

Natural Language Processing

| | |
|-------------------|--|
| Authors: | Steven Bird, Ewan Klein, Edward Loper |
| Version: | 0.9.6 (draft only, please send feedback to authors) |
| Copyright: | © 2001-2008 the authors |
| License: | Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License |
| Revision: | |
| Date: | |

Contents

- [Preface](#)
 - [Audience](#)
 - [Emphasis](#)
 - [What You Will Learn](#)
 - [Organization](#)
 - [Why Python?](#)
 - [Learning Python for Natural Language Processing](#)
 - [The Design of NLTK](#)
 - [For Instructors](#)
 - [Acknowledgments](#)
 - [About the Authors](#)
- [1. Language Processing and Python](#)
 - [Computing with Language: Texts and Words](#)
 - [A Closer Look at Python: Texts as Lists of Words](#)
 - [Computing with Language: Simple Statistics](#)
 - [Back to Python: Making Decisions and Taking Control](#)
 - [Automatic Natural Language Understanding](#)
 - [Summary](#)
- [2. Text Corpora and Lexical Resources](#)
 - [Accessing Text Corpora](#)
 - [Conditional Frequency Distributions](#)
 - [More Python: Reusing Code](#)
 - [Lexical Resources](#)
 - [WordNet](#)
 - [Summary](#)
 - [Further Reading \(NOTES\)](#)
 - [Exercises](#)
- [3. Processing Raw Text](#)
 - [Accessing Text from the Web and from Disk](#)
 - [Strings: Text Processing at the Lowest Level](#)
 - [Regular Expressions for Detecting Word Patterns](#)
 - [Useful Applications of Regular Expressions](#)
 - [Normalizing Text](#)
 - [Regular Expressions for Tokenizing Text](#)
 - [Sentence Segmentation](#)
 - [Formatting: From Lists to Strings](#)
 - [Conclusion](#)
 - [Summary](#)
 - [Further Reading \(NOTES\)](#)
 - [Exercises](#)
- [4. Categorizing and Tagging Words](#)
 - [Applications of Tagging](#)
 - [Tagged Corpora](#)
 - [Mapping Words to Properties Using Python Dictionaries](#)
 - [Automatic Tagging](#)
 - [N-Gram Tagging](#)

- [Transformation-Based Tagging](#)
- [The TnT Tagger](#)
- [How to Determine the Category of a Word](#)
- [Summary](#)
- [Further Reading](#)
- [Exercises](#)
- [**5. Data-Intensive Language Processing**](#)
 - [Introduction](#)
 - [Exploratory Data Analysis](#)
 - [Selecting a Corpus](#)
 - [Search](#)
 - [Data Modeling](#)
 - [Evaluation](#)
 - [Classification Methods](#)
 - [Decision Trees](#)
 - [Maximum Entropy Classifiers](#)
 - [Exercises](#)
- [**6. Structured Programming in Python**](#)
 - [Back to the Basics](#)
 - [Functions](#)
 - [Iterators](#)
 - [Algorithm Design Strategies](#)
 - [Visualizing Language Data \(DRAFT\)](#)
 - [Object-Oriented Programming in Python](#)
 - [Further Reading](#)
 - [Exercises](#)
- [**7. Shallow Linguistic Processing**](#)
 - [Information Extraction](#)
 - [Chunking](#)
 - [Developing and Evaluating Chunkers](#)
 - [Building Nested Structure with Cascaded Chunkers](#)
 - [Conclusion](#)
 - [Further Reading](#)
 - [Exercises](#)
- [**8. Grammars and Parsing**](#)
 - [Some Grammatical Dilemmas](#)
 - [What's the Use of Syntax?](#)
 - [Context Free Grammar](#)
 - [Parsing With Context Free Grammar](#)
 - [Chart Parsing](#)
 - [Summary \(notes\)](#)
 - [Further Reading](#)
 - [Exercises](#)
- [**9. Advanced Topics in Parsing**](#)
 - [A Problem of Scale](#)
 - [Treebanks \(notes\)](#)
 - [Probabilistic Parsing](#)
 - [Grammar Induction](#)
 - [Dependency Grammar](#)
 - [Further Reading](#)
 - [Exercises](#)
- [**10. Feature Based Grammar**](#)
 - [Introduction](#)
 - [Why Features?](#)
 - [Computing with Feature Structures](#)
 - [Extending a Feature-Based Grammar](#)
 - [Summary](#)
 - [Further Reading](#)
 - [Exercises](#)

- [11. Analyzing the Meaning of Sentences](#)
 - [Natural Language Understanding](#)
 - [Propositional Logic](#)
 - [First-Order Logic](#)
 - [The Semantics of English Sentences](#)
 - [Inference Tools](#)
 - [Discourse Semantics](#)
 - [Summary](#)
 - [Further Reading](#)
 - [Exercises](#)
- [12. Linguistic Data Management \(DRAFT\)](#)
 - [Introduction](#)
 - [Linguistic Databases](#)
 - [Creating Data](#)
 - [Converting Data Formats](#)
 - [Analyzing Language Data](#)
 - [Summary](#)
 - [Further Reading](#)
 - [Exercises](#)
- [Appendix: Tagsets](#)
 - [Brown Tagset](#)
 - [CLAWS5 Tagset](#)
 - [UPenn Tagset](#)
- [Appendix: Text Processing with Unicode](#)
 - [What is Unicode?](#)
 - [Extracting encoded text from files](#)
 - [Using your local encoding in Python](#)
- [Appendix: NLP in Python vs other Programming Languages](#)
- [Appendix: NLTK Modules and Corpora](#)
- [Appendix: Python and NLTK Cheat Sheet \(Draft\)](#)
 - [Python](#)
 - [NLTK](#)

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 like 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 the number of times the letter *t* occurs in a paragraph of text. 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, handheld computers (PDAs) 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 comprehensive introduction to the field of NLP. It can be used for individual study or as the textbook a course on natural language processing or computational linguistics. The book is intensely practical, containing hundreds of fully-worked examples and graded exercises. It is based on the Python programming language together with an open source library called the *Natural Language Toolkit* (NLTK). NLTK includes software, data, and documentation, all freely downloadable from <http://www.nltk.org/>. Distributions are provided for Windows, Macintosh and Unix platforms. We encourage you, the reader, 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, it includes people in human-computer interaction, business information analysis, and Web software development. Within academia, this includes people in areas from humanities computing and corpus linguistics through to computer science and artificial intelligence.

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

New to Programming?:

The book is suitable for readers with no prior knowledge of programming, and the early chapters contain many examples that you can simply copy and try for yourself, together with graded exercises. If you decide you need a more general introduction to Python, we recommend you read *Learning Python* (O'Reilly) in conjunction with this book.

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.

Already dreaming in Python?:

Skip the Python examples and dig into the interesting language analysis material that starts in [Chapter 1](#). Soon you'll be applying your skills to this exciting new application area.

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 learnt 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, sometimes whimsical.

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 analyse 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 below:

Table I.1

| Goals | Background in Arts and Humanities | Background in Science and Engineering |
|---------------------|---|--|
| Language Analysis | Programming to manage language data, explore linguistic models, models, and test empirical claims | Language as a source of interesting problems in data modeling, data mining, and knowledge discovery |
| Language Technology | Learning to program, with applications to familiar problems to work in language technology or other technical field | Knowledge of linguistic algorithms and data structures for high quality, maintainable language processing software |

Organization

The early chapters are organized in order of conceptual difficulty, starting with a gentle introduction to language processing and Python, before proceeding on to fundamental topics such as tokenization, tagging, and evaluation. After this, a sequence of chapters covers topics in grammars and parsing, which have long been central tasks in language processing. The last third of the book contains chapters on advanced topics, which can be read independently of each other.

Each chapter consists of an introduction, a sequence of sections that progress from elementary to advanced material, and finally a summary and suggestions for further reading. Most sections include exercises that 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 are encouraged to skip these); ☺ is for non-programming exercises for reflection or discussion. The exercises are important for consolidating the material in each section, and we strongly encourage you to try a few before continuing with the rest of the chapter.

Within each chapter, we'll be switching between different styles of presentation. In one style, natural language will be the driver. We'll analyze language, explore linguistic concepts, and use programming examples to support the discussion. Sometimes we'll present Python constructs that have not been introduced systematically; this way you will see useful idioms early, and might not appreciate their workings until later. In the other style, the programming language will be the driver. We'll analyze programs, explore algorithms, and use linguistic examples to support the discussion.

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 above, `split()` had no argument because we were splitting the string wherever there was white space, and we could therefore use empty parentheses. To split a string into sentences delimited by a period, we would write `split('.')`. Finally — and most importantly — Python is highly readable, so much so that it is fairly easy to guess what the above 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 a scripting 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 data processing.

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 tokenization, part-of-speech tagging, and syntactic parsing; and standard implementations for each task which can be combined to solve complex problems.

NLTK comes with extensive documentation. In addition to this book, the website <http://www.nltk.org/> provides API documentation which covers every module, class and function in the toolkit, specifying parameters and giving examples of usage. The website also provides module **guides**; these contain extensive examples and test cases, and are intended for users, developers and instructors.

Learning Python for Natural Language Processing

This book contains self-paced learning materials including many examples and exercises. An effective way to learn is simply to work through the materials. The program fragments can be copied directly into a Python interactive session. Any questions concerning the book, or Python and NLP more generally, can be posted to the NLTK-Users mailing list (see <http://www.nltk.org/>).

Python Environments:

The easiest way to start developing Python code, and to run interactive Python demonstrations, is to use the simple editor and interpreter GUI that comes with Python called *IDLE*, the *Integrated Development Environment for Python*.

NLTK Community: NLTK has a large and growing user base. There are mailing lists for announcements about NLTK, for developers and for teachers. <http://www.nltk.org/> lists many courses around the world where NLTK and materials from this book have been adopted, a useful source of extra materials including slides and exercises.

The Design of NLTK

NLTK was designed with four primary goals in mind:

- | | |
|-----------------------|--|
| Simplicity: | We have tried to provide an intuitive and appealing framework along with substantial building blocks, so you can gain a practical knowledge of NLP without getting bogged down in the tedious house-keeping usually associated with processing annotated language data. We have provided software distributions for several platforms, along with platform-specific instructions, to make the toolkit easy to install. |
| Consistency: | We have made a significant effort to ensure that all the data structures and interfaces are consistent, making it easy to carry out a variety of tasks using a uniform framework. |
| Extensibility: | The toolkit easily accommodates new components, whether those components replicate or extend existing functionality. Moreover, the toolkit is organized so that it is usually obvious where extensions would fit into the toolkit's infrastructure. |
| Modularity: | The interaction between different components of the toolkit uses simple, well-defined interfaces. It is possible to complete individual projects using small parts of the toolkit, without needing to understand how they interact with the rest of the toolkit. Modularity also makes it easier to change and extend the toolkit. |

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

For Instructors

Natural Language Processing (NLP) is often taught within the confines of a single-semester course at 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 making it possible to view algorithms step-by-step. Most NLTK components include a demonstration which 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, entering them in a Python session, observing what they do, and modifying them to explore some empirical or theoretical issue.

The book contains hundreds of examples and exercises which 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 data-sets (corpora), and a flexible and

extensible architecture. Additional support for teaching using NLTK is available on the NLTK website, and on a closed mailing list for instructors.

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 — before introducing non-trivial control structures like comprehensions and conditionals. These idioms permit us to do useful language processing from the start. Once this motivation is in place we deal with the fundamental concepts systematically. Thus we cover the same ground as more conventional approaches, without expecting readers to be interested in the programming language for its own sake.

Table I.2:

Suggested Course Plans; Lectures/Lab Sessions per Chapter

| Chapter | Arts and Humanities | Science and Engineering |
|--------------------------------------|---------------------|-------------------------|
| 1 Language Processing and Python | 2-4 | 2 |
| 2 Text Corpora and Lexical Resources | 2-4 | 2 |
| 3 Processing Raw Text | 2-4 | 2 |
| 4 Categorizing and Tagging Words | 2-4 | 2-4 |
| 5 Data-Intensive Language Processing | 0-2 | 2-4 |
| 6 Structured Programming | 2-4 | 0 |
| 7 Partial Parsing and Interpretation | 2 | 2 |
| 8 Grammars and Parsing | 2-4 | 2-4 |
| 9 Advanced Parsing | 0 | 1-4 |
| 10 Feature Based Grammar | 2-4 | 1-4 |
| 11 Logical Semantics | 1 | 1-4 |
| 12 Linguistic Data Management | 0-2 | 0-4 |
| 13 Conclusion | 1 | 1 |
| Total | 18-36 | 18-36 |

Acknowledgments

NLTK was originally created as part of a computational linguistics course in the Department of Computer and Information Science at the University of Pennsylvania in 2001. 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.

In particular, we're grateful to the following people for their feedback, comments on earlier drafts, advice, contributions: Michaela Atterer, Greg Aumann, Kenneth Beesley, Steven Bethard, Ondrej Bojar, Trevor Cohn, Grev Corbett, James Curran, Jean Mark Gawron, Baden Hughes, Gwillim Law, Mark Liberman, Christopher Maloof, Stefan Müller, Stuart Robinson, Jussi Salmela, Rob Speer. Many others have contributed to the toolkit, and they are listed at <http://www.nltk.org/>. We are grateful to many colleagues and students for feedback on the text.

We are grateful to the US National Science Foundation, the Linguistic Data Consortium, and the Universities of Pennsylvania, Edinburgh, and Melbourne for supporting our work on this book.

About the Authors



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

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. After completing his undergraduate training in computer science and mathematics at the University of Melbourne, Steven went to the University of Edinburgh to study computational linguistics, and completed his PhD in 1990 under the supervision of Ewan Klein. He later moved to Cameroon to conduct linguistic fieldwork on the Grassfields Bantu languages under the auspices of the Summer Institute of Linguistics. More recently, he spent several years as Associate Director of the Linguistic Data Consortium where he led an R&D team to create models and tools for large databases of annotated text. Back at Melbourne University, he established a language technology research group and has taught at all levels of the undergraduate computer science curriculum. Steven is Vice President of the Association for Computational Linguistics.

Ewan Klein is Professor of Language Technology in the School of Informatics at the University of Edinburgh. He completed a PhD on formal semantics at the University of Cambridge in 1978. After some years working at the Universities of Sussex and Newcastle upon Tyne, Ewan took up a teaching position at Edinburgh. He was involved in the establishment of Edinburgh's Language Technology Group 1993, and has been closely associated with it ever since. From 2000–2002, he took leave from the University to act as Research Manager for the Edinburgh-based Natural Language Research Group of Edify Corporation, Santa Clara, and was responsible for spoken dialogue processing. Ewan is a past President of the European Chapter of the Association for Computational Linguistics and was a founding member and Coordinator of the European Network of Excellence in Human Language Technologies (ELSNET). He has been involved in leading numerous academic-industrial collaborative projects, the most recent of which is a biological text mining initiative funded by ITI Life Sciences, Scotland, in collaboration with Cognia Corporation, NY.

Edward Loper is a doctoral student in the Department of Computer and Information Sciences at the University of Pennsylvania, conducting research on machine learning in natural language processing. Edward was a student in Steven's graduate course on computational linguistics in the fall of 2000, and went on to be a TA and share in the development of NLTK. In addition to NLTK, he has helped develop other major packages for documenting and testing Python software, `epydoc` and `doctest`.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#). Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

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 tackle 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. is the Python programming language suitable 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 understanding 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/>.

1.1 Computing with Language: Texts and Words

We're all very familiar with text, since we read and write it every day. But 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. Before we can do this, we have to get started with the Python interpreter.

Getting Started

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*→*Mac Python*, 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 greater (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 be sure not to type in the `>>>` prompt 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 or floating point division depending on whether you type `1/3` or `1.0/3.0`.

These examples demonstrate how you can work interactively with the interpreter, allowing you to experiment and explore. Now let's try a nonsensical expression to see how the interpreter handles it:

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

Here we have produced a **syntax error**. 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 "standard input").

Note

The chapter contains many examples and exercises; there is no better way to learn to NLP than to dive in and try these yourself. However, before continuing you need to install NLTK,

downloadable for free from <http://www.nltk.org/>. Once you've done this, install the data required for the book by typing `nltk.download()` at the Python prompt, and download the book collection.

Searching Text

Now that we can use the Python interpreter, let's see how we can harness its power to process text. 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 *` — i.e. load NLTK's `book` module, which contains the examples you'll be working with as you read this chapter. After printing a welcome message, it loads the text of several books, including *Moby Dick*. Type the following, taking care to get spelling and punctuation exactly right:

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading: text1, ..., text8 and sent1, ..., sent8
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
>>> text1
<Text: Moby Dick by Herman Melville 1851>
>>> text2
<Text: Sense and Sensibility by Jane Austen 1811>
>>>
```

We can examine the contents of a text in a variety of ways. A concordance view shows us every occurrence of a given word, together with some context. Here we look up the word *monstrous*.

```
>>> text1.concordance("monstrous")
mong the former , one was of a most monstrous size . . . This came towards us , o
ION OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have re
all over with a heathenish array of monstrous clubs and spears . Some were thickl
ed as you gazed , and wondered what monstrous cannibal and savage could ever have
that has survived the flood ; most monstrous and most mountainous ! That Himmale
they might scout at Moby Dick as a monstrous fable , or still worse and more det
ath of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere lo
ling Scenes . In connexion with the monstrous pictures of whales , I am strongly
>>>
```

Note

Your Turn: Try searching for other words; you can use the up-arrow key 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 *US Presidential Inaugural Addresses* to see examples of English dating 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 few minutes 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.

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. We can also determine the *location* of a word in the text: how many words in from the beginning it appears. This positional information can be displayed using a so-called **dispersion plot**. Each stripe represents an instance of a word and each row represents the entire text. In [Figure 1.1](#) we see some striking patterns of word usage over the last 220 years. You can produce this plot as shown below. You might like to try different words, 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"])
>>>
```

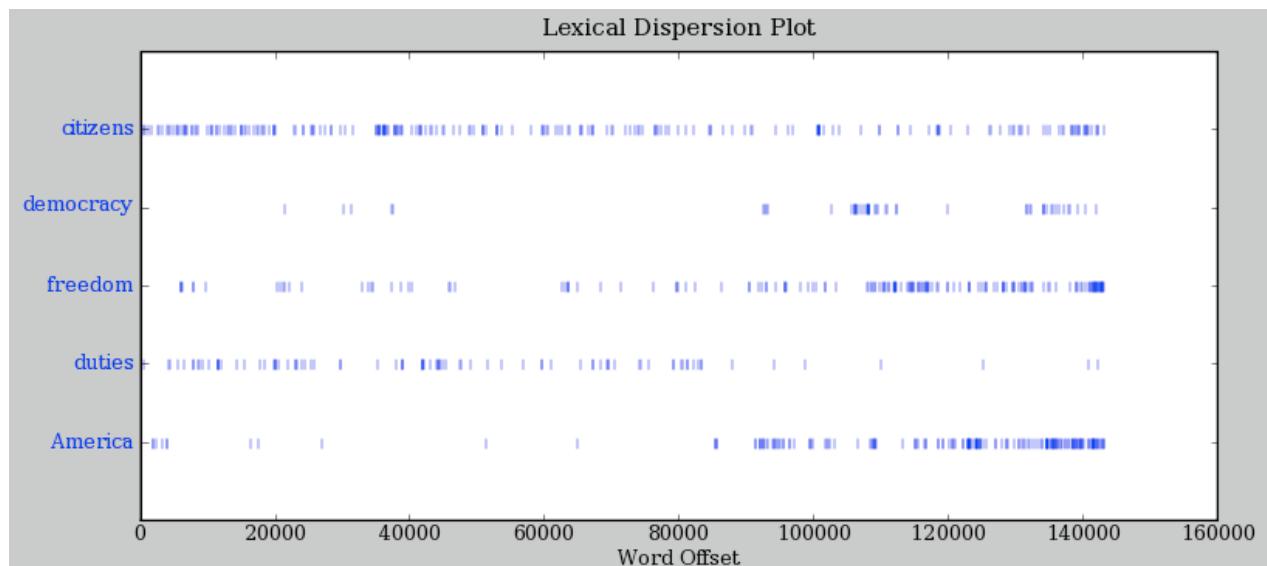


Figure 1.1: Lexical Dispersion Plot for Words in US Presidential Inaugural Addresses

Note

You need to have Python's Numpy and Pylab packages installed in order to produce the graphical plots used in this book. Please see <http://www.nltk.org/> for installation instructions.

A concordance permits us to see words in context, e.g. we saw that *monstrous* appeared in the context *the monstrous pictures*. What other words appear in the same contexts that *monstrous* appears in? We can find out as follows:

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

Observe that we get different results for different books. Melville and Austen use this word quite differently. For Austen *monstrous* has positive connotations, and might even function as an intensifier, like the word *very*. Let's examine the contexts that are shared by *monstrous* and *very*

```
>>> text2.common_contexts(["monstrous", "very"])
be_glad am_glad a.pretty is.pretty a_lucky
>>>
```

Note

Your Turn: Pick another word and compare its usage in two different texts, using the `similar()` and `common_contexts()` methods.

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 "generate" function:

```
>>> 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 first time you run this, 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 re-uses common words and phrases from the source text and gives us a sense of its style and content.

Note

When text is printed, punctuation has been split off from the previous word. Although this is not correct formatting for English text, we do this to make it clear that punctuation does not belong to the word. This is called "tokenization", and you will learn about it in [Chapter 3](#).

Counting Vocabulary

The most obvious fact about texts that emerges from the previous section 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'll use the text of *Moby Dick* again:

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

That's a quarter of a million words long! But how many distinct words does this text 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 words that it uses, and in Python we can list the vocabulary of `text3` with the command: `set(text3)` (many screens of words will fly past). Now try the following:

```
>>> sorted(set(text3))
[',', '.', '(', ')', ',', '.', '.', ':', ';', '?', '?'],
['A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abri', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]
>>> len(set(text3))
2789
>>> len(text3) / len(set(text3))
16
>>>
```

Here we can see 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 length of the set. There are fewer than 3,000 distinct words in this book. Finally, we can calculate a measure of the lexical richness of the text and learn that each word is used 16 times on average.

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
>>> 100.0 * text4.count('a') / len(text4)
1.4587672822333748
>>>
```

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 might like to repeat such calculations on several texts, but it is tedious to keep retyping it for different texts. Instead, we can come up with our own name for a task, e.g. "score", and associate it with a block of code. Now we only have to type a short name instead of one or more complete lines of Python code, and we can re-use it as often as we like:

```
>>> def score(text):
...     return len(text) / len(set(text))
...
>>> score(text3)
16
>>> score(text5)
4
>>>
```

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. To finish the indented block just enter a blank line.

The keyword `def` is short for "define", and the above code defines a *function*:dt" called "score". We used the function by typing its name, followed by an open parenthesis, the name of the text, then a close parenthesis. This is just what we did for the `len` and `set` functions earlier. These parentheses will show up often: their role is to separate the name of a task — such as `score` — from the data that the task is to be performed on — such as `text3`. Functions are an advanced concept in programming and we only mention them at the outset to give newcomers a sense of the power and creativity of programming. Later we'll see how to use such functions when tabulating data, like [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 functions.

Table 1.1:

Lexical Diversity of Various Genres in the Brown Corpus

| Genre | Token Count | Type Count | Score |
|-------------------|-------------|------------|-------|
| 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 building blocks of the Python programming language. Let's 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, and we can ask for its length:

```
>>> sent1
['Call', 'me', 'Ishmael', '.']
>>> len(sent1)
4
>>> score(sent1)
1
>>>
```

We can even apply our own "score" function to it. Some more lists have been defined for you, one for the opening sentence of each of our texts, sent2 ... sent8. We inspect two of them here; you can see the rest for yourself using the Python interpreter.

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

Note

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 above in [Section 1.1](#), e.g. sorted(ex1), len(set(ex1)), ex1.count('the').

We can also do arithmetic operations with lists in Python. Multiplying a list by a number, e.g. sent1 * 2, creates a longer list with multiple copies of the items in the original list. Adding two lists, e.g. sent4 + sent1, creates a new list with everything from the first list, followed by everything from the second list:

```
>>> sent1 * 2
['Call', 'me', 'Ishmael', '.', 'Call', 'me', 'Ishmael', '.']
>>> sent4 + sent1
['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the',
'House', 'of', 'Representatives', ':', 'Call', 'me', 'Ishmael', '.']
>>>
```

Note

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

Indexing Lists

As we have seen, a text in Python is just a list of words, represented using a particular combination of brackets and quotes. Just as with an ordinary page of text, we can count up the total number of words using len(text1), and count the occurrences of a particular word using text1.count('heaven'). And just as we can pick out the first, tenth, or even 14,278th word in a printed text, we can identify the elements of a list by their number, or **index**, by following the name of the text with the index inside brackets. We can also find the index of the first occurrence of any word:

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

Indexes turn out to be a common way to access the words of a text, or — more generally — the elements of a 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',
...           'word11', 'word12', 'word13', 'word14', 'word15',
...           'word16', 'word17', 'word18', 'word19', 'word20']
>>> sent[0]
'word1'
>>> sent[19]
'word20'
>>>
```

Notice that our indexes start from zero: `sent` element zero, written `sent[0]`, is the first word, `'word1'`, while `sent` element 19 is `'word20'`. 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.

Note

This is initially confusing, but typical of modern programming languages. You'll quickly get the hang of this 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 tell it to go too far, by using an index value that is too large, we get an error:

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

This time it is not a syntax error, for 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:

```
>>> sent[17:20]
['word18', 'word19', 'word20']
>>> sent[17]
'word18'
>>> sent[18]
'word19'
>>> sent[19]
'word20'
>>>
```

Thus, the slice `17:20` includes `sent` elements 17, 18, and 19. By convention, `m:n` means elements $m \dots n-1$. We can omit the first number if the slice begins at the start of the list, and we can omit the second number if the slice goes to the end:

```
>>> sent[:3]
['word1', 'word2', 'word3']
>>> text2[141525:]
['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, e.g. putting `sent[0]` on the left of the equals sign. We can also replace an entire slice with new material:

```
>>> sent[0] = 'First Word'
>>> sent[19] = 'Last Word'
>>> sent[1:19] = ['Second Word', 'Third Word']
>>> sent
['First Word', 'Second Word', 'Third Word', 'Last Word']
```

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 above. 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 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 copied from the right side to the left. It might help to think of it as a left-arrow. 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:

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

It is good to choose meaningful variable names to help you — and anyone who reads your Python code — to understand 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`. A variable name cannot be any of Python's reserved words, such as `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 can use variables to hold intermediate steps of a computation. This may make the Python 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. You can use underscores anywhere in a name, but not a hyphen, since this gets interpreted 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**):

```

>>> name = 'Monty'
>>> name[0]
'M'
>>> name[:4]
'Mont'
>>> 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](#).

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 will 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 with this section, check your understanding of the previous section by predicting the output of the following code, and using the interpreter to check if you got it right. If you found it difficult 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', '.']
>>> words = set(saying)
>>> words = sorted(words)
>>> words[:2]

```

Frequency Distributions

How could we automatically identify the words of a text that are most informative about the topic and genre of the text? Let's begin by finding the most frequent words 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 1.2](#). We would need thousands of counters and it would be a laborious process, so laborious that we would rather assign the task to a machine.

Word Tally

| | | | |
|-----------|--|--|--|
| the | | | |
| been | | | |
| message | | | |
| persevere | | | |
| nation | | | |

Figure 1.2: Counting Words Appearing in a Text (a frequency distribution)

The table in [Figure 1.2](#) 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 words in the text — 260,819 in the case of *Moby Dick* — 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*.

```
>>> fdist1 = FreqDist(text1)
>>> fdist1
<FreqDist with 260819 samples>
>>> vocabulary1 = fdist1.keys()
>>> vocabulary1[:50]
[',', '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
 >>>

Note

Your Turn: Try the above frequency distribution example for yourself, for `text2`. Be careful 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 in the above list 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 English text is taken up with such words? We can generate a cumulative frequency plot for these words, using `fdist1.plot(cumulative=True)`, to produce the graph in [Figure 1.3](#). These 50 words account for nearly half the book!

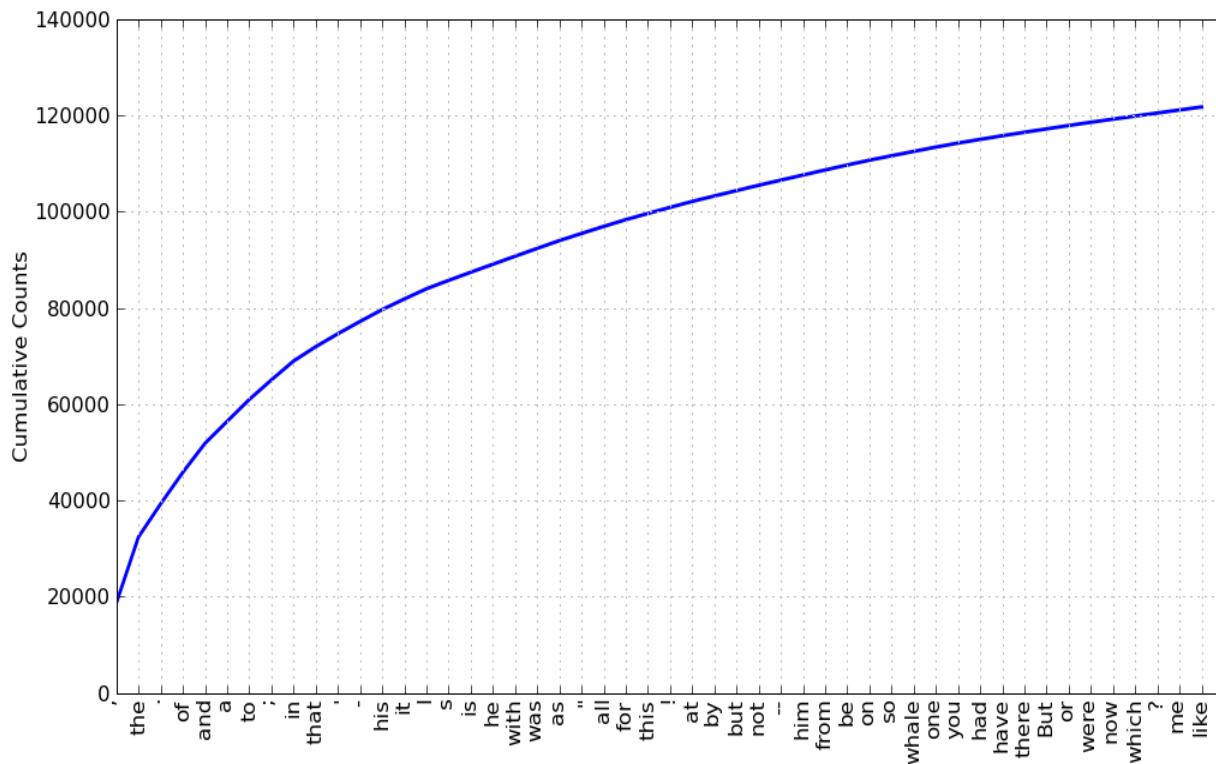


Figure 1.3: Cumulative Frequency Plot for 50 Most Frequent Words in *Moby Dick*

If the frequent words don't help us, how about the words that occur once only, the so-called **hapaxes**. See them using `fdist1.hapaxes()`. This list contains *lexicographer*, *cetological*, *contraband*, *expostulations*, and about 9,000 others. It seems that there's too many rare words, and without seeing the context we probably can't guess what half of them mean in any case! Neither frequent nor infrequent words help, so 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 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)
- $\{w \mid w \in V \& P(w)\}$
 - `[w for w in V if p(w)]`

The equivalent Python expression is given in (1b). Notice 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 if `len(w)` is greater than 15; all other words will be ignored. We will discuss this syntax more carefully later.

Note

Your Turn: Try out the above statements in the Python interpreter, and try changing the text, and changing the length condition. Also try changing 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*, while those in `text5` reflect its informal content: *booooooooooooooglyyyyyyy* and *yuuuuuuuuuuuuuummmmmmmmmmmmm*. 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 like (*antiphilosopists*). Here are all words from the chat corpus that are longer than 7 characters, that occur more than 7 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 thousands of words, produces some informative output.

Bigrams and Collocations

Frequency distributions are very powerful. Here we briefly explore a more advanced application that uses word pairs, also known as **bigrams**. We can convert a list of words to a list of bigrams as follows:

```
>>> 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()
United States; fellow citizens; has been; have been; those who;
```

```

Declaration Independence; Old World; Indian tribes; District Columbia;
four years; Chief Magistrate; and the; the world; years ago; Santo
Domingo; Vice President; the people; for the; specie payments; Western
Hemisphere
>>>

```

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]
[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)
>>> fdist
<FreqDist with 260819 samples>
>>> fdist.samples()
[3, 1, 4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20]
>>>

```

The material being counted up in the frequency distribution consists of the numbers `[1, 4, 4, 2, ...]`, and the result is a distribution containing a quarter of a million items, one per word. There are only twenty distinct items being counted, the numbers 1 through 20. Let's look at the frequency of each sample:

```

>>> 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 50,000 (20%) of the words of the book. Further analysis of word length might help us understand differences between authors, genres or languages. [Table 1.2](#) summarizes the methods defined in frequency distributions.

Table 1.2:

Methods Defined for NLTK's Frequency Distributions

| Example | Description |
|--|--|
| <code>fdist = FreqDist(samples)</code> | create a frequency distribution containing the given samples |
| <code>fdist.inc(sample)</code> | increment the count for this sample |
| <code>fdist['monstrous']</code> | count of the number of times a given sample occurred |
| <code>fdist.freq('monstrous')</code> | frequency of a given sample |
| <code>fdist.N()</code> | total number of samples |
| <code>fdist.keys()</code> | the samples sorted in order of decreasing frequency |
| <code>for sample in fdist:</code> | iterate over the samples, in order of decreasing frequency |
| <code>fdist.max()</code> | sample with the greatest count |
| <code>fdist.tabulate()</code> | tabulate the frequency distribution |
| <code>fdist.plot()</code> | graphical plot of the frequency distribution |
| <code>fdist.plot(cumulative=True)</code> | cumulative plot of the frequency distribution |
| <code>fdist1 < fdist2</code> | samples in <code>fdist1</code> occur less frequently than in <code>fdist2</code> |

Our discussion of frequency distributions has introduced some important Python concepts, and we will look at them systematically in [Section 1.4](#). We've also touched on the topic of normalization, and we'll explore this in depth in [Chapter 3](#).

1.4 Back to Python: Making Decisions and Taking Control

So far, our little programs have had some interesting qualities: (i) the ability to work with language, and (ii) 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 like `<` 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 |
|--------------------|--|
| <code><</code> | less than |
| <code><=</code> | less than or equal to |
| <code>==</code> | equal to (note this is two not one = sign) |
| <code>!=</code> | not equal to |
| <code>></code> | greater than |
| <code>>=</code> | greater than or equal to |

We can use these to select different words from a sentence of news text. Here are some examples — only the operator is changed from one line to the next. They all use `sent7`, the first sentence from `text7` (Wall Street Journal).

```
>>> [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', '.']
>>>
```

Notice the pattern in all of these examples: `[w for w in text if condition]`. In these cases 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 |
|------------------------------|---|
| <code>s.startswith(t)</code> | <code>s</code> starts with <code>t</code> |
| <code>s.endswith(t)</code> | <code>s</code> ends with <code>t</code> |
| <code>t in s</code> | <code>t</code> is contained inside <code>s</code> |
| <code>s.islower()</code> | all cased characters in <code>s</code> are lowercase |
| <code>s.isupper()</code> | all cased characters in <code>s</code> are uppercase |
| <code>s.isalpha()</code> | all characters in <code>s</code> are alphabetic |
| <code>s.isalnum()</code> | all characters in <code>s</code> are alphanumeric |
| <code>s.isdigit()</code> | all characters in <code>s</code> are digits |
| <code>s.istitle()</code> | <code>s</code> is titlecased (all words have initial capital) |

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 'gmt' in term)
['Sovereignty', 'sovereignty', 'sovereignty']
>>> sorted(item for item in set(text6) if item.istitle())
['A', 'Aaaaaaaaaah', 'Aaaaaaaaaah', 'Aaaaah', '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 `and` and `or`: $c_1 \text{ and } c_2$, $c_1 \text{ or } c_2$.

Note

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 above 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.

Note

The above notation 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 incorrect result (why?). Don't worry if you don't feel confident with these already. You might like to try some of the exercises at the end of this chapter, or wait til we come back to these again in the next chapter.

Nested Code Blocks

Most programming languages permit us to execute a block of code when a **conditional expression**, or `if` statement, is satisfied. In the following program, we have created a variable called `word` containing the string value '`'cat'`'. The `if` statement checks whether the conditional expression `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'
...
word length is less than 5
>>>
```

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

If we change the conditional expression to `len(word) >= 5`, to check that the length of `word` is greater than or equal to 5, then the conditional expression 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. Don't forget 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 only print the item 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 'lowercase word'
...     elif token.istitle():
...         print 'titlecase word'
...     else:
...         print 'punctuation'
...
titlecase word
lowercase word
titlecase word
punctuation
>>>
```

As you can see, even with this small amount of Python knowledge, you can start to build multi-line 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 *c_ie* and *ce_i* 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.

```
>>> confusing = sorted(w for w in set(text2) if 'cie' in w or 'cei' in w)
>>> for word in confusing:
...     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, dictionaries, and a programming language. However, we're also interested in exploiting our knowledge of language and computation by building useful language technologies.

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 behaviour 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 analysing 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 processing components and systems, to give you a sense 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

Now, 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. Its 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 her 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, e.g.: *the book by Chesterton* (agentive); *the cup by the stove* (locative); and *submit by Friday* (temporal). Observe in (3c) 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 learnt to do this in elementary school, but its 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 (4c), 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* (the thieves or the 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 verb (as agent, patient, instrument, and so on).

Generating Language Output

If we can automatically solve such problems, we will have understood enough of the text to perform some 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 above 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 the above examples — working out the sense of a word, the subject of a verb, 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. We'll come back to some of these topics later in the book.

Spoken Dialog 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 below:

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 the above 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 complete processing pipeline for NLP. [Figure 1.4](#) shows the architecture of a simple dialogue system.

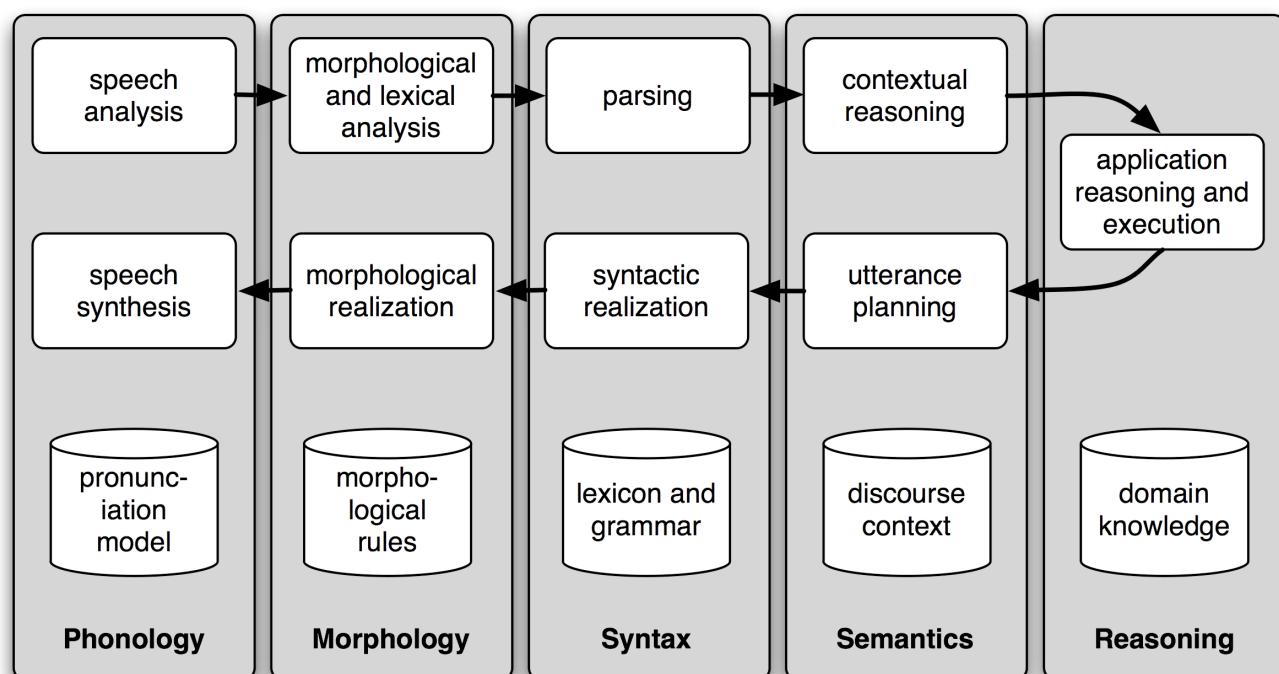


Figure 1.4: Simple Pipeline Architecture for a Spoken Dialogue System

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.

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 which allow competitors to develop their systems, but not enough data to brute-force approaches using standard machine learning techniques. Consequently, some linguistic analysis is crucial. In the above 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)
- David Golinkin is the editor or author of eighteen books, and over 150 responsa, articles, sermons and books
 - Golinkin has written eighteen books

In order to determine whether or not 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 that book; (iii) if someone is editor or author of eighteen books, then one cannot conclude that he/she is author of eighteen books.

Limitations of NLP

Despite the research-led advances in tasks like 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 holy grail of natural language understanding, using superficial yet powerful counting and symbol manipulation techniques, but *without* recourse to this unrestricted knowledge and reasoning capability.

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 vision 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.
- We get the vocabulary of a text `t` using `sorted(set(t))`.
- To get the vocabulary, collapsing case distinctions and ignoring punctuation, we can write `set(w.lower() for w in text if w.isalpha())`.
- We operate on each item of a text using `[f(x) for x in text]`.
- 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).

- WordNet is a semantically-oriented dictionary of English, consisting of synonym sets — or synsets — and organized into a hierarchical network.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

2 Text Corpora and Lexical Resources

Practical work in Natural Language Processing usually involves a variety of established bodies of linguistic data. Such a body of text is called a **corpus** (plural **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 re-use code effectively?

This chapter continues to present programming concepts by example, in the context of a linguistic processing task. We will wait till 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 any large body of text. Many, but not all, 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 we glued them end-to-end and treated them as a single text. In this section we will examine a variety of text corpora and will see how to select individual texts, and how to work with them.

The Gutenberg Corpus

NLTK includes a small selection of texts from the Project Gutenberg <http://www.gutenberg.org/> electronic text archive containing some 25,000 free electronic books. We begin by getting the Python interpreter to load the NLTK package, then ask to see `nltk.corpus.gutenberg.files()`, the files in NLTK's corpus of Gutenberg texts:

```
>>> import nltk
>>> nltk.corpus.gutenberg.files()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.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
```

Note

You cannot carry out concordancing (and other tasks from [Section 1.1](#)) using a text defined this way. Instead you have to make the following statement:

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

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

```
>>> from nltk.corpus import gutenberg
>>> gutenberg.files()
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
'blake-poems.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.txt',
'chesterton-thursday.txt', 'melville-moby_dick.txt', 'milton-paradise.txt',
'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt',
'whitman-leaves.txt')
```

Let's write a short program to display other information about each text:

```
>>> for file in gutenberg.files():
...     num_chars = len(gutenberg.raw(file))
...     num_words = len(gutenberg.words(file))
...     num_sents = len(gutenberg.sents(file))
...     num_vocab = len(set(w.lower() for w in gutenberg.words(file)))
...     print num_chars/num_words, num_words/num_sents, num_words/num_vocab, file
...
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 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.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 has displayed 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 is always 4. Average sentence length and lexical diversity appear to be characteristics of particular authors.

This example also showed how we can access the "raw" text of the book, not split up into words. 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[1038]
['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', ...], ...]
```

Note

Most NLTK corpus readers include a variety of access methods apart from `words()`. We access the raw file contents using `raw()`, and get the content sentence by sentence using `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 Caribbean*, personal advertisements, and wine reviews:

```
>>> from nltk.corpus import webtext
>>> for f in webtext.files():
...     print f, webtext.raw(f)[:70]
...
firefox.txt Cookie Manager: "Don't allow sites that set removed cookies to set fut
grail.txt SCENE 1: [wind] [clop clop clop] KING ARTHUR: Whoa there! [clop clop
overheard.txt White guy: So, do you have any plans for this evening? Asian girl: Yea
pirates.txt PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted Elliott & Terry Ross
singles.txt 25 SEXY MALE, seeks attrac older single lady, for discreet encounters.
wine.txt Lovely delicate, fragrant Rhone wine. Polished leather and strawberrie
```

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', '.']
```

The 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: <i>Society Reportage</i> |
| B02 | cb02 | editorial | Christian Science Monitor: <i>Editorials</i> |
| C17 | cc17 | reviews | Time Magazine: <i>Reviews</i> |
| D12 | cd12 | religion | Underwood: <i>Probing the Ethics of Realtors</i> |
| E36 | ce36 | hobbies | Norling: <i>Renting a Car in Europe</i> |
| F25 | cf25 | lore | Boroff: <i>Jewish Teenage Culture</i> |
| G22 | cg22 | belles_lettres | Reiner: <i>Coping with Runaway Technology</i> |
| H15 | ch15 | government | US Office of Civil and Defence Mobilization: <i>The Family Fallout Shelter</i> |
| J17 | cj19 | learned | Mosteller: <i>Probability with Statistical Applications</i> |
| K04 | ck04 | fiction | W.E. Du Bois: <i>Worlds of Color</i> |
| L13 | cl13 | mystery | Hitchens: <i>Footsteps in the Night</i> |
| M01 | cm01 | science_fiction | Heinlein: <i>Stranger in a Strange Land</i> |

| ID | File | Genre | Description |
|-----|------|-----------|---|
| N14 | cn15 | adventure | Field: <i>Rattlesnake Ridge</i> |
| P12 | cp12 | romance | Callaghan: <i>A Passion in Rome</i> |
| R06 | cr06 | humor | Thurber: <i>The Future, If Any, of Comedy</i> |

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(files=['cg22'])
['Does', 'our', 'society', 'have', 'a', 'runaway', ',', ...]
>>> brown.sents(categories=['news', 'editorial', 'reviews'])
[[['The', 'Fulton', 'County'], ['The', 'jury', 'further'], ...], ...]
```

We can use the Brown Corpus to study 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:

```
>>> 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
```

Note

Your Turn: Choose a different section of the Brown Corpus, and adapt the above method 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. To save re-typing, we can put the above code into a function, and use the function several times over. (We discuss functions in more detail in [Section 2.3](#).) However, there is an even better way, using NLTK's support for conditional frequency distributions ([Section 2.2](#)), as follows:

```
>>> cfd = nltk.ConditionalFreqDist((g,w)
...         for g in brown.categories()
...         for w in brown.words(categories=g))
>>> 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*, suggesting a focus on the future, while the most frequent modal in the romance genre is *could*, suggesting a focus on possibilities.

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" (for training and testing algorithms that automatically detect the topic of a document, as we will explore further in [Chapter 5](#)).

```
>>> from nltk.corpus import reuters
>>> reuters.files()
['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 name or a list of names.

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

```
>>> 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', ...]
```

Note

Many other English text corpora are provided with NLTK. For a list see [Appendix D.1](#). For more examples of how to access NLTK corpora, please consult the online guide at <http://nltk.org/doc/guides/corpora.html>.

US Presidential Inaugural Addresses

In [section 1.1](#), we looked at the US Presidential Inaugural Addresses corpus, but treated it as a single text. The graph in [Figure 1.1](#), used word offset as one of the axes, but this is difficult to interpret. 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.files()
['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', ...]
>>> [file[:4] for file in inaugural.files()]
['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 file name, we extracted the first four characters, using `file[:4]`.

Let's look at how the words *America* and *citizen* are used over time. The following code will count similar words, such as plurals of these words, or the word *Citizens* as it would appear at the start of a sentence (how?). The result is shown in [Figure 2.1](#).

```
>>> cfd = nltk.ConditionalFreqDist((target, file[:4])
...     for file in inaugural.files())
...     for w in inaugural.words(file))
```

```
...             for target in ['america', 'citizen']
...                 if w.lower().startswith(target))
>>> cfd.plot()
```

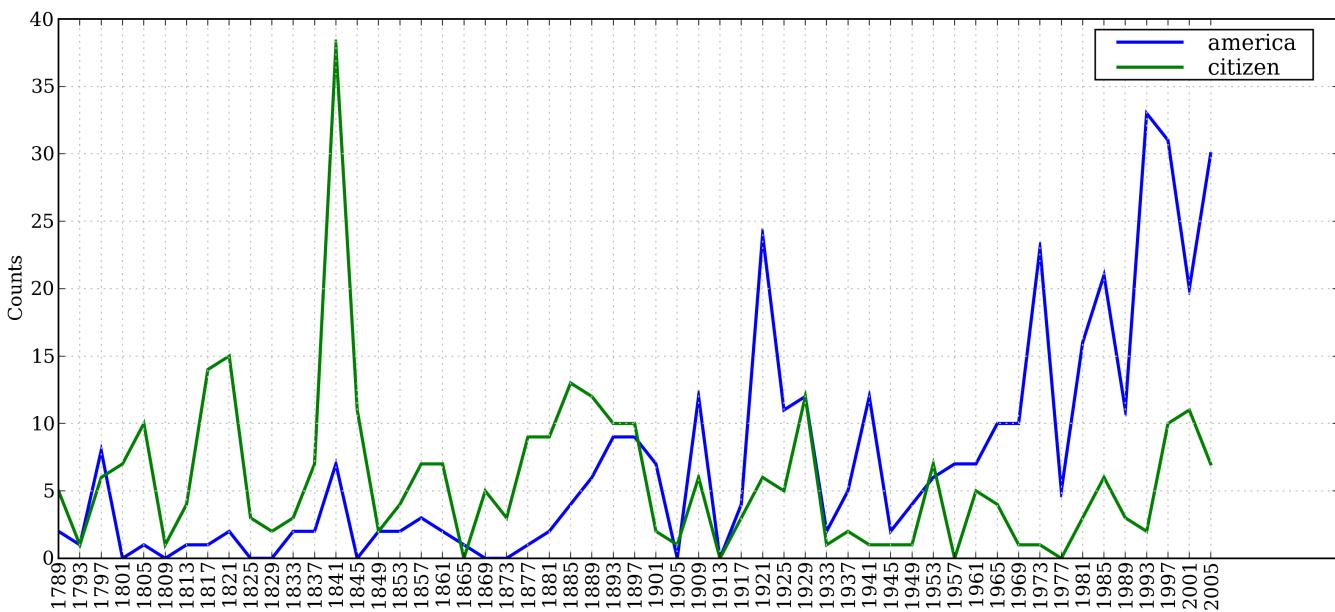


Figure 2.1: Conditional Frequency Distribution for Two Words in the Inaugural Address Corpus

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 [Appendix B](#)).

```
>>> 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.udhr.files()
('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', ...]
>>> nltk.corpus.indian.words('hindi.pos')
['\xe0\xaa\xae\x0\xaa\x5\x82\xe0\xaa\xb0\xe0\xaa\x5\x8d\xe0\xaa\x3',
'\xe0\xaa\xaa\xe0\xaa\x5\x8d\xe0\xaa\xb0\xe0\xaa\x4\xaa\xe0\xaa\xbf\xe0\xaa\xac\xe0\xaa\x82\xe0\xaa\x4\xaa7',
```

The last of these corpora, `udhr`, contains the Universal Declaration of Human Rights in over 300 languages. (Note that the names of the files in this corpus include information about character encoding, and for now we will stick with texts in ISO Latin-1, or ASCII)

Let's use a conditional frequency distribution to examine the differences in word lengths, for a selection of languages included in this corpus. The output is shown in [Figure 2.2](#) (run the program yourself to see a color plot).

```
>>> 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)
```

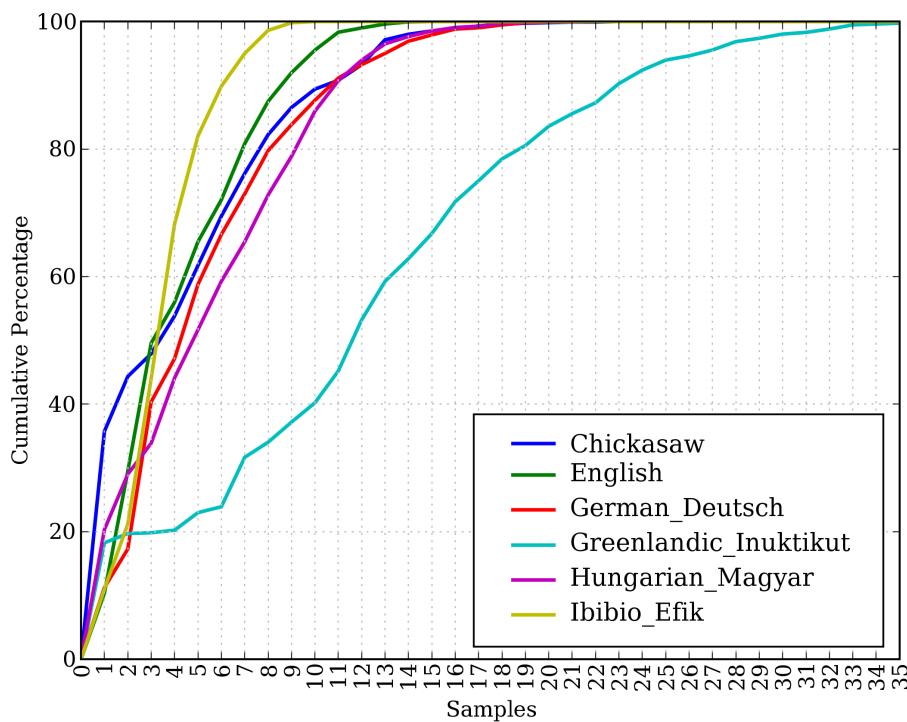


Figure 2.2: Cumulative Word Length Distributions for Several Languages

Note

Your Turn: Pick a language of interest in `udhr.files()`, 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 no government or industrial support for developing language resources, and individual efforts are piecemeal and hard to discover or re-use. Some languages have no established writing system, or are endangered. A good place to check is the search service of the *Open Language Archives Community*, at <http://www.language-archives.org/>. This service indexes the catalogs of dozens of language resource archives and publishers.

Note

The most complete inventory of the world's languages is *Ethnologue*, <http://www.ethnologue.com/>.

Text Corpus Structure

The corpora we have seen exemplify a variety of common corpus structures, 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, since a text can be relevant to more than one topic. Occasionally, text collections have temporal structure, news collections being the most common.

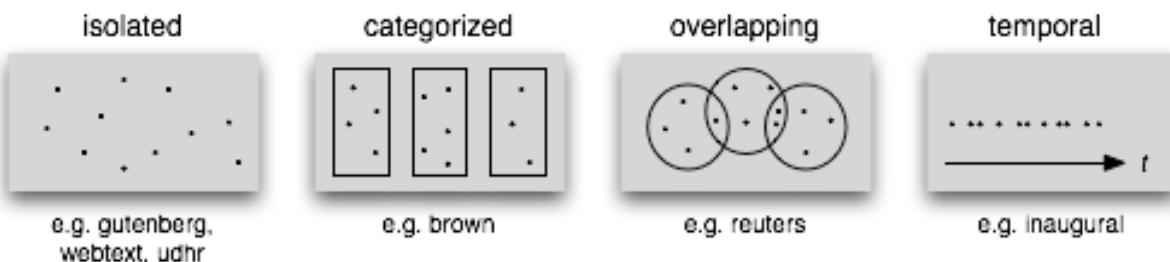


Figure 2.3: Common Structures for Text Corpora (one point per text)

NLTK's corpus readers support efficient access to a variety of corpora, and can easily be extended to work with new corpora [REF]. [Table 2.2](#) lists the basic methods provided by the corpus readers.

Table 2.2:

Basic Methods Defined in NLTK's Corpus Package

| Example | Description |
|----------------------------|---|
| files() | the files of the corpus |
| categories() | the categories of the corpus |
| abspath(file) | the location of the given file on disk |
| words() | the words of the whole corpus |
| words(files=[f1, f2, f3]) | the words of the specified files |
| words(categories=[c1, c2]) | the words of the specified categories |
| sents() | the sentences of the specified categories |
| sents(files=[f1, f2, f3]) | the sentences of the specified files |
| sents(categories=[c1, c2]) | the sentences of the specified categories |

Note

For more information about NLTK's Corpus Package, type `help(nltk.corpus.reader)` at the Python prompt, or see <http://nltk.org/doc/guides/corpus.html>. You will probably have other text sources, stored in files on your computer or accessible via the web. We'll discuss how to work with these in [Chapter 3](#).

Loading your own Corpus

If you have a collection of text files that you would like to access using the above methods, you can easily load them with the help of NLTK's `PlaintextCorpusReader` as follows:

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

The second parameter of the `PlaintextCorpusReader` can be a list of file pathnames, like `['a.txt', 'test/b.txt']`, or a pattern that matches all file pathnames, like `'[abc]/.*\.txt'` (see [Section 3.3](#) for information about regular expressions).

2.2 Conditional Frequency Distributions

We introduced frequency distributions in [Chapter 1](#), and 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. 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 to enable study of 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.

| Condition: News | | Condition: Romance | |
|-----------------|--|--------------------|--|
| the | | | |
| cute | | | |
| Monday | | | |
| could | | | |
| will | | | |

Figure 2.4: Counting Words Appearing in a Text Collection (a conditional frequency distribution)

Conditions and Events

As we saw in [Chapter 1](#), a frequency distribution counts observable events, such as the appearance of words in a text. A conditional frequency distribution needs to pair each such event with a condition. So instead of processing a text (a sequence of words), we have to process a sequence of pairs:

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

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

[TUPLES]

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.

```
>>> cfd = nltk.ConditionalFreqDist((g,w)
...                                     for g in brown.categories()
...                                     for w in brown.words(categories=g))
```

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

```
>>> genre_word = [(g,w) for g in ['news', 'romance'] for w in brown.words(categories=g)]
>>> len(genre_word)
170576
```

So pairs at the beginning of the list `genre_word` will be of the form (`'news'`, `word`) while those at the end will be of the form (`'romance'`, `word`). (Recall that `[-4:]` gives us a slice consisting of the last four items of a sequence.)

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

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

```
>>> cf = nltk.ConditionalFreqDist(genre_word)
>>> cf
<ConditionalFreqDist with 2 conditions>
>>> cf.conditions()
```

```
[ 'news', 'romance' ]
```

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

```
>>> cfd['news']
<FreqDist with 100554 samples>
>>> cfd['romance']
<FreqDist with 70022 samples>
>>> 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
```

Apart from combining two or more frequency distributions, and being easy to initialize, a `ConditionalFreqDist` provides some useful methods for tabulation and plotting. We can optionally specify which conditions to display with a `conditions=` parameter. When we omit it, we get all the conditions.

Note

Your Turn: 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 `cfд.tabulate(samples=days)`. Now try the same thing using `plot` in place of `tabulate`.

Other Conditions

The plot in [Figure 2.2](#) is based on a conditional frequency distribution where the condition is the name of the language and the counts being plotted are derived from word lengths. It exploits the fact that the filename for each language is the language name followed by ``-Latin1`` (the character encoding).

```
>>> cfd = nltk.ConditionalFreqDist((lang, len(word))
...     for lang in languages
...     for word in udhr.words(lang + '-Latin1'))
```

The plot in [Figure 2.1](#) is based on a conditional frequency distribution where the condition is either of two words *america* or *citizen*, and the counts being plotted are the number of times the word occurs in a particular speech. It exploits the fact that the filename for each speech, e.g. `1865-Lincoln.txt` contains the year as the first four characters.

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

This code will generate the tuple `('america', '1865')` for every instance of a word whose lowercased form starts with "america" — such as "Americans" — in the file `1865-Lincoln.txt`.

Generating Random Text with Bigrams

We can use a conditional frequency distribution to create a table of bigrams (word pairs). (We introduced bigrams 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 [Figure 2.5](#), 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.

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

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

[Figure 2.5 \(random.py\)](#): Figure 2.5: Generating Random Text in the Style of Genesis

Summary

Table 2.3:

Methods Defined for NLTK's Conditional Frequency Distributions

| Example | Description |
|--|--|
| <code>cfdist = ConditionalFreqDist(pairs)</code> | create a conditional frequency distribution |
| <code>cfdist.conditions()</code> | alphabetically sorted list of conditions |
| <code>cfdist[condition]</code> | the frequency distribution for this condition |
| <code>cfdist[condition][sample]</code> | frequency for the given sample for this condition |
| <code>cfdist.tabulate()</code> | tabulate the conditional frequency distribution |
| <code>cfdist.plot()</code> | graphical plot of the conditional frequency distribution |
| <code>cfdist1 < cfdist2</code> | samples in <code>cfdist1</code> occur less frequently than in <code>cfdist2</code> |

2.3 More Python: Reusing Code

By this time you've probably retyped a lot of code. 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 multi-line 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:

```
msg = 'Monty Python'
```

Save this program in a file called `test.py`, then go to the `Run` menu, and select the command `Run Module`. The result in the main IDLE window should look like this:

```
>>> ===== RESTART =====
>>>
>>>
```

Now, where is the output showing the value of `msg`? The answer is that the program in `test.py` will show a value only if you explicitly tell it to, using the `print` statement. So add another line to `test.py` so that it looks as follows:

```
msg = 'Monty Python'
print msg
```

Select Run Module again, and this time you should get output that looks like this:

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

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, and consulting the interactive help facility. Once you're ready, you can paste the code (minus any >>> 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.

Note

Our inline code examples will continue to 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 above.

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. It usually has some inputs, also 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](#):

```
>>> def score(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 the above example, all the work of the function is done in the return statement. Here's an equivalent definition which does the same work using multiple lines of code. We'll change the parameter name to remind you that this is an arbitrary choice:

```
>>> def score(my_text_data):
...     word_count = len(my_text_data)
...     vocab_size = len(set(my_text_data))
...     richness_score = word_count / vocab_size
...     return richness_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. Notice also that defining a function like this produces no output. Functions do nothing until they are "called" (or "invoked").

Let's return to our earlier scenario, and actually define a simple plural function. The function plural() in [Figure 2.6](#) takes a singular noun and generates a plural form (one which is not always correct).

```
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'
    return word + 's'
```

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

[Figure 2.6 \(plural.py\)](#): Figure 2.6: Example of a Python function

(There is much more to be said about functions, but we will hold off until [Section 6.2](#).)

Modules

Over time you will find that you create a variety of useful little text processing functions, and you end up copy-pasting 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 any copying and pasting.

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')
fan
```

Our plural function has an error, and we'll need to fix it. This time, we won't produce another version, but instead we'll fix the existing one. Thus, at every stage, there is only one version of our plural function, and no confusion about which one we should use.

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**.

[Work in somewhere: In general, we use `import` statements when we want to get access to Python code that doesn't already come as part of core Python. This code will exist somewhere as one or more files. Each such file corresponds to a Python **module** — this is a way of grouping together code and data that we regard as reusable. When you write down some Python statements in a file, you are in effect creating a new Python module. And you can make your code depend on another module by using the `import` statement.]

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 folder / directory.)

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 a defined a text `my_text`, then `vocab = sorted(set(my_text))` builds the vocabulary of `my_text`, while `word_freq = FreqDist(my_text)` counts the frequency of each word in the text. Both of `vocab` and `word_freq` are simple lexical resources. Similarly, a concordance ([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 2.7](#).

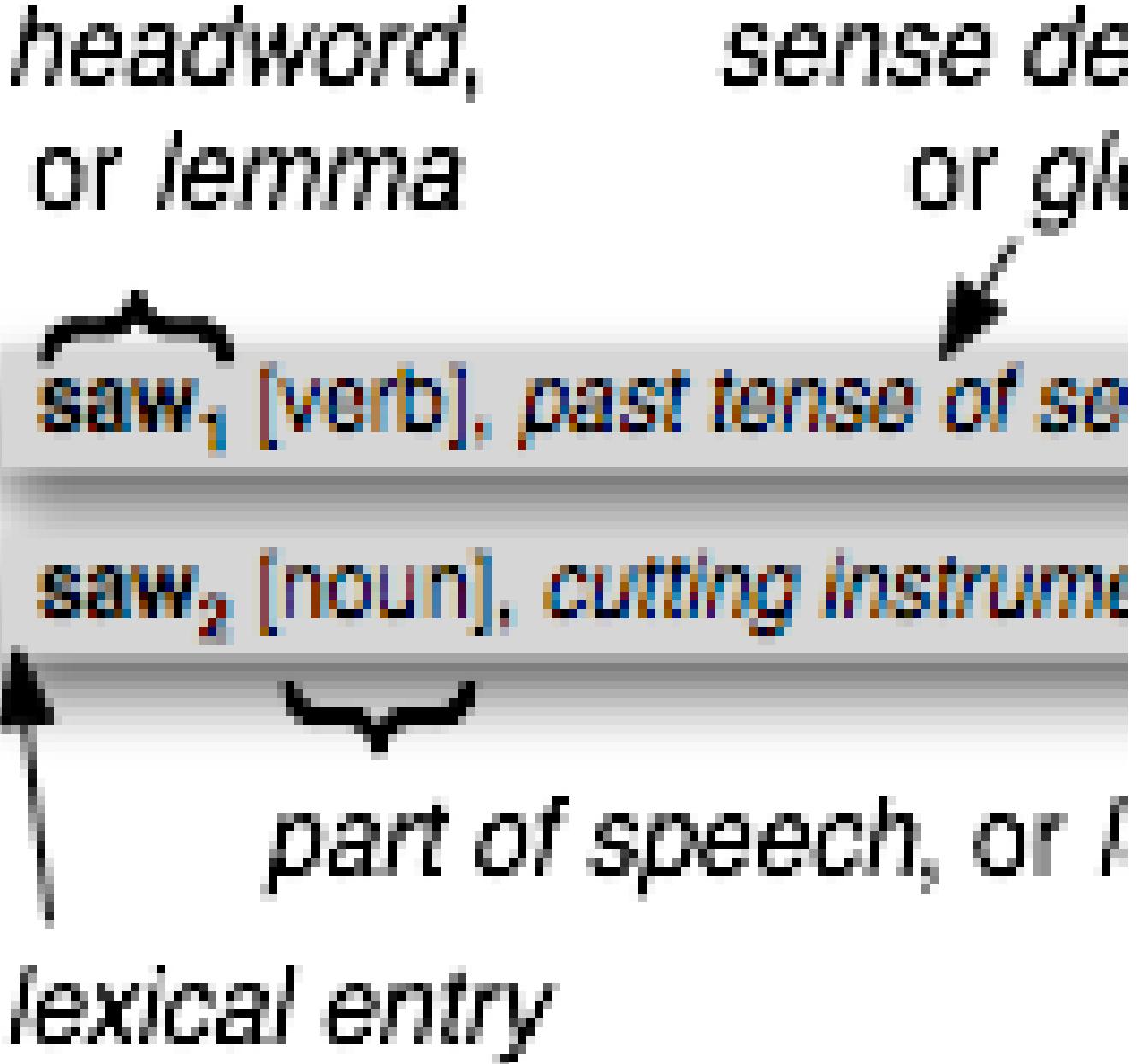


Figure 2.7: Lexicon Terminology

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 spell checkers. We can use it to find unusual or mis-spelt words in a text corpus, as shown in [Figure 2.8](#).

```
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'))
['abbeylead', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary',
 'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham',
 'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]
```

```
>>> unusual_words(nltk.corpus.nps_chat.words())
['aaaaaaaaaaaaaaa', 'aaahhhh', 'abou', 'abouted', 'abs', 'ack', 'acros',
'actualy', 'adduser', 'addy', 'adoted', 'adreniline', 'ae', 'afe', 'affari', 'afk',
'agaibn', 'agurlwithbigguns', 'ahah', 'ahahah', 'ahahaha', 'ahem', 'ahh', ...]
```

Figure 2.8 (unusual.py): Figure 2.8: Using a Lexical Resource to Filter a Text

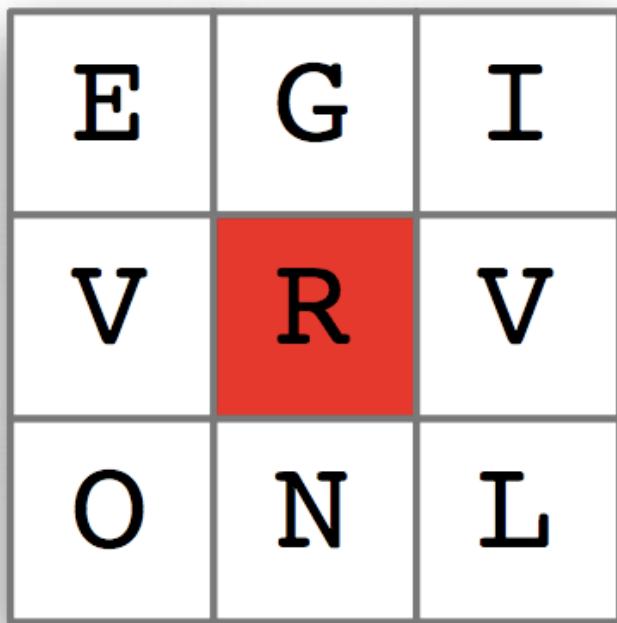
There is also a corpus of **stopwords**, that is, high-frequency words like *the*, *to* and *also* that we sometimes want to filter out of a document before further processing. Stopwords usually have little lexical content, and their presence in a text fail to distinguish it from other texts.

```
>>> 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 1.0 * 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.



How many words shown here must contain the word. No plus 21 words, good!

Figure 2.9: A Word Puzzle Known as "Target"

A wordlist is useful for solving word puzzles, such as the one in [Figure 2.9](#). Our program iterates through every word and, for each one, checks whether it meets the conditions. The obligatory letter and length constraint are easy to check (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 latter appear twice (here, the letter *v*). We use the `FreqDist` comparison method 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
...     and obligatory in w
...     and nltk.FreqDist(w) <= puzzle_letters]
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

Note

Your Turn: Can you think of an English word that contains *gnt*? Write Python code to find any such words in the wordlist.

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 which appear in both files, i.e. names that are ambiguous for gender:

```
>>> names = nltk.corpus.names
>>> names.files()
('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 2.10](#), produced by the following code:

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

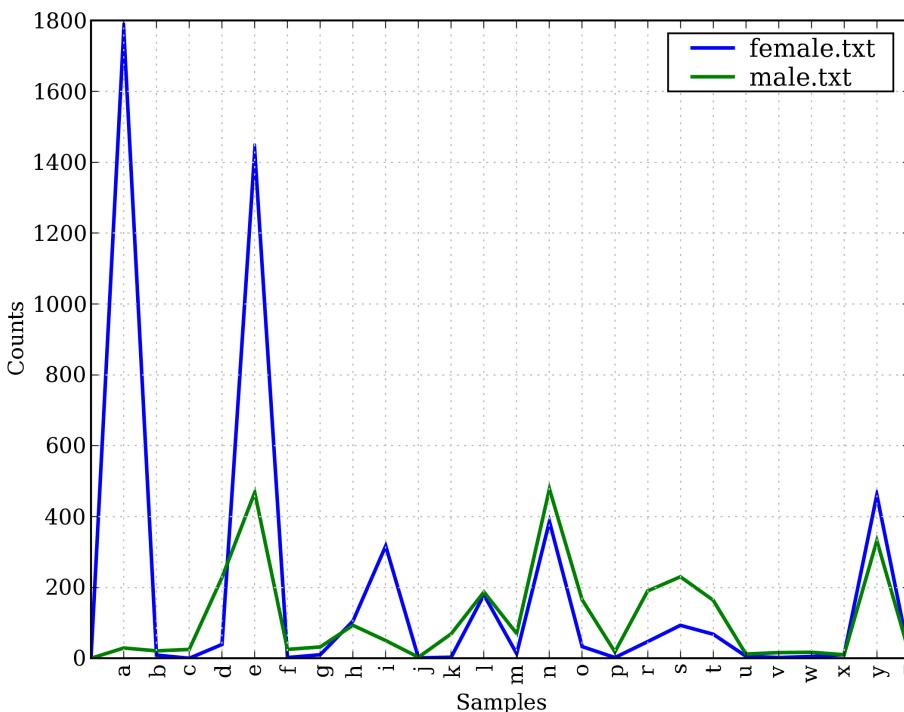


Figure 2.10: Frequency of Final Letter of Female vs Male Names

A Pronouncing Dictionary

As we have seen, the entries in a wordlist lack internal structure — they are just words. 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 US 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 US 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. 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:
...     if len(pron) == 3:
...         ph1, ph2, ph3 = pron
...         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 above program scans the lexicon looking for entries whose pronunciation consists of three phones (`len(pron) == 3`). If the condition is true, we assign the contents of `pron` to three new variables `ph1`, `ph2` and `ph3`. Notice the unusual form of the statement which does that work: `ph1, ph2, ph3 = pron`.

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 spelt in several ways: *nics*, *niks*, *nix*, 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 [int(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', ...]
```

Note that this example has a user-defined function inside the condition of a list comprehension.

Rather than iterating over the whole dictionary, we can also access it by looking up particular words. (This uses Python's dictionary data structure, which we will study in [Section 4.3](#).)

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

We look up a dictionary by specifying its name, followed by a **key** (such as the word *fire*) inside square brackets:

`prondict['fire']`. If we try to look up a non-existent key, we get a `KeyError`, as we did when indexing a list with an integer that was too large. The word *blog* is missing from the pronouncing dictionary, so we tweak our version by assigning a value for this key (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.files()
('be', 'bg', 'bs', 'ca', 'cs', 'cu', 'de', 'en', 'es', 'fr', 'hr', 'it', 'la', 'mk',
'n1', '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.

```
>>> fr2en = swadesh.entries(['fr', 'en'])
>>> fr2en
[('je', 'I'), ('tu', 'vous'), ('you (singular)', 'thou'), ('il', 'he'), ('nous', 'we'), ...]
>>> 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, 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 will return to Python's dictionary data type `dict()` in [Section 4.3](#).) We can compare words in various Germanic and Romance languages:

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

Shoebox and Toolbox Lexicons

Perhaps the single most popular tool used by linguists for managing data is *Toolbox*, previously known as *Shoebox* (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, like `('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 12](#).

Note

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), 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 (8a). If we replace the word *motorcar* in (8a) by *automobile*, to get (8b), the meaning of the sentence stays pretty much the same:

- (8)
- 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**. Let's 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 the above 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 these help humans 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, and here's how to access them:

```
>>> wn.synset('car.n.01').lemmas
[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')
Lemma('car.n.01.automobile')
>>> wn.lemma('car.n.01.automobile').synset
Synset('car.n.01')
>>> wn.lemma('car.n.01.automobile').name
'automobile'
```

Unlike the words *automobile* and *motorcar*, the word *car* itself 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')]
```

Observe that there is a one-to-one correspondence between the synsets of car and the lemmas of car.

Note

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 we used above.

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 2.11](#). The edges between nodes indicate the hypernym/hyponym relation...

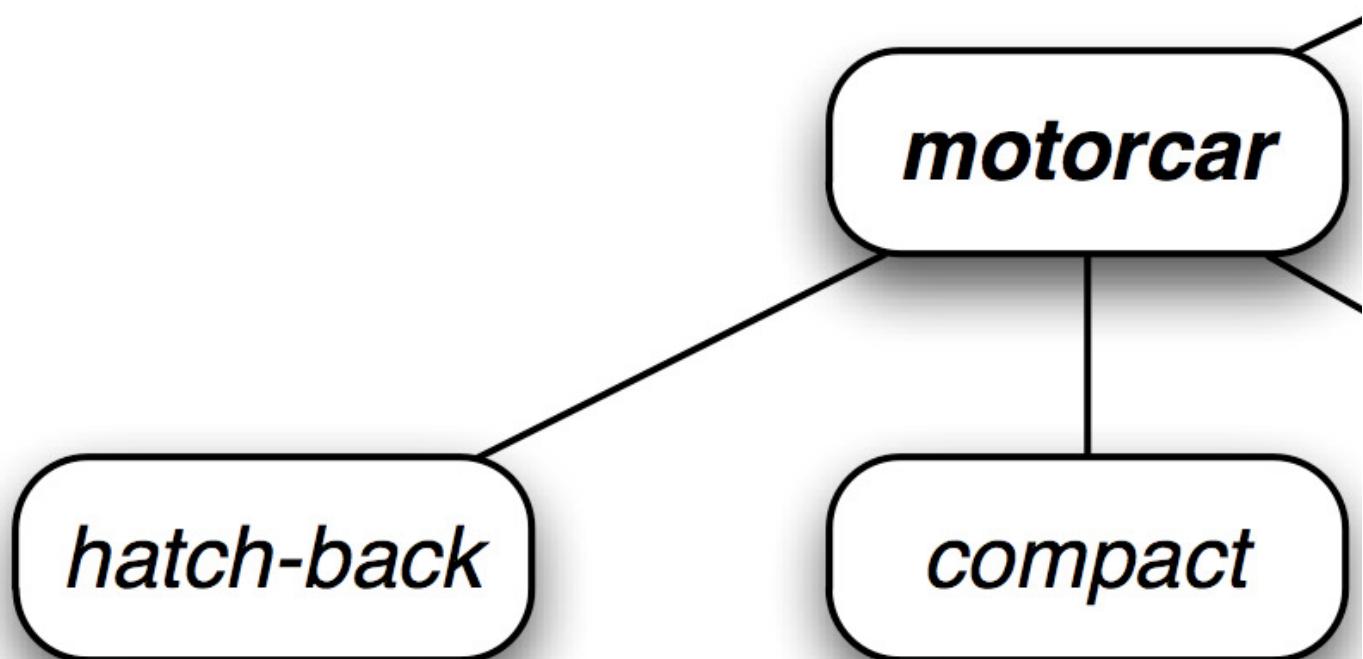


Figure 2.11: Fragment of WordNet Concept Hierarchy

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_wagon', '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 either as a vehicle or as a container.

```
>>> motorcar.hypernyms()
[Synset('motor_vehicle.n.01')]
>>> [synset.name for synset in motorcar.hypernym_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')]
```

Note

NLTK includes a convenient web-browser interface to WordNet `nltk.wordnet.browser()`

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; the `part_meronyms()`. The *substance* a tree is made of include *heartwood* and *sapwood*; the `substance_meronyms()`. A collection of trees forms a *forest*; 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')]

```

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 like *vehicle* will match documents containing specific terms like *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. If two synsets share a very specific hypernym — one that is low down in the hypernym hierarchy — they must be closely related.

```

>>> 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')
>>> orca.lowest_common_hypernyms(minke)
[Synset('whale.n.02')]
>>> orca.lowest_common_hypernyms(tortoise)
[Synset('vertebrate.n.01')]
>>> orca.lowest_common_hypernyms(novel)
[Synset('entity.n.01')]

```

Of course we know that *whale* is very specific, *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('whale.n.02').min_depth()
13
>>> wn.synset('vertebrate.n.01').min_depth()
8
>>> wn.synset('entity.n.01').min_depth()
0

```

The WordNet package includes a variety of sophisticated measures that incorporate this basic insight. For example, *path_similarity* assigns a score in the range 0–1, based on the shortest path that connects the concepts in the hypernym hierarchy (-1 is returned in those cases where a path cannot be found). Comparing a synset with itself will return 1.

```

>>> orca.path_similarity(minke)
0.14285714285714285
>>> orca.path_similarity(tortoise)
0.071428571428571425
>>> orca.path_similarity(novel)
0.04166666666666664

```

This is a convenient interface, and gives us the same relative ordering as before. Several other similarity measures are available (see `help(wn)`).

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.
- To find out about some variable `v` that you have created, type `help(v)` to read the help entry for this kind of object.
- Some functions are not available by default, but must be accessed using Python's `import` statement.

2.7 Further Reading (NOTES)

Natural Language Processing

Several websites have useful information about NLP, including conferences, resources, and special-interest groups, e.g. www.lt-world.org, www.aclweb.org, www.elsnet.org. The website of the *Association for Computational Linguistics*, at www.aclweb.org, contains an overview of computational linguistics, including copies of introductory chapters from recent textbooks. Wikipedia has entries for NLP and its subfields (but don't confuse natural language processing with the other NLP: neuro-linguistic programming.) The new, second edition of *Speech and Language Processing*, is a more advanced textbook that builds on the material presented here. Three books provide comprehensive surveys of the field: [[Cole, 1997](#)], [[Dale, Moisl, & Somers, 2000](#)], [[Mitkov, 2002](#)]. Several NLP systems have online interfaces that you might like to experiment with, e.g.:

- WordNet: <http://wordnet.princeton.edu/>
- Translation: <http://world.altavista.com/>
- ChatterBots: <http://www.loebner.net/Prizef/loebner-prize.html>
- Question Answering: <http://www.answerbus.com/>
- Summarization: <http://newsblaster.cs.columbia.edu/>

Python

[[Rossum & Drake, 2006](#)] is a Python tutorial by Guido van Rossum, the inventor of Python and Fred Drake, the official editor of the Python documentation. It is available online at <http://docs.python.org/tut/tut.html>. A more detailed but still introductory text is [[Lutz & Ascher, 2003](#)], which covers the essential features of Python, and also provides an overview of the standard libraries. A more advanced text, [[Rossum & Drake, 2006](#)] is the official reference for the Python language itself, and describes the syntax of Python and its built-in datatypes in depth. It is also available online at <http://docs.python.org/ref/ref.html>. [[Beazley, 2006](#)] is a succinct reference book; although not suitable as an introduction to Python, it is an excellent resource for intermediate and advanced programmers. Finally, it is always worth checking the official *Python Documentation* at <http://docs.python.org/>.

Two freely available online texts are the following:

- Josh Cogliati, *Non-Programmer's Tutorial for Python*, http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python/Contents
- Jeffrey Elkner, Allen B. Downey and Chris Meyers, *How to Think Like a Computer Scientist: Learning with Python* (Second Edition), <http://openbookproject.net/thinkCSpy/>

Learn more about functions in Python by reading Chapter 4 of [[Lutz & Ascher, 2003](#)].

Archives of the CORPORA mailing list.

[\[Woods, Fletcher, & Hughes, 1986\]](#)

LDC, ELRA

The online API documentation at <http://www.nltk.org/> contains extensive reference material for all NLTK modules.

Although WordNet was originally developed for research in psycholinguistics, it is widely used in NLP and Information Retrieval. WordNets are being developed for many other languages, as documented at <http://www.globalwordnet.org/>.

For a detailed comparison of wordnet similarity measures, see [\[Budanitsky & Hirst, 2006\]](#).

2.8 Exercises

1. ☀ How many words are there in `text2`? How many distinct words are there?
2. ☀ Compare the lexical diversity scores for humor and romance fiction in [Table 1.1](#). Which genre is more lexically diverse?
3. ☀ Produce a dispersion plot of the four main protagonists in *Sense and Sensibility*: Elinor, Marianne, Edward, Willoughby. What can you observe about the different roles played by the males and females in this novel? Can you identify the couples?
4. ☀ 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.
5. ☀ Create a variable `phrase` containing a list of words. Experiment with the operations described in this chapter, including addition, multiplication, indexing, slicing, and sorting.
6. ☀ 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`?
7. ☀ Using the Python interactive interpreter, experiment with the examples in this section. Think of a short phrase and represent it as a list of strings, e.g. `['Monty', 'Python']`. Try the various operations for indexing, slicing and sorting the elements of your list.
8. ☀ Investigate the holonym / meronym relations for some nouns. Note that there are three kinds (member, part, substance), so access is more specific, e.g., `wordnet.MEMBER_MERONYM`, `wordnet.SUBSTANCE_HOLONYM`.
9. ☀ The polysemy of a word is the number of senses it has. Using WordNet, we can determine that the noun *dog* has 7 senses with: `len(nltk.wordnet.N['dog'])`. Compute the average polysemy of nouns, verbs, adjectives and adverbs according to WordNet.
10. ☀ 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 if you try to access a non-existent entry, e.g. `d['xyz']`?
11. ☀ Try deleting an element from a dictionary, using the syntax `del d['abc']`. Check that the item was deleted.
12. ☀ Create a dictionary `e`, to represent a single lexical entry for some word of your choice. Define keys like `headword`, `part-of-speech`, `sense`, and `example`, and assign them suitable values.
13. ☀ Try the examples in this section, then try the following.
 1. Create a variable called `msg` and put a message of your own in this variable. Remember that strings need to be quoted, so you will need to type something like: `msg = "I like NLP!"`
 2. Now 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.
 3. Try various arithmetic expressions using this string, e.g. `msg + msg`, and `5 * msg`.
 4. Define a new string `hello`, and then try `hello + msg`. Change the `hello` string so that it ends with a space character, and then try `hello + msg` again.
14. ☀ Consider the following two expressions which have the same result. Which one will typically be more relevant in NLP? Why?
 1. `"Monty Python"[6:12]`
 2. `["Monty", "Python"][1]`
15. ☀ Define a string `s = 'colorless'`. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.
16. ☀ Try the slice examples from this section using the interactive interpreter. Then try some more of your own. Guess what the result will be before executing the command.
17. ☀ 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`.

18. ☀ 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?
19. ☀ We can also specify a "step" size for the slice. The following returns every second character within the slice:
`msg[6:11:2]`. It also works in the reverse direction: `msg[10:5:-2]` Try these for yourself, then experiment with different step values.
20. ☀ What happens if you ask the interpreter to evaluate `msg[::-1]`? Explain why this is a reasonable result.
21. ☀ Define a conditional frequency distribution over the Names corpus that allows you to see which initial letters are more frequent for males vs females (cf. [Figure 2.10](#)).
22. ☀ Use the corpus module to read `austen-persuasion.txt`. How many word tokens does this book have? How many word types?
23. ☀ 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.
24. ☀ 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?
25. ● Consider the following Python expression: `len(set(text4))`. State the purpose of this expression. Describe the two steps involved in performing this computation.
26. ● 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 which have quite different meanings across the two texts, such as *monstrous* in *Moby Dick* and in *Sense and Sensibility*?
27. ● Use `text9.index(??)` to find the index of the word *sunset*. By a process of trial and error, find the slice for the complete sentence that contains this word.
28. ● Using list addition, and the `set` and `sorted` operations, compute the vocabulary of the sentences `sent1 ... sent8`.
29. ● What is the difference between `sorted(set(w.lower() for w in text1))` and `sorted(w.lower() for w in set(text1))`? Which one will give a larger value? Will this be the case for other texts?
30. ● Write the slice expression to produce the last two words of `text2`.
31. ● Read the BBC News article: *UK's Vicky Pollards 'left behind'* <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>.
32. ● Assign a new value to `sent`, namely the sentence `['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']`, then write code to perform the following tasks:
 1. Print all words beginning with `'sh'`:
 2. Print all words longer than 4 characters.
33. ● 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?
34. ● What is the difference between the following two tests: `w.isupper(), not w.islower()`?
35. ● 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?
36. ● 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?
37. ● What is the branching factor of the noun hypernym hierarchy? (For all noun synsets that have hyponyms, how many do they have on average?)
38. ● Define a function `supergloss(s)` that takes a synset `s` as its argument and returns a string consisting of the concatenation of the glosses of `s`, all hypernyms of `s`, and all hyponyms of `s`.
39. ○ Review the mappings in [Table 4.4](#). Discuss any other examples of mappings you can think of. What type of information do they map from and to?
40. ● Write a program to find all words that occur at least three times in the Brown Corpus.
41. ● Write a program to generate a table of 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?
42. ● Modify the text generation program in [Figure 2.5](#) further, to do the following tasks:
 1. Store the `n` most likely words in a list `lwords` then randomly choose a word from the list using `random.choice()`.
 2. 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.
3. Now train your system using two distinct genres and experiment with generating text in the hybrid genre. Discuss your observations.
 43. ● Write a program to print the most frequent bigrams (pairs of adjacent words) of a text, omitting non-content words, in order of decreasing frequency.
 44. ● Write a program to create a table of word frequencies by genre, like the one given above for modals. Choose your own words and try to find words whose presence (or absence) is typical of a genre. Discuss your findings.
 45. ● Write a function that finds the 50 most frequently occurring words of a text that are not stopwords.
 46. ● Write a function `tf()` that takes a word and the name of a section of the Brown Corpus as arguments, and computes the text frequency of the word in that section of the corpus.
 47. ● Write a program to guess the number of syllables contained in a text, making use of the CMU Pronouncing Dictionary.
 48. ● Define a function `hedge(text)` which processes a text and produces a new version with the word '`'like'` between every third word.
 49. ★ **Zipf's Law:** Let $f(w)$ be the frequency of a word w in free text. Suppose that all the words of a text are ranked according to their frequency, with the most frequent word first. Zipf's law states that the frequency of a word type is inversely proportional to its rank (i.e. $f.r = k$, for some constant k). For example, the 50th most common word type should occur three times as frequently as the 150th most common word type.
 1. 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?
 2. Generate random text, e.g. using `random.choice("abcdefg ")`, taking care to include the space character. You will need to `import random` first. Use the string concatenation operator to accumulate characters into a (very) long string. Then tokenize this string, and generate the Zipf plot as before, and compare the two plots. What do you make of Zipf's Law in the light of this?
 50. ★ Modify the `generate_model()` function in [Figure 2.5](#) to use Python's `random.choose()` method to randomly pick the next word from the available set of words.
 51. ★ 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.
 52. ★ Use one of the predefined similarity measures to score the similarity of each of the following pairs of words. Rank the pairs in order of decreasing similarity. How close is your ranking to the order given here? (Note that this order was established experimentally by [\[Miller & Charles, 1998\]](#).)

::

car-automobile, gem-jewel, journey-voyage, boy-lad, coast-shore, asylum-madhouse, magician-wizard, midday-noon, furnace-stove, food-fruit, bird-cock, bird-crane, tool-implement, brother-monk, lad-brother, crane-implement, journey-car, monk-oracle, cemetery-woodland, food-rooster, coast-hill, forest-graveyard, shore-woodland, monk-slave, coast-forest, lad-wizard, chord-smile, glass-magician, rooster-voyage, noon-string.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

3 Processing Raw Text

The most important source of texts is undoubtedly the Web. Its 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 do the same kinds of analysis

we did with text corpora in earlier chapters?

3. What features of the Python programming language are needed to do this?

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.

Note

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

Note

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

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

The variable `raw` contains a string with 1,176,831 characters. This is the raw content of the book, including many details we are not interested in such as whitespace, line breaks and blank lines. Instead, we want to break it up 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. From now on we will call these **tokens**.

```
>>> text = nltk.wordpunct_tokenize(raw)
>>> type(text)
<class 'nltk.text.Text'>
>>> len(text)
255809
>>> text[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and', 'Punishment', ',', 'by']
```

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 like 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]
```

The `find()` and `rfind()` ("reverse find") functions help us get the right index values. Now the raw text 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 reality: 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 section on files below. However, if you're going to do this a lot, 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 reported 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.wordpunct_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 two generations. blonde hair is caused by a recessive gene . In order for a child to have blonde hair , it must have the gene on both sides of the family in the genome. there is a disadvantage of having that gene or by chance . They don ' t disappear because if having the gene was a disadvantage and I do not think

Note

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

Processing Google Results

[how to extract google hits]

LanguageLog example for *absolutely*

Table 3.1:

Absolutely vs Definitely (Liberman 2005, LanguageLog.org)

| Google hits | <i>adore</i> | <i>love</i> | <i>like</i> | <i>prefer</i> |
|-------------------|--------------|-------------|-------------|---------------|
| <i>absolutely</i> | 289,000 | 905,000 | 16,200 | 644 |
| <i>definitely</i> | 1,460 | 51,000 | 158,000 | 62,600 |
| ratio | 198:1 | 18:1 | 1:10 | 1:97 |

Reading Local Files

Note

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 `doc.txt` inside the directory that IDLE offers in the pop-up dialogue box. Next, in the Python interpreter, open the file using `f = open('doc.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()` function 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 it in the usual way:

```
>>> file = nltk.data.find('corpora/gutenberg/melville-moby_dick.txt')
>>> raw = open(file, '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 — like PDF and MSWord — that can only be opened using specialized software. Third-party libraries such as `pypdf` and `pywin32` can be used to access these formats. Extracting text from multi-column documents can be particularly challenging. For once-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.

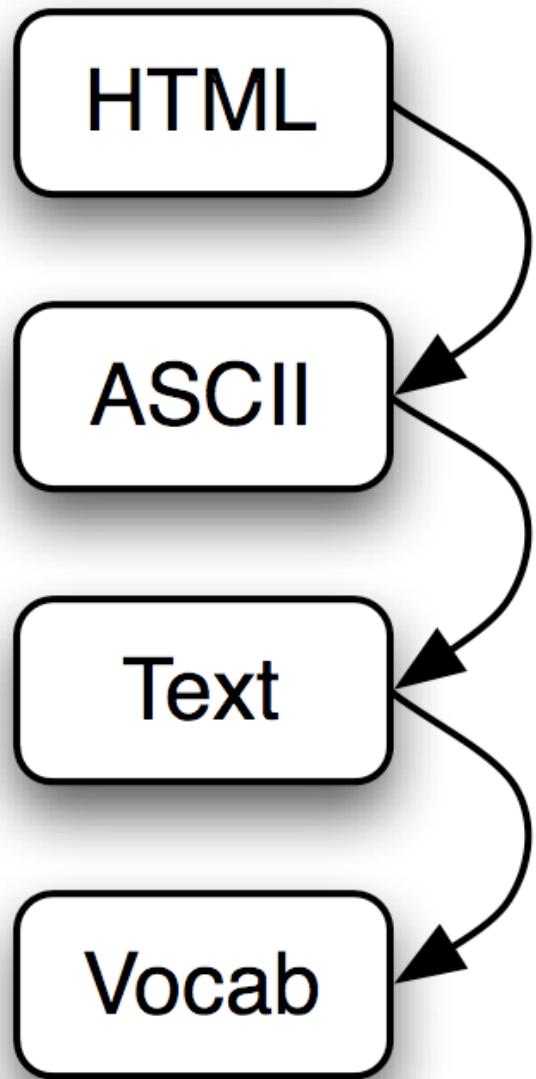
Getting User Input

Another source of text is a user interacting with our program. We can prompt the user to type a line of input using the Python function `raw_input()`. We can save that to a variable and 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.wordpunct_tokenize(s)), "words."
You typed 8 words.
```

Summary

[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 3.5](#)).



```

html = urlopen(ur
raw = nltk.clean_
raw = raw[750:235

tokens = nltk.wor
tokens = tokens[2
text = nltk.Text(

words = [w.lower(
vocab = sorted(se

```

Figure 3.1: The Processing Pipeline

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.wordpunct_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
```

You may also have noticed that our analogy between operations on strings and numbers works for multiplication and addition, but not subtraction or division:

```
>>> '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.

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 focussed 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

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

The `+` operation can be used with strings, and is known as **concatenation**. 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. The Python interpreter has no way of knowing that you want a space; it does *exactly* what it is told. Given the example of `+`, you might be able guess what multiplication will do:

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

Caution!

Be careful to distinguish between the string ' ', which is a single whitespace character, and '' , which is the empty string.

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 below:

```
>>> 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):

```
>>> 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. Using positive and negative indexes, we have 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), and: `5 = len(monty) - 7`.

```
>>> monty[-1]
'n'
>>> 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,
...
colorless green ideas sleep furiously
```

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 below). 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. For example, the following code accesses the substring starting at index 6, up to (but not including) index 10:

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

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 indices — 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[0:-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 can also find the position of a substring within a string, using `find()`:

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

Analyzing Strings

- character frequency plot, e.g. get text in some language using `language_x = nltk.corpus.udhr.raw(x)`, then construct its frequency distribution `fdist = FreqDist(language_x)`, then view the distribution with `fdist.keys()` and `fdist.plot()`.
- functions involving strings, e.g. determining past tense
- built-ins, `find()`, `rfind()`, `index()`, `rindex()`
- revisit string tests like `endswith()` from chapter 1

The Difference between Lists and Strings

Strings and lists are both kind 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 to 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, sentence, 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. So 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 3.6](#)). Conversely, when we want to write our results to a file, or to a terminal, we will usually format them as a string ([Section 3.8](#)).

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.

3.3 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 [Figure 1.4](#). Regular expressions give us a more powerful and flexible method for describing the character patterns we are interested in.

Note

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».

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() if w.islower()]
```

Using Basic Meta-Characters

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 8-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', ...]
```

Note

Your Turn: The caret symbol `^` matches the start of the word, just like the `$` matches the end of the word. What results do we get with the above example 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) using `len(w for w in text if re.search('e-?mail', w))`.

Ranges and Closures



Figure 3.2: T9: Text on 9 Keys

The **T9** system is used for entering text on mobile phones. Two or more words that are entered using the same sequence of keystrokes are known as **textonyms**. For example, both *hole* and *golf* are entered using 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 six 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.

Note

Your Turn: Look for some "finger-twisters", by searching for words that only use part of the number-pad. For example «`^[_-o]+$`» will match words that only use keys 4, 5, 6 in the center row, and «`^[_-fj-o]+$`» 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+$', w)]
['miiiiiiiiiiinnnnnnnnneeeeeeee', 'miiiiiinnnnnnnnneeeeeeee', 'mine',
'mmmmmmmmmiiiiiiinnnnnnnnneeeeeeee']
>>> [w for w in chat_words if re.search('^ha]+$', w)]
['a', 'aaaaaaaaaaaaaaa', 'aaahhh', 'ah', 'ahah', 'ahahah', 'ahh',
'ahahahaha', 'ahhh', 'ahhhh', 'ahhhhh', 'ahhhhhhhhhhhh', 'h', 'ha', 'haaa',
'hah', 'haha', 'hahaa', 'hahah', '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 *mmmmmm*. Note that the "`+`" and "`*`" symbols are sometimes referred to as **Kleene closures**, or simply **closures**.

The "`^`" operator has another function when it appears inside square brackets. For example «`[^aeiouAEIOU]`» matches any character other than a vowel. We can search the 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.

Note

Your Turn: Study the following examples and work out what the `\`, `{}` and `|` notations mean:

```
>>> 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}$', w)]
['10-day', '10-lap', '10-year', '100-share', '12-point', '12-year', ...]
>>> [w for w in wsj if re.search('^[a-z]{5,}-[a-z]{2,3}-[a-z]{1,6}$', 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', ...]
```

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 brace characters are used to specify the number of repeats of the previous item.

The meta-characters we have seen are summarized in [Table 3.2](#).

Table 3.2:

Basic Regular Expression Meta-Characters, Including Wildcards, Ranges and Closures

| Operator | Behavior |
|----------|---|
| . | Wildcard, matches any character |
| ^abc | Matches some pattern <i>abc</i> at the start of a string |
| abc\$ | Matches some pattern <i>abc</i> at the end of a string |
| [abc] | Matches a set of characters |
| [A-Z0-9] | Matches 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 <i>Kleene Closure</i>) |
| + | 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 <i>n</i> repeats where <i>n</i> is a non-negative integer |
| {m, n} | At least <i>m</i> and no more than <i>n</i> repeats (<i>m, n</i> optional) |
| (ab c) + | Parentheses that indicate the scope of the operators |

3.4 Useful Applications of Regular Expressions

The above examples all involved searching for words *w* that match some regular expression *regexp* using `re.search(regexp, w)`. Apart from checking if 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 = 'supercalifragulisticexpialidocious'
>>> re.findall('[aeiou]', word)
['u', 'e', 'a', 'i', 'a', 'u', 'i', 'i', 'e', 'i', 'a', 'i', 'o', 'i', 'o', 'u']
>>> len(re.findall('[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('[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), ...]
```

Note

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's interesting things to do with the pieces, like 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 *dclrt*, and *inalienable* becomes *inlnble*, retaining any initial or final vowel sequences. This regular expression matches initial vowel sequences, final vowel sequences, and all consonants; everything else is ignored. We use `re.findall()` to extract all the matching pieces, and `''.join()` to join them together (see [Section 3.8](#) for more about the join operation).

```
>>> regexp = '^([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 rcgnrn of the inhrrnt 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 rsld in brbrs acts whch hve outrgd the cnsnce 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('([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*.

If we want to be able to inspect the words behind the numbers in the above 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, e.g. `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('([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', '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', 'kasuari')`, `('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* word. 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 which 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('^(.*(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 for scoping the disjunction but not for selecting output, we have to add `:?` (just one of many arcane subtleties of regular expressions). Here's the revised version.

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

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

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

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

```
>>> re.findall('^(.*?)(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 stemming function and apply it to a whole text:

```
>>> def stem(word):
...     regexp = '^(.*)?(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.wordpunct_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 like *distribut* and *deriv*, but these are acceptable stems.

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).

```
>>> from nltk.corpus import gutenberg, nps_chat
>>> moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))
>>> moby.findall("<a>(<.*>)<man>")
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.search("<.*><.*><bro>")
you rule bro; telling you bro; u twizted bro
>>> chat.search("<1.*>{3,}")
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
```

Note

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.draw.finding_nemo()` which provides a graphical interface for exploring regular expressions.

3.5 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.

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 these handle a wide range of irregular cases. The Porter Stemmer strips affixes and knows about some special cases, e.g. that *lie* not *ly* is the stem of *lying*.

```
>>> 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 [Figure 3.3](#), which uses *object oriented* programming techniques that will be covered in Chapter REF, and string formatting techniques to be covered in [section 3.8](#)).

```
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 --
hit 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
```

[Figure 3.3 \(stemmer_indexing.py\)](#): Figure 3.3: Indexing a Text Using a Stemmer

Lemmatization

The WordNet lemmatizer only removes affixes if the resulting word is in its dictionary (and this additional checking process makes it slower). 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 lexical items.

[3.6 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 til 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, we need to match any number of spaces, tabs, or newlines.

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

Note

When using regular expressions that contain the backslash character, you should prefix the string with the letter `r` (meaning "raw"), which instructs the Python interpreter to treat them as literal backslashes.

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 [define] and also the complement of this class `\W`. So, we can split 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 [explain why]. We get the same result using `re.findall(r'\w+', raw)`, using a pattern that matches the words instead of the spaces.

```
>>> 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', '']
```

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 `\w`) 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. Let's generalize the `\w+` in the above 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', '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 above expression also included «[-.]» which causes the double hyphen, ellipsis, and open bracket to be tokenized separately.

[Table 3.3](#) lists the regular expression character class symbols we have seen in this section.

Table 3.3:

Regular Expression Symbols

| Symbol | Function |
|--------|--|
| \b | Word boundary (zero width) |
| \d | Any decimal digit (equivalent to [0-9]) |
| \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 like `re.findall`, except it is more efficient and it 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%
...     | \.\\.\\.
...     | [] [.,;'"?():-_`] # these are separate tokens
...
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

The `regexp_tokenize()` function has an optional `gaps` parameter. When set to `True`, the regular expression is applied to the gaps between tokens (cf `re.split()`).

Note

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

Dealing with Contractions

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*). [MORE]

3.7 Sentence Segmentation

[Explain how sentence segmentation followed by word tokenization can give different results to word tokenization on its own.]

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
```

In other cases, the text is only available 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 [Tibor & Jan, 2006], along with supporting data for English. Here is an example of its use in segmenting the text of a novel:

```
>>> 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, so very sad and
'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 place they will r
'It is because after they have\npassed Sloane Square they know that the next station must be\nVict
'Oh, their wild rapture!',
 'oh,\ntheir eyes like stars and their souls again in Eden, if the next\nstation were unaccountably
'"\\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.

3.8 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, such as 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.

Converting Between Strings and Lists (notes)

We specify the string to be used as the "glue", followed by a period, followed by the `join()` function.

```
>>> 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;.'
```

So `' '.join(silly)` means: take all the items in `silly` and concatenate them as one big string, using `' '` as a spacer between the items. (Many people find the notation for `join()` rather unintuitive.)

Notice that `join()` only works on a list of strings (what we have been calling a text).

Formatting Output

The output of a program is usually structured to make the information easily digestible by a reader. Instead of running some code and then manually inspecting the contents of a variable, we would like the code to tabulate some output. There are many ways we might want to format the output of a program. For instance, we might want to place the length value in parentheses *after* the word, and print all the output on a single line:

```
>>> 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), than (4)
```

However, this approach has some problems. First, the `print` statement intermingles variables and punctuation, making it a little difficult to read. Second, the output has spaces around every item that was printed. Third, we have to convert the length of the word to a string so that we can surround it with parentheses. A cleaner way to produce structured output uses Python's **string formatting expressions**. Before diving into clever formatting tricks, however, let's look at a really simple example. We are going to use a special symbol, `%s`, as a placeholder in strings. Once we have a string containing this placeholder, we follow it with a single `%` and then a value `v`. Python then returns a new string where `v` has been slotted in to replace `%s`:

```
>>> "I want a %s right now" % "coffee"
'I want a coffee right now'
```

In fact, 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:

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

We oversimplified things when we said that placeholders were of the form `%s`; in fact, this is a complex object, called a **conversion specifier**. This has to start with the `%` character, and ends with conversion character such as `s` or `d`. The `%s` specifier tells Python that the corresponding variable is a string (or should be converted into a string), while the `%d` specifier indicates that the corresponding variable should be converted into a decimal representation. The string containing conversion specifiers is called a **format string**.

Picking up on the `print` example that we opened this section with, here's how we can use two different kinds of conversion specifier:

```
>>> for word in saying:
...     print "%s (%d)," % (word, len(word)),
After (5), all (3), is (2), said (4), and (3), done (4), , (1), more (4), is (2), said (4), than (4)
```

To summarize, string formatting is accomplished with a three-part object having the syntax: `format % values`. The `format` section is a string containing format specifiers such as `%s` and `%d` that Python will replace with the supplied values. The `values` section of a formatting string is a parenthesized list containing exactly as many items as there are format specifiers in the `format` section. In the case that there is just one item, the parentheses can be left out.

In the above example, we used a trailing comma to suppress the printing of a newline. Suppose, on the other hand, that we want to introduce some additional newlines in our output. We can accomplish this by inserting the "special" character `\n` into the `print` string:

```
>>> for i, word in enumerate(saying[:6]):
...     print "Word = %s\nIndex = %s" % (word, i)
...
Word = After
Index = 0
Word = all
Index = 1
Word = is
Index = 2
Word = said
Index = 3
Word = and
Index = 4
Word = done
```

 Index = 5

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, e.g. given a dictionary `wordcount` consisting of words and their frequencies we could do:

```
>>> wordcount = {'cat':3, 'dog':4, 'snake':1}
>>> for word in sorted(wordcount):
...     print word, '->', wordcount[word], ';',
cat -> 3 ; dog -> 4 ; 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 formatting strings:

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

Lining Things Up

So far our formatting strings have contained specifications of fixed width, such as `%6s`, a string that is padded to width 6 and right-justified. We can include a minus sign to make it left-justified. 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:

```
>>> '%6s' % 'dog'
'dog'
>>> '%-6s' % 'dog'
'dog'
>>> width = 6
>>> '%-*s' % (width, 'dog')
'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:

```
>>> "accuracy for %d words: %2.4f%%" % (9375, 100.0 * 3205/9375)
'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. Note the clear separation between the language processing work, and the tabulation of results.

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

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

[Figure 3.4 \(modal_tabulate.py\)](#): Figure 3.4: Frequency of Modals in Different Sections of the Brown Corpus

Recall from the listing in [Figure 3.3](#) 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 customise the width of a column to be the smallest value required to fit all the words, using `width = min(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.

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

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 above. We can also do it using Python's backquote notation, which converts any object into a string. Let's write the total number of words to our file, before closing it.

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

3.9 Conclusion

In this chapter we saw that we can do a variety of interesting language processing tasks that focus solely on words. Tokenization turns out to be far more difficult than expected. No single solution works well across-the-board, and we must decide what

counts as a token depending on the application domain. We also looked at normalization (including lemmatization) and saw how it collapses distinctions between tokens. In the next chapter we will look at word classes and automatic tagging.

3.10 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`' [1] gives the value o. 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()`
- texts found on the web may contain unwanted material (such as headers, footers, markup), that need to be removed before we do any linguistic processing.
- a word token is an individual occurrence of a word in a particular context
- a word type is the vocabulary item, independent of any particular use of that item
- 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
- 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, and we can use `re.sub()` to replace substrings of one sort with another.
- 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'`.
- Normalization of words collapses distinctions, and is useful when indexing texts.

3.11 Further Reading (NOTES)

To learn about Unicode, see [1](#).

A.M. Kuchling. *Regular Expression HOWTO*, <http://www.amk.ca/python/howto/regex/>

For more examples of processing words with NLTK, please see the guides at <http://nltk.org/doc/guides/tokenize.html>, <http://nltk.org/doc/guides/stem.html>, and <http://nltk.org/doc/guides/wordnet.html>. A guide on accessing NLTK corpora is available at: <http://nltk.org/doc/guides/corpus.html>. Chapters 2 and 3 of [\[Jurafsky & Martin, 2008\]](#) contain more advanced material on regular expressions and morphology.

For languages with a non-Roman script, tokenizing text is even more challenging. For example, in Chinese text there is no visual representation of word boundaries. 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." The problem of tokenizing Chinese text is a major focus of SIGHAN, the ACL Special Interest Group on Chinese Language Processing <http://sighan.org/>.

Regular Expressions

There are many references for regular expressions, both practical and theoretical. [\[Friedl, 2002\]](#) is a comprehensive and detailed manual in using regular expressions, covering their syntax in most major programming languages, including Python.

For an introductory tutorial to using regular expressions in Python with the `re` module, see A. M. Kuchling, *Regular Expression HOWTO*, <http://www.amk.ca/python/howto/regex/>.

Chapter 3 of [\[Mertz, 2003\]](#) provides a more extended tutorial on Python's facilities for text processing with regular expressions.

<http://www.regular-expressions.info/> is a useful online resource, providing a tutorial and references to tools and other sources of information.

Unicode Regular Expressions: <http://www.unicode.org/reports/tr18/>

Regex Library: <http://regexlib.com/>

3.12 Exercises

1. ☀ Describe the class of strings matched by the following regular expressions.

1. `[a-zA-Z] +`
2. `[A-Z] [a-z] *`
3. `p[aeiou] {,2}t`
4. `\d+ (\.\d+)?`
5. `([^aeiou] [aeiou] [^aeiou]) *`
6. `\w+ | [^\w\$\s] +`

Test your answers using `re_show()`.

2. ☀ Write regular expressions to match the following classes of strings:

1. A single determiner (assume that *a*, *an*, and *the* are the only determiners).
2. An arithmetic expression using integers, addition, and multiplication, such as 2^*3+8 .

3. ☀ 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://nltk.org/').read()`.

4. ☀ 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.

1. Use `nltk.regexp_tokenize()` to create a tokenizer that tokenizes the various kinds of punctuation in this text. Use a single regular expression, with inline comments using the `re.VERBOSE` flag.
2. Use `nltk.regexp_tokenize()` to create a tokenizer that tokenizes the following kinds of expression: monetary amounts; dates; names of people and companies.

5. ☀ 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)]
```

6. ☀ Split `sent` on some other character, such as '`s`'.

7. ☀ We pointed out that when `phrase` is a list, `phrase.reverse()` returns a modified version of `phrase` rather than a new list. On the other hand, we can use the slice trick mentioned in the exercises for the previous section, `[::-1]` to create a new reversed list without changing `phrase`. Show how you can confirm this difference in behavior.

8. ☀ We have seen how to represent a sentence as a list of words, where each word is a sequence of characters. What does `phrasel[2][2]` do? Why? Experiment with other index values.

9. ☀ Write a `for` loop to print out the characters of a string, one per line.

10. ☀ What is the difference between calling `split` on a string with no argument or 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.)

11. ☀ Create a variable `words` containing a list of words. Experiment with `words.sort()` and `sorted(words)`. What is the difference?

12. ☀ Earlier, we asked you to use a text editor to create a file called `test.py`, containing the single line `msg = '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 `msg` 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.msg` at the prompt.

13. ● 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?
14. ● Write code to access a favorite webpage and extract some text from it. For example, access a weather site and extract the forecast top temperature for your town or city today.
15. ● Write a function `unknown()` that takes a URL as its argument, and returns a list of unknown words that occur on that webpage. 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.
16. ● 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.
17. ● Define a function `ghits()` that takes a word as its argument and builds a Google query string of the form `http://www.google.com/search?q=word`. Strip the HTML markup and normalize whitespace. Search for a substring of the form `Results 1 - 10 of about`, followed by some number *n*, and extract *n*. Convert this to an integer and return it.
18. ● The above example of extracting (name, domain) pairs from text does not work when there is more than one email address on a line, because the `+` operator is "greedy" and consumes too much of the input.
1. Experiment with input text containing more than one email address per line, such as that shown below. What happens?
 2. Using `re.findall()`, write another regular expression to extract email addresses, replacing the period character with a range or negated range, such as `[a-z]+` or `[^>]+`.
 3. Now try to match email addresses by changing the regular expression `.+` to its "non-greedy" counterpart, `.+?`
- ```
>>> s = """
... austen-emma.txt:hart@vmd.cso.uiuc.edu (internet) hart@uiucvmd (bitnet)
... austen-emma.txt:Internet (72600.2026@compuserve.com); TEL: (212-254-5093)
... austen-persuasion.txt:Editing by Martin Ward (Martin.Ward@uk.ac.durham)
... blake-songs.txt:Prepared by David Price, email ccx074@coventry.ac.uk
... """
```
19. ● 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+>`.
20. ● Write code to convert text into *hAck3r* again, this time 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`.
21. ● *Pig Latin* is a simple transliteration of English. 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*. [http://en.wikipedia.org/wiki/Pig\\_Latin](http://en.wikipedia.org/wiki/Pig_Latin)
1. Write a function to convert a word to Pig Latin.
  2. Write code that converts text, instead of individual words.

3. Extend it further to preserve capitalization, to keep `qu` together (i.e. so that `quiet` becomes `ietquay`), and to detect when `y` is used as a consonant (e.g. `yellow`) vs a vowel (e.g. `style`).
22. ● 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.
23. ● 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?
24. ● 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  $\mu_w$  to be the average number of letters per word, and  $\mu_s$  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:  $4.71 * \frac{\mu_w}{\mu_s} + 0.5 * \frac{\mu_s}{\mu_w} - 21.43$ . 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, while `nltk.corpus.brown.sents()` produces a sequence of sentences.
25. ● 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.
26. ● Process the list `saying` 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.
27. ● 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:
1. Split `silly` into a list of strings, one per word, using Python's `split()` operation, and save this to a variable called `bland`.
  2. Extract the second letter of each word in `silly` and join them into a string, to get `'eoldrrnnna'`.
  3. Combine the words in `bland` back into a single string, using `join()`. Make sure the words in the resulting string are separated with whitespace.
  4. Print the words of `silly` in alphabetical order, one per line.
28. ● 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`.
1. What happens when you look up a substring, e.g. `'inexpressible'.index('re')`?
  2. Define a variable `words` containing a list of words. Now use `words.index()` to look up the position of an individual word.
  3. Define a variable `silly` as in the exercise above. 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`.
29. ● Write code to abbreviate text by removing all the vowels. Define `sentence` to hold any string you like, then initialize a new string `result` to hold the empty string `''`. Now write a `for` loop to process the string, one character at a time, and append any non-vowel characters to the result string.
30. ● Write code to convert nationality adjectives like *Canadian* and *Australian* to their corresponding nouns *Canada* and *Australia*. (see [http://en.wikipedia.org/wiki/List\\_of\\_adjectival\\_forms\\_of\\_place\\_names](http://en.wikipedia.org/wiki/List_of_adjectival_forms_of_place_names))
31. ★ An interesting challenge for tokenization is words that have been split across a line-break. E.g. if `long-term` is split, then we have the string `long-\nterm`.
1. Write a regular expression that identifies words that are hyphenated at a line-break. The expression will need to

- include the `\n` character.
2. Use `re.sub()` to remove the `\n` character from these words.
32. ★ Read the Wikipedia entry on *Soundex*. Implement this algorithm in Python.
33. ★ Define a function `percent(word, text)` that calculates how often a given word occurs in a text, and expresses the result as a percentage.
34. ★ Obtain raw texts from two or more genres and compute their respective reading difficulty scores as in the previous exercise. E.g. compare ABC Rural News and ABC Science News (`nltk.corpus.abc`). Use `Punkt` to perform sentence segmentation.
35. ★ 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)
['aiui', 'eaiou', 'eouio', 'euoia', 'oauaio', 'uiieioa']
```

36. ★ 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 an open research problem.)

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

## 4 Categorizing and Tagging Words

Back in elementary school you learnt 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.

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**. 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.

### 4.1 Applications of Tagging

Automatic tagging has several applications. We have already seen an example of how to exploit tags in corpus analysis — we get a clear understanding of the distribution of *often* by looking at the tags of adjacent words. Automatic tagging also helps predict the behavior of previously unseen words. For example, if we encounter the word *scrobbbling* we can probably infer that

it is a verb, with the root *scrobble*, and likely to occur after forms of the auxiliary *to be* (e.g. *he was scrobbing*). Parts-of-speech are also used in speech synthesis and recognition. For example, *wind/NN*, as in *the wind blew*, is pronounced with a short vowel, whereas *wind/VB*, as in *to wind the clock*, is pronounced with a long vowel. Other examples can be found where the stress pattern differs depending on whether the word is a noun or a verb, e.g. *contest, insult, present, protest, rebel, suspect*. Without knowing the part-of-speech we cannot be sure of pronouncing the word correctly. Finally, there are many applications where automatic part-of-speech tagging is a vital step that feeds into later processing. We will look at many examples of this in later chapters.

## Evidence for Lexical Categories: Distributional Similarity

Before we go further, let's look for words based on similar distribution in a text. We will look up *woman* (a noun), *bought* (a verb), *over* (a preposition), and *the* (a determiner), using NLTK's `Text.similar()` function:

```
>>> text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())
>>> text.similar('woman')
Building word-context index...
man number fact end time world use kind state matter house result way
group part day rest sense couple be
>>> text.similar('bought')
able been made found used was had said have that given in expected as
told put taken got seen done
>>> text.similar('over')
of in to on at for was is with that from and into by all as out up back the
>>> text.similar('the')
a his this that it their one her an all in its any which our some he
these my be
```

This function takes a word *w*, finds all contexts *w1w w2*, then finds all words *w'* that appear in the same context, i.e. *w1w'w2*. (You can find the implementation online at <http://nltk.org/nltk/text.py>)

Observe that searching for *woman* finds *nouns*; searching for *bought* finds *verbs*; searching for *over* generally finds *prepositions*; searching for *the* finds several *determiners*.

These groups of words are so important that they have several names, all in common use: **word classes**, **lexical categories**, and **parts of speech**. We'll use these names interchangeably.

### Note

Remember that our program samples assume you begin your interactive session or your program with: `import nltk, re, pprint`

## 4.2 Tagged Corpora

### 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 4.1](#)).

**Table 4.1:**

Simplified Part-of-Speech Tagset

| Tag | Meaning      | Examples                                    |
|-----|--------------|---------------------------------------------|
| ADJ | adjective    | <i>new, good, high, special, big, local</i> |
| ADV | adverb       | <i>really, already, still, early, now</i>   |
| CNJ | conjunction  | <i>and, or, but, if, while, although</i>    |
| DET | determiner   | <i>the, a, some, most, every, no</i>        |
| EX  | existential  | <i>there, there's</i>                       |
| FW  | foreign word | <i>dolce, ersatz, esprit, quo, maitre</i>   |

| Tag | Meaning            | Examples                                   |
|-----|--------------------|--------------------------------------------|
| MOD | modal verb         | <i>will, can, would, may, must, should</i> |
| N   | noun               | <i>year, home, costs, time, education</i>  |
| NP  | proper noun        | <i>Alison, Africa, April, Washington</i>   |
| NUM | number             | <i>twenty-four, fourth, 1991, 14:24</i>    |
| PRO | pronoun            | <i>he, their, her, its, my, I, us</i>      |
| P   | preposition        | <i>on, of, at, with, by, into, under</i>   |
| TO  | the word <i>to</i> | <i>to</i>                                  |
| UH  | interjection       | <i>ah, bang, ha, whee, hmpf, oops</i>      |
| V   | verb               | <i>is, has, get, do, make, see, run</i>    |
| VD  | past tense         | <i>said, took, told, made, asked</i>       |
| VG  | present participle | <i>making, going, playing, working</i>     |
| VN  | past participle    | <i>given, taken, begun, sung</i>           |
| WH  | wh determiner      | <i>who, which, when, what, where, how</i>  |

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', 'NN', '.', 'CNJ', 'PRO', 'ADV', 'VD', ...]
```

### Note

**Your Turn:** Plot the above frequency distribution 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.draw.pos_concordance()`. Use it to search for any combination of words and POS tags, e.g. N N N N, hit/VD, hit/VN, the ADJ man.

## 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 ./

However, other tagged corpus files represent their part-of-speech tags in different ways. NLTK's corpus readers provide a uniform interface to these various formats so that you don't have to be concerned with them. By contrast with the text extract shown above, the corpus reader for the Brown Corpus presents the data as follows:

```
>>> list(nltk.corpus.brown.tagged_words())[:25]
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'),
 ('Grand', 'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'),
 ('Friday', 'NR'), ('an', 'AT'), ('investigation', 'NN'), ...]
```

Part-of-speech tags have been converted to uppercase, since this has become standard practice since the Brown Corpus was published.

Whenever a corpus contains tagged text, it 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; please see [Appendix A](#) for a comprehensive list of tags for some popular tagsets. (Note that each NLTK corpus has a `README` file which may also have documentation on tagsets.) 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\x9a\x6\xae\xe0\x9a\x6\xb9\xe0\x9a\x6\xbf\xe0\x9a\x6\xb7\xe0\x9a\x7\x87\xe0\x9a\x6\xb0', 'NN'),
 ('\xe0\x9a\x6\xb8\xe0\x9a\x6\xa8\xe0\x9a\x7\x8d\xe0\x9a\x6\xa4\xe0\x9a\x6\xbe\xe0\x9a\x6\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()
[('E1', '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 4.1](#) shows the output of the demonstration code `nltk.corpus.indian.demo()`.

Bangla: কুঢ়মেরগুলি/'NN' আকার/'NN' বাংলা/'NNP' বা/'CC' ভারতদের/'NNP' ?/None  
ন্য/'JJ' ?/None এইচলদের/'NN' প্ৰচলতি/'JJ' কুঢ়ের/'NN' ঘৰ/'NN' নয়/'VM' ক্ৰি/'SYM'  
Hindi: पाकिस्तन/'NNP' को/'PREP' पूर्व/'JJ' प्रधानमंत्री/'NN' ब्रेनजटोर/'NNPC' भूट्टो/'NNP'  
पर/'PREP' लगे/'VFM' भ्रष्टाचार/'NN' के/'PREP' आपोपो/'NN' के/'PREP' खिलाफ/'PREP' भूट्टो/'NNP'  
द্বारा/'PREP' दायर/'NVB' की/'VFM' गई/'VAUX' याचिकা/'NN' की/'PREP' सुनवाई/'NN'  
সংগ্রহণ/'NN' কো/'PREP' বকিলোঁ/'NN' কী/'PREP' হড়তাল/'NN' কে/'PREP' কারণ/'PREP'  
স্থগিত/'JVB' কর/'VFM' দীর্ঘ/'VAUX' গাই/'VAUX' !/'PUNC'  
Marathi: ग्रामीण/'JJ' जिल्हाध्यक्ष/'NN' बालासहेब/'NNPC' भोसले/'NNP' यांच्या/'PRP' ?/None  
दयक्षतेबाली/'NN' पक्षाची/'NN' आज/'NN' बै?/?None क/'NN' ज्ञाली/'VM' .//'SYM'  
Telugu: ఫలాదులు/'NN' సుంచి/'PREP' వచ్చినవ్యులు/'NN' పు/'PREP' సాంక్షణ్యా/'NN'

**Figure 4.1:** POS-Tagged Data from Four Indian Languages

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 typically function on a sentence at a time.

## 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 4.2](#).

**Table 4.2:**

Syntactic Patterns involving some Nouns

| Word     | After a determiner                            | Subject of the verb                     |
|----------|-----------------------------------------------|-----------------------------------------|
| woman    | <i>the woman who I saw yesterday ...</i>      | the woman <i>sat down</i>               |
| Scotland | <i>the Scotland I remember as a child ...</i> | Scotland <i>has five million people</i> |

| Word         | After a determiner                                 | Subject of the verb                                     |
|--------------|----------------------------------------------------|---------------------------------------------------------|
| book         | <i>the book I bought yesterday ...</i>             | this book <i>recounts</i> the colonization of Australia |
| intelligence | <i>the intelligence displayed by the child ...</i> | Mary's intelligence <i>impressed</i> 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*, *eat* in [Table 4.3](#). In the context of a sentence, verbs typically express a relation involving the referents of one or more noun phrases.

**Table 4.3:**

Syntactic Patterns involving some Verbs

| Word | Simple          | With modifiers and adjuncts (italicized)                |
|------|-----------------|---------------------------------------------------------|
| fall | Rome fell       | Dot com stocks <i>suddenly</i> fell <i>like a stone</i> |
| eat  | Mice eat cheese | John ate the pizza <i>with gusto</i>                    |

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 which 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?

#### Note

**Your Turn:** Given the list of past participles specified by `cf2d['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 in 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.

#### Note

**Your Turn:** If you are uncertain about some of these parts of speech, study them using `nltk.draw.pos_concordance()`, or watch some of the *Schoolhouse Rock!* grammar videos available at YouTube, or consult the Further Reading section at the end of this chapter.

## Tuples

By convention in NLTK, a tagged token is represented using a Python **tuple**. Python tuples are just like lists, except for one important difference: tuples cannot be changed in place, for example by `sort()` or `reverse()`. In other words, like strings, they are immutable. Tuples are formed with the comma operator, and typically enclosed using parentheses. Like lists, tuples can be indexed and sliced:

```
>>> t = ('walk', 'fem', 3)
>>> t[0]
'walk'
>>> t[1:]
('fem', 3)
>>> t[0] = 'run'
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

A tagged token is represented using a tuple consisting of just two items. 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'), ... ('.', '.')]
```

## Unsimplified Tags

Let's find the most frequent nouns of each noun part-of-speech type. The program in [Figure 4.2](#) 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.

```
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']
```

[Figure 4.2 \(findtags.py\)](#): Figure 4.2: Program to Find the Most Frequent Noun Tags

When we come to constructing part-of-speech taggers later in this chapter, we will use the unsimplified tags.

## Exploring Tagged Corpora (NOTES)

We can continue the kinds of exploration of corpora we saw in previous chapters, but exploiting the 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_learned_tagged = brown.tagged_words(categories='learned', simplify_tags=True)
>>> tags = [b[1] for (a, b) in nltk.ibigrams(brown_learned_tagged) if a[0] == 'often']
```

```
>>> list(nltk.FreqDist(tags))
['VN', 'V', 'VD', 'DET', 'ADJ', 'ADV', 'P', 'CNJ', ',', 'TO', 'VG', 'WH', 'VBZ', '.']
```

Notice that the most high-frequency parts of speech following *often* are verbs. Nouns never appear in this position (in this particular corpus).

## 4.3 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 dictionary data type. In this section we look at dictionaries and see how they can represent a variety of language information, including parts of speech.

### Indexing Lists vs 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 4.3](#).

|   |         |
|---|---------|
| 0 | Call    |
| 1 | me      |
| 2 | Ishmael |
| 3 | .       |

**Figure 4.3:** List Look-up

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. Look-up using words is familiar to anyone who has used a dictionary. Some more examples are shown in [Figure 4.4](#).

Phone List

|       |      |
|-------|------|
| Alex  | x154 |
| Dana  | x642 |
| Kim   | x911 |
| Les   | x120 |
| Sandy | x124 |

Domain Name Resolution

|                 |              |
|-----------------|--------------|
| aclweb.org      | 128.231.23.4 |
| amazon.com      | 12.118.92.43 |
| google.com      | 28.31.23.124 |
| python.org      | 18.21.3.144  |
| sourceforge.net | 51.98.23.53  |

Word Frequency Table

|               |     |
|---------------|-----|
| computational | 25  |
| language      | 196 |
| linguistics   | 17  |
| natural       | 56  |
| processing    | 57  |

Figure 4.4: Dictionary Look-up

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 round as with a list. In general, we would like to be able to map between arbitrary types of information. [Table 4.4](#) lists a variety of linguistic objects, along with what they map.

Table 4.4:

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 4.4](#), 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['colorless'] = 'ADJ'
>>> pos['ideas'] = 'N'
>>> pos['sleep'] = 'V'
>>> pos['furiously'] = 'ADV'
```

So, for example, `pos['colorless'] = 'ADJ'` 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`. 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 indices, 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`.

```
>>> pos
{'furiously': 'ADV', 'ideas': 'N', 'colorless': 'ADJ', 'sleep': 'V'}
```

Here, the contents of the dictionary are shown as **key-value pairs**, separated by a colon. The order of the key-value pairs is different from the order in which they were originally entered; this is because dictionaries are not sequences but mappings (cf. [Figure 4.4](#)), and the keys are not inherently ordered.

Alternatively, to just find the keys, we can convert the dictionary to a list — or use the dictionary in a context where a list is expected, as the parameter of `sorted()` or in a `for` loop:

```
>>> list(pos)
['ideas', 'furiously', 'colorless', 'sleep']
>>> sorted(pos)
['colorless', 'furiously', 'ideas', 'sleep']
>>> [w for w in pos if w.endswith('s')]
['colorless', 'ideas']
```

### Note

When you type `list(pos)` you might see a different order to the one shown above. 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:

```
>>> 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()):
... 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 only get 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'] = '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 only be 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.

## Default Dictionaries

Since Python 2.5, a special kind of dictionary has been available, which can automatically create a default entry for a given key. (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 right kind of initial entry, e.g. `int` or `list`:

```
>>> frequency = nltk.defaultdict(int)
>>> frequency['colorless'] = 4
>>> frequency['ideas']
0
>>> pos = nltk.defaultdict(list)
>>> pos['sleep'] = ['N', 'V']
>>> pos['ideas']
[]
```

If we want to supply our parameter to create a initial value, we have to supply it as a function. Let's return to our part-of-speech example, and create a dictionary whose default value for any entry is '`N`'.

```
>>> pos = nltk.defaultdict(lambda: 'N')
>>> pos['colorless'] = 'ADJ'
>>> pos['blog']
'N'
```

### Note

The above example used a *lambda expression*, an advanced feature we will study in [section 6.2](#). For now you just need to know that `lambda: 'N'` creates a function, and when we call this function it produces the value '`N`':

```
>>> f = lambda: 'N'
>>> f()
'N'
```

## Incrementally Updating a Dictionary

We can employ dictionaries to count occurrences, emulating the method for tallying words shown in [Figure 1.2](#) of [Chapter 1](#). 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.

```
>>> counts = nltk.defaultdict(int)
>>> for (word, tag) in brown_news_tagged:
... 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', ...]
```

---

**Figure 4.5 (dictionary.py): Figure 4.5: Incrementally Updating a Dictionary, and Sorting by Value**

The listing in [Figure 4.5](#) illustrates an important idiom for sorting a dictionary by its values, in order to show the words in decreasing order of frequency. The `sorted()` function takes `key` and `reverse` as parameters, and the required key is the second element of each word-tag pair. Although the second member of a pair is normally accessed with index `[1]`, this expression on its own (i.e. `key=[1]`) cannot be assigned as a parameter value in the function definition since `[1]` looks like a list containing the integer 1. An alternative is to set the value of `key` to be `itemgetter(1)`, a function which has the same effect as indexing into a tuple:

```
>>> pair = ('NP', 8336)
>>> pair[1]
8336
>>> itemgetter(1)(pair)
8336
```

There's a second useful programming idiom at the beginning of [Figure 4.5](#), 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:
 update my_dictionary[item_key] 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', 'abbreviatey',
 'abdominally', 'abhorrently', 'abidingly', 'abiogenetically', 'abiologically', ...]
>>> last_letters['zy']
['blazy', 'bleezy', 'blowzy', 'boozy', 'breezy', 'bronzy', 'buzzy', 'Chazy', 'cozy', ...]
```

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['aegilnrt']
['alerting', 'altering', 'integral', 'relating', 'triangle']
```

Since accumulating words like this is such a common task, NLTK provides a more convenient way of creating a `defaultdict(list)`:

```
>>> anagrams = nltk.Index(''.join(sorted(w)), w) for w in words)
```

#### Note

`nltk.FreqDist` is essentially a `defaultdict(int)` with extra support for initialization, sorting and plotting that are needed in language processing. Similarly `nltk.Index` is a `defaultdict(list)` with extra support for initialization.

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))
>>> for ((w1,t1), (w2,t2)) in nltk.bigrams(brown_news_tagged):
... pos[(t1,w2)][t2] += 1
...
>>> pos[('N', 'that')]
defaultdict(<type 'int'>, {'V': 10, 'CNJ': 145, 'WH': 112})
>>> pos[('DET', 'right')]
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). There is some new notation here (the `lambda`), and we will return to this in [chapter 6](#). For now, notice how we iterated over the bigrams of the tagged corpus, processing a pair of word-tag pairs for each iteration. Each time through the loop we updated our `pos` dictionary's entry for `(t1,w2)`, a tag and its *following* word. The entry for `('DET', 'right')` is itself a dictionary of counts. A POS tagger could use such information to decide to tag the word *right* as `ADJ` when it is preceded by a determiner.

## 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:

```
>>> [key for (key, value) in counts.items() if value == 16]
['call', 'sleepe', 'take', 'where', 'Your', 'Father', 'looke', 'owne']
```

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 incrementally add new values to the dictionary `pos2`, 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())
```

## Summary

Thanks to their versatility, Python dictionaries are extremely useful in most areas of NLP. We already made heavy use of dictionaries in [Chapter 1](#), since NLTK's `FreqDist` objects are just a special case of dictionaries for counting things. [Table 4.5](#) lists the most important dictionary methods you should know.

**Table 4.5:**

Summary of Python's Dictionary Methods

| Example             | Description                                                |
|---------------------|------------------------------------------------------------|
| <code>d = {}</code> | create an empty dictionary and assign it to <code>d</code> |

| Example                                    | Description                                           |
|--------------------------------------------|-------------------------------------------------------|
| <code>d[key] = value</code>                | assign a value to a given dictionary key              |
| <code>list(d), d.keys()</code>             | the list of keys of the dictionary                    |
| <code>sorted(d)</code>                     | the keys of the dictionary, sorted                    |
| <code>key in d</code>                      | test whether a particular key is in the dictionary    |
| <code>for key in d</code>                  | iterate over the keys of the dictionary               |
| <code>d.values()</code>                    | the list of values in the dictionary                  |
| <code>dict([(k1,v1), (k2,v2), ...])</code> | create a dictionary from a list of key-value pairs    |
| <code>d1.update(d2)</code>                 | add all items from <code>d2</code> to <code>d1</code> |
| <code>defaultdict(int)</code>              | a dictionary whose default value is zero              |

## 4.4 Automatic Tagging

In this and the following sections we will explore various ways to automatically add part-of-speech tags to some text. We'll begin by loading the data we will be using.

```
>>> from nltk.corpus import brown
>>> brown_news_tagged = brown.tagged(categories='news')
>>> brown_news_text = brown.words(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):

```
>>> nltk.FreqDist(tag for (word, tag) in brown_news_tagged).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.wordpunct_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:

```
>>> nltk.tag.accuracy(default_tagger, brown_news_tagged)
0.13089484257215028
```

Default taggers assign their tag to every single word, even words that have never been encountered before. As it happens, most new words are 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.

```

>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_news_text[3:4])
[('``', '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'), ..., ('.', 'NN')]
>>> nltk.tag.accuracy(regexp_tagger, brown_news_tagged)
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 re-specifying 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.

## The Lookup Tagger

A lot of high-frequency words do not have the `NN` tag. Let's find some of these words and their tags. 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".

```

>>> fd = nltk.FreqDist(brown_news_text)
>>> cfd = nltk.ConditionalFreqDist(brown_news_tagged)
>>> 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)
>>> nltk.tag.accuracy(baseline_tagger, brown.tagged_sents(categories='news'))
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 nearly half of all words correctly. Let's see what it does on some untagged input text:

```

>>> baseline_tagger.tag(brown_news_text[3])
[('``', ''), ('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`, a process known as backoff.

## Getting Better Coverage with Backoff

How do we combine these taggers? We want to use the lookup table first, and if it is unable to assign a tag, then use the default tagger. We do this by specifying the default tagger as a parameter to the lookup tagger. The lookup tagger will invoke the default tagger when it can't assign a tag itself.

```

>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags, backoff=nltk.DefaultTagger('NN'))
>>> nltk.tag.accuracy(baseline_tagger, brown_news_tagged)
0.58177695566561249

```

We can put all this together to write a simple (but somewhat inefficient) program to create and evaluate lookup taggers having a

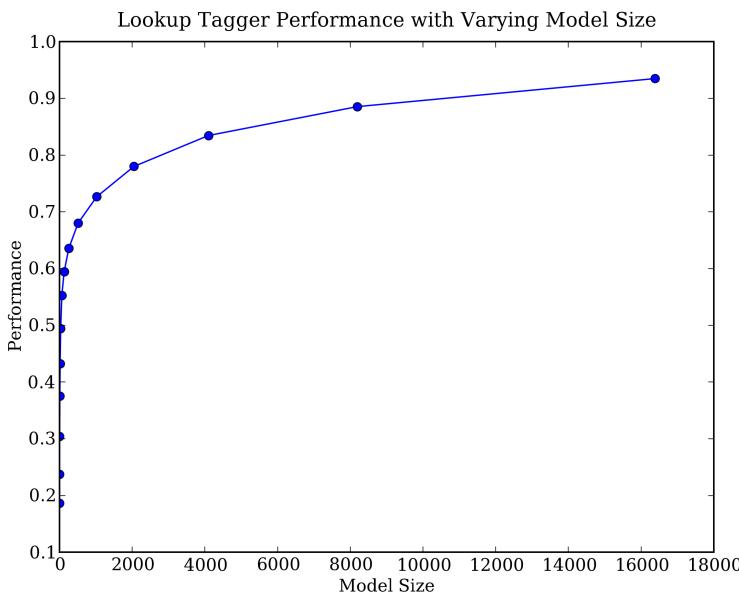
range of sizes, as shown in [Figure 4.6](#). We include a backoff tagger that tags everything as a noun. A consequence of using this backoff tagger is that the lookup tagger only has to store word/tag pairs for words other than nouns.

```
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 nltk.tag.accuracy(baseline_tagger, 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()
```

[Figure 4.6 \(baseline\\_tagger.py\)](#): [Figure 4.6](#): Lookup Tagger Performance with Varying Model Size



[Figure 4.7](#): Lookup Tagger

Observe 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; we will return to this in [Section 6.2](#)).

## Evaluation

In the above 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 1.4](#); 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 don't usually 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 which is 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.

### Note

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.

## 4.5 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 4.4](#)), 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, then evaluate:

```
>>> brown_news_tagged = brown.tagged_sents(categories='news')
>>> unigram_tagger = nltk.UnigramTagger(brown_news_tagged)
>>> sent = brown.sents(categories='news')[2007]
>>> unigram_tagger.tag(sent)
[('Various', None), ('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'), ('.', '.')]
>>> nltk.tag.accuracy(unigram_tagger, brown_news_tagged)
0.8550331165343994
```

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, 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 above 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 also be useless for tagging new text. Instead, we should split the data, training on 90% and testing on the remaining 10%:

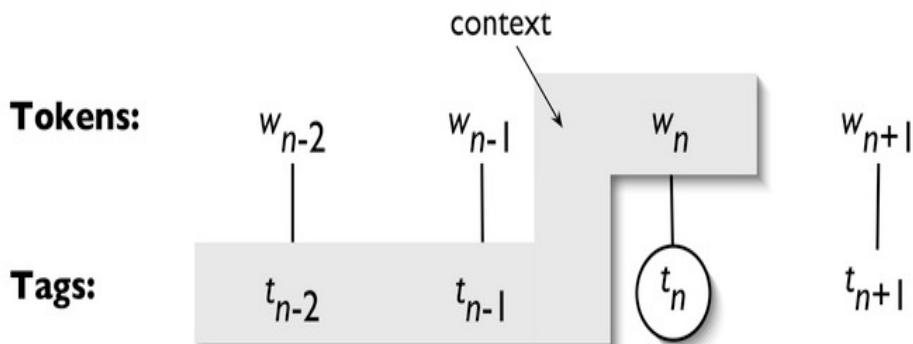
```
>>> size = int(len(brown_news_tagged) * 0.9)
>>> brown_news_train = brown_news_tagged[:size]
>>> brown_news_test = brown_news_tagged[size:]
>>> unigram_tagger = nltk.UnigramTagger(brown_news_train)
>>> nltk.tag.accuracy(unigram_tagger, brown_news_test)
0.77294926741752212
```

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.

### 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 only consider 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 4.8](#). 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 4.8](#), we have  $n=3$ ; that is, we consider 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.



**Figure 4.8:** Tagger Context

### Note

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(brown_news_train)
>>> bigram_tagger.tag(sent)
[('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(categories='news')[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:

```
>>> nltk.tag.accuracy(bigram_tagger, brown_news_test)
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 `regexp_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(brown_news_train, backoff=t0)
>>> t2 = nltk.BigramTagger(brown_news_train, backoff=t1)
>>> nltk.tag.accuracy(t2, brown_news_test)
0.81281770158477029
```

### Note

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, e.g.

`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 a 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*. During training, a unigram tagger will probably learn that this "word" 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. Full exploration of this method is left to the exercises.

## 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 re-use. 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(categories='news')
... 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) / float(cfd.N())
0.049297702067999993
```

Thus, one out of twenty trigrams is ambiguous [EXAMPLES]. 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 an empirical upper 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:

```
>>> def tag_list(tagged_sents):
... return [tag for sent in tagged_sents for (word, tag) in sent]
>>> def apply_tagger>tagger, corpus):
... return [tagger.tag(tag.untag(sent)) for sent in corpus]
>>> gold = tag_list(brown.tagged_sents(categories='editorial'))
>>> test = tag_list(apply_tagger(t2, brown.tagged_sents(categories='editorial')))
>>> print nltk.ConfusionMatrix(gold, test)
```

### [EXAMPLE OF CONFUSION MATRIX]

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. [MORE]

In general, observe that the tagging process simultaneously collapses distinctions (i.e., lexical identity is usually lost when all personal pronouns are tagged PRP), while introducing distinctions and removing ambiguities (e.g. deal tagged as VB or NN). This move facilitates classification and prediction. When we introduce finer distinctions in a tagset, we get better information about linguistic context, but we have to do more work to classify the current token (there are more tags to choose from). Conversely, with fewer distinctions (as with the simplified tagset), we have less work to do for classifying the current token, but less information about the context to draw on.

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 [Church, Young, & Bloothooft, 1996], the ambiguity can only be resolved 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, on chunk parsing, we shall see that it can.

## 4.6 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, large sparse arrays which 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, a statistical tagging method which performs very well using models that are only a tiny fraction of the size of n-gram taggers.

### Brill Tagging

Brill tagging is a kind of *transformation-based learning*, named after its inventor [REF]. 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 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 then fix up the details, with successively finer changes. Let's look at an example involving the following sentence:

- (9) 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 `NN`. [Table 4.6](#) illustrates this process, first tagging with the unigram tagger, then applying the rules to fix the errors.

**Table 4.6:**

Steps in Brill Tagging

|                |    |          |        |    |        |     |            |                |
|----------------|----|----------|--------|----|--------|-----|------------|----------------|
| <b>Phrase</b>  | to | increase | grants | to | states | for | vocational | rehabilitation |
| <b>Gold</b>    | TO | VB       | NNS    | IN | NNS    | IN  | JJ         | NN             |
| <b>Unigram</b> | TO | NN       | NNS    | TO | NNS    | IN  | JJ         | NN             |
| <b>Rule 1</b>  |    | VB       |        |    |        |     |            |                |
| <b>Rule 2</b>  |    |          |        | IN |        |     |            |                |
| <b>Output</b>  | 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 2-3 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.

### Using NLTK's Brill Tagger

[Figure 4.9](#) demonstrates NLTK's Brill tagger...

```
>>> nltk.tag.brill.demo
Training Brill tagger on 80 sentences...
Finding initial useful rules...
Found 6555 useful rules.
```

| S                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | F                                                                                                                                                                                           | r                                                                                                                                                                                                                                                                                                                                       | O  |                                                      | Score = Fixed - Broken                          |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|------------------------------------------------------|-------------------------------------------------|--------------|-----------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| c                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | i                                                                                                                                                                                           | o                                                                                                                                                                                                                                                                                                                                       | t  |                                                      | Fixed = num tags changed incorrect -> correct   |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| o                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | x                                                                                                                                                                                           | k                                                                                                                                                                                                                                                                                                                                       | h  |                                                      | Broken = num tags changed correct -> incorrect  |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| r                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | e                                                                                                                                                                                           | e                                                                                                                                                                                                                                                                                                                                       | e  |                                                      | Other = num tags changed incorrect -> incorrect |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| e                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    | d                                                                                                                                                                                           | n                                                                                                                                                                                                                                                                                                                                       | r  |                                                      |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| <hr/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |    |                                                      |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| 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' |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| <hr/>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |    |                                                      |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| >>> print(open("errors.out").read())                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |    |                                                      |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| <table border="1"> <thead> <tr> <th>left context</th><th>word/test-&gt;gold</th><th>right context</th></tr> </thead> <tbody> <tr><td>, in/IN the/DT guests/NNS /POS honor/NN ,/, the/DT NN ,/, the/DT speedway/NN DT speedway/NN hauled/VBD dway/NN hauled/VBD out/RP hauled/VBD out/RP four/CD P four/CD drivers/NNS ,/, NNS and/CC even/RB the/DT ter/IN the/DT race/NN ,/, s/NNS drooled/VBD like/IN schoolboys/NNP-&gt;NN</td><td>Then/NN-&gt;RB /VBD-&gt;POS speedway/JJ-&gt;NN haule/NN-&gt;VBD out/NNP-&gt;RP four/NNP-&gt;CD drivers/NNP-&gt;NNS crews/NN-&gt;NNS official/NNP-&gt;JJ After/VBD-&gt;IN Fortune/IN-&gt;NNP schoolboys/NNP-&gt;NNS cars/NN-&gt;NNS</td><td>,/, in/IN the/DT guests/N honor/NN ,/, the/DT speed haul/BB out/RP four/CD out/RP four/CD drivers/NN four/CD drivers/NNS ,/, c drivers/NNS ,/, crews/NNS ,/, crews/NNS and/CC even/CC even/RB the/DT off Indianapolis/NNP 500/CD a the/DT race/NN ,/, Fortun 500/CD executives/NNS dro over/IN the/DT cars/NNS a and/CC drivers/NNS ./.</td></tr> </tbody> </table> |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |    |                                                      |                                                 | left context | word/test->gold | right context | , in/IN the/DT guests/NNS /POS honor/NN ,/, the/DT NN ,/, the/DT speedway/NN DT speedway/NN hauled/VBD dway/NN hauled/VBD out/RP hauled/VBD out/RP four/CD P four/CD drivers/NNS ,/, NNS and/CC even/RB the/DT ter/IN the/DT race/NN ,/, s/NNS drooled/VBD like/IN schoolboys/NNP->NN | Then/NN->RB /VBD->POS speedway/JJ->NN haule/NN->VBD out/NNP->RP four/NNP->CD drivers/NNP->NNS crews/NN->NNS official/NNP->JJ After/VBD->IN Fortune/IN->NNP schoolboys/NNP->NNS cars/NN->NNS | ,/, in/IN the/DT guests/N honor/NN ,/, the/DT speed haul/BB out/RP four/CD out/RP four/CD drivers/NN four/CD drivers/NNS ,/, c drivers/NNS ,/, crews/NNS ,/, crews/NNS and/CC even/CC even/RB the/DT off Indianapolis/NNP 500/CD a the/DT race/NN ,/, Fortun 500/CD executives/NNS dro over/IN the/DT cars/NNS a and/CC drivers/NNS ./. |
| left context                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | word/test->gold                                                                                                                                                                             | right context                                                                                                                                                                                                                                                                                                                           |    |                                                      |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |
| , in/IN the/DT guests/NNS /POS honor/NN ,/, the/DT NN ,/, the/DT speedway/NN DT speedway/NN hauled/VBD dway/NN hauled/VBD out/RP hauled/VBD out/RP four/CD P four/CD drivers/NNS ,/, NNS and/CC even/RB the/DT ter/IN the/DT race/NN ,/, s/NNS drooled/VBD like/IN schoolboys/NNP->NN                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Then/NN->RB /VBD->POS speedway/JJ->NN haule/NN->VBD out/NNP->RP four/NNP->CD drivers/NNP->NNS crews/NN->NNS official/NNP->JJ After/VBD->IN Fortune/IN->NNP schoolboys/NNP->NNS cars/NN->NNS | ,/, in/IN the/DT guests/N honor/NN ,/, the/DT speed haul/BB out/RP four/CD out/RP four/CD drivers/NN four/CD drivers/NNS ,/, c drivers/NNS ,/, crews/NNS ,/, crews/NNS and/CC even/CC even/RB the/DT off Indianapolis/NNP 500/CD a the/DT race/NN ,/, Fortun 500/CD executives/NNS dro over/IN the/DT cars/NNS a and/CC drivers/NNS ./. |    |                                                      |                                                 |              |                 |               |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                         |

**Figure 4.9 (brill\_demo.py):** Figure 4.9: NLTK's Brill tagger

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.

## 4.7 The TnT Tagger

[NLTK contains a pure Python implementation of the TnT tagger `nltk.tag.tnt`, and also an interface to an external TnT tagger `nltk_contrib.tag.tnt`. These will be described in a later version of this chapter.]

## 4.8 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**). (Since the present participle and the gerund cannot be systematically distinguished, they are often tagged with the same tag, i.e. VBG in the Brown Corpus tagset).

### 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:

- (10)
- 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 that we are unfamiliar with. 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 **morpho-syntactic** 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:

- (11)
- 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.

- (12)
- 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.

- (13)
- 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 4.7](#).

**Table 4.7:**

Some morphosyntactic distinctions in the Brown tagset

| Form  | Category             | Tag |
|-------|----------------------|-----|
| go    | base                 | VB  |
| goes  | 3rd 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.

## 4.9 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`, `VB`,
- The process of automatically assigning parts of speech to words in text is called part-of-speech tagging, POS tagging, or just tagging.
- 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.
- 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 = {'furiouly': 'adv', 'ideas': 'n', 'colorless': 'adj'}`.
- Ngram 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 only see a tiny fraction of possible contexts.

## 4.10 Further Reading

[Recommended readings on lexical categories...]

[Appendix A](#) contains details of popular tagsets.

For more examples of tagging with NLTK, please see the tagging HOWTO on the NLTK website. Chapters 4 and 5 of [\[Jurafsky & Martin, 2008\]](#) contain more advanced material on n-grams and part-of-speech tagging.

There are several other important approaches to tagging involving *Transformation-Based Learning*, *Markov Modeling*, and *Finite State Methods*. (We will discuss some of these in [Chapter 5](#).) In [Chapter 7](#) we will see a generalization of tagging called *chunking* in which a contiguous sequence of words is assigned a single tag.

Part-of-speech tagging is just one kind of tagging, one that does not depend on deep linguistic analysis. 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 below. 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.

1. *Speech Synthesis Markup Language (W3C SSML)*: That `is` a `<emphasis>big</emphasis>` car!
2. *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")
3. *Morphological tagging, from the Turin University Italian Treebank*: E' `italiano`, come progetto e `realizzazione`, il primo (PRIMO ADJ ORDIN M SING) porto turistico dell' Albania .

Tagging exhibits several properties that are characteristic of natural language processing. First, tagging involves *classification*: words have properties; many words share the same property (e.g. `cat` and `dog` are both nouns), while some words can have multiple such properties (e.g. `wind` is a noun and a verb). Second, in tagging, disambiguation occurs via *representation*: we augment the representation of tokens with part-of-speech tags. Third, training a tagger involves *sequence learning from annotated corpora*. Finally, tagging uses *simple, general, methods* such as conditional frequency distributions and transformation-based learning.

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.

```
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
```

List of available taggers: <http://www-nlp.stanford.edu/links/statnlp.html>

NLTK's HMM tagger, `nltk.HiddenMarkovModelTagger`

- HMM and Brill Tagging are discussed in Sections 5.5 and 5.6 of [\[Jurafsky & Martin, 2008\]](#).

[\[Church, Young, & Blothooft, 1996\]](#)

## 4.11 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 if knowledge of the part-of-speech tags removes the ambiguity.
2. ☀ Working with someone else, take turns to pick 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. ● Write programs to process the Brown Corpus and find answers to the following questions:
  1. Which nouns are more common in their plural form, rather than their singular form? (Only consider regular plurals, formed with the `-s` suffix.)
  2. Which word has the greatest number of distinct tags. What are they, and what do they represent?
  3. List tags in order of decreasing frequency. What do the 20 most frequent tags represent?
  4. Which tags are nouns most commonly found after? What do these tags represent?
4. ● Explore the following issues that arise in connection with the lookup tagger:
  1. What happens to the tagger performance for the various model sizes when a backoff tagger is omitted?
  2. Consider the curve in [Figure 4.7](#); 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?
5. ● 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.)

6. ● Generate some statistics for tagged data to answer the following questions:
  1. What proportion of word types are always assigned the same part-of-speech tag?
  2. How many words are ambiguous, in the sense that they appear with at least two tags?
  3. What percentage of word *occurrences* in the Brown Corpus involve these ambiguous words?
7. ● Above we gave an example of the `nltk.tag.accuracy()` function. It has two arguments, a tagger and some tagged text, and it 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 accuracy score would be `0.5`. Can you figure out how the `nltk.tag.accuracy()` function works?
  1. A tagger takes a list of words as input, and produces a list of tagged words as output. However, `nltk.tag.accuracy()` is given correctly tagged text as its input. What must the `nltk.tag.accuracy()` function do with this input before performing the tagging?
  2. Once the supplied tagger has created newly tagged text, how would `nltk.tag.accuracy()` go about comparing it with the original tagged text and computing the accuracy score?
8. ☀ 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 [\(3d\)](#).
9. ● Write code to search the Brown Corpus for particular words and phrases according to tags, to answer the following questions:
  1. Produce an alphabetically sorted list of the distinct words tagged as `MD`.
  2. Identify words that can be plural nouns or third person singular verbs (e.g. *deals*, *flies*).
  3. Identify three-word prepositional phrases of the form `IN + DET + NN` (eg. *in the lab*).
  4. What is the ratio of masculine to feminine pronouns?
10. ● In the introduction we saw a table involving frequency counts for the verbs *adore*, *love*, *like*, *prefer* and preceding qualifiers such as *really*. Investigate the full range of qualifiers (Brown tag `QL`) that appear before these four verbs.
11. ● 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 `nltk.RegexpTagger`) that tests for at least five other patterns in the spelling of words. (Use inline documentation to explain the rules.)
12. ● Consider the regular expression tagger developed in the exercises in the previous section. Evaluate the tagger using `nltk.tag.accuracy()`, and try to come up with ways to improve its performance. Discuss your findings. How does objective evaluation help in the development process?
13. ★ There are 264 distinct words in the Brown Corpus having exactly three possible tags.
  1. 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.
  2. For the word with the greatest number of distinct tags, print out sentences from the corpus containing the word, one for each possible tag.
14. ★ 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*?
15. ☀ Train a unigram tagger and run it on some new text. Observe that some words are not assigned a tag. Why not?
16. ☀ Train an affix tagger `AffixTagger()` and run it on some new text. Experiment with different settings for the affix length and the minimum word length. Can you find a setting that seems to perform better than the one described above? Discuss your findings.
17. ☀ 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?
18. ● Write a program that calls `AffixTagger()` repeatedly, using different settings for the affix length and the minimum word length. What parameter values give the best overall performance? Why do you think this is the case?
19. ● 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  $10^2$ .
20. ● 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.
21. ● Inspect the confusion matrix for the bigram tagger `t2` defined in [Section 4.5](#), 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.
22. ● 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.

23. ● 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 part way 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?
24. ● Modify the program in [Figure 4.7](#) 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?
25. ★ Create a default tagger and various unigram and n-gram taggers, incorporating backoff, and train them on part of the Brown corpus.
  1. Create three different combinations of the taggers. Test the accuracy of each combined tagger. Which combination works best?
  2. Try varying the size of the training corpus. How does it affect your results?
26. ★ Our approach for tagging an unknown word has been to consider the letters of the word (using `RegexpTagger()` and `AffixTagger()`), 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` vs `NP$`) would probably be helpful. I.e. we need a default tagger that is sensitive to the preceding tag.
  1. 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 Python classes discussed in [Section 6.6](#).)
  2. 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.
  3. Evaluate the contribution of this new unigram tagger.
27. ★ Write code to preprocess tagged training data, replacing all but the most frequent  $n$  words with the special word *UNK*. Train an n-gram backoff tagger on this data, then use it to tag some new text. Note that you will have to preprocess the text to replace unknown words with *UNK*, and post-process the tagged output to replace the *UNK* words with the words from the original input.
28. ★ Consider the code in [4.5](#) which determines the upper bound for accuracy of a trigram tagger. Consult the Abney reading and review his discussion of the impossibility of exact tagging. 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?
29. ★ 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 ngram backoff taggers in cases where contexts encountered during testing were not seen during training.
30. ● 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?
31. ★ Inspect the diagnostic files created by the tagger `rules.out` and `errors.out`. Obtain the demonstration code (<http://nltk.org/nltk/tag/brill.py>) 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`.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

## 5 Data-Intensive Language Processing

### 5.1 Introduction

Language is full of patterns. In [Chapter 3](#) we saw that frequent use of the modal verb *will* is characteristic of news text, and more generally, that we can use the frequency of a small number of diagnostic words in order to automatically guess the genre of a text ([Table 1.1](#)). In [Chapter 4](#) we saw that words ending in *-ed* tend to be past tense verbs, and more generally, that the

internal structure of words tells us something about their part of speech. Detecting and understanding such patterns is central to many NLP tasks, particularly those that try to access the meaning of a text.

In order to study and model these linguistic patterns we need to be able to write programs to process large quantities of annotated text. In this chapter we will focus on data-intensive language processing, covering manual approaches to exploring linguistic data in [Section 5.2](#) and automatic approaches in [Section 5.5](#).

We have already seen a simple application of classification in the case of part-of-speech tagging ([Chapter 4](#)). Although this is a humble beginning, it actually holds the key for a range of more difficult classification tasks, including those mentioned above. Recall that adjectives (tagged JJ) tend to precede nouns (tagged NN), and that we can use this information to predict that the word *deal* is a noun in the context *good deal* (and not a verb, as in *to deal cards*).

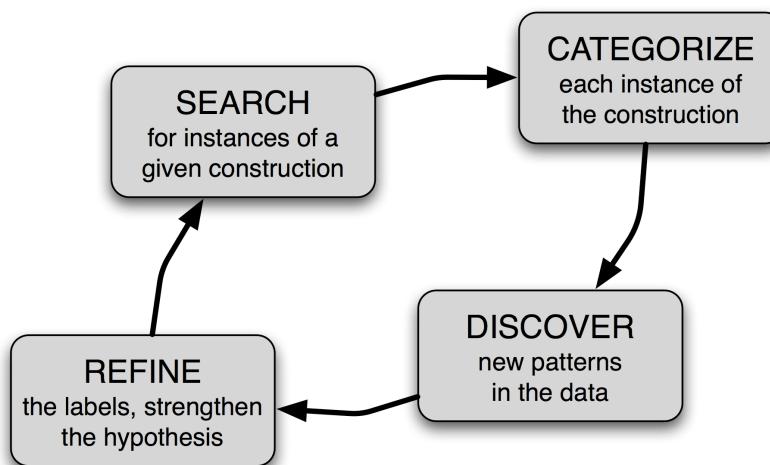
## 5.2 Exploratory Data Analysis

As language speakers, we all have intuitions about how language works, and what patterns it contains. Unfortunately, those intuitions are notoriously unreliable. We tend to notice unusual words and constructions, and to be oblivious to high-frequency cases. Many public commentators go further to make pronouncements about statistics and usage which turn out to be false. Many examples are documented on *LanguageLog*, e.g.

- Strunk and White's injunction against using adjectives and adverbs <http://itre.cis.upenn.edu/~myl/languagelog/archives/001905.html>
- Brizenden's claim that women use 20,000 words a day while men use 7,000 <http://itre.cis.upenn.edu/~myl/languagelog/archives/003420.html>
- Payack's claim that vocabulary size of English stood at 986,120 on 2006-01-26 <http://itre.cis.upenn.edu/~myl/languagelog/archives/002809.html>

In order to get an accurate idea of how language works, and what patterns it contains, we must study language — in a wide variety of forms and contexts — as impartial observers. To help facilitate this endeavour, researchers and organizations have created many large collections of real-world language, or **corpora**. These corpora are collected from a wide variety of sources, including literature, journalism, telephone conversations, instant messaging, and web pages.

**Exploratory data analysis**, the focus of this section, is a technique for learning about a specific linguistic pattern, or **construction**. It consists of four steps, illustrated in [Figure 5.1](#).



**Figure 5.1:** Exploratory Corpus Analysis

First, we must find the occurrences of the construction that we're interested in, by searching the corpus. Ideally, we would like to find all occurrences of the construction, but sometimes that may not be possible, and we have to be careful not to over-generalize our findings. In particular, we should be careful not to conclude that something doesn't happen simply because we were unable to find any examples; it's also possible that our corpus is deficient.

Once we've found the constructions of interest, we can then categorize them, using two sources of information: content and context. In some cases, like identifying date and time expressions in text, we can simply write a set of rules to cover the various cases. In general, we can't just enumerate the cases but we have to manually annotate a corpus of text and then train systems to

do the task automatically.

Having collected and categorized the constructions of interest, we can proceed to look for patterns. Typically, this involves describing patterns as combinations of categories, and counting how often different patterns occur. We can check for both graded distinctions and categorical distinctions...

- center-embedding suddenly gets bad after two levels
- examples from probabilistic syntax / gradient grammaticality

Finally, the information that we discovered about patterns in the corpus can be used to refine our understanding of how constructions work. We can then continue to perform exploratory data analysis, both by adjusting our characterizations of the constructions to better fit the data, and by building on our better understanding of simple constructions to investigate more complex constructions.

Although we have described exploratory data analysis as a cycle of four steps, it should be noted that any of these steps may be skipped or re-arranged, depending on the nature of the corpus and the constructions that we're interested in understanding. For example, we can skip the search step if we already have a corpus of the relevant constructions; and we can skip categorization if the constructions are already labeled.

## 5.3 Selecting a Corpus

In exploratory data analysis, we learn about a specific linguistic pattern by objectively examining how it is used. We therefore must begin by selecting a corpus (i.e., a collection of language data) containing the pattern we are interested in. Often, we can use one of the many existing corpora that have been made freely-available by the researchers who assembled them. Sometimes, we may choose instead to assemble a **derived corpus** by combining several existing corpora, or by selecting out specific subsets of a corpus (e.g., only news stories containing interviews). Occasionally, we may decide to build a new corpus from scratch (e.g., if we wish to learn about a previously undocumented language).

The results of our analysis will be highly dependent on the corpus that we select. This should hardly be surprising, since many linguistic phenomena pattern differently in different contexts -- for example, <>add a good example -- child vs adult? written vs spoken? some specific phenomenon?>. But when selecting the corpus for analysis, it is important to understand how the characteristics of the corpus will affect results of the the data analysis.

### Source of Language Data

Language is used for many different purposes, in many different contexts. For example, language can be used in a novel to tell a complex fictional story, or it can be used in an internet chat room to exchange rumors about celebrities. It can be used in a newspaper to report a story about a sports team, or in a workplace to form a collaborative plan for building a new product. Although some linguistic phenomena will act uniformly across these different contexts, other phenomena may vary depending.

Therefore, one of the most important characteristics defining a corpus is the source (or sources) from which its language data is drawn, which will determine the types of language data it includes. Attributes that characterize the type of language data contained in a corpus include:

- Domain: What subject matters does the language data talk about?
- Mode: Does the corpus contain spoken language data, written language data, or both?
- Number of Speakers: Does the corpus contain texts that are produced by a single speaker, such as books or news stories, or does it contain dialogues?
- Register: Is the language data in the corpus formal or informal?
- Communicative Intent: For what purpose was the langauge generated
  - e.g., to communicate, to entertain, or to persuade?
- Dialect: Do the speakers use any specific dialects?
- Language: What language or languages are used?

When making conclusions on the basis of exploratory data analysis, it is important to consider the extent to which those conclusions are dependent on the type of language data included in the corpus. For example, if we discover a pattern in a corpus of newspaper articles, we should not necessarily assume that the same pattern will hold in spoken discourse.

In order to allow more general conclusions to be drawn about linguistic patterns, several **balanced corpora** have been created, which include language data from a wide variety of different language sources. For example, the Brown Corpus contains documents ranging from science fiction to howto guidebooks to legislative council transcripts. But it's worth noting that since language use is so diverse, it would be almost impossible to create a single corpus that includes *all* of the contexts in which language gets used. Thus, even balanced corpora should be considered to cover only a subset of the possible linguistic sources (even if that subset is larger than the subset covered by many other corpora).

## Information Content

Corpora can vary in the amount of information they contain about the language data they describe. At a minimum, a corpus will typically contain at least a sequence of sounds or orthographic symbols. At the other end of the spectrum, a corpus could contain a large amount of information about the syntactic structure, morphology, prosody, and semantic content of every sentence. This extra information is called **annotation**, and can be very helpful when performing exploratory data analysis. For example, it may be much easier to find a given linguistic pattern if we can search for specific syntactic structures; and it may be easier to categorize a linguistic pattern if every word has been tagged with its word sense.

Corpora vary widely in the amount and types of annotation that they include. Some common types of information that can be annotated include:

- Word Tokenization: In written English, word tokenization is often marked implicitly using whitespace. However, in spoken English, and in some written languages, word tokenization may need to be explicitly marked.
- Sentence Segmentation: As we saw in [Chapter 3](#), sentence segmentation can be more difficult than it seems. Some corpora therefore use explicit annotations to mark sentence segmentation.
- Paragraph Segmentation: Paragraphs and other structural elements (headings, chapters, etc.) may be explicitly annotated.
- Part of Speech: The syntactic category of each word in a document.
- Syntactic Structure: A tree structure showing how each sentence is constructed from its constituent pieces.
- etc.

Unfortunately, there is not much consistency between existing corpora in how they represent their annotations. However, two general classes of annotation representation should be distinguished. **Inline annotation** modifies the original document by inserting special symbols or control sequences that carry the annotated information. For example, when part-of-speech tagging a document, the string "fly" might be replaced with the string "fly/NN", to indicate that the word *fly* is a noun in this context. In contrast, **standoff annotation** does not modify the original document, but instead creates a new file that adds annotation information using pointers into the original document. For example, this new document might contain the string "<word start=8 end=11 pos='NN' />", to indicate that the word starting at character 8 and ending at character 11 is a noun.

## Corpus Size

The size of corpora can vary widely, from tiny corpora containing just a few hundred sentences up to enormous corpora containing a billion words or more. In general, we perform exploratory data analysis using the largest appropriate corpus that's available. This ensures that the results of our analysis don't just reflect a quirk of the particular language data contained in the corpus.

However, in some circumstances, we may be forced to perform our analysis using small corpora. For example, if we are examining linguistic patterns in a language that is not well studied, or if our analysis requires specific annotations, then no large corpora may be available. In these cases, we should be careful when interpreting the results of an exploratory data analysis. In particular, we should avoid concluding that a linguistic pattern or phenomenon never occurs, just because we did not find it in our small sample of language data.

**Table 5.1:**

Example Corpora. This table summarizes some important properties of several popular corpora.

| Corpus Name   | Contents     | Size     | Annotations | etc. |
|---------------|--------------|----------|-------------|------|
| Penn Treebank | News stories | 1m words | etc.        |      |
| Web (google)  | etc.         |          |             |      |

## 5.4 Search

Once we've selected or constructed a corpus, the next step is to search that corpus for instances of the linguistic phenomenon we're interested in. The techniques that we use to search the corpus will depend on whether the corpus is annotated or not.

### Searching Unannotated Data

Unannotated corpora consist of a large quantity of "raw text," with no extra linguistic information. We therefore typically rely on search patterns to find the constructions of interest. For example, if we are investigating what adjectives can be used to describe the word "man," we could use the following code to search for relevant phrases:

```
>>> gutenberg = nltk.corpus.gutenberg.raw()
>>> re.findall(r'a \w+ man', gutenberg.lower())
['a young man', 'a just man', 'a wild man', 'a young man', 'a dead man',
 'a plain man', 'a hairy man', 'a smooth man', 'a certain man', ...]
```

For some linguistic phenomena, it can be fairly straightforward to build appropriate search patterns — especially when the linguistic phenomenon is tightly coupled with specific words. However, for other linguistic phenomena, it may be necessary to be very creative to think up appropriate search patterns. (For example, if we are interested in learning about type-instance relations, we could try searching for the pattern "*x* and other *ys*", which will match phrases such as "tea and other refreshments.") If we are unable to come up with an appropriate search pattern for a given linguistic phenomenon, then we should consider whether we might need to switch to a corpus containing annotations that might help us to locate the occurrences of the linguistic phenomenon we're interested in.

When dealing with an unannotated corpus, it's usually possible to build search patterns that find some examples of the linguistic phenomenon we're interested in. However, we should keep two limitations in mind. First, most patterns that we write will occasionally match examples that we are not interested in. For example, the search pattern "*x* and other *ys*" matches the string "eyes and other limbs," where the larger context is "other eyes and other limbs;" but we wouldn't want to conclude that an eye is a kind of limb.

Second, we should keep in mind that it can be difficult to write a pattern that finds every occurrence of a given phenomenon. Thus, as was the case when working with small corpora, we need to be very careful about drawing any negative conclusions: just because we did not find an example of a given pattern, does not mean that it never happens. We may have just not been creative enough to think of a pattern that would match the context where it occurs.

### Searching the Web

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 that search engines hold, when it comes to exploratory data analysis, is that of corpus size: since you are searching such a large set of documents, you are more likely to find any linguistic phenomenon you are interested in. Furthermore, you can make use of very specific patterns, which would only match 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.

However, search engines also have several significant disadvantages. First, you are usually fairly severely restricted in the types of patterns you can search for. Unlike local corpora, where you can build arbitrarily complex regular expressions, search engines generally just allow you to search for individual words or strings of words, and sometimes for fill-in-the-blank patterns. Second, search engines use advanced techniques to filter web pages, in an attempt to block out spam and pornography. These techniques may unintentionally skew the results of a web-based exploratory data analysis.

Finally, it should be noted that the web page counts provided by search engines can be misleading. Many of the pages may contain the strings that you searched for, but they may be used in unexpected ways. For example, many web pages include

non-language text that might give you false matches. Web pages that intersperse images with text may cause unexpected matches. And the web page count may include pages with duplicated content, which do not actually represent separate occurrences of the search pattern. As an example of these problems, most search engines will return millions of hits for the search pattern "the of", even though we know that this pair of words should (almost) never occur together in English sentences.

## Searching annotated corpora

Large unannotated corpora are widely available; and it is fairly easy to search these corpora for simple string patterns. However, many of the linguistic phenomena that we may be interested can not be captured with simple string patterns. For example, it would be very difficult to write a search pattern that finds all verbs that take sentential complements.

Partially to help address these concerns, a large number of manually annotated corpora have been created. These corpora are augmented with a wide variety of extra information about the language data they contain. The exact types and extents of this additional information will vary from corpus to corpus, but common annotation types include tokenization and segmentation information; information about words' parts of speech or word sense; and information about sentences' syntactic structure. We can make use of this extra information to perform more advanced searches over the corpus. Often, this extra information will make it possible to find all of (or most of) the occurrences of a given linguistic phenomenon, especially if the phenomenon is closely related to the type of information contained in the corpus's annotations.

However, building annotated corpora is a very labor-intensive process, and these corpora therefore tend to be significantly smaller than corresponding unannotated corpora. As we mentioned when discussing corpus size, we should therefore be careful about concluding that a linguistic pattern or phenomenon never occurs, just because we did not find it in a small annotated corpus.

Because the format and information content of annotations can vary widely from one corpus to the next, there is no single tool for searching annotated corpora. One solution is to use one of the many specialized tools have been built for searching popular corpus formats. For example, the `tgrep` tool can be used to search for specific syntactic patterns in a corpus that is annotated with the syntactic structure of each sentence. This can be a very efficient solution if your corpus is in the required format, and if the tool supports the search pattern you wish to construct.

But a more general solution is to simply write a short Python script that loads the corpus and then scans through it for any occurrences of the linguistic pattern you are interested in. For example, if we are interested in searching for occurrences of the pattern "`<Verb> to <Verb>`" in the Brown corpus, we could use the following short script:

```
>>> from nltk.corpus import brown
>>> for sent in brown.tagged_sents():
... # Look at each 3-word window in the sentence.
... for triple in nltk.trigrams(sent):
... # Get the part-of-speech tags for this 3-word window.
... tags = [t for (w,t) in triple]
... # Check if they match our pattern.
... if (tags[0].startswith('V') and tags[1]=='TO' and
... tags[2].startswith('V')):
... print triple
[('combined', 'VBN'), ('to', 'TO'), ('achieve', 'VB')]
[('continue', 'VB'), ('to', 'TO'), ('place', 'VB')]
[('serve', 'VB'), ('to', 'TO'), ('protect', 'VB')]
[('wanted', 'VBD'), ('to', 'TO'), ('wait', 'VB')]
[('allowed', 'VBN'), ('to', 'TO'), ('place', 'VB')]
[('expected', 'VBN'), ('to', 'TO'), ('become', 'VB')]
...
[('seems', 'VBZ'), ('to', 'TO'), ('overtake', 'VB')]
[('want', 'VB'), ('to', 'TO'), ('buy', 'VB')]
```

In [Chapter 7](#), we'll learn about the "regular expression chunker," which can be used to make this search script even simpler:

```
>>> cp = nltk.RegexpChunker("CHUNK: {<V.*> <TO> <V.*>}")
>>> brown = nltk.corpus.brown
>>> for sent in brown.tagged_sents():
... tree = cp.parse(sent)
... for subtree in tree.subtrees():
... if subtree.node == 'CHUNK': print subtree
...
(CHUNK combined/VBN to/TO achieve/VB)
```

```
(CHUNK continue/VB to/TO place/VB)
(CHUNK serve/VB to/TO protect/VB)
(CHUNK wanted/VBD to/TO wait/VB)
(CHUNK allowed/VBN to/TO place/VB)
(CHUNK expected/VBN to/TO become/VB)
...
(CHUNK seems/VBZ to/TO overtake/VB)
(CHUNK want/VB to/TO buy/VB)
```

As a second example, the script shown in [Listing 5.2](#) uses a simple filter to find all verbs in a given corpus that take sentential complements.

```
def filter(tree):
 child_nodes = [child.node for child in tree
 if isinstance(child, nltk.Tree)]
 return (tree.node == 'VP') and ('S' in child_nodes)

>>> treebank = nltk.corpus.treebank
>>> for tree in treebank.parsed_sents()[:5]:
... for subtree in tree.subtrees(filter):
... print subtree
(VP
 (VBN named)
 (S
 (NP-SBJ (-NONE- *-1))
 (NP-PRD
 (NP (DT a) (JJ nonexecutive) (NN director)))
 (PP
 (IN of)
 (NP (DT this) (JJ British) (JJ industrial) (NN conglomerate))))))
... SB: NB a later discussion of XML will include XPath, another method for tree search
```

[Figure 5.2 \(sentential\\_complement.py\)](#): Figure 5.2

## Searching Automatically Annotated Data

In some cases, we may find that the hand-annotated corpora that are available are too small to perform the analysis we desire; but that unannotated corpora do not contain the information we need to perform a search. One solution in these cases is to build an automatically annotated corpus, and then to search that corpus.

For example, if we wish to search for a relatively rare syntactic configuration, we might first train an automatic parser using a corpus that has been hand-annotated with syntactic parses (such as the Treebank corpus); and then use that parser to generate parse trees for a much larger unannotated corpus. We could then use those automatically generated parse trees to create a new annotated corpus, and finally we could search this annotated corpus for the syntactic configuration we're interested in.

- is this safe?
  - yes, sometimes.
  - look at the automatic system's accuracy, and think about how it might affect your search
    - could the automatic system make a systematic error that would prevent you from finding an important class of instances?
    - the more closely tied the annotation & the phenomenon are, the more likely you are to get into trouble.

## Categorizing

Once we've found the occurrences we're interested in, the next step is to categorize them. In general, we're interested in two things:

- features of the phenomenon itself
- features of the context that we think are relevant to the phenomenon.

Categorization can be automatic or manual

- automatic: when the decision can be made deterministically. e.g., what is the previous word?

- manual: when the decision needs human judgement. example.. animacy?

Encoding this information -- features. We need to encode this info in a concrete way. Use a feature dictionary for each occurrence, mapping feature names (eg 'prevword') to concrete values (eg a string, int). Features are typically simple-valued, but don't necessarily need to be. (Though they will need to be for automatic methods.. coming up)

```

def features(word):
 return dict(len = len(word),
 last1 = word[-1:],
 last2 = word[-2:],
 last3 = word[-3:])

>>> data = [(features(word), tag) for (word, tag) in nltk.corpus.brown.tagged_words('news')]
>>> train = data[1000:]
>>> test = data[:1000]
>>> classifier = nltk.NaiveBayesClassifier.train(train)
>>> nltk.classify.accuracy(classifier, test)
0.8110

```

[Figure 5.3 \(tagging.py\)](#): Figure 5.3

```

def preprocess(sents):
 tokens = []
 boundaries = []
 for sent in sents:
 for token in sent:
 tokens.append(token)
 boundaries.append(False)
 boundaries[-1] = True
 return (tokens, boundaries)

def get_instances(tokens, boundaries):
 instances = []
 for i in range(len(tokens)):
 if tokens[i] in ".?!" :
 try:
 instances.append(
 (dict(upper = tokens[i+1][0].isupper(),
 abbrev = len(tokens[i-1]) == 1),
 boundaries[i]))
 except IndexError:
 pass
 return instances

>>> tokens, boundaries = preprocess(nltk.corpus.abc.sents())
>>> data = get_instances(tokens, boundaries)
>>> train = data[1000:]
>>> test = data[:1000]
>>> classifier = nltk.NaiveBayesClassifier.train(train)
>>> nltk.classify.accuracy(classifier, test)
0.9960

```

[Figure 5.4 \(segmentation.py\)](#): Figure 5.4

## Counting

Now that we've got our occurrences coded up, we want to look at how often different combinations occur.

- we can look for both graded and categorical distinctions
  - for categorical distinctions, we don't necessarily require that the counts be zero; every rule has its exception.

Example: what makes a name sound male or female? Walk through it, explain some features, do some counts using python.

## 5.5 Data Modeling

Exploratory data analysis helps us to understand the linguistic patterns that occur in natural language corpora. Once we have a basic understanding of those patterns, we can attempt to create **models** that capture those patterns. Typically, these models will be constructed automatically, using algorithms that attempt to select a model that accurately describes an existing corpus; but it is also possible to build analytically motivated models. Either way, these explicit models serve two important purposes: they

help us to understand the linguistic patterns, and they can be used to make predictions about new language data.

The extent to which explicit models can give us insight into linguistic patterns depends largely on what kind of model is used. Some models, such as decision trees, are relatively transparent, and give us direct information about which factors are important in making decisions, and about which factors are related to one another. Other models, such as multi-level neural networks, are much more opaque -- although it can be possible to gain insight by studying them, it typically takes a lot more work.

But all explicit models can make predictions about new "**unseen**" language data that was not included in the corpus used to build the model. These predictions can be evaluated to assess the accuracy of the model. Once a model is deemed sufficiently accurate, it can then be used to automatically predict information about new language data. These predictive models can be combined into systems that perform many useful language processing tasks, such as document classification, automatic translation, and question answering.

## What do models tell us?

Before we delve into the mechanics of different models, it's important to spend some time looking at exactly what automatically constructed models can tell us about language.

One important consideration when dealing with language models is the distinction between descriptive models and explanatory models. Descriptive models capture patterns in the data but they don't provide any information about *why* the data contains those patterns. For example, as we saw in [Table 3.1](#), the synonyms *absolutely* and *definitely* are not interchangeable: we say *absolutely adore* not *definitely adore*, and *definitely prefer* not *absolutely prefer*. In contrast, explanatory models attempt to capture properties and relationships that underlie the linguistic patterns. For example, we might introduce the abstract concept of "polar adjective", as one that has an extreme meaning, and categorize some adjectives like *adore* and *detest* as polar. Our explanatory model would contain the constraint that *absolutely* can only combine with polar adjectives, and *definitely* can only combine with non-polar adjectives. In summary, descriptive models provide information about correlations in the data, while explanatory models go further to postulate causal relationships.

Most models that are automatically constructed from a corpus are descriptive models; in other words, they can tell us what features are relevant to a given patterns or construction, but they can't necessarily tell us how those features and patterns relate to one another. If our goal is to understand the linguistic patterns, then we can use this information about which features are related as a starting point for further experiments designed to tease apart the relationships between features and patterns. On the other hand, if we're just interested in using the model to make predictions (e.g., as part of a language processing system), then we can use the model to make predictions about new data, without worrying about the precise nature of the underlying causal relationships.

## Supervised Classification

One of the most basic tasks in data modeling is **classification**. In classification tasks, we wish to choose the correct **class label** for a given input. Each input is considered in isolation from all other inputs, and set of labels is defined in advance. Some examples of classification tasks are:

### Note

The basic classification task has a number of interesting variants: for example, in multi-class classification, each instance may be assigned multiple labels; in open-class classification, the set of labels is not defined in advance; and in sequence classification, a list of inputs are jointly classified.

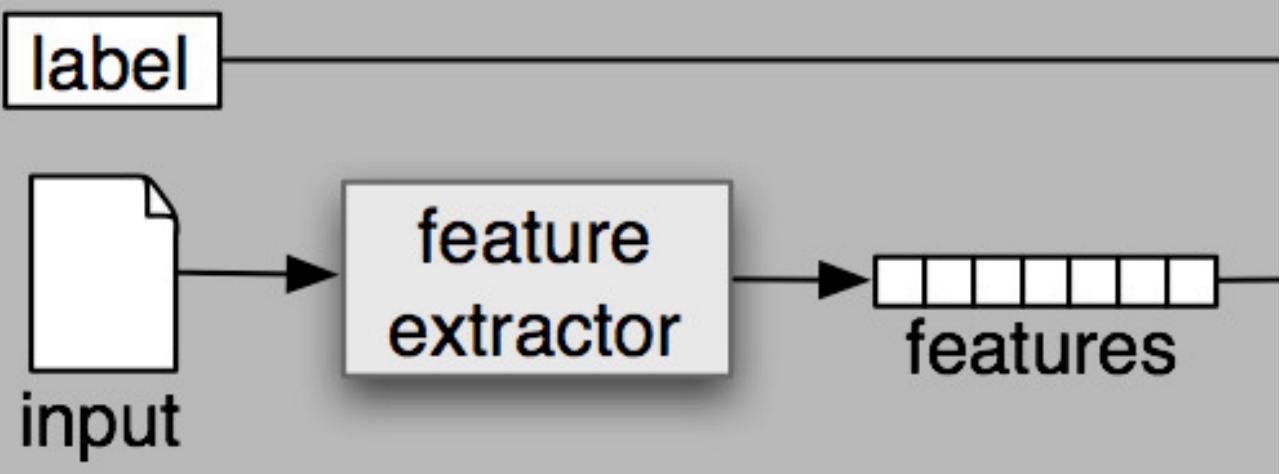
Classification models are typically trained using a corpus that contains the correct label for each input. This **training corpus** is typically constructed by manually annotating each input with the correct label, but for some tasks it is possible to automatically construct training corpora. Classification models that are built based on training corpora that contain the correct label for each input are called **supervised** classification models.

## Feature Extraction

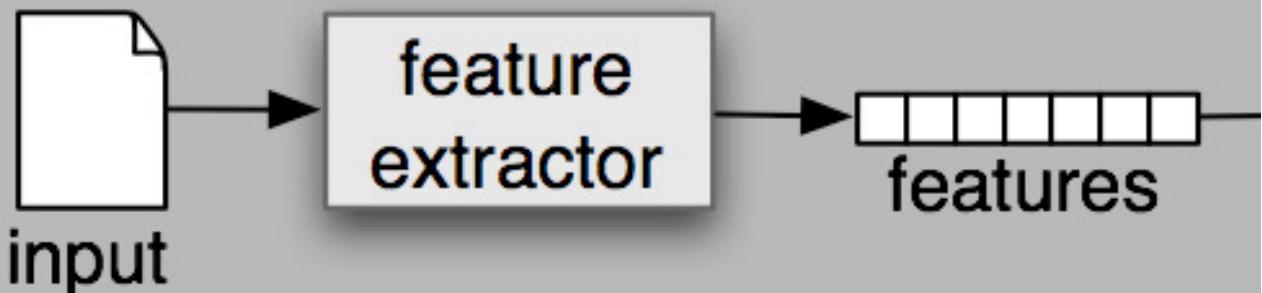
The first step in creating a model is deciding what information about the input might be relevant to the classification task; and

how to encode that information. In other words, we must decide which **features** of the input are relevant, and how to **encode** those features. Most automatic learning methods restrict features to have simple value types, such as booleans, numbers, and strings. But note that just because a feature has a simple type, does not necessarily mean that the feature's value is simple to express or compute; indeed, it is even possible to use very complex and informative values, such as the output of a second supervised classifier, as features.

## (a) Training



## (b) Prediction



**Figure 5.5:** Supervised Classification. (a) During training, a feature extractor is used to convert each input value to a feature set. Pairs of feature sets and labels are fed into the machine learning algorithm to generate a model. (b) During prediction, the same feature extractor is used to convert unseen inputs to feature sets. These feature sets are then fed into the model, which generates predicted labels.

For NLTK's classifiers, the features for each input are stored using a dictionary that maps feature names to corresponding values. Feature names are case-sensitive strings that typically provide a short human-readable description of the feature. Feature values are simple-typed values, such as booleans, numbers, and strings. For example, if we had built an `animal_classifier` model for classifying animals, then we might provide it with the following feature set:

```
>>> animal = {'fur': True, 'legs': 4,
... 'size': 'large', 'spots': True}
>>> animal_classifier.classify(animal)
'leopard'
```

Generally, feature sets are constructed from inputs using a **feature extraction** function. This function takes an input value, and possibly its context, as parameters, and returns a corresponding feature set. This feature set can then be passed to the machine learning algorithm for training, or to the learned model for prediction. For example, we might use the following function to extract features for a document classification task:

```
def extract_features(document):
 features = {}
 for word in document:
 features['contains(%s)' % word] = True
 return features

>>> extract_features(nltk.corpus.brown.words('cj79'))
{'contains(of)': True, 'contains(components)': True,
'contains(some)': True, 'contains(that)': True,
'contains(passage)': True, 'contains(table)': True, ...}
```

[Figure 5.6 \(feature\\_extractor.py\)](#): Figure 5.6

In addition to a feature extractor, we need to select or build a training corpus, consisting of a list of examples and corresponding class labels. For many interesting tasks, appropriate corpora have already been assembled. Given a feature extractor and a training corpus, we can train a classifier. First, we run the feature extractor on each instance in the training corpus, and building a list of (featureset, label) tuples. Then, we pass this list to the classifier's constructor:

```
>>> train = [(extract_features(word), label)
... for (word, label) in labeled_words]
>>> classifier = nltk.NaiveBayesClassifier.train(train)
```

The constructed model `classifier` can then be used to predict the labels for unseen inputs:

```
>>> test_featuresets = [extract_features(word)
... for word in unseen_labeled_words]
>>> predicted = classifier.batch_classify(test)
```

### Note

When working with large corpora, constructing a single list that contains the features of every instance can use up a large amount of memory. In these cases, we can make use of the function `nltk.classify.apply_features`, which returns an object that acts like a list but does not store all values in memory:

```
>>> train = apply_features(extract_features, labeled_words)
>>> test = apply_features(extract_features, unseen_words)
```

Selecting relevant features, and deciding how to encode them for the learning method, can have an enormous impact on its ability to extract a good model. Much of the interesting work in modeling a phenomenon is deciding what features might be relevant, and how we can represent them. Although it's often possible to get decent performance by using a fairly simple and obvious set of features, there are usually significant gains to be had by using carefully constructed features based on an understanding of the task at hand.

Typically, feature extractors are built through a process of trial-and-error, guided by intuitions about what information is relevant to the problem at hand. It's often useful to start with a "kitchen sink" approach, including all the features that you can think of, and then checking to see which features actually appear to be helpful. However, there are usually limits to the number of features that you should use with a given learning algorithm -- if you provide too many features, then the algorithm will have a higher chance of relying on idiosyncrasies of your training data that don't generalize well to new examples. This problem is known as **overfitting**, and can especially problematic when working with small training sets.

Once a basic system is in place, a very productive method for refining the feature set is **error analysis**. First, the training corpus is split into two pieces: a training subcorpus, and a **development** subcorpus. The model is trained on the training subcorpus, and then run on the development subcorpus. We can then examine individual cases in the development subcorpus where the model predicted the wrong label, and try to determine what additional pieces of information would allow it to make the right decision (or which existing pieces of information are tricking it into making the wrong decision). The feature set can then be adjusted

accordingly, and the error analysis procedure can be repeated, ideally using a different development/training split.

## Example: Predicting Name Genders

In [section 5.2](#), we looked at some of the factors that might influence whether an English name sounds more like a male name or a female name. Now we can build a simple model for this classification task. We'll use the same `names` corpus that we used for exploratory data analysis, divided into a training set and an evaluation set:

```
>>> from nltk.corpus import names
>>> import random
>>>
>>> # Construct a list of classified names, using the names corpus.
>>> namelist = ([name, 'male') for name in names.words('male')] +
... [(name, 'female') for name in names.words('female')])
>>>
>>> # Randomly split the names into a test & train set.
>>> random.shuffle(namelist)
>>> train = namelist[500:]
>>> test = namelist[:500]
```

Next, we'll build a simple feature extractor, using some of the features that appeared to be useful in the exploratory data analysis. We'll also throw in a number of features that seem like they might be useful:

```
def gender_features(name):
 features = {}
 features["firstletter"] = name[0].lower()
 features["lastletter"] = name[-1].lower()
 for letter in 'abcdefghijklmnopqrstuvwxyz':
 features["count(%s)" % letter] = name.lower().count(letter)
 features["has(%s)" % letter] = (letter in name.lower())
 return features

>>> gender_features('John')
{'count(j)': 1, 'has(d)': False, 'count(b)': 0, ...}
```

[Figure 5.7 \(gender\\_features.py\)](#): Figure 5.7

Now that we have a corpus and a feature extractor, we can train a classifier. We'll use a "Naive Bayes" classifier, which will be described in more detail in [section 5.8.1](#).

Now we can use the classifier to predict the gender for unseen names:

```
>>> classifier.classify(gender_features('Blorgy'))
'male'
>>> classifier.classify(gender_features('Alaphina'))
'female'
```

And using the test corpus, we can check the overall accuracy of the classifier across a collection of unseen names with known labels:

```
>>> test_featuresets = [(gender_features(n), g) for (n,g) in test]
>>> print nltk.classify.accuracy(classifier, test_featuresets)
0.688
```

## Example: Predicting Sentiment

Movie review domain; ACL 2004 paper by Lillian Lee and Bo Pang. Movie review corpus included with NLTK.

```
import nltk, random

TEST_SIZE = 500

def word_features(doc):
 words = nltk.corpus.movie_reviews.words(doc)
 return nltk.FreqDist(words), doc[0]
```

```

def get_data():
 featuresets = apply(word_features, nltk.corpus.movie_reviews.files())
 random.shuffle(featuresets)
 return featuresets[TEST_SIZE:], featuresets[:TEST_SIZE]

>>> train_featuresets, test_featuresets = get_data()
>>> c1 = nltk.NaiveBayesClassifier.train(train_featuresets)
>>> print nltk.classify.accuracy(c1, test_featuresets)
0.774
>>> c2 = nltk.DecisionTreeClassifier.train(train_featuresets)
>>> print nltk.classify.accuracy(c2, test_featuresets)
0.576

```

**Figure 5.8 (movie\_reviews.py): Figure 5.8**

Initial work on a classifier to use frequency of modal verbs to classify documents by genre:

```

import nltk, math
modals = ['can', 'could', 'may', 'might', 'must', 'will']

def modal_counts(tokens):
 return nltk.FreqDist(word for word in tokens if word in modals)

just the most frequent modal verb
def modal_features1(tokens):
 return dict(most_frequent_modal = modal_counts(tokens).max())

one feature per verb, set to True if the verb occurs more than once
def modal_features2(tokens):
 fd = modal_counts(tokens)
 return dict((word, (fd[word]>1)) for word in modals)

one feature per verb, with a small number of scalar values
def modal_features3(tokens):
 fd = modal_counts(tokens)
 features = {}
 for word in modals:
 try:
 features[word] = int(-math.log10(float(fd[word])/len(tokens)))
 except OverflowError:
 features[word] = 1000
 return features

4 bins per verb based on frequency
def modal_features4(tokens):
 fd = modal_counts(tokens)
 features = {}
 for word in modals:
 freq = float(fd[word])/len(tokens)
 for logfreq in range(3,7):
 features["%s(%d)" % (word, logfreq)] = (freq < 10**(-logfreq))
 return features

>>> genres = ['hobbies', 'humor', 'science_fiction', 'news', 'romance', 'religion']
>>> train = [(modal_features4(nltk.corpus.brown.words(g)[:2000]), g) for g in genres]
>>> test = [(modal_features4(nltk.corpus.brown.words(g)[2000:4000]), g) for g in genres]
>>> classifier = nltk.NaiveBayesClassifier.train(train)
>>> print 'Accuracy: %6.4f' % nltk.classify.accuracy(classifier, test)

```

**Figure 5.9 (modals.py): Figure 5.9**

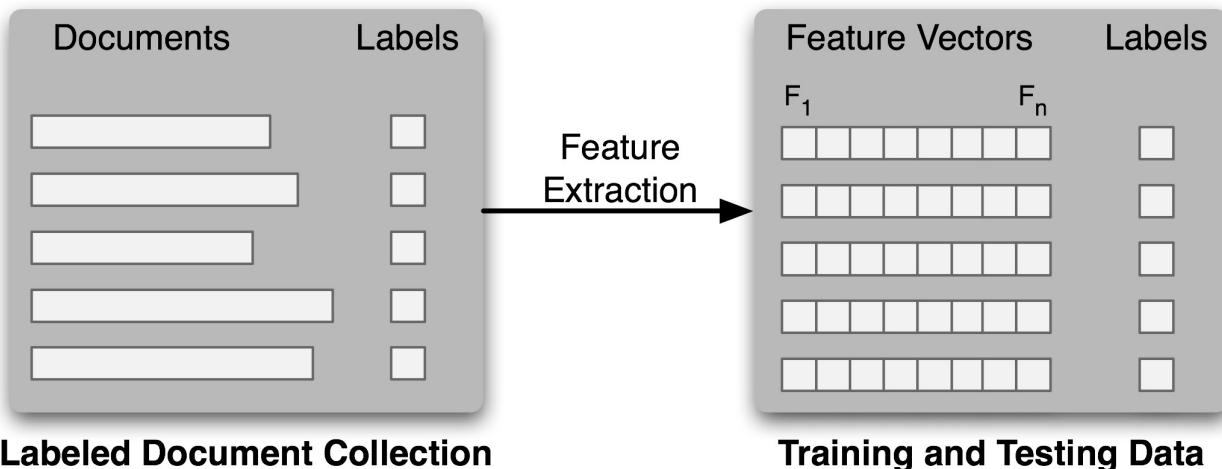


Figure 5.10: Feature Extraction

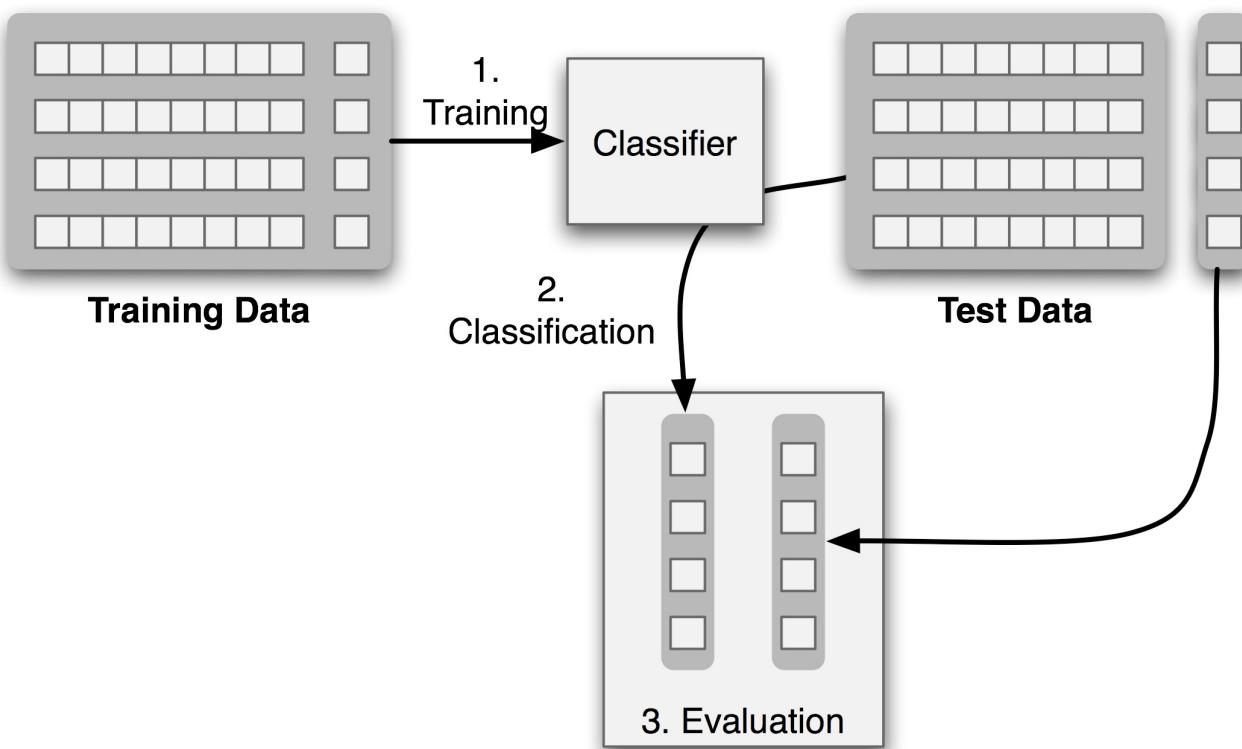


Figure 5.11: Document Classification

## 5.6 Evaluation

In order to decide whether a classification model is accurately capturing a pattern, we must evaluate that model. The result of this evaluation is important for deciding how trustworthy the model is, and for what purposes we can use it. Evaluation can also be a useful tool for guiding us in making future improvements to the model.

### Evaluation Set

Most evaluation techniques calculate a score for a model by comparing the labels that it generates for the inputs in an **evaluation set** with the correct labels for those inputs. This evaluation set typically has the same format as the training corpus. However, it is very important that the evaluation set be distinct from the training corpus: if we simply re-used the training corpus as the evaluation set, then a model that simply memorized its input, without learning how to generalize to new examples, would receive very high scores. Similarly, if we use a development corpus, then it must be distinct from the evaluation set as well. Otherwise, we risk building a model that does not generalize well to new inputs; and our evaluation scores may be

misleadingly high.

If we are actively developing a model, by adjusting the features that it uses or any hand-tuned parameters, then we may want to make use of two evaluation sets. We would use the first evaluation set while developing the model, to evaluate whether specific changes to the model are beneficial. However, once we've made use of this first evaluation set to help develop the model, we can no longer trust that it will give us an accurate idea of how well the model would perform on new data. We therefore save the second evaluation set until our model development is complete, at which point we can use it to check how well our model will perform on new input values.

When building an evaluation set, we must be careful to ensure that it is sufficiently different from the training corpus that it will effectively evaluate the performance of the model on new inputs. For example, if our evaluation set and training corpus are both drawn from the same underlying data source, then the results of our evaluation will only tell us how well the model is likely to do on other texts that come from the same (or a similar) data source.

## Accuracy

The simplest metric that can be used to evaluate a classifier, **accuracy**, measures the percentage of inputs in the evaluation set that the classifier correctly labeled. For example, a name gender classifier that predicts the correct name 60 times in an evaluation set containing 80 names would have an accuracy of  $60/80 = 75\%$ . The function `nltk.classify.accuracy` can be used to calculate the accuracy of a classifier model on a given evaluation set:

```
>>> classifier = nltk.NaiveBayesClassifier.train(train)
>>> print 'Accuracy: %4.2f' % nltk.classify.accuracy(classifier, test)
0.75
```

When interpreting the accuracy score of a classifier, it is important to take into consideration the frequencies of the individual class labels in the evaluation set. For example, consider a classifier that determines the correct word sense for each occurrence of the word "bank." If we evaluate this classifier on financial newswire text, then we may find that the `financial-institution` sense is used 19 times out of 20. In that case, an accuracy of 95% would hardly be impressive, since we could achieve that accuracy with a model that always returns the `financial-institution` sense. However, if we instead evaluate the classifier on a more balanced corpus, where the most frequent word sense has a frequency of 40%, then a 95% accuracy score would be a much more positive result.

- Simplest metric: accuracy. Describe what it is, where it can be limited in usefulness.

## Precision and Recall

**system output:**  
**retrieved documents** —

**information need:** —  
**relevant documents**

### Figure 5.12: True and False Positives and Negatives

Consider [Figure 5.12](#). The intersection of these sets defines four regions: the true positives (TP), true negatives (TN), false positives (FP) or Type I errors, and false negatives (FN) or Type II errors. Two standard measures are *precision*, the fraction of guessed chunks that were correct  $TP/(TP+FP)$ , and *recall*, the fraction of correct chunks that were identified  $TP/(TP+FN)$ . A third measure, the *F measure*, is the harmonic mean of precision and recall, i.e.  $1/(0.5/\text{Precision} + 0.5/\text{Recall})$ .

## Cross-Validation

To do evaluation, we need to keep some of the data back -- don't test on train. But that means we have less data available to train. Also, what if our training set has ideoyncracies?

Cross-validation: run training&testing multiple times, with different training sets.

- Lets us get away with smaller training sets
- Lets us get a feel for how much the performance varies based on different training sets.

## Error Analysis

The metrics above give us a general feel for how well a system does, but doesn't tell us much about why it gets that performance .. are there patterns in what it gets wrong? If so, that can help us to improve the system, or if we can't improve it, then at least make us more aware of what the limitations of the system are, and what kind of data it will produce more reliable or less reliable results for.

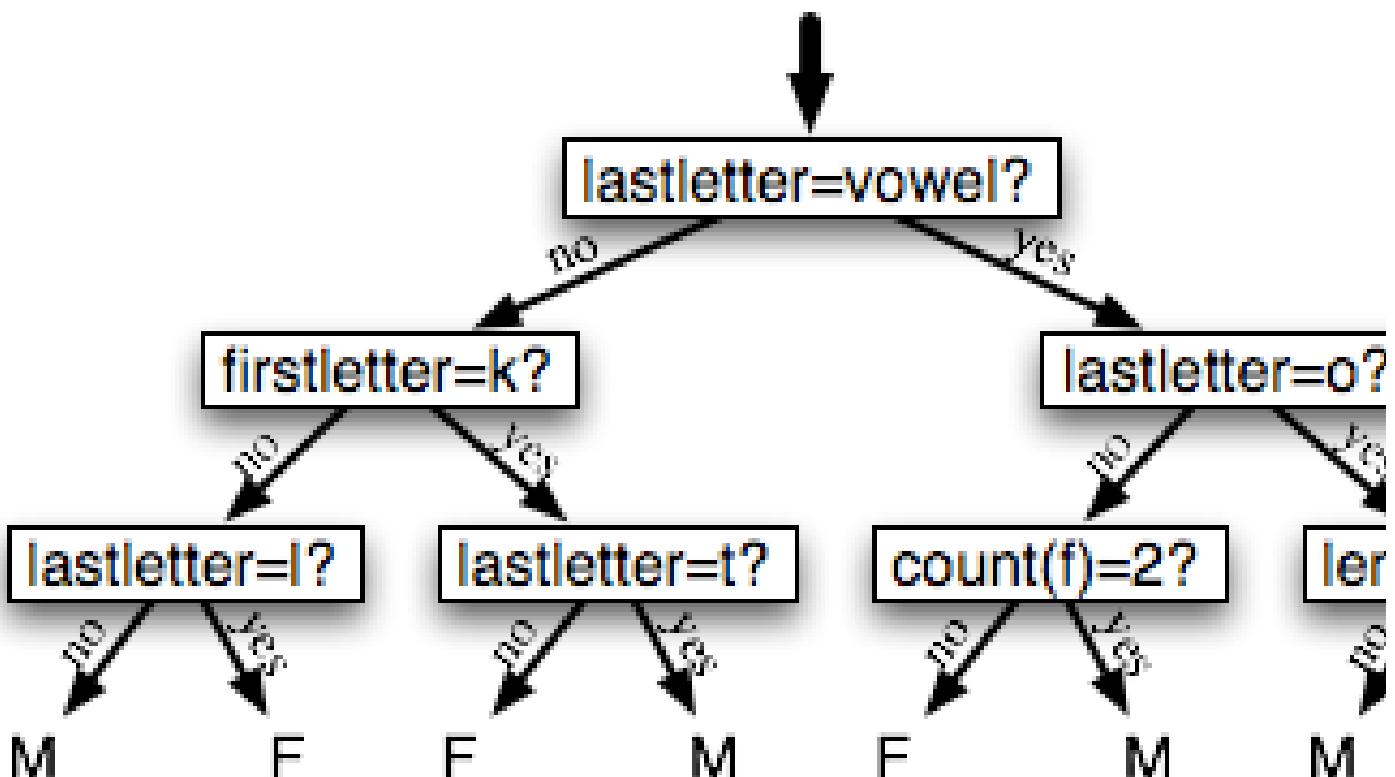
Talk some about how to do error analysis?

## 5.7 Classification Methods

In this section, we'll take a closer look at three machine learning methods that can be used to automatically build classification models: Decision Trees, Naive Bayes classifiers, and Maximum Entropy classifiers. As we've seen, it's possible treat these learning methods as black boxes, simply training models and using them for prediction without understanding how they work. But there's a lot to be learned from taking a closer look at how these learning methods select models based on the data in a training corpus. An understanding of these methods can help guide our selection of appropriate features, and especially our decisions about how those features should be encoded. And an understanding of the generated models can allow us to extract useful information about which features are most informative, and how those features relate to one another.

## 5.8 Decision Trees

A **decision tree** is a tree-structured flowchart used to choose labels for input values. This flowchart consists of **decision nodes**, which check feature values, and **leaf nodes**, which assign labels. To choose the label for an input value, we begin at the flowchart's initial decision node, known as its **root node**. This node contains a condition that checks one of the input value's features, and selects a branch based on that feature's value. Following the branch that describes our input value, we arrive at a new decision node, with a new condition on the input value's features. We continue following the branch selected by each node's condition, until we arrive at a leaf node, which provides a label for the input value. [Figure 5.13](#) shows an example decision tree model for the name gender task.



**Figure 5.13:** Decision Tree model for the name gender task. Note that tree diagrams are conventionally drawn "upside down," with the root at the top, and the leaves at the bottom.

Once we have a decision tree, it is thus fairly straight forward to use it to assign labels to new input values. What's less straight forward is how we can build a decision tree that models a given training corpus. But before we look at the learning algorithm for building decision trees, we'll consider a simpler task: picking the best "decision stump" for a corpus. A **decision stump** is a decision tree with a single node, that decides how to classify inputs based on a single feature. It contains one leaf for each possible feature value, specifying the class label that should be assigned to inputs whose features have that value. In order to build a decision stump, we must first decide which feature should be used. The simplest method is to just build a decision stump for each possible feature, and see which one achieves the highest accuracy on the training data; but we'll discuss some other alternatives below. Once we've picked a feature, we can build the decision stump by assigning a label to each leaf based on the most frequent label for the selected examples in the training corpus (i.e., the examples where the selected feature has that value).

Given the algorithm for choosing decision stumps, the algorithm for growing larger decision trees is straightforward. We begin by selecting the overall best decision stump for the corpus. We then check the accuracy of each of the leaves on the training corpus. Any leaves that do not achieve sufficiently good accuracy are then replaced by new decision stumps, trained on the subset of the training corpus that is selected by the path to the leaf. For example, we could grow the decision tree in [Figure 5.13](#) by replacing the leftmost leaf with a new decision stump, trained on the subset of the training corpus names that do not start with a "k" or end with a vowel or an "l."

As we mentioned before, there are a number of methods that can be used to select the most informative feature for a decision stump. One popular alternative is to use **information gain**, a measure of how much more organized the input values become when we divide them up using a given feature. To measure how disorganized the original set of input values are, we calculate entropy of their labels, which is defined as:

$$\text{Entropy}(S) = \sum_{\{\text{label}\}} \text{freq}(\text{label}) * \log_2(\text{freq}(\text{label}))$$

(how are we doing markup for math? -- also inline math?)

If most input values have the same label, then the entropy of their labels will be low. In particular, labels that have low frequency will not contribute much to the entropy (since the first term,  $\text{freq}(\text{label})$ , will be low); and labels with high frequency will also not contribute much to the entropy (since  $\log_2(\text{freq}(\text{label}))$  will be low). On the other hand, if the input values have a wide variety of labels, then there will be many labels with a "medium" frequency, where neither  $\text{freq}(\text{label})$  nor

$\log_2(\text{freq}(\text{label}))$  is low, so the entropy will be high.

Once we have calculated the entropy of the original set of input values' labels, we can figure out how much more organized the labels become once we apply the decision stump. To do so, we calculate the entropy for each of the decision stump's leaves, and take the average of those leaf entropy values (weighted by the number of samples in each leaf). The information gain is then equal to the original entropy minus this new, reduced entropy. The higher the information gain, the better job the decision stump does of dividing the input values into coherent groups, so we can build decision trees by selecting the decision stumps with the highest information gain.

Another consideration for decision trees is efficiency. The simple algorithm for selecting decision stumps described above must construct a candidate decision stump for every possible feature; and this process must be repeated for every node in the constructed decision tree. A number of algorithms have been developed to cut down on the training time by storing and reusing information about previously evaluated examples. <<references>>.

Decision trees have a number of useful qualities. To begin with, they're simple to understand, and easy to interpret. This is especially true near the top of the decision tree, where it is usually possible for the learning algorithm to find very useful features. Decision trees are especially well suited to cases where many hierarchical categorical distinctions can be made. For example, decision trees can be very effective at modelling phylogeny trees.

However, decision trees also have a few disadvantages. One problem is that, since each branch in the decision tree splits the training data, the amount of training data available to train nodes lower in the tree can become quite small. As a result, these lower decision nodes may **overfit** the training corpus, learning patterns that reflect idiosyncrasies of the training corpus, rather than genuine patterns in the underlying problem. One solution to this problem is to stop dividing nodes once the amount of training data becomes too small. Another solution is to grow a full decision tree, but then to **prune** decision nodes that do not improve performance on a development corpus.

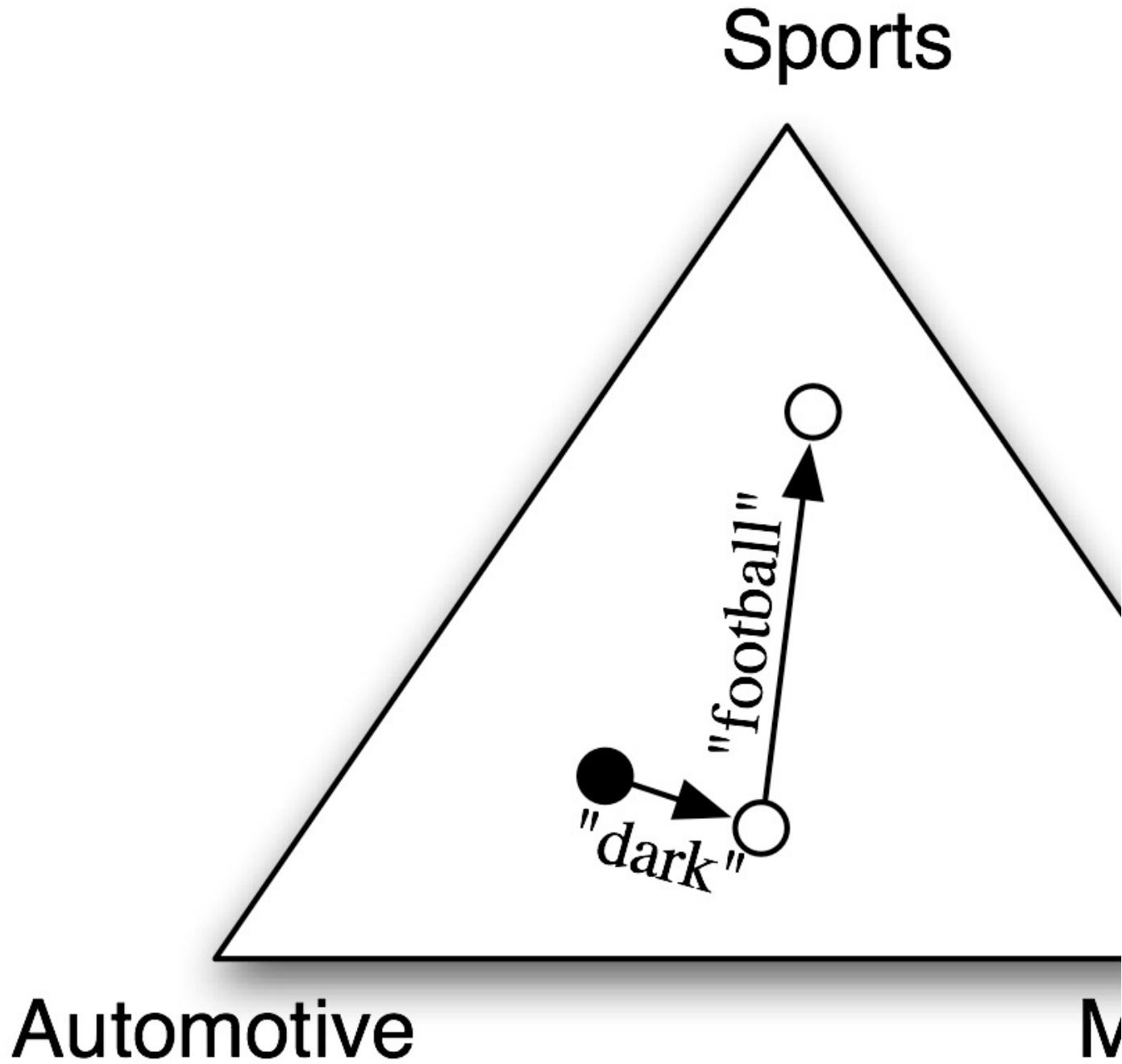
A second problem with decision trees is that they force features to be checked in a specific order, even when features may act relatively independently of one another. For example, when classifying documents into topics (such as sports, automotive, or murder mystery), features such as `hasword(football)` are highly indicative of a specific label, regardless of what other the feature values are. Since there is limited space near the top of the decision tree, most of these features will need to be repeated on many different branches in the tree. And since the number of branches increases exponentially as we go down the tree, the amount of repetition can be very large.

A related problem is that decision trees are not good at making use of features that are weak predictors of the correct label. Since these features make relatively small incremental improvements, they tend to occur very low in the decision tree. But by the time the decision tree learner has descended far enough to use these features, there is not enough training data left to reliably determine what effect they should have. If we could instead look at the effect of these features across the entire training corpus, then we might be able to make some conclusions about how they should affect the choice of label.

The fact that decision trees require that features be checked in a specific order limits their ability to make use of features that are relatively independent of one another. The Naive Bayes classification method, which we'll discuss next, overcomes this limitation by allowing all features to act "in parallel."

## Naive Bayes Classifiers

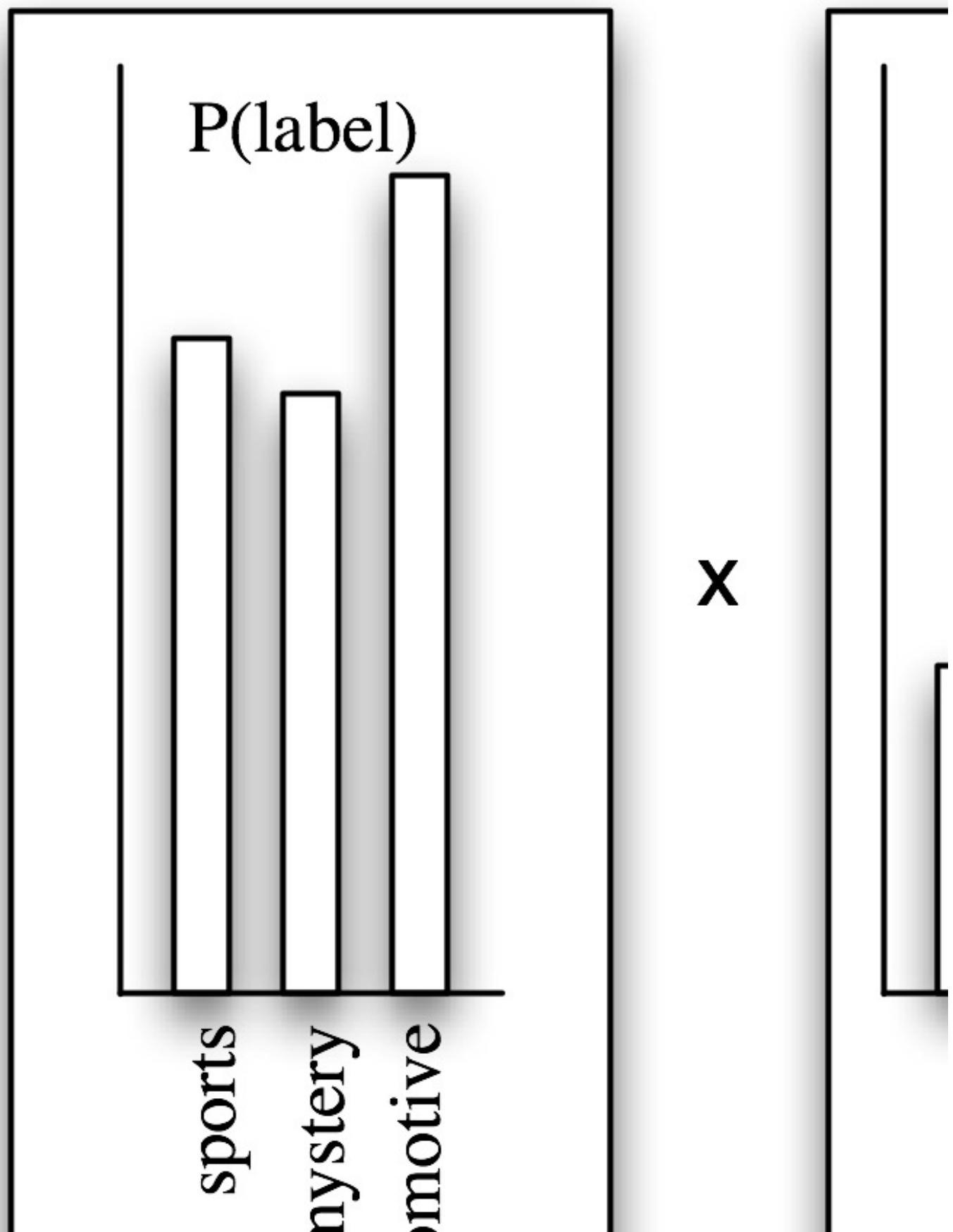
In **Naive Bayes** classifiers, every feature gets a say in determining which label should be assigned to a given input value. To choose a label for an input value, the Naive Bayes classifier begins by calculating the **prior probability** of each label, which is determined by checking frequency of each label in the training corpus. The contribution from each feature is then combined with this prior probability, to arrive at a likelihood estimate for each label. The label whose likelihood estimate is the highest is then assigned to the input value. [Figure 5.14](#) illustrates this process.



**Figure 5.14:** An abstract illustration of the procedure used by the Naive Bayes classifier to choose the topic for a document. In the training corpus, most documents are automotive, so the classifier starts out at a pointer closer to the "automotive" label. But it then considers the effect of each feature. In this example, the input document contains the word "dark," which is a weak indicator for murder mysteries; but it also contains the word "football," which is a strong indicator for sports documents. After every feature has made its contribution, the classifier checks which label it is closest to, and assigns that label to the input.

Individual features make their contribution to the overall decision by "voting against" labels that don't occur with that feature very often. In particular, the likelihood score for each label is reduced by multiplying it by the probability that an input value with that label would have the feature. For example, if the word "run" occurs in 12% of the sports documents, 10% of the murder mystery documents, and 2% of the automotive documents, then the likelihood score for the sports label will be multiplied by 0.12; the likelihood score for the murder mystery label will be multiplied by 0.1; and the likelihood score for the automotive label will be multiplied by 0.02. The overall effect will be to reduce the score of the murder mystery label slightly more than the score of the sports label; and to significantly reduce the automotive label with respect to the other two labels. This overall process is illustrated in [Figure 5.15](#).

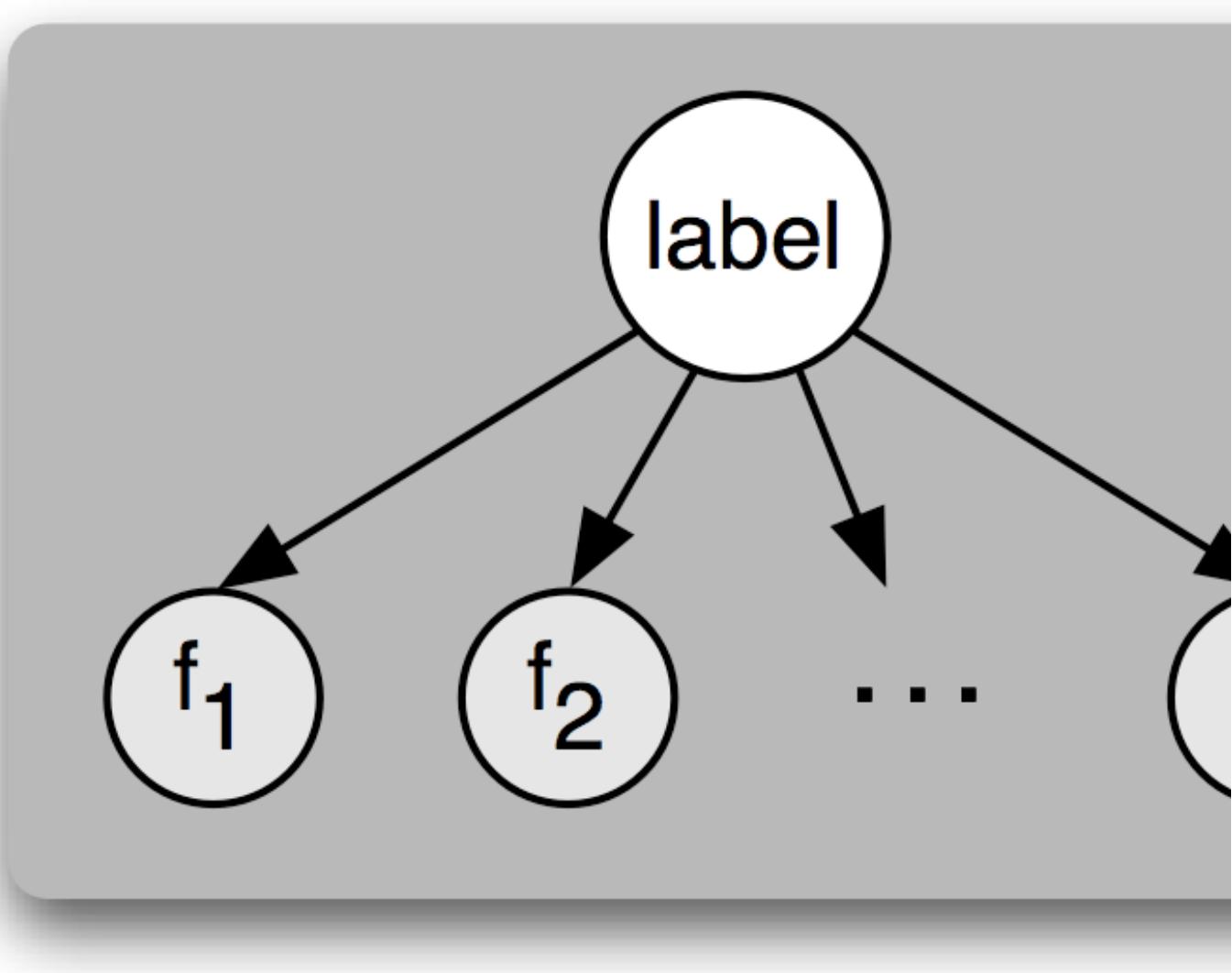
# Prior Probabilities



**Figure 5.15:** Calculating label likelihoods with Naive Bayes. Naive Bayes begins by calculating the prior probability of each label, based on how frequently each label occurs in the training data. Every feature then contributes to the likelihood estimate for each label, by multiplying it by the probability that input values with that label will have that feature. The resulting likelihood score can be thought of as an estimate of the probability that a randomly selected value from the training corpus would have both the given label and the set of features, assuming that the feature probabilities are all independent.

## Underlying Probabilistic Model

Another way of understanding the Naive Bayes classifier is that it chooses the most likely label for an input, under the assumption that every input value is generated by first choosing a class label for that input value, and then generating each feature, entirely independent of every other feature. Of course, this assumption is unrealistic: features are often highly dependent on one another in ways that don't just reflect differences in the class label. We'll return to some of the consequences of this assumption at the end of this section. But making this simplifying assumption makes it much easier to combine the contributions of the different features, since we don't need to worry about how they should interact with one another.



**Figure 5.16: A Bayesian Network Graph** illustrating the generative process that is assumed by the Naive Bayes classifier. To generate a labeled input, the model first chooses a label for the input; and then it generates each of the input's features based on that label. Every feature is assumed to be entirely independent of every other feature, given the label.

Based on this assumption, we can calculate an expression for  $P(\text{label}|\text{features})$ , the probability that an input will have a particular label, given that it has a particular set of features. To choose a label for a new input, we can then simply pick the label  $l$  that maximizes  $P(l|\text{features})$ .

To begin, we note that  $P(\text{label}|\text{features})$  is equal to the probability that an input has a particular label *and* the specified set of

features, divided by the probability that it has the specified set of features:

$$P(\text{label}|\text{features}) = P(\text{features}, \text{label})/P(\text{features})$$

Next, we note that  $P(\text{features})$  will be the same for every choice of label, so if we are simply interested in finding the most likely label, it suffices to calculate  $P(\text{features}, \text{label})$ , which we'll call the label likelihood.

### Note

If we want to generate a probability estimate for each label, rather than just choosing the most likely label, then the easiest way to compute  $P(\text{features})$  is to simply calculate the sum over labels of  $P(\text{features}, \text{label})$ :

$$P(\text{features}) = \sum_{\{l \text{ in label}\}} P(\text{features}, \text{label})$$

The label likelihood can be expanded out as the probability of the label times the probability of the features given the label:

$$P(\text{features}, \text{label}) = P(\text{label}) * P(\text{features}|\text{label})$$

Furthermore, since the features are all independent of one another (given the label), we can separate out the probability of each individual feature:

$$P(\text{features}, \text{label}) = P(\text{label}) * \prod_{\{f \text{ in features}\}} P(f|\text{label})$$

This is exactly the equation we discussed above for calculating the label likelihood:  $P(\text{label})$  is the prior probability for a given label, and each  $P(f|\text{label})$  is the contribution of a single feature to the label likelihood.

## Zero Counts and Smoothing

The simplest way to calculate  $P(f|\text{label})$ , the contribution of a feature  $f$  toward the label likelihood for a label  $\text{label}$ , is to take the percentage of training instances with the given label that also have the given feature:

$$P(f|\text{label}) = \text{count}(f, \text{label}) / \text{count}(\text{label})$$

However, this simple approach can become problematic when a feature *never* occurs with a given label in the training corpus. In this case, our calculated value for  $P(f|\text{label})$  will be zero, which will cause the label likelihood for the given label to be zero. Thus, the input will never be assigned this label, regardless of how well the other features fit the label.

The basic problem here is with our calculation of  $P(f|\text{label})$ , the probability that an input will have a feature, given a label. In particular, just because we haven't seen a feature/label combination occur in the training corpus, doesn't mean it's impossible for that combination to occur. For example, we may not have seen any murder mystery documents that contained the word "football," but we wouldn't want to conclude that it's completely impossible for such documents to exist.

Thus, although  $\text{count}(f, \text{label})/\text{count}(\text{label})$  is a good estimate for  $P(f|\text{label})$  when  $\text{count}(f, \text{label})$  is relatively high, this estimate becomes less reliable when  $\text{count}(f)$  becomes smaller. Therefore, when building Naive Bayes models, we usually make use of more sophisticated techniques, known as **smoothing** techniques, for calculating  $P(f|\text{label})$ , the probability of a feature given a label. For example, the "Expected Likelihood Estimation" for the probability of a feature given a label basically adds 0.5 to each  $\text{count}(f, \text{label})$  value; and the "Heldout Estimation" uses a heldout corpus to calculate the relationship between feature frequencies and feature probabilities. For more information on smoothing techniques, see <>ref -- manning & schutze?>>.

## Non-Binary Features

We have assumed here that each feature is binary -- in other words that each input either has a feature or does not. Label-valued features (e.g., a color feature which could be red, green, blue, white, or orange) can be converted to binary features by replacing them with features such as "color-is-red". Numeric features can be converted to binary features by **binning**, which replaces them with features such as "4<x<6".

Another alternative is to use regression methods to model the probabilities of numeric features. For example, if we assume that the height feature is gaussian, then we could estimate  $P(\text{height}|\text{label})$  by finding the mean and variance of the heights of the

inputs with each label. In this case,  $P(f=v|label)$  would not be a fixed value, but would vary depending on the value of  $v$ .

## The Naivete of Independence

The reason that Naive Bayes classifiers are called "naive" is that it's unreasonable to assume that all features are independent of one another (given the label). In particular, almost all real-world problems contain features with varying degrees of dependence on one another. If we had to avoid any features that were dependent on one another, it would be very difficult to construct good feature sets that provide the required information to the machine learning algorithm.

So what happens when we ignore the independence assumption, and use the Naive Bayes classifier with features that are not independent? One problem that arises is that the classifier can end up "double-counting" the effect of highly correlated features, pushing the classifier closer to a given label than is justified.

To see how this can occur, consider a name gender classifier that contains two identical features,  $f\_1$  and  $f\_2$ . In other words,  $f\_2$  is an exact copy of  $f\_1$ , and contains no new information. Nevertheless, when the classifier is considering an input, it will include the contribution of both  $f\_1$  and  $f\_2$  when deciding which label to choose. Thus, the information content of these two features is given more weight than it should be.

Of course, we don't usually build Naive Bayes classifiers that contain two identical features. However, we do build classifiers that contain features which are dependent on one another. For example, the features `ends-with(a)` and `ends-with(vowel)` are dependent on one another, because if an input value has the first feature, then it must also have the second feature. For features like these, the duplicated information may be given more weight than is justified by the training corpus.

## The Cause of Double-Counting

The basic problem that causes this double-counting issue is that during training, feature contributions are computed separately; but when using the classifier to choose labels for new inputs, those feature contributions are combined. One solution, therefore, is to consider the possible interactions between feature contributions during training. We could then use those interactions to adjust the contributions that individual features make.

To make this more precise, we can rewrite the equation used to calculate out the likelihood of a label, separating out the contribution made by each feature (or label):

$$P(features, label) = w[label] * \prod_{f \in features} w[f, label]$$

Here,  $w[label]$  is the "starting score" for a given label; and  $w[f, label]$  is the contribution made by a given feature towards a label's likelihood. We call these values  $w[label]$  and  $w[f, label]$  the **parameters** or **weights** for the model. Using the Naive Bayes algorithm, we set each of these parameters independently:

$$w[label] = P(label)$$

$$w[f, label] = P(f|label)$$

However, in the next section, we'll look at a classifier that considers the possible interactions between these parameters when choosing their values.

## 5.9 Maximum Entropy Classifiers

The **Maximum Entropy** classifier uses a model that is very similar to the model used by the Naive Bayes classifier. But rather than using probabilities to set the model's parameters, it uses search techniques to find a set of parameters that will maximize the performance of the classifier. In particular, it looks for the set of parameters that maximizes the **total likelihood** of the training corpus, which is defined as:

```
\sum_{(x) in corpus} P(label(x) | features(x))
```

Where  $P(label|features)$ , the probability that an input whose features are `features` will have class label `label`, is defined as:

$$P(label|features) = P(label, features) / \sum_{label} P(label, features)$$

Because of the potentially complex interactions between the effects of related features, there is no way to directly calculate the model parameters that maximize the likelihood of the training corpus. Therefore, Maximum Entropy classifiers choose the model parameters using **iterative optimization** techniques, which initialize the model's parameters to random values, and then repeatedly refine those parameters to bring them closer to the optimal solution. The iterative optimization techniques guarantee that each refinement of the parameters will bring them closer to the optimal values; but do not necessarily provide a means of determining when those optimal values have been reached. Because the parameters for Maximum Entropy classifiers are selected using iterative optimization techniques, they can take a long time to train. This is especially true when the size of the training corpus, the number of features, and the number of labels are all large.

#### Note

Some iterative optimization techniques are much faster than others. When training Maximum Entropy models, avoid the use of Generalized Iterative Scaling (GIS) or Improved Iterative Scaling (IIS), which are both considerably slower than the Conjugate Gradient (CG) and the BFGS optimization methods.

- the technique, of fixing the form of the model, and searching for model parameters that optimize some evaluation metric is called optimization.
- a number of other machine learning algorithms can be thought of as optimization systems.

## 5.10 Exercises

1. ☀ Read up on one of the language technologies mentioned in this section, such as word sense disambiguation, semantic role labeling, question answering, machine translation, named entity detection. Find out what type and quantity of annotated data is required for developing such systems. Why do you think a large amount of data is required?
2. ☀ Exercise: compare the performance of different machine learning methods. (they're still black boxes at this point)
3. ☀ The synonyms *strong* and *powerful* pattern differently (try combining them with *chip* and *sales*).
4. ● Accessing extra features from WordNet to augment those that appear directly in the text (e.g. hypernym of any monosemous word)
5. ★ Task involving PP Attachment data; predict choice of preposition from the nouns.
6. ★ Suppose you wanted to automatically generate a prose description of a scene, and already had a word to uniquely describe each entity, such as *the jar*, and simply wanted to decide whether to use *in* or *on* in relating various items, e.g. *the book is in the cupboard vs the book is on the shelf*. Explore this issue by looking at corpus data; writing programs as needed.

(14)

- a. in the car *vs* on the train
- b. in town *vs* on campus
- c. in the picture *vs* on the screen
- d. in Macbeth *vs* on Letterman

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

## 6 Structured Programming in Python

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 we write well-structured, readable programs that you and others will be able to re-use 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 we 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.

#### Note

Remember that our program samples assume you begin your interactive session or your program with: `import nltk, re, pprint`

## 6.1 Back to the Basics

### Iteration

If you've come to Python from another programming language, you may be inclined to make heavy use of loop variables, and no use of list comprehensions. In this section we'll see how avoiding loop variables can lead to more readable code. We'll also look at the relationship between loops with nested blocks vs list comprehensions, and discuss when to use either construct.

Let's look at a familiar technique for iterating over the members of a list by initializing an index `i` and then incrementing the index each time we pass through the loop:

```
>>> sent = ['I', 'am', 'the', 'Walrus']
>>> i = 0
>>> while i < len(sent):
... print sent[i].lower(),
... i += 1
...
i am the walrus
```

Although this does the job, it is not idiomatic Python. It is almost never a good idea to use loop variables in this way. Observe that Python's `for` statement allows us to achieve the same effect, and that the code is not just more succinct, but more readable:

```
>>> for w in sent:
... print w.lower(),
...
i am the walrus
```

This is much more readable, and `for` loops with nested code blocks will be our preferred way of producing printed output. However, the above two programs have the same subtle problem: both print a trailing space character. If we meant to include this output with surrounding markup, we would see something like `<sent>i am the walrus </sent>`. Similarly, both programs need an extra `print` statement in order to produce a newline character at the end. A third solution using a list comprehension is more compact again

```
>>> print ''.join(w.lower() for w in sent)
i am the walrus
```

This doesn't produce an extra space character, and does produce the required newline. However, it is less readable thanks to the use of `' '.join()`, and we will usually prefer to use the second solution above for code that prints output.

Another case where loop variables seem to be necessary is for printing the value of 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] * 100.0 / 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 better solution uses two list comprehensions as shown below. We sacrifice some efficiency by having two passes through the data, but the result is more transparent.

```
>>> 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. In most cases we actually prefer to get all solutions, though it's easy to find them all and only report one back to the user. In contrast, it is difficult to modify the first program to find all solutions. 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 very fast. 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.

List comprehensions have a surprising range of uses. Here's an example of how they can be used to generate all combinations of some collections of words. Here we generate all combinations of two determiners, two adjectives, and two nouns. The list comprehension is split across three lines for readability.

```
>>> [(det,adj,noun) for det in ('two', 'three')
... for adj in ('old', 'blind')
... for noun in ('men', 'mice')]
[('two', 'old', 'men'), ('two', 'old', 'mice'), ('two', 'blind', 'men'),
 ('two', 'blind', 'mice'), ('three', 'old', 'men'), ('three', 'old', 'mice'),
 ('three', 'blind', 'men'), ('three', 'blind', 'mice')]
```

Our use of list comprehensions has helped us avoid loop variables. However, 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 that will be discussed in the next 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])]]
```

## Assignment

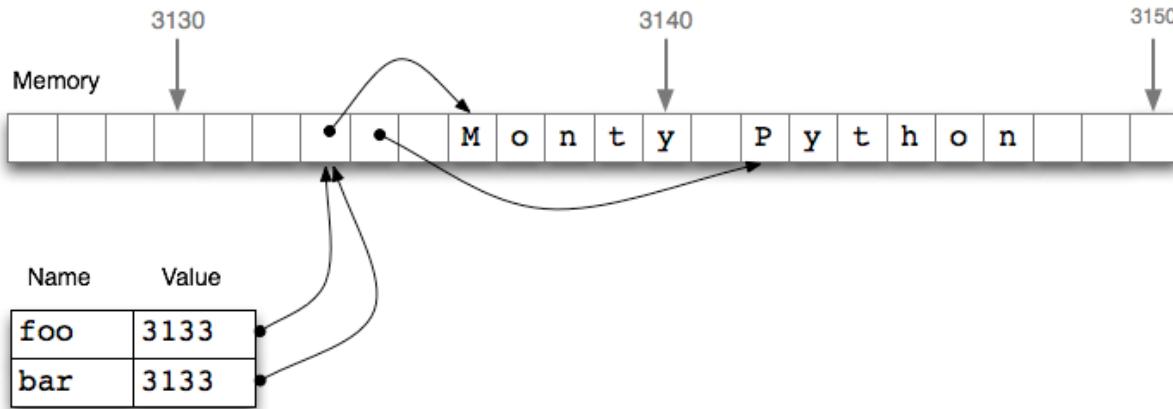
Python's assignment statement operates on *values*. But what is a value? Consider the following code fragment:

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

This code shows that when we write `bar = foo`, 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'`, the value of `bar` is not affected.

However, assignment statements do not always involve making copies in this way. An important subtlety of Python is that the "value" of a structured object (such as a list) is actually a *reference* to the object. In the following example, we assign the reference of `foo` to the new variable `bar`. When we modify something inside `foo`, we can see that the contents of `bar` have also been changed.

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

**Figure 6.1:** List Assignment and Computer Memory

Thus, the line `bar = foo` 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 6.1](#), we see that a list `sent1` 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 `sent2 = sent1`, it is just the object reference 3133 that gets copied.

This behavior extends to other aspects of the Python language. In [Section 6.2](#) we will see how it effects the way parameters are passed into functions. Here's an example of how it applies to copying:

```
>>> 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.

#### Note

**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.

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 the object references, but not the objects themselves. 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.

#### Note

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()`.

## Sequences

We have seen three kinds of sequence object: strings, lists, and tuples. As sequences, they have some common properties: they can be indexed and they have a length:

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

We can iterate over the items in a sequence `s` in a variety of useful ways, as shown in [Table 6.1](#).

**Table 6.1:**

Various ways to iterate over sequences

| Python Expression                             | Comment                                                       |
|-----------------------------------------------|---------------------------------------------------------------|
| <code>for item in s</code>                    | iterate over the items of <code>s</code>                      |
| <code>for item in sorted(s)</code>            | iterate over the items of <code>s</code> in order             |
| <code>for item in set(s)</code>               | iterate over unique elements of <code>s</code>                |
| <code>for item in reversed(s)</code>          | iterate over elements of <code>s</code> in reverse            |
| <code>for item in set(s).difference(t)</code> | iterate over elements of <code>s</code> not in <code>t</code> |
| <code>for item in random.shuffle(s)</code>    | iterate over elements of <code>s</code> in random order       |

The sequence functions illustrated in [Table 6.1](#) 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)`.

Notice in the above 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.

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[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 and which can be handy for language processing. Thus, `zip()` takes the items of two sequences and "zips" them together into a single list of pairs. Given a sequence `s`, `enumerate(s)` returns an iterator that produces a pair of an index and the item at that index.

```

>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> tags = ['NNP', 'VBD', 'IN', 'DT', 'NN']
>>> zip(words, tags)
[('I', 'NNP'), ('turned', 'VBD'), ('off', 'IN'),
 ('the', 'DT'), ('spectroroute', 'NN')]
>>> list(enumerate(words))
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]

```

## 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() [1]
>>> wordlens = [(len(word), word) for word in words] [2]
>>> wordlens
[(1, 'I'), (6, 'turned'), (3, 'off'), (3, 'the'), (12, 'spectroroute')]
>>> wordlens.sort() [3]
>>> ''.join([word for (count, word) in wordlens]) [4]
'I off the turned spectroroute'

```

Each of the above lines of code contains a significant feature. Line [1] demonstrates that a simple string is actually an object with methods defined on it, such as `split()`. Line [2] shows the construction of a list of tuples, where each tuple consists of a number (the word length) and the word, e.g. (3, 'the'). Line [3] sorts the list, modifying the list in-place. Finally, line [4] discards the length information then joins the words back into a single string.

We began by talking about the commonalities in these sequence types, but the above 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', 'DT', ['Di:', 'D@']),
... ('off', 'IN', ['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. Note that these pronunciations are stored using a list. (Why?)

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**, while tuples are **immutable**. In other words, lists can be modified, while tuples cannot. Here are some of the operations on lists that do in-place modification of the list. None of these operations is permitted on a tuple, a fact you should confirm for yourself.

```

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

```

## Stacks and Queues

Lists are a particularly versatile data type. We can use lists to implement higher-level data types such as stacks and queues. A **stack** is a container that has a **last-in-first-out** policy for adding and removing items (see [Figure 6.2](#)).

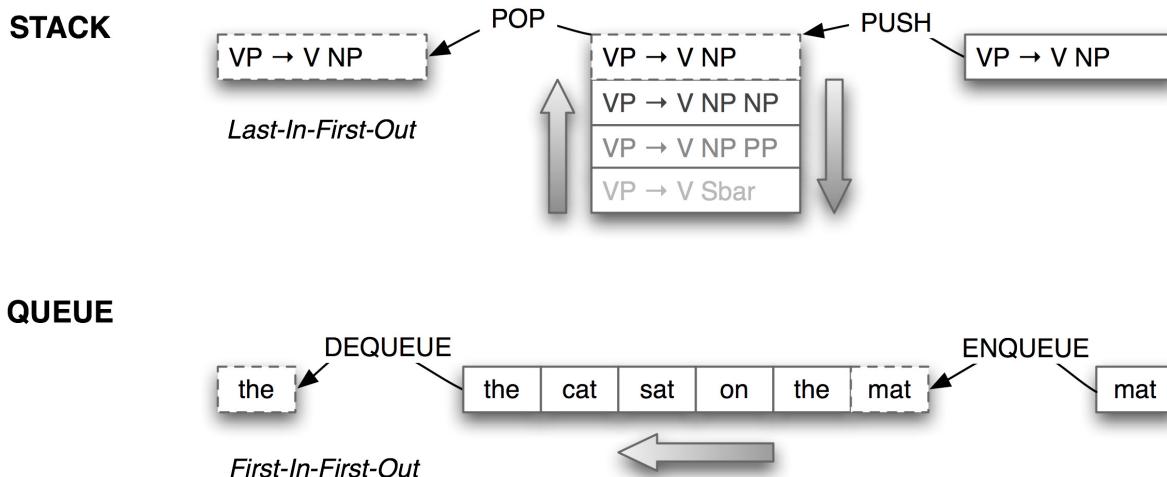


Figure 6.2: Stacks and Queues

Stacks are used to keep track of the current context in computer processing of natural languages (and programming languages too). We will seldom have to deal with stacks explicitly, as the implementation of NLTK parsers, treebank corpus readers, (and even Python functions), all use stacks behind the scenes. However, it is important to understand what stacks are and how they work.

```
def check_parens(tokens):
 stack = []
 for token in tokens:
 if token == '(': # push
 stack.append(token)
 elif token == ')': # pop
 stack.pop()
 return stack

>>> phrase = "(the cat) (sat (on (the mat))"
>>> print check_parens(phrase.split())
['(', ')']
```

Figure 6.3 (check\_parens.py): Figure 6.3: Check parentheses are balanced

In Python, we can treat a list as a stack by limiting ourselves to the three operations defined on stacks: `append(item)` (to push `item` onto the stack), `pop()` to pop the item off the top of the stack, and `[-1]` to access the item on the top of the stack. The program in Figure 6.3 processes a sentence with phrase markers, and checks that the parentheses are balanced. The loop pushes material onto the stack when it gets an open parenthesis, and pops the stack when it gets a close parenthesis. We see that two are left on the stack at the end; i.e. the parentheses are not balanced.

Although the program in Figure 6.3 is a useful illustration of stacks, it is overkill because we could have done a direct count: `phrase.count('(') == phrase.count(')')`. However, we can use stacks for more sophisticated processing of strings containing nested structure, as shown in Figure 6.4. Here we build a (potentially deeply-nested) list of lists. Whenever a token other than a parenthesis is encountered, we add it to a list at the appropriate level of nesting. The stack cleverly keeps track of this level of nesting, exploiting the fact that the item at the top of the stack is actually shared with a more deeply nested item. (Hint: add diagnostic print statements to the function to help you see what it is doing.)

```
def convert_parens(tokens):
 stack = [[]]
 for token in tokens:
 if token == '(': # push
 sublist = []
 stack[-1].append(sublist)
 stack.append(sublist)
 elif token == ')': # pop
 stack.pop()
 else: # update top of stack
 stack[-1].append(token)
 return stack[0]
```

```
>>> phrase = "(the cat) (sat (on (the mat)))"
>>> print convert_parens(phrase.split())
[['the', 'cat'], ['sat', ['on', ['the', 'mat']]]]
```

**Figure 6.4 (convert\_parens.py):** Figure 6.4: Convert a nested phrase into a nested list using a stack

Lists can be used to represent another important data structure. A **queue** is a container that has a **first-in-first-out** policy for adding and removing items (see [Figure 6.2](#)). Queues are used for scheduling activities or resources. As with stacks, we will seldom have to deal with queues explicitly, as the implementation of NLTK n-gram taggers ([Section 4.5](#)) and chart parsers ([Section 8.5](#)) use queues behind the scenes. However, we will take a brief look at how queues are implemented using lists.

```
>>> queue = ['the', 'cat', 'sat']
>>> queue.append('on')
>>> queue.append('the')
>>> queue.append('mat')
>>> queue.pop(0)
'the'
>>> queue.pop(0)
'cat'
>>> queue
['sat', 'on', 'the', 'mat']
```

## Conditionals

In the condition part of an `if` statement, a nonempty 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.

## 6.2 Functions

Once you have been programming for a while, you will find that you need to perform a task that you have done in the past. In fact, over time, the number of completely novel things you have to do in creating a program decreases significantly. Half of the work may involve simple tasks that you have done before. Thus it is important for your code to be *re-usable*. One effective way to do this is to abstract commonly used sequences of steps into a **function**.

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()`:

```

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

```

[Figure 6.5 \(get\\_text.py\)](#): Figure 6.5: Read text from a file

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.

Notice that a function definition consists of the keyword `def` (short for "define"), followed by the function name, followed by a sequence of parameters enclosed in parentheses, then a colon. The following lines contain an indented block of code, the **function body**.

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 above 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 the above 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 re-use 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 that we forget some important step, or introduce 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.

- [More: overview of section]

## Function Arguments

- multiple arguments
- named arguments
- default values

Python is a **dynamically typed** language. It does not force us to declare the type of a variable when we write a program. This feature is often useful, as it 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.

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. Observe that the `tag()` function in [Figure 6.6](#) behaves sensibly for string arguments, but that it does not complain when it is passed a dictionary.

```

def tag(word):
 if word in ['a', 'the', 'all']:
 return 'DT'
 else:
 return 'NN'

>>> tag('the')
'DT'
>>> tag('dog')

```

```
'NN'
>>> tag({'lexeme':'turned', 'pos':'VBD', 'pron':['t3:nd', 't3`nd']})
'NN'
```

[Figure 6.6 \(tag1.py\)](#): Figure 6.6: A tagger that tags anything

It would be helpful if the author of this function took 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 and return a diagnostic value, such as Python's special empty value, `None`, as shown in [Figure 6.7](#).

```
def tag(word):
 if not type(word) is str:
 return None
 if word in ['a', 'the', 'all']:
 return 'DT'
 else:
 return 'NN'
```

[Figure 6.7 \(tag2.py\)](#): Figure 6.7: A tagger that only tags strings

However, this approach is dangerous because the calling program may not detect the error, and the diagnostic return value may be propagated to later parts of the program with unpredictable consequences. A better solution is shown in [Figure 6.8](#).

```
def tag(word):
 if not type(word) is str:
 raise ValueError, "argument to tag() must be a string"
 if word in ['a', 'the', 'all']:
 return 'DT'
 else:
 return 'NN'
```

[Figure 6.8 \(tag3.py\)](#): Figure 6.8: A tagger that generates an error message when not passed a string

This produces an error that cannot be ignored, since it halts program execution. Additionally, the error message is easy to interpret. (We will see an even better approach, known as "duck typing" in Section [XREF].)

Another aspect of defensive programming concerns the return statement of a function. In order to be confident that all execution paths through a function lead to a return statement, it is best to have a single return statement at the end of the function definition. This approach has a further benefit: it makes it more likely that the function will only return a single type. Thus, the following version of our `tag()` function is safer:

```
>>> def tag(word):
... result = 'NN' # default value, a string
... if word in ['a', 'the', 'all']: # in certain cases...
... result = 'DT' # overwrite the value
... return result # all paths end here
```

A return statement can be used to pass multiple values back to the calling program, by packing them into a tuple. Here we define a function that returns a tuple consisting of the average word length of a sentence, and the inventory of letters used in the sentence. It would have been clearer to write two separate functions.

```
>>> def proc_words(words):
... avg_wordlen = sum(len(word) for word in words)/len(words)
... chars_used = ''.join(sorted(set(''.join(words))))
... return avg_wordlen, chars_used
>>> proc_words(['Not', 'a', 'good', 'way', 'to', 'write', 'functions'])
(3, 'Nacdefginorstuvwxyz')
```

Functions do not need to have a return statement at all. 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. Consider the following three sort functions; the last approach is dangerous because a programmer could use it without realizing that it had modified its input.

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

```

...
 l.sort()
...
 return l

```

## An Important Subtlety

Back in [Section 6.1](#) you saw that in Python, 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**). Consider [Figure 6.9](#). Function `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 = 'cat'
 properties['pos'] = 'noun'

>>> w = ''
>>> p = {}
>>> set_up(w, p)
>>> w
''
>>> p
{'pos': 'noun'}

```

[Figure 6.9 \(call\\_by\\_value.py\)](#): Figure 6.9

To understand why `w` was not changed, it is necessary to understand call-by-value. 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 had no effect on the external value of `w`. This parameter passing is identical to the following sequence of assignments:

```

>>> w = ''
>>> word = w
>>> word = 'cat'
>>> w
''

```

In the case of the structured object, matters are quite different. When we called `set_up(w, p)`, the value of `p` (an empty dictionary) was assigned to a new local variable `properties`. Since the value of `p` is an object reference, both variables now reference the same memory location. Modifying something inside `properties` will also change `p`, just as if we had done the following sequence of assignments:

```

>>> p = {}
>>> properties = p
>>> properties['pos'] = 'noun'
>>> p
{'pos': 'noun'}

```

Thus, to understand Python's call-by-value parameter passing, it is enough to understand Python's assignment operation. We will address some closely related issues in our discussion of variable scope 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, e.g.

```

>>> 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 [Figure 6.10](#). 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.

```
def freq_words(url, freqdist, n):
 text = nltk.clean_url(url)
 for word in nltk.wordpunct_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']
```

[Figure 6.10 \(freq\\_words1.py\)](#): Figure 6.10

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 [Figure 6.11](#) we **refactor** this function, and simplify its interface by providing a single `url` parameter.

```
def freq_words(url):
 freqdist = nltk.FreqDist()
 text = nltk.clean_url(url)
 for word in nltk.wordpunct_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']
```

[Figure 6.11 \(freq\\_words2.py\)](#): Figure 6.11

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.wordpunct_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']
```

## Modularity

Variable scope

- local and global variables
- scope rules
- global variables introduce dependency on context and limits the reusability of a function
- importance of avoiding side-effects
- functions hide implementation details

## Documentation (notes)

- some guidelines for literate programming (e.g. variable and function naming)
- documenting functions (user-level and developer-level documentation)

NLTK's docstrings are (mostly) written using the "epytext" markup language. Using an explicit markup language allows us to generate prettier online documentation, e.g. see:

<http://nltk.org/doc/api/nltk.tree.Tree-class.html>

[Examples of function-level docstrings with epytext markup.]

## Functions as Arguments

So far the arguments we have passed into functions have been simple objects like strings, or structured objects like lists. These arguments allow us to parameterize the behavior of a function. As a result, functions are very flexible and powerful abstractions, permitting us to repeatedly apply the *same operation* on *different data*. 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 parameters 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', '.']
```

Surprisingly, `len` and `last_letter` are objects that can be passed around like lists and dictionaries. Notice that parentheses are only used after a function name if we are invoking the function; when we are simply passing the function around as an object these are not used.

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 above `last_letter()` function in multiple places, and thus no need to give it a name. 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 lexicographic 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', ',', '.']
```

## Higher-Order Functions

In 6.1 we saw an example of filtering out some items in a list comprehension, using an `if` test. Sometimes list comprehensions get cumbersome, since they can mention the same variable many times, e.g.: `[word for word in sent if property(word)]`. We can perform the same task more succinctly as follows:

```
>>> def is_lexical(word):
... return word.lower() not in ['a', 'of', 'the', 'and', 'will', ',', '.', '.']
>>> filter(is_lexical, sent)
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

The function `is_lexical(word)` returns `True` just in case `word`, when normalized to lowercase, is not in the given list. This function is itself used as an argument to `filter()`. The `filter()` function applies its first argument (a function) to each item of its second (a sequence), only passing it through if the function returns true for that item. Thus `filter(f, seq)` is equivalent to `[item for item in seq if f(item)]`.

Another helpful function, which like `filter()` applies a function to a sequence, is `map()`. Here is a simple way to find the average length of a sentence in a section of the Brown Corpus:

```
>>> lengths = map(len, nltk.corpus.brown.sents(categories='news'))
>>> sum(lengths) / float(len(lengths))
21.7508111616
```

Instead of `len()`, we could have passed in any other function we liked:

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> def is_vowel(letter):
... return letter.lower() in "aeiou"
>>> def vowelcount(word):
... return len(filter(is_vowel, word))
>>> map(vowelcount, sent)
[1, 1, 2, 1, 1, 3]
```

Instead of using `filter()` to call a named function `is_vowel`, we can define a lambda expression as follows:

```
>>> map(lambda w: len(filter(lambda c: c.lower() in "aeiou", w)), sent)
[1, 1, 2, 1, 1, 3]
```

We can check that all or any items meet some condition:

```
>>> all(len(w) > 4 for w in sent)
False
>>> any(len(w) > 4 for w in sent)
True
```

The higher order functions like `map` and `filter` are certainly useful, but in general it is better to stick to using list comprehensions since they are often more readable.

## Named Arguments

One of the difficulties in re-using functions is remembering the order of arguments. Consider the following function, that finds the `n` most frequent words that are at least `min_len` characters long:

```
>>> def freq_words(file, min, num):
... text = open(file).read()
... tokens = nltk.wordpunct_tokenize(text)
... freqdist = nltk.FreqDist(t for t in tokens if len(t) >= min)
... return freqdist.keys()[:num]
>>> freq_words('ch01.rst', 4, 10)
['words', 'that', 'text', 'word', 'Python', 'with', 'this', 'have', 'language', 'from']
```

This function has three arguments. It follows the convention of listing the most basic and substantial argument first (the file). However, it might be hard to remember the order of the second and third arguments on subsequent use. We can make this function more readable by using **keyword arguments**. These appear in the function's argument list with an equals sign and a default value:

```
>>> def freq_words(file, min=1, num=10):
... text = open(file).read()
... tokens = nltk.wordpunct_tokenize(text)
... freqdist = nltk.FreqDist(t for t in tokens if len(t) >= min)
... return freqdist.keys()[:num]
```

Now there are several equivalent ways to call this function: `freq_words('ch01.rst', 4, 10)`, `freq_words('ch01.rst', min=4, num=10)`, `freq_words('ch01.rst', num=10, min=4)`.

When we use an integrated development environment such as IDLE, simply typing the name of a function at the command prompt will list the arguments. Using named arguments helps someone to re-use the code...

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, e.g.:

```
>>> def freq_words(file, min=1, num=10, trace=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.wordpunct_tokenize(text):
... if len(word) >= min:
... freqdist.inc(word)
... if trace and freqdist.N() % 100 == 0: print "."
... if trace: print
... return freqdist.keys()[:num]
```

## 6.3 Iterators

[itertools, bigrams vs ibigrams, efficiency, ...]

### 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). The standard way to do this is to initialize an empty list, accumulate the material, then return the list, as shown in function `find_nouns1()` in [Listing 6.12](#).

```
def find_nouns1(tagged_text):
 nouns = []
 for word, tag in tagged_text:
 if tag[:2] == 'NN':
 nouns.append(word)
 return nouns

>>> tagged_text = [('the', 'DT'), ('cat', 'NN'), ('sat', 'VBD'),
... ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
>>> find_nouns1(tagged_text)
['cat', 'mat']
```

[Figure 6.12 \(find\\_nouns1.py\)](#): Figure 6.12: Accumulating Output into a List

A superior way to perform this operation is define the function to be a **generator**, as shown in [Listing 6.13](#). The first time this function is called, it gets as far as the `yield` statement and stops. 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.

```
def find_nouns2(tagged_text):
 for word, tag in tagged_text:
 if tag[:2] == 'NN':
 yield word

>>> tagged_text = [('the', 'DT'), ('cat', 'NN'), ('sat', 'VBD'),
... ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
>>> find_nouns2(tagged_text)
<generator object at 0x14b2f30>
>>> for noun in find_nouns2(tagged_text):
... print noun,
cat mat
>>> list(find_nouns2(tagged_text))
['cat', 'mat']
```

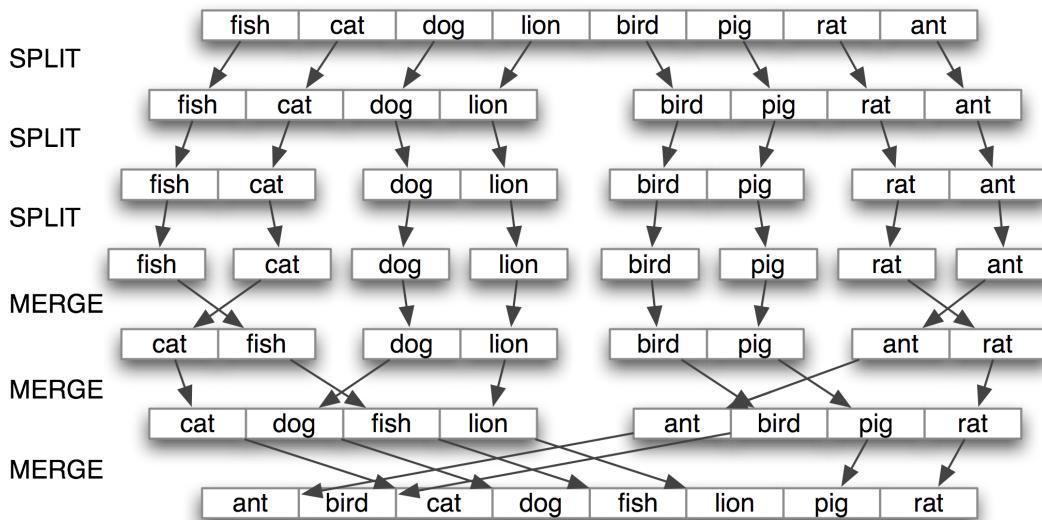
[Figure 6.13 \(find\\_nouns2.py\)](#): Figure 6.13: Defining a Generator Function

If we call the function directly we see that it returns a "generator object", which is not very useful to us. Instead, we can iterate over it directly, using `for noun in find_nouns(tagged_text)`, or convert it into a list, using `list(find_nouns(tagged_text))`.

## 6.4 Algorithm Design Strategies

A major part of algorithmic problem solving is selecting or adapting an appropriate algorithm for the problem at hand. Whole books are written on this topic (e.g. [Levitin, 2004]) 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. [Figure 6.14](#) illustrates this approach for sorting a list of words.



**Figure 6.14:** Sorting by Divide-and-Conquer (Mergesort)

Another strategy is **decrease-and-conquer**. In this approach, a small amount of work on a problem of size  $n$  permits us to reduce it to a problem of size  $n/2$ . [Figure 6.15](#) illustrates this approach for the problem of finding the index of an item in a sorted list.

./images/binary-search.png

**Figure 6.15:** Searching by Decrease-and-Conquer (Binary Search)

A third well-known strategy is **transform-and-conquer**. 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 duplicates entries in a list, we can **pre-sort** the list, then look for adjacent identical items, as shown in [Figure 6.16](#).

```
def duplicates(words):
 prev = None
 dup = [None]
 for word in sorted(words):
 if word == prev and word != dup[-1]:
 dup.append(word)
 else:
 prev = word
 return dup[1:]

>>> duplicates(['cat', 'dog', 'cat', 'pig', 'dog', 'cat', 'ant', 'cat'])
['cat', 'dog']
```

**Figure 6.16 (presorting.py):** Figure 6.16: Presorting a list for duplicate detection

### Recursion (notes)

We first saw recursion in [Chapter 3](#), in a function that navigated the hypernym hierarchy of WordNet...

Iterative solution:

```
>>> def factorial(n):
... result = 1
... for i in range(n):
... result *= (i+1)
... return result
```

Recursive solution (base case, induction step)

```
>>> def factorial(n):
... if n == 1:
... return n
... else:
... return n * factorial(n-1)
```

[Simple example of recursion on strings.]

Generating all permutations of words, to check which ones are grammatical:

```
>>> def perms(seq):
... if len(seq) <= 1:
... yield seq
... else:
... for perm in perms(seq[1:]):
... for i in range(len(perm)+1):
... yield perm[:i] + seq[0:1] + perm[i:]
>>> list(perms(['police', 'fish', 'cream']))
[['police', 'fish', 'cream'], ['fish', 'police', 'cream'],
 ['fish', 'cream', 'police'], ['police', 'cream', 'fish'],
 ['cream', 'police', 'fish'], ['cream', 'fish', 'police']]
```

## Deeply Nested Objects (notes)

We can use recursive functions to build deeply-nested objects. Building a letter trie, [Figure 6.17](#).

```
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

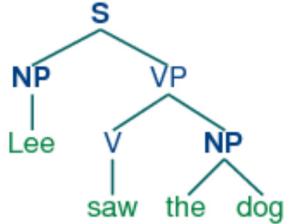
>>> trie = {}
>>> insert(trie, 'chat', 'cat')
>>> insert(trie, 'chien', 'dog')
>>> trie['c']['h']
{'a': {'t': {'value': 'cat'}}, 'i': {'e': {'n': {'value': 'dog'}}}}
>>> trie['c']['h']['a']['t']['value']
'cat'
>>> pprint.pprint(trie)
{'c': {'h': {'a': {'t': {'value': 'cat'}},
 'i': {'e': {'n': {'value': 'dog'}}}}}}
```

[Figure 6.17 \(trie.py\)](#): Figure 6.17: Building a Letter Trie

## Trees

A tree is a set of connected nodes, each of which is labeled with a category. It common to use a 'family' metaphor to talk about the relationships of nodes in a tree: for example, S is the **parent** of VP; conversely VP is a **daughter** (or **child**) of S. Also, since NP and VP are both daughters of S, they are also **sisters**. Here is an example of a tree:

(15)



Although it is helpful to represent trees in a graphical format, for computational purposes we usually need a more text-oriented representation. We will use the same format as the Penn Treebank, a combination of brackets and labels:

```

(S
 (NP Lee)
 (VP
 (V saw)
 (NP
 (Det the)
 (N dog))))
```

Here, the node value is a constituent type (e.g., NP or VP), and the children encode the hierarchical contents of the tree.

Although we will focus on syntactic trees, trees can be used to encode *any* homogeneous hierarchical structure that spans a sequence of linguistic forms (e.g. morphological structure, discourse structure). In the general case, leaves and node values do not have to be strings.

In NLTK, trees are created with the `Tree` constructor, which takes a node value and a list of zero or more children. Here's a couple of simple trees:

```

>>> tree1 = nltk.Tree('NP', ['John'])
>>> print tree1
(NP John)
>>> tree2 = nltk.Tree('NP', ['the', 'man'])
>>> print tree2
(NP the man)
```

We can incorporate these into successively larger trees as follows:

```

>>> tree3 = nltk.Tree('VP', ['saw', tree2])
>>> tree4 = nltk.Tree('S', [tree1, tree3])
>>> print tree4
(S (NP John) (VP saw (NP the man)))
```

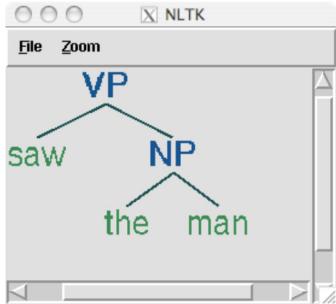
Here are some of the methods available for tree objects:

```

>>> print tree4[1]
(VP saw (NP the man))
>>> tree4[1].node
'VP'
>>> tree4.leaves()
['John', 'saw', 'the', 'man']
>>> tree4[1,1,1]
'man'
```

The printed representation for complex trees can be difficult to read. In these cases, the `draw` method can be very useful. It opens a new window, containing a graphical representation of the tree. The tree display window allows you to zoom in and out; to collapse and expand subtrees; and to print the graphical representation to a postscript file (for inclusion in a document).

```
>>> tree3.draw()
```



[To do: recursion on trees]

## 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 sub-problems. Instead of computing solutions to these sub-problems 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  $n$ . He 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 (2).

(16)  $V_4 =$   
 $LL, LSS$   
 i.e. L prefixed to each item of  $V_2 = \{L, SS\}$   
 $SSL, SLS, SSSS$   
 i.e. S prefixed to each item of  $V_3 = \{SL, LS, SSS\}$

```

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:

```

```

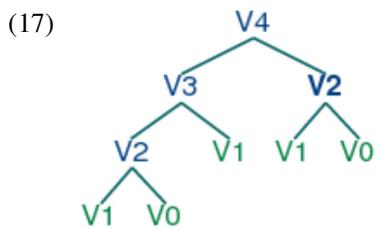
s = ["S" + prosody for prosody in virahanka4(n-1)]
l = ["L" + prosody for prosody in virahanka4(n-2)]
return s + l

>>> 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']

```

**Figure 6.18 (virahanka.py): Figure 6.18: Three Ways to Compute Sanskrit Meter**

With this observation, we can write a little recursive function called `virahanka1()` to compute these meters, shown in [Figure 6.18](#). 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 (3).



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 sub-problem 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 [Figure 6.18](#). 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 is to use a Python **decorator** called `memoize`, which takes care of the housekeeping work done by `virahanka3()` without cluttering up the program.

This concludes our brief introduction to dynamic programming. We will encounter it again in [Chapter 9](#).

### Note

Dynamic programming is a kind of **memoization**. A memoized function stores results of previous calls to the function along with the supplied parameters. If the function is subsequently called with those parameters, it returns the stored result instead of recalculating it.

### Timing (notes)

We can easily test the efficiency gains made by the use of dynamic programming, or any other putative performance enhancement, using the `timeit` module:

```

>>> from timeit import Timer
>>> Timer("PYTHON CODE", "INITIALIZATION CODE").timeit()

```

[MORE]

## 6.5 Visualizing Language Data (DRAFT)

Python has some libraries that are useful for visualizing language data. In this section we will explore two of these, PyLab and NetworkX. The PyLab package supports sophisticated plotting functions with a MATLAB-style interface, and is available from <http://matplotlib.sourceforge.net/>. The NetworkX package is for displaying network diagrams, and is available from <https://networkx.lanl.gov/>.

### PyLab

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 [Figure 3.4](#) we saw a table of numbers showing the frequency of particular modal verbs in the Brown Corpus, classified by genre. The program in [Figure 6.19](#) presents the same information in graphical format. The output is shown in [Figure 6.20](#) (a color figure in the online version).

```

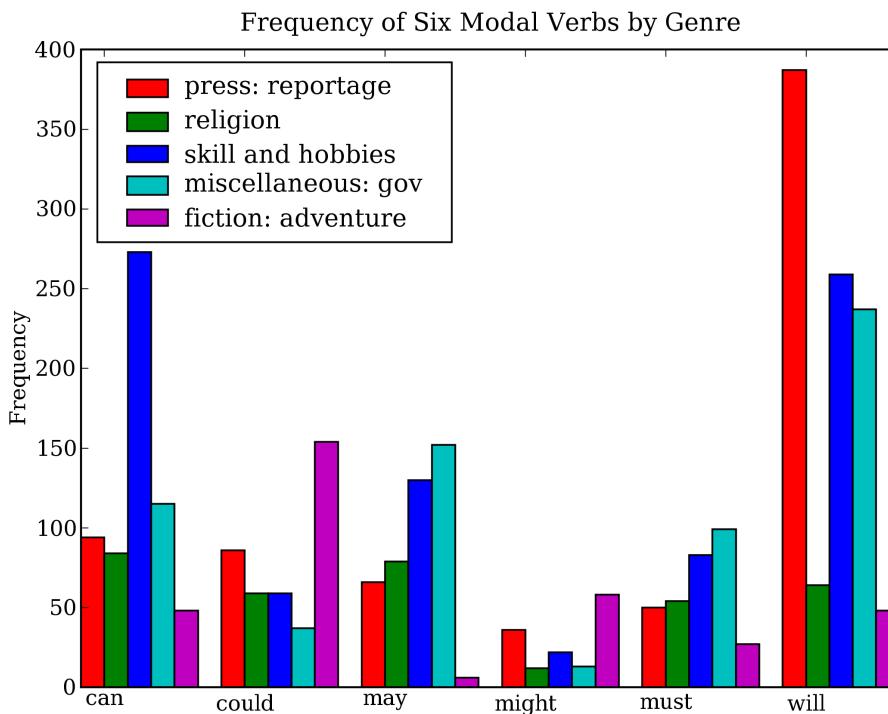
colors = 'rgbcmky' # red, green, blue, cyan, magenta, yellow, black
def bar_chart(categories, words, counts):
 "Plot a bar chart showing counts for each word by category"
 import pylab
 ind = pylab.arange(len(words))
 width = 1.0 / (len(categories) + 1)
 bar_groups = []
 for c in range(len(categories)):
 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((g,w)
... for g in genres
... for w in nltk.corpus.brown.words(categories=g)
... if w in modals)

...
>>> counts = {}
>>> for genre in genres:
... counts[genre] = [cfdist[genre][word] for word in modals]
>>> bar_chart(genres, modals, counts)

```

[Figure 6.19 \(modal\\_plot.py\)](#): Figure 6.19: Frequency of Modals in Different Sections of the Brown Corpus



**Figure 6.20:** Bar Chart Showing Frequency of Modals in Different Sections of Brown Corpus

From the bar chart it is immediately obvious that *may* and *must* have almost identical relative frequencies. The same goes for *could* and *might*.

## Using PyLab on the Web

We can generate data visualizations on the fly, based on user input via the web... To do this we have to specify the `Agg` backend for `matplotlib` before importing `pylab`, as follows:

```
>>> import matplotlib
>>> matplotlib.use('Agg')
>>> import pylab
```

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()`. We specify the filename and dpi, then print HTML markup that directs the web browser to load the file.

```
>>> pylab.savefig('modals.png')
>>> print 'Content-Type: text/html'
>>> print
>>> print '<html><body>'
>>> print ''
>>> print '</body></html>'
```

## Network Diagrams

[Section on networkx and displaying network diagrams; example with WordNet visualization]

## 6.6 Object-Oriented Programming in Python

Object-Oriented Programming is a programming paradigm in which complex structures and processes are decomposed into **classes**, each encapsulating a single data type and the legal operations on that type. In this section we show you how to create simple data classes and processing classes by example. For a systematic introduction to Object-Oriented design, please see the

Further Reading section at the end of this chapter.

## Data Classes: Trees in NLTK

An important data type in language processing is the syntactic tree. Here we will review the parts of the NLTK code that defines the `Tree` class.

The first line of a class definition is the `class` keyword followed by the class name, in this case `Tree`. This class is derived from Python's built-in `list` class, permitting us to use standard list operations to access the children of a tree node.

```
>>> class Tree(list):
```

Next we define the `initializer __init__()`; Python knows to call this function when you ask for a new tree object by writing `t = Tree(node, children)`. The constructor's first argument is special, and is standardly called `self`, giving us a way to refer to the current object from within its definition. This particular constructor calls the list initializer (similar to calling `self = list(children)`), then defines the `node` property of a tree.

```
... def __init__(self, node, children):
... list.__init__(self, children)
... self.node = node
```

Next we define another special function that Python knows to call when we index a Tree. The first case is the simplest, when the index is an integer, e.g. `t[2]`, we just ask for the list item in the obvious way. The other cases are for handling slices, like `t[1:2]`, or `t[:]`.

```
... def __getitem__(self, index):
... if isinstance(index, int):
... return list.__getitem__(self, index)
... else:
... if len(index) == 0:
... return self
... elif len(index) == 1:
... return self[int(index[0])]
... else:
... return self[int(index[0])][index[1:]]
```

This method was for accessing a child node. Similar methods are provided for setting and deleting a child (using `__setitem__`) and `__delitem__`).

Two other special member functions are `__repr__()` and `__str__()`. The `__repr__()` function produces a string representation of the object, one that can be executed to re-create the object, and is accessed from the interpreter simply by typing the name of the object and pressing 'enter'. The `__str__()` function produces a human-readable version of the object; here we call a pretty-printing function we have defined called `pp()`.

```
... def __repr__(self):
... childstr = ' '.join([repr(c) for c in self])
... return '(%s: %s)' % (self.node, childstr)
... def __str__(self):
... return self.pp()
```

Next we define some member functions that do other standard operations on trees. First, for accessing the leaves:

```
... def leaves(self):
... leaves = []
... for child in self:
... if isinstance(child, Tree):
... leaves.extend(child.leaves())
... else:
... leaves.append(child)
... return leaves
```

Next, for computing the height:

```

... def height(self):
... max_child_height = 0
... for child in self:
... if isinstance(child, Tree):
... max_child_height = max(max_child_height, child.height())
... else:
... max_child_height = max(max_child_height, 1)
... return 1 + max_child_height

```

And finally, for enumerating all the subtrees (optionally filtered):

```

... def subtrees(self, filter=None):
... if not filter or filter(self):
... yield self
... for child in self:
... if isinstance(child, Tree):
... for subtree in child.subtrees(filter):
... yield subtree

```

## Processing Classes: N-gram Taggers in NLTK

This section will discuss the `tag.ngram` module.

## 6.7 Further Reading

[\[Harel, 2004\]](#)

[\[Levitin, 2004\]](#)

[\[Knuth, 2006\]](#)

<http://docs.python.org/lib/typesseq-strings.html>

<http://www.jwz.org/doc/worse-is-better.html>

## 6.8 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 underscore; 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. ☀ Does the method for creating a sliding window of n-grams behave correctly for the two limiting cases:  $n = 1$ , and  $n = \text{len(sent)}$ ?
6. ☀ 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?
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 kind of non-Boolean expressions in Boolean contexts, and see whether they evaluate as true or false.

8. ● 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.
1. Now try the same exercise but instead assign `sent2 = sent1[:]`. Modify `sent1` again and see what happens to `sent2`. Explain.
  2. 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.
  3. 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.
9. ● Write code that starts with a string of words and results in a new string consisting of the same words, but where the first word swaps places with the second, and so on. For example, '`the cat sat on the mat`' will be converted into '`cat the on sat mat the`'.
10. ● 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.
11. ● 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.
12. ● Write a function `novel10(text)` that prints any word that appeared in the last 10% of a text that had not been encountered earlier.
13. ● 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.
14. ● Write code that builds a dictionary of dictionaries of sets.
15. ● Use `sorted()` and `set()` to get a sorted list of tags used in the Brown corpus, removing duplicates.
16. ● Write code to convert text into *hAck3r*, where characters are mapped according to the following table:

**Table 6.2**

|         |   |   |   |   |   |        |     |
|---------|---|---|---|---|---|--------|-----|
| Input:  | e | i | o | l | s | .      | ate |
| Output: | 3 | 1 | 0 | 1 | 5 | 5w33t! | 8   |

17. ● 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>).
1. 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}
```

  2. Process a corpus (e.g. `nltk.corpus.state_union`) and for each document, count how many of its words have the number 666.
  3. 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.

18. ● Write a function `shorten(text, n)` to process a text, omitting the `n` most frequently occurring words of the text. How readable is it?
19. ☀ Write code that removes whitespace at the beginning and end of a string, and normalizes whitespace between words to be a single space character.
  1. do this task using `split()` and `join()`
  2. do this task using regular expression substitutions
20. ☀ What happens when the formatting strings `%6s` and `%-6s` are used to display strings that are longer than six characters?
21. ☀ 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.python.org/lib/typeseq-strings.html](http://docs.python.org/lib/typesseq-strings.html)), and use this method to display today's date in two different formats.
22. ● Figure 4.6 in Chapter 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.
23. ☀ Review the answers that you gave for the exercises in 6.1, and rewrite the code as one or more functions.
24. ● In this section we saw examples of some special functions such as `filter()` and `map()`. Other functions in this family are `zip()` and `reduce()`. Find out what these do, and write some code to try them out. What uses might they have in language processing?
25. ● 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.
26. ● 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()`?
27. ● As you saw, `zip()` combines two lists into a single list of pairs. What happens when the lists are of unequal lengths? Define a function `myzip()` that does something different with unequal lists.
28. ● Import the `itemgetter()` function from the `operator` module in Python's standard library (i.e. `from operator import itemgetter`). Create a list `words` containing several words. Now try calling: `sorted(words, key=itemgetter(1))`, and `sorted(words, key=itemgetter(-1))`. Explain what `itemgetter()` is doing.
29. ● 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')`).
30. ● Read about string edit distance and the Levenshtein Algorithm. Try the implementation provided in `nltk.edit_dist()`. How is this using dynamic programming? Does it use the bottom-up or top-down approach? [See also <http://norvig.com/spell-correct.html>]
31. ● The Catalan numbers arise in many applications of combinatorial mathematics, including the counting of parse trees (Chapter 9). The series can be defined as follows:  $C_0 = 1$ , and  $C_{n+1} = \sum_{i=0..n} (C_i C_{n-i})$ .
  1. Write a recursive function to compute  $n$ th Catalan number  $C_n$
  2. Now write another function that does this computation using dynamic programming
  3. Use the `timeit` module to compare the performance of these functions as  $n$  increases.
32. ★ **Authorship identification:** Reproduce some of the results of [\[Zhao & Zobel, 2007\]](#).
33. ★ **Gender-specific lexical choice:** Reproduce some of the results of <http://www.clintoneast.com/articles/words.php>
34. ★ Write a recursive function that pretty prints a trie in alphabetically sorted order, as follows

chat: 'cat' --ien: 'dog' -????: ???

35. ★ 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?

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

## 7 Shallow Linguistic Processing

Two fundamental tasks in language processing are segmentation and labeling. Recall that tokenizations *segments* a sequence of characters into tokens, while tagging *labels* each token. However, segmentation and labeling are generic operations and can be used for a variety of more sophisticated language processing tasks. The goal of this chapter is to answer the following questions:

1. How can we use segmentation and labeling to probe the structure and meaning of text?
2. What are some robust methods for identifying the entities and relationships described in a text?
3. Which corpora are appropriate for this work, and how do we use them for training and evaluating our models?

Along the way, we'll see how we can do segmentation and labeling at the level of phrases.

### 7.1 Information Extraction

Information comes in many shapes and sizes. One important form is **structured data**, where there is a regular and predictable organization of facts. For example, we might be interested in the relation between companies and locations. Given a particular company, we would like to be able to identify the locations in which it does business; conversely, given a location, we would like to discover which companies do business in that location. If our data is in tabular form, such as the little example in [Table 7.1](#), then answering these queries is straightforward.

**Table 7.1:**

Locations data

| OrgName             | LocationName |
|---------------------|--------------|
| Omnicon             | New York     |
| DDB Needham         | New York     |
| Kaplan Thaler Group | New York     |
| BBDO South          | Atlanta      |
| Georgia-Pacific     | Atlanta      |

Assuming the data in `Locations` is a table within a relational database, the question [\(1\)](#) can be translated into the SQL query [\(2\)](#).

(18) Which organizations operate in Atlanta?

(19) `"select OrgName from locations where LocationName = 'Atlanta'"`

When executed, [\(2\)](#) will return the required values:

**Table 7.2:**

Companies that operate in Atlanta

| OrgName |
|---------|
|         |

| OrgName         |
|-----------------|
| BBDO South      |
| Georgia-Pacific |

Things are more tricky if we try to get similar information out of text. For example, consider the following snippet (extracted from the IEER corpus, document 'NYT19980315.0085').

- (20) The fourth Wells account moving to another agency is the packaged paper-products division of Georgia-Pacific Corp., which arrived at Wells only last fall. Like Hertz and the History Channel, it is also leaving for an Omnicom-owned agency, the BBDO South unit of BBDO Worldwide. BBDO South in Atlanta, which handles corporate advertising for Georgia-Pacific, will assume additional duties for brands like Angel Soft toilet tissue and Sparkle paper towels, said Ken Haldin, a spokesman for Georgia-Pacific in Atlanta.

If you read through (3), you will glean the information required to answer (1). But how do we get a machine to understand enough about (3) to return the answers in [Table 7.2](#)? This is obviously a much harder task. Unlike [Table 7.1](#), (3) contains no structure that links organization names with location names. As we will see in [Chapter 11](#), one approach to this problem involves building a very general representation of meaning. Here we take a different approach, and decide in advance that we are only going to look for very specific kinds of information in text, such as the relation between an organization and the locations it operates in. In other words, rather than trying to use text like (3) to answer (1) directly, we first convert the **unstructured data** of natural language sentences into the structured data of [Table 7.1](#). Then we reap the benefits of powerful query tools such as SQL. This method of getting meaning from text is called **Information Extraction**.

Information Extraction has many applications, including business intelligence, resume harvesting, media analysis, sentiment detection, patent search, and email scanning. A particularly important area of current research involves the attempt to extract structured data out of electronically-available scientific literature, especially in the domain of biology and medicine.

## Named Entity Recognition

Information Extraction is usually broken down into at least two major steps: **Named Entity Recognition** and **Relation Extraction**. Named Entities are noun phrases that denote specific types of individuals such as organizations, persons, dates, and so on. Thus, we might use the following XML annotations to mark-up the named entities in (21a):

- (21)
- a. ... said William Gale, an economist at the Brookings Institution, the research group in Washington.
  - b. ... said <ne type='PERSON'>William Gale</ne>, an economist at the <ne type='ORGANIZATION'>Brookings Institution</ne>, the research group in <ne type='LOCATION'>Washington</ne>.

How do we go about identifying named entities? One option would be to look up each word in an appropriate list of names. For example, in the case of locations, we could use the Alexandria Gazetteer or the Getty Gazetteer. However, doing this blindly runs into problems, as shown in [Figure 7.1](#).



**Figure 7.1:** Location Detection by Simple Lookup for a News Story

Observe that the gazetteer has good coverage of locations in many countries, and incorrectly finds locations like Sanchez in the

Dominican Republic. Of course we could omit such locations from the gazetteer, but then we won't be able to identify them when they do appear in a document.

It gets even harder in the case of names for people or organizations. Any list of such names will probably have poor coverage. New organizations come into existence every day, so if we are trying to deal with contemporary newswire or blog entries, it is unlikely that we will be able to recognize many of the entities using gazetteer lookup.

Another major source of difficulty is caused by the fact that many named entity terms are ambiguous. Thus *May* and *North* are likely to be parts of named entities for DATE and LOCATION, respectively, but could both be part of a PERSON; conversely *Christian Dior* looks like a PERSON but is more likely to be of type ORGANIZATION. A term like *Yankee* will be ordinary modifier in some contexts, but will be marked as an entity of type ORGANIZATION in the phrase *Yankee infielders*.

Another challenge is posed by multi-part names like *Stanford University*, and by names that contain other names such as *Cecil H. Green Library* and *Escondido Village Conference Service Center*. In Named Entity Recognition, therefore, we need to be able to identify the beginning and end of multi-token sequences.

To summarize, we cannot reliably detect named entities by looking them up in a gazetteer, and it is also hard to develop rules that will correctly recognize ambiguous named entities on the basis of their context of occurrence. Although lookup may contribute to a solution, most contemporary approaches to Named Entity Recognition treat it as a statistical classification task that requires training data for good performance. This task is facilitated by adopting an appropriate data representation, such as the IOB tags that we saw being deployed in the CoNLL chunk data. For example, here are a representative few lines from the CONLL 2002 (conll2002) Dutch training data:

```
Eddy N B-PER
Bonte N I-PER
is V O
woordvoerder N O
van Prep O
diezelfde Pron O
Hogeschool N B-ORG
. Punc O
```

In this representation, there is one token per line, each with its part-of-speech tag and its named entity tag.

Identifying the boundaries of specific types of word sequences is also required when we want to recognize pieces of syntactic structure. Suppose for example that as a preliminary to Named Entity Recognition, we have decided that it would be useful to just pick out noun phrases from a piece of text. To carry this out in a complete way, we would probably want to use a proper syntactic parser. But parsing can be quite challenging and computationally expensive — is there an easier alternative? The answer is Yes: we can look for sequences of part-of-speech tags in a tagged text, using one or more patterns that capture the typical ingredients of a noun phrase.

For example, here is some Wall Street Journal text with noun phrases marked using brackets:

- (22) [ The/DT market/NN ] for/IN [ system-management/NN software/NN ] for/IN [ Digital/NNP ] [ 's/POS hardware/NN ] is/VBZ fragmented/JJ enough/RB that/IN [ a/DT giant/NN ] such/JJ as/IN [ Computer/NNP Associates/NNPS ] should/MD do/VB well/RB there/RB ./.

From the point of view of theoretical linguistics, we seem to have been rather unorthodox in our use of the term "noun phrase"; although all the bracketed strings are noun phrases, not every noun phrase has been captured. We will discuss this issue in more detail shortly. For the moment, let's say that we are identifying noun "chunks" rather than full noun phrases.

## Relation Extraction

When named entities have been identified in a text, we then want to extract relations that hold between them. As indicated earlier, we will typically be looking for relations between specified types of named entity. One way of approaching this task is to initially look for all triples of the form  $X, \alpha, Y$ , where  $X$  and  $Y$  are named entities of the required types, and  $\alpha$  is the string of words that intervenes between  $X$  and  $Y$ . We can then use regular expressions to pull out just those instances of  $\alpha$  that express the relation that we are looking for. The following example searches for strings that contain the word *in*. The special character expression `(?!\\b.+ing\\b)` is a negative lookahead condition that allows us to disregard strings such as *success in supervising the transition of*, where *in* is followed by a gerundive verb.

```
>>> IN = re.compile(r'.*\bin\b(?!\\b.+ing\\b) ')
```

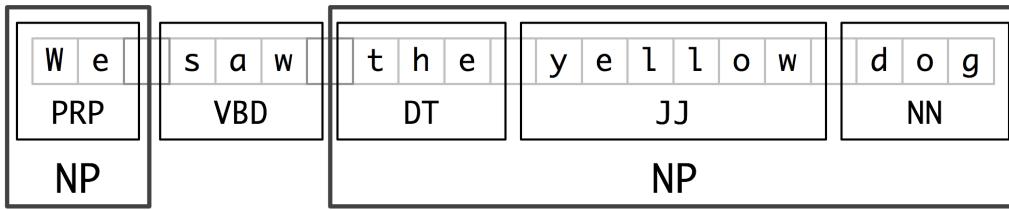
```
>>> for doc in nltk.corpus.ieer.parsed_docs('NYT_19980315'):
... for rel in nltk.sem.extract_rels('ORG', 'LOC', doc, pattern = IN):
... print nltk.sem.show_raw_rtuple(rel)
[ORG: 'WHYY'] 'in' [LOC: 'Philadelphia']
[ORG: 'McGlashan & Sarrail'] 'firm in' [LOC: 'San Mateo']
[ORG: 'Freedom Forum'] 'in' [LOC: 'Arlington']
[ORG: 'Brookings Institution'] ', the research group in' [LOC: 'Washington']
[ORG: 'Idealab'] ', a self-described business incubator based in' [LOC: 'Los Angeles']
[ORG: 'Open Text'] ', based in' [LOC: 'Waterloo']
[ORG: 'WGBH'] 'in' [LOC: 'Boston']
[ORG: 'Bastille Opera'] 'in' [LOC: 'Paris']
[ORG: 'Omnicom'] 'in' [LOC: 'New York']
[ORG: 'DDB Needham'] 'in' [LOC: 'New York']
[ORG: 'Kaplan Thaler Group'] 'in' [LOC: 'New York']
[ORG: 'BBDO South'] 'in' [LOC: 'Atlanta']
[ORG: 'Georgia-Pacific'] 'in' [LOC: 'Atlanta']
```

Searching for the keyword *works in* reasonably well, though it will also retrieve false positives such as [ORG: House Transportation Committee], secured the most money *in* the [LOC: New York]; there is unlikely to be simple string-based method of excluding filler strings such as this.

```
>>> vnv = """
... (
... is/V|
... was/V|
... werd/V|
... wordt/V
...)
... .*
... van/Prep
...
"""
>>> VAN = re.compile(vnv, re.VERBOSE)
>>> for r in nltk.sem.extract_rels('PER', 'ORG', corpus='conll2002-ned', pattern=VAN):
... print show_tuple(r)
```

## 7.2 Chunking

In chunking, we carry out segmentation and labeling of multi-token sequences, as illustrated in [Figure 7.2](#). The smaller boxes show word-level segmentation and labeling, while the large boxes show higher-level segmentation and labeling. It is these larger pieces that we will call **chunks**, and the process of identifying them is called **chunking**.



**Figure 7.2:** Segmentation and Labeling at both the Token and Chunk Levels

Like tokenization, chunking can skip over material in the input. Tokenization omits white space and punctuation characters. Chunking uses only a subset of the tokens and leaves others out.

In the following sections we will explore chunking in some depth, beginning with the definition and representation of chunks. We will see regular expression and n-gram approaches to chunking, and will develop and evaluate chunkers using the CoNLL-2000 chunking corpus.

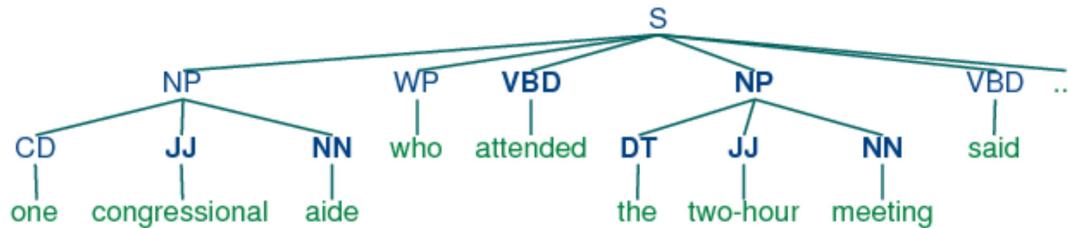
### Chunking vs Parsing

Chunking is akin to parsing in the sense that it can be used to build hierarchical structure over text. There are several important differences, however. First, as noted above, chunking is not exhaustive, and typically ignores some items in the surface string. In fact, chunking is sometimes called **partial parsing**. Second, where parsing constructs nested structures that are arbitrarily deep, chunking creates structures of fixed depth (typically depth 2). These chunks often correspond to the lowest level of grouping

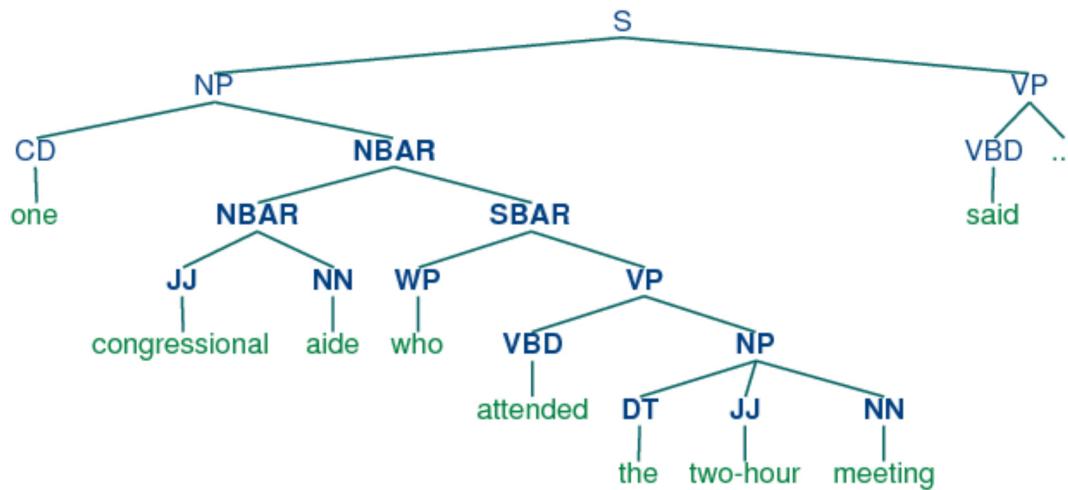
identified in the full parse tree. This is illustrated in (6b) below, which shows an NP chunk structure and a completely parsed counterpart:

(23)

a.



b.



A significant motivation for chunking is its robustness and efficiency relative to parsing. As we will see in [Chapter 8](#), parsing has problems with robustness, given the difficulty in gaining broad coverage while minimizing ambiguity. Parsing is also relatively inefficient: the time taken to parse a sentence grows with the cube of the length of the sentence, while the time taken to chunk a sentence only grows linearly.

## Representing Chunks: Tags vs Trees

As befits its intermediate status between tagging and parsing, chunk structures can be represented using either tags or trees. The most widespread file representation uses so-called **IOB tags**. In this scheme, each token is tagged with one of three special chunk tags, **I** (inside), **O** (outside), or **B** (begin). A token is tagged as **B** if it marks the beginning of a chunk. Subsequent tokens within the chunk are tagged **I**. All other tokens are tagged **O**. The **B** and **I** tags are suffixed with the chunk type, e.g. **B-NP**, **I-NP**. Of course, it is not necessary to specify a chunk type for tokens that appear outside a chunk, so these are just labeled **O**. An example of this scheme is shown in [Figure 7.3](#).

|             |          |            |            |            |
|-------------|----------|------------|------------|------------|
| We          | saw      | the        | yellow     | dog        |
| PRP<br>B-NP | VBD<br>O | DT<br>B-NP | JJ<br>I-NP | NN<br>I-NP |

**Figure 7.3:** Tag Representation of Chunk Structures

IOB tags have become the standard way to represent chunk structures in files, and we will also be using this format. Here is an example of the file representation of the information in [Figure 7.3](#):

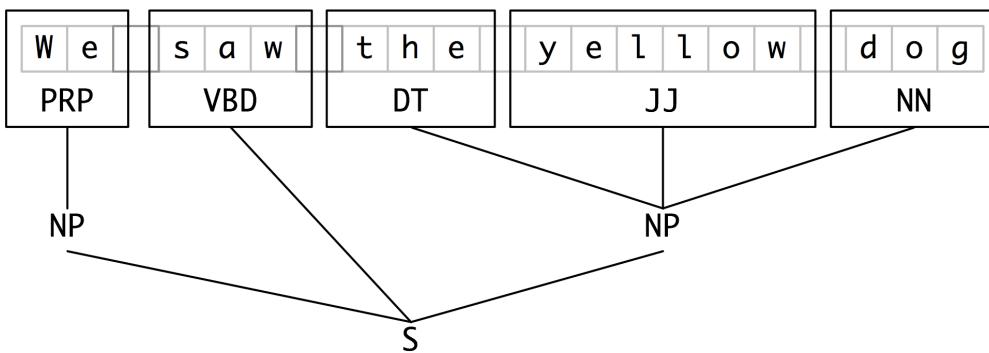
```

We PRP B-NP
saw VBD O
the DT B-NP
little JJ I-NP
yellow JJ I-NP
dog NN I-NP

```

In this representation, there is one token per line, each with its part-of-speech tag and its chunk tag. We will see later that this format permits us to represent more than one chunk type, so long as the chunks do not overlap.

As we saw earlier, chunk structures can also be represented using trees. These have the benefit that each chunk is a constituent that can be manipulated directly. An example is shown in [Figure 7.4](#).



**Figure 7.4:** Tree Representation of Chunk Structures

NLTK uses trees for its internal representation of chunks, and provides methods for reading and writing such trees to the IOB format. By now you should understand what chunks are, and how they are represented. In the next section, you will see how to build a simple chunker.

## Chunkers

A **chunker** finds contiguous, non-overlapping spans of related tokens and groups them together into chunks. Chunkers often operate on tagged texts, and use the tags to make chunking decisions. In this section we will see how to write a special type of regular expression over part-of-speech tags, and then how to combine these into a chunk grammar. Then we will set up a chunker to chunk some tagged text according to the grammar.

Chunking in NLTK begins with tagged tokens.

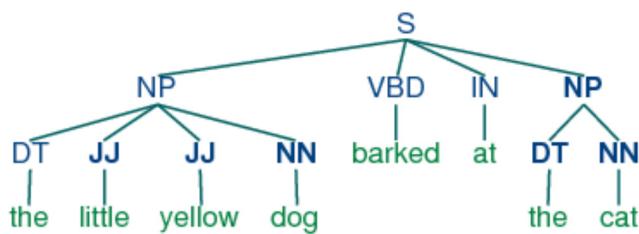
```
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
```

Next, we write regular expressions over tag sequences. The following example identifies noun phrases that consist of an optional determiner, followed by any number of adjectives, then a noun.

```
>>> cp = nltk.RegexpParser("NP: { <DT>? <JJ>* <NN> }")
```

We create a chunker `cp` that can then be used repeatedly to parse tagged input. The result of chunking is a tree.

```
>>> cp.parse(tagged_tokens).draw()
```



### Note

Remember that our program samples assume you begin your interactive session or your program with: `import nltk, re, pprint`

## Tag Patterns

A **tag pattern** is a sequence of part-of-speech tags delimited using angle brackets, e.g. <DT><JJ><NN>. Tag patterns are the same as the regular expression patterns we have already seen, except for two differences that make them easier to use for chunking. First, angle brackets group their contents into atomic units, so "<NN>+" matches one or more repetitions of the tag NN; and "<NN|JJ>" matches the NN or JJ. Second, the period wildcard operator is constrained not to cross tag delimiters, so that "<N.\*>" matches any single tag starting with N, e.g. NN, NNS.

Now, consider the following noun phrases from the Wall Street Journal:

```
another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
```

We can match these using a slight refinement of the first tag pattern above: <DT>?<JJ.\*>\*<NN.\*>+. This can be used to chunk any sequence of tokens beginning with an optional determiner DT, followed by zero or more adjectives of any type JJ.\* (including relative adjectives like earlier/JJR), followed by one or more nouns of any type NN.\*. It is easy to find many more difficult examples:

```
his/PRP$ Mansion/NNP House/NNP speech/NN
the/DT price/NN cutting/VBG
3/CD %/NN to/TO 4/CD %/NN
more/JJR than/IN 10/CD %/NN
the/DT fastest/JJS developing/VBG trends/NNS
's/POS skill/NN
```

Your challenge will be to come up with tag patterns to cover these and other examples. A good way to learn about tag patterns is via a graphical interface `nltk.draw.rechunkparser.demo()`.

## Chunking with Regular Expressions

The chunker begins with a flat structure in which no tokens are chunked. Patterns are applied in turn, successively updating the chunk structure. Once all of the patterns have been applied, the resulting chunk structure is returned. [Figure 7.5](#) shows a simple chunk grammar consisting of two patterns. The first pattern matches an optional determiner or possessive pronoun (recall that | indicates disjunction), zero or more adjectives, then a noun. The second rule matches one or more proper nouns. We also define some tagged tokens to be chunked, and run the chunker on this input.

```
grammar = r"""
NP: {<DT|PP\$>?<JJ>*<NN>} # chunk determiner/possessive, adjectives and nouns
 {<NNP>+} # chunk sequences of proper nouns
"""

cp = nltk.RegexpParser(grammar)
tagged_tokens = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"),
 ("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]

>>> print cp.parse(tagged_tokens)
(S
 (NP Rapunzel/NNP)
 let/VBD
 down/RP
 (NP her/PP$ long/JJ golden/JJ hair/NN))
```

[Figure 7.5 \(chunker1.py\)](#): Figure 7.5: Simple Noun Phrase Chunker

### Note

The \$ symbol is a special character in regular expressions, and therefore needs to be escaped with the backslash \ in order to match the tag PP\$.

If a tag pattern matches at overlapping locations, the first match takes precedence. For example, if we apply a rule that matches two consecutive nouns to a text containing three consecutive nouns, then only the first two nouns will be chunked:

```
>>> nouns = [("money", "NN"), ("market", "NN"), ("fund", "NN")]
>>> grammar = "NP: {<NN><NN>} # Chunk two consecutive nouns"
>>> cp = nltk.RegexpParser(grammar)
>>> print cp.parse(nouns)
(S (NP money/NN market/NN) fund/NN)
```

Once we have created the chunk for *money market*, we have removed the context that would have permitted *fund* to be included in a chunk. This issue would have been avoided with a more permissive chunk rule, e.g. `NP: {<NN>+}`.

## Splitting and Merging (incomplete)

[Notes: the above approach creates chunks that are too large, e.g. *the cat the dog chased* would be given a single NP chunk because it does not detect that determiners introduce new chunks. For this we would need a rule to split an NP chunk prior to any determiner, using a pattern like: `"NP: <.*>{<DT>}`. We can also merge chunks, e.g. `"NP: <NN>{ }<NN>"`.]

## Chinking

Sometimes it is easier to define what we *don't* want to include in a chunk than it is to define what we *do* want to include. In these cases, it may be easier to build a chunker using a method called **chinking**.

Following [[Church, Young, & Bloethooft, 1996](#)], we define a **chink** as a sequence of tokens that is not included in a chunk. In the following example, *barked/VBD at/IN* is a chink:

```
[the/DT little/JJ yellow/JJ dog/NN] barked/VBD at/IN [the/DT cat/NN]
```

Chinking is the process of removing a sequence of tokens from a chunk. If the sequence of tokens spans an entire chunk, then the whole chunk is removed; if the sequence of tokens appears in the middle of the chunk, these tokens are removed, leaving two chunks where there was only one before. If the sequence is at the beginning or end of the chunk, these tokens are removed, and a smaller chunk remains. These three possibilities are illustrated in [Table 7.3](#).

**Table 7.3:**

Three chinking rules applied to the same chunk

|                  |                         |                           |                         |
|------------------|-------------------------|---------------------------|-------------------------|
| <i>Input</i>     | [a/DT little/JJ dog/NN] | [a/DT little/JJ dog/NN]   | [a/DT little/JJ dog/NN] |
| <i>Operation</i> | Chink "DT JJ NN"        | Chink "JJ"                | Chink "NN"              |
| <i>Pattern</i>   | "}DT JJ NN{"            | "}JJ{"                    | "}NN{"                  |
| <i>Output</i>    | a/DT little/JJ dog/NN   | [a/DT] little/JJ [dog/NN] | [a/DT little/JJ] dog/NN |

In the following grammar, we put the entire sentence into a single chunk, then excise the chink:

```
grammar = r"""
NP:
{<.*>+} # Chunk everything
}<VBD|IN>+{ # Chink sequences of VBD and IN
"""

tagged_tokens = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
 ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
cp = nltk.RegexpParser(grammar)

>>> print cp.parse(tagged_tokens)
(S
 (NP the/DT little/JJ yellow/JJ dog/NN)
 barked/VBD
 at/IN
 (NP the/DT cat/NN))
>>> from nltk.corpus import conll2000
>>> test_sents = conll2000.chunked_sents('test.txt', chunk_types=('NP',))
>>> print nltk.chunk.accuracy(cp, test_sents)
0.581041433607
```

[Figure 7.6 \(chinker.py\)](#): Figure 7.6: Simple Chinker

A chunk grammar can use any number of chunking and chinking patterns in any order.

## Multiple Chunk Types (incomplete)

So far we have only developed NP chunkers. However, as we saw earlier in the chapter, the CoNLL chunking data is also annotated for PP and VP chunks. Here is an example, to show the structure we get from the corpus and the flattened version that will be used as input to the parser.

```
>>> from nltk.corpus import conll2000
>>> example = conll2000.chunked_sents('train.txt')[99]
>>> print example
(S
 (PP Over/IN)
 (NP a/DT cup/NN)
 (PP of/IN)
 (NP coffee/NN)
 ,
 (NP Mr./NNP Stone/NNP)
 (VP told/VBD)
 (NP his/PRP$ story/NN)
 .)
>>> print example.flatten()
(S
 Over/IN
 a/DT
 cup/NN
 of/IN
 coffee/NN
 ,
 Mr./NNP
 Stone/NNP
 told/VBD
 his/PRP$
 story/NN
 .)
```

Now we can set up a multi-stage chunk grammar, as shown in [Figure 7.7](#). It has a stage for each of the chunk types.

```
cp = nltk.RegexpParser(r"""
 NP: {<DT>?<JJ>*<NN.*>+} # noun phrase chunks
 VP: {<TO>?<VB.*>} # verb phrase chunks
 PP: {<IN>} # prepositional phrase chunks
 """)

>>> from nltk.corpus import conll2000
>>> example = conll2000.chunked_sents('train.txt')[99]
>>> print cp.parse(example.flatten(), trace=1)
Input:
<IN> <DT> <NN> <IN> <NN> <,> <NNP> <NNP> <VBD> <PRP$> <NN> <.>
noun phrase chunks:
<IN> <DT> <NN> } <IN> <NN> } <,> { <NNP> <NNP> } <VBD> <PRP$> { <NN> } <.>
Input:
<IN> <NP> <IN> <NP> <,> <NP> <VBD> <PRP$> <NP> <.>
verb phrase chunks:
<IN> <NP> <IN> <NP> <,> <NP> { <VBD> } <PRP$> <NP> <.>
Input:
<IN> <NP> <IN> <NP> <,> <NP> <VP> <PRP$> <NP> <.>
prepositional phrase chunks:
{<IN>} <NP> {<IN>} <NP> <,> <NP> <VP> <PRP$> <NP> <.>
(S
 (PP Over/IN)
 (NP a/DT cup/NN)
 (PP of/IN)
 (NP coffee/NN)
 ,
 (NP Mr./NNP Stone/NNP)
 (VP told/VBD)
 his/PRP$
 (NP story/NN)
 .)
```

[Figure 7.7 \(multistage\\_chunker.py\)](#): Figure 7.7: A Multistage Chunker

## 7.3 Developing and Evaluating Chunkers

Now you have a taste of what chunking can do, but we have not explained how to carry out a quantitative evaluation of chunkers. For this, we need to get access to a corpus that has been annotated not only with parts-of-speech, but also with chunk information. We will begin by looking at the mechanics of converting IOB format into an NLTK tree, then at how this is done on a larger scale using a chunked corpus directly. We will see how to use the corpus to score the accuracy of a chunker, then look some more flexible ways to manipulate chunks. Our focus throughout will be on scaling up the coverage of a chunker.

### Developing Chunkers

Creating a good chunker usually requires several rounds of development and testing, during which existing rules are refined and new rules are added. In order to diagnose any problems, it often helps to trace the execution of a chunker, using its `trace` argument. The tracing output shows the rules that are applied, and uses braces to show the chunks that are created at each stage of processing. In [Figure 7.8](#), two chunk patterns are applied to the input sentence. The first rule finds all sequences of three tokens whose tags are `DT`, `JJ`, and `NN`, and the second rule finds any sequence of tokens whose tags are either `DT` or `NN`. We set up two chunkers, one for each rule ordering, and test them on the same input.

```
tagged_tokens = [("The", "DT"), ("enchantress", "NN"), ("clutched", "VBD"),
 ("the", "DT"), ("beautiful", "JJ"), ("hair", "NN")]
cp1 = nltk.RegexpParser(r"""
 NP: {<DT><JJ><NN>} # Chunk det+adj+noun
 {<DT|NN>+} # Chunk sequences of NN and DT
 """)
cp2 = nltk.RegexpParser(r"""
 NP: {<DT|NN>+} # Chunk sequences of NN and DT
 {<DT><JJ><NN>} # Chunk det+adj+noun
 """)

>>> print cp1.parse(tagged_tokens, trace=1)
Input:
<DT> <NN> <VBD> <DT> <JJ> <NN>
Chunk det+adj+noun:
<DT> <NN> <VBD> {<DT> <JJ> <NN>}
Chunk sequences of NN and DT:
{<DT> <NN>} <VBD> {<DT> <JJ> <NN>}
(S
 (NP The/DT enchantress/NN)
 clutched/VBD
 (NP the/DT beautiful/JJ hair/NN))
>>> print cp2.parse(tagged_tokens, trace=1)
Input:
<DT> <NN> <VBD> <DT> <JJ> <NN>
Chunk sequences of NN and DT:
{<DT> <NN>} <VBD> {<DT>} <JJ> {<NN>}
Chunk det+adj+noun:
{<DT> <NN>} <VBD> {<DT>} <JJ> {<NN>}
(S
 (NP The/DT enchantress/NN)
 clutched/VBD
 (NP the/DT)
 beautiful/JJ
 (NP hair/NN))
```

[Figure 7.8 \(chunker2.py\)](#): [Figure 7.8](#): Two Noun Phrase Chunkers Having Identical Rules in Different Orders

Observe that when we chunk material that is already partially chunked, the chunker will only create chunks that do not partially overlap existing chunks. In the case of `cp2`, the second rule did not find any chunks, since all chunks that matched its tag pattern overlapped with existing chunks. As you can see, you need to be careful to put chunk rules in the right order.

You may have noted that we have added explanatory comments, preceded by `#`, to each of our tag rules. Although it is not strictly necessary to do this, it's a helpful reminder of what a rule is meant to do, and it is used as a header line for the output of a rule application when tracing is on.

You might want to test out some of your rules on a corpus. One option is to use the Brown corpus. However, you need to remember that the Brown tagset is different from the Penn Treebank tagset that we have been using for our examples so far in this chapter (see [Appendix A](#) for full details). Because the Brown tagset uses `NP` for proper nouns, in this example we have

followed Abney in labeling noun chunks as NX.

```
>>> grammar = (r"""
... NX: {<AT|AP|PP\$>?<JJ.*>?<NN.*>} # Chunk article/numeral/possessive+adj+noun
... {<NP>+} # Chunk one or more proper nouns
... """
)
>>> cp = nltk.RegexpParser(grammar)
>>> sent = nltk.corpus.brown.tagged_sents(categories='news')[112]
>>> print cp.parse(sent)
(S
 (NX His/PP$ contention/NN)
 was/BEDZ
 denied/VBN
 by/IN
 (NX several/AP bankers/NNS)
 ,/
 including/IN
 (NX Scott/NP Hudson/NP)
 of/IN
 (NX Sherman/NP)
 ,/
 (NX Gaynor/NP B./NP Jones/NP)
 of/IN
 (NX Houston/NP)
 ,/
 (NX J./NP B./NP Brady/NP)
 of/IN
 (NX Harlingen/NP)
 and/CC
 (NX Howard/NP Cox/NP)
 of/IN
 (NX Austin/NP)
 ./)
```

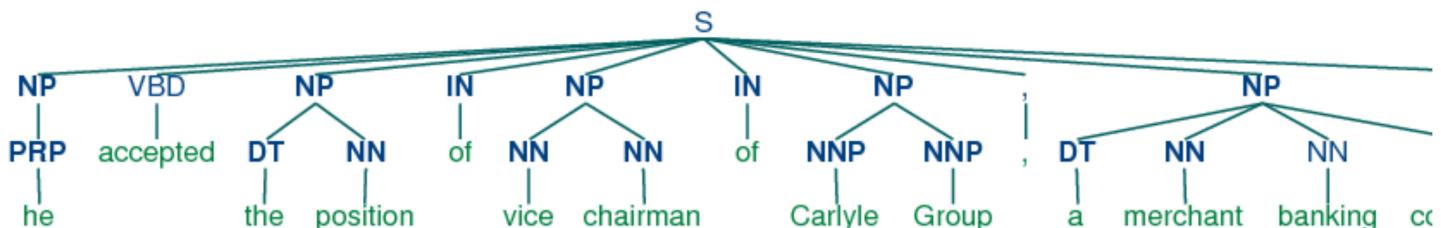
## Reading IOB Format and the CoNLL 2000 Corpus

Using the `corpora` module we can load Wall Street Journal text that has been tagged then chunked using the IOB notation. The chunk categories provided in this corpus are NP, VP and PP. As we have seen, each sentence is represented using multiple lines, as shown below:

```
he PRP B-NP
accepted VBD B-VP
the DT B-NP
position NN I-NP
...
...
```

A conversion function `chunk.conllstr2tree()` builds a tree representation from one of these multi-line strings. Moreover, it permits us to choose any subset of the three chunk types to use. The example below produces only NP chunks:

```
>>> text = """
... he PRP B-NP
... accepted VBD B-VP
... the DT B-NP
... position NN I-NP
... of IN B-PP
... vice NN B-NP
... chairman NN I-NP
... of IN B-PP
... Carlyle NNP B-NP
... Group NNP I-NP
... , O
... a DT B-NP
... merchant NN I-NP
... banking NN I-NP
... concern NN I-NP
... . O
...
>>> nltk.chunk.conllstr2tree(text, chunk_types=('NP',)).draw()
```



We can use the NLTK corpus module to access a larger amount of chunked text. The CoNLL 2000 corpus contains 270k words of Wall Street Journal text, divided into "train" and "test" portions, annotated with part-of-speech tags and chunk tags in the IOB format. We can access the data using an NLTK corpus reader called `conll2000`. Here is an example that reads the 100th sentence of the "train" portion of the corpus:

```

>>> print nltk.corpus.conll2000.chunked_sents('train.txt')[99]
(S
 (PP Over/IN)
 (NP a/DT cup/NN)
 (PP of/IN)
 (NP coffee/NN)
 ,
 (NP Mr./NNP Stone/NNP)
 (VP told/VBD)
 (NP his/PRP$ story/NN)
 .)

```

This showed three chunk types, for NP, VP and PP. We can also select which chunk types to read:

```

>>> print nltk.corpus.conll2000.chunked_sents('train.txt', chunk_types=('NP',))[99]
(S
 Over/IN
 (NP a/DT cup/NN)
 of/IN
 (NP coffee/NN)
 ,
 (NP Mr./NNP Stone/NNP)
 told/VBD
 (NP his/PRP$ story/NN)
 .)

```

## Simple Evaluation and Baselines

Armed with a corpus, it is now possible to carry out some simple evaluation. We start off by establishing a baseline for the trivial chunk parser `cp` that creates no chunks:

```

>>> cp = nltk.RegexpParser("")
>>> print nltk.chunk.accuracy(cp, conll2000.chunked_sents('train.txt', chunk_types=('NP',)))
0.440845995079

```

This indicates that more than a third of the words are tagged with `o` (i.e., not in an NP chunk). Now let's try a naive regular expression chunker that looks for tags (e.g., CD, DT, JJ, etc.) beginning with letters that are typical of noun phrase tags:

```

>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> print nltk.chunk.accuracy(cp, conll2000.chunked_sents('train.txt', chunk_types=('NP',)))
0.874479872666

```

As you can see, this approach achieves pretty good results. In order to develop a more data-driven approach, let's define a function `chunked_tags()` that takes some chunked data and sets up a conditional frequency distribution. For each tag, it counts up the number of times the tag occurs inside an NP chunk (the `True` case, where `ctag` is `B-NP` or `I-NP`), or outside a chunk (the `False` case, where `ctag` is `O`). It returns a list of those tags that occur inside chunks more often than outside chunks.

```

def chunked_tags(train):
 """Generate a list of tags that tend to appear inside chunks"""
 cfdist = nltk.ConditionalFreqDist()

```

```

for t in train:
 for word, tag, chtag in nltk.chunk.tree2conlltags(t):
 if chtag == "O":
 cfdist[tag].inc(False)
 else:
 cfdist[tag].inc(True)
return [tag for tag in cfdist.conditions() if cfdist[tag].max() == True]

>>> train_sents = conll2000.chunked_sents('train.txt', chunk_types=('NP',))
>>> print chunked_tags(train_sents)
['PRP$', 'WDT', 'JJ', 'WP', 'DT', '#', '$', 'NN', 'FW', 'POS',
 'PRP', 'NNS', 'NNP', 'PDT', 'RBS', 'EX', 'WP$', 'CD', 'NNPS', 'JJS', 'JJR']

```

**Figure 7.9 (chunker3.py):** Figure 7.9: Capturing the conditional frequency of NP Chunk Tags

The next step is to convert this list of tags into a tag pattern. To do this we need to "escape" all non-word characters, by preceding them with a backslash. Then we need to join them into a disjunction. This process would convert a tag list `['NN', 'NN\$']` into the tag pattern `<NN|NN\$>`. The following function does this work, and returns a regular expression chunker:

```

def baseline_chunker(train):
 chunk_tags = [re.sub(r'(\W)', r'\\1', tag)
 for tag in chunked_tags(train)]
 grammar = 'NP: {<%s>+}' % '|'.join(chunk_tags)
 return nltk.RegexpParser(grammar)

```

**Figure 7.10 (chunker4.py):** Figure 7.10: Deriving a Regexp Chunker from Training Data

The final step is to train this chunker and test its accuracy (this time on the "test" portion of the corpus, i.e., data not seen during training):

```

>>> train_sents = conll2000.chunked_sents('train.txt', chunk_types=('NP',))
>>> test_sents = conll2000.chunked_sents('test.txt', chunk_types=('NP',))
>>> cp = baseline_chunker(train_sents)
>>> print nltk.chunk.accuracy(cp, test_sents)
0.914262194736

```

## Evaluating Chunk Parsers

An easy way to evaluate a chunk parser is to take some already chunked text, strip off the chunks, rechunk it, and compare the result with the original chunked text. The `ChunkScore.score()` function takes the correctly chunked sentence as its first argument, and the newly chunked version as its second argument, and compares them. It reports the fraction of actual chunks that were found (recall), the fraction of hypothesized chunks that were correct (precision), and a combined score, the F-measure (the harmonic mean of precision and recall).

A number of different metrics can be used to evaluate chunk parsers. We will concentrate on a class of metrics that can be derived from two sets:

- **guessed**: The set of chunks returned by the chunk parser.
- **correct**: The correct set of chunks, as defined in the test corpus.

We will set up an analogy between the correct set of chunks and a user's so-called "information need", and between the set of returned chunks and a system's returned documents (cf precision and recall, from [Chapter 5](#)).

During evaluation of a chunk parser, it is useful to flatten a chunk structure into a tree consisting only of a root node and leaves:

```

>>> correct = nltk.chunk.tagstr2tree(
... "[the/DT little/JJ cat/NN] sat/VBD on/IN [the/DT mat/NN]")
>>> print correct.flatten()
(S the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN)

```

We run a chunker over this flattened data, and compare the resulting chunked sentences with the originals, as follows:

```

>>> grammar = r"NP: {<PRP|DT|POS|JJ|CD|N.*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("cat", "NN"),
... ("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

```

```
>>> chunkscore = nltk.chunk.ChunkScore()
>>> guess = cp.parse(correct.flatten())
>>> chunkscore.score(correct, guess)
>>> print chunkscore
ChunkParse score:
 Precision: 100.0%
 Recall: 100.0%
 F-Measure: 100.0%
```

`ChunkScore` is a class for scoring chunk parsers. It can be used to evaluate the output of a chunk parser, using precision, recall, f-measure, missed chunks, and incorrect chunks. It can also be used to combine the scores from the parsing of multiple texts. This is quite useful if we are parsing a text one sentence at a time. The following program listing shows a typical use of the `ChunkScore` class. In this example, `chunkparser` is being tested on each sentence from the Wall Street Journal tagged files.

```
>>> grammar = r"NP: {<DT|JJ|NN>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> chunkscore = nltk.chunk.ChunkScore()
>>> for file in nltk.corpus.treebank_chunk.files()[:5]:
... for chunk_struct in nltk.corpus.treebank_chunk.chunked_sents(file):
... test_sent = cp.parse(chunk_struct.flatten())
... chunkscore.score(chunk_struct, test_sent)
>>> print chunkscore
ChunkParse score:
 Precision: 42.3%
 Recall: 29.9%
 F-Measure: 35.0%
```

The overall results of the evaluation can be viewed by printing the `ChunkScore`. Each evaluation metric is also returned by an accessor method: `precision()`, `recall`, `f_measure`, `missed`, and `incorrect`. The `missed` and `incorrect` methods can be especially useful when trying to improve the performance of a chunk parser. Here are the missed chunks:

```
>>> from random import shuffle
>>> missed = chunkscore.missed()
>>> shuffle(missed)
>>> print missed[:10]
[((('A', 'DT'), ('Lorillard', 'NNP'), ('spokeswoman', 'NN'))),
 (('even', 'RB'), ('brief', 'JJ'), ('exposures', 'NNS'))),
 (((its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS'))),
 (('30', 'CD'), ('years', 'NNS'))),
 (('workers', 'NNS')),,
 (('preliminary', 'JJ'), ('findings', 'NNS'))),
 (('Medicine', 'NNP')),,
 (('Consolidated', 'NNP'), ('Gold', 'NNP'), ('Fields', 'NNP'), ('PLC', 'NNP'))),
 (((its', 'PRP$'), ('Micronite', 'NN'), ('cigarette', 'NN'), ('filters', 'NNS'))),
 (('researchers', 'NNS'))]
```

Here are the incorrect chunks:

```
>>> incorrect = chunkscore.incorrect()
>>> shuffle(incorrect)
>> print incorrect[:10]
[((('New', 'JJ'), ('York-based', 'JJ'))),
 (('Micronite', 'NN'), ('cigarette', 'NN'))),
 (((a', 'DT'), ('forum', 'NN'), ('likely', 'JJ'))),
 (((later', 'JJ'))),
 (('preliminary', 'JJ')),,
 (((New', 'JJ'), ('York-based', 'JJ'))),
 (((resilient', 'JJ'))),
 (((group', 'NN'))),
 (((the', 'DT'))),
 (((Micronite', 'NN'), ('cigarette', 'NN')))]
```

## Training N-Gram Chunkers

Our approach to chunking has been to try to detect structure based on the part-of-speech tags. We have seen that the IOB format represents this extra structure using another kind of tag. The question arises as to whether we could use the same *n*-gram tagging methods we saw in [Chapter 4](#), applied to a different vocabulary. In this case, rather than trying to determine the correct

part-of-speech tag, given a word, we are trying to determine the correct chunk tag, given a part-of-speech tag.

The first step is to get the `word, tag, chunk` triples from the CoNLL 2000 corpus and map these to `tag, chunk` pairs:

```
>>> chunk_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(chtree)]
... for chtree in conll2000.chunked_sents('train.txt')]
```

We will now train two  $n$ -gram taggers over this data. To start off, we train and score a **unigram chunker** on the above data, just as if it was a tagger:

```
>>> unigram_chunker = nltk.UnigramTagger(chunk_data)
>>> print nltk.tag.accuracy(unigram_chunker, chunk_data)
0.781378851068
```

This chunker does reasonably well. Let's look at the errors it makes. Consider the opening phrase of the first sentence of the CONLL chunking data, here shown with part-of-speech tags:

Confidence/NN in/IN the/DT pound/NN is/VBZ widely/RB expected/VBN to/TO take/VB another/DT sharp/JJ dive/NN

We can try out the unigram chunker on this first sentence by creating some "tokens" using `[t for t, c in chunk_data[0]]`, then running our chunker over them using `list(unigram_chunker.tag(tokens))`. The unigram chunker only looks at the tags, and tries to add chunk tags. Here is what it comes up with:

NN/I-NP IN/B-PP DT/B-NP NN/I-NP VBZ/B-VP RB/O VBN/I-VP TO/B-PP VB/I-VP DT/B-NP JJ/I-NP NN/I-NP

Notice that it tags all instances of `NN` with `I-NP`, because nouns usually do not appear at the beginning of noun phrases in the training data. Thus, the first noun `Confidence/NN` is tagged incorrectly. However, `pound/NN` and `dive/NN` are correctly tagged as `I-NP`; they are not in the initial position that should be tagged `B-NP`. The chunker incorrectly tags `widely/RB` as `O`, and it incorrectly tags the infinitival `to/TO` as `B-PP`, as if it was a preposition starting a prepositional phrase.

[Why these problems might go away if we look at the previous chunk tag?]

Let's run a bigram chunker:

```
>>> bigram_chunker = nltk.BigramTagger(chunk_data, backoff=unigram_chunker)
>>> print nltk.tag.accuracy(bigram_chunker, chunk_data)
0.893220987404
```

We can run the bigram chunker over the same sentence as before using `list(bigram_chunker.tag(tokens))`. Here is what it comes up with:

NN/B-NP IN/B-PP DT/B-NP NN/I-NP VBZ/B-VP RB/I-VP VBN/I-VP TO/I-VP VB/I-VP DT/B-NP JJ/I-NP  
NN/I-NP

This is 100% correct.

## 7.4 Building Nested Structure with Cascaded Chunkers

So far, our chunk structures have been relatively flat. Trees consist of tagged tokens, optionally grouped under a chunk node such as `NP`. However, it is possible to build chunk structures of arbitrary depth, simply by creating a multi-stage chunk grammar. These stages are processed in the order that they appear. The patterns in later stages can refer to a mixture of part-of-speech tags and chunk types. [Figure 7.11](#) has patterns for noun phrases, prepositional phrases, verb phrases, and sentences. This is a four-stage chunk grammar, and can be used to create structures having a depth of at most four.

```
grammar = r"""
NP: {<DT|JJ|NN.*+} # Chunk sequences of DT, JJ, NN
PP: {<IN><NP>} # Chunk prepositions followed by NP
VP: {<VB.*><NP|PP|S>+$} # Chunk rightmost verbs and arguments/adjuncts
S: {<NP><VP>} # Chunk NP, VP
"""

cp = nltk.RegexpParser(grammar)
```

```

tagged_tokens = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),
 ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]

```

```

>>> print cp.parse(tagged_tokens)
(S
 (NP Mary/NN)
 saw/VBD
 (S
 (NP the/DT cat/NN)
 (VP sit/VB (PP on/IN (NP the/DT mat/NN))))))

```

[Figure 7.11 \(cascaded\\_chunker.py\)](#): Figure 7.11: A Chunker that Handles NP, PP, VP and S

Unfortunately this result misses the VP headed by *saw*. It has other shortcomings too. Let's see what happens when we apply this chunker to a sentence having deeper nesting.

```

>>> tagged_tokens = [("John", "NNP"), ("thinks", "VBZ"), ("Mary", "NN"),
... ("saw", "VBD"), ("the", "DT"), ("cat", "NN"), ("sit", "VB"),
... ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(tagged_tokens)
(S
 (NP John/NNP)
 thinks/VBZ
 (NP Mary/NN)
 saw/VBD
 (S
 (NP the/DT cat/NN)
 (VP sit/VB (PP on/IN (NP the/DT mat/NN))))))

```

The solution to these problems is to get the chunker to loop over its patterns: after trying all of them, it repeats the process. We add an optional second argument `loop` to specify the number of times the set of patterns should be run:

```

>>> cp = nltk.RegexpParser(grammar, loop=2)
>>> print cp.parse(tagged_tokens)
(S
 (NP John/NNP)
 thinks/VBZ
 (S
 (NP Mary/NN)
 (VP
 saw/VBD
 (S
 (NP the/DT cat/NN)
 (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))))

```

This cascading process enables us to create deep structures. However, creating and debugging a cascade is quite difficult, and there comes a point where it is more effective to do full parsing (see [Chapter 8](#)).

## 7.5 Conclusion

In this chapter we have explored efficient and robust methods that can identify linguistic structures in text. Using only part-of-speech information for words in the local context, a "chunker" can successfully identify simple structures such as noun phrases and verb groups. We have seen how chunking methods extend the same lightweight methods that were successful in tagging. The resulting structured information is useful in information extraction tasks and in the description of the syntactic environments of words. The latter will be invaluable as we move to full parsing.

There are a surprising number of ways to chunk a sentence using regular expressions. The patterns can add, shift and remove chunks in many ways, and the patterns can be sequentially ordered in many ways. One can use a small number of very complex rules, or a long sequence of much simpler rules. One can hand-craft a collection of rules, and one can write programs to analyze a chunked corpus to help in the development of such rules. The process is painstaking, but generates very compact chunkers that perform well and that transparently encode linguistic knowledge.

It is also possible to chunk a sentence using the techniques of n-gram tagging. Instead of assigning part-of-speech tags to words, we assign IOB tags to the part-of-speech tags. Bigram tagging turned out to be particularly effective, as it could be sensitive to the chunk tag on the previous word. This statistical approach requires far less effort than rule-based chunking, but creates large models and delivers few linguistic insights.

Like tagging, chunking cannot be done perfectly. For example, as pointed out by [Church, Young, & Bloothooft, 1996], we cannot correctly analyze the structure of the sentence *I turned off the spectroroute* without knowing the meaning of *spectroroute*; is it a kind of road or a type of device? Without knowing this, we cannot tell whether *off* is part of a prepositional phrase indicating direction (tagged *B-PP*), or whether *off* is part of the verb-particle construction *turn off* (tagged *I-VP*).

A recurring theme of this chapter has been **diagnosis**. The simplest kind is manual, when we inspect the tracing output of a chunker and observe some undesirable behavior that we would like to fix. Sometimes we discover cases where we cannot hope to get the correct answer because the part-of-speech tags are too impoverished and do not give us sufficient information about the lexical item. A second approach is to write utility programs to analyze the training data, such as counting the number of times a given part-of-speech tag occurs inside and outside an NP chunk. A third approach is to evaluate the system against some gold standard data to obtain an overall performance score. We can even use this to parameterize the system, specifying which chunk rules are used on a given run, and tabulating performance for different parameter combinations. Careful use of these diagnostic methods permits us to optimize the performance of our system. We will see this theme emerge again later in chapters dealing with other topics in natural language processing.

## 7.6 Further Reading

For more examples of chunking with NLTK, please see the guide at <http://nltk.org/doc/guides/chunk.html>.

The popularity of chunking is due in great part to pioneering work by Abney e.g., [Church, Young, & Bloothooft, 1996]. Abney's Cass chunker is available at <http://www.vinartus.net/spa/97a.pdf>

The word **chink** initially meant a sequence of stopwords, according to a 1975 paper by Ross and Tukey [Church, Young, & Bloothooft, 1996].

The IOB format (or sometimes **BIO Format**) was developed for NP chunking by [Ramshaw & Marcus, 1995], and was used for the shared NP bracketing task run by the *Conference on Natural Language Learning* (CoNLL) in 1999. The same format was adopted by CoNLL 2000 for annotating a section of Wall Street Journal text as part of a shared task on NP chunking.

Section 13.5 of [Jurafsky & Martin, 2008] contains a discussion of chunking.

Alexandria Gazetteer: <http://www.alexandria.ucsb.edu/gazetteer>

Getty Gazetteer

We are grateful to Nicola Stokes for providing the example in [Figure 7.1](#).

## 7.7 Exercises

1. ☀ **Chunk Grammar Development:** Try developing a series of chunking rules using the graphical interface accessible via `nltk.draw.rechunkparser.demo()`
2. ☀ **Chunking Demonstration:** Run the chunking demonstration: `nltk.chunk.demo()`
3. ☀ **IOB Tags:** The IOB format categorizes tagged tokens as `I`, `O` and `B`. Why are three tags necessary? What problem would be caused if we used `I` and `O` tags exclusively?
4. ☀ Write a tag pattern to match noun phrases containing plural head nouns, e.g. "many/JJ researchers/NNS", "two/CD weeks/NNS", "both/DT new/JJ positions/NNS". Try to do this by generalizing the tag pattern that handled singular noun phrases.
5. ● Write a tag pattern to cover noun phrases that contain gerunds, e.g. "the/DT receiving/VBG end/NN", "assistant/NN managing/VBG editor/NN". Add these patterns to the grammar, one per line. Test your work using some tagged sentences of your own devising.
6. ● Write one or more tag patterns to handle coordinated noun phrases, e.g. "July/NNP and/CC August/NNP", "all/DT your/PRP\$ managers/NNS and/CC supervisors/NNS", "company/NN courts/NNS and/CC adjudicators/NNS".
7. ● **Chunker Evaluation:** Carry out the following evaluation tasks for any of the chunkers you have developed earlier. (Note that most chunking corpora contain some internal inconsistencies, such that any reasonable rule-based approach will produce errors.)
  1. Evaluate your chunker on 100 sentences from a chunked corpus, and report the precision, recall and F-measure.
  2. Use the `chunkscore.missed()` and `chunkscore.incorrect()` methods to identify the errors made by your chunker. Discuss.
  3. Compare the performance of your chunker to the baseline chunker discussed in the evaluation section of this

chapter.

8. ★ **Transformation-Based Chunking:** Apply the n-gram and Brill tagging methods to IOB chunk tagging. Instead of assigning POS tags to words, here we will assign IOB tags to the POS tags. E.g. if the tag DT (determiner) often occurs at the start of a chunk, it will be tagged B (begin). Evaluate the performance of these chunking methods relative to the regular expression chunking methods covered in this chapter.
9. ☀ Pick one of the three chunk types in the CoNLL corpus. Inspect the CoNLL corpus and try to observe any patterns in the POS tag sequences that make up this kind of chunk. Develop a simple chunker using the regular expression chunker `nltk.RegexpParser`. Discuss any tag sequences that are difficult to chunk reliably.
10. ☀ An early definition of *chunk* was the material that occurs between chinks. Develop a chunker that starts by putting the whole sentence in a single chunk, and then does the rest of its work solely by chinking. Determine which tags (or tag sequences) are most likely to make up chinks with the help of your own utility program. Compare the performance and simplicity of this approach relative to a chunker based entirely on chunk rules.
11. ● Develop a chunker for one of the chunk types in the CoNLL corpus using a regular-expression based chunk grammar `RegexpChunk`. Use any combination of rules for chunking, chinking, merging or splitting.
12. ● Sometimes a word is incorrectly tagged, e.g. the head noun in "12/CD or/CC so/RB cases/VBZ". Instead of requiring manual correction of tagger output, good chunkers are able to work with the erroneous output of taggers. Look for other examples of correctly chunked noun phrases with incorrect tags.
13. ★ We saw in the tagging chapter that it is possible to establish an upper limit to tagging performance by looking for ambiguous n-grams, n-grams that are tagged in more than one possible way in the training data. Apply the same method to determine an upper bound on the performance of an n-gram chunker.
14. ★ Pick one of the three chunk types in the CoNLL corpus. Write functions to do the following tasks for your chosen type:
  1. List all the tag sequences that occur with each instance of this chunk type.
  2. Count the frequency of each tag sequence, and produce a ranked list in order of decreasing frequency; each line should consist of an integer (the frequency) and the tag sequence.
  3. Inspect the high-frequency tag sequences. Use these as the basis for developing a better chunker.
15. ★ The baseline chunker presented in the evaluation section tends to create larger chunks than it should. For example, the phrase: [every/DT time/NN] [she/PRP] sees/VBZ [a/DT newspaper/NN] contains two consecutive chunks, and our baseline chunker will incorrectly combine the first two: [every/DT time/NN she/PRP]. Write a program that finds which of these chunk-internal tags typically occur at the start of a chunk, then devise one or more rules that will split up these chunks. Combine these with the existing baseline chunker and re-evaluate it, to see if you have discovered an improved baseline.
16. ★ Develop an NP chunker that converts POS-tagged text into a list of tuples, where each tuple consists of a verb followed by a sequence of noun phrases and prepositions, e.g. the little cat sat on the mat becomes ('sat', 'on', 'NP') ...
17. ★ The Penn Treebank contains a section of tagged Wall Street Journal text that has been chunked into noun phrases. The format uses square brackets, and we have encountered it several times during this chapter. The Treebank corpus can be accessed using: `for sent in nltk.corpus.treebank_chunk.chunked_sents(file)`. These are flat trees, just as we got using `nltk.corpus.conll2000.chunked_sents()`.
  1. The functions `nltk.tree pprint()` and `nltk.chunk.tree2conllstr()` can be used to create Treebank and IOB strings from a tree. Write functions `chunk2brackets()` and `chunk2iob()` that take a single chunk tree as their sole argument, and return the required multi-line string representation.
  2. Write command-line conversion utilities `bracket2iob.py` and `iob2bracket.py` that take a file in Treebank or CoNLL format (resp) and convert it to the other format. (Obtain some raw Treebank or CoNLL data from the NLTK Corpora, save it to a file, and then use `for line in open(filename)` to access it from Python.)
18. ● The bigram chunker scores about 90% accuracy. Study its errors and try to work out why it doesn't get 100% accuracy.
19. ● Experiment with trigram chunking. Are you able to improve the performance any more?
20. ★ An n-gram chunker can use information other than the current part-of-speech tag and the *n-1* previous chunk tags. Investigate other models of the context, such as the *n-1* previous part-of-speech tags, or some combination of previous chunk tags along with previous and following part-of-speech tags.
21. ★ Consider the way an n-gram tagger uses recent tags to inform its tagging choice. Now observe how a chunker may re-use this sequence information. For example, both tasks will make use of the information that nouns tend to follow adjectives (in English). It would appear that the same information is being maintained in two places. Is this likely to become a problem as the size of the rule sets grows? If so, speculate about any ways that this problem might be addressed.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008

the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

## 8 Grammars and Parsing

Earlier chapters focused on words: how to identify them, analyze their structure, assign them to lexical categories, and access their meanings. We have also seen how to identify patterns in word sequences or n-grams. However, these methods only scratch the surface. We need a way to deal with the ambiguity that natural language is famous for. We also need to be able to cope with the fact that there are an infinite number of possible sentences, and we can only write finite programs to analyze their structures and discover their meanings.

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

1. How can we use a formal grammar to describe the structure of an infinite set of sentences?
2. How do we represent the structure of sentences using syntax trees?
3. How do parsers analyze a sentence and automatically build a syntax tree?

Along the way, we will cover the fundamentals of English syntax, and see that there are systematic aspects of meaning that are much easier to capture once we have identified the structure of sentences.

### 8.1 Some Grammatical Dilemmas

#### Infinite Possibilities

Sentences have an interesting property that they can be embedded inside larger sentences. Consider the following examples:

(24)

- a. Usain Bolt broke the 100m record
- b. The Jamaica Observer reported that Usain Bolt broke the 100m record
- c. Andre said The Jamaica Observer reported that Usain Bolt broke the 100m record
- d. I think Andre said the Jamaica Observer reported that Usain Bolt broke the 100m record

If we replaced whole sentences with the symbol *S*, we would see patterns like *Andre said S* and *I think S*. These are templates for taking a sentence and constructing a bigger sentence. There are other templates we can use, like *S but S*, and *S when S*. With a bit of ingenuity we can construct some really long sentences using these templates. Here's an impressive example from a Winnie the Pooh story by A.A. Milne, *In which Piglet is Entirely Surrounded by Water*:

You can imagine Piglet's joy when at last the ship came in sight of him. In after-years he liked to think that he had been in Very Great Danger during the Terrible Flood, but the only danger he had really been in was the last half-hour of his imprisonment, when Owl, who had just flown up, sat on a branch of his tree to comfort him, and told him a very long story about an aunt who had once laid a seagull's egg by mistake, and the story went on and on, rather like this sentence, until Piglet who was listening out of his window without much hope, went to sleep quietly and naturally, slipping slowly out of the window towards the water until he was only hanging on by his toes, at which moment, luckily, a sudden loud squawk from Owl, which was really part of the story, being what his aunt said, woke the Piglet up and just gave him time to jerk himself back into safety and say, "How interesting, and did she?" when -- well, you can imagine his joy when at last he saw the good ship, Brain of Pooh (Captain, C. Robin; 1st Mate, P. Bear) coming over the sea to rescue him...

This long sentence actually has a simple structure that begins *S but S when S*. We can see from this example that there is no upper bound on the length of a sentence. Its striking that we can produce and understand sentences of arbitrary length that we've never heard before. Its not hard to concoct an entirely novel sentence, one that has probably never been used before in the history of the language, yet all speakers of the language will agree about its meaning.

The purpose of a grammar is to provide a compact definition of this infinite set of sentences. It achieves this using **recursion**, with the help of grammar **productions** of the form  $S \rightarrow S$  and  $S$ , as we will explore in [8.3](#). In [11](#) we will extend this, to automatically build up the meaning of a sentence out of the meanings of its parts.

## Ubiquitous Ambiguity

A well-known example of ambiguity is shown in [\(2\)](#) (from Groucho Marx, *Animal Crackers*, 1930):

- (25) While hunting in Africa, I shot an elephant in my pajamas. How an elephant got into my pajamas I'll never know.

Let's take a closer look at the ambiguity in the phrase: *I shot an elephant in my pajamas*. First we need to define a simple grammar:

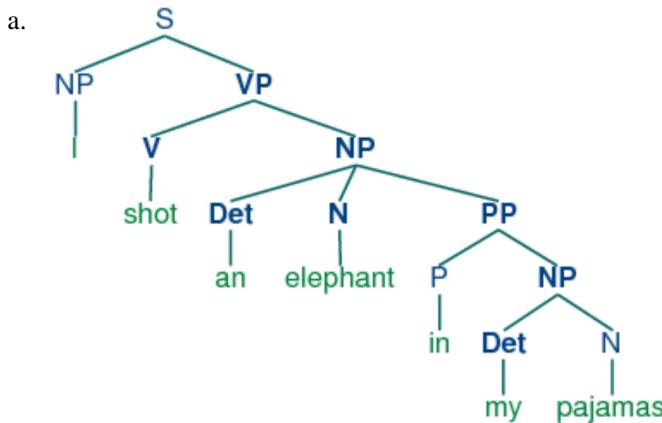
```
>>> groucho_grammar = nltk.parse_cfg("""
... S -> NP VP
... PP -> P NP
... NP -> Det N | Det N PP | 'I'
... VP -> V NP | VP PP
... Det -> 'an' | 'my'
... N -> 'elephant' | 'pajamas'
... V -> 'shot'
... P -> 'in'
... """)
```

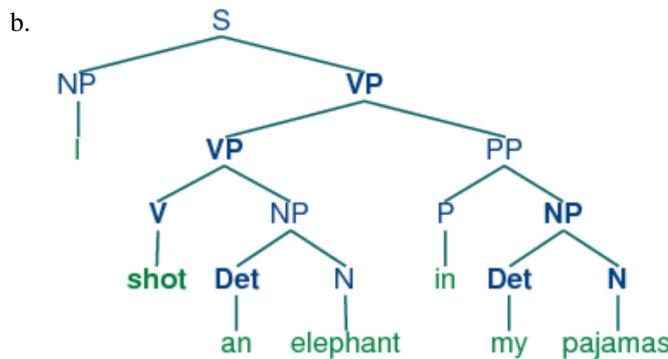
This grammar permits the sentence to be analyzed in two ways, depending on whether the prepositional phrase *in my pajamas* describes the elephant or the shooting event.

```
>>> sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
>>> parser = nltk.ChartParser(groucho_grammar)
>>> trees = parser.nbest_parse(sent)
>>> for tree in trees:
... print tree
...
(S
 (NP I)
 (VP
 (V shot)
 (NP (Det an) (N elephant)) (PP (P in) (NP (Det my) (N pajamas))))))
(S
 (NP I)
 (VP
 (V shot)
 (NP (Det an) (N elephant)))
 (PP (P in) (NP (Det my) (N pajamas))))
```

The program produces two bracketed structures, which we can depict as trees, as shown in [\(3b\)](#):

- (26)





Notice that there's no ambiguity concerning the meaning of any of the words. E.g. the word *shot* doesn't refer to the act of using a gun in the first sentence, and using a camera in the second sentence.

### Note

**Your Turn:** Consider the following sentences and see if you can think of two quite different interpretations: *Fighting animals could be dangerous. Visiting relatives can be tiresome..* Is ambiguity of the individual words to blame? If not, what is the cause of the ambiguity?

Perhaps another kind of syntactic variation, word order, is easier to understand. We know that the two sentences *Kim likes Sandy* and *Sandy likes Kim* have different meanings, and that *likes Sandy Kim* is simply ungrammatical. Similarly, we know that the following two sentences are equivalent:

- (27)
- The farmer *loaded* the cart with sand
  - The farmer *loaded* sand into the cart

However, consider the semantically similar verbs *filled* and *dumped*. Now the word order cannot be altered (ungrammatical sentences are prefixed with an asterisk.)

- (28)
- The farmer *filled* the cart with sand
  - \*The farmer *filled* sand into the cart
  - \*The farmer *dumped* the cart with sand
  - The farmer *dumped* sand into the cart

This chapter presents grammars and parsing, as the formal and computational methods for investigating and modeling the linguistic phenomena we have been touching on (or tripping over). As we shall see, patterns of well-formedness and ill-formedness in a sequence of words can be understood with respect to the underlying **phrase structure** of the sentences. We can develop formal models of these structures using grammars and parsers. As before, the motivation is natural language *understanding*. How much more of the meaning of a text can we access when we can reliably recognize the linguistic structures it contains? Having read in a text, can a program 'understand' it enough to be able to answer simple questions about "what happened" or "who did what to whom." Also as before, we will develop simple programs to process annotated corpora and perform useful tasks.

## Old School Grammar

Early experiences with the kind of grammar taught in school are sometimes perplexing, if not downright frustrating. Written work is graded by a teacher who red-lined all the grammar errors they wouldn't put up with. Like the plural pronoun or the dangling preposition in the last sentence, or sentences like this one that lack a main verb. If you learnt English as a second

language, you might have found it difficult to discover which of these errors actually need to be fixed (or should that be: *needs* to be fixed?). As a consequence, many people are afraid of grammar. However, as users of language we depend on our knowledge of grammar in order to produce and understand sentences. To see why grammar matters, consider the following two sentences, which have an important difference of meaning:

(29)

- a. The vice-presidential candidate, who was wearing a \$10,000 outfit, smiled broadly.
- b. The vice-presidential candidate who was wearing a \$10,000 outfit smiled broadly.

In (29a), we assume there is just one candidate, and say two things about her: that she was wearing an expensive outfit and that she smiled. In (29b), on the other hand, we use the description *who was wearing a \$10,000 outfit* as a means of identifying which particular candidate we are referring to. In the above examples, punctuation is a clue to grammatical structure. In particular, it tells us whether the relative clause is restrictive or non-restrictive.

In contrast, other grammatical concerns are nothing more than vestiges of antiquated style. For example, consider the injunction that *however* — when used to mean *nevertheless* — must not appear at the start of a sentence. Pullum argues that Strunk and White [Strunk & White, 1999] were merely insisting that English usage should conform to "an utterly unimportant minor statistical detail of style concerning adverb placement in the literature they knew" [Pullum, 2005]. This is a case where, a *descriptive* observation about language use became a *prescriptive* requirement.

In NLP we usually discard such prescriptions, and use grammar to formalize observations about language as it is used, particularly as it is used in corpora. We create our own formal grammars and write programs to parse sentences. This is a far cry from "old school" grammar. It is a thoroughly objective approach that makes grammatical structures explicit with the help of corpora, formal grammars, and parsers.

## 8.2 What's the Use of Syntax?

### Syntactic Ambiguity

We have seen that sentences can be ambiguous. If we overheard someone say *I went to the bank*, we wouldn't know whether it was a river bank or a financial institution. This ambiguity concerns the meaning of the word *bank*, and is a kind of **lexical ambiguity**.

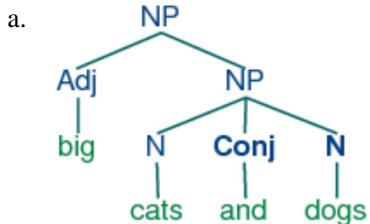
However, other kinds of ambiguity cannot be explained in terms of ambiguity of specific words. Consider a phrase involving an adjective with a conjunction: *big cats and dogs*. Does *big* have wider scope than *and*, or is it the other way round? In fact, both interpretations are possible, and we can represent the different scopes using parentheses:

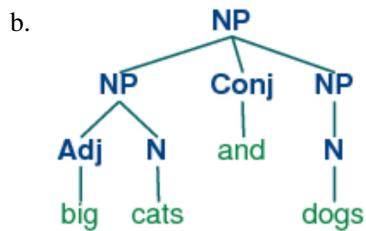
(30)

- a. big (cats and dogs)
- b. (big cats) and dogs

One convenient way of representing this scope difference at a structural level is by means of a **tree diagram**, as shown in (8b).

(31)





Note that linguistic trees grow upside down: the node labeled **s** is the **root** of the tree, while the **leaves** of the tree are labeled with the words.

In NLTK, you can easily produce trees like this yourself with the following commands:

```

>>> tree = nltk.Tree(' (NP (Adj big) (NP (N cats) (Conj and) (N dogs))) ')
>>> tree.draw()

```

We can construct other examples of syntactic ambiguity involving the coordinating conjunctions *and* and *or*, e.g. *Kim left or Dana arrived and everyone cheered*. We can describe this ambiguity in terms of the relative semantic **scope** of *or* and *and*.

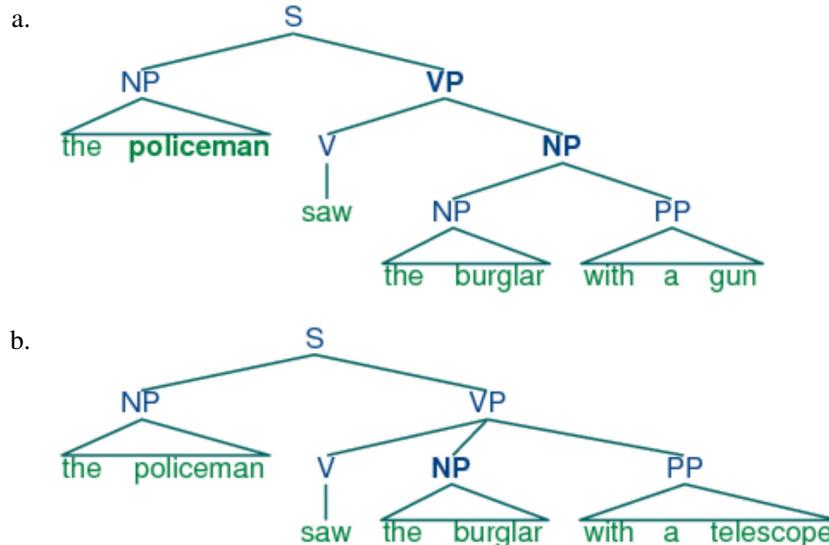
For our third illustration of ambiguity, we look at prepositional phrases. Consider a sentence like: *I saw the man with a telescope*. Who has the telescope? To clarify what is going on here, consider the following pair of sentences:

(32)

- a. The policeman saw a burglar *with a gun*. (not some other burglar)
- b. The policeman saw a burglar *with a telescope*. (not with his naked eye)

In both cases, there is a prepositional phrase introduced by *with*. In the first case this phrase modifies the noun *burglar*, and in the second case it modifies the verb *saw*. We could again think of this in terms of scope: does the prepositional phrase (PP) just have scope over the NP *a burglar*, or does it have scope over the whole verb phrase? As before, we can represent the difference in terms of tree structure:

(33)



In (32)a, the PP attaches to the NP, while in (32)b, the PP attaches to the VP.

We can generate these trees in Python as follows:

```

>>> s1 = '(S (NP the policeman) (VP (V saw) (NP (NP the burglar) (PP with a gun))))'
>>> s2 = '(S (NP the policeman) (VP (V saw) (NP the burglar) (PP with a telescope)))'
>>> tree1 = nltk.bracket_parse(s1)
>>> tree2 = nltk.bracket_parse(s2)

```

We can discard the structure to get the list of **leaves**, and we can confirm that both trees have the same leaves (except for the last word). We can also see that the trees have different **heights** (given by the number of nodes in the longest branch of the tree, starting at S and descending to the words):

```
>>> tree1.leaves()
['the', 'policeman', 'saw', 'the', 'burglar', 'with', 'a', 'gun']
>>> tree1.leaves()[:-1] == tree2.leaves()[:-1]
True
>>> tree1.height() == tree2.height()
False
```

In general, how can we determine whether a prepositional phrase modifies the preceding noun or verb? This problem is known as **prepositional phrase attachment ambiguity**. The **Prepositional Phrase Attachment Corpus** makes it possible for us to study this question systematically. The corpus is derived from the IBM-Lancaster Treebank of Computer Manuals and from the Penn Treebank, and distills out only the essential information about PP attachment. Consider the sentence from the WSJ in (34a). The corresponding line in the Prepositional Phrase Attachment Corpus is shown in (34b).

(34)

- a. Four of the five surviving workers have asbestos-related diseases, including three with recently diagnosed cancer.
- b. 16 including three with cancer N

That is, it includes an identifier for the original sentence, the head of the relevant verb phrase (i.e., *including*), the head of the verb's NP object (*three*), the preposition (*with*), and the head noun within the prepositional phrase (*cancer*). Finally, it contains an "attachment" feature (N or V) to indicate whether the prepositional phrase attaches to (modifies) the noun phrase or the verb phrase. Here are some further examples:

(35) 47830 allow visits between families N  
 47830 allow visits on peninsula V  
 42457 acquired interest in firm N  
 42457 acquired interest in 1986 V

The PP attachments in (12) can also be made explicit by using phrase groupings as in (13).

(36) allow (NP visits (PP between families))  
 allow (NP visits) (PP on peninsula)  
 acquired (NP interest (PP in firm))  
 acquired (NP interest) (PP in 1986)

Observe in each case that the argument of the verb is either a single complex expression (*visits (between families)*) or a pair of simpler expressions (*visits*) (*on peninsula*).

We can access the Prepositional Phrase Attachment Corpus from NLTK as follows:

```
>>> nltk.corpus.pptattach.tuples('training')[9]
('16', 'including', 'three', 'with', 'cancer', 'N')
```

If we go back to our first examples of PP attachment ambiguity, it appears as though it is the PP itself (e.g., *with a gun* versus *with a telescope*) that determines the attachment. However, we can use this corpus to find examples where other factors come into play. For example, it appears that the verb is the key factor in (14).

(37) 8582 received offer from group V  
 19131 rejected offer from group N

## Constituency

We claimed earlier that one of the motivations for building syntactic structure was to help make explicit how a sentence says "who did what to whom". Let's just focus for a while on the "who" part of this story: in other words, how can syntax tell us what the subject of a sentence is? At first, you might think this task is rather simple — so simple indeed that we don't need to bother with syntax. In a sentence such as *The fierce dog bit the man* we know that it is the dog that is doing the biting. So we could say that the noun phrase immediately preceding the verb is the subject of the sentence. And we might try to make this more explicit in terms of sequences part-of-speech tags. Let's try to come up with a simple definition of noun phrase; we might start off with

something like this, based on our knowledge of noun phrase chunking ([Chapter 7](#)):

- (38) DT JJ\* NN

We're using regular expression notation here in the form of `JJ*` to indicate a sequence of zero or more `JJs`. So this is intended to say that a noun phrase can consist of a determiner, possibly followed by some adjectives, followed by a noun. Then we can go on to say that if we can find a sequence of tagged words like this that precedes a word tagged as a verb, then we've identified the subject. But now think about this sentence:

- (39) The child with a fierce dog bit the man.

This time, it's the child that is doing the biting. But the tag sequence preceding the verb is:

- (40) DT NN IN DT JJ NN

Our previous attempt at identifying the subject would have incorrectly come up with *the fierce dog* as the subject. So our next hypothesis would have to be a bit more complex. For example, we might say that the subject can be identified as any string matching the following pattern before the verb:

- (41) DT JJ\* NN (IN DT JJ\* NN)\*

In other words, we need to find a noun phrase followed by zero or more sequences consisting of a preposition followed by a noun phrase. Now there are two unpleasant aspects to this proposed solution. The first is esthetic: we are forced into repeating the sequence of tags (DT JJ\* NN) that constituted our initial notion of noun phrase, and our initial notion was in any case a drastic simplification. More worrying, this approach still doesn't work! For consider the following example:

- (42) The seagull that attacked the child with the fierce dog bit the man.

This time the seagull is the culprit, but it won't be detected as subject by our attempt to match sequences of tags. So it seems that we need a richer account of how words are *grouped* together into patterns, and a way of referring to these groupings at different points in the sentence structure. This idea of grouping is often called syntactic **constituency**.

As we have just seen, a well-formed sentence of a language is more than an arbitrary sequence of words from the language. Certain kinds of words usually go together. For instance, determiners like *the* are typically followed by adjectives or nouns, but not by verbs. Groups of words form intermediate structures called phrases or **constituents**. These constituents can be identified using standard syntactic tests, such as substitution, movement and coordination. For example, if a sequence of words can be replaced with a pronoun, then that sequence is likely to be a constituent. According to this test, we can infer that the italicized string in the following example is a constituent, since it can be replaced by *they*:

- (43)

- a. *Ordinary daily multivitamin and mineral supplements* could help adults with diabetes fight off some minor infections.
- b. *They* could help adults with diabetes fight off some minor infections.

In order to identify whether a phrase is the subject of a sentence, we can use the construction called **Subject-Auxiliary Inversion** in English. This construction allows us to form so-called Yes-No Questions. That is, corresponding to the statement in (44a), we have the question in (44b):

- (44)

- a. All the cakes have been eaten.
- b. Have *all the cakes* been eaten?

Roughly speaking, if a sentence already contains an auxiliary verb, such as *has* in (44a), then we can turn it into a Yes-No Question by moving the auxiliary verb 'over' the subject noun phrase to the front of the sentence. If there is no auxiliary in the statement, then we insert the appropriate form of *do* as the fronted auxiliary and replace the tensed main verb by its base form:

- (45)

- a. The fierce dog bit the man.

- b. Did *the fierce dog* bite the man?

As we would hope, this test also confirms our earlier claim about the subject constituent of (19):

- (46) Did *the seagull that attacked the child with the fierce dog* bite the man?

To sum up then, we have seen that the notion of constituent brings a number of benefits. By having a constituent labeled NOUN PHRASE, we can provide a unified statement of the classes of word that constitute that phrase, and reuse this statement in describing noun phrases wherever they occur in the sentence. Second, we can use the notion of a noun phrase in defining the subject of sentence, which in turn is a crucial ingredient in determining the "who does what to whom" aspect of meaning.

## 8.3 Context Free Grammar

As we have seen, languages are infinite — there is no principled upper-bound on the length of a sentence. Nevertheless, we would like to write (finite) programs that can process well-formed sentences. It turns out that we can characterize what we mean by well-formedness using a grammar. The way that finite grammars are able to describe an infinite set uses **recursion**. (We already came across this idea when we looked at regular expressions: the finite expression  $a^+$  is able to describe the infinite set  $\{a, aa, aaa, aaaa, \dots\}$ ). Apart from their compactness, grammars usually capture important structural and distributional properties of the language, and can be used to map between sequences of words and abstract representations of meaning. Even if we were to impose an upper bound on sentence length to ensure the language was finite, we would probably still want to come up with a compact representation in the form of a grammar.

A **grammar** is a formal system that specifies which sequences of words are well-formed in the language, and that provides one or more phrase structures for well-formed sequences. We will be looking at **context-free grammar** (CFG), which is a collection of **productions** of the form  $S \rightarrow NP VP$ . This says that a constituent S can consist of sub-constituents NP and VP. Similarly, the production  $V \rightarrow 'saw' \mid ``walked''$  means that the constituent V can consist of the string *saw* or *walked*. For a phrase structure tree to be well-formed relative to a grammar, each non-terminal node and its children must correspond to a production in the grammar.

### A Simple Grammar

Let's start off by looking at a simple context-free grammar. By convention, the left-hand-side of the first production is the **start-symbol** of the grammar, and all well-formed trees must have this symbol as their root label.

- (47)  $S \rightarrow NP VP$   
 $NP \rightarrow Det N \mid Det N PP$   
 $VP \rightarrow V \mid V NP \mid V NP PP$   
 $PP \rightarrow P NP$

- $Det \rightarrow 'the' \mid 'a'$   
 $N \rightarrow 'man' \mid 'park' \mid 'dog' \mid 'telescope'$   
 $V \rightarrow 'saw' \mid 'walked'$   
 $P \rightarrow 'in' \mid 'with'$

This grammar contains productions involving various syntactic categories, as laid out in [Table 8.1](#).

**Table 8.1:**

#### Syntactic Categories

| Symbol | Meaning              | Example                 |
|--------|----------------------|-------------------------|
| S      | sentence             | <i>the man walked</i>   |
| NP     | noun phrase          | <i>a dog</i>            |
| VP     | verb phrase          | <i>saw a park</i>       |
| PP     | prepositional phrase | <i>with a telescope</i> |
| ...    | ...                  | ...                     |
| Det    | determiner           | <i>the</i>              |
| N      | noun                 | <i>dog</i>              |
| V      | verb                 | <i>walked</i>           |

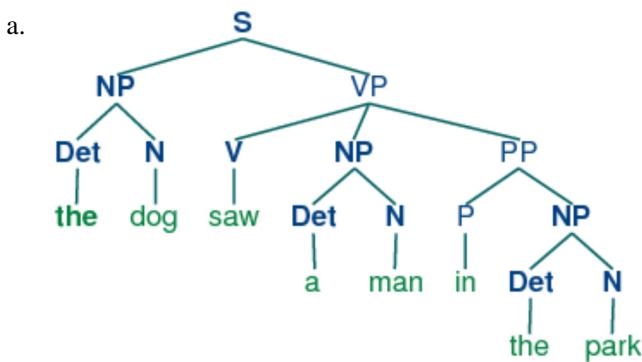
| Symbol | Meaning     | Example |
|--------|-------------|---------|
| P      | preposition | in      |

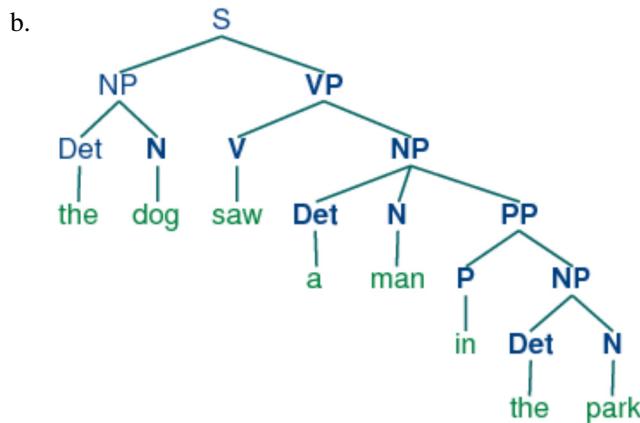
In our following discussion of grammar, we will use the following terminology. The grammar consists of productions, where each production involves a single **non-terminal** (e.g. S, NP), an arrow, and one or more non-terminals and **terminals** (e.g. *walked*). The productions are often divided into two main groups. The **grammatical productions** are those without a terminal on the right hand side. The **lexical productions** are those having a terminal on the right hand side. A special case of non-terminals are the **pre-terminals**, which appear on the left-hand side of lexical productions. We will say that a grammar **licenses** a tree if each non-terminal X with children Y<sub>1</sub> ... Y<sub>n</sub> corresponds to a production in the grammar of the form: X → Y<sub>1</sub> ... Y<sub>n</sub>.

In order to get started with developing simple grammars of your own, you will probably find it convenient to play with the recursive descent parser demo, `nltk.draw.rdparsert.demo()`. The demo opens a window that displays a list of grammar productions in the left hand pane and the current parse diagram in the central pane:

The demo comes with the grammar in (24) already loaded. We will discuss the parsing algorithm in greater detail below, but for the time being you can get an idea of how it works by using the *autostep* button. If we parse the string *The dog saw a man in the park* using the grammar in (24), we end up with two trees:

(48)





Since our grammar licenses two trees for this sentence, the sentence is said to be **structurally ambiguous**. The ambiguity in question is called a prepositional phrase attachment ambiguity, as we saw earlier in this chapter. As you may recall, it is an ambiguity about attachment since the *PP in the park* needs to be attached to one of two places in the tree: either as a daughter of *VP* or else as a daughter of *NP*. When the *PP* is attached to *VP*, the seeing event happened in the park. However, if the *PP* is attached to *NP*, then the man was in the park, and the agent of the seeing (the dog) might have been sitting on the balcony of an apartment overlooking the park. As we will see, dealing with ambiguity is a key challenge in parsing.

## Recursion in Syntactic Structure

A grammar is said to be **recursive** if a category occurring on the left hand side of a production (such as *S* in this case) also appears on the right hand side of a production. If this dual occurrence takes place in *one and the same production*, then we have **direct recursion**; otherwise we have **indirect recursion**. There is no recursion in (24). However, the grammar in (26) illustrates both kinds of recursive production:

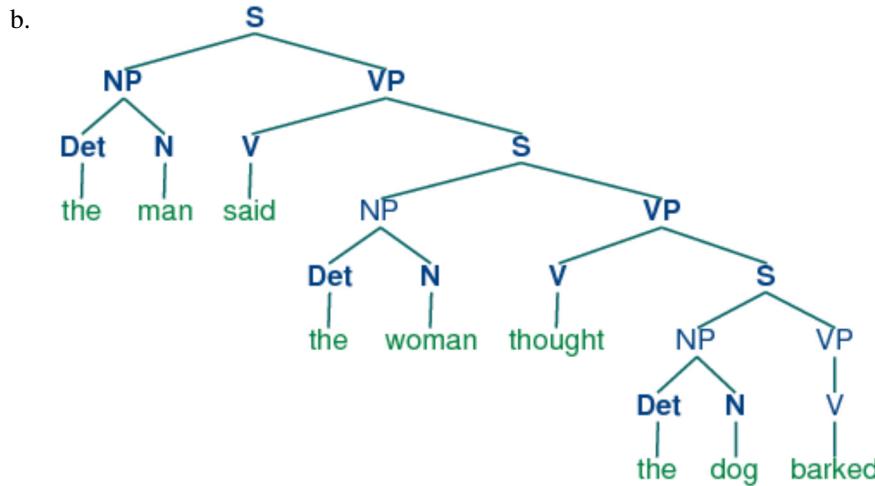
- (49)  $S \rightarrow NP\ VP$   
 $NP \rightarrow Det\ Nom \mid Det\ Nom\ PP \mid PropN$   
 $Nom \rightarrow Adj\ Nom \mid N$   
 $VP \rightarrow V \mid V\ NP \mid V\ NP\ PP \mid V\ S$   
 $PP \rightarrow P\ NP$
- $PropN \rightarrow 'John' \mid 'Mary'$   
 $Det \rightarrow 'the' \mid 'a'$   
 $N \rightarrow 'man' \mid 'woman' \mid 'park' \mid 'dog' \mid 'lead' \mid 'telescope' \mid 'butterfly'$   
 $Adj \rightarrow 'fierce' \mid 'black' \mid 'big' \mid 'European'$   
 $V \rightarrow 'saw' \mid 'chased' \mid 'barked' \mid 'disappeared' \mid 'said' \mid 'reported'$   
 $P \rightarrow 'in' \mid 'with'$

Notice that the production *NOM*  $\rightarrow$  *ADJ NOM* (where *NOM* is the category of nominals) involves direct recursion on the category *NOM*, whereas indirect recursion on *S* arises from the combination of two productions, namely *S*  $\rightarrow$  *NP VP* and *VP*  $\rightarrow$  *V S*.

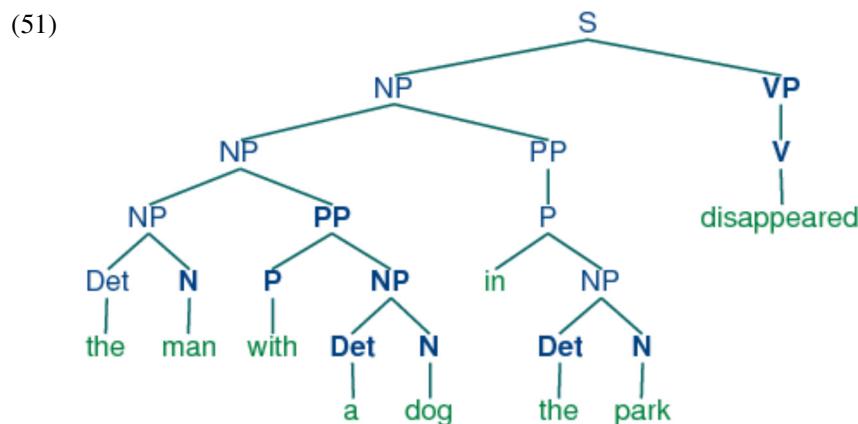
To see how recursion is handled in this grammar, consider the following trees. Example [nested-nominals](#) involves nested nominal phrases, while [nested-sentences](#) contains nested sentences.

- (50)
- a.
- 
- ```

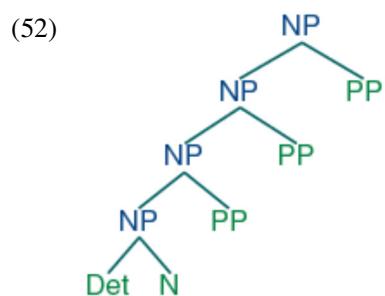
graph TD
    S --- NP1[NP]
    S --- VP[VP]
    NP1 --- Det1[Det]
    NP1 --- Nom1[Nom]
    Nom1 --- Adj1[Adj]
    Nom1 --- Nom2[Nom]
    Adj1 --- fierce[fierce]
    Nom2 --- Adj2[Adj]
    Nom2 --- N1[N]
    Adj2 --- black[black]
    N1 --- dog[dog]
    VP --- V[V]
    VP --- NP2[NP]
    NP2 --- Det2[Det]
    NP2 --- Nom3[Nom]
    Nom3 --- Adj3[Adj]
    Nom3 --- Nom4[Nom]
    Adj3 --- big[big]
    Nom4 --- Adj4[Adj]
    Nom4 --- N2[N]
    Adj4 --- European[European]
    N2 --- butterfly[butterfly]
    
```



If you did the exercises for the last section, you will have noticed that the recursive descent parser fails to deal properly with the following production: $NP \rightarrow NP\ PP$. From a linguistic point of view, this production is perfectly respectable, and will allow us to derive trees like this:



More schematically, the trees for these compound noun phrases will be of the following shape:



The structure in (29) is called a **left recursive** structure. These occur frequently in analyses of English, and the failure of recursive descent parsers to deal adequately with left recursion means that we will need to find alternative approaches.

Heads, Complements and Modifiers

Let us take a closer look at verbs. The grammar (26) correctly generates examples like (30d), corresponding to the four productions with VP on the left hand side:

- (53)
- The woman gave the telescope to the dog
 - The woman saw a man

- c. A man said that the woman disappeared
- d. The dog barked

That is, *gave* can occur with a following NP and PP; *saw* can occur with a following NP; *said* can occur with a following S; and *barked* can occur with no following phrase. In these cases, NP, PP and S are called **complements** of the respective verbs, and the verbs themselves are called **heads** of the verb phrase.

However, there are fairly strong constraints on what verbs can occur with what complements. Thus, we would like our grammars to mark the following examples as ungrammatical:

- (54)
- a. *The woman disappeared the telescope to the dog
 - b. *The dog barked a man
 - c. *A man gave that the woman disappeared
 - d. *A man said

Note

It is possible to create examples that involve 'non-standard' but interpretable combinations of verbs and complements. Thus, we can, at a stretch, interpret *the man disappeared the dog* to mean that the man made the dog disappear. We will ignore such examples here.

How can we ensure that our grammar correctly excludes the ungrammatical examples in (31d)? We need some way of constraining grammar productions which expand VP so that verbs *only* co-occur with their correct complements. We do this by dividing the class of verbs into **subcategories**, each of which is associated with a different set of complements. For example, **transitive verbs** such as *saw*, *kissed* and *hit* require a following NP object complement. Borrowing from the terminology of chemistry, we sometimes refer to the **valency** of a verb, that is, its capacity to combine with a sequence of arguments and thereby compose a verb phrase.

Let's introduce a new category label for such verbs, namely TV (for Transitive Verb), and use it in the following productions:

- (55) $\begin{array}{l} \text{VP} \rightarrow \text{TV NP} \\ \text{TV} \rightarrow \text{'saw'} \mid \text{'kissed'} \mid \text{'hit'} \end{array}$

Now **the dog barked the man* is excluded since we haven't listed *barked* as a V_TR, but *the woman saw a man* is still allowed. [Table 8.2](#) provides more examples of labels for verb subcategories.

Table 8.2:

Verb Subcategories

| Symbol | Meaning | Example |
|--------|-------------------|-------------------------------|
| IV | intransitive verb | <i>barked</i> |
| TV | transitive verb | <i>saw a man</i> |
| DatV | dative verb | <i>gave a dog to a man</i> |
| SV | sentential verb | <i>said that a dog barked</i> |

The revised grammar for VP will now look like this:

- (56) $\begin{array}{l} \text{VP} \rightarrow \text{DATV NP PP} \\ \text{VP} \rightarrow \text{TV NP} \\ \text{VP} \rightarrow \text{SV S} \\ \text{VP} \rightarrow \text{IV} \end{array}$

```

DATV → 'gave' | 'donated' | 'presented'
TV → 'saw' | 'kissed' | 'hit' | 'sang'
SV → 'said' | 'knew' | 'alleged'
IV → 'barked' | 'disappeared' | 'elapsed' | 'sang'

```

Notice that according to (33), a given lexical item can belong to more than one subcategory. For example, *sang* can occur both with and without a following NP complement.

Scaling Up

So far, we have only considered "toy grammars," small grammars that illustrate the key aspects of parsing. But there is an obvious question as to whether the approach can be scaled up to cover large corpora of natural languages. How hard would it be to construct such a set of productions by hand? In general, the answer is: *very hard*. Even if we allow ourselves to use various formal devices that give much more succinct representations of grammar productions (some of which will be discussed in Chapter 9), it is still extremely difficult to keep control of the complex interactions between the many productions required to cover the major constructions of a language. In other words, it is hard to modularize grammars so that one portion can be developed independently of the other parts. This in turn means that it is difficult to distribute the task of grammar writing across a team of linguists. Another difficulty is that as the grammar expands to cover a wider and wider range of constructions, there is a corresponding increase in the number of analyses which are admitted for any one sentence. In other words, ambiguity increases with coverage.

Despite these problems, some large collaborative projects have achieved interesting and impressive results in developing rule-based grammars for several languages. Examples are the Lexical Functional Grammar (LFG) Pargram project (<http://www2.parc.com/istl/groups/nltt/pargram/>), the Head-Driven Phrase Structure Grammar (HPSG) LinGO Matrix framework (<http://www.delph-in.net/matrix/>), and the Lexicalized Tree Adjoining Grammar XTAG Project (<http://www.cis.upenn.edu/~xtag/>).

Context Free Grammar in NLTK

In NLTK, context free grammars are defined in the `nltk.grammar` module. In Figure 8.1 we define a grammar and use it to parse a simple sentence. You will learn more about parsing in the next section.

```

grammar = nltk.parse_cfg("""
S -> NP VP
VP -> V NP | V NP PP
V -> "saw" | "ate"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "dog" | "cat" | "cookie" | "park"
PP -> P NP
P -> "in" | "on" | "by" | "with"
""")

>>> sent = "Mary saw Bob".split()
>>> rd_parser = nltk.RecursiveDescentParser(grammar)
>>> for p in rd_parser.nbest_parse(sent):
...     print p
(S (NP Mary) (VP (V saw) (NP Bob)))

```

Figure 8.1 (cfg.py): Figure 8.1: Context Free Grammars in NLTK

8.4 Parsing With Context Free Grammar

A **parser** processes input sentences according to the productions of a grammar, and builds one or more constituent structures that conform to the grammar. A grammar is a declarative specification of well-formedness — it is actually just a string, not a program. A parser is a procedural interpretation of the grammar. It searches through the space of trees licensed by a grammar to find one that has the required sentence along its fringe.

A parser permits a grammar to be evaluated against a collection of test sentences, helping linguists to discover mistakes in their grammatical analysis. A parser can serve as a model of psycholinguistic processing, helping to explain the difficulties that humans have with processing certain syntactic constructions. Many natural language applications involve parsing at some point; for example, we would expect the natural language questions submitted to a question-answering system to undergo parsing as an initial step.

In this section we see two simple parsing algorithms, a top-down method called recursive descent parsing, and a bottom-up method called shift-reduce parsing.

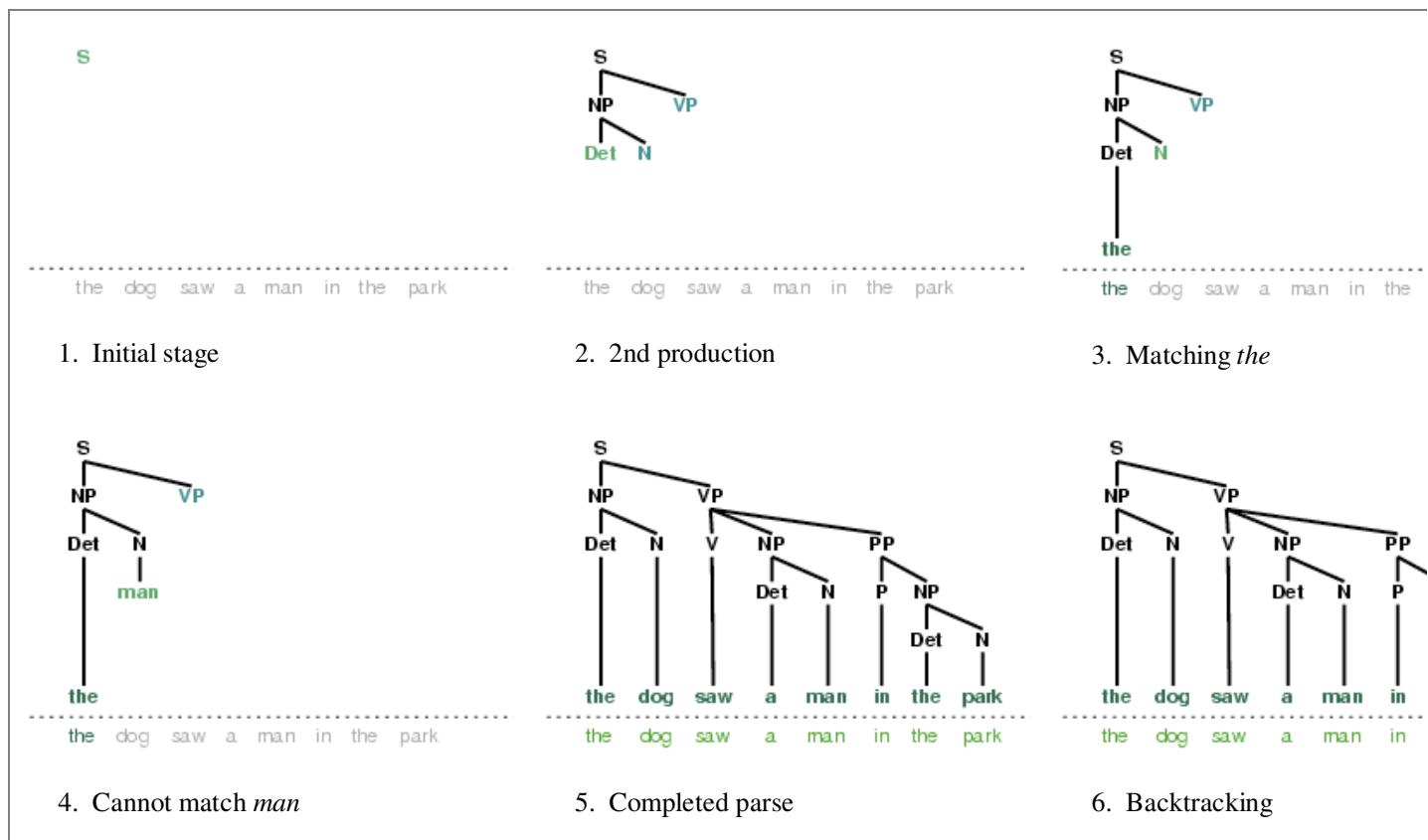
Recursive Descent Parsing

The simplest kind of parser interprets a grammar as a specification of how to break a high-level goal into several lower-level subgoals. The top-level goal is to find an S. The $S \rightarrow NP\ VP$ production permits the parser to replace this goal with two subgoals: find an NP, then find a VP. Each of these subgoals can be replaced in turn by sub-sub-goals, using productions that have NP and VP on their left-hand side. Eventually, this expansion process leads to subgoals such as: find the word *telescope*. Such subgoals can be directly compared against the input string, and succeed if the next word is matched. If there is no match the parser must back up and try a different alternative.

The recursive descent parser builds a parse tree during the above process. With the initial goal (find an S), the S root node is created. As the above process recursively expands its goals using the productions of the grammar, the parse tree is extended downwards (hence the name *recursive descent*). We can see this in action using the parser demonstration `nltk.draw.rdparsert.demo()`. Six stages of the execution of this parser are shown in [Table 8.3](#).

Table 8.3:

Six Stages of a Recursive Descent Parser



During this process, the parser is often forced to choose between several possible productions. For example, in going from step 3 to step 4, it tries to find productions with N on the left-hand side. The first of these is $N \rightarrow man$. When this does not work it backtracks, and tries other N productions in order, under it gets to $N \rightarrow dog$, which matches the next word in the input sentence. Much later, as shown in step 5, it finds a complete parse. This is a tree that covers the entire sentence, without any dangling edges. Once a parse has been found, we can get the parser to look for additional parses. Again it will backtrack and explore other choices of production in case any of them result in a parse.

NLTK provides a recursive descent parser:

```
>>> rd_parser = nltk.RecursiveDescentParser(grammar)
>>> sent = 'Mary saw a dog'.split()
>>> for t in rd_parser.nbest_parse(sent):
```

```
...     print t
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

Note

`RecursiveDescentParser()` takes an optional parameter `trace`. If `trace` is greater than zero, then the parser will report the steps that it takes as it parses a text.

Recursive descent parsing has three key shortcomings. First, left-recursive productions like $NP \rightarrow NP\ PP$ send it into an infinite loop. Second, the parser wastes a lot of time considering words and structures that do not correspond to the input sentence. Third, the backtracking process may discard parsed constituents that will need to be rebuilt again later. For example, backtracking over $VP \rightarrow V\ NP$ will discard the subtree created for the NP . If the parser then proceeds with $VP \rightarrow V\ NP\ PP$, then the NP subtree must be created all over again.

Recursive descent parsing is a kind of **top-down parsing**. Top-down parsers use a grammar to *predict* what the input will be, before inspecting the input! However, since the input is available to the parser all along, it would be more sensible to consider the input sentence from the very beginning. This approach is called **bottom-up parsing**, and we will see an example in the next section.

Shift-Reduce Parsing

A simple kind of bottom-up parser is the **shift-reduce parser**. In common with all bottom-up parsers, a shift-reduce parser tries to find sequences of words and phrases that correspond to the *right hand side* of a grammar production, and replace them with the *left-hand side*, until the whole sentence is reduced to an S .

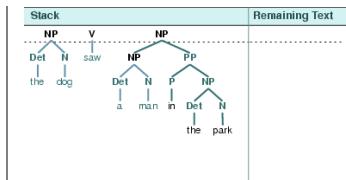
The shift-reduce parser repeatedly pushes the next input word onto a stack ([Section 6.1](#)); this is the **shift** operation. If the top n items on the stack match the n items on the right hand side of some production, then they are all popped off the stack, and the item on the left-hand side of the production is pushed on the stack. This replacement of the top n items with a single item is the **reduce** operation. (This reduce operation may only be applied to the top of the stack; reducing items lower in the stack must be done before later items are pushed onto the stack.) The parser finishes when all the input is consumed and there is only one item remaining on the stack, a parse tree with an S node as its root.

The shift-reduce parser builds a parse tree during the above process. If the top of stack holds the word *dog*, and if the grammar has a production $N \rightarrow dog$, then the reduce operation causes the word to be replaced with the parse tree for this production. For convenience we will represent this tree as $N(dog)$. At a later stage, if the top of the stack holds two items $Det(the)$ $N(dog)$ and if the grammar has a production $NP \rightarrow DET\ N$ then the reduce operation causes these two items to be replaced with $NP(Det(the), N(dog))$. This process continues until a parse tree for the entire sentence has been constructed. We can see this in action using the parser demonstration `nltk.draw.srparser.demo()`. Six stages of the execution of this parser are shown in [Figure 8.4](#).

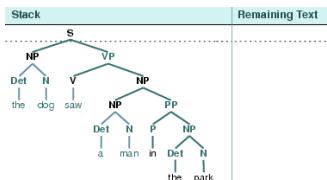
Table 8.4:

Six Stages of a Shift-Reduce Parser

| Stack | Remaining Text | Stack | Remaining Text |
|------------------------------|-------------------------------|------------------------------------|---------------------------|
| | the dog saw a man in the park | the | dog saw a man in the park |
| 1. Initial State | | 2. After one shift | |
| Stack | Remaining Text | Stack | Remaining Text |
| Det N the dog | saw a man in the park | NP V NP in | the park |
| 3. After reduce shift reduce | | 4. After recognizing the second NP | |



5. Complex NP



6. Final Step

NLTK provides `ShiftReduceParser()`, a simple implementation of a shift-reduce parser. This parser does not implement any backtracking, so it is not guaranteed to find a parse for a text, even if one exists. Furthermore, it will only find at most one parse, even if more parses exist. We can provide an optional `trace` parameter that controls how verbosely the parser reports the steps that it takes as it parses a text:

```
>>> sr_parse = nltk.ShiftReduceParser(grammar)
>>> sent = 'Mary saw a dog'.split()
>>> print sr_parse.parse(sent)
(S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

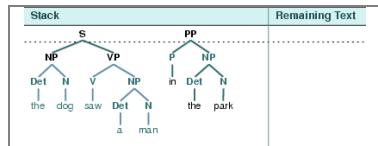
Note

Your Turn: Run the above parser in tracing mode to see the sequence of shift and reduce operations, using `sr_parse = nltk.ShiftReduceParser(grammar, trace=2)`

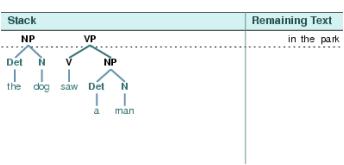
Shift-reduce parsers have a number of problems. A shift-reduce parser may fail to parse the sentence, even though the sentence is well-formed according to the grammar. In such cases, there are no remaining input words to shift, and there is no way to reduce the remaining items on the stack, as exemplified in [Table 8.51](#). The parser entered this blind alley at an earlier stage shown in [Table 8.52](#), when it reduced instead of shifted. This situation is called a **shift-reduce conflict**. At another possible stage of processing shown in [Table 8.53](#), the parser must choose between two possible reductions, both matching the top items on the stack: $VP \rightarrow VP\ NP\ PP$ or $NP \rightarrow NP\ PP$. This situation is called a **reduce-reduce conflict**.

Table 8.5:

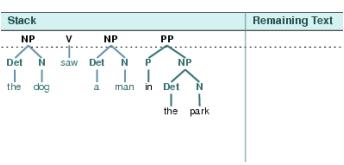
Conflict in Shift-Reduce Parsing



1. Dead end



2. Shift-reduce conflict



3. Reduce-reduce conflict

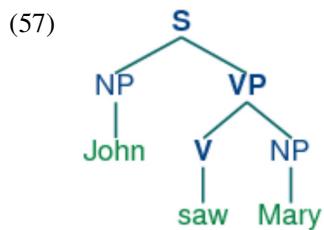
Shift-reduce parsers may implement policies for resolving such conflicts. For example, they may address shift-reduce conflicts by shifting only when no reductions are possible, and they may address reduce-reduce conflicts by favoring the reduction operation that removes the most items from the stack. No such policies are failsafe however.

The advantages of shift-reduce parsers over recursive descent parsers is that they only build structure that corresponds to the words in the input. Furthermore, they only build each sub-structure once, e.g. $\text{NP}(\text{Det}(\text{the}), \text{N}(\text{man}))$ is only built and pushed onto the stack a single time, regardless of whether it will later be used by the $\text{VP} \rightarrow \text{V NP PP}$ reduction or the $\text{NP} \rightarrow \text{NP PP}$ reduction.

The Left-Corner Parser

One of the problems with the recursive descent parser is that it can get into an infinite loop. This is because it applies the grammar productions blindly, without considering the actual input sentence. A left-corner parser is a hybrid between the bottom-up and top-down approaches we have seen.

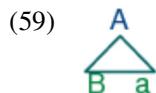
Grammar (26) allows us to produce the following parse of *John saw Mary*:



Recall that the grammar in (26) has the following productions for expanding NP:

- (58)
- $\text{NP} \rightarrow \text{DT NOM}$
 - $\text{NP} \rightarrow \text{DT NOM PP}$
 - $\text{NP} \rightarrow \text{PROPN}$

Suppose we ask you to first look at tree (57), and then decide which of the NP productions you'd want a recursive descent parser to apply first — obviously, (58c) is the right choice! How do you know that it would be pointless to apply (58a) or (58b) instead? Because neither of these productions will derive a string whose first word is *John*. That is, we can easily tell that in a successful parse of *John saw Mary*, the parser has to expand NP in such a way that NP derives the string *John* α . More generally, we say that a category B is a **left-corner** of a tree rooted in A if $A \Rightarrow^* B \alpha$.



A **left-corner parser** is a top-down parser with bottom-up filtering. Unlike an ordinary recursive descent parser, it does not get trapped in left recursive productions. Before starting its work, a left-corner parser preprocesses the context-free grammar to build a table where each row contains two cells, the first holding a non-terminal, and the second holding the collection of possible left corners of that non-terminal. Table 8.6 illustrates this for the grammar from (26).

Table 8.6:

Left-Corners in (26)

| Category | Left-Corners (pre-terminals) |
|----------|------------------------------|
| S | NP |
| NP | Det, PropN |
| VP | V |
| PP | P |

Each time a production is considered by the parser, it checks that the next input word is compatible with at least one of the

pre-terminal categories in the left-corner table.

[TODO: explain how this effects the action of the parser, and why this solves the problem.]

Summary

A context-free phrase structure grammar (CFG) is a formal model for describing whether a given string can be assigned a particular constituent structure. Given a set of syntactic categories, the CFG uses a set of productions to say how a phrase of some category A can be analyzed into a sequence of smaller parts $a_1 \dots a_n$. But a grammar is a static description of a set of strings; it does not tell us what sequence of steps we need to take to build a constituent structure for a string. For this, we need to use a parsing algorithm. We presented two such algorithms: Top-Down Recursive Descent Bottom-Up Shift-Reduce (8.4). As we pointed out, both parsing approaches suffer from important shortcomings. The Recursive Descent parser cannot handle left-recursive productions (e.g., productions such as $NP \rightarrow NP\ PP$), and blindly expands categories top-down without checking whether they are compatible with the input string. The Shift-Reduce parser is not guaranteed to find a valid parse for the input even if one exists, and builds substructure without checking whether it is globally consistent with the grammar. As we will describe further below, the Recursive Descent parser is also inefficient in its search for parses.

8.5 Chart Parsing

The simple parsers discussed above suffer from limitations in both completeness and efficiency. In order to remedy these, we will apply the algorithm design technique of dynamic programming to the parsing problem. As we saw in [Section 6.4](#), dynamic programming stores intermediate results and re-uses them when appropriate, achieving significant efficiency gains. This technique can be applied to syntactic parsing, allowing us to store partial solutions to the parsing task and then look them up as necessary in order to efficiently arrive at a complete solution. This approach to parsing is known as **chart parsing**, and is the focus of this section.

Well-Formed Substring Tables

Dynamic programming allows us to build the PP in *my pyjamas* just once. The first time we build it we save it in a table, then we look it up when we need to use it as a subconstituent of either the object NP or the higher VP . This table is known as a **well-formed substring table** (or WFST for short). We will show how to construct the WFST bottom-up so as to systematically record what syntactic constituents have been found.

Let's set our input to be the sentence in (3b). It is helpful to think of the input as being indexed like a Python list. We have illustrated this in [Figure 8.2](#).

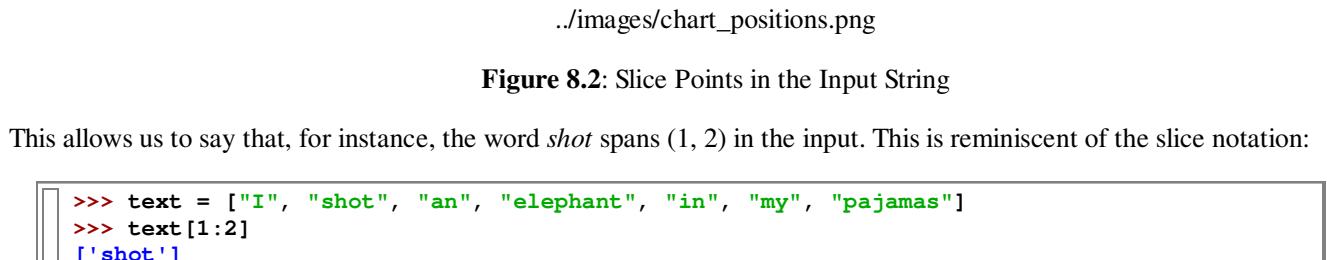
./images/chart_positions.png

Figure 8.2: Slice Points in the Input String

This allows us to say that, for instance, the word *shot* spans (1, 2) in the input. This is reminiscent of the slice notation:

```
>>> text = ["I", "shot", "an", "elephant", "in", "my", "pajamas"]
>>> text[1:2]
['shot']
```

In a WFST, we record the position of the words by filling in cells in a triangular matrix: the vertical axis will denote the start position of a substring, while the horizontal axis will denote the end position (thus *shot* will appear in the cell with coordinates (1, 2)). To simplify this presentation, we will assume each word has a unique lexical category, and we will store this (not the word) in the matrix. So cell (1, 2) will contain the entry v . More generally, if our input string is $a_1 a_2 \dots a_n$, and our grammar contains a production of the form $A \rightarrow a_i$, then we add A to the cell $(i-1, i)$.

So, for every word in `text`, we can look up in our grammar what category it belongs to.

```
>>> groucho_grammar.productions(rhs=text[1])
[v -> 'shot']
```

For our WFST, we create an $(n-1) \times (n-1)$ matrix as a list of lists in Python, and initialize it with the lexical categories of each token, in the `init_wfst()` function in [Figure 8.3](#). We also define a utility function `display()` to pretty-print the WFST for us.

As expected, there is a V in cell (1, 2).

```

def init_wfst(tokens, grammar):
    numtokens = len(tokens)
    wfst = [[None for i in range(numtokens+1)] for j in range(numtokens+1)]
    for i in range(numtokens):
        productions = grammar.productions(rhs=tokens[i])
        wfst[i][i+1] = productions[0].lhs()
    return wfst

def complete_wfst(wfst, tokens, grammar, trace=False):
    index = dict((p.rhs(), p.lhs()) for p in grammar.productions())
    numtokens = len(tokens)
    for span in range(2, numtokens+1):
        for start in range(numtokens+1-span):
            end = start + span
            for mid in range(start+1, end):
                nt1, nt2 = wfst[start][mid], wfst[mid][end]
                if nt1 and nt2 and (nt1, nt2) in index:
                    wfst[start][end] = index[(nt1, nt2)]
                    if trace:
                        print "[%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s]" % \
                            (start, nt1, mid, nt2, end, start, index[(nt1, nt2)], end)
    return wfst

def display(wfst, tokens):
    print '\nWFST ' + ' '.join(['%4d' % i for i in range(1, len(wfst))])
    for i in range(len(wfst)-1):
        print "%d " % i,
        for j in range(1, len(wfst)):
            print "%-4s" % (wfst[i][j] or '.'),
        print

>>> tokens = "I shot an elephant in my pajamas".split()
>>> wfst0 = init_wfst(tokens, groucho_grammar)
>>> display(wfst0, tokens)
WFST 1 2 3 4 5 6 7
0 NP . . . . .
1 . V . . . .
2 . . Det . . .
3 . . . N . .
4 . . . . P .
5 . . . . . Det .
6 . . . . . .
>>> wfst1 = complete_wfst(wfst0, tokens, groucho_grammar)
>>> display(wfst1, tokens)
WFST 1 2 3 4 5 6 7
0 NP . . S . .
1 . V . VP . .
2 . . Det NP . .
3 . . . N . .
4 . . . . P .
5 . . . . Det NP .
6 . . . . .

```

[Figure 8.3 \(wfst.py\)](#): Figure 8.3: Acceptor Using Well-Formed Substring Table (based on CYK algorithm)

Returning to our tabular representation, given that we have DET in cell (2, 3) for the word *an*, and N in cell (3, 4) for the word *elephant*, what should we put into cell (2, 4) for *an elephant*? We need to find a production of the form $A \rightarrow \text{DET } N$. Consulting the grammar, we know that we can enter NP in cell (0,2).

More generally, we can enter A in (i, j) if there is a production $A \rightarrow B C$, and we find nonterminal B in (i, k) and C in (k, j) . The program in [Figure 8.3](#) uses this rule to complete the WFST. By setting `trace` to `True` when calling the function `complete_wfst()`, we see tracing output that shows the WFST being constructed:

```

>>> wfst1 = complete_wfst(wfst0, tokens, groucho_grammar, trace=True)
[2] Det [3] N [4] ==> [2] NP [4]
[5] Det [6] N [7] ==> [5] NP [7]
[1] V [2] NP [4] ==> [1] VP [4]
[4] P [5] NP [7] ==> [4] PP [7]
[0] NP [1] VP [4] ==> [0] S [4]

```

| | | | | | | | | |
|-----|----|-----|----|-----|---------------|-----|----|-----|
| [1] | VP | [4] | PP | [7] | \Rightarrow | [1] | VP | [7] |
| [0] | NP | [1] | VP | [7] | \Rightarrow | [0] | S | [7] |

For example, this says that since we found `Det` at `wfst[0][1]` and `N` at `wfst[1][2]`, we can add `NP` to `wfst[0][2]`.

Note

To help us easily retrieve productions by their right hand sides, we create an index for the grammar. This is an example of a space-time trade-off: we do a reverse lookup on the grammar, instead of having to check through entire list of productions each time we want to look up via the right hand side.

We conclude that there is a parse for the whole input string once we have constructed an `S` node in cell $(0, 7)$, showing that we have found a sentence that covers the whole input.

Notice that we have not used any built-in parsing functions here. We've implemented a complete, primitive chart parser from the ground up!

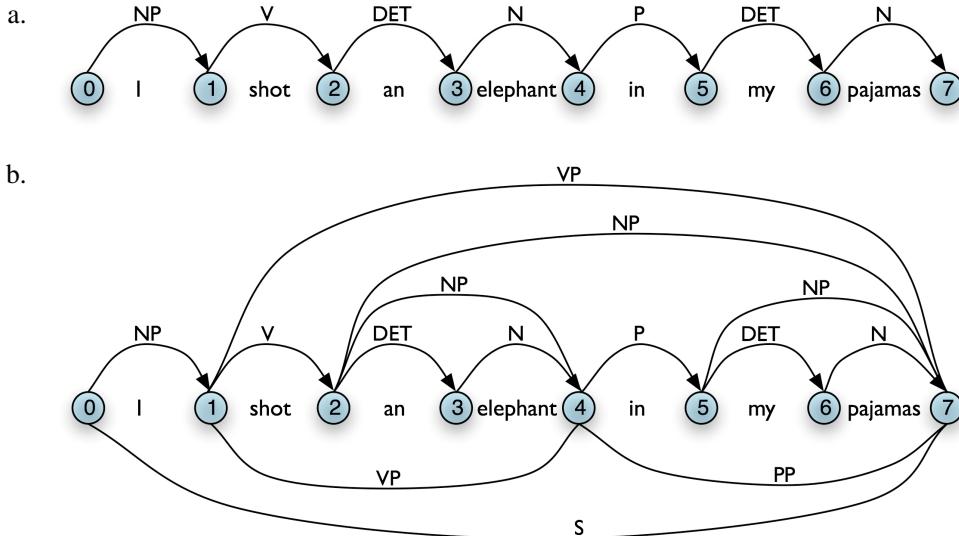
WFST's have several shortcomings. First, as you can see, the WFST is not itself a parse tree, so the technique is strictly speaking **recognizing** that a sentence is admitted by a grammar, rather than parsing it. Second, it requires every non-lexical grammar production to be *binary* (see the discussion of normal forms in [Section 9.4](#)). Although it is possible to convert an arbitrary CFG into this form, we would prefer to use an approach without such a requirement. Third, as a bottom-up approach it is potentially wasteful, being able to propose constituents in locations that would not be licensed by the grammar.

Finally, the WFST did not represent the structural ambiguity in the sentence (i.e. the two verb phrase readings). The `VP` in cell $(2, 8)$ was actually entered twice, once for a `V NP` reading, and once for a `VP PP` reading. These are different hypotheses, and the second overwrote the first (as it happens this didn't matter since the left hand side was the same.) Chart parsers use a slightly richer data structure to solve these problems, known as the **active chart**.

Active Charts

The same information can be represented in a directed acyclic graph, as shown in [\(60a\)](#) for the initialized WFST and [\(60b\)](#) for the completed WFST.

(60)



In general, a chart parser hypothesizes constituents (i.e. adds edges) based on the grammar, the tokens, and the constituents already found. Any constituent that is compatible with the current knowledge can be hypothesized; even though many of these hypothetical constituents will never be used in the final result. A WFST just records these hypotheses.

All of the edges that we've seen so far represent complete constituents. However, it is helpful to record *incomplete* constituents, to document the work already done by the parser. For example, when a top-down parser processes $VP \rightarrow V NP PP$, it may find V and NP but not the PP . This work can be reused when processing $VP \rightarrow V NP$. Thus, we will record the hypothesis that "the V constituent *likes* is the beginning of a VP ."

We can do this by adding a **dot** to the edge's right hand side. Material to the left of the dot records what has been found so far; material to the right of the dot specifies what still needs to be found in order to complete the constituent. For example, the edge in (38) records the hypothesis that "a VP starts with the V *likes*, but still needs an NP to become complete":

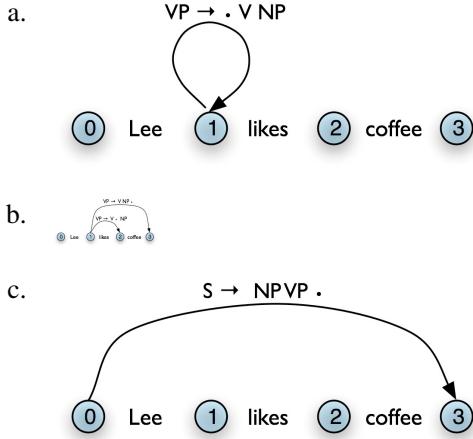
(61) 

These **dotted edges** are used to record all of the hypotheses that a chart parser makes about constituents in a sentence.

Types of Edge

Let's take stock. An edge $[VP \rightarrow \bullet V NP PP, (i, i)]$ records the hypothesis that a VP begins at location i , and that we anticipate finding a sequence $V NP PP$ starting here. This is known as a **self-loop edge**; see (62a). An edge $[VP \rightarrow V \bullet NP PP, (i, j)]$ records the fact that we have discovered a V spanning (i, j) , and hypothesize a following $NP PP$ sequence to complete a VP beginning at i . This is known as an **incomplete edge**; see (62b). An edge $[VP \rightarrow V NP PP \bullet, (i, k)]$ records the discovery that a VP consisting of the sequence $V NP PP$ has been discovered for the span (i, j) . This is known as a **complete edge**; see (62c). If a complete edge spans the entire sentence, and has the grammar's start symbol as its left-hand side, then the edge is called a **parse edge**, and it encodes one or more parse trees for the sentence; see (62c).

(62)



The Chart Parser

To parse a sentence, a chart parser first creates an empty chart spanning the sentence. It then finds edges that are licensed by its knowledge about the sentence, and adds them to the chart one at a time until one or more parse edges are found. The edges that it adds can be licensed in one of three ways:

1. The *input* can license an edge: each word w_i in the input licenses the complete edge $[w_i \rightarrow \bullet, (i, i+1)]$.
2. The *grammar* can license an edge: each grammar production $A \rightarrow \alpha$ licenses the self-loop edge $[A \rightarrow \bullet \alpha, (i, i)]$ for every $i, 0 \leq i < n$.
3. The *current chart contents* can license an edge: a suitable pair of existing edges triggers the addition of a new edge.

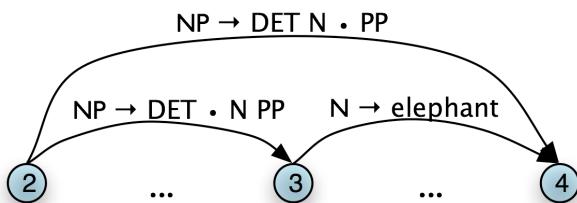
Chart parsers use a set of **rules** to heuristically decide when an edge should be added to a chart. This set of rules, along with a specification of when they should be applied, forms a **strategy**.

The Fundamental Rule

One rule is particularly important, since it is used by every chart parser: the **Fundamental Rule**. This rule is used to combine an incomplete edge that's expecting a nonterminal B with a following, complete edge whose left hand side is B . The rule is defined and illustrated in (40). We will use α , β , and γ to denote (possibly empty) sequences of terminals or non-terminals.

(63)

Fundamental Rule If the chart contains the edges $[A \rightarrow \alpha \bullet B \beta, (i, j)]$ and $[B \rightarrow \gamma \bullet, (j, k)]$ then add a new edge $[A \rightarrow \alpha B \bullet \beta, (i, k)]$.



In the new edge, the dot has moved one place to the right. Its span is the combined span of the original edges. Note that in adding this new edge we do not remove the other two, because they might be used again.

Bottom-Up Parsing

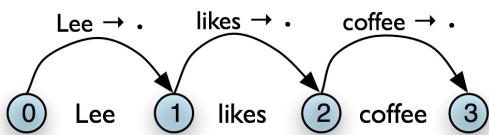
As we saw with the shift-reduce parser in 8.4, bottom-up parsing starts from the input string, and tries to find sequences of words and phrases that correspond to the *right hand side* of a grammar production. The parser then replaces these with the left-hand side of the production, until the whole sentence is reduced to an S. Bottom-up chart parsing is an extension of this approach in which hypotheses about structure are recorded as edges on a chart. In terms of our earlier terminology, bottom-up chart parsing can be seen as a parsing strategy; in other words, bottom-up is a particular choice of heuristics for adding new edges to a chart.

The general procedure for chart parsing is inductive: we start with a base case, and then show how we can move from a given state of the chart to a new state. Since we are working bottom-up, the base case for our induction will be determined by the words in the input string, so we add new edges for each word. Now, for the induction step, suppose the chart contains an edge labeled with constituent A . Since we are working bottom-up, we want to build constituents that can have an A as a daughter. In other words, we are going to look for productions of the form $B \rightarrow A \beta$ and use these to label new edges.

Let's look at the procedure a bit more formally. To create a bottom-up chart parser, we add to the Fundamental Rule two new rules: the **Bottom-Up Initialization Rule**; and the **Bottom-Up Predict Rule**. The Bottom-Up Initialization Rule says to add all edges licensed by the input.

(64)

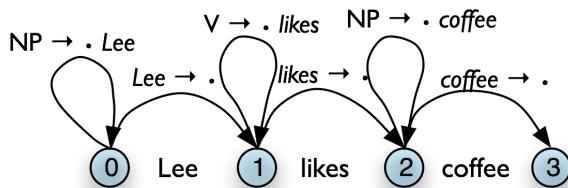
Bottom-Up Initialization Rule For every word w_i add the edge $[w_i \rightarrow \bullet, (i, i+1)]$



Next, suppose the chart contains a complete edge e whose left hand category is A . Then the Bottom-Up Predict Rule requires the parser to add a self-loop edge at the left boundary of e for each grammar production whose right hand side begins with category A .

(65)

Bottom-Up Predict Rule For each complete edge $[A \rightarrow \alpha \bullet, (i, j)]$ and each production $B \rightarrow A \beta$, add the self-loop edge $[B \rightarrow \bullet A \beta, (i, i)]$



The next step is to use the Fundamental Rule to add edges like $[NP \rightarrow Lee \bullet, (0, 1)]$, where we have "moved the dot" one position to the right. After this, we will now be able to add new self-loop edges such as $[S \rightarrow \bullet NP VP, (0, 0)]$ and $[VP \rightarrow \bullet VP NP, (1, 1)]$, and use these to build more complete edges.

Using these three rules, we can parse a sentence as shown in (43).

(66)

Bottom-Up Strategy

Create an empty chart spanning the sentence.

Apply the Bottom-Up Initialization Rule to each word.

Until no more edges are added:

 Apply the Bottom-Up Predict Rule everywhere it applies.

 Apply the Fundamental Rule everywhere it applies.

Return all of the parse trees corresponding to the parse edges in the chart.

Note

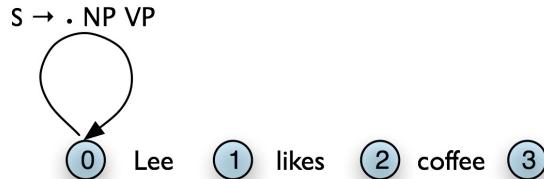
Your Turn: NLTK provides a useful interactive tool for visualizing the operation of a chart parser: `nltk.draw.chart.demo()`. The tool comes with a pre-defined input string and grammar, but both of these can be readily modified with options inside the *Edit* menu.

Top-Down Parsing

Top-down chart parsing works in a similar way to the recursive descent parser, in that it starts off with the top-level goal of finding an S . This goal is broken down into the subgoals of trying to find constituents such as NP and VP predicted by the grammar. To create a top-down chart parser, we use the Fundamental Rule as before plus three other rules: the **Top-Down Initialization Rule**, the **Top-Down Expand Rule**, and the **Top-Down Match Rule**. The Top-Down Initialization Rule in (44) captures the fact that the root of any parse must be the start symbol S .

(67)

Top-Down Initialization Rule For each production $s \rightarrow \alpha$ add the self-loop edge $[s \rightarrow \bullet\alpha, (0, 0)]$



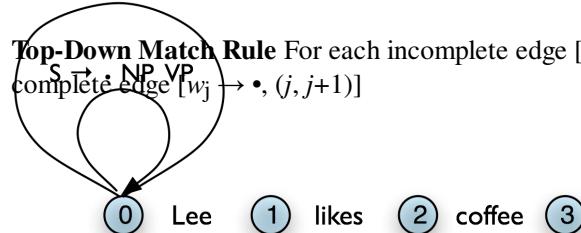
In our running example, we are predicting that we will be able to find an NP and a VP starting at 0, but have not yet satisfied these subgoals. In order to find an NP we need to invoke a production that has NP on its left hand side. This work is done by the Top-Down Expand Rule (45). This tells us that if our chart contains an incomplete edge whose dot is followed by a nonterminal B , then the parser should add any self-loop edges licensed by the grammar whose left-hand side is B .

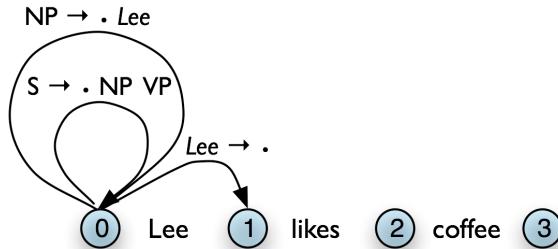
(68)

Top-Down Expand Rule If the production $A \rightarrow \alpha \bullet B \beta$ is in the grammar to B , then add a self-loop edge for $A \rightarrow \bullet B \beta$ to the chart. If the chart contains an incomplete edge whose dot is followed by a terminal w_j , then the parser should add an edge if the terminal corresponds to the current input symbol.

(69)

Top-Down Match Rule For each incomplete edge $[A \rightarrow \alpha \bullet w_j \beta, (i, j)]$, where w_j is the j^{th} word of the input, add a new complete edge $[w_j \rightarrow \bullet, (j, j+1)]$





Here we see our example chart after applying the Top-Down Match rule. After this, we can apply the fundamental rule to add the edge [$NP \rightarrow Lee \bullet, (0, 1)$].

Using these four rules, we can parse a sentence top-down as shown in (47).

(70) **Top-Down Strategy**

```
Create an empty chart spanning the sentence.
Apply the Top-Down Initialization Rule (at node 0)
Until no more edges are added:
    Apply the Top-Down Expand Rule everywhere it applies.
    Apply the Top-Down Match Rule everywhere it applies.
    Apply the Fundamental Rule everywhere it applies.
Return all of the parse trees corresponding to the parse edges in
the chart.
```

The Earley Algorithm

The Earley algorithm [Earley, 1970] is a parsing strategy that resembles the Top-Down Strategy, but deals more efficiently with matching against the input string. Table 8.7 shows the correspondence between the parsing rules introduced above and the rules used by the Earley algorithm.

Table 8.7:

Terminology for rules in the Earley algorithm

| Top-Down/Bottom-Up | Earley |
|-------------------------------|----------------------|
| Top-Down Initialization Rule | Top-Down Expand Rule |
| Top-Down/Bottom-Up Match Rule | Predictor Rule |
| Fundamental Rule | Scanner Rule |
| | Completer Rule |

Let's look in more detail at the Scanner Rule. Suppose the chart contains an incomplete edge with a lexical category P immediately after the dot, the next word in the input is w , P is a part-of-speech label for w . Then the Scanner Rule admits a new complete edge in which P dominates w . More precisely:

(71) **Scanner Rule** For each incomplete edge

[$A \rightarrow a \bullet P\beta, (i, j)$] where w_j is the j^{th} word of the input and P is a valid part of speech for w_j , add the new complete edges [$P \rightarrow w_j \bullet, (j, j+1)$] and [$w_j \rightarrow \bullet, (j, j+1)$]

To illustrate, suppose the input is of the form *I saw ...*, and the chart already contains the edge [$VP \rightarrow \bullet v \dots, (1, 1)$]. Then the Scanner Rule will add to the chart the edges [$v \rightarrow 'saw', (1, 2)$] and [$'saw' \rightarrow \bullet, (1, 2)$]. So in effect the Scanner Rule packages up a sequence of three rule applications: the Bottom-Up Initialization Rule for [$w \rightarrow \bullet, (j, j+1)$], the Top-Down Expand Rule for [$P \rightarrow \bullet w_j, (j, j)$], and the Fundamental Rule for [$P \rightarrow w_j \bullet, (j, j+1)$]]. This is considerably more efficient than the Top-Down Strategy, that adds a new edge of the form [$P \rightarrow \bullet w, (j, j)$] for every lexical rule $P \rightarrow w$, regardless of whether w can be found in the input. By contrast with Bottom-Up Initialization, however, the Earley algorithm proceeds strictly left-to-right through the input, applying all applicable rules at that point in the chart, and never backtracking.

Note

Your Turn: The NLTK chart parser demo, `nltk.draw.chart.demo()`, allows the option of parsing according to the Earley algorithm.

Chart Parsing in NLTK

NLTK defines a simple yet flexible chart parser, `ChartParser`. A new chart parser is constructed from a grammar and a strategy. The strategy is applied until no new edges are added to the chart. NLTK defines two ready-made strategies: `TD_STRATEGY`, a basic top-down strategy; and `BU_STRATEGY`, a basic bottom-up strategy. When constructing a chart parser, you can use either of these strategies, or create your own. We've already seen how to define a chart parser in [section 8.1](#). This time we'll specify a strategy and turn on tracing:

```
>>> sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
>>> parser = nltk.ChartParser(groucho_grammar, nltk.parse.BU_STRATEGY)
>>> trees = parser.nbest_parse(sent, trace=2)
```

8.6 Summary (notes)

- Sentences have internal organization, or constituent structure, that can be represented using a tree; notable features of constituent structure are: recursion, heads, complements, modifiers
- A grammar is a compact characterization of a potentially infinite set of sentences; we say that a tree is well-formed according to a grammar, or that a grammar licenses a tree.
- Syntactic ambiguity arises when one sentence has more than one syntactic structure (e.g. prepositional phrase attachment ambiguity).
- A parser is a procedure for finding one or more trees corresponding to a grammatically well-formed sentence.
- A simple top-down parser is the recursive descent parser (summary, problems)
- A simple bottom-up parser is the shift-reduce parser (summary, problems)
- It is difficult to develop a broad-coverage grammar...

8.7 Further Reading

For more examples of parsing with NLTK, please see the guide at <http://nltk.org/doc/guides/parse.html>.

There are many introductory books on syntax. [O'Grady1989LI] is a general introduction to linguistics, while [Radford, 1988] provides a gentle introduction to transformational grammar, and can be recommended for its coverage of transformational approaches to unbounded dependency constructions. The most widely used term in linguistics for formal grammar is *generative grammar*, though it has nothing to do with generation.

[Burton-Roberts, 1997] is very practically oriented textbook on how to analyze constituency in English, with extensive exemplification and exercises. [Huddleston & Pullum, 2002] provides an up-to-date and comprehensive analysis of syntactic phenomena in English.

Chapter 12 of [Jurafsky & Martin, 2008] covers formal grammars of English; Sections 13.1-3 cover simple parsing algorithms and techniques for dealing with ambiguity; Chapter 16 covers the Chomsky hierarchy and the formal complexity of natural language.

- LALR(1), LR(k)
- Marcus parser
- Lexical Functional Grammar (LFG)
 - [Pargram project](#)
 - [LFG Portal](#)
- Head-Driven Phrase Structure Grammar (HPSG) [LinGO Matrix framework](#)
- Lexicalized Tree Adjoining Grammar [XTAG Project](#)

8.8 Exercises

1. ☀ Can you come up with grammatical sentences that have probably never been uttered before? (Take turns with a partner.) What does this tell you about human language?
2. ☀ Recall Strunk and White's prohibition against sentence-initial *however* used to mean "although". Do a web search for *however* used at the start of the sentence. How widely used is this construction?
3. ☀ Consider the sentence *Kim arrived or Dana left and everyone cheered*. Write down the parenthesized forms to show the relative scope of *and* and *or*. Generate tree structures corresponding to both of these interpretations.
4. ☀ The `Tree` class implements a variety of other useful methods. See the `Tree` help documentation for more details, i.e. import the `Tree` class and then type `help(Tree)`.

5. ☀ Building trees:

1. Write code to produce two trees, one for each reading of the phrase *old men and women*
2. Encode any of the trees presented in this chapter as a labeled bracketing and use `nltk.bracket_parse()` to check that it is well-formed. Now use `draw()` to display the tree.
3. As in (a) above, draw a tree for *The woman saw a man last Thursday*.
6. ☀ Write a recursive function to traverse a tree and return the depth of the tree, such that a tree with a single node would have depth zero. (Hint: the depth of a subtree is the maximum depth of its children, plus one.)
7. ☀ Analyze the A.A. Milne sentence about Piglet, by underlining all of the sentences it contains then replacing these with `S` (e.g. the first sentence becomes `S when:Ix` S`). Draw a tree structure for this "compressed" sentence. What are the main syntactic constructions used for building such a long sentence?
8. ☀ In the recursive descent parser demo, experiment with changing the sentence to be parsed by selecting *Edit Text* in the *Edit* menu.
9. ☀ Can the grammar in (24) be used to describe sentences that are more than 20 words in length?
10. ☀ Use the graphical chart-parser interface to experiment with different rule invocation strategies. Come up with your own strategy that you can execute manually using the graphical interface. Describe the steps, and report any efficiency improvements it has (e.g. in terms of the size of the resulting chart). Do these improvements depend on the structure of the grammar? What do you think of the prospects for significant performance boosts from cleverer rule invocation strategies?
11. ☀ With pen and paper, manually trace the execution of a recursive descent parser and a shift-reduce parser, for a CFG you have already seen, or one of your own devising.
12. ☀ We have seen that a chart parser adds but never removes edges from a chart. Why?
13. ● You can modify the grammar in the recursive descent parser demo by selecting *Edit Grammar* in the *Edit* menu. Change the first expansion production, namely `NP -> Det N PP`, to `NP -> NP PP`. Using the *Step* button, try to build a parse tree. What happens?
14. ● Extend the grammar in (26) with productions that expand prepositions as intransitive, transitive and requiring a PP complement. Based on these productions, use the method of the preceding exercise to draw a tree for the sentence *Lee ran away home*.
15. ● Pick some common verbs and complete the following tasks:
 1. Write a program to find those verbs in the Prepositional Phrase Attachment Corpus `nltk.corpus.ppattach`. Find any cases where the same verb exhibits two different attachments, but where the first noun, or second noun, or preposition, stay unchanged (as we saw in our discussion of syntactic ambiguity in [Section 8.2](#)).
 2. Devise CFG grammar productions to cover some of these cases.
16. ● Write a program to compare the efficiency of a top-down chart parser compared with a recursive descent parser ([Section 8.4](#)). Use the same grammar and input sentences for both. Compare their performance using the `timeit` module (Section XREF).

17. ● Compare the performance of the top-down, bottom-up, and left-corner parsers using the same grammar and three grammatical test sentences. Use `timeit` to log the amount of time each parser takes on the same sentence (Section XREF). Write a function that runs all three parsers on all three sentences, and prints a 3-by-3 grid of times, as well as row and column totals. Discuss your findings.
18. ● Read up on "garden path" sentences. How might the computational work of a parser relate to the difficulty humans have with processing these sentences? http://en.wikipedia.org/wiki/Garden_path_sentence
19. ● To compare multiple trees in a single window, we can use the `draw_trees()` method. Define some trees and try it out:

```
>>> from nltk.draw.tree import draw_trees
>>> draw_trees(tree1, tree2, tree3)
```

20. ● Using tree positions, list the subjects of the first 100 sentences in the Penn treebank; to make the results easier to view, limit the extracted subjects to subtrees whose height is 2.
21. ● Inspect the Prepositional Phrase Attachment Corpus and try to suggest some factors that influence PP attachment.
22. ● In this section we claimed that there are linguistic regularities that cannot be described simply in terms of n-grams. Consider the following sentence, particularly the position of the phrase *in his turn*. Does this illustrate a problem for an approach based on n-grams?

What was more, the in his turn somewhat youngish Nikolay Parfenovich also turned out to be the only person in the entire world to acquire a sincere liking to our "discriminated-against" public procurator.
(Dostoevsky: The Brothers Karamazov)

23. ● Write a recursive function that produces a nested bracketing for a tree, leaving out the leaf nodes, and displaying the non-terminal labels after their subtrees. So the above example about Pierre Vinken would produce: [[[NNP NNP] NP , [ADJP [CD NNS] NP JJ] ADJP] NP-SBJ MD [VB [DT NN] NP [IN [DT JJ NN] NP] PP-CLR [NNP CD] NP-TMP] VP .] S
Consecutive categories should be separated by space.
1. ● Download several electronic books from Project Gutenberg. Write a program to scan these texts for any extremely long sentences. What is the longest sentence you can find? What syntactic construction(s) are responsible for such long sentences?
2. ★ One common way of defining the subject of a sentence S in English is as *the noun phrase that is the daughter of S and the sister of VP*. Write a function that takes the tree for a sentence and returns the subtree corresponding to the subject of the sentence. What should it do if the root node of the tree passed to this function is not S, or it lacks a subject?
3. ★ Write a function that takes a grammar (such as the one defined in [Figure 8.1](#)) and returns a random sentence generated by the grammar. (Use `grammar.start()` to find the start symbol of the grammar; `grammar.productions(lhs)` to get the list of productions from the grammar that have the specified left-hand side; and `production.rhs()` to get the right-hand side of a production.)
4. ★ **Lexical Acquisition:** As we saw in [Chapter 7](#), it is possible to collapse chunks down to their chunk label. When we do this for sentences involving the word *gave*, we find patterns such as the following:

```
gave NP
gave up NP in NP
gave NP up
gave NP NP
gave NP to NP
```

1. Use this method to study the complementation patterns of a verb of interest, and write suitable grammar productions.
2. Identify some English verbs that are near-synonyms, such as the *dumped/filled/loaded* example from earlier in this chapter. Use the chunking method to study the complementation patterns of these verbs. Create a grammar to cover these cases. Can the verbs be freely substituted for each other, or are their constraints? Discuss your findings.
5. ★ **Left-corner parser:** Develop a left-corner parser based on the recursive descent parser, and inheriting from `ParseI`.

6. ★ Extend NLTK's shift-reduce parser to incorporate backtracking, so that it is guaranteed to find all parses that exist (i.e. it is **complete**).
7. ★ Modify the functions `init_wfst()` and `complete_wfst()` so that when a non-terminal symbol is added to a cell in the WFST, it includes a record of the cells from which it was derived. Implement a function that will convert a WFST in this form to a parse tree.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#). Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

9 Advanced Topics in Parsing

9.1 A Problem of Scale

Parsing builds trees over sentences, according to a phrase structure grammar. Now, all the examples we gave in [Chapter 8](#) only involved toy grammars containing a handful of productions. What happens if we try to scale up this approach to deal with realistic corpora of language? Unfortunately, as the coverage of the grammar increases and the length of the input sentences grows, the number of parse trees grows rapidly. In fact, it grows at an astronomical rate.

Let's explore this issue with the help of a simple example. The word *fish* is both a noun and a verb. We can make up the sentence *fish fish fish fish*, meaning *fish like to fish for other fish*. (Try this with *police* if you prefer something more sensible.) Here is a toy grammar for the "fish" sentences.

```
>>> grammar = nltk.parse_cfg("""
... S -> NP V NP
... NP -> NP Sbar
... Sbar -> NP V
... NP -> 'fish'
... V -> 'fish'
... """)
```

Now we can try parsing a longer sentence, *fish fish fish fish fish*, which amongst other things, means 'fish that other fish fish are in the habit of fishing fish themselves'. We use the NLTK chart parser, which is presented later on in this chapter. This sentence has two readings.

```
>>> tokens = ["fish"] * 5
>>> cp = nltk.ChartParser(grammar, nltk.parse.TD_STRATEGY)
>>> for tree in cp.nbest_parse(tokens):
...     print tree
(S (NP (NP fish) (Sbar (NP fish) (V fish))) (V fish) (NP fish))
(S (NP fish) (V fish) (NP (NP fish) (Sbar (NP fish) (V fish))))
```

As the length of this sentence goes up (3, 5, 7, ...) we get the following numbers of parse trees: 1; 2; 5; 14; 42; 132; 429; 1,430; 4,862; 16,796; 58,786; 208,012; ... (These are the Catalan numbers, which we saw in an exercise in [Section 6.4](#)). The last of these is for a sentence of length 23, the average length of sentences in the WSJ section of Penn Treebank. For a sentence of length 50 there would be over 10^{12} parses, and this is only half the length of the Piglet sentence ([Section 8.1](#)), which young children process effortlessly. No practical NLP system could construct millions of trees for a sentence and choose the appropriate one in the context. It's clear that humans don't do this either!

Note that the problem is not with our choice of example. [\[Church & Patil, 1982\]](#) point out that the syntactic ambiguity of PP attachment in sentences like (1) also grows in proportion to the Catalan numbers.

(72) Put the block in the box on the table.

So much for structural ambiguity; what about lexical ambiguity? As soon as we try to construct a broad-coverage grammar, we are forced to make lexical entries highly ambiguous for their part of speech. In a toy grammar, *a* is only a determiner, *dog* is only a noun, and *runs* is only a verb. However, in a broad-coverage grammar, *a* is also a noun (e.g. *part a*), *dog* is also a verb (meaning to follow closely), and *runs* is also a noun (e.g. *ski runs*). In fact, all words can be referred to by name: e.g. *the verb 'ate' is spelled with three letters*; in speech we do not need to supply quotation marks. Furthermore, it is possible to *verb* most nouns. Thus a parser for a broad-coverage grammar will be overwhelmed with ambiguity. Even complete gibberish will often have a reading, e.g. *the a are of I*. As [Klavans & Resnik, 1996] has pointed out, this is not word salad but a grammatical noun phrase, in which *are* is a noun meaning a hundredth of a hectare (or 100 sq m), and *a* and *I* are nouns designating coordinates, as shown in [Figure 9.1](#).

| | | | | | | | | | |
|---|--|--|--|--|--|--|--|--|--|
| a | | | | | | | | | |
| b | | | | | | | | | |
| c | | | | | | | | | |

A B C D E F G H I

Figure 9.1: The a are of I

Even though this phrase is unlikely, it is still grammatical and a broad-coverage parser should be able to construct a parse tree for it. Similarly, sentences that seem to be unambiguous, such as *John saw Mary*, turn out to have other readings we would not have anticipated (as Abney explains). This ambiguity is unavoidable, and leads to horrendous inefficiency in parsing seemingly innocuous sentences.

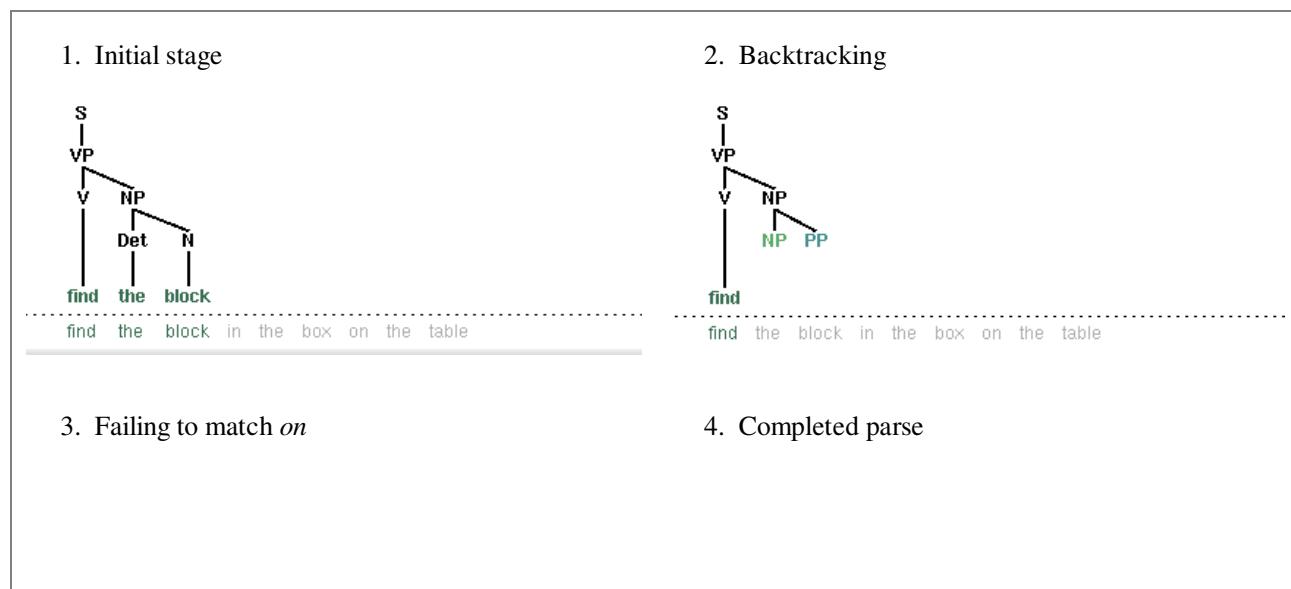
Let's look more closely at this issue of efficiency. The top-down recursive-descent parser presented in [Chapter 8](#) can be very inefficient, since it often builds and discards the same sub-structure many times over. We see this in [Figure 9.1](#), where a phrase *the block* is identified as a noun phrase several times, and where this information is discarded each time we backtrack.

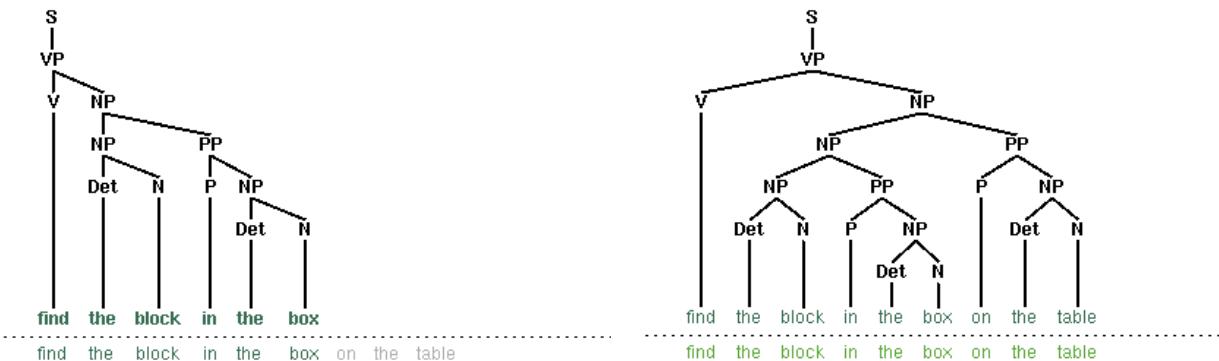
Note

You should try the recursive-descent parser demo if you haven't already:
`nltk.draw.srparser.demo()`

Table 9.1:

Backtracking and Repeated Parsing of Subtrees





In this chapter, we will present two independent methods for dealing with ambiguity. The first is *chart parsing*, which uses the algorithmic technique of dynamic programming to derive the parses of an ambiguous sentence more *efficiently*. The second is *probabilistic parsing*, which allows us to *rank* the parses of an ambiguous sentence on the basis of evidence from corpora.

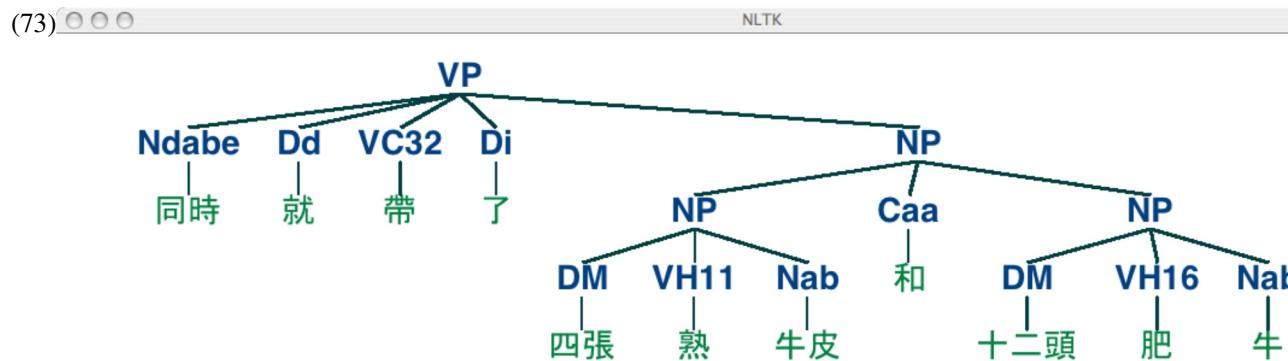
9.2 Treebanks (notes)

The `corpus` module defines the `treebank` corpus reader, which contains a 10% sample of the Penn Treebank corpus.

```
>>> print nltk.corpus.treebank.parsed_sents('wsj_0001.mrg')[0]
(S
  (NP-SBJ
    (NP (NNP Pierre) (NNP Vinken))
    (, ,)
    (ADJP (NP (CD 61) (NNS years)) (JJ old))
    (, ,))
  (VP
    (MD will)
    (VP
      (VB join)
      (NP (DT the) (NN board))
      (PP-CLR
        (IN as)
        (NP (DT a) (JJ nonexecutive) (NN director)))
      (NP-TMP (NNP Nov.) (CD 29))))
    (. .))
```

NLTK also includes a sample from the *Sinica Treebank Corpus*, consisting of 10,000 parsed sentences drawn from the *Academia Sinica Balanced Corpus of Modern Chinese*. Here is a code fragment to read and display one of the trees in this corpus.

```
>>> nltk.corpus.sinica_treebank.parsed_sents()[3450].draw()
```



9.3 Probabilistic Parsing

As we pointed out in the introduction to this chapter, dealing with ambiguity is a key challenge to broad coverage parsers. We have shown how chart parsing can help improve the efficiency of computing multiple parses of the same sentences. But the sheer number of parses can be just overwhelming. We will show how probabilistic parsing helps to manage a large space of parses. However, before we deal with these parsing issues, we must first back up and introduce weighted grammars.

Weighted Grammars

We begin by considering the verb *give*. This verb requires both a direct object (the thing being given) and an indirect object (the recipient). These complements can be given in either order, as illustrated in [example \(74b\)](#). In the "prepositional dative" form, the indirect object appears last, and inside a prepositional phrase, while in the "double object" form, the indirect object comes first:

- (74)
- a. Kim gave a bone to the dog
 - b. Kim gave the dog a bone

Using the Penn Treebank sample, we can examine all instances of prepositional dative and double object constructions involving *give*, as shown in [Figure 9.2](#).

```
def give(t):
    return t.node == 'VP' and len(t) > 2 and t[1].node == 'NP' \
        and (t[2].node == 'PP-DTV' or t[2].node == 'NP') \
        and ('give' in t[0].leaves() or 'gave' in t[0].leaves())
def sent(t):
    return ' '.join(token for token in t.leaves() if token[0] not in '*-0')
def print_node(t, width):
    output = "%s %s: %s / %s: %s" %
        (sent(t[0]), t[1].node, sent(t[1]), t[2].node, sent(t[2]))
    if len(output) > width:
        output = output[:width] + "..."
    print output
>>> for tree in nltk.corpus.treebank.parsed_sents():
...     for t in tree.subtrees(give):
...         print_node(t, 72)
gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad sp...
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: them / NP:
give NP: only French history questions / PP-DTV: to students in a Europe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving , nor to ...
give NP: your Foster Savings Institution / NP: the gift of hope and free...
give NP: market operators / NP: the authority to suspend trading in futu...
gave NP: quick approval / PP-DTV: to $ 3.18 billion in supplemental appr...
give NP: the Transportation Department / NP: up to 50 days to review any...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right , but not the obligation , to buy a cal...
gave NP: Mr. Thomas / NP: only a `` qualified '' rating , rather than `` ...
give NP: the president / NP: line-item veto power
```

[Figure 9.2 \(give.py\)](#): Figure 9.2: Usage of Give and Gave in the Penn Treebank sample

We can observe a strong tendency for the shortest complement to appear first. However, this does not account for a form like *give NP: federal judges / NP: a raise*, where animacy may be playing a role. In fact there turn out to be a large number of contributing factors, as surveyed by [\[Bresnan & Hay, 2006\]](#).

How can such tendencies be expressed in a conventional context free grammar? It turns out that they cannot. However, we can address the problem by adding weights, or probabilities, to the productions of a grammar.

A **probabilistic context free grammar** (or *PCFG*) is a context free grammar that associates a probability with each of its

productions. It generates the same set of parses for a text that the corresponding context free grammar does, and assigns a probability to each parse. The probability of a parse generated by a PCFG is simply the product of the probabilities of the productions used to generate it.

The simplest way to define a PCFG is to load it from a specially formatted string consisting of a sequence of weighted productions, where weights appear in brackets, as shown in [Figure 9.3](#).

```
grammar = nltk.parse_pcfg("""
    S      -> NP VP [1.0]
    VP     -> TV NP [0.4]
    VP     -> IV [0.3]
    VP     -> DatV NP NP [0.3]
    TV     -> 'saw' [1.0]
    IV     -> 'ate' [1.0]
    DatV   -> 'gave' [1.0]
    NP     -> 'telescopes' [0.8]
    NP     -> 'Jack' [0.2]
""")

>>> print grammar
Grammar with 9 productions (start state = S)
S -> NP VP [1.0]
VP -> TV NP [0.4]
VP -> IV [0.3]
VP -> DatV NP NP [0.3]
TV -> 'saw' [1.0]
IV -> 'ate' [1.0]
DatV -> 'gave' [1.0]
NP -> 'telescopes' [0.8]
NP -> 'Jack' [0.2]
```

[Figure 9.3 \(pcfg1.py\)](#): [Figure 9.3](#): Defining a Probabilistic Context Free Grammar (PCFG)

It is sometimes convenient to combine multiple productions into a single line, e.g. $VP \rightarrow TV\ NP\ [0.4] \mid IV\ [0.3] \mid DatV\ NP\ NP\ [0.3]$. In order to ensure that the trees generated by the grammar form a probability distribution, PCFG grammars impose the constraint that all productions with a given left-hand side must have probabilities that sum to one. The grammar in [Figure 9.3](#) obeys this constraint: for S , there is only one production, with a probability of 1.0; for VP , $0.4+0.3+0.3=1.0$; and for NP , $0.8+0.2=1.0$. The parse tree returned by `parse()` includes probabilities:

```
>>> viterbi_parser = nltk.ViterbiParser(grammar)
>>> print viterbi_parser.parse(['Jack', 'saw', 'telescopes'])
(S (NP Jack) (VP (TV saw) (NP telescopes))) (p=0.064)
```

The next two sections introduce two probabilistic parsing algorithms for PCFGs. The first is an A* parser that uses dynamic programming to find the single most likely parse for a given text. Whenever it finds multiple possible parses for a subtree, it discards all but the most likely parse. The second is a bottom-up chart parser that maintains a queue of edges, and adds them to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, allowing the parser to expand more likely edges before less likely ones. Different queue orderings are used to implement a variety of different search strategies. These algorithms are implemented in the `nltk.parse.viterbi` and `nltk.parse.pchart` modules.

A* Parser

An **A* Parser** is a bottom-up PCFG parser that uses dynamic programming to find the single most likely parse for a text [\[Klein & Manning, 2003\]](#). It parses texts by iteratively filling in a **most likely constituents table**. This table records the most likely tree for each span and node value. For example, after parsing the sentence "I saw the man with the telescope" with the grammar `cfg.toy_pcfg1`, the most likely constituents table contains the following entries (amongst others):

Table 9.2:

Fragment of Most Likely Constituents Table

| Span | Node | Tree | Prob |
|-------|------|--------------------|------|
| [0:1] | NP | (NP I) | 0.15 |
| [6:7] | NP | (NN telescope) | 0.5 |
| [5:7] | NP | (NP the telescope) | 0.2 |

| Span | Node | Tree | Prob |
|-------|------|---|--------------|
| [4:7] | PP | (PP with (NP the telescope)) | 0.122 |
| [0:4] | S | (S (NP I) (VP saw (NP the man))) | 0.01365 |
| [0:7] | S | (S (NP I) (VP saw (NP (NP the man) (PP with (NP the telescope)))))) | 0.0004163250 |

Once the table has been completed, the parser returns the entry for the most likely constituent that spans the entire text, and whose node value is the start symbol. For this example, it would return the entry with a span of [0:6] and a node value of "S".

Note that we only record the *most likely* constituent for any given span and node value. For example, in the table above, there are actually two possible constituents that cover the span [1:6] and have "VP" node values.

1. "saw the man, who has the telescope":

```
(VP saw
  (NP (NP John)
    (PP with (NP the telescope))))
```

2. "used the telescope to see the man":

```
(VP saw
  (NP John) (PP with (NP the telescope)))
```

Since the grammar we are using to parse the text indicates that the first of these tree structures has a higher probability, the parser discards the second one.

Filling in the Most Likely Constituents Table: Because the grammar used by `ViterbiParse` is a PCFG, the probability of each constituent can be calculated from the probabilities of its children. Since a constituent's children can never cover a larger span than the constituent itself, each entry of the most likely constituents table depends only on entries for constituents with *shorter* spans (or equal spans, in the case of unary and epsilon productions).

`ViterbiParse` takes advantage of this fact, and fills in the most likely constituent table incrementally. It starts by filling in the entries for all constituents that span a single element of text. After it has filled in all the table entries for constituents that span one element of text, it fills in the entries for constituents that span two elements of text. It continues filling in the entries for constituents spanning larger and larger portions of the text, until the entire table has been filled.

To find the most likely constituent with a given span and node value, `ViterbiParse` considers all productions that could produce that node value. For each production, it checks the most likely constituents table for sequences of children that collectively cover the span and that have the node values specified by the production's right hand side. If the tree formed by applying the production to the children has a higher probability than the current table entry, then it updates the most likely constituents table with the new tree.

Handling Unary Productions and Epsilon Productions: A minor difficulty is introduced by unary productions and epsilon productions: an entry of the most likely constituents table might depend on another entry with the same span. For example, if the grammar contains the production `v → VP`, then the table entries for `VP` depend on the entries for `v` with the same span. This can be a problem if the constituents are checked in the wrong order. For example, if the parser tries to find the most likely constituent for a `VP` spanning [1:3] before it finds the most likely constituents for `v` spanning [1:3], then it can't apply the `v → VP` production.

To solve this problem, `ViterbiParse` repeatedly checks each span until it finds no new table entries. Note that cyclic grammar productions (e.g. `v → v`) will *not* cause this procedure to enter an infinite loop. Since all production probabilities are less than or equal to 1, any constituent generated by a cycle in the grammar will have a probability that is less than or equal to the original constituent; so `ViterbiParse` will discard it.

In NLTK, we create Viterbi parsers using `ViterbiParse()`. Note that since `ViterbiParse` only finds the single most likely parse, that `nbest_parse()` will never return more than one parse.

```
grammar = nltk.parse_pcfg('''
NP  -> NNS [0.5] | JJ NNS [0.3] | NP CC NP [0.2]
NNS -> "cats" [0.1] | "dogs" [0.2] | "mice" [0.3] | NNS CC NNS [0.4]
JJ  -> "big" [0.4] | "small" [0.6]
CC  -> "and" [0.9] | "or" [0.1]
''')
```

```

viterbi_parser = nltk.ViterbiParser(grammar)
>>> sent = 'big cats and dogs'.split()
>>> print viterbi_parser.parse(sent)
(NP (JJ big) (NNS (NNS cats) (CC and) (NNS dogs))) (p=0.000864)

```

Figure 9.4 (viterbi_parse.py): Figure 9.4

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays the constituents that are considered, and indicates which ones are added to the most likely constituent table. It also indicates the likelihood for each constituent.

```

>>> viterbi_parser.trace(3)
>>> print viterbi_parser.parse(sent)
Inserting tokens into the most likely constituents table...
Insert: |=...| big
Insert: |.=...| cats
Insert: |...=..| and
Insert: |....=| dogs
Finding the most likely constituents spanning 1 text elements...
Insert: |=...| JJ -> 'big' [0.4] 0.4000000000
Insert: |.=...| NNS -> 'cats' [0.1] 0.1000000000
Insert: |...=..| NP -> NNS [0.5] 0.0500000000
Insert: |...=..| CC -> 'and' [0.9] 0.9000000000
Insert: |....=| NNS -> 'dogs' [0.2] 0.2000000000
Insert: |....=| NP -> NNS [0.5] 0.1000000000
Finding the most likely constituents spanning 2 text elements...
Insert: |==...| NP -> JJ NNS [0.3] 0.0120000000
Finding the most likely constituents spanning 3 text elements...
Insert: |.====| NP -> NP CC NP [0.2] 0.0009000000
Insert: |.====| NNS -> NNS CC NNS [0.4] 0.0072000000
Insert: |.====| NP -> NNS [0.5] 0.0036000000
Discard: |.====| NP -> NP CC NP [0.2] 0.0009000000
Discard: |.====| NP -> NP CC NP [0.2] 0.0009000000
Finding the most likely constituents spanning 4 text elements...
Insert: |====| NP -> JJ NNS [0.3] 0.0008640000
Discard: |====| NP -> NP CC NP [0.2] 0.0002160000
Discard: |====| NP -> NP CC NP [0.2] 0.0002160000
(NP (JJ big) (NNS (NNS cats) (CC and) (NNS dogs))) (p=0.000864)

```

A Bottom-Up PCFG Chart Parser

The A* parser described in the previous section finds the single most likely parse for a given text. However, when parsers are used in the context of a larger NLP system, it is often necessary to produce several alternative parses. In the context of an overall system, a parse that is assigned low probability by the parser might still have the best overall probability.

For example, a probabilistic parser might decide that the most likely parse for "I saw John with the cookie" is the structure with the interpretation "I used my cookie to see John"; but that parse would be assigned a low probability by a semantic system. Combining the probability estimates from the parser and the semantic system, the parse with the interpretation "I saw John, who had my cookie" would be given a higher overall probability.

A probabilistic bottom-up chart parser maintains an **edge queue**, and adds these edges to the chart one at a time. The ordering of this queue is based on the probabilities associated with the edges, and this allows the parser to insert the most probable edges first. Each time an edge is added to the chart, it may become possible to insert further edges, so these are added to the queue. The bottom-up chart parser continues adding the edges in the queue to the chart until enough complete parses have been found, or until the edge queue is empty.

Like an edge in a regular chart, a probabilistic edge consists of a dotted production, a span, and a (partial) parse tree. However, unlike ordinary charts, this time the tree is weighted with a probability. Its probability is the product of the probability of the production that generated it and the probabilities of its children. For example, the probability of the edge `[Edge: S → NP•VP, 0:2]` is the probability of the PCFG production `S → NP VP` multiplied by the probability of its NP child. (Note that an edge's tree only includes children for elements to the left of the edge's dot.)

Bottom-Up PCFG Strategies

The edge queue is a sorted list of edges that can be added to the chart. It is initialized with a single edge for each token, with the

form [Edge: token → •]. As each edge from the queue is added to the chart, it may become possible to add further edges, according to two rules: (i) the Bottom-Up Initialization Rule can be used to add a self-loop edge whenever an edge whose dot is in position 0 is added to the chart; or (ii) the Fundamental Rule can be used to combine a new edge with edges already present in the chart. These additional edges are queued for addition to the chart.

By changing the sort order used by the queue, we can control the strategy that the parser uses to explore the search space. Since there are a wide variety of reasonable search strategies, `BottomUpChartParser()` does not define any sort order. Instead, different strategies are implemented in subclasses of `BottomUpChartParser()`.

Lowest Cost First: The simplest way to order the edge queue is to sort edges by the probabilities of their associated trees. This ordering concentrates the efforts of the parser on those edges that are more likely to be correct analyses of their corresponding input tokens. Now, the probability of an edge's tree provides an upper bound on the probability of any parse produced using that edge. The probabilistic "cost" of using an edge to form a parse is one minus its tree's probability. Thus, inserting the edges with the most likely trees first results in a **lowest-cost-first search strategy**. Lowest-cost-first search turns out to be optimal: the first solution it finds is guaranteed to be the best solution (cf the A* parser).

However, lowest-cost-first search can be rather inefficient. Recall that a tree's probability is the product of the probabilities of all the productions used to generate it. Consequently, smaller trees tend to have higher probabilities than larger ones. Thus, lowest-cost-first search tends to work with edges having small trees before considering edges with larger trees. Yet any complete parse of the text will necessarily have a large tree, and so this strategy will tend to produce complete parses only once most other edges are processed.

Let's consider this problem from another angle. The basic shortcoming with lowest-cost-first search is that it ignores the probability that an edge's tree will be part of a complete parse. The parser will try parses that are locally coherent even if they are unlikely to form part of a complete parse. Unfortunately, it can be quite difficult to calculate the probability that a tree is part of a complete parse. However, we can use a variety of techniques to approximate that probability.

Best-First Search: This method sorts the edge queue in descending order of the edges' span, no the assumption that edges having a larger span are more likely to form part of a complete parse. This is a **best-first search strategy**, since it inserts the edges that are closest to producing complete parses before trying any other edges. However, best-first search is *not* optimal: the first solution it finds is not guaranteed to be the best solution. However, it will usually find a complete parse much more quickly than lowest-cost-first search.

Beam Search: When large grammars are used to parse a text, the edge queue can grow quite long. The edges at the end of a long queue are unlikely to be used. Therefore, it is reasonable to remove these edges from the queue. This strategy is known as **beam search**; it only keeps the best partial results. The bottom-up chart parsers take an optional parameter `beam_size`; whenever the edge queue grows longer than this, it is pruned. This parameter is best used in conjunction with `InsideChartParser()`. Beam search reduces the space requirements for lowest-cost-first search, by discarding edges that are not likely to be used. But beam search also loses many of lowest-cost-first search's more useful properties. Beam search is not optimal: it is not guaranteed to find the best parse first. In fact, since it might prune a necessary edge, beam search is not complete: it won't find every parse, and it is not even guaranteed to return a parse if one exists.

The code in [Figure 9.5](#) demonstrates how we define and use these probabilistic chart parsers in NLTK.

```
inside_parser = nltk.InsideChartParser(grammar)
longest_parser = nltk.LongestChartParser(grammar)
beam_parser = nltk.InsideChartParser(grammar, beam_size=20)

>>> print inside_parser.parse(sent)
(NP (JJ big) (NNS (NNS cats) (CC and) (NNS dogs))) (p=0.000864)
>>> for tree in inside_parser.nbest_parse(sent):
...     print tree
(NP
 (JJ big)
 (NNS (NNS cats) (CC and) (NNS dogs))) (p=0.000864)
(NP
 (NP (JJ big) (NNS cats))
 (CC and)
 (NP (NNS dogs))) (p=0.000216)
```

[Figure 9.5 \(bottom_up_chart_parsers.py\): Figure 9.5](#)

The `trace` method can be used to set the level of tracing output that is generated when parsing a text. Trace output displays edges as they are added to the chart, and shows the probability for each edges' tree.

Note

Your Turn: Run the above example using tracing, by calling `inside_parser.trace(3)` before running the parser.

9.4 Grammar Induction

As we have seen, PCFG productions are just like CFG productions, adorned with probabilities. So far, we have simply specified these probabilities in the grammar. However, it is more usual to *estimate* these probabilities from training data, namely a collection of parse trees or *treebank*.

The simplest method uses *Maximum Likelihood Estimation*, so called because probabilities are chosen in order to maximize the likelihood of the training data. The probability of a production $VP \rightarrow V \ NP \ PP$ is $p(V, NP, PP | VP)$. We calculate this as follows:

$$P(V, NP, PP | VP) = \frac{\text{count}(VP \rightarrow V \ NP \ PP)}{\text{count}(VP \rightarrow \dots)}$$

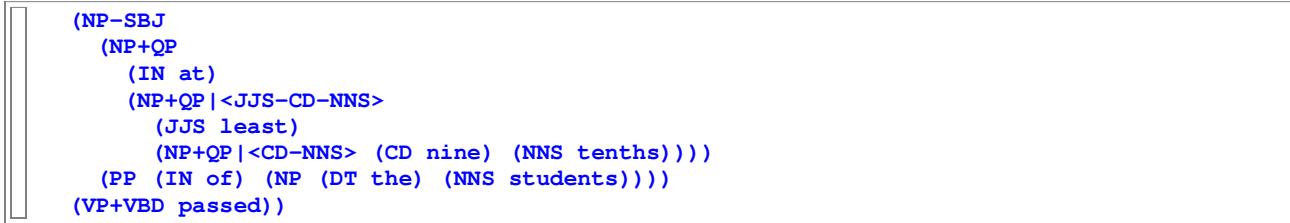
Here is a simple program that induces a grammar from the first three parse trees in the Penn Treebank corpus:

```
>>> from itertools import islice
>>> productions = []
>>> S = nltk.Nonterminal('S')
>>> for tree in nltk.corpus.treebank.parsed_sents('wsj_0002.mrg'):
...     productions += tree.productions()
>>> grammar = nltk.induce_pcfg(S, productions)
>>> for production in grammar.productions()[:10]:
...     print production
CC -> 'and' [1.0]
NNP -> 'Agnew' [0.166666666667]
JJ -> 'industrial' [0.2]
NP -> CD NNS [0.142857142857]
, -> ',' [1.0]
S -> NP-SBJ NP-PRD [0.5]
VP -> VBN S [0.5]
NNP -> 'Rudolph' [0.166666666667]
NP -> NP PP [0.142857142857]
NNP -> 'PLC' [0.166666666667]
```

Normal Forms

Grammar induction usually involves normalizing the grammar in various ways. NLTK trees support binarization (Chomsky Normal Form), parent annotation, Markov order-N smoothing, and unary collapsing:

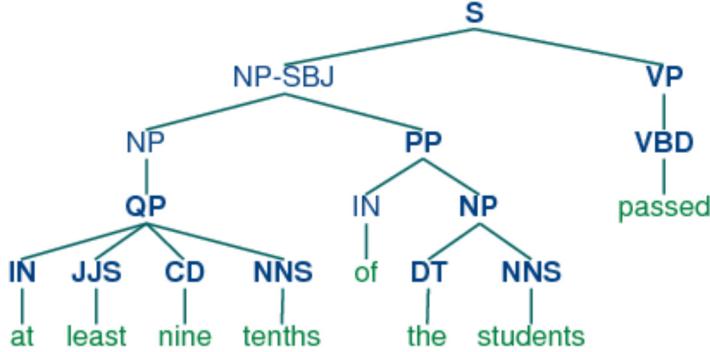
```
>>> treebank_string = """(S (NP-SBJ (NP (QP (IN at) (JJS least) (CD nine) (NNS tenths))) )
...   (PP (IN of) (NP (DT the) (NNS students)))) (VP (VBD passed)))"""
>>> t = nltk.bracket_parse(treebank_string)
>>> print t
(S
(NP-SBJ
  (NP (QP (IN at) (JJS least) (CD nine) (NNS tenths)))
  (PP (IN of) (NP (DT the) (NNS students))))
  (VP (VBD passed)))
>>> t.collapse Unary(collapsePOS=True)
>>> print t
(S
(NP-SBJ
  (NP+QP (IN at) (JJS least) (CD nine) (NNS tenths))
  (PP (IN of) (NP (DT the) (NNS students))))
  (VP+VBD passed))
>>> t.chomsky_normal_form()
>>> print t
(S
```



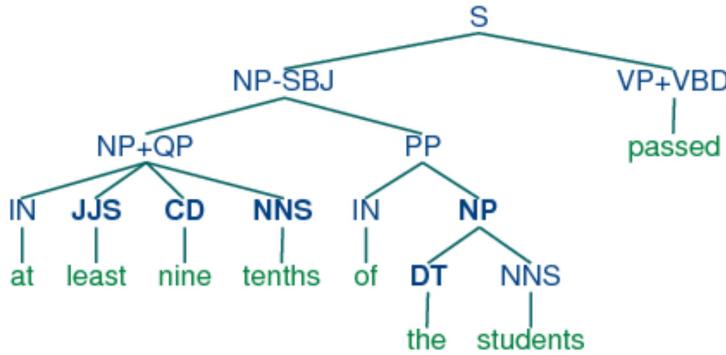
These trees are shown in (4c).

(75)

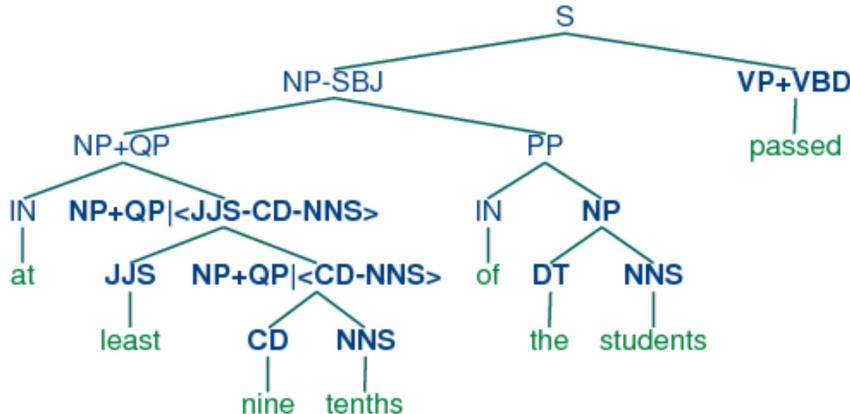
a.



b.



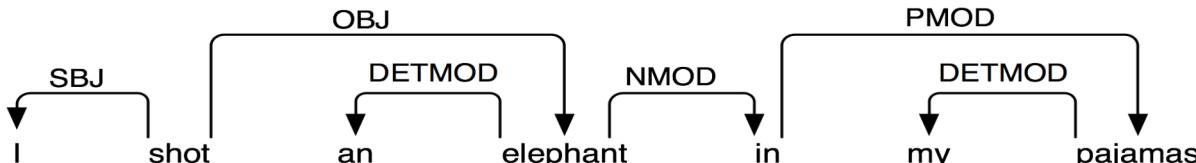
c.



9.5 Dependency Grammar

Context Free Grammar is founded on the notion that sequences of adjacent words can be grouped into constituents, such as as prepositional phrases. A separate tradition in syntax, recently gaining popularity in NLP, is known as dependency grammar. Here, the most basic notion is that of dependency: words are **dependent** on another words. The root of a sentence is usually taken to be the main verb, and every other word is either dependent on the root, or connects to it through a path of dependencies. [Figure \(76\)](#) illustrates a dependency graph, where the head of the arrow points to the head of a dependency.

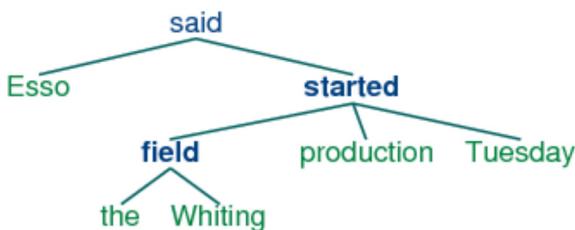
(76)



As you will see, the arcs in [Figure \(76\)](#) are labeled with the particular dependency relation that holds between a dependent and its head. For example, *Esso* bears the subject relation to *said* (which is the head of the whole sentence), and *Tuesday* bears a verbal modifier (VMOD) relation to *started*.

An alternative way of representing the dependency relationships is illustrated in the [tree \(77\)](#), where dependents are shown as daughters of their heads.

(77)



One format for encoding dependency information places each word on a line, followed by its part-of-speech tag, the index of its head, and the label of the dependency relation (cf. [\[Nivre, Hall, & Nilsson, 2006\]](#)). The index of a word is implicitly given by the ordering of the lines (with 1 as the first index). This is illustrated in the following code snippet:

```

>>> from nltk import DependencyGraph
>>> dg = DependencyGraph("""Esso      NNP 2      SUB
... said      VBD 0      ROOT
... the       DT 5      NMOD
... Whiting   NNP 5      NMOD
... field     NN 6      SUB
... started   VBD 2      VMOD
... production NN 6      OBJ
... Tuesday   NNP 6      VMOD""")
  
```

As you will see, this format also adopts the convention that the head of the sentence is dependent on an empty node, indexed as 0. We can use the `deptree()` method of a `DependencyGraph()` object to build an NLTK tree like that illustrated earlier in [\(6\)](#).

```

>>> tree = dg.deptree()
>>> tree.draw()
  
```

Projective Dependency Parsing

```

>>> grammar = nltk.parse_dependency_grammar("""
... 'fell' -> 'price' | 'stock'
... 'price' -> 'of' 'the'
... 'of' -> 'stock'
... 'stock' -> 'the'
... """)
>>> print grammar
Dependency grammar with 5 productions
'fell' -> 'price'
'fell' -> 'stock'
'price' -> 'of' 'the'
'of' -> 'stock'
'stock' -> 'the'
  
```

```

>>> dp = nltk.ProjectiveDependencyParser(grammar)
>>> for t in dp.parse(['the', 'price', 'of', 'the', 'stock', 'fell']):
...     print tree
(fell (price the of the) stock)
(fell (price the of) (stock the))
(fell (price the (of (stock the))))
  
```

Non-Projective Dependency Parsing

```
>>> grammar = nltk.parse_dependency_grammar("""
... 'taught' -> 'play' | 'man'
... 'man' -> 'the'
... 'play' -> 'golf' | 'dog' | 'to'
... 'dog' -> 'his'
... """)
>>> print grammar
Dependency grammar with 7 productions
'taught' -> 'play'
'taught' -> 'man'
'man' -> 'the'
'play' -> 'golf'
'play' -> 'dog'
'play' -> 'to'
'dog' -> 'his'
```

```
>>> dp = nltk.NonprojectiveDependencyParser(grammar)
>>> for g in dp.parse(['the', 'man', 'taught', 'his', 'dog', 'to', 'play', 'golf']):
...     print g
[{'address': 0, 'deps': 3, 'rel': 'TOP', 'tag': 'TOP', 'word': None},
 {'address': 1, 'deps': [], 'word': 'the'},
 {'address': 2, 'deps': [1], 'word': 'man'},
 {'address': 3, 'deps': [2, 7], 'word': 'taught'},
 {'address': 4, 'deps': [], 'word': 'his'},
 {'address': 5, 'deps': [4], 'word': 'dog'},
 {'address': 6, 'deps': [], 'word': 'to'},
 {'address': 7, 'deps': [5, 6, 8], 'word': 'play'},
 {'address': 8, 'deps': [], 'word': 'golf'}]
```

Note

The dependency parser modules also support probabilistic dependency parsing.

9.6 Further Reading

Section 13.4 of [\[Jurafsky & Martin, 2008\]](#) covers chart parsing, and Chapter 14 contains a more formal presentation of statistical parsing.

- [\[Manning, 2003\]](#)
- [\[Klein & Manning, 2003\]](#)
- [\[Charniak, 1997\]](#)

9.7 Exercises

1. ☀ Consider the sequence of words: *Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo*. This is a grammatically correct sentence, as explained at http://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo. Consider the tree diagram presented on this Wikipedia page, and write down a suitable grammar. Normalize case to lowercase, to simulate the problem that a listener has when hearing this sentence. Can you find other parses for this sentence? How does the number of parse trees grow as the sentence gets longer? (More examples of these sentences can be found at http://en.wikipedia.org/wiki/List_of_homophonous_phrases).
2. ● Consider the algorithm in [Figure 8.3](#). Can you explain why parsing context-free grammar is proportional to n^3 , where n is the length of the input sentence?
3. ● Modify the functions `init_wfst()` and `complete_wfst()` so that the contents of each cell in the WFST is a set of non-terminal symbols rather than a single non-terminal.
4. ● Process each tree of the Treebank corpus sample `nltk.corpus.treebank` and extract the productions with the help of `Tree.productions()`. Discard the productions that occur only once. Productions with the same left hand side, and

similar right hand sides can be collapsed, resulting in an equivalent but more compact set of rules. Write code to output a compact grammar.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#). Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

10 Feature Based Grammar

10.1 Introduction

Imagine you are building a spoken dialogue system to answer queries about train schedules in Europe. (1) illustrates one of the input sentences that the system should handle.

(78) Which stations does the 9.00 express from Amsterdam to Paris stop at?

The information that the customer is seeking is not exotic — the system back-end just needs to look up the list of stations on the route, and reel them off. But you have to be careful in giving the correct semantic interpretation to (1). You don't want to end up with the system trying to answer (2) instead:

(79) Which station does the 9.00 express from Amsterdam terminate at?

Part of your solution might use domain knowledge to figure out that if a speaker knows that the train is a train to Paris, then she probably isn't asking about the terminating station in (1). But the solution will also involve recognizing the syntactic structure of the speaker's query. In particular, your analyzer must recognize that there is a syntactic connection between the question phrase *which stations*, and the phrase *stop at* at the end (1). The required interpretation is made clearer in the "quiz question version" shown in (3), where the question phrase fills the "gap" that is implicit in (1):

(80) The 9.00 express from Amsterdam to Paris stops at which stations?

The **long-distance dependency** between an initial question phrase and the gap that it semantically connects to cannot be recognized by techniques we have presented in earlier chapters. For example, we can't use *n*-gram based language models; in practical terms, it is infeasible to observe the *n*-grams for a big enough value of *n*. Similarly, chunking grammars only attempt to capture local patterns, and therefore just don't "see" long-distance dependencies. In this chapter, we will show how syntactic features can be used to provide a simple yet effective technique for keeping track of long-distance dependencies in sentences.

Features are helpful too for dealing with purely local dependencies. Consider the German questions (4).

(81)

- a. Welche Studenten kennen Franz?
which student.PL know.PL Franz
'which students know Franz?'
- b. Welche Studenten kennt Franz?
which student.PL know.SG Franz
'which students does Franz know?'

The only way of telling which noun phrase is the subject of *kennen* ('know') and which is the object is by looking at the agreement inflection on the verb — word order is no help to us here. Since verbs in German agree in number with their subjects, the plural form *kennen* requires *Welche Studenten* as subject, while the singular form *kennt* requires *Franz* as subject. The fact that subjects and verbs must agree in number can be expressed within the CFGs that we presented in [Chapter 8](#). But capturing the fact that the interpretations of (81a) and (81b) differ is more challenging. In this chapter, we will only examine the syntactic aspect of local dependencies such as number agreement. In [Chapter 11](#), we will demonstrate how feature-based grammars can

be extended so that they build a representation of meaning in parallel with a representation of syntactic structure.

10.2 Why Features?

We have already used the term "feature" a few times, without saying what it means. What's special about feature-based grammars? The core ideas are probably already familiar to you. To make things concrete, let's look at the simple phrase *these dogs*. It's composed of two words. We'll be a bit abstract for the moment, and call these words *a* and *b*. We'll be modest, and assume that we do not know *everything* about them, but we can at least give a partial description. For example, we know that the orthography of *a* is *these*, its phonological form is *DH IY Z*, its part-of-speech is DET, and its number is plural. We can use dot notation to record these observations:

- (82) *a.spelling* = *these*
a.phonology = *DH IY Z*
a.pos = DET
a.number = plural

Thus (5) is a *partial description* of a word; it lists some attributes, or features, of the word, and declares their values. There are other attributes that we might be interested in, which have not been specified; for example, what head the word is dependent on (using the notion of dependency discussed in [Chapter 8](#)), and what the lemma of the word is. But this omission of some attributes is exactly what you would expect from a partial description!

We will start off this chapter by looking more closely at the phenomenon of syntactic agreement; we will show how agreement constraints can be expressed elegantly using features, and illustrate their use in a simple grammar. Feature structures are a general data structure for representing information of any kind; we will briefly look at them from a more formal point of view, and explain how to create feature structures in Python. In the final part of the chapter, we demonstrate that the additional expressiveness of features opens out a wide spectrum of possibilities for describing sophisticated aspects of linguistic structure.

Syntactic Agreement

Consider the following contrasts:

- (83)
- a. this dog
 - b. *these dog

- (84)
- a. these dogs
 - b. *this dogs

In English, nouns are usually morphologically marked as being singular or plural. The form of the demonstrative also varies: *this* (singular) and *these* (plural). Examples (6b) and (7b) show that there are constraints on the use of demonstratives and nouns within a noun phrase: either both are singular or both are plural. A similar constraint holds between subjects and predicates:

- (85)
- a. the dog runs
 - b. *the dog run

- (86)
- a. the dogs run
 - b. *the dogs runs

Here we can see that morphological properties of the verb co-vary with syntactic properties of the subject noun phrase. This co-variance is called **agreement**. If we look further at verb agreement in English, we will see that present tense verbs typically have two inflected forms: one for third person singular, and another for every other combination of person and number:

Table 10.1:

Agreement Paradigm for English Regular Verbs

| | singular | plural |
|---------|----------------|----------|
| 1st per | I run | we run |
| 2nd per | you run | you run |
| 3rd per | he/she/it runs | they run |

We can make the role of morphological properties a bit more explicit as illustrated in (10) and (11). These representations indicate that the verb agrees with its subject in person and number. (We use "3" as an abbreviation for 3rd person, "SG" for singular and "PL" for plural.)

- (87) the dog run-s
dog.3.SG run-3.SG

- (88) the dog-s run
dog.3.PL run-3.PL

Let's see what happens when we encode these agreement constraints in a context-free grammar. We will begin with the simple CFG in (12).

- (89) $S \rightarrow NP\ VP$
 $NP \rightarrow DET\ N$
 $VP \rightarrow V$
 $DET \rightarrow 'this'$
 $N \rightarrow 'dog'$
 $V \rightarrow 'runs'$

Example (89) allows us to generate the sentence *this dog runs*; however, what we really want to do is also generate *these dogs run* while blocking unwanted strings such as **this dogs run* and **these dog runs*. The most straightforward approach is to add new non-terminals and productions to the grammar:

- (90) $S_SG \rightarrow NP_SG\ VP_SG$
 $S_PL \rightarrow NP_PL\ VP_PL$
 $NP_SG \rightarrow DET_SG\ N_SG$
 $NP_PL \rightarrow DET_PL\ N_PL$
 $VP_SG \rightarrow V_SG$
 $VP_PL \rightarrow V_PL$
 $DET_SG \rightarrow 'this'$
 $DET_PL \rightarrow 'these'$
 $N_SG \rightarrow 'dog'$
 $N_PL \rightarrow 'dogs'$
 $V_SG \rightarrow 'runs'$
 $V_PL \rightarrow 'run'$

It should be clear that this grammar will do the required task, but only at the cost of duplicating our previous set of productions.

Using Attributes and Constraints

We spoke informally of linguistic categories having *properties*; for example, that a noun has the property of being plural. Let's make this explicit:

- (91) $N[\text{NUM}=pl]$

In (14), we have introduced some new notation which says that the category N has a **feature** called NUM (short for 'number') and that the value of this feature is *pl* (short for 'plural'). We can add similar annotations to other categories, and use them in lexical entries:

- (92) $DET[\text{NUM}=sg] \rightarrow 'this'$
 $DET[\text{NUM}=pl] \rightarrow 'these'$
 $N[\text{NUM}=sg] \rightarrow 'dog'$
 $N[\text{NUM}=pl] \rightarrow 'dogs'$

$V[\text{NUM}=\text{sg}] \rightarrow \text{'runs'}$
 $V[\text{NUM}=\text{pl}] \rightarrow \text{'run'}$

Does this help at all? So far, it looks just like a slightly more verbose alternative to what was specified in (13). Things become more interesting when we allow *variables* over feature values, and use these to state constraints:

(93)

- a. $S \rightarrow NP[\text{NUM}=?n] VP[\text{NUM}=?n]$
- b. $NP[\text{NUM}=?n] \rightarrow DET[\text{NUM}=?n] N[\text{NUM}=?n]$
- c. $VP[\text{NUM}=?n] \rightarrow V[\text{NUM}=?n]$

We are using " $?n$ " as a variable over values of NUM; it can be instantiated either to *sg* or *pl*. Its scope is limited to individual productions. That is, within (93a), for example, $?n$ must be instantiated to the same constant value; we can read the production as saying that whatever value NP takes for the feature NUM, VP must take the same value.

In order to understand how these feature constraints work, it's helpful to think about how one would go about building a tree. Lexical productions will admit the following local trees (trees of depth one):

(94)

- a. $\text{Det}[\text{NUM}=\text{sg}]$

- b. $\text{Det}[\text{NUM}=\text{pl}]$

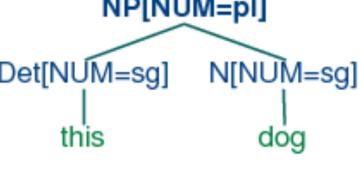
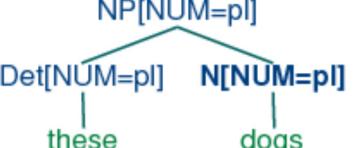

(95)

- a. $N[\text{NUM}=\text{sg}]$

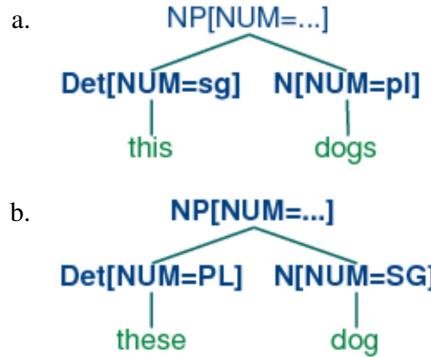
- b. $N[\text{NUM}=\text{pl}]$


Now (93b) says that whatever the NUM values of N and DET are, they have to be the same. Consequently, (93b) will permit (94a) and (95a) to be combined into an NP as shown in (96a) and it will also allow (94b) and (95b) to be combined, as in (96b). By contrast, (97a) and (97b) are prohibited because the roots of their constituent local trees differ in their values for the NUM feature.

(96)

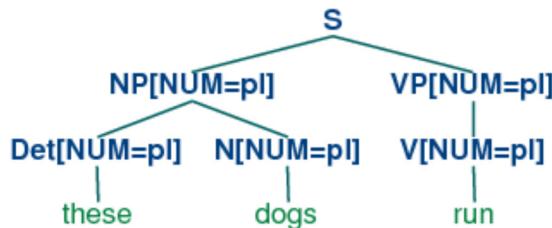
- a. $NP[\text{NUM}=\text{pl}]$

- b. $NP[\text{NUM}=\text{pl}]$


(97)



Production (93c) can be thought of as saying that the NUM value of the head verb has to be the same as the NUM value of the VP mother. Combined with (93a), we derive the consequence that if the NUM value of the subject head noun is *pl*, then so is the NUM value of the VP's head verb.

(98)



The grammar in [listing 10.1](#) illustrates most of the ideas we have introduced so far in this chapter, plus a couple of new ones.

```

>>> nltk.data.show_cfg('grammars/book_grammars/feat0.fcfg')
% start S
# #####
# Grammar Rules
# #####
# S expansion rules
S -> NP[NUM=?n] VP[NUM=?n]
# NP expansion rules
NP[NUM=?n] -> N[NUM=?n]
NP[NUM=?n] -> PropN[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
NP[NUM=pl] -> N[NUM=pl]
# VP expansion rules
VP[TENSE=?t, NUM=?n] -> IV[TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> TV[TENSE=?t, NUM=?n] NP
# #####
# Lexical Rules
# #####
Det[NUM=sg] -> 'this' | 'every'
Det[NUM=pl] -> 'these' | 'all'
Det -> 'the' | 'some'
PropN[NUM=sg] -> 'Kim' | 'Jody'
N[NUM=sg] -> 'dog' | 'girl' | 'car' | 'child'
N[NUM=pl] -> 'dogs' | 'girls' | 'cars' | 'children'
IV[TENSE=pres, NUM=sg] -> 'disappears' | 'walks'
TV[TENSE=pres, NUM=sg] -> 'sees' | 'likes'
IV[TENSE=pres, NUM=pl] -> 'disappear' | 'walk'
TV[TENSE=pres, NUM=pl] -> 'see' | 'like'
IV[TENSE=past, NUM=?n] -> 'disappeared' | 'walked'
TV[TENSE=past, NUM=?n] -> 'saw' | 'liked'
  
```

[Figure 10.1 \(feat0cfg.py\)](#): Figure 10.1: Example Feature-Based Grammar

Notice that a syntactic category can have more than one feature; for example, $V[TENSE=pres, NUM=pl]$. In general, we can add as many features as we like.

Notice also that we have used feature variables in lexical entries as well as grammatical productions. For example, *the* has been assigned the category $DET[NUM=?n]$. Why is this? Well, you know that the definite article *the* can combine with both singular

and plural nouns. One way of describing this would be to add two lexical entries to the grammar, one each for the singular and plural versions of *the*. However, a more elegant solution is to leave the NUM value **underspecified** and letting it agree in number with whatever noun it combines with.

A final detail about [10.1](#) is the statement `%start S`. This a "directive" that tells the parser to take S as the start symbol for the grammar.

In general, when we are trying to develop even a very small grammar, it is convenient to put the productions in a file where they can be edited, tested and revised. We have saved [10.1](#) as a file named '`feat0.fcfg`' in the NLTK data distribution, and it can be accessed using `nltk.data.load()`.

We can inspect the productions and the lexicon using the commands `print g.earley_grammar()` and `pprint(g.earley_lexicon())`.

Next, we can tokenize a sentence and use the `nbest_parse()` function to invoke the Earley chart parser.

```
>>> tokens = 'Kim likes children'.split()
>>> from nltk.parse import load_earley
>>> cp = load_earley('grammars/book_grammars/feat0.fcfg', trace=2)
>>> trees = cp.nbest_parse(tokens)
    .K.1.c.
Processing queue 0
Predictor |> . . . | [0:0] S[] -> * NP[NUM=?n] VP[NUM=?n] {}
Predictor |> . . . | [0:0] NP[NUM=?n] -> * N[NUM=?n] {}
Predictor |> . . . | [0:0] NP[NUM=?n] -> * PropN[NUM=?n] {}
Predictor |> . . . | [0:0] NP[NUM=?n] -> * Det[NUM=?n] N[NUM=?n] {}
Predictor |> . . . | [0:0] NP[NUM='pl'] -> * N[NUM='pl'] {}
Scanner |[-] . . | [0:1] 'Kim'
Scanner |[-] . . | [0:1] PropN[NUM='sg'] -> 'Kim' *
Processing queue 1
Completer |[-] . . | [0:1] NP[NUM='sg'] -> PropN[NUM='sg'] *
Completer |[-] . . | [0:1] S[] -> NP[NUM=?n] * VP[NUM=?n] {?n: 'sg'}
Predictor |. > . . | [1:1] VP[NUM=?n, TENSE=?t] -> * IV[NUM=?n, TENSE=?t] {}
Predictor |. > . . | [1:1] VP[NUM=?n, TENSE=?t] -> * TV[NUM=?n, TENSE=?t] NP[] {}
Scanner |. [-] . | [1:2] 'likes'
Scanner |. [-] . | [1:2] TV[NUM='sg', TENSE='pres'] -> 'likes' *
Processing queue 2
Completer |. [-] . | [1:2] VP[NUM=?n, TENSE=?t] -> TV[NUM=?n, TENSE=?t] * NP[] {?n: 'sg', ?t: 'pres'}
Predictor |. . > . | [2:2] NP[NUM=?n] -> * N[NUM=?n] {}
Predictor |. . > . | [2:2] NP[NUM=?n] -> * PropN[NUM=?n] {}
Predictor |. . > . | [2:2] NP[NUM=?n] -> * Det[NUM=?n] N[NUM=?n] {}
Predictor |. . > . | [2:2] NP[NUM='pl'] -> * N[NUM='pl'] {}
Scanner |. . [-] | [2:3] 'children'
Scanner |. . [-] | [2:3] N[NUM='pl'] -> 'children' *
Processing queue 3
Completer |. . [-] | [2:3] NP[NUM='pl'] -> N[NUM='pl'] *
Completer |. . [-] | [1:3] VP[NUM='sg', TENSE='pres'] -> TV[NUM='sg', TENSE='pres'] NP[] *
Completer |[=====]| [0:3] S[] -> NP[NUM='sg'] VP[NUM='sg'] *
Completer |[=====]| [0:3] [INIT] [] -> S[] *
```

[Figure 10.2 \(featurecharttrace.py\)](#): Figure 10.2: Trace of Feature-Based Chart Parser

Observe that the parser works directly with the underspecified productions given by the grammar. That is, the Predictor rule does not attempt to compile out all admissible feature combinations before trying to expand the non-terminals on the left hand side of a production. However, when the Scanner matches an input word against a lexical production that has been predicted, the new edge will typically contain fully specified features; e.g., the edge $[PropN[NUM = sg] \rightarrow 'Kim', (0, 1)]$. Recall from [Chapter 8](#) that the Fundamental (or Completer) Rule in standard CFGs is used to combine an incomplete edge that's expecting a nonterminal B with a following, complete edge whose left hand side matches B . In our current setting, rather than checking for a complete match, we test whether the expected category B will **unify** with the left hand side B' of a following complete edge. We will explain in more detail in [Section 10.3](#) how unification works; for the moment, it is enough to know that as a result of unification, any variable values of features in B will be instantiated by constant values in the corresponding feature structure in B' , and these instantiated values will be used in the new edge added by the Completer. This instantiation can be seen, for example, in the edge $[NP[NUM=sg] \rightarrow PropN[NUM=sg] \bullet, (0, 1)]$ in [10.2](#), where the feature NUM has been assigned the value sg.

Finally, we can inspect the resulting parse trees (in this case, a single one).

```
>>> for tree in trees: print tree
```

```
(S []
  (NP [NUM='sg'] (PropN [NUM='sg'] Kim))
  (VP [NUM='sg', TENSE='pres']
    (TV [NUM='sg', TENSE='pres'] likes)
    (NP [NUM='pl'] (N [NUM='pl'] children))))
```

Terminology

So far, we have only seen feature values like *sg* and *pl*. These simple values are usually called **atomic** — that is, they can't be decomposed into subparts. A special case of atomic values are **boolean** values, that is, values that just specify whether a property is true or false of a category. For example, we might want to distinguish **auxiliary** verbs such as *can*, *may*, *will* and *do* with the boolean feature **AUX**. Then our lexicon for verbs could include entries such as (22). (Note that we follow the convention that boolean features are not written F +, F - but simply +F, -F, respectively.)

- (99) V[TENSE=pres, +AUX=+] → 'can'
 V[TENSE=pres, +AUX=+] → 'may'
 V[TENSE=pres, -AUX -] → 'walks'
 V[TENSE=pres, -AUX -] → 'likes'

We have spoken informally of attaching "feature annotations" to syntactic categories. A more general approach is to treat the whole category — that is, the non-terminal symbol plus the annotation — as a bundle of features. Consider, for example, the object we have written as (23).

- (100) N[NUM=sg]

The syntactic category N, as we have seen before, provides part of speech information. This information can itself be captured as a feature value pair, using POS to represent "part of speech":

- (101) [POS=N, NUM=sg]

In fact, we regard (24) as our "official" representation of a feature-based linguistic category, and (23) as a convenient abbreviation. A bundle of feature-value pairs is called a **feature structure** or an **attribute value matrix** (AVM). A feature structure that contains a specification for the feature POS is a **linguistic category**.

In addition to atomic-valued features, we allow features whose values are themselves feature structures. For example, we might want to group together agreement features (e.g., person, number and gender) as a distinguished part of a category, as shown in (25).

- (102)
- $$\begin{aligned} \text{POS} &= N \\ \text{AGR} &= \left[\begin{array}{l} \text{PER} = 3 \\ \text{NUM} = pl \\ \text{GND} = fem \end{array} \right] \end{aligned}$$

AVM as included figure:

| | | | | | | | |
|-----|--|-----|---|-----|----|-----|-----|
| POS | N | | | | | | |
| AGR | <table border="1"> <tbody> <tr> <td>PER</td><td>3</td></tr> <tr> <td>NUM</td><td>pl</td></tr> <tr> <td>GND</td><td>fem</td></tr> </tbody> </table> | PER | 3 | NUM | pl | GND | fem |
| PER | 3 | | | | | | |
| NUM | pl | | | | | | |
| GND | fem | | | | | | |

In this case, we say that the feature AGR has a **complex** value.

There is no particular significance to the *order* of features in a feature structure. So (25) is equivalent to (25).

$$(103) \quad \begin{aligned} \text{AGR} &= \left[\begin{array}{l} \text{NUM} = \text{pl} \\ \text{PER} = 3 \\ \text{GND} = \text{fem} \end{array} \right] \\ \text{POS} &= N \end{aligned}$$

Once we have the possibility of using features like AGR, we can refactor a grammar like [10.1](#) so that agreement features are bundled together. A tiny grammar illustrating this point is shown in [\(27\)](#).

$$(104) \quad \begin{aligned} S &\rightarrow NP[\text{AGR}=?n] VP[\text{AGR}=?n] \\ NP[\text{AGR}=?n] &\rightarrow \text{PROPN}[\text{AGR}=?n] \\ VP[\text{TENSE}=?t, \text{AGR}=?n] &\rightarrow \text{COP}[\text{TENSE}=?t, \text{AGR}=?n] \text{ Adj} \\ \text{COP}[\text{TENSE}=\text{pres}, \text{AGR}=[\text{NUM}=\text{sg}, \text{PER}=3]] &\rightarrow \text{'is'} \\ \text{PROPN}[\text{AGR}=[\text{NUM}=\text{sg}, \text{PER}=3]] &\rightarrow \text{'Kim'} \\ \text{Adj} &\rightarrow \text{'happy'} \end{aligned}$$

10.3 Computing with Feature Structures

In this section, we will show how feature structures can be constructed and manipulated in Python. We will also discuss the fundamental operation of unification, which allows us to combine the information contained in two different feature structures.

Feature Structures in Python

Feature structures are declared with the `FeatStruct()` constructor. Atomic feature values can be strings or integers.

```
>>> fs1 = nltk.FeatStruct(TENSE='past', NUM='sg')
>>> print fs1
[ NUM = 'sg' ]
[ TENSE = 'past' ]
```

A feature structure is actually just a kind of dictionary, and so we access its values by indexing in the usual way. We can use our familiar syntax to *assign* values to features:

```
>>> fs1 = nltk.FeatStruct(PER=3, NUM='pl', GND='fem')
>>> print fs1['GND']
fem
>>> fs1['CASE'] = 'acc'
```

We can also define feature structures that have complex values, as discussed earlier.

```
>>> fs2 = nltk.FeatStruct(POS='N', AGR=fs1)
>>> print fs2
[ CASE = 'acc' ]
[ AGR = [ GND = 'fem' ] ]
[ [ NUM = 'pl' ] ]
[ [ PER = 3 ] ]
[ ]
[ POS = 'N' ]
>>> print fs2['AGR']
[ CASE = 'acc' ]
[ GND = 'fem' ]
[ [ NUM = 'pl' ] ]
[ [ PER = 3 ] ]
>>> print fs2['AGR']['PER']
3
```

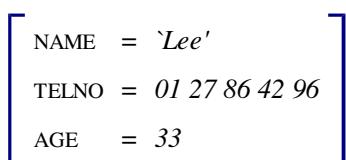
An alternative method of specifying feature structures is to use a bracketed string consisting of feature-value pairs in the format `feature=value`, where values may themselves be feature structures:

```
>>> nltk.FeatStruct("[POS='N', AGR=[PER=3, NUM='pl', GND='fem']]")
[AGR=[GND='fem', NUM='pl', PER=3], POS='N']
```

Feature Structures as Graphs

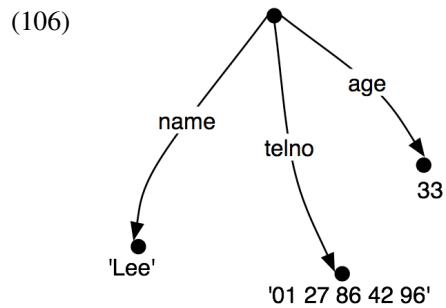
Feature structures are not inherently tied to linguistic objects; they are general purpose structures for representing knowledge. For example, we could encode information about a person in a feature structure:

```
>>> person01 = nltk.FeatStruct(name='Lee', telno='01 27 86 42 96', age=33)
```

(105) 

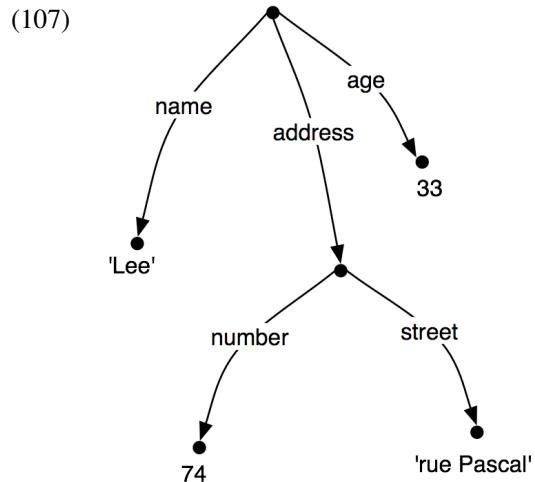
| | | |
|-------|---|----------------|
| NAME | = | 'Lee' |
| TELNO | = | 01 27 86 42 96 |
| AGE | = | 33 |

It is sometimes helpful to view feature structures as graphs; more specifically, **directed acyclic graphs** (DAGs). (29) is equivalent to the AVM (28).



The feature names appear as labels on the directed arcs, and feature values appear as labels on the nodes that are pointed to by the arcs.

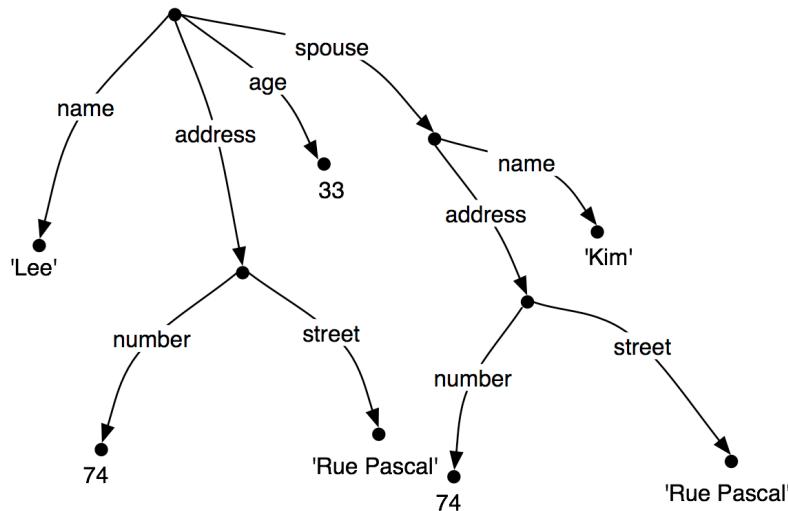
Just as before, feature values can be complex:



When we look at such graphs, it is natural to think in terms of paths through the graph. A **feature path** is a sequence of arcs that can be followed from the root node. We will represent paths as tuples. Thus, ('address', 'street') is a feature path whose value in (30) is the string "rue Pascal".

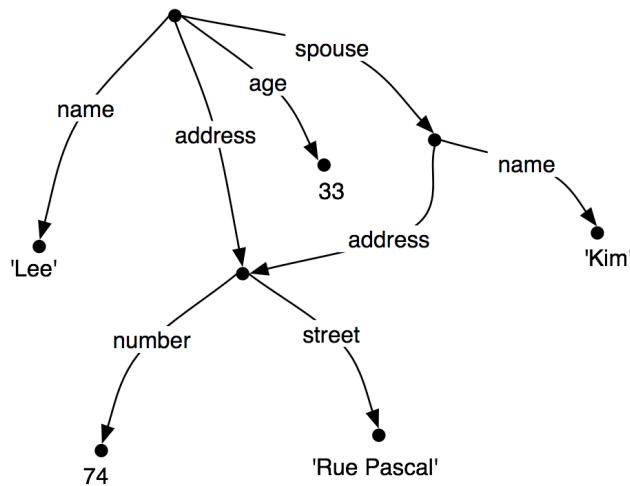
Now let's consider a situation where Lee has a spouse named "Kim", and Kim's address is the same as Lee's. We might represent this as (31).

(108)



However, rather than repeating the address information in the feature structure, we can "share" the same sub-graph between different arcs:

(109)



In other words, the value of the path (`'ADDRESS'`) in (32) is identical to the value of the path (`'SPOUSE'`, `'ADDRESS'`). DAGs such as (32) are said to involve **structure sharing** or **reentrancy**. When two paths have the same value, they are said to be **equivalent**.

There are a number of notations for representing reentrancy in matrix-style representations of feature structures. We adopt the following convention: the first occurrence of a shared feature structure is prefixed with an integer in parentheses, such as (1), and any subsequent reference to that structure uses the notation `->(1)`, as shown below.

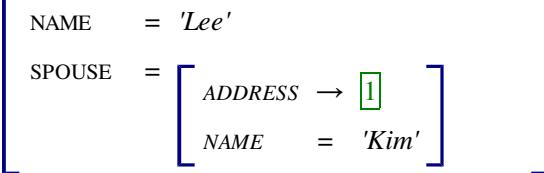
```

>>> fs = nltk.FeatStruct("""[NAME='Lee', ADDRESS=(1)[NUMBER=74, STREET='rue Pascal'],
... SPOUSE=[NAME='Kim', ADDRESS->(1)]]""")
>>> print fs
[ ADDRESS = (1) [ NUMBER = 74 ] ]
[ [ STREET = 'rue Pascal' ] ]
[ ]
[ [ NAME = 'Lee' ] ]
[ [ ] ]
[ [ SPOUSE = [ ADDRESS -> (1) ] ] ]
[ [ [ NAME = 'Kim' ] ] ]
  
```

This is similar to more conventional displays of AVMs, as shown in (33).

(110)

$$\text{ADDRESS} = \begin{bmatrix} \text{NUMBER} & = & 74 \\ \text{STREET} & = & 'rue Pascal' \end{bmatrix}^{\boxed{1}}$$



The bracketed integer is sometimes called a **tag** or a **coindex**. The choice of integer is not significant. There can be any number of tags within a single feature structure.

```
>>> fs1 = nltk.FeatStruct("[A='a', B=(1)[C='c'], D->(1), E->(1)]")
```

$$(111) \quad \left[\begin{array}{l} A = 'a' \\ B = \left[C = 'c' \right] 1 \\ D \rightarrow 1 \\ E \rightarrow 1 \end{array} \right]$$

Subsumption and Unification

It is standard to think of feature structures as providing **partial information** about some object, in the sense that we can order feature structures according to how general they are. For example, (112a) is more general (less specific) than (112b), which in turn is more general than (112c).

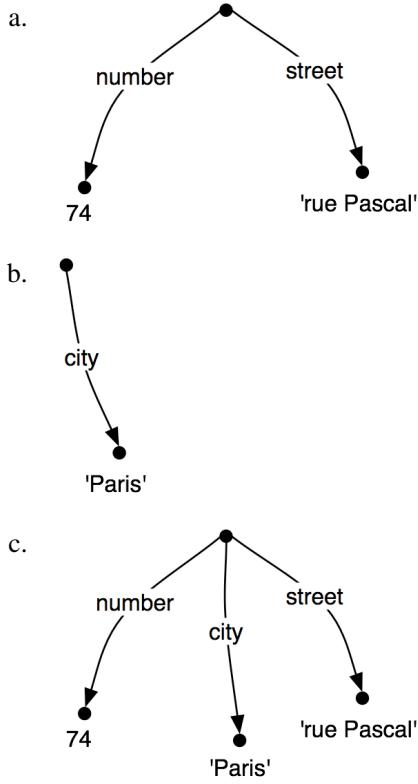
- $$(112) \quad \begin{array}{l} a. \quad \left[\text{NUMBER} = 74 \right] \\ b. \quad \left[\begin{array}{l} \text{NUMBER} = 74 \\ \text{STREET} = 'rue Pascal' \end{array} \right] \\ c. \quad \left[\begin{array}{l} \text{NUMBER} = 74 \\ \text{STREET} = 'rue Pascal' \\ \text{CITY} = 'Paris' \end{array} \right] \end{array}$$

This ordering is called **subsumption**; a more general feature structure **subsumes** a less general one. If FS_0 subsumes FS_1 (formally, we write $FS_0 \sqsubseteq FS_1$), then FS_1 must have all the paths and path equivalences of FS_0 , and may have additional paths and equivalences as well. Thus, (31) subsumes (32), since the latter has additional path equivalences.. It should be obvious that subsumption only provides a partial ordering on feature structures, since some feature structures are incommensurable. For example, (36) neither subsumes nor is subsumed by (112a).

$$(113) \quad \left[\text{TELNO} = 01\ 27\ 86\ 42\ 96 \right]$$

So we have seen that some feature structures are more specific than others. How do we go about specializing a given feature structure? For example, we might decide that addresses should consist of not just a street number and a street name, but also a city. That is, we might want to *merge* graph (114b) with (114a) to yield (114c).

(114)



Merging information from two feature structures is called **unification** and is supported by the `unify()` method.

```
>>> fs1 = nltk.FeatStruct(NUMBER=74, STREET='rue Pascal')
>>> fs2 = nltk.FeatStruct(CITY='Paris')
>>> print fs1.unify(fs2)
[ CITY  = 'Paris'      ]
[ NUMBER = 74          ]
[ STREET = 'rue Pascal' ]
```

Unification is formally defined as a binary operation: $FS_0 \sqcap FS_1$. Unification is symmetric, so

(115) $FS_0 \sqcap FS_1 = FS_1 \sqcap FS_0$.

The same is true in Python:

```
>>> print fs2.unify(fs1)
[ CITY  = 'Paris'      ]
[ NUMBER = 74          ]
[ STREET = 'rue Pascal' ]
```

If we unify two feature structures which stand in the subsumption relationship, then the result of unification is the most specific of the two:

(116) If $FS_0 \sqsubseteq FS_1$, then $FS_0 \sqcap FS_1 = FS_1$

For example, the result of unifying (112b) with (112c) is (112c).

Unification between FS_0 and FS_1 will fail if the two feature structures share a path π , but the value of π in FS_0 is a distinct atom from the value of π in FS_1 . This is implemented by setting the result of unification to be `None`.

```
>>> fs0 = nltk.FeatStruct(A='a')
>>> fs1 = nltk.FeatStruct(A='b')
>>> fs2 = fs0.unify(fs1)
>>> print fs2
```

None

Now, if we look at how unification interacts with structure-sharing, things become really interesting. First, let's define (31) in Python:

```
>>> fs0 = nltk.FeatStruct("""[NAME=Lee,
...                         ADDRESS=[NUMBER=74,
...                                   STREET='rue Pascal'],
...                         SPOUSE=[NAME=Kim,
...                                   ADDRESS=[NUMBER=74,
...                                             STREET='rue Pascal']]]]""")
```

(117)

$$\begin{aligned} \text{ADDRESS} &= \left[\begin{array}{l} \text{NUMBER} = 74 \\ \text{STREET} = \text{'rue Pascal'} \end{array} \right] \\ \text{NAME} &= \text{'Lee'} \\ \text{SPOUSE} &= \left[\begin{array}{l} \text{ADDRESS} = \left[\begin{array}{l} \text{NUMBER} = 74 \\ \text{STREET} = \text{'rue Pascal'} \end{array} \right] \\ \text{NAME} = \text{'Kim'} \end{array} \right] \end{aligned}$$

What happens when we augment Kim's address with a specification for CITY? (Notice that `fs1` includes the whole path from the root of the feature structure down to CITY.)

```
>>> fs1 = nltk.FeatStruct("[SPOUSE = [ADDRESS = [CITY = Paris]]]]")
```

(41) shows the result of unifying `fs0` with `fs1`:

(118)

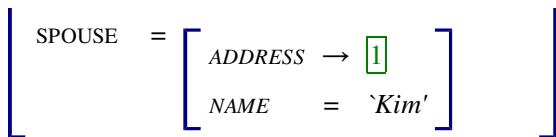
$$\begin{aligned} \text{ADDRESS} &= \left[\begin{array}{l} \text{NUMBER} = 74 \\ \text{STREET} = \text{'rue Pascal'} \end{array} \right] \\ \text{NAME} &= \text{'Lee'} \\ \text{SPOUSE} &= \left[\begin{array}{l} \text{ADDRESS} = \left[\begin{array}{l} \text{CITY} = \text{'Paris'} \\ \text{NUMBER} = 74 \\ \text{STREET} = \text{'rue Pascal'} \end{array} \right] \\ \text{NAME} = \text{'Kim'} \end{array} \right] \end{aligned}$$

By contrast, the result is very different if `fs1` is unified with the structure-sharing version `fs2` (also shown as (32)):

```
>>> fs2 = nltk.FeatStruct("""[NAME=Lee, ADDRESS=(1) [NUMBER=74, STREET='rue Pascal'],
...                         SPOUSE=[NAME=Kim, ADDRESS->(1)]]]""")
```

(119)

$$\begin{aligned} \text{ADDRESS} &= \left[\begin{array}{l} \text{CITY} = \text{'Paris'} \\ \text{NUMBER} = 74 \\ \text{STREET} = \text{'rue Pascal'} \end{array} \right] \\ \text{NAME} &= \text{'Lee'} \end{aligned}$$



Rather than just updating what was in effect Kim's "copy" of Lee's address, we have now updated *both* their addresses at the same time. More generally, if a unification involves specializing the value of some path π , then that unification simultaneously specializes the value of *any path that is equivalent to π* .

As we have already seen, structure sharing can also be stated using variables such as $?x$.

```
>>> fs1 = nltk.FeatStruct("[ADDRESS1=[NUMBER=74, STREET='rue Pascal']]")
>>> fs2 = nltk.FeatStruct("[ADDRESS1=?x, ADDRESS2=?x]")
>>> print fs2
[ ADDRESS1 = ?x ]
[ ADDRESS2 = ?x ]
>>> print fs2.unify(fs1)
[ ADDRESS1 = (1) [ NUMBER = 74 ] ]
[ [ STREET = 'rue Pascal' ] ]
[ ]
[ ADDRESS2 -> (1) ]
```

10.4 Extending a Feature-Based Grammar

Subcategorization

In [Chapter 8](#), we proposed to augment our category labels to represent different kinds of verb. We introduced labels such as IV and TV for intransitive and transitive verbs respectively. This allowed us to write productions like the following:

$$(120) \quad \begin{array}{l} VP \rightarrow IV \\ VP \rightarrow TV \ NP \end{array}$$

Although we know that IV and TV are two kinds of V, from a formal point of view IV has no closer relationship with TV than it does with NP. As it stands, IV and TV are just atomic nonterminal symbols from a CFG. This approach doesn't allow us to say anything about the class of verbs in general. For example, we cannot say something like "All lexical items of category V can be marked for tense", since *bark*, say, is an item of category IV, not V. A simple solution, originally developed for a grammar framework called Generalized Phrase Structure Grammar (GPSG), stipulates that lexical categories may bear a SUBCAT feature whose values are integers. This is illustrated in a modified portion of [10.1](#), shown in [\(44\)](#).

$$(121) \quad \begin{array}{l} VP[TENSE=?t, NUM=?n] \rightarrow V[SUBCAT=0, TENSE=?t, NUM=?n] \\ VP[TENSE=?t, NUM=?n] \rightarrow V[SUBCAT=1, TENSE=?t, NUM=?n] \ NP \\ VP[TENSE=?t, NUM=?n] \rightarrow V[SUBCAT=2, TENSE=?t, NUM=?n] \ Sbar \\ \\ V[SUBCAT=0, TENSE=pres, NUM=sg] \rightarrow 'disappears' \mid 'walks' \\ V[SUBCAT=1, TENSE=pres, NUM=sg] \rightarrow 'sees' \mid 'likes' \\ V[SUBCAT=2, TENSE=pres, NUM=sg] \rightarrow 'says' \mid 'claims' \\ \\ V[SUBCAT=0, TENSE=pres, NUM=pl] \rightarrow 'disappear' \mid 'walk' \\ V[SUBCAT=1, TENSE=pres, NUM=pl] \rightarrow 'see' \mid 'like' \\ V[SUBCAT=2, TENSE=pres, NUM=pl] \rightarrow 'say' \mid 'claim' \\ \\ V[SUBCAT=0, TENSE=past, NUM=?n] \rightarrow 'disappeared' \mid 'walked' \\ V[SUBCAT=1, TENSE=past, NUM=?n] \rightarrow 'saw' \mid 'liked' \\ V[SUBCAT=2, TENSE=past, NUM=?n] \rightarrow 'said' \mid 'claimed' \end{array}$$

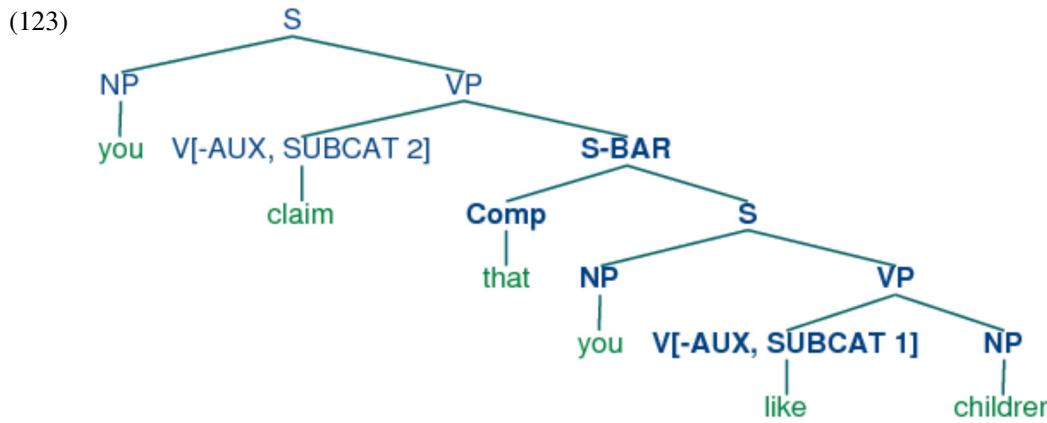
When we see a lexical category like $V[SUBCAT=1]$, we can interpret the SUBCAT specification as a pointer to the production in which $V[SUBCAT=1]$ is introduced as the head daughter in a VP production. By convention, there is a one-to-one correspondence between SUBCAT values and the productions that introduce lexical heads. It's worth noting that the choice of integer which acts as a value for SUBCAT is completely arbitrary — we could equally well have chosen 3999, 113 and 57 as our two values in [\(44\)](#). On this approach, SUBCAT can *only* appear on lexical categories; it makes no sense, for example, to specify a SUBCAT value on VP.

In our third class of verbs above, we have specified a category S-BAR. This is a label for subordinate clauses such as the complement of *claim* in the example *You claim that you like children*. We require two further productions to analyze such

sentences:

- (122) S-BAR → Comp S
Comp → 'that'

The resulting structure is the following.



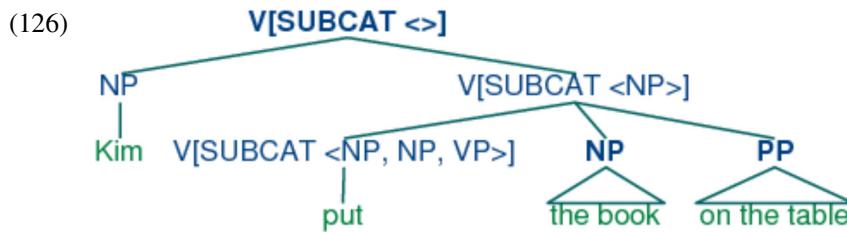
An alternative treatment of subcategorization, due originally to a framework known as categorial grammar, is represented in feature-based frameworks such as PATR and Head-driven Phrase Structure Grammar. Rather than using SUBCAT values as a way of indexing productions, the SUBCAT value directly encodes the valency of a head (the list of arguments that it can combine with). For example, a verb like *put* that takes NP and PP complements (*put the book on the table*) might be represented as (47):

- (124) v[SUBCAT {NP, NP, PP}]

This says that the verb can combine with three arguments. The leftmost element in the list is the subject NP, while everything else — an NP followed by a PP in this case — comprises the subcategorized-for complements. When a verb like *put* is combined with appropriate complements, the requirements which are specified in the SUBCAT are discharged, and only a subject NP is needed. This category, which corresponds to what is traditionally thought of as VP, might be represented as follows.

- (125) v[SUBCAT {NP}]

Finally, a sentence is a kind of verbal category that has *no* requirements for further arguments, and hence has a SUBCAT whose value is the empty list. The tree (126) shows how these category assignments combine in a parse of *Kim put the book on the table*.

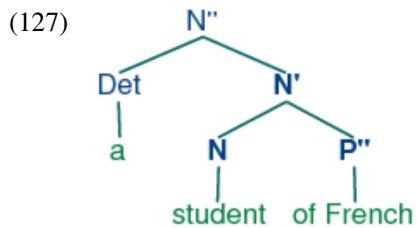


Heads Revisited

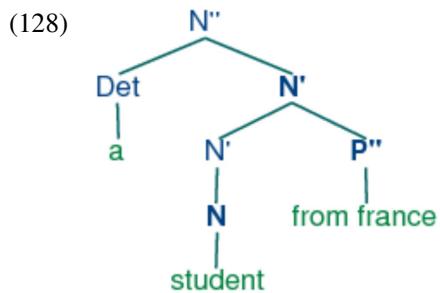
We noted in the previous section that by factoring subcategorization information out of the main category label, we could express more generalizations about properties of verbs. Another property of this kind is the following: expressions of category V are heads of phrases of category VP. Similarly (and more informally) Ns are heads of NPs, As (i.e., adjectives) are heads of APs, and Ps (i.e., adjectives) are heads of PPs. Not all phrases have heads — for example, it is standard to say that coordinate phrases (e.g., *the book and the bell*) lack heads — nevertheless, we would like our grammar formalism to express the mother / head-daughter relation where it holds. Now, although it looks as though there is something in common between, say, V and VP, this is more of a handy convention than a real claim, since V and VP formally have no more in common than V and DET.

X-bar syntax (cf. [Jacobs & Rosenbaum, 1970], [Jackendoff, 1977]) addresses this issue by abstracting out the notion of **phrasal level**. It is usual to recognize three such levels. If N represents the lexical level, then N' represents the next level up, corresponding to the more traditional category NOM, while N'' represents the phrasal level, corresponding to the category NP.

(The primes here replace the typographically more demanding horizontal bars of [Jacobs & Rosenbaum, 1970]). (50) illustrates a representative structure.



The head of the structure (50) is N while N' and N'' are called (**phrasal**) **projections** of N. N'' is the **maximal projection**, and N is sometimes called the **zero projection**. One of the central claims of X-bar syntax is that all constituents share a structural similarity. Using X as a variable over N, V, A and P, we say that directly subcategorized *complements* of the head are always placed as sisters of the lexical head, whereas *adjuncts* are placed as sisters of the intermediate category, X'. Thus, the configuration of the P'' adjunct in (51) contrasts with that of the complement P'' in (50).



The productions in (52) illustrate how bar levels can be encoded using feature structures.

- (129)
- $$\begin{aligned} S &\rightarrow N[\text{BAR}=2] \quad V[\text{BAR}=2] \\ N[\text{BAR}=2] &\rightarrow \text{DET } N[\text{BAR}=1] \\ N[\text{BAR}=1] &\rightarrow N[\text{BAR}=1] \quad P[\text{BAR}=2] \\ N[\text{BAR}=1] &\rightarrow N[\text{BAR}=0] \quad P[\text{BAR}=2] \end{aligned}$$

Auxiliary Verbs and Inversion

Inverted clauses — where the order of subject and verb is switched — occur in English interrogatives and also after 'negative' adverbs:

- (130)
- a. Do you like children?
 - b. Can Jody walk?

- (131)
- a. Rarely do you see Kim.
 - b. Never have I seen this dog.

However, we cannot place just any verb in pre-subject position:

- (132)
- a. *Like you children?
 - b. *Walks Jody?

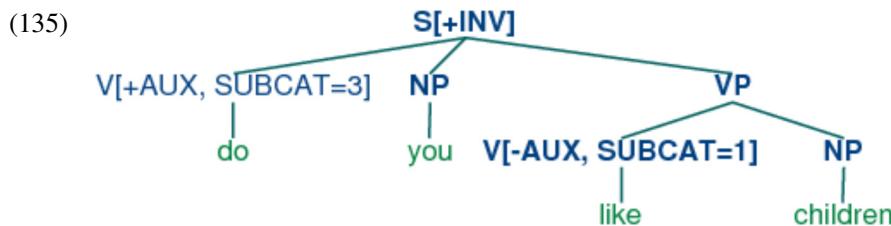
- (133)
- a. *Rarely see you Kim.

- b. *Never saw I this dog.

Verbs that can be positioned initially in inverted clauses belong to the class known as **auxiliaries**, and as well as *do*, *can* and *have* include *be*, *will* and *shall*. One way of capturing such structures is with the following production:

- (134) S [+inv] → V [+AUX] NP VP

That is, a clause marked as [+INV] consists of an auxiliary verb followed by a VP. (In a more detailed grammar, we would need to place some constraints on the form of the VP, depending on the choice of auxiliary.) (58) illustrates the structure of an inverted clause.



Unbounded Dependency Constructions

Consider the following contrasts:

- (136)
- a. You like Jody.
 - b. *You like.

- (137)
- a. You put the card into the slot.
 - b. *You put into the slot.
 - c. *You put the card.
 - d. *You put.

The verb *like* requires an NP complement, while *put* requires both a following NP and PP. Examples (59) and (60) show that these complements are *obligatory*: omitting them leads to ungrammaticality. Yet there are contexts in which obligatory complements can be omitted, as (61) and (62) illustrate.

- (138)
- a. Kim knows who you like.
 - b. This music, you really like.

- (139)
- a. Which card do you put into the slot?
 - b. Which slot do you put the card into?

That is, an obligatory complement can be omitted if there is an appropriate **filler** in the sentence, such as the question word *who* in (138a), the preposed topic *this music* in (138b), or the *wh* phrases *which card/slot* in (62). It is common to say that sentences like (61) – (62) contain **gaps** where the obligatory complements have been omitted, and these gaps are sometimes made explicit using an underscore:

- (140)
- a. Which card do you put __ into the slot?

- b. Which slot do you put the card into __?

So, a gap can occur if it is **licensed** by a filler. Conversely, fillers can only occur if there is an appropriate gap elsewhere in the sentence, as shown by the following examples.

(141)

- a. *Kim knows who you like Jody.
- b. *This music, you really like hip-hop.

(142)

- a. *Which card do you put this into the slot?
- b. *Which slot do you put the card into this one?

The mutual co-occurrence between filler and gap leads to (61) – (62) is sometimes termed a "dependency". One issue of considerable importance in theoretical linguistics has been the nature of the material that can intervene between a filler and the gap that it licenses; in particular, can we simply list a finite set of strings that separate the two? The answer is No: there is no upper bound on the distance between filler and gap. This fact can be easily illustrated with constructions involving sentential complements, as shown in (66).

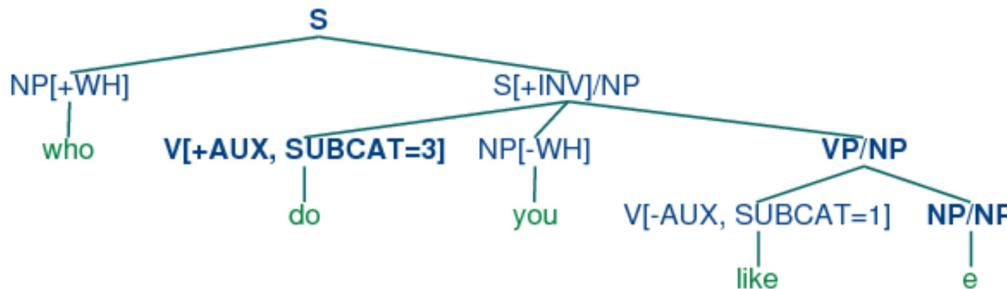
(143)

- a. Who do you like __?
- b. Who do you claim that you like __?
- c. Who do you claim that Jody says that you like __?

Since we can have indefinitely deep recursion of sentential complements, the gap can be embedded indefinitely far inside the whole sentence. This constellation of properties leads to the notion of an **unbounded dependency construction**; that is, a filler-gap dependency where there is no upper bound on the distance between filler and gap.

A variety of mechanisms have been suggested for handling unbounded dependencies in formal grammars; we shall adopt an approach due to Generalized Phrase Structure Grammar that involves something called **slash categories**. A slash category is something of the form Y/XP; we interpret this as a phrase of category Y that is missing a sub-constituent of category XP. For example, S/NP is an S that is missing an NP. The use of slash categories is illustrated in (67).

(144)



The top part of the tree introduces the filler *who* (treated as an expression of category NP[+WH]) together with a corresponding gap-containing constituent S/NP. The gap information is then "percolated" down the tree via the VP/NP category, until it reaches the category NP/NP. At this point, the dependency is discharged by realizing the gap information as the empty string *e* immediately dominated by NP/NP.

Do we need to think of slash categories as a completely new kind of object in our grammars? Fortunately, no, we don't — in fact, we can accommodate them within our existing feature-based framework. We do this by treating slash as a feature, and the category to its right as a value. In other words, our "official" notation for S/NP will be S[SLASH=NP]. Once we have taken this step, it is straightforward to write a small grammar for analyzing unbounded dependency constructions. 10.3 illustrates the main principles of slash categories, and also includes productions for inverted clauses. To simplify presentation, we have omitted any

specification of tense on the verbs.

```
>>> nltk.data.show_cfg('grammars/book_grammars/feat1.fcfg')
% start S
# #####
# Grammar Rules
# #####
S[-INV] -> NP_S/NP
S[-INV]/?x -> NP VP/?x
S[+INV]/?x -> V[+AUX] NP VP/?x
S-BAR/?x -> Comp S[-INV]/?x
NP/NP ->
VP/?x -> V[SUBCAT=1, -AUX] NP/?x
VP/?x -> V[SUBCAT=2, -AUX] S-BAR/?x
VP/?x -> V[SUBCAT=3, +AUX] VP/?x
# #####
# Lexical Rules
# #####
V[SUBCAT=1, -AUX] -> 'see' | 'like'
V[SUBCAT=2, -AUX] -> 'say' | 'claim'
V[SUBCAT=3, +AUX] -> 'do' | 'can'
NP[-WH] -> 'you' | 'children' | 'girls'
NP[+WH] -> 'who'
Comp -> 'that'
```

[Figure 10.3 \(slashcfg.py\)](#): Figure 10.3: Grammar for Simple Long-distance Dependencies

The grammar in [Figure 10.3](#) contains one gap-introduction production, namely

$$(145) \quad S[-INV] \rightarrow NP \ S/NP$$

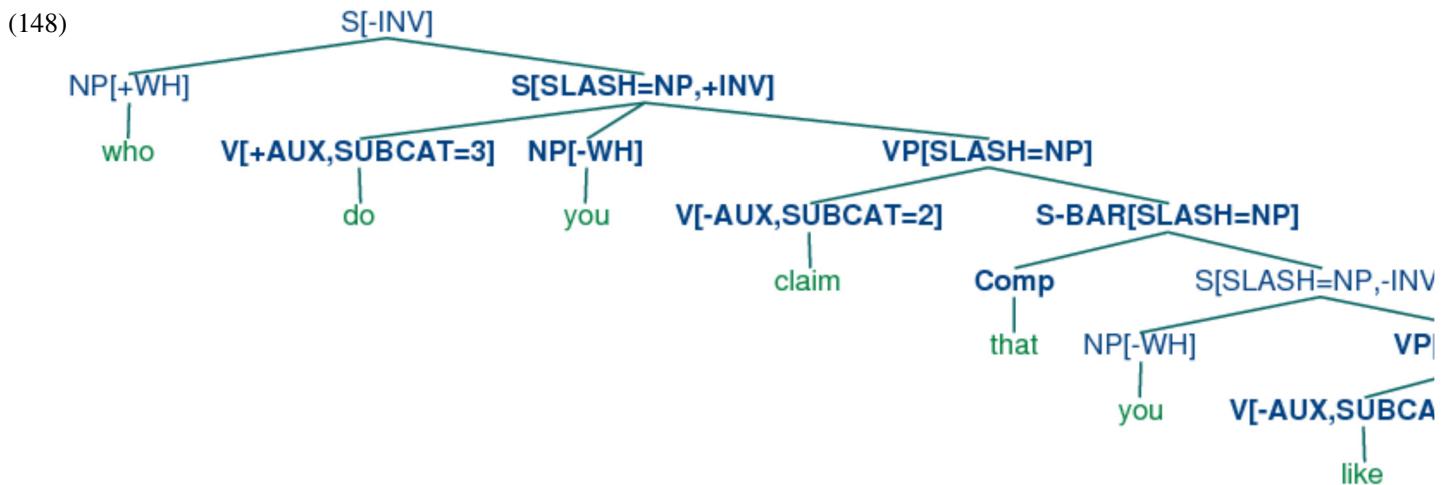
In order to percolate the slash feature correctly, we need to add slashes with variable values to both sides of the arrow in productions that expand S, VP and NP. For example,

$$(146) \quad VP/?x \rightarrow V \ S-BAR/?x$$

says that a slash value can be specified on the VP mother of a constituent if the same value is also specified on the S-BAR daughter. Finally, [\(147\)](#) allows the slash information on NP to be discharged as the empty string.

$$(147) \quad NP/NP \rightarrow$$

Using [10.3](#), we can parse the string *who do you claim that you like* into the tree shown in [\(71\)](#).



Case and Gender in German

Compared with English, German has a relatively rich morphology for agreement. For example, the definite article in German varies with case, gender and number, as shown in [Table 10.2](#).

Table 10.2:

Morphological Paradigm for the German definite Article

| Case | Masc | Fem | Neut | Plural |
|-------------|-------------|------------|-------------|---------------|
| <i>Nom</i> | der | die | das | die |
| <i>Gen</i> | des | der | des | der |
| <i>Dat</i> | dem | der | dem | den |
| <i>Acc</i> | den | die | das | die |

Subjects in German take the nominative case, and most verbs govern their objects in the accusative case. However, there are exceptions like *helfen* that govern the dative case:

(149)

- a. Die Katze sieht den Hund
the.NOM.FEM.SG cat.3.FEM.SG see.3.SG the.ACC.MASC.SG dog.3.MASC.SG
'the cat sees the dog'
- b. *Die Katze sieht dem Hund
the.NOM.FEM.SG cat.3.FEM.SG see.3.SG the.DAT.MASC.SG dog.3.MASC.SG
- c. Die Katze hilft dem Hund
the.NOM.FEM.SG cat.3.FEM.SG help.3.SG the.DAT.MASC.SG dog.3.MASC.SG
'the cat helps the dog'
- d. *Die Katze hilft den Hund
the.NOM.FEM.SG cat.3.FEM.SG help.3.SG the.ACC.MASC.SG dog.3.MASC.SG

The grammar [10.4](#) illustrates the interaction of agreement (comprising person, number and gender) with case.

```
>>> nltk.data.show_cfg('grammars/book_grammars/german.fcfg')
% start S
# Grammar Rules
S -> NP [CASE=nom, AGR=?a] VP [AGR=?a]
NP [CASE=?c, AGR=?a] -> PRO [CASE=?c, AGR=?a]
NP [CASE=?c, AGR=?a] -> Det [CASE=?c, AGR=?a] N [CASE=?c, AGR=?a]
VP [AGR=?a] -> IV [AGR=?a]
VP [AGR=?a] -> TV [OBJCASE=?c, AGR=?a] NP [CASE=?c]
# Lexical Rules
# Singular determiners
# masc
Det [CASE=nom, AGR=[GND=masc, PER=3, NUM=sg]] -> 'der'
Det [CASE=dat, AGR=[GND=masc, PER=3, NUM=sg]] -> 'dem'
Det [CASE=acc, AGR=[GND=masc, PER=3, NUM=sg]] -> 'den'
# fem
Det [CASE=nom, AGR=[GND=fem, PER=3, NUM=sg]] -> 'die'
Det [CASE=dat, AGR=[GND=fem, PER=3, NUM=sg]] -> 'der'
Det [CASE=acc, AGR=[GND=fem, PER=3, NUM=sg]] -> 'die'
# Plural determiners
Det [CASE=nom, AGR=[PER=3, NUM=pl]] -> 'die'
Det [CASE=dat, AGR=[PER=3, NUM=pl]] -> 'den'
Det [CASE=acc, AGR=[PER=3, NUM=pl]] -> 'die'
# Nouns
N [AGR=[GND=masc, PER=3, NUM=sg]] -> 'hund'
N [CASE=nom, AGR=[GND=masc, PER=3, NUM=pl]] -> 'hunde'
N [CASE=dat, AGR=[GND=masc, PER=3, NUM=pl]] -> 'hunden'
N [CASE=acc, AGR=[GND=masc, PER=3, NUM=pl]] -> 'hunde'
N [AGR=[GND=fem, PER=3, NUM=sg]] -> 'katze'
N [AGR=[GND=fem, PER=3, NUM=pl]] -> 'katzen'
# Pronouns
PRO [CASE=nom, AGR=[PER=1, NUM=sg]] -> 'ich'
PRO [CASE=acc, AGR=[PER=1, NUM=sg]] -> 'mich'
PRO [CASE=dat, AGR=[PER=1, NUM=sg]] -> 'mir'
PRO [CASE=nom, AGR=[PER=2, NUM=sg]] -> 'du'
PRO [CASE=nom, AGR=[PER=3, NUM=sg]] -> 'er' | 'sie' | 'es'
PRO [CASE=nom, AGR=[PER=1, NUM=pl]] -> 'wir'
PRO [CASE=acc, AGR=[PER=1, NUM=pl]] -> 'uns'
```

```

PRO [CASE=dat, AGR=[PER=1,NUM=pl]] -> 'uns'
PRO [CASE=nom, AGR=[PER=2,NUM=pl]] -> 'ihr'
PRO [CASE=nom, AGR=[PER=3,NUM=pl]] -> 'sie'

# Verbs
IV [AGR=[NUM=sg,PER=1]] -> 'komme'
IV [AGR=[NUM=sg,PER=2]] -> 'kommst'
IV [AGR=[NUM=sg,PER=3]] -> 'kommt'
IV [AGR=[NUM=pl, PER=1]] -> 'kommen'
IV [AGR=[NUM=pl, PER=2]] -> 'kommt'
IV [AGR=[NUM=pl, PER=3]] -> 'kommen'
TV [OBJCASE=acc, AGR=[NUM=sg,PER=1]] -> 'sehe' | 'mag'
TV [OBJCASE=acc, AGR=[NUM=sg,PER=2]] -> 'siehst' | 'magst'
TV [OBJCASE=acc, AGR=[NUM=sg,PER=3]] -> 'sieht' | 'mag'
TV [OBJCASE=dat, AGR=[NUM=sg,PER=1]] -> 'folge' | 'helfe'
TV [OBJCASE=dat, AGR=[NUM=sg,PER=2]] -> 'folgst' | 'hilfst'
TV [OBJCASE=dat, AGR=[NUM=sg,PER=3]] -> 'folgt' | 'hilft'

```

Figure 10.4 (germancfg.py): Figure 10.4: Example Feature-Based Grammar

As you will see, the feature OBJCASE is used to specify the case that the verb governs on its object.

10.5 Summary

- The traditional categories of context-free grammar are atomic symbols. An important motivation feature structures is to capture fine-grained distinctions that would otherwise require a massive multiplication of atomic categories.
- By using variables over feature values, we can express constraints in grammar productions that allow the realization of different feature specifications to be inter-dependent.
- Typically we specify fixed values of features at the lexical level and constrain the values of features in phrases to unify with the corresponding values in their daughters.
- Feature values are either atomic or complex. A particular sub-case of atomic value is the Boolean value, represented by convention as [+/- F].
- Two features can share a value (either atomic or complex). Structures with shared values are said to be re-entrant. Shared values are represented by numerical indices (or tags) in AVMs.
- A path in a feature structure is a tuple of features corresponding to the labels on a sequence of arcs from the root of the graph representation.
- Two paths are equivalent if they share a value.
- Feature structures are partially ordered by subsumption. FS_0 subsumes FS_1 when FS_0 is more general (less informative) than FS_1 .
- The unification of two structures FS_0 and FS_1 , if successful, is the feature structure FS_2 that contains the combined information of both FS_0 and FS_1 .
- If unification specializes a path π in FS , then it also specializes every path π' equivalent to π .
- We can use feature structures to build succinct analyses of a wide variety of linguistic phenomena, including verb subcategorization, inversion constructions, unbounded dependency constructions and case government.

10.6 Further Reading

For more examples of feature-based parsing with NLTK, please see the guides at <http://nltk.org/doc/guides/featgram.html>, <http://nltk.org/doc/guides/featstruct.html>, and <http://nltk.org/doc/guides/grammartestsuites.html>.

For an excellent introduction to the phenomenon of agreement, see [\[Corbett, 2006\]](#).

The earliest use of features in theoretical linguistics was designed to capture phonological properties of phonemes. For example, a sound like /b/ might be decomposed into the structure [+LABIAL, +VOICE]. An important motivation was to capture generalizations across classes of segments; for example, that /n/ gets realized as /m/ preceding any +LABIAL consonant. Within Chomskyan grammar, it was standard to use atomic features for phenomena like agreement, and also to capture generalizations across syntactic categories, by analogy with phonology. A radical expansion of the use of features in theoretical syntax was advocated by Generalized Phrase Structure Grammar (GPSG; [\[Gazdar, Klein, & and, 1985\]](#)), particularly in the use of features with complex values.

Coming more from the perspective of computational linguistics, [\[Dahl & Saint-Dizier, 1985\]](#) proposed that functional aspects of language could be captured by unification of attribute-value structures, and a similar approach was elaborated by [\[Grosz &](#)

[\[Stickel, 1983\]](#) within the PATR-II formalism. Early work in Lexical-Functional grammar (LFG; [\[Bresnan, 1982\]](#)) introduced the notion of an **f-structure** that was primarily intended to represent the grammatical relations and predicate-argument structure associated with a constituent structure parse. [\[Shieber, 1986\]](#) provides an excellent introduction to this phase of research into feature-based grammars.

One conceptual difficulty with algebraic approaches to feature structures arose when researchers attempted to model negation. An alternative perspective, pioneered by [\[Kasper & Rounds, 1986\]](#) and [\[Johnson, 1988\]](#), argues that grammars involve *descriptions* of feature structures rather than the structures themselves. These descriptions are combined using logical operations such as conjunction, and negation is just the usual logical operation over feature descriptions. This description-oriented perspective was integral to LFG from the outset (cf. [\[Huang & Chen, 1989\]](#), and was also adopted by later versions of Head-Driven Phrase Structure Grammar (HPSG; [\[Sag & Wasow, 1999\]](#)). A comprehensive bibliography of HPSG literature can be found at <http://www.cl.uni-bremen.de/HPSG-Bib/>.

Feature structures, as presented in this chapter, are unable to capture important constraints on linguistic information. For example, there is no way of saying that the only permissible values for NUM are *sg* and *pl*, while a specification such as [NUM=*masc*] is anomalous. Similarly, we cannot say that the complex value of AGR *must* contain specifications for the features PER, NUM and GND, but *cannot* contain a specification such as [SUBCAT=3]. **Typed feature structures** were developed to remedy this deficiency. To begin with, we stipulate that feature values are always typed. In the case of atomic values, the values just are types. For example, we would say that the value of NUM is the type *num*. Moreover, *num* is the most general type of value for NUM. Since types are organized hierarchically, we can be more informative by specifying the value of NUM is a **subtype** of *num*, namely either *sg* or *pl*.

In the case of complex values, we say that feature structures are themselves typed. So for example the value of AGR will be a feature structure of type *agr*. We also stipulate that all and only PER, NUM and GND are **appropriate** features for a structure of type *agr*. A good early review of work on typed feature structures is [\[Emele & Zajac, 1990\]](#). A more comprehensive examination of the formal foundations can be found in [\[Carpenter, 1992\]](#), while [\[Copestake, 2002\]](#) focuses on implementing an HPSG-oriented approach to typed feature structures.

There is a copious literature on the analysis of German within feature-based grammar frameworks. [\[Nerbonne, Netter, & Pollard, 1994\]](#) is a good starting point for the HPSG literature on this topic, while [\[M{"u}ller, 2002\]](#) gives a very extensive and detailed analysis of German syntax in HPSG.

Chapter 15 of [\[Jurafsky & Martin, 2008\]](#) discusses feature structures, the unification algorithm, and the integration of unification into parsing algorithms.

10.7 Exercises

1. ☀ What constraints are required to correctly parse strings like *I am happy* and *she is happy* but not **you is happy* or **they am happy*? Implement two solutions for the present tense paradigm of the verb *be* in English, first taking Grammar (13) as your starting point, and then taking Grammar (27) as the starting point.
2. ☀ Develop a variant of grammar 10.1 that uses a feature COUNT to make the distinctions shown below:

(150)

- a. The boy sings.
- b. *Boy sings.

(151)

- a. The boys sing.
- b. Boys sing.

(152)

- a. The boys sing.
- b. Boys sing.

(153)

- a. The water is precious.
- b. Water is precious.

3. ● Develop a feature-based grammar that will correctly describe the following Spanish noun phrases:

(154) un cuadro hermos-o
INDEF.SG.MASC picture beautiful-SG.MASC
'a beautiful picture'

(155) un-os cuadro-s hermos-os
INDEF- picture-PL beautiful-PL.MASC
'beautiful pictures'

(156) un-a cortina hermos-a
INDEF- curtain beautiful-SG.FEM
'a beautiful curtain'

(157) un-as cortina-s hermos-as
INDEF- curtain beautiful-PL.FEM
'beautiful curtains'

4. ● Develop a wrapper for the `earley_parser` so that a trace is only printed if the input string fails to parse.

5. ☀ Write a function `subsumes()` which holds of two feature structures `fs1` and `fs2` just in case `fs1` subsumes `fs2`.

6. ● Consider the feature structures shown in [Figure 10.5](#).

[XX]

NOTE: This example is somewhat broken -- nltk doesn't support reentrance for base feature values. (See email ~7/23/08 to the nltk-users mailing list for details.)

```

fs1 = nltk.FeatStruct("[A = (1)b, B= [C ->(1)]]")
fs2 = nltk.FeatStruct("[B = [D = d]]")
fs3 = nltk.FeatStruct("[B = [C = d]]")
fs4 = nltk.FeatStruct("[A = (1)[B = b], C->(1)]")
fs5 = nltk.FeatStruct("[A = [D = (1)e], C = [E -> (1)] ]")
fs6 = nltk.FeatStruct("[A = [D = (1)e], C = [B -> (1)] ]")
fs7 = nltk.FeatStruct("[A = [D = (1)e, F = (2)[], C = [B -> (1), E -> (2)] ]")
fs8 = nltk.FeatStruct("[A = [B = b], C = [E = [G = e]]]"]
fs9 = nltk.FeatStruct("[A = (1)[B = b], C -> (1)]")

```

[Figure 10.5 \(featstructures.py\): Figure 10.5](#)

Work out on paper what the result is of the following unifications. (Hint: you might find it useful to draw the graph structures.)

1. fs_1 and fs_2
2. fs_1 and fs_3
3. fs_4 and fs_5
4. fs_5 and fs_6
5. fs_7 and fs_8
6. fs_7 and fs_9

Check your answers using Python.

7. ● List two feature structures that subsume $[A=?x, B=?x]$.
8. ● Ignoring structure sharing, give an informal algorithm for unifying two feature structures.
9. ☀ Modify the grammar illustrated in [\(44\)](#) to incorporate a BAR feature for dealing with phrasal projections.
10. ☀ Modify the German grammar in [10.4](#) to incorporate the treatment of subcategorization presented in [10.4](#).
11. ● Extend the German grammar in [10.4](#) so that it can handle so-called verb-second structures like the following:
 (158) Heute sieht der hund die katze.
12. ★ Morphological paradigms are rarely completely regular, in the sense of every cell in the matrix having a different realization. For example, the present tense conjugation of the lexeme WALK only has two distinct forms: *walks* for the 3rd person singular, and *walk* for all other combinations of person and number. A successful analysis should not require redundantly specifying that 5 out of the 6 possible morphological combinations have the same realization. Propose and implement a method for dealing with this.
13. ★ So-called **head features** are shared between the mother and head daughter. For example, TENSE is a head feature that is shared between a VP and its head V daughter. See [\[Gazdar, Klein, & and, 1985\]](#) for more details. Most of the features we have looked at are head features — exceptions are SUBCAT and SLASH. Since the sharing of head features is predictable, it should not need to be stated explicitly in the grammar productions. Develop an approach that automatically accounts for this regular behavior of head features.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

[**11 Analyzing the Meaning of Sentences**](#)

[**11.1 Natural Language Understanding**](#)

So far, this book has concentrated on relatively concrete aspects of language, particularly on words, and the patterns in which they occur. In the area of *syntax*, we have studied the organization of words into parts of speech, and from there we moved on to how sequences of words are built up to make phrases and sentences.

By contrast, we have said little about the **semantics** of sentences, that is, about how language is used to convey meanings. However, meaning is a very slippery notion, and there is much debate and disagreement about what is involved in a

computational model of linguistic meaning. In this chapter, we will concentrate on one main approach to the semantics of sentences, and we will not attempt to incorporate any contribution from lexical semantics, even though this is vital for many practical tasks.

In order to pin down 'the meaning of meaning' more closely, let's suppose that you are a native speaker of English, and have started to learn German. I ask if you understand what (1) means:

- (159) Die Kamera gefällt Stefan.

If you know the meanings of the individual words in (1), and know how these meanings are combined to make up the meaning of the whole sentence, you might say that (1) means the same as (160a), or more idiomatically, (160b):

- (160)
- a. The camera pleases Stefan.
 - b. Stefan likes the camera.

Now, I will probably take this as evidence that you do grasp the meaning of (1). But isn't this because you have translated from one language, German, into another, namely English? If I didn't already understand English, we would be no further forward! Nevertheless, there are some important insights here, ones that we will return to shortly.

Here's another way of looking at things. Imagine there is a situation *s* where there are two entities, Stefan and a specific camera, say Agfa Isola Serial No. KX7750. In addition, there is a relation holding between the two entities, which we will call the *please* relation. If you understand the meaning of (1), then you know that it is true in situation *s*. In part, you know this because you know that *Die Kamera* refers to Agfa Isola Serial No. KX7750, *Stefan* refers to Stefan, and *geflaumlaulllt* refers to the *please* relation. You also know that the grammatical subject of *geflaumlaulllt* plays the role of the entity that causes pleasure, while the object plays the role of the entity who is pleased.

We have just presented two contrasting views of semantics. In the first case, we tried to get at the meaning of (1) by translating it into another language which we already understood. Moreover, we assumed that we could tell whether the translation was successful, in the sense of being able to say that (1) and (2) 'have the same meaning'. In the second case, rather than translating (1) into another sentence, we tried to say what it is *about* by relating it to a situation in the world. In both cases, we talked informally about how the meaning of the parts of (1) could be combined to make up the meaning of the whole. This idea about how meanings should be constructed is so fundamental that it has been given a special name, that is, the **Principle of Compositionality**. (See [Gleitman & Liberman, 1995] for this formulation.)

Principle of Compositionality:

The meaning of a whole is a function of the meanings of the parts and of the way they are syntactically combined.

So far, so good. But what about computational semantics? Can a computer understand the meaning of a sentence? And how could we tell if it did? This is similar to asking 'Can a computer think?' Alan Turing [Turing, 1950] famously proposed to answer this by examining the ability of a computer to hold sensible conversations with a human. Suppose you are having a chat session with a person and a computer, but you are not told at the outset which is which. If you cannot identify which of your partners is the computer after chatting to each of them, then the computer has successfully imitated a human. If a computer succeeds in passing itself off as human in this 'imitation game', then according to Turing, we should be prepared to say that the computer *can* think and can be said to be intelligent. So Turing sidestepped the question of somehow examining the internal states of a computer by instead using its *behaviour* as evidence of intelligence. By the same reasoning, we could argue that in order to say that a computer understands English, it just needs to behave as though it did. What is important here is not so much the specifics of Turing's imitation game, but rather the proposal to judge a capacity for natural language understanding in terms of observable behaviour on the part of the computer.

Querying a Database

There are a variety of tasks that a computer could perform which might be considered as involving some amount of natural language understanding. Right now, we will look at one in particular: we type in a query in English, and the computer responds with an appropriate (and if we are lucky, correct) answer. For example:

- (161)
- a. Which country is Athens in?

b. Greece.

Building software that allows this task to be carried out in a general way is extremely hard, but solving it for a particular case is close to trivial. We start off by assuming that we have data about cities and countries in a structured form. To be concrete, we will use a database table whose first few rows are shown in [Table 11.1](#).

Note

The data illustrated in [Table 11.1](#) is drawn from the Chat-80 system described by [\[Warren & Pereira, 1982\]](#), which will be discussed in more detail later in the chapter. Population figures are given in thousands, but note that the data used in these examples dates back at least to the 1980s, and was already somewhat out of date at the point when [\[Warren & Pereira, 1982\]](#) was published.

Table 11.1:

city_table: A table of cities, countries and populations

| City | Country | Population |
|------------|----------------|------------|
| athens | greece | 1368 |
| bangkok | thailand | 1178 |
| barcelona | spain | 1280 |
| berlin | east_germany | 3481 |
| birmingham | united_kingdom | 1112 |

The obvious way to retrieve answers from this tabular data involves writing queries in a database query language such as SQL.

Note

SQL (Structured Query Language) is a language designed for retrieving and managing data in relational databases. If you want to find out more about SQL, <http://www.w3schools.com/sql/> is a convenient online reference.

For example, executing the query (4) will pull out the value '`greece`':

(162) `SELECT Country FROM city_table WHERE City = 'athens'`

This specifies a result set consisting of all values for the column `Country` in data rows where the value of the `City` column is '`athens`'.

How can we get the same effect using English as our input to the query system? The framework offered by the feature-based grammar formalism developed in [Chapter 10](#) makes it easy to translate from English to SQL. The grammar `sql10.fcfg` illustrates how to assemble a meaning representation for a sentence in tandem with parsing the sentence. Each phrase structure rule is supplemented with a recipe for constructing a value for the feature `sem`. You can see that these recipes are extremely simple; in each case, we use the string concatenation operation `+` (see [Chapter 10](#) [NB give section]) to splice the values for the daughter constituents to make a value for the mother constituent.

```
>>> nltk.data.show_cfg('grammars/book_grammars/sql10.fcfg')
% start S
S[sem=(?np + WHERE + ?vp)] -> NP[sem=?np] VP[sem=?vp]
VP[sem=(?v + ?pp)] -> IV[sem=?v] PP[sem=?pp]
VP[sem=(?v + ?ap)] -> IV[sem=?v] AP[sem=?ap]
NP[sem=(?det + ?n)] -> Det[sem=?det] N[sem=?n]
PP[sem=(?p + ?np)] -> P[sem=?p] NP[sem=?np]
AP[sem=?pp] -> A[sem=?a] PP[sem=?pp]
NP[sem='Country="greece"'] -> 'Greece'
NP[sem='Country="china"'] -> 'China'
```

```

Det[sem='SELECT'] -> 'Which' | 'What'
N[sem='City FROM city_table'] -> 'cities'
IV[sem=''] -> 'are'
A -> 'located'
P[sem=''] -> 'in'

```

This allows us to parse a query into SQL.

```

>>> from nltk.parse import load_earley
>>> from string import join
>>> cp = load_earley('grammars/sql0.fcfg')
>>> query = 'What cities are located in China'
>>> trees = cp.nbest_parse(query.split())
>>> answer = trees[0].node['sem']
>>> q = join(answer)
>>> print q
SELECT City FROM city_table WHERE Country="china"

```

Note

Your Turn: Run the parser with maximum tracing on, i.e., `cp = load_earley('grammars/sql0.fcfg', trace=3)`, and examine how the values of `sem` are built up as complete edges are added to the chart.

Finally, we execute the query over the database `city.db` and retrieve some results.

```

>>> from nltk.sem import chat80
>>> rows = chat80.sql_query('samples/city.db', q)
>>> for r in rows: print "%s" % r,
canton chungking dairen harbin kowloon mukden peking shanghai sian tientsin

```

Let's summarize what we've just learned. We have constructed a task that involves the computer returning useful data in response to a natural language query, and we implemented this by translating a small subset of English into SQL. So to the extent that our software already understands SQL, in the sense of being able to execute SQL queries against a database, it also understands queries such as *What cities are located in China*. This parallels being able to translate from German into English as an example of natural language understanding.

The grammar `sql0.fcfg`, together with the NLTK Earley parser, is instrumental in carrying out the translation. How adequate is this grammar? On the positive side, it follows the Principle of Compositionality in assigning meaning to the parts of a sentence and then combining them to obtain a meaning for the whole. For example, *what cities* receives the meaning `SELECT City FROM city_table`. This approach can be extended to deal with a range of similar examples. However, there are also several less satisfactory aspects to `sql0.fcfg`. First off, we have hard-wired an embarrassing amount of detail about the database: we need to know the name of the relevant table (e.g., `city_table`) and the names of the fields. But our database could have contained exactly the same rows of data yet used a different table name and different field names, in which case the SQL queries would not be executable. Equally, we could have stored our data in a different format, such as XML, in which case retrieving the same results would require us to translate our English queries into an XML query language rather than SQL. These considerations suggest that we should be translating English into something that is more abstract and generic than something like SQL.

In order to sharpen the point, let's consider another English query and its translation:

(163)

- a. What cities are in China and have populations above 1,000,000?
- b. `SELECT City FROM city_table WHERE Country = 'china' AND Population > 1000`

Note

Your Turn: Extend the grammar `sql0.fcfg` so that it will translate (163a) into (163b), and check the values returned by the query.

You will probably find it easiest to first extend the grammar to handle queries like *What cities have populations above 1,000,000* before tackling conjunction. After you have had a go at this task, you can compare your solution to `grammars/sql1.fcfg` in the NLTK data distribution.

Observe that the *and* conjunction in (163a) is translated into an `AND` in the SQL counterpart, (163b). The latter tells us to select results from rows where two conditions are true together: the value of the `Country` column is '`china`' and the value of the `Population` column is greater than 1000. This interpretation for *and* invokes an idea we mentioned earlier, namely it talks about what is true in some particular situation. Moreover, although we haven't exhausted the meaning of *and* in English, we have given it a meaning which is independent of any query language. In fact, we have given it the standard interpretation from standard logic. In the following sections, we will explore an approach in which sentences of natural language are translated into logic instead of an executable query language. One advantage is that logical formalisms are more abstract and therefore more generic. If we wanted to, once we had our translation into logic, we could then translate it further into a task-specific query language. In fact, most serious attempts to query databases via natural language have adopted this methodology; cf. [Androulidakis, Ritchie, & Thanisch, 1995].

Natural Language and Logic

We have already touched on two fundamental notions in semantics. The first is that declarative sentences are *true or false in certain situations*. The second is that definite noun phrases and proper nouns *refer to things in the world*. To return to our earlier example, we suggested that (1) was true in a situation, where Stefan likes the camera in question, here illustrated in in Figure 11.1.

`./images/stefan_and_camera_with_arrows.png`

Figure 11.1

Once we have adopted the notion of truth in situation, we have a powerful tool for reasoning. In particular, we can look at sets of sentences, and ask whether they could be true together in some situation. For example, the sentences in (6) can be both true, while those in (7) and (8) cannot be. In other words, the sentences in (6) are **consistent**, while those in (7) and (8) are **inconsistent**.

(164)

- a. Sylvania is to the north of Freedonia.
- b. Freedonia is a republic.

(165)

- a. The capital of Freedonia has a population of 9,000.
- b. No city in Freedonia has a population of 9,000.

(166)

- a. Sylvania is to the north of Freedonia.
- b. Freedonia is to the north of Sylvania.

We have chosen sentences about fictional countries (featured in the Marx Brothers' 1933 movie *Duck Soup*) to emphasize that your ability to reason about these examples does *not* depend on what is true or false in the actual world. If you know the meaning of *no*, and that the capital of a country is a city in that country, then you should be able to conclude that the two sentences in (7) are inconsistent, regardless of where Freedonia is or what the population of its capital is. That is, there's no possible situation in which both sentences could be true. Similarly, if you know that the relation expressed by *to the north of* is asymmetric, then you should be able to conclude that the two sentences in (8) are inconsistent.

Broadly speaking, logic-based approaches to natural language semantics focus on those aspects of natural language which contribute to our judgments of consistency and inconsistency. The syntax of a logical language is designed to make these features formally explicit. As a result, determining properties like consistency can often be reduced to symbolic manipulation,

that is, to a task that can be carried out by a computer.

A **model** for a set Γ of sentences is a formal representation of a situation in which all the sentences in Γ are true. Standardly, models are represented in set theory. The domain D of discourse (all the entities we currently care about) is a set of individuals, while properties and relations are usually treated as sets built up from D . Let's look at a concrete example. Our domain D will consist of three children, Stefan, Klaus and Evi, represented respectively as s , k and e . We write this as $D = \{s, k, e\}$. The property *boy* is the set consisting of Stefan and Klaus, the property *girl* is the set consisting of Evi, and the property *is running* is the set consisting of Stefan and Evi. [11.2](#) is a graphical rendering of the model.

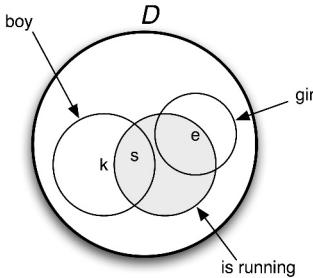


Figure 11.2: A picture of a model

[11.2 Propositional Logic](#)

A logical language is designed to capture aspects of natural language which contribute to determining whether a set of sentences is consistent. This in turn requires that a logical representation of a sentence S helps make the **truth-conditions** of a S explicit.

(167)

- a. [Shanghai is to the north of Guangzhou] and [the urban population of Shanghai is 15.5 million].
- b. [Klaus chased Evi] and [Evi ran away].

If we temporarily ignore the internal details of the sentences enclosed by square brackets, then both of them have the same structure:

(168) P and Q

We will sometimes say that [\(10\)](#) is the **logical form** of sentences like [\(9\)](#). We can state the truth-conditions for [\(10\)](#) along the following lines:

(169) A sentence of the form ' P and Q ' is true in a situation s if P is true in s and Q is true in s .

The inconsistency of [\(8\)](#) arises from the asymmetry of the relation *x is to the north of y* (where *x* and *y* are place-holders for noun phrases). Informally, we can state the following rule:

if *x* is to the north of *y* then *y* is not to the north of *x*.

In propositional logic, negation expressions such as *not* are treated as sentence operators. That is, the logical form of [\(170a\)](#) is something like [\(170b\)](#).

(170)

- a. Sylvania is not to the north of Freedonia.
- b. It is not the case that [Sylvania is to the north of Freedonia].

A more succinct version of [\(170b\)](#) is $\neg P$, where \neg symbolizes *it is not the case that*. A sentence of the form ' $\neg P$ ' is true in a situation s if P is not true in s . Consequently, we can make the following argument.

(171)

- [Freedonia is to the north of Sylvania].

Therefore, \neg [Sylvania is to the north of Freedonia].

So the two sentences in (8) can be augmented to the following set:

- (172)
- [Sylvania is to the north of Freedonia]
 - [Freedonia is to the north of Sylvania]
 - \neg [Sylvania is to the north of Freedonia]

Thus, the set in (14) contains a pair of sentences whose logical forms are P and $\neg P$. But a fundamental assumption of classical logic is that a sentence cannot be both true and false in a situation. Any set of sentences which contains both P and $\neg P$ is inconsistent. In this way, we can see how reducing sentences to their logical forms allows us to determine inconsistency.

Propositional logic is a formal system for systematically representing just those parts of linguistic structure which correspond to certain sentential conjunctions such as *and*, *or* and *if..., then...*. In the formalization of propositional logic, the counterparts of such conjunctions are sometimes called **boolean operators**. The basic expressions of propositional logic are **propositional symbols**, which we shall continue to write as P , Q , R , etc. There are varying conventions for representing boolean operators. Here's one choice: a unary operator \neg (*not*), and four binary operators \wedge (*and*), \vee (*or*), \rightarrow (*if..., then...*) and \equiv (*if and only if*). Then the set of **formulas** of propositional logic can be defined as follows:

1. Every propositional symbol is a formula.
2. If φ is a formula, then so is $\neg \varphi$.
3. If φ and ψ are formulas, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ ($\varphi \equiv \psi$).

The module `nltk.sem.logic` allows us to parse formulas of propositional logic. However, it only recognizes ASCII versions of the boolean operators:

```
>>> from nltk.sem import logic
>>> logic.boolean_ops()
negation      -
conjunction   &
disjunction   |
implication   ->
equivalence   <->
```

Within `logic`, `LogicParser()` parses logical expressions into various subclasses of `Expression`:

```
>>> lp = logic.LogicParser()
>>> lp.parse('-(P & Q)')
<NegatedExpression -(P & Q)>
>>> lp.parse('P & Q')
<AndExpression (P & Q)>
>>> lp.parse('P | (R -> Q)')
<OrExpression (P | (R -> Q))>
>>> lp.parse('P <-> --P')
<IffExpression (P <-> --P)>
```

Recall that we interpret sentences of a logical language relative to a model, which is a very simplified version of the world. A model for propositional logic needs to assign the values `True` or `False` to every possible formula. We do this inductively: first, every propositional symbol is assigned a value, and then we compute the value of complex formulas by consulting the meanings of the Boolean connectives and applying them to the values of the formula's components. A `Valuation` is a mapping from basic expressions of the logic to their values. Here's an example:

```
>>> val = nltk.sem.Valuation([('P', True), ('Q', True), ('R', False)])
```

We initialize a `Valuation` with a list of pairs, each of which consists of a semantic symbol and a semantic value. The resulting object is essentially just a dictionary that maps logical expressions (treated as strings) to appropriate values.

```
>>> val['P']
True
```

As we will see later, our models need to be somewhat more complicated in order to handle the more complex logical forms discussed in the next section; for the time being, just ignore the `dom` and `g` parameters in the following declarations.

```
>>> dom = set([])
>>> g = nltk.sem.Assignment(dom)
```

Now let's initialize a model `m` that uses `val`:

```
>>> m = nltk.sem.Model(dom, val)
```

Every instance of `Model` defines appropriate truth functions for the boolean connectives; these can be invoked to show how input values are mapped to output values. Here are the truth conditions for `&` and `->` (encoded via the functions `AND()` and `IMPLIES()` respectively):

```
>>> ops = [m.AND, m.IMPLIES]
>>> for o in ops:
...     print "===="
...     print o.func_name
...     for first in [True, False]:
...         for second in [True, False]:
...             print "%-5s %-5s => %s" % (first, second, o(first, second))
=====
AND
True  True  => True
True  False => False
False True  => False
False False => False
=====
IMPLIES
True  True  => True
True  False => False
False True  => True
False False => True
```

If you reflect on the truth conditions of `->`, you will see that it departs in many cases from our usual intuitions about the conditional in English. A formula of the form `P -> Q` is only false when `P` is true and `Q` is false. If `P` is false (say `P` corresponds to *The moon is made of green cheese*) and `Q` is true (say `Q`` corresponds to `Two plus two equals four`:`lx:`) then `~P -> Q` will come out true.

Every model comes with an `evaluate()` method, which will determine the semantic value of logical expression, such as formulas of propositional logic; of course, these values depend on the initial truth values we assigned to propositional symbols such as '`P`', '`Q`' and '`R`'.

```
>>> print m.evaluate('(P & Q)', g)
True
>>> print m.evaluate('-(P & Q)', g)
False
>>> print m.evaluate('(P & R)', g)
False
>>> print m.evaluate('(P | R)', g)
True
```

Note

Your Turn: Experiment with evaluating different formulas of propositional logic. Does the model give the values that you expected?

Up until now, we have treated the internal structure of sentences as being opaque to logical analysis. However, if we are to formalize arguments such as (13) in a way that allows us to compute with them, we have to be able look inside clauses. This is

what we turn to in the next section.

11.3 First-Order Logic

Syntax

In first-order logic, propositions are analyzed into predicates and arguments, which takes us a step closer to the structure of natural languages. The standard construction rules for FOL recognize **terms** such as individual variables and individual constants, and **predicates** which take differing numbers of arguments. For example, *Angus walks* might be formalized as *walk(angus)* and *Angus sees Bertie* as *see(angus, bertie)*. We will call *walk* a **unary predicate**, and *see* a **binary predicate**. The symbols used as predicates do not have intrinsic meaning, although it is hard to remember this. Returning to one of our earlier examples, there is no *logical* difference between (173a) and (173b).

(173)

- a. please(agfa_kx7750, stefan)
- b. gefällen(agfa_kx7750, stefan)

By itself, FOL has nothing substantive to say about lexical semantics — the meaning of individual words — although some theories of lexical semantics can be encoded in FOL. Whether an atomic predication like *see(angus, bertie)* is true or false in a situation is not a matter of logic, but depends on the particular valuation that we have chosen for the constants *see*, *angus* and *bertie*. For this reason, such expressions are called **non-logical constants**. By contrast, **logical constants** (such as the boolean operators) always receive the same interpretation in every model for FOL.

We should mention here that one binary predicate which has special status, and that is equality, as in formulas such as *angus = aj*. Equality is regarded as a logical constant, since for individual terms *t1* and *t2*, the formula *t1 = t2* is true if and only if *t1* and *t2* refer to one and the same entity.

It is often helpful to inspect the syntactic structure of expressions of FOL, and the usual way of doing this is to assign **types** to expressions. Following the tradition of Montague grammar [refs], we will use two **basic types**: *e* is the type of entities, while *t* is the type of formulas, i.e., expressions which have truth values. Given these two basic types, we can form **complex types** for functors, i.e., expressions which denote functions. That is, given any types σ and τ , $\langle \sigma, \tau \rangle$ is a complex type corresponding to functions from ' σ things' to ' τ things'. For example, $\langle e, t \rangle$ is the type of expressions from entities to truth values, namely unary predicates. The `LogicParser` can be invoked so that it carries out type checking.

```
>>> tlp = logic.LogicParser(type_check=True)
>>> parsed = tlp.parse('walk (angus)')
>>> parsed.argument
<ConstantExpression angus>
>>> parsed.argument.type
e
>>> parsed.function
<ConstantExpression walk>
>>> parsed.function.type
<e,t>
```

A binary predicate is given type $\langle e, \langle e, t \rangle \rangle$, that is, the type of something which combines with an argument of type *e* to make a unary predicate.

```
>>> parsed = tlp.parse('see(a, b)')
>>> parsed.function
<ApplicationExpression see(a)>
>>> parsed.function.type
<e,t>
>>> parsed.function.function
<ConstantExpression see>
>>> parsed.function.function.type
<e,<e,t>>
```

In FOL, arguments of predicates can also be individual variables such as *x*, *y* and *z*. Individual variables are similar to personal pronouns like *he*, *she* and *it*, in that we need to know about the context of use in order to figure out their denotation.

One way of interpreting the pronoun in (16) is by pointing to a relevant individual in the local context.

(174) He disappeared.

Another way is to supply a textual antecedent for the pronoun *he*, for example by uttering (175a) prior to (16). Here, we say that *she* is **coreferential** with the noun phrase *Cyril*. As a result, (16) is semantically equivalent to (175b).

(175)

- a. Cyril is Angus's dog.
- b. Cyril disappeared.

Consider by contrast the occurrence of *she* in (176a). In this case, it is **bound** by the indefinite NP *a poodle*, and this is a different relationship than coreference. If we replace the pronoun *she* by *a poodle*, the result (176b) is *not* semantically equivalent to (176a).

(176)

- a. Angus had a dog but he disappeared.
- b. Angus had a dog but a dog disappeared.

Corresponding to (177a), we can construct an **open formula** (177b) with two occurrences of the variable *x*. (We ignore tense to simplify exposition.)

(177)

- a. He is a dog and he disappeared.
- b. $\text{dog}(x) \wedge \text{disappear}(x)$

By placing an **existential quantifier** $\exists x$ ('for some *x*') in front of (177b), we can **bind** these variables, as in (178a), which means (178b) or, more idiomatically, (178c).

(178)

- a. $\exists x. \text{dog}(x) \wedge \text{disappear}(x)$
- b. At least one entity is a dog and disappeared.
- c. A dog disappeared.

The NLTK rendering of (178a) is (21).

(179) `exists x. (dog(x) & disappear(x))`

If all variable occurrences in a formula are bound, the formula is said to be **closed**. Any variables which are not bound in a formula are **free** in that formula. We mentioned before that the `parse()` method of `nltk.sem.logic`'s `LogicParser` returns objects of class `Expression`. As such, each expression `expr` comes with a method `free()` which returns the set of variables that are free in `expr`.

```
>>> print lp.parse('dog(cyril)').free()
set([])
>>> print lp.parse('dog(x)').free()
set([Variable('x')])
>>> print lp.parse('own(angus, cyril)').free()
set([])
>>> print lp.parse('exists x.dog(x)').free()
set([])
>>> print lp.parse(r'exists x.own(y, x)').free()
set([Variable('y')])
```

In addition to the existential quantifier, FOL offers us the **universal quantifier** $\forall x$ ('for all *x*'), illustrated in (22).

(180)

- a. $\forall x. \text{dog}(x) \rightarrow \text{disappear}(x)$
- b. Everything has the property that if it is a dog, it disappears.
- c. Every dog disappeared.

The NLTK rendering of (180a) is (23).

(181) `all x.(dog(x) -> disappear(x))`

One important property of (180a) often trips people up. The logical rendering in effect says that *if* something is a dog, then it runs away, but makes no commitment to the existence of dogs. So in a situation where there are no dogs, (180a) will still come out true. (Remember that '`p implies q`' is true when '`p`' is false.) Now you might argue that (180a) does presuppose the existence of dogs, and that the logic formalization is wrong. But it is possible to find other examples which lack such a presupposition. For instance, we might explain that the value of the Python expression `re.sub('ate', '8', astring)` is the result of replacing all occurrences of '`ate`' in `astring` by '`8`', even though there may in fact be no such occurrences.

In Section 1.3, we pointed out that mathematical set notation was a helpful method of specifying properties P of words that we wanted to select from a document. We illustrated this with (24), which we glossed as "the set of all w such that w is an element of V (the vocabulary) and w has property P ".

(182) $\{w \mid w \in V \& P(w)\}$

It turns out to be extremely useful to add something to FOL that will achieve the same effect. We do this with the **lambda operator** λx . The λ counterpart to (24) is (25). (Since we are not trying to do set-theory here, we just treat V as a unary predicate.)

(183) $\lambda w. (V(w) \wedge P(w))$

Note

λ expression were originally designed to represent computable functions and to provide a foundation for mathematics and logic. The theory in which λ expressions are studied is known as the λ -calculus.

λ is a binding operator, just as the FOL quantifiers are. If we have an open formula such as (26), then we can bind the variable x with the λ operator, as shown in (28).

(184)

- a. $(\text{walk}(x) \wedge \text{chew_gum}(x))$
- b. $\lambda x. (\text{walk}(x) \wedge \text{chew_gum}(x))$

(184b) is represented in NLTK as

(185) `\x. (walk(x) & chew_gum(x))`

Actually, this is a slight oversimplification, since `\` also happens to be a special character in Python. We could escape it (with another `\`), but in practice we will use raw strings, as illustrated here:

```
>>> e = lp.parse(r'\x. (walk(x) & chew_gum(x))')
>>> e
<LambdaExpression \x. (walk(x) & chew_gum(x))>
>>> e.free()
set([])
>>> print lp.parse(r'\x. (walk(x) & chew_gum(y))')
\x. (walk(x) & chew_gum(y))
```

We have a special name for the result of binding the variables in an expression: **lambda abstraction**. When one initially encounters λ -abstracts, it can be hard to get an intuitive sense of their meaning. A couple of English glosses for (184b) are: "be an x such that x walks and x chews gum" or "have the property of walking and chewing gum". Within the tradition of formal semantics of natural language, it has often been suggested that λ -abstracts are good representations for verb phrases (or subjectless clauses), particularly when these occur as arguments in their own right. This is illustrated in (186a) and its translation (186b).

(186)

- a. To walk and chew-gum is hard
- b. $\text{hard}(\lambda x. (\text{walk}(x) \wedge \text{chew_gum}(x)))$

So the general picture is this: given an open formula φ with free variable x , abstracting over x yields a property expression $\lambda x. \varphi$ — the property of being an x such that φ . (186b) illustrates a case where we say something about a property, namely that it is hard. But what we usually do with properties is attribute them to individuals. And in fact if φ is an open formula, then the abstract $\lambda x. \varphi$ can be used as a unary predicate. In (29), (184b) is predicated of the term *gerald*.

(187) $\lambda x. (\text{walk}(x) \wedge \text{chew_gum}(x)) (\text{gerald})$

Given the intended interpretation of (29), we would like it to be semantically equivalent to (30).

(188) $(\text{walk}(\text{gerald}) \wedge \text{chew_gum}(\text{gerald}))$

In fact, the equivalence of (29) and (30) follows from one of the axioms of the λ calculus, namely **beta-conversion**.

The rule of β -conversion

$$\lambda x. \alpha(t) = \alpha[t/x]$$

This rule can be read as follows: if we apply a λ -abstract $\lambda x. \alpha$ to a term t , the result is the same as removing the $\lambda x.$ operator and replacing all free occurrences of x in α by t . (This second operation is what is meant by the notation $\alpha[t/x]$.) So, for example, if we apply the rule of β -conversion to (29), we remove the initial ' $\lambda x.$ ' from $\lambda x. (\text{walk}(x) \wedge \text{chew_gum}(x))$ and then replace all free occurrences of x in $((\text{walk}(x) \wedge \text{chew_gum}(x))$ by the term *gerald*. β -conversion is effected in the `logic` module with the `simplify()` method.

```
>>> print lp.parse(r'\x. (\text{walk}(x) \wedge \text{chew\_gum}(x)) (\text{gerald})')
\x. (\text{walk}(x) \wedge \text{chew\_gum}(x)) (\text{gerald})
>>> print lp.parse(r'\x. (\text{walk}(x) \wedge \text{chew\_gum}(x)) (\text{gerald})').simplify()
(\text{walk}(\text{gerald}) \wedge \text{chew\_gum}(\text{gerald}))
```

Although we have so far only considered cases where the body of the λ abstract is an open formula, i.e., of type t , this is not a necessary restriction; the body can be any well-formed expression. Here's an example with two λ s.

(189) $\lambda x. \lambda y. (\text{poodle}(x) \wedge \text{own}(y, x))$

Just as (184b) plays the role of a unary predicate, (31) works like a binary predicate: it can be applied to two arguments. The `LogicParser` allows nested λ s to be written in abbreviated form, as illustrated in the second case below.

```
>>> print lp.parse(r'\x. \y. (\text{dog}(x) \wedge \text{own}(y, x)) (\text{cyril})').simplify()
\y. (\text{dog}(\text{cyril}) \wedge \text{own}(\text{y}, \text{cyril}))
>>> print lp.parse(r'\x y. (\text{dog}(x) \wedge \text{own}(y, x)) (\text{cyril}, \text{angus})').simplify()
(\text{dog}(\text{cyril}) \wedge \text{own}(\text{angus}, \text{cyril}))
```

Alphabetic Variants

When carrying out β -reduction, some care has to be taken with variables. Consider, for example, the λ terms (190a) and (190b), which differ only in the identity of a free variable.

(190)

- a. $\lambda y. \text{see}(y, x)$
- b. $\lambda y. \text{see}(y, z)$

Suppose now that we apply the λ -term $\lambda P. \exists x. P(x)$ to each of these terms:

(191)

- a. $\lambda P. \exists x. P(x) (\lambda y. see(y, x))$
- b. $\lambda P. \exists x. P(x) (\lambda y. see(y, z))$

In principle, the results of the application should be semantically equivalent. But if we let the free variable x in (190a) be 'captured' by the existential quantifier in (191a), then after reduction, the results will be different:

(192)

- a. $\exists x. see(x, x)$
- b. $\exists x. see(x, z)$

(192a) means there is some x that sees him/herself, whereas (192b) means that there is some x that sees an unspecified individual z . What has gone wrong here? Clearly, we want to forbid the kind of variable capture shown in (192a), and it seems that we have been too literal about the label of the particular variable bound by the existential quantifier in the functor expression of (191a). In fact, given any variable-binding expression (involving \forall , \exists or λ), the particular name chosen for the bound variable is completely arbitrary. For example, (193a) and (193b) are equivalent; they are called **α -equivalents** (or **alphabetic variants**).

(193)

- a. $\exists x. P(x)$
- b. $\exists y. P(y)$

The process of relabeling bound variables (which takes us from (193a) to (193b)) is known as **α -conversion**. When we test for equality of VariableBinderExpressions in the `logic` module (i.e., using `==`), we are in fact testing for α -equivalence:

```
>>> e1 = lp.parse('exists x.P(x)')
>>> print e1
exists x.P(x)
>>> e2 = e1.alpha_convert(nltk.sem.Variable('z'))
>>> print e2
exists z.P(z)
>>> e1 == e2
True
```

When β -reduction is carried out on an application $f(a)$, we check whether there are free variables in a which also occur as bound variables in any subterms of f . Suppose, as in the example discussed above, that x is free in a , and that f contains the subterm $\exists x. P(x)$. In this case, we produce an alphabetic variant of $\exists x. P(x)$, say, $\exists z. P(z)$, and then carry on with the reduction. This relabeling is carried out automatically by the β -reduction code in `logic`, and the results can be seen in the following example.

```
>>> e3 = lp.parse('\P.exists x.P(x) (\y.see(y, x))')
>>> print e3
(\P.exists x.P(x)) (\y.see(y, x))
>>> print e3.simplify()
exists z1.see(z1, x)
```

Summary

We'll take this opportunity to restate our earlier syntactic rules for propositional logic and add the formation rules for quantifiers. In addition, we make explicit the types of the expressions involved. We'll adopt the convention that $\langle e^n, t \rangle$ is the type of a predicate which combines with n arguments of type e to yield an expression of type t . In this case, we say that n is the **arity** of the predicate.

1. If P is a predicate of type $\langle e^n, t \rangle$, and t_1, \dots, t_n are terms of type e , then $P(t_1, \dots, t_n)$ is of type t .
2. If ϕ is of type t , then so is $\neg\phi$.
3. If ϕ and ψ are of type t , then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$ and $(\phi \equiv \psi)$.
4. If ϕ is of type t , and x is a variable of type e , then $\exists x.\phi$ and $\forall x.\phi$ are of type t .
5. If α is of type τ , and x is a variable of type e , then $\lambda x.\alpha$ is of type $\langle e, \tau \rangle$.

[Table 11.2](#) summarizes the logical constants of the `logic` module, and two of the methods of `Expressions`.

Table 11.2

| Example | Description |
|------------|------------------------------------|
| - | negation |
| & | conjunction |
| | disjunction |
| -> | implication |
| <-> | equivalence |
| = | equality |
| != | inequality |
| exists | existential quantifier |
| all | universal quantifier |
| \ | lambda operator |
| e.free() | show free variables of e |
| e.simplify | carry out β -conversion on e |

Model-checking

We have looked at the syntax of FOL, and in [Section 11.4](#) we will examine the task of translating English into FOL. Yet as we argued in [Section 11.1](#), this only gets us further forward if we can give a meaning to sentences of FOL. In other words, we need to give a *truth-conditional semantics* to FOL. From the point of view of computational semantics, there are obvious limits in how far one can push this approach. Although we want to talk about sentences being true or false in situations, we only have the means of representing situations in the computer in a symbolic manner. Despite this limitation, it is still possible to gain a clearer picture of truth-conditional semantics by encoding models in NLTK.

Given a first-order logic language L , a model M for L is a pair $\langle D, Val \rangle$, where D is a nonempty set called the **domain** of the model, and Val is a function called the **valuation function** which assigns values from D to expressions of L as follows:

1. For every individual constant c in L , $Val(c)$ is an element of D .
2. For every predicate symbol P of arity $n \geq 0$, $Val(P)$ is a function from D^n to $\{True, False\}$. (If the rank of P is 0, then $Val(P)$ is simply a truth value, the P is regarded as a propositional symbol.)

According to (2), if P is of arity n , then $Val(P)$ will be a function f from pairs of elements of D to $\{True, False\}$. In the models we shall build in NLTK, we'll adopt a more convenient alternative, in which $Val(P)$ is a set S of pairs, defined as follows:

$$(194) \quad S = \{s \mid f(s) = True\}$$

Such an f is called the **characteristic function** of S .

Relations are represented semantically in NLTK in the standard set-theoretic way: as sets of tuples. For example, let's suppose we have a domain of discourse consisting of the individuals Bertie, Olive and Cyril, where Bertie is a boy, Olive is a girl and Cyril is a dog. For mnemonic reasons, we use `b`, `o` and `c` as the corresponding labels in the model. We can declare the domain as follows:

```
>>> dom = set(['b', 'o', 'c'])
```

We will use the utility function `parse_valuation()` to convert a list of strings of the form `symbol => value` into a `Valuation` object.

```
>>> from nltk.sem import parse_valuation
>>> v = """
... bertie => b
```

```

... olive => o
... cyril => c
... boy => {b}
... girl => {o}
... dog => {c}
... walk => {o, c}
... see => {(b, o), (c, b), (o, c)}
...
"""
>>> val = parse_valuation(v)
>>> print val
{'bertie': 'b',
 'boy': set([('b',)]),
 'cyril': 'c',
 'dog': set([('c',)]),
 'girl': set([('o',)]),
 'olive': 'o',
 'see': set([(('o', 'c'), ('c', 'b'), ('b', 'o'))]),
 'walk': set([('c',), ('o',)])}

```

So according to this valuation, the value of `see` is a set of tuples such that Bertie sees Olive, Cyril sees Bertie, and Olive sees Cyril.

Note

Your Turn: Draw a picture of the domain of `m` and the sets corresponding to each of the unary predicates, by analogy with the diagram shown in [11.2](#).

You may have noticed that our unary predicates (i.e., `boy`, `girl`, `dog`) also come out as sets of singleton tuples, rather than just sets of individuals. This is a convenience which allows us to have a uniform treatment of relations of any arity. A predication of the form $P(\tau_1, \dots, \tau_n)$, where P is of arity n , comes out true just in case the tuple of values corresponding to (τ_1, \dots, τ_n) belongs to the set of tuples in the value of P .

```

>>> ('c', 'b') in val['see']
True
>>> ('b',) in val['boy']
True

```

Individual Variables and Assignments

In our models, the counterpart of a context of use is a variable **Assignment**. This is a mapping from individual variables to entities in the domain. Assignments are created using the `Assignment` constructor, which also takes the model's domain of discourse as a parameter. We are not required to actually enter any bindings, but if we do, they are in a $(variable, value)$ format similar to what we saw earlier for valuations.

```

>>> g = nltk.sem.Assignment(dom, [('x', 'o'), ('y', 'c')])
>>> g
{'y': 'c', 'x': 'o'}

```

In addition, there is a `print()` format for assignments which uses a notation closer to that often found in logic textbooks:

```

>>> print g
g[c/y][o/x]

```

Let's now look at how we can evaluate an atomic formula of FOL. First, we create a model, then we use the `evaluate()` method to compute the truth value.

```

>>> m = nltk.sem.Model(dom, val)
>>> m.evaluate('see(olive, y)', g)
True

```

What's happening here? Essentially, we are evaluating the formula `see('o', 'c')` just as in our earlier example. However,

when the interpretation function encounters the variable '`y`', rather than checking for a value in `val`, it asks the variable assignment `g` to come up with a value:

```
>>> g['y']
'c'
```

Since we already know that '`o`' and '`c`' stand in the *see* relation, the value `True` is what we expected. In this case, we can say that assignment `g` **satisfies** the formula '`see(olive, y)`'. By contrast, the following formula evaluates to `False` relative to `g` — check that you see why this is.

```
>>> m.evaluate('see(y, x)', g)
False
```

In our approach (though not in standard first-order logic), variable assignments are *partial*. For example, `g` says nothing about any variables apart from '`x`' and '`y`'. The method `purge()` clears all bindings from an assignment.

```
>>> g.purge()
>>> g
{}
```

If we now try to evaluate a formula such as '`see(olive, y)`' relative to `g`, it is like trying to interpret a sentence containing a *him* when we don't know what *him* refers to. In this case, the evaluation function fails to deliver a truth value.

```
>>> m.evaluate('see(olive, y)', g)
'Undefined'
```

Quantification and Scope

One of the crucial insights of modern logic is that the notion of variable satisfaction can be used to provide an interpretation to quantified formulas. Let's use [exists1](#) as an example.

When is it true? Let's think about all the individuals in our domain, i.e., in `dom`. We want to check whether any of these individuals have the property of being a girl and walking. In other words, we want to know if there is some `u` in `dom` such that `g[u/x]` satisfies the open formula [\(37\)](#).

(195) `girl(x) & walk(x)`

Consider the following:

```
>>> m.evaluate('exists x.(girl(x) & walk(x))', g)
True
```

`evaluate()` returns `True` here because there is some `u` in `dom` such that [\(37\)](#) is satisfied by an assignment which binds '`x`' to `u`. In fact, '`o`' is such a `u`:

```
>>> m.evaluate('girl(x) & walk(x)', g.add('x', 'o'))
True
```

One useful tool offered by NLTK is the `satisfiers()` method. This returns a set of all the individuals that satisfy an open formula. The method parameters are a parsed formula, a variable, and an assignment. Here are a few examples:

```
>>> fmla1 = lp.parse('girl(x) | boy(x)')
>>> m.satisfiers(fmla1, 'x', g)
set(['b', 'o'])
>>> fmla2 = lp.parse('girl(x) -> walk(x)')
>>> m.satisfiers(fmla2, 'x', g)
set(['c', 'b', 'o'])
>>> fmla3 = lp.parse('walk(x) -> girl(x)')
>>> m.satisfiers(fmla3, 'x', g)
set(['b', 'o'])
```

It's useful to think about why `fmla2` and `fmla3` receive the values they do. The truth conditions for \rightarrow mean that `fmla2` is

equivalent to `-girl(x) | walk(x)`, which is satisfied by something which either isn't a girl or walks. Since neither `b` (Bertie) nor `c` (Cyril) are girls, according to model `m`, they both satisfy the whole formula. And of course '`o`' satisfies the formula because '`o`' satisfies both disjuncts. Now, since every member of the domain of discourse satisfies `fmla2`, the corresponding universally quantified formula is also true.

```
>>> m.evaluate('all x.(girl(x) -> walk(x))', g)
True
```

In other words, a universally quantified formula $\forall x.\varphi$ is true with respect to `g` just in case for every u , φ is true with respect to `g[u/x]`.

Note

Your Turn: Try to figure out, first with pencil and paper, and then using `m.evaluate()`, what the truth values are for '`all x.(girl(x) & walk(x))`' and '`exists x.(boy(x) -> walk(x))`'. Make sure you understand why they receive these values.

Quantifier Scope Ambiguity

What happens when we want to give a formal representation of a sentence with *two* quantifiers, such as the following?

(196) Everybody admires someone.

There are (at least) two ways of expressing (38) in FOL:

- (197)
- a. `all x.(person(x) -> exists y.(person(y) & admire(x,y)))`
 - b. `exists y.(person(y) & all x.(person(x) -> admire(x,y)))`

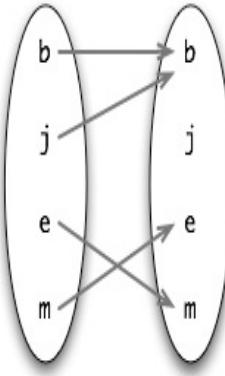
Can we use both of these? Then answer is Yes, but they have different meanings. (197b) is logically stronger than (197a): it claims that there is a unique person, say Bruce, who is admired by everyone. (197a), on the other hand, just requires that for every person u , we can find some person u' whom u admires; but this could be a different person u' in each case. We distinguish between (197a) and (197b) in terms of the **scope** of the quantifiers. In the first, \forall has wider scope than \exists , while in (197b), the scope ordering is reversed. So now we have two ways of representing the meaning of (38), and they are both quite legitimate. In other words, we are claiming that (38) is *ambiguous* with respect to quantifier scope, and the formulas in (39) give us a formal means of making the two readings explicit. However, we are not just interested in associating two distinct representations with (38). We also want to show in detail how the two representations lead to different conditions for truth in a formal model.

In order to examine the ambiguity more closely, let's fix our valuation as follows:

```
>>> from nltk.sem import parse_valuation
>>> v2 = """
... bruce => b
... cyril => c
... elspeth => e
... julia => j
... matthew => m
... person => {b, e, j, m}
... admire => {(j, b), (b, b), (m, e), (e, m), (c, a)}
...
>>> val2 = parse_valuation(v2)
```

We can use the graph in (40) to visualize the *admire* relation.

(198)



In (40), an arrow between two individuals x and y indicates that x admires y . So j and b both admire b (Bruce is very vain), while e admires m and m admires e . In this model, formula (197a) above is true but (197b) is false. One way of exploring these results is by using the `satisfiers()` method of `Model` objects.

```

>>> dom2 = val2.domain
>>> m2 = nltk.sem.Model(dom2, val2)
>>> g2 = nltk.sem.Assignment(dom2)
>>> fmla4 = lp.parse('(person(x) -> exists y.(person(y) & admire(x, y)))')
>>> m2.satisfiers(fmla4, 'x', g2)
set(['m', 'c', 'b', 'e', 'j'])

```

This shows that `fmla4` holds of every individual in the domain. By contrast, consider the formula `fmla5` below; this has no satisfiers for the variable y .

```

>>> fmla5 = lp.parse('(person(y) & all x.(person(x) -> admire(x, y)))')
>>> m2.satisfiers(fmla5, 'y', g2)
set([])
>>>

```

That is, there is no person that is admired by everybody. Taking a different open formula, `fmla6`, we can verify that there is a person, namely Bruce, who is admired by both Julia and Bruce.

```

>>> fmla6 = lp.parse('(person(y) & all x.((x = bruce | x = julia) -> admire(x, y)))')
>>> m2.satisfiers(fmla6, 'y', g2)
set(['b'])

```

Note

Your Turn: Devise a new model based on `m2` such that (197a) comes out false in your model; similarly, devise a new model such that (197b) comes out true.

11.4 The Semantics of English Sentences

Compositional Semantics in Feature-Based Grammar

At the beginning of the chapter we briefly illustrated a method of building semantic representations on the basis of a syntactic parse, using the grammar framework developed in [Chapter 10](#). This time, rather than constructing an SQL query, we will build a logical form. (41) illustrates a first approximation to the kind of analyses we would like to build.

(199) $S[\text{sem}=\langle \text{bark(cyril)} \rangle]$ 

In (41), the `sem` value at the root node shows a semantic representation for the whole sentence, while the `sem` values at lower nodes show semantic representations for constituents of the sentence. Since the values of `sem` have to be treated in special manner, they are distinguished from other feature values by being enclosed in angle brackets.

So far, so good, but how do we write grammar rules which will give us this kind of result? Our approach will be similar to that adopted for the grammar `sql10.fcfg` at the start of this chapter, in that we will assign semantic representations to lexical nodes, and then compose the semantic representations for phrases from those of the daughters. However, in the present case we will use function application rather than string concatenation as the mode of composition. To be more specific, suppose we have a NP and VP constituents with appropriate values for their `sem` nodes. Then the `sem` value of an S is handled by a rule like (42). (Observe that in the case where the value of `sem` is a variable, we omit the angle brackets.)

(200) $S[\text{sem} = \langle ?vp(\text{?np}) \rangle] \rightarrow \text{NP}[\text{sem}=?\text{subj}] \text{ VP}[\text{sem}=?\text{vp}]$

(42) tells us that given some `sem` value `?subj` for the subject NP and some `sem` value `?vp` for the VP, the `sem` value of the S mother is constructed by applying `?vp` as a functor to `?np`. From this, we can conclude that `?vp` has to denote a function which has the denotation of `?np` in its domain. (42) is a nice example of building semantics using the principle of compositionality.

To complete the grammar is very straightforward; all we require are the rules shown below.

```

VP [sem=?v] -> IV[sem=?v]
NP [sem=<cyril>] -> 'Cyril'
IV[sem=<bark>] -> 'barks'
  
```

The VP rule says that the mother's semantics is the same as the head daughter's. The two lexical rules just introduce non-logical constants to serve as the semantic values of *Cyril* and *barks* respectively.

Quantified NPs

You might be thinking this is all too easy — surely there is a bit more to building compositional semantics. What about quantifiers, for instance? Right, this is a crucial issue. For example, we want (201a) to be given the logical form in (201b). How can this be accomplished?

(201)

- a. A dog barks.
- b. $\exists x. (\text{dog}(x) \wedge \text{bark}(x))$

Let's make the assumption that our *only* operation for building complex semantic representations is function application. Then our problem is this: how do we give a semantic representation to the quantified NPs *a dog* so that it can be combined with '*bark*' to give the result in (201b)? As a first step, let's make the subject's `sem` value act as the functor rather than the argument. (This is sometimes called **type-raising**.) Now we are looking for way of instantiating `?np` so that `[sem=<?np(bark)>]` is equivalent to `[sem=<\exists x. (\text{dog}(x) \wedge \text{bark}(x))>]`. Doesn't this look a bit reminiscent of carrying out β -reduction in the λ -calculus? In other words, we want a λ term *M* to replace '`?np`' so that applying *M* to '*bark*' yields (201b). To do this, we replace the occurrence of '*bark*' in (201b) by a predicate variable '*P*', and bind the variable with λ , as shown in (44).

(202) $\lambda P. \exists x. (\text{dog}(x) \wedge P(x))$

We have used a different style of variable in (44) — that is '*P*' rather than '*x*' or '*y*' — to signal that we are abstracting over a different kind of thing — not an individual, but a function of type $\langle e, t \rangle$. So the type of (44) as a whole is $\langle \langle e, t \rangle, t \rangle$. We will take this to be the type of NPs in general. To illustrate further, a universally quantified NP will look like (45).

(203) $\lambda P. \forall x. (\text{dog}(x) \rightarrow P(x))$

We are pretty much done now, except that we also want to carry out a further abstraction plus application for the process of combining the semantics of the determiner *a* namely (47) with the semantics of *dog*.

(204) $\lambda Q. P.\exists x. (Q(x) \wedge P(x))$

Applying (47) as a functor to '*dog*' yields (44), and applying that to '*bark*' gives us $\lambda P.\exists x. (\text{dog}(x) \wedge P(x)) (\text{bark})$. Finally, carrying out β -reduction yields just what we wanted, namely (201b).

Transitive Verbs

Our next challenge is to deal with sentences containing transitive verbs, such as (47).

(205) Angus chase a dog.

The output semantics that we want to build is $\exists x. (\text{dog}(x) \wedge \text{chase}(\text{angus}, x))$. Let's look at how we can use λ -abstraction to get this result. A significant constraint on possible solutions is to require that the semantic representation of *a dog* be independent of whether the NP acts as subject or object of the sentence. In other words, we want to get the formula above as our output while sticking to (44) as the NP semantics. A second constraint is that VPs should have a uniform type of interpretation regardless of whether they consist of just an intransitive verb or a transitive verb plus object. More specifically, we stipulate that VPs are always of type $\langle e, t \rangle$. Given these constraints, here's a semantic representation for *owns a dog* which does the trick.

(206) $\lambda y. \exists x. (\text{dog}(x) \wedge \text{chase}(y, x))$

Think of (48) as the property of being a *y* such that for some dog *x*, *y* chases *x*; or more colloquially, being a *y* who chases a dog. Our task now resolves to designing a semantic representation for *chases* which can combine with (44) so as to allow (48) to be derived.

Let's carry out a kind of inverse β -reduction on (48), giving rise to (49).

(207) $\lambda P. \exists x. (\text{dog}(x) \wedge P(x)) (\lambda z. \text{chase}(y, z))$

(49) may be slightly hard to read at first; you need to see that it involves applying the quantified NP representation from (44) to ' $\lambda z. \text{chase}(y, z)$ '. (49) is equivalent via β -reduction to (48).

Now let's replace the functor in (49) by a variable '*X*' of the same type as an NP; that is, of type $\langle \langle e, t \rangle, t \rangle$.

(208) $X(\lambda z. \text{chase}(y, z))$

The representation of a transitive verb will have to apply to an argument of the type of '*X*' to yield a functor of the type of VPs, that is, of type $\langle e, t \rangle$. We can ensure this by abstracting over both the '*X*' variable in (50) and also the subject variable '*y*'. So the full solution is reached by giving *chases* the semantic representation shown in (51).

(209) $\lambda X y. X(\lambda x. \text{chase}(y, x))$

If (51) is applied to (44), the result after β -reduction is equivalent to (48), which is what we wanted all along:

```
>>> tvp = lp.parse(r'\X x.X(\y.chase(x,y))')
>>> np = lp.parse(r'(\P.\exists x. (\text{dog}(x) \wedge P(x)))')
>>> vp = logic.ApplicationExpression(tvp, np)
>>> print vp
(\X x.X(\y.chase(x,y))) (\P.\exists x. (\text{dog}(x) \wedge P(x)))
>>> print vp.simplify()
\X.\exists x1. (\text{dog}(z1) \wedge \text{chase}(x, z1))
```

In order to build a semantic representation for a sentence, we also need to combine in the semantics of the subject NP. If the latter is a quantified expression like *every girl*, everything proceeds in the same way as we showed for *a dog barks* earlier on; the subject is translated as a functor which is applied to the semantic representation of the VP. However, we now seem to have created another problem for ourselves with proper names. So far, these have been treated semantically as individual constants, and these cannot be applied as functors to expressions like (48). Consequently, we need to come up with a different semantic representation for them. What we do in this case is re-interpret proper names so that they too are functors, like quantified NPs. Here is the required λ expression for *Angus*.

(210) \P.P(angus)

(52) denotes the characteristic function corresponding to the set of all properties which are true of Angus. Converting from an individual constant to an expression like (51) is known as **type raising**, and allows us to flip functors with arguments. That is, type raising means that we can replace a Boolean-valued application such as $f(a)$ with an equivalent application $\lambda P.P(a)(f)$.

The grammar `sem3.fcfg` contains a small set of rules for parsing and translating simple examples of the kind that we have been looking at. Here's a slightly more complicated example.

```
>>> parser = load_earley('grammars/sem3.fcfg', trace=0)
>>> sentence = 'Angus gives a bone to every dog'
>>> tokens = sentence.split()
>>> trees = parser.nbest_parse(tokens)
>>> for tree in trees:
...     print tree.node['sem']
all z2.(dog(z2) -> exists z1.(bone(z1) & give(angus,z1,z2)))
```

NLTK provides some utilities to make it easier to derive and inspect semantic interpretations. The function `text_interpret()` is intended for batch interpretation of a list of input sentences. It builds a dictionary `d` where for each sentence `sent` in the input, `d[sent]` is a list of pairs (`synrep, semrep`) consisting of trees and semantic representations for `sent`. The value itself is a list, since `sent` may be syntactically ambiguous; in the following example, however, there is only one parse tree per sentence in the list.

```
>>> sents = ['Irene walks', 'Cyril bites an ankle']
>>> results = nltk.sem.text_interpret(sents, 'grammars/sem3.fcfg')
>>> for sent in sents:
...     for (synrep, semrep) in results[sent]:
...         print synrep
(S[sem=<walk(irene)>]
 (NP[-loc, num='sg', sem=<\P.P(irene)>]
  (PropN[-loc, num='sg', sem=<\P.P(irene)>] Irene))
 (VP[num='sg', sem=<\x.walk(x)>]
  (IV[num='sg', sem=<\x.walk(x)>, tns='pres'] walks)))
(S[sem=<exists z3.(ankle(z3) & bite(cyril,z3))>]
 (NP[-loc, num='sg', sem=<\P.P(cyril)>]
  (PropN[-loc, num='sg', sem=<\P.P(cyril)>] Cyril))
 (VP[num='sg', sem=<\x.exists z3.(ankle(z3) & bite(x,z3))>]
  (TV[num='sg', sem=<\X x.X(\y.bite(x,y))>, tns='pres'] bites)
 (NP[num='sg', sem=<\Q.exists x.(ankle(x) & Q(x))>]
  (Det[num='sg', sem=<\P Q.exists x.(P(x) & Q(x))>] an)
 (Nom[num='sg', sem=<ankle>] (N[num='sg', sem=<ankle>] ankle))))))
```

We have seen now how to convert English sentences into logical forms, and earlier we saw how logical forms could be checked as true or false in a model. Putting these two mappings together, we can check the truth value of English sentences in a given model. Let's take model `m` as defined above. The utility `text_evaluate()` resembles `text_interpret()` except that we need to pass a model and a variable assignment as parameters. The output is a triple `(synrep, semrep, value)` where `synrep, semrep` are as before, and `value` is a truth value.

```
>>> sent = 'Cyril sees every boy'
>>> results = nltk.sem.text_evaluate([sent], 'grammars/sem3.fcfg', m, g)
>>> for (syntree, semrel, value) in results[sent]:
...     print semrep
...     print value
exists z3.(ankle(z3) & bite(cyril,z3))
True
```

A more extensive example of evaluating English sentences in a model can be found in [howto Chat80 REF].

Quantifier Ambiguity Revisited

One important limitation of our approach so far is that it does not deal with scope ambiguity. Instead, quantifier scope ordering directly reflects scope in the parse tree. As a result, a sentence like (38), repeated here, will always be translated as (212a), not (212b).

(211) Every girl chases a dog.

(212)

- a. $\text{all } x.(\text{girl}(x) \rightarrow \text{exists } y.(\text{dog}(y) \wedge \text{chase}(x, y)))$
- b. $\text{exists } y.\text{dog}(y) \wedge \text{all } x.(\text{girl}(x) \rightarrow \text{chase}(x, y))$

There are numerous approaches to dealing with scope ambiguity, and we will look very briefly at one of the simplest. To start with, let's briefly consider the structure of scoped formulas. [Figure 11.3](#) depicts the way in which the two readings of (53) differ.

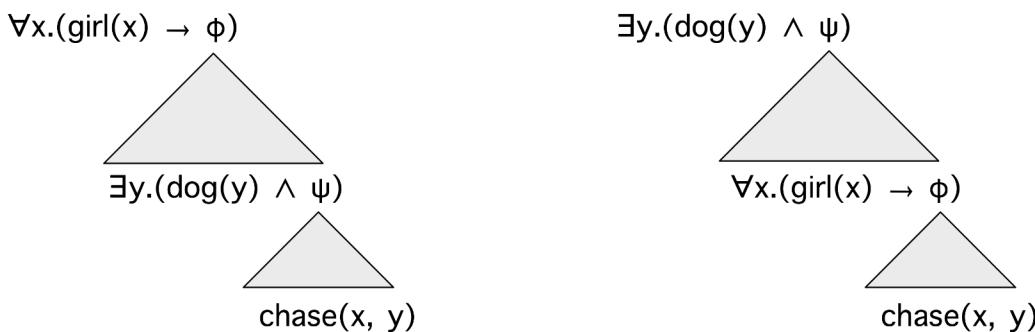


Figure 11.3: Quantifier Scopings

Let's consider the lefthand structure first. At the top, we have the quantifier corresponding to *every girl*. The ϕ can be thought of as a placeholder for whatever is inside the scope of the quantifier. Moving downwards, we see that we can plug in the quantifier corresponding to *a dog* as an instantiation of ϕ . This gives a new placeholder ψ , representing the scope of *a dog*, and into this we can plug the 'core' of the semantics, namely the open sentence corresponding to *x chases y*. The structure on righthand is identical, except we have swapped round the order of the two quantifiers.

In the approach known as **Cooper storage**, a semantic representation is no longer an expression of FOL, but instead a pair consisting of a 'core' semantic representation plus a list of **binding operators**. For the moment, think of a binding operator as being identical to the semantic representation of a quantified NP such as (45) or (46). Following along the lines indicated in [Figure 11.3](#), let's assume that we have constructed a Cooper-storage style semantic representation of [sentence \(211\)](#), and let's take our core to be the open formula `chase(x, y)`. Given a list of binding operators corresponding to the two NPs in (53), we pick a binding operator off the list, and combine it with the core.

```
\P.exists y. (dog(y) & P(y)) (\z2.chase(z1, z2))
```

Then we take the result, and apply the next binding operator from the list to it.

```
\P.all x. (girl(x) -> P(x)) (\z1.exists x. (dog(x) & chase(z1, x)))
```

Once the list is empty, we have a conventional logical form for the sentence. Combining binding operators with the core in this way is called **S-Retrieval**. If we are careful to allow every possible order of binding operators (for example, by taking all permutation orders of the list), then we will be able to generate every possible scope ordering of quantifiers.

The next question to address is how we build up a core+store representation compositionally. As before, each phrasal and lexical rule in the grammar will have a `sem` feature, but now there will be embedded features `core` and `store`. To illustrate the machinery, we will examine a simpler example, namely *Cyril smiles*. Here's a lexical rule for the verb *smiles* (taken from the grammar `storage.fcfg`) which looks pretty innocuous.

```
IV[sem=[core=<\x.smile(x)>, store=()]] -> 'smiles'
```

The rule for the proper name *Cyril* is more complex.

```
NP[sem=[core=<@x>, store=<bo(\P.P(cyril), @x)>]] -> 'Cyril'
```

The `bo` predicate has two subparts: the standard (type-raised) representation of a proper name, and the expression `@x`, which is called the **address** of the binding operator. (We'll explain the need for the address variable shortly.) `@x` is a metavariable, that is, a variable that ranges over individual variables of the logic and, as you will see, it also constitutes the content of the value of `core`. The rule for `VP` just percolates up the semantics of the `IV`, and most of the interesting work is done by the `S` rule.

```
VP[sem=?s] -> IV[sem=?s]
```

```
S[sem=[core=<?vp (?subj)>, store=(?b1+?b2)] ] ->
NP[sem=[core=?subj, store=?b1]] VP[sem=[core=?vp, store=?b2] ]
```

The `core` value at the `s` node is the result of applying the `VP`'s `core` value, namely `\x.smile(x)`, to the subject `NP`'s value. The latter will not be `@x`, but rather an instantiation of `@x`, say `z3`. After β -reduction, `<?vp (?subj)>` will be unified with `<smile(z3)>`. Now, when `@x` is instantiated as part of the parsing process, it will be instantiated uniformly. In particular, the occurrence of `@x` in the subject `NP`'s `store` will also be mapped to `z3`, yielding the element `bo(\P.P(cyril), z3)`. These steps can be seen in the following parse tree.

```
(S[sem=[core=<smile(z3)>, store=(bo(\P.P(cyril),z3))]]
 (NP[sem=[core=<z3>, store=(bo(\P.P(cyril),z3))]] Cyril)
 (VP[sem=[core=<\x.smile(x)>, store=()]]
 (IV[sem=[core=<\x.smile(x)>, store=()]] smiles)))
```

Let's move on to our more complex example, (53), and see what the storage style `sem` value is, after parsing with grammar `storage.fcfg`.

```
core = <chase(z3,z4)>
store = (bo(\P.all x.(girl(x) -> P(x)),z3), bo(\P.exists x.(dog(x) & P(x)),z4))
```

It should be clearer now why the address variables are an important part of the binding operator. Recall that during S-retrieval, we will be taking binding operators off the `store` list and applying them successively to the core. Suppose we start with `bo(\P.all x.(girl(x) -> P(x)),z3)`, which we want to combine with `chase(z3,z4)`. The quantifier part of binding operator is `\P.all x.(girl(x) -> P(x))`, and to combine this with `chase(z3,z4)`, the latter needs to first turned into a λ -abstract. How do we know which variable to abstract over? This is what the address `z3` tells us; i.e. that *every girl* has the role of chaser rather than chasee.

The module `nltk.sem.cooper_storage` deals with the task of turning storage-style semantic representations into standard logical forms. First, we construct a `CooperStore` instance, and inspect its `store` and `core`.

```
>>> from nltk.sem import cooper_storage as cs
>>> sentence = 'every girl chases a dog'
>>> trees = cs.parse_with_bindops(sentence, grammar='grammars/storage.fcfg')
>>> semrep = trees[0].node['sem']
>>> cs_semrep = cs.CooperStore(semrep)
>>> print cs_semrep.core
chase(z3,z4)
>>> for bo in cs_semrep.store:
...     print bo
bo(\P.all x.(girl(x) -> P(x)),z3)
bo(\P.exists x.(dog(x) & P(x)),z4)
```

Finally we call `s_retrieve()` and check the readings.

```
>>> cs_semrep.s_retrieve(trace=True)
Permutation 1
  (\P.all x.(girl(x) -> P(x))) (\z3.chase(z3,z4))
  (\P.exists x.(dog(x) & P(x))) (\z4.all x.(girl(x) -> chase(x,z4)))
Permutation 2
  (\P.exists x.(dog(x) & P(x))) (\z4.chase(z3,z4))
  (\P.all x.(girl(x) -> P(x))) (\z3.exists x.(dog(x) & chase(z3,x)))
```

```
>>> for reading in cs_semrep.readings:
...     print reading
exists x.(dog(x) & all z5.(girl(z5) -> chase(z5,x)))
all x.(girl(x) -> exists z6.(dog(z6) & chase(x,z6)))
```

NLTK contains implementations of two other approaches to scope ambiguity, namely **hole semantics** as described in [Blackburn & Bos, 2005] and **Glue semantics** as described in [Dalrymple, 1999]. We do not have space to discuss these here, but documentation can be found at [howto REFS].

11.5 Inference Tools

In order to perform inference over semantic representations, NLTK can call both theorem provers and model builders. The

library includes a pure Python tableau-based first order theorem prover; this is intended to allow students to study tableau methods for theorem proving, and provides an opportunity for experimentation. In addition, NLTK provides interfaces to two third party tools, namely the theorem prover Prover9, and the model builder Mace4 [McCune, 2008].

The `get_prover(G, A)` method by default calls Prover9, and takes as parameters a proof goal `G` and a list `A` of assumptions. Here, we verify that if every dog barks, and Rover is a dog, then it is true that Rover barks:

```
>>> from nltk.inference import inference
>>> a = lp.parse('all x.(dog(x) -> bark(x))')
>>> b = lp.parse('dog(rover)')
>>> c = lp.parse('bark(rover)')
>>> prover = inference.get_prover(c, [a,b])
>>> prover.prove()
True
```

A theorem prover can also be used to check the logical equivalence of expressions. For two expressions A and B , we can pass ($A \equiv B$) into a theorem prover and know that the theorem will be proved if and only if the expressions are logically equivalent. NLTK's standard equality operator for `Expressions` (`==`) is able to handle situations where two expressions are identical up to α -conversion. However, it would be impractical for NLTK to invoke a wider range of logic rules every time we checked for equality of two expressions. Consequently, both the `logic` and `drt` modules in NLTK have a separate method, `tp_equals`, for checking 'equality' up to logical equivalence.

```
>>> a = lp.parse('all x.walk(x)')
>>> b = lp.parse('all y.walk(y)')
>>> a == b
True
>>> c = lp.parse('-(P(x) & Q(x))')
>>> d = lp.parse('¬P(x) | ¬Q(x)')
>>> c == d
False
>>> c.tp_equals(d)
True
```

11.6 Discourse Semantics

Discourse Representation Theory

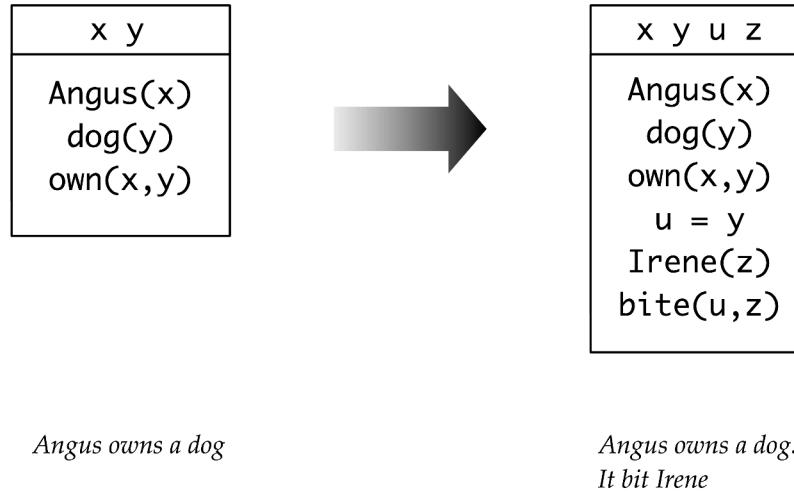
The standard approach to quantification in FOL is limited to single sentences. Yet there seem to be examples where the scope of a quantifier can extend over two or more sentences. Consider:

(213) Angus owns a dog. It bit Irene.

(55) is interpreted as (56).

(214) $\exists x.(dog(x) \wedge own(Angus, x) \wedge bite(x, Irene))$

That is, the NP *a dog* acts like a quantifier which binds the *it* in the second sentence. Discourse Representation Theory (DRT, [Kamp & Reyle, 1993]) was developed with the specific goal of providing a means for handling this and other semantic phenomena which seem to be characteristic of discourse — sentences in a sequence whose interpretation depends in part on what preceded them. A **discourse representation structure** (DRS) presents the meaning of discourse in terms of a list of **discourse referents** (the things under discussion in the discourse) together with a list of **conditions** on those discourse referents. These look respectively like the individual variables and atomic open formulas of FOL. Figure 11.4 illustrates how DRS for the first sentence in (55) is augmented to become a DRS for both sentences.

**Figure 11.4:** Building a DRS

When the second sentence of (55) is processed, it is interpreted in the context of what is already present 11.4. The pronoun *it* triggers the addition of a new discourse referent, say *u*, and we need to find an **anaphoric antecedent** for it — that is, we want to figure out what *it* refers to. In DRT, the task of finding antecedents for anaphoric pronouns is framed in terms of linking the pronoun to a discourse referent already within the current DRS, and *y* is the obvious choice. (We will say more about anaphora resolution shortly.) This processing step gives rise to a new condition *u = y*. The remaining content contributed by the second sentence is also merged with the content of the first, and this is shown on the righthand side of Figure 11.4.

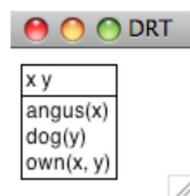
Figure 11.4 illustrates how a DRS can represent more than just a single sentence. In this case, it is a two-sentence discourse, but in principle a single DRS could correspond to the interpretation of a whole text. We can inquire into the truth conditions of the righthand DRS in figure 11.4. Informally, it is true in some situation *s* if there are entities **a**, **c** and **i** in *s* corresponding to the discourse referents in the DRS such that all the conditions true in *s*; that is, **a** is named *Angus*, **c** is a dog, **a** owns **c**, **i** is named *Irene* and **c** bit **i**.

In order to process DRSs computationally, we need to convert them into a linear format. The `nltk.sem.drt` module uses a `DRS()` constructor which takes lists of discourse referents and conditions as initialization parameters.

```
DRS([x, y], [Angus(x), dog(y), own(x, y)])
```

The easiest way to build a `DRS` object in NLTK is by parsing a string representation. Once we have done this, we can use the `draw()` method to visualize the result, as shown in 11.5.

```
>>> from nltk.sem import drt
>>> dp = drt.DrtParser()
>>> drs1 = dp.parse('([x,y], [Angus(x), dog(y), own(x, y)])')
>>> print drs1
([x,y], [Angus(x), dog(y), own(x, y)])
>>> drs1.draw()
```

**Figure 11.5:** DRS Screenshot

When we discussed the truth conditions of the DRSs in Figure 11.4, we assumed that the topmost discourse referents were interpreted like existentially quantifiers, while the conditions were interpreted as though they are conjoined. In fact, every DRS can be translated into a formula of FOL, and the `toFol()` method implements this translation.

```
>>> from nltk.sem import drt
>>> dp = drt.DrtParser()
```

```
>>> drs1 = dp.parse('([x,y], [Angus(x), dog(y), own(x, y)])')
>>> print drs1
([x,y], [Angus(x), dog(y), own(x,y)])
>>> print drs1.toFol()
exists x y. (Angus(x) & (dog(y) & own(x,y)))
```

In addition to the functionality available for FOL expressions, DRT Expressions have a DRS-concatenation operator, represented as the + symbol. The concatenation of two DRSs is a single DRS containing the merged discourse referents and the conditions from both arguments. DRS-concatenation automatically α -converts bound variables to avoid name-clashes. The + symbol is overloaded so that DRT expressions can be added together easily.

```
>>> drs2 = dp.parse('([x], [walk(x)]) + ([y], [run(y)])')
>>> print drs2
(([x], [walk(x)]) + ([y], [run(y)]))
>>> print drs2.simplify()
([x,y], [walk(x), run(y)])
```

While all the conditions see so far have been atomic, it is possible to embed one DRS within another, and this is how universal quantification is handled. In `drs3`, there are no top-level discourse referents, and the sole condition is made up of two sub-DRSIs, connected by an implication. Again, we can use `toFol()` to get a handle on the truth conditions.

```
>>> dds3 = dp.parse('([], [(([x], [dog(x)]) -> ([y], [ankle(y), bite(x, y))))])')
>>> print dds3.toFol()
all x. (dog(x) -> exists y. (ankle(y) & bite(x,y)))
```

Anaphora Resolution

We pointed out earlier that DRT is designed to allow anaphoric pronouns to be interpreted by linking to existing discourse referents. DRT sets constraints on which discourse referents are 'accessible' as possible antecedents, but is not intended to explain how a particular antecedent is chosen from the set of candidates. The module `nltk.sem.drt_resolve_anaphora` adopts a similarly conservative strategy: if the DRS contains a condition of the form `PRO(x)`, the method `resolve_anaphora()` replaces this with a condition of the form `x = [...]`, where [...] is a list of possible antecedents.

```
>>> dds4 = dp.parse('([x,y], [Angus(x), dog(y), own(x, y)])')
>>> dds5 = dp.parse('([u,z], [PRO(u), Irene(z), bite(u,z)])')
>>> dds6 = dds4 + dds5
>>> print dds6.simplify()
([x,y,u,z], [Angus(x), dog(y), own(x,y), PRO(u), Irene(z), bite(u,z)])
>>> print dds6.simplify().resolve_anaphora()
([x,y,u,z], [Angus(x), dog(y), own(x,y), (u = [x,y,z]), Irene(z), bite(u,z)])
```

Since the algorithm for anaphora resolution has been separated into its own module, this facilitates swapping in alternative procedures which try to make more intelligent guesses about the correct antecedent.

Since the treatment of DRSs in `nltk.sem.drt` is fully compatible with the existing machinery for handling λ abstraction, it is straightforward to build compositional semantic representations which are based on DRT rather than FOL. The general approach is illustrated in the following rule for indefinites (which is part of the grammar `drt.fcfg`). For ease of comparison, we have added the parallel rule for indefinites from `sem3.fcfg`.

```
Det [num=sg, sem=<\P Q.((DRS([x], [])+P(x))+Q(x))>] -> 'a'
Det [num=sg, sem=<\P Q. exists x.(P(x) & Q(x))>] -> 'a'
```

To get a better idea of how the DRT rule works, look at this subtree for the NP *a dog*.

```
(NP [num='sg', sem=<\Q.(([x], [dog(x)]) + Q(x))>]
(Det [num='sg', sem=<\P Q.(((x,[]) + P(x)) + Q(x))>] a)
(Nom [num='sg', sem=<\x.([x], [dog(x)])>]
(N [num='sg', sem=<\x.([x], [dog(x)])>] dog)))
```

The λ abstract for the indefinite is applied as a functor to $\lambda x.([x], [dog(x)])$ which leads to $\lambda Q.(((x,[]) + ([x], [dog(x)])) + Q(x))$; after simplification, we get $\lambda Q.(([x], [dog(x)]) + Q(x))$ as the representation for the NP as a whole.

In order to parse with grammar `drt.fcfg`, we specify in the call to `load_earley()` that `sem` values in feature structures are to

parsed using `drt.DrtParser` in place of the default `LogicParser`.

```
>>> from nltk.parse import load_earley
>>> parser = load_earley('grammars/drt.fcfg', trace=0, logic_parser=drt.DrtParser())
>>> trees = parser.nbest_parse('Angus owns a dog'.split())
>>> print trees[0].node['sem'].simplify()
([x, z2], [Angus(x), dog(z2), own(x, z2)])
```

Discourse Processing

When we interpret a sentence, we use a rich context for interpretation, determined in part by the preceding context and in part by our background assumptions. DRT provides a theory of how the meaning of a sentence is integrated into a representation of the prior discourse, but two things have been glaringly absent from the processing approach just discussed. First, there has been no attempt to incorporate any kind of inference; and second, we have only processed individual sentences. These omissions are redressed by the module `nltk.inference.discourse`, which is inspired by the CURT system of [Blackburn & Bos, 2005].

Whereas a discourse is a sequence s_1, \dots, s_n of sentences, a *discourse thread* is a sequence s_1-r_i, \dots, s_n-r_j of readings, one for each sentence in the discourse. The module processes sentences incrementally, keeping track of all possible threads when there is ambiguity. For simplicity, the following example ignores scope ambiguity.

```
>>> from nltk.inference.discourse import DiscourseTester as DT
>>> dt = DT(['A student dances', 'Every student is a person'])
>>> dt.readings()
s0 readings:
s0-r0: exists x.(student(x) & dance(x))
s1 readings:
s1-r0: all x.(student(x) -> person(x))
```

When a new sentence is added to the current discourse, setting the parameter `consistchk=True` causes consistency to be checked by invoking the model checker for each thread, i.e., sequence of admissible readings. In this case, the user has the option of retracting the sentence in question.

```
>>> dt.add_sentence('No person dances', consistchk=True)
Inconsistent discourse d0 ['s0-r0', 's1-r0', 's2-r0']:
s0-r0: exists x.(student(x) & dance(x))
s1-r0: all x.(student(x) -> person(x))
s2-r0: -exists x.(person(x) & dance(x))
>>> dt.retract_sentence('No person dances', quiet=False)
Current sentences are
s0: A student dances
s1: Every student is a person
```

In a similar manner, we use `informchk=True` to check whether the new sentence is informative relative to the current discourse (by asking the theorem prover to derive it from the discourse).

```
>>> dt.add_sentence('A person dances', informchk=True)
Sentence 'A person dances' under reading 'exists x.(person(x) & dance(x))':
Not informative relative to thread 'd0'
```

It is also possible to pass in an additional set of assumptions as background knowledge and use these to filter out inconsistent readings; see <http://nltk.org/doc/guides/discourse.html> for more details.

The `discourse` module can accommodate semantic ambiguity and filter out readings that are not admissible. By invoking both Glue Semantics and DRT, the following example processes the two-sentence discourse *Every dog chases a boy. He runs*. As shown, the first sentence has two possible readings, while the second sentence contains an anaphoric pronoun, indicated as `PRO(x)`.

```
>>> from nltk.inference.discourse import DrtGlueReadingCommand as RC
>>> dt = DT(['Every dog chases a boy', 'He runs'], RC())
>>> dt.readings()
s0 readings:
s0-r0: ([] , [([x], [dog(x)]) -> ([z15], [boy(z15), chase(x, z15)])))
s0-r1: ([z16], [boy(z16), (([x], [dog(x)]) -> ([] , [chase(x, z16)])))])
s1 readings:
```

s1-r0: ([x], [PRO(x), run(x)])

When we examine the two threads `d0` and `d1`, we see that that reading `s0-r0`, where *every dog* out-scopes a *boy*, is deemed inadmissible because the pronoun in the second sentence cannot be resolved. By contrast, in thread `d1` the pronoun (relettered to `z24`) has been bound *via* the equation `(z24 = z20)`.

Inadmissible readings are filtered out by passing the parameter `filter=True`.

```
>>> dt.readings(show_thread_readings=True)
d0: ['s0-r0', 's1-r0'] : INVALID: AnaphoraResolutionException
d1: ['s0-r1', 's1-r0'] : ([z20,z24], [boy(z20), (([x], [dog(x)]) ->
([], [chase(x,z20)])), (z24 = z20), run(z24)])
>>> dt.readings(filter=True, show_thread_readings=True)
d1: ['s0-r1', 's1-r0'] : ([z26,z29], [boy(z26), (([x], [dog(x)]) ->
([], [chase(x,z26)])), (z29 = z26), run(z29)])
```

```
>>> dt.readings(show_thread_readings=True)
d0: ['s0-r0', 's1-r0'] : INVALID: AnaphoraResolutionException
d1: ['s0-r1', 's1-r0'] : ([z20,z24], [boy(z20), (([x], [dog(x)]) ->
([], [chase(x,z20)])), (z24 = z20), run(z24)])
```

11.7 Summary

In this chapter, we presented a standard approach to natural language meaning by translating sentences of natural language into first-order logic. From a computational point of view, a strong argument in favor of FOL is that it strikes a reasonable balance between expressiveness and logical tractability. On the one hand, it is flexible enough to represent many aspects of the logical structure of natural language. On the other hand, automated theorem proving for FOL has been well studied, and although inference in FOL is not decidable, in practice many reasoning problems are efficiently solvable using modern theorem provers (cf. [\[Blackburn & Bos, 2005\]](#) for discussion).

- Semantic Representations (SRs) for English are constructed using a language based on the λ -calculus, together with Boolean connectives, equality, and first-order quantifiers.
- β -reduction in the λ -calculus corresponds semantically to application of a function to an argument. Syntactically, it involves replacing a variable bound by λ in the functor with the expression that provides the argument in the function application.
- If two λ -abstracts differ only in the label of the variable bound by λ , they are said to be α equivalents. Relabeling a variable bound by a λ is called α -conversion.
- Currying of a binary function turns it into a unary function whose value is again a unary function.
- FSRL has both a syntax and a semantics. The semantics is determined by recursively evaluating expressions in a model.
- A key part of constructing a model lies in building a valuation which assigns interpretations to non-logical constants. These are interpreted as either curried characteristic functions or as individual constants.
- The interpretation of Boolean connectives is handled by the model; these are interpreted as characteristic functions.
- An open expression is an expression containing one or more free variables. Open expressions only receive an interpretation when their free variables receive values from a variable assignment.
- Quantifiers are interpreted by constructing, for a formula $\varphi[x]$ open in variable x , the set of individuals which make $\varphi[x]$ true when an assignment g assigns them as the value of x . The quantifier then places constraints on that set.
- A closed expression is one that has no free variables; that is, the variables are all bound. A closed sentence is true or false with respect to all variable assignments.
- Given a formula with two nested quantifiers Q_1 and Q_2 , the outermost quantifier Q_1 is said to have wide scope (or scope over Q_2). English sentences are frequently ambiguous with respect to the scope of the quantifiers they contain.
- English sentences can be associated with an SR by treating `sem` as a feature. The `sem` value of a complex expressions typically involves functional application of the `sem` values of the component expressions.
- Model valuations need not be built by hand, but can also be extracted from relational tables, as in the Chat-80 example.

11.8 Further Reading

For more examples of semantic analysis with NLTK, please see the guides at <http://nltk.org/doc/guides/sem.html> and <http://nltk.org/doc/guides/logic.html>.

The use of characteristic functions for interpreting expressions of natural language was primarily due to Richard Montague. [Dowty, Wall, & Peters, 1981] gives a comprehensive and reasonably approachable introduction to Montague's grammatical framework.

A more recent and wide-reaching study of the use of a λ based approach to natural language can be found in [Carpenter, 1997].

[Heim & Kratzer, 1998] is a thorough application of formal semantics to transformational grammars in the Government-Binding model.

[Blackburn & Bos, 2005] is the first textbook devoted to computational semantics, and provides an excellent introduction to the area.

<http://www.fil.ion.ucl.ac.uk/~asaygin/tt/ttest.html>

11.9 Exercises

1. ● Modify the `sem.evaluate` code so that it will give a helpful error message if an expression is not in the domain of a model's valuation function.
2. ★ Specify and implement a typed functional language with quantifiers, Boolean connectives and equality. Modify `sem.evaluate` to interpret expressions of this language.
3. ★ Extend the `chat80` code so that it will extract data from a relational database using SQL queries.
4. ★ Taking [WarrenPereira1982] as a starting point, develop a technique for converting a natural language query into a form that can be evaluated more efficiently in a model. For example, given a query of the form '`'(P(x) & Q(x))'`', convert it to '`'(Q(x) & P(x))'` if the extension of '`'Q'`' is smaller than the extension of '`'P'`'.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

12 Linguistic Data Management (DRAFT)

12.1 Introduction

Language resources of all kinds are proliferating on the Web. These include data such as lexicons and annotated text, and software tools for creating and manipulating the data. As we have seen in previous chapters, language resources are essential in most areas of NLP. This has been made possible by three significant technological developments over the past decade. First, inexpensive mass storage technology permits large resources to be stored in digital form, while the Extensible Markup Language (XML) and Unicode provide flexible ways to represent structured data and give it good prospects for long-term survival. Second, digital publication has been a practical and efficient means of sharing language resources. Finally, search engines, mailing lists, and online resource catalogs make it possible for people to discover the existence of the resources they may be seeking.

Together with these technological advances have been three other developments that have shifted the NLP community in the direction of data-intensive approaches. First, the "shared task method," an initiative of government sponsors, supports major sections within the community to identify a common goal for the coming year, and provides "gold standard" data on which competing systems can be evaluated. Second, data publishers such as the Linguistic Data Consortium have negotiated with hundreds of data providers (including newswire services in many countries), and created hundreds of annotated corpora stored in well-defined and consistent formats. Finally, organizations that purchase NLP systems, or that publish NLP papers, now expect the quality of the work to be demonstrated using standard datasets.

Although language resources are central to NLP, we still face many obstacles in using them. First, the resource we are looking for may not exist, and so we have to think about *creating* a new language resource, and doing a sufficiently careful job that it serves our future needs, thanks to its coverage, balance, and documentation of the sources. Second, a resource may exist but its

creator didn't document its existence anywhere, leaving us to recreate the resource; however, to save further wasted effort we should learn about *publishing* metadata the documents the existence of a resource, and even how to publish the resource itself, in a form that is easy for others to re-use. Third, the resource may exist and may be obtained, but is in an incompatible format, and so we need to set about *converting* the data into a different format. Finally, the resource may be in the right format, but the available software is unable to perform the required analysis task, and so we need to develop our own program for *analyzing* the data. This chapter covers each of these issues — creating, publishing, converting, and analyzing — using many examples drawn from practical experience managing linguistic data. However, before embarking on this sequence of issues, we start by examining the organization of linguistic data.

12.2 Linguistic Databases

Linguistic databases span a multidimensional space of cases, which we can divide up in several ways: the scope and design of the data collection; the goals of the creators; the nature of the material included; the goals and methods of the users (which are often not anticipated by the creators). Three examples follow.

In one type of linguistic database, the design unfolds interactively in the course of the creator's explorations. This is the pattern typical of traditional "field linguistics," in which material from elicitation sessions is analyzed repeatedly as it is gathered, with tomorrow's elicitation often based on questions that arise in analyzing today's. The resulting field notes are then used during subsequent years of research, and may serve as an archival resource indefinitely — the field notes of linguists and anthropologists working in the early years of the 20th century remain an important source of information today.

Computerization is an obvious boon to work of this type, as exemplified by the popular program **Shoebox** — now about two decades old and re-released as **Toolbox** — which replaces the field linguist's traditional shoebox full of file cards.

Another pattern is represented by experimental approaches in which a body of carefully-designed material is collected from a range of subjects, then analyzed to evaluate a hypothesis or develop a technology. Today, such databases are collected and analyzed in digital form. Among scientists (such as phoneticians or psychologists), they are rarely published and therefore rarely preserved. Among engineers, it has become common for such databases to be shared and re-used at least within a laboratory or company, and often to be published more widely. Linguistic databases of this type are the basis of the "common task" method of research management, which over the past 15 years has become the norm in government-funded research programs in speech- and language-related technology.

Finally, there are efforts to gather a "reference corpus" for a particular language. Large and well-documented examples include the **American National Corpus** (ANC) and the **British National Corpus** (BNC). The goal in such cases is to produce a set of linguistic materials that cover the many forms, styles and uses of a language as widely as possible. The core application is typically lexicographic, that is, the construction of dictionaries based on a careful study of patterns of use. These corpora were constructed by large consortia spanning government, industry, and academia. Their planning and execution took more than five years, and indirectly involved hundreds of person-years of effort. There is also a long and distinguished history of other humanistic reference corpora, such as the Thesaurus Linguae Graecae.

There are no hard boundaries among these categories. Accumulations of smaller bodies of data may come in time to constitute a sort of reference corpus, while selections from large databases may form the basis for a particular experiment. Further instructive examples follow.

A linguist's field notes may include extensive examples of many genres (proverbs, conversations, narratives, rituals, and so forth), and may come to constitute a reference corpus of modest but useful size. There are many extinct languages for which such material is all the data we will ever have, and many more endangered languages for which such documentation is urgently needed. Sociolinguists typically base their work on analysis of a set of recorded interviews, which may over time grow to create another sort of reference corpus. In some labs, the residue of decades of work may comprise literally thousands of hours of recordings, many of which have been transcribed and annotated to one extent or another. The **CHILDES** corpus, comprising transcriptions of parent-child interactions in many languages, contributed by many individual researchers, has come to constitute a widely-used reference corpus for language acquisition research. Speech technologists aim to produce training and testing material of broad applicability, and wind up creating another sort of reference corpus. To date, linguistic technology R&D has been the primary source of published linguistic databases of all sorts (see e.g. <http://www.ldc.upenn.edu/>).

As large, varied linguistic databases are published, phoneticians or psychologists are increasingly likely to base experimental investigations on balanced, focused subsets extracted from databases produced for entirely different reasons. Their motivations include the desire to save time and effort, the desire to work on material available to others for replication, and sometimes a desire to study more naturalistic forms of linguistic behavior. The process of choosing a subset for such a study, and making the measurements involved, is usually in itself a non-trivial addition to the database. This recycling of linguistic databases for new

purposes is a normal and expected consequence of publication. For instance, the Switchboard database, originally collected for speaker identification research, has since been used as the basis for published studies in speech recognition, word pronunciation, disfluency, syntax, intonation and discourse structure.

At present, only a tiny fraction of the linguistic databases that are collected are published in any meaningful sense. This is mostly because publication of such material was both time-consuming and expensive, and because use of such material by other researchers was also both expensive and technically difficult. However, general improvements in hardware, software and networking have changed this, and linguistic databases can now be created, published, stored and used without inordinate effort or large expense.

In practice, the implications of these cost-performance changes are only beginning to be felt. The main problem is that adequate tools for creation, publication and use of linguistic data are not widely available. In most cases, each project must create its own set of tools, which hinders publication by researchers who lack the expertise, time or resources to make their data accessible to others. Furthermore, we do not have adequate, generally accepted standards for expressing the structure and content of linguistic databases. Without such standards, general-purpose tools are impossible — though at the same time, without available tools, adequate standards are unlikely to be developed, used and accepted. Just as importantly, there must be a critical mass of users and published material to motivate maintenance of data and access tools over time.

Relative to these needs, the present chapter has modest goals, namely to equip readers to take linguistic databases into their own hands by writing programs to help create, publish, transform and analyze the data. In the rest of this section we take a close look at the fundamental data types, an exemplary speech corpus, and the lifecycle of linguistic data.

Lexicon

Abstraction: fielded records

| | | | | |
|-----|-------|-------|-------|-------|
| key | field | field | field | field |
| key | field | field | field | field |

Eg: dictionary

wake: /weɪk/, [v], cease to sleep...
walk: /wɔ:k/, [v], progress by lifting and setting down each foot...

Eg: comparative wordlist

wake; aufwecken; acordar
walk; gehen; andar
write; schreiben; enscrever

Eg: verb paradigm

| | | |
|-------|-------|---------|
| wake | woke | woken |
| write | wrote | written |
| wring | wrung | wrung |

Text

Abstraction: time series

| | | | |
|-------------|-------------|-------------|-----|
| token attrs | token attrs | token attrs | ... |
|-------------|-------------|-------------|-----|

time →

Eg: written text

A long time ago, Sun and Moon lived together. They were good brothers. ...

Eg: POS-tagged text

A/DT long/JJ time/NN ago/RB ./,
Sun/NNP and/CC Moon/NNP
lived/VBD together/RB ./.

Eg: interlinear text

| | | |
|---------|------|------------|
| Ragaipa | irai | vateri |
| ragai | -pa | ira |
| PP.1.SG | -BEN | RP.3.SG.M |
| | | -ABS |
| | | give -2.SG |

Figure 12.1: Basic Linguistic Datatypes: Lexicons and Texts

Fundamental Data Types

Linguistic data management deals with a variety of data types, the most important being lexicons and texts. A **lexicon** is a database of words, minimally containing part of speech information and glosses. For many lexical resources, it is sufficient to use a **record** structure, i.e. a key plus one or more fields, as shown in [Figure 12.1](#). A lexical resource could be a conventional dictionary or comparative wordlist, as illustrated. Several related linguistic data types also fit this model. For example in a

phrasal lexicon, the key field is a phrase rather than a single word. A thesaurus can be derived from a lexicon by adding topic fields to the entries and constructing an index over those fields. We can also construct special tabulations (known as paradigms) to illustrate contrasts and systematic variation, as shown in [Figure 12.1](#) for three verbs.

At the most abstract level, a **text** is a representation of a real or fictional speech event, and the time-course of that event carries over into the text itself. A text could be a small unit, such as a word or sentence, or a complete narrative or dialogue. It may come with annotations such as part-of-speech tags, morphological analysis, discourse structure, and so forth. As we saw in the IOB tagging technique ([Chapter 7](#)), it is possible to represent higher-level constituents using tags on individual words. Thus the abstraction of text shown in [Figure 12.1](#) is sufficient.

Corpus Structure: a Case Study of TIMIT

The TIMIT corpus of read speech was the first annotated speech database to be widely distributed, and it has an especially clear organization. TIMIT was developed by a consortium including Texas Instruments and MIT (hence the name), and was designed to provide data for the acquisition of acoustic-phonetic knowledge and to support the development and evaluation of automatic speech recognition systems.

Like the Brown Corpus, which displays a balanced selection of text genres and sources, TIMIT includes a balanced selection of dialects, speakers, and materials. For each of eight dialect regions, 50 male and female speakers having a range of ages and educational backgrounds each read ten carefully chosen sentences. Two sentences, read by all speakers, were designed to bring out dialect variation:

- (215)
- a. she had your dark suit in greasy wash water all year
 - b. don't ask me to carry an oily rag like that

The remaining sentences were chosen to be phonetically rich, involving all phones (sounds) and a comprehensive range of diphones (phone bigrams). Additionally, the design strikes a balance between multiple speakers saying the same sentence in order to permit comparison across speakers, and having a large range of sentences covered by the corpus to get maximal coverage of diphones. Thus, five sentences read by each speaker, are also read by six other speakers (comparability). The remaining three sentences read by each speaker were unique to that speaker (coverage).

NLTK includes a sample from the TIMIT corpus. You can access its documentation in the usual way, using `help(corpus.timit)`. Print `corpus.timit.items` to see a list of the 160 recorded utterances in the corpus sample. Each item name has complex internal structure, as shown in [Figure 12.2](#).

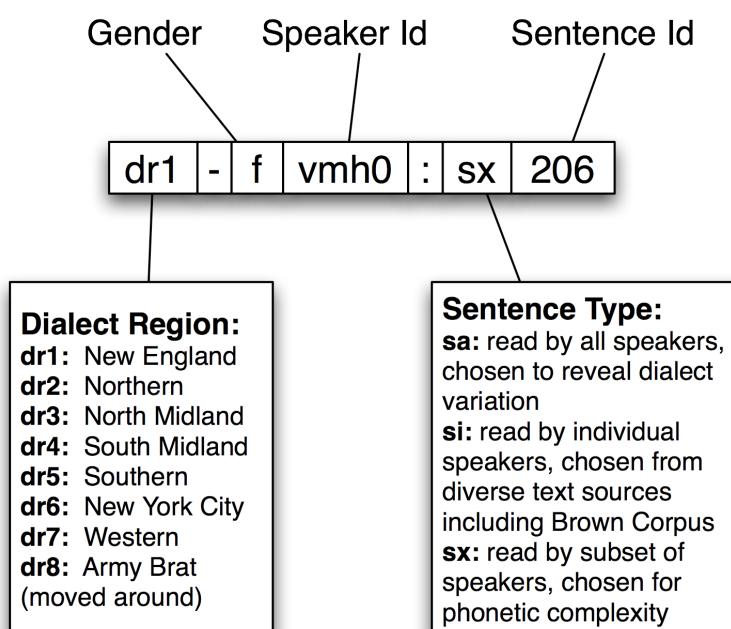


Figure 12.2: Structure of a TIMIT Identifier in the NLTK Corpus Package

Each item has a phonetic transcription, which can be accessed using the `phones()` method. We can access the corresponding word tokens in the customary way. Both access methods permit an optional argument `offset=True` which includes the start and end offsets of the corresponding span in the audio file.

```
>>> phonetic = nltk.corpus.timit.phones('dr1-fvmh0/sal1')
>>> phonetic
['h#', 'sh', 'iy', 'hv', 'ae', 'dcl', 'y', 'ix', 'dcl', 'd', 'aa', 'kcl',
's', 'ux', 'tcl', 'en', 'gcl', 'g', 'r', 'iy', 's', 'iy', 'w', 'aa',
'sh', 'epi', 'w', 'aa', 'dx', 'ax', 'q', 'ao', 'l', 'y', 'ih', 'ax', 'h#']
>>> nltk.corpus.timit.word_times('dr1-fvmh0/sal1')
[('she', 7812, 10610), ('had', 10610, 14496), ('your', 14496, 15791),
('dark', 15791, 20720), ('suit', 20720, 25647), ('in', 25647, 26906),
('greasy', 26906, 32668), ('wash', 32668, 37890), ('water', 38531, 42417),
('all', 43091, 46052), ('year', 46052, 50522)]
```

In addition to this text data, TIMIT includes a lexicon that provides the canonical pronunciation of every word:

```
>>> timitdict = nltk.corpus.timit.transcription_dict()
>>> timitdict['greasy'] + timitdict['wash'] + timitdict['water']
['g', 'r', 'iy1', 's', 'iy', 'w', 'aol', 'sh', 'w', 'ao1', 't', 'axr']
>>> phonetic[17:30]
['g', 'r', 'iy', 's', 'iy', 'w', 'aa', 'sh', 'epi', 'w', 'aa', 'dx', 'ax']
```

This gives us a sense of what a speech processing system would have to do in producing or recognizing speech in this particular dialect (New England). Finally, TIMIT includes demographic data about the speakers, permitting fine-grained study of vocal, social, and gender characteristics.

```
>>> nltk.corpus.timit.spkrinfo('dr1-fvmh0')
SpeakerInfo(id='VMH0', sex='F', dr='1', use='TRN', recdate='03/11/86',
birthdate='01/08/60', ht='5\'05"', race='WHT', edu='BS',
comments='BEST NEW ENGLAND ACCENT SO FAR')
```

TIMIT illustrates several key features of corpus design. First, the corpus contains two layers of annotation, at the phonetic and orthographic levels. In general, a text or speech corpus may be annotated at many different linguistic levels, including morphological, syntactic, and discourse levels. Moreover, even at a given level there may be different labeling schemes or even disagreement amongst annotators, such that we want to represent multiple versions. A second property of TIMIT is its balance across multiple dimensions of variation, for coverage of dialect regions and diphones. The inclusion of speaker demographics brings in many more independent variables, that may help to account for variation in the data, and which facilitate later uses of the corpus for purposes that were not envisaged when the corpus was created, e.g. sociolinguistics. A third property is that there is a sharp division between the original linguistic event captured as an audio recording, and the annotations of that event. The same holds true of text corpora, in the sense that the original text usually has an external source, and is considered to be an immutable artifact. Any transformations of that artifact which involve human judgment — even something as simple as tokenization — are subject to later revision, thus it is important to retain the source material in a form that is as close to the original as possible.

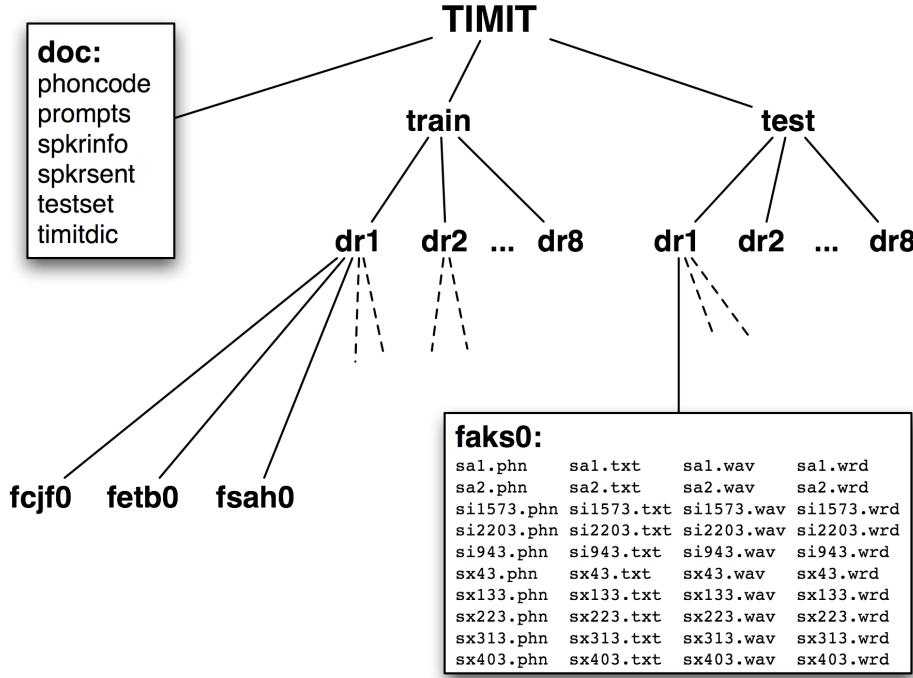


Figure 12.3: Structure of the Published TIMIT Corpus

A fourth feature of TIMIT is the hierarchical structure of the corpus. With 4 files per sentence, and 10 sentences for each of 500 speakers, there are 20,000 files. These are organized into a tree structure, shown schematically in [Figure 12.3](#). At the top level there is a split between training and testing sets, which gives away its intended use for developing and evaluating statistical models.

Finally, notice that even though TIMIT is a speech corpus, its transcriptions and associated data are just text, and can be processed using programs just like any other text corpus. Therefore, many of the computational methods described in this book are applicable. Moreover, notice that all of the data types included in the TIMIT corpus fall into our two basic categories of lexicon and text (cf. the discussion of fundamental data types in [section 12.2](#)). Even the speaker demographics data is just another instance of the lexicon data type.

This last observation is less surprising when we consider that text and record structures are the primary domains for the two subfields of computer science that focus on data management, namely text retrieval and databases. A notable feature of linguistic data management is that usually brings both data types together, and that it can draw on results and techniques from both fields.

The Lifecycle of Linguistic Data: Evolution vs Curation

Once a corpus has been created and disseminated, it typically gains a life of its own, as others adapt it to their needs. This may involve reformatting a text file (e.g. converting to XML), renaming files, retokenizing the text, selecting a subset of the data to enrich, and so forth. Multiple research groups may do this work independently, as exemplified in [Figure 12.4](#). At a later date, when someone wants to combine sources of information from different version, the task may be extremely onerous.

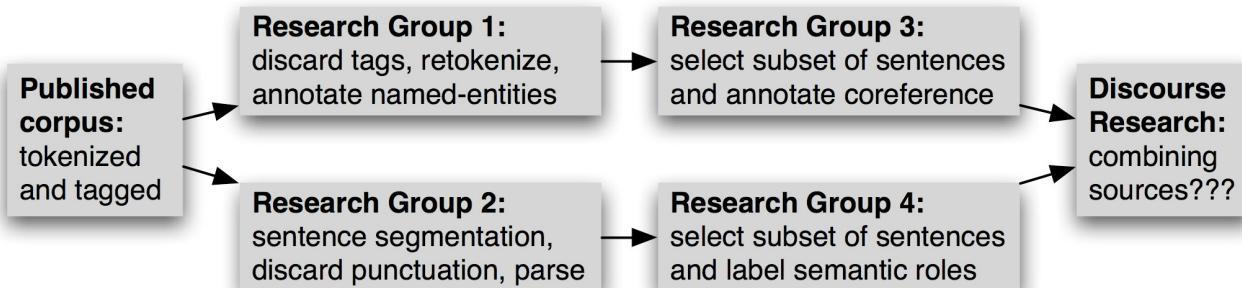


Figure 12.4: Evolution of a Corpus

The task of using derived corpora is made even more difficult by the lack of any record about how the derived version was created, and which version is the most up-to-date.

An alternative to this chaotic situation is for all corpora to be centrally curated, and for committees of experts to revise and extend a reference corpus at periodic intervals, considering proposals for new content from third-parties, much like a dictionary is edited. However, this is impractical.

A better solution is to have a canonical, immutable primary source, which supports incoming references to any sub-part, and then for all annotations (including segmentations) to reference this source. This way, two independent tokenizations of the same text can be represented without touch the source text, as can any further labeling and grouping of those annotations. This method is known as **standoff annotation**.

[More discussion and examples]

12.3 Creating Data

Scenarios: fieldwork, web, manual entry using local tool, machine learning with manual post-editing

Conventional office software is widely used in computer-based language documentation work, given its familiarity and ready availability. This includes word processors and spreadsheets.

Spiders

- what they do: basic idea is simple
- python code to find all the anchors, extract the href, and make an absolute URL for fetching
- many issues: starting points, staying within a single site, only getting HTML
- various stand-alone tools for spidering, and mirroring

Creating Language Resources Using Word Processors

Word processing software is often used in creating dictionaries and interlinear texts. As the data grows in size and complexity, a larger proportion of time is spent maintaining consistency. Consider a dictionary in which each entry has a part-of-speech field, drawn from a set of 20 possibilities, displayed after the pronunciation field, and rendered in 11-point bold. No conventional word processor has search or macro functions capable of verifying that all part-of-speech fields have been correctly entered and displayed. This task requires exhaustive manual checking. If the word processor permits the document to be saved in a non-proprietary format, such as text, HTML, or XML, we can sometimes write programs to do this checking automatically.

Consider the following fragment of a lexical entry: "sleep [sli:p] **vi** condition of body and mind...". We can enter this in MSWord, then "Save as Web Page", then inspect the resulting HTML file:

```

<p class=MsoNormal>sleep
<span style='mso-spacerun:yes'> </span>
[<span class=SpellE>sli:p</span>]
<span style='mso-spacerun:yes'> </span>
<b><span style='font-size:11.0pt'>vi</span></b>
<span style='mso-spacerun:yes'> </span>
  
```

```
<i>a condition of body and mind ...<o:p></o:p></i>
</p>
```

Observe that the entry is represented as an HTML paragraph, using the `<p>` element, and that the part of speech appears inside a `` element. The following program defines the set of legal parts-of-speech, `legal_pos`. Then it extracts all 11-point content from the `dict.htm` file and stores it in the set `used_pos`. Observe that the search pattern contains a parenthesized sub-expression; only the material that matches this sub-expression is returned by `re.findall`. Finally, the program constructs the set of illegal parts-of-speech as `used_pos - legal_pos`:

```
>>> legal_pos = set(['n', 'v.t.', 'v.i.', 'adj', 'det'])
>>> pattern = re.compile(r'"font-size:11.0pt">([a-z.]+)<"')
>>> document = open("dict.htm").read()
>>> used_pos = set(re.findall(pattern, document))
>>> illegal_pos = used_pos.difference(legal_pos)
>>> print list(illegal_pos)
['v.i', 'intrans']
```

This simple program represents the tip of the iceberg. We can develop sophisticated tools to check the consistency of word processor files, and report errors so that the maintainer of the dictionary can correct the original file *using the original word processor*.

We can write other programs to convert the data into a different format. For example, the program in [Figure 12.5](#) strips out the HTML markup using `nltk.clean_html()`, extracts the words and their pronunciations, and generates output in "comma-separated value" (CSV) format:

```
def lexical_data(html_file):
    SEP = '_ENTRY'
    html = open(html_file).read()
    html = re.sub(r'<p>', SEP + '<p>', html)
    text = nltk.clean_html(html)
    text = ' '.join(text.split())
    for entry in text.split(SEP):
        if entry.count(' ') > 2:
            yield entry.split(' ', 3)

>>> import csv
>>> writer = csv.writer(open("dict1.csv", "wb"))
>>> writer.writerows(lexical_data("dict.htm"))
```

[Figure 12.5 \(html2csv.py\)](#): Figure 12.5: Converting HTML Created by Microsoft Word into Comma-Separated Values

Creating Language Resources Using Spreadsheets and Databases

Spreadsheets. These are often used for wordlists or paradigms. A comparative wordlist may be stored in a spreadsheet, with a row for each cognate set, and a column for each language. Examples are available from www.rosettaproject.org. Programs such as Excel can export spreadsheets in the CSV format, and we can write programs to manipulate them, with the help of Python's `csv` module. For example, we may want to print out cognates having an edit-distance of at least three from each other (i.e. 3 insertions, deletions, or substitutions).

Databases. Sometimes lexicons are stored in a full-fledged relational database. When properly normalized, these databases can implement many well-formedness constraints. For example, we can require that all parts-of-speech come from a specified vocabulary by declaring that the part-of-speech field is an *enumerated type*. However, the relational model is often too restrictive for linguistic data, which typically has many optional and repeatable fields (e.g. dictionary sense definitions and example sentences). Query languages such as SQL cannot express many linguistically-motivated queries, e.g. *find all words that appear in example sentences for which no dictionary entry is provided*. Now supposing that the database supports exporting data to CSV format, and that we can save the data to a file `dict.csv`:

```
"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"
```

Now we can express this query as shown in [Figure 12.6](#).

```
def undefined_words(csv_file):
    import csv
```

```

lexemes = set()
defn_words = set()
for row in csv.reader(open(csv_file)):
    lexeme, pron, pos, defn = row
    lexemes.add(lexeme)
    defn_words.union(defn.split())
return sorted(defn_words.difference(lexemes))

>>> print undefined_words("dict.csv")
[..., 'a', 'and', 'body', 'by', 'cease', 'condition', 'down', 'each',
'foot', 'lifting', 'mind', 'of', 'progress', 'setting', 'to']

```

[Figure 12.6 \(undefined_words.py\)](#): Figure 12.6: Finding definition words not themselves defined

Creating Language Resources Using Toolbox

Over the last two decades, several dozen tools have been developed that provide specialized support for linguistic data management. Perhaps the single most popular tool used by linguists for managing data is *Toolbox*, previously known as *Shoebox* (freely downloadable from <http://www.sil.org/computing/toolbox/>). In this section we discuss a variety of techniques for manipulating Toolbox data in ways that are not supported by the Toolbox software. (The methods we discuss could be applied to other record-structured data, regardless of the actual file format.)

A Toolbox file consists of a collection of *entries* (or *records*), where each record is made up of one or more *fields*. Here is an example of an entry taken from a Toolbox dictionary of Rotokas, already mentioned in [Section 2.4](#).

```

lx kaa ps N pt MASC cl isi ge cooking banana tkp banana bilong kukim pt itoo sf FLORA dt 12/Aug/2005 ex
Taeavi iria kaa isi kovopauvea kaparapasia. xp Taeavi i bin planim gaden banana bilong kukim tasol long paia. xe
Taeavi planted banana in order to cook it.

```

This lexical entry contains the following fields: `lx` lexeme; `ps` part-of-speech; `pt` part-of-speech; `cl` classifier; `ge` English gloss; `tkp` Tok Pisin gloss; `sf` Semantic field; `dt` Date last edited; `ex` Example sentence; `xp` Pidgin translation of example; `xe` English translation of example. These field names are preceded by a backslash, and must always appear at the start of a line. The characters of the field names must be alphabetic. The field name is separated from the field's contents by whitespace. The contents can be arbitrary text, and can continue over several lines (but cannot contain a line-initial backslash).

We can use the `toolbox.xml()` method to access a Toolbox file and load it into an `ElementTree` object.

```

>>> from nltk.corpus import toolbox
>>> lexicon = toolbox.xml('rotokas.dic')

```

There are two ways to access the contents of the `lexicon` object, by indexes and by paths. Indexes use the familiar syntax, thus `lexicon[3]` returns entry number 3 (which is actually the fourth entry counting from zero). And `lexicon[3][0]` returns its first field:

```

>>> lexicon[3][0]
<Element lx at 77bd28>
>>> lexicon[3][0].tag
'lx'
>>> lexicon[3][0].text
'kaa'

```

The second way to access the contents of the `lexicon` object uses paths. The `lexicon` is a series of `record` objects, each containing a series of field objects, such as `lx` and `ps`. We can conveniently address all of the lexemes using the path `record/lx`. Here we use the `findall()` function to search for any matches to the path `record/lx`, and we access the text content of the element, normalizing it to lowercase.

```

>>> [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
['kaa', 'kaa', 'kaa', 'kaakaaro', 'kaakaaviko', 'kaakaavo', 'kaakaoko',
'kaakasi', 'kaakau', 'kaakauko', 'kaakito', 'kaakuupato', ..., 'kuvuto']

```

It is often convenient to add new fields that are derived automatically from existing ones. Such fields often facilitate search and analysis. For example, in [Figure 12.7](#) we define a function `cv()` which maps a string of consonants and vowels to the corresponding CV sequence, e.g. `kakapua` would map to `CVCVCCV`. This mapping has four steps. First, the string is converted to lowercase, then we replace any non-alphabetic characters `[^a-z]` with an underscore. Next, we replace all vowels with `v`.

Finally, anything that is not a v or an underscore must be a consonant, so we replace it with a c. Now, we can scan the lexicon and add a new cv field after every lx field. [Figure 12.7](#) shows what this does to a particular entry; note the last line of output, which shows the new CV field.

```
from nltk.etree.ElementTree import SubElement

def cv(s):
    s = s.lower()
    s = re.sub(r'^[a-z]', r'_', s)
    s = re.sub(r'[aeiou]', r'V', s)
    s = re.sub(r'^V_', r'C', s)
    return (s)

def add_cv_field(entry):
    for field in entry:
        if field.tag == 'lx':
            cv_field = SubElement(entry, 'cv')
            cv_field.text = cv(field.text)

>>> lexicon = toolbox.xml('rotokas.dic')
>>> add_cv_field(lexicon[53])
>>> print nltk.corpus.reader.toolbox.to_sfm_string(lexicon[53])
\lx kaeviro
\ps V
\pt A
\ge lift off
\ge take off
\tkp go antap
\sc MOTION
\vx 1
\nt used to describe action of plane
\dt 03/Jun/2005
\ex Pita kaeviroroe kepa kekesia oa vuripierevo kiuvu.
\xp Pita i go antap na lukim haus win i bagarapim.
\xe Peter went to look at the house that the wind destroyed.
\cv CVVCVCV
```

[Figure 12.7 \(add_cv_field.py\)](#): Figure 12.7: Adding a new cv field to a lexical entry

Finally, we take a look at simple methods to generate summary reports, giving us an overall picture of the quality and organisation of the data.

First, suppose that we wanted to compute the average number of fields for each entry. This is just the total length of the entries (the number of fields they contain), divided by the number of entries in the lexicon:

```
>>> sum(len(entry) for entry in lexicon) / len(lexicon)
13
```

```
grammar = nltk.parse_cfg('''
S -> Head PS Glosses Comment Date Examples
Head -> Lexeme Root
Lexeme -> "lx"
Root -> "rt" |
PS -> "ps"
Glosses -> Gloss Glosses |
Gloss -> "ge" | "gp"
Date -> "dt"
Examples -> Example Ex_Pidgin Ex_English Examples |
Example -> "ex"
Ex_Pidgin -> "xp"
Ex_English -> "xe"
Comment -> "cmt" |
''')

def validate_lexicon(grammar, lexicon):
    rd_parser = nltk.RecursiveDescentParser(grammar)
    for entry in lexicon[10:20]:
        marker_list = [field.tag for field in entry]
        if rd_parser.get_parse_list(marker_list):
            print "+", ":".join(marker_list)
```

```

    else:
        print "-", ":".join(marker_list)

>>> lexicon = toolbox.xml('rotokas.dic')[10:20]
>>> validate_lexicon(grammar, lexicon)
- lx:ps:ge:gp:sf:nt:dt:ex:xp:xe:ex:xp:xe
- lx:rt:ps:ge:gp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:gp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:gp:nt:sf:dt
- lx:ps:ge:gp:dt:cmt:ex:xp:xe:ex:xp:xe
+ lx:ps:ge:ge:gp:cmt:dt:ex:xp:xe
+ lx:rt:ps:ge:gp:cmt:dt:ex:xp:xe:ex:xp:xe
+ lx:rt:ps:ge:ge:gp:dt
- lx:rt:ps:ge:ge:gp:dt:cmt:ex:xp:xe:ex:xp:xe
+ lx:rt:ps:ge:gp:dt

```

Figure 12.8 (toolbox_validation.py): Figure 12.8: Validating Toolbox Entries Using a Context Free Grammar

We could try to write down a grammar for lexical entries, and look for entries which do not conform to the grammar. In general, toolbox entries have nested structure. Thus they correspond to a tree over the fields. We can check for well-formedness by parsing the field names. In [Figure 12.9](#) we set up a putative grammar for the entries, then parse each entry. Those that are accepted by the grammar prefixed with a '+', and those that are rejected are prefixed with a '-'.

```

import os.path, sys
from nltk_contrib import toolbox

grammar = r"""
lexfunc: {<lf>(<lv><ln|le>*)*}
example: {<rf|xv><xn|xe>*}
sense:   {<sn><ps><pn|gv|dv|gn|gp|dn|rн|ge|de|re>*<example>*<lexfunc>*}
record:  {<lx><hm><sense>+<dt>}
"""

>>> from nltk.etree.ElementTree import ElementTree
>>> db = toolbox.ToolboxData()
>>> db.open(nltk.data.find('corpora/toolbox/iu_mien_samp.db'))
>>> lexicon = db.chunk_parse(grammar, encoding='utf8')
>>> toolbox.data.indent(lexicon)
>>> tree = ElementTree(lexicon)
>>> tree.write(sys.stdout, encoding='utf8')

```

Figure 12.9 (chunk_toolbox.py): Figure 12.9: Chunking a Toolbox Lexicon

Interlinear Text

The NLTK corpus collection includes many interlinear text samples (though no suitable corpus reader as yet).

General Ontology for Linguistic Description (GOLD) <http://www.linguistics-ontology.org/>

Creating Metadata for Language Resources

OLAC metadata extends the **Dublin Core** metadata set with descriptors that are important for language resources.

The container for an OLAC metadata record is the element `<olac>`. Here is a valid OLAC metadata record from the Pacific And Regional Archive for Digital Sources in Endangered Cultures (PARADISEC):

```

<olac:olac xsi:schemaLocation="http://purl.org/dc/elements/1.1/ http://www.language-archives.org/OLAC/1.
  http://purl.org/dc/terms/ http://www.language-archives.org/OLAC/1.0/dcterms.xsd
  http://www.language-archives.org/OLAC/1.0/ http://www.language-archives.org/OLAC/1.0/olac.xsd">
  <dc:title>Tiraq Field Tape 019</dc:title>
  <dc:identifier>AB1-019</dc:identifier>
  <dcterms:hasPart>AB1-019-A.mp3</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-A.wav</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-B.mp3</dcterms:hasPart>
  <dcterms:hasPart>AB1-019-B.wav</dcterms:hasPart>
  <dc:contributor xsi:type="olac:role" olac:code="recorder">Brotchie, Amanda</dc:contributor>
  <dc:subject xsi:type="olac:language" olac:code="x-sil-MME"/>
  <dc:language xsi:type="olac:language" olac:code="x-sil-BCY"/>
  <dc:language xsi:type="olac:language" olac:code="x-sil-MME"/>
  <dc:format>Digitised: yes;</dc:format>
  <dc:type>primary_text</dc:type>

```

```
<dcterms:accessRights>standard, as per PDSC Access form</dcterms:accessRights>
<dc:description>SIDE A<p>1. Elicitation Session - Discussion and
translation of Lise's and Marie-Claire's Songs and Stories from
Tape 18 (Tamedal)<p><p>SIDE B<p>1. Elicitation Session: Discussion
of and translation of Lise's and Marie-Claire's songs and stories
from Tape 018 (Tamedal)<p>2. Kastom Story 1 - Bislama
(Alec). Language as given: Tiraq</dc:description>
</olac:olac>
```

NLTK Version 0.9 includes support for reading an OLAC record, for example:

```
>>> file = nltk.data.find('corpora/treebank/olac.xml')
>>> xml = open(file).read()
>>> nltk.olac pprint.olac(xml)
identifier : LDC99T42
title : Treebank-3
type : (olac:linguistic-type=primary_text)
description : Release type: General
creator : Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz and Ann Taylor (olac:r
identifier : ISBN: 1-58563-163-9
description : Online documentation: http://www.ldc.upenn.edu/Catalog/docs/treebank3/
subject : English (olac:language=x-sil-ENG)
```

Linguistic Annotation

Annotation graph model

multiple overlapping trees over shared data

Large annotation tasks require multiple annotators. How consistently can a group of annotators perform? It is insufficient to report that there is 80% agreement, as we have no way to tell if this is good or bad. I.e. for an easy task such as tagging, this would be a bad score, while for a difficult task such as semantic role labeling, this would be an exceptionally good score.

The **Kappa** coefficient K measures agreement between two people making category judgments, correcting for expected chance agreement. For example, suppose an item is to be annotated, and four coding options are equally likely. Then people coding randomly would be expected to agree 25% of the time. Thus, an agreement of 25% will be assigned K = 0, and better levels of agreement will be scaled accordingly. For an agreement of 50%, we would get K = 0.333, as 50 is a third of the way from 25 to 100.

12.4 Converting Data Formats

- write our own parser and formatted print
- use existing libraries, e.g. csv

Formatting Entries

We can also print a formatted version of a lexicon. It allows us to request specific fields without needing to be concerned with their relative ordering in the original file.

```
>>> lexicon = toolbox.xml('rotokas.dic')
>>> for entry in lexicon[70:80]:
...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     ge = entry.findtext('ge')
...     print "%s (%s) '%s'" % (lx, ps, ge)
kakae (???) 'small'
kakae (CLASS) 'child'
kakaevira (ADV) 'small-like'
kakapikoa (???) 'small'
kakapikoto (N) 'newborn baby'
kakapu (V) 'place in sling for purpose of carrying'
kakapua (N) 'sling for lifting'
kakara (N) 'arm band'
Kakarapaia (N) 'village name'
kakarau (N) 'frog'
```

We can use the same idea to generate HTML tables instead of plain text. This would be useful for publishing a Toolbox lexicon on the web. It produces HTML elements `<table>`, `<tr>` (table row), and `<td>` (table data).

```
>>> html = "<table>\n"
>>> for entry in lexicon[70:80]:
...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     ge = entry.findtext('ge')
...     html += " <tr><td>%s</td><td>%s</td><td>%s</td></tr>\n" % (lx, ps, ge)
>>> html += "</table>"
>>> print html


|            |       |                                        |
|------------|-------|----------------------------------------|
| kakae      |       | small                                  |
| kakae      | CLASS | child                                  |
| kakaevira  | ADV   | small-like                             |
| kakapikoa  | ???   | small                                  |
| kakapikoto | N     | newborn baby                           |
| kakapu     | V     | place in sling for purpose of carrying |
| kakapua    | N     | sling for lifting                      |
| kakara     | N     | arm band                               |
| Kakarapaia | N     | village name                           |
| kakarau    | N     | frog                                   |


```

XML output

```
>>> import sys
>>> from nltk.etree.ElementTree import ElementTree
>>> tree = ElementTree(lexicon[3])
>>> tree.write(sys.stdout)
<record>
  <lx>kaa</lx>
  <ps>N</ps>
  <pt>MASC</pt>
  <cl>isi</cl>
  <ge>cooking banana</ge>
  <tkp>banana bilong kukim</tkp>
  <pt>itoo</pt>
  <sf>FLORA</sf>
  <dt>12/Aug/2005</dt>
  <ex>Taeavi iria kaa isi kovopaevea kaparapasia.</ex>
  <xp>Taeavi i bin planim gaden banana bilong kukim tasol long paia.</xp>
  <xe>Taeavi planted banana in order to cook it.</xe>
</record>
```

Exercises

- Create a spreadsheet using office software, containing one lexical entry per row, consisting of a headword, a part of speech, and a gloss. Save the spreadsheet in CSV format. Write Python code to read the CSV file and print it in Toolbox format, using `lx` for the headword, `ps` for the part of speech, and `gl` for the gloss.

12.5 Analyzing Language Data

I.e. linguistic exploration

Export to statistics package via CSV

In this section we consider a variety of analysis tasks.

Reduplication: First, we will develop a program to find reduplicated words. In order to do this we need to store all verbs, along with their English glosses. We need to keep the glosses so that they can be displayed alongside the wordforms. The following code defines a Python dictionary `lexgloss` which maps verbs to their English glosses:

```
>>> lexgloss = {}
>>> for entry in lexicon:
```

```

...     lx = entry.findtext('lx')
...     ps = entry.findtext('ps')
...     if lx and ps and ps[0] == 'V':
...         lexgloss[lx] = entry.findtext('ge')
kasi (burn); kasikasi (angry)
kee (shatter); keekee (chipped)
kauo (jump); kauokaupo (jump up and down)
kea (confused); keakea (lie)
kaape (unable to meet); kaapekaape (embrace)
kapo (fasten.cover.strip); kapokapo (fasten.cover.strip)
kavo (collect); kavokavo (perform sorcery)
karu (open); karukaru (open)
kare (return); karekare (return)
kari (rip); karikari (tear)
kae (blow); kaekae (tempt)

```

Next, for each verb `lex`, we will check if the lexicon contains the reduplicated form `lex+lex`. If it does, we report both forms along with their glosses.

```

>>> for lex in lexgloss:
...     if lex+lex in lexgloss:
...         print "%s (%s); %s (%s)" %\
...             (lex, lexgloss[lex], lex+lex, lexgloss[lex+lex])
kuvu (fill.up); kuvukuvu (fill up)
kitu (store); kitukitu (scrub clothes)
kiru (have sore near mouth); kirukiru (crisp)
kopa (swallow); kopakopa (gulp.down)
kasi (burn); kasikasi (angry)
koi (high pitched sound); koikoi (groan with pain)
kee (shatter); keekee (chipped)
kauo (jump); kauokaupo (jump up and down)
kea (confused); keakea (lie)
kovo (work); kovokovo (play)
kove (fell); kovekove (drip repeatedly)
kaape (unable to meet); kaapekaape (embrace)
kapo (fasten.cover.strip); kapokapo (fasten.cover.strip)
koa (skin); koakoa (bark a tree)
kipu (paint); kipukipu (rub.on)
koe (spoon out a solid); koekoe (spoon out)
kotu (bite); kotukotu (gnash teeth)
kavo (collect); kavokavo (perform sorcery)
kuri (scrape); kurikuri (scratch repeatedly)
karu (open); karukaru (open)
kare (return); karekare (return)
kari (rip); karikari (tear)
kiro (write); kirokiro (write)
kae (blow); kaekae (tempt)
koru (make return); korukoru (block)
kosi (exit); kosikosi (exit)

```

Complex Search Criteria: Phonological description typically identifies the segments, alternations, syllable canon and so forth. It is relatively straightforward to count up the occurrences of all the different types of CV syllables that occur in lexemes.

In the following example, we first import the regular expression and probability modules. Then we iterate over the lexemes to find all sequences of a non-vowel `[^aeiou]` followed by a vowel `[aeiou]`.

```

>>> fd = nltk.FreqDist()
>>> tokenizer = nltk.RegexpTokenizer(pattern=r'[^aeiou][aeiou]')
>>> lexemes = [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
>>> for lex in lexemes:
...     for syl in tokenizer.tokenize(lex):
...         fd.inc(syl)

```

Now, rather than just printing the syllables and their frequency counts, we can tabulate them to generate a useful display.

```

>>> for vowel in 'aeiou':
...     for cons in 'ptkvsr':
...         print '%s%s:%4d' % (cons, vowel, fd[cons+vowel]),

```

```

...     print
pa: 83 ta: 47 ka: 428 va: 93 sa: 0 ra: 187
pe: 31 te: 8 ke: 151 ve: 27 se: 0 re: 63
pi: 105 ti: 0 ki: 94 vi: 105 si: 100 ri: 84
po: 34 to: 148 ko: 430 vo: 48 so: 2 ro: 89
pu: 51 tu: 37 ku: 175 vu: 49 su: 1 ru: 79

```

Consider the `t` and `s` columns, and observe that `ti` is not attested, while `si` is frequent. This suggests that a phonological process of palatalization is operating in the language. We would then want to consider the other syllables involving `s` (e.g. the single entry having `su`, namely *kasuari* 'cassowary' is a loanword).

Prosodically-motivated search: A phonological description may include an examination of the segmental and prosodic constraints on well-formed morphemes and lexemes. For example, we may want to find trisyllabic verbs ending in a long vowel. Our program can make use of the fact that syllable onsets are obligatory and simple (only consist of a single consonant). First, we will encapsulate the syllabic counting part in a separate function. It gets the CV template of the word `cv(word)` and counts the number of consonants it contains:

```

>>> def num_cons(word):
...     template = cv(word)
...     return template.count('C')

```

We also encapsulate the vowel test in a function, as this improves the readability of the final program. This function returns the value `True` just in case `char` is a vowel.

```

>>> def is_vowel(char):
...     return (char in 'aeiou')

```

Over time we may create a useful collection of such functions. We can save them in a file `utilities.py`, and then at the start of each program we can simply import all the functions in one go using `from utilities import *`. We take the entry to be a verb if the first letter of its part of speech is a `V`. Here, then, is the program to display trisyllabic verbs ending in a long vowel:

```

>>> for entry in lexicon:
...     lx = entry.findtext('lx')
...     if lx:
...         ps = entry.findtext('ps')
...         if num_cons(lx) == 3 and ps[0] == 'V' \
...             and is_vowel(lx[-1]) and is_vowel(lx[-2]):
...             ge = entry.findtext('ge')
...             print "%s (%s) '%s'" % (lx, ps, ge)
kaetupie (V) 'tighten'
kakupie (V) 'shout'
kapatau (V) 'add to'
kapuapie (V) 'wound'
kapupie (V) 'close tight'
kapuupie (V) 'close'
karepie (V) 'return'
karivai (V) 'have an appetite'
kasipie (V) 'care for'
kasirao (V) 'hot'
kaukaupie (V) 'shine intensely'
kavorou (V) 'covet'
kavupie (V) 'leave.behind'
kekepie (V) 'show'
keruria (V) 'persistent'
ketoopie (V) 'make sprout from seed'
kipapie (V) 'wan samting tru'
koatapie (V) 'put in'
koetapie (V) 'investigate'
koikoipie (V) 'make groan with pain'
kokepie (V) 'make.rain'
kokoruu (V) 'insect-infested'
kokovae (V) 'sing'
kokovua (V) 'shave the hair line'
kopipiie (V) 'kill'
korupie (V) 'take outside'
kosipie (V) 'make exit'
kovopie (V) 'make work'

```

```
kukuvai (V) 'shelter head'
kuvaupie (V) 'desert'
```

Finding Minimal Sets: In order to establish a contrast segments (or lexical properties, for that matter), we would like to find pairs of words which are identical except for a single property. For example, the words pairs *mace* vs *maze* and *face* vs *faze* — and many others like them — demonstrate the existence of a phonemic distinction between *s* and *z* in English. NLTK provides flexible support for constructing minimal sets, using the `MinimalSet()` class. This class needs three pieces of information for each item to be added: `context`: the material that must be fixed across all members of a minimal set; `target`: the material that changes across members of a minimal set; `display`: the material that should be displayed for each item.

Table 12.1:

Examples of Minimal Set Parameters

| Minimal Set | Context | Target | Display |
|---------------------------|-------------------|--------------|---------|
| <i>bib, bid, big</i> | first two letters | third letter | word |
| <i>deal (N), deal (V)</i> | whole word | pos | (pos) |

We begin by creating a list of parameter values, generated from the full lexical entries. In our first example, we will print minimal sets involving lexemes of length 4, with a target position of 1 (second segment). The `context` is taken to be the entire word, except for the target segment. Thus, if `lex` is *kasi*, then `context` is `lex[:1] + '_' + lex[2:]`, or *k_si*. Note that no parameters are generated if the lexeme does not consist of exactly four segments.

```
>>> pos = 1
>>> ms = nltk.MinimalSet((lex[:pos] + '_' + lex[pos+1:], lex[pos], lex)
...                                     for lex in lexemes if len(lex) == 4)
```

Now we print the table of minimal sets. We specify that each context was seen at least 3 times.

```
>>> for context in ms.contexts(3):
...     print context + ':',
...     for target in ms.targets():
...         print "%-4s" % ms.display(context, target, "-"),
...     print
k_si: kasi -    kesi kusi kosi
k_va: kava -    -    kuva kova
k_ru: karu kiru keru kuru koru
k_pu: kapu kipu -    -    kopu
k_ro: karo kiro -    -    koro
k_ri: kari kiri keri kuri kori
k_pa: kapa -    kepa -    kopa
k_ra: kara kira kera -    kora
k_ku: kaku -    -    kuku koku
k_ki: kaki kiki -    -    koki
```

Observe in the above example that the context, target, and displayed material were all based on the lexeme field. However, the idea of minimal sets is much more general. For instance, suppose we wanted to get a list of wordforms having more than one possible part-of-speech. Then the target will be part-of-speech field, and the context will be the lexeme field. We will also display the English gloss field.

```
>>> entries = [(e.findtext('lx'), e.findtext('ps'), e.findtext('ge'))
...               for e in lexicon
...               if e.findtext('lx') and e.findtext('ps') and e.findtext('ge')]
>>> ms = nltk.MinimalSet((lx, ps[0], "%s (%s)" % (ps[0], ge))
...                         for (lx, ps, ge) in entries)
>>> for context in ms.contexts()[:10]:
...     print "%10s:" % context, "; ".join(ms.display_all(context))
kokovara: N (unripe coconut); V (unripe)
kapua: N (sore); V (have sores)
koie: N (pig); V (get pig to eat)
kovo: C (garden); N (garden); V (work)
kavori: N (crayfish); V (collect crayfish or lobster)
korita: N (cutlet?); V (dissect meat)
keru: N (bone); V (harden like bone)
```

```

kirokiro: N (bush used for sorcery); V (write)
kaapie: N (hook); V (snag)
kou: C (heap); V (lay egg)

```

The following program uses `MinimalSet` to find pairs of entries in the corpus which have different attachments based on the *verb* only.

```

>>> ms = nltk.MinimalSet()
>>> for entry in nltk.corpus.p�attach.attachments('training'):
...     target = entry.attachment
...     context = (entry.noun1, entry.prep, entry.noun2)
...     display = (target, entry.verb)
...     ms.add(context, target, display)
>>> for context in ms.contexts():
...     print context, ms.display_all(context)

```

Here is one of the pairs found by the program.

- (216) received (NP offer) (PP from group)
 rejected (NP offer (PP from group))

This finding gives us clues to a structural difference: the verb *receive* usually comes with two following arguments; we receive something *from* someone. In contrast, the verb *reject* only needs a single following argument; we can reject something without needing to say where it originated from.

12.6 Summary

- diverse motivations for corpus collection
- corpus structure, balance, documentation
- OLAC

12.7 Further Reading

Shoebox/Toolbox and other tools for field linguistic data management: Full details of the Shoebox data format are provided with the distribution [[Buseman, Buseman, & Early, 1996](#)], and with the latest distribution, freely available from <http://www.sil.org/computing/toolbox/>. Many other software tools support the format. More examples of our efforts with the format are documented in [[Tamanji, Hirotani, & Hall, 1999](#)], [[Robinson, Aumann, & Bird, 2007](#)]. Dozens of other tools for linguistic data management are available, some surveyed by [[Bird & Simons, 2003](#)].

Some Major Corpora: The primary sources of linguistic corpora are the *Linguistic Data Consortium* and the *European Language Resources Agency*, both with extensive online catalogs. More details concerning the major corpora mentioned in the chapter are available: American National Corpus [[Reppen, Ide, & Suderman, 2005](#)], British National Corpus [[{BNC}, 1999](#)], Thesaurus Linguae Graecae [[{TLG}, 1999](#)], Child Language Data Exchange System (CHILDES) [[MacWhinney, 1995](#)], TIMIT [[S., Lamel, & William, 1986](#)]. The following papers give accounts of work on corpora that put them to entirely different uses than were envisaged at the time they were created [[Graff & Bird, 2000](#)], [[Cieri & Strassel, 2002](#)].

Annotation models and tools: An extensive set of models and tools are available, surveyed at <http://www.exmaralda.org/annotation/>. The initial proposal for standoff annotation was [[Thompson & McKelvie, 1997](#)]. The Annotation Graph model was proposed by [[Bird & Liberman, 2001](#)].

Scoring measures: Full details of the two scoring methods are available: Kappa: [[Carletta, 1996](#)], Windowdiff: [[Pevzner & Hearst, 2002](#)].

12.8 Exercises

1. ☀ Write a program to filter out just the date field (`dt`) without having to list the fields we wanted to retain.
2. ☀ Print an index of a lexicon. For each lexical entry, construct a tuple of the form (`gloss, lexeme`), then sort and print them all.
3. ☀ What is the frequency of each consonant and vowel contained in lexeme fields?
4. ● In [Figure 12.7](#) the new field appeared at the bottom of the entry. Modify this program so that it inserts the new

subelement right after the `lx` field. (Hint: create the new `cv` field using `Element('cv')`, assign a text value to it, then use the `insert()` method of the parent element.)

5. ● Write a function that deletes a specified field from a lexical entry. (We could use this to sanitize our lexical data before giving it to others, e.g. by removing fields containing irrelevant or uncertain content.)
6. ● Write a program that scans an HTML dictionary file to find entries having an illegal part-of-speech field, and reports the *headword* for each entry.
7. ● Write a program to find any parts of speech (`ps` field) that occurred less than ten times. Perhaps these are typing mistakes?
8. ● We saw a method for discovering cases of whole-word reduplication. Write a function to find words that may contain partial reduplication. Use the `re.search()` method, and the following regular expression: `(.+)\\1`
9. ● We saw a method for adding a `cv` field. There is an interesting issue with keeping this up-to-date when someone modifies the content of the `lx` field on which it is based. Write a version of this program to add a `cv` field, replacing any existing `cv` field.
10. ● Write a function to add a new field `sy1` which gives a count of the number of syllables in the word.
11. ● Write a function which displays the complete entry for a lexeme. When the lexeme is incorrectly spelled it should display the entry for the most similarly spelled lexeme.
12. ● Write a function that takes a lexicon and finds which pairs of consecutive fields are most frequent (e.g. `ps` is often followed by `pt`). (This might help us to discover some of the structure of a lexical entry.)
13. ★ Obtain a comparative wordlist in CSV format, and write a program that prints those cognates having an edit-distance of at least three from each other.
14. ★ Build an index of those lexemes which appear in example sentences. Suppose the lexeme for a given entry is `w`. Then add a single cross-reference field `xref` to this entry, referencing the headwords of other entries having example sentences containing `w`. Do this for all entries and save the result as a toolbox-format file.

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

A Appendix: Tagsets

A.1 Brown Tagset

Based on <http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html>

Table A.1

| Brown Tag | Description | Examples |
|-----------|--|---|
| (| opening parenthesis | (|
|) | closing parenthesis |) |
| * | negator | not n't |
| , | comma | , |
| -- | dash | -- |
| . | sentence terminator | . ? ; ! : |
| : | colon | : |
| ABL | determiner/pronoun, pre-qualifier | quite such rather |
| ABN | determiner/pronoun, pre-quantifier | all half many nary |
| ABX | determiner/pronoun, double conjunction or pre-quantifier | both |
| AP | determiner/pronoun, post-determiner | many other next more last former little several enough most least only very few fewer past same Last latter less single plenty 'nough lesser certain various manye next-to-last particular final previous present nuf |

| Brown Tag | Description | Examples |
|-----------|--|---|
| AP\$ | determiner/pronoun, post-determiner, genitive | other's |
| AP+AP | determiner/pronoun, post-determiner, hyphenated pair | many-much |
| AT | article | the an no a every th' ever' ye |
| BE | verb 'to be', infinitive or imperative | be |
| BED | verb 'to be', past tense, 2nd person singular or all persons plural | were |
| BED* | verb 'to be', past tense, 2nd person singular or all persons plural, negated | weren't |
| BEDZ | verb 'to be', past tense, 1st and 3rd person singular | was |
| BEDZ* | verb 'to be', past tense, 1st and 3rd person singular, negated | wasn't |
| BEG | verb 'to be', present participle or gerund | being |
| BEM | verb 'to be', present tense, 1st person singular | am |
| BEM* | verb 'to be', present tense, 1st person singular, negated | ain't |
| BEN | verb 'to be', past participle | been |
| BER | verb 'to be', present tense, 2nd person singular or all persons plural | are art |
| BER* | verb 'to be', present tense, 2nd person singular or all persons plural, negated | aren't ain't |
| BEZ | verb 'to be', present tense, 3rd person singular | is |
| BEZ* | verb 'to be', present tense, 3rd person singular, negated | isn't ain't |
| CC | conjunction, coordinating | and or but plus & either neither nor yet 'n' and/or minus an' |
| CD | numeral, cardinal | two one 1 four 2 1913 71 74 637 1937 8 five three million 87-31 29-5 seven 1,119 fifty-three 7.5 billion hundred 125,000 1,700 60 100 six ... 1960's 1961's .404's |
| CD\$ | numeral, cardinal, genitive | that as after whether before while like because if since for than |
| CS | conjunction, subordinating | altho until so unless though providing once lest s'posin' till whereas whereupon supposing tho' albeit then so's 'fore do dost |
| DO | verb 'to do', uninflected present tense, infinitive or imperative | don't |
| DO* | verb 'to do', uninflected present tense or imperative, negated | d'you |
| DO+PPSS | verb 'to do', past or present tense + pronoun, personal, nominative, not 3rd person singular | did done didn't does |
| DOD | verb 'to do', past tense | doesn't don't |
| DOD* | verb 'to do', past tense, negated | this each another that 'nother |
| DOZ | verb 'to do', present tense, 3rd person singular | another's |
| DOZ* | verb 'to do', present tense, 3rd person singular, negated | that's |
| DT | determiner/pronoun, singular | that'll this'll |
| DT\$ | determiner/pronoun, singular, genitive | any some |
| DT+BEZ | determiner/pronoun + verb 'to be', present tense, 3rd person singular | these those them |
| DT+MD | determiner/pronoun + modal auxiliary | them's |
| DTI | determiner/pronoun, singular or plural | |
| DTS | determiner/pronoun, plural | |
| DTS+BEZ | pronoun, plural + verb 'to be', present tense, 3rd person singular | |

| Brown Tag | Description | Examples |
|-----------|--|---|
| DTX | determiner, pronoun or double conjunction | neither either one |
| EX | existential there | there |
| EX+BEZ | existential there + verb 'to be', present tense, 3rd person singular | there's |
| EX+HVD | existential there + verb 'to have', past tense | there'd |
| EX+HVZ | existential there + verb 'to have', present tense, 3rd person singular | there's |
| EX+MD | existential there + modal auxillary | there'll there'd |
| FW-* | foreign word: negator | pas non ne |
| FW-AT | foreign word: article | la le el un die der ein keine eine das las les Il |
| FW-AT+NN | foreign word: article + noun, singular, common | l'orchestre l'identite l'arcade l'ange l'assistance l'activite L'Universite l'independance L'Union L'Unita l'ossevatore L'Astree L'Imperiale |
| FW-AT+NP | foreign word: article + noun, singular, proper | |
| FW-BE | foreign word: verb 'to be', infinitive or imperative | sit |
| FW-BER | foreign word: verb 'to be', present tense, 2nd person singular or all persons plural | sind sunt etes |
| FW-BEZ | foreign word: verb 'to be', present tense, 3rd person singular | ist est |
| FW-CC | foreign word: conjunction, coordinating | et ma mais und aber och nec y |
| FW-CD | foreign word: numeral, cardinal | une cinq deux sieben unam zwei |
| FW-CS | foreign word: conjunction, subordinating | bevor quam ma |
| FW-DT | foreign word: determiner/pronoun, singular | hoc |
| FW-DT+BEZ | foreign word: determiner + verb 'to be', present tense, 3rd person singular | c'est |
| FW-DTS | foreign word: determiner/pronoun, plural | haec |
| FW-HV | foreign word: verb 'to have', present tense, not 3rd person singular | habe |
| FW-IN | foreign word: preposition | ad de en a par con dans ex von auf super post sine sur sub avec per inter sans pour pendant in di della des du aux zur d'un del dell' d'etat d'hotel d'argent d'identite d'art |
| FW-IN+AT | foreign word: preposition + article | |
| FW-IN+NN | foreign word: preposition + noun, singular, common | |
| FW-IN+NP | foreign word: preposition + noun, singular, proper | d'Yquem d'Eiffel |
| FW-JJ | foreign word: adjective | |
| FW-JJR | foreign word: adjective, comparative | avant Espagnol sinfonica Siciliana Philharmonique grand |
| FW-JJT | foreign word: adjective, superlative | publique haute noire bouffe Douce meme humaine bel |
| FW-NN | foreign word: noun, singular, common | serieuses royaux anticus presto Sovietskaya Bayerische comique schwarzen ... |
| FW-NN\$ | foreign word: noun, singular, common, genitive | fortiori |
| FW-NNS | foreign word: noun, plural, common | optimo |
| FW-NP | foreign word: noun, singular, proper | ballet esprit ersatz mano chatte goutte sang Fledermaus oud def kolkhoz roi troika canto boite blutwurst carne muzyka bonheur monde piece force ... |
| FW-NPS | foreign word: noun, plural, proper | corporis intellectus arte's dei aeternitatis senioritatis curiae patronne's chambre's al culpas vopos boites haflis kolkhozes augen tyrannis alpha-beta-gammes metis banditos rata phis negociants crus Einsatzkommandos kamikaze wohaws sabinas zorrillas palazzi engages coureurs corroborees yori Ubermenschen ... Karshilama Dieu Rundfunk Afrique Espanol Afrika Spagna Gott Carthago deus Svenskarna Atlantes Dieux |

| Brown Tag | Description | Examples |
|------------|---|--|
| FW-NR | foreign word: noun, singular, adverbial | heute morgen aujourd'hui hoy |
| FW-OD | foreign word: numeral, ordinal | 18e 17e quintus |
| FW-PN | foreign word: pronoun, nominal | hoc |
| FW-PP\$ | foreign word: determiner, possessive | mea mon deras vos |
| FW-PPL | foreign word: pronoun, singular, reflexive | se |
| FW-PPL+VBZ | foreign word: pronoun, singular, reflexive + verb, present tense, 3rd person singular | s'excuse s'accuse |
| FW-PPO | pronoun, personal, accusative | lui me moi mi |
| FW-PPO+IN | foreign word: pronoun, personal, accusative + preposition | mecum tecum |
| FW-PPS | foreign word: pronoun, personal, nominative, 3rd person singular | il |
| FW-PPSS | foreign word: pronoun, personal, nominative, not 3rd person singular | ich vous sie je |
| FW-PPSS+HV | foreign word: pronoun, personal, nominative, not 3rd person singular + verb 'to have', present tense, not 3rd person singular | j'ai |
| FW-QL | foreign word: qualifier | minus |
| FW-RB | foreign word: adverb | bas assai deja um wiederum cito velociter vielleicht simpliciter non zu domi nuper sic forsan olim oui semper tout despues hors forisque |
| FW-RB+CC | foreign word: adverb + conjunction, coordinating | d'entretenir |
| FW-TO+VB | foreign word: infinitival to + verb, infinitive | sayonara bien adieu arigato bonjour adios bueno tchalo ciao o nolo contendere vive fermate faciunt esse vade noli tangere dites duces meminisse iuvabit gosaimasu voulez habla ksu'u'pel'i'af'o lacheln miuchi say allons strafe portant stabat peccavi audivi nolens volens appellant seq. oblitterans servanda dicendi delenda |
| FW-UH | foreign word: interjection | vue verstrichen rasa verboten engages gouverne sinkt sigue diapiace |
| FW-VB | foreign word: verb, present tense, not 3rd person singular, imperative or infinitive | quo qua quod que quok quibusdam qui have hast haven't ain't hafta had hadn't having had has hath hasn't ain't |
| FW-VBD | foreign word: verb, past tense | |
| FW-VBG | foreign word: verb, present participle or gerund | |
| FW-VBN | foreign word: verb, past participle | |
| FW-VBZ | foreign word: verb, present tense, 3rd person singular | |
| FW-WDT | foreign word: WH-determiner | |
| FW-WPO | foreign word: WH-pronoun, accusative | |
| FW-WPS | foreign word: WH-pronoun, nominative | |
| HV | verb 'to have', uninflected present tense, infinitive or imperative | |
| HV* | verb 'to have', uninflected present tense or imperative, negated | |
| HV+TO | verb 'to have', uninflected present tense + infinitival to | |
| HVD | verb 'to have', past tense | |
| HVD* | verb 'to have', past tense, negated | |
| HVG | verb 'to have', present participle or gerund | |
| HVN | verb 'to have', past participle | |
| HVZ | verb 'to have', present tense, 3rd person singular | |
| HVZ* | verb 'to have', present tense, 3rd person singular, negated | |

| Brown Tag | Description | Examples |
|-----------|---|---|
| IN | preposition | of in for by considering to on among at through with under into regarding than since despite according per before toward against as after during including between without except upon out over ... f'ovuh t'hi-im |
| IN+IN | preposition, hyphenated pair | |
| IN+PPO | preposition + pronoun, personal, accusative | |
| JJ | adjective | ecent over-all possible hard-fought favorable hard meager fit such widespread outmoded inadequate ambiguous grand clerical effective orderly federal foster general proportionate ... Great's big-large long-far |
| JJ\$ | adjective, genitive | greater older further earlier later freer franker wider better deeper firmer tougher faster higher bigger worse younger lighter nicer slower happier frothier Greater newer Elder ... lighter'n |
| JJ+JJ | adjective, hyphenated pair | top chief principal northernmost master key head main tops utmost innermost foremost uppermost paramount topmost best largest coolest calmest latest greatest earliest simplest strongest newest fiercest unhappiest worst youngest worthiest fastest hottest fittest lowest finest smallest staunchest ... should may might will would must can could shall ought need wilt |
| JJR | adjective, comparative | cannot couldn't wouldn't can't won't shouldn'tshan't mustn't musn't shouldda musta coulda must've woulda could've |
| JJR+CS | adjective + conjunction, coordinating | willya |
| JJS | adjective, semantically superlative | oughta |
| JJT | adjective, superlative | failure burden court fire appointment awarding compensation Mayor interim committee fact effect airport management surveillance jail doctor intern extern night weekend duty legislation Tax Office ... season's world's player's night's chapter's golf's football's baseball's club's U.'s coach's bride's bridegroom's board's county's firm's company's superintendent's mob's Navy's ... water's camera's sky's kid's Pa's heat's throat's father's money's undersecretary's granite's level's wife's fat's Knife's fire's name's hell's leg's sun's roulette's cane's guy's kind's baseball's ... Pa'd |
| MD | modal auxillary | |
| MD* | modal auxillary, negated | |
| MD+HV | modal auxillary + verb 'to have', uninflected form | |
| MD+PPSS | modal auxillary + pronoun, personal, nominative, not 3rd person singular | |
| MD+TO | modal auxillary + infinitival to | |
| NN | noun, singular, common | |
| NN\$ | noun, singular, common, genitive | |
| NN+BEZ | noun, singular, common + verb 'to be', present tense, 3rd person singular | |
| NN+HVD | noun, singular, common + verb 'to have', past tense | |
| NN+HVZ | noun, singular, common + verb 'to have', present tense, 3rd person singular | |
| NN+IN | noun, singular, common + preposition | |
| NN+MD | noun, singular, common + modal auxillary | |
| NN+NN | noun, singular, common, hyphenated pair | |
| NNS | noun, plural, common | |
| NNS\$ | noun, plural, common, genitive | |
| NNS+MD | noun, plural, common + modal auxillary | |

| Brown Tag | Description | Examples |
|-----------|---|---|
| NP | noun, singular, proper | Fulton Atlanta September-October Durwood Pye Ivan Allen Jr. Jan. Alpharetta Grady William B. Hartsfield Pearl Williams Aug. Berry J. M. Cheshire Griffin Opelika Ala. E. Pelham Snodgrass ... |
| NP\$ | noun, singular, proper, genitive | Green's Landis' Smith's Carreon's Allison's Boston's Spahn's Willie's Mickey's Milwaukee's Mays' Howsam's Mantle's Shaw's Wagner's Rickey's Shea's Palmer's Arnold's Broglio's ... W.'s Ike's Mack's Jack's Kate's Katharine's Black's Arthur's Seaton's Buckhorn's Breed's Penny's Rob's Kitty's Blackwell's Myra's Wally's Lucille's Springfield's Arlene's Bill's Guardino's Celie's Skolman's Crosson's Tim's Wally's |
| NP+BEZ | noun, singular, proper + verb 'to be', present tense, 3rd person singular | Gyp'll John'll |
| NP+HVZ | noun, singular, proper + verb 'to have', present tense, 3rd person singular | Chases Aderholds Chapelles Armisteads Lockies Carbones |
| NP+MD | noun, singular, proper + modal auxillary | French Marskmen Toppers Franciscans Romans Cadillacs |
| NPS | noun, plural, proper | Masons Blacks Catholics British Dixiecrats Mississippians Congresses ... |
| NPS\$ | noun, plural, proper, genitive | Republicans' Orioles' Birds' Yanks' Redbirds' Bucs' Yankees' Stevenses' Geraghtys' Burkes' Wackers' Achaeans' Dresbachs' Russians' Democrats' Gershwins' Adventists' Negroes' Catholics' ... |
| NR | noun, singular, adverbial | Friday home Wednesday Tuesday Monday Sunday Thursday yesterday tomorrow tonight West East Saturday west left east downtown north northeast southeast northwest North South right ... |
| NR\$ | noun, singular, adverbial, genitive | Saturday's Monday's yesterday's tonight's tomorrow's Sunday's Wednesday's Friday's today's Tuesday's West's Today's South's today'll |
| NR+MD | noun, singular, adverbial + modal auxillary | Sundays Mondays Saturdays Wednesdays Souths Fridays first 13th third nineteenth 2d 61st second sixth eighth ninth twenty-first eleventh 50th eighteenth- Thirty-ninth 72nd 1/20th twentieth mid-19th thousandth 350th sixteenth 701st ... |
| NRS | noun, plural, adverbial | none something everything one anyone nothing nobody everybody everyone anybody anything someone no-one nothin |
| OD | numeral, ordinal | one's someone's anybody's nobody's everybody's anyone's everyone's |
| PN | pronoun, nominal | nothing's everything's somebody's nobody's someone's |
| PN\$ | pronoun, nominal, genitive | nobody'd |
| PN+BEZ | pronoun, nominal + verb 'to be', present tense, 3rd person singular | nobody's somebody's one's |
| PN+HVD | pronoun, nominal + verb 'to have', past tense | someone'll somebody'll anybody'd our its his their my your her out thy mine thine |
| PN+HVZ | pronoun, nominal + verb 'to have', present tense, 3rd person singular | ours mine his hers theirs yours itself himself myself yourself herself oneself ownself |
| PN+MD | pronoun, nominal + modal auxillary | themselves ourselves yourselves |
| PP\$ | determiner, possessive | them it him me us you 'em her thee we'unus |
| PP\$\$ | pronoun, possessive | it he she thee |
| PPL | pronoun, singular, reflexive | it's he's she's |
| PPLS | pronoun, plural, reflexive | |
| PPO | pronoun, personal, accusative | |
| PPS | pronoun, personal, nominative, 3rd person singular | she'd he'd it'd |
| PPS+BEZ | pronoun, personal, nominative, 3rd person singular + verb 'to be', present tense, 3rd person singular | |
| PPS+HVD | pronoun, personal, nominative, 3rd person singular + verb 'to have', past tense | |

| Brown Tag | Description | Examples |
|-----------|---|---|
| PPS+HVZ | pronoun, personal, nominative, 3rd person singular + verb 'to have', present tense, 3rd person singular | it's he's she's |
| PPS+MD | pronoun, personal, nominative, 3rd person singular + modal auxillary | he'll she'll it'll he'd it'd she'd |
| PPSS | pronoun, personal, nominative, not 3rd person singular | they we I you ye thou you'uns |
| PPSS+BEM | pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tense, 1st person singular | I'm Ahm |
| PPSS+BER | pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tense, 2nd person singular or all persons plural | we're you're they're |
| PPSS+BEZ | pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tense, 3rd person singular | you's |
| PPSS+BEZ* | pronoun, personal, nominative, not 3rd person singular + verb 'to be', present tense, 3rd person singular | 'tain't |
| PPSS+HV | pronoun, personal, nominative, not 3rd person singular + verb 'to have', uninflected present tense | I've we've they've you've |
| PPSS+HVD | pronoun, personal, nominative, not 3rd person singular + verb 'to have', past tense | I'd you'd we'd they'd |
| PPSS+MD | pronoun, personal, nominative, not 3rd person singular + modal auxillary | you'll we'll I'll we'd I'd they'll they'd you'd |
| PPSS+VB | pronoun, personal, nominative, not 3rd person singular + verb 'to verb', uninflected present tense | y'know |
| QL | qualifier, pre | well less very most so real as highly fundamentally even how much remarkably somewhat more completely too thus ill deeply little overly halfway almost impossibly far severly such ... indeed enough still 'nuff only often generally also nevertheless upon together back newly no likely meanwhile near then heavily there apparently yet outright fully aside consistently specifically formally ever just ... else's here's there's |
| QLP | qualifier, post | |
| RB | adverb | |
| RB\$ | adverb, genitive | |
| RB+BEZ | adverb + verb 'to be', present tense, 3rd person singular | |
| RB+CS | adverb + conjunction, coordinating | |
| RBR | adverb, comparative | |
| RBR+CS | adverb, comparative + conjunction, coordinating | well's soon's further earlier better later higher tougher more harder longer sooner less faster easier louder farther oftener nearer cheaper slower tighter lower worse heavier quicker ... more'n |
| RBT | adverb, superlative | most best highest uppermost nearest brightest hardest fastest deepest farthest loudest ... |
| RN | adverb, nominal | here afar then |
| RP | adverb, particle | up out off down over on in about through across after out'n outta |
| RP+IN | adverb, particle + preposition | to t' |
| TO | infinitival to | t'jawn t'lah |
| TO+VB | infinitival to + verb, infinitive | |

| Brown Tag | Description | Examples |
|------------|---|--|
| UH | interjection | Hurrah bang whee hmpf ah goodbye oops oh-the-pain-of-it ha crunch say oh why see well hello lo alas tarantara rum-tum-tum gosh hell keerist Jesus Keeerist boy c'mon 'mon goddamn bah hoo-pig damn ... |
| VB | verb, base: uninflected present, imperative or infinitive | investigate find act follow inure achieve reduce take remedy re-set distribute realize disable feel receive continue place protect eliminate elaborate work permit run enter force ... wanna |
| VB+AT | verb, base: uninflected present or infinitive + article | lookit |
| VB+IN | verb, base: uninflected present, imperative or infinitive + preposition | die-dead |
| VB+JJ | verb, base: uninflected present, imperative or infinitive + adjective | let's lemme gimme |
| VB+PPO | verb, uninflected present tense + pronoun, personal, accusative | g'ahn c'mon |
| VB+RP | verb, imperative + adverbial particle | wanta wanna |
| VB+TO | verb, base: uninflected present, imperative or infinitive + infinitival to | say-speak |
| VB+VB | verb, base: uninflected present, imperative or infinitive; hyphenated pair | said produced took recommended commented urged found added praised charged listed became announced brought attended wanted voted defeated received got stood shot scheduled feared promised made ... |
| VBD | verb, past tense | modernizing improving purchasing Purchasing lacking enabling pricing keeping getting picking entering voting warning making strengthening setting neighboring attending participating moving ... |
| VBG | verb, present participle or gerund | gonna |
| VBG+TO | verb, present participle + infinitival to | conducted charged won received studied revised operated accepted combined experienced recommended effected granted seen protected adopted retarded notarized selected composed gotten printed ... |
| VBN | verb, past participle | gotta |
| VBN+TO | verb, past participle + infinitival to | deserves believes receives takes goes expires says opposes starts permits expects thinks faces votes teaches holds calls fears spends collects backs eliminates sets flies gives seeks reads ... |
| VBZ | verb, present tense, 3rd person singular | which what whatever whichever whichever-the-hell what're |
| WDT | WH-determiner | whaddya |
| WDT+BER | WH-determiner + verb 'to be', present tense, 2nd person singular or all persons plural | what's |
| WDT+BER+PP | WH-determiner + verb 'to be', present, 2nd person singular or all persons plural + pronoun, personal, nominative, not 3rd person singular | whaddya |
| WDT+BEZ | WH-determiner + verb 'to be', present tense, 3rd person singular | what'd |
| WDT+DO+PPS | WH-determiner + verb 'to do', uninflected present tense + pronoun, personal, nominative, not 3rd person singular | what's |
| WDT+DOD | WH-determiner + verb 'to do', past tense | whose whoever |
| WDT+HVZ | WH-determiner + verb 'to have', present tense, 3rd person singular | whom that who |
| WP\$ | WH-pronoun, genitive | that who whoever whosoever what whatsoever |
| WPO | WH-pronoun, accusative | |
| WPS | WH-pronoun, nominative | |

| Brown Tag | Description | Examples |
|-----------|--|--|
| WPS+BEZ | WH-pronoun, nominative + verb 'to be', present, 3rd person singular | that's who's |
| WPS+HVD | WH-pronoun, nominative + verb 'to have', past tense | who'd |
| WPS+HVZ | WH-pronoun, nominative + verb 'to have', present tense, 3rd person singular | who's that's |
| WPS+MD | WH-pronoun, nominative + modal auxillary | who'll that'd who'd that'll |
| WQL | WH-qualifier | however how |
| WRB | WH-adverb | however when where why whereby wherever how whenever whereon wherein wherewith wheare wherefore whereof howsabout where're |
| WRB+BER | WH-adverb + verb 'to be', present, 2nd person singular or all persons plural | how's where's |
| WRB+BEZ | WH-adverb + verb 'to be', present, 3rd person singular | howda |
| WRB+DO | WH-adverb + verb 'to do', present, not 3rd person singular | where'd how'd |
| WRB+DOD | WH-adverb + verb 'to do', past tense | whyn't |
| WRB+DOD* | WH-adverb + verb 'to do', past tense, negated | how's |
| WRB+DOZ | WH-adverb + verb 'to do', present tense, 3rd person singular | why'n |
| WRB+IN | WH-adverb + preposition | where'd |
| WRB+MD | WH-adverb + modal auxillary | |

A.2 CLAWS5 Tagset

Based on <http://ucrel.lancs.ac.uk/claws5tags.html>

Table A.2

| CLAWS5 Tag | Description | Examples |
|------------|---|--------------------------------|
| AJ0 | adjective (unmarked) | GOOD, OLD |
| AJC | comparative adjective | BETTER, OLDER |
| AJS | superlative adjective | BEST, OLDEST |
| AT0 | article | THE, A, AN |
| AV0 | adverb (unmarked) | OFTEN, WELL, LONGER, FURTHEST |
| AVP | adverb particle | UP, OFF, OUT |
| AVQ | wh-adverb | WHEN, HOW, WHY |
| CJC | coordinating conjunction | AND, OR |
| CJS | subordinating conjunction | ALTHOUGH, WHEN |
| CJT | the conjunction THAT | THAT |
| CRD | cardinal numeral | 3, FIFTY-FIVE, 6609 (excl ONE) |
| DPS | possessive determiner form | YOUR, THEIR |
| DT0 | general determiner | THESE, SOME |
| DTQ | wh-determiner | WHOSE, WHICH |
| EX0 | existential THERE | THERE |
| ITJ | interjection or other isolate | OH, YES, MHM |
| NN0 | noun (neutral for number) | AIRCRAFT, DATA |
| NN1 | singular noun | PENCIL, GOOSE |
| NN2 | plural noun | PENCILS, GEESE |
| NP0 | proper noun | LONDON, MICHAEL, MARS |
| NULL | the null tag (for items not to be tagged) | SIXTH, 77TH, LAST |
| ORD | ordinal | NONE, EVERYTHING |
| PNI | indefinite pronoun | YOU, THEM, OURS |
| PNP | personal pronoun | WHO, WHOEVER |
| PNQ | wh-pronoun | |

| CLAWS5 Tag | Description | Examples |
|------------|--|-----------------------|
| PNX | reflexive pronoun | ITSELF, OURSELVES |
| POS | the possessive (or genitive morpheme) 'S or ' | |
| PRF | the preposition OF | OF |
| PRP | preposition (except for OF) | FOR, ABOVE, TO |
| PUL | punctuation - left bracket | (or [) |
| PUN | punctuation - general mark | . ! , : ; - ? ... |
| PUQ | punctuation - quotation mark | `` '' |
| PUR | punctuation - right bracket |) or] |
| TO0 | infinitive marker TO | TO |
| UNC | "unclassified" items which are not words of the English lexicon | |
| VBB | the "base forms" of the verb "BE" (except the infinitive) | AM, ARE |
| VBD | past form of the verb "BE" | WAS, WERE |
| VBG | -ing form of the verb "BE" | BEING |
| VBI | infinitive of the verb "BE" | BE |
| VBN | past participle of the verb "BE" | BEEN |
| VBZ | -s form of the verb "BE" | IS, 'S |
| VDB | base form of the verb "DO" (except the infinitive) | DO |
| VDD | past form of the verb "DO" | DID |
| VDG | -ing form of the verb "DO" | DOING |
| VDI | infinitive of the verb "DO" | DO |
| VDN | past participle of the verb "DO" | DONE |
| VDZ | -s form of the verb "DO" | DOES |
| VHB | base form of the verb "HAVE" (except the infinitive) | HAVE |
| VHD | past tense form of the verb "HAVE" | HAD, 'D |
| VHG | -ing form of the verb "HAVE" | HAVING |
| VHI | infinitive of the verb "HAVE" | HAVE |
| VHN | past participle of the verb "HAVE" | HAD |
| VHZ | -s form of the verb "HAVE" | HAS, 'S |
| VM0 | modal auxiliary verb | CAN, COULD, WILL, 'LL |
| VVB | base form of lexical verb (except the infinitive) | TAKE, LIVE |
| VVD | past tense form of lexical verb | TOOK, LIVED |
| VVG | -ing form of lexical verb | TAKING, LIVING |
| VVI | infinitive of lexical verb | TAKE, LIVE |
| VVN | past participle form of lex. verb | TAKEN, LIVED |
| VVZ | -s form of lexical verb | TAKES, LIVES |
| XX0 | the negative NOT or N'T | NOT |
| ZZ0 | alphabetical symbol | A, B, c, d |

A.3 UPenn Tagset

Based on <http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html>

Table A.3

| UPenn Tag | Description | Examples |
|-----------|------------------------|---|
| \$ | dollar | \$ -\$ --\$ A\$ C\$ HK\$ M\$ NZ\$ S\$ U.S.\$ US\$ |
| `` | opening quotation mark | `` `` |
| " | closing quotation mark | ' '' |
| (| opening parenthesis | ([{ |
|) | closing parenthesis |)] } |
| , | comma | , |
| -- | dash | -- |

| UPenn Tag | Description | Examples |
|-----------|---|---|
| . | sentence terminator | . ! ? |
| : | colon or ellipsis | : ; ... |
| CC | conjunction, coordinating | & 'n and both but either et for less minus neither nor or plus so therefore times v. versus vs. whether yet |
| CD | numeral, cardinal | mid-1890 nine-thirty forty-two one-tenth ten million 0.5 one forty-seven 1987 twenty '79 zero two 78-degrees eighty-four IX '60s .025 fifteen 271,124 dozen quintillion DM2,000 ... |
| DT | determiner | all an another any both del each either every half la many much nary neither no some such that the them these this those there |
| EX | existential there | gemeinschaft hund ich jeux habeas Haementeria Herr K'ang-si |
| FW | foreign word | vous lutihaw alai je jour objets salutaris fille quibusdam pas trop Monte terram fiche oui corporis ... |
| IN | preposition or conjunction, subordinating | astride among upon whether out inside pro despite on by throughout below within for towards near behind atop around if like until below next into if beside ... |
| JJ | adjective or numeral, ordinal | third ill-mannered pre-war regrettable oiled calamitous first separable ectoplasmic battery-powered participatory fourth still-to-be-named multilingual multi-disciplinary ... |
| JJR | adjective, comparative | bleaker braver breezier briefer brighter brisker broader bumper busier calmer cheaper choosier cleaner clearer closer colder commoner costlier cozier creamier crunchier cuter ... |
| JJS | adjective, superlative | calmest cheapest choicest classiest cleanest clearest closest commonest corniest costliest crassest creepiest crudest cutest darkest deadliest dearest deepest densest dinkiest ... |
| LS | list item marker | A A. B B. C C. D E F First G H I J K One SP-44001 SP-44002 SP-44005 SP-44007 Second Third Three Two * a b c d first five four one six three two |
| MD | modal auxiliary | can cannot could couldn't dare may might must need ought shall should shouldn't will would |
| NN | noun, common, singular or mass | common-carrier cabbage knuckle-duster Casino afghan shed thermostat investment slide humour falloff slick wind hyena override subhumanity machinist ... |
| NNP | noun, proper, singular | Motown Venneboerger Czestochwa Ranzer Conchita Trumplane Christos Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI Darryl CTCA Shannon A.K.C. Meltex Liverpool ... |
| NNPS | noun, proper, plural | Americans Americas Amharas Amityvilles Amusements Anarcho-Syndicalists Andalusians Andes Andruses Angels Animals Anthony Antilles Antiques Apache Apaches Apocrypha ... |
| NNS | noun, common, plural | undergraduates scotches bric-a-brac products bodyguards facets coasts divestitures storehouses designs clubs fragrances averages subjectivists apprehensions muses factory-jobs ... |
| PDT | pre-determiner | all both half many quite such sure this |
| POS | genitive marker | ' s |
| PRP | pronoun, personal | hers herself him himself hisself it itself me myself one oneself ours ourselves ownself self she thee theirs them themselves they thou thy us |
| PRP\$ | pronoun, possessive | her his mine my our ours their thy your |
| RB | adverb | occasionally unabatantly maddeningly adventurously professedly stirringly prominently technologically magisterially predominately swiftly fiscally pitilessly ... |
| RBR | adverb, comparative | further gloomier grander graver greater grimmer harder harsher healthier heavier higher however larger later leaner lengthier less-perfectly lesser lonelier longer louder lower more ... |

| UPenn Tag | Description | Examples |
|-----------|--|--|
| RBS | adverb, superlative | best biggest bluntest earliest farthest first furthest hardest heartiest highest largest least less most nearest second tightest worst |
| RP | particle | aboard about across along apart around aside at away back before behind by crop down ever fast for forth from go high i.e. in into just later low more off on open out over per pie raising start teeth that through under unto up up-pp upon whole with you % & ' " .) . * + , . < = > @ A[fj] U.S U.S.S.R * *** |
| SYM | symbol | to Goodbye Goody Gosh Wow Jeepers Jee-sus Hubba Hey Kee-reist Oops amen huh howdy uh dammit whammo shucks heck anyways whodunnit honey golly man baby diddle hush sonuvabitch ... |
| TO | "to" as preposition or infinitive marker | |
| UH | interjection | |
| VB | verb, base form | ask assemble assess assign assume atone attention avoid bake balkanize bank begin behold believe bend benefit bevel beware bless boil bomb boost brace break bring broil brush build ... dipped pleaded swiped regummed soaked tidied convened halted registered cushioned exacted snubbed strode aimed adopted belied figgered speculated wore appreciated contemplated ... |
| VBD | verb, past tense | telegraphing stirring focusing angering judging stalling lactating hankerin' alleging veering capping approaching traveling besieging encrypting interrupting erasing wincing ... multihulled dilapidated aerosolized chaired languished panelized used experimented flourished imitated reunified factored condensed sheared unsettled primed dubbed desired ... predominate wrap resort sue twist spill cure lengthen brush terminante appear tend stray glisten obtain comprise detest tease attract emphasize mold postpone sever return wag ... bases reconstructs marks mixes displeases seals carps weaves snatches slumps stretches authorizes smolders pictures emerges stockpiles seduces fizzes uses bolsters slaps speaks pleads ... |
| VBG | verb, present participle or gerund | that what whatever which whichever that what whatever whatsoever which who whom whosoever whose |
| VBN | verb, past participle | |
| VBP | verb, present tense, not 3rd person singular | how however whence whenever where whereby whereever wherein whereof why |
| VBZ | verb, present tense, 3rd person singular | |
| WDT | WH-determiner | |
| WP | WH-pronoun | |
| WP\$ | WH-pronoun, possessive | |
| WRB | Wh-adverb | |

B Appendix: 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 "њ" 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.

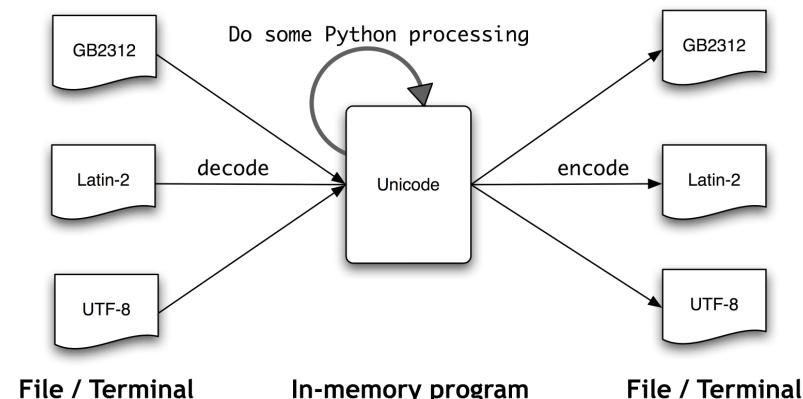
B.1 What is Unicode?

Unicode supports over a million characters. Each of these characters is assigned a number, called a **code point**. In Python, code points are written in the form \uXXXX, where XXXX is the number in 4-digit hexadecimal form.

Within a program, Unicode code points can be manipulated directly, but when Unicode characters are stored in files or displayed on a terminal they must be encoded as one or more bytes. Some encodings (such as ASCII and Latin-2) use a single byte, so they can only support a small subset of Unicode, suited to a single language. Other encodings (such as UTF-8) use

multiple bytes and can represent the full range of Unicode.

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**. The following diagram illustrates.



From a Unicode perspective, characters are abstract entities which 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.

B.2 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), encoded as Latin-2, also known as ISO-8859-2. The function `nltk.data.find()` locates the file for us.

```
>>> import nltk.data
>>> path = nltk.data.find('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>.

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 `\uXXXX` representations. Code points above the ASCII 0-127 range but below 256 are represented in the two-digit form `\xXX`.

```
>>> lines = f.readlines()
>>> for l in lines:
...     l = l[:-1]
...     uni = l.encode('unicode_escape')
...     print uni
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 \u0151al\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 above illustrates a Unicode escape string, namely preceded by the `\u` escape string, namely `\u0144`. The relevant Unicode character will be displayed on the screen as the glyph ñ. 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 `\uXXXX` escape sequence inside a Unicode string literal. We find the integer ordinal of a character using `ord()`. For example:

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

The hexadecimal 4 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, `ñ` 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 `\xXX`) 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 `ñ` on your screen.

Note

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
>>> line = lines[2]
>>> print line.encode('unicode_escape')
Niemc\xf3w pod koniec II wojny \u015bwiatowej na Dolny \u0151al\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 code sample above, and if your system supports UTF-8, you should see an output like the following:

ó U+00f3 LATIN SMALL LETTER O WITH ACUTE
 ó U+015b LATIN SMALL LETTER S WITH ACUTE
 ź U+015a LATIN CAPITAL LETTER S WITH ACUTE
 á 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'
```

The NLTK `tokenizer` module allows Unicode strings as input, and correspondingly yields Unicode strings as output.

```
>>> from nltk.tokenize import WordTokenizer
>>> tokenizer = WordTokenizer()
>>> tokenizer.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']
```

B.3 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: <encoding> -*`' as the first or second line of your file. Note that `<encoding>` has to be a string like '`latin-1`', '`big5`' or '`utf-8`'.

Note

If you are using Emacs as your editor, the coding specification will also be interpreted as a specification of the editor's coding for the file. Not all of the valid Python names for codings are accepted by Emacs.

The following screenshot illustrates the use of UTF-8 encoded string literals within the IDLE editor:

```
# -*- coding: utf-8 -*-

import re
sent = """
Przewiezione przez Niemców pod koniec II wojny światowej na Dolny
Śląsk, zostały odnalezione po 1945 r. na terytorium Polski.
"""

u = sent.decode('utf8')
u.lower()
print u.encode('utf8')

SACUTE = re.compile('ś|š')
replaced = re.sub(SACUTE, '[acute]', sent)
print replaced
```

Note

The above example requires that an appropriate font is set in IDLE's preferences. In this case, we chose Courier CE.

The above example also illustrates how regular expressions can use encoded strings.

Further Reading

There are a number of online discussions of Unicode in general, and of Python facilities for handling Unicode. The following are worth consulting:

- 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>

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

C Appendix: NLP in Python vs other Programming Languages

Many programming languages have been used for NLP. As explained in the Preface, we have chosen Python because we believe it is well-suited to the special requirements of NLP. Here we present a brief survey of several programming languages, for the simple task of reading a text and printing the words that end with `ing`. We begin with the Python version, which we believe is readily interpretable, even by non Python programmers:

```
import sys
for line in sys.stdin:
    for word in line.split():
        if word.endswith('ing'):
```

```
print word
```

Like Python, Perl is a scripting language. However, its syntax is obscure. For instance, it is difficult to guess what kind of entities are represented by: <>, \$, my, and split, in the following program:

```
while (<>) {
    foreach my $word (split) {
        if ($word =~ /ing$/) {
            print "$word\n";
        }
    }
}
```

We agree that "it is quite easy in Perl to write programs that simply look like raving gibberish, even to experienced Perl programmers" (Hammond 2003:47). Having used Perl ourselves in research and teaching since the 1980s, we have found that Perl programs of any size are inordinately difficult to maintain and re-use. Therefore we believe Perl is no longer a particularly suitable choice of programming language for linguists or for language processing.

Prolog is a logic programming language which has been popular for developing natural language parsers and feature-based grammars, given the inbuilt support for search and the *unification* operation which combines two feature structures into one. Unfortunately Prolog is not easy to use for string processing or input/output, as the following program code demonstrates for our linguistic example:

```
main :-
    current_input(InputStream),
    read_stream_to_codes(InputStream, Codes),
    codesToWords(Codes, Words),
    maplist(string_to_list, Words, Strings),
    filter(endsWithIng, Strings, MatchingStrings),
    writeMany(MatchingStrings),
    halt.

codesToWords([], []).
codesToWords([Head | Tail], Words) :-
    ( char_type(Head, space) ->
        codesToWords(Tail, Words)
    ;
        getWord([Head | Tail], Word, Rest),
        codesToWords(Rest, Words0),
        Words = [Word | Words0]
    ).

getWord([], [], []).
getWord([Head | Tail], Word, Rest) :-
    (
        ( char_type(Head, space) ; char_type(Head, punct) )
    -> Word = [], Tail = Rest
    ;
        getWord(Tail, Word0, Rest),
        Word = [Head | Word0]
    ).

filter(Predicate, List0, List) :-
    ( List0 = [] -> List = []
    ;
        List0 = [Head | Tail],
        ( apply(Predicate, [Head]) ->
            filter(Predicate, Tail, List1),
            List = [Head | List1]
        ;
            filter(Predicate, Tail, List)
        )
    ).

endsWithIng(String) :- sub_string(String, _Start, _Len, 0, 'ing').

writeMany([]).
writeMany([Head | Tail]) :- write(Head), nl, writeMany(Tail).
```

Java is an object-oriented language incorporating native support for the Internet, that was originally designed to permit the same executable program to be run on most computer platforms. Java has replaced COBOL as the standard language for business enterprise software:

```
import java.io.*;
public class IngWords {
```

```
public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new
        InputStreamReader(
            System.in)));
    String line = in.readLine();
    while (line != null) {
        for (String word : line.split(" ")) {
            if (word.endsWith("ing"))
                System.out.println(word);
        }
        line = in.readLine();
    }
}
```

The C programming language is a highly-efficient low-level language that is popular for operating system and networking software:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    int i = 0;
    int c = 1;
    char buffer[1024];

    while (c != EOF) {
        c = fgetc(stdin);
        if ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
            buffer[i++] = (char) c;
            continue;
        } else {
            if (i > 2 && (strncmp(buffer+i-3, "ing", 3) == 0 || strncmp(buffer+i-3, "ING", 3) == 0)) {
                buffer[i] = 0;
                puts(buffer);
            }
            i = 0;
        }
    }
    return 0;
}
```

LISP is a so-called functional programming language, in which all objects are lists, and all operations are performed by (nested) functions of the form `(function arg1 arg2 ...)`. Many of the earliest NLP systems were implemented in LISP:

```

(defpackage "REGEXP-TEST" (:use "LISP" "REGEXP"))
(in-package "REGEXP-TEST")

(defun has-suffix (string suffix)
  "Open a file and look for words ending in _ing."
  (with-open-file (f string)
    (with-loop-split (s f " ")
      (mapcar #'(lambda (x) (has_suffix suffix x)) s)))))

(defun has_suffix (suffix string)
  (let* ((suffix_len (length suffix))
         (string_len (length string))
         (base_len (- string_len suffix_len)))
    (if (string-equal suffix string :start1 0 :end1 NIL :start2 base_len :end2 NIL
                      (print string)))))

(has-suffix "test.txt" "ing")

```

Ruby is a more recently developed scripting language than Python, best known for its convenient web application framework, *Ruby on Rails*. Here are two Ruby programs for finding words ending in *ing*.

```
ARGF.each { |line|
  line.split.find_all { |word|
    word.match(/ing$/)
  }.each { |word|
    puts word
  }
}
```

```

for line in ARGF
  for word in line.split
    if word.match(/ing$/) then
      puts word
    end
  end
end

```

Haskell is another functional programming language which permits a much more compact (but incomprehensible) solution of our simple task:

```

import Data.List
main = putStrLn . unlines . filter ("ing" `isSuffixOf`) . words =<< getContent

```

The unix shell can also be used for simple linguistic processing. Here is a simple pipeline for finding the *ing* words. The first step transliterates any whitespace character to a newline, so that each word of the text occurs on its own line, and the second step finds all lines ending in *ing*

```
tr [:space:] '\n' | grep ing$
```

(We are grateful to the following people for furnishing us with these program samples: Tim Baldwin, Trevor Cohn, David Duke, Rod Farmer, Andrew Hardie, Aaron Harnly, Edward Ivanovic, and Lars Yencken.)

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

D Appendix: NLTK Modules and Corpora

NLTK Organization: NLTK is organized into a collection of task-specific packages. Each package is a combination of data structures for representing a particular kind of information such as trees, and implementations of standard algorithms involving those structures such as parsers. This approach is a standard feature of *object-oriented design*, in which components encapsulate both the resources and methods needed to accomplish a particular task.

The most fundamental NLTK components are for identifying and manipulating individual words of text. These include: `tokenize`, for breaking up strings of characters into word tokens; `tag`, for adding part-of-speech tags, including regular-expression taggers, n-gram taggers and Brill taggers; and the Porter stemmer.

The second kind of module is for creating and manipulating structured linguistic information. These components include: `tree`, for representing and processing parse trees; `featurestructure`, for building and unifying nested feature structures (or attribute-value matrices); `cfd`, for specifying context-free grammars; and `parse`, for creating parse trees over input text, including chart parsers, chunk parsers and probabilistic parsers.

Several utility components are provided to facilitate processing and visualization. These include: `draw`, to visualize NLP structures and processes; `probability`, to count and collate events, and perform statistical estimation; and `corpora`, to access tagged linguistic corpora.

A further group of components is not part of NLTK proper. These are a wide selection of third-party contributions, often developed as student projects at various institutions where NLTK is used, and distributed in a separate package called *NLTK Contrib*. Several of these student contributions, such as the Brill tagger and the HMM module, have now been incorporated into NLTK. Although these contributed components are not maintained, they may serve as a useful starting point for future student projects.

In addition to software and documentation, NLTK provides substantial corpus samples. Many of these can be accessed using the `corpora` module, avoiding the need to write specialized file parsing code before you can do NLP tasks. These corpora include:

Brown Corpus — 1.15 million words of tagged text in 15 genres; a 10% sample of the Penn Treebank corpus, consisting of 40,000 words of syntactically parsed text; a selection of books from Project Gutenberg totally 1.7 million words; and other corpora for chunking, prepositional phrase attachment, word-sense disambiguation, text categorization, and information extraction.

Table D.1

| Corpus | Corpora and Corpus Samples Distributed with NLTK Compiler | Contents |
|-----------------------------|--|---|
| Alpino Dutch Treebank | van Noord | 140k words, tagged and parsed (Dutch) |
| Australian ABC News | Bird | 2 genres, 660k words, sentence-segmented |
| Brown Corpus | Francis, Kucera | 15 genres, 1.15M words, tagged, categorized |
| CESS-CAT Catalan Treebank | CLiC-UB et al | 500k words, tagged and parsed |
| CESS-ESP Spanish Treebank | CLiC-UB et al | 500k words, tagged and parsed |
| CMU Pronouncing Dictionary | CMU | 127k entries |
| CoNLL 2000 Chunking Data | Tjong Kim Sang | 270k words, tagged and chunked |
| CoNLL 2002 Named Entity | Tjong Kim Sang | 700k words, pos- and named-entity-tagged (Dutch, Spanish) |
| Floresta Treebank | Diana Santos et al | 9k sentences (Portuguese) |
| Genesis Corpus | Misc web sources | 6 texts, 200k words, 6 languages |
| Gutenberg (sel) | Hart, Newby, et al | 14 texts, 1.7M words |
| 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 | Sentiment Polarity Dataset 2.0 |
| Names Corpus | Kantrowitz, Ross | 8k male and female names |
| NIST 1999 Info Extr (sel) | Garofolo | 63k words, newswire and named-entity SGML markup |
| NPS Chat Corpus | Forsyth, Martell | 10k IM chat posts, POS-tagged and dialogue-act tagged |
| PP Attachment Corpus | Ratnaparkhi | 28k prepositional phrases, tagged as noun or verb modifiers |
| Presidential Addresses | Ahrens | 485k words, formatted text |
| Proposition Bank | Palmer | 113k propositions, 3300 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, part-of-speech and sense tagged |
| SENSEVAL 2 Corpus | Ted Pedersen | 600k words, part-of-speech and sense tagged |
| Shakespeare XML texts (sel) | Jon Bosak | 8 books |
| Stopwords Corpus | Porter et al | 2,400 stopwords for 11 languages |
| Switchboard Corpus (sel) | LDC | 36 phonecalls, transcribed, parsed |
| Univ Decl of Human Rights | United Nations | 480k words, 300+ languages |
| US Pres Addr Corpus | Ahrens | 480k words |
| Penn Treebank (sel) | LDC | 40k words, tagged and parsed |
| TIMIT Corpus (sel) | NIST/LDC | audio files and transcripts for 16 speakers |
| 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 |

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

E Appendix: Python and NLTK Cheat Sheet (Draft)

E.1 Python

Strings

```
>>> x = 'Python'; y = 'NLTK'; z = 'Natural Language Processing'
>>> x + '/' + y
'Python/NLTK'
>>> 'LT' in y
True
>>> x[2:]
'thon'
>>> x[::-1]
'nohtyP'
>>> len(x)
6
>>> z.count('a')
4
>>> z.endswith('ing')
True
>>> z.index('Language')
8
>>> ' '.join([x,y,z])
'Python; NLTK; Natural Language Processing'
>>> y.lower()
'nltk'
>>> z.replace(' ', '\n')
'Natural\nLanguage\nProcessing'
>>> print z.replace(' ', '\n')
Natural
Language
Processing
>>> z.split()
['Natural', 'Language', 'Processing']
```

For more information, type `help(str)` at the Python prompt.

Lists

```
>>> x = ['Natural', 'Language']; y = ['Processing']
>>> x[0]
'Natural'
>>> list(x[0])
['N', 'a', 't', 'u', 'r', 'a', 'l']
>>> x + y
['Natural', 'Language', 'Processing']
>>> 'Language' in x
True
>>> len(x)
2
>>> x.index('Language')
1
```

The following functions modify the list in-place:

```
>>> x.append('Toolkit')
>>> x
['Natural', 'Language', 'Toolkit']
>>> x.insert(0, 'Python')
>>> x
['Python', 'Natural', 'Language', 'Toolkit']
>>> x.reverse()
>>> x
['Toolkit', 'Language', 'Natural', 'Python']
>>> x.sort()
>>> x
['Language', 'Natural', 'Python', 'Toolkit']
```

For more information, type `help(list)` at the Python prompt.

Dictionaries

```
>>> d = {'natural': 'adj', 'language': 'noun'}
>>> d['natural']
'adj'
>>> d['toolkit'] = 'noun'
>>> d
{'natural': 'adj', 'toolkit': 'noun', 'language': 'noun'}
>>> 'language' in d
True
>>> d.items()
[('natural', 'adj'), ('toolkit', 'noun'), ('language', 'noun')]
>>> d.keys()
['natural', 'toolkit', 'language']
>>> d.values()
['adj', 'noun', 'noun']
```

For more information, type `help(dict)` at the Python prompt.

Regular Expressions

Note

to be written

E.2 NLTK

Many more examples can be found in the NLTK Guides, available at <http://nltk.org/doc/guides>.

Corpora

```
>>> import nltk
>>> dir(nltk.corpus)
```

Tokenization

```
>>> text = '''NLTK, the Natural Language Toolkit, is a suite of program
... modules, data sets and tutorials supporting research and teaching in
... computational linguistics and natural language processing.'''
>>> import nltk
>>> nltk.LineTokenizer().tokenize(text)
['NLTK, the Natural Language Toolkit, is a suite of program', 'modules,
data sets and tutorials supporting research and teaching in', 'computational
linguistics and natural language processing.']
>>> nltk.WhitespaceTokenizer().tokenize(text)
['NLTK', 'the', 'Natural', 'Language', 'Toolkit', 'is', 'a', 'suite',
'of', 'program', 'modules', 'data', 'sets', 'and', 'tutorials',
'supporting', 'research', 'and', 'teaching', 'in', 'computational',
'linguistics', 'and', 'natural', 'language', 'processing.']
>>> nltk.WordPunctTokenizer().tokenize(text)
['NLTK', ',', 'the', 'Natural', 'Language', 'Toolkit', ',', 'is', 'a',
'suite', 'of', 'program', 'modules', ',', 'data', 'sets', 'and',
'tutorials', 'supporting', 'research', 'and', 'teaching', 'in',
'computational', 'linguistics', 'and', 'natural', 'language',
'processing', '.']
>>> nltk.RegexpTokenizer(' ', gaps=True).tokenize(text)
['NLTK', 'the Natural Language Toolkit', 'is a suite of program\nmodules',
'data sets and tutorials supporting research and teaching in\ncomputational
linguistics and natural language processing.']
```

Stemming

```
>>> tokens = nltk.WordPunctTokenizer().tokenize(text)
>>> stemmer = nltk.RegexpStemmer('ing$|s$|e$')
>>> for token in tokens:
...     print stemmer.stem(token),
NLTK , th Natural Languag Toolkit , i a suit of program module ,
data set and tutorial support research and teach in computational
linguistic and natural languag process .
>>> stemmer = nltk.PorterStemmer()
>>> for token in tokens:
...     print stemmer.stem(token),
NLTK , the Natur Languag Toolkit , is a suit of program modul ,
data set and tutori support research and teach in comput linguist
and natur languag process .
```

Tagging

Note

to be written

About this document...

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008

Index

—
[\(phrasal\) projections \(10.4.2\)](#)

A

[A* Parser \(9.3.2\)](#)
[A* parser \(9.3.3\)](#)
[accuracy \(5.6.2\)](#)
[active chart \(8.5.1\)](#)
[address \(11.4.4\)](#)
[adjectives \(4.2.5\)](#)
[adverbs \(4.2.5\)](#)
[agreement \(10.2.1\)](#)
[alphabetic variants \(11.3.2\)](#)
[American National Corpus \(12.2\)](#)
[anaphora resolution \(1.5.2\)](#)
[anaphoric antecedent \(11.6.1\)](#)
[annotation \(5.3.2\)](#)
[antecedent \(1.5.2\)](#)
[appropriate \(10.6\)](#)
[arity \(11.3.3\)](#)
[articles \(4.2.5\)](#)
[artificial intelligence \(I.1\)](#)
[assignment \(1.2.3\)](#)
[Assignment \(11.3.5\)](#)
[atomic \(10.2.3\)](#)

[attribute value matrix \(10.2.3\)](#)

[auxiliaries \(10.4.3\)](#)

[auxiliary \(10.2.3\)](#)

B

[backtracks \(8.4.1\)](#)

[balanced corpora \(5.3.1\)](#)

[basic types \(11.3.1\)](#)

[Bayesian Network Graph \(5.8.2\)](#)

[beam search \(9.3.4\)](#)

[best-first search strategy \(9.3.4\)](#)

[beta-conversion \(11.3.1\)](#)

[bigrams \(1.3.3\)](#)

[binary predicate \(11.3.1\)](#)

[bind \(11.3.1\)](#)

[binding operators \(11.4.4\)](#)

[binning \(5.8.4\)](#)

[BIO Format \(7.6\)](#)

[Boolean context \(6.8\)](#)

[boolean \(10.2.3\)](#)

[boolean operators \(11.2\)](#)

[bottom-up \(6.4.4\)](#)

[Bottom-Up Initialization Rule \(8.5.6\)](#)

[Bottom-Up Initialization Rule \(8.5.6\)](#)

[bottom-up parsing \(8.4.1\)](#)

[Bottom-Up Predict Rule \(8.5.6\)](#)

[Bottom-Up Predict Rule \(8.5.6\)](#)

[Bottom-Up Strategy \(8.5.6\)](#)

[bound \(11.3.1\)](#)

[British National Corpus \(12.2\)](#)

[business information analysis \(1.1\)](#)

C

[call-by-value \(6.2.2\)](#)

[call structure \(6.4.4\)](#)

[Catalan numbers \(9.1\)](#)

[characteristic function \(11.3.4\)](#)

[chart parsing \(8.5\)](#)

[child \(6.4.3\)](#)

[CHILDES \(12.2\)](#)

[chink \(7.2.7\)](#)

[chink \(7.6\)](#)

[chinking \(7.2.7\)](#)

[chunker \(7.2.3\)](#)

[chunking \(7.2\)](#)

[chunks \(7.2\)](#)

[class label \(5.5.2\)](#)

[classes \(6.6\)](#)

[classification \(5.5.2\)](#)

[closed class \(4.8.4\)](#)

[closed \(11.3.1\)](#)

[closures \(3.3.2\)](#)

[coindex \(10.3.2\)](#)

[comparative wordlist \(2.4.3\)](#)

[complements \(8.3.3\)](#)

[complete edge \(8.5.3\)](#)

[complete \(8.8\)](#)

[complete \(9.3.4\)](#)

[complex \(10.2.3\)](#)

[complex types \(11.3.1\)](#)

[components \(1.5.4\)](#)

[computer science \(1.1\)](#)
[concatenation \(1.2.1\)](#)
[concatenation \(3.2.1\)](#)
[conditional expression \(1.4.3\)](#)
[conditional frequency distribution \(2.2\)](#)
[conditions \(11.6.1\)](#)
[confusion matrix \(4.5.7\)](#)
[consistent \(11.1.2\)](#)
[constituency \(8.2.2\)](#)
[constituents \(8.2.2\)](#)
[construction \(5.2\)](#)
[context-free grammar \(8.3\)](#)
[control \(1.4\)](#)
[control structure \(1.4.3\)](#)
[conversion \(11.3.2\)](#)
[conversion specifier \(3.8.2\)](#)
[Cooper storage \(11.4.4\)](#)
[copy \(6.1.2\)](#)
[coreferential \(11.3.1\)](#)
[corpora \(2\)](#)
[corpora \(5.2\)](#)
[corpus \(2\)](#)
[corpus linguistics \(1.1\)](#)

D

[daughter \(6.4.3\)](#)
[decision nodes \(5.8\)](#)
[decision stump \(5.8\)](#)
[decision tree \(5.8\)](#)
[decorator \(6.4.4\)](#)
[decrease-and-conquer \(6.4\)](#)
[dependent \(9.5\)](#)
[derived corpus \(5.3\)](#)
[determiners \(4.2.5\)](#)
[development \(5.5.3\)](#)
[diagnosis \(7.5\)](#)
[dictionary \(4.3.2\)](#)
[direct recursion \(8.3.2\)](#)
[directed acyclic graphs \(10.3.2\)](#)
[discourse referents \(11.6.1\)](#)
[discourse representation structure \(11.6.1\)](#)
[dispersion plot \(1.1.2\)](#)
[divide-and-conquer \(6.4\)](#)
[docstring \(6.2\)](#)
[domain \(11.3.4\)](#)
[dot \(8.5.2\)](#)
[dotted edges \(8.5.2\)](#)
[Dublin Core \(12.3.6\)](#)
[dynamic programming \(8.5\)](#)
[dynamically typed \(6.2.1\)](#)

E

[edge queue \(9.3.3\)](#)
[edge queue \(9.3.4\)](#)
[encode \(5.5.3\)](#)
[entails \(2.5.3\)](#)
[equivalent \(10.3.2\)](#)
[equivalents \(11.3.2\)](#)
[error analysis \(5.5.3\)](#)
[evaluation set \(5.6.1\)](#)
[event \(1.3.1\)](#)

[existential quantifier \(11.3.1\)](#)
[Exploratory data analysis \(5.2\)](#)
[export \(3.8.3\)](#)

F

[f-structure \(10.6\)](#)
[feature extraction \(5.5.3\)](#)
[feature \(10.2.2\)](#)
[feature path \(10.3.2\)](#)
[feature structure \(10.2.3\)](#)
[features \(5.5.3\)](#)
[fields \(6.1.4\)](#)
[filler \(10.4.4\)](#)
[first-in-first-out \(6.1.5\)](#)
[format string \(3.8.2\)](#)
[formulas \(11.2\)](#)
[free \(11.3.1\)](#)
[frequency distribution \(1.3.1\)](#)
[function body \(6.2\)](#)
[function \(2.3.2\)](#)
[function \(6.2\)](#)
[Fundamental Rule \(8.5.5\)](#)
[Fundamental Rule \(8.5.5\)](#)

G

[gaps \(10.4.4\)](#)
[General Ontology for Linguistic Description \(12.3.5\)](#)
[generator \(6.3.1\)](#)
[gerund \(4.8.1\)](#)
[Glue semantics \(11.4.4\)](#)
[gold standard \(4.4.5\)](#)
[grammar \(8.3\)](#)
[grammatical productions \(8.3.1\)](#)
[guides \(I.5\)](#)

H

[hapaxes \(1.3.1\)](#)
[head features \(10.7\)](#)
[heads \(8.3.3\)](#)
[heights \(8.2.1\)](#)
[hole semantics \(11.4.4\)](#)
[human-computer interaction \(I.1\)](#)
[humanities computing \(I.1\)](#)
[hyponyms \(2.5.2\)](#)

I

[identifiers \(1.2.3\)](#)
[immutable \(3.2.6\)](#)
[immutable \(6.1.4\)](#)
[incomplete edge \(8.5.3\)](#)
[inconsistent \(11.1.2\)](#)
[index \(1.2.2\)](#)
[indirect recursion \(8.3.2\)](#)
[Information Extraction \(7.1\)](#)
[information gain \(5.8\)](#)
[initializer \(6.6.1\)](#)
[Inline annotation \(5.3.2\)](#)
[interpreter \(1.1.1\)](#)
[IOB tags \(7.2.2\)](#)
[iterative optimization \(5.9\)](#)

K

[Kappa \(12.3.7\)](#)

[key \(2.4.2\)](#)
[key \(4.3.2\)](#)
[key-value pairs \(4.3.2\)](#)
[keyword arguments \(6.2.8\)](#)
[Kleene closures \(3.3.2\)](#)

L

[lambda abstraction \(11.3.1\)](#)
[lambda expressions \(6.2.6\)](#)
[lambda operator \(11.3.1\)](#)
[last-in-first-out \(6.1.5\)](#)
[leaf nodes \(5.8\)](#)
[leaves \(8.2.1\)](#)
[leaves \(8.2.1\)](#)
[left-corner \(8.4.3\)](#)
[left-corner parser \(8.4.3\)](#)
[left recursive \(8.3.2\)](#)
[lexical ambiguity \(8.2.1\)](#)
[lexical categories \(4.1.1\)](#)
[lexical productions \(8.3.1\)](#)
[lexicon \(12.2.1\)](#)
[library \(2.3.3\)](#)
[licensed \(10.4.4\)](#)
[licenses \(8.3.1\)](#)
[linguistic category \(10.2.3\)](#)
[list \(1.2.1\)](#)
[logical constants \(11.3.1\)](#)
[logical form \(11.2\)](#)
[long-distance dependency \(10.1\)](#)
[lowest-cost-first search strategy \(9.3.4\)](#)

M

[machine translation \(1.5.3\)](#)
[mapping \(4.3\)](#)
[maximal projection \(10.4.2\)](#)
[Maximum Entropy \(5.9\)](#)
[memoization \(6.4.4\)](#)
[meronyms \(2.5.3\)](#)
[method \(1.5\)](#)
[modals \(4.2.5\)](#)
[model \(11.1.2\)](#)
[models \(5.5\)](#)
[module \(2.3.3\)](#)
[module \(2.3.3\)](#)
[morpho-syntactic \(4.8.5\)](#)
[most likely constituents table \(9.3.2\)](#)
[mutable \(3.2.6\)](#)
[mutable \(6.1.4\)](#)

N

[n-gram tagger \(4.5.3\)](#)
[Naive Bayes \(5.8.1\)](#)
[Named Entity Recognition \(7.1.1\)](#)
[natural language \(I\)](#)
[Natural Language Processing \(I\)](#)
[newlines \(3.1.4\)](#)
[non-logical constants \(11.3.1\)](#)
[non-terminal \(8.3.1\)](#)
[normalized \(3.5\)](#)
[noun phrase \(8.2.2\)](#)

O

[open formula \(11.3.1\)](#)
[out-of-vocabulary \(4.5.5\)](#)
[overfit \(5.8\)](#)
[overfitting \(5.5.3\)](#)

P

[package \(2.3.3\)](#)
[parameters \(2.3.2\)](#)
[parameters \(5.8.6\)](#)
[parent \(6.4.3\)](#)
[parse edge \(8.5.3\)](#)
[parser \(8.4\)](#)
[part-of-speech tagging \(4\)](#)
[partial information \(10.3.3\)](#)
[partial parsing \(7.2.1\)](#)
[parts of speech \(4.1.1\)](#)
[personal pronouns \(4.2.5\)](#)
[phrasal level \(10.4.2\)](#)
[phrase structure \(8.1.2\)](#)
[POS-tagging \(4\)](#)
[pre-sort \(6.4\)](#)
[pre-terminals \(8.3.1\)](#)
[precision/recall trade-off \(4.5.3\)](#)
[predicates \(11.3.1\)](#)
[prepositional phrase attachment ambiguity \(8.2.1\)](#)
[prepositional phrase attachment ambiguity \(8.3.1\)](#)
[Prepositional Phrase Attachment Corpus \(8.2.1\)](#)
[present participle \(4.8.1\)](#)
[Principle of Compositionality \(11.1\)](#)
[prior probability \(5.8.1\)](#)
[probabilistic context free grammar \(9.3.1\)](#)
[productions \(8.1.1\)](#)
[productions \(8.3\)](#)
[Propositional logic \(11.2\)](#)
[propositional symbols \(11.2\)](#)
[prune \(5.8\)](#)

Q

[question answering \(1.5.3\)](#)
[queue \(6.1.5\)](#)

R

[recognizing \(8.5.1\)](#)
[record \(6.1.4\)](#)
[record \(12.2.1\)](#)
[recursion \(8.1.1\)](#)
[recursion \(8.3\)](#)
[recursive \(8.3.2\)](#)
[reduce \(8.4.2\)](#)
[reduce-reduce conflict \(8.4.2\)](#)
[reentrancy \(10.3.2\)](#)
[refactor \(6.2.3\)](#)
[Relation Extraction \(7.1.1\)](#)
[relational operators \(1.4.1\)](#)
[return value \(2.3.2\)](#)
[root \(8.2.1\)](#)
[root node \(5.8\)](#)
[rules \(8.5.4\)](#)
[runtime error \(1.2.2\)](#)

S

[S-Retrieval \(11.4.4\)](#)

[satisfies \(11.3.5\)](#)
[Scanner Rule \(8.5.8\)](#)
[scope \(8.2.1\)](#)
[scope \(11.3.7\)](#)
[self-loop edge \(8.5.3\)](#)
[semantic role labeling \(1.5.2\)](#)
[semantics \(11.1\)](#)
[sequence \(3.2.6\)](#)
[shift \(8.4.2\)](#)
[shift-reduce conflict \(8.4.2\)](#)
[shift-reduce parser \(8.4.2\)](#)
[Shoebox \(12.2\)](#)
[sisters \(6.4.3\)](#)
[slash categories \(10.4.4\)](#)
[slicing \(1.2.2\)](#)
[smoothing \(5.8.3\)](#)
[stack \(6.1.5\)](#)
[standoff annotation \(5.3.2\)](#)
[standoff annotation \(12.2.3\)](#)
[start-symbol \(8.3.1\)](#)
[stopwords \(2.4.1\)](#)
[strategy \(8.5.4\)](#)
[string formatting expressions \(3.8.2\)](#)
[string \(3.2\)](#)
[strings \(1.2.4\)](#)
[structurally ambiguous \(8.3.1\)](#)
[structure sharing \(10.3.2\)](#)
[structured data \(7.1\)](#)
[stylistics \(2.1.3\)](#)
[subcategories \(8.3.3\)](#)
[Subject-Auxiliary Inversion \(8.2.2\)](#)
[subsumes \(10.3.3\)](#)
[subsumption \(10.3.3\)](#)
[subtype \(10.6\)](#)
[supervised \(5.5.2\)](#)
[Swadesh wordlists \(2.4.3\)](#)
[synonyms \(2.5.1\)](#)
[synset \(2.5.1\)](#)
[syntax error \(1.1.1\)](#)

T

[T9 \(3.3.2\)](#)
[tag \(10.3.2\)](#)
[tag pattern \(7.2.4\)](#)
[tagged \(4.2.2\)](#)
[tagging \(4\)](#)
[tagset \(4\)](#)
[terminals \(8.3.1\)](#)
[terms \(11.3.1\)](#)
[text \(12.2.1\)](#)
[texonyms \(3.3.2\)](#)
[tokenization \(3.1.1\)](#)
[tokens \(3.1.1\)](#)
[Toolbox \(12.2\)](#)
[Top-Down Expand Rule \(8.5.7\)](#)
[Top-Down Expand Rule \(8.5.7\)](#)
[top-down \(6.4.4\)](#)
[Top-Down Initialization Rule \(8.5.7\)](#)
[Top-Down Initialization Rule \(8.5.7\)](#)
[Top-Down Match Rule \(8.5.7\)](#)

[Top-Down Match Rule \(8.5.7\)](#)

[top-down parsing \(8.4.1\)](#)

[Top-Down Strategy \(8.5.7\)](#)

[total likelihood \(5.9\)](#)

[train \(4.5.1\)](#)

[training corpus \(5.5.2\)](#)

[training \(4.5.1\)](#)

[transform-and-conquer \(6.4\)](#)

[transitive verbs \(8.3.3\)](#)

[tree diagram \(8.2.1\)](#)

[truth-conditions \(11.2\)](#)

[tuple \(4.2.6\)](#)

[Turing Test \(1.5.4\)](#)

[type-raising \(11.4.2\)](#)

[type raising \(11.4.3\)](#)

[Typed feature structures \(10.6\)](#)

[types \(11.3.1\)](#)

U

[unary predicate \(11.3.1\)](#)

[unbounded dependency construction \(10.4.4\)](#)

[underspecified \(10.2.2\)](#)

[unification \(10.3.3\)](#)

[unify \(10.2.2\)](#)

[unigram chunker \(7.3.5\)](#)

[unique beginners \(2.5.2\)](#)

[universal quantifier \(11.3.1\)](#)

[unseen \(5.5\)](#)

[unstructured data \(7.1\)](#)

V

[valency \(8.3.3\)](#)

[valuation function \(11.3.4\)](#)

[value \(4.3.2\)](#)

[variable \(1.2.3\)](#)

W

[Web software development \(1.1\)](#)

[weights \(5.8.6\)](#)

[well-formed substring table \(8.5.1\)](#)

[wildcard \(3.3.1\)](#)

[word classes \(4.1.1\)](#)

[word sense disambiguation \(1.5.1\)](#)

[WordNet \(2.5\)](#)

Z

[zero projection \(10.4.2\)](#)

Index

-

[\(phrasal\) projections \(10.4.2\)](#)

A

[A* Parser \(9.3.2\)](#)

[A* parser \(9.3.3\)](#)

[accuracy \(5.6.2\)](#)

[active chart \(8.5.1\)](#)

[address \(11.4.4\)](#)

[adjectives \(4.2.5\)](#)

[adverbs \(4.2.5\)](#)

[agreement \(10.2.1\)](#)
[alphabetic variants \(11.3.2\)](#)
[American National Corpus \(12.2\)](#)
[anaphora resolution \(1.5.2\)](#)
[anaphoric antecedent \(11.6.1\)](#)
[annotation \(5.3.2\)](#)
[antecedent \(1.5.2\)](#)
[appropriate \(10.6\)](#)
[arity \(11.3.3\)](#)
[articles \(4.2.5\)](#)
[artificial intelligence \(I.1\)](#)
[Assignment \(11.3.5\)](#)
[Assignment \(11.3.5\)](#)
[atomic \(10.2.3\)](#)
[attribute value matrix \(10.2.3\)](#)
[auxiliaries \(10.4.3\)](#)
[auxiliary \(10.2.3\)](#)

B

[backtracks \(8.4.1\)](#)
[balanced corpora \(5.3.1\)](#)
[basic types \(11.3.1\)](#)
[Bayesian Network Graph \(5.8.2\)](#)
[beam search \(9.3.4\)](#)
[best-first search strategy \(9.3.4\)](#)
[beta-conversion \(11.3.1\)](#)
[bigrams \(1.3.3\)](#)
[binary predicate \(11.3.1\)](#)
[bind \(11.3.1\)](#)
[binding operators \(11.4.4\)](#)
[binning \(5.8.4\)](#)
[BIO Format \(7.6\)](#)
[Boolean context \(6.8\)](#)
[boolean \(10.2.3\)](#)
[boolean operators \(11.2\)](#)
[bottom-up \(6.4.4\)](#)
[Bottom-Up Initialization Rule \(8.5.6\)](#)
[Bottom-Up Initialization Rule \(8.5.6\)](#)
[bottom-up parsing \(8.4.1\)](#)
[Bottom-Up Predict Rule \(8.5.6\)](#)
[Bottom-Up Predict Rule \(8.5.6\)](#)
[Bottom-Up Strategy \(8.5.6\)](#)
[bound \(11.3.1\)](#)
[British National Corpus \(12.2\)](#)
[business information analysis \(I.1\)](#)

C

[call-by-value \(6.2.2\)](#)
[call structure \(6.4.4\)](#)
[Catalan numbers \(9.1\)](#)
[characteristic function \(11.3.4\)](#)
[chart parsing \(8.5\)](#)
[child \(6.4.3\)](#)
[CHILDES \(12.2\)](#)
[chink \(7.2.7\)](#)
[chink \(7.6\)](#)
[chinking \(7.2.7\)](#)
[chunker \(7.2.3\)](#)
[chunking \(7.2\)](#)
[chunks \(7.2\)](#)
[class label \(5.5.2\)](#)

[classes \(6.6\)](#)
[classification \(5.5.2\)](#)
[closed class \(4.8.4\)](#)
[closed \(11.3.1\)](#)
[closures \(3.3.2\)](#)
[coindex \(10.3.2\)](#)
[comparative wordlist \(2.4.3\)](#)
[complements \(8.3.3\)](#)
[complete edge \(8.5.3\)](#)
[complete \(9.3.4\)](#)
[complete \(9.3.4\)](#)
[complex \(10.2.3\)](#)
[complex types \(11.3.1\)](#)
[components \(1.5.4\)](#)
[computer science \(1.1\)](#)
[concatenation \(3.2.1\)](#)
[concatenation \(3.2.1\)](#)
[conditional expression \(1.4.3\)](#)
[conditional frequency distribution \(2.2\)](#)
[conditions \(11.6.1\)](#)
[confusion matrix \(4.5.7\)](#)
[consistent \(11.1.2\)](#)
[constituency \(8.2.2\)](#)
[constituents \(8.2.2\)](#)
[construction \(5.2\)](#)
[context-free grammar \(8.3\)](#)
[control \(1.4\)](#)
[control structure \(1.4.3\)](#)
[conversion \(11.3.2\)](#)
[conversion specifier \(3.8.2\)](#)
[Cooper storage \(11.4.4\)](#)
[copy \(6.1.2\)](#)
[coreferential \(11.3.1\)](#)
[corpora \(5.2\)](#)
[corpora \(5.2\)](#)
[corpus \(2\)](#)
[corpus linguistics \(1.1\)](#)

D

[daughter \(6.4.3\)](#)
[decision nodes \(5.8\)](#)
[decision stump \(5.8\)](#)
[decision tree \(5.8\)](#)
[decorator \(6.4.4\)](#)
[decrease-and-conquer \(6.4\)](#)
[dependent \(9.5\)](#)
[derived corpus \(5.3\)](#)
[determiners \(4.2.5\)](#)
[development \(5.5.3\)](#)
[diagnosis \(7.5\)](#)
[dictionary \(4.3.2\)](#)
[direct recursion \(8.3.2\)](#)
[directed acyclic graphs \(10.3.2\)](#)
[discourse referents \(11.6.1\)](#)
[discourse representation structure \(11.6.1\)](#)
[dispersion plot \(1.1.2\)](#)
[divide-and-conquer \(6.4\)](#)
[docstring \(6.2\)](#)
[domain \(11.3.4\)](#)
[dot \(8.5.2\)](#)

[dotted edges \(8.5.2\)](#)
[Dublin Core \(12.3.6\)](#)
[dynamic programming \(8.5\)](#)
[dynamically typed \(6.2.1\)](#)

E

[edge queue \(9.3.3\)](#)
[edge queue \(9.3.4\)](#)
[encode \(5.5.3\)](#)
[entails \(2.5.3\)](#)
[equivalent \(10.3.2\)](#)
[equivalents \(11.3.2\)](#)
[error analysis \(5.5.3\)](#)
[evaluation set \(5.6.1\)](#)
[event \(1.3.1\)](#)
[existential quantifier \(11.3.1\)](#)
[Exploratory data analysis \(5.2\)](#)
[export \(3.8.3\)](#)

F

[f-structure \(10.6\)](#)
[feature extraction \(5.5.3\)](#)
[feature \(10.2.2\)](#)
[feature path \(10.3.2\)](#)
[feature structure \(10.2.3\)](#)
[features \(5.5.3\)](#)
[fields \(6.1.4\)](#)
[filler \(10.4.4\)](#)
[first-in-first-out \(6.1.5\)](#)
[format string \(3.8.2\)](#)
[formulas \(11.2\)](#)
[free \(11.3.1\)](#)
[frequency distribution \(1.3.1\)](#)
[function body \(6.2\)](#)
[function \(6.2\)](#)
[function \(6.2\)](#)
[Fundamental Rule \(8.5.5\)](#)
[Fundamental Rule \(8.5.5\)](#)

G

[gaps \(10.4.4\)](#)
[General Ontology for Linguistic Description \(12.3.5\)](#)
[generator \(6.3.1\)](#)
[gerund \(4.8.1\)](#)
[Glue semantics \(11.4.4\)](#)
[gold standard \(4.4.5\)](#)
[grammar \(8.3\)](#)
[grammatical productions \(8.3.1\)](#)
[guides \(1.5\)](#)

H

[hapaxes \(1.3.1\)](#)
[head features \(10.7\)](#)
[heads \(8.3.3\)](#)
[heights \(8.2.1\)](#)
[hole semantics \(11.4.4\)](#)
[human-computer interaction \(1.1\)](#)
[humanities computing \(1.1\)](#)
[hyponyms \(2.5.2\)](#)

I

[identifiers \(1.2.3\)](#)
[immutable \(6.1.4\)](#)

[immutable \(6.1.4\)](#)
[incomplete edge \(8.5.3\)](#)
[inconsistent \(11.1.2\)](#)
[index \(1.2.2\)](#)
[indirect recursion \(8.3.2\)](#)
[Information Extraction \(7.1\)](#)
[information gain \(5.8\)](#)
[initializer \(6.6.1\)](#)
[Inline annotation \(5.3.2\)](#)
[interpreter \(1.1.1\)](#)
[IOB tags \(7.2.2\)](#)
[iterative optimization \(5.9\)](#)

K

[Kappa \(12.3.7\)](#)
[key \(4.3.2\)](#)
[key \(4.3.2\)](#)
[key-value pairs \(4.3.2\)](#)
[keyword arguments \(6.2.8\)](#)
[Kleene closures \(3.3.2\)](#)

L

[lambda abstraction \(11.3.1\)](#)
[lambda expressions \(6.2.6\)](#)
[lambda operator \(11.3.1\)](#)
[last-in-first-out \(6.1.5\)](#)
[leaf nodes \(5.8\)](#)
[leaves \(8.2.1\)](#)
[leaves \(8.2.1\)](#)
[left-corner \(8.4.3\)](#)
[left-corner parser \(8.4.3\)](#)
[left recursive \(8.3.2\)](#)
[lexical ambiguity \(8.2.1\)](#)
[lexical categories \(4.1.1\)](#)
[lexical productions \(8.3.1\)](#)
[lexicon \(12.2.1\)](#)
[library \(2.3.3\)](#)
[licensed \(10.4.4\)](#)
[licenses \(8.3.1\)](#)
[linguistic category \(10.2.3\)](#)
[list \(1.2.1\)](#)
[logical constants \(11.3.1\)](#)
[logical form \(11.2\)](#)
[long-distance dependency \(10.1\)](#)
[lowest-cost-first search strategy \(9.3.4\)](#)

M

[machine translation \(1.5.3\)](#)
[mapping \(4.3\)](#)
[maximal projection \(10.4.2\)](#)
[Maximum Entropy \(5.9\)](#)
[memoization \(6.4.4\)](#)
[meronyms \(2.5.3\)](#)
[method \(1.5\)](#)
[modals \(4.2.5\)](#)
[model \(11.1.2\)](#)
[models \(5.5\)](#)
[module \(2.3.3\)](#)
[module \(2.3.3\)](#)
[morpho-syntactic \(4.8.5\)](#)
[most likely constituents table \(9.3.2\)](#)
[mutable \(6.1.4\)](#)

[mutable \(6.1.4\)](#)**N**

[n-gram tagger \(4.5.3\)](#)
[Naive Bayes \(5.8.1\)](#)
[Named Entity Recognition \(7.1.1\)](#)
[natural language \(I\)](#)
[Natural Language Processing \(I\)](#)
[newlines \(3.1.4\)](#)
[non-logical constants \(11.3.1\)](#)
[non-terminal \(8.3.1\)](#)
[normalized \(3.5\)](#)
[noun phrase \(8.2.2\)](#)

O

[open formula \(11.3.1\)](#)
[out-of-vocabulary \(4.5.5\)](#)
[overfit \(5.8\)](#)
[overfitting \(5.5.3\)](#)

P

[package \(2.3.3\)](#)
[parameters \(5.8.6\)](#)
[parameters \(5.8.6\)](#)
[parent \(6.4.3\)](#)
[parse edge \(8.5.3\)](#)
[parser \(8.4\)](#)
[part-of-speech tagging \(4\)](#)
[partial information \(10.3.3\)](#)
[partial parsing \(7.2.1\)](#)
[parts of speech \(4.1.1\)](#)
[personal pronouns \(4.2.5\)](#)
[phrasal level \(10.4.2\)](#)
[phrase structure \(8.1.2\)](#)
[POS-tagging \(4\)](#)
[pre-sort \(6.4\)](#)
[pre-terminals \(8.3.1\)](#)
[precision/recall trade-off \(4.5.3\)](#)
[predicates \(11.3.1\)](#)
[prepositional phrase attachment ambiguity \(8.2.1\)](#)
[prepositional phrase attachment ambiguity \(8.3.1\)](#)
[Prepositional Phrase Attachment Corpus \(8.2.1\)](#)
[present participle \(4.8.1\)](#)
[Principle of Compositionality \(11.1\)](#)
[prior probability \(5.8.1\)](#)
[probabilistic context free grammar \(9.3.1\)](#)
[productions \(8.1.1\)](#)
[productions \(8.3\)](#)
[Propositional logic \(11.2\)](#)
[propositional symbols \(11.2\)](#)
[prune \(5.8\)](#)

Q

[question answering \(1.5.3\)](#)
[queue \(6.1.5\)](#)

R

[recognizing \(8.5.1\)](#)
[record \(12.2.1\)](#)
[record \(12.2.1\)](#)
[recursion \(8.1.1\)](#)
[recursion \(8.3\)](#)
[recursive \(8.3.2\)](#)

[reduce \(8.4.2\)](#)
[reduce-reduce conflict \(8.4.2\)](#)
[reentrancy \(10.3.2\)](#)
[refactor \(6.2.3\)](#)
[Relation Extraction \(7.1.1\)](#)
[relational operators \(1.4.1\)](#)
[return value \(2.3.2\)](#)
[root \(8.2.1\)](#)
[root node \(5.8\)](#)
[rules \(8.5.4\)](#)
[runtime error \(1.2.2\)](#)

S

[S-Retrieval \(11.4.4\)](#)
[satisfies \(11.3.5\)](#)
[Scanner Rule \(8.5.8\)](#)
[scope \(11.3.7\)](#)
[scope \(11.3.7\)](#)
[self-loop edge \(8.5.3\)](#)
[semantic role labeling \(1.5.2\)](#)
[semantics \(11.1\)](#)
[sequence \(3.2.6\)](#)
[shift \(8.4.2\)](#)
[shift-reduce conflict \(8.4.2\)](#)
[shift-reduce parser \(8.4.2\)](#)
[Shoebox \(12.2\)](#)
[sisters \(6.4.3\)](#)
[slash categories \(10.4.4\)](#)
[slicing \(1.2.2\)](#)
[smoothing \(5.8.3\)](#)
[stack \(6.1.5\)](#)
[standoff annotation \(12.2.3\)](#)
[standoff annotation \(12.2.3\)](#)
[start-symbol \(8.3.1\)](#)
[stopwords \(2.4.1\)](#)
[strategy \(8.5.4\)](#)
[string formatting expressions \(3.8.2\)](#)
[string \(3.2\)](#)
[strings \(1.2.4\)](#)
[structurally ambiguous \(8.3.1\)](#)
[structure sharing \(10.3.2\)](#)
[structured data \(7.1\)](#)
[stylistics \(2.1.3\)](#)
[subcategories \(8.3.3\)](#)
[Subject-Auxiliary Inversion \(8.2.2\)](#)
[subsumes \(10.3.3\)](#)
[subsumption \(10.3.3\)](#)
[subtype \(10.6\)](#)
[supervised \(5.5.2\)](#)
[Swadesh wordlists \(2.4.3\)](#)
[synonyms \(2.5.1\)](#)
[synset \(2.5.1\)](#)
[syntax error \(1.1.1\)](#)

T

[T9 \(3.3.2\)](#)
[tag \(10.3.2\)](#)
[tag pattern \(7.2.4\)](#)
[tagged \(4.2.2\)](#)
[tagging \(4\)](#)
[tagset \(4\)](#)

[terminals \(8.3.1\)](#)
[terms \(11.3.1\)](#)
[text \(12.2.1\)](#)
[textonyms \(3.3.2\)](#)
[tokenization \(3.1.1\)](#)
[tokens \(3.1.1\)](#)
[Toolbox \(12.2\)](#)
[Top-Down Expand Rule \(8.5.7\)](#)
[Top-Down Expand Rule \(8.5.7\)](#)
[top-down \(6.4.4\)](#)
[Top-Down Initialization Rule \(8.5.7\)](#)
[Top-Down Initialization Rule \(8.5.7\)](#)
[Top-Down Match Rule \(8.5.7\)](#)
[Top-Down Match Rule \(8.5.7\)](#)
[top-down parsing \(8.4.1\)](#)
[Top-Down Strategy \(8.5.7\)](#)
[total likelihood \(5.9\)](#)
[train \(4.5.1\)](#)
[training corpus \(5.5.2\)](#)
[training \(4.5.1\)](#)
[transform-and-conquer \(6.4\)](#)
[transitive verbs \(8.3.3\)](#)
[tree diagram \(8.2.1\)](#)
[truth-conditions \(11.2\)](#)
[tuple \(4.2.6\)](#)
[Turing Test \(1.5.4\)](#)
[type-raising \(11.4.2\)](#)
[type raising \(11.4.3\)](#)
[Typed feature structures \(10.6\)](#)
[types \(11.3.1\)](#)

U

[unary predicate \(11.3.1\)](#)
[unbounded dependency construction \(10.4.4\)](#)
[underspecified \(10.2.2\)](#)
[unification \(10.3.3\)](#)
[unify \(10.2.2\)](#)
[unigram chunker \(7.3.5\)](#)
[unique beginners \(2.5.2\)](#)
[universal quantifier \(11.3.1\)](#)
[unseen \(5.5\)](#)
[unstructured data \(7.1\)](#)

V

[valency \(8.3.3\)](#)
[valuation function \(11.3.4\)](#)
[value \(4.3.2\)](#)
[variable \(1.2.3\)](#)

W

[Web software development \(1.1\)](#)
[weights \(5.8.6\)](#)
[well-formed substring table \(8.5.1\)](#)
[wildcard \(3.3.1\)](#)
[word classes \(4.1.1\)](#)
[word sense disambiguation \(1.5.1\)](#)
[WordNet \(2.5\)](#)

Z

[zero projection \(10.4.2\)](#)

This chapter is a draft from *Natural Language Processing*, by [Steven Bird](#), [Ewan Klein](#) and [Edward Loper](#), Copyright © 2008 the authors. It is distributed with the *Natural Language Toolkit* [<http://www.nltk.org/>], Version 0.9.6, under the terms of the *Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License* [<http://creativecommons.org/licenses/by-nc-nd/3.0/us/>].

This document is Revision: 7166 Mon Dec 8 21:47:15 EST 2008