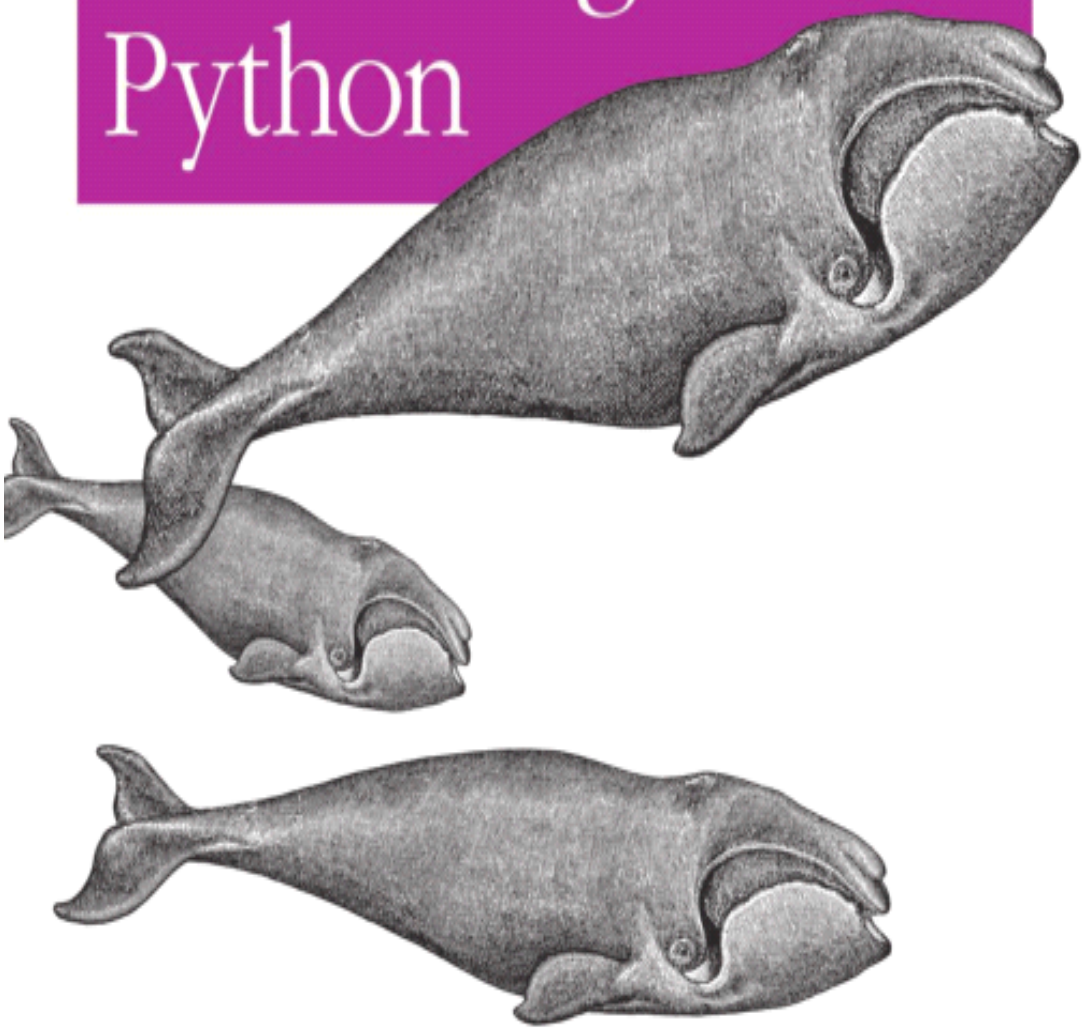
Analyzing Text with the Natural Language Toolkit

Natural Language Processing with Python



O'REILLY®

Steven Bird, Ewan Klein & Edward Loper

PYTHON 自然语言处理中文翻译

作者：Steven Bird, Ewan Klein & Edward Loper
英文版出版社：O'REILLY
翻译：陈涛（weibo.com/chentao1999）

译者的话

作为一个自然语言处理的初学者，看书看到“训练模型”，这模型那模型的，一直不知道模型究竟是什么东西。看了这本书，从预处理数据到提取特征集，训练模型，测试修改等，一步一步实际操作了之后，才对模型一词有了直观的认识（算法的中间结果，存储在计算机中的一个个 pkl 文件，测试的时候直接用，前面计算过的就省了）。以后听人谈“模型”的时候也有了底气。当然，模型还有很多其他含义。还有动词的“配价”、各种搭配、客观逻辑对根据文法生成的句子的约束如何实现？不上机动手做做，很难真正领悟。

自然语言处理理论书籍很多，讲实际操作的不多，能讲的这么系统的更少。从这个角度讲，本书是目前世界上最好的自然语言处理实践教程。初学者若在看过理论之后能精读本书，必定会有获益。这也是翻译本书的目的之一。

本书是译者课余英文翻译练习，抛砖引玉。书中存在很多问题，尤其是第 10 章命题逻辑和一阶逻辑推理在自然语言处理中的应用。希望大家多多指教。可以在微博上找到我（weibo.com/chentao1999）。虽然读中文翻译速度更快，但直接读原文更能了解作者的本意。

原书作者在书的最后列出了迫切需要帮助改进的条目，对翻译本书建议使用目标语言的例子，目前本书还只能照搬英文的例子，希望有志愿者能加入本书的中文化进程中，为中文自然语言处理做出贡献。

将本书作学习和研究之用，欢迎传播、复制、修改。山寨产品请留下译者姓名和微博。用于商业目的，请与原书版权所有者联系，译者不承担由此产生的责任。

译者
2012 年 4 月 7 日

PYTHON 自然语言处理

从输入法联想提示（predictive text）、email 过滤到自动文本摘要、机器翻译，大量的语言相关的技术都离不开自然语言处理的支持，而这本书提供了自然语言处理非常方便的入门指南。通过它，你将学到如何写能处理大量非结构化文本的 Python 程序。你将获得有丰富标注的涵盖语言学各种数据结构的数据集，而且你将学到分析书面文档内容和结构的主要算法。

通过大量的例子和联系，《PYTHON 自然语言处理》将会帮助你：

- **从非结构化文本中提取信息，无论是猜测主题还是识别“命名实体”。**
- **分析文本的语言学结构，包括文法和语义分析**
- **访问流行的语言学数据集，包括 WordNet 和 treebanks**
- **整合从语言学到人工智能的多个领域的技术**

通过使用 Python 程序设计语言和自然语言工具包（NTLK）的开源函数库，本书将帮助你获得自然语言处理的实际经验。如果你对开发 Web 应用、分析多种语言的新闻来源或者收集濒危语言感兴趣，或者仅仅对以程序员的视角看人类语言如何运作好奇，你将发现《PYTHON 自然语言处理》不仅迷人而且极其有用。

“少有的一本书，用如此清晰的方法如此优美整洁的代码处理如此复杂的问题……这是一本从中可以学习自然语言处理的书。”

——Ken Getz,
MCW Technologies 高级顾问

Steven Bird 是墨尔本大学计算机科学和软件工程系副教授，宾夕法尼亚大学语言学数据联盟高级研究助理。

Ewan Klein 是爱丁堡大学信息学院语言技术教授。

Edward Loper 是宾夕法尼亚大学基于机器学习的自然语言处理方向的刚毕业的博士，现在是波士顿的 BBN Technologies 的研究员。

oreilly.com
US $44.99 CAN $56.99
ISBN: 978-0-596-51649-9
54499
9 780596 516499

Safari Books Online
Free online edition for 45 days with purchase of this book. Details on last page.

PYTHON 自然语言处理

Steven Bird, Ewan Klein & Edward Loper

O'REILLY®

北京 • 剑桥 • 法纳姆 • 科隆 • 塞瓦斯托波尔 • 台北 • 东京

Python 自然语言处理

by Steven Bird, Ewan Klein, and Edward Loper

Copyright © 2009 Steven Bird, Ewan Klein, and Edward Loper. All rights reserved.
Printed in the United States of America.

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

可以购买O'Reilly出版的书用于教育、商业或者销售推广使用。大多数图书都有网络版（http://my.safaribooksonline.com）。更多的信息请联系我们的企业/机构销售部门：(800) 998-9938 or corporate@oreilly.com.。

编辑：Julie Steele
制作编辑：Loranah Dimant
拷贝编辑：Genevieve d'Entremont
校对：Loranah Dimant
索引编者：Ellen Troutman Zaig
封面设计：Karen Montgomery
内页设计：David Futato
插画：Robert Romano

版本说明：
2009年六月：第一版

Nutshell Handbook, the Nutshell Handbook标志, 以及O'Reilly标志是O'Reilly Media, Inc.的注册商标。《PYTHON自然语言处理》，露脊鲸图案以及相关的商品外观是O'Reilly Media, Inc.的商标。

制造商和经销商为了区分他们的产品而声明一些名称为商标。这些名称也出现在本书中，O'Reilly Media, Inc.知道这是商标，使用盖帽或者小的盖帽来印刷。

在本书编写过程中已经采取一切可能的预防措施，所以出版商和作者对书中的错误和遗漏以及使用此书包含的信息所造成的损害不承担责任。

ISBN: 978-0-596-51649-9
[M]
1244726609

目录

PYTHON 自然语言处理中文翻译 2
译者的话 2
PYTHON 自然语言处理 3
目录 6
前言 15
读者 15
强调 16
你将学到什么？ 16
篇章结构 16
为什么使用 Python？ 17
软件安装需求 18
自然语言工具包（NLTK） 18
教师请看 19
本书使用的约定 20
使用例子代码 20
Safari®联机丛书 21
如何联系我们 21
致谢 22
版税 22
第 1 章 语言处理与 Python 23
1.1 语言计算：文本和单词 23
Python 入门 23
NLTK 入门 24
搜索文本 26
计数词汇 28
1.2 近观 Python：将文本当做词链表 30
链表 30
索引列表 32
变量 34
字符串 35
1.3 计算语言：简单的统计 36
频率分布 36
细粒度的选择词 38
词语搭配和双连词（bigrams） 39
计数其他东西 39
1.4 回到 Python:决策与控制 41
条件 41
对每个元素进行操作 42
嵌套代码块 43

条件循环 44
1.5 自动理解自然语言 45
词意消歧 46
指代消解 46
自动生成语言 46
机器翻译 47
人机对话系统 48
文本的含义 49
NLP 的局限性 49
1.6 小结 50
1.7 深入阅读 50
1.8 练习 51
第 2 章 获得文本语料和词汇资源 54
2.1 获取文本语料库 54
古腾堡语料库 54
网络和聊天文本 56
布朗语料库 57
路透社语料库 59
就职演说语料库 59
标注文本语料库 60
在其他语言的语料库 62
文本语料库的结构 64
载入你自己的语料库 65
2.2 条件频率分布 66
条件和事件 66
按文体计数词汇 66
绘制分布图和分布表 67
使用双连词生成随机文本 68
2.3 更多关于 Python：代码重用 70
使用文本编辑器创建程序 70
函数 70
模块 71
2.4 词典资源 72
词汇列表语料库 73
发音的词典 75
比较词表 78
2.5 WordNet 79
意义与同义词 79
WordNet 的层次结构 81
更多的词汇关系 82
语义相似度 83
2.6 小结 84
2.7 深入阅读 85
2.8 练习 85

第 3 章 加工原料文本 88
3.1 从网络和硬盘访问文本 88
电子书 88
处理的 HTML 90
处理搜索引擎的结果 91
读取本地文件 92
从 PDF、MS Word 及其他二进制格式中提取文本 93
捕获用户输入 93
NLP 的流程 93
3.2 字符串：最底层的文本处理 94
字符串的基本操作 95
输出字符串 96
访问单个字符 97
访问子字符串 98
更多的字符串操作 99
链表与字符串的差异 99
3.3 使用 Unicode 进行文字处理 100
什么是 Unicode？ 100
从文件中提取已编码文本 101
在 Python 中使用本地编码 103
3.4 使用正则表达式检测词组搭配 104
使用基本的元字符 104
范围与闭包 105
3.5 正则表达式的有益应用 107
提取字符块 107
在字符块上做更多事情 108
查找词干 109
搜索已分词文本 110
3.6 规范化文本 111
词干提取器 112
词形归并 113
3.7 用正则表达式为文本分词 113
分词的简单方法 114
NLTK 的正则表达式分词器 115
分词的进一步问题 116
3.8 分割 116
断句 116
分词 117
3.9 格式化：从链表到字符串 120
从链表到字符串 120
字符串与格式 120
排列 122
将结果写入文件 123
文本换行 124

3.10 小结.....124
3.11 深入阅读.....125
3.12 练习.....126
第 4 章 编写结构化程序.....131
4.1 回到基础.....131
赋值.....131
等式.....133
条件语句.....133
4.2 序列.....134
序列类型上的操作.....135
合并不同类型的序列.....136
产生器表达式.....138
4.3 风格的问题.....138
Python 代码风格.....138
过程风格与声明风格.....139
计数器的一些合理用途.....141
4.4 函数：结构化编程的基础.....142
函数的输入和输出.....142
参数传递.....143
变量的作用域.....144
参数类型检查.....145
功能分解.....145
文档说明函数.....147
4.5 更多关于函数.....148
作为参数的函数.....148
累计函数.....149
高阶函数.....150
参数的命名.....150
4.6 程序开发.....152
Python 模块的结构.....152
多模块程序.....153
误差源头.....154
调试技术.....155
防御性编程.....156
4.7 算法设计.....157
递归.....157
权衡空间与时间.....159
动态规划.....161
4.8 Python 库的样例.....163
Matplotlib 绘图工具.....163
NetworkX.....165
CSV.....166
NumPy.....166
其他 Python 库.....167

4.9 小结.....167
4.10 深入阅读.....168
4.11 练习.....168
第 5 章 分类和标注词汇.....172
5.1 使用词性标注器.....172
5.2 标注语料库.....173
表示已标注的标识符.....173
读取已标注的语料库.....174
简化的词性标记集.....175
名词.....176
动词.....177
形容词和副词.....178
未简化的标记.....178
探索已标注的语料库.....179
5.3 使用 Python 字典映射词及其属性.....181
索引链表 VS 字典.....181
Python 字典.....182
定义字典.....184
默认字典.....184
递增地更新字典.....185
复杂的键和值.....187
颠倒字典.....187
5.4 自动标注.....188
默认标注器.....189
正则表达式标注器.....189
查询标注器.....190
评估.....192
5.5 N-gram 标注.....192
一元标注（Unigram Tagging）.....192
分离训练和测试数据.....193
一般的 N-gram 的标注.....193
组合标注器.....194
标注生词.....195
存储标注器.....195
性能限制.....196
跨句子边界标注.....197
5.6 基于转换的标注.....197
5.7 如何确定一个词的分类.....199
形态学线索.....199
句法线索.....199
语义线索.....200
新词.....200
词性标记集中的形态学.....200
5.8 小结.....201

5.9 深入阅读.....201
5.10 练习.....202
第六章 学习分类文本.....206
6.1 有监督分类.....206
性别鉴定.....207
选择正确的特征.....208
文档分类.....211
探索上下文语境.....213
序列分类.....214
其他序列分类方法.....216
6.2 有监督分类的更多例子.....216
句子分割.....216
识别对话行为类型.....217
识别文字蕴含.....218
扩展到大型数据集.....219
6.3 评估.....219
测试集.....220
准确度.....220
精确度和召回率.....221
混淆矩阵.....222
交叉验证.....222
6.4 决策树.....223
熵和信息增益.....224
6.5 朴素贝叶斯分类器.....225
潜在概率模型.....227
零计数和平滑.....227
非二元特征.....228
独立的朴素.....228
双重计数的原因.....228
6.6 最大熵分类器.....229
最大熵模型.....229
熵的最大化.....230
生成式分类器对比条件式分类器.....231
6.7 为语言模式建模.....231
模型告诉我们什么？.....232
6.8 小结.....232
6.9 进一步阅读.....232
6.10 练习.....233
第七章 从文本提取信息.....235
7.1 信息提取.....235
信息提取结构.....236
7.2 分块.....237
名词短语分块.....237
标记模式.....238

用正则表达式分块.....239
探索文本语料库.....239
加缝隙.....240
块的表示：标记与树.....241
7.3 开发和评估分块器.....242
读取 IOB 格式与 CoNLL2000 分块语料库.....242
简单评估和基准.....243
训练基于分类器的分块器.....245
7.4 语言结构中的递归.....249
用级联分块器构建嵌套结构.....249
树.....250
树遍历.....251
7.5 命名实体识别.....252
7.6 关系抽取.....254
7.7 小结.....255
7.8 进一步阅读.....256
7.9 练习.....256
第 8 章 分析句子结构.....259
8.1 一些语法困境.....259
语言数据和无限可能性.....259
普遍存在的歧义.....260
8.2 文法有什么用？.....262
超越 n-grams.....262
8.3 上下文无关文法.....264
一种简单的文法.....264
写你自己的文法.....266
句法结构中的递归.....267
8.4 上下文无关文法分析.....268
递归下降分析.....268
移进-归约分析.....270
左角落分析器.....271
符合语句规则的子串表.....271
8.5 依存关系和依存文法.....274
配价与词汇.....276
扩大规模.....277
8.6 文法开发.....278
树库和文法.....278
有害的歧义.....279
加权文法.....281
8.7 小结.....283
8.8 进一步阅读.....283
8.9 练习.....284
第 9 章 建立基于特征的文法.....287
9.1 文法特征.....287

句法协议.....288
使用属性和约束.....290
术语.....293
9.2 处理特征结构.....295
包含和统一.....297
9.3 扩展基于特征的文法.....300
子类别.....300
核心词回顾.....302
助动词与倒装.....303
无限制依赖成分.....304
德语中的格和性别.....307
9.4 小结.....310
9.5 进一步阅读.....310
9.6 练习.....311
第 10 章 分析句子的意思.....314
10.1 自然语言理解.....314
查询数据库.....314
自然语言、语义和逻辑.....317
10.2 命题逻辑.....319
10.3 一阶逻辑.....321
句法.....322
一阶定理证明.....324
一阶逻辑语言总结.....325
真值模型.....325
独立变量和赋值.....327
量化.....328
量词范围歧义.....329
模型的建立.....330
10.4 英语句子的语义.....332
基于特征的文法中的合成语义学.....332
λ 演算.....333
量化的 NP.....335
及物动词.....336
再述量词歧义.....338
10.5 段落语义层.....341
段落表示理论.....341
段落处理.....343
10.6 小结.....345
10.7 进一步阅读.....345
10.8 练习.....346
第 11 章 语言数据管理.....349
11.1 语料库结构：一个案例研究.....349
TIMIT 的结构.....349
主要设计特点.....351

基本数据类型.....352
11.2 语料库生命周期.....353
语料库创建的三种方案.....353
质量控制.....353
维护与演变.....354
11.3 数据采集.....355
从网上获取数据.....355
从字处理器文件获取数据.....356
从电子表格和数据库中获取数据.....357
转换数据格式.....358
决定要包含的标注层.....359
标准和工具.....359
处理濒危语言时特别注意事项.....360
11.4 使用 XML.....362
语言结构中使用 XML.....362
XML 的作用.....363
ElementTree 接口.....364
使用 ElementTree 访问 Toolbox 数据.....366
格式化条目.....368
11.5 使用 Toolbox 数据.....368
为每个条目添加一个字段.....368
验证 Toolbox 词汇.....369
11.6 使用 OLAC 元数据描述语言资源.....372
元数据是什么？.....372
OLAC：开放语言档案社区.....372
11.7 小结.....373
11.8 进一步阅读.....374
11.9 练习.....374
后记：语言的挑战.....376
语言处理与符号处理.....376
当代哲学划分.....377
NLTK 的路线图.....378
Envoi 379
参考文献.....380
NLTK 索引.....380
一般索引.....380
关于作者.....381
书的末页.....382

前言

这是一本关于自然语言处理的书。所谓“自然语言”，是指人们日常交流使用的语言，如英语，印地语，葡萄牙语等。相对于编程语言和数学符号这样的人工语言，自然语言随着一代人传给另一代人而不断演化，因而很难用明确的规则来刻画。从广义上讲，“自然语言处理”（Natural Language Processing 简称 NLP）包含所有用计算机对自然语言进行的操作，从最简单的通过计数词出现的频率来比较不同的写作风格，到最复杂的完全“理解”人所说的话，至少要能达到对人的话语作出有效反应的程度。

基于 NLP 的技术应用日益广泛。例如：手机和手持电脑支持输入法联想提示和手写识别；网络搜索引擎能搜到非结构化文本中的信息；机器翻译能把中文文本翻译成西班牙文。通过提供更自然的人机界面和更复杂的存储信息获取手段，语言处理正在这个多语种的信息社会中扮演更核心的角色。

这本书提供自然语言处理领域非常方便的入门指南。它可以用来自学，也可以作为自然语言处理或计算语言学课程的教科书，或是人工智能、文本挖掘、语料库语言学课程的补充读物。本书的实践性很强，包括几百个实际可用的例子和分级练习。

本书基于 Python 编程语言及其上的一个名为自然语言工具包（Natural Language Toolkit，简称 NLTK）的开源库。NLTK 包含大量的软件、数据和文档，所有这些都可以从 http://www.nltk.org/免费下载。NLTK 的发行版本支持 Windows、Macintosh 和 Unix 平台。我们强烈建议你下载 Python 和 NLTk，与我们一起尝试书中的例子和练习。

读者

NLP 是科学、经济、社会和文化的一个重要因素。NLP 正在迅速成长，它的很多理论和方法在大量新的语言技术中得到应用。所以对很多行业的人来说掌握 NLP 知识十分重要。在应用领域包括从事人机交互、商业信息分析、web 软件开发的人；在学术界包括从人文计算学、语料库语言学到计算机科学和人工智能领域的人。（学术界的很多人把 NLP 叫称为“计算语言学”。）

本书旨在帮助所有想要学习如何编写程序分析书面语言的人，不管他们以前的编程经验如何：

初学编程？

本书的最初几章适合没有编程经验的读者，只要你不怕应对新概念和学习新的计算机技能。遍布书中的例子和数以百计的分级练习，你都可以复制下来亲自尝试一下。如果你需要一个更一般性的关于 Python 的介绍，在 http://docs.python.org/有 Python 资源列表。

初学 Python？

有经验的程序员可以很快掌握书中用到的 Python 代码，而专注于自然语言处理。所有涉及到的 Python 的特征都经过精心解释和举例说明，你很快就会体会到 Python 用在这个应用领域是多么合适。

已经精通 Python？

你可以略读 Python 的例子而钻研第一章一开始就有的有趣的语言分析材料。你能很快在这个迷人的领域展现你的技能。

强调

本书是一本**实用**的介绍 NLP 的书。你将通过例子来学习，编写真正的程序，体会到能够通过实践验证自己想法的价值。如果你没有学过编程，本书将教你如何**编程**。与其他编程书籍不同的是，我们提供了丰富的来自 NLP 领域的实例和练习。我们撰写本书的方法也是讲究**原则和条理**的，无论是严谨的语言学还是计算分析学，我们不回避所涉及到的基础理论。我们曾经试图在理论与实践之间寻求**折中**，确定它们之间的联系与边界。最终我们认识到只要能从中受益而感到**快乐**这些都是无关紧要的，所以我们竭尽所能插入了很多既有益又有趣的应用和例子，有些甚至有些异想天开。

请注意本书并不是一本工具书。本书讲述的 Python 和 NLP 是精心挑选的，并通过教程的形式展现的。关于参考材料，请查阅 http://python.org/和 http://www.nltk.org/，那里有大量可搜索的资源。

本书也不是高深的计算机科学文章。书中的内容属于初级和中级，目标读者是那些想要学习如何使用 Python 和自然语言分析包来分析文本的人。

你将学到什么？

通过钻研本书，你将学到：

- 十分简单的程序如何就能帮你处理和分析语言数据，以及如何写这些程序
- NLP 与语言学的关键概念是如何用来描述和分析语言的
- NLP 中的数据结构和算法是怎样的
- 语言数据是如何存储为标准格式，以及如何使用数据来评估 NLP 技术的性能

根据读者知识背景和学习 NLP 的动机不同，从本书中获得的技能和知识也将不同，详情见表 P-1：

表 P-1. 读者的目标和背景不同，阅读本书可获得的技能和知识

目标	艺术人文背景	理工背景
语言分析	操控大型语料库，设计语言模型，验证由经验得出的假设。	使用数据建模，数据挖掘和知识发现的技术来分析自然语言。
语言技术	应用 NLP 技术构筑健壮的系统处理语言学任务。	使用健壮的语言处理软件中的语言学算法和数据结构

篇章结构

本书前几章按照概念的难易程度编排。先是实用性很强的语言处理的入门介绍，讲述如何使用很短的 Python 程序分析感兴趣的文本信息（1-3 章）。接着是结构化程序设计章节（第 4 章），用来巩固散布在前面几章中的编程要点。之后，速度加快，我们用一系列章节讲述语言处理的主要内容：标注、分类和信息提取（5-7 章）。接下来的三章探索分析句子、识别句法结构和构建表示句意的方法（8-10 章）。最后一章讲述如何有效管理语言数据（第 11 章）。本书结尾处的后记简要讨论了 NLP 的过去和未来。

每一章中我们都在两种不同的叙述风格间切换。一种风格是以自然语言为主线。我们分析语言，探索语言学概；在讨论中使用编程的例子。我们经常会使用尚未系统介绍的 Python 结构，这样你可以在钻研这些程序如何运作的细节之前了解它们的效能。就像学习一门外语的惯用表达一样，你能够买到好吃的糕点而不必先学会复杂的提问句型。叙述的另一种风

格是以程序设计语言为主线。我们将分析程序、探索算法，而语言学例子将扮演配角。

每章结尾都有一系列分级练习，用于巩固学到的知识。练习按照如下的标准分级：○初级练习：对范例代码作稍微修改等简单的练习；◑中级练习：深入探索材料的一个方面，需要仔细的分析和设计；●高级练习：开放的任务，挑战你对材料的理解并迫使你独立思考解决的方案（新学编程的读者应该跳过这些）。

每一章都有深入阅读环节和放在 http://www.nltk.org/网上的一个“额外”环节，用来介绍更深入的材料和一些网络资源。所有实例代码都可从网上下载。

为什么使用 Python？

Python 是一种简单但功能强大的编程语言，非常适合处理语言数据。Python 可以从 http://www.python.org/免费下载，能够在各种平台上安装运行。

下面的 4 行 Python 程序就可以操作 *file.txt* 文件，输出所有后缀是“ing”的词。

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

这段程序演示了 Python 的一些主要特征。首先，使用空白符号缩进代码，从而使 if 后面的代码都在前面一行 for 语句的范围之内；这保证了检查单词是否以“ing”结尾的测试对所有单词都进行。第二，Python 是面向对象语言。每一个变量都是包含特定属性和方法的对象。例如：变量“line”的值不仅仅是一行字符串，它是一个 string 对象，包含用来把字符串分割成词的 split()方法（或叫操作、函数）。我们在对象名称后面写句号（点）再写方法名称就可以调用对象的一个方法，即 line.splie()。第三，方法的参数写在括号内。例如：上面的例子中的 word.endswith('ing')，参数“ing”表示我们需要找的是“ing”结尾的词而不是别的结尾的词。最后也是最重要的，Python 的可读性如此之强以至于可以相当容易的猜出程序的功能，即使你以前从未写过一行代码。

我们选择 Python 是因为它的学习曲线比较平缓，文法和语义都很清晰，具有良好的处理字符串的功能。作为解释性语言，Python 便于交互式编程。作为面向对象语言，Python 允许数据和方法被方便的封装和重用。作为动态语言，Python 允许属性等到程序运行时才被添加到对象，允许变量自动类型转换，提高开发效率。Python 自带强大的标准库，包括图形编程、数值处理和网络连接等组件。

Python 在世界各地的工业、科研、教育领域应用广泛。它因为提高了软件的生产效率、质量和可维护性而备受称赞。http://www.python.org/about/success/中列举了许多成功使用 Python 的故事。

NLTK 定义了一个使用 Python 进行 NLP 编程的基础工具。它提供重新表示自然语言处理相关数据的基本类，词性标注、文法分析、文本分类等任务的标准接口以及这些任务的标准实现，可以组合起来解决复杂的问题。

NLTK 自带大量文档。作为本书的补充，http://www.nltk.org/网站提供的 API 文档涵盖工具包中每一个模块、类和函数，详细说明了各种参数，还给出了用法示例。该网站还为广大用户、开发人员和导师提供了很多包含大量的例子和测试用例的 HOWTO。

软件安装需求

为了充分利用好本书，你应该安装一些免费的软件包。http://www.nltk.org/上有这些软件包当前的下载链接和安装说明。

Python

本书的例子都假定你正在使用 Python 2.4 或 2.5 版本。一旦 NLTK 的依赖库支持 Python3.0，我们将把 NLTK 移植到 Python 3.0。

NLTK

本书的代码示例使用 NLTK 2.0 版本。NLTK 的后续版本将是兼容的。

NLTK-Data

包含本书中分析和处理的语言语料库。

NumPy（推荐）

这是一个科学计算库，支持多维数组和线性代数，在某些计算概率、标记、聚类和分类任务中用到。

Matplotlib（推荐）

这是一个用于数据可视化的 2D 绘图库，本书在产生线图和条形图的程序例子中用到。

NetworkX（可选）

这是一个用于存储和操作由节点和边组成的网络结构的函数库。可视化语义网络还需要安装 Graphviz 库。

Prover9（可选）

这是一个使用一阶等式逻辑定理的自动证明器，用于支持语言处理中的推理。

自然语言工具包（NLTK）

NLTK 创建于 2001 年，最初是宾州大学计算机与信息科学系计算语言学课程的一部分。从那以后，在数十名贡献者的帮助下不断发展壮大。如今，它已被几十所大学的课程所采纳，并作为许多研究项目的基础。表 P -2 列出了 NLTK 的一些最重要的模块。

表 P-2. 语言处理任务与相应 NLTK 模块以及功能描述

语言处理任务	NLTK 模块	功能描述
获取和处理语料库	nltk.corpus	语料库和词典的标准化接口
字符串处理	nltk.tokenize, nltk.stem	分词，句子分解提取主干
搭配发现	nltk.collocations	t-检验，卡方，点互信息 PMI
词性标识符	nltk.tag	n-gram，backoff，Brill，HMM，TnT
分类	nltk.classify, nltk.cluster	决策树，最大熵，贝叶斯，EM，k-means
分块	nltk.chunk	正则表达式，n-gram，命名实体
解析	nltk.parse	图表，基于特征，一致性，概率，依赖
语义解释	nltk.sem, nltk.inference	λ演算，一阶逻辑，模型检验
指标评测	nltk.metrics	精度，召回率，协议系数
概率与估计	nltk.probability	频率分布，平滑概率分布
应用	nltk.app, nltk.chat	图形化的关键词排序，分析器，WordNet 查看器，聊天机器人
语言学领域的工作	nltk.toolbox	处理 SIL 工具箱格式的数据

NLTK 设计中的四个主要目标：

简易性

提供一个直观的框架和大量模块，使用户获取 NLP 知识而不必陷入像标注语言数据那样繁琐的事务中。

一致性

提供一个具有一致的接口和数据结构并且方法名称容易被猜到的统一的框架。

可扩展性

提供一种结构，新的软件模块包括同一个任务中的不同的实现和相互冲突的方法都可以方便添加进来。

模块化

提供可以独立使用而与工具包的其他部分无关的组件。

对比上述目标，我们回避了工具包的潜在实用性。首先，虽然工具包提供了广泛的工具，但它不是面面俱全的。它是一个工具包而不是一个系统，它将会随着 NLP 领域一起演化。第二，虽然这个工具包的效率足以支持实际的任务，但它运行时的性能还没有高度优化。这种优化往往涉及更复杂的算法或使用 C 或 C++等较低一级的编程语言来实现。这将影响工具包的可读性且更难以安装。第三，我们试图避开巧妙的编程技巧，因为我们相信清楚直白的实现比巧妙却可读性差的方法好。

教师请看

自然语言处理一般是在本科或研究生层次的高年级开设的为期一个学期的课程。很多教师都发现，在如此短的时间里涵盖理论和实践两个方面是十分困难的。有些课程注重理论而排挤实践练习，剥夺了学生编写程序自动处理语言带来的挑战和兴奋感。另一些课程仅仅教授语言学编程而不包含任何重要的 NLP 内容。最初开发 NLTK 就是为了解决这个问题，使在一个学期里同时教授大量理论和实践成为可能，无论学生事先有没有编程经验。

算法和数据结构在所有 NLP 教学大纲中都十分重要。它们本身可能非常枯燥，而 NLTK 提供的交互式图形用户界面能一步一步看到算法过程，使它们变得鲜活。大多说 NLTK 组件都有一个无需用户输入任何数据就能执行有趣的任务的示范性例子。学习本书的一个有效的方法就是交互式重现书中的例子，把它们输入到 Python 会话控制台，观察它们做了些什么，修改它们去探索试验或理论问题。

本书包含了数百个练习，可作为学生作业的基础。最简单的练习涉及用指定的方式修改已有的程序片段来回答一个具体的问题。另一个极端，NLTK 为研究生水平的研究项目提供了一个灵活的框架，包括所有的基本数据结构和算法的标准实现，几十个广泛使用的数据集（语料库）的接口，以及一个灵活可扩展的体系结构。NLTK 网站上还有其他资源支持教学中使用 NLTK。

我们相信本书是唯一为学生提供在学习编程的环境中学习 NLP 的综合性框架。各个章节和练习通过 NLTK 紧密耦合，并将各章材料分割开，为学生（即使是那些以前没有编程经验的学生）提供一个实用的 NLP 的入门指南。学完这些材料后，学生将准备好尝试一本更加深层次的教科书，例如：《语音和语言处理》，作者是 Jurafsky 和 Martin (Prentice Hall 出版社，2008 年)。

本书介绍编程概念的顺序与众不同。以一个重要的数据类型：字符串列表（链表）开始，然后介绍重要的控制结构如推导和条件式等。这些概念允许我们在一开始就做一些有用的语言处理。有了这样做的冲动，我们回过头来系统的介绍一些基础概念，如字符串，循环，文件等。这样的方法同更传统的方法达到了同样的效果而不必要求读者自己已经对编程感兴趣。

表 P-3 列出了两个课程计划表。第一个适用于艺术人文专业，第二个适用于理工科。其他的课程计划应该涵盖前 5 章，然后把剩余的时间投入单独的领域，例如：文本分类（第 6、7 章）、文法（第 8、9 章）、语义（第 10 章）或者语言数据管理（第 11 章）。

表 P-3. 课程计划建议；每一章近似的课时数

章节	艺术人文专业	理工科
第 1 章，语言处理与 Python	2-4	2
第 2 章，获得文本语料和词汇资源	2-4	2
第 3 章，处理原始文本	2-4	2
第 4 章，编写结构化程序	2-4	1-2
第 5 章，分类和标注单词	2-4	2-4
第 6 章，学习本文分类	0-2	2-4
第 7 章，从文本提取信息	2	2-4
第 8 章，句子结构分析	2-4	2-4
第 9 章，构建基于特征的文法	2-4	1-4
第 10 章，分析句子的意义	1-2	1-4
第 11 章，语言学数据管理	1-2	1-4
总计	18-36	18-36

本书使用的约定

本书使用以下印刷约定：

黑体

表示新的术语。

斜体

用在段落中表示语言学例子、文本的名称和 URL，文件名和后缀名也用斜体。

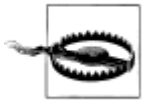
`等宽字体`

用来表示程序清单，用在段落中表示变量、函数名、语句或关键字等程序元素。也用来表示程序名。

`等宽斜体`

表示应由用户提供的值或上下文决定的值来代替文本中的值，也在程序代码例子中表示元变量。

此图标表示提示、建议或一般性注意事项。



此图标表示警告或重要提醒。

使用例子代码

本书是为了帮你完成你的工作的。一般情况下，你都可以在你的程序或文档中使用本书

中的代码。不需要得到我们获得允许，除非你要大量的复制代码。例如，写程序用到书中几段代码不需要许可。销售和分发包含 O'Reilly 书籍中例子的 CD-ROM 需要获得许可。援引本书和书中的例子来回答问题不需要许可。大量的将本书中的例子纳入你的产品文档将需要获得许可。

我们希望但不是一定要求被参考文献引用。一个引用通常包括标题，作者，出版者和 ISBN。例如："Python 自然语言处理，Steven Bird，Ewan Klein 和 Edward Loper。版权所有 2009 Steven Bird, Ewan Klein 和 Edward Loper, 978-0-596-51649-9。"如果你觉得你使用本书的例子代码超出了上面列举的一般用途或许可，随时通过 permissions@oreilly.com 联系我们。

Safari®联机丛书

当你看到你喜爱的技术书的封面上印有 Safari®联机丛书的图标时，这意味着这本书可以在 O'Reilly 网络 Safari 书架上找到。

Safari 提供比电子书更好的解决方案。它是一个虚拟图书馆，你可以轻松搜索数以千计的顶尖技术书籍，剪切和粘贴例子代码，下载一些章节，并在你需要最准确最新的信息时快速找到答案。欢迎免费试用 http://my.safaribooksonline.com。

如何联系我们

关于本书的意见和咨询请写信给出版商：

O'Reilly Media 公司
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938（在美国或加拿大）
707-829-0515（国际或本地）
707-829-0104（传真）

我们为本书的勘误表、例子等信息制作了一个网页。你可以访问这个页面：

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

作者通过 NLTK 网站提供了各章的其他材料：

http://www.nltk.org/

要发表评论或询问有关这本书的技术问题，发送电子邮件至：

bookquestions@oreilly.com

欲了解更多有关我们的书籍、会议、资源中心和 O'Reilly 网络的信息，请参阅我们的网站：

http://www.oreilly.com

致谢

作者感激为本书早期手稿提供反馈意见的人，他们是：Doug Arnold, Michaela Atterer, Greg Aumann, Kenneth Beesley, Steven Bethard, Ondrej Bojar, Chris Cieri, Robin Cooper, Grev Corbett, James Curran, Dan Garrette, Jean Mark Gawron, Doug Hellmann, Nitin Indurkhya, Mark Liberman, Peter Ljunglöf, Stefan Müller, Robin Munn, Joel Nothman, Adam Przepiorkowski, Brandon Rhodes, Stuart Robinson, Jussi Salmela, Kyle Schlansker, Rob Speer 和 Richard Sproat。感谢许许多多的学生和同事，他们关于课堂材料的意见演化成本书的这些章节，其中包括巴西，印度和美国的 NLP 与语言学暑期学校的参加者。没有 NLTK 开发社区的成员们的努力这本书也不会存在，他们为建设和壮大 NLTK 无私奉献他们的时间和专业知识，他们的名字都记录在 NLTK 网站上。

非常感谢美国国家科学基金会、语言数据联盟、Edward Clarence Dyason 奖学金、宾州大学、爱丁堡大学和墨尔本大学对我们在本书相关的工作上的支持。

感谢 Julie Steele、Abby Fox、Loranah Dimant 以及其他 O'Reilly 团队成员。他们组织大量 NLP 和 Python 社区成员全面审阅我们的手稿，很高兴的为满足我们的需要定制 O'Reilly 的生成工具。感谢他们一丝不苟的审稿工作。

最后，我们对我们的合作伙伴欠了巨额的感情债，他们是 Kay、Mimo 和 Jee。感谢在我们写作本书的几年里他们付出的爱心、耐心和支持。我们希望我们的孩子——Andrew, Alison、Kirsten、Leonie 和 Maaike——能从这些页面中赶上我们对语言和计算的热情。

版税

出售这本书的版税将被用来支持自然语言工具包的发展。

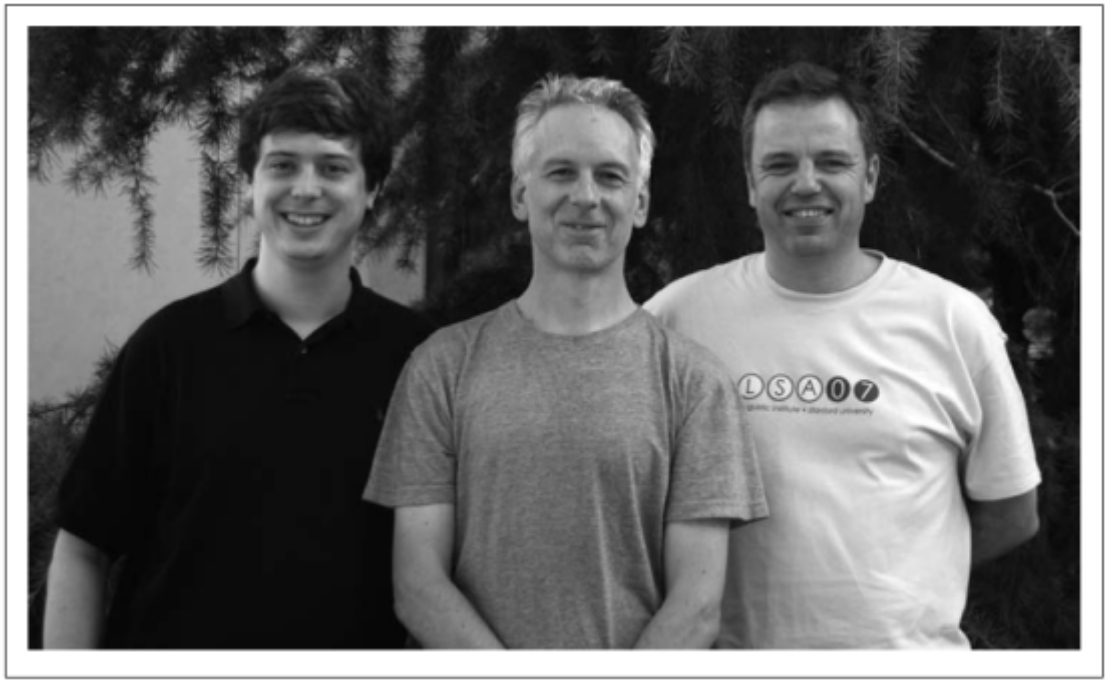


图 P-1. Edward Loper，Ewan Klein 和 Steven Bird，斯坦福大学，2007 年 7 月

第 1 章 语言处理与 Python

上百万字的文本，是容易拿到手的。假设我们会写一些简单的程序，那我们可以用它来做些什么？在本章中，我们将解决以下几个问题：

1. 将简单的程序与大量的文本结合起来，我们能实现什么？
2. 我们如何能自动提取概括文本风格和内容的关键词和短语？
3. Python 编程语言为上述工作提供了哪些工具和技术？
4. 自然语言处理中的有哪些有趣的挑战？

本章分为完全不同风格的两部分。在“语言计算”部分，我们将选取一些语言相关的编程任务而不去解释它们是如何实现的。在“近观 Python”部分，我们将系统地回顾关键的编程概念。两种风格将按章节标题区分，而后面几章将混合两种风格而不作明显的区分。我们希望这种风格的介绍能使你对接下来将要碰到的内容有一个真实的体味，与此同时，涵盖语言学与计算机科学的基本概念。如果你对这两个方面已经有了基本的了解，可以跳到 1.5 节。我们将在后续的章节中重复所有要点，如果错过了什么，你可以很容易地在 http://www.nltk.org/上查询在线参考材料。如果这些材料对你而言是全新的，那么本章将引发比解答本身更多的问题，这些问题将在本书的其余部分讨论。

1.1 语言计算：文本和单词

我们都对文本非常熟悉，因为我们每天都读到和写到。在这里，把文本视为我们写的程序的原始数据，这些程序以很多有趣的方式处理和分析文本。但在我们能写这些程序之前，我们必须得从 Python 解释器开始。

Python 入门

Python 对用户友好的一个方式是你可以在交互式**解释器**——将要运行你的 Python 代码的程序——里面直接打字。你可以通过一个简单的叫做交互式开发环境（Interactive DeveLopment Environment，简称 IDLE）的图形接口来访问 Python 解释器。在 Mac 上，你可以在“应用程序→MacPython”中找到；在 Windows 中，你可以在“程序→Python”中找到。在 Unix 下，你可以在 shell 输入“idle”来运行 Python（如果没有安装，尝试输入 python）。解释器将会输入关于你的 Python 的版本简介，请检查你是否运行在 Python 2.4 或 2.5（这里是 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.
>>>
```

如果你无法运行 Python 解释器可能是因为没有正确安装 Python。请访问 http://python.org/查阅详细操作说明。

>>>提示符表示 Python 解释器正在等待输入。复制这本书的例子时，自己不要键入“>>>”。现在，让我们开始把 Python 当作计算器使用：

一旦解释器计算并显示出答案，提示符就会出现。这表示 Python 解释器在等待另一个指令。

轮到你来：输入一些你自己的表达式。你可以使用星号（*）表示乘法，左斜线表示除法，你可以用括号括起表达式。请注意：除法并不总是像你可能期望的那样。当你输入 1/3 时是整数除法（小数会被四舍五入），输入 1.0/3.0 时是“浮点数”（或十进制）除法。要想获得通常我们期望的除法（在 Python3.0 中是标准），你需要输入：from __future__ import division。

前面的例子演示了如何交互式的使用 Python 解释器，试验 Python 语言中各种表达式，看看它们做些什么。现在让我们尝试一个无意义的表达式，看看解释器如何处理：

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

产生了一个语法错误。在 Python 中，指令以加号结尾是没有意义的。Python 解释器会指出发生错误的行（<stdin>的第 1 行，<stdin>表示“标准输入”）。

现在我们学会使用 Python 解释器了，已经准备好可以开始处理语言数据了。

NLTK 入门

首先应该安装 NLTk。可以从 http://www.nltk.org/免费下载。按照说明下载适合你的操作系统的版本。

安装完 NLTK 之后，像前面那样启动 Python 解释器。在 Python 提示符后面输入下面两个命令来安装本书所需的数据，然后选择 book，如图 1-1 所示。

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

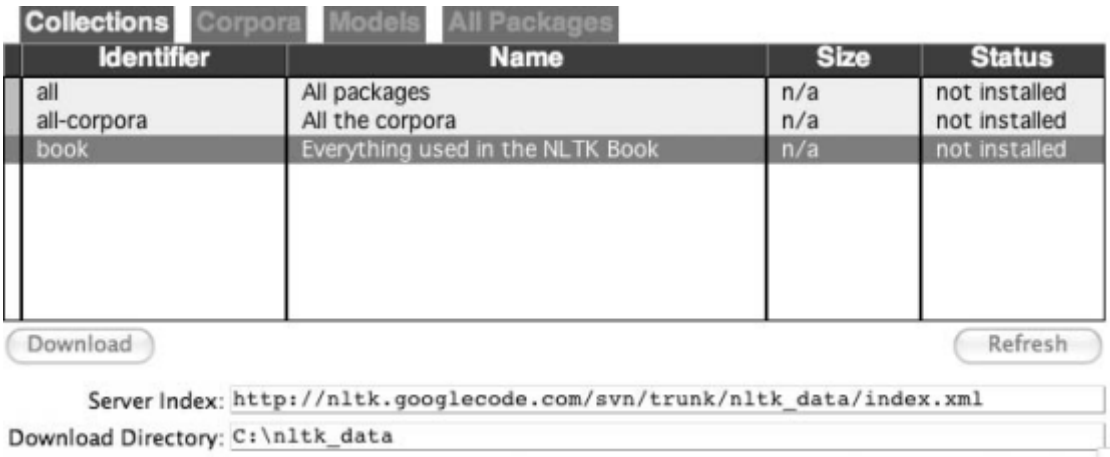


图 1-1. 下载 NLTK 图书集：使用 nltk.download()浏览可用的软件包。下载器上的 Collections 选项卡显示软件包如何被打包分组。选择 book 标记所在行，可以获取本书的例子和练习所需的全部数据。这些数据包括约 30 个压缩文件，需要 100MB 硬盘空间。完整的数据集（即下载器中的 all）在本书写作期间大约是这个大小的 5 倍，还在不断扩充。

一旦数据被下载到你的机器，你就可以使用 Python 解释器加载其中一些。第一步是在 Python 提示符后输入一个特殊的命令，告诉解释器去加载一些我们要用的文本：from nltk.book import * 。这条语句是说“从 NLTK 的 book 模块加载所有的东西”。这个 book 模块包含你阅读本章所需的所有数据。在输出欢迎信息之后，将会加载几本书的文本（这将需要几秒钟）。下面连同你将看到的输出一起再次列出这条命令，注意拼写和标点符号的正确性，记住不要输入>>>。

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

任何时候我们想要找到这些文本，只需要在 Python 提示符后输入它们的名字。

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

现在我们可以和这些数据一起来使用 Python 解释器，我们已经准备好上手了。

搜索文本

除了阅读文本之外，还有很多方法可以用来研究文本内容。词语索引视图显示一个指定单词的每一次出现，连同一些上下文一起显示。下面我们输入 text1 后面跟一个点，再输入函数名 concordance，然后将 monstrous 放在括号里，来查一下《白鲸记》中的词 *monstrous*：

```
>>> text1.concordance("monstrous")
Building index...
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himm
they might scout at Moby Dick as a monstrous fable , or still worse and more de
h of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
>>>
```

轮到你来：尝试搜索其他词。为了方便重复输入，你也许会用到上箭头，Ctrl-↑或者 Alt-p 获取之前输入的命令，然后修改要搜索的词。你也可以搜索我们已经列入的其他文本。例如：使用 text2.concordance("affection") 搜索《理智与情感》中的词 *affection*；使用 text3.concordance("lived") 搜索《创世纪》找出某人活了多久；你也可以看看 text4，《就职演说语料》，回到 1789 年看看那时英语的例子，搜索如 *nation*, *terror*, *god* 这样的词，看看随着时间推移这些词的使用如何不同；我们也包括了 text5，《NPS 聊天语料库》，你可以在里面搜索一些网络词，如 *im*, *ur*, *lol*。（注意这个语料库未经审查！）

在你花了一小会儿研究这些文本之后，我们希望你对语言的丰富性和多样性有一个新的认识。在下一章中，你将学习获取更广泛的文本，包括英语以外其他语言的文本。

词语索引使我们看到词的上下文。例如：我们看到 *monstrous* 出现的上下文，如 *the ___ pictures* 和 *the ___ size*。还有哪些词出现在相似的上下文中？我们可以通过在被查询的文本名后添加函数名 similar，然后在括号中插入相关的词来查找到。

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

remarkably extremely vast
>>>

观察我们从不同的文本中得到的不同结果。Austen(奥斯丁，英国女小说家)使用这些词与 Melville 完全不同；在她那里，*monstrous* 是正面的意思，有时它的功能像词 *very* 一样作强调成分。

函数common_contexts允许我们研究两个或两个以上的词共同的上下文，如*monstrous*和*very*。我们必须用方括号和圆括号把这些词括起来，中间用逗号分割。

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

轮到你来：挑选另一对词，使用 similar()和 common_contexts()函数比较它们在两个不同文本中的用法。

自动检测出现在文本中的特定的词，并显示同样上下文中出现的一些词，这只是一个方面。我们也可以判断词在文本中的位置：从文本开头算起在它前面有多少词。这个位置信息可以用**离散图**表示。每一个竖线代表一个单词，每一行代表整个文本。图1-2中我们看到在过去220年中的一些显著的词语用法模式（在一个由就职演说语料首尾相连的人为组合的文本中）。可以用下面的方法画出这幅图。你也许会想尝试更多的词（如：*liberty*，*constitution*）和不同的文本。你能在看到这幅图之前预测一个词的分布吗？跟以前一样，请保证引号、逗号、中括号及小括号的使用完全正确。

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

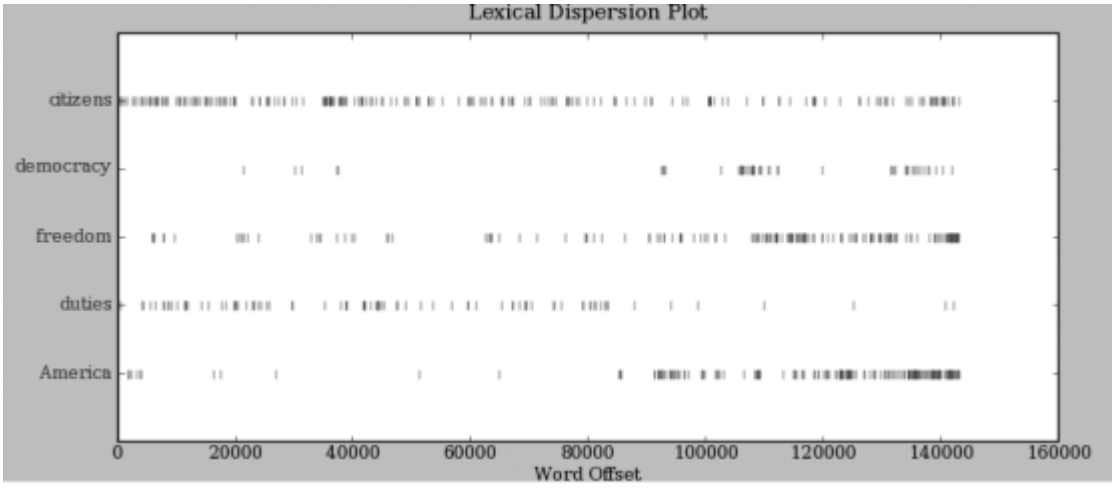


图 1-2. 美国总统就职演说词汇分布图：可以用来研究随时间推移语言使用上的变化

重要事项：为了画出这本书中用到的图形，你需要安装 Python 的 NumPy 的 Matplotlib 包。请参阅 http://www.nltk.org/上的安装说明。

现在轻松一下，让我们尝试以我们刚才看到的不同风格产生一些随机文本。要做到这一点，我们需要输入文本的名字后面跟函数名 generate。(需要带括号，但括号里没有也什么。)

```
>>> 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
>>>
```

请注意，第一次运行此命令时，由于要搜集词序列的统计信息而执行的比较慢。每次运行它，输出的文本都会不同。现在试试产生就职演说风格或互联网聊天室风格的随机文本。虽然文本是随机的，但它重用了源文本中常见的词和短语，从而使我们能感觉到它的风格和内容。

在 generate 产生输出时，标点符号被从前面的词分裂出去。虽然这不是正确的英文格式，但我们之所以这么做是为了弄清楚文字和标点符号是彼此独立的。你将在第 3 章学到更多这方面的内容。

计数词汇

关于前面例子中出现的文本，最明显的事实是它们所使用的词汇不同。在本节中，我们将看到如何使用计算机以各种有用的方式计数词汇。像以前一样，你将会马上开始用 Python 解释器进行试验，即使你可能还没有系统的研究过 Python。修改这些例子测试一下你是否理解它们，尝试一下本章结尾处的练习。

首先，让我们以文本中出现的词和标点符号为单位算出文本从头到尾的长度。我们使用函数 len 获取长度，请看在《创世纪》中使用的例子：

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

《创世纪》有 44764 个词和标点符号或者叫“标识符”。一个**标识符**是表示一个我们想要放在一组对待的字符序列——如：hairy、his 或者:)——的术语。当我们计数文本中标识符的个数时，如 *to be or not to be* 这句话，我们计数这些序列出现的次数。因此，我们的例句中出现了 *to* 和 *be* 各两次，*or* 和 *not* 各一次。然而在例句中只有 4 个不同的词。《创世纪》中有多少不同的词？要用 Python 来回答这个问题，我们处理问题的方法将稍有改变。一个文本词汇表只是它用到的标识符的集合，因为在集合中所有重复的元素都只算一个。Python 中我们可以使用命令：set(text3)获得 text3 的词汇表。当你这样做时，屏幕上的很多词就会被掠过。现在尝试以下操作：

```
>>> sorted(set(text3)) ①
['!', '"', '(', ')', ',', ',)', '.', '.)', ':', ';', ';)', '?', '?)',
'A', 'Abel', 'Abelmizraim', 'Abidah', 'Abide', 'Abimael', 'Abimelech',
'Abr', 'Abrah', 'Abraham', 'Abram', 'Accad', 'Achbor', 'Adah', ...]
>>> len(set(text3)) ②
2789
>>>
```

用 sorted()包裹起 Python 表达式 set(text3)①，我们得到一个词汇项的排序表，这个表以各种标点符号开始，然后是以 *A* 开头的词汇。大写单词排在小写单词前面。我们通过求集合中项目的个数间接获得词汇表的大小。再次使用 len 来获得这个数值②。尽管小说中有 44,764 个标识符，但只有 2,789 个不同的词汇或“词类型”。一个**词类型**是指一个词在一个文本中独一无二的出现形式或拼写。也就是说，这个词在词汇表中是唯一的。我们计数的 2,789 个项目中包括标点符号，所以我们把这些叫做唯一项目**类型**而不是词类型。

现在，让我们对文本词汇丰富度进行测量。下一个例子向我们展示了每个字平均被使用了 16 次（我们需要确保 Python 使用的是浮点除法）：

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

接下来，让我们专注于特定的词。计数一个词在文本中出现的次数，计算一个特定的词在文本中占据的百分比。

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

轮到你来：text5 中 *lol* 出现了多少次？它占文本全部词数的百分比是多少？

你也许想要对几个文本重复这些计算，但重新输入公式是乏味的。你可以自己命名一个任务，如“lexical_diversity”或“percentage”，然后用一个代码块关联它。现在，你只需输入一个很短的名字就可以代替一行或多行 Python 代码，而且你想用多少次就用多少次。执行一个任务的代码段叫做一个**函数**。我们使用关键字 def 给函数定义一个简短的名字。下面的例子演示如何定义两个新的函数，lexical_diversity()和 percentage()：

```
>>> def lexical_diversity(text): ①
...     return len(text) / len(set(text)) ②
...
>>> def percentage(count, total): ③
...     return 100 * count / total
...
```

注意！

当遇到第一行末尾的冒号后，Python 解释器提示符由>>>变为...。...提示符表示 Python 期望在后面是一个**缩进代码块**。缩进是输入四个空格还是敲击 Tab 键，这由你决定。要结束一个缩进代码段，只需输入一个空行。

lexical_diversity()①的定义中，我们指定了一个 text **参数**。这个参数是我们想要计算词汇多样性的实际文本的一个“占位符”。第②行是使用这个函数时想要重现的代码段。类似地，percentage()③定义了两个参数：count 和 total。

只要 Python 知道了 lexical_diversity()和 percentage()是指定代码段的名字，我们

就可以继续使用这些函数：

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

扼要重述一下，我们使用或叫**调用**一个如 lexical_diversity()这样的函数，只要输入它的名字后面跟一个左括号，再输入文本名字，然后是右括号。这些括号经常出现，它们的作用是分割任务名——如：lexical_diversity()——与任务将要处理的数据——如：text3。调用函数时放在参数位置的数据值叫做函数的**实参**。

在本章中你已经遇到了几个函数，如：len()，set()和 sorted()。通常我们会在函数名后面加一对空括号，像 len()中的那样，这只是为了表明这是一个函数而不是其他的 Python 表达式。函数是编程中的一个重要概念，我们在一开始提到它们，是为了让新同学体会编程的强大和富有创造力。如果你现在觉得有点混乱，请不要担心。

稍后我们将看到如何使用函数列表显示数据，像表 1-1 显示的那样。表的每一行将包含不同数据的相同的计算，我们用函数来做这种重复性的工作。

表 1-1. Brown 语料库中各种文体的词汇多样性

体裁	标识符	类型	词汇多样性
技能和爱好	82345	11935	6.9
幽默	21695	5017	4.3
小说：科学	14470	3233	4.5
新闻：报告文学	100554	14394	7.0
小说：浪漫	70022	8452	8.3
宗教	39399	6373	6.2

1.2 近观 Python：将文本当做词链表

你已经见过 Python 编程语言的一些重要元素。让我们花几分钟系统的复习一下。

链表

文本是什么？在一个层面上，它是一页纸上的符号序列就像这页纸一样。在另一个层面上，它是章节的序列，每一章由小节序列组成，小节由段落序列组成，以此类推。然而，对于我们而言，我们认为文本不外乎是词和标点符号的序列。下面是我们如何在 Python 中表示文本，如何表示《白鲸记》的开篇句：

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

在提示符后面，我们输入自己命名的 sent1，后跟一个等号，然后是一些引用的词汇，

中间用逗号分割并用括号包围。这个方括号内的东西在 Python 中叫做**链表**（list，也叫列表）：它就是我们存储文本的方式。我们可以通过输入它的名字①来查阅它。我们可以查询它的长度②，甚至可以在我们自己的函数 lexical_diversity()中使用它③。

```
>>> sent1 ①
['Call', 'me', 'Ishmael', '.']
>>> len(sent1) ②
4
>>> lexical_diversity(sent1) ③
1.0
>>>
```

已经为你定义了一些链表，每个文本开始的句子定义为 sent2...sent9。在这里我们检查其中的两个。你可以自己在 Python 解释器中尝试其余的（如果你得到一个错误说：sent2 没有定义，你需要先输入 from nltk.book import *）。

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

轮到你来：通过输入名字、等号和一个词链表，组建一些你自己的句子，如 ex1 = ['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']。重复一些我们先前在 1.1 节看到的其他 Python 操作，如：sorted(ex1)，len(set(ex1))，ex1.count('the')。

令人惊喜的是，我们可以对链表使用 Python 加法运算。两个链表相加①创造出一个新的链表，包括第一个链表的全部，后面跟着第二个链表的全部。

```
>>> ['Monty', 'Python'] + ['and', 'the', 'Holy', 'Grail'] ①
['Monty', 'Python', 'and', 'the', 'Holy', 'Grail']
```

这种加法的特殊用途叫做**连接**；它将多个链表组合为一个链表。我们可以把句子连接起来组成一个文本。

不必逐字的输入链表，可以使用简短的名字来引用预先定义好的链表。

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

如果我们想要向链表中增加一个元素该如何？这种操作叫做**追加**。当我们对一个链表使用 append()时，链表自身会随着操作而更新。

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

>>>

索引列表

正如我们已经看到的，Python 中的一个文本是一个词汇的链表，用括号和引号的组合来表示。就像处理一页普通的文本，我们可以使用 len(text1)计算 text1 的词数，使用 text1.count('heaven')计算一个文本中出现的特定的词，如 *heaven*。

稍微花些耐心，我们可以挑选出一篇打印出来的文本中的第 1 个、第 173 个或第 14278 个词。类似的，我们也可以通过它在链表中出现的次序找出一个 Python 链表的元素。表示这个位置的数字叫做这个元素的**索引**。在文本名称后面的方括号里写下索引，Python 就会表示出文本中这个索引处——例如 173——的元素。

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

我们也可以反过来做；找出一个词第一次出现的索引。

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

索引是一种常见的用来获取文本中词汇的方式，或者，更一般的，访问链表中的元素的方式。Python 也允许我们获取子链表，从大文本中任意抽取语言片段，术语叫做**切片**。

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

索引有一些微妙，我们将在一个构造的句子的帮助下探讨这些：

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

请注意，索引从零开始：第 0 个元素写作 sent[0]，其实是第 1 个词“word1”；而句子的第 9 个元素是“word10”。原因很简单：Python 从计算机内存中的链表获取内容的时候，我们要告诉它向前多少个元素。因此，向前 0 个元素使它留在第一个元素上。

这种从零算起的做法刚开始接触会有些混乱，但这是现代编程语言普遍使用的。如果你已经掌握了 19XY 是 20 世纪中的一年这样的计数世纪的系统，或者如果你生活在一个建筑物楼层编号从 1 开始的国家，你很开就会掌握它的窍门，步行 n-1 级楼梯到第 n 层。

现在，如果我们不小心使用的索引过大就会得到一个错误：

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

这次不是一个语法错误，因为程序片段在语法上是正确的。相反，它是一个**运行时错误**，它会产生一个回溯消息显示错误的上下文、错误的名称：IndexError 以及简要的解释说明。

让我们再次使用构造的句子仔细看看切片，这里我们发现切片 5:8 包含索引 *5*，*6* 和 *7* 的句子元素。

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

按照惯例，m:n 表示元素 *m…n-1*。正如下一个例子显示的那样，如果切片从链表第一个元素开始，我们可以省略第一个数字①；如果切片到链表最后一个元素处结尾，我们可以省略第二个数字②：

```
>>> sent[:3] ①
['word1', 'word2', 'word3']
>>> text2[141525:] ②
['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']
>>>
```

我们可以通过指定它的索引值来修改链表中的元素。在接下来的例子中，我们把 sent[0] 放在等号左侧①。我们也可以用新内容替换掉一整个片段②。最后一个尝试报错的原因是这个链表只有四个元素而要获取 4 后面的元素就产生了错误③。

```
>>> sent[0] = 'First' ①
>>> sent[9] = 'Last'
>>> len(sent)
10
```

```
>>> sent[1:9] = ['Second', 'Third'] ②
>>> sent
['First', 'Second', 'Third', 'Last']
>>> sent[9] ③
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

轮到你来：花几分钟定义你自己的句子，使用前文中的方法修改个别词和词组（切片）。尝试本章结尾关于链表的练习，检验你是否理解。

变量

从 1.1 节一开始，你已经访问过名为 text1，text2 等的文本。像这样只输入简短的名字来引用一本 250,000 字的书节省了很多打字时间。一般情况下，我们可以对任何我们关心的计算命名。我们在前面的小节中已经这样做了，如下所示，定义一个变量 sent1：

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

这样的语句形式是：*变量 = 表达式*。Python 将计算右边的表达式把结果保存在变量中。这个过程叫做**赋值**。它并不产生任何输出，你必须在新的一行输入变量的名字来检查它的内容。等号可能会有些误解，因为信息是从右边流到左边的。你把它想象成一个左箭头可能会有帮助。变量的名字可以是任何你喜欢的名字，如：my_sent、sentence、xyzzy 等。变量必须以字母开头，可以包含数字和下划线。下面是变量和赋值的一些例子：

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

请记住，排序表中大写字母出现在小写字母之前。

请注意，在前面的例子中，我们将 my_sent 的定义分成两行。Python 表达式可以被分割成多行，只要它出现在任何一种括号内。Python 使用...提示符表示期望更多的输入。在这些连续的行中有多少缩进都没有关系，只是加入缩进通常会便于阅读。

最好是选择有意义的变量名，它能提醒你代码的含义，也帮助别人读懂你的 Python 代码。Python 并不会理解这些名称的意义。它只是盲目的服从你的指令，如果你输入一些令人困惑的代码，例如：one = 'two'或者 two = 3，它也不会反对。唯一的限制是变量名不能是 Python 的保留字，如 def，if，not 和 import。如果你使用了保留字，Python 会产生

一个语法错误：

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

我们将经常使用变量来保存计算的中间步骤，尤其是当这样做使代码更容易读懂时。因此，len(set(text1))也可以写作:

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

注意！

为 Python 变量选择名称（或**标识符**）时请注意：首先，应该以字母开始，后面跟数字（0 到 9）或字母。因此，abc23 是好的，23abc 会导致一个语法错误。名称是大小写敏感的。这意味着 myVar 和 myvar 是不同的变量。变量名不能包含空格，但可以用下划线把单词分开，如 my_var。注意不要插入连字符来代替下划线：my-var 不对，因为 Python 会把-解释为减号。

字符串

我们用来访问链表元素的一些方法也可以用在单独的词或**字符串**上。例如可以把一个字符串指定给一个变量①，索引一个字符串②，切片一个字符串③。

```
>>> name = 'Monty' ①
>>> name[0] ②
'M'
>>> name[:4] ③
'Mont'
>>>
```

我们还可以对字符串执行乘法和加法：

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

我们可以把词用链表连接起来组成单个字符串，或者把字符串分割成一个链表，如下面所示：

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

```
>>>
```

我们将在第 3 章回到字符串的主题。目前，我们已经有了两个重要的基石——链表和字符串——已经准备好可以重新做一些语言分析了。

1.3 计算语言：简单的统计

让我们重新开始探索用我们的计算资源处理大量文本的方法。我们在 1.1 节已经开始讨论了，在那里我们看到如何搜索词及其上下文，如何汇编一个文本中的词汇，如何产生一种文体的随机文本等。

在本节中，我们重新拾起是什么让一个文本不同与其他文本这样的问题，并使用程序自动寻找特征词汇和文字表达。正如在 1.1 节中那样，你可以通过复制它们到 Python 解释器中来尝试 Python 语言的新特征，你将在下一节中系统的了解这些功能。

在这之前，你可能会想通过预测下面的代码的输出来检查你对上一节的理解。你可以使用解释器来检查你是否正确。如果你不确定如何做这个任务，你最好在继续之前复习一下上一节的内容。

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

频率分布

我们如何能自动识别文本中最能体现文本的主题和风格的词汇？试想一下，要找到一本书中使用最频繁的 50 个词你会怎么做？一种方法是为每个词项设置一个计数器，如图 1-3 显示的那样。计数器可能需要几千行代码，这将是一个极其繁琐的过程——如此繁琐以至于我们宁愿把任务交给机器来做。

Word Tally

the	𝍸 𝍸 𝍸 ||||
been	𝍸 𝍸 |
message	||||
persevere	|
nation	𝍸 |||

图 1-3. 计数一个文本中出现的词（频率分布）

图 1-3 中的表被称为**频率分布**，它告诉我们在文本中的每一个词项的频率。（一般情况下，它能计数任何观察得到的事件。）这是一个“分布”因为它告诉我们文本中词标识符的总数是如何分布在词项中的。因为我们经常需要在语言处理中使用频率分布，NLTK 中内置

了它们。让我们使用 FreqDist 寻找《白鲸记》中最常见的 50 个词。尝试下面的例子，然后阅读接下来的解释。

```
>>> fdist1 = FreqDist(text1) ①
>>> fdist1 ②
<FreqDist with 260819 outcomes>
>>> vocabulary1 = fdist1.keys() ③
>>> vocabulary1[:50] ④
[',', '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
>>>
```

第一次调用 FreqDist 时，传递文本的名称作为参数①。我们可以看到已经被计算出来的《白鲸记》中的总的词数（“结果”）——高达 260,819②。表达式 keys()为我们提供了文本中所有不同类型的链表③，我们可以通过切片看看这个链表的前 50 项④。

轮到你来：使用 text2 尝试前面的频率分布的例子。注意正确使用括号和大写字母。如果你得到一个错误消息：NameError：name 'FreqDist' is not defined，你需要在一开始输入 nltk.book import *。

上一个例子中是否有什么词有助于我们把握这个文本的主题或风格呢？只有一个词，*whale*，稍微有些信息量！它出现了超过 900 次。其余的词没有告诉我们关于文本的信息，它们只是“管道”英语。这些词在文本中占多少比例？我们可以产生一个这些词汇的累积频率图，使用 fdist1.plot(50, cumulative=True)产生图 1-4。这 50 个词占了书的将近一半!

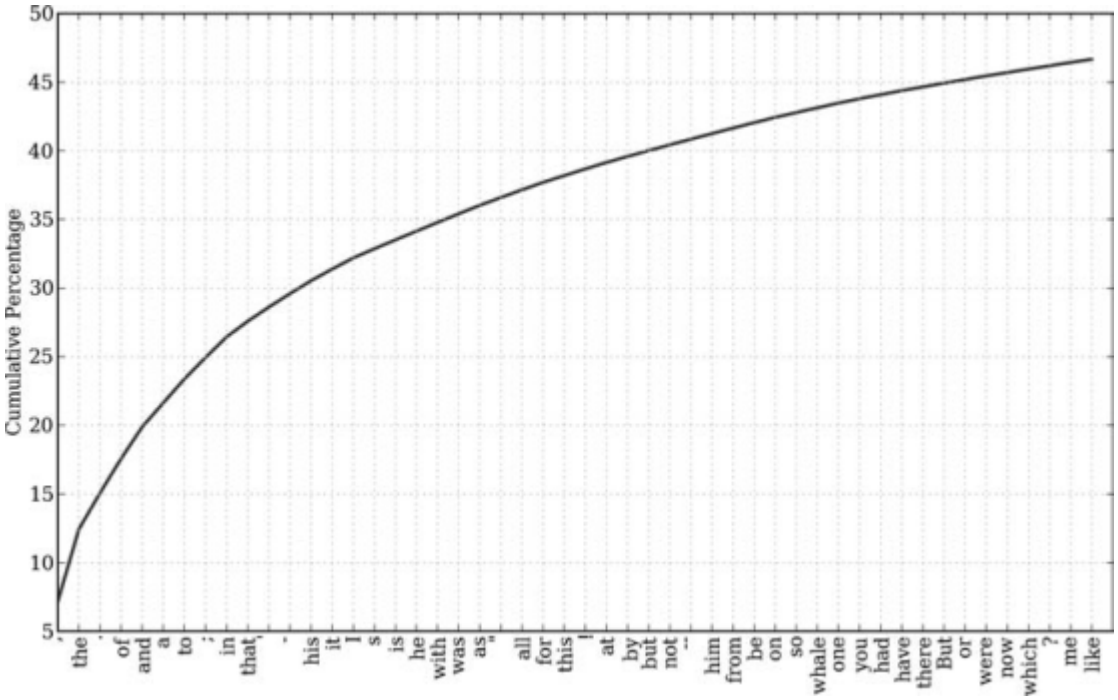


图 1-4. 《白鲸记》中 50 个最常用词的累积频率图，这些词占了所有标识符的将近一半

如果高频词对我们没有帮助，那些只出现了一次的词（所谓的 **hapaxes**）又如何呢？输

入 fdist1.hapaxes()查看它们。此链表包括 *lexicographer*，*cetological*，*contraband*，*expostulations* 等 9,000 多个词。看来低频词太多了，没看到上下文我们很可能有一半的 hapaxes 猜不出它们的意义！既然高频词和低频词都没有帮助，我们需要尝试其他的办法。

细粒度的选择词

接下来，让我们看看文本中的*长词*，也许它们有更多的特征和信息量。为此我们采用集合论的一些符号。我们想要找出文本词汇表长度中超过 15 个字符的词，定义这个性质为 P，则 P(w)为真当且仅当词 w 的长度大余 15 个字符。现在我们可以用（1a）中的数学集合符号表示我们感兴趣的词汇。它的含义是：此集合中所有 w 都满足 w 是集合 V（词汇表）的一个元素且 w 有性质 P。

(1) a. {w | w ∈ V & P(w)}

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

（1b）给出了对应的 Python 表达式。（请注意，它产生一个链表，而不是集合，这意味着可能会有相同的元素。）观察这两个表达式它们是多么相似。让我们进行下一步，编写可执行的 Python 代码：

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

对于词汇表 V 中的每一个词 w，我们检查 len(w) 是否大于 15；所有其他词汇将被忽略。我们将在后面更仔细的讨论这里的语法。

轮到你来：在 Python 解释器中尝试上面的表达式，改变文本和长度条件做一些实验。如果改变变量名，你的结果会产生什么变化吗？如使用[word for word in vocab if ...]？

让我们回到寻找文本特征词汇的任务上来。请注意，text4 中的长词反映国家主题——*constitutionally*（按宪法规定的，本质地），*transcontinental*（横贯大陆的）——而 text 5 中的长词是非正规表达方式，例如 *boooooooooooglyyyyyy* 和 *yuuuuuuuuuuuummmmmmmmmmmm*。我们是否已经成功的自动提取出文本的特征词汇呢？好的，这些很长的词通常是 hapaxes（即唯一的），也许找长词出现的频率会更好。这样看起来更有前途，因为这样忽略了短高频词（如 the）和长低频词（如 antiphilosophists）。以下是聊天语料库中所有长度超过 7 个字符出现次数超过 7 次的词：

```
>>> 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']
>>>

注意我们是如何使用两个条件：len(w) > 7 保证词长都超过七个字母，fdist5[w]> 7 保证这些词出现超过 7 次。最后，我们已成功地自动识别出与文本内容相关的高频词。这很小的一步却是一个重要的里程碑：一小块代码，处理数以万计的词，产生一些有信息量的输出。

词语搭配和双连词（bigrams）

一个**搭配**是异乎寻常的经常在一起出现的词序列。*red wine* 是一个搭配而 *the wine* 不是。一个搭配的特点是其中的词不能被类似的词置换。例如：*maroon wine*（粟色酒）听起来就很奇怪。

要获取搭配，我们先从提取文本词汇中的词对也就是双连词开始。使用函数 bigrams() 很容易实现。

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

在这里我们看到词对 *than-done* 是一个双连词，在 Python 中写成('than','done')。现在，除非我们更加注重包含不常见词的的情况，搭配基本上就是频繁的双连词。特别的，我们希望找到比我们基于单个词的频率预期得到的更频繁出现的双连词。collocations()函数为我们做这些（我们将在以后看到它是如何工作）：

```
>>> text4.collocations()
Building collocations list
United States; fellow citizens; years ago; Federal Government; General
Government; American people; Vice President; Almighty God; Fellow
citizens; Chief Magistrate; Chief Justice; God bless; Indian tribes;
public debt; foreign nations; political parties; State governments;
National Government; United Nations; public money
>>> text8.collocations()
Building collocations list
medium build; social drinker; quiet nights; long term; age open;
financially secure; fun times; similar interests; Age open; poss
rship; single mum; permanent relationship; slim build; seeks lady;
Late 30s; Photo pls; Vibrant personality; European background; ASIAN
LADY; country drives
>>>
```

文本中出现的搭配很能体现文本的风格。为了找到 *red wine* 这个搭配，我们将需要处理更大的文本。

计数其他东西

计数词汇是有用的，我们也可以计数其他东西。例如，我们可以查看文本中词长的分布，

通过创造一长串数字的链表的 FreqDist，其中每个数字是文本中对应词的长度。

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

我们以导出 text1 中每个词的长度的链表开始①，然后 FreqDist 计数链表中每个数字出现的次数②。结果③是一个包含 25 万左右个元素的分布，每一个元素是一个数字，对应文本中一个词标识符。但是只有 20 个不同的元素，从 1 到 20，因为只有 20 个不同的词长。也就是说，有由 1 个字符，2 个字符，...，20 个字符组成的词，而没有由 21 个或更多字符组成的词。有人可能会问不同长度的词的频率是多少？（例如，文本中有多少长度为 4 的词？长度为 5 的词是否比长度为 4 的词多？等等）。下面我们回答这个问题：

```
>>> 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
>>>
```

由此我们看到，最频繁的词长度是 3，长度为 3 的词有 50,000 多个（约占书中全部词汇的 20%）。虽然我们不会在这里追究它，关于词长的进一步分析可能帮助我们了解作者、文体或语言之间的差异。表 1-2 总结了 NLTK 频率分布类中定义的函数。

表 1-2. NLTK 频率分布类中定义的函数

例子	描述
fdist = FreqDist(samples)	创建包含给定样本的频率分布
fdist.inc(sample)	增加样本
fdist['monstrous']	计数给定样本出现的次数
fdist.freq('monstrous')	给定样本的频率
fdist.N()	样本总数
fdist.keys()	以频率递减顺序排序的样本链表
for sample in fdist:	以频率递减的顺序遍历样本
fdist.max()	数值最大的样本
fdist.tabulate()	绘制频率分布表
fdist.plot()	绘制频率分布图

fdist.plot(cumulative=True)	绘制累积频率分布图
fdist1 < fdist2	测试样本在 fdist1 中出现的频率是否小于 fdist2

关于频率分布的讨论中引入了一些重要的 Python 概念，我们将在 1.4 节系统的学习它们。

1.4 回到 Python:决策与控制

到目前为止，我们的小程序有了一些有趣的特征：处理语言的能力和通过自动化节省人力的潜力。程序设计的一个关键特征是让机器能按照我们的意愿决策，遇到特定条件时执行特定命令，或者对文本数据从头到尾不断循环遍历直到条件满足。这一特征被称为**控制**，是本节的重点。

条件

Python 广泛支持多种运算符，如：<和> =可以测试值之间的关系。全部的**关系运算符**见表 1-3：

表 1-3. 数值比较运算符

运算符	关系
<	小于
<=	小于等于
==	等于（注意是两个“=”号而不是一个）
!=	不等于
>	大于
>=	大于等于

我们可以使用这些从新闻文本句子中选出不同的词。下面是一些例子——注意行与行之间只是运算符不同。它们都使用 sent7，text7（华尔街日报）的第一句话。像以前一样，如果你得到一个错误说，sent7 没有定义，你需要事先输入：from nltk.book import *。

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

>>>

所有这些例子都有一个共同的模式：[w for w in text if *condition*]，其中 *condition* 是 Python 中的一个“测试”，得到真（true）或者假（false）。在前面的代码例子所示的情况中，条件始终是数值比较。然而，我们也可以使用表 1-4 中列出的函数测试词汇的各种属性。

表 1-4. 一些词比较运算符

函数	含义
s.startswith(t)	测试 s 是否以 t 开头
s.endswith(t)	测试 s 是否以 t 结尾
t in s	测试 s 是否包含 t
s.islower()	测试 s 中所有字符是否都是小写字母
s.isupper()	测试 s 中所有字符是否都是大写字母
s.isalpha()	测试 s 中所有字符是否都是字母
s.isalnum()	测试 s 中所有字符是否都是字母或数字
s.isdigit()	测试 s 中所有字符是否都是数字
s.istitle()	测试 s 是否首字母大写（s 中所有的词都首字母大写）

下面是一些用来从我们的文本中选择词汇的运算符的例子：以*-ableness* 结尾的词；包含 *gnt* 的词；首字母大写的词；完全由数字组成的词。

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

我们还可以创建更复杂的条件。如果 c 是一个条件，那么 not c 也是一个条件。如果我们有两个条件 c1 和 c2，那么我们可以使用合取和析取将它们合并形成一个新的条件：c1 and c2 以及 c1 or c2。

轮到你来：运行下面的例子，尝试解释每一条指令中所发生的事情。然后，试着自己组合一些条件。

```
>>> 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])
```

对每个元素进行操作

在 1.3 节中，我们看到计数词汇以外的其他项目的一些例子。让我们仔细看看我们所使用的符号：

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

这些表达式形式为[f(w) for ...]或[w.f() for ...]，其中 f 是一个函数，用来计算词长或把字母转换为大写。现阶段你还不需要理解两种表示方法：f(w)与 w.f()之间的差异，而只需学习对链表上的所有元素执行相同的操作的这种 Python 习惯用法（idiom）。在前面的例子中，遍历 text1 中的每一个词，一个接一个的赋值给变量 w 并在变量上执行指定的操作。

上面描述的表示法被称为“链表推导”，这是我们的第一个 Python 习惯用法的例子，一中固定的表示法，我们习惯使用的方法，省去了每次分析的烦恼。掌握这些习惯用法是成为一流 Python 程序员的一个重要组成部分。

让我们回到计数词汇的问题，这里使用相同的习惯用法：

```
>>> len(text1)
260819
>>> len(set(text1))
19317
>>> len(set([word.lower() for word in text1]))
17231
>>>
```

由于我们不重复计算像 *This* 和 *this* 这样仅仅大小写不同的词，就已经从词汇表计数中抹去了 2,000 个！还可以更进一步，通过过滤掉所有非字母元素，从词汇表中消除数字和标点符号。

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

这个例子稍微有些复杂：将所有纯字母组成的词小写。也许只计数小写的词会更简单一些，但这却是一个错误的答案（为什么？）。

如果你对链表推导不那么充满信心，请不要担心，因为在下面的章节中你会看到更多的例子及解释。

嵌套代码块

大多数编程语言允许我们在**条件表达式**或者说 if 语句条件满足时执行代码块。我们已经看到了如[w for w in sent7 if len(w) < 4]这样的条件测试的例子。在下面的程序中，我们创建一个叫 word 的变量包含字符串值“cat”。在 if 语句中检查 len(word)< 5 是否为真。cat 的长度确实小于 5，所以 if 语句下的代码块被调用，print 语句被执行，向用户显示一条消息。别忘了要缩进，在 print 语句前输入四个空格。

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

```
... ①
word length is less than 5
>>>
```

使用 Python 解释器时，我们必须添加一个额外的空白行①，这样它才能检测到嵌套块结束。

如果我们改变测试条件为 len(word) >= 5，检查词的长度是否大于或等于 5，那么测试将不再为真。此时，if 语句后面的代码段将不会被执行，没有消息显示给用户：

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

一个 if 语句叫做一个**控制结构**，因为它控制缩进块中的代码是否运行。另一个控制结构是 for 循环。尝试下面的代码，请记住输入冒号和四个空格：

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

这叫做一个循环，因为 Python 以循环的方式执行里面的代码。它以 word='Call'赋值开始，使用变量 word 命名链表的第一个元素。然后，显示 word 的值给用户。接下来回到 for 语句，执行 word = 'me'赋值，然后显示这个新值给用户，以此类推。它以这种方式不断运行，直到链表中所有项都被处理完。

条件循环

现在，我们可以将 if 语句和 for 语句结合。循环链表中每一项，只输出结尾字母是 *l* 的词。我们将为变量挑选另一个名字以表明 Python 并不在意变量名的意义。

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

你会发现在 if 和 for 语句所在行末尾——缩进开始之前——有一个冒号。事实上，所有的 Python 控制结构都以冒号结尾。冒号表示当前语句与后面的缩进块有关联。

我们也可以指定当 if 语句的条件不满足时采取的行动。在这里，我们看到 elif(else if) 语句和 else 语句。请注意，这些在缩进代码前也有冒号。

```
>>> for token in sent1:
...     if token.islower():
```

```
...             print token, 'is a lowercase word'
...         elif token.istitle():
...             print token, 'is a titlecase word'
...         else:
...             print token, 'is punctuation'
...
Call is a titlecase word
me is a lowercase word
Ishmael is a titlecase word
. is punctuation
>>>
```

正如你看到的，即便只有这么一点儿 Python 知识，你就已经可以开始构建多行的 Python 程序。分块开发程序，在整合它们之前测试每一块代码是否达到你的预期是很重要的。这也是 Python 交互式解释器的价值所在，也是为什么你必须适应它的原因。

最后，让我们把一直在探索的习惯用法组合起来。首先，我们创建一个包含 *cie* 或者 *cei* 的词的链表，然后循环输出其中的每一项。请注意 print 语句结尾处的逗号，它告诉 Python 在同一行输出。

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

1.5 自动理解自然语言

我们一直在各种文本和 Python 编程语言的帮助自下而上的探索语言。然而，我们也对通过构建有用的语言技术，开拓我们的语言和计算知识感兴趣。现在，我们将借此机会从代码的细节中退出来，描绘一下自然语言处理的全景图。

纯粹应用层面，我们大家都需要帮助才能在网络上的文本中找到有用的信息。搜索引擎在网络的发展和普及中发挥了关键作用，但也有一些缺点。它需要技能、知识和一点运气才能找到这样一些问题的答案：“*我用有限的预算能参观费城和匹兹堡的哪些景点？*”，“*专家们怎么评论数码单反相机？*”，“*过去的一周里可信的评论员都对钢材市场做了哪些预测？*”。让计算机来自动回答这些问题，涉及包括信息提取、推理与总结在内的广泛的语言处理任务，将需要在一个更大规模更稳健的层面实施，这超出了我们当前的能力。

哲学层面，构建智能机器是人工智能长久以来的挑战，语言理解是智能行为的重要组成部分。这一目标多年来一直被看作是太困难了。然而，随着 NLP 技术日趋成熟，分析非结构化文本的方法越来越健壮，应用越来越广泛，对自然语言理解的期望变成一个合理的目标再次浮现。

在本节中，我们将描述一些语言理解技术，给你一种有趣的挑战正在等着你的感觉。

词意消歧

在**词意消歧**中，我们要算出特定上下文中的词被赋予的是哪个意思。思考存在歧义的词 serve 和 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

在包含短语 he served the dish 的句子中，你可以知道 serve 和 dish 都用的是它们与食物相关的含义。在短短的 3 个词的地方，讨论的话题不太可能从体育转向陶器。这也许会迫使你眼前产生一幅怪异的画面：一个职业网球手正把他的郁闷发泄到放在网球场边上的陶瓷茶具上。换句话说，自动消除歧义需要使用上下文，利用相邻词汇有相近含义这样一个简单的事实。在另一个有关上下文影响的例子是词 by，它有几种含义，例如：the book by Chesterton（施事——Chesterton 是书的作者）；the cup by the stove（位置格——炉子在杯子旁边）；submit by Friday（时间——星期五前提交）。观察(3)中斜体字的含义有助于我们解释 by 的含义。

(3) a. The lost children were found by the *searchers* (施事)
b. The lost children were found by the *mountain* (位置格)
c. The lost children were found by the *afternoon* (时间)

指代消解

一种更深刻的语言理解是解决“谁对谁做了什么”，即检测主语和动词的宾语。虽然你在小学已经学会了这些，但它比你想象的更难。在句子 the thieves stole the paintings 中，很容易分辨出谁做了偷窃的行为。考虑(4)中句子的三种可能，尝试确定是什么被出售、被抓和被发现（其中一种情况是有歧义的）。

(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*.

要回答这个问题涉及到寻找代词 they 的**先行词** thieves 或者 paintings。处理这个问题的计算技术包括**指代消解**（anaphora resolution）——确定代词或名词短语指的是什么——和**语义角色标注**（semantic role labeling）——确定名词短语如何与动词相关联（如施事，受事，工具等）。

自动生成语言

如果我们能够解决自动语言理解等问题，我们将能够继续那些包含自动生成语言的任务，如**自动问答**和**机器翻译**。在自动问答中，一台机器要能够回答用户关于特定文本集的问题：

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

机器的回答表明，它已经正确的计算出 they 是指 paintings，而不是 thieves。在机器翻译中，机器要能够把文本翻译成另一种语言文字，并准确传达原文的意思。在把例子文本译

成法文过程中，我们不得不在第二句选择代词的性别：ils（男性）如果 thieves 被出售，elles（女性）如果 paintings 被出售。正确的翻译实际上取决于对代词的正确理解。

(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)

所有这些例子中，弄清楚词的含义、动作的主语以及代词的先行词是理解句子含义的步骤，也是我们希望语言理解系统能够做到的事情。

机器翻译

长久以来，机器翻译（MT）都是语言理解的圣杯，人们希望能找到从根本上提供高品质的符合语言习惯的任意两种语言之间的翻译。其历史可以追溯到冷战初期，当时自动翻译的许诺带来大量的政府赞助，它也是 NLP 本身的起源。

今天，特定语言之间实用的翻译系统已经存在，有些已经集成到搜索引擎中了。但是，这些系统有一些严重的缺点。我们可以在 NLTK 的“babelizer”的帮助下探索它们（当你使用 `from nltk.book import *` 导入本章的材料时，已经自动装载了）。这个程序把提交的英文句子翻译成指定语言，然后把结果重新翻译回英文。这样重复 12 次结束或者得到曾经产生过的翻译时（表示一个循环）结束。

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

请看，该系统正确地把 Alice Springs 从英文翻译成了德文（第 1 行），但在翻译回英文时却以 Alice jump 结束（第 2 行）。介词 before 一开始被翻译成对应的德文介词 vor，但后来变成了连词 bevor（第 5 行）。第 5 行以后的句子变的没有意义（但请注意以逗号指示的各种分句，以及从 jump 到 leap 的变化）。翻译系统不能识别一个词是某个名字的一部分，并且弄错了语法结构。语法问题在下面的例子中更加明显。是约翰发现了猪，还是猪找到约翰？

```
>>> babelize_shell()
Babel> The pig that John found looked happy
```

Babel> german

Babel> run

0> The pig that John found looked happy

1> Das Schwein, das John fand, schaute gl?cklich

2> The pig, which found John, looked happy

机器翻译是困难的，因为一个给定的词可能有几种不同的解释（取决于它的意思），也因为必须改变词序才能与目标语言的语法结构保持一致。今天，这些困难遇到新情况，从新闻和政府网站发布的两种或两种以上的语言文档中可以收集到大量的相似文本。给出一个德文和英文双语的文档或者一个双语词典，我们就可以自动配对组成句子，这个过程叫做**文本对齐**。一旦我们有一百万或更多的句子对，就可以检测出相应的词和短语，并建立一个能用来翻译新文本的模型。

人机对话系统

在人工智能的历史，主要的智能测试是一个语言学测试，叫做**图灵测试**：一个响应用户文本输入的对话系统能否表现的自然到我们无法区分它是人工生成的响应？相比之下，今天的商业对话系统能力是非常有限的，但在较小的给定领域仍然有些作用，就像我们在这里看到的：

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.

你不能要求这个系统提供驾驶指示或附近餐馆的细节，除非所需的信息已经被保存并且合适的问题答案对已经被纳入语言处理系统。

请看，这个系统似乎了解用户的目标：用户询问电影上映的时间，系统正确的判断出用户是想要看电影。这一推断看起来如此明显，你可能都没有注意到它，一个自然语言系统需要被赋予这种自然的交互能力。没有它，当问到："*你知道拯救大兵瑞恩什么时候上映？*"时，系统可能只会回答一个冷冷的毫无用处的"*是的*"。然而，商业对话系统的开发者使用上下文语境假设和业务逻辑确保在用户以不同方式表达需求或提供信息时对特定应用都能有效处理。因此，如果你输入 When is ...或者 I want to know when ...或者 Can you tell me when ...时，这些简单的规则总是对应着放映时间，这就足够系统提供有益的服务了。

对话系统给我们一个机会来说说一般认为的 NLP 流程。图 1-5 显示了一个简单的对话系统架构。沿图的顶部从左向右是一些语言理解**组件**的"管道"。这些组件从语音输入经过文法分析到某种意义的重现。图的中间，从右向左是这些组件的逆向流程，将概念转换为语音。这些组件构成了系统的动态方面。在图的底部是一些有代表性的静态信息：语言相关的数据仓库，这些用于处理的组件在其上运作。

轮到你来：作为一个原始的对话系统的例子，尝试与 NLTK 的 chatbot 谈话。使用之前请运行 nltk.chat.chatbots()。（记住要先 import nltk。）

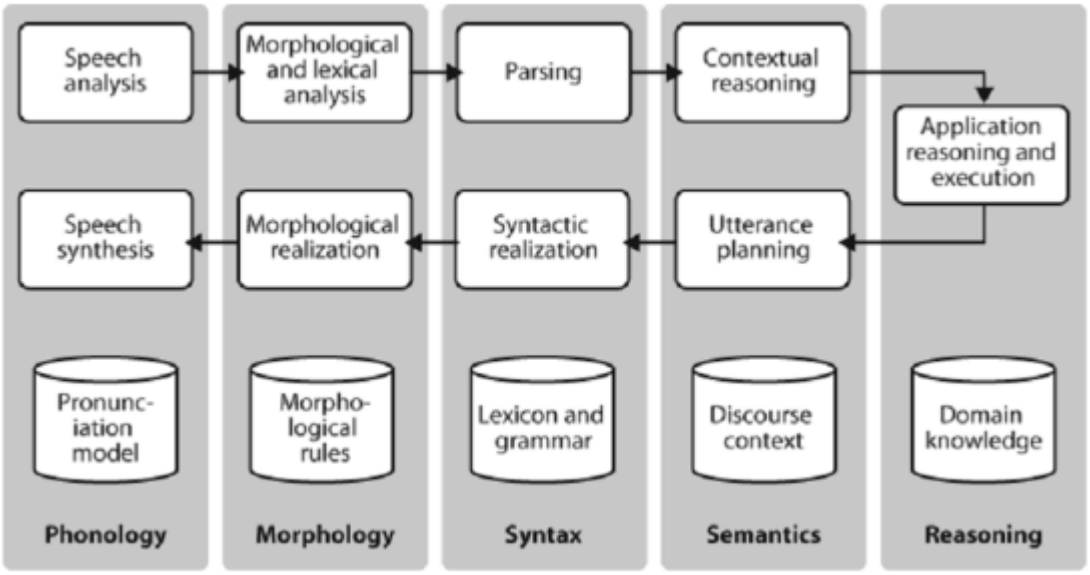


图 1-5. 简单的语音对话系统的流程架构：分析语音输入（左上），识别单词，文法分析和在上下文中解释，应用相关的具体操作（右上）;响应规划，实现文法结构，然后是适当的词形变化，最后到语音输出；处理的每个过程都蕴含不同类型的语言学知识。

文本的含义

近年来，一个叫做文本含义识别(Recognizing Textual Entailment 简称 RTE)的公开的“共享任务”使语言理解所面临的挑战成为关注焦点。基本情形很简单：假设你想找到证据来支持一个假设：Sandra Goudie 被 Max Purnell 击败了。而你有一段简短的文字似乎是有关的，例如：Sandra Goudie 在 2002 年国会选举首次当选，通过击败工党候选人 Max Purnell 将现任绿党下院议员 Jeanette Fitzsimons 推到第三位，以微弱优势赢得了 Coromandel 席位。文本是否为你接受假说提供了足够的证据呢？在这种特殊情况下，答案是“否”。你可以很容易得出这样的结论，但使用自动方法做出正确决策是困难的。RTE 挑战为竞赛者开发他们的系统提供数据，但这些数据对“蛮力”机器学习技术（我们将在第 6 章讲述这一主题）来说是不够的。因此，一些语言学分析是至关重要的。在前面的例子中，很重要的一点是让系统知道 Sandra Goudie 是假设中被击败的人，而不是文本中击败别人的人。思考下面的文本-假设对，这是任务困难性的另一个例证：

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

为了确定假说是否得到文本的支持，该系统需要以下背景知识：

（一）如果有人是一本书的作者，那么他/她写了这本书;

（二）如果有人是一本书的编辑，那么他/她（完全）没有写这本书;

（三）如果有人是 18 本书的编辑或作者，则无法断定他/她是 18 本书的作者。

NLP 的局限性

尽管在很多如 RTE 这样的任务中研究取得了进展，但在现实世界的应用中已经部署的

语言理解系统仍不能进行常识推理或以一种一般的可靠的方式描绘这个世界的知识。我们在等待这些困难的人工智能问题得到解决的同时，接受一些在推理和知识能力上存在严重限制的自然语言系统是有必要的。因此，从一开始，自然语言处理研究的一个重要目标一直是使用浅显但强大的技术代替无边无际的知识和推理能力，促进构建“语言理解”技术的艰巨任务的不断取得进展。事实上，这是本书的目标之一，我们希望你能掌握这些知识和技能，构建有效的自然语言处理系统，并为构建智能机器这一长期的理想做出贡献。

1.6 小结

- 在 Python 中文本用链表来表示：['Monty', 'Python']。我们可以使用索引、分片和 len()函数对链表进行操作。
- 词“token”（标识符）是指文本中给定词的特定出现；词“type”（类型）则是指词作为一个特定序列字母的唯一形式。我们使用 len(text)计数词的标识符，使用 len(set(text))计数词的类型。
- 我们使用 sorted(set(t))获得文本 t 的词汇表。
- 我们使用[f(x) for x in text]对文本的每一项目进行操作。
- 为了获得没有大小写区分和忽略标点符号的词汇表，我们可以使用 set([w.lower() for w in text if w.isalpha()])。
- 我们使用 for 语句对文本中的每个词进行处理，例如 for win t:或者 for word in text:。后面必须跟冒号和一块在每次循环被执行的缩进的代码。
- 我们使用 if 语句测试一个条件：if len(word)<5:。后面必须跟冒号和一块仅当条件为真时执行的缩进的代码。
- 频率分布是项目连同它们的频率计数的集合(例如：一个文本中的词与它们出现的频率)。
- 函数是指定了名字并且可以重用的代码块。函数通过 def 关键字定义，例如在 def mult(*x*, *y*)中 *x* 和 *y* 是函数的参数，起到实际数据值的占位符的作用。
- 函数是通过指定它的名字及一个或多个放在括号里的实参来调用，就像这样：mult(3, 4)或者 len(text1)。

1.7 深入阅读

本章介绍了一些编程、自然语言处理和语言学的新概念，都混在一起。其中不少将会在下面的章节得到巩固。你可能也想咨询与本章相关的在线材料（在 http://www.nltk.org/），包括额外的背景资料的链接以及在线 NLP 系统的链接。你可能还喜欢在维基百科中阅读一些语言学和自然语言处理相关的概念（如：搭配，图灵测试，类型-标识符的区别等）。

你应该自己去熟悉 http://docs.python.org/上的 Python 文档，那里链接了许多教程和全面的参考材料。http://wiki.python.org/moin/BeginnersGuide 上有《Python 初学者指南》。关于 Python 的各种问题在 http://www.python.org/doc/faq/general/的 FAQ 中得到回答。

随着你对 NLTK 研究的深入，你可能想订阅有关新版工具包的邮件列表。还有一个 NLTK 用户邮件列表，用户在学习如何使用 Python 和 NLTK 做语言分析工作时使用它来相互帮助。在 http://www.nltk.org/中有这些列表的详情。

如果需要第 1.5 节所讲述的话题以及更一般的 NLP 的相关信息，你可能想阅读以下的优秀图书中的一本：

- Indurkhya, Nitin 和 Fred Damerau（合编，2010）自然语言处理手册（Handbook of Natural Language Processing）（第二版），Chapman & Hall/CRC。
- Jurafsky, Daniel 和 James Martin（2008）语音和语言处理（Speech and Language Processing）（第二版），Prentice Hall。
- Mitkov, Ruslan（主编，2002 年），牛津计算语言学手册（The Oxford Handbook of Computational Linguistics）。牛津大学出版社。（预计在 2010 年出第二版）

计算语言学协会（The Association for Computational Linguistics，简称 ACL）是代表 NLP 领域的国际组织。ACL 网站上有许多有用的资源，包括：有关国际和地区的会议及研讨会的信息；链接到数以百计有用资源的 ACL Wiki；包含过去 50 年以来大多数 NLP 研究文献的 ACL 选集，里面的论文全部建立索引且可免费下载。

一些介绍语言学的优秀的教科书：(Finegan, 2007), (O’Grady et al., 2004), (OSU, 2007)。你可能想查阅 LanguageLog，一个流行的语言学博客，不定期发布一些本书中描述的技术应用。

1.8 练习

1. ○尝试使用 Python 解释器作为一个计算器，输入表达式，如 12/(4+1)。
2. ○26 个字母可以组成 26 的 10 次方或者 26**10 个 10 字母长的字符串。也就是 141167095653376L（结尾处的 L 只是表示这是 Python 长数字格式）。100 个字母长度的字符串可能有多少个？
3. ○Python 乘法运算可应用于链表。当你输入['Monty', 'Python'] * 20 或者 3 * sent1 会发生什么？
4. ○复习 1.1 节关于语言计算的内容。在 text2 中有多少个词？有多少个不同的词？
5. ○比较表格 1-1 中幽默和言情小说的词汇多样性得分，哪一个文体中词汇更丰富？
6. ○制作《理智与情感》中四个主角：Elinor，Marianne，Edward 和 Willoughby 的分布图。在这部小说中关于男性和女性所扮演的不同角色，你能观察到什么？你能找出一对夫妻吗？
7. ○查找 text5 中的搭配。
8. ○思考下面的 Python 表达式：len(set(text4))。说明这个表达式的用途。描述在执行此计算中涉及的两个步骤。
9. ○复习 1.2 节关于链表和字符串的内容。
   a. 定义一个字符串，并且将它分配给一个变量，如：my_string = 'My String'(在字符串中放一些更有趣的东西)。用两种方法输出这个变量的内容，一种是通过简单地输入变量的名称，然后按回车；另一种是通过使用 print 语句。
   b. 尝试使用 my_string+ my_string 或者用它乘以一个数将字符串添加到它自身，例如：my_string* 3。请注意，连接在一起的字符串之间没有空格。怎样能解决这个问题？
10. ○使用的语法 my_sent = ["My", "sent"]，定义一个词链表变量 my_sent（用你自己的词或喜欢的话）。
    a. 使用' '.join(my_sent)将其转换成一个字符串。
    b. 使用 split()在你指定的地方将字符串分割回链表。

11. ○定义几个包含词链表的变量，例如：phrase1，phrase2 等。将它们连接在一起组成不同的组合（使用加法运算符），最终形成完整的句子。len(phrase1 + phrase2)与 len(phrase1) + len(phrase2)之间的关系是什么？
12. ○考虑下面两个具有相同值的表达式。哪一个在 NLP 中更常用？为什么？
    a. "Monty Python"[6:12]
    b. ["Monty", "Python"][1]
13. ○我们已经看到如何用词链表表示一个句子，其中每个词是一个字符序列。sent1[2][2]代表什么意思？为什么？请用其他的索引值做实验。
14. ○在变量 sent3 中保存的是 text3 的第一句话。在 sent3 中 *the* 的索引值是 1，因为 sent3[1]的值是“the”。sent3 中“the”的其它出现的索引值是多少？
15. ○复习 1.4 节讨论的条件语句。在聊天语料库（text5）中查找所有以字母 b 开头的词，按字母顺序显示出来。
16. ○在 Python 解释器提示符下输入表达式 range(10)。再尝试 range(10, 20), range(10, 20, 2)和 range(20, 10, -2)。在后续章节中我们将看到这个内置函数的多用用途。
17. ◑使用 text9.index()查找词 sunset 的索引值。你需要将这个词作为一个参数插入到圆括号之间。通过尝试和出错的过程中，找到完整的句子中包含这个词的切片。
18. ◑使用链表加法、set 和 sorted 操作，计算句子 sent1...sent8 的词汇表。
19. ◑下面两行之间的差异是什么？哪一个的值比较大？其他文本也是同样情况吗？
    ```
    >>> sorted(set([w.lower() for w in text1]))
    >>> sorted([w.lower() for w in set(text1)])
    ```
20. ◑w.isupper()和 not w.islower()这两个测试之间的差异是什么？
21. ◑写一个切片表达式提取 text2 中最后两个词。
22. ◑找出聊天语料库（text5）中所有四个字母的词。使用频率分布函数（FreqDist），以频率从高到低显示这些词。
23. ◑复习 1.4 节中条件循环的讨论。使用 for 和 if 语句组合循环遍历《巨蟒和圣杯》（text6）的电影剧本中的词，输出所有的大写词，每行输出一个。
24. ◑写表达式找出 text6 中所有符合下列条件的词。结果应该是词链表的形式：['word1', 'word2', ...]。
    a. 以 ize 结尾
    b. 包含字母 z
    c. 包含字母序列 pt
    d. 除了首字母外是全部小写字母的词（即 titlecase）
25. ◑定义 sent 为词链表['she', 'sells', 'sea', 'shells', 'by', 'the', 'sea', 'shore']。编写代码执行以下任务：
    a. 输出所有 sh 开头的单词
    b. 输出所有长度超过 4 个字符的词
26. ◑下面的 Python 代码是做什么的？sum([len(w) for w in text1])，你可以用它来算出一个文本的平均字长吗？
27. ◑定义一个名为 vocab_size(text)的函数，以文本作为唯一的参数，返回文本的词汇量。
28. ◑定义一个函数 percent(word, text)，计算一个给定的词在文本中出现的频率，结果以百分比表示。
29. ◑我们一直在使用集合存储词汇表。试试下面的 Python 表达式：set(sent3) < set(t

ext1)。实验在 set()中使用不同的参数。它是做什么用的？你能想到一个实际的应用吗？

第 2 章 获得文本语料和词汇资源

在自然语言处理的实际项目中，通常要使用大量的语言数据或者**语料库**。本章的目的是要回答下列问题：

1. 什么是有用的文本语料和词汇资源，我们如何使用 Python 获取它们？
2. 哪些 Python 结构最适合这项工作？
3. 编写 Python 代码时我们如何避免重复的工作？

本章继续通过语言处理任务的例子展示编程概念。在系统的探索每一个 Python 结构之前请耐心等待。如果你看到一个例子中含有一些不熟悉的东西，请不要担心。只需去尝试它，看看它做些什么——如果你很勇敢——通过使用不同的文本或词替换代码的某些部分来进行修改。这样，你会将任务与编程习惯用法关联起来，并在后续的学习中了解怎么会这样和为什么是这样。

2.1 获取文本语料库

正如刚才提到的，一个文本语料库是一大段文本。许多语料库的设计都要考虑一个或多个文体间谨慎的平衡。我们曾在第 1 章研究过一些小的文本集合，例如美国总统就职演说。这种特殊的语料库实际上包含了几十个单独的文本——每个人一个演讲——但为了处理方便，我们把它们头尾连接起来当做一个文本对待。第 1 章中也使用变量预先定义好了一些文本，我们通过输入 from book import * 来访问它们。然而，因为我们希望能够处理其他文本，本节中将探讨各种文本语料库。我们将看到如何选择单个文本，以及如何处理它们。

古腾堡语料库

NLTK 包含古腾堡项目（Project Gutenberg）电子文本档案的经过挑选的一小部分文本。该项目大约有 25,000（现在是 36,000 了）本免费电子图书，放在 http://www.gutenberg.org/上。我们先要用 Python 解释器加载 NLTK 包，然后尝试 nltk.corpus.gutenberg.fileids()，下面是这个语料库中的文件标识符：

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

让我们挑选这些文本的第一个——简·奥斯丁的《爱玛》——并给它一个简短的名称 emma，然后找出它包含多少个词：

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

```
>>> len(emma)
192427
```

在 1.1 节中，我们讲述了如何使用 text1.concordance()命令对像 text1 这样的文本进行索引。然而，这是假设你正在使用由 from nltk.book import * 导入的 9 个文本之一。现在你开始研究 nltk.corpus 中的数据，像前面的例子一样，你必须采用以下语句对来处理索引和 1.1 节中的其它任务。

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

在我们定义的 emma 时，我们调用了 NLTK 中的 corpus 包中的 gutenberg 对象的 words()函数。但因为总是要输入这么长的名字很繁琐，Python 提供了另一个版本的 import 语句，示例如下：

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

让我们写一个简短的程序，通过循环遍历前面列出的 gutenberg 文件标识符链表相应的 fileid，然后计算统计每个文本。为了使输出看起来紧凑，我们使用函数 int()来确保数字都是整数。

```
>>> for fileid in gutenberg.fileids():
...     num_chars = len(gutenberg.raw(fileid)) ①
...     num_words = len(gutenberg.words(fileid))
...     num_sents = len(gutenberg.sents(fileid))
...     num_vocab = len(set([w.lower() for w in gutenberg.words(fileid)]))
...     print int(num_chars/num_words), int(num_words/num_sents), int(num_words/num_vocab), fileid
...
4 21 26 austen-emma.txt
4 23 16 austen-persuasion.txt
4 24 22 austen-sense.txt
4 33 79 bible-kjv.txt
4 18 5 blake-poems.txt
4 17 14 bryant-stories.txt
4 17 12 burgess-busterbrown.txt
4 16 12 carroll-alice.txt
4 17 11 chesterton-ball.txt
4 19 11 chesterton-brown.txt
4 16 10 chesterton-thursday.txt
4 18 24 edgeworth-parents.txt
4 24 15 melville-moby_dick.txt
4 52 10 milton-paradise.txt
4 12 8 shakespeare-caesar.txt
4 13 7 shakespeare-hamlet.txt
4 13 6 shakespeare-macbeth.txt
4 35 12 whitman-leaves.txt
```

这个程序显示每个文本的三个统计量：平均词长、平均句子长度和本文中每个词出现的平均次数（我们的词汇多样性得分）。请看，平均词长似乎是英语的一个一般属性，因为它的值总是 4。（事实上，平均词长是 3 而不是 4，因为 num_chars 变量计数了空白字符。）相比之下，平均句子长度和词汇多样性看上去是作者个人的特点。

前面的例子也表明我们怎样才能获取“原始”文本①而不用把它分割成标识符。raw()函数给我们没有进行过任何语言学处理的文件的内容。因此，例如：len(gutenberg.raw('blake-poems.txt')告诉我们文本中出现的词汇个数，包括词之间的空格。sents()函数把文本划分成句子，其中每一个句子是一个词链表。

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

除了 words()、raw()和 sents()以外，大多数 NLTK 语料库阅读器还包括多种访问方法。一些语料库提供更加丰富的语言学内容，例如：词性标注，对话标记，句法树等。在后面的章节中，我们将看到这些。

网络和聊天文本

虽然古腾堡项目包含成千上万的书籍，它代表既定的文学。考虑较不正式的语言也是很重要的。NLTK 的网络文本小集合的内容包括 Firefox 交流论坛，在纽约无意听到的对话，《加勒比海盗》的电影剧本，个人广告和葡萄酒的评论：

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

还有一个即时消息聊天会话语料库，最初由美国海军研究生院为研究自动检测互联网幼童虐待癖而收集的。语料库包含超过 10,000 张帖子，以“UserNNN”形式的通用名替换掉用户名，手工编辑消除任何其他身份信息，制作而成。语料库被分成 15 个文件，每个文件

包含几百个按特定日期和特定年龄的聊天室（青少年、20 岁、30 岁、40 岁、再加上一个通用的成年人聊天室）收集的帖子。文件名中包含日期、聊天室和帖子数量，例如：10-19-20s_706posts.xml 包含 2006 年 10 月 19 日从 20 多岁聊天室收集的 706 个帖子。

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

布朗语料库

布朗语料库是第一个百万词级的英语电子语料库的，由布朗大学于 1961 年创建。这个语料库包含 500 个不同来源的文本，按照文体分类，如：新闻、社论等。表 2-1 给出了各个文体的例子（完整列表，请参阅 http://icame.uib.no/brown/bcm-los.html）。

表 2-1. 布朗语料库每一部分的示例文档

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

我们可以将语料库作为词链表或者句子链表来访问（每个句子本身也是一个词链表）。我们可以指定特定的类别或文件阅读：

```
>>> from nltk.corpus import brown
>>> brown.categories()
['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies',
'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance',
'science_fiction']
```

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

布朗语料库是一个研究文体之间的系统性差异——一种叫做**文体学**的语言学研究——很方便的资源。让我们来比较不同文体中的情态动词的用法。第一步：产生特定文体的计数。记住做下面的实验之前要 import nltk：

```
>>> from nltk.corpus import brown
>>> news_text = brown.words(categories='news')
>>> fdist = nltk.FreqDist([w.lower() for w in news_text])
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> for m in modals:
...     print m + ':', fdist[m],
...
can: 94 could: 87 may: 93 might: 38 must: 53 will: 389
```

轮到你来：选择布朗语料库的不同部分，修改前面的例子，计数包含 wh 的词，如：*what*，*when*，*where*，*who* 和 *why*。

下面，我们来统计每一个感兴趣的文体。我们使用 NLTK 提供的带条件的频率分布函数。在 2.2 节中会系统的把下面的代码一行行拆开来讲解。现在，你可以忽略细节，只看输出。

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

请看：新闻文体中最常见的情态动词是 will，而言情文体中最常见的情态动词是 could。你能预言这些吗？这种可以区分文体的词计数方法将在第 6 章中再次谈及。

路透社语料库

路透社语料库包含 10,788 个新闻文档，共计 130 万字。这些文档分成 90 个主题，按照“训练”和“测试”分为两组。因此，fileid 为“test/14826”的文档属于测试组。这样分割是为了训练和测试算法的，这种算法自动检测文档的主题，我们将在第 6 章中看到。

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

与布朗语料库不同，路透社语料库的类别是有互相重叠的，只是因为新闻报道往往涉及多个主题。我们可以查找由一个或多个文档涵盖的主题，也可以查找包含在一个或多个类别中的文档。为方便起见，语料库方法既接受单个的 fileid 也接受 fileids 列表作为参数。

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

类似的，我们可以以文档或类别为单位查找我们想要的词或句子。这些文本中最开始的几个词是标题，按照惯例以大写字母存储。

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

就职演说语料库

1.1 节中，我们看到了就职演说语料库，但是把它当作一个单独的文本对待。在图 1-2 中使用的“词偏移”就像是一个坐标轴，它是语料库中词的索引数，从第一个演讲的第一个词开始算起。然而，语料库实际上是 55 个文本的集合，每个文本都是一个总统的演说。这个集合的一个有趣特性是它的时间维度：

```
>>> from nltk.corpus import inaugural
```

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

请注意，每个文本的年代都出现在它的文件名中。要从文件名中获得年代，我们使用 fileid[:4]提取前四个字符。

让我们来看看词汇 *america* 和 *citizen* 随时间推移的使用情况。下面的代码使用 w.lower()①将就职演说语料库中的词汇转换成小写。然后用 startswith()①检查它们是否以“目标”词汇 america 或 citizen 开始。因此，它会计算如 *American's* 和 *Citizens* 等词。我们将在 2.2 节学习条件频率分布，现在只考虑输出，如图 2-1 所示。

```
>>> cfd = nltk.ConditionalFreqDist(
...           (target, file[:4])
...           for fileid in inaugural.fileids()
...           for w in inaugural.words(fileid)
...           for target in ['america', 'citizen']
...           if w.lower().startswith(target)) ①
>>> cfd.plot()
```

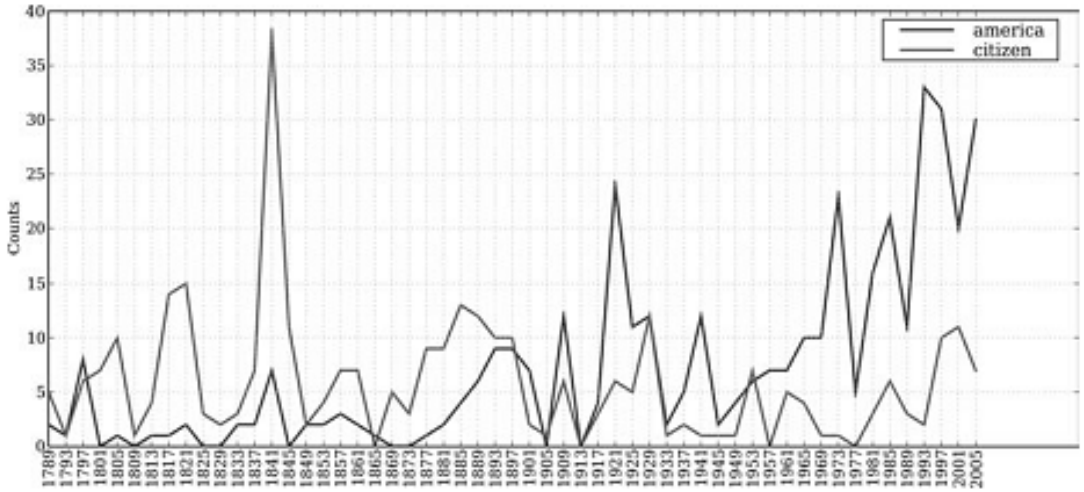


图 2-1. 条件频率分布图：计数就职演说语料库中所有以 america 或 citizen 开始的词。每个演讲单独计数。这样就能观察出随时间变化用法上的演变趋势。计数没有与文档长度进行归一化处理。

标注文本语料库

许多文本语料库都包含语言学标注，有词性标注、命名实体、句法结构、语义角色等。NLTK 中提供了很方便的方式来访问这些语料库中的几个，还有一个包含语料库和语料样本的数据包，用于教学和科研的话可以免费下载。表 2-2 列出了其中一些语料库。有关下载信息请参阅 http://www.nltk.org/data。关于如何访问 NLTK 语料库的其它例子，请在 http://www.nltk.org/howto 查阅语料库的 HOWTO。

表 2-2. NLTK 中的一些语料库和语料库样本

语料库	编译器	内容

Brown Corpus	Francis, Kucera	15 genres, 1.15M words, tagged, categorized
CESS Treebanks	CLiC-UB	1M words, tagged and parsed (Catalan, Spanish)
Chat-80 Data Files	Pereira & Warren	World Geographic Database
CMU Pronouncing Dictionary	CMU	127k entries
CoNLL 2000 Chunking Data	CoNLL	270k words, tagged and chunked
CoNLL 2002 Named Entity	CoNLL	700k words, pos- and named-entity-tagged (Dutch, Spanish)
CoNLL 2007 Dependency Treebanks (sel)	CoNLL	150k words, dependency parsed (Basque, Catalan)
Dependency Treebank	Narad	Dependency parsed version of Penn Treebank sample
Floresta Treebank	Diana Santos et al	9k sentences, tagged and parsed (Portuguese)
Gazetteer Lists	Various	Lists of cities and countries
Genesis Corpus	Misc web sources	6 texts, 200k words, 6 languages
Gutenberg (selections)	Hart, Newby, et al	18 texts, 2M words
Inaugural Address Corpus	CSpan	US Presidential Inaugural Addresses (1789-present)
Indian POS-Tagged Corpus	Kumaran et al	60k words, tagged (Bangla, Hindi, Marathi, Telugu)
MacMorpho Corpus	NILC, USP, Brazil	1M words, tagged (Brazilian Portuguese)
Movie Reviews	Pang, Lee	2k movie reviews with sentiment polarity classification
Names Corpus	Kantrowitz, Ross	8k male and female names
NIST 1999 Info Extr (selections)	Garofolo	63k words, newswire and named-entity SGML markup
NPS Chat Corpus	Forsyth, Martell	10k IM chat posts, POS-tagged and dialogue-act tagged
PP Attachment Corpus	Ratnaparkhi	28k prepositional phrases, tagged as noun or verb modifiers
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	Pedersen	600k words, part-of-speech and sense tagged
Shakespeare texts (selections)	Bosak	8 books in XML format
State of the Union Corpus	CSPAN	485k words, formatted text
Stopwords Corpus	Porter et al	2,400 stopwords for 11 languages
Swadesh Corpus	Wiktionary	comparative wordlists in 24 languages
Switchboard Corpus (selections)	LDC	36 phonecalls, transcribed, parsed
Univ Decl of Human Rights	United Nations	480k words, 300+ languages
Penn Treebank (selections)	LDC	40k words, tagged and parsed
TIMIT Corpus (selections)	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

在其他语言的语料库

NLTK 包含多国语言语料库。某些情况下你在使用这些语料库之前需要学习如何在 Python 中处理字符编码（见 3.3 节）。

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

这些语料库的最后，udhr，是超过 300 种语言的世界人权宣言。这个语料库的 fileids

包括有关文件所使用的字符编码，如：UTF8 或者 Latin1。让我们用条件频率分布来研究“世界人权宣言”（udhr）语料库中不同语言版本中的字长差异。图 2-2 中所示的输出（自己运行程序可以看到一个彩色图）。注意：True 和 False 是 Python 内置的布尔值。

```
>>> 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)
```

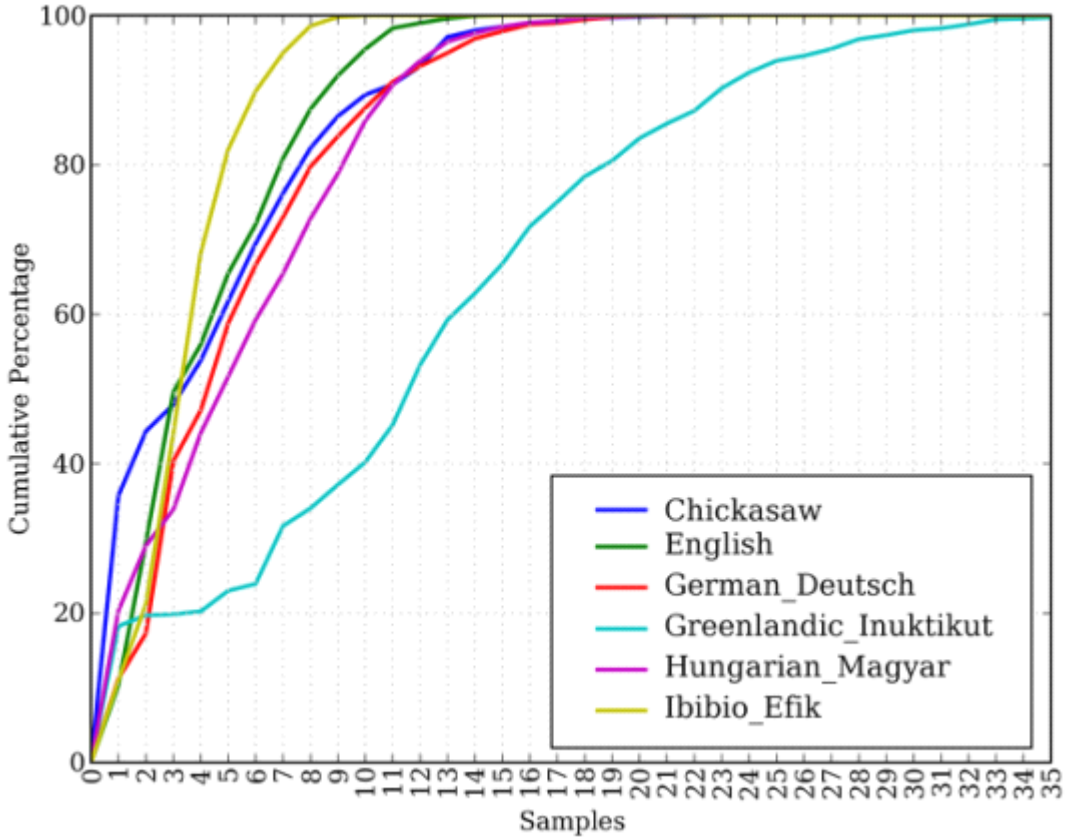


图 2-2. 累积字长分布：内容是“世界人权宣言”的 6 个翻译版本；此图显示：5 个或 5 个以下字母组成的词在 Ibibio 语言的文本中占约 80%，在德语文本中占 60%，在 Inuktitut 文本中占 25%。

轮到你来：在 udhr.fileids()中选择一种感兴趣的语言，定义一个变量 raw_text= udhr.raw(+)。使用 nltk.FreqDist(raw_text).plot()画出此文本的字母频率分布图。

不幸的是，许多语言没有大量的语料库。通常是政府或工业对发展语言资源的支持不够。个人的努力是零碎的，难以发现或重用。有些语言没有既定的书写系统，或濒临灭绝。（见 2.7 节有关如何寻找语言资源的建议。）

文本语料库的结构

到目前为止，我们已经看到了大量的语料库结构。图 2-3 中总结了它们。最简单的一种没有任何结构，仅仅是一个文本集合。通常，文本会按照其可能对应的文体、来源、作者、语言等分类。有时，这些类别会重叠，尤其是在按主题分类的情况下，因为一个文本可能与多个主题相关。偶尔的，文本集有一个时间结构，新闻集合是最常见的例子。

NLTK 语料库阅读器支持高效的访问大量语料库，并且能用于处理新的语料库。表 2-3 列出了语料库阅读器提供的函数。

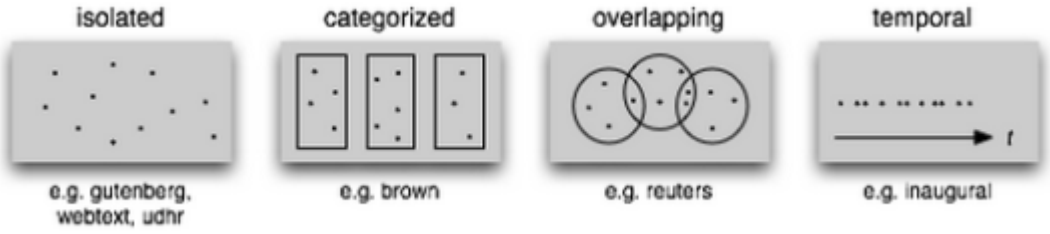


图 2-3. 文本语料库的常见结构：最简单的一种语料库是一些孤立的没有什么特别的组织的文本集合；一些语料库按如文体（布朗语料库）等分类组织结构；一些分类会重叠，如主题类别（路透社语料库）；另外一些语料库可以表示随时间变化语言用法的改变（就职演说语料库）。

表 2-3. NLTK 中定义的基本语料库函数：使用 help(nltk.corpus.reader)可以找到更多的文档，也可以阅读 http://www.nltk.org/howto 上的在线语料库的 HOWTO。

示例	描述
fileids()	语料库中的文件
fileids([categories])	这些分类对应的语料库中的文件
categories()	语料库中的分类
categories([fileids])	这些文件对应的语料库中的分类
raw()	语料库的原始内容
raw(fileids=[f1,f2,f3])	指定文件的原始内容
raw(categories=[c1,c2])	指定分类的原始内容
words()	整个语料库中的词汇
words(fileids=[f1,f2,f3])	指定文件中的词汇
words(categories=[c1,c2])	指定分类中的词汇
sents()	指定分类中的句子
sents(fileids=[f1,f2,f3])	指定文件中的句子
sents(categories=[c1,c2])	指定分类中的句子
abspath(fileid)	指定文件在磁盘上的位置
encoding(fileid)	文件的编码（如果知道的话）
open(fileid)	打开指定语料库文件的文件流
root()	到本地安装的语料库根目录的路径

在这里我们说明一些语料库访问方法之间的区别：

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

载入你自己的语料库

如果你有自己收集的文本文件，并且想使用前面讨论的方法访问它们，你可以很容易地在 NLTK 中的 PlaintextCorpusReader 帮助下载入它们。检查你的文件在文件系统中的位置；在下面的例子中，我们假定你的文件在/usr/share/dict 目录下。不管是什么位置，将变量 corpus_root①的值设置为这个目录。PlaintextCorpusReader 初始化函数②的第二个参数可以是一个如['a.txt', 'test/b.txt']这样的 fileids 链表，或者一个匹配所有 fileids 的模式，如：'[abc]/.*\.txt'（关于正则表达式的信息见 3.4 节）。

```
>>> from nltk.corpus import PlaintextCorpusReader
>>> corpus_root = '/usr/share/dict' ①
>>> wordlists = PlaintextCorpusReader(corpus_root, '.*') ②
>>> wordlists.fileids()
['README', 'connectives', 'propernames', 'web2', 'web2a', 'words']
>>> wordlists.words('connectives')
['the', 'of', 'and', 'to', 'a', 'in', 'that', 'is', ...]
```

下面是另一个例子，假设你在本地硬盘上有自己的宾州树库（第 3 版）的拷贝，放在 C:\corpora。我们可以使用 BracketParseCorpusReader 访问这些语料。我们指定 corpus_root 为存放语料库中解析过的《华尔街日报》部分①的位置，并指定 file_pattern 与它的子文件夹中包含的文件匹配②（用前斜杠）。

```
>>> from nltk.corpus import BracketParseCorpusReader
>>> corpus_root = r"C:\corpora\penntreebank\parsed\mrg\wsj" ①
>>> file_pattern = r".*/wsj_.*\.mrg" ②
>>> ptb = BracketParseCorpusReader(corpus_root, file_pattern)
>>> ptb.fileids()
['00/wsj_0001.mrg', '00/wsj_0002.mrg', '00/wsj_0003.mrg', '00/wsj_0004.mrg', ...]
>>> len(ptb.sents())
49208
```

```
>>> ptb.sents(fileids='20/wsj_2013.mrg')[19]
['The', '55-year-old', 'Mr.', 'Noriega', 'is', "n't", 'as', 'smooth', 'as', 'the',
'shah', 'of', 'Iran', ',', 'as', 'well-born', 'as', 'Nicaragua', "'s", 'Anastasio',
'Somoza', ',', 'as', 'imperial', 'as', 'Ferdinand', 'Marcos', 'of', 'the', 'Philippines',
'or', 'as', 'bloody', 'as', 'Haiti', "'s", 'Baby', 'Doc', 'Duvalier', '.']
```

2.2 条件频率分布

我们在 1.3 节介绍了频率分布。我们看到给定某个词汇或其他项目的链表变量 mylist，FreqDist(mylist)会计算链表中每个项目出现的次数。在这里，我们将推广这一想法。

当语料文本被分为几类（文体、主题、作者等）时，我们可以计算每个类别独立的频率分布。这将允许我们研究类别之间的系统性差异。在上一节中，我们是用 NLTK 的 ConditionalFreqDist 数据类型实现的。**条件频率分布**是频率分布的集合，每个频率分布有一个不同的“条件”。这个条件通常是文本的类别。图 2-4 描绘了一个带两个条件的条件频率分布的片段，一个是新闻文本，一个是言情文本。

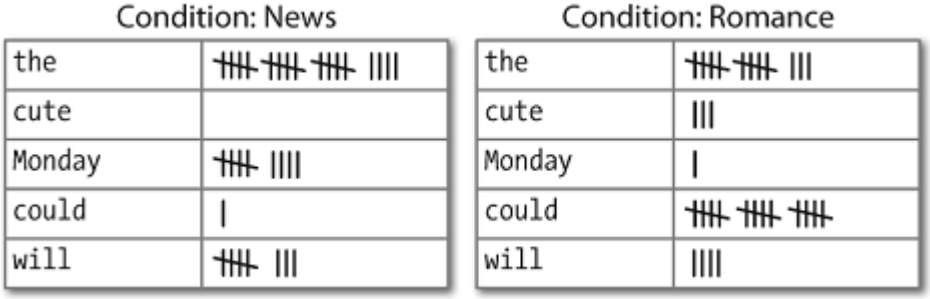


图 2-4. 计数文本集合中单词出现次数（条件频率分布）。

条件和事件

频率分布计算观察到的事件，如文本中出现的词汇。条件频率分布需要给每个时间关联一个条件，所以不是处理一个词序列①，我们必须处理的是一个配对序列②。

```
>>> text = ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...] ①
>>> pairs = [('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ...] ②
```

每对的形式是：（条件，事件）。如果我们按文体处理整个布朗语料库，将有 15 个条件（每个文体一个条件）和 1,161,192 个事件（每一个词一个事件）。

按文体计数词汇

2.1 节中，我们看到一个以布朗语料库的每一部分为条件的条件频率分布，并按照每个条件计数词汇。FreqDist()以一个简单的链表作为输入，ConditionalFreqDist()以一个配对链表作为输入。

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

```
...             for genre in brown.categories()
...             for word in brown.words(categories=genre))
```

让我们拆开来看，只看两个文体：新闻和言情。对于每个文体②，我们遍历文体中的每个词③以产生文体与词的配对① 。

```
>>> genre_word = [(genre, word) ①
...                  for genre in ['news', 'romance'] ②
...                  for word in brown.words(categories=genre)] ③
>>> len(genre_word)
170576
```

因此，在下面的代码中我们可以看到：链表 genre_word 的前几个配对将是('news', *word*)①的形式，而最后几个配对将是('romance', *word*)②的形式。

```
>>> genre_word[:4]
[('news', 'The'), ('news', 'Fulton'), ('news', 'County'), ('news', 'Grand')] ①
>>> genre_word[-4:]
[('romance', 'afraid'), ('romance', 'not'), ('romance', '"'), ('romance', '.')] ②
```

现在，我们可以使用此配对链表创建一个 ConditionalFreqDist，并将它保存在一个变量 cfd 中。像往常一样，我们可以输入变量的名称来检查它①，并确认它有两个条件②：

```
>>> cfd = nltk.ConditionalFreqDist(genre_word)
>>> cfd ①
<ConditionalFreqDist with 2 conditions>
>>> cfd.conditions()
['news', 'romance'] ②
```

让我们访问这两个条件，它们每一个都只是一个频率分布：

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

绘制分布图和分布表

除了组合两个或两个以上的频率分布和更容易初始化之外，ConditionalFreqDist 还为制表和绘图提供了一些有用的方法。

图 2-1 是基于下面的代码产生的一个条件频率分布绘制的。条件是词 america 或 citizen②，被绘图的计数是在特定演讲中出现的词的次数。它利用了每个演讲的文件名——例如 1865-Lincoln.txt——的前 4 个字符包含年代的事实①。这段代码为文件 1865-Lincoln.txt 中每个小写形式以 america 开头的词——如：Americans——产生一个配对('america', '1865')。

```
>>> from nltk.corpus import inaugural
>>> cfd = nltk.ConditionalFreqDist(
```

```
...                 (target, fileid[:4]) ①
...                 for fileid in inaugural.fileids()
...                 for w in inaugural.words(fileid)
...                 for target in ['america', 'citizen'] ②
...                 if w.lower().startswith(target))
```

图 2-2 也是基于下面的代码产生的一个条件频率分布绘制的。这次的条件是语言的名称，图中的计数来源于词长①。它利用了每一种语言的文件名是语言名称后面跟'-Latin1'（字符编码）的事实。

```
>>> from nltk.corpus import udhr
>>> languages = ['Chickasaw', 'English', 'German_Deutsch',
...     'Greenlandic_Inuktikut', 'Hungarian_Magyar', 'Ibibio_Efik']
>>> cfd = nltk.ConditionalFreqDist(
...                 (lang, len(word)) ①
...                 for lang in languages
...                 for word in udhr.words(lang + '-Latin1'))
```

在 plot()和 tabulate()方法中，我们可以使用 conditions= parameter 来选择指定哪些条件显示。如果我们忽略它，所有条件都会显示。同样，我们可以使用 samples= parameter 来限制要显示的样本。这使得载入大量数据到一个条件频率分布，然后通过选定条件和样品，绘图或制表的探索成为可能。这也使我们能全面控制条件和样本的显示顺序。例如：我们可以为两种语言和长度少于 10 个字符的词汇绘制累计频率数据表，如下所示。我们解释一下上排最后一个单元格中数值的含义是英文文本中 9 个或少于 9 个字符长的词有 1,638 个。

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

轮到你来：处理布朗语料库的新闻和言情文体，找出一周中最有新闻价值并且是最浪漫的日子。定义一个变量 days 包含星期的链表，如['Monday', ...]。然后使用 cfd.tabulate(samples=days)为这些词的计数制表。接下来用绘图替代制表尝试同样的事情。你可以在额外的参数 conditions=['Monday', ...]的帮助下控制星期输出的顺序。

你可能已经注意到：我们已经在使用的条件频率分布看上去像链表推导，但是不带方括号。通常，我们使用链表推导作为一个函数的参数，如：set([w.lower for w in t])，忽略掉方括号而只写 set(w.lower() for w in t)是允许的。（更多的讲解请参见 4.2 节“生成器表达式”的讨论。）

使用双连词生成随机文本

我们可以使用条件频率分布创建一个双连词表（词对，在 1.3 节介绍过）。bigrams()函数接受一个词汇链表，并建立一个连续的词对链表。

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

在例 2-1 中，我们把每个词作为一个条件，对每个词我们有效的创建它的后续词的频率分布。函数 generate_model()包含一个简单的循环来生成文本。当我们调用这个函数时，我们选择一个词（如“living”）作为我们的初始内容。然后，进入循环后，我们输入变量 word 的当前值，重新设置 word 为上下文中最可能的标识符（使用 max()）。下一次进入循环，我们使用那个词作为新的初始内容。正如你通过检查输出可以看到的，这种简单的文本生成方法往往会在循环中卡住。另一种方法是从可用的词汇中随机选择下一个词。

*例 2-1. 产生随机文本：此程序获得《创世记》文本中所有的双连词，然后构造一个条件频率分布来记录哪些词汇最有可能跟在给定词的后面；例如：living 后面最可能的词是 creature；*generate_model()*函数使用这些数据和种子词随机产生文本。*

```
def generate_model(cfdist, word, num=15):
    for i in range(num):
        print word,
        word = cfdist[word].max()
text = nltk.corpus.genesis.words('english-kjv.txt')
bigrams = nltk.bigrams(text)
cfd = nltk.ConditionalFreqDist(bigrams) ①

>>> 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
```

条件频率分布是一个对许多 NLP 任务都有用的数据结构。表 2-4 总结了它们常用的方法。

表 2-4. NLTK 中的条件频率分布：定义、访问和可视化一个计数的条件频率分布的常用方法和习惯用法

示例	描述
cfdist= ConditionalFreqDist(pairs)	从配对链表中创建条件频率分布
cfdist.conditions()	将条件按字母排序
cfdist[condition]	此条件下的频率分布
cfdist[condition][sample]	此条件下给定样本的频率
cfdist.tabulate()	为条件频率分布制表
cfdist.tabulate(samples, conditions)	指定样本和条件限制下制表
cfdist.plot()	为条件频率分布绘图
cfdist.plot(samples, conditions)	指定样本和条件限制下绘图
cfdist1 < cfdist2	测试样本在 cfdist1 中出现次数是否小于在 cfdist2 中出现次数

2.3 更多关于 Python：代码重用

到现在，你可能已经在 Python 交互式解释器中输入和重新输入了大量的代码。如果你在重新输入一个复杂的例子时弄得一团糟，你就必须重新输入。使用箭头键来访问和修改以前的命令是有用的，但也很有限。在本节中，我们将讲述代码重用的两个重要方式：文本编辑器和 Python 函数。

使用文本编辑器创建程序

Python 交互式解释器中你一输入命令就执行。通常，使用文本编辑器组织多行程序，然后让 Python 一次运行整个程序会更好。使用 IDLE，你可以通过“文件”菜单打开一个新窗口来做到这些。现在就尝试一下，输入下面的一行程序：

```
print 'Monty Python'
```

将这段程序保存为文件，取名为 monty.py。然后打开“运行”菜单，选择运行模块命令。（我们将很快学习到什么是模块。） IDLE 主窗口的结果应该像这样：

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

你也可以输入 `from monty import *` ，它将做同样的事情。

从现在起，你可以选择使用交互式解释器或文本编辑器来创建你的程序。使用解释器测试你的想法往往比较方便，修改一行代码直到达到你期望的效果。测试好之后，你就可以将代码粘贴到文本编辑器（去除所有>>>和...提示符），继续扩展它。最后将程序保存为文件这样你以后就不用再重新输入了。给文件一个小而准确的名字，使用所有的小写字母，用下划线分割词汇，使用.py 文件名后缀，例如：monty_python.py。

要点：我们的内联代码的例子包含>>>和...提示符，好像我们正在直接与解释器交互。随着程序变得更加复杂，你应该在编辑器中输入它们，没有提示符，如前面所示的那样在编辑器中运行它们。当我们在这本书中提供更长的程序时，我们将不使用提示符以提醒你在文件中输入它而不是使用解释器。你可以看到示例 2-1 已经这样了。请注意，这个例子还包括两行代码带有 Python 提示符。这是任务的互动部分，在这里你观察一些数据，并调用一个函数。请记住，像例 2-1 这样的所有示例代码都可以从 http://www.nltk.org/下载。

函数

假设你正在分析一些文本，这些文本包含同一个词的不同形式，你的一部分程序需要将给定的单数名词变成复数形式。假设需要在两个地方做这样的事，一个是处理一些文本，另一个是处理用户的输入。

比起重复相同的代码好几次，把这些事情放在一个**函数**中会更有效和可靠。一个函数是命名的代码块，执行一些明确的任务，就像我们在 1.1 节中所看到的那样。一个函数通常被定义来使用一些称为**参数**的变量接受一些输入，并且它可能会产生一些结果，也称为**返回值**。我们使用

关键字 def 加函数名以及所有输入参数来定义一个函数，接下来是函数的主体。这里是我们在第 1.1 节看到的函数（包括使除法像我们期望的那样运算的 import 语句）：

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

我们使用关键字 return 表示函数作为输出而产生的值。在这个例子中，函数所有的工作都在 return 语句中完成。下面是一个等价的定义，使用多行代码做同样的事。我们将把参数名称从 text 变为 my_text_data，注意这只是一个任意的选择：

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

请注意，我们已经在函数体内部创造了一些新的变量。这些是**局部变量**，不能在函数体外访问。现在我们已经定义一个名为 lexical_diversity 的函数。但只定义它不会产生任何输出！函数在被“调用”之前不会做任何事情。

让我们回到前面的场景，实际定义一个简单的函数来处理英文的复数词。例 2-2 中的函数 plural()接受单数名词，产生一个复数形式，虽然它并不总是正确的。（我们将在 4.4 节中以更长的篇幅讨论这个函数。）

例 2-2. 一个 Python 函数：这个函数试图生成任何英语名词的复数形式。关键字 def（define）后面跟着函数的名称，然后是包含在括号内的参数和一个冒号；函数体是缩进代码块；它试图识别词内的模式，相应的对词进行处理；例如:如果这个词以 y 结尾，删除它们并添加 ies。

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

endswith()函数总是与一个字符串对象一起使用（如：例 2-2 中的 word）。要调用此函数，我们使用对象的名字，一个点，然后跟函数的名称。这些函数通常被称为**方法**。

模块

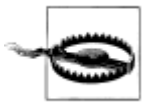
随着时间的推移，你将会发现你创建了大量小而有用的文字处理函数，结果你不停的把它们从老程序复制到新程序中。哪个文件中包含的才是你要使用的函数的最新版本？如果你能把你的劳动成果收集在一个单独的地方，而且访问以前定义的函数不必复制，生活将会更加轻松。

要做到这一点，请将你的函数保存到一个文件：textproc.py。现在，你可以简单的通过从文件导入它来访问你的函数：

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

显然，我们的复数函数明显存在错误，因为 fan 的复数是 fans。不必再重新输入这个函数的新版本，我们可以简单的编辑现有的。因此，在任何时候我们的复数函数只有一个版本，不会再有使用哪个版本的困扰。

在一个文件中的变量和函数定义的集合被称为一个 Python **模块**（module）。相关模块的集合称为一个**包**（package）。处理布朗语料库的 NLTK 代码是一个模块，处理各种不同的语料库的代码的集合是一个包。NLTK 的本身是包的集合，有时被称为一个**库**（library）。



注意！

如果你正在创建一个包含一些你自己的 Python 代码的文件，一定不要将文件命名为 nltk.py：这可能会在导入时占据“真正的”NLTK 包。当 Python 导入模块时，它先查找当前目录（文件夹）。

2.4 词典资源

词典或者词典资源是一个词和/或短语以及一些相关信息的集合，例如：词性和词意定义等相关信息。词典资源附属于文本，通常在文本的帮助下创建和丰富。例如：如果我们定义了一个文本 my_text，然后 vocab = sorted(set(my_text))建立 my_text 的词汇表，同时 word_freq = FreqDist(my_text)计数文本中每个词的频率。vocab 和 word_freq 都是简单的词汇资源。同样，如我们在 1.1 节中看到的，词汇索引为我们提供了有关词语用法的信息，可能在编写词典时有用。图 2-5 中描述了词汇相关的标准术语。一个**词项**包括**词目**（也叫**词条**）以及其他附加信息，例如：词性和词意定义。两个不同的词拼写相同被称为**同音异义词**。

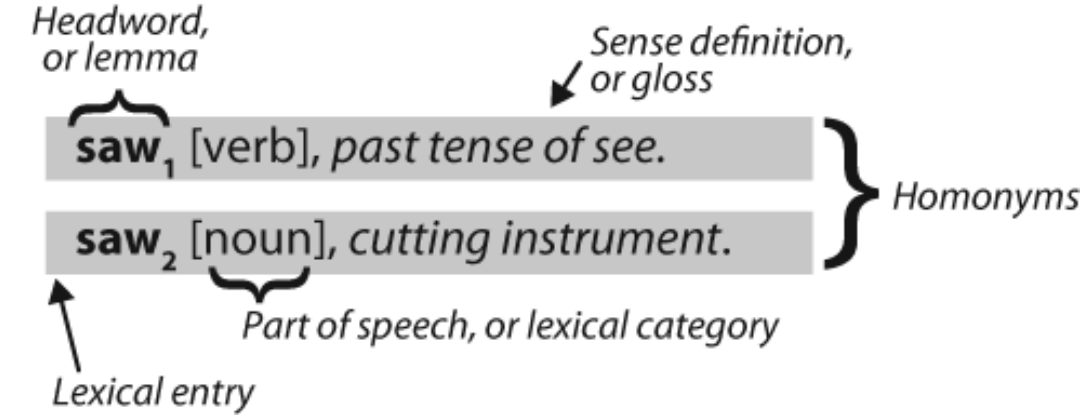


图 2-5. 词典术语：两个拼写相同的词条（同音异义词）的词汇项，包括词性和注释信息。

一种简单的词典资源是除了一个词汇列表外什么也没有。复杂的词典资源包括在词汇项内和跨词汇项的复杂的结构。在本节，我们来看看 NLTK 中的一些词典资源。

词汇列表语料库

NLTK 包括一些仅仅包含词汇列表的语料库。词汇语料库是 Unix 中的/usr/dict/words 文件，被一些拼写检查程序使用。我们可以用它来寻找文本语料中不寻常的或拼写错误的词汇，如例 2-3 所示。

例 2-3. 过滤文本：此程序计算文本的词汇表，然后删除所有在现有的词汇列表中出现的元素，只留下罕见或拼写错误的词。

```
def unusual_words(text):
    text_vocab = set(w.lower() for w in text if w.isalpha())
    english_vocab = set(w.lower() for w in nltk.corpus.words.words())
    unusual = text_vocab.difference(english_vocab)
    return sorted(unusual)
>>> unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))
['abbeyland', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary',
'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham',
'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]
>>> unusual_words(nltk.corpus.nps_chat.words())
['aaaaaaaaaaaaaaaa', 'aaahhhh', 'abou', 'abourted', 'abs', 'ack', 'acros',
'actualy', 'adduser', 'addy', 'adoted', 'adreniline', 'ae', 'afe', 'affari', 'afk',
'agaibn', 'agurlwithbigguns', 'ahah', 'ahahah', 'ahahh', 'ahahha', 'ahem', 'ahh', ...]
```

还有一个**停用词**语料库，就是那些高频词汇，如：the，to，我们有时在进一步的处理之前想要将它们从文档中过滤。停用词通常几乎没有什么词汇内容，而它们的出现会使区分文本变困难。

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

让我们定义一个函数来计算文本中没有在停用词列表中的词的比例。

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

因此，在停用词的帮助下，我们筛选掉文本中三分之一的词。请注意，我们在这里结合了两种不同类型的语料库，使用词典资源来过滤文本语料的内容。

How many words of four letters or more can you make from those shown here? Each letter may be used once per word. Each word must contain the center letter and there must be at least one nine-letter word. No plurals ending in "s"; no foreign words; no proper names. 21 words, good; 32 words, very good; 42 words, excellent.

图 2-6. 一个字母拼词谜题：在由随机选择的字母组成的网格中，选择里面的字母组成词。这个谜题叫做“目标”。图中文字的意思是：用这里显示的字母你能组成多少个 4 字母或者更多字母的词？每个字母在每个词中只能被用一次。每个词必须包括中间的字母并且必须至少有一个 9 字母的词。没有复数以“s”结尾；没有外来词；没有姓名。能组出 21 个词就是“好”；32 个词，“很好”；42 个词，“非常好”。

一个词汇列表对解决如图 2-6 中这样的词的谜题很有用。我们的程序遍历每一个词，对于每一个词检查是否符合条件。检查必须出现的字母②和长度限制①是很容易的（这里我们只查找 6 个或 6 个以上字母的词）。只使用指定的字母组合作为候选方案，尤其是一些指定的字母出现了两次（这里如字母 v）这样的检查是很棘手的。FreqDist 比较法③允许我们检查每个字母在候选词中的频率是否小于或等于相应的字母在拼词谜题中的频率。

```
>>> puzzle_letters = nltk.FreqDist('egivrvonl')
>>> obligatory = 'r'
>>> wordlist = nltk.corpus.words.words()
>>> [w for w in wordlist if len(w) >= 6 ①
...                      and obligatory in w ②
...                      and nltk.FreqDist(w) <= puzzle_letters] ③
['glover', 'gorlin', 'govern', 'grovel', 'ignore', 'involver', 'lienor',
'linger', 'longer', 'lovering', 'noiler', 'overling', 'region', 'renvoi',
'revolving', 'ringle', 'roving', 'violer', 'virole']
```

另一个词汇列表是名字语料库，包括 8000 个按性别分类的名字。男性和女性的名字存储在单独的文件中。让我们找出同时出现在两个文件中的名字即性别暧昧的名字：

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

正如大家都知道的，以字母 a 结尾的名字几乎都是女性。我们可以在图 2-7 中看到这一点以及一些其它的模式，该图是由下面的代码产生的。请记住 name[-1]是 name 的最后一个字母。

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

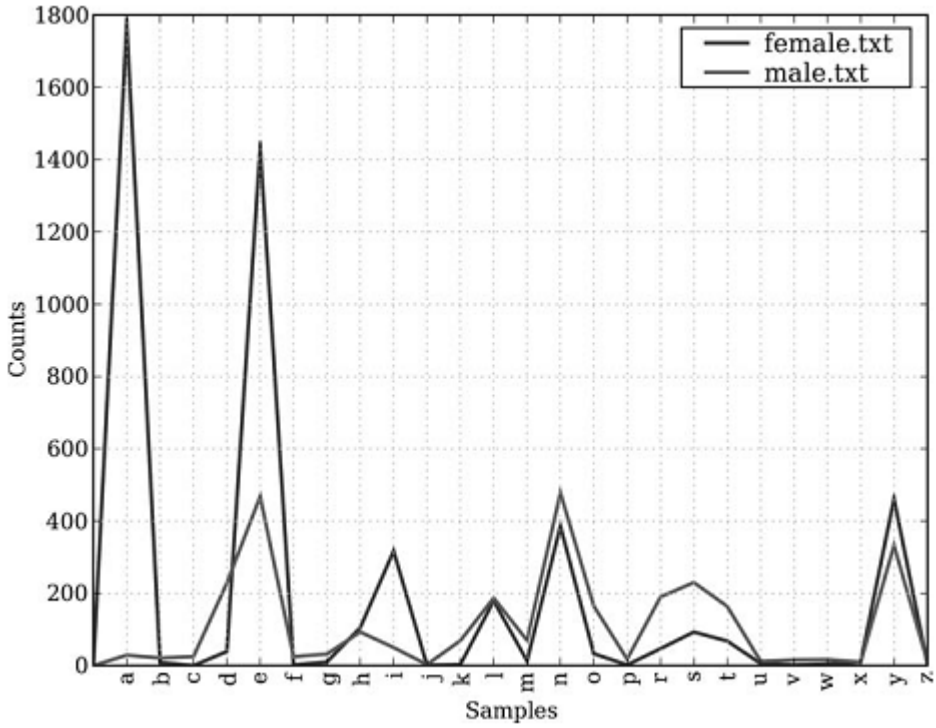


图 2-7. 条件频率分布：此图显示男性和女性名字的结尾字母；大多数以 a，e 或 i 结尾的名字是女性；以 h 和 l 结尾的男性和女性同样多；以 k，o，r，s 和 t 结尾的更可能是男性。

发音的词典

一个稍微丰富的词典资源是一个表格（或电子表格），在每一行中含有一个词加一些性质。NLTK 中包括美国英语的 CMU 发音词典，它是为语音合成器使用而设计的。

```
>>> 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'])
```

对每一个词，这个词典资源提供语音的代码——不同的声音不同的标签——叫做音素。请看 fire 有两个发音（美国英语中）：单音节 `F AY1 R` 和双音节 `F AY1 ER0`。CMU 发音词典中的符号是从 Arpabet 来的，更多的细节请参考 http://en.wikipedia.org/wiki/Arpabet。

每个条目由两部分组成，我们可以用一个复杂的 for 语句来一个一个的处理这些。我们没有写 `for entry in entries:`，而是用两个变量名 `word` 和 `pron` 替换 `entry`①。现在，

每次通过循环时，word 被分配条目的第一部分，pron 被分配条目的第二部分：

```
>>> for word, pron in entries: ①
...     if len(pron) == 3: ②
...         ph1, ph2, ph3 = pron ③
...         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
```

上面的程序扫描词典中那些发音包含三个音素的条目②。如果条件为真，就将 pron 的内容分配给三个新的变量：ph1，ph2 和 ph3。请注意实现这个功能的语句的形式并不多见。

这里是同样的 for 语句的另一个例子，这次使用内部的链表推导。这段程序找到所有发音结尾与 nicks 相似的词汇。你可以使用此方法来找到押韵的词。

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

请注意，有几种方法来拼读一个读音：nics，niks，nix 甚至 ntic's 加一个无声的 t，如词 atlantic's。让我们来看看其他一些发音与书写之间的不匹配。你可以总结一下下面的例子的功能，并解释它们是如何实现的？

```
>>> [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']
```

音素包含数字表示主重音(1)，次重音(2)和无重音(0)。作为我们最后的一个例子，我们定义一个函数来提取重音数字，然后扫描我们的词典，找到具有特定重音模式的词汇。

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

这段程序的精妙之处在于：我们的用户自定义函数 stress()调用一个内含条件的链表推理，还有一个双层嵌套循环。这里有些复杂，等你有了更多的使用链表推导的经验后，你可能会想回过来重新阅读。

我们可以使用条件频率分布来帮助我们找到词汇的最小受限集合。在这里，我们找到所有 p 开头的三音素词②，并按照它们的第一个和最后一个音素来分组①。

```
>>> p3 = [(pron[0]+'-'+pron[2], word) ①
...           for (word, pron) in entries
...           if pron[0] == 'P' and len(pron) == 3] ②
>>> cfd = nltk.ConditionalFreqDist(p3)
>>> for template in cfd.conditions():
...       if len(cfd[template]) > 10:
...             words = cfd[template].keys()
...             wordlist = ' '.join(words)
...             print template, wordlist[:70] + "..."
...
P-CH perch puche poche peach petsche poach pietsch putsch pautsch piche pet...
P-K pik peek pic pique paque polk perc poke perk pac pock poch purk pak pa...
P-L pil poehl pille pehl pol pall pohl pahl paul perl pale paille perle po...
P-N paine payne pon pain pin pawn pinn pun pine paign pen pyne pane penn p...
P-P pap paap pipp paup pape pup pep poop pop pipe paape popp pip peep pope...
P-R paar poor par poore pear pare pour peer pore parr por pair porr pier...
P-S pearse piece posts pasts peace perce pos pers pace puss pesce pass pur...
P-T pot puett pit pete putt pat purt pet peart pott pett pait pert pote pa...
P-Z pays p.s pao's pais paws p.'s pas pez paz pei's pose poise peas paiz p...
```

我们可以通过查找特定词汇来访问它，而不必遍历整个词典。我们将使用 Python 的词典数据结构，在 5.3 节我们将系统的学习它。通过指定词典的名字后面跟一个包含在方括号里的**关键字**（例如：词 fire）来查词典①。

```
>>> prondict = nltk.corpus.cmudict.dict()
>>> prondict['fire'] ①
[['F', 'AY1', 'ER0'], ['F', 'AY1', 'R']]
>>> prondict['blog'] ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'blog'
>>> prondict['blog'] = [['B', 'L', 'AA1', 'G']] ③
>>> prondict['blog']
[['B', 'L', 'AA1', 'G']]
```

如果我们试图查找一个不存在的关键字②，就会得到一个 KeyError。这与我们使用一个过大的整数索引一个链表时产生一个 IndexError 是类似的。词 blog 在发音词典中没有，所以我们对我们自己版本的词典稍作调整，为这个关键字分配一个值③（这对 NLTK 语料库是没有影响的；下一次我们访问它，blog 依然是空的）。

我们可以用任何词典资源来处理文本，如：过滤掉具有某些词典属性的词（如名词），或者映射文本中每一个词。例如：下面的文本到发音函数在发音词典中查找文本中每个词：

```
>>> 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']
```

比较词表

表格词典的另一个例子是**比较词表**。NLTK 中包含了所谓的**斯瓦迪士核心词列表**（Swadesh wordlists），几种语言中约 200 个常用词的列表。语言标识符使用 ISO639 双字母码。

```
>>> from nltk.corpus import swadesh
>>> swadesh.fileids()
['be', 'bg', 'bs', 'ca', 'cs', 'cu', 'de', 'en', 'es', 'fr', 'hr', 'it', 'la', 'mk',
'nl', 'pl', 'pt', 'ro', 'ru', 'sk', 'sl', 'sr', 'sw', 'uk']
>>> swadesh.words('en')
['I', 'you (singular), thou', 'he', 'we', 'you (plural)', 'they', 'this', 'that',
'here', 'there', 'who', 'what', 'where', 'when', 'how', 'not', 'all', 'many', 'some',
'few', 'other', 'one', 'two', 'three', 'four', 'five', 'big', 'long', 'wide', ...]
```

我们可以通过在 entries()方法中指定一个语言链表来访问多语言中的同源词。更进一步，我们可以把它转换成一个简单的词典（我们将在 5.3 节学到 dict()函数）。

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

通过添加其他源语言，我们可以让我们这个简单的翻译器更为有用。让我们使用 dict()函数把德语-英语和西班牙语-英语对相互转换成一个词典，然后用这些添加的映射更新我们原来的翻译词典。

```
>>> 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'
```

我们可以比较日尔曼语族和拉丁语族的不同：

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

词汇工具：Toolbox 和 Shoebox

可能最流行的语言学家用来管理数据的工具是 Toolbox（工具箱），以前叫做 Shoebox（鞋柜），因为它用满满的档案卡片占据了语言学家的旧鞋盒。Toolbox 可以免费从 http://www.sil.org/computing/toolbox/下载。

一个 Toolbox 文件由一个大量条目的集合组成，其中每个条目由一个或多个字段组成。大多数字段都是可选的或重复的，这意味着这个词汇资源不能作为一个表格或电子表格来处理。

下面是一个罗托卡特语（Rotokas）的词典。我们只看第一个条目，词 kaa 的意思是 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.')]), ...]
```

条目包括一系列的属性-值对，如('ps', 'V')，表示词性是'V'(动词)，('ge', 'gag')表示英文注释是'gag'。最后的 3 个配对包含一个罗托卡特语例句和它的巴布亚皮钦语及英语翻译。

Toolbox 文件松散的结构是我们在现阶段很难更好的利用它。XML 提供了一种强有力的方式来处理这种语料库，我们将在第 11 章回到这个的主题。

罗托卡特语是巴布亚新几内亚的布干维尔岛上使用的一种语言。这个词典资源由 Stuart Robinson 贡献给 NLTK。罗托卡特语以仅有 12 个音素（彼此对立的声音）而闻名。详情请参考：http://en.wikipedia.org/wiki/Rotokas_language

2.5 WordNet

WordNet 是面向语义的英语词典，类似与传统辞典，但具有更丰富的结构。NLTK 中包括英语 WordNet，共有 155,287 个词和 117,659 个同义词集合。我们将以寻找同义词和它们在 WordNet 中如何访问开始。

意义与同义词

考虑句子(1a)。如果我们用 automobile 替换掉(1a)中的词 motorcar，变成(1b)，句子的意思几乎保持不变：

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

b. Benz is credited with the invention of the automobile.

因为句子中所有其他成分都保持不变，我们可以得出结论：motorcar 和 automobile 有相同的含义即它们是**同义词**。在 WordNet 的帮助下，我们可以探索这些词：

```
>>> from nltk.corpus import wordnet as wn
```

```
>>> wn.synsets('motorcar')
[Synset('car.n.01')]
```

因此，motorcar 只有一个可能的含义，它被定义为 car.n.01，car 的第一个名词意义。car.n.01 被称为 **synset** 或“**同义词集**”，意义相同的词（或“词条”）的集合：

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

同义词集中的每个词可以有多种含义，例如：car 也可能是火车车厢、一个货车或电梯厢。但我们只对这个同义词集中所有词来说最常用的一个意义感兴趣。同义词集也有一些一般的定义和例句：

```
>>> 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']
```

虽然定义帮助人们了解一个同义词集的本意，同义词集中的词往往对我们的程序更有用。为了消除歧义，我们将这些词标注为 car.n.01.automobile，car.n.01.motorcar 等。这种同义词集和词的配对叫做**词条**。我们可以得到指定同义词集的所有词条①，查找特定的词条②，得到一个词条对应的同义词集③，也可以得到一个词条的“名字”④：

```
>>> wn.synset('car.n.01').lemmas ①
[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') ②
Lemma('car.n.01.automobile')
>>> wn.lemma('car.n.01.automobile').synset ③
Synset('car.n.01')
>>> wn.lemma('car.n.01.automobile').name ④
'automobile'
```

与词 automobile 和 motorcar 这些意义明确的只有一个同义词集的词不同，词 car 是含糊的，有五个同义词集：

```
>>> 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']
```

为方便起见，我们可以用下面的方式访问所有包含词 car 的词条。

```
>>> 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')]
```

轮到你来：写下词 dishd 的你能想到的所有意思。现在，在 WordNet 的帮助下使用前面所示的相同的操作探索这个词。

WordNet 的层次结构

WordNet 的同义词集对应于抽象的概念，它们并不总是有对应的英语词汇。这些概念在层次结构中相互联系在一起。一些概念也很一般，如实体、状态、事件；这些被称为**独一无二的根同义词集**。其他的，如：油老虎和有仓门式后背的汽车等就比较具体的多。图 2-8 展示了一个概念层次的一小部分。

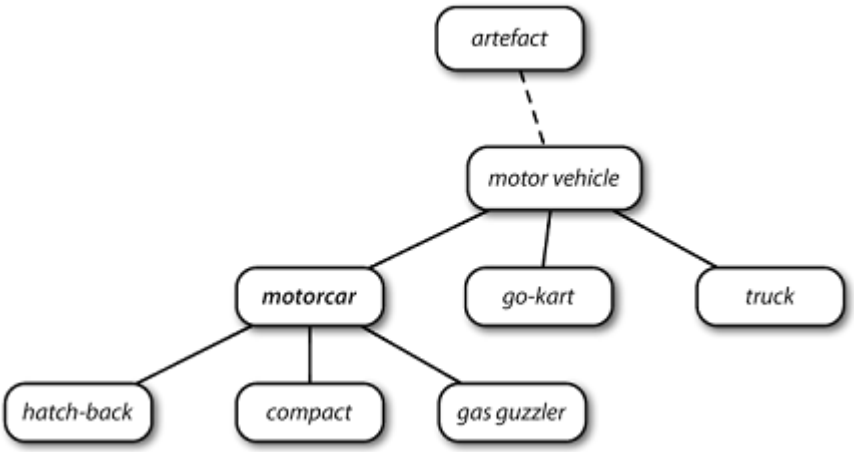


图 2-8. WordNet 概念层次片段：每个节点对应一个同义词集;边表示上位词/下位词关系，即上级概念与从属概念的关系。

WordNet 使在概念之间漫游变的容易。例如：一个如*摩托车*这样的概念，我们可以看到它的更加具体（直接）的概念——**下位词**。

```
>>> motorcar = wn.synset('car.n.01')
>>> types_of_motorcar = motorcar.hyponyms()
>>> types_of_motorcar[26]
Synset('ambulance.n.01')
>>> sorted([lemma.name for synset in types_of_motorcar for lemma in synset.lemmas])
['Model_T', 'S.U.V.', 'SUV', 'Stanley_Steamer', 'ambulance', 'beach_waggon',
'beach_wagon', 'bus', 'cab', 'compact', 'compact_car', 'convertible',
'coupe', 'cruiser', 'electric', 'electric_automobile', 'electric_car',
'estate_car', 'gas_guzzler', 'hack', 'hardtop', 'hatchback', 'heap',
'horseless_carriage', 'hot-rod', 'hot_rod', 'jalopy', 'jeep', 'landrover',
'limo', 'limousine', 'loaner', 'minicar', 'minivan', 'pace_car', 'patrol_car',
'phaeton', 'police_car', 'police_cruiser', 'prowl_car', 'race_car', 'racer',
'racing_car', 'roadster', 'runabout', 'saloon', 'secondhand_car', 'sedan',
'sport_car', 'sport_utility', 'sport_utility_vehicle', 'sports_car', 'squad_car',
'station_waggon', 'station_wagon', 'stock_car', 'subcompact', 'subcompact_car',
'taxi', 'taxicab', 'tourer', 'touring_car', 'two-seater', 'used-car', 'waggon',
'wagon']
```

我们也可以通过访问上位词来浏览层次结构。有些词有多条路径，因为它们可以归类在

一个以上的分类中。car.n.01 与 entity.n.01 之间有两条路径，因为 wheeled_vehicle.n.01 可以同时被归类为车辆和容器。

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

我们可以用如下方式得到一个最一般的上位（或根上位）同义词集：

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

轮到你来：尝试NLTK 中便捷的图形化 WordNet浏览器：nltk.app.wordnet()。沿着上位词与下位词之间的链接，探索 WordNet 的层次结构。

更多的词汇关系

上位词和下位词被称为**词汇关系**，因为它们是同义集之间的关系。这个关系定位上下为“是一个”层次。WordNet 网络另一个重要的漫游方式是从物品到它们的部件（**部分**）或到它们被包含其中的东西（**整体**）。例如：一棵树的部分是它的树干，树冠等；这些都是 part_meronyms()。一棵树的实质是包括心材和边材组成的，即 substance_meronyms()。树木的集合形成了一个森林，即 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')]
```

来看看可以获得多么复杂的东西，考虑具有几个密切相关意思的词 mint。我们可以看到 mint.n.04 是 mint.n.02 的一部分，是组成 mint.n.05 的材质。

```
>>> 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')]
```

动词之间也有关系。例如：走路的动作包括抬脚的动作，所以走路**蕴涵**着抬脚。一些动词有多个蕴涵：

```
>>> 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')]
```

词条之间的一些词汇关系，如：**反义词**:

```
>>> 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')]
```

你可以使用 dir() 查看词汇关系和同义词集上定义的其它方法。例如：尝试 dir(wn.synset('harmony.n.02'))。

语义相似度

我们已经看到同义词集之间构成复杂的词汇关系网络。给定一个同义词集，我们可以遍历 WordNet 网络来查找相关含义的同义词集。知道哪些词是语义相关的，对索引文本集合非常有用，当搜索一个一般性的用语——例如：*车辆*——时就可以匹配包含具体用语——例如*豪华轿车*——的文档。

回想一下每个同义词集都有一个或多个上位词路径连接到一个根上位词，如 entity.n.01。连接到同一个根的两个同义词集可能有一些共同的上位词（见图 2-8）。如果两个同义词集共用一个非常具体的上位词——在上位词层次结构中处于较低层的上位词——它们一定有密切的联系。

```
>>> right = wn.synset('right_whale.n.01')
>>> orca = wn.synset('orca.n.01')
>>> minke = wn.synset('minke_whale.n.01')
>>> tortoise = wn.synset('tortoise.n.01')
>>> novel = wn.synset('novel.n.01')
>>> right.lowest_common_hypernyms(minke)
```

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

当然，我们知道，鲸鱼是非常具体的（须鲸更是如此），脊椎动物是更一般的，而实体完全是抽象的一般的。我们可以通过查找每个同义词集深度量化这个一般性的概念：

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

WordNet 同义词集的集合上定义了类似的函数能够深入的观察。例如：path_similarityassigns 是基于上位词层次结构中相互连接的概念之间的最短路径在 0-1 范围的打分（两者之间没有路径就返回-1）。同义词集与自身比较将返回 1。考虑以下的相似度：露脊鲸与小须鲸、逆戟鲸、乌龟以及小说。数字本身的意义并不大，当我们从海洋生物的语义空间转移到非生物时它是减少的。

```
>>> right.path_similarity(minke)
0.25
>>> right.path_similarity(orca)
0.16666666666666666
>>> right.path_similarity(tortoise)
0.07692307692307692
>>> right.path_similarity(novel)
0.043478260869565216
```

还有一些其它的相似性度量方法；你可以输入 help(wn)获得更多信息。NLTK 还包括 VerbNet，一个连接到 WordNet 的动词的层次结构的词典。

2.6 小结

- 文本语料库是一个大型结构化文本的集合。NLTK 包含了许多语料库，如：布朗语料库 nltk.corpus.brown。
- 有些文本语料库是分类的，例如通过文体或者主题分类；有时候语料库的分类会相互重叠。
- 条件频率分布是一个频率分布的集合，每个分布都有一个不同的条件。它们可以用于通过给定内容或者文体对词的频率计数。
- 行数较多的 Python 程序应该使用文本编辑器来输入，保存为.py 后缀的文件，并使用 import 语句来访问。

- Python 函数允许你将一段特定的代码块与一个名字联系起来，然后重用这些代码想用多少次就用多少次。
- 一些被称为“方法”的函数与一个对象联系在起来，我们使用对象名称跟一个点然后跟方法名称来调用它，就像：x.funct(y)或者 word.isalpha()。
- 要想找到一些关于变量 v 的信息，可以在 Pyhon 交互式解释器中输入 help(v)来阅读这一类对象的帮助条目。
- WordNet 是一个面向语义的英语词典，由同义词的集合—或称为同义词集（synsets）—组成，并且组织成一个网络。
- 默认情况下有些函数是不能使用的，必须使用 Python 的 import 语句来访问。

2.7 深入阅读

本章的附加材料发布在 http://www.nltk.org/，包括网络上免费提供的资源的链接。语料库方法总结请参阅语料库 HOWTO（http://www.nltk.org/howto），在线 API 文档中也有更广泛的资料。公开发行的语料库的重要来源是*语言数据联盟*（(LDC）和*欧洲语言资源局*（ELRA）。提供几十种语言的数以百计的已标注文本和语音语料库。非商业许可证允许这些数据用于教学和科研目的。其中一些语料库也提供商业许可（但需要较高的费用）。

这些语料库和许多其他语言资源使用 OLAC 元数据格式存档，可以通过 http://www.language-archives.org/上的 OLAC 主页搜索到。语料库列表（见 http://gandalf.aksis.uib.no/corpora/sub.html）是一个讨论语料库内容的邮件列表，你可以通过搜索列表档案来找到资源或发布资源到列表中。Ethnologue 是最完整的世界上的语言的清单，http://www.ethnologue.com/。7000 种语言中只有几十中有大量适合 NLP 使用的数字资源。

本章触及**语料库语言学**领域。在这一领域的其他有用的书籍包括(Biber, Conrad, & Reppen, 1998)、(McEnery, 2006)、(Meyer, 2002)、(Sampson & McCarthy, 2005)和(Scott & Tribble, 2006)。在语言学中海量数据分析的深入阅读材料有(Baayen, 2008)、(Gries, 2009)和(Woods, Fletcher, & Hughes, 1986)。

WordNet 的原始描述在(Fellbaum, 1998)中。虽然 WordNet 最初是为心理语言学研究开发的，它目前在自然语言处理和信息检索领域被广泛使用。WordNets 正在开发许多其他语言的版本，在 http://www.globalwordnet.org/中有记录。学习 WordNet 相似性度量可以阅读(Budanitsky & Hirst, 2006)。

本章触及的其它主题是语音和词汇语义学，读者可以参考(Jurafsky & Martin, 2008)的第 7 和第 20 章。

2.8 练习

1. ○创建一个变量 phrase 包含一个词的链表。实验本章描述的操作，包括加法、乘法、索引、切片和排序。
2. ○使用语料库模块处理 austen-persuasion.txt。这本书中有多少词标识符？多少词类型？
3. ○使用布朗语料库阅读器 nltk.corpus.brown.words()或网络文本语料库阅读器 nltk.corpus.webtext.words()来访问两个不同文体的一些样例文本。
4. ○使用 state_union 语料库阅读器，访问《国情咨文报告》的文本。计数每个文档中出现的 men、women 和 people。随时间的推移这些词的用法有什么变化？

5. ○考查一些名词的整体部分关系。请记住，有 3 种整体部分关系，所以你需要使用 member_meronyms()，part_meronyms()，substance_meronyms()，member_holonyms()，part_holonyms()以及 substance_holonyms()。
6. ○在比较词表的讨论中，我们创建了一个对象叫做 translate，通过它你可以使用德语和意大利语词汇查找对应的英语词汇。这种方法可能会出现什么问题？你能提出一个办法来避免这个问题吗？
7. ○根据 Strunk 和 White 的《Elements of Style》，词 however 在句子开头使用是“in whatever way”或“to whatever extent”的意思，而没有“nevertheless”的意思。他们给出了正确用法的例子：However you advise him, he will probably do as he thinks best.（http://www.bartleby.com/141/strunk3.html）。使用词汇索引工具在我们一直在思考的各种文本中研究这个词的实际用法。也可以看 LanguageLog 发布在 http://itre.cis.upenn.edu/~myl/languagelog/archives/001913.html 上的“Fossilized prejudices about ‘however’”。
8. ◑在名字语料库上定义一个条件频率分布，显示哪个首字母在男性名字中比在女性名字中更常用（见图 2-7）。
9. ◑挑选两个文本，研究它们之间在词汇、词汇丰富性、文体等方面的差异。你能找出几个在这两个文本中词意相当不同的词吗？例如：在《白鲸记》与《理智与情感》中的 monstrous。
10. ◑阅读 BBC 新闻文章：“UK’s Vicky Pollards ‘left behind’” http://news.bbc.co.uk/1/hi/education/6173441.stm。文章给出了有关青少年语言的以下统计：“使用最多的 20 个词，包括 yeah, no, but 和 like，占所有词的大约三分之一”。对于大量文本源来说，所有词标识符的三分之一有多少词类型？你从这个统计中得出什么结论？更多相关信息请阅读 LanguageLog 上的 http://itre.cis.upenn.edu/~myl/languagelog/archives/003993.html。
11. ◑调查模式分布表，寻找其他模式。试着用你自己对不同文体的印象理解来解释它们。你能找到其他封闭的词汇归类，展现不同文体的显著差异吗？
12. ◑CMU 发音词典包含某些词的多个发音。它包含多少种不同的词？具有多个可能的发音的词在这个词典中的比例是多少？
13. ◑没有下位词的名词同义词集所占的百分比是多少？你可以使用 wn.all_synsets('n') 得到所有名词同义词集。
14. ◑定义函数 supergloss(s)，使用一个同义词集 s 作为它的参数，返回一个字符串，包含 s 的定义和 s 所有的上位词与下位词的定义的连接字符串。
15. ◑写一个程序，找出所有在布朗语料库中出现至少 3 次的词。
16. ◑写一个程序，生成如表 1-1 所示的词汇多样性得分表（例如：标识符/类型的比例）。包括布朗语料库文体的全集（nltk.corpus.brown.categories()）。哪个文体词汇多样性最低（每个类型的标识符数最多）？这是你所期望的吗？
17. ◑写一个函数，找出一个文本中最常出现的 50 个词，停用词除外。
18. ◑写一个程序，输出一个文本中 50 个最常见的双连词（相邻词对），忽略包含停用词的双连词。
19. ◑写一个程序，按文体创建一个词频表，以 2.1 节给出的词频表为范例，选择你自己的词汇，并尝试找出那些在一个文体中很突出或很缺乏的词汇。讨论你的研究结果。
20. ◑写一个函数 word_freq()，用一个词和布朗语料库中的一个部分的名字作为参数，计算这部分语料中词的频率。
21. ◑写一个程序，估算一个文本中的音节数，利用 CMU 发音词典。
22. ◑定义一个函数 hedge(text)，处理一个文本和产生一个新的版本在每三个词之间插入一个词 like。

23. ●**齐夫定律**：f(w)是一个自由文本中的词 w 的频率。假设一个文本中的所有词都按照它们的频率排名，频率最高的在最前面。齐夫定律指出一个词类型的频率与它的排名成反比（即 f×r=k，k 是某个常数）。例如：最常见的第 50 个词类型出现的频率应该是最常见的第 150 个词型出现频率的 3 倍。
    a. 写一个函数来处理一个大文本,使用 pylab.plot 画出相对于词的排名的词的频率。你认可齐夫定律吗？（提示：使用对数刻度会有帮助。）所绘的线的极端情况是怎样的？
    b. 随机生成文本，如：使用 random.choice("abcdefg ")，注意要包括空格字符。你需要事先 import random。使用字符串连接操作将字符累积成一个很长的字符串。然后为这个字符串分词，产生前面的齐夫图，比较这两个图。此时你怎么看齐夫定律？
24. ●修改例 2-1 的文本生成程序，进一步完成下列任务：
    a. 在一个词链表中存储 n 个最相似的词，使用 random.choice()从链表中随机选取一个词。（你将需要事先 import random）
    b. 选择特定的文体，如：布朗语料库中的一部分或者《创世纪》翻译或者古腾堡语料库中的文本或者一个网络文本。在此语料上训练一个模型，产生随机文本。你可能要实验不同的起始字。文本的可理解性如何？讨论这种方法产生随机文本的长处和短处。
    c. 现在使用两种不同文体训练你的系统，使用混合文体文本做实验。讨论你的观察结果。
25. ●定义一个函数 find_language()，用一个字符串作为其参数，返回包含这个字符串作为词汇的语言的列表。使用《世界人权宣言》（udhr）的语料，将你的搜索限制在 Latin-1 编码的文件中。
26. ●名词上位词层次的分枝因素是什么？也就是说，对于每一个具有下位词——上位词层次中的子女——的名词同义词集，它们平均有几个下位词？你可以使用 wn.all_synsets('n')获得所有名词同义词集。
27. ●一个词的多义性是它所有含义的个数。利用 WordNet，使用 len(wn.synsets('dog', 'n'))我们可以判断名词 dog 有 7 种含义。计算 WordNet 中名词、动词、形容词和副词的平均多义性。
28. ●使用预定义的相似性度量之一给下面的每个词对的相似性打分。按相似性减少的顺序排名。你的排名与这里给出的顺序有多接近？(Miller & Charles, 1998)实验得出的顺序：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.

第 3 章 加工原料文本

文本的最重要来源无疑是网络。探索现成的文本集合，如我们在前面章节中看到的语料库，是很方便的。然而，在你心中可能有你自己的文本来源，需要学习如何访问它们。

本章的目的是要回答下列问题：

1. 我们怎样才能编写程序访问本地和网络上的文件，从而获得无限的语言材料？
2. 我们如何把文档分割成单独的词和标点符号，这样我们就可以开始像前面章节中在文本语料上做的那样的分析？
3. 我们怎样编程程序产生格式化的输出，并把结果保存在一个文件中？

为了解决这些问题，我们将讲述 NLP 中的关键概念，包括分词和词干提取。在此过程中，你会巩固你的 Python 知识并且了解关于字符串、文件和正则表达式知识。既然这些网络上的文本都是 HTML 格式的，我们也将看到如何去除 HTML 标记。

重点：从本章开始往后我们的例子程序将假设你以下面的导入语句开始你的交互式会话或程序：

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

3.1 从网络和硬盘访问文本

电子书

NLTK 语料库集合中有古腾堡项目的一小部分样例文本。然而，你可能对分析古腾堡项目的其它文本感兴趣。你可以在 http://www.gutenberg.org/catalog/ 上浏览 25,000 本免费在线书籍的目录，获得 ASCII 码文本文件的 URL。虽然 90%的古腾堡项目的文本是英语的，它还包括超过 50 种语言的材料，包括加泰罗尼亚语、中文、荷兰语、芬兰语、法语、德语、意大利语、葡萄牙语和西班牙语（每种语言都有超过 100 个文本）。

编号 2554 的文本是《罪与罚》的英文翻译，我们可以如下方式访问它。

```
>>> 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'
```

进程 read()将需要几秒钟来下载这本大书。如果你使用的 Internet 代理 Python 不能正确检测出来，你可能需要用下面的方法手动指定代理：

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

变量 raw 包含一个有 176,831 个字符的字符串。（我们使用 type(raw)可以看到它是一个字符串。）这是这本书原始的内容，包括很多我们不感兴趣的细节，如空格、换行符和空行。请注意，文件中行尾的\r 和\n，这是 Python 用来显示特殊的回车和换行字符的方式（这个文件一定是在 Windows 机器上创建的）。对于语言处理，我们要将字符串分解为词和标点符号，正如我们在第 1 章中所看到的。这一步被称为**分词**，它产生我们所熟悉的结构，一个词汇和标点符号的链表。

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

请注意，NLTK 需要分词，但所有前面的打开一个 URL 读入一个字符串的任务都没有分词。如果我们现在采取进一步的步骤从这个链表创建一个 NLTK 文本，我们可以进行我们在第 1 章看到的所有的其他语言的处理，也包括常规的链表操作，例如切片：

```
>>> 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
```

请注意，古腾堡项目以一个排列的形式出现。这是因为从古腾堡项目下载的每个文本都包含一个首部，里面有文本的名称、作者、扫描和校对文本的人的名字、许可证等信息。有时这些信息出现在文件末尾页脚处。我们不能可靠地检测出文本内容的开始和结束，因此在从原始文本中挑出内容之前，我们需要手工检查文件以发现标记内容开始和结尾的独特的字符串。

```
>>> raw.find("PART I")
5303
>>> raw.rfind("End of Project Gutenberg's Crime")
1157681
```

```
>>> raw = raw[5303:1157681] ①
>>> raw.find("PART I")
0
```

方法 find()和 rfind()（反向的 find）帮助我们得到字符串切片需要用到的正确的索引值①。我们用这个切片重新给 raw 赋值，所以现在 raw 以“PART I”开始一直到（但不包括）标记内容结尾的句子。

这是我们第一次接触到网络的实际内容：在网络上找到的文本可能含有不必要的内容，并没有一个自动的方法来去除它。但只需要少量的额外工作，我们就可以提取出我们需要的材料。

处理的 HTML

网络上的文本大部分是 HTML 文件的形式。你可以使用网络浏览器将网页作为文本保存为本地文件，然后按照后面关于文件的小节描述的那样来访问它。不过，如果你要经常这样做，最简单的办法是直接让 Python 来做这份工作。第一步是像以前一样使用 urlopen。为了好玩，我们将挑选被称为“Blondes to die out in 200 years”的 BBC 新闻故事，一个都市传奇被 BBC 作为确立的科学事实流传下来。

```
>>> 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'
```

输入 print html 可以看到 HTML 的全部内容，包括 meta 元标签、图像标签、map 标签、JavaScript、表单和表格。

从 HTML 中提取文本是极其常见的任务，NLTK 提供了一个辅助函数 nltk.clean_html()将 HTML 字符串作为参数，返回原始文本。然后我们可以对原始文本进行分词，获得我们熟悉的文本结构：

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

其中仍然含有不需要的内容，包括网站导航及有关报道等。通过一些尝试和出错你可以找到内容索引的开始和结尾，并选择你感兴趣的标识符，按照前面讲的那样初始化一个文本。

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

更多更复杂的有关处理 HTML 的内容，可以使用 http://www.crummy.com/software/BeautifulSoup/上的 Beautiful Soup 软件包。

处理搜索引擎的结果

网络可以被看作未经标注的巨大的语料库。网络搜索引擎提供了一个有效的手段，搜索大量文本作为有关的语言学的例子。搜索引擎的主要优势是规模：因为你正在寻找这样庞大的一个文件集，会更容易找到你感兴趣语言模式。而且，你可以使用非常具体的模式，仅仅在较小的范围匹配一两个例子，但在网络上可能匹配成千上万的例子。网络搜索引擎的第二个优势是非常容易使用。因此，它是一个非常方便的工具，可以快速检查一个理论是否合理。请看表 3-1 的一个例子。

表 3-1. 搭配的谷歌命次数：搭配的命中次数包括词 absolutely *或* definitely，*跟着* adore，love，like *或* prefer *其中的一个。(Liberman, in LanguageLog, 2005)*

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

不幸的是，搜索引擎有一些显著的缺点。首先，允许的搜索方式的范围受到严格限制。不同于本地驱动器中的语料库，你可以编写程序来搜索任意复杂的模式，搜索引擎一般只允许你搜索单个词或词串，有时也允许使用通配符。其次，搜索引擎给出的结果不一致，并且在不同的时间或在不同的地理区域会给出非常不同的结果。当内容在多个站点重复时，搜索结果会增加。最后，搜索引擎返回的结果中的标记可能会不可预料的改变，基于模式的方法定位特定的内容将无法使用（通过使用搜索引擎 APIs 可以改善这个问题）。

轮到你来：在网络中搜索“the of”（引号内的内容）。基于巨大的数据，我们是否可以得出“the of”是英语中常用搭配的结论呢？

处理 RSS 订阅

博客圈是文本的重要来源，无论是正式的还是非正式的。在一个叫做 Universal Feed Parser 的第三方 Python 库（可从 http://feedparser.org/免费下载）的帮助下，我们可以访问一个博客的内容，如下所示：

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

```
>>> nltk.word_tokenize(nltk.clean_html(llog.entries[2].content[0].value))
[u'Today', u'I', u'was', u'chatting', u'with', u'three', u'of', u'our', u'visiting',
u'graduate', u'students', u'from', u'the', u'PRC', u'.', u'Thinking', u'that', u'I',
u'was', u'being', u'au', u'courant', u',', u'I', u'mentioned', u'the', u'expression',
u'DUI4XIANG4', u'\u5c0d\u8c61', u'(', u'boy', u'/', u'girl', u'friend', u')', ...]
```

请注意，结果字符串有一个 u 前缀表示它们是 Unicode 字符串（见 3.3 节）。伴随着一些更深入的工作，我们可以编写程序创建一个博客帖子的小语料库，并以此作为我们 NLP 的工作基础。

读取本地文件

为了读取本地文件，我们需要使用 Python 内置的 open()函数，然后是 read()方法。假设你有一个文件 document.txt，你可以像这样加载它的内容：

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

轮到你来：使用文本编辑器创建一个名为 document.txt 的文件，然后输入几行文字，保存为纯文本。如果你使用 IDLE，在“文件”菜单中选择“新建窗口”命令，在新窗口中输入所需的文本，然后在 IDLE 提供的弹出式对话框中的文件夹内保存文件为 document.txt。然后在 Python 解释器中使用 f = open('document.txt')打开这个文件，并使用 print f.read()检查其内容。

当你尝试这样做时可能会出各种各样的错误。如果解释器无法找到你的文件，你会看到类似这样的错误：

```
>>> 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'
```

要检查你正试图打开的文件是否在正确的目录中，使用 IDLE“文件”菜单上的“打开”命令；这将显示 IDLE 当前目录下所有文件的清单。另一种方法是在 Python 中检查当前目录：

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

另一个你在访问一个文本文件时可能遇到的问题是换行的约定，这个约定因操作系统不同而不同。内置的 open()函数的第二个参数用于控制如何打开文件：open('document.txt', 'rU')。'r'意味着以只读方式打开文件（默认），'U'表示“通用”，它让我们忽略不同的换行约定。

假设你已经打开了该文件，有几种方法可以阅读此文件。read()方法创建了一个包含整个文件内容的字符串：

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

回想一下'\n'字符是换行符；这相当于按键盘上的 Enter 开始一个新行。

我们也可以使用一个 for 循环一次读文件中的一行：

```
>>> f = open('document.txt', 'rU')
```

```
>>> for line in f:
...     print line.strip()
Time flies like an arrow.
Fruit flies like a banana.
```

在这里，我们使用 strip() 方法删除输入行结尾的换行符。

NLTK 中的语料库文件也可以使用这些方法来访问。我们只需使用 nltk.data.find()来获取语料库项目的文件名。然后就可以使用我们刚才讲的方式打开和阅读它。

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

从 PDF、MS Word 及其他二进制格式中提取文本

ASCII 码文本和 HTML 文本是人可读的格式。文字常常以二进制格式出现，如 PDF 和 MSWord，只能使用专门的软件打开。第三方函数库如 pypdf 和 pywin32 提供了对这些格式的访问。从多列文档中提取文本是特别具有挑战性的。一次性转换几个文件，会比较简单些，用一个合适的应用程序打开文件，以文本格式保存到本地驱动器，然后以如下所述的方式访问它。如果该文档已经在网络上，你可以在 Google 的搜索框输入它的 URL。搜索结果通常包括这个文档的 HTML 版本的链接，你可以将它保存为文本。

捕获用户输入

有时我们想捕捉用户与我们的程序交互时输入的文本。调用 Python 函数 raw_input() 提示用户输入一行数据。保存用户输入到一个变量后，我们可以像其他字符串那样操纵它。

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

NLP 的流程

图 3-1 总结了我们在本节涵盖的内容，包括我们在第 1 章中所看到的建立一个词汇表的过程。（其中一个步骤，规范化，将在 3.6 节讨论。）

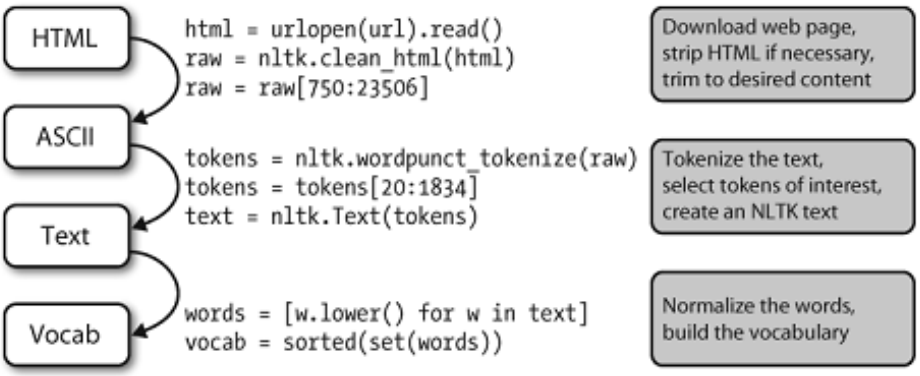


图 3-1. 处理流程：打开一个 URL，读里面 HTML 格式的内容，去除标记，并选择字符的切片，然后分词，是否转换为 nltk.Text 对象是可选择的。我们也可以将所有词汇小写并提取词汇表。

在这条流程后面还有很多操作。要正确理解它，这样有助于明确其中提到的每个变量的类型。使用 type(x)我们可以找出任一 Python 对象 x 的类型，如 type(1)是<int>因为 1 是一个整数。

当我们载入一个 URL 或文件的内容时，或者当我们去掉 HTML 标记时，我们正在处理字符串，也就是 Python 的<str>数据类型（在 3.2 节，我们将学习更多有关字符串的内容）：

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

当我们将一个字符串分词，会产生一个（词的）链表，这是 Python 的<list>类型。规范化和排序链表产生其它链表：

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

一个对象的类型决定了它可以执行哪些操作。比如说我们可以追加元素到一个链表，但不能追加元素到一个字符串：

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

同样的，我们可以连接字符串与字符串，列出链表内容，但我们不能连接字符串与链表：

```
>>> 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
```

在下一节，我们将更加细致的探讨字符串，并进一步探索字符串和链表之间的关系。

3.2 字符串：最底层的文本处理

现在是时候研究一个之前我们一直故意避开的基本数据类型了。在前面的章节中，我们侧重于将文本作为一个词链表。我们并没有细致的探讨词汇以及它们是如何在编程语言中被处理的。通过使用 NLTK 中的语料库接口，我们可以忽略这些文本所在的文件。一个词的内容，一个文件的内容在编程语言中是由一个叫做**字符串**的基本数据类型来表示的。在本节

中，我们将详细探讨字符串，并展示字符串与词汇、文本和文件之间的联系。

字符串的基本操作

可以使用单引号①或②双引号来指定字符串，如下面的例子代码所示。如果一个字符串中包含一个单引号，我们必须在单引号前加反斜杠③让 Python 知道这是字符串中的单引号。或者也可以将这个字符串放入双引号中②。否则，字符串内的单引号④将被解释为字符串结束标志，Python 解释器会报告一个语法错误：

```
>>> monty = 'Monty Python' ①
>>> monty
'Monty Python'
>>> circus = "Monty Python's Flying Circus" ②
>>> circus
"Monty Python's Flying Circus"
>>> circus = 'Monty Python\'s Flying Circus' ③
>>> circus
"Monty Python's Flying Circus"
>>> circus = 'Monty Python's Flying Circus' ④
  File "<stdin>", line 1
    circus = 'Monty Python's Flying Circus'
                            ^
SyntaxError: invalid syntax
```

有时字符串跨好几行。Python 提供了多种方式表示它们。在下面的例子中，一个包含两个字符串的序列被连接为一个字符串。我们需要使用反斜杠①或者括号②，这样解释器就知道第一行的表达式不完整。

```
>>> couplet = "Shall I compare thee to a Summer's day?"\
...           "Thou are more lovely and more temperate:" ①
>>> print couplet
Shall I compare thee to a Summer's day?Thou are more lovely and more temperate:
>>> couplet = ("Rough winds do shake the darling buds of May,"
...           "And Summer's lease hath all too short a date:") ②
>>> print couplet
Rough winds do shake the darling buds of May,And Summer's lease hath all too short a date:
```

不幸的是，这些方法并没有展现给我们十四行诗的两行之间的换行。为此，我们可以使用如下所示的三重引号的字符串：

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

```
And Summer's lease hath all too short a date:
```

现在我们可以定义字符串，也可以在上面尝试一些简单的操作。首先，让我们来看看+操作，被称为**连接①**。此操作产生一个新字符串，它是两个原始字符串首尾相连粘贴复制而成。请注意，连接不会做一些比较聪明的事，例如在词汇之间插入空格。我们甚至可以对字符串用乘法②：

```
>>> 'very' + 'very' + 'very' ①
'veryveryvery'
>>> 'very' * 3 ②
'veryveryvery'
```

轮到你来：试运行下面的代码，然后尝试使用你对字符串+和*操作的理解，弄清楚它是如何运作的。要小心区分字符串' '，这是一个空格符，和字符串''，这是一个空字符串。

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

我们已经看到加法和乘法运算不仅仅适用于数字也适用于字符串。但是，请注意，我们不能对字符串用减法或除法：

```
>>> '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'
```

这些错误消息是 Python 的另一个例子，告诉我们：对数据类型陷入困惑。第一种情况告诉我们减法操作（即：-）不能适用于对象类型 str（字符串），而第二种情况告诉我们除法的两个操作数不能分别为 str 和 int。

输出字符串

到目前为止，当我们想看看变量的内容或想看到计算的结果，我们就把变量的名称输入到解释器。我们还可以使用 print 语句来看一个变量的内容：

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

请注意这次是没有引号的。当我们通过输入变量的名字到解释器中来检查它时，解释器输出 Python 中的变量的值。因为它是一个字符串，结果被引用。然而，当我们告诉解释器输出这个变量时，我们没有看到引号字符，因为字符串的内容里面没有引号。

print 语句可以多种方式显示多行，就像这样：

```
>>> 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
```

访问单个字符

正如我们在 1.2 节看到的链表，字符串也是被索引的，从零开始。当我们索引一个字符串时，我们得到它的一个字符（或字母）。一个单独的字符并没有什么特别，它只是一个长度为 1 的字符串。

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

与链表一样，如果我们尝试访问一个超出字符串范围的索引时，会得到了一个错误：

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

也与链表一样，我们可以使用字符串的负数索引，其中-1 是最后一个字符的索引①。正数和负数的索引给我们两种方式指示一个字符串中的任何位置。在这种情况下，当一个字符串长度为 12 时，索引 5 和-7 都指示相同的字符（一个空格）。（请注意，`5 = len(monty) - 7`）

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

我们可以写一个 `for` 循环，遍历字符串中的字符。`print` 语句结尾加一个逗号，这是为了告诉 Python 不要在行尾输出换行符。

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

我们也可以计数单个字符。通过将所有字符小写来忽略大小写的区分，并过滤掉非字母字符。

```
>>> 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']

这段代码按照出现频率最高排在最先的顺序显示出英文字母（这是一个相当复杂的过程，我们会在以后更仔细地解释）。你可能想用 fdist.plot()可视化这个分布。一个文本相关的字母频率特征可以用在文本语言自动识别中。

访问子字符串

一个子字符串是我们为进一步处理而从一个字符串中取出的任意连续片段。我们可以很容易地访问子字符串就像我们对链表进行切片一样（见图 3-2）。例如：下面的代码访问从索引 6 到索引 10（但不包括）的子字符串：

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

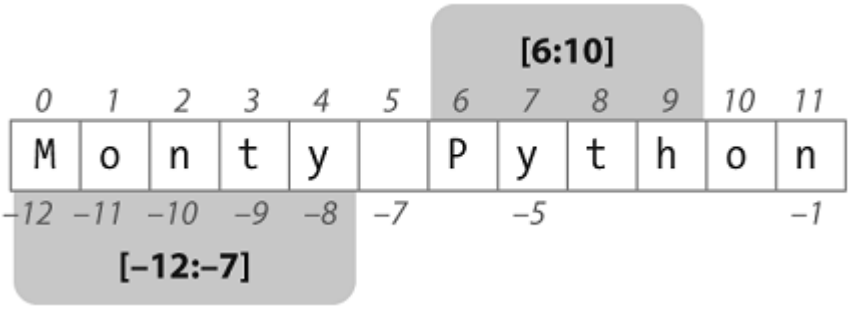


图 3-2. 字符串切片：伴随字符串 Monty Python *显示它的正数和负数索引；两个子字符串都选择使用“切片”符号。切片*[m，n]*包含从位置 m 到 n-1 中的字符。*

在这里，我们看到的字符是'P'，'y'，'t'和'h'，它们分别对应于 monty[6]...monty[9]而不是 monty[10]。这是因为切片开始于第一个索引，但结束于最后一个索引的前一个。

我们也可以使用负数索引切片——也是同样的规则，从第一个索引开始到最后一个索引的前一个结束；在这里是在空格字符前结束。

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

与链表切片一样，如果我们省略了第一个值，子字符串将从字符串的开头开始。

如果我们省略了第二个值，则子字符串直到字符串的结尾结束：

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

我们使用 in 操作符测试一个字符串是否包含一个特定的子字符串，如下所示：

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

我们也可以使用 find()找到一个子字符串在字符串内的位置：

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

轮到你来：造一句话，将它分配给一个变量，例如：sent = 'my sentence...'。写切片表达式抽取个别词。（这显然不是一种方便的方式来处理文本中的词！）

更多的字符串操作

Python 对处理字符串的支持很全面。表 3-2 所示是一个总结，其中包括一些我们还没有看到的操作。关于字符串的更多信息，可在 Python 提示符下输入 help(str)。

表 3-2. 有用的字符串方法：表 1-4 中字符串测试之外的字符串上的操作；所有的方法都产生一个新的字符串或链表

方法	功能
s.find(t)	字符串 s 中包含 t 的第一个索引（没找到返回-1）
s.rfind(t)	字符串 s 中包含 t 的最后一个索引（没找到返回-1）
s.index(t)	与 s.find(t)功能类似，但没找到时引起 ValueError
s.rindex(t)	与 s.rfind(t)功能类似，但没找到时引起 ValueError
s.join(text)	连接字符串 s 与 text 中的词汇
s.split(t)	在所有找到 t 的位置将 s 分割成链表（默认为空白符）
s.splitlines()	将 s 按行分割成字符串链表
s.lower()	将字符串 s 小写
s.upper()	将字符串 s 大写
s.titlecase()	将字符串 s 首字母大写
s.strip()	返回一个没有首尾空白字符的 s 的拷贝
s.replace(t, u)	用 u 替换 s 中的 t

链表与字符串的差异

字符串和链表都是一种序列。我们可以通过索引抽取它们中的一部分，可以给它们切片，可以使用连接将它们合并在一起。但是，字符串和链表之间不能连接。

```
>>> 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']
```

当我们在一个 Python 程序中打开并读入一个文件，我们得到一个对应整个文件内容的字符串。如果我们使用一个 for 循环来处理这个字符串元素，所有我们可以挑选出的只是单个的字符——我们不选择粒度。相比之下，链表中的元素可以很大也可以很小，只要我们喜欢。例如：它们可能是段落、句子、短语、单词、字符。所以，链表的优势是我们可以灵活的决定它包含的元素，相应的后续的处理也变得灵活。因此，我们在一段 NLP 代码中可能做的第一件事情就是将一个字符串分词放入一个字符串链表中（3.7 节）。相反，当我们要将结果写入到一个文件或终端，我们通常会将它们格式化为一个字符串（3.9 节）。

链表与字符串的功能有很大不同。链表增加了使你可以改变其中的元素的能力：

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

另一方面，如果我们尝试在一个字符串上这么做——将 query 的第 0 个字符修改为 'F'——我们得到：

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

这是因为字符串是**不可变的**：一旦你创建了一个字符串，就不能改变它。然而，链表是**可变的**，其内容可以随时修改。作为一个结论，链表支持修改原始值的操作，而不是产生一个新的值。

轮到你来：通过尝试本章结尾的一些练习，巩固你的字符串知识。

3.3 使用 Unicode 进行文字处理

我们的程序经常需要处理不同的语言和不同的字符集。“纯文本”的概念是虚构的。如果你住在讲英语国家，你可能在使用 ASCII 码而没有意识到这一点。如果你住在欧洲，你可能使用一种扩展拉丁字符集，包含丹麦语和挪威语中的“ø”，匈牙利语中的“ő”，西班牙和布列塔尼语中的“ñ”，捷克语和斯洛伐克语中的“ň”。在本节中，我们将概述如何使用 Unicode 处理使用非 ASCII 字符集的文本。

什么是 Unicode？

Unicode 支持超过一百万种字符。每个字符分配一个编号，称为**编码点**。在 Python 中，编码点写作\uXXXX 的形式，其中 XXXX 是四位十六进制形式数。

在一个程序中，我们可以像普通字符串那样操纵 Unicode 字符串。然而，当 Unicode 字

符被存储在文件或在终端上显示，它们必须被编码为字节流。一些编码（如 ASCII 和 Latin-2）中每个编码点使用单字节，所以它们可以只支持 Unicode 的一个小的子集就足够一种语言使用了。其它的编码（如 UTF-8）使用多个字节，可以表示全部的 Unicode 字符。

文件中的文本都是有特定编码的，所以我们需要一些机制来将文本翻译成 Unicode——翻译成 Unicode 叫做**解码**。相对的，要将 Unicode 写入一个文件或终端，我们首先需要将 Unicode 转化为合适的编码——这种将 Unicode 转化为其它编码的过程叫做**编码**，如图 3-3 所示。

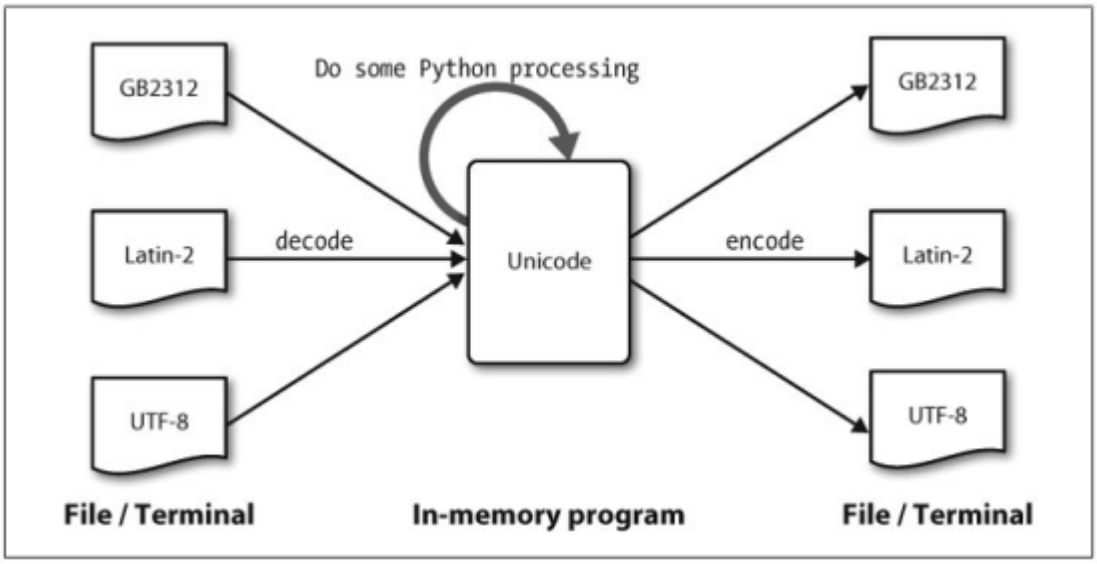


图 3-3. Unicode 的解码和编码

从 Unicode 的角度来看，字符是可以实现一个或多个**字形**的抽象的实体。只有字形可以出现在屏幕上或被打印在纸上。一个字体是一个字符到字形映射。

从文件中提取已编码文本

假设我们有一个小的文本文件，我们知道它是如何编码的。例如：polish-lat2.txt 顾名思义是波兰语的文本片段（来源波兰语 Wikipedia；可以在 http://pl.wikipedia.org/wiki/Biblioteka_Pruska 中看到）。此文件是 Latin-2 编码的，也称为 ISO-8859-2。nltk.data.find()函数为我们定位文件。

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

Python 的 codecs 模块提供了将编码数据读入为 Unicode 字符串和将 Unicode 字符串以编码形式写出的函数。codecs.open()函数有一个 encoding 参数来指定被读取或写入的文件的编码。让我们导入 codecs 模块，以“latin2”为 encoding 参数，调用它以 Unicode 打开我们的波兰语文件。

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

关于 codecs 允许的 encoding 参数列表，可以在 http://docs.python.org/lib/standard-encodings.html 中看到。请注意我们可以使用 f = codecs.open(path, 'w', encoding='utf-8')写入 Unicode 编码数据到一个文件。

从文件对象 f 读出的文本将以 Unicode 返回。正如我们较早前指出的，要在终端上查看这个文本，我们需要使用合适的编码对它进行编码。Python 特定的编码 unicode_escape 是一个虚拟的编码，它把所有非 ASCII 字符转换成它们的\uXXXX 形式。编码点在 ASCII

码 0-127 的范围以外但低于 256 的使用两位数字的形式\xXX 表示。

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

在这个输出的第一行有一个以\u 转义字符串开始的 Unicode 转义字符串，即\u0144。相关的 Unicode 字符将会以字形ń显示在屏幕上。在前面例子中的第三行中，我们看到\xF3，对应字形为ó，在 128-255 的范围内。

在 Python 中，一个 Unicode 字符串常量可以通过在字符串常量前面加一个 u 也就是 u'hello'来指定。任意 Unicode 字符通过在 Unicode 字符串常量内使用\uXXXX 转义序列来定义。我们使用 ord()查找一个字符的整数序数。例如：

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

97 的十六进制四位数表示是 0061，所以我们可以使用相应的转义序列定义一个 Unicode 字符串常量：

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

请注意，Python 的 print 语句假设 Unicode 字符的默认编码是 ASCII 码。然而，ń不在 ASCII 码范围之内，所以除非我们指定编码否则不能被输出。在下面的例子中，我们指定 print 使用 repr()转化的字符串，repr()输出 UTF-8 转义序列（以\xXX 的形式），而不是试图显示字形。

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

如果您的操作系统和区域的设置支持 UTF-8 编码字符，你输入 Python 命令：print nacute_utf，应该能够在你的屏幕上看到ń。

决定屏幕上显示的字形的因素很多。如果你确定你的编码正确但你的 Python 代码仍然未能显示出你预期的字形，你应该检查你的系统上是否安装了所需的字体。

unicodedata 模块使我们可以检查 Unicode 字符的属性。在下面的例子中，我们选择超出 ASCII 范围的波兰语文本的第三行中的所有字符，输出它们的 UTF-8 转义值，然后是

使用标准 Unicode 约定的它们的编码点整数（即以 U+为前缀的十六进制数字），随后是它们的 Unicode 名称。

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

如果你在前面的例子中格式化字符串时用%s 替换%r（产生 repr()值），如果你的系统支持 UTF-8，你应该看到类似下面的输出：

```
ó U+00f3 LATIN SMALL LETTER O WITH ACUTE
ś U+015b LATIN SMALL LETTER S WITH ACUTE
Ś U+015a LATIN CAPITAL LETTER S WITH ACUTE
ą U+0105 LATIN SMALL LETTER A WITH OGONEK
ł U+0142 LATIN SMALL LETTER L WITH STROKE
```

另外，根据你的系统的具体情况，你可能需要用“latin2”替换示例中的编码“utf8”。

下一个例子展示 Python 字符串函数和 re 模块是如何接收 Unicode 字符串的。

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

NLTK 分词器允许 Unicode 字符串作为输入，并输出相应地 Unicode 字符串。

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

在 Python 中使用本地编码

如果你习惯了使用特定的本地编码字符，你可能希望能够在一个 Python 文件中使用你的字符串输入及编辑的标准方法。为了做到这一点，你需要在你的文件的第一行或第二行中包含字符串：'# -*- coding: <coding>-*-' 。请注意，<coding>是一个像'latin-1',

'big5'或者'utf-8'的字符串（见图 3-4）。图 3-4 也展示了正则表达式中如何使用已编码字符串。

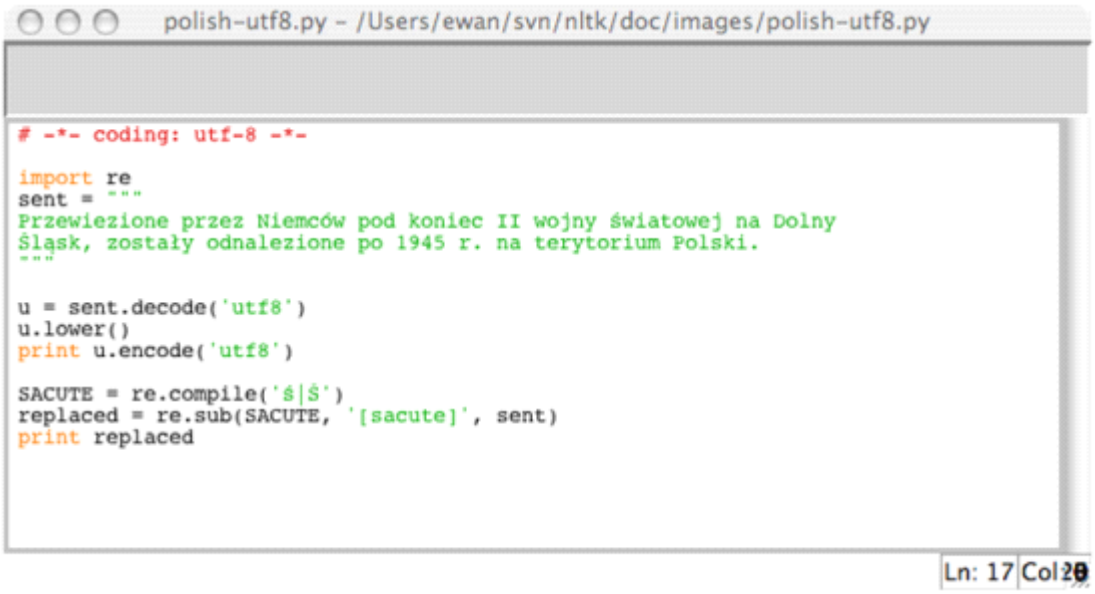


图 3-4. Unicode 与 IDLE：IDLE 编辑器中用 UTF-8 编码字符串常量；这需要在 IDLE 属性中设置了相应的字体；这里我们选择“Courier CE”。

3.4 使用正则表达式检测词组搭配

许多语言处理任务都涉及模式匹配。例如：我们可以使用 endswith('ed')找到以“ed”结尾的词。在表 1-4 中我们可以看到各种“词测试”。正则表达式给我们一个更加强大和灵活的方法描述我们感兴趣的字符模式。

介绍正则表达式的其他出版物有很多，它们围绕正则表达式的语法组织，应用于搜索文本文件。我们不再赘述这些，只专注于在语言处理的不同阶段如何使用正则表达式。像往常一样，我们将采用基于问题的方式，只在解决实际问题需要时才介绍新特性。在我们的讨论中，我们将使用箭头来表示正则表达式，就像这样：«patt»。

在 Python 中使用正则表达式，需要使用 import re 导入 re 函数库。还需要一个用于搜索的词汇链表；我们再次使用*词汇语料库*（2.4 节），对它进行预处理消除某些名称。

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

使用基本的元字符

让我们使用正则表达式«ed$»查找以 ed 结尾的词汇。函数 re.search(p, s)检查字符串 s 中是否有模式 p。我们需要指定感兴趣的字符，然后使用美元符号，它是正则表达式中有特殊用途的符号，用来匹配单词的末尾：

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

通配符“.”匹配任何单个字符。假设我们有一个 8 个字母组成的词的字谜室，j 是其第三个字母，t 是其第六个字母。空白单元格中的每个地方，我们用一个句点：

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

轮到你来：插入符号“^”匹配字符串的开始，就像“$”符号匹配字符串的结尾。如果我们不用这两个符号，刚才例子中我们会得到什么样的结果？

最后，符号“?”表示前面的字符是可选的。因此«^e-?mail $»将匹配 email 和 e-mail。我们可以使用 sum(1 for w in text if re.search('^e-? mail$', w))计数一个文本中这个词（任一拼写形式）出现的总次数。

范围与闭包

T9 系统用于在手机上输入文本（见图 3-5）。两个或两个以上的词汇以相同的击键顺序输入，这叫做**输入法联想提示**。例如：hole 和 golf 都是通过输入序列 4653。还有哪些其它词汇由相同的序列产生？这里我们使用正则表达式«^[ghi][mno][jlk][def]$»：

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

表达式的第一部分：«^[ghi]»匹配以 g、h 或 i 开始的词。表达式的下一部分：«[mno]»限制了第二个字符是 m、n 或 o。第三部分和第四部分也是限制。四个字母要满足所有这些限制。注意，方括号内的字符的顺序是没有关系的，所以我们可以写«^[hig][nom][ljk][fed]$»匹配同样的词汇。

图 3-5. T9：9 个键上的文字。

轮到你来：来看一些“手指绕口令”，只用一部分数字键盘搜索词汇。例如：«^[ghijklmno]+$»，或更为简洁的：«^[g-o]+$»，将匹配只使用中间行的 4、5、6 键的词汇，«^[a-fj-o]+$»将匹配使用右上角 2、3、5、6 键的词汇。“-”和“+”是什么意思？

让我们进一步探索“+”符号。请注意，它可以适用于单个字母或括号内的字母集：

```
>>> 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+$', w)]
['miiiiiiiiiiiiinnnnnnnnnnneeeeeeeee', 'miiiiiinnnnnnnnnneeeeeee', 'mine',
'mmmmmmmmiiiiiiiinnnnnnnnneeeeeeee']
>>> [w for w in chat_words if re.search('^[ha]+$', w)]
['a', 'aaaaaaaaaaaaaaaa', 'aaahhhh', 'ah', 'ahah', 'ahahah', 'ahh',
'ahhahahaha', 'ahhh', 'ahhhh', 'ahhhhhh', 'ahhhhhhhhhhhhhh', 'h', 'ha', 'haaa',
'hah', 'haha', 'hahaaa', 'hahah', 'hahaha', 'hahahaa', 'hahahah', 'hahahaha', ...]
```

很显然，“+”表示的是“前面的项目的一个或多个实例，”它可以是单独的字母如 m，可以是一个集合如[fed]，可以是一个范围如[d-f]。现在让我们用“*”替换“+”，它表示“前面的项目的零个或多个实例。”正则表达式«^m*i*n*e*$»将匹配所有我们用«^m+i+n+e+$»找到的，同时包括其中一些字母不出现的词汇，例如：me、min 和 mmmmm。请注意“+”和“*”符号有时被称为的 **Kleene 闭包**，或者干脆**闭包**。

运算符“^”当它出现在方括号内的第一个字符位置时有另外的功能。例如：«[^aeiouAEIOU]»匹配除元音字母之外的所有字母。我们可以搜索 NPS 聊天语料库中完全由非元音字母组成的词汇，使用«^[^aeiouAEIOU]+$»查找诸如:):)、grrr、cyb3r 和 zzzzzzzz 这样的词。请注意其中包含非字母字符。

下面是另外一些正则表达式的例子，用来寻找匹配特定模式的词汇标识符，这些例子演示如何使用一些新的符号：\，{}，()和|。

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

轮到你来：研究前面的例子，在你继续阅读之前尝试弄清楚\，{}，()和|这些符号的功能。

你可能已经知道反斜杠表示其后面的字母不再有特殊的含义而是按照字面的表示匹配词中特定的字符。因此，虽然“.”有特殊含义，但“\.”值匹配一个句号。大括号表达式，如{3,5}，表示前面的项目重复指定次数。管道字符表示从其左边的内容和右边的内容中选择一个。圆括号表示一个操作符的范围，它们可以与管道（或叫析取）符号一起使用，如：«w(i|e|ai|oo)t»，匹配 wit、wet、wait 和 woot。你可以省略这个例子里的最后一个表达式中的括号，使用«ed|ing$»搜索看看会发生什么，这是很有益处的。

我们所看到的元字符总结在表 3-3 中：

表 3-3. 正则表达式基本元字符，其中包括通配符，范围和闭包

操作符	行为
.	通配符，匹配所有字符
^abc	匹配以 abc 开始的字符串
abc$	匹配以 abc 结尾的字符串
[abc]	匹配字符集合中的一个
[A-Z0-9]	匹配字符一个范围

ed|ing|s	匹配指定的一个字符串（析取）
*	前面的项目零个或多个，如 a*, [a-z]* (也叫 Kleene 闭包)
+	前面的项目 1 个或多个，如 a+, [a-z]+
?	前面的项目零个或 1 个（即：可选）如：a?, [a-z]?
{n}	重复 n 次，n 为非负整数
{n,}	至少重复 n 次
{,n}	重复不多于 n 次
{m,n}	至少重复 m 次不多于 n 次
a(b|c)+	括号表示操作符的范围

对 Python 解释器而言，一个正则表达式与任何其他字符串没有两样。如果字符串中包含一个反斜杠后面跟一些特殊字符，Python 解释器将会特殊处理它们。例如：“\b”会被解释为一个退格符号。一般情况下，当使用含有反斜杠的正则表达式时，我们应该告诉解释器一定不要解释字符串里面的符号，而仅仅是将它直接传递给 re 库来处理。我们通过给字符串加一个前缀“r”来表明它是一个**原始字符串**。例如：原始字符串 r'\band\b'包含两个“\b”符号会被 re 库解释为匹配词的边界而不是解释为退格字符。如果你能逐渐习惯使用 r'...'表示正则表达式——就像从现在开始我们将做的那样——你将会避免去想这些解释上的歧义。

3.5 正则表达式的有益应用

前面的例子都涉及到使用 re.search(regexp, w)匹配一些正则表达式 regexp 来搜索词 w。除了检查一个正则表达式是否匹配一个单词外，我们还可以使用正则表达式从词汇中提取的特征或以特殊的方式来修改词。

提取字符块

通过 re.findall() (“find all”即找到所有)方法找出所有（无重叠的）匹配指定正则表达式的。让我们找出一个词中的元音，再计数它们：

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

让我们来看看一些文本中的两个或两个以上的元音序列，并确定它们的相对频率：

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

轮到你来：在 W3C 日期时间格式中，日期像这样表示：2009-12-31。用正则表达式替换下面 Python 代码中的“?”，将字符串'2009-12-31'转换为一个整数链表[2009, 12, 31]:

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

在字符块上做更多事情

一旦我们会使用 re.findall()从词中提取字符块，就可以在这些字符块上做一些有趣的事情，例如将它们粘贴在一起或用它们绘图。

英文文本是高度冗余的，忽略掉词内部的元音仍然可以很容易的阅读，有些时候这很明显。例如：declaration 变成 dclrtn，inalienable 变成 inlnble，保留所有词首或词尾的元音序列。在下一个例子中，正则表达式匹配词首元音序列，词尾元音序列和所有的辅音；其它的被忽略。这三个阶段从左到右处理，如果词匹配了三个部分之一，正则表达式后面的部分将被忽略。我们使用 re.findall()提取所有匹配的词中的字符，然后使用''.join()将它们连接在一起（见 3.9 节的连接操作）。

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

接下来，让我们将正则表达式与条件频率分布结合起来。在这里，我们将从罗托卡特语词汇中提取所有辅音-元音序列，如 ka 和 si。因为每部分都是成对的，它可以被用来初始化一个条件频率分布。然后我们为每对的频率列表。

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

考查 s 行和 t 行，我们看到它们是部分的“互补分布”，这个证据表明它们不是这种语言中的不同的音素。从而我们可以令人信服的从罗托卡特语字母表中去除 s，简单加入一个

发音规则：当字母 t 跟在 i 后面时发 s 的音。（注意单独的条目“su”即 kasuari，“cassowary”是从英语中借来的）。

如果我们想要检查表格中数字背后的词汇，有一个索引允许我们迅速找到包含一个给定的辅音-元音对的单词的列表将会有帮助。例如：cv_index['su']应该给我们所有含有“su”的词汇。下面是我们如何能做到这一点：

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

这段代码依次处理每个词 w，对每一个词找出匹配正则表达式«[ptksvr][aeiou]»的所有子字符串。对于词 kasuari，它找到 ka、su 和 ri。从而，链表 cv_word_pairs 将包含('ka', 'kasuari')、('su', 'kasuari')和('ri', 'kasuari')。更进一步使用 nltk.Index()转换成有用的索引。

查找词干

在使用网络搜索引擎时，我们通常不介意（甚至没有注意到）文档中的词汇与我们的搜索条件的后缀形式是否相同。查询“laptops”会找到含有“laptop”的文档，反之亦然。事实上，“laptop”与“laptops”只是字典中的同一个词（或词条）的两种形式。

对于一些语言处理任务，我们想忽略词语结尾，只是处理词干。抽出一个词的词干的方法有很多种。这里的是一种简单直观的方法，直接去掉任何看起来像一个后缀的字符：

```
>>> 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
```

虽然我们最终将使用 NLTK 中内置的词干，看看我们如何能够使用正则表达式处理这个任务是有趣的。我们的第一步是建立一个所有后缀的连接。我们需要把它放在括号内以限制这个连接的范围。

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

在这里，尽管正则表达式匹配整个单词，re.findall()只是给我们后缀。这是因为括号有第二个功能：选择要提取的子字符串。如果我们要使用括号来指定连接的范围，但不想选择要输出的字符串，必须添加“?:”，它是许多神秘奥妙的正则表达式之一。下面是改进后的版本：

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

然而，实际上，我们会想将词分成词干和后缀。所以，我们应该只是用括号括起正则表达式的这两个部分：

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

```
[('process', 'ing')]
```

这看起来很有用途，但仍然有一个问题。让我们来看看另外的词“processes”：

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

正则表达式错误地找到了后缀“-s”，而不是后缀“-es”。这表明另一个微妙之处：“*”操作符是“贪婪的”，所以表达式的“.*”部分试图尽可能多的匹配输入的字符串。如果我们使用“非贪婪”版本的“*”操作符，写成“*?”，我们就得到我们想要的：

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

我们甚至可以通过使第二个括号中的内容变成可选，来得到空后缀：

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

这种方法仍然有许多问题，（你能发现它们吗？）但我们仍将继续定义一个函数来获取词干，并将它应用到整个文本：

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

请注意我们的正则表达式将“ponds”的“s”删除，但也将“basis”的“is”删除。它产生一些非词（未被确认、收录的词）如 distribut 与 deriv，但这些都是在一些应用中可接受的词干。

搜索已分词文本

你可以使用一种特殊的正则表达式搜索一个文本中多个词（这里的文本是一个标识符列表）。例如：“<a> <man>”找出文本中所有“a man”的实例。尖括号用于标记标识符的边界，尖括号之间的所有空白都被忽略（这只对 NLTK 中的 findall()方法处理文本有效）。在下面的例子中，我们使用<.*>①，它将匹配所有单个标识符，将它括在括号里，于是只匹配词（例如：monied）而不匹配短语（例如：a monied man）。第二个例子找出以词“bro”结尾的三个词组成的短语②。最后一个例子找出以字母“l”开始的三个或更多词组成的序列③。

```
>>> from nltk.corpus import gutenberg, nps_chat
>>> moby = nltk.Text(gutenberg.words('melville-moby_dick.txt'))
>>> moby.findall(r"<a> (<.*>) <man>") ①
```

```
monied; nervous; dangerous; white; white; white; pious; queer; good;
mature; white; Cape; great; wise; wise; butterless; white; fiendish;
pale; furious; better; certain; complete; dismasted; younger; brave;
brave; brave; brave
>>> chat = nltk.Text(nps_chat.words())
>>> chat.findall(r"<.*> <.*> <bro>") ②
you rule bro; telling you bro; u twizted bro
>>> chat.findall(r"<l.*>{3,}") 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
```

轮到你来：巩固你对正则表达式模式与替换的理解，使用 nltk.re_show(p, s)，它能标注字符串 s 中所有匹配模式 p 的地方，以及 nltk.app.nemo()，它能提供一个探索正则表达式的图形界面。更多的练习，可以尝试本章结尾的正则表达式的一些练习。

当我们研究的语言现象与特定词语相关时建立搜索模式是很容易的。在某些情况下，一个小小的创意可能会花很大功夫。例如：在大型文本语料库中搜索“x and other ys”形式的表达式能让我们发现上位词（见 2.5 节）：

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

只要有足够多的文本，这种做法会给我们一整套有用的分类标准信息，而不需要任何手工劳动。然而，我们的搜索结果中通常会包含误报，即我们想要排除的情况。例如，结果“demands and other factors”表明“demands”是类型“factor”的一个实例，但是这句话实际上是关于要求增加工资的。尽管如此，我们仍可以通过手工纠正这些搜索的结果来构建自己的英语概念本体。

这种自动和人工处理相结合的方式是最常见的建造新的语料库的方式。我们将在第 11 章继续讲述这些。

搜索语料也会有遗漏的问题，即漏掉了我们想要包含的情况。仅仅因为我们找不到任何一个搜索模式的实例，就断定一些语言现象在一个语料库中不存在，是很冒险的。也许我们只是没有足够仔细的思考合适的模式。

轮到你来：查找模式“as x as y”的实例以发现实体及其属性信息。

3.6 规范化文本

在前面的程序例子中，我们在处理文本词汇前经常要将文本转换为小写，即 set(w.lo

wer() for w in text)。通过使用 lower()我们将文本**规范化**为小写，这样一来“The”与“the”的区别被忽略。我们常常想比这走得更远，例如：去掉所有的词缀以及提取词干的任务等。更进一步的步骤是确保结果形式是字典中确定的词，即叫做词形归并的任务。我们依次讨论这些。首先，我们需要定义我们将在本节中使用的数据：

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

词干提取器

NLTK 中包括了一些现成的词干提取器，如果你需要一个词干提取器，你应该优先使用它们中的一个，而不是使用正则表达式制作自己的词干提取器，因为 NLTK 中的词干提取器能处理的不规则的情况很广泛。Porter 和 Lancaster 词干提取器按照它们自己的规则剥离词缀。请看 Porter 词干提取器正确处理了词 lying（将它映射为 lie），而 Lancaster 词干提取器并没有处理好。

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

词干提取过程没有明确定义，我们通常选择心目中最适合我们的应用的词干提取器。如果你要索引一些文本和使搜索支持不同词汇形式的话，Porter 词干提取器是一个很好的选择（例 3-1 所示，它采用了面向对象编程技术，这超出了本书的范围，字符串格式化技术将在 3.9 节讲述，enumerate()将在 4.2 节解释）。

例 3-1. 使用词干提取器索引文本。

```
class IndexedText(object):
    def __init__(self, stemmer, text):
        self._text = text
        self._stemmer = stemmer
        self._index = nltk.Index((self._stem(word), i)
                                 for (i, word) in enumerate(text))
    def concordance(self, word, width=40):
        key = self._stem(word)
        wc = width/4                    # words of context
        for i in self._index[key]:
            lcontext = ' '.join(self._text[i-wc:i])
```

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

词形归并

WordNet 词形归并器删除词缀产生的词都是在它的字典中的词。这个额外的检查过程使词形归并器比刚才提到的词干提取器要慢。请注意，它并没有处理“lying”，但它将“women”转换为“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', '.']
```

如果你想编译一些文本的词汇，或者想要一个有效词条（或中心词）列表，WordNet词形归并器是一个不错的选择。

另一个规范化任务涉及识别**非标准词**，包括数字、缩写、日期以及任何此类标识符到一个特殊的词汇的映射。例如：每一个十进制数可以被映射到一个单独的标识符 0.0，每首字母缩写可以映射为 AAA。这使词汇量变小，提高了许多语言建模任务的准确性。

3.7 用正则表达式为文本分词

分词是将字符串切割成可识别的构成一块语言数据的语言单元。虽然这是一项基础任务，我们能够一直拖延到现在为止才讲，是因为许多语料库已经分过词了，也因为 NLTK

中包括一些分词器。现在你已经熟悉了正则表达式，你可以学习如何使用它们来为文本分词，并对此过程中有更多的掌控权。

分词的简单方法

文本分词的一种非常简单的方法是在空格符处分割文本。考虑以下摘自《爱丽丝梦游仙境》中的文本：

```
>>> 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,'..."""
```

我们可以使用 raw.split() 在空格符处分割原始文本。使用正则表达式能做同样的事情，匹配字符串中的所有空格符①是不够的，因为这将导致分词结果包含“\n”换行符；我们需要匹配任何数量的空格符、制表符或换行符②：

```
>>> re.split(r' ', raw) ①
['"When", "I'M", 'a', "Duchess,"', 'she', 'said', 'to', 'herself,', '(not', 'in',
'a', 'very', 'hopeful', 'tone\nthough),', '"I", "won't", 'have', 'any', 'pepper',
'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup', 'does', 'very\nwell', 'without--Maybe',
"it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-tempered,'..."]
>>> re.split(r'[ \t\n]+', raw) ②
['"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,'..."]
```

正则表达式«[\t\n]+»匹配一个或多个空格、制表符（\t）或换行符（\n）。其他空白字符，如回车和换页符，确实应该包含的太多。于是，我们将使用一个 re 库内置的缩写“\s”，它表示匹配所有空白字符。前面的例子中第二条语句可以改写为 re.split(r'\s+', raw)。

要点：记住在正则表达式前加字母“r”，它告诉 Python 解释器按照字面表示对待字符串而不去处理正则表达式中包含的反斜杠字符。

在空格符处分割文本给我们如“(not”和“herself,”这样的标识符。另一种方法是使用 Python 提供给我们的字符类“\w”匹配词中的字符，相当于[a-zA-Z0-9_]。也定义了这个类的补“\W”即所有字母、数字和下划线以外的字符。我们可以在一个简单的正则表达式中用\W 来分割所有单词字符以外的输入。

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

可以看到，在开始和结尾都给了我们一个空字符串（要了解原因请尝试'xx'.split('x')）。通过 re.findall(r'\w+', raw)使用模式匹配词汇而不是空白符号，我们得到相同的标识符，但没有空字符串。现在，我们正在匹配词汇，我们处在扩展正则表达式覆盖更广泛的情况的位置。正则表达式«\w+|\S\w*»将首先尝试匹配词中字符的所有序列。如果没有找到匹配的，它会尝试匹配后面跟着词中字符的任何非空白字符（“\S”是“\s”的补）。这意味着

标点会与跟在后面的字母（如's）在一起，但两个或两个以上的标点字符序列会被分割。

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

让我们扩展前面表达式中的“\w+”，允许连字符和撇号：«\w+([-']\w+)*»。这个表达式表示“\w+”后面跟零个或更多“[-']\w+”的实例；它会匹配 hot-tempered 和 it's。（我们需要在这个表达式中包含“?:”，原因前面已经讨论过。）我们还将添加一个模式来匹配引号字符让它们与它们包括的文字分开。

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

在这个例子中的表达式也包括«[-.(]+»，这会使双连字符、省略号和左括号被单独分词。

表 3-4 列出了我们已经在本节中看到的正则表达式字符类符号，以及一些其他有用的符号。

表 3-4. 正则表达式符号

符号	功能
\b	词边界（零宽度）
\d	任一十进制数字（相当于[0-9]）
\D	任何非数字字符（等价于[^ 0-9]）
\s	任何空白字符（相当于[\t\n\r\f\v]）
\S	任何非空白字符（相当于[^ \t\n\r\f\v]）
\w	任何字母数字字符（相当于[a-zA-Z0-9_]）
\W	任何非字母数字字符（相当于[^a-zA-Z0-9_]）
\t	制表符
\n	换行符

NLTK 的正则表达式分词器

函数 nltk.regexp_tokenize()与 re.findall()类似（我们一直在使用它进行分词）。然而，nltk.regexp_tokenize()分词效率更高，且不需要特殊处理括号。为了增强可读性，我们将正则表达式分几行写，每行添加一个注释。特别的“(?x)”“verbose 标志”告诉 Python 去掉嵌入的空白字符和注释。

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

```
...     | \.\.\.                  # ellipsis
...     | [][.,;'"?():-_`]   # these are separate tokens
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

使用 verbose 标志时，可以不再使用' '来匹配一个空格字符；使用“\s”代替。regexp_tokenize()函数有一个可选的 gaps 参数。设置为 True 时，正则表达式指定标识符间的距离，就像使用 re.split()一样。

我们可以使用 set(tokens).difference(wordlist)，通过比较分词结果与一个词表，然后报告任何没有在词表出现的标识符，来评估一个分词器。你可能想先将所有标记变成小写。

分词的进一步问题

现在，分词是一个比你可能预期的要更为艰巨的任务。没有单一的解决方案能在所有领域都行之有效，我们必须根据应用领域的需要决定那些是标识符。

在开发分词器时，访问已经手工标注好的原始文本是有益的，这可以让你的分词器的输出结果与高品质（或称“黄金标准”）的标注进行比较。NLTK 语料库集合包括宾州树库的数据样本，包括《华尔街日报》原始文本（nltk.corpus.treebank_raw.raw()）和分好词的版本（nltk.corpus.treebank.words()）。

分词的最后一个问题是缩写的存在，如“didn't”。如果我们想分析一个句子的意思，将这种形式规范化为两个独立的形式：“did”和“n't”(不是 not)可能更加有用。我们可以通过查表来做这项工作。

3.8 分割

本节将讨论更高级的概念，你在第一次阅读本章时可能更愿意跳过本节。

分词是一个更普遍的**分割**问题的一个实例。在本节中，我们将看到这个问题的另外两个实例，它们使用与到目前为止我们已经在本章看到的完全不同的技术。

断句

在词级水平处理文本通常假定能够将文本划分成单个句子。正如我们已经看到，一些语料库已经提供在句子级别的访问。在下面的例子中，我们计算布朗语料库中每个句子的平均词数：

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

在其他情况下，文本可能只是作为一个字符流。在将文本分词之前，我们需要将它分割成句子。NLTK 通过包含 Punkt 句子分割器(Kiss & Strunk, 2006)简化了这些。这里是使用它为一篇小说文本断句的例子。（请注意，如果在你读到这篇文章时分割器内部数据已经更

新过，你会看到不同的输出。）

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

请注意，这个例子其实是一个单独的句子，报道 Lucian Gregory 先生的演讲。然而，引用的演讲包含几个句子，这些已经被分割成几个单独的字符串。这对于大多数应用程序都是合理的行为。

断句是困难的，因为句号会被用来标记缩写而另一些句号同时标记缩写和句子结束，就像发生在缩写如“U.S.A.”上的那样。断句的另一种方法见 6.2 节。

分词

对于一些书写系统，由于没有词边界的可视表示这一事实，文本分词变得更加困难。

例如：在中文中，三个字符的字符串：爱国人(ai4 “love” [verb], guo3 “country”, ren2 “person”) 可以被分词为“爱国/人”，“country-loving person”，或者“爱/国人”，“love country-person”。

类似的问题在口语语言处理中也会出现，听者必须将连续的语音流分割成单个的词汇。当我们事先不认识这些词时，这个问题就演变成一个特别具有挑战性的版本。语言学习者会面对这个问题，例如小孩听父母说话。

考虑下面的人为构造的例子，单词的边界已被去除：

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

我们的第一个挑战仅仅是表示这个问题：我们需要找到一种方法来分开文本内容与分词标志。我们可以给每个字符标注一个布尔值来指示这个字符后面是否有一个分词标志（这个想法将在第 7 章“分块”中大量使用）。让我们假设说话人会给语言学习者一个说话时的停顿，这往往是对应一个延长的暂停。这里是一种表示方法，包括初始的分词和最终分词目标。

```
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "0000000000000001000000000010000000000000000100000000000"
>>> seg2 = "0100100100100001001001000010100100010010000100010010000"
```

观察由 0 和 1 组成的分词表示字符串。它们比源文本短一个字符，因为长度为 n 文本可以在 n-1 个地方被分割。例 3-2 中的函数 segment()演示了我们可以从这个表示回到初始

分词的文本。

例 3-2. 从分词表示字符串 seg1 *和* seg2 *中重建文本分词。* seg1 *和* seg2 *表示假设的一些儿童讲话的初始和最终分词。函数* segment() *可以使用它们重现分词的文本。*

```
def segment(text, segs):
    words = []
    last = 0
    for i in range(len(segs)):
        if segs[i] == '1':
            words.append(text[last:i+1])
            last = i+1
    words.append(text[last:])
    return words
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "000000000000000100000000001000000000000000010000000000"
>>> seg2 = "010010010010000100100100001010010001001000010001001000"
>>> segment(text, seg1)
['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']
>>> segment(text, seg2)
['do', 'you', 'see', 'the', 'kitty', 'see', 'the', 'doggy', 'do', 'you',
 'like', 'the', kitty', 'like', 'the', 'doggy']
```

现在分词的任务变成了一个搜索问题：找到将文本字符串正确分割成词汇的字位串。我们假定学习者接收词，并将它们存储在一个内部词典中。给定一个合适的词典，是能够由词典中的词的序列来重构源文本的。读过(Brent & Cart-wright, 1995)之后，我们可以定义一个**目标函数**，一个打分函数，我们将基于词典的大小和从词典中重构源文本所需的信息量尽力优化它的值。我们在图 3-6 中说明了这些。

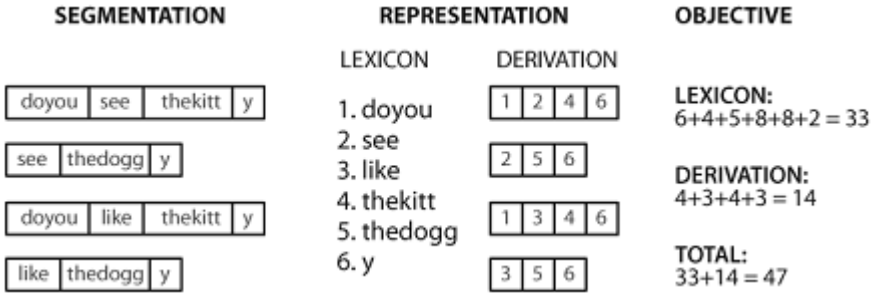


图 3-6. 计算目标函数：给定一个假设的源文本的分词（左），推导出一个词典和推导表，它能让源文本重构，然后合计每个词项（包括边界标志）与推导表的字符数，作为分词质量的得分；得分值越小表明分词越好。

实现这个目标函数是很简单的，如例子 3-3 所示。

例 3-3. 计算存储词典和重构源文本的成本。

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

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

最后一步是寻找最大化目标函数值的 0 和 1 的模式，例 3-4 中所示。请注意，最好的分词包括像“thekitty”这样的“词”，因为数据中没有足够的证据进一步分割这个词。

例 3-4. 使用模拟退火算法的非确定性搜索：一开始仅搜索短语分词；随机扰动 0 和 1，它们与“温度”成比例；每次迭代温度都会降低，扰动边界会减少。

```
from random import randint
def flip(segs, pos):
    return segs[:pos] + str(1-int(segs[pos])) + segs[pos+1:]
def flip_n(segs, n):
    for i in range(n):
        segs = flip(segs, randint(0,len(segs)-1))
    return segs
def anneal(text, segs, iterations, cooling_rate):
    temperature = float(len(segs))
    while temperature > 0.5:
        best_segs, best = segs, evaluate(text, segs)
        for i in range(iterations):
            guess = flip_n(segs, int(round(temperature)))
            score = evaluate(text, guess)
            if score < best:
                best, best_segs = score, guess
        score, segs = best, best_segs
        temperature = temperature / cooling_rate
        print evaluate(text, segs), segment(text, segs)
    print
    return segs
>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"
>>> seg1 = "0000000000000001000000000010000000000000000100000000000"
>>> anneal(text, seg1, 5000, 1.2)
60 ['doyouseetheki', 'tty', 'see', 'thedoggy', 'doyouliketh', 'ekittylike', 'thedoggy']
58 ['doy', 'ouseetheki', 'ttysee', 'thedoggy', 'doy', 'o', 'ulikethekittylike', 'thedoggy']
56 ['doyou', 'seetheki', 'ttysee', 'thedoggy', 'doyou', 'liketh', 'ekittylike', 'thedoggy']
54 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'likethekittylike', 'thedoggy']
```

```
53 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']
51 ['doyou', 'seethekittysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']
42 ['doyou', 'see', 'thekitty', 'see', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']
'000010010000000100100000001000010001000000010001000000'
```

有了足够的数据，就可能以一个合理的准确度自动将文本分割成词汇。这种方法可用于为那些词的边界没有任何视觉表示的书写系统分词。

3.9 格式化：从链表到字符串

我们经常会写程序输出一个单独的数据项，例如一个语料库中满足一些复杂的标准的特定的元素，或者一个单独的总数统计，例如一个词计数器或一个标注器的性能。更多的时候，我们写程序来产生一个结构化的结果；例如：一个数字或语言形式的表格，或原始数据的格式变换。当要表示的结果是语言时，文字输出通常是最自然的选择。然而当结果是数值时，可能最好是图形输出。在本节中，你将会学到呈现程序输出的各种方式。

从链表到字符串

我们用于文本处理的最简单的一种结构化对象是词链表。当我们希望把这些输出到显示器或文件时，必须把这些词的链表转换成字符串。在 Python 做这些，我们使用的 join()方法，并指定作为“胶水”使用的字符串：

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

' '.join(silly)的意思是：取出 silly 中的所有项目，将它们连接成一个大的字符串，使用' '作为项目之间的间隔符，即 join()是一个你想要用来作为胶水的字符串的一个方法。（许多人感到 join()的这种表示方法是违反直觉的。）join()方法只适用于一个字符串的链表——我们一直把它叫做一个文本——在 Python 中享有某些特权的一个复杂的类型。

字符串与格式

我们已经看到了有两种方式显示一个对象的内容：

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

```
world
>>> word
'cat'
>>> sentence
'hello\nworld'
```

print 命令产生的是 Python 尝试以人最可读的形式输出的一个对象的内容。第二种方法——叫做变量提示——向我们显示可用于重新创建该对象的字符串。重要的是要记住这些都仅仅是字符串，为了你——用户——的方便而显示的。它们并不会给我们实际对象的内部表示的任何线索。

还有许多其他有用的方法来将一个对象作为字符串显示。这可能是为了人阅读的方便，或是因为我们希望**导出**我们的数据到一个特定的能被外部程序使用的文件格式。

格式化输出通常包含变量和预先指定的字符串的一个组合。例如：给定一个频率分布 fdist，我们可以这样做：

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

除了不必要的空格符问题，输出包含变量和常量交替出现的表达式是难以阅读和维护的。一个更好的解决办法是使用**字符串格式化表达式**。

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

要了解这里发生了什么事情，让我们在字符串格式化表达式上面测试一下。（现在，这将是你探索新语法的常用方法。）

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

特殊符号%s 和%d 是字符串和整数（十进制数）的占位符。我们可以将这些嵌入在一个字符串中，然后使用%操作符把它们组合起来。让我们更深入的解开这段代码，以便更仔细的观察它的行为：

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

我们可以有多个占位符，但%操作符后面必须指定数目完全相同的数值的元组：

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

我们还可以间接的提供占位符的值。下面是使用 for 循环的一个例子：

```
>>> template = 'Lee wants a %s right now'
```

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

符号%s 和%d 被称为**转换说明符**。它们以%字符开始，以一个转换字符如 s（表示字符串）或 d（十进制整数）结束。其中包含转换说明符的字符串被称为**格式字符串**。我们组合一个格式字符串和%操作符以及一个值的元组，来创建一个完整的字符串格式化表达式。

排列

到目前为止，我们的格式化字符串可以在页面（或屏幕）上输出任意的宽度，如%s 和%d。我们也可以指定宽度，如%6s，产生一个宽度为 6 的字符串。缺省情况它是右对齐的①，但我们可以包括一个减号使它左对齐②。我们事先不知道要显示的值应该有多宽时，可以在格式化字符串中用*替换宽度值，然后指定一个变量③。

```
>>> '%6s' % 'dog' ①
' dog'
>>> '%-6s' % 'dog' ②
'dog '
>>> width = 6
>>> '%-*s' % (width, 'dog') ③
'dog
```

其他控制字符用于十进制整数和浮点数。因为百分号%在格式化字符串中有特殊解释，我们要在它前面加另一个%才能输出它。

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

格式化字符串的一个重要用途是用于数据制表。回想一下，在 2.1 节中，我们看到从条件频率分布中制表的数据。让我们自己来制表，行使对标题和列宽的完全控制，如例 3-5 所示。注意语言处理工作与结果制表之间是明确分离的。

例 3-5. 布朗语料库的不同部分的频率模型

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

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

回顾在例 3-1 列出的，我们使用了格式化字符串“%*s”。这使我们可以使用变量指定一个字段的宽度。

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

我们可以使用 width = max(len(w) for w in words)自动定制列的宽度，使其足够容纳所有的词。要记住 print 语句结尾处的逗号增加了一个额外的空格，这样能够防止列标题相互重叠。

将结果写入文件

我们已经看到了如何读取文本文件（3.1 节）。将输出写入文件往往也很有用。下面的代码打开可写文件 output.txt，将程序的输出保存到文件。

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

轮到你来：在写入文件之前，我们为每个字符串追加\n 会有什么影响？如果你使用的是 Windows 机器，你可能想用 word + "\r\n"代替。如果我们输入 output_file.write(word)会发生什么？

当我们将非文本数据写入文件时，我们必须先将它转换为字符串。正如我们前面所看到的，可以使用格式化字符串来做这一转换。关闭文件之前，让我们把总词数写入我们的文件。

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

注意！

你应该避免文件名包含空格字符，例如：output file.txt，避免使用除了大小写区别外其他都相同的文件名，例如：Output.txt 与 output.TXT。

文本换行

当程序的输出是文档式的而不是像表格时，通常会有必要包装一下以便可以方便地显示它。考虑下面的输出，它的行尾溢出了，且使用了一个复杂的 print 语句。

```
>>> 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),
```

我们可以在 Python 的 textwrap 模块的帮助下采取换行。为了最大程度的清晰，我们将每一个步骤分在一行：

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

请注意，在 more 与其下面的数字之间有一个换行符。如果我们希望避免这种情况，可以重新定义格式化字符串，使它不包含空格（例如：'%s_(%d),'），然后不输出 wrapped 的值而是输出 wrapped.replace('_', ' ')。

3.10 小结

- 在本书中，我们将文本作为一个词链表。“原始文本”是一个潜在的长字符串，其中包含文字和用于设置格式的空白字符，也是我们通常存储和可视化文本的原料。
- 在 Python 中指定一个字符串使用单引号或双引号：'Monty Python'，"Monty Python"。
- 字符串中的字符是使用索引来访问的，索引从零计数：'Monty Python'[0]的值是 M。求字符串的长度可以使用 len()。
- 子字符串使用切片符号来访问： 'Monty Python'[1:5]的值是 onty。如果省略起始索引，子字符串从字符串的开始处开始；如果省略结尾索引，切片会一直到字符串的结尾处结束。
- 字符串可以被分割成链表：'Monty Python'.split()得到['Monty', 'Python']。链表可以连接成字符串：'/'.join(['Monty', 'Python'])得到'Monty/Python'。
- 我们可以使用 text = open(f).read()从一个文件 f 读取文本。可以使用 text = url

open(u).read()从一个 URL u 读取文本。我们可以使用 for line in open(f)遍历一个文本文件的每一行。

- 在网上找到的文本可能包含不需要的内容（如页眉、页脚和标记），在我们做任何语言处理之前需要去除它们。
- 分词是将文本分割成基本单位或标记，例如词和标点符号等。基于空格符的分词对于许多应用程序都是不够的，因为它会捆绑标点符号和词。NLTK 提供了一个现成的分词器 nltk.word_tokenize()。
- 词形归并是一个过程，将一个词的各种形式（如：appeared，appears）映射到这个词标准的或引用的形式，也称为词位或词元（如：appear）。
- 正则表达式是用来指定模式的一种强大而灵活的方法。只要导入了 re 模块，我们就可以使用 re.findall()来找到一个字符串中匹配一个模式的所有子字符串。
- 如果一个正则表达式字符串包含一个反斜杠，你应该使用原始字符串与一个 r 前缀：r'regexp'，告诉 Python 不要预处理这个字符串。
- 当某些字符前使用了反斜杠时，例如：\n，处理时会有特殊的含义（换行符）；然而，当反斜杠用于正则表达式通配符和操作符时，如：\.，\|，\$，这些字符失去其特殊的含义，只按字面表示匹配。
- 一个字符串格式化表达式 template % arg_tuple 包含一个格式字符串 template，它由如%-6s 和%0.2d 这样的转换标识符符组成。

3.11 深入阅读

本章的额外材料发布在 http://www.nltk.org/，包括网络上免费提供的资源的链接。记得在 http://docs.python.org/咨询 Python 的参考材料。（例如：此文档涵盖“通用换行符支持”，解释了各种操作系统如何规定不同的换行符。）

更多的使用 NLTK 处理词汇的例子请参阅 http://www.nltk.org/howto 上的分词、词干提取以及语料库 HOWTO 文档。(Jurafsky & Martin, 2008)的第 2、3 章包含正则表达式和形态学的更高级的材料。Python 文本处理更广泛的讨论请参阅(Mertz, 2003)。规范非标准词的信息请参阅(Sproat et al., 2001)。

关于正则表达式的参考材料很多，无论是理论的还是实践的。在 Python 中使用正则表达式的一个入门教程，请参阅 Kuchling’s Regular Expression HOWTO，http://www.amk.ca/python/howto/regex/。

关于使用正则表达式的全面而详细的手册，请参阅(Friedl, 2002)，其中涵盖包括 Python 在内大多数主要编程语言的语法。其他材料还包括(Jurafsky & Martin, 2008)的第 2.1 节，(Mertz, 2003)的第 3 章。

网上有许多关于 Unicode 的资源。以下是与处理 Unicode 的 Python 的工具有关的有益的讨论：

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

SIGHAN，ACL 中文语言处理特别兴趣小组（http://sighan.org/），重点关注中文文本分

词的问题。我们分割英文文本的方法依据(Brent & Cartwright, 1995)；这项工作属于语言获取领域(Niyogi, 2006)。

搭配是**多词表达式**的一种特殊情况。一个多词表达式是一个小短语，仅从它的词汇不能预测它的意义和其他属性，例如：part-of-speech(Baldwin & Kim, 2010)。

模拟退火是一种启发式算法，找寻在一个大型的离散的搜索空间上的一个函数的最佳值的最好近似，基于对金属冶炼中的退火的模拟。该技术在许多人工智能文本中都有描述。(Hearst, 1992)描述了使用如 x and other ys 的搜索模式发现文本中下位词的方法。

3.12 练习

1. ○定义一个字符串 `s = 'colorless'`。写一个 Python 语句将其变为“colourless”，只使用切片和连接操作。
2. ○我们可以使用切片符号删除词汇形态上的结尾。例如：`'dogs'[:-1]`删除了 `dogs` 的最后一个字符，留下 `dog`。使用切片符号删除下面这些词的词缀（我们插入了一个连字符指示词缀的边界，请在你的字符串中省略掉连字符）： dish-es, run-ning, nation-ality, un-do, pre-heat。
3. ○我们看到如何产生一个 IndexError，通过索引超出一个字符串的末尾。构造一个向左走的太远走到字符串的前面的索引，这有可能吗？
4. ○我们可以为分片指定一个“步长”。下面的表达式间隔一个字符返回一个片内字符：monty[6:11:2]。也可以反向进行：monty[10:5:-2]。自己尝试一下，然后实验不同的步长。
5. ○如果你让解释器处理 monty[::-1]会发生什么？解释为什么这是一个合理的结果。
6. ○说明以下的正则表达式匹配的字符串类：
   a. [a-zA-Z]+
   b. [A-Z][a-z]*
   c. p[aeiou]{,2}t
   d. \d+(\.\d+)?
   e. ([^aeiou][aeiou][^aeiou])*
   f. \w+|[^\w\s]+
   使用 nltk.re_show()测试你的答案。
7. ○写正则表达式匹配下面字符串类：
   a. 一个单独的限定符（假设只有 a，an 和 the 为限定符）
   b. 整数加法和乘法的算术表达式，如：2*3+8。
8. ○写一个工具函数以 URL 为参数，返回删除所有的 HTML 标记的 URL 的内容。使用 urllib.urlopen 访问的 URL 的内容，例如：raw_contents = urllib.urlopen('http://www.nltk.org/').read()
9. ○将一些文字保存到文件 corpus.txt。定义一个函数 load(f)以要读取的文件名为其唯一参数，返回包含文件中文本的字符串。
   a. 使用 nltk.regexp_tokenize()创建一个分词器分割这个文本中的各种标点符号。使用一个多行的正则表达式，行内要有注释，使用 verbose 标志(?x)。
   b. 使用 nltk.regexp_tokenize()创建一个分词器分割以下几种表达式：货币金额；日期；个人和组织的名称。
10. ○将下面的循环改写为链表推导：

```
>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']
>>> result = []
```

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

11. ○定义一个字符串 raw 包含你自己选择的句子。现在，以空格以为的其它字符分割 raw，例如：‘s’。
12. ○写一个 for 循环输出一个字符串的字符，每行一个。
13. ○不带参数的 split 与以' '作为参数的 split，即 sent.split()与 sent.split(' ')的区别是什么？当被分割的字符串包含制表符、连续的空格或一个制表符与空格的序列会发生什么？（在 IDLE 中你将需要使用'\t'来输入制表符。）
14. ○创建一个变量 words，包含一个词链表。实验 words.sort()和 sorted(words)。它们有什么区别？
15. ○通过在 Python 提示符输入以下表达式，探索字符串和整数的区别："3" * 7 和 3 * 7。尝试使用 int("3")和 str(3)进行字符串和整数之间的转换。
16. ○早些时候，我们要求你用文本编辑器创建一个名为 test.py 的文件，只包含一行 monty = 'Monty Python'。如果你还没有这样做（或无法找到该文件），去吧！现在就做。接下来，打开一个新的 Python 会话，并在提示符下输入表达式 monty。你会从解释器得到一个错误。现在，请尝试以下代码（注意你要丢弃文件名中的.py）：

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

    这一次，Python 应该返回一个值。你也可以尝试 import test，在这种情况下，Python 应该能够处理提示符处的表达式 test.monty。
17. ○格式化字符串%6s 与%-6s 用来显示长度大于 6 个字符的字符串时，会发生什么？
18. ◑阅读语料库中的一些文字，为它们分词，输出其中出现的所有 wh-类型词的列表。（英语中的 wh-类型词被用在疑问句，关系从句和感叹句：who，which，what 等。）按顺序输出它们。在这个列表中有因为有大小写或标点符号的存在而重复的词吗？
19. ◑创建一个文件，包含词汇和（任意指定）频率，其中每行包含一个词，一个空格和一个正整数，如：fuzzy 53。使用 open(filename).readlines()将文件读入 Python 链表。接下来，使用 split()将每一行分成两个字段，并使用 int()将其中的数字转换为一个整数。结果要是一个链表形式：[['fuzzy', 53], ...]。
20. ◑编写代码来访问喜爱的网页，并从中提取一些文字。例如，访问一个天气网站，提取你所在的城市今天的最高温度预报。
21. ◑写一个函数 unknown()，以一个 URL 为参数，返回一个那个网页出现的未知词链表。为了做到这一点，请提取所有由小写字母组成的子字符串(使用 re.findall())，并去除所有在 Words 语料库中出现的项目（nltk.corpus.words）。尝试手动分类这些词，并讨论你的发现。
22. ◑使用上面建议的正则表达式处理网址 http://news.bbc.co.uk/，检查处理结果。你会看到那里仍然有相当数量的非文本数据，特别是 JavaScript 命令。你可能还会发现句子分割没有被妥善保留。定义更深入的正则表达式，改善此网页文本的提取。
23. ◑你能写一个正则表达式以这样的方式来分词吗？将词 don't 分为 do 和 n't？解释为什么这个正则表达式无法正常工作：«n't|\w+»。
24. ◑尝试编写代码将文本转换成 hAck3r，使用正则表达式和替换，其中 e→3，i→1，o→0，l→|，s→5，.→5w33t!，ate→8。在转换之前将文本规范化为小写。自己添加更多的

替换。现在尝试将 s 映射到两个不同的值：词开头的 s 映射为$，词内部的 s 映射为 5。

25. ◑Pig Latin 是英语文本的一个简单的变换。文本中每个词的按如下方式变换：将出现在词首的所有辅音（或辅音群）移到词尾，然后添加 ay，例如：string→ingstray，idle→idleay（见 http://en.wikipedia.org/wiki/Pig_Latin）。
    a. 写一个函数转换一个词为 Pig Latin。
    b. 写代码转换文本而不是单个的词。
    c. 进一步扩展它，保留大写字母，将 qu 保持在一起（例如：这样 quiet 会变成 ietquay），并检测 y 是作为一个辅音（如：yellow）还是元音（如：style）。
26. ◑下载一种包含元音和谐的语言（如匈牙利语）的一些文本，提取词汇的元音序列，并创建一个元音二元语法表。
27. ◑Python 的 random 模块包括函数 choice()，它从一个序列中随机选择一个项目；例如：choice("aehh ")会产生四种可能的字符中的一个，字母 h 的几率是其它字母的两倍。写一个表达式产生器，从字符串"aehh "产生 500 个随机选择的字母的序列，并将这个表达式写入函数"".join()调用中，将它们连接成一个长字符串。你得到的结果应该看起来像失去控制的喷嚏或狂笑：he haha ee heheeh eha。使用 split()和 join()再次规范化这个字符串中的空格。
28. ◑考虑下面的摘自 MedLine 语料库的句子中的数字表达式：The corresponding free cortisol fractions in these sera were 4.53 +/- 0.15% and 8.16 +/- 0.23%, respectively.我们应该说数字表达式 4.53 +/- 0.15%是三个词吗？或者我们应该说它是一个单独的复合词？或者我们应该说它实际上是九个词，因为它读作“four point five three,plus or minus fifteen percent”?或者我们应该说这不是一个“真正的”词，因为它不会出现在任何词典中？讨论这些不同的可能性。你能想出产生这些答案中至少两个以上可能性的应用领域吗？
29. ◑可读性测量用于为一个文本的阅读难度打分，给语言学习者挑选适当难度的文本。在一个给定的文本中，让我们定义μ_w为每个词的平均字母数，μ_s为每个句子的平均词数。文本自动可读性指数（Automated Readability Index，缩写 ARI）被定义为：$4.71\mu_w + 0.5\mu_s - 21.43$。计算布朗语料库各部分的 ARI 得分，包括 f 部分（popular lore）和 j（learned）。利用 nltk.corpus.brown.words()产生一个词汇序列，nltk.corpus.brown.sents()产生一个句子的序列的事实。
30. ◑使用 Porter 词干提取器规范化一些已标注的文本叫做为每个词提取词干。用 Lancaster 词干提取器做同样的事情，看看你是否能观察到一些差别。
31. ◑定义变量 saying 包含链表['After', 'all', 'is', 'said','and', 'done', ',', 'more', 'is', 'said', 'than', 'done', '.']。使用 for 循环处理这个链表，并将结果存储在一个新的链表 lengths 中。提示：使用 lengths = []，分配一个空链表给 lengths 开始。然后每次循环中用 append()添加另一个长度值到链表中。
32. ◑定义一个变量 silly 包含字符串：'newly formed bland ideas are inexpressible in an infuriating way'。（这碰巧是合法的解释，讲英语西班牙语双语者可以适用于乔姆斯基著名的无意义短语：colorless green ideas sleep furiously，来自维基百科）。现在编写代码执行以下任务：
    a. 分割 silly 为一个字符串链表，每一个词一个字符串，使用 Python 的 split()操作，并保存到叫做 bland 的变量中。
    b. 提取 silly 中每个词的第二个字母，将它们连接成一个字符串，得到'eoldrnnnna'。
    c. 使用 join()将 bland 中的词组合回一个单独的字符串。确保结果字符串中的词以空格隔开。

d. 按字母顺序输出 silly 中的词，每行一个。

33. ◑index()函数可用于查找序列中的项目。例如：'inexpressible'.index('e')告诉我们字母 e 的第一个位置的索引值。
    a. 当你查找一个子字符串如：'inexpressible'.index('re')会发生什么？
    b. 定义一个变量 words 包含一个词链表。现在使用 words.index()来查找一个单独的词的位置。
    c. 定义一个练习 32 中的变量 silly。使用 index()函数结合链表切片，建立一个包括 silly 中 in 之前（但不包括 in）的所有的词的链表 phrase。
34. ◑编写代码，将国家的形容词转换为它们对应的名词形式，如将 Canadian 和 Australian 转换为 Canada 和 Australia（见 http://en.wikipedia.org/wiki/List_of_adjectival_forms_of_place_names）。
35. ◑阅读 LanguageLog 中关于短语的 as best as p can 和 as best p can 形式的帖子，其中 p 是一个代名词。在一个语料库和 3.5 节中描述的搜索已标注的文本的 findall()方法的帮助下，调查这一现象。帖子在 http://itre.cis.upenn.edu/ myl/languagelog/archives/002733.html
36. ◑研究《创世记》的 lolcat 版本（使用 nltk.corpus.genesis.words('lolcat.txt')可以访问），和将文本转换为 http://www.lolcatbible.com/index.php?title=How_to_speak_lolcat. 处的 lolspeak 的规则。定义正则表达式将英文词转换成相应的 lolspeak 词。
37. ◑使用 help(re.sub)和参照本章的深入阅读，阅读有关 re.sub()函数使用正则表达式进行字符串替换的内容。使用 re.sub 编写代码从一个 HTML 文件中删除 HTML 标记，规范化空格。
38. ●分词的一个有趣的挑战是已经被分割的跨行的词。例如：如果 long-term 被分割，我们就得到字符串 long-\nterm。
    a. 写一个正则表达式，识别连字符连结的跨行处的词汇。这个表达式将需要包含\n 字符。
    b. 使用 re.sub()从这些词中删除\n 字符。
    c. 你如何确定一旦换行符被删除后不应该保留连字符的词汇，如：'encyclo-\npedia'？
39. ●阅读维基百科 Soundex 条目。用 Python 实现这个算法。
40. ●获取两个或多个文体的原始文本，计算它们各自的在前面关于阅读难度的练习中描述的阅读难度得分。例如：比较 ABC 农村新闻和 ABC 科学新闻（nltk.corpus.abc）。使用 Punkt 处理句子分割。
41. ●将下面的嵌套循环重写为嵌套链表推导：

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

42. ●使用 WordNet 为一个文本集合创建语义索引。扩展例 3.1 中的一致性搜索程序，使用它的第一个同义词集偏移索引每个词，例如：wn.synsets('dog')[0].offset（或者使用上位词层次中的一些祖先的偏移，这是可选的）。
43. ●在多语言语料库如世界人权宣言语料库（nltk.corpus.udhr），和 NLTK 的频率分布和关系排序的功能(nltk.FreqDist, nltk.spearman_correlation)的帮助下，开发一个系统，猜测未知文本。为简单起见，使用一个单一的字符编码和几种语言。
44. ●写一个程序处理文本，发现一个词以一种新的意义被使用的情况。对于每一个词计算这个词所有同义词集与这个词的上下文的所有同义词集之间的 WordNet 相似性。（请注意，这是一个粗略的办法，要做的很好是困难的，开放性研究问题。）
45. ●阅读关于规范化非标准词的文章(Sproat et al., 2001)，实现一个类似的文字规范系统。

第 4 章 编写结构化程序

现在，你对 Python 编程语言处理自然语言的能力已经有了体会。不过，如果你是 Python 或者编程新手，你可能仍然要努力对付 Python，而尚未感觉到你在完全控制它。在这一章中，我们将解决以下问题：

1. 怎么能写出结构良好、可读的程序，你和其他人将能够很容易的重新使用它？
2. 基本结构块，如循环、函数以及赋值，是如何执行的？
3. Python 编程的陷阱有哪些，你怎么能避免它们吗？

一路上，你将巩固基本编程结构的知识，了解更多关于以一种自然和简洁的方式使用 Python 语言特征的内容，并学习一些有用的自然语言数据可视化技术。如前所述，本章包含许多例子和练习（和以前一样，一些练习会引入新材料）。编程新手的读者应仔细做完它们，并在需要时查询其他编程介绍；有经验的程序员可以快速浏览本章。

在这本书的其他章节中，为了讲述 NLP 的需要，我们已经组织了一些编程的概念。在这里，我们回到一个更传统的方法，材料更紧密的与编程语言的结构联系在一起。这里不会完整的讲述编程语言，我们只关注对 NLP 最重要的语言结构和习惯用法。

4.1 回到基础

赋值

赋值似乎是最基本的编程概念，不值得单独讨论。不过，也有一些令人吃惊的微妙之处。思考下面的代码片段：

```
>>> foo = 'Monty'
>>> bar = foo ①
>>> foo = 'Python' ②
>>> bar
'Monty'
```

这个结果与预期的完全一样。当我们在代码中写 `bar = foo`①时，`foo` 的值（字符串 `'Monty'`）被赋值给 `bar`。也就是说，`bar` 是 `foo` 的一个副本，所以当我们在第②行用一个新的字符串`'Python'`覆盖 `foo` 时，`bar` 的值不会受到影响。

然而，赋值语句并不总是以这种方式复制副本。赋值总是一个表达式的值的复制，但值并不总是你可能希望的那样。特别是结构化对象的“值”，例如一个链表，实际上是一个对象的引用。在下面的例子中，①将 `foo` 的引用分配给新的变量 `bar`。现在，当我们修改 `foo` 内的东西②，我们可以看到，`bar` 的内容也已改变。

```
>>> foo = ['Monty', 'Python']
>>> bar = foo ①
>>> foo[1] = 'Bodkin' ②
>>> bar
```

['Monty', 'Bodkin']

bar = foo①行并不会复制变量的内容，只有它的“引用对象”。要了解这里发生了什么事，我们需要知道链表是如何存储在计算机内存的。在图 4.1 中，我们看到一个链表 foo 是存储在位置 3133 处的一个对象的引用。当我们赋值 bar = foo 时，仅仅是 3133 位置处的引用被复制。这种行为延伸到语言的其他方面，如参数传递（4.4 节）。

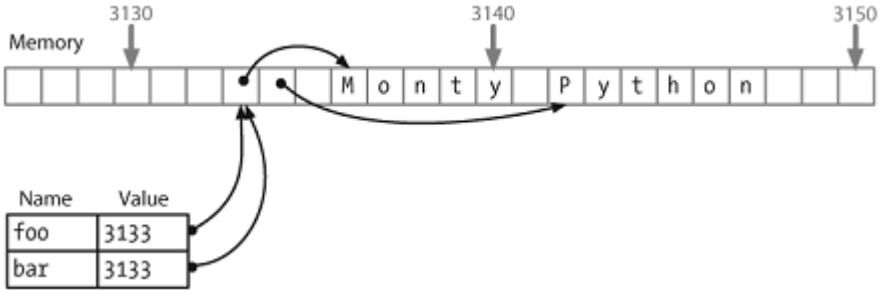


图 4.1. 链表赋值与计算机内存：两个链表对象 foo *和* bar *引用计算机内存中的相同的位置；更新* foo *将会修改* bar*，反之亦然。*

让我们做更多的实验，通过创建一个持有空链表的变量 empty，然后在下一行使用它三次。

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

请看，改变链表中嵌套链表内的一个项目，它们全改变了。这是因为三个元素中的每一个实际上都只是一个内存中的同一链表的引用。

轮到你来：用乘法创建一个链表的链表：nested = [[]] * 3。现在修改链表中的一个元素，观察所有的元素都改变了。使用 Python 的 id()函数找出任一对象的数字标识符，并验证 id(nested[0])，id(nested[1])与 id(nested[2])是一样的。

现在请注意，当我们分配一个新值给链表中的一个元素时，它并不会传送给其他元素：

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

我们一开始用含有 3 个引用的链表，每个引用指向一个空链表对象。然后，我们通过给它追加'Python'，修改这个对象，结果变成包含 3 个到一个链表对象['Python']的引用的链表。下一步，我们使用到一个新对象['Monty']的引用来覆盖三个元素中的一个。这最后一步修改嵌套链表内的 3 个对象引用中的 1 个。然而，['Python']对象并没有改变，仍然是在我们的嵌套链表的链表中的两个位置被引用。关键是要明白通过一个对象引用修改一个对象与通过覆盖一个对象引用之间的区别。

重要：要从链表 foo 复制项目到一个新的链表 bar，你可以写 bar = foo[:]。这会复制链表中的对象引用。复制结构，而不复制任何对象引用使用 copy.deepcopy()。

等式

Python 提供了两种方法来检查一对项目是否相同。is 操作符测试对象的标识符。我们可以用它来验证我们早先的对对象的观察。首先，我们创建一个链表，其中包含同一对象的多个副本，证明它们不仅对于==完全相同，而且它们是同一个对象：

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

现在，让我们将一个新的 python 放入嵌套中。可以很容易地表明这些对象不完全相同。

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

你可以再做几对测试，发现哪个位置包含闯入者，函数 id() 使检测更加容易：

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

这表明链表中的第二个项目有一个独特的标识符。如果你尝试自己运行这段代码，希望看到结果链表中的不同数字，如果“闯入者”（不同的数字）在不同的位置，请不要奇怪。

有两种等式可能看上去有些奇怪。然而，这真的只是类型与标识符式的区别，与自然语言相似，这里在一种编程语言中呈现出来。

条件语句

在 if 语句的条件部分，一个非空字符串或链表被判定为真，而一个空字符串或链表的被判定为假。

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

```
...             print element
...
cat
['dog']
```

也就是说，我们不必在条件中写：if len(element) > 0:。

使用 if...elif 而不是在一行中使用两个 if 语句有什么区别？嗯，考虑以下情况：

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

因为表达式中 if 子句条件满足，Python 就不会比较 elif 子句，所有程序永远不会输出 2。相反，如果我们用一个 if 替换 elif，那么程序会输出 1 和 2。所以 elif 子句比单独的 if 子句潜在的给我们更多信息；当它被判定为真时，告诉我们不仅条件满足而且前面的 if 子句的条件不满足。

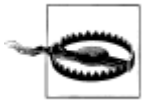
all()函数和 any()函数可以应用到一个链表（或其他序列），来检查是否全部或任一项目满足一些条件：

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

4.2 序列

到目前为止，我们已经看到了两种序列对象：字符串和链表。另一种序列被称为**元组**。元组由逗号操作符①构造，而且通常使用括号括起来。实际上，我们已经在前面的章节中看到过它们，它们有时也被称为“配对”，因为总是有两名成员。然而，元组可以有任何数目的成员。与链表和字符串一样，元组可以被索引②和切片③，并有长度④。

```
>>> t = 'walk', 'fem', 3 ①
>>> t
('walk', 'fem', 3)
>>> t[0] ②
'walk'
>>> t[1:] ③
('fem', 3)
>>> len(t) ④
```



注意！元组使用逗号操作符来构造。括号是一个 Python 语法的一般功能，设计用于分组。定义一个包含单个元素'snark'的元组是通过添加一个尾随的逗号，像这样：'snark',。空元组是一个特殊的情况下，使用空括号()定义。

让我们直接比较字符串、链表和元组，在各个类型上做索引、切片和长度操作：

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

请注意在此代码示例中，我们在一行代码中计算多个值，中间用逗号分隔。这些用逗号分隔的表达式其实就是元组——如果没有歧义，Python 允许我们忽略元组周围的括号。当我们输出一个元组时，括号始终显示。通过以这种方式使用元组，我们隐式的将这些项目聚集在一起。

轮到你来：定义一个集合，如：使用 set(text)，当把它转换成一个链表或遍历其成员时，看看会发生什么。

序列类型上的操作

我们可以用各种有用的方式遍历一个序列 s 中的项目，如表 4.1 所示。

表 4.1. 遍历序列的各种方式

Python 表达式	评论
for item in s	遍历 s 中的元素
for item in sorted(s)	按顺序遍历 s 中的元素
for item in set(s)	遍历 s 中的无重复的元素
for item in reversed(s)	按逆序遍历 s 中的元素
for item in set(s).difference(t)	遍历在集合 s 中不在集合 t 的元素
for item in random.shuffle(s)	按随机顺序遍历 s 中的元素

表格 4.1 所示的序列的功能可以多种方式相互结合；例如：要获得无重复的逆序排列的 s 的元素，可以使用 reversed(sorted(set(s)))。

我们可以在这些序列类型之间相互转换。例如：tuple(s)将任何种类的序列转换成一个元组，list(s)将任何种类的序列转换成一个链表。我们可以使用 join()函数将一个字符串链表转换成单独的字符串，例如：':'.join(words)。

其他一些对象，如 FreqDist，也可以转换成一个序列（使用 list()）且支持迭代：

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

...
4 3 2 1 1 1

在接下来的例子中，我们使用元组重新安排我们的链表中的内容。（可以省略括号，因为逗号比赋值的优先级更高。）

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

这是一种地道和可读的移动链表内的项目的方式。它相当于下面的传统方式不使用元组做上述任务（注意这种方法需要一个临时变量 tmp）。

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

正如我们已经看到的，Python 有序列处理函数，如 sorted()和 reversed()，它们重新排列序列中的项目。也有修改序列结构的函数，可以很方便的处理语言。因此，zip()取两个或两个以上的序列中的项目，将它们“压缩”打包成单个的配对链表。给定一个序列 s，enumerate(s)返回一个包含索引和索引处项目的配对。

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

对于一些 NLP 的任务，有必要将一个序列分割成两个或两个以上的部分。例如：我们可能需要用 90%的数据来“训练”一个系统，剩余 10%进行测试。要做到这一点，我们指定想要分割数据的位置①，然后在这个位置分割序列②。

```
>>> text = nltk.corpus.nps_chat.words()
>>> cut = int(0.9 * len(text)) ①
>>> training_data, test_data = text[:cut], text[cut:] ②
>>> text == training_data + test_data ③
True
>>> len(training_data) / len(test_data) ④
9
```

我们可以验证在此过程中的原始数据没有丢失，也不是复制③。我们也可以验证两块大小的比例是我们预期的④。

合并不同类型的序列

让我们综合关于这三种类型的序列的知识，一起使用链表推导处理一个字符串中的词，按它们的长度排序。

```
>>> words = 'I turned off the spectroroute'.split() ①
>>> wordlens = [(len(word), word) for word in words] ②
```

```
>>> wordlens.sort() ③
>>> ' '.join(w for (_, w) in wordlens) ④
'I off the turned spectroroute'
```

上述代码段中每一行都包含一个显著的特征。一个简单的字符串实际上是一个其上定义了方法的对象，如 split()①。我们使用链表推导建立一个元组的链表②，其中每个元组由一个数字（词长）和这个词组成，例如：(3, 'the')。我们使用 sort()方法③就地排序链表。最后，丢弃长度信息，并将这些词连接回一个字符串④。（下划线④只是一个普通的 Python 变量，我们约定可以用下划线表示我们不会使用其值的变量。）

上面的代码说明了这些序列类型的重要的区别，我们开始谈论这些序列类型的共性。首先，字符串出现在开头和结尾：这是很典型的，我们的程序先读一些文本，最后产生输出给我们看。链表和元组在中间，但使用的目的不同。一个链表是一个典型的具有*相同类型*的对象的序列，它的*长度是任意的*。我们经常使用链表保存词序列。相反，一个元组通常是*不同类型*的对象的集合，*长度固定*。我们经常使用一个元组来保存一个**纪录**：与一些实体相关的不同**字段**的集合。使用链表与使用元组之间的区别需要一些时间来习惯，所以这里是另一个例子：

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

在这里，用一个链表表示词典，因为它是一个单一类型的对象的集合——词汇条目——没有预定的长度。个别条目被表示为一个元组，因为它是一个有不同的解释的对象的集合，例如：正确的拼写形式、词性、发音（以 SAMPA 计算机可读的拼音字母表示，见 http://www.phon.ucl.ac.uk/home/sampa/）。请注意，这些发音都是用链表存储的。（为什么呢？）。

决定何时使用元组还是链表的一个好办法是看一个项目的内容是否取决与它的位置。例如：一个已标注的词标识符由两个具有不同解释的字符串组成，我们选择解释第一项为词标识符，第二项为标注。因此，我们使用这样的元组：('grail', 'noun')；一个形式为('noun', 'grail')的元组将是无意义的，因为这将是一个词 noun 被标注为 grail。相反，一个文本中的元素都是标识符，位置并不重要。因此，我们使用这样的链表：['venetian','blind']。一个形式为['blind', 'venetian']的链表也同样有效。词的语言学意义可能会有所不同，但作为标识符的链表项的解释是不变的。

链表和元组之间的使用上的区别已经讲过了。然而，还有一个更加基本的区别：在 Python 中，列表是可变的，而元组是不可变的。换句话说，列表可以被修改，而元组不能。这里是一些在链表上的操作，就地修改一个链表：

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

轮到你来：使用 lexicon = tuple(lexicon)将词典转换为一个元组。然后尝试上述操作，确认它们都不能运用在元组上。

产生器表达式

我们一直在大量使用列表推导，因为用它处理文本结构紧凑和可读性好。下面是一个例子：分词和规范化一个文本：

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

假设我们现在想要进一步处理这些词。我们可以将上面的表达式插入到一些其他函数的调用中①，Python 允许我们省略方括号②。

```
>>> max([w.lower() for w in nltk.word_tokenize(text)]) ①
'word'
>>> max(w.lower() for w in nltk.word_tokenize(text)) ②
'word'
```

第二行使用了**产生器表达式**。这不仅仅是标记（形式上看起来）方便：在许多语言处理的案例中，产生器表达式会更高效。在①中，链表对象的存储空间必须在 max()的值被计算之前分配。如果文本非常大的，这将会很慢。在②中，数据流向调用它的函数。由于调用的函数只是简单的要找最大值——按字典顺序排在最后的词——它可以处理数据流，而无需存储迄今为止的最大值以外的任何值。

4.3 风格的问题

编程是作为一门科学的艺术。无可争议的程序设计的“圣经”：Donald Knuth 的 2500 页的多卷作品被称为《计算机程序设计艺术》。已经有许多书籍是关于文学化编程的，它们认为人类，不只是电脑，必须阅读和理解程序。在这里，我们挑选了一些编程风格的问题，它们对你的代码的可读性，包括代码布局、程序与声明的风格、使用循环变量都有重要的影响。

Python 代码风格

编写程序时，你会做许多微妙的选择：名称、间距、注释等等。当你在看别人编写的代码时，风格上的不必要的差异使其难以理解。因此，Python 语言的设计者出版了 Python 代码风格指南：http://www.python.org/dev/peps/pep-0008/。风格指南中提出的基本价值是一致性，目的是最大限度地提高代码的可读性。我们在这里简要回顾一下它的一些主要建议，并请读者阅读完整的指南，里面有对实例的详细的讨论。

代码布局中每个缩进级别应使用 4 个空格。你应该确保当你在一个文件中写 Python 代码时，避免使用 tab 缩进，因为它可能由于不同的文本编辑器的不同解释而产生混乱。每行应少于 80 个字符长，如果必要的话，你可以在圆括号、方括号或花括号内换行，因为 Python 能够探测到该行与下一行是连续的，就像下面的例子：

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

```
...                 (genre, word)
...                 for genre in brown.categories()
...                 for word in brown.words(categories=genre))

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

如果你需要在圆括号、方括号或大括号中换行，通常可以添加额外的括号，也可以在行尾需要换行的地方添加一个反斜杠：

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

键入空格来代替制表符很快就会成为一件苦差事。许多程序编辑器内置对 Python 的支持，能自动缩进代码，突出任何语法错误（包括缩进错误）。了解 Python 编辑器列表，请见 http://wiki.python.org/moin/PythonEditors。

过程风格与声明风格

我们刚才已经看到可以不同的方式执行相同的任务，其中蕴含着对执行效率的影响。另一个影响程序开发的因素是编程风格。思考下面的计算布朗语料库中词的平均长度的程序：

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

在这段程序中，我们使用变量 count 跟踪遇到的标识符的数量，total 储存所有词的长度的总和。这是一个低级别的风格，与机器代码（即计算机的 CPU 所执行的基本操作）相差不远。两个变量就像 CPU 的两个寄存器，积累许多中间环节产生的值，和直到最后也毫无意义的值。我们说，这段程序是以过程风格编写，一步一步口授机器操作。现在，考虑下面的程序，计算同样的事情：

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

第一行使用生成器表达式累加标示符的长度，第二行像前面一样计算平均值。每行代码执行一个完整的、有意义的工作，可以高级别的属性，如：“total 是标识符长度的总和”，的方式来理解。实施细节留给 Python 解释器。第二段程序使用内置函数，在一个更抽象的

层面构成程序；生成的代码是可读性更好。让我们看一个极端的例子：

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

等效的声明版本使用熟悉的内置函数，可以立即知道代码的目的：

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

另一种情况，对于每行输出一个计数值，一个循环计数器似乎是必要的。然而，我们可以使用 enumerate()处理序列 s，为 s 中每个项目产生一个(i, s[i])形式的元组，以(0, s[0])开始。下面我们枚举频率分布的值，捕获变量 rank 和 word 中的整数-字符串对。按照产生排序项列表时的需要，输出 rank+1 使计数从 1 开始。

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

到目前为止，使用循环变量存储最大值或最小值，有时很诱人。让我们用这种方法找出文本中最长的词。

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

然而，一个更加清楚的解决方案是使用两个链表推导，它们的形式现在应该很熟悉：

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

请注意，我们的第一个解决方案找到第一个长度最长的词，而第二种方案找到所有最长的词（通常是我们想要的）。虽然有两个解决方案之间的理论效率的差异，主要的开销是到内存中读取数据；一旦数据准备好，第二阶段处理数据可以瞬间高效完成。我们还需要平衡我们对程序的效率与程序员的效率的关注。一种快速但神秘的解决方案将是更难理解和维护的。

计数器的一些合理用途

在有些情况下，我们仍然要在链表推导中使用循环变量。例如：我们需要使用一个循环变量中提取链表中连续重叠的 n-grams：

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

确保循环变量范围的正确相当棘手的。因为这是 NLP 中的常见操作，NLTK 提供了支持函数 bigrams(text)、trigrams(text)和一个更通用的 ngrams(text, n)。

下面是我们如何使用循环变量构建多维结构的一个例子。例如：建立一个 m 行 n 列的数组，其中每个元素是一个集合，我们可以使用一个嵌套的链表推导：

```
>>> 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([])]]
```

请看循环变量 i 和 j 在产生对象过程中没有用到；它们只是需要一个语法正确的 for 语句。这种用法的另一个例子，请看表达式['very' for i in range(3)]产生一个包含三个'very'实例的链表，没有整数。

请注意，由于我们前面所讨论的有关对象复制的原因，使用乘法做这项工作是不正确的。

```
>>> 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])]]
```

迭代是一个重要的编程概念。采取其他语言中的习惯用法是很诱人的。然而，Python 提供一些优雅和高度可读的替代品，正如我们已经看到。

4.4 函数：结构化编程的基础

函数提供了程序代码打包和重用的有效途径，已经在 2.3 节中解释过。例如：假设我们发现我们经常要从 HTML 文件读取文本。这包括以下几个步骤：打开文件，将它读入，规范化空白符号，剥离 HTML 标记。我们可以将这些步骤收集到一个函数中，并给它一个名字，如 get_text()，如例 4.2 所示。

例 4-1. 从文件读取文本

```
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
```

现在，任何时候我们想从一个 HTML 文件得到干净的文字，都可以用文件的名字作为唯一的参数调用 get_text()。它会返回一个字符串，我们可以将它指定给一个变量，例如: contents = get_text("test.html")。每次我们要使用这一系列的步骤，只需要调用这个函数。

使用函数可以为我们的程序节约空间。更重要的是：我们为函数选择名称可以提高程序可读性。在上面的例子中，只要我们的程序需要从文件读取干净的文本，我们不必弄乱这四行代码的程序，只需要调用 get_text()。这种命名方式有助于提供一些“语义解释”——它可以帮助我们的程序的读者理解程序的“意思”。

请注意，上面的函数定义包含一个字符串。函数定义内的第一个字符串被称为 **docstring**。它不仅为阅读代码的人记录函数的功能，也使从文件加载这段代码的程序员更容易访问：

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

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

我们已经看到，函数有助于提高我们的工作的可重用性和可读性。它还有益于程序的可靠性。当我们重用已开发和测试过的代码时，可以更有信心，因为它已经正确处理过各种案件。我们还可以避免忘记了一些重要步骤，或引入 bug 的风险。调用我们的函数的程序也提高了可靠性。该程序的作者可以处理一个较短的程序，其组成部分的行为也更透明。

简而言之，顾名思义，函数捕捉功能。它是一个代码段，可以给取一个有意义的名字，并执行一个明确的任务。函数使我们可以对细节不予考虑，而去看一个更大的图案，更高效地编程。

本节的其余部分将仔细的分析函数，探索它的机制，讨论使你的程序更容易阅读的方法。

函数的输入和输出

我们使用函数的参数传递信息给函数，参数是括号括起的变量和常量的列表，在函数定

义中跟在函数名称之后。下面是一个完整的例子：

```
>>> def repeat(msg, num):  ①
...     return ' '.join([msg] * num)
>>> monty = 'Monty Python'
>>> repeat(monty, 3) ②
'Monty Python Monty Python Monty Python'
```

我们首先定义函数的两个参数：msg 和 num①。然后调用函数，并传递给它两个参数: monty 和 3②；这些参数填补了参数提供的“占位符”，为函数体中出现的 msg 和 num 提供值。

我们看到在下面的例子中不需要有任何参数：

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

函数通常会通过 return 语句将其结果返回给调用它的程序，正如我们刚才看到的。对于调用程序，它看起来就像函数调用已被函数结果替代：

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

一个 Python 函数并不是一定需要有一个 return 语句。有些函数做它们的工作的同时会附带输出结果、修改文件或者更新参数的内容。(这种函数在其他一些编程语言中被称为“过程”)。

考虑以下三个排序功能。第三个是危险的，因为程序员可能没有意识到它已经修改了给它的输入。一般情况下，函数应该修改参数的内容（my_sort1()）或返回一个值（my_sort2()），而不是两个都做（my_sort3()）。

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

参数传递

早在 4.1 节中，你就已经看到了赋值操作，而一个结构化对象的值是该对象的引用。函数也是一样的。Python 按它的值来解释函数的参数（这被称为**按值传递**）。在下面的代码中，set_up()有两个参数，都在函数内部被修改。我们一开始将一个空字符串分配给 w，将一个空链表分配给 p。调用该函数后，w 没有变，而 p 改变了：

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

```
...
>>> w = ''
>>> p = []
>>> set_up(w, p)
>>> w
''
>>> p
['noun']
```

请注意，w 没有被函数改变。当我们调用 set_up(w, p)时，w（空字符串）的值被分配到一个新的变量 word。在函数内部 word 值被修改。然而，这种变化并没有传播给 w。这个参数传递过程与下面的赋值序列是一样的：

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

让我们来看看链表 p 上发生了什么。当我们调用 set_up(w, p)，p 的值（一个空列表的引用）被分配到一个新的本地变量 properties，所以现在这两个变量引用相同的内存位置。函数修改 properties，而这种变化也反映在 p 值上，正如我们所看到的。函数也分配给 properties 一个新的值（数字 5）；这并不能修改该内存位置上的内容，而是创建了一个新的局部变量。这种行为就好像是我们做了下列赋值序列：

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

因此，要理解 Python 按值传递参数，只要了解它是如何赋值的就足够了。记住，你可以使用 id() 函数和 is 操作符来检查每个语句执行之后你对对象标识符的理解。

变量的作用域

函数定义为变量创建了一个新的局部的**范围**。当你在函数体内部分配一个新的变量时，这个名字只在该函数内部被定义。函数体外或者在其它函数体内，这个名字是不可见的。这一行为意味着你可以选择变量名而不必担心它与你的其他函数定义中使用的名称冲突。

当你在一个函数体内部使用一个现有的名字时，Python 解释器先尝试按照函数本地的名字来解释。如果没有发现，解释器检查它是否是一个模块内的全局名称。如果没有成功，最后，解释器会检查是否是 Python 内置的名字。这就是所谓的名称解析的 **LGB 规则**：本地（local），全局（global），然后内置（built-in）。

注意！一个函数可以使用 global 声明创建一个新的全局变量。然而，这种做法应尽可能避免。在函数内部定义全局变量会导致上下文依赖性而限制函数的便携性（或重用性）。一般来说，你应该使用参数作为函数的输入，返回值作为函数的输出。

参数类型检查

我们写程序时，Python 不会强迫我们声明变量的类型，这允许我们定义参数类型灵活的函数。例如：我们可能希望一个标注只是一个词序列，而不管这个序列被表示为一个链表、元组或是迭代器（一种新的序列类型，我们将在下面讨论）。

然而，我们常常想写一些能被他人利用的程序，并希望以一种防守的风格，当函数没有被正确调用时提供有益的警告。下面的 tag()函数的作者假设其参数将始终是一个字符串。

```
>>> def tag(word):
...     if word in ['a', 'the', 'all']:
...         return 'det'
...     else:
...         return 'noun'
...
>>> tag('the')
'det'
>>> tag('knight')
'noun'
>>> tag(["'Tis", 'but', 'a', 'scratch'])  ①
'noun'
```

该函数对参数'the'和'knight'返回合理的值，传递给它一个链表①，看看会发生什么——它没有抱怨，虽然它返回的结果显然是不正确的。此函数的作者可以采取一些额外的步骤来确保 tag()函数的参数 word 是一个字符串。一种天真的做法是使用 if not type(word) is str 检查参数的类型，如果 word 不是一个字符串，简单地返回 Python 特殊的空值：None。这是一个略微的改善，因为该函数在检查参数类型，并试图对错误的输入返回一个“特殊的”诊断结果。然而，它也是危险的，因为调用程序可能不会检测 None 是故意设定的“特殊”值，这种诊断的返回值可能被传播到程序的其他部分产生不可预测的后果。如果这个词是一个 Unicode 字符串这种方法也会失败。因为它的类型是 unicode 而不是 str。这里有一个更好的解决方案，使用 assert 语句和 Python 的 basestring 的类型一起，它是 unicode 和 str 的产生类型。

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

如果 assert 语句失败，它会产生一个不可忽视的错误而停止程序执行。此外，该错误信息是容易理解的。程序中添加断言能帮助你找到逻辑错误，是一种**防御性编程**。一个更根本的方法是在本节后面描述的使用 docstrings 为每个函数记录参数。

功能分解

结构良好的程序通常都广泛使用函数。当一个程序代码块增长到超过 10-20 行，如果将代码分成一个或多个函数，每一个有明确的目的，这将对可读性有很大的帮助。这类似于

好文章被划分成段，每段话表示一个主要思想。

函数提供了一种重要的抽象。它们让我们将多个动作组合成一个单一的复杂的行动，并给它关联一个名称。（比较我们组合动作 go 和 bring back 为一个单一的更复杂的动作 fetch。）当我们使用函数时，主程序可以在一个更高的抽象水平编写，使其结构更透明，例如：

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

适当使用函数使程序更具可读性和可维护性。另外，重新实现一个函数已成为可能——使用更高效的代码替换函数体——不需要关心程序的其余部分。

思考例 4-2 中 freq_words 函数。它更新一个作为参数传递进来的频率分布的内容，并输出前 n 个最频繁的词的链表。

例 4-2. 设计不佳的函数用来计算高频词。

```
def freq_words(url, freqdist, n):
    text = nltk.clean_url(url)
    for word in nltk.word_tokenize(text):
    freqdist.inc(word.lower())
    print freqdist.keys()[:n]
>>> constitution = "http://www.archives.gov/national-archives-experience" \
... "/charters/constitution_transcript.html"
>>> fd = nltk.FreqDist()
>>> freq_words(constitution, fd, 20)
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

这个函数有几个问题。该函数有两个副作用：它修改了第二个参数的内容，并输出它已计算的结果的经过选择的子集。如果我们在函数内部初始化 FreqDist()对象（在它被处理的同一个地方），并且去掉选择集而将结果显示给调用程序的话，函数会更容易理解和更容易在其他地方重用。在例 4-3 中，我们**重构**此函数，并通过提供一个单一的 url 参数简化其接口。

例 4-3. 精心设计的函数用来计算高频词

```
def freq_words(url):
    freqdist = nltk.FreqDist()
    text = nltk.clean_url(url)
    for word in nltk.word_tokenize(text):
    freqdist.inc(word.lower())
    return freqdist
>>> fd = freq_words(constitution)
>>> print fd.keys()[:20]
['the', 'of', 'charters', 'bill', 'constitution', 'rights', ',',
'declaration', 'impact', 'freedom', '-', 'making', 'independence']
```

请注意，我们现在已经将 freq_words 的工作简化到可以用 3 行代码完成的程度：

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

'declaration', 'impact', 'freedom', '-', 'making', 'independence']

文档说明函数

如果我们已经将工作分解成函数分解的很好了，那么应该很容易使用通俗易懂的语言描述每个函数的目的，并且在函数的定义顶部的 docstring 中提供这些描述。这条语句不应该解释函数是如何实现的；实际上，应该能够不改变这条语句，使用不同的方法，重新实现这个函数。

对于最简单的函数，一个单行的 docstring 通常就足够了（见例 4-1）。你应该提供一个在一行中包含一个完整的句子的三重引号引起来的字符串。对于不寻常的函数，你还是应该在第一行提供一个一句话总结，因为很多的 docstring 处理工具会索引这个字符串。它后面应该有一个空行，然后是更详细的功能说明（见 http://www.python.org/dev/peps/pep-0257/的 docstring 约定的更多信息）。

docstring 中可以包括一个 **doctest 块**，说明使用的函数和预期的输出。这些都可以使用 Python 的 docutils 模块自动测试。docstring 中应当记录函数的每个参数的类型和返回类型。至少可以用纯文本来做这些。请注意，NLTK 中使用“epytext”标记语言来记录参数。这种格式可以自动转换成富结构化的 API 文档（见 http://www.nltk.org/），并包含某些“字段”的特殊处理，例如：@param 允许清楚地记录函数的输入和输出。例 4-4 演示了一个完整的 docstring。

例如 4-4 一个完整的 docstring 的演示，包括一行总结，一个更详细的解释，一个 doctest 例子以及特定参数、类型、返回值和异常的 epytext 标记。

```
def accuracy(reference, test):
    """
    Calculate the fraction of test items that equal the corresponding reference items.
    Given a list of reference values and a corresponding list of test values,
    return the fraction of corresponding values that are equal.
    In particular, return the fraction of indexes
    {0<i<=len(test)} such that C{test[i] == reference[i]}.
    >>> accuracy(['ADJ', 'N', 'V', 'N'], ['N', 'N', 'V', 'ADJ'])
    0.5
@param reference: An ordered list of reference values.
@type reference: C{list}
@param test: A list of values to compare against the corresponding
    reference values.
@type test: C{list}
@rtype: C{float}
@raise ValueError: If C{reference} and C{length} do not have the
    same length.
    """
    if len(reference) != len(test):
        raise ValueError("Lists must have the same length.")
    num_correct = 0
    for x, y in izip(reference, test):
        if x == y:
```

```
            num_correct += 1
    return float(num_correct) / len(reference)
```

4.5 更多关于函数

本节将讨论更高级的特性，你在第一次阅读本章时可能更愿意跳过此节。

作为参数的函数

到目前为止，我们传递给函数的参数一直都是简单的对象，如字符串或链表等结构化对象。Python 也允许我们传递一个函数作为另一个函数的参数。现在，我们可以抽象出操作，对相同数据进行不同操作。正如下面的例子表示的，我们可以传递内置函数 len()或用户定义的函数 last_letter()作为另一个函数的参数：

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

对象 len 和 last_letter 可以像链表和字典那样被传递。请注意，只有在我们调用该函数时，才在函数名后使用括号；当我们只是将函数作为一个对象，括号被省略。

Python 提供了更多的方式来定义函数作为其他函数的参数，即所谓的 **lambda 表达式**。试想在很多地方没有必要使用上述的 last_letter()函数，因此没有必要给它一个名字。我们可以写以下内容是等价的：

```
>>> extract_property(lambda w: w[-1])
['e', 'e', 'f', 'e', 'e', ',', 'd', 'e', 's', 'l', 'e', 'e', 'f', 's', '.']
```

我们的下一个例子演示传递一个函数给 sorted()函数。当我们用唯一的参数（需要排序的链表）调用后者，它使用内置的比较函数 cmp()。然而，我们可以提供自己的排序函数，例如：按长度递减排序。

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

累计函数

这些函数以初始化一些存储开始，迭代和处理输入的数据，最后返回一些最终的对象（一个大的结构或汇总的结果）。做到这一点的一个标准的方式是初始化一个空链表，累计材料，然后返回这个链表，如例 4-5 所示函数 search1()。

例 4-5. 累计输出到一个链表

```
def search1(substring, words):
    result = []
    for word in words:
        if substring in word:
            result.append(word)
    return result
def search2(substring, words):
    for word in words:
        if substring in word:
            yield word
print "search1:"
for item in search1('zz', nltk.corpus.brown.words()):
    print item
print "search2:"
for item in search2('zz', nltk.corpus.brown.words()):
    print item
```

函数 search2()是一个产生器。第一次调用此函数，它运行到 yield 语句然后停下来。调用程序获得第一个词，没有任何必要的处理。一旦调用程序对另一个词做好准备，函数会从停下来的地方继续执行，直到再次遇到 yield 语句。这种方法通常更有效，因为函数只产生调用程序需要的数据，并不需要分配额外的内存来存储输出（参见前面关于产生器表达式的讨论）。

下面是一个更复杂的产生器的例子，产生一个词链表的所有排列。为了强制 permutations()函数产生所有它的输出，我们将它包装在 list()调用中①。

```
>>> def permutations(seq):
...     if len(seq) <= 1:
...         yield seq
...     else:
...         for perm in permutations(seq[1:]):
...             for i in range(len(perm)+1):
...                 yield perm[:i] + seq[0:1] + perm[i:]
...
>>> list(permutations(['police', 'fish', 'buffalo'])) ①
[['police', 'fish', 'buffalo'], ['fish', 'police', 'buffalo'],
 ['fish', 'buffalo', 'police'], ['police', 'buffalo', 'fish'],
 ['buffalo', 'police', 'fish'], ['buffalo', 'fish', 'police']]
```

permutations 函数使用了一种技术叫递归，将在 4.7 节讨论。产生一组词的排列对于创建测试一个语法的数据（第 8 章）十分有用。

高阶函数

Python 提供了一些具有函数式编程语言标准特征的高阶函数，如：Haskell。我们将在这里演示它们，与使用链表推导的相对应的表达一起。

让我们从定义一个函数 is_content_word()开始，它检查一个词是否来自一个开放的实词类。使用此函数作为 offilter()的第一个参数，它对作为它的第二个参数的序列中的每个项目运用该函数，只保留该函数返回 True 的项目。

```
>>> def is_content_word(word):
... return word.lower() not in ['a', 'of', 'the', 'and', 'will', ',', '.']
>>> sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
... 'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
>>> filter(is_content_word, sent)
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
>>> [w for w in sent if is_content_word(w)]
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

另一个高阶函数是 map()，将一个函数运用到一个序列中的每一项。它是我们在本节早先看到的函数 extract_property()的一个通用版本。这里是一个简单的方法找出布朗语料库新闻部分中的句子的平均长度，后面跟着的是使用链表推导计算的等效版本：

```
>>> lengths = map(len, nltk.corpus.brown.sents(categories='news'))
>>> sum(lengths) / len(lengths)
21.7508111616
>>> lengths = [len(w) for w in nltk.corpus.brown.sents(categories='news')]
>>> sum(lengths) / len(lengths)
21.7508111616
```

在上面的例子中，我们指定了一个用户定义的函数 is_content_word()和一个内置函数 len()。我们还可以提供一个 lambda 表达式。这里是两个等效的例子，计数每个词中的元音的数量。

```
>>> map(lambda w: len(filter(lambda c: c.lower() in "aeiou", w)), sent)
[2, 2, 1, 1, 2, 0, 1, 1, 2, 1, 2, 2, 1, 3, 0]
>>> [len([c for c in w if c.lower() in "aeiou"]) for w in sent]
[2, 2, 1, 1, 2, 0, 1, 1, 2, 1, 2, 2, 1, 3, 0]
```

链表推导为基础的解决方案通常比基于高阶函数的解决方案可读性更好，我们在整个这本书的青睐于使用前者。

参数的命名

当有很多参数时，很容易混淆正确的顺序。我们可以通过名字引用参数，甚至可以给它们分配默认值以供调用程序没有提供该参数时使用。现在参数可以按任意顺序指定，也可以省略。

```
>>> def repeat(msg='<empty>', num=1):
... return msg * num
>>> repeat(num=3)
'<empty><empty><empty>'
>>> repeat(msg='Alice')
'Alice'
>>> repeat(num=5, msg='Alice')
'AliceAliceAliceAliceAlice'
```

这些被称为**关键字参数**。如果我们混合使用这两种参数，就必须确保未命名的参数在命名的参数前面。必须是这样，因为未命名参数是根据位置来定义的。我们可以定义一个函数，接受任意数量的未命名和命名参数，并通过一个就地的参数链表*args 和一个就地的关键字参数字典**kwargs 来访问它们。

```
>>> def generic(*args, **kwargs):
...     print args
...     print kwargs
...
>>> generic(1, "African swallow", monty="python")
(1, 'African swallow')
{'monty': 'python'}
```

当*args 作为函数参数时，它实际上对应函数所有的未命名参数。下面是另一个这方面的 Python 语法的演示，思考处理可变数目的参数的函数 zip()。我们将使用变量名*song 来表示名字*args 并没有什么特别的。

```
>>> song = [['four', 'calling', 'birds'],
...         ['three', 'French', 'hens'],
...         ['two', 'turtle', 'doves']]
>>> zip(song[0], song[1], song[2])
[('four', 'three', 'two'), ('calling', 'French', 'turtle'), ('birds', 'hens', 'doves')]
>>> zip(*song)
[('four', 'three', 'two'), ('calling', 'French', 'turtle'), ('birds', 'hens', 'doves')]
```

应该从这个例子中明白输入*song 仅仅是一个方便的记号，相当于输入了 song[0]，song[1]，song[2]。

下面是另一个在函数的定义中使用关键字参数的例子，有三种等效的方法来调用这个函数：

```
>>> def freq_words(file, min=1, num=10):
...     text = open(file).read()
...     tokens = nltk.word_tokenize(text)
...     freqdist = nltk.FreqDist(t for t in tokens if len(t) >= min)
...     return freqdist.keys()[:num]
>>> fw = freq_words('ch01.rst', 4, 10)
>>> fw = freq_words('ch01.rst', min=4, num=10)
>>> fw = freq_words('ch01.rst', num=10, min=4)
```

已命名的参数的另一个作用是它们允许选择性使用参数。因此，我们可以在我们高兴使用默认值的地方省略任何参数：freq_words('ch01.rst', min=4)，freq_words('ch01.rst', 4)。可选参数的另一个常见用途是作为标志使用。这里是同一个的函数的修订版本，

如果设置了 verbose 标志将会报告其进展情况：

```
>>> def freq_words(file, min=1, num=10, verbose=False):
...     freqdist = FreqDist()
...     if trace: print "Opening", file
...     text = open(file).read()
...     if trace: print "Read in %d characters" % len(file)
...     for word in nltk.word_tokenize(text):
...         if len(word) >= min:
...             freqdist.inc(word)
...             if trace and freqdist.N() % 100 == 0: print "."
...     if trace: print
...     return freqdist.keys()[:num]
```

注意！

注意不要使用可变对象作为参数的默认值。这个函数的一系列调用将使用同一个对象，有时会出现离奇的结果，就像我们稍后会在关于调试的讨论中看到的那样。

4.6 程序开发

编程是一种技能，需要获得几年的各种编程语言和任务的经验。关键的高层次能力是算法设计及其在结构化编程中的实现。关键的低层次的能力包括熟悉语言的语法结构，以及排除故障的程序（不能表现预期的行为的程序）的各种诊断方法的知识。

本节描述一个程序模块的内部结构，以及如何组织一个多模块的程序。然后描述程序开发过程中出现的各种错误，你可以做些什么来解决这些问题，更好的是，从一开始就避免它们。

Python 模块的结构

程序模块的目的是把逻辑上相关的定义和函数结合在一起，以方便重用和更高层次的抽象。Python 模块只是一些单独的.py 文件。例如：如果你在处理一种特定的语料格式，读取和写入这种格式的函数可以放在一起。这两种格式所使用的常量，如字段分隔符或一个 EXTN = ".inf"文件扩展名，可以共享。如果要更新格式，你就会知道只有一个文件需要改变。同样的，一个模块可以包含用于创建和操纵一种特定的数据结构，如语法树，的代码，或执行特定的处理任务，如绘制语料统计图表，的代码。

当你开始编写 Python 模块，有一些例子来模拟是有益的。你可以使用变量__file__定位你的系统中任一 NLTK 模块的代码：

```
>>> nltk.metrics.distance.__file__
'/usr/lib/python2.5/site-packages/nltk/metrics/distance.pyc'
```

这将返回模块已编译.pyc 文件的位置，在你的机器上你可能看到的位置不同。你需要打开的文件是对应的.py 源文件，它和.pyc 文件放在同一目录下。另外，你可以在网站上查看该模块的最新版本：http://code.google.com/p/nltk/source/browse/trunk/nltk/nltk/metrics/distance.py。

与其他 NLTK 的模块一样，distance.py 以一组注释行开始，包括一行模块标题和作者信息。（由于代码会被发布，也包括代码可用的 URL、版权声明和许可信息。）接下来是模块级的 docstring，三重引号的多行字符串，其中包括当有人输入 help(nltk.metrics.distance)将被输出的关于模块的信息。

```
# Natural Language Toolkit: Distance Metrics
#
# Copyright (C) 2001-2009 NLTK Project
# Author: Edward Loper <edloper@gradient.cis.upenn.edu>
#         Steven Bird <sb@csse.unimelb.edu.au>
#         Tom Lippincott <tom@cs.columbia.edu>
# URL: <http://www.nltk.org/>
# For license information, see LICENSE.TXT
#
"""
Distance Metrics.
Compute the distance between two items (usually strings).
As metrics, they must satisfy the following three requirements:
1. d(a, a) = 0
2. d(a, b) >= 0
3. d(a, c) <= d(a, b) + d(b, c)
"""
```

在这之后是所有需要的模块的导入语句，然后是所有全局变量，接着是组成模块主要部分的一系列函数的定义。有些模块定义“类”，面向对象编程的主要部分，这超出了这本书的范围。（大多数 NLTK 的模块还包括一个 demo()函数，可以用来演示模块使用方法的例子。）

模块的一些变量和函数仅用于模块内部。它们的名字应该以下划线开头，如_helper()，因为这将隐藏名称。如果另一个模块使用习惯用法：from module import * 导入这个模块，这些名称将不会被导入。你可以选择性的列出一个模块的外部可访问的名称，使用像这样的一个特殊的内置变量：__all__ = ['edit_distance', 'jaccard_distance']。

多模块程序

一些程序汇集多种任务，例如：从语料库加载数据、对数据进行一些分析、然后将其可视化。我们可能已经有了稳定的模块来加载数据和实现数据可视化。我们的工作可能会涉及到那些分析任务的编码，只是从现有的模块调用一些函数。图 4-2 描述了这种情景。

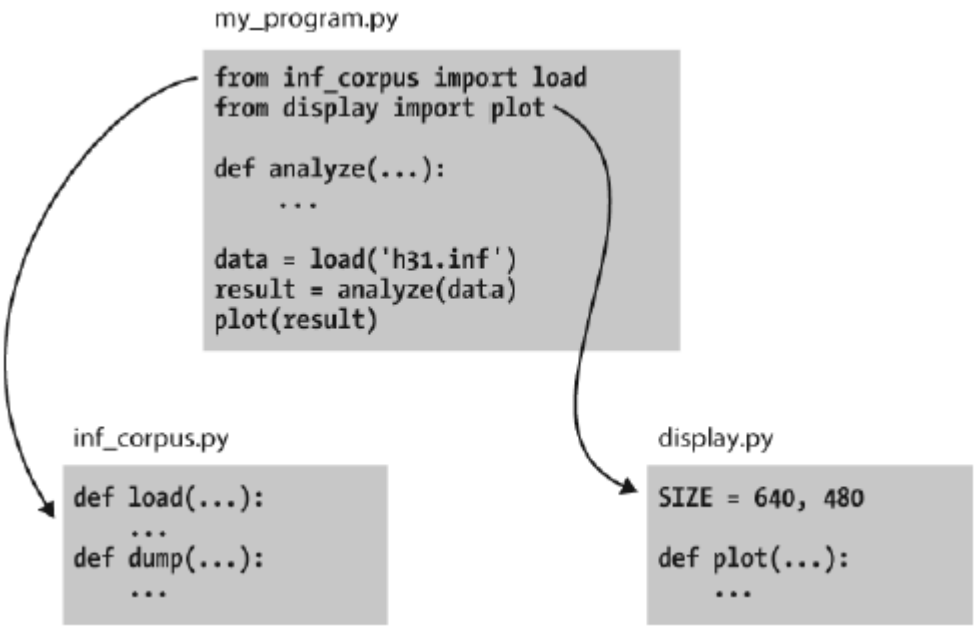


图 4-7. 一个多模块程序的结构：主程序 my_program.py 从其他两个模块导入函数；独特的分析任务在主程序本地进行，而一般的载入和可视化任务被分离开以便可以重用和抽象。

通过将我们的工作分成几个模块和使用 import 语句访问别处定义的函数，我们可以保持各个模块简单，易于维护。这种做法也将导致越来越多的模块的集合，使我们有可能建立复杂的涉及模块间层次结构的系统。设计这样的系统是一个复杂的软件工程任务，这超出了本书的范围。

误差源头

掌握编程技术取决于当程序不按预期运作时各种解决问题的技能的总结。一些琐碎的东西，如放错位置的符号，可能导致程序的行为异常。我们把这些叫做“bugs”，因为它们与它们所导致的损害相比较小。它们不知不觉的潜入我们的代码，只有在很久以后，我们在一些新的数据上运行程序时才会发现它们的存在。有时，一个错误的修复仅仅是暴露出另一个，于是我们得到了鲜明的印象：bug 在移动。我们唯一的安慰是 bugs 是自发的而不是程序员的错误。

繁琐浮躁不谈，调试代码是很难的，因为有那么多的方式出现故障。我们对输入数据、算法甚至编程语言的理解可能是错误的。让我们分别来看看每种情况的例子。

首先，输入的数据可能包含一些意想不到的字符。例如：WordNet 的同义词集名称的形式是 tree.n.01，逗号分割的单个部分。最初 NLTK 的 WordNet 模块使用 split('.')分解这些名称。然而，当有人试图寻找词 PhD 时，这种方法就不能用了，它的同义集名称是 ph.d..n.01，包含 4 个逗号而不是预期的 2 个。解决的办法是使用 rsplit('.', 2)利用最右边的逗号，最多分割两次，留下字符串 ph.d.不变。虽然在模块发布之前已经测试过，但就在几个星期前有人检测到这个问题（见 http://code.google.com/p/nltk/issues/detail?id=297）。

第二，提供的函数可能不会像预期的那样运作。例如：在测试 NLTK 中的 WordNet 接口时，一名作者注意到没有同义词集定义了反义词，而底层数据库提供了大量的反义词的信息。这看着像是 WordNet 接口中的一个错误，结果却是对 WordNet 本身的误解：反义词在词条中定义，而不是在义词集中。唯一的“bug”是对接口的一个误解（见 http://code.google.com/p/nltk/issues/detail?id=98）。

第三，我们对 Python 语义的理解可能出错。很容易做出关于两个操作符的相对范围的错误的假设。例如："%s.%s.%02d" % "ph.d.", "n", 1 产生一个运行时错误 TypeError:not enough arguments for format string。这是因为百分号操作符优先级高于逗号运算符。解决办法是添加括号强制限定所需的范围。作为另一个例子，假设我们定义一个函数来收集一个文本中给定长度的所有标识符。该函数有文本和词长作为参数，还有一个额外的参数，允许指定结果的初始值作为参数：

```
>>> def find_words(text, wordlength, result=[]):
...         for word in text:
...             if len(word) == wordlength:
...                 result.append(word)
...         return result
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 3) ①
['omg', 'teh', 'teh', 'mat']
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 2, ['ur']) ②
['ur', 'on']
>>> find_words(['omg', 'teh', 'lolcat', 'sitted', 'on', 'teh', 'mat'], 3) ③
['omg', 'teh', 'teh', 'mat', 'omg', 'teh', 'teh', 'mat']
```

我们第一次调用 find_words()①得到所有预期的三个字母的词。第二次，我们为 result 指定一个初始值：一个单元素链表['ur']，如预期，结果中有这个词连同我们的文本中的其他双字母的词。现在，我们再次使用①中相同的参数调用 find_words()③，但我们得到了不同的结果！我们每次不使用第三个参数调用 find_words()，结果都只会延长前次调用的结果，而不是以在函数定义中指定的空链表 result 开始。程序的行为并不如预期，因为我们错误地认为在函数被调用时会创建默认值。然而，它只创建了一次，在 Python 解释器加载这个函数时。这一个链表对象会被使用，只要没有给函数提供明确的值。

调试技术

由于大多数代码错误是因为程序员的不正确的假设，你检测 bug 要做的第一件事是检查你的假设。通过给程序添加 print 语句定位问题，显示重要的变量的值，并显示程序的进展程度。

如果程序产生一个“异常”——运行时错误——解释器会输出一个**堆栈跟踪**，精确定位错误发生时程序执行的位置。如果程序取决于输入数据，尽量将它减少到能产生错误的最小尺寸。

一旦你已经将问题定位在一个特定的函数或一行代码，你需要弄清楚是什么出了错误。使用交互式命令行重现错误发生时的情况往往是有益的。定义一些变量，然后复制粘贴可能出错的代码行到会话中，看看会发生什么。检查你对代码的理解，通过阅读一些文档和测试与你正在试图做的事情相同的其他代码示例。尝试将你的代码解释给别人听，也许她会看出出错的地方。

Python 提供了一个**调试器**，它允许你监视程序的执行，指定程序暂停运行的行号（即**断点**），逐步调试代码段和检查变量的值。你可以如下方式在你的代码中调用调试器：

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.myfunction()')
```

它会给出一个提示(Pdb)，你可以在那里输入指令给调试器。输入 help 来查看命令的

完整列表。输入 step(或只输入 s)将执行当前行然后停止。如果当前行调用一个函数，它将进入这个函数并停止在第一行。输入 next(或只输入 n)是类似的，但它会在当前函数中的下一行停止执行。break（或 b）命令可用于创建或列出断点。输入 continue（或 c）会继续执行直到遇到下一个断点。输入任何变量的名称可以检查它的值。

我们可以使用 Python 调试器来查找 find_words()函数的问题。请记住问题是在第二次调用函数时产生的。我们一开始将不使用调试器而调用该函数①，使用可能的最小输入。第二次我们使用调试器调用它②。

```
>>> import pdb
>>> find_words(['cat'], 3) ①
['cat']
>>> pdb.run("find_words(['dog'], 3)") ②
> <string>(1)<module>()
(Pdb) step
--Call--
> <stdin>(1)find_words()
(Pdb) args
text = ['dog']
wordlength = 3
result = ['cat']
```

在这里，我们只输入了两个命令到调试器：step 将我们带入函数体内部，args 显示它的参数的值。我们立即看到 result 有一个初始值['cat']，而不是如预期的空链表。调试器帮助我们定位问题，促使我们检查我们对 Python 函数的理解。

防御性编程

为了避免一些调试的痛苦，养成防御性的编程习惯是有益的。不要写 20 行程序然后测试它，而是自下而上的打造一些明确可以运作的小的程序片。每次你将这些程序片组合成更大的单位都要仔细的看它是否能如预期的运作。考虑在你的代码中添加 assert 语句，指定变量的属性，例如：assert(isinstance(text, list))。如果 text 的值在你的代码被用在一些较大的环境中时变为了一个字符串，将产生一个 AssertionError，于是你会立即得到问题的通知。

一旦你觉得你发现了错误，作为一个假设查看你的解决方案。在重新运行该程序之前尝试预测你修正错误的影响。如果 bug 不能被修正，不要陷入盲目修改代码希望它会奇迹般地重新开始运作的陷阱。相反，每一次修改都要尝试阐明错误是什么和为什么这样修改会解决这个问题的假设。如果这个问题没有解决就撤消这次修改。

当你开发你的程序时，扩展其功能，并修复所有 bug，维护一套测试用例是有益的。这被称为**回归测试**，因为它是用来检测代码“回归”的地方——修改代码后会带来一个意想不到的副作用是以前能运作的程序不运作了的地方。Python 以 doctest 模块的形式提供了一个简单的回归测试框架。这个模块搜索一个代码或文档文件查找类似与交互式 Python 会话这样的文本块，这种形式你已经在这本书中看到了很多次。它执行找到的 Python 命令，测试其输出是否与原始文件中所提供的输出匹配。每当有不匹配时，它会报告预期值和实际值。有关详情，请查询在 http://docs.python.org/library/doctest.html 上的 doctest 文档。除了回归测试它的值，doctest 模块有助于确保你的软件文档与你的代码保持同步。

也许最重要的防御性编程策略是要清楚的表述你的代码，选择有意义的变量和函数名，

并通过将代码分解成拥有良好文档的接口的函数和模块尽可能的简化代码。

4.7 算法设计

本节将讨论更高级的概念，你第一次阅读本章可能更愿意跳过此节。

解决算法问题的一个重要部分是为手头的问题选择或改造一个合适的算法。有时会有几种选择，能否选择最好的一个取决于对每个选择随数据增长如何执行的知识。关于这个话题的书很多，我们只介绍一些关键概念和精心描述在自然语言处理中最普遍的做法。

最有名的策略被称为**分而治之**。我们解决一个大小为 n 的问题通过将其分成两个大小为 n/2 的问题，解决这些问题，组合它们的结果成为原问题的结果。例如：假设我们有一堆卡片，每张卡片上写了一个词。我们可以排序这一堆卡片，通过将它分成两半分别给另外两个人来排序（他们又可以做同样的事情）。然后，得到两个排好序的卡片堆，将它们并成一个单一的排序堆就是一项容易的任务了。参见图 4-3 这个过程的说明。

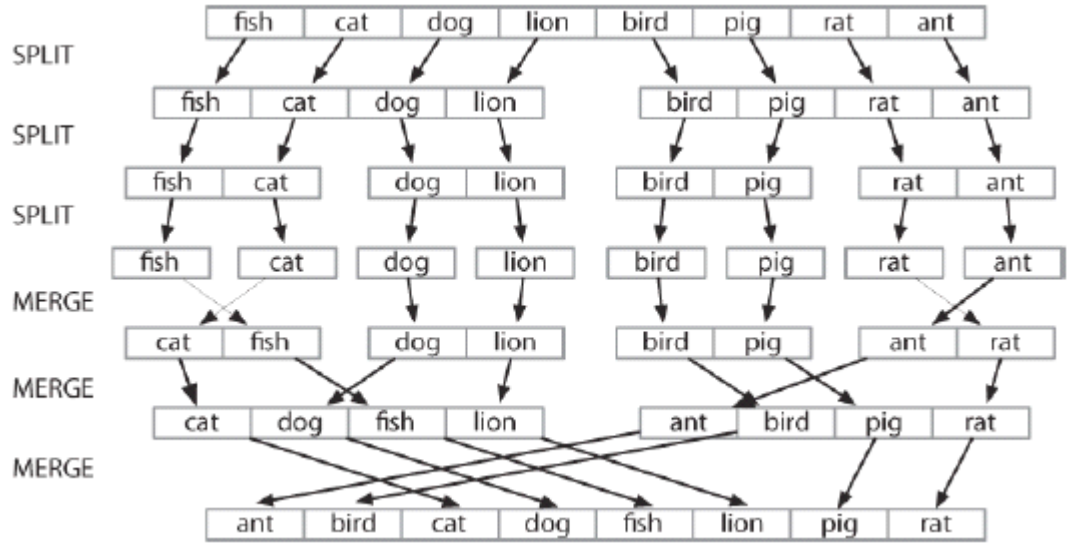


图 4.8：通过分而治之排序：对一个数组排序，我们将其分成两半并对每一半进行排序（递归）；将每个排好序的一半合并成一个完整的链表（再次递归）；这个算法被称为“归并排序“。

另一个例子是在词典中查找一个词的过程。我们打开在书的中部附近的一个地方，比较我们的词与当前页面上的词。如果它在词典中的词前面，我们就在词典的前一半重复上面的过程；如果它在后面，我们就使用词典的后一半。这种搜索方法被称为二分查找，因为它的每一步都将问题分裂成一半。

算法设计的另一种方法，我们解决问题通过将它转化为一个我们已经知道如何解决的问题的一个实例。例如：为了检测链表中的重复项，我们可以**预排序**这个链表，然后通过一次扫描检查是否有相邻的两个元素是相同的。

递归

上面的关于排序和搜索的例子有一个引人注目的特征：解决一个大小为 n 的问题，可以将其分成两半，然后处理一个或多个大小为 n/2 的问题。一种一般的方式来实现这种方法是使用**递归**。我们定义一个函数 f，从而简化了问题，并调用自身来解决一个或多个同样问题的更简单的实例。然后组合它们的结果成为原问题的解答。

例如：假设我们有 n 个词，要计算出它们结合在一起有多少不同的方式能组成一个词序列。如果我们只有一个词（n=1），只是一种方式组成一个序列。如果我们有 2 个词，就有 2 种方式将它们组成一个序列。3 个词有 6 种可能性。一般的，n 个词有 n × n-1 × … × 2 × 1 种方式（即 n 的阶乘）。我们可以将这些编写成如下代码：

```
>>> def factorial1(n):
...     result = 1
...     for i in range(n):
...         result *= (i+1)
...     return result
```

但是，也可以使用一种递归算法来解决这个问题，该算法基于以下观察：假设我们有办法为 n-1 个不同的词构建所有的排列。然后对于每个这样的排列，有 n 个地方我们可以插入一个新词：开始、结束或任意两个词之间的 n-2 个空隙。因此，我们简单的将 n-1 个词的解决方案数乘以 n 的值。我们还需要**基础案例**，也就是说，如果我们有一个词，只有一个顺序。我们可以将这些编写成如下代码：

```
>>> def factorial2(n):
...     if n == 1:
...         return 1
...     else:
...         return n * factorial2(n-1)
```

这两种算法解决同样的问题。一个使用迭代，而另一个使用递归。我们可以用递归处理深层嵌套的对象，例如：WordNet 的上位词层次。让我们计数给定同义词集 s 为根的上位词层次的大小。我们会找到 s 的每个下位词的大小，然后将它们加到一起（我们也将加 1 表示同义词集本身）。下面的函数 size1()做这项工作；注意函数体中包括 size1()的递归调用：

```
>>> def size1(s):
...     return 1 + sum(size1(child) for child in s.hyponyms())
```

我们也可以设计一种这个问题的迭代解决方案处理层的层次结构。第一层是同义词集本身①，然后是同义词集所有的下位词，之后是所有下位词的下位词。每次循环通过查找上一层的所有下位词计算下一层③。它也保存了到目前为止遇到的同义词集的总数②。

```
>>> def size2(s):
...     layer = [s] ①
...     total = 0
...     while layer:
...         total += len(layer) ②
...         layer = [h for c in layer for h in c.hyponyms()] ③
...     return total
```

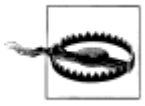
迭代解决方案不仅代码更长而且更难理解。它迫使我们程序式的思考问题，并跟踪 layer 和 total 随时间变化发生了什么。让我们满意的是两种解决方案均给出了相同的结果。我们将使用 import 语句的一个新的形式，允许我们缩写名称 wordnet 为 wn：

```
>>> from nltk.corpus import wordnet as wn
>>> dog = wn.synset('dog.n.01')
>>> size1(dog)
190
>>> size2(dog)
190
```

作为递归的最后一个例子，让我们用它来构建一个深嵌套的对象。一个**字母查找树**（letter trie）是一种可以用来索引词汇的数据结构，一次一个字母。（根据词检索 word retrieval 而得名）。例如：如果 trie 包含一个字母的查找树，那么 trie['c']是一个较小的查找树包含所有以 c 开头的词。例 4-6 演示使用 Python 字典（5.3 节）构建查找树的递归过程。插入词 chien（狗的法语），我们将 c 分类，递归的掺入 hien 到 trie['c']的子查找树中。递归继续直到词中没有剩余的字母，于是我们存储的了预期值（本例中是词 chien）。

例 4-6. 构建一个字母查找树：一个递归函数建立一个嵌套的字典结构，每一级嵌套包含给定前缀的所有单词，子查找树含有所有可能的后续词。

```
def insert(trie, key, value):
    if key:
        first, rest = key[0], key[1:]
        if first not in trie:
            trie[first] = {}
        insert(trie[first], rest, value)
    else:
        trie['value'] = value
>>> trie = nltk.defaultdict(dict)
>>> insert(trie, 'chat', 'cat')
>>> insert(trie, 'chien', 'dog')
>>> insert(trie, 'chair', 'flesh')
>>> insert(trie, 'chic', 'stylish')
>>> trie = dict(trie) # for nicer printing
>>> trie['c']['h']['a']['t']['value']
'cat'
>>> pprint.pprint(trie)
{'c': {'h': {'a': {'t': {'value': 'cat'}},
                   {'i': {'r': {'value': 'flesh'}}},
           'i': {'e': {'n': {'value': 'dog'}}}
                   {'c': {'value': 'stylish'}}}}}
```



注意！

尽管递归编程结构简单，但它是有代价的。每次函数调用时，一些状态信息需要推入堆栈，这样一旦函数执行完成可以从离开的地方继续执行。出于这个原因，迭代的解决方案往往比递归解决方案的更高效。

权衡空间与时间

我们有时可以显著的加快程序的执行，通过建设一个辅助的数据结构，例如：索引。例 4-7 实现一个简单的电影评论语料库的全文检索系统。通过索引文档集合，它提供更快的查找。

例 4-7. 一个简单的全文检索系统

```
def raw(file):
    contents = open(file).read()
    contents = re.sub(r'<.*?>', ' ', contents)
```

```
        contents = re.sub('\s+', ' ', contents)
        return contents
    def snippet(doc, term): # buggy
        text = ' '*30 + raw(doc) + ' '*30
        pos = text.index(term)
        return text[pos-30:pos+30]
    print "Building Index..."
    files = nltk.corpus.movie_reviews.abspaths()
    idx = nltk.Index((w, f) for f in files for w in raw(f).split())
    query = ''
    while query != "quit":
        query = raw_input("query> ")
        if query in idx:
            for doc in idx[query]:
                print snippet(doc, query)
        else:
            print "Not found"
```

一个更微妙的空间与时间折中的例子涉及使用整数标识符替换一个语料库的标识符。我们为语料库创建一个词汇表，每个词都被存储一次的链表，然后转化这个链表以便我们能通过查找任意词来找到它的标识符。每个文档都进行预处理，使一个词链表变成一个整数链表。现在所有的语言模型都可以使用整数。见例 4-8 中的如何为一个已标注的语料库做这个的例子的列表：

例 4-8. 预处理已标注的语料库数据，将所有的词和标注转换成整数

```
    def preprocess(tagged_corpus):
        words = set()
        tags = set()
        for sent in tagged_corpus:
            for word, tag in sent:
                words.add(word)
                tags.add(tag)
        wm = dict((w,i) for (i,w) in enumerate(words))
        tm = dict((t,i) for (i,t) in enumerate(tags))
        return [[(wm[w], tm[t]) for (w,t) in sent] for sent in tagged_corpus]
```

空间时间权衡的另一个例子是维护一个词汇表。如果你需要处理一段输入文本检查所有的词是否在现有的词汇表中，词汇表应存储为一个集合，而不是一个链表。集合中的元素会自动索引，所以测试一个大的集合的成员将远远快于测试相应的链表的成员。

我们可以使用 timeit 模块测试这种说法。Timer 类有两个参数：一个是多次执行的语句，一个是只在开始执行一次的设置代码。我们将分别使用一个整数的链表①和一个整数的集合②模拟 10 万个项目的词汇表。测试语句将产生一个随机项，它有 50%的机会在词汇表中③。

```
    >>> from timeit import Timer
    >>> vocab_size = 100000
    >>> setup_list = "import random; vocab = range(%d)" % vocab_size ①
    >>> setup_set = "import random; vocab = set(range(%d))" % vocab_size ②
```

```
>>> statement = "random.randint(0, %d) in vocab" % vocab_size * 2 ③
>>> print Timer(statement, setup_list).timeit(1000)
2.78092288971
>>> print Timer(statement, setup_set).timeit(1000)
0.0037260055542
```

执行 1000 次链表成员资格测试总共需要 2.8 秒，而在集合上的等效试验仅需 0.0037 秒，也就是说快了三个数量级！

动态规划

动态规划（Dynamic programming）是一种自然语言处理中被广泛使用的算法设计的一般方法。“programming”一词的用法与你可能想到的感觉不同，是规划或调度的意思。动态规划用于解决包含多个重叠的子问题的问题。不是反复计算这些子问题，而是简单的将它们的计算结果存储在一个查找表中。在本节的余下部分，我们将介绍动态规划，但在一个相当不同的背景：句法分析，下介绍。

Pingala 是大约生活在公元前 5 世纪的印度作家，作品有被称为《Chandas Shastra》的梵文韵律专著。Virahanka 大约在公元 6 世纪延续了这项工作，研究短音节和长音节组合产生一个长度为 n 的旋律的组合数。短音节，标记为 S，占一个长度单位，而长音节，标记为 L，占 2 个长度单位。Pingala 发现，例如：有 5 种方式构造一个长度为 4 的旋律：V_4 = {LL, SSL, SLS, LSS, SSSS}。请看，我们可以将 V_4 分成两个子集，以 L 开始的子集和以 S 开始的子集，如（1）所示。

(1) V_4 =
    LL, LSS
      i.e. L prefixed to each item of V_2 = {L, SS}
    SSL, SLS, SSSS
      i.e. S prefixed to each item of V_3 = {SL, LS, SSS}

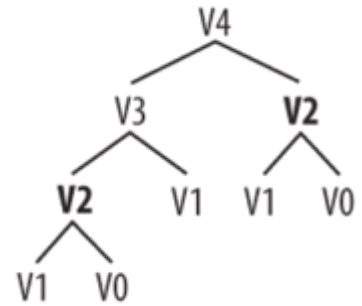
有了这个观察结果，我们可以写一个小的递归函数称为 virahanka1()来计算这些旋律，如例 4-9 所示。请注意，要计算 V_4，我们先要计算 V_3 和 V_2。但要计算 V_3，我们先要计算 V_2 和 V_1。在（2）中描述了这种**调用结构**。

例 4-9. 四种方法计算梵文旋律：（一）迭代；（二）自底向上的动态规划；（三）自上而下的动态规划；（四）内置默记法。

```
def virahanka1(n):
    if n == 0:
        return [""]
    elif n == 1:
        return ["S"]
    else:
        s = ["S" + prosody for prosody in virahanka1(n-1)]
        l = ["L" + prosody for prosody in virahanka1(n-2)]
        return s + l
def virahanka2(n):
    lookup = [[""], ["S"]]
    for i in range(n-1):
        s = ["S" + prosody for prosody in lookup[i+1]]
```

```
            l = ["L" + prosody for prosody in lookup[i]]
            lookup.append(s + l)
        return lookup[n]
    def virahanka3(n, lookup={0:[""], 1:["S"]}):
        if n not in lookup:
            s = ["S" + prosody for prosody in virahanka3(n-1)]
            l = ["L" + prosody for prosody in virahanka3(n-2)]
            lookup[n] = s + l
        return lookup[n]
    from nltk import memoize
    @memoize
    def virahanka4(n):
        if n == 0:
            return [""]
        elif n == 1:
            return ["S"]
        else:
            s = ["S" + prosody for prosody in virahanka4(n-1)]
            l = ["L" + prosody for prosody in virahanka4(n-2)]
            return s + l
    >>> virahanka1(4)
    ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
    >>> virahanka2(4)
    ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
    >>> virahanka3(4)
    ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
    >>> virahanka4(4)
    ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

(2)



正如你可以看到，V_2 计算了两次。这看上去可能并不像是一个重大的问题，但事实证明，当 n 变大时使用这种递归技术计算 V_{20}，我们将计算 V_2 4,181 次；对 V_{40} 我们将计算 V_2 63245986 次！一个更好的选择是将 V_2 的值存储在一个表中，当我们需要它时查这个表。如 V_3 等其他值也是同样的。函数 virahanka2()实现动态规划方法解决这个问题。它的工作原理是使用问题的所有较小的实例的计算结果填充一个表格（叫做 lookup），一旦我们得到了我们感兴趣的值就立即停止。此时，我们读出值，并返回它。最重要的是，每个子问题只计算了一次。

请注意，virahanka2()所采取的办法是解决较大问题前先解决较小的问题。因此，这被称为**自下而上**的方法进行动态规划。不幸的是，对于某些应用它还是相当浪费资源的，因为它计算的一些子问题在解决主问题时可能并不需要。采用**自上而下**的方法进行动态规划可避免这种计算的浪费，如例 4-9 中函数 virahanka3()所示。不同于自下而上的方法，这种方法是递归的。通过检查是否先前已存储了结果，它避免了 virahanka1()的巨大浪费。如果没有存储，就递归的计算结果，并将结果存储在表中。最后一步返回存储的结果。最后一种方法，invirahanka4()，使用一个 Python 的“装饰器”称为默记法（memoize），它会做 virahanka3()所做的繁琐的工作而不会搞乱程序。这种“默记”过程中会存储每次函数调用的结果以及使用到的参数。如果随后的函数调用了同样的参数，它会返回存储的结果，而不是重新计算。（这方面的 Python 语法超出了本书的范围。）

就这样结束了，我们简要介绍了动态规划。在 8.4 节中，我们会再次遇到它。

4.8 Python 库的样例

Python 有数以百计的第三方库，扩展 Python 功能的专门的软件包。NLTK 就是一个这样的库。要全面了解 Python 编程能力，你应该熟悉几个其他的库。它们中的大多数需要在你的计算机上进行手动安装。

Matplotlib 绘图工具

Python 有一些有用的可视化语言数据的库。Matplotlib 包支持 MATLAB 风格接口的复杂的绘图函数，从 http://matplotlib.sourceforge.net/可以得到它。

到目前为止，我们一直专注文字介绍和使用格式化输出语句按列得到输出。以图形的形式显示数值数据往往是非常有用的，因为这往往更容易检测到模式。例如：在例 3-5 中，我们看到一个数字的表格，显示按类别划分的布朗语料库中的特殊情态动词的频率。例 4-10 中的程序以图形格式展示同样的信息。输出显示在图 4-4（一个图形显示的彩色图）。

例 4-10. 布朗语料库中不同部分的情态动词频率。

```
colors = 'rgbcmyk' # red, green, blue, cyan, magenta, yellow, black
def bar_chart(categories, words, counts):
    "Plot a bar chart showing counts for each word by category"
    import pylab
    ind = pylab.arange(len(words))
    width = 1 / (len(categories) + 1)
    bar_groups = []
    for c in range(len(categories)):
        bars = pylab.bar(ind+c*width, counts[categories[c]], width,
                         color=colors[c % len(colors)])
        bar_groups.append(bars)
    pylab.xticks(ind+width, words)
    pylab.legend([b[0] for b in bar_groups], categories, loc='upper left')
    pylab.ylabel('Frequency')
    pylab.title('Frequency of Six Modal Verbs by Genre')
    pylab.show()
```

```
>>> genres = ['news', 'religion', 'hobbies', 'government', 'adventure']
>>> modals = ['can', 'could', 'may', 'might', 'must', 'will']
>>> cfdist = nltk.ConditionalFreqDist(
...                   (genre, word)
...                   for genre in genres
...                   for word in nltk.corpus.brown.words(categories=genre)
...                   if word in modals)
...
>>> counts = {}
>>> for genre in genres:
...       counts[genre] = [cfdist[genre][word] for word in modals]
>>> bar_chart(genres, modals, counts)
```

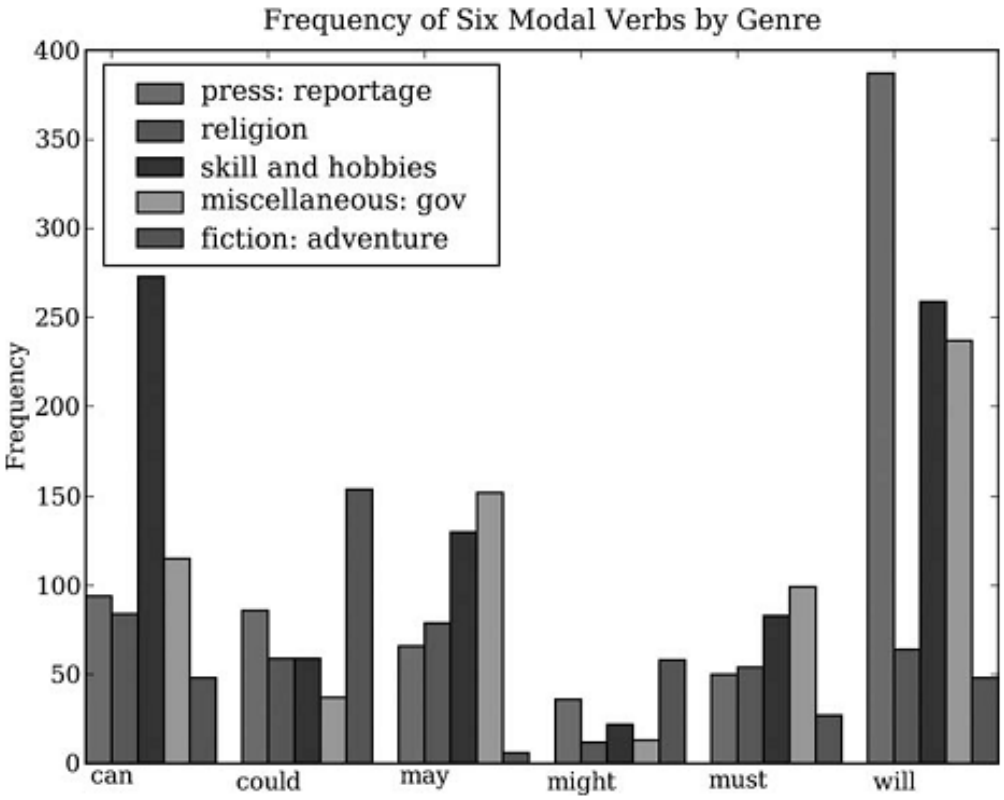


图 4-4. 条形图显示布朗语料库中不同部分的情态动词频率：这个可视化图形由例 4-10 中的程序产生。

从该柱状图可以立即看出 may 和 must 有几乎相同的相对频率。could 和 might 也一样。

也可以动态的产生这些数据的可视化图形。例如：一个使用表单输入的网页可以允许访问者指定搜索参数，提交表单，看到一个动态生成的可视化图形。要做到这一点，我们必须为 matplotlib 指定 Agg 后台，它是一个产生栅格（像素）图像的库①。下一步，我们像以前一样使用相同的 PyLab 方法，但不是用 pylab.show()显示结果在图形终端，而是使用 pylab.savefig()把它保存到一个文件②。我们指定文件名和 dpi，然后输出一些指示网页浏览器来加载该文件的 HTML 标记。

```
>>> import matplotlib
>>> matplotlib.use('Agg') ①
>>> pylab.savefig('modals.png') ②
```

```
>>> print 'Content-Type: text/html'
>>> print
>>> print '<html><body>'
>>> print '<img src="modals.png"/>'
>>> print '</body></html>'
```

NetworkX

NetworkX 包定义和操作被称为**图**的由节点和边组成的结构。它可以从 https://networkx.lanl.gov/得到。NetworkX 可以和 Matplotlib 结合使用可视化如 WordNet 的网络结构（语义网络，我们在 2.5 节介绍过）。例 4-11 中的程序初始化一个空的图③，然后遍历 WordNet 上位词层次为图添加边①。请注意，遍历是递归的②，使用在 4.7 节讨论的编程技术。结果显示如图 4-5 所示。

例 4-11. 使用 NetworkX 和 Matplotlib 库。

```
import networkx as nx
import matplotlib
from nltk.corpus import wordnet as wn
def traverse(graph, start, node):
    graph.depth[node.name] = node.shortest_path_distance(start)
    for child in node.hyponyms():
        graph.add_edge(node.name, child.name) ①
        traverse(graph, start, child) ②
def hyponym_graph(start):
    G = nx.Graph() ③
    G.depth = {}
    traverse(G, start, start)
    return G
def graph_draw(graph):
    nx.draw_graphviz(graph,
          node_size = [16 * graph.degree(n) for n in graph],
          node_color = [graph.depth[n] for n in graph],
          with_labels = False)
    matplotlib.pyplot.show()
>>> dog = wn.synset('dog.n.01')
>>> graph = hyponym_graph(dog)
>>> graph_draw(graph)
```

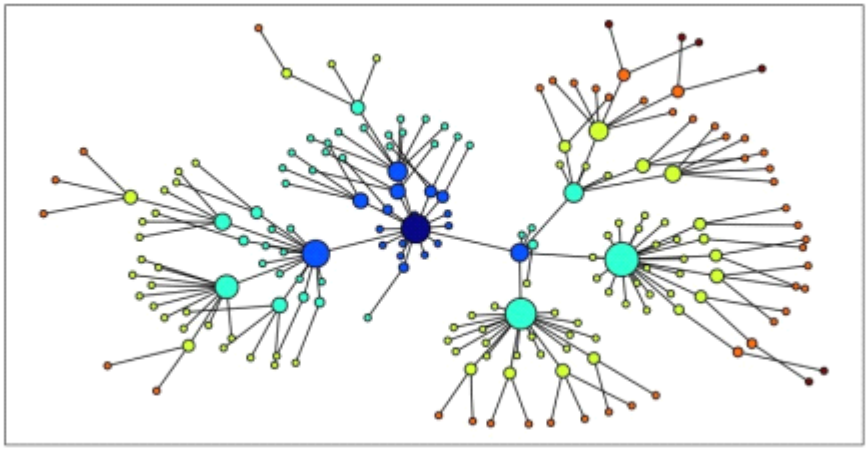


图 4-5. 使用 NetworkX 和 Matplotlib 可视化数据：WordNet 的上位词层次的部分显示，开始于 dog.n.01（中间最黑的节点）；节点的大小对应节点的孩子的数目，颜色对应节点到 dog.n.01 的距离；此可视化图形由例 4-11 中的程序产生。

CSV

语言分析工作往往涉及数据统计表，包括有关词项的信息、试验研究的参与者名单或从语料库提取的语言特征。这里有一个 CSV（Comma-separated values，逗号分隔型取值格式）格式的简单的词典片段：

```
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
```

我们可以使用 Python 的 CSV 库读写这种格式存储的文件。例如：我们可以打开一个叫做 lexicon.csv 的 CSV 文件①，并遍历它的行②：

```
>>> import csv
>>> input_file = open("lexicon.csv", "rb") ①
>>> for row in csv.reader(input_file): ②
...         print row
['sleep', 'sli:p', 'v.i', 'a condition of body and mind ...']
['walk', 'wo:k', 'v.intr', 'progress by lifting and setting down each foot ...']
['wake', 'weik', 'intrans', 'cease to sleep']
```

每一行是一个字符串列表。如果字段包含有数值数据，它们将作为字符串出现，所以都必须使用 int()或 float()转换。

NumPy

NumPy 包（基本的数值运算包）对 Python 中的数值处理提供了大量的支持。NumPy 有一个多维数组对象，它可以很容易初始化和访问：

```
>>> from numpy import array
```

```
>>> cube = array([ [[0,0,0], [1,1,1], [2,2,2]],
... [[3,3,3], [4,4,4], [5,5,5]],
... [[6,6,6], [7,7,7], [8,8,8]] ])
>>> cube[1,1,1]
4
>>> cube[2].transpose()
array([[6, 7, 8],
[6, 7, 8],
[6, 7, 8]])
>>> cube[2,1:]
array([[7, 7, 7],
[8, 8, 8]])
```

NumPy 包括线性代数函数。在这里我们进行矩阵的奇异值分解，**潜在语义分析**中使用的操作，它能帮助识别一个文档集合中的隐含概念。

```
>>> from numpy import linalg
>>> a=array([[4,0], [3,-5]])
>>> u,s,vt = linalg.svd(a)
>>> u
array([[-0.4472136 , -0.89442719],
[-0.89442719, 0.4472136 ]])
>>> s
array([ 6.32455532, 3.16227766])
>>> vt
array([[-0.70710678, 0.70710678],
[-0.70710678, -0.70710678]])
```

NLTK 中的聚类包 nltk.cluster 中广泛使用 NumPy 数组，支持包括 k-means 聚类、高斯 EM 聚类、组平均凝聚聚类以及聚类分析图。有关详细信息，请输入 help(nltk.cluster)。

其他 Python 库

还有许多其他的 Python 库，你可以使用 http://pypi.python.org/处的 Python 包索引找到它们。许多库提供了外部软件接口，例如：关系数据库（如：mysql-python）和大数据集合（如 PyLucene）。许多其他库能访问各种文件格式，如 PDF、MSWord 和 XML（pypdf, pywin32，xml.etree）、RSS 源（如 feedparser）以及电子邮箱（如 imaplib，email）。

4.9 小结

- Python 赋值和参数传递使用对象引用，例如：如果 a 是一个链表，我们分配 b = a，然后任何 a 上的操作都将修改 b，反之亦然。
- is 操作测试是否两个对象是相同的内部对象，而==测试是否两个对象是相等的。两者的区别和标识符与类型的区别相似。
- 字符串、链表和元组是不同类型的序列对象，支持常见的操作如：索引、切片、len()、sorted()和使用 in 的成员测试。

- 我们可以通过以写方式打开文件来写入文本到一个文件：ofile = open('output.txt', 'w')，然后加入内容到文件：ofile.write("Monty Python")，最后关闭文件 ofile.close()。
- 声明式的编程风格通常会产生更简洁更可读的代码；手动递增循环变量通常是不必要的。枚举一个序列，使用 enumerate()。
- 函数是一个重要的编程抽象，需要理解的关键概念有：参数传递、变量的范围和 docstrings。
- 函数作为一个命名空间：函数内部定义的名称在该函数外不可见，除非这些名称被宣布为是全局的。
- 模块允许将材料与本地的文件逻辑的关联起来。一个模块作为一个命名空间：在一个模块中定义的名称——如变量和函数——在其他模块中不可见，除非这些名称被导入。
- 动态规划是一种在 NLP 中广泛使用的算法设计技术，它存储以前的计算结果，以避免不必要的重复计算。

4.10 深入阅读

本章已经触及编程中的许多主题，一些是 Python 特有的，一些是相当普遍的。我们只是触及了表面，你可能想要阅读更多有关这些主题的内容，可以从 http://www.nltk.org/处的本章深入阅读材料开始。

Python 网站提供了大量文档。理解内置的函数和标准类型是很重要的，在 http://docs.python.org/library/functions.html 和 http://docs.python.org/library/stdtypes.html 处有描述。我们已经学习了产生器以及它们对提高效率的重要性；关于迭代器的信息，一个密切相关的话题请看 http://docs.python.org/library/itertools.html。查询你喜欢的 Python 书籍中这些主题的更多信息。使用 Python 进行多媒体处理的一个优秀的资源是(Guzdial, 2005)，包括声音文件。

使用在线 Python 文档时，要注意你安装的版本可能与你正在阅读的文档的版本不同。你可以使用 import sys; sys.version 轻松地查询你有什么样的版本。特定版本的文档在 http://www.python.org/doc/versions/处。

算法设计是计算机科学中一个丰富的领域。一些很好的出发点是(Harel, 2004)，(Levitin, 2004)和(Knuth, 2006)。(Hunt & Thomas, 2000)和(McConnell, 2004)为软件开发实践提供了有益的指导。

4.11 练习

1. ○使用 Python 的帮助功能，找出更多关于序列对象的内容。在解释器中输入 help(str)，help(list)和 help(tuple)。这会给你一个每种类型支持的函数的完整列表。一些函数名字有些特殊，两侧有下划线；正如帮助文档显示的那样，每个这样的函数对应于一些较为熟悉的东西。例如 x.__getitem__(y)仅仅是以长篇大论的方式使用 x[y]。
2. ○确定三个同时在元组和链表上都可以执行的操作。确定三个不能在元组上执行的链表操作。命名一段使用链表替代元组会产生一个 Python 错误的代码。
3. ○了解如何创建一个由单个项目组成的元组。至少有两种方法可以做到这一点。
4. ○创建一个链表 words = ['is', 'NLP', 'fun', '?']。使用一系列赋值语句（如 words[1] = words[2]）和临时变量 tmp 将这个链表转换为链表['NLP', 'is', 'fun', '!']。现在，使用元组赋值做相同的转换。

5. ○通过输入 help(cmp)阅读关于内置的比较函数 cmp 的内容。它与比较运算符在行为上有何不同？
6. ○创建一个 n-grams 的滑动窗口的方法在下面两种极端情况下是否正确：n = 1 和 n = len(sent)？
7. ○我们指出当空字符串和空链表出现在 if 从句的条件部分时，它们的判断结果是 False。在这种情况下，它们被说成出现在一个布尔上下文中。实验各种不同的布尔上下文中的非布尔表达式，看它们是否被判断为 True 或 False。
8. ○使用不等号比较字符串，如：'Monty' < 'Python'。当你做'Z' < 'a'时会发生什么？尝试具有共同前缀的字符串对，如：'Monty' < 'Montague'。阅读有关“字典排序”的内容以便了解这里发生了什么事。尝试比较结构化对象，如：('Monty', 1) < ('Monty', 2)。这与预期一样吗？
9. ○写代码删除字符串开头和结尾处的空白，并规范化词之间的空格为一个单独的空格字符。
   a. 使用 split()和 join()做这个任务。
   b. 使用正则表达式替换做这个任务。
10. ○写一个程序按长度对词排序。定义一个辅助函数 cmp_len，它在词长上使用 cmp 比较函数。
11. ◑创建一个词链表并将其存储在变量 sent1。现在赋值 sent2 = sent1。修改 sent1 中的一个项目，验证 sent2 改变了。
    a. 现在尝试同样的练习，但使用 sent2 = sent1[:]替换赋值语句。再次修改 sent1 看看 sent2 会发生什么。解释。
    b. 现在定义 text1 为一个字符串链表的链表（例如：表示由多个句子组成的文本）。现在赋值 text2 = text1[:]，分配一个新值给其中一个词，例如：text1[1][1] = 'Monty'。检查这对 text2 做了什么。解释。
    c. 导入 Python 的 deepcopy()函数（即 from copy import deepcopy），查询其文档，使用它生成任一对象的新副本。
12. ◑使用链表乘法初始化 n×m 的空字符串链表的链表，例如：word_table = [[''] * n] * m。当你设置其中一个值时会发生什么事，例如：word_table[1][2] = "hello"？解释为什么会出现这种情况。现在写一个表达式，使用 range()构造一个链表，表明它没有这个问题。
13. ◑写代码初始化一个称为 word_vowels 的二维数组的集合，处理一个词链表，添加每个词到 word_vowels[l][V]，其中 l 是词的长度，v 是它包含的元音的数量。
14. ◑写一个函数 novel10(text)输出所有在一个文本最后 10%出现而之前没有遇到过的词。
15. ◑写一个程序将一个句子表示成一个单独的字符串，分割和计数其中的词。让它输出每一个词和词的频率，每行一个，按字母顺序排列。
16. ◑阅读有关 Gematria 的内容，它是一种方法，分配一个数字给词汇，具有相同数字的词之间映射以发现文本隐藏的含义。（http://en.wikipedia.org/wiki/Gematria, http://essenes.net/gemcal.htm）。
    a. 写一个函数 gematria()，根据 letter_vals 中的字母值，累加一个词的字母的数值：

```
>>> letter_vals = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':80, 'g':3, 'h':8,
... 'i':10, 'j':10, 'k':20, 'l':30, 'm':40, 'n':50, 'o':70, 'p':80, 'q':100,
... 'r':200, 's':300, 't':400, 'u':6, 'v':6, 'w':800, 'x':60, 'y':10, 'z':7}
```

b. 处理一个语料库（如 nltk.corpus.state_union）对每个文档计数它有多少词的字母数值为 666。

c. 写一个函数 decode()来处理文本，随机替换词汇为它们的 Gematria 等值的词，以发现文本的“隐藏的含义”。

17. ◑写一个函数 shorten(text, n)处理文本，省略文本中前 n 个最频繁出现的词。它的可读性会如何？
18. ◑写代码输出词汇的索引，允许别人根据其含义查找词汇（或它们的发言；词汇条目中包含的任何属性）。
19. ◑写一个链表推导排序 WordNet 中与给定同义词集接近的同义词集的链表。例如：给定同义词集 minke_whale.n.01、orca.n.01、novel.n.01 和 tortoise.n.01，按照它们与 right_whale.n.01 的 path_distance()对它们进行排序。
20. ◑写一个函数处理一个词链表（含重复项），返回一个按照频率递减排序的词链表（没有重复项）。例如：如果输入链表中包含词 table 的 10 个实例，chair 的 9 个实例，那么在输出链表中 table 会出现在 chair 前面。
21. ◑写一个函数以一个文本和一个词汇表作为它的参数，返回在文本中出现但不在词汇表中的一组词。两个参数都可以表示为字符串链表。你能使用 set.difference()在一行代码中做这些吗？
22. ◑从 Python 的标准库的 operator 模块导入 itemgetter()函数（即 from operator import itemgetter）。创建一个包含几个词的链表 words。现在尝试调用：sorted(words, key=itemgetter(1))和 sorted(words, key=itemgetter(-1))。解释 itemgetter()正在做什么。
23. ◑写一个递归函数 lookup(trie, key)在查找树中查找一个关键字，返回找到的值。扩展这个函数，当一个词由其前缀唯一确定时返回这个词（例如：vanguard 是以 vang-开头的唯一的词，所以 lookup(trie, 'vang')应该返回与 lookup(trie, 'vanguard')相同的内容）。
24. ◑阅读关于“关键字联动”的内容（(Scott & Tribble, 2006)的第 5 章）。从 NLTK 的莎士比亚语料库中提取关键字，使用 NetworkX 包，画出关键字联动网络。
25. ◑阅读有关字符串编辑距离和 Levenshtein 算法的内容。尝试 nltk.edit_dist()提供的实现。这用的是动态规划的何种方式？它使用的是自下而上或自上而下的方法吗？[也请参阅 http://norvig.com/spell-correct.html]
26. ◑Catalan 数出现在组合数学的许多应用中，包括解析树的计数（8.6 节）。该级数可以定义如下：$C_0 = 1$，$C_{n+1} = \Sigma_{0..n}(C_i C_{n-i})$。

a. 编写一个递归函数计算第 n 个 Catalan 数 C_n。

b. 现在写另一个函数使用动态规划做这个计算。

c. 使用 timeit 模块比较当 n 增加时这些函数的性能。

27. ●重现有关著作权鉴定的(Zhao & Zobel, 2007)中的一些结果
28. ●研究性别特异词汇选择，看你是否可以重现一些 http://www.clintoneast.com/articles/words.php 的结果。
29. ●写一个递归函数漂亮的按字母顺序排列输出一个查找树，例如：

```
chair: 'flesh'
---t: 'cat'
--ic: 'stylish'
---en: 'dog'
```

30. ●在查找树数据结构的帮助下，编写一个递归函数处理文本，定位在每个词的独特点，

并丢弃每个词的其余部分。这样压缩了多少？产生的文本可读性如何？

31. ●以一个单独的长字符串的形式，获取一些原始文本。使用 Python 的 textwrap 模块将它分割成多行。现在，写代码在词之间添加额外的空格，以调整输出。每一行必须具有相同的宽度，每行约均匀分布。没有行以空白开始或结束。
32. ●开发一个简单的挖掘总结工具，输出一个文档中包含最高总词频的句子。使用 FreqDist()计数词频，并使用 sum 累加每个句子中词的频率。按照句子的得分排序。最后，输出文档中 n 个最高得分的句子。仔细检查你的程序的设计，尤其是你的关于这种双重的排序的方法。确保程序写的尽可能明了。
33. ●开发你自己的 NgramTagger 类，从 NLTK 中的类继承，封装第 5 章所述的已标注的训练和测试数据的词汇表缩减方法。确保 unigram 和默认回退标注器有机会获得全部词汇。
34. ●阅读下面的关于形容词的语义倾向文章。使用 NetworkX 包可视化一个形容词的网络，其中的边表示相同和不同的语义倾向（见 http://www.aclweb.org/anthology/P97-1023）。
35. ●设计一个算法找出一个文档集合中“统计学上不可能的短语”（见 http://www.amazon.com/gp/search-inside/sipshelp.html）。
36. ●写一个程序实现发现一种 n×n 的四方联词的蛮力算法：纵横字谜，它的第 n 行的词与第 n 列的词相同。讨论见 http://itre.cis.upenn.edu/ myl/languagelog/archives/002679.html。

第 5 章 分类和标注词汇

早在小学你就学过名词、动词、形容词和副词之间的差异。这些“词类”不是闲置的文法家的发明，而是对许多语言处理任务都有用的分类。正如我们将看到的，这些分类源于对文本中词的分布的简单的分析。本章的目的是要回答下列问题：

1. 什么是词汇分类，在自然语言处理中它们是如何使用？
2. 一个好的存储词汇和它们的分类的 Python 数据结构是什么？
3. 我们如何自动标注文本中词汇的词类？

一路上，我们将介绍 NLP 的一些基本技术，包括序列标注、N-gram 模型、回退和评估。这些技术在许多方面都很有用，标注为我们提供了一个表示它们的简单的上下文。我们还将看到标注为何是典型的 NLP 流水线中继分词之后的第二个步骤。

将词汇按它们的**词性**（parts-of-speech，POS）分类以及相应的标注它们的过程被称为**词性标注**（part-of-speech tagging, POS tagging）或干脆简称**标注**。词性也称为**词类**或**词汇范畴**。用于特定任务的标记的集合被称为一个**标记集**。我们在本章的重点是利用标记和自动标注文本。

5.1 使用词性标注器

一个**词性标注器**（part-of-speech tagger 或 POS tagger）处理一个词序列，为每个词附加一个词性标记（不要忘记 `import nltk`）：

```
>>> text = nltk.word_tokenize("And now for something completely different")
>>> nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'),
('completely', 'RB'), ('different', 'JJ')]
```

在这里我们看到 and 是 CC，并列连词；now 和 completely 是 RB，副词；for 是 IN，介词；something 是 NN，名词；different 是 JJ，形容词。

NLTK 中提供了每个标记的文档，可以使用标记来查询，如：`nltk.help.upenn_tagset('RB')`，或正则表达式，如：nltk.help.upenn_brown_tagset('NN.*')。一些语料库有标记集文档的 README 文件；见 `nltk.`*name*`.readme()`，用语料库的名称替换 name。

让我们来看看另一个例子，这次包括一些同形同音异义词：

```
>>> text = nltk.word_tokenize("They refuse to permit us to obtain the refuse permit")
>>> nltk.pos_tag(text)
[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'),
('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]
```

请注意 refuse 和 permit 都以一般现在时动词（VBP）和名词（NN）形式出现。例如：refUSE 是一个动词，意为“拒绝”，而 REFuse 是一个名词，意思是“垃圾”（即它们不是同音词）。因此，我们需要知道正在使用哪一个词以便能正确读课文。（出于这个原因，文本转语音系统通常要进行词性标注。）

轮到你来：很多词，如 ski 和 race，可以作为名词或动词而发音没有区别。你能想出其他的吗？提示：想想一个常见的东西，尝试把词 to 放到它前面，看它是否也是一个动词；或者想想一个动作，尝试把 the 放在它前面，看它是否也是一个名词。现在用这个词的两种用途造句，在这句话上运行 POS 标注器。

词汇类别，如“名词”，和词性标记，如 NN，看上去似乎有其用途，但在细节上将使许多读者感到晦涩。你可能想知道为什么要引进这种额外的信息。这些类别中很多都源于对文本中词的分布的浅层的分析。考虑下面的分析，涉及 woman（名词），bought（动词），over（介词）和 the（限定词）。text.similar()方法为一个词 w 找出所有上下文 w1ww2，然后找出所有出现在相同上下文中的词 w'，即 w1w'w2。

```
>>> text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())
>>> text.similar('woman')
Building word-context index...
man time day year car moment world family house country child boy
state job way war girl place room word
>>> text.similar('bought')
made said put done seen had found left given heard brought got been
was set told took in felt that
>>> text.similar('over')
in on to of and for with from at by that into as up out down through
is all about
>>> text.similar('the')
a his this their its her an that our any all one these my in your no
some other and
```

请看搜索 woman 找到名词；搜索 bought 找到的大部分是动词；搜索 over 一般会找到介词；搜索 the 找到几个限定词。一个标注器能够正确识别一个句子的上下文中的这些词的标记。例如：The woman bought over $150,000 worth of clothes。

一个标注器也可以为我们对未知词的认识过程建模；例如：我们可以根据词根 scrobble 猜测 scrobbling 可能是一个动词，并有可能发生在 he was scrobbling 这样的上下文中。

5.2 标注语料库

表示已标注的标识符

按照 NLTK 的约定，一个已标注的标识符使用一个由标识符和标记组成的元组来表示。我们可以使用函数 str2tuple()从表示一个已标注的标识符的标准字符串创建一个这样的特殊元组：

```
>>> tagged_token = nltk.tag.str2tuple('fly/NN')
>>> tagged_token
('fly', 'NN')
>>> tagged_token[0]
'fly'
>>> tagged_token[1]
```

'NN'

我们可以直接从一个字符串构造一个已标注的标识符的链表。第一步是对字符串分词以便能访问单独的*词/标记*字符串，然后将每一个转换成一个元组（使用 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/PP
... said/VBD ``/`` ARE/BER well/QL operated/VBN and/CC follow/VB generally/R
... 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'), ... ('.', '.')]
```

读取已标注的语料库

NLTK 中包括的若干语料库**已标注**了词性。下面是一个你用文本编辑器打开一个布朗语料库的文件就能看到的例子：

```
The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl said/vbd Friday/nr an/at inves-
tigation/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 ./.
```

其他语料库使用各种格式存储词性标记。NLTK 中的语料库阅读器提供了一个统一的接口，使你不必理会这些不同的文件格式。与刚才提取并显示的上面的文件不同，布朗语料库的语料库阅读器按如下所示的方式表示数据。注意：部分词性标记已转换为大写的；自从布朗语料库发布以来，这已成为标准的做法。

```
>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ...]
>>> nltk.corpus.brown.tagged_words(simplify_tags=True)
[('The', 'DET'), ('Fulton', 'N'), ('County', 'N'), ...]
```

只要语料库包含已标注的文本，NLTK 的语料库接口都将有一个 tagged_words()方法。下面是一些例子，再次使用布朗语料库所示的输出格式：

```
>>> 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'), (',', ','), ...]
```

并非所有的语料库都采用同一组标记；看前面提到的标记集的帮助函数和 readme()方法中的文档。最初，我们想避免这些标记集的复杂化，所以我们使用一个内置的到一个简化的标记集的映射：

```
>>> 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'), (',', ','), ...]

NLTK 中还有其他几种语言的已标注语料库，包括中文，印地语，葡萄牙语，西班牙语，荷兰语和加泰罗尼亚语。这些通常含有非 ASCII 文本，当输出较大的结构如列表时，Python 总是以十六进制显示这些。

```
>>> nltk.corpus.sinica_treebank.tagged_words()
[('\xe4\xb8\x80', 'Neu'), ('\xe5\x8f\x8b\xe6\x83\x85', 'Nad'), ...]
>>> nltk.corpus.indian.tagged_words()
[('\xe0\xa6\xae\xe0\xa6\xb9\xe0\xa6\xbf\xe0\xa6\xb7\xe0\xa7\x87\xe0\xa6\xb0', 'NN'),
('\xe0\xa6\xb8\xe0\xa6\xa8\xe0\xa7\x8d\xe0\xa6\xa4\xe0\xa6\xbe\xe0\xa6\xa8', 'NN'),
...]
>>> nltk.corpus.mac_morpho.tagged_words()
[('Jersei', 'N'), ('atinge', 'V'), ('m\xe9dia', 'N'), ...]
>>> nltk.corpus.conll2002.tagged_words()
[('Sao', 'NC'), ('Paulo', 'VMI'), ('(', 'Fpa'), ...]
>>> nltk.corpus.cess_cat.tagged_words()
[('El', 'da0ms0'), ('Tribunal_Suprem', 'np0000o'), ...]
```

如果你的环境设置正确，有适合的编辑器和字体，你应该能够以人可读的方式显示单个字符串。例如：图 5-1 显示的使用 nltk.corpus.indian 访问的数据。

如果语料库也被分割成句子，将有一个 tagged_sents()方法将已标注的词划分成句子，而不是将它们表示成一个大链表。在我们开始开发自动标注器时，这将是有益的，因为它们在句子链表上被训练和测试，而不是词。

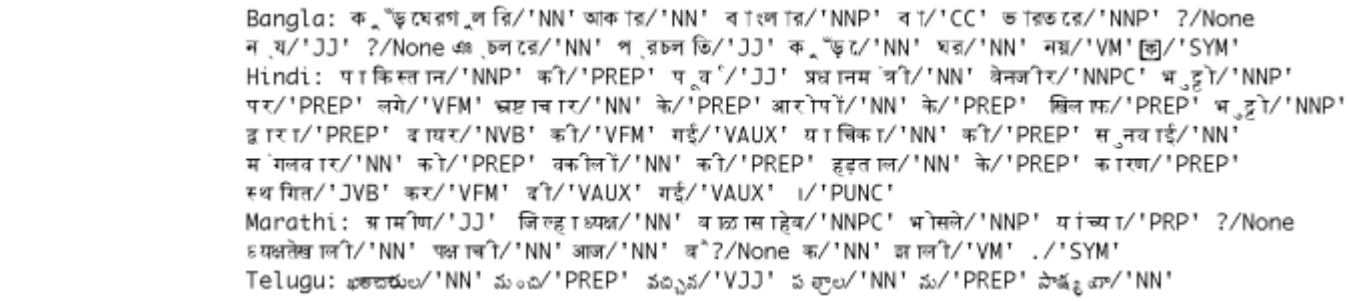


图 5-1. 四种印度语言的词性标注数据：孟加拉语、印地语、马拉地语和泰卢固语。

简化的词性标记集

已标注的语料库使用许多不同的标记集约定来标注词汇。为了帮助我们开始，我们将看一看一个简化的标记集（表 5-1 所示）。

表 5-1. 简化的标记集

标记	含义	例子
ADJ	形容词	new, good, high, special, big, local
ADV	动词	really, already, still, early, now
CNJ	连词	and, or, but, if, while, although
DET	限定词	the, a, some, most, every, no
EX	存在量词	there, there's
FW	外来词	dolce, ersatz, esprit, quo, maitre
MOD	情态动词	will, can, would, may, must, should

N	名词	year, home, costs, time, education
NP	专有名词	Alison, Africa, April, Washington
NUM	数词	twenty-four, fourth, 1991, 14:24
PRO	代词	he, their, her, its, my, I, us
P	介词	on, of, at, with, by, into, under
TO	词 to	to
UH	感叹词	ah, bang, ha, whee, hmpf, oops
V	动词	is, has, get, do, make, see, run
VD	过去式	said, took, told, made, asked
VG	现在分词	making, going, playing, working
VN	过去分词	given, taken, begun, sung
WH	Wh 限定词	who, which, when, what, where, how

让我们来看看这些标记中哪些是布朗语料库的新闻类中最常见的：

```
>>> from nltk.corpus import brown
>>> brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
>>> tag_fd = nltk.FreqDist(tag for (word, tag) in brown_news_tagged)
>>> tag_fd.keys()
['N', 'P', 'DET', 'NP', 'V', 'ADJ', ',', '.', 'CNJ', 'PRO', 'ADV', 'VD', ...]
```

轮到你来：使用 tag_fd.plot(cumulative=True)为上面显示的频率分布绘图。标注为上述列表中的前五个标记的词的百分比是多少？

我们可以使用这些标记做强大的搜索，结合一个图形化的 POS 一致性工具 nltk.app.concordance()。用它来寻找任一词和 POS 标记的组合，如：N N N N, hit/VD, hit/VN 或 the ADJ man。

名词

名词一般指的是人、地点、事情或概念，例如：女人、苏格兰、图书、情报。名词可能出现在限定词和形容词之后，可以是动词的主语或宾语，如表 5-2 所示。

表 5-2 一些名词的句法模式

词	限定词之后	动词的主语
woman	the woman who I saw yesterday ...	the woman sat down
Scotland	the Scotland I remember as a child ...	Scotland has five million people
book	the book I bought yesterday ...	this book recounts the colonization of Australia
intelligence	the intelligence displayed by the child ...	Mary's intelligence impressed her teachers

简化的名词标记对普通名词是 N，如：书，对专有名词是 NP，如苏格兰。

让我们检查一些已标注的文本，看看哪些词类出现在一个名词前，频率最高的在最前面。首先，我们构建一个双连词链表，它的成员是它们自己的词-标记对，例如：(('The', 'DET'), ('Fulton', 'NP'))和(('Fulton', 'NP'), ('County', 'N'))。然后，我们构建了一个双连词的标记部分的 FreqDist。

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

这证实了我们的断言：名词出现在限定词和形容词之后，包括数字形容词（数词，标注

为 NUM）。

动词

动词是用来描述事件和行动的词，例如：fall 和 eat，如表 5-3 所示。在一个句子中，动词通常表示涉及一个或多个名词短语所指示物的关系。

表 5-3. 一些动词的句法模式

词	例子	修饰符与修饰语（斜体字）
fall	Rome fell	Dot com stocks *suddenly* fell *like* a stone
eat	Mice eat cheese	John ate the pizza *with* gusto

新闻文本中最常见的动词是什么？让我们按频率排序所有动词：

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

请注意，频率分布中计算的项目是词-标记对。由于词汇和标记是成对的，我们可以把词作作为条件，标记作为事件，使用条件-事件对的链表初始化一个条件频率分布。这让我们看到了一个给定的词的标记的频率顺序列表。

```
>>> cfd1 = nltk.ConditionalFreqDist(wsj)
>>> cfd1['yield'].keys()
['V', 'N']
>>> cfd1['cut'].keys()
['V', 'VD', 'N', 'VN']
```

我们可以颠倒配对的顺序，这样标记作为条件，词汇作为事件。现在我们可以看到对于一个给定的标记可能的词。

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

要弄清 VD（过去式）和 VN（过去分词）之间的区别，让我们找到可以同是 VD 和 VN 的词汇，看看一些它们周围的文字：

```
>>> [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')]
```

在这种情况下，我们可以看到过去分词 kicked 前面是助动词 have 的形式。这是普遍真实的吗？

轮到你来：通过 cfd2['VN'].keys()指定一个过去分词的链表，尝试收集所有直接在链表中项目前面的词-标记对。

形容词和副词

另外两个重要的词类是**形容词**和**副词**。形容词修饰名词，可以作为修饰符（如：the large pizza 中的 large）或谓语（如：the pizza is large）。英语形容词可以有内部结构（如：the falling stocks 中的 fall+ing）。副词修饰动词，指定时间、方式、地点或动词描述的事件的方向（如：the stocks fell quickly 中的 quickly）。副词也可以修饰的形容词（如：Mary's teacher was really nice 中的 really）。

英语中还有几个封闭的词类，如介词，**冠词**（也常称为**限定词**）（如：the，a），**情态动词**（如：should，may），**人称代词**（如：she，they）。每个词典和语法对这些词的分类都不同。

轮到你来：如果你对这些词性中的一些不确定，使用 nltk.app.concordance()学习它们，或看 YouTube 上的一些《School-house Rock!》语法视频，或查询 5-9 节。

未简化的标记

让我们找出每个名词类型中最频繁的名词。例 5-1 中的程序找出所有以 NN 开始的标记，并为每个标记提供了几个示例词汇。你会看到有许多名词的变种；最重要的含有$的名词所有格,含有 S 的复数名词（因为复数名词通常以 s 结尾），以及含有 P 的专有名词。此外，大多数的标记都有后缀修饰符：-NC 表示引用，-HL 表示标题中的词，-TL 表示标题（布朗标记的特征）。

例 5-1. 找出最频繁的名词标记的程序

```
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']
```

当我们开始在本章后续部分创建词性标注器时，我们将使用这些未简化的标记。

探索已标注的语料库

让我们简要地回过来探索语料库，我们在前面的章节中看到过，这次我们探索 POS 标记。

假设我们正在研究词 often，想看看它是如何在文本中使用的。我们可以试着看看跟在 often 后面的词汇：

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

然而，它使用 `tagged_words()`方法查看跟随词的词性标记可能更有指导性。

```
>>> brown_lrnd_tagged = brown.tagged_words(categories='learned', simplify_tags=True)
>>> tags = [b[1] for (a, b) in nltk.ibigrams(brown_lrnd_tagged) if a[0] == 'often']
>>> fd = nltk.FreqDist(tags)
>>> fd.tabulate()
VN V VD DET ADJ ADV P CNJ , TO VG WH VBZ .
15 12 8 5 5 4 4 3 3 1 1 1 1 1
```

请注意 often 后面最高频率的词性是动词。名词从来没有在这个位置出现（在这个特别的语料中）。

接下来，让我们看一些较大范围的上下文，找出涉及特定标记和词序列的词（在这种情况下，“<Verb>到<Verb>”）。在例 5-2 中，我们考虑句子中的每个三词窗口①，检查它们是否符合我们的标准②。如果标记匹配，我们输出对应的词③。

例 5-2. 使用 POS 标记寻找三词短语。

```
from nltk.corpus import brown
def process(sentence):
    for (w1,t1), (w2,t2), (w3,t3) in nltk.trigrams(sentence): ①
        if (t1.startswith('V') and t2 == 'TO' and t3.startswith('V')): ②
            print w1, w2, w3 ③
>>> for tagged_sent in brown.tagged_sents():
...     process(tagged_sent)
...
combined to achieve
```

```
continue to place
serve to protect
wanted to wait
allowed to place
expected to become
...
```

最后，让我们看看与它们的标记关系高度模糊不清的词。了解为什么要标注这样的词是因为它们各自的上下文可以帮助我们弄清楚标记之间的区别。

```
>>> brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
>>> data = nltk.ConditionalFreqDist((word.lower(), tag)
...                                 for (word, tag) in brown_news_tagged)
>>> for word in data.conditions():
...     if len(data[word]) > 3:
...         tags = data[word].keys()
...         print word, ' '.join(tags)
...
best ADJ ADV NP V
better ADJ ADV V DET
close ADV ADJ V N
cut V N VN VD
even ADV DET ADJ V
grant NP N V -
hit V VD VN N
lay ADJ V NP VD
left VD ADJ N VN
like CNJ V ADJ P -
near P ADV ADJ DET
open ADJ V N ADV
past N ADJ DET P
present ADJ ADV V N
read V VN VD NP
right ADJ N DET ADV
second NUM ADV DET N
set VN V VD N -
that CNJ V WH DET
```

轮到你来：打开 POS 一致性工具 **nltk.app.concordance()**和加载完整的布朗语料库（简化标记集）。现在挑选一些上面代码例子末尾处列出的词，看看词的标记如何与词的上下文相关。例如：搜索 near 会看到所有混合在一起的形式，搜索 near/ADJ 会看到它作为形容词使用，near N 会看到只是名词跟在后面的情况，等等。

5.3 使用 Python 字典映射词及其属性

正如我们已经看到， (word, tag)形式的一个已标注词是词和词性标记的关联。一旦我们开始做词性标注，我们将会创建分配一个标记给一个词的程序，标记是在给定上下文中最可能的标记。我们可以认为这个过程是从词到标记的**映射**。在 Python 中最自然的方式存储映射是使用所谓的**字典**数据类型（在其他的编程语言又称为**关联数组**或**哈希数组**）。在本节中，我们来看看字典，看它如何能表示包括词性在内的各种不同的语言信息。

索引链表 VS 字典

我们已经看到，文本在 Python 中被视为一个词链表。链表的一个重要的属性是我们可以通过给出其索引来“看”特定项目，例如：text1[100]。请注意我们如何指定一个数字，然后取回一个词。我们可以把链表看作一种简单的表格，如图 5-2 所示。

0	Call
1	me
2	Ishmael
3	.

图 5-2. 链表查找：一个整数索引帮助我们访问 Python 链表的内容。

对比这种情况与频率分布（1.3 节），在那里我们指定一个词然后取回一个数字，如：fdist['monstrous']，它告诉我们一个给定的词在文本中出现的次数。用词查询类似与使用一本字典。一些例子如图 5-3 所示。

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

图 5-3. 字典查询：我们使用一个关键字，如某人的名字、一个域名或一个英文单词，访问一个字典的条目；映射（map）、哈希表（hashmap）、哈希（hash）、关联数组（associative array）是字典的其他名字。

在电话簿中，我们用名字查找一个条目得到一个数字。当我们在浏览器中输入一个域名，计算机查找它得到一个 IP 地址。一个词频表允许我们查一个词找出它在一个文本集合中的频率。在所有这些情况，都是从名称映射到数字，而不是其他如链表那样的方式。在一般情况下，我们希望能够在任意类型的信息之间映射。表 5-4 列出了各种语言学对象以及它们的映射。

表 5-4. 语言学对象从键到值的映射

语言学对象	映射来自	映射到
文档索引	词	页面列表（找到词的地方）
同义词	词意	同义词列表

词典	中心词	词条项（词性、意思定义、词源）
比较单词列表	注释术语	同源词（词列表，每种语言一个）
词形分析	表面形式	形态学分析（词素组件列表）

大多数情况下，从一个“词”映射到一些结构化对象。例如：一个文档索引从一个词（可以表示为一个字符串）映射到页面列表（表示为一个整数列表）。本节中，我们将看到在 Python 中如何表示这些映射。

Python 字典

Python 提供了一个**字典**数据类型，可用来做任意类型之间的映射。它更像是一个传统的字典，给你一种高效的方式来查找事物。然而，正如我们从表 5-4 看到的，它具有更广泛的用途。

为了说明这一点，我们定义 **pos** 为一个空字典，然后给它添加四个项目，指定一些词的词性。使用熟悉的方括号将条目添加到字典。

```
>>> pos = {}
>>> pos
{}
>>> pos['colorless'] = 'ADJ'  ①
>>> pos
{'colorless': 'ADJ'}
>>> pos['ideas'] = 'N'
>>> pos['sleep'] = 'V'
>>> pos['furiously'] = 'ADV'
>>> pos  ②
{'furiously': 'ADV', 'ideas': 'N', 'colorless': 'ADJ', 'sleep': 'V'}
```

例子中，①说的是 colorless 的词性是形容词，或者更具体地说：在字典 pos 中，键'colorless'被分配了值'ADJ'。当我们检查 pos 的值②时，我们看到一个键-值对的集合。一旦我们以这样的方式填充了字典，就可以使用键来检索值：

```
>>> pos['ideas']
'N'
>>> pos['colorless']
'ADJ'
```

当然，我们可能会无意中使用一个尚未分配值的键。

```
>>> pos['green']
Traceback (most recent call last):
File "<stdin>", line 1, in ?
KeyError: 'green'
```

这就提出了一个重要的问题。与链表和字符串中我们可以用 len()算出哪些整数是合法索引不同，我们如何算出一个字典的合法键？如果字典不是太大，我们可以简单地通过查看变量 pos 检查它的内容。正如在前面②行中所看到，这为我们提供了键-值对。请注意它们的顺序与最初放入它们的顺序不同，这是因为字典不是序列而是映射（见图 5-3），它的键并不按固有的顺序。

另外，要找到键，我们可以将字典转换成一个链表①或在需要使用链表的地方使用字典，如作为 sorted()的参数②或用在 for 循环中③。

```
>>> list(pos) ①
['ideas', 'furiously', 'colorless', 'sleep']
>>> sorted(pos) ②
['colorless', 'furiously', 'ideas', 'sleep']
>>> [w for w in pos if w.endswith('s')] ③
['colorless', 'ideas']
```

当你输入 list(pos)时，你看到的可能会与这里显示的顺序不同。如果你想看到有序的键，只需要对它们进行排序。

与使用一个 for 循环遍历字典中的所有键一样，我们可以使用 for 循环输出字典的内容：

```
>>> for word in sorted(pos):
... print word + ":", pos[word]
...
colorless: ADJ
furiously: ADV
sleep: V
ideas: N
```

最后，字典的方法 keys()、values()和 items()允许我们访问作为单独的链表的键、值以及键-值对。我们甚至可以按它们的第一个元素排序元组①（如果第一个元素相同，就使用它们的第二个元素）。

```
>>> 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()): ①
... print key + ":", val
...
colorless: ADJ
furiously: ADV
ideas: N
sleep: V
```

我们要确保当我们在字典中查找某词时，一个键只得到一个值。现在假设我们试图用字典来存储可同时作为动词和名词的词 sleep：

```
>>> pos['sleep'] = 'V'
>>> pos['sleep']
'V'
>>> pos['sleep'] = 'N'
>>> pos['sleep']
'N'
```

一开始，pos['sleep']给的值是'V'。但是，它立即被一个新值'N'覆盖了。换句话说，字典中只能有'sleep'的一个条目。然而，有一个方法可以在该项目中存储多个值：我们使用一个链表值，例如：pos['sleep'] = ['N', 'V']。事实上，这就是我们在 2.4 节中看到的

CMU 发音字典，它为一个词存储多个发音。

定义字典

我们可以使用键-值对格式创建字典。有两种方式做这个，我们通常会使用第一个：

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

请注意：字典的键必须是不可改变的类型，如字符串和元组。如果我们尝试使用可变键定义字典会得到一个 TypeError：

```
>>> pos = {['ideas', 'blogs', 'adventures']: 'N'}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

默认字典

如果我们试图访问一个不在字典中的键，会得到一个错误。然而，如果一个字典能为这个新键自动创建一个条目并给它一个默认值，如 0 或者一个空链表，将是有用的。自从 Python 2.5 以来，一种特殊的称为 defaultdict 的字典已经出现。（考虑到有读者使用 Python 2.4，NLTK 提供了 nltk.defaultdict。）为了使用它，我们必须提供一个参数，用来创建默认值，如：int、float、str、list、dict、tuple。

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

这些默认值实际上是将其他对象转换为指定类型的函数（例如：int("2")、list("2")）。当它们被调用的时候没有参数——也就是说，int()、list()——分别返回 0 和[]。

前面的例子中指定字典项的默认值为一个特定的数据类型的默认值。然而，也可以指定任何我们喜欢的默认值，只要提供可以无参数的被调用产生所需值的函数的名子。让我们回到我们的词性的例子，创建一个任一条目的默认值是'N'的字典①。当我们访问一个不存在的条目②时，它会自动添加到字典③。

```
>>> pos = nltk.defaultdict(lambda: 'N') ①
>>> pos['colorless'] = 'ADJ'
>>> pos['blog'] ②
'N'
>>> pos.items()
```

```
[('blog', 'N'), ('colorless', 'ADJ')] ③
```

这个例子使用一个 lambda 表达式，在 4.4 节介绍过。这个 lambda 表达式没有指定参数，所以我们用不带参数的括号调用它。因此，下面的 f 和 g 的定义是等价的：

```
>>> f = lambda: 'N'
>>> f()
'N'
>>> def g():
... return 'N'
>>> g()
'N'
```

让我们来看看默认字典如何被应用在较大规模的语言处理任务中。许多语言处理任务——包括标注——费很大力气来正确处理文本中只出现过一次的词。如果有一个固定的词汇和没有新词会出现的保证，它们会有更好的表现。我们可以预处理一个文本，在一个默认字典的帮助下，替换低频词汇为一个特殊的“超出词汇表”标识符，UNK（out of vocabulary）。（你能不看下面的想出如何做吗？）

我们需要创建一个默认字典，映射每个词为它们的替换词。最频繁的 n 个词将被映射到它们自己。其他的被映射到 UNK。

```
>>> alice = nltk.corpus.gutenberg.words('carroll-alice.txt')
>>> vocab = nltk.FreqDist(alice)
>>> v1000 = list(vocab)[:1000]
>>> mapping = nltk.defaultdict(lambda: 'UNK')
>>> for v in v1000:
... mapping[v] = v
...
>>> alice2 = [mapping[v] for v in alice]
>>> alice2[:100]
['UNK', 'Alice', "'", 's', 'Adventures', 'in', 'Wonderland', 'by', 'UNK', 'UNK',
'UNK', 'UNK', 'CHAPTER', 'I', '.', 'UNK', 'the', 'Rabbit', '-', 'UNK', 'Alice',
'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by', 'her',
'sister', 'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 'to', 'do',
':', 'once', 'or', 'twice', 'she', 'had', 'UNK', 'into', 'the', 'book', 'her',
'sister', 'was', 'UNK', ',', 'but', 'it', 'had', 'no', 'pictures', 'or', 'UNK',
'in', 'it', ',', "'", 'and', 'what', 'is', 'the', 'use', 'of', 'a', 'book', ",'",
'thought', 'Alice', "'", 'without', 'pictures', 'or', 'conversation', '?"', ...]
>>> len(set(alice2))
1001
```

递增地更新字典

我们可以使用字典计数出现的次数，模拟图 1-3 所示的计数词汇的方法。首先初始化一个空的 defaultdict，然后处理文本中每个词性标记。如果标记以前没有见过，就默认计数为零。每次我们遇到一个标记，就使用+=运算符递增它的计数（见例 5-3）。

例 5-3. 递增地更新字典，按值排序。

```
>>> counts = nltk.defaultdict(int)
>>> from nltk.corpus import brown
>>> for (word, tag) in brown.tagged_words(categories='news'):
...     counts[tag] += 1
...
>>> counts['N']
22226
>>> list(counts)
['FW', 'DET', 'WH', "''", 'VBZ', 'VB+PPO', '``', ')', 'ADJ', 'PRO', '*', '-', ...]
>>> from operator import itemgetter
>>> sorted(counts.items(), key=itemgetter(1), reverse=True)
[('N', 22226), ('P', 10845), ('DET', 10648), ('NP', 8336), ('V', 7313), ...]
>>> [t for t, c in sorted(counts.items(), key=itemgetter(1), reverse=True)]
['N', 'P', 'DET', 'NP', 'V', 'ADJ', ',', '.', 'CNJ', 'PRO', 'ADV', 'VD', ...]
```

例 5-3 的列表演示了一个重要的按值排序一个字典的习惯用法，按频率递减顺序显示词汇。sorted()的第一个参数是要排序的项目，它是由一个 POS 标记和一个频率组成的元组的链表。第二个参数使用函数 itemgetter()指定排序键。在一般情况下，itemgetter(n)返回一个函数，这个函数可以在一些其他序列对象上被调用获得这个序列的第 n 个元素的。

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

sorted()的最后一个参数指定项目是否应被按相反的顺序返回，即频率值递减。

在例 5-3 的开头还有第二个有用的习惯用法，那里我们初始化一个 defaultdict，然后使用 for 循环来更新其值。下面是一个示意图版本：

```
>>> my_dictionary = nltk.defaultdict(function to create default value)
>>> for item in sequence:
...         my_dictionary[item_key] is updated with information about item
```

下面是这种模式的另一个实例，我们按它们最后两个字母索引词汇：

```
>>> last_letters = nltk.defaultdict(list)
>>> words = nltk.corpus.words.words('en')
>>> for word in words:
...         key = word[-2:]
...         last_letters[key].append(word)
...
>>> last_letters['ly']
['abactinally', 'abandonedly', 'abasedly', 'abashedly', 'abashlessly', 'abbreviately',
'abdominally', 'abhorrently', 'abidingly', 'abiogenetically', 'abiologically', ...]
>>> last_letters['zy']
['blazy', 'bleezy', 'blowzy', 'boozy', 'breezy', 'bronzy', 'buzzy', 'Chazy', ...]
```

下面的例子使用相同的模式创建一个颠倒顺序的词字典。（你可能会试验第 3 行，以便能弄清楚为什么这个程序能运行。）

```
>>> anagrams = nltk.defaultdict(list)
>>> for word in words:
...         key = ''.join(sorted(word))
...         anagrams[key].append(word)
...
>>> anagrams['aeilnrt']
['entrail', 'latrine', 'ratline', 'reliant', 'retinal', 'trenail']
```

由于积累这样的词是如此常用的任务，NLTK 以 nltk.Index()的形式提供一个创建 defaultdict(list)更方便的方式。

```
>>> anagrams = nltk.Index((''.join(sorted(w)), w) for w in words)
>>> anagrams['aeilnrt']
['entrail', 'latrine', 'ratline', 'reliant', 'retinal', 'trenail']
```

nltk.Index 是一个额外支持初始化的 defaultdict(list)。类似的，nltk.FreqDist 本质上是一个额外支持初始化的 defaultdict(int)（附带排序和绘图方法）。

复杂的键和值

我们可以使用具有复杂的键和值的默认字典。让我们研究一个词可能的标记的范围，给定词本身和它前一个词的标记。我们将看到这些信息如何被一个 POS 标注器使用。

```
>>> pos = nltk.defaultdict(lambda: nltk.defaultdict(int))
>>> brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
>>> for ((w1, t1), (w2, t2)) in nltk.ibigrams(brown_news_tagged): ①
...         pos[(t1, w2)][t2] += 1 ②
...
>>> pos[('DET', 'right')] ③
defaultdict(<type 'int'>, {'ADV': 3, 'ADJ': 9, 'N': 3})
```

这个例子使用一个字典，它的条目的默认值也是一个字典（其默认值是 int()，即 0）。请注意我们如何遍历已标注语料库的双连词，每次遍历处理一个词-标记对①。每次通过循环时，我们更新字典 pos 中的条目 (t1, w2)，一个标记和它后面的词②。当我们在 pos 中查找一个项目时，我们必须指定一个复合键③，然后得到一个字典对象。一个 POS 标注器可以使用这些信息来决定词 right，前面是一个限定词时，应标注为 ADJ。

颠倒字典

字典支持高效查找，只要你想获得任意键的值。如果 d 是一个字典，k 是一个键，输入 d[K]，就立即获得值。给定一个值查找对应的键要慢一些和麻烦一些：

```
>>> counts = nltk.defaultdict(int)
>>> for word in nltk.corpus.gutenberg.words('milton-paradise.txt'):
...     counts[word] += 1
...
```

```
>>> [key for (key, value) in counts.items() if value == 32]
['brought', 'Him', 'virtue', 'Against', 'There', 'thine', 'King', 'mortal',
'every', 'been']
```

如果我们希望经常做这样的一种“反向查找”，建立一个映射值到键的字典是有用的。在没有两个键具有相同的值情况，这是一个容易的事。只要得到字典中的所有键-值对，并创建一个新的值-键对字典。下一个例子演示了用键-值对初始化字典 pos 的另一种方式。

```
>>> pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
>>> pos2 = dict((value, key) for (key, value) in pos.items())
>>> pos2['N']
'ideas'
```

首先让我们将我们的词性字典做的更实用些，使用字典的 update()方法加入再一些词到 pos 中，创建多个键具有相同的值的情况。这样一来，刚才看到的反向查找技术就将不再起作用（为什么不？）。作为替代，我们不得不使用 append()积累词和每个词性，如下所示：

```
>>> 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']
```

现在，我们已经颠倒了字典 pos，可以查任意词性找到所有具有此词性的词。可以使用 NLTK 中的索引支持更容易的做同样的事，如下所示：

```
>>> pos2 = nltk.Index((value, key) for (key, value) in pos.items())
>>> pos2['ADV']
['peacefully', 'furiously']
```

表 5-5 了给出 Python 字典方法的总结。

表 5-5. Python 字典方法：常用的方法与字典相关习惯用法的总结

示例	说明
d = {}	创建一个空的字典，并将分配给 d
d[key] = value	分配一个值给一个给定的字典键
d.keys()	字典的键的链表
list(d)	字典的键的链表
sorted(d)	字典的键，排序
key in d	测试一个特定的键是否在字典中
for key in d	遍历字典的键
d.values()	字典中的值的链表
dict([(k1,v1), (k2,v2), ...])	从一个键-值对链表创建一个字典
d1.update(d2)	添加 d2 中所有项目到 d1
defaultdict(int)	一个默认值为 0 的字典

5.4 自动标注

本章的其余部分将探讨以不同的方式来给文本自动添加词性标记。我们将看到一个词的

标记依赖于这个词和它在句子中的上下文。出于这个原因，我们将处理（已标注）句子层次而不是词汇层次的数据。我们以加载将要使用的数据开始。

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
```

默认标注器

最简单的标注器是为每个标识符分配同样的标记。这似乎是一个相当平庸的一步，但它建立了标注器性能的一个重要的底线。为了得到最好的效果，我们用最有可能的标记标注每个词。让我们找出哪个标记是最有可能的（现在使用未简化标记集）：

```
>>> tags = [tag for (word, tag) in brown.tagged_words(categories='news')]
>>> nltk.FreqDist(tags).max()
'NN'
```

现在我们可以创建一个将所有词都标注成 NN 的标注器。

```
>>> raw = 'I do not like green eggs and ham, I do not like them Sam I am!'
>>> tokens = nltk.word_tokenize(raw)
>>> default_tagger = nltk.DefaultTagger('NN')
>>> default_tagger.tag(tokens)
[('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('green', 'NN'),
('eggs', 'NN'), ('and', 'NN'), ('ham', 'NN'), (',', 'NN'), ('I', 'NN'),
('do', 'NN'), ('not', 'NN'), ('like', 'NN'), ('them', 'NN'), ('Sam', 'NN'),
('I', 'NN'), ('am', 'NN'), ('!', 'NN')]
```

不出所料，这种方法的表现相当不好。在一个典型的语料库中，它只标注正确了八分之一的标识符，正如我们在这里看到的：

```
>>> default_tagger.evaluate(brown_tagged_sents)
0.13089484257215028
```

默认的标注器给每一个单独的词分配标记，即使是之前从未遇到过的词。碰巧的是，一旦我们处理了几千词的英文文本之后，大多数新词都将是名词。正如我们将看到的，这意味着，默认标注器可以帮助我们提高语言处理系统的稳定性。我们将很快回来讲述这个。

正则表达式标注器

正则表达式标注器基于匹配模式分配标记给标识符。例如：我们可能会猜测任一以 ed 结尾的词都是动词过去分词，任一以's 结尾的词都是名词所有格。可以用一个正则表达式的列表表示这些：

```
>>> 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)
... ]
```

请注意，这些是顺序处理的，第一个匹配上的会被使用。现在我们可以建立一个标注器，并用它来标记一个句子。做完这一步会有约五分之一是正确的。

```
>>> regexp_tagger = nltk.RegexpTagger(patterns)
>>> regexp_tagger.tag(brown_sents[3])
[('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative', 'NN'), ('handful', 'NN'),
('of', 'NN'), ('such', 'NN'), ('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),
("''", 'NN'), (',', 'NN'), ('the', 'NN'), ('jury', 'NN'), ('said', 'NN'), (',', 'NN'),
('``', 'NN'), ('considering', 'VBG'), ('the', 'NN'), ('widespread', 'NN'), ...]
>>> regexp_tagger.evaluate(brown_tagged_sents)
0.20326391789486245
```

最终的正则表达式«.*»是一个全面捕捉的，标注所有词为名词。除了作为正则表达式标注器的一部分重新指定这个，这与默认标注器是等效的（只是效率低得多）。有没有办法结合这个标注器和默认标注器呢？我们将很快看到如何做到这一点。

轮到你来：看看你能不能想出一些模式，提高上面所示的正则表达式标注器的性能。（请注意：6.1 节描述部分自动化这类工作的方法。）

查询标注器

很多高频词没有 NN 标记。让我们找出 100 个最频繁的词，存储它们最有可能的标记。然后我们可以使用这个信息作为“查找标注器”（NLTK UnigramTagger）的模型：

```
>>> fd = nltk.FreqDist(brown.words(categories='news'))
>>> cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
>>> most_freq_words = fd.keys()[:100]
>>> likely_tags = dict((word, cfd[word].max()) for word in most_freq_words)
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags)
>>> baseline_tagger.evaluate(brown_tagged_sents)
0.45578495136941344
```

现在应该并不奇怪，仅仅知道 100 个最频繁的词的标记就使我们能正确标注很大一部分标识符（近一半，事实上）。让我们来看看它在一些未标注的输入文本上做的如何：

```
>>> sent = brown.sents(categories='news')[3]
>>> baseline_tagger.tag(sent)
[('``', '``'), ('Only', None), ('a', 'AT'), ('relative', None),
('handful', None), ('of', 'IN'), ('such', None), ('reports', None),
('was', 'BEDZ'), ('received', None), ("''", "''"), (',', ','),
('the', 'AT'), ('jury', None), ('said', 'VBD'), (',', ','),
('``', '``'), ('considering', None), ('the', 'AT'), ('widespread', None),
('interest', None), ('in', 'IN'), ('the', 'AT'), ('election', None),
(',', ','), ('the', 'AT'), ('number', None), ('of', 'IN'),
('voters', None), ('and', 'CC'), ('the', 'AT'), ('size', None),
('of', 'IN'), ('this', 'DT'), ('city', None), ("''", "''"), ('.', '.')]
```

许多词都被分配了一个 None 标签，因为它们不在 100 个最频繁的词之中。在这些情况下，我们想分配默认标记 NN。换句话说，我们要先使用查找表，如果它不能指定一个标记就使用默认标注器，这个过程叫做**回退**（5.5 节）。我们可以做到这个，通过指定一个标注器作为另一个标注器的参数，如下所示。现在查找标注器将只存储名词以外的词的词-标记对，只要它不能给一个词分配标记，它将会调用默认标注器。

```
>>> baseline_tagger = nltk.UnigramTagger(model=likely_tags,
...                                      backoff=nltk.DefaultTagger('NN'))
```

让我们把所有这些放在一起，写一个程序来创建和评估具有一定范围的查找标注器（例如 5-4）。

例 5-4. 查找标注器的性能，使用不同大小的模型。

```
def performance(cfd, wordlist):
    lt = dict((word, cfd[word].max()) for word in wordlist)
    baseline_tagger = nltk.UnigramTagger(model=lt, backoff=nltk.DefaultTagger('NN'))
    return baseline_tagger.evaluate(brown.tagged_sents(categories='news'))
def display():
    import pylab
    words_by_freq = list(nltk.FreqDist(brown.words(categories='news')))
    cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
    sizes = 2 ** pylab.arange(15)
    perfs = [performance(cfd, words_by_freq[:size]) for size in sizes]
    pylab.plot(sizes, perfs, '-bo')
    pylab.title('Lookup Tagger Performance with Varying Model Size')
    pylab.xlabel('Model Size')
    pylab.ylabel('Performance')
    pylab.show()
>>> display()
```

观察图 5-4，随着模型规模的增长，最初的性能增加迅速，最终达到一个稳定水平，这时模型的规模大量增加性能的提高很小。（这个例子使用 pylab 绘图软件包，在 4.8 节讨论过）。

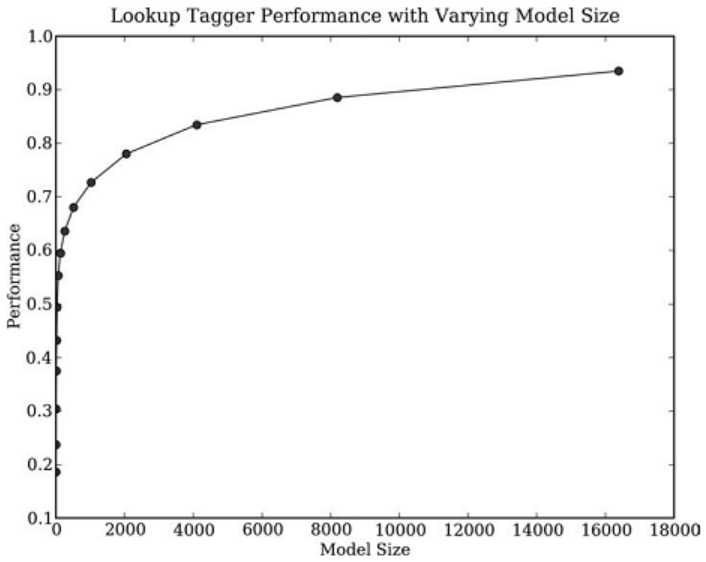


图 5-4. 查找标注器

评估

在前面的例子中，你会注意到对准确性得分的强调。事实上，这些工具的性能评估是 NLP 的一个中心主题。回想图 1-5 的处理流程；一个模块输出中的任何错误都在下游模块大大的放大。

我们对比专家分配的标记来评估一个标注器的性能。由于我们通常很难获得专业和公正的人的判断，所以使用**黄金标准**测试数据来代替。这是一个已经手动标注并作为自动系统评估标准而被接受的语料库。当标注器对给定词猜测的标记与黄金标准标记相同，标注器被视为是正确的，。

当然，设计和实施原始的黄金标准标注的也是人，更深入的分析可能会显示黄金标准中的错误，或者可能最终会导致一个修正的标记集和更复杂的指导方针。然而，黄金标准就目前有关的自动标注器的评估而言被定义成“正确的”。

开发一个已标注语料库是一个重大的任务。除了数据，它会产生复杂的工具、文档和实践，为确保高品质的标注。标记集和其他编码方案不可避免地依赖于一些理论主张，不是所有的理论主张都被共享。然而，语料库的创作者往往竭尽全力使他们的工作尽可能理论中立，以最大限度地提高其工作的有效性。我们将在第 11 章讨论创建一个语料库的挑战。

5.5 N-gram 标注

一元标注（Unigram Tagging）

一元标注器基于一个简单的统计算法：对每个标识符分配这个独特的标识符最有可能的标记。例如：它将分配标记 JJ 给词 frequent 的所有出现，因为 frequent 用作一个形容词（例如：a frequent word）比用作一个动词（例如：I frequent this cafe）更常见。一个一元标注器的行为就像一个查找标注器（5.4 节），除了有一个更方便的建立它的技术，称为**训练**。在下面的代码例子中，我们训练一个一元标注器，用它来标注一个句子，然后评估：

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='news')
>>> brown_sents = brown.sents(categories='news')
>>> unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
>>> unigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments', 'NNS'),
('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace', 'NN'), ('type', 'NN'),
(',', ','), ('being', 'BEG'), ('on', 'IN'), ('the', 'AT'), ('ground', 'NN'),
('floor', 'NN'), ('so', 'QL'), ('that', 'CS'), ('entrance', 'NN'), ('is', 'BEZ'),
('direct', 'JJ'), ('.', '.')]
>>> unigram_tagger.evaluate(brown_tagged_sents)
0.9349006503968017
```

我们**训练**一个 UnigramTagger，通过在我们初始化标注器时指定已标注的句子数据作为参数。训练过程中涉及检查每个词的标记，将所有词的最可能的标记存储在一个字典里面，

这个字典存储在标注器内部。

分离训练和测试数据

现在，我们正在一些数据上训练一个标注器，必须小心不要在相同的数据上测试，如我们在前面的例子中的那样。一个只是记忆它的训练数据，而不试图建立一个一般的模型的标注器会得到一个完美的得分，但在标注新的文本时将是无用的。

相反，我们应该分割数据，90％为测试数据，其余 10％为测试数据：

```
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> size
4160
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
>>> unigram_tagger = nltk.UnigramTagger(train_sents)
>>> unigram_tagger.evaluate(test_sents)
0.81202033290142528
```

虽然得分更糟糕了，但是我们现在对这种标注器是无用的（如：它在前所未见的文本上的性能）有了一个更好地了解。

一般的 N-gram 的标注

在基于 unigrams 处理一个语言处理任务时，我们使用上下文中的一个项目。标注的时候，我们只考虑当前的标识符，与更大的上下文隔离。给定一个模型，我们能做的最好的是为每个词标注其先验的最可能的标记。这意味着我们将使用相同的标记标注一个词，如 wind，不论它出现的上下文是 the wind 还是 to wind。

一个 **n-gram 标注器**是一个 unigram 标注器的一般化，它的上下文是当前词和它前面 n-1 个标识符的词性标记，如图 5-5 所示。要选择的标记是圆圈里的 t_n，灰色阴影的是上下文。在图 5-5 所示的 n-gram 标注器的例子中，我们让 n= 3，也就是说，我们考虑当前词的前两个词的标记。一个 n-gram 标注器挑选在给定的上下文中最有可能的标记。

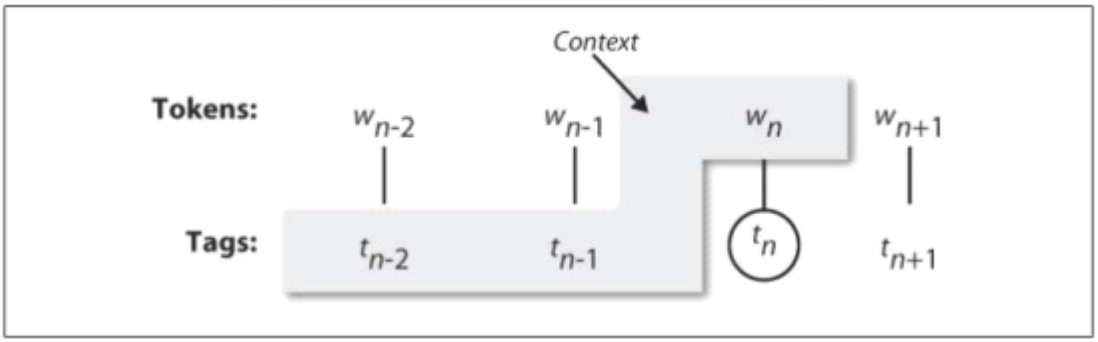


图 5-5. 标注器上下文

1-gram 标注器是一元标注器（unigram tagger）另一个名称：即用于标注一个标识符的上下文的只是标识符本身。2-gram 标注器也称为二元标注器（bigram taggers），3-gram 标注器也称为三元标注器（trigram taggers）。

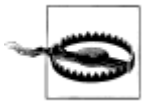
NgramTagger 类使用一个已标注的训练语料库来确定对每个上下文哪个词性标记最有可能。在这里，我们看到一个 n-gram 标注器的特殊情况，即一个 bigram 标注器。首先，我们训练它，然后用它来标注未标注的句子：

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments', 'NNS'),
('are', 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace', 'NN'),
('type', 'NN'), (',', ','), ('being', 'BEG'), ('on', 'IN'), ('the', 'AT'),
('ground', 'NN'), ('floor', 'NN'), ('so', 'CS'), ('that', 'CS'),
('entrance', 'NN'), ('is', 'BEZ'), ('direct', 'JJ'), ('.', '.')]
>>> unseen_sent = brown_sents[4203]
>>> bigram_tagger.tag(unseen_sent)
[('The', 'AT'), ('population', 'NN'), ('of', 'IN'), ('the', 'AT'), ('Congo', 'NP'),
('is', 'BEZ'), ('13.5', None), ('million', None), (',', None), ('divided', None),
('into', None), ('at', None), ('least', None), ('seven', None), ('major', None),
('``', None), ('culture', None), ('clusters', None), ("''", None), ('and', None),
('innumerable', None), ('tribes', None), ('speaking', None), ('400', None),
('separate', None), ('dialects', None), ('.', None)]
```

请注意，bigram 标注器能够标注训练中它看到过的句子中的所有词，但对一个没见过的句子表现很差。只要遇到一个新词（如 13.5），就无法给它分配标记。它不能标注下面的词（如：million），即使是在训练过程中看到过的，只是因为在训练过程中从来没有见过它前面有一个 None 标记的词。因此，标注器标注句子的其余部分也失败了。它的整体准确度得分非常低：

```
>>> bigram_tagger.evaluate(test_sents)
0.10276088906608193
```

当 n 越大，上下文的特异性就会增加，我们要标注的数据中包含训练数据中不存在的上下文的几率也增大。这被称为数据稀疏问题，在 NLP 中是相当普遍的。因此，我们的研究结果的精度和覆盖范围之间需要有一个权衡（这与信息检索中的**精度/召回权衡**有关）。



注意！

N-gram 标注器不应考虑跨越句子边界的上下文。因此，NLTK 的标注器被设计用于句子链表，一个句子是一个词链表。在一个句子的开始，t_{n-1} 和前面的标记被设置为 None。

组合标注器

解决精度和覆盖范围之间的权衡的一个办法是尽可能的使用更精确的算法，但却在很多时候落后于具有更广覆盖范围的算法。例如：我们可以按如下方式组合 bigram 标注器、unigram 标注器和一个默认标注器：

1. 尝试使用 bigram 标注器标注标识符。
2. 如果 bigram 标注器无法找到一个标记，尝试 unigram 标注器。
3. 如果 unigram 标注器也无法找到一个标记，使用默认标注器。

大多数 NLTK 标注器允许指定一个回退标注器。回退标注器自身可能也有一个回退标注器：

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents, backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.84491179108940495
```

轮到你来：通过定义一个名为 t3 的 TrigramTagger，扩展前面的例子，它是 t2 的回退标注器。

请注意，我们在标注器初始化时指定回退标注器，从而使训练能利用回退标注器。

于是，如果在一个上下文中 bigram 标注器将分配与它的 unigram 回退标注器一样的标记，那么 bigram 标注器丢弃训练的实例。这样保持尽可能小的 bigram 标注器模型。我们可以进一步指定一个标注器需要看到一个上下文的多个实例才能保留它。例如：nltk.BigramTagger(sents, cutoff=2, backoff=t1)将会丢弃那些只看到一次或两次的上下文。

标注生词

我们标注生词的方法仍然是回退到一个正则表达式标注器或一个默认标注器。这些都无法利用上下文。因此，如果我们的标注器遇到词 blog，训练过程中没有看到过，它会分配相同的标记，不论这个词出现的上下文是 the blog 还是 to blog。我们怎样才能更好地处理这些生词，或**词汇表以外**的项目？

一个有用的基于上下文标注生词的方法是限制一个标注器的词汇表为最频繁的 n 个词，使用 5.3 节中的方法替代每个其他的词为一个特殊的词 UNK。训练时，一个 unigram 标注器可能会学到 UNK 通常是一个名词。然而，n-gram 标注器会检测它的一些其他标记中的上下文。例如：如果前面的词是 to（标注为 TO），那么 UNK 可能会被标注为一个动词。

存储标注器

在大语料库上训练一个标注器可能需要大量的时间。没有必要在每次我们需要的时候训练一个标注器，很容易将一个训练好的标注器保存到一个文件以后重复使用。让我们保存我们的标注器 t2 到文件 t2.pkl：

```
>>> from cPickle import dump
>>> output = open('t2.pkl', 'wb')
>>> dump(t2, output, -1)
>>> output.close()
```

现在，我们可以在一个单独的 Python 进程中载入我们保存的标注器：

```
>>> from cPickle import load
>>> input = open('t2.pkl', 'rb')
>>> tagger = load(input)
```

```
>>> input.close()
```
现在，让我们的检查它是否可以用来标注：
```
>>> 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'), ('.', '.')]
```

性能限制

一个 n-gram 标注器的性能上限是什么？考虑一个 trigram 标注器的情况。它遇到多少词性歧义的情况？我们可以根据经验决定这个问题的答案：

```
>>> cfd = nltk.ConditionalFreqDist(
...             ((x[1], y[1], z[0]), z[1])
...             for sent in brown_tagged_sents
...             for x, y, z in nltk.trigrams(sent))
>>> ambiguous_contexts = [c for c in cfd.conditions() if len(cfd[c]) > 1]
>>> sum(cfd[c].N() for c in ambiguous_contexts) / cfd.N()
0.049297702068029296
```

因此，1/20 的 trigrams 是有歧义的。给定当前单词及其前两个标记，根据训练数据，在 5%的情况中，有一个以上的标记可能合理地分配给当前词。假设我们总是挑选在这种含糊不清的上下文中最有可能的标记，可以得出 trigram 标注器性能的一个下界。

调查标注器性能的另一种方法是研究它的错误。有些标记可能会比别的更难分配，可能需要专门对这些数据进行预处理或后处理。一个方便的方式查看标注错误是**混淆矩阵**。它用图表表示期望的标记（黄金标准）与实际由标注器产生的标记：

```
>>> test_tags = [tag for sent in brown.sents(categories='editorial')
...                   for (word, tag) in t2.tag(sent)]
>>> gold_tags = [tag for (word, tag) in brown.tagged_words(categories='editorial')]
>>> print nltk.ConfusionMatrix(gold, test)
```

基于这样的分析，我们可能会决定修改标记集。或许标记之间很难做出的区分可以被丢弃，因为它在一些较大的处理任务的上下文中并不重要。

分析标注器性能界限的另一种方式来自人类标注者之间并非 100%的意见一致。

一般情况下，标注过程会消除区别：例如：当所有的人称代词被标注为 PRP 时，词的特性通常会失去。与此同时，标注过程引入了新的区别从而去除了含糊之处：例如：deal 标注为 VB 或 NN。这种消除某些区别并引入新的区别的特点是标注的一个重要的特征，有利于分类和预测。当我们引入一个标记集的更细的划分时，在 n-gram 标注器决定什么样的标记分配给一个特定的词时，可以获得关于左侧上下文的更详细的信息。然而，标注器同时也将需要做更多的工作来划分当前的标识符，只是因为有更多可供选择的标记。相反，使用较少的区别（如简化的标记集），标注器有关上下文的信息会减少，为当前标识符分类的选择范围也较小。

我们已经看到，训练数据中的歧义导致标注器性能的上限。有时更多的上下文能解决这

些歧义。然而，在其他情况下，如(Abney, 1996)中指出的，只有参考语法或现实世界的知识，才能解决歧义。尽管有这些缺陷，词性标注在用统计方法进行自然语言处理的兴起过程中起到了核心作用。1990 年代初，统计标注器令人惊讶的精度是一个惊人的示范：可以不用更深的语言学知识解决一小部分语言理解问题，即词性消歧。这个想法能再推进吗？第 7 章中，我们将看到，它可以。

跨句子边界标注

一个 n-gram 标注器使用最近的标记作为为当前的词选择标记的指导。当标记一个句子的第一个词时，trigram 标注器将使用前面两个标识符的词性标记，这通常会是前面句子的最后一个词和句子结尾的标点符号。然而，在前一句结尾的词的类别与下一句的开头的通常没有关系。

为了应对这种情况，我们可以使用已标注句子的链表来训练、运行和评估标注器，如例 5-5 所示。

例 5-5. 句子层面的 N-gram 标注

```
brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')
size = int(len(brown_tagged_sents) * 0.9)
train_sents = brown_tagged_sents[:size]
test_sents = brown_tagged_sents[size:]
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_sents, backoff=t0)
t2 = nltk.BigramTagger(train_sents, backoff=t1)
>>> t2.evaluate(test_sents)
0.84491179108940495
```

5.6 基于转换的标注

n-gram 标注器的一个潜在的问题是它们的 n-gram 表的大小（或语言模型）。如果使用各种语言技术的标注器部署在移动计算设备上，在模型大小和标注器性能之间取得平衡是很重要的。使用回退标注器的 n-gram 标注器可能存储 trigram 和 bigram 表，这是很大的稀疏阵列，可能有数亿条条目。

第二个问题是关于上下文的。n-gram 标注器从前面的上下文中获得的唯一的信息是标记，虽然词本身可能是一个有用的信息源。n-gram 模型使用上下文中的词的其他特征为条件是不切实际的。在本节中，我们考察 Brill 标注，一种归纳标注方法，它的性能很好，使用的模型只有 n-gram 标注器的很小一部分。

Brill 标注是一种*基于转换的学习*，以它的发明者命名。一般的想法很简单：猜每个词的标记，然后返回和修复错误的。在这种方式中，Brill 标注器陆续将一个不良标注的文本转换成一个更好的。与 n-gram 标注一样，这是有监督的学习方法，因为我们需要已标注的训练数据来评估标注器的猜测是否是一个错误。然而，不像 n-gram 标注，它不计数观察结果，只编制一个转换修正规则链表。

Brill 标注的的过程通常是与绘画类比来解释的。假设我们要画一棵树，包括大树枝、树枝、小枝、叶子和一个统一的天蓝色背景的所有细节。不是先画树然后尝试在空白处画蓝

色，而是简单的将整个画布画成蓝色，然后通过在蓝色背景上上色“修正”树的部分。以同样的方式，我们可能会画一个统一的褐色的树干再回过头来用更精细的刷子画进一步的细节。Brill 标注使用了同样的想法：以大笔画开始，然后修复细节，一点点的细致的改变。让我们看看下面的例子：

(1) The President said he will ask Congress to increase grants to states for vocational rehabilitation.（总统表示：他将要求国会增加拨款给各州用于职业康复。）

我们将研究两个规则的运作：（a）当前面的词是 TO 时，替换 NN 为 VB；（b）当下一个标记是 NNS 时，替换 TO 为 IN。表 5-6 说明了这一过程，首先使用 unigram 标注器标注，然后运用规则修正错误。

表 5-6. Brill 标注的步骤

Phrase	to	increase	grants	to	states	for	vocational	rehabilitation
Unigram	TO	NN	NNS	TO	NNS	IN	JJ	NN
Rule1		VB						
Rule2				IN				
Output	TO	VB	NNS	IN	NNS	IN	JJ	NN
Gold	TO	VB	NNS	IN	NNS	IN	JJ	NN

在此表中，我们看到两个规则。所有这些规则由以下形式的模板产生：“替换 T1 为 T2 在上下文 C 中。”典型的上下文是之前或之后的词的内容或标记，或者当前词的两到三个词范围内出现的一个特定标记。在其训练阶段，T1，T2 和 C 的标注器猜测值创造出数以千计的候选规则。每一条规则都根据其净收益打分：它修正的不正确标记的数目减去它错误修改的正确标记的数目。

Brill 标注器的另一个有趣的特性：规则是语言学可解释的。与采用潜在的巨大的 n-gram 表的 n-gram 标注器相比，我们并不能从直接观察这样的一个表中学到多少东西，而 Brill 标注器学到的规则可以。例 5-6 演示了 NLTK 中的 Brill 标注器。

例 5-6. Brill 标注器演示：标注器有一个“X→Y 如果前面的词是 Z”的形式的模板集合；这些模板中的变量是创建“规则”的特定词和标记的实例；规则的得分是它纠正的错误例子的数目减去正确的情况下它误报的数目；除了训练标注器，演示还显示了剩余的错误。

```
>>> nltk.tag.brill.demo()
Training Brill tagger on 80 sentences...
Finding initial useful rules...
    Found 6555 useful rules.
             B      |
     S   F   r   O  |          Score = Fixed - Broken
     c   i   o   t  |   R      Fixed = num tags changed incorrect -> correct
     o   x   k   h  |   u      Broken = num tags changed correct -> incorrect
     r   e   e   e  |   l      Other = num tags changed incorrect -> incorrect
     e   d   n   r  |   e
------------------+-------------------------------------------------
    12  13   1   4  | NN -> VB if the tag of the preceding word is 'TO'
     8   9   1  23  | NN -> VBD if the tag of the following word is 'DT'
     8   8   0   9  | NN -> VBD if the tag of the preceding word is 'NNS'
     6   9   3  16  | NN -> NNP if the tag of words i-2...i-1 is '-NONE-'
     5   8   3   6  | NN -> NNP if the tag of the following word is 'NNP'
     5   6   1   0  | NN -> NNP if the text of words i-2...i-1 is 'like'
```

```
    5    5    0    3   | NN -> VBN if the text of the following word is '*-1'
    ...
>>> print(open("errors.out").read())
                  left context |       word/test->gold        | right context
-------------------------------+---------------------------+----------------------
                               |         Then/NN->RB       | ,/, in/IN the/DT guests/N
, in/IN the/DT guests/NNS |          '/VBD->POS       | honor/NN ,/, the/DT speed
'/POS honor/NN ,/, the/DT |      speedway/JJ->NN      | hauled/VBD out/RP four/CD
NN ,/, the/DT speedway/NN|        hauled/NN->VBD  | out/RP four/CD drivers/NN
DT speedway/NN hauled/VBD|         out/NNP->RP     | four/CD drivers/NNS ,/, c
dway/NN hauled/VBD out/RP|         four/NNP->CD    | drivers/NNS ,/, crews/NNS
hauled/VBD out/RP four/CD |      drivers/NNP->NNS | ,/, crews/NNS and/CC even
P four/CD drivers/NNS ,/,    |      crews/NN->NNS     | and/CC even/RB the/DT off
NNS and/CC even/RB the/DT|      official/NNP->JJ    | Indianapolis/NNP 500/CD a
                               |        After/VBD->IN    | the/DT race/NN ,/, Fortun
ter/IN the/DT race/NN ,/,     |      Fortune/IN->NNP    | 500/CD executives/NNS dro
s/NNS drooled/VBD like/IN |  schoolboys/NNP->NNS | over/IN the/DT cars/NNS a
olboys/NNS over/IN the/DT |         cars/NN->NNS      | and/CC drivers/NNS ./.
```

5.7 如何确定一个词的分类

我们已经详细研究了词类，现在转向一个更基本的问题：如何决定一个词属于哪一类？首先应该考虑什么？在一般情况下，语言学家使用形态学、句法和语义线索确定一个词的类别。

形态学线索

一个词的内部结构可能为这个词分类提供有用的线索。举例来说：-ness 是一个后缀，与形容词结合产生一个名词，如 happy→happiness，ill→illness。因此，如果我们遇到的一个以-ness 结尾的词，很可能是一个名词。同样的，-ment 是与一些动词结合产生一个名词的后缀，如 govern→government 和 establish→establishment。

英语动词也可以是形态复杂的。例如：一个动词的**现在分词**以-ing 结尾，表示正在进行的还没有结束的行动（如：falling，eating）的意思。-ing 后缀也出现在从动词派生的名词中，如：the falling of the leaves（这被称为**动名词**）。

句法线索

另一个信息来源是一个词可能出现的典型的上下文语境。例如：假设我们已经确定了名词类。那么我们可以说，英语形容词的句法标准是它可以立即出现在一个名词前，或紧跟在词 be 或 very 后。根据这些测试，near 应该被归类为形容词：

(2) a. the near window
    b. The end is (very) near.

语义线索

最后，一个词的意思对其词汇范畴是一个有用的线索。例如：名词的众所周知的一个定义是根据语义的："一个人、地方或事物的名称。"在现代语言学，词类的语义标准受到怀疑，主要是因为它们很难规范化。然而，语义标准巩固了我们对许多词类的直觉，使我们能够在不熟悉的语言中很好的猜测词的分类。例如：如果我们都知道荷兰语词 verjaardag 的意思与英语词 birthday 相同，那么我们可以猜测 verjaardag 在荷兰语中是一个名词。然而，一些修补是必要的：虽然我们可能翻译 zij is vandaag jarig as it's her birthday today，词 jarig 在荷兰语中实际上是形容词，与英语并不完全相同。

新词

所有的语言都学习新的词汇。最近添加到牛津英语词典中的一个单词列表包括 cyberslacker、fatoush、blamestorm、SARS、cantopop、bupkis、noughties、muggle 和 robata。请注意，所有这些新词都是名词，这反映在名词被称为**开放类**。相反，介词被认为是一个**封闭类**，也就是说，只有有限的词属于这个类别（例如：above、along、at、below、beside、between、during、for、from、in、near、on、outside、over、past、through、towards、under、up、with），词类成员随着很长时间的推移才逐渐改变。

词性标记集中的形态学

普通标记集经常捕捉一些**构词**信息，即词借助它们的句法角色获得的一种形态标记的信息。例如：思考下面句子中词 go 的不同语法形式的选集：

(3) a. Go away!
b. He sometimes goes to the cafe.
c. All the cakes have gone.
d. We went on the excursion.

这些形态中的每一个——go、goes、gone 和 went——是形态学上的区别。思考形式 goes。它出现在受限制的语法环境中，需要一个第三人称单数的主语。因此，下面的句子是不合语法的。

(4) a. *They sometimes goes to the cafe.
b. *I sometimes goes to the cafe.

相比之下，gone 是过去分词形式；需要出现在 have 后面（在这个上下文中不能被 goes 替代），不能作为一个从句的主要动词出现。

(5) a. *All the cakes have goes.
b. *He sometimes gone to the cafe.

我们可以很容易地想象一个刚才讨论的有四种不同语法形式的标记集都被标注为 VB。虽然这对于某些目的已经足够了，更细粒度的标记集提供有关这些形式的有用信息，可以帮助尝试检测标记序列模式的其它处理者。布朗标记集捕捉这些区别，如表 5-7 中的总结。

表 5-7. 布朗标记集的一些构词区别

形式	类别	标记
go	基本	VB

goes	第三人称单数	VBZ
gone	过去分词	VBN
going	动名词	VBG
went	一般过去时	VBD

除了这组动词标记，动词 to be 的各种形式也有特殊的标记：be/BE，being/BEG，am/BEM，are/BER，is/BEZ，been/BEN，were/BED 和 was/BEDZ（加上额外的动词否定形式的标记）。总的来说，这种动词细粒度标记意味着使用此标记集的自动标注器能有效开展有限数量的形态分析。

大多数词性标注集使用相同的基本类别，如名词、动词、形容词和介词。然而，标记集的相互区别不仅在于它们如何细致的将词分类，也在于它们如何界定其类别。例如：is 在一个标记集可能会被简单的标注为动词，而在另一个标记集中被标注为 lexeme be 的不同形式（如在布朗语料库中）。这种标记集的变化是不可避免的，因为词性标记被以不同的方式用于不同的任务。换句话说，没有一个“正确的方式”来分配标记，只有根据目标不同或多或少有用的方法。

5.8 小结

- 词可以组成类，如名词、动词、形容词以及副词。这些类被称为词汇范畴或者词性。词性被分配短标签或者标记，如 NN 和 VB。
- 给文本中的词自动分配词性的过程称为词性标注、POS 标注或只是标注。
- 自动标注是 NLP 流程中重要的一步，在各种情况下都十分有用，包括预测先前未见过的词的行为、分析语料库中词的使用以及文本到语音转换系统。
- 一些语言学语料库，如布朗语料库，已经做了词性标注。
- 有多种标注方法，如默认标注器、正则表达式标注器、unigram 标注器、n-gram 标注器。这些都可以结合一种叫做回退的技术一起使用。
- 标注器可以使用已标注语料库进行训练和评估。
- 回退是一个组合模型的方法：当一个较专业的模型（如 bigram 标注器）不能为给定内容分配标记时，我们回退到一个较一般的模型（如 unigram 标注器）
- 词性标注是 NLP 中一个重要的早期的序列分类任务：利用局部上下文语境中的词和标记对序列中任意一点的分类决策。
- 字典用来映射任意类型之间的信息，如字符串和数字：freq['cat']=12。我们使用大括号来创建字典：pos = {}，pos = {'furiously': 'adv', 'ideas': 'n', 'colorless': 'adj'}。
- N-gram 标注器可以定义较大数值的 n，但是当 n 大于 3 时，我们常常会面临数据稀疏问题；即使使用大量的训练数据，我们看到的也只是可能的上下文的一小部分。
- 基于转换的标注学习一系列“改变标记 s 为标记 t 在上下文 c 中”形式的修复规则，每个规则会修复错误，也可能引入（较小的）错误。

5.9 深入阅读

本章的额外材料发布在 http://www.nltk.org/，包括网上免费提供的资源的链接。关于使用 NLTK 标注的更多的例子，请看在 http://www.nltk.org/howto 上的标注 HOWTO。 (Jurafsky & Martin, 2008)的第 4 章和第 5 章包含更多 n-grams 和词性标注的高级材料。其他标注

方法涉及到机器学习方法（第 6 章）。第 7 章中，我们将看到被称为分块的标注的泛化，在那里为连续的词序列分配一个单独的标记。

关于标记集的文档，请参阅 nltk.help.upenn_tagset()和 nltk.help.brown_tagset()。词汇范畴在很多语言学教科书中都有介绍，包括本书第 1 章中所列的书籍。

还有许多其他类型的标注。可以按语音合成器的指令标注，表示哪些词应强调；可以按词意数标注，表示词的哪个意思被使用；也可以按形态特征标注。每一个这些类别的标记的例子如下所示。由于篇幅所限，我们只显示一个词的标记。还要注意的是前两个例子使用 XML 风格的标签，尖括号中的元素括起已标注的词。

Speech Synthesis Markup Language (W3C SSML)

That is a <emphasis>big</emphasis> car!

SemCor: Brown Corpus tagged with WordNet senses

Space in any <wf pos="NN" lemma="form" wnsn="4">form</wf> is completely measured by the three dimensions. (Wordnet form/nn sense 4: “shape, form, configuration, contour, conformation”)

Morphological tagging, from the Turin University Italian Treebank

E' italiano , come progetto e realizzazione , il primo (PRIMO ADJ ORDIN M SING) porto turistico dell' Albania .

请注意，标注也可以在更高层次进行。下面是一个对话的行为标注的例子，来自 NLTK 中的 NPS 聊天语料库（Forsyth & Martell, 2007）。每个回合的对话按照它的交际功能分类：

Statement User117 Dude..., I wanted some of that
ynQuestion User120 m I missing something?
Bye User117 I'm gonna go fix food, I'll be back later.
System User122 JOIN
System User2 slaps User122 around a bit with a large trout.
Statement User121 18/m pm me if u tryin to chat

5.10 练习

1. ○网上搜索“spoof newspaper headlines”，找到这种宝贝：British Left Waffles on Falkland Islands 和 Juvenile Court to Try Shooting Defendant。手工标注这些头条，看看词性标记的知识是否可以消除歧义。
2. ○和别人一起，轮流挑选一个既可以是名词也可以是动词的词（如：contest）；让对方预测哪一个可能是布朗语料库中频率最高的。检查对方的预测，为几个回合打分。
3. ○分词和标注下面的句子：They wind back the clock, while we chase after the wind。包含哪些不同的发音和词类？
4. ○回顾表 5.4 中的映射。讨论你能想到的映射的其他的例子。它们从什么类型的信息映射到什么类型的信息？
5. ○在交互模式下使用 Python 解释器，实验本章中字典的例子。创建一个字典 d，添加一些条目。如果你尝试访问一个不存在的条目，如 d['xyz']，会发生什么？
6. ○尝试从字典 d 删除一个元素，使用语法 del d['abc']。检查被删除的项目。
7. ○创建两个字典，d1 和 d2，为每个添加一些条目。现在发出命令 d1.update(d2)。这做了什么？它可能是有什么用？
8. ○创建一个字典 e，表示你选择的一些词的一个单独的词汇条目。定义键如：headword、part-of-speech、sense 和 example，分配给它们适当的值。

9. ○自己验证 go 和 went 在分布上的限制，也就是说，它们不能自由地在第 5.7 节(3)中演示的那种上下文中互换。
10. ○训练一个 unigram 标注器，在一些新的文本上运行。观察有些词没有分配到标记。为什么没有？
11. ○了解词缀标注器（输入 help(nltk.AffixTagger)）。训练一个词缀标注器，在一些新的文本上运行。设置不同的词缀长度和最小词长做实验。讨论你的结果。
12. ○训练一个没有回退标注器的 bigram 标注器，在一些训练数据上运行。然后，在一些新的数据运行它。标注器的性能会发生什么？为什么呢？
13. ○我们可以使用字典指定由一个格式化字符串替换的值。阅读关于格式化字符串的 Python 库文档（http://docs.python.org/lib/typesseq-strings.html），使用这种方法以两种不同的格式显示今天的日期。
14. ◑使用 sorted()和 set()获得布朗语料库使用的标记的排序的链表，删除重复。
15. ◑写程序处理布朗语料库，找到以下问题的答案：
    a. 哪些名词常以它们复数形式而不是它们的单数形式出现？（只考虑常规的复数形式，-s 后缀形式的）。
    b. 哪个词的不同标记数目最多？它们是什么，它们代表什么？
    c. 按频率递减的顺序列出标记。前 20 个最频繁的标记代表什么？
    d. 名词后面最常见的是哪些标记？这些标记代表什么？
16. ◑探索有关查找标注器的以下问题：
    a. 回退标注器被省略时，模型大小变化，标注器的性能会发生什么？
    b. 思考图 5-4 的曲线；为查找标注器推荐一个平衡内存和性能的好的规模。你能想出在什么情况下应该尽量减少内存使用，什么情况下性能最大化而不必考虑内存使用？
17. ◑查找标注器的性能上限是什么？假设其表的大小没有限制。（提示：写一个程序算出被分配了最有可能的标记的词的标识符的平均百分比。）
18. ◑生成已标注数据的一些统计数据，回答下列问题：
    a. 总是被分配相同词性的词类的比例是多少？
    b. 多少词是有歧义的，即：从某种意义上说，它们至少和两个标记一起出现？
    c. 布朗语料库中这些有歧义的词的标识符的百分比是多少？
19. ◑evaluate()方法算出一个文本上运行的标注器的精度。例如：如果提供的已标注文本是[('the', 'DT'), ('dog','NN')]，标注器产生的输出是[('the', 'NN'), ('dog', 'NN')]，那么得分为 0.5。让我们尝试找出评价方法是如何工作的：
    a. 一个标注器 t 将一个词汇链表作为输入，产生一个已标注词链表作为输出。t.evaluate()只以一个正确标注的文本作为唯一的参数。执行标注之前必须对输入做些什么？
    b. 一旦标注器创建了新标注的文本，evaluate()方法可能如何比较它与原来标注的文本，计算准确性得分？
    c. 现在，检查源代码来看看这个方法是如何实现的。检查 nltk.tag.api.__file__找到源代码的位置，使用编辑器打开这个文件（一定要使用文件 api.py，而不是编译过的二进制文件 api.pyc）。
20. ◑编写代码，搜索布朗语料库，根据标记查找特定的词和短语，回答下列问题：
    a. 产生一个标注为 MD 的不同的词的按字母顺序排序的列表。
    b. 识别可能是复数名词或第三人称单数动词的词（如 deals，flies）。
    c. 识别三个词的介词短语形式 IN + DET + NN（如 in the lab）。

d. 男性与女性代词的比例是多少？

21. ◑表 3-1 中我们看到动词 adore、love、like 和 prefer 及前面的限定符如 really 的频率计数的表格。探讨这四个动词前出现的所有限定符（布朗标记 QL）。
22. ◑定义可以用来做生词的回退标注器的 regexp_tagger。这个标注器只检查基数词。通过特定的前缀或后缀字符串进行测试，它应该能够猜测其他标记。例如：我们可以标注所有-s 结尾的词为复数名词。定义一个正则表达式标注器（使用 RegexpTagger()），测试至少 5 个单词拼写的其他模式。（使用内联文档解释规则。）
23. ◑考虑上一练习中开发的正则表达式标注器，使用它的 accuracy()方法评估标注器，尝试想办法提高其性能。讨论你找到的结果。客观的评估如何帮助开发过程？
24. ◑数据稀疏问题有多严重？调查 n-gram 标注器当 n 从 1 增加到 6 时的性能。为准确性得分制表。估计这些标注器需要的训练数据，假设词汇量大小为 10^5 而标记集的大小为 10^2。
25. ◑获取另一种语言的一些已标注数据，在其上测试和评估各种标注器。如果这种语言是形态复杂的，或者有词类的任何字形线索（如：capitalization），可以考虑为它开发一个正则表达式标注器（排在 unigram 标注器之后，默认标注器之前）。对比同样的运行在英文数据上的标注器，你的 tagger(s)的准确性如何？讨论你在运用这些方法到这种语言时遇到的问题。
26. ◑例 5-4 绘制曲线显示查找标注器的性能随模型的大小增加的变化。绘制当训练数据量变化时 unigram 标注器的性能曲线。
27. ◑检查 5.5 节中定义的 bigram 标注器 t2 的混淆矩阵，确定简化的一套或多套标记。定义字典做映射，在简化的数据上评估标注器。
28. ◑使用简化的标记集测试标注器（或制作一个你自己的，通过丢弃每个标记名中除第一个字母外所有的字母）。这种标注器需要做的区分更少，但由它获得的信息也更少。讨论你的结果。
29. ◑回顾一个 bigram 标注器训练过程中遇到生词，标注句子的其余部分为 None 的例子。一个 bigram 标注器只处理了句子的一部分就失败了，即使句子中没有包含生词（即使句子在训练过程中使用过），这可能吗？在什么情况下会出现这种情况呢？你可以写一个程序，找到一些这方面的例子吗？
30. ◑预处理布朗新闻数据，替换低频词为 UNK，但留下标记不变。在这些数据上训练和评估一个 bigram 标注器。这样有多少帮助？unigram 标注器和默认标注器的贡献是什么？
31. ◑修改例 5-4 中的程序，通过将 pylab.plot()替换为 pylab.semilogx()，在 X 轴上使用对数刻度。关于结果图形的形状，你注意到了什么？梯度告诉你什么呢？
32. ◑阅读 Brill 标注器演示函数的文档，使用 help(nltk.tag.brill.demo)。通过设置不同的参数值试验这个标注器。是否有任何训练时间（语料库大小）和性能之间的权衡？
33. ◑写代码构建一个集合的字典的字典。用它来存储一套可以跟在具有给定 POS 标记的给定词后面的 POS 标记，例如：$word_i \rightarrow tag_i \rightarrow tag_i+1$。
34. ●布朗语料库中有 264 个不同的词有 3 种可能的标签。
    a. 打印一个表格，一列中是整数 1..10，另一列是语料库中有 1..10 个不同标记的不同词的数目。
    b. 对有不同的标记数量最多的词，输出语料库中包含这个词的句子，每个可能的标记一个。
35. ●写一个程序，按照词 must 后面的词的标记为它的上下文分类。这样可以区分 must 的“必须”和“应该”两种词意上的用法吗？

36. ●创建一个正则表达式标注器和各种 unigram 以及 n-gram 标注器，包括回退，在布朗语料库上训练它们。
    a. 创建这些标注器的 3 种不同组合。测试每个组合标注器的准确性。哪种组合效果最好？
    b. 尝试改变训练语料的规模。它是如何影响你的结果的？
37. ●我们标注生词的方法一直要考虑这个词的字母（使用 RegexpTagger()），或完全忽略这个词，将它标注为一个名词（使用 nltk.DefaultTagger()）。这些方法对于有新词却不是名词的文本不会很好。思考句子：I like to blog on Kim's blog。如果 blog 是一个新词，那么查看前面的标记（TO 和 NP$）可能会有所帮助，即我们需要一个对前面的标记敏感的默认标注器。
    a. 创建一种新的 unigram 标注器，查看前一个词的标记，而忽略当前词。（做到这一点的最好办法是修改 UnigramTagger()的源代码，需要 Python 中的面向对象编程的知识。）
    b. 将这个标注器加入到回退标注器序列（包括普通的 trigram 和 bigram 标注器），放在常用默认标注器的前面。
    c. 评价这个新的 unigram 标注器的贡献。
38. ●思考 5-5 节中的代码，它确定一个 trigram 标注器的准确性上限。回顾 Abney 的关于精确标注的不可能性的讨论（Abney, 2006）。解释为什么正确标注这些例子需要获取词和标记以外的其他种类的信息。你如何估计这个问题的规模？
39. ●使用 nltk.probability 中的一些估计技术，例如 Lidstone 或 Laplace 估计，开发一种统计标注器，它在训练中没有遇到而测试中遇到的上下文中表现优于 n-gram 回退标注器。
40. ●检查 Brill 标注器创建的诊断文件 rules.out 和 errors.out。通过访问源代码（http://www.nltk.org/code）获得演示代码，创建你自己版本的 Brill 标注器。并根据你从检查 rules.out 了解到的，删除一些规则模板。增加一些新的规则模板，这些模板使用那些可能有助于纠正你在 errors.out 看到的错误的上下文。
41. ●开发一个 n-gram 回退标注器，允许在标注器初始化时指定“anti-n-grams”，如：["the","the"]。一个 anti-n-grams 被分配一个数字 0，被用来防止这个 n-gram 回退（如避免估计 P(the | the)而只做 P(the)）。
42. ●使用布朗语料库开发标注器时，调查三种不同的方式来定义训练和测试数据之间的分割：genre(category)、source(fileid)和句子。比较它们的相对性能，并讨论哪种方法最合理。（你可能要使用 n-交叉验证，在 6.3 节中讨论的，以提高评估的准确性。）
43. ●开发你自己的 NgramTagger，从 NLTK 中的类继承，封装本章中所述的已标注的训练和测试数据的词汇表缩减方法。确保 unigram 和默认回退标注器有机会获得全部词汇。

第六章 学习分类文本

模式识别是自然语言处理的一个核心部分。以-ed 结尾的词往往是过去时态动词（第 5 章）。频繁使用 will 是新闻文本的暗示（第 3 章）。这些可观察到的模式——词的结构和词频——恰好与特定方面的含义关联，如：时态和主题。但我们怎么知道从哪里开始寻找，形式的哪一方面关联含义的哪一方面？

本章的目标是要回答下列问题：

1. 我们怎样才能识别语言数据中能明显用于对其分类的特征？
2. 我们怎样才能构建语言模型，用于自动执行语言处理任务？
3. 从这些模型中我们可以学到哪些关于语言的知识？

一路上，我们将研究一些重要的机器学习技术，包括决策树、朴素贝叶斯分类器和最大熵分类。我们会掩盖这些技术的数学和统计的基础，集中关注如何以及何时使用它们（更多的技术背景知识见 6.9 节）。在看这些方法之前，我们首先需要知道这个主题的范围十分广泛。

6.1 有监督分类

分类是为给定的输入选择正确的**类标签**的任务。在基本的分类任务中，每个输入被认为是与所有其它输入隔离的，并且标签集是预先定义的。这里是分类任务的一些例子：

- 判断一封电子邮件是否是垃圾邮件。
- 从一个固定的主题领域列表中，如“体育”、“技术”和“政治”，决定新闻报道的主题是什么。
- 决定词 bank 给定的出现是用来指河的坡岸、一个金融机构、向一边倾斜的动作还是在金融机构里的存储行为。

基本的分类任务有许多有趣的变种。例如：在多类分类中，每个实例可以分配多个标签；在开放性分类中，标签集是事先没有定义的；在序列分类中，一个输入链表作为一个整体分类。

如果分类的建立基于包含每个输入的正确标签的训练语料，被称为**有监督分类**。有监督分类使用的框架图如图 6-1 所示：

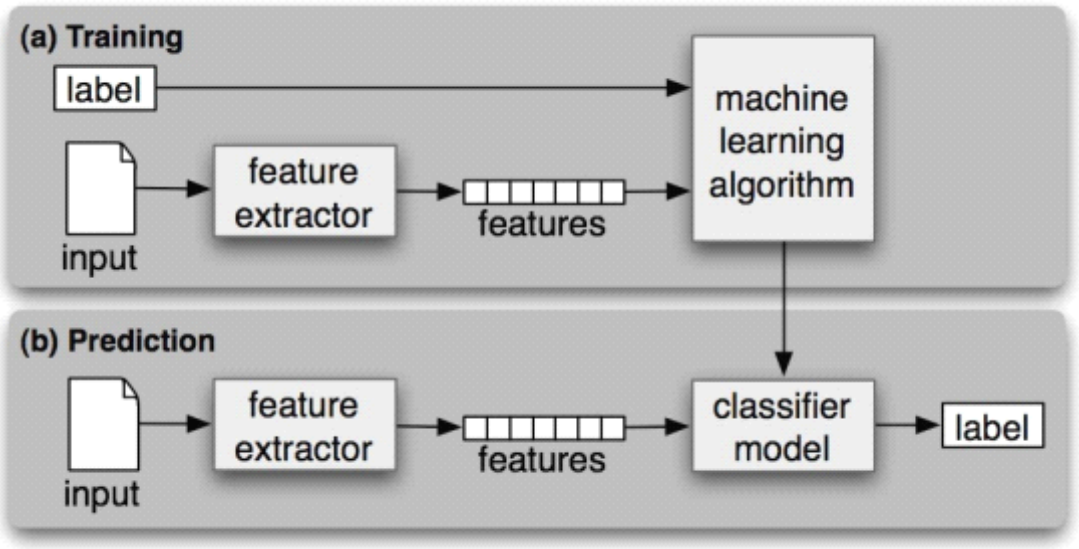


图 6-1. 有监督分类。（a）在训练过程中，特征提取器用来将每一个输入值转换为特征集。这些特征集捕捉每个输入中应被用于对其分类的基本信息，我们将在下一节中讨论它。特征集与标签的配对被送入机器学习算法，生成模型。（b）在预测过程中，相同的特征提取器被用来将未见过的输入转换为特征集。之后，这些特征集被送入模型产生预测标签。

在本节的其余部分，我们将着眼于分类器如何能够解决各种各样的任务。我们讨论的目的不是要范围全面，而是给出在文本分类器的帮助下执行的任务的一个代表性的例子。

性别鉴定

在 2.4 节中，我们看到，男性和女性的名字有一些鲜明的特点。以 a，e 和 i 结尾的很可能是女性，而以 k，o，r，s 结尾的很可能是男性。让我们建立一个分类器更精确地模拟这些差异。

创建一个分类器的第一步是决定输入的什么样的**特征**是相关的，以及如何为那些特征**编码**。在这个例子中，我们一开始只是寻找一个给定的名称的最后一个字母。以下**特征提取器**函数建立一个字典，包含有关给定名称的相关信息：

```
>>> def gender_features(word):
...     return {'last_letter': word[-1]}
>>> gender_features('Shrek')
{'last_letter': 'k'}
```

这个函数返回的字典被称为**特征集**，映射特征名称到它们的值。特征名称是区分大小写的字符串，通常提供一个简短的人可读的特征描述。特征值是简单类型的值，如布尔、数字和字符串。

大多数分类方法要求特征使用简单的类型进行编码，如布尔类型、数字和字符串。但要注意仅仅因为一个特征是简单类型，并不一定意味着该特征的值易于表达或计算；的确，它可以用非常复杂的和有信息量的值作为特征，如：第 2 个有监督分类器的输出。

现在，我们已经定义了一个特征提取器，我们需要准备一个例子和对应类标签的链表：

```
>>> from nltk.corpus import names
>>> import random
```

```
>>> names = ([(name, 'male') for name in names.words('male.txt')] +
...          [(name, 'female') for name in names.words('female.txt')])
>>> random.shuffle(names)
```

接下来，我们使用特征提取器处理名称数据，并划分特征集的结果链表为一个**训练集**和一个**测试集**。训练集用于训练一个新的“朴素贝叶斯”分类器。

```
>>> featuresets = [(gender_features(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

在本章的后面，我们将学习更多关于朴素贝叶斯分类器的内容。现在，让我们只是在上面测试一些没有出现在训练数据中的名字：

```
>>> classifier.classify(gender_features('Neo'))
'male'
>>> classifier.classify(gender_features('Trinity'))
'female'
```

请看《黑客帝国》中这些角色的名字被正确分类。尽管这部科幻电影的背景是在 2199 年，但它仍然符合我们有关名字和性别的预期。我们可以在大数据量的未见过的数据上系统地评估这个分类器：

```
>>> print nltk.classify.accuracy(classifier, test_set)
0.758
```

最后，我们可以检查分类器，确定哪些特征对于区分名字的性别是最有效的。

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
             last_letter = 'a'              female : male   =     38.3 : 1.0
             last_letter = 'k'                male : female =     31.4 : 1.0
             last_letter = 'f'                male : female =     15.3 : 1.0
             last_letter = 'p'                male : female =     10.6 : 1.0
             last_letter = 'w'                male : female =     10.6 : 1.0
```

此列表显示训练集中以 a 结尾的名字中女性是男性的 38 倍，而以 k 结尾名字中男性是女性的 31 倍。这些比率称为**似然比**，可以用于比较不同特征-结果关系。

轮到你来：修改 gender_features()函数，为分类器提供名称的长度、它的第一个字母以及任何其他看起来可能有用的特征。再用这些新特征训练分类器，并测试其准确性。

在处理大型语料库时，构建一个包含每一个实例的特征的单独的链表会使用大量的内存。在这些情况下，使用函数 nltk.classify.apply_features，返回一个行为像一个链表而不会在内存存储所有特征集的对象：

```
>>> from nltk.classify import apply_features
>>> train_set = apply_features(gender_features, names[500:])
>>> test_set = apply_features(gender_features, names[:500])
```

选择正确的特征

选择相关的特征，并决定如何为一个学习方法编码它们，这对学习方法提取一个好的模

型可以产生巨大的影响。建立一个分类器的很多有趣的工作之一是找出哪些特征可能是相关的，以及我们如何能够表示它们。虽然使用相当简单而明显的特征集往往可以得到像样的性能，但是使用精心构建的基于对当前任务的透彻理解的特征，通常会显著提高收益。

典型地，特征提取通过反复试验和错误的过程建立的，由哪些信息是与问题相关的直觉指引的。它通常以“厨房水槽”的方法开始，包括你能想到的所有特征，然后检查哪些特征是实际有用的。我们在例 6-1 中对名字性别特征采取这种做法。

例 6-1. 一个特征提取器，过拟合性别特征。这个特征提取器返回的特征集包括大量指定的特征，从而导致对于相对较小的名字语料库过拟合。

```
def gender_features2(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_features2('John')
{'count(j)': 1, 'has(d)': False, 'count(b)': 0, ...}
```

然而，你要用于一个给定的学习算法的特征的数目是有限的——如果你提供太多的特征，那么该算法将高度依赖你的训练数据的特，性而一般化到新的例子的效果不会很好。这个问题被称为**过拟合**，当运作在小训练集上时尤其会有问题。例如：如果我们使用例 6-1 中所示的特征提取器训练朴素贝叶斯分类器，将会过拟合这个相对较小的训练集，造成这个系统的精度比只考虑每个名字最后一个字母的分类器的精度低约 1%。

```
>>> featuresets = [(gender_features2(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.748
```

一旦初始特征集被选定，完善特征集的一个非常有成效的方法是**错误分析**。首先，我们选择一个**开发集**，包含用于创建模型的语料数据。然后将这种开发集分为**训练集**和**开发测试集**。

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
```

训练集用于训练模型，开发测试集用于进行错误分析，测试集用于系统的最终评估。由于下面讨论的原因，我们将一个单独的开发测试集用于错误分析而不是使用测试集是很重要的。在图 6-2 显示了将语料数据划分成不同的子集。

已经将语料分为适当的数据集，我们使用训练集训练一个模型①，然后在开发测试集上运行②。

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set) ①
>>> print nltk.classify.accuracy(classifier, devtest_set) ②
```

0.765

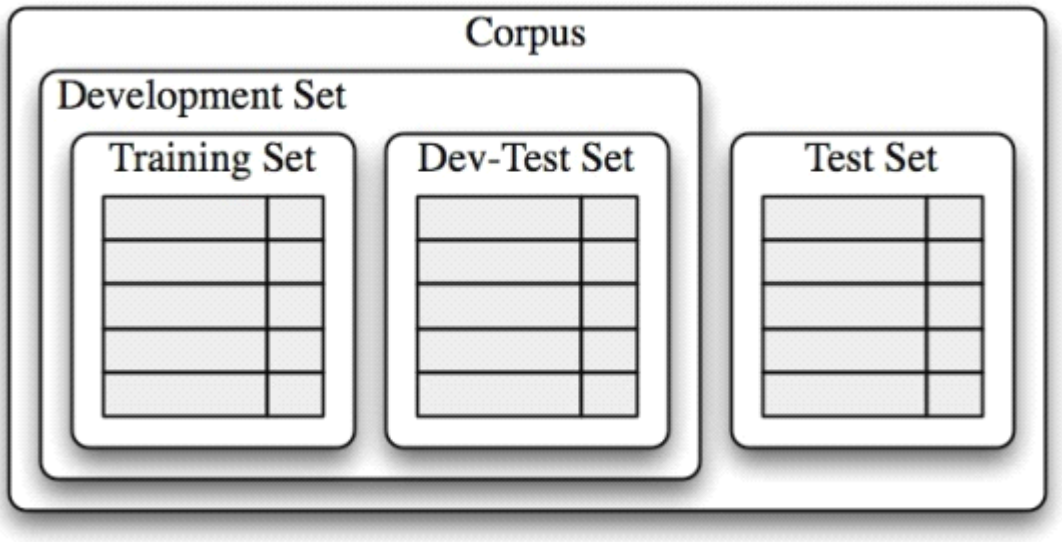


图 6-2. 用于训练有监督分类器的语料数据组织图。语料数据分为两类：开发集和测试集。开发集通常被进一步分为训练集和开发测试集。

使用开发测试集，我们可以生成一个分类器预测名字性别时的错误列表。

```
>>> errors = []
>>> for (name, tag) in devtest_names:
...     guess = classifier.classify(gender_features(name))
...     if guess != tag:
...         errors.append( (tag, guess, name) )
```

然后，可以检查个别错误案例，在那里该模型预测了错误的标签，尝试确定什么额外信息将使其能够作出正确的决定（或者现有的哪部分信息导致其做出错误的决定）。然后可以相应的调整特征集。我们已经建立的名字分类器在开发测试语料上产生约 100 个错误：

```
>>> for (tag, guess, name) in sorted(errors): # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
...     print 'correct=%-8s guess=%-8s name=%-30s' %
(tag, guess, name)
    ...
correct=female guess=male name=Cindelyn
    ...
correct=female guess=male name=Katheryn
correct=female guess=male name=Kathryn
    ...
correct=male guess=female name=Aldrich
    ...
correct=male guess=female name=Mitch
    ...
correct=male guess=female name=Rich
    ...
```

浏览这个错误列表，它明确指出一些多个字母的后缀可以指示名字性别。例如：yn 结尾的名字显示以女性为主，尽管事实上，n 结尾的名字往往是男性；以 ch 结尾的名字通常是男性，尽管以 h 结尾的名字倾向于是女性。因此，调整我们的特征提取器包括两个字母后

缀的特征：

```
>>> def gender_features(word):
...         return {'suffix1': word[-1:],
...                 'suffix2': word[-2:]}
```

使用新的特征提取器重建分类器，我们看到测试数据集上的性能提高了近 3 个百分点（从 76.5%到 78.2%）：

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, devtest_set)
0.782
```

这个错误分析过程可以不断重复，检查存在于由新改进的分类器产生的错误中的模式，每一次错误分析过程被重复，我们应该选择一个不同的开发测试/训练分割，以确保该分类器不会开始反映开发测试集的特质。

但是，一旦我们已经使用了开发测试集帮助我们开发模型，关于这个模型在新数据会表现多好，我们将不能再相信它会给我们一个准确地结果！因此，保持测试集分离、未使用过，直到我们的模型开发完毕是很重要的。在这一点上，我们可以使用测试集评估模型在新的输入值上执行的有多好。

文档分类

2.1 节中，我们看到了语料库的几个例子，那里文档已经按类别标记。使用这些语料库，我们可以建立分类器，自动给新文档添加适当的类别标签。首先，我们构造一个标记了相应类别的文档清单。对于这个例子，我们选择电影评论语料库，将每个评论归类为正面或负面。

```
>>> from nltk.corpus import movie_reviews
>>> documents = [(list(movie_reviews.words(fileid)), category)
...               for category in movie_reviews.categories()
...               for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(documents)
```

接下来，我们为文档定义一个特征提取器，这样分类器就会知道哪些方面的数据应注意（见例 6-2）。对于文档主题识别，我们可以为每个词定义一个特性表示该文档是否包含这个词。为了限制分类器需要处理的特征的数目，我们一开始构建一个整个语料库中前 2000 个最频繁词的链表①。然后，定义一个特征提取器②，简单地检查这些词是否在一个给定的文档中。

例 6-2. 一个文档分类的特征提取器，其特征表示每个词是否在一个给定的文档中。

```
all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = all_words.keys()[:2000]
def document_features(document):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains(%s)' % word] = (word in document_words)
    return features
>>> print document_features(movie_reviews.words('pos/cv957_8737.txt'))
```

{'contains(waste)': False, 'contains(lot)': False, ...}

我们计算③中文档的所有词的集合，而不仅仅检查 if word in document，因为检查一个词是否在一个集合中出现比检查它是否在一个链表中出现要快的多（见 4.7 节）。

现在，我们已经定义了我们的特征提取器，可以用它来训练一个分类器，为新的电影评论加标签（例 6-3）。为了检查产生的分类器可靠性如何，我们在测试集上计算其准确性①。再一次的，我们可以使用 show_most_informative_features()来找出哪些特征是分类器发现最有信息量的②。

例 6-3. 训练和测试一个分类器进行文档分类。

```
featuresets = [(document_features(d), c) for (d,c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set) ①
0.81
>>> classifier.show_most_informative_features(5) ②
Most Informative Features
    contains(outstanding) = True              pos : neg    =     11.1 : 1.0
         contains(seagal) = True              neg : pos    =      7.7 : 1.0
    contains(wonderfully) = True              pos : neg    =      6.8 : 1.0
          contains(damon) = True              pos : neg    =      5.9 : 1.0
          contains(wasted) = True             neg : pos    =      5.8 : 1.0
```

显然在这个语料库中，提到 Seagal 的评论中负面的是正面的大约 8 倍，而提到 Damon 的评论中正面的是负面的大约 6 倍。

词性标注

第 5 章中，我们建立了一个正则表达式标注器，通过查找词内部的组成，为词选择词性标记。然而，这个正则表达式标注器是手工制作的。作为替代，我们可以训练一个分类器来算出哪个后缀最有信息量。首先，让我们找出最常见的后缀：

```
>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
...         word = word.lower()
...         suffix_fdist.inc(word[-1:])
...         suffix_fdist.inc(word[-2:])
...         suffix_fdist.inc(word[-3:])
>>> common_suffixes = suffix_fdist.keys()[:100]
>>> print common_suffixes
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the',
'y', 'r', 'to', 'in', 'f', 'o', 'ed', 'nd', 'is', 'on', 'l',
'g', 'and', 'ng', 'er', 'as', 'ing', 'h', 'at', 'es', 'or',
're', 'it', '``', 'an', "''", 'm', ';', 'i', 'ly', 'ion', ...]
```

接下来，我们将定义一个特征提取器函数，检查给定的单词的这些后缀：

```
>>> def pos_features(word):
```

```
...     features = {}
...     for suffix in common_suffixes:
...         features['endswith(%s)' % suffix] = word.lower().endswith(suffix)
...     return features
```

特征提取函数的行为就像有色眼镜一样，强调我们的数据中的某些属性（颜色），并使其无法看到其他属性。分类器在决定如何标记输入时，将完全依赖它们强调的属性。在这种情况下，分类器将只基于一个给定的词拥有（如果有）哪个常见后缀的信息来做决定。

现在，我们已经定义了我们的特征提取器，可以用它来训练一个新的“决策树”的分类器（将在 6.4 节讨论）：

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in tagged_words]
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.DecisionTreeClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.62705121829935351
>>> classifier.classify(pos_features('cats'))
'NNS'
```

决策树模型的一个很好的性质是它们往往很容易解释。我们甚至可以指示 NLTK 将它们以伪代码形式输出：

```
>>> print classifier.pseudocode(depth=4)
if endswith(,) == True: return ','
if endswith(,) == False:
  if endswith(the) == True: return 'AT'
  if endswith(the) == False:
    if endswith(s) == True:
      if endswith(is) == True: return 'BEZ'
      if endswith(is) == False: return 'VBZ'
    if endswith(s) == False:
      if endswith(.) == True: return '.'
      if endswith(.) == False: return 'NN'
```

在这里，我们可以看到分类器一开始检查一个词是否以逗号结尾——如果是，它会得到一个特别的标记“,”。接下来，分类器检查词是否以“the”结尾，这种情况它几乎肯定是一个限定词。这个“后缀”被决策树早早使用是因为词 the 太常见。分类器继续检查词是否以 s 结尾，如果是，那么它极有可能得到动词标记 VBZ（除非它是这个词 is，它有特殊标记 BEZ），如果不是，那么它往往是名词（除非它是标点符号“.”）。实际的分类器包含这里显示的 if-then 语句下面进一步的嵌套，参数 depth=4 只显示决定树的顶端部分。

探索上下文语境

通过增加特征提取函数，我们可以修改这个词性标注器来利用各种词内部的其他特征，例如：词长、它所包含的音节数或者它的前缀。然而，只要特征提取器仅仅看着目标词，我们就没法添加依赖词出现的上下文语境特征。然而上下文语境特征往往提供关于正确标记的强大线索——例如：标注词 fly，如果知道它前面的词是“a”将使我们能够确定它是一个名

词，而不是一个动词。

为了采取基于词的上下文的特征，我们必须修改以前为我们的特征提取器定义的模式。不是只传递已标注的词，我们将传递整个（未标注的）句子，以及目标词的索引。例 6-4 演示了这种方法，使用依赖上下文的特征提取器定义一个词性标记分类器。

例 6-4. 一个词性分类器，它的特征检测器检查一个词出现的上下文以便决定应该分配的词性标记。特别的，前面的词被作为一个特征。

```
def pos_features(sentence, i):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
    return features
>>> pos_features(brown.sents()[0], 8)
{'suffix(3)': 'ion', 'prev-word': 'an', 'suffix(2)': 'on', 'suffix(1)': 'n'}
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> featuresets = []
>>> for tagged_sent in tagged_sents:
...     untagged_sent = nltk.tag.untag(tagged_sent)
...     for i, (word, tag) in enumerate(tagged_sent):
...         featuresets.append(
(pos_features(untagged_sent, i), tag) )
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
```

0.78915962207856782 很显然，利用上下文特征提高了我们的词性标注器的性能。例如：分类器学到一个词如果紧跟在词 large 或 gubernatorial 后面，极可能是名词。然而，它无法学到：一个词如果它跟在形容词后面可能是名词，这样更一般的，因为它没有获得前面这个词的词性标记。在一般情况下，简单的分类器总是将每一个输入与所有其他输入独立对待。在许多情况下，这非常有意义。例如：关于名字是否倾向于男性或女性的决定可以通过一个事件一个事件具体分析来做出。然而，有很多情况，如词性标注，我们感兴趣的是解决彼此密切相关的分类问题。

序列分类

为了捕捉相关的分类任务之间的依赖关系，我们可以使用**联合分类器**模型，收集有关输入，选择适当的标签。在词性标注的例子中，各种不同的**序列分类器**模型可以被用来为一个给定的句子中的所有的词共同选择词性标签。

一种序列分类器策略，称为**连续分类**或**贪婪序列分类**，是为第一个输入找到最有可能的类标签，然后使用这个问题的答案帮助找到下一个输入的最佳的标签。这个过程可以不断重复直到所有的输入都被贴上标签。这是被 5.5 节的 bigram 标注器采用的方法，它一开始为

句子的第一个词选择词性标记，然后为每个随后的词选择标记，基于词本身和前面词的预测的标记。

在例 6-5 演示了这一策略。首先，我们必须扩展我们的特征提取函数使其具有参数 history，它提供一个我们到目前为止已经为句子预测的标记的链表①。history 中的每个标记对应句子中的一个词。但是请注意，history 将只包含我们已经归类的词的标记，也就是目标词左侧的词。因此，虽然是有可能查看目标词右边的词的某些特征，但查看那些词的标记是不可能的（因为我们还未产生它们）。

已经定义了特征提取器，我们可以继续建立我们的序列分类器②。在训练中，我们使用已标注的标记为征提取器提供适当的历史信息，但标注新的句子时，我们基于标注器本身的输出产生历史信息。

例 6-5. 使用连续分类器进行词性标注。

```
def pos_features(sentence, i, history): ①
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
        features["prev-tag"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
        features["prev-tag"] = history[i-1]
    return features
class ConsecutivePosTagger(nltk.TaggerI): ②
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = pos_features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.NaiveBayesClassifier.train(train_set)
    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = pos_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> size = int(len(tagged_sents) * 0.1)
>>> train_sents, test_sents = tagged_sents[size:], tagged_sents[:size]
>>> tagger = ConsecutivePosTagger(train_sents)
```

```
>>> print tagger.evaluate(test_sents)
0.79796012981
```

其他序列分类方法

这种方法的一个缺点是我们的决定一旦做出无法更改。例如：如果我们决定将一个词标注为名词，但后来发现的证据表明应该是一个动词，就没有办法回去修复我们的错误。这个问题的一个解决方案是采取转型策略。转型联合分类的工作原理是为输入的标签创建一个初始值，然后反复提炼那个值，尝试修复相关输入之间的不一致。Brill 标注器，5.6 节描述的，是这种策略的一个很好的例子。

另一种方案是为词性标记所有可能的序列打分，选择总得分最高的序列。隐马尔可夫模型就采取这种方法。**隐马尔可夫模型**类似于连续分类器，它不光看输入也看已预测标记的历史。然而，不是简单地找出一个给定的词的单个最好的标签，而是为标记产生一个概率分布。然后将这些概率结合起来计算标记序列的概率得分，最高概率的标记序列会被选中。不幸的是，可能的标签序列的数量相当大。给定 30 个标签的标记集，有大约 600 万（30^{10}）种方式来标记一个 10 个词的句子。为了避免单独考虑所有这些可能的序列，隐马尔可夫模型要求特征提取器只看最近的标记（或最近的 n 个标记，其中 n 是相当小的）。由于这种限制，它可以使用动态规划（4.7 节），有效地找出最有可能的标记序列。特别是，对每个连续的词索引 i，每个可能的当前及以前的标记都被计算得分。这种同样基础的方法被两个更先进的模型所采用，它们被称为**最大熵马尔可夫模型**和**线性链条件随机场模型**；但为标记序列打分用的是不同的算法。

6.2 有监督分类的更多例子

句子分割

句子分割可以看作是一个标点符号的分类任务：每当我们遇到一个可能会结束一个句子的符号，如句号或问号，我们必须决定它是否终止了当前句子。

第一步是获得一些已被分割成句子的数据，将它转换成一种适合提取特征的形式：

```
>>> sents = nltk.corpus.treebank_raw.sents()
>>> tokens = []
>>> boundaries = set()
>>> offset = 0
>>> for sent in nltk.corpus.treebank_raw.sents():
...     tokens.extend(sent)
...     offset += len(sent)
...     boundaries.add(offset-1)
```

在这里，`tokens` 是单独句子标识符的合并链表，`boundaries` 是一个包含所有句子边界标识符索引的集合。下一步，我们需要指定用于决定标点是否表示句子边界的数据特征：

```
>>> def punct_features(tokens, i):
...     return {'next-word-capitalized': tokens[i+1][0].isupper(),
...             'prevword': tokens[i-1].lower(),
```

```
...                 'punct': tokens[i],
...                 'prev-word-is-one-char': len(tokens[i-1]) == 1}
```

基于这一特征提取器，我们可以通过选择所有的标点符号创建一个加标签的特征集的链表，然后标注它们是否是边界标识符：

```
>>> featuresets = [(punct_features(tokens, i), (i in boundaries))
...                for i in range(1, len(tokens)-1)
...                if tokens[i] in '.?!']
```

使用这些特征集，我们可以训练和评估一个标点符号分类器：

```
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0.97419354838709682
```

使用这种分类器进行断句，我们只需检查每个标点符号，看它是否是作为一个边界标识符，在边界标识符处分割词链表。在例 6.8 中的清单显示了如何可以做到这一点。

例 6-6. 基于分类的断句器

```
def segment_sentences(words):
    start = 0
    sents = []
    for i, word in words:
        if word in '.?!' and classifier.classify(words, i) == True:
            sents.append(words[start:i+1])
            start = i+1
    if start < len(words):
        sents.append(words[start:])
```

识别对话行为类型

处理对话时，将对话看作说话者执行的动作是很有用的。对于表述行为的陈述句这种解释是最简单的。例如：I forgive you or I bet you can't climb that hill。但是问候、问题、回答、断言和说明都可以被认为是基于语言的行动类型。识别对话中言语下的**对话行为**是理解谈话的重要的第一步。

NPS 聊天语料库，在 2.1 节中的展示过，包括超过 10,000 个来自即时消息会话的帖子。这些帖子都已经被贴上 15 种对话行为类型中的一种标签，例如：“陈述”，“情感”，“yn 问题”，“Continuer”。因此，我们可以利用这些数据建立一个分类器，识别新的即时消息帖子的对话行为类型。第一步是提取基本的消息数据。我们将调用 xml_posts()来得到一个数据结构，表示每个帖子的 XML 注释：

```
>>> posts = nltk.corpus.nps_chat.xml_posts()[:10000]
```

下一步，我们将定义一个简单的特征提取器，检查帖子包含什么词：

```
>>> def dialogue_act_features(post):
...     features = {}
...     for word in nltk.word_tokenize(post):
...         features['contains(%s)' % word.lower()] = True
...     return features
```

最后，我们通过为每个帖子提取特征（使用 post.get('class') 获得一个帖子的对话行为类型）构造训练和测试数据，并创建一个新的分类器：

```
>>> featuresets = [(dialogue_act_features(post.text), post.get('class'))
...                for post in posts]
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.66
```

识别文字蕴含

识别文字蕴含（Recognizing textual entailment(RTE)）是判断文本 T 的一个给定片段是否蕴含着另一个叫做“假设”的文本（已经在 1.5 节讨论过）。迄今为止，已经有 4 个 RTE 挑战赛，在那里共享的开发和测试数据会提供给参赛队伍。这里是挑战赛 3 开发数据集中的文本/假设对的两个例子。标签 True 表示蕴含成立，False 表示蕴含不成立。

Challenge 3, Pair 34 (True)

T: Parviz Davudi was representing Iran at a meeting of the Shanghai Co-operation Organisation (SCO), the fledgling association that binds Russia, China and four former Soviet republics of central Asia together to fight terrorism.

H: China is a member of SCO.

Challenge 3, Pair 81 (False)

T: According to NC Articles of Organization, the members of LLC company are

H. Nelson Beavers, III, H. Chester Beavers and Jennie Beavers Stewart.

H: Jennie Beavers Stewart is a share-holder of Carolina Analytical Laboratory.

应当强调，文字和假设之间的关系并不一定是逻辑蕴涵，而是一个人是否会得出结论：文本提供了合理的证据证明假设是真实的。

我们可以把 RTE 当作一个分类任务，尝试为每一对预测真/假标签。虽然这项任务的成功做法似乎看上去涉及语法分析、语义和现实世界的知识的组合，RTE 的许多早期的尝试使用粗浅的分析基于文字和假设之间的在词级别的相似性取得了相当不错的结果。在理想情况下，我们希望如果有一个蕴涵那么假设所表示的所有信息也应该在文本中表示。相反，如果假设中有的资料文本中没有，那么就没有蕴涵。

在我们的 RTE 特征探测器（例 6-7）中，我们让词（即词类型）作为信息的代理，我们的特征计数词重叠的程度和假设中有而文本中没有的词的程度（由 hyp_extra()方法获取）。不是所有的词都是同样重要的——命名实体，如人、组织和地方的名称，可能会更为重要，这促使我们分别为 words 和 nes（命名实体）提取不同的信息。此外，一些高频虚词作为“停用词”被过滤掉。

图 6-7. “认识文字蕴涵”的特征提取器。RTEFeatureExtractor 类建立了一个除去一些停用词后在文本和假设中都有的词汇包，然后计算重叠和差异。

```
def rte_features(rtepair):
    extractor = nltk.RTEFeatureExtractor(rtepair)
    features = {}
    features['word_overlap'] = len(extractor.overlap('word'))
    features['word_hyp_extra'] = len(extractor.hyp_extra('word'))
```

```
        features['ne_overlap'] = len(extractor.overlap('ne'))
        features['ne_hyp_extra'] = len(extractor.hyp_extra('ne'))
        return features
```

为了说明这些特征的内容，我们检查前面显示的文本/假设对 34 的一些属性：

```
>>> rtepair = nltk.corpus.rte.pairs(['rte3_dev.xml'])[33]
>>> extractor = nltk.RTEFeatureExtractor(rtepair)
>>> print extractor.text_words
set(['Russia', 'Organisation', 'Shanghai', 'Asia', 'four', 'at',
'operation', 'SCO', ...])
>>> print extractor.hyp_words
set(['member', 'SCO', 'China'])
>>> print extractor.overlap('word')
set([])
>>> print extractor.overlap('ne')
set(['SCO', 'China'])
>>> print extractor.hyp_extra('word')
set(['member'])
```

这些特征表明假设中所有重要的词都包含在文本中，因此有一些证据支持标记这个为 True。

nltk.classify.rte_classify 模块使用这些方法在合并的 RTE 测试数据上取得了刚刚超过 58%的准确率。这个数字并不是很令人印象深刻的，还需要大量的努力，更多的语言学处理，才能达到更好的结果。

扩展到大型数据集

Python 提供了一个良好的环境进行基本的文本处理和特征提取。然而，它处理机器学习方法需要的密集数值计算不能够如 C 语言那样的低级语言那么快。因此，如果你尝试在大型数据集使用纯 Python 的机器学习实现（如 nltk.NaiveBayesClassifier），你可能会发现学习算法会花费大量的时间和内存。

如果你打算用大量训练数据或大量特征来训练分类器，我们建议你探索 NLTK 与外部机器学习包的接口。只要这些软件包已安装，NLTK 可以透明地调用它们（通过系统调用来训练分类模型，明显比纯 Python 的分类实现快。请看 NLTK 网站上推荐的 NLTK 支持的机器学习包列表。

6.3 评估

为了决定一个分类模型是否准确地捕捉了模式，我们必须评估该模型。评估的结果对于决定模型是多么值得信赖以及我们如何使用它是非常重要。评估也可以是一个有效的工具，用于指导我们在未来改进模型。

测试集

大多数评估技术为模型计算一个得分，通过比较它在**测试集**（或**评估集**）中为输入生成的标签与那些输入的正确标签。该测试集通常与训练集具有相同的格式。然而，测试集与训练语料不同是非常重要的：如果我们简单地重复使用训练集作为测试集，那么一个只记住了它的输入而没有学会如何推广到新的例子的模型会得到误导人的高分。

建立测试集时，往往是一个可用于测试的和可用于训练的数据量之间的权衡。对于有少量平衡的标签和一个多样化的测试集的分类任务，只要 100 个评估实例就可以进行有意义的评估。但是，如果一个分类任务有大量的标签或包括非常罕见的标签，那么选择的测试集的大小就要保证出现次数最少的标签至少出现 50 次。此外，如果测试集包含许多密切相关的实例——例如：来自一个单独文档中的实例——那么测试集的大小应增加，以确保这种多样性的缺乏不会扭曲评估结果。当有大量已标注数据可用时，只使用整体数据的 10%进行评估常常会在安全方面犯错。

选择测试集时另一个需要考虑的是测试集中实例与开发集中的实例的相似程度。这两个数据集越相似，我们对将评估结果推广到其他数据集的信心就越小。例如：考虑词性标注任务。在一种极端情况，我们可以通过从一个反映单一的文体（如新闻）的数据源随机分配句子，创建训练集和测试集：

```
>>> import random
>>> from nltk.corpus import brown
>>> tagged_sents = list(brown.tagged_sents(categories='news'))
>>> random.shuffle(tagged_sents)
>>> size = int(len(tagged_sents) * 0.1)
>>> train_set, test_set = tagged_sents[size:], tagged_sents[:size]
```

在这种情况下，我们的测试集和训练集将是非常相似的。训练集和测试集均取自同一文体，所以我们不能相信评估结果可以推广到其他文体。更糟糕的是，因为调用 `random.shuffle()`，测试集中包含来自训练使用过的相同的文档的句子。如果文档中有相容的模式（也就是说，如果一个给定的词与特定词性标记一起出现特别频繁），那么这种差异将体现在开发集和测试集。一个稍好的做法是确保训练集和测试集来自不同的文件：

```
>>> file_ids = brown.fileids(categories='news')
>>> size = int(len(file_ids) * 0.1)
>>> train_set = brown.tagged_sents(file_ids[size:])
>>> test_set = brown.tagged_sents(file_ids[:size])
```

如果我们要执行更令人信服的评估，可以从与训练集中文档联系更少的文档中获取测试集。

```
>>> train_set = brown.tagged_sents(categories='news')
>>> test_set = brown.tagged_sents(categories='fiction')
```

如果我们在此测试集上建立了一个性能很好的分类器，那么我们完全可以相信它有能力很好的泛化到用于训练它的数据以外的。

准确度

用于评估一个分类最简单的度量是**准确度**，测量测试集上分类器正确标注的输入的比例。例如：一个名字性别分类器，在包含 80 个名字的测试集上预测正确的名字有 60 个，它

有 60/80= 75%的准确度。nltk.classify.accuracy()函数会在给定的测试集上计算分类器模型的准确度：

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

解释一个分类器的准确性得分，考虑测试集中单个类标签的频率是很重要的。例如：考虑一个决定词 bank 每次出现的正确的词意的分类器。如果我们在金融新闻文本上评估分类器，那么我们可能会发现，金融机构的意思 20 个里面出现了 19 次。在这种情况下，95%的准确度也难以给人留下深刻印象，因为我们可以实现一个模型，它总是返回金融机构的意义。然而，如果我们在一个更加平衡的语料库上评估分类器，那里的最频繁的词意只占 40%，那么 95%的准确度得分将是一个更加积极的结果。（在 11.2 节测量标注一致性程度时也会有类似的问题。）

精确度和召回率

另一个准确度分数可能会产生误导的实例是在“搜索”任务中，如：信息检索，我们试图找出与特定任务有关的文档。由于不相关的文档的数量远远多于相关文档的数量，一个将每一个文档都标记为无关的模型的准确度分数将非常接近 100%。

因此，对搜索任务使用不同的测量集是很常见的，基于图 6-3 所示的四个类别的每一个中的项目的数量：

- **真阳性**是相关项目中我们正确识别为相关的。
- **真阴性**是不相关项目中我们正确识别为不相关的。
- **假阳性**（或 **I 型错误**）是不相关项目中我们错误识别为相关的。
- **假阴性**（或 **II 型错误**）是相关项目中我们错误识别为不相关的。

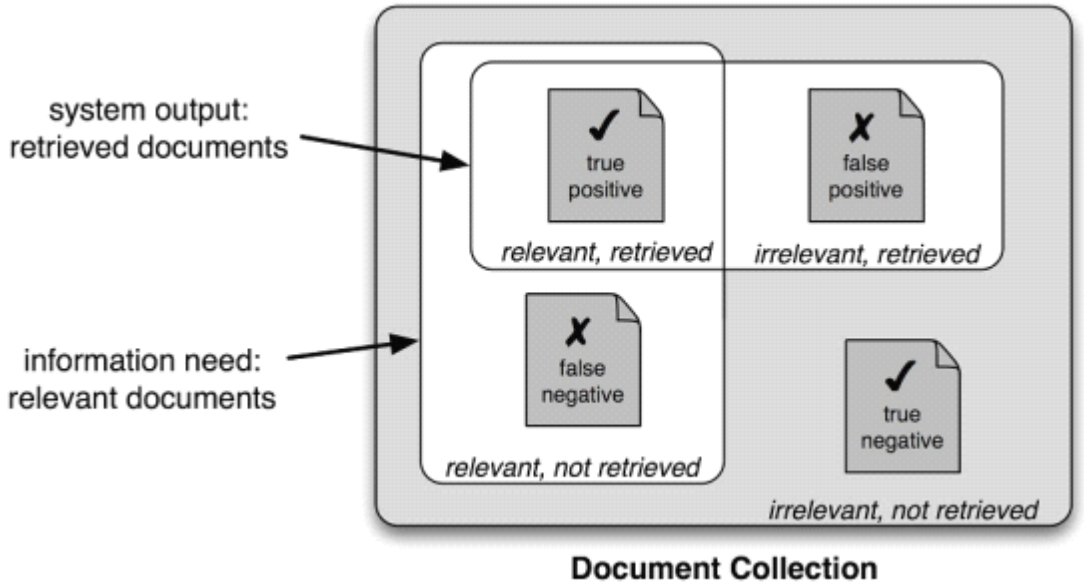


图 6-3. 真与假的阳性和阴性。

给定这四个数字，我们可以定义以下指标：

- **精确度**（Precision），表示我们发现的项目中有多少是相关的，TP/(TP+ FP)。
- **召回率**（Recall），表示相关的项目中我们发现了多少，TP/(TP+ FN)。
- **F-度量值**（F-Measure）（或 F-得分，F-Score），组合精确度和召回率为一个单独的

得分，被定义为精确度和召回率的调和平均数(2 × Precision × Recall)/(Precision +Recall)。

混淆矩阵

当处理有 3 个或更多的标签的分类任务时，基于模型错误类型细分模型的错误是有信息量的。一个**混淆矩阵**是一个表，其中每个 cells[i,j]表示正确的标签 i 被预测为标签 j 的次数。因此，对角线项目（即 cells[i,i]）表示正确预测的标签，非对角线项目表示错误。在下面的例子中，我们为 5.4 节中开发的 unigram 标注器生成一个混淆矩阵：

```
>>> 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(nltk.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')))
>>> cm = nltk.ConfusionMatrix(gold, test)
    |                                                 N                       |
    |       N       I       A       J               N               V       N |
    |       N       N       T       J       .       S       ,       B       P |
------+-----------------------------------------------------------------------------+
 NN | <11.8%>  0.0%     .    0.2%     .    0.0%     .    0.3%    0.0% |
 IN |   0.0% <9.0%>     .       .     .    0.0%     .       .       . |
 AT |      .      . <8.6%>     .     .       .     .       .       . |
 JJ |   1.6%      .      .  <4.0%>     .       .     .    0.0%    0.0% |
  . |      .      .      .      . <4.8%>     .     .       .       . |
 NS |   1.5%      .      .      .      . <3.2%>     .       .    0.0% |
  , |      .      .      .      .      .      . <4.4%>     .       . |
  B |   0.9%      .      .   0.0%     .       .     . <2.4%>     .  |
 NP |   1.0%      .      .   0.0%     .       .     .       . <1.9%> |
-------+-----------------------------------------------------------------------------+
(row = reference; col = test)
```

这个混淆矩阵显示常见的错误，包括 NN 替换为了 JJ（1.6%的词），NN 替换为了 NNS（1.5%的词）。注意点（.）表示值为 0 的单元格，对角线项目——对应正确的分类——用尖括号标记。

交叉验证

为了评估我们的模型，我们必须为测试集保留一部分已标注的数据。正如我们已经提到，如果测试集是太小了，我们的评价可能不准确。然而，测试集设置较大通常意味着训练集设置较小，如果已标注数据的数量有限，这样设置对性能会产生重大影响。

这个问题的解决方案之一是在不同的测试集上执行多个评估，然后组合这些评估的得分，这种技术被称为**交叉验证**。特别是，我们将原始语料细分为 N 个子集称为**折叠**（folds）。对于每一个这些的折叠，我们使用除这个折叠中的数据外其他所有数据训练模型，然后在这

个折叠上测试模型。即使个别的折叠可能是太小了而不能在其上给出准确的评价分数，综合评估得分是基于大量的数据，因此是相当可靠的。

第二，同样重要的，采用交叉验证的优势是，它可以让我们研究不同的训练集上性能变化有多大。如果我们从所有 N 个训练集得到非常相似的分数，然后我们可以相当有信心，得分是准确的。另一方面，如果 N 个训练集上分数很大不同，那么，我们应该对评估得分的准确性持怀疑态度。

6.4 决策树

接下来的三节中，我们将仔细看看可用于自动生成分类模型的三种机器学习方法：决策树、朴素贝叶斯分类器和最大熵分类器。正如我们所看到的，可以把这些学习方法看作黑盒子，直接训练模式，使用它们进行预测而不需要理解它们是如何工作的。但是，仔细看看这些学习方法如何基于一个训练集上的数据选择模型，会学到很多。了解这些方法可以帮助指导我们选择相应的特征，尤其是我们关于那些特征如何编码的决定。理解生成的模型可以让我们更好的提取信息，哪些特征对有信息量，那些特征之间如何相互关联。

决策树是一个简单的为输入值选择标签的流程图。这个流程图由检查特征值的**决策节点**和分配标签的**叶节点**组成。为输入值选择标签，我们以流程图的初始决策节点（称为其**根节点**）开始。此节点包含一个条件，检查输入值的特征之一，基于该特征的值选择一个分支。沿着这个描述我们输入值的分支，我们到达了一个新的决策节点，有一个关于输入值的特征的新的条件。我们继续沿着每个节点的条件选择的分支，直到到达叶节点，它为输入值提供了一个标签。图 6-4 显示名字性别任务的决策树模型的例子。

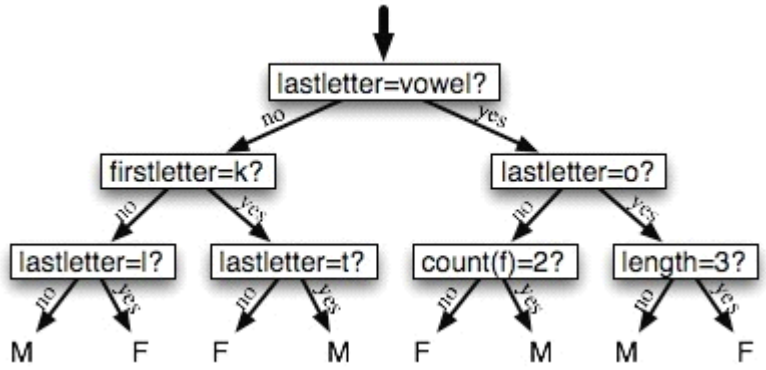


图 6-4. 名字性别任务的决策树模型。请注意树图按照习惯画出“颠倒的”，根在上面，叶子在下面。

一旦我们有了一个决策树，就可以直接用它来分配标签给新的输入值。不那么直接的是我们如何能够建立一个模拟给定的训练集的决策树。在此之前，我们看一下建立决策树的学习算法，思考一个简单的任务：为语料库选择最好的“决策树桩”。**决策树桩**是只有一个节点的决策树，基于一个特征决定如何为输入分类。每个可能的特征值一个叶子，为特征有那个值的输入指定类标签。要建立决策树桩，我们首先必须决定哪些特征应该使用。最简单的方法是为每个可能的特征建立一个决策树桩，看哪一个在训练数据上得到最高的准确度，也有其他的替代方案，我们将在下面讨论。一旦我们选择了一个特征，就可以通过分配一个标签给每个叶子，基于在训练集中所选的例子的最频繁的标签，建立决策树桩（即选择特征具有那个值的例子）。

给出了选择决策树桩的算法，生长出较大的决策树的算法就很简单了。首先，我们选择分类任务的整体最佳的决策树桩。然后，我们在训练集上检查每个叶子的准确度。没有达到足够的准确度的叶片被新的决策树桩替换，新决策树桩是在根据到叶子的路径选择的训练语

料的子集上训练的。例如：我们可以使图 6-4 中的决策树生长，通过替换最左边的叶子为新的决策树桩，这个新的决策树桩是在名字不以 k 开始或以一个元音或 l 结尾的训练集的子集上训练的。

熵和信息增益

正如之前提到的，有几种方法来为决策树桩确定最有信息量的特征。一种流行的替代方法，被称为**信息增益**，当我们用给定的特征分割输入值时，衡量它们变得更有序的程度。要衡量原始输入值集合如何无序，我们计算它们的标签的熵，如果输入值的标签非常不同，熵就高；如果输入值的标签都相同，熵就低。特别是，熵被定义为每个标签的概率乘以那个标签的 log 概率的总和。

(1) $H = \Sigma_{l \in labels} P(l) \times \log_2 P(l)$.

例如：图 6-5 显示了在名字性别预测任务中标签的熵如何取决于男性名字对女性名字的比例。请注意，如果大多数输入值具有相同的标签（例如，如果 P(male)接近 0 或接近 1），那么熵很低。特别的，低频率的标签不会贡献多少给熵（因为 P(l)很小），高频率的标签对熵也没有多大帮助（因为 log2P(l)很小）。另一方面，如果输入值的标签变化很多，那么有很多“中等”频率的标签，它们的 P(l)和 log2P(l)都不小，所以熵很高。例 6-8 演示了如何计算标签链表的熵。

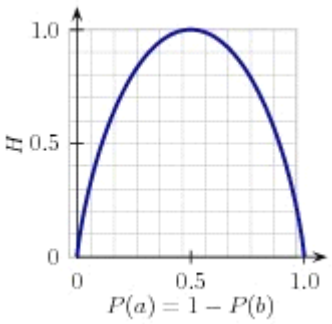


图 6-5. 名字性别预测任务中标签的熵，作为给定名字集合中是男性的百分比的函数。

例 6-8. 计算标签链表的熵。

```
import math
def entropy(labels):
    freqdist = nltk.FreqDist(labels)
    probs = [freqdist.freq(l) for l in nltk.FreqDist(labels)]
    return -sum([p * math.log(p,2) for p in probs])
>>> print entropy(['male', 'male', 'male', 'male'])
0.0
>>> print entropy(['male', 'female', 'male', 'male'])
0.811278124459

>>> print entropy(['female', 'male', 'female', 'male'])
1.0
>>> print entropy(['female', 'female', 'male', 'female'])
0.811278124459
>>> print entropy(['female', 'female', 'female', 'female'])
```

0.0

一旦我们已经计算了原始输入值的标签集的墒，就可以判断应用了决策树桩之后标签会变得多么有序。为了这样做，我们计算每个决策树桩的叶子的熵，利用这些叶子熵值的平均值（加权每片叶子的样本数量）。信息增益等于原来的熵减去这个新的减少的熵。信息增益越高，将输入值分为相关组的决策树桩就越好，于是我们可以通过选择具有最高信息增益的决策树桩来建立决策树。

决策树的另一个考虑因素是效率。前面描述的选择决策树桩的简单算法必须为每一个可能的特征构建候选决策树桩，并且这个过程必须在构造决策树的每个节点上不断重复。已经开发了一些算法通过存储和重用先前评估的例子的信息减少训练时间。

决策树有一些有用的性质。首先，它们简单明了，容易理解。决策树顶部附近尤其如此，这通常使学习算法可以找到非常有用的特征。决策树特别适合有很多层次的分类区别的情况。例如：决策树可以非常有效地捕捉进化树。

然而，决策树也有一些缺点。一个问题是，由于决策树的每个分支会划分训练数据，在训练树的低节点，可用的训练数据量可能会变得非常小。因此，这些较低的决策节点可能**过拟合**训练集，学习模式反映训练集的特质而不是问题底层显著的语言学模式。对这个问题的一个解决方案是当训练数据量变得太小时停止分裂节点。另一种方案是长出一个完整的决策树，但随后进行**剪枝剪去**在开发测试集上不能提高性能的决策节点。

决策树的第二个问题是，它们强迫特征按照一个特定的顺序进行检查，即使特征可能是相对独立的。例如：按主题分类文档（如体育、汽车或谋杀之谜）时，特征如 hasword(football)，极可能表示一个特定标签，无论其他的特征值是什么。由于决定树顶部附近的空间有限，大部分这些特征将需要在树中的许多不同的分支中重复。因为越往树的下方，分支的数量成指数倍增长，重复量可能非常大。

一个相关的问题是决策树不善于利用对正确的标签具有较弱预测能力的特征。由于这些特征的影响相对较小，它们往往出现在决策树非常低的地方。决策树学习的时间远远不够用到这些特征，也不能留下足够的训练数据来可靠地确定它们应该有什么样的影响。如果我们能够在整个训练集中看看这些特征的影响，那么我们也许能够做出一些关于它们是如何影响标签的选择的结论，。

决策树需要按一个特定的顺序检查特征的事实，限制了它们的利用相对独立的特征的能力。我们下面将讨论的朴素贝叶斯分类方法克服了这一限制，允许所有特征“并行”的起作用。

6.5 朴素贝叶斯分类器

在**朴素贝叶斯**分类器中，每个特征都得到发言权，来确定哪个标签应该被分配到一个给定的输入值。为一个输入值选择标签，朴素贝叶斯分类器以计算每个标签的**先验概率**开始，它由在训练集上检查每个标签的频率来确定。之后，每个特征的贡献与它的先验概率组合，得到每个标签的似然估计。似然估计最高的标签会分配给输入值。图 6-6 说明了这一过程。

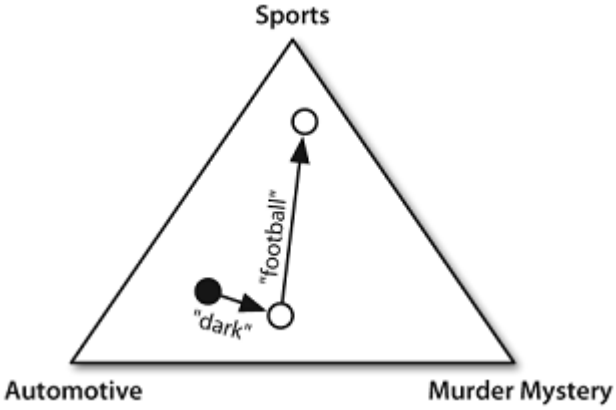


图 6-6. 使用朴素贝叶斯分类器为文档选择主题的程序的抽象图解。在训练语料中，大多数文档是有关汽车的，所以分类器从接近“汽车”的标签的点上开始。但它会考虑每个特征的影响。在这个例子中，输入文档中包含的词 dark，它是谋杀之谜的一个不太强的指标，也包含词 football，它是体育文档的一个有力指标。每个特征都作出了贡献之后，分类器检查哪个标签最接近，并将该标签分配给输入。

个别特征对整体决策作出自己的贡献，通过“投票反对”那些不经常出现的特征的标签。特别是，每个标签的似然得分由于与输入值具有此特征的标签的概率相乘而减小。例如：如果词 run 在 12%的体育文档中出现，在 10%的谋杀之谜的文档中出现，在 2%的汽车文档中出现，那么体育标签的似然得分将被乘以 0.12，谋杀之谜标签将被乘以 0.1，汽车标签将被乘以 0.02。整体效果是：略高于体育标签的得分的谋杀之谜标签的得分会减少，而汽车标签相对于其他两个标签会显著减少。这个过程如图 6-7 和 6-8 所示。

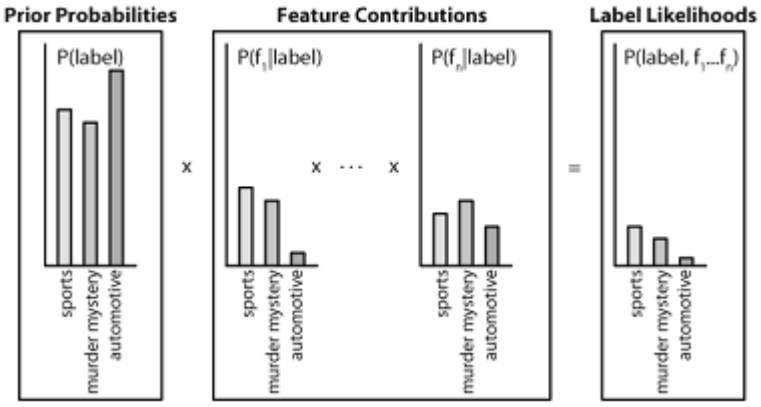


图 6-7. 计算朴素贝叶斯的标签似然得分。朴素贝叶斯以计算每个标签的先验概率开始，基于每个标签出现在训练数据中的频率。然后每个特征都用于估计每个标签的似然估计，通过用输入值中有那个特征的标签的概率乘以它。似然得分结果可以认为是从具有给定的标签和特征集的训练集中随机选取的值的概率的估计，假设所有特征概率是独立的。

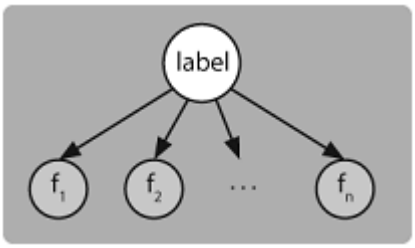


图 6-8. 一个贝叶斯网络图演示朴素贝叶斯分类器假定的生成过程。要生成一个标记的输入，模型先为输入选择标签，然后基于该标签生成每个输入的特征。对给定标签，每个特征被认

为是完全独立于所有其他特征的。

潜在概率模型

理解朴素贝叶斯分类器的另一种方式是它为输入选择最有可能的标签，基于下面的假设：每个输入值是通过首先为那个输入值选择一个类标签，然后产生每个特征的方式产生的，每个特征与其他特征完全独立。当然，这种假设是不现实的，特征往往高度依赖彼此。我们将在本节结尾回过来讨论这个假设的一些后果。这简化的假设，称为**朴素贝叶斯假设**（或**独立性假设**），使得它更容易组合不同特征的贡献，因为我们不必担心它们相互影响。

基于这个假设，我们可以计算表达式 P(label|features)，给定一个特别的特征集一个输入具有特定标签的概率。要为一个新的输入选择标签，我们可以简单地选择使 P(l|features)最大的标签 l。

一开始，我们注意到 P(label|features)等于具有特定标签和特定特征集的输入的概率除以具有特定特征集的输入的概率：

(2) P(label|features) = P(features, label)/P(features)

接下来，我们注意到 P(features)对每个标签选择都相同。因此，如果我们只是对寻找最有可能的标签感兴趣，只需计算 P(features,label)，我们称之为该标签的似然。

如果我们想生成每个标签的概率估计，而不是只选择最有可能的标签，那么计算 P(features)的最简单的方法是简单的计算 P(features, label)在所有标签上的总和：

（3）$P(features) = \Sigma_{label \in labels} P(features, label)$

标签的似然可以展开为标签的概率乘以给定标签的特征的概率：

(4) P(features, label) = P(label) × P(features|label)

此外，因为特征都是独立的（给定标签），我们可以分离每个独立特征的概率：

(5) $P(features, label) = P(label) \times \Pi_{f \in features} P(f|label)$

这正是我们前面讨论的用于计算标签可能性的方程式：P(label)是一个给定标签的先验概率，每个 P(f|label)是一个单独的特征对标签可能性的贡献。

零计数和平滑

最简单的方法计算 P(f|label)，特征 f 对标签 label 的标签可能性的贡献，是取得具有给定特征和给定标签的训练实例的百分比：

(6) P(f|label) = count(f, label)/count(label)

然而，当训练集中有特征从来没有和给定标签一起出现时，这种简单的方法会产生一个问题。在这种情况下，我们的 P(f|label)计算值将是 0，这将导致给定标签的标签可能性为 0。从而，输入将永远不会被分配给这个标签，不管其他特征有多么适合这个标签。

这里的基本问题与我们计算 P(f|label)有关，对于给定标签输入将具有一个特征的概率。特别的，仅仅因为我们在训练集中没有看到特征/标签组合出现，并不意味着该组合不会出现。例如：我们可能不会看到任何谋杀之迷文档中包含词 football，但我们不希望作出结论认为在这些文档中存在是完全不可能。

虽然 count(f,label)/count(label)当 count(f,label)相对高时是 P(f|label)的好的估计，当 count(f)变小时这个估计变得不那么可靠。因此，建立朴素贝叶斯模型时，我们通常采用更复杂

的技术，被称为**平滑**技术，用于计算 P(f|label)，给定标签的特征的概率。例如：给定标签的一个特征的概率的**期望似然估计**基本上给每个 count(f,label)值加 0.5，**Heldout 估计**使用一个 heldout 语料库计算特征频率与特征概率之间的关系。nltk.probability 模块提供了多种平滑技术的支持。

非二元特征

我们这里假设每个特征是二元的，即每个输入要么有这个特征要么没有。标签值特征（例如：颜色特征，可能有红色、绿色、蓝色、白色或橙色）可通过用二元特征，如“颜色是红色”，替换它们，将它们转换为二元特征。数字特征可以通过**装箱**转换为二元特征，装箱是用特征，如“4<X <6”，替换它们。

另一种方法是使用回归方法模拟数字特征的概率。例如：如果我们假设特征 height 具有贝尔曲线分布，那么我们可以通过找到每个标签的输入的 height 的均值和方差来估算 P(height|label)。在这种情况下，P(f=v|label)将不会是一个固定值，会依赖 v 的值变化。

独立的朴素

朴素贝叶斯分类器被称为“naive（天真、朴素）”的原因是它不切实际地假设所有特征相互独立（给定标签）。特别的，几乎所有现实世界的问题含有的特征都不同程度的彼此依赖。如果我们要避免任何依赖其他特征的特征，那将很难构建良好的功能集，提供所需的信息给机器学习算法。

如果我们忽略了独立性假设，使用特征不独立的朴素贝叶斯分类器会发生什么？产生的一个问题是分类器“双重计数”高度相关的特征的影响，将分类器推向更接近给定的标签而不是合理的标签。

来看看这种情况怎么出现的，思考一个包含两个相同的特征 f1 和 f2 的名字性别分类器。换句话说，f2 是 f1 的精确副本，不包含任何新的信息。当分类器考虑输入，在决定选择哪一个标签时，它会同时包含 f1 和 f2 的贡献。因此，这两个特征的信息内容将被赋予比它们应得的更多的比重。

当然，我们通常不会建立包含两个相同的特征的朴素贝叶斯分类器。不过，我们会建立包含相互依赖特征的分类器。例如：特征 ends-with(a)和 ends-with(vowel)是彼此依赖，因为如果一个输入值有第一个特征，那么它也必有第二个特征。对于这些功能，重复的信息可能会被训练集赋予比合理的更多的比重。

双重计数的原因

双重计数问题的原因是在训练过程中特征的贡献被分开计算，但当使用分类器为新输入选择标签时，这些特征的贡献被组合。因此，一个解决方案是考虑在训练中特征的贡献之间可能的相互作用。然后，我们就可以使用这些相互作用调整独立特征所作出的贡献。

为了使这个更精确，我们可以重写计算标签的可能性的方程，分离出每个功能（或标签）所作出的贡献：

$$(7)\ P(features,\ label) = w[label] \times \prod_{f \in features} w[f,\ label]$$

在这里，w[label]是一个给定标签的“初始分数”，w[f, label]是给定特征对一个标签的

可能性所作的贡献。我们称这些值 w[label]和 w[f, label]为模型的**参数**或**权重**。使用朴素贝叶斯算法，我们单独设置这些参数：

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

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

然而，在下一节中，我们将来看一个分类器，它在选择这些参数的值时会考虑它们之间可能的相互作用的。

6.6 最大熵分类器

最大熵分类器使用了一个与朴素贝叶斯分类器使用的模型非常相似的模型。不是使用概率设置模型的参数，它使用搜索技术找出一组将最大限度地提高分类器性能的参数。特别的，它查找使训练语料的**整体可能性**最大的参数组。其定义如下：

(10) $P(features) = \Sigma_{x \in corpus} P(label(x)|features(x))$

其中 P(label|features)，一个特征为 features 将有类标签 label 的输入的概率，被定义为：

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

由于相关特征的影响之间的潜在的复杂的相互作用，没有办法直接计算最大限度地提高训练集的可能性的模型参数。因此，最大熵分类器采用**迭代优化**技术选择模型参数，该技术用随机值初始化模型的参数，然后反复优化这些参数，使它们更接近最优解。这些迭代优化技术保证每次参数的优化都会使它们更接近最佳值，但不一定提供方法来确定是否已经达到最佳值。由于最大熵分类器使用迭代优化技术选择参数，它们花费很长的时间来学习。当训练集的大小、特征的数目以及标签的数目都很大时尤其如此。

一些迭代优化技术比别的快得多。当训练最大熵模型时，应避免使用广义迭代缩放（Generalized Iterative Scaling，GIS）或改进的迭代缩放（Improved Iterative Scaling，IIS），这两者都比共轭梯度（Conjugate Gradient，CG）和 BFGS 优化方法慢很多。

最大熵模型

最大熵分类器模型是一朴素贝叶斯分类器模型的泛化。像朴素贝叶斯模型一样，最大熵分类器为给定的输入值计算每个标签的可能性，通过将适合于输入值和标签的参数乘在一起。朴素贝叶斯分类器模型为每个标签定义一个参数，指定其先验概率，为每个（特征，标签）对定义一个参数，指定独立的特征对一个标签的可能性的贡献。

相比之下，最大熵分类器模型留给用户来决定什么样的标签和特征组合应该得到自己的参数。特别的，它可以使用一个单独的参数关联一个特征与一个以上的标签；或者关联一个以上的特征与一个给定的标签。这有时会允许模型“概括”相关的标签或特征之间的一些差异。

每个接收它自己的参数的标签和特征的组合被称为一个联合特征。请注意，**联合特征**是有标签的的值的属性，而（简单）特征是未加标签的值的属性。

描述和讨论最大熵模型的文字中，术语“特征 features”往往指联合特征；术语“上下文 contexts”指我们一直说的（简单）特征。

通常情况下，用来构建最大熵模型的联合特征完全镜像朴素贝叶斯模型使用的联合特

征。特别的，每个标签定义的联合特征对应于 w[label]，每个（简单）特征和标签组合定义的联合特征对应于 w[f, label]。给定一个最大熵模型的联合特征，分配到一个给定输入的标签的得分与适用于该输入和标签的联合特征相关联的参数的简单的乘积。

(12) $P(input, label) = \prod_{joint\text{-}features(input,label)} w[joint\text{-}feature]$

熵的最大化

进行最大熵分类的直觉是我们应该建立一个模型，捕捉单独的联合特征的频率，不必作任何无根据的假设。举一个例子将有助于说明这一原则。

假设我们被分配从 10 个可能的任务的列表（标签从 A-J）中为一个给定的词找出正确词意的任务。首先，我们没有被告知其他任何关于词或词意的信息。我们可以为 10 种词意选择的概率分布很多，例如：

	A	B	C	D	E	F	G	H	I	J
(i)	10%	10%	10%	10%	10%	10%	10%	10%	10%	10%
(ii)	5%	15%	0%	30%	0%	8%	12%	0%	6%	24%
(iii)	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%

虽然这些分布都可能是正确的，我们很可能会选择分布（i），因为没有任何更多的信息，也没有理由相信任何词的词意比其他的更有可能。另一方面，分布（ii）及（iii）反映的假设不被我们已知的信息支持。

直觉上这种分布（i）比其他的更“公平”，解释这个的一个方法是引用熵的概念。在决策树的讨论中，我们描述了熵作为衡量一套标签是如何“无序”。特别的，如果是一个单独的标签则熵较低，但如果标签的分布比较均匀则熵较高。在我们的例子中，我们选择了分布（i）因为它的标签概率分布均匀——换句话说，因为它的熵较高。一般情况下，**最大熵原理**是说在与我们所知道的一致的的分布中，我们会选择熵最高的。

接下来，假设我们被告知词意 A 出现的次数占 55%。再一次，有许多分布与这一条新信息一致，例如：

	A	B	C	D	E	F	G	H	I	J
(iv)	55%	45%	0%	0%	0%	0%	0%	0%	0%	0%
(v)	55%	5%	5%	5%	5%	5%	5%	5%	5%	5%
(vi)	55%	3%	1%	2%	9%	5%	0%	25%	0%	0%

但是，我们可能会选择最少无根据的假设的分布——在这种情况下，分布（v）。

最后，假设我们被告知词 up 出现在 nearby 上下文中的次数占 10%，当它出现在这个上下文中时有 80%的可能使用词意 A 或 C。从这个意义上讲，将使用 A 或 C。在这种情况下，我们很难手工找到合适的分布；然而，可以验证下面的看起来适当的分布：

		A	B	C	D	E	F	G	H	I	J
(vii)	+up	5.1%	0.25%	2.9%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%	0.25%
	-up	49.9%	4.46%	4.46%	4.46%	4.46%	4.46%	4.46%	4.46%	4.46%	4.46%

特别的，与我们所知道的一致的分布：如果我们将 A 列的概率加起来是 55%，如果我们将第 1 行的概率加起来是 10%；如果我们将+up 行词意 A 和 C 的概率加起来是 8%（或+up 行的 80%）。此外，其余的概率“均匀分布”。

纵观这个例子，我们将自己限制在与我们所知道的一致的分布上。其中，我们选择最高熵的分布。这正是最大熵分类器所做的。特别的，对于每个联合特征，最大熵模型计算该特征的“经验频率”——即它出现在训练集中的频率。然后，它搜索熵最大的分布，同时也预测每个联合特征正确的频率。

生成式分类器对比条件式分类器

朴素贝叶斯分类器和最大熵分类器之间的一个重要差异是它们可以被用来回答问题的类型。朴素贝叶斯分类器是一个**生成式**分类器的例子，建立一个模型，预测 P(input, label)，即(input, label)对的联合概率。因此，生成式模型可以用来回答下列问题：

1. 一个给定输入的最可能的标签是什么？
2. 对于一个给定输入，一个给定标签有多大可能性？
3. 最有可能的输入值是什么？
4. 一个给定输入值的可能性有多大？
5. 一个给定输入具有一个给定标签的可能性有多大？
6. 对于一个可能有两个值中的一个值（但我们不知道是哪个）的输入，最可能的标签是什么？

另一方面，最大熵分类器是**条件式**分类器的一个例子。条件式分类器建立模型预测 P(label|input)——一个给定输入值的标签的概率。因此，条件式模型仍然可以被用来回答问题 1 和 2。然而，条件式模型不能用来回答剩下的问题 3-6。

一般情况下，生成式模型确实比条件式模型强大，因为我们可以从联合概率 P(input, label)计算出条件概率 P(label|input)，但反过来不行。然而，这种额外的能力是要付出代价的。由于该模型更强大的，它也有更多的“自由参数”需要学习的。而训练集的大小是固定的。因此，使用一个更强大的模型时，我们可用来训练每个参数的值的数据也更少，使其难以找到最佳参数值。结果是一个生成式模型回答问题 1 和 2 可能不会与条件式模型一样好，因为条件式模型可以集中精力在这两个问题上。然而，如果我们确实需要像 3-6 问题的答案，那么我们别无选择，只能使用生成式模型。

生成式模型与条件式模型之间的差别类似与一张地形图和一张地平线的图片之间的区别。虽然地形图可用于回答问题的更广泛，制作一张精确的地形图也明显比制作一张精确的地平线图片更加困难。

6.7 为语言模式建模

分类器可以帮助我们理解自然语言中存在的语言模式，允许我们建立明确的**模型**捕捉这些模式。通常情况下，这些模型使用有监督的分类技术，但也可以建立分析型激励模型。无论哪种方式，这些明确的模型有两个重要目的：它们帮助我们了解语言模式，它们可以被用来预测新的语言数据。

明确的模型可以让我们洞察语言模式，在很大程度上取决于使用哪种模型。一些模型，如决策树，相对透明，直接给我们信息：哪些因素是决策中重要的，哪些因素是彼此相关的。另一些模型，如多级神经网络，比较不透明。虽然有可能通过研究它们获得洞察力，但通常需要大量的工作。

但是，所有明确的模型都可以预测新的**未见过**的建立模型时未包括在语料中的语言数据。这些预测进行评估获得模型的准确性。一旦模型被认为足够准确，它就可以被用来自动预测新的语言数据信息。这些预测模型可以组合成系统，执行很多有用的语言处理任务，例如：文档分类、自动翻译、问答系统。

模型告诉我们什么？

理解我们可以从自动构建的模型中学到关于语言的什么是很重要的。处理语言模型时一个重要的考虑因素是描述性模型与解释性模型之间的区别。描述性模型捕获数据中的模式，但它们并不提供任何有关数据包含这些模式的原因的信息。例如：我们在表 3.1 中看到的，同义词 absolutely 和 definitely 是不能互换的：我们说 absolutely adore 而不是 definitely adore，definitely prefer 而不是 absolutely prefer。与此相反，解释性模型试图捕捉造成语言模式的属性和关系。例如：我们可能会介绍一个抽象概念“极性形容词”为一个具有极端意义的形容词，并对一些形容词进行分类，如：adore 和 detest 是相反的两极。我们的解释性模式将包含约束：absolutely 只能与极性形容词结合，definitely 只能与非极性形容词结合。总之，描述性模型提供数据内相关性的信息，而解释性模型再进一步假设因果关系。

大多数从语料库自动构建的模型是描述性模型；换句话说，它们可以告诉我们哪些特征与一个给定的模式或结构有关，但它们不一定能告诉我们这些特征和模式之间如何关联。如果我们的目标是理解语言模式，那么我们就可以使用哪些特征是相关的这一信息作为出发点，设计进一步的实验弄清特征与模式之间的关系。另一方面，如果我们只是对利用该模型进行预测，例如：作为一种语言处理系统的一部分，感兴趣，那么我们可以使用该模型预测新的数据，而不用担心潜在的因果关系的细节。

6.8 小结

- 为语料库中的语言数据建模可以帮助我们理解语言模型，也可以用于预测新语言数据。
- 有监督分类器使用加标签的训练语料库来建立模型，基于输入的特征，预测那个输入的标签。
- 有监督分类器可以执行多种 NLP 任务，包括文档分类、词性标注、语句分割、对话行为类型识别以及确定蕴含关系和很多其他任务。
- 训练一个有监督分类器时，你应该把语料分为三个数据集：用于构造分类器模型的训练集，用于帮助选择和调整模型特性的开发测试集，以及用于评估最终模型性能的测试集。
- 评估一个有监督分类器时，重要的是你要使用新鲜的没有包含在训练集或开发测试集中的数据。否则，你的评估结果可能会不切实际地乐观。
- 决策树可以自动地构建树结构的流程图，用于为输入变量值基于它们的特征加标签，虽然它们易于解释，但不适合处理特性值在决定合适标签过程中相互影响的情况。
- 在朴素贝叶斯分类器中，每个特征决定应该使用哪个标签的贡献是独立的。它允许特征值间有关联，但当两个或更多的特征高度相关时将会有问题。
- 最大熵分类器使用的基本模型与朴素贝叶斯相似；不过，它们使用了迭代优化来寻找使训练集的概率最大化的特征权值集合。
- 大多数从语料库自动构建的模型都是描述性的，也就是说，它们让我们知道哪些特征与给定的模式或结构相关，但它们没有给出关于这些特征和模式之间的因果关系的任何信息。

6.9 进一步阅读

关于本章和如何安装外部机器学习包，如 Weka、Mallet、TADM 和 MegaM 的进一步的

材料请查询 http://www.nltk.org/。更多使用 NLTK 进行分类和机器学习的例子，请参阅在 http://www.nltk.org/howto 的分类 HOWTO。

机器学习的一般介绍我们推荐(Alpaydin, 2004)。更多机器学习理论的数学专著，见(Hastie, Tibshirani & Friedman, 2009)。使用机器学习技术进行自然语言处理的优秀图书包括：(Abney, 2008)、(Daelemans & Bosch, 2005)、(Feldman & Sanger, 2007)、(Segaran, 2007)和(Weiss et al., 2004)。语言问题上的平滑技术的更多信息，请参阅(Manning & Schütze, 1999)。序列建模尤其是隐马尔可夫模型的更多信息，请参阅(Manning & Schütze, 1999)或(Jurafsky & Martin, 2008)。(Manning, Raghavan & Schütze, 2008)的第 13 章讨论了利用朴素贝叶斯分类文本。

许多在本章中讨论的机器学习算法都是数值计算密集的，使用纯 Python 编码时它们运行的很慢。关于 Python 中的数值密集型算法效率的信息，请参阅(Kiusalaas, 2005)。

本章中所描述的分类技术应用范围可以非常广泛。例如：(Agirre & Edmonds, 2007)使用分类器处理词义消歧；(Melamed, 2001)使用分类器创建平行文本。最近的包括文本分类的教科书有(Manning, Raghavan & Schütze, 2008)和 (Croft, Metzler & Strohman, 2009)。

目前机器学习技术应用到 NLP 问题的研究大部分是由政府主办的“挑战”，在那里提供给一些研究机构相同的开发语料库，要求建立一个系统，使用保留的测试集比较系统的结果。这些挑战比赛的例子包括 CoNLL 共享任务（CoNLL Shared Tasks）、文字蕴含识别比赛（Recognizing Textual Entailment competitions），ACE 比赛（ACE competitions）和 AQUAINT 比赛（AQUAINT competitions）。这些挑战的网页链接的列表请参阅 http://www.nltk.org/。

6.10 练习

1. ○阅读在本章中提到的语言技术，如词意消歧、语义角色标注、问答系统、机器翻译，命名实体识别之一。找出开发这种系统需要什么类型和多少数量的已标注的数据。为什么你会认为需要大量的数据？
2. ○使用任何本章所述的三种分类器之一，以及你能想到的特征，尽量好的建立一个名字性别分类器。从将名字语料库分成 3 个子集开始：500 个词为测试集，500 个词为开发测试集，剩余 6900 个词为训练集。然后从示例的名字性别分类器开始，逐步改善。使用开发测试集检查你的进展。一旦你对你的分类器感到满意，在测试集上检查它的最终性能。相比在开发测试集上的性能，它在测试集上的性能如何？这是你期待的吗？
3. ○Senseval 2 语料库包含旨在训练词-词义消歧分类器的数据。它包含四个词的数据：hard、interest、line 和 serve。选择这四个词中的一个，加载相应的数据：

```
>>> from nltk.corpus import senseval
>>> instances = senseval.instances('hard.pos')
>>> size = int(len(instances) * 0.1)
>>> train_set, test_set = instances[size:], instances[:size]
```

   使用这个数据集，建立一个分类器，预测一个给定的实例的正确的词意标签。关于使用 Senseval 2 语料库返回的实例对象的信息请参阅 http://www.nltk.org/howto 上的语料库 HOWTO。
4. ○使用本章讨论过的电影评论文档分类器，产生对分类器最有信息量的 30 个特征的列表。你能解释为什么这些特定特征具有信息量吗？你能在它们中找到什么惊人的发现吗？
5. ○选择一个本章所描述的分类任务，如名字性别检测、文档分类、词性标注或对话行为分类。使用相同的训练和测试数据，相同的特征提取器，建立该任务的三个分类器：决

策树、朴素贝叶斯分类器和最大熵分类器。比较你所选任务上这三个分类器的性能。你如何看待如果你使用了不同的特征提取器，你的结果可能会不同？

6. ○同义词 strong 和 powerful 的模式不同（尝试将它们与 chip 和 sales 结合）。哪些特征与这种区别有关？建立一个分类器，预测每个词何时该被使用。

7. ◑对话行为分类器为每个帖子分配标签，不考虑帖子的上下文背景。然而，对话行为是高度依赖上下文的，一些对话行序列可能比别的更相近。例如：ynQuestion 对话行为更容易被一个 **yanswer** 回答而不是以一个**问候**来回答。利用这一事实，建立一个连续的分类器，为对话行为加标签。一定要考虑哪些特征可能是有用的。参见例 6-5 的词性标记的连续分类器的代码，获得一些想法。

8. ◑词特征在处理文本分类中是非常有用的，因为在一个文档中出现的词对于其语义内容是什么具有强烈的指示作用。然而，很多词很少出现，一些在文档中的最有信息量的词可能永远不会出现在我们的训练数据中。一种解决方法是使用一个**词典**，它描述了词之间的不同。使用 WordNet 词典，加强本章介绍的电影评论文档分类器，使用概括一个文档中出现的词的特征，使之更容易匹配在训练数据中发现的词。

9. ●PP 附件语料库是描述介词短语附着决策的语料库。语料库中的每个实例被编码为 PP Attachment 对象：

```
>>> from nltk.corpus import ppattach
>>> ppattach.attachments('training')
[PPAttachment(sent='0', verb='join', noun1='board',
              prep='as', noun2='director', attachment='V'),
 PPAttachment(sent='1', verb='is', noun1='chairman',
              prep='of', noun2='N.V.', attachment='N'),
 ...]
>>> inst = ppattach.attachments('training')[1]
>>> (inst.noun1, inst.prep, inst.noun2)
('chairman', 'of', 'N.V.')
```

选择 inst.attachment 为 N 的唯一实例：

```
>>> nattach = [inst for inst in ppattach.attachments('training')
...            if inst.attachment == 'N']
```

使用此子语料库，建立一个分类器，尝试预测哪些介词是用来连接一对给定的名词。例如：给定的名词对 team 和 researchers，分类器应该预测出介词 of。更多的使用 PP 附件语料库的信息，参阅 http://www.nltk.org/howto 上的语料库 HOWTO。

10. ●假设你想自动生成一个场景的散文描述，每个实体已经有了一个唯一描述此实体的词，例如：the book，只是想决定在有关的各项目中是否使用 in 或 on，例如：the book is in the cupboard 对 the book is on the shelf。通过查找语料数据，编写需要的程序，探讨这个问题。思考下面的例子：

(13) a. in the car versus on the train
     b. in town versus on campus
     c. in the picture versus on the screen
     d. in Macbeth versus on Letterman

第七章 从文本提取信息

对于任何给定的问题，很可能已经有人把答案写在某个地方了。以电子形式提供的自然语言文本的数量真的惊人，并且与日俱增。然而，自然语言的复杂性使访问这些文本中的信息非常困难。NLP 目前的技术水平仍然有很长的路要走才能够从不受限制的文本建立通用的意义重现。如果我们不是集中我们的精力在问题或“实体关系”的有限集合，例如：“不同的设施位于何处”或“谁被什么公司雇用”上，我们就能取得重大进展。本章的目的是要回答下列问题：

1. 我们如何能构建一个系统，从非结构化文本中提取结构化数据？
2. 有哪些稳健的方法识别一个文本中描述的实体和关系？
3. 哪些语料库适合这项工作，我们如何使用它们来训练和评估我们的模型？

一路上，我们将应用最后两章中的技术来解决分块和命名实体识别。

7.1 信息提取

信息有很多种“形状”和“大小”。一个重要的形式是**结构化数据**：实体和关系的可预测的规范的结构。例如：我们可能对公司和地点之间的关系感兴趣。给定一个公司，我们希望能够确定它做业务的位置；反过来，给定位置，我们会想发现哪些公司在该位置做业务。如果我们的数据是表格形式，如表 7.1 中的例子，那么回答这些问题就很简单了。

表 7-1. 位置数据

机构名	位置名
Omnicom	New York
DDB Needham	New York
Kaplan Thaler Group	New York
BBDO South	Atlanta
Georgia-Pacific	Atlanta

如果这个位置数据被作为一个元组(entity, relation, entity)的链表存储在 Python 中，那么这个问题：“哪些组织在亚特兰大经营？”可翻译如下：

```
>>> print [org for (e1, rel, e2) if rel=='IN' and e2=='Atlanta']
['BBDO South', 'Georgia-Pacific']
```

如果我们尝试从文本中获得相似的信息，事情就比较麻烦了。例如：思考下面的片段(来自 nltk.corpus.ieer，fileid 为 NYT19980315.0085)。

(1) 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.

如果你通读了（1），你将收集到回答例子问题所需的信息。但我们如何能让一台机器理

解(1)返回链表['BBDO South', 'Georgia-Pacific']作为答案呢？这显然是一个困难得多的任务。与表 7.1 不同，（1）不包含连结组织名和位置名的结构。

这个问题的解决方法之一是建立一个非常通用的意义重现（第 10 章）。在这一章中，我们采取不同的方法，提前定为我们将只查找文本中非常具体的各种信息，如：组织和地点之间的关系。不是试图用文字像（1）那样直接回答这个问题，我们首先将自然语言句子这样的**非结构化数据**转换成表 7-1 的结构化数据。然后，利用强大的查询工具，如 SQL。这种从文本获取意义的方法被称为**信息提取**。

信息提取有许多应用，包括商业智能、简历收获、媒体分析、情感检测、专利检索、电子邮件扫描。当前研究的一个特别重要的领域是提取出电子科学文献的结构化数据，特别是在生物学和医学领域。

信息提取结构

图 7-1 显示了一个简单的信息提取系统的结构。它开始于使用第 3 章和第 5 章讨论过的几个程序处理文档：首先，使用句子分割器将该文档的原始文本分割成句，使用分词器将每个句子进一步细分为词。接下来，对每个句子进行词性标注，在下一步**命名实体识别**中将证明这是非常有益的。在这一步，我们寻找每个句子中提到的潜在的有趣的实体。最后，我们使用**关系识别**搜索文本中不同实体间的可能关系。

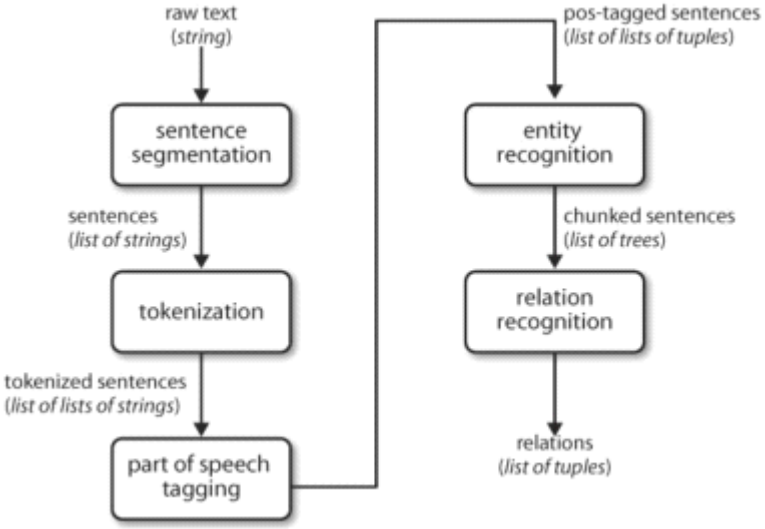


图 7-1. 信息提取系统的简单的流水线结构。该系统以一个文档的原始文本作为其输入，生成(entity, relation, entity)元组的一个链表作为输出。例如：假设一个文档表明 Georgia-Pacific 公司位于 Atlanta，它可能产生元组([ORG: 'Georgia-Pacific'] 'in' [LOC: 'Atlanta'])。

要执行前面三项任务，我们可以定义一个函数，简单地连接 NLTK 中默认的句子分割器①，分词器②和词性标注器③：

```
>>> def ie_preprocess(document):
...     sentences = nltk.sent_tokenize(document) ①
...     sentences = [nltk.word_tokenize(sent) for sent in sentences] ②
...     sentences = [nltk.pos_tag(sent) for sent in sentences] ③
```

请记住我们的例子程序假设你以 import nltk, re, pprint 开始交互式会话或程序。

接下来，命名实体识别中，我们分割和标注可能组成一个有趣关系的实体。通常情况下，这些将被定义为名词短语，例如 the knights who say "ni"或者适当的名称如 Monty Python。在一些任务中，同时考虑不明确的名词或名词块也是有用的，如 every student 或 cats，这些不必要一定与定义 NP 和适当名称一样的方式指示实体。

最后，在提取关系时，我们搜索对文本中出现在附近的实体对之间的特殊模式，并使用这些模式建立元组记录实体之间的关系。

7.2 分块

我们将用于实体识别的基本技术是**分块（chunking）**，分割和标注图 7-2 所示的多标识符序列。小框显示词级标识符和词性标注，大框显示较高级别的分块。每个这种较大的框叫做**一大块（chunk）**。就像分词忽略空白符，分块通常选择标识符的一个子集。同样像分词一样，分块构成的源文本中的片段不能重叠。

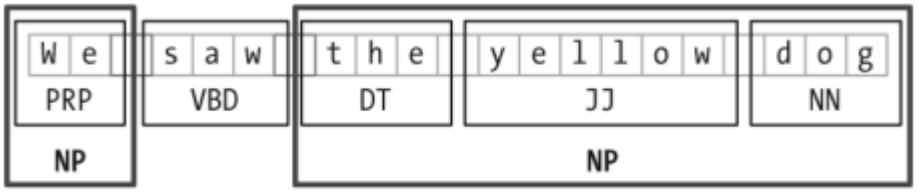


图 7-2. 词标识符和块级别的分割与标注。

在本节中，我们将在较深的层面探讨分块，以块的定义和表示开始。我们将看到正则表达式和 N-gram 的方法分块，使用 CoNLL-2000 分块语料库开发和评估分块器。我们将在 7.5 节和 7.6 节回到命名实体识别和关系抽取的任务。

名词短语分块

我们将首先思考**名词短语分块**，或 **NP-分块**（**NP-chunking**），在那里我们寻找单独名词短语对应的块。例如：这里是一些《华尔街日报》文本，其中的 NP-块用方括号标记：

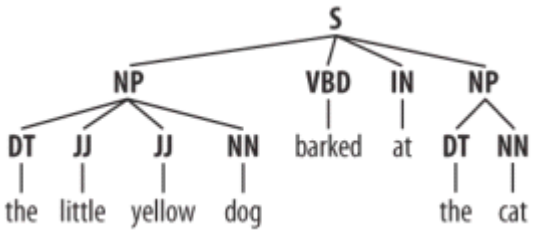
(2) [The/DT market/NN] for/IN [system-management/NN software/NN] for/IN [Digital/NNP] [’s/POS hardware/NN] is/VBZ fragmented/JJ enough/RB that/IN [a/DT giant/NN] such/JJ as/IN [Computer/NNP Associates/NNPS] should/MD do/VB well/RB there/RB ./.

正如我们可以看到，NP-块往往是比完整的名词短语的小片段。例如：the market for system-management software for Digital's hardware 是一个单独的名词短语（含两个嵌套的名词短语），它中间有一个简单的 NP-块 the market。这种差异的动机之一是 NP-块被定义为不包含其他的 NP-块。因此，修饰一个名词的任何介词短语或从句将不包括在相应的 NP-块内，因为它们几乎可以肯定包含更多的名词短语。

NP-分块信息最有用的来源之一是词性标记。这是在我们的信息提取系统中进行词性标注的动机之一。我们在例 7-1 中用一个已经标注词性的例句来演示这种方法。为了创建一个 NP-块，我们将首先定义一个**块语法**，规则句子应如何分块。在本例中，我们将用一个正则表达式规则定义一个简单的语法②。这条规则是说一个 NP-块由一个可选的限定词（DT）后面跟着任何数目的形容词（JJ）然后是一个名词（NN）组成。使用此语法，我们创建了一个块分析器③，测试我们的例句④。结果是一棵树，我们可以输出⑤或图形显示⑥。

例 7-1. 一个简单的基于正则表达式的 NP 分块器的例子。

```
>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),  ①
... ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]
>>> grammar = "NP: {<DT>?<JJ>*<NN>}"  ③
>>> cp = nltk.RegexpParser(grammar)  ③
>>> result = cp.parse(sentence)  ④
>>> print result ⑤
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
>>> result.draw() ⑥
```



标记模式

组成一个块语法的规则使用**标记模式**来描述已标注的词的序列。一个标记模式是一个用尖括号分隔的词性标记序列，如<DT>?<JJ>*<NN>。标记模式类似于正则表达式模式（3.4 节）。现在，思考下面的来自《华尔街日报》的名词短语：

```
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
```

我们可以使用轻微改进的上述第一个标记模式来匹配这些名词短语，如<DT>?<JJ.*>*<NN.*>+。这将把任何以一个可选的限定词开头，后面跟零个或多个任何类型的形容词（包括相对形容词，如 earlier/JJR），后面跟一个或多个任何类型的名词的标识符序列分块。然而，很容易找到许多该规则不包括的更复杂的例子：

```
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
```

轮到你来：尝试用标记模式覆盖这些案例。使用图形界面 nltk.app.chunkparser()测试它们。使用此工具提供的帮助资料继续完善你的标记模式。

用正则表达式分块

要找到一个给定的句子的块结构，RegexpParser 分块器以一个没有标识符被分块的平面结构开始。轮流应用分块规则，依次更新块结构。所有的规则都被调用后，返回块结构。

例 7-2 显示了一个由 2 个规则组成的简单的块语法。第一条规则匹配一个可选的限定词或所有格代名词，零个或多个形容词，然后跟一个名词。第二条规则匹配一个或多个专有名词。我们还定义了一个进行分块的例句①，并在此输入上运行这个分块器②。

例 7-2. 简单的名词短语分块器。

```
grammar = r"""
  NP: {<DT|PP\$>?<JJ>*<NN>}   # chunk determiner/possessive, adjectives and nouns
      {<NNP>+}                # chunk sequences of proper nouns
"""
cp = nltk.RegexpParser(grammar)
sentence = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"), ①
            ("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]
>>> print cp.parse(sentence) ②
(S
  (NP Rapunzel/NNP)
  let/VBD
  down/RP
  (NP her/PP$ long/JJ golden/JJ hair/NN))
```

$符号是正则表达式中的一个特殊字符，必须使用转义符\来匹配 PP$标记。

如果标记模式匹配位置重叠，最左边的匹配优先。例如：如果我们应用一个匹配两个连续的名词文本的规则到一个包含三个连续的名词的文本，则只有前两个名词将分块：

```
>>> 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)
```

一旦我们创建了块 money market，我们就已经消除了允许 fund 被包含在一个块中的上下文。这个问题可以避免，使用一种更加宽容的块规则，如：NP: {<NN>+}。

我们已经为每个块规则添加了一个注释。这些都是可选的。当它们的存在时，分块器将它作为其跟踪输出的一部分输出这些注释。

探索文本语料库

在 5.2 节中，我们看到了我们如何在已标注的语料库中提取匹配的特定的词性标记序列的短语。我们可以使用分块器更容易的做同样的工作，具体如下：

```
>>> cp = nltk.RegexpParser('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)
```

轮到你来：将上面的例子封装在函数 find_chunks()内，以一个如"CHUNK: {<V.*> <TO><V.*>}"的块字符串作为参数。用它来搜索语料库寻找其他几个模式，如四个或更多的连续的名词即"NOUNS:{<N.*>{4,}}"。

加缝隙

有时定义我们想从一个块排除什么比较容易。我们可以为不包括在一大块中的一个标识符序列定义一个**缝隙**。在下面的例子中，barked/VBD at/IN 是一个缝隙：

[the/DT little/JJ yellow/JJ dog/NN] barked/VBD at/IN [the/DT cat/NN]

加缝隙是从一大块中去除一个标识符序列的过程。如果匹配的标识符序列贯穿一整块，那么这一整块会被去除；如果标识符序列出现在块中间，这些标识符会被去除，在以前只有一个块的地方留下两个块。如果序列在块的周边，这些标记被去除，留下一个较小的块。表 7-2 演示了这三种可能性。

表 7-2. 三个加缝隙规则应用于同一个块。

	整个块	块中间	块结尾
输入	[a/DT little/JJ dog/NN]	[a/DT little/JJ dog/NN]	[a/DT little/JJ dog/NN]
操作	Chink “DT JJ NN”	Chink “JJ”	Chink “NN”
模式	}DT JJ NN{	}JJ{	}NN{
输出	a/DT little/JJ dog/NN	[a/DT] little/JJ [dog/NN]	[a/DT little/JJ] dog/NN

在例 7-3 中，我们将整个句子作为一个块，然后练习加缝隙。

例 7-3. 简单的加缝器。

```
grammar = r"""
  NP:
    {<.*>+}          # Chunk everything
    }<VBD|IN>+{      # Chink sequences of VBD and IN
  """
sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
            ("dog", "NN"), ("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]
```

```
cp = nltk.RegexpParser(grammar)
>>> print cp.parse(sentence)
(S
  (NP the/DT little/JJ yellow/JJ dog/NN)
  barked/VBD
  at/IN
  (NP the/DT cat/NN))
```

块的表示：标记与树

作为标注和分析之间的中间状态（第 8 章），块结构可以使用标记或树来表示。使用最广泛的表示是**IOB 标记**。在这个方案中，每个标识符被用三个特殊的块标签之一标注，I（inside，内部），O（outside，外部）或 B（begin，开始）。一个标识符被标注为 B，如果它标志着一个块的开始。块内的标识符子序列被标注为 I。所有其他的标识符被标注为 O。B 和 I 标记加块类型的后缀，如 B-NP, I-NP。当然，没有必要指定出现在块外的标识符的类型，所以这些都只标注为 O。这个方案的例子如图 7 — 3 所示。

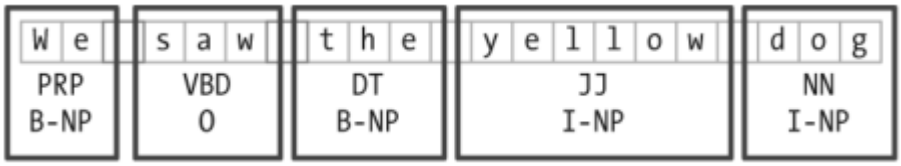


图 7-3. 块结构的标记标识符

IOB 标记已成为文件中表示块结构的标准方式，我们也将使用这种格式。下面是图 7-3 中的信息如何出现在一个文件中的：

```
We PRP B-NP
saw VBD O
the DT B-NP
little JJ I-NP
yellow JJ I-NP
dog NN I-NP
```

在此表示中，每个标识符一行，和它的词性标记与块标记一起。这种格式允许我们表示多个块类型，只要块不重叠。正如我们前面所看到的，块的结构也可以使用树表示。这有利于使每块作为一个组成部分可以直接操作。一个例子如图 7-4 所示。

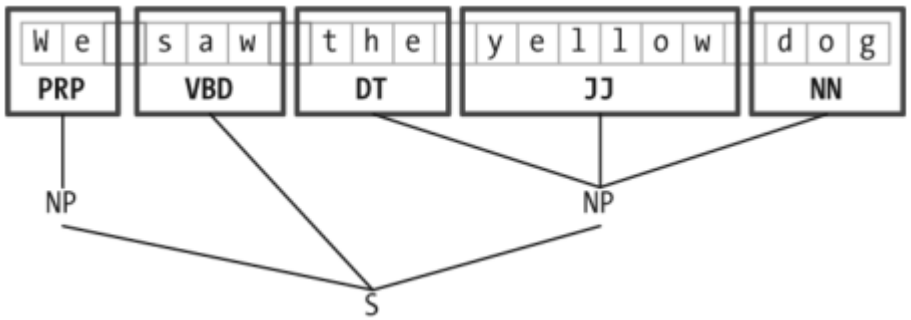


图 7-4. 块结构的树表示。

NLTK 中树作为块的内部表示，并提供这些树与 IOB 格式互换的方法。

7.3 开发和评估分块器

现在你对分块的作用有了一些了解，但我们并没有解释如何评估分块器。和往常一样，这需要一个合适的已标注语料库。我们一开始寻找将 IOB 格式转换成 NLTK 树的机制，然后是使用已分块的语料库如何在一个更大的规模上做这个。我们将看到如何为一个分块器相对一个语料库的准确性打分，再看看一些数据驱动方式搜索 NP 块。我们整个的重点在于扩展一个分块器的覆盖范围。

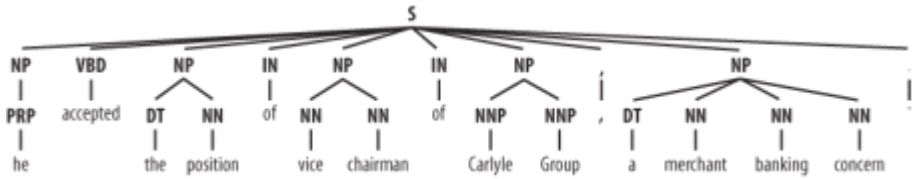
读取 IOB 格式与 CoNLL2000 分块语料库

使用 corpora 模块，我们可以加载已标注的《华尔街日报》文本，然后使用 IOB 符号分块。这个语料库提供的块类型有 NP，VP 和 PP。正如我们已经看到的，每个句子使用多行表示，如下所示：

```
he PRP B-NP
accepted VBD B-VP
the DT B-NP
position NN I-NP
...
```

转换函数 chunk.conllstr2tree()用这些多行字符串建立一个树表示。此外，它允许我们选择使用三个块类型的任何子集，这里只是 NP 块：

```
>>> 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()
```



我们可以使用 NLTK 的 corpus 模块访问较大量的已分块文本。CoNLL2000 分块语料库包含 27 万词的《华尔街日报文本》，分为“训练”和“测试”两部分，标注有词性标记和 IOB 格式分块标记。我们可以使用 nltk.corpus.conll2000 访问这些数据。下面是一个读取语料库的“训练”部分的 100 个句子的例子：

```
>>> from nltk.corpus import conll2000
>>> print 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)
  ./.)
```

正如你看到的，CoNLL2000 分块语料库包含三种块类型：NP 块，我们已经看到了；VP 块如 has already delivered；PP 块如 because of。因为现在我们唯一感兴趣的是 NP 块，我们可以使用 chunk_types 参数选择它们：

```
>>> print 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)
  ./.)
```

简单评估和基准

现在，我们可以访问一个已分块语料，可以评估分块器。我们开始为琐碎的不创建任何块的块分析器 cp 建立一个基准（baseline）：

```
>>> from nltk.corpus import conll2000
>>> cp = nltk.RegexpParser("")
>>> test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
>>> print cp.evaluate(test_sents)
```

```
ChunkParse score:
    IOB Accuracy:  43.4%
    Precision:      0.0%
    Recall:         0.0%
    F-Measure:      0.0%
```

IOB 标记准确性表明超过三分之一的词被标注为 O，即没有在 NP 块中。然而，由于我们的标注器没有找到任何块，其精度、召回率和 F-度量均为零。现在让我们尝试一个初级的正则表达式分块器，查找以名词短语标记的特征字母（如 CD、DT 和 JJ）开头的标记。

```
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> print cp.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  87.7%
    Precision:     70.6%
    Recall:        67.8%
    F-Measure:     69.2%
```

正如你看到的，这种方法达到相当好的结果。但是，我们可以采用更多数据驱动的方法改善它，在这里我们使用训练语料找到对每个词性标记最有可能的块标记（I、O 或 B）。换句话说，我们可以使用 unigram 标注器（5.4 节）建立一个分块器。但不是尝试确定每个词的正确的词性标记，而是给定每个词的词性标记，尝试确定正确的块标记。

在例 7-4 中，我们定义了 UnigramChunker 类，使用 unigram 标注器给句子加块标记。这个类的大部分代码只是用来在 NLTK 的 ChunkParserI 接口使用的分块树表示和嵌入式标注器使用的 IOB 表示之间镜像转换。类定义了两个方法：一个构造函数①，当我们建立一个新的 UnigramChunker 时调用；一个 parse 方法③，用来给新句子分块。

例 7-4. 使用 unigram 标注器对名词短语分块。

```
class UnigramChunker(nltk.ChunkParserI):
    def __init__(self, train_sents): ①
        train_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(sent)]
                      for sent in train_sents]
        self.tagger = nltk.UnigramTagger(train_data) ②
    def parse(self, sentence): ③
        pos_tags = [pos for (word,pos) in sentence]
        tagged_pos_tags = self.tagger.tag(pos_tags)
        chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
        conlltags = [(word, pos, chunktag) for ((word,pos),chunktag)
                     in zip(sentence, chunktags)]
        return nltk.chunk.conlltags2tree(conlltags)
```

构造函数①需要训练句子的一个链表，这将是块树的形式。它首先将训练数据转换成适合训练标注器的形式，使用 tree2conlltags 映射每个块树到一个词，标记，块三元组的链表。然后使用转换好的训练数据训练一个 unigram 标注器，并存储在 self.tagger 供以后使用。

parse 方法②取一个已标注的句子作为其输入，以从那句话提取词性标记开始。然后使用在构造函数中训练过的标注器 self.tagger，为词性标记标注 IOB 块标记。接下来，提取块标记，与原句组合，产生 conlltags。最后，使用 conlltags2tree 将结果转换成一个块

树。

现在我们有了 UnigramChunker，可以使用 CoNLL2000 分块语料库训练它，并测试其性能：

```
>>> test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
>>> train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
>>> unigram_chunker = UnigramChunker(train_sents)
>>> print unigram_chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  92.9%
    Precision:     79.9%
    Recall:        86.8%
    F-Measure:     83.2%
```

这个分块器相当不错，达到整体 F 度量 83%的得分。让我们来看一看通过使用 unigram 标注器分配一个标记给每个语料库中出现的词性标记，它学到了什么：

```
>>> postags = sorted(set(pos for sent in train_sents
...                      for (word,pos) in sent.leaves()))
>>> print unigram_chunker.tagger.tag(postags)
[('#', 'B-NP'), ('$', 'B-NP'), ('"', 'O'), ('(', 'O'), (')', 'O'),
 (',', 'O'), ('.', 'O'), (':', 'O'), ('CC', 'O'), ('CD', 'I-NP'),
 ('DT', 'B-NP'), ('EX', 'B-NP'), ('FW', 'I-NP'), ('IN', 'O'),
 ('JJ', 'I-NP'), ('JJR', 'B-NP'), ('JJS', 'I-NP'), ('MD', 'O'),
 ('NN', 'I-NP'), ('NNP', 'I-NP'), ('NNPS', 'I-NP'), ('NNS', 'I-NP'),
 ('PDT', 'B-NP'), ('POS', 'B-NP'), ('PRP', 'B-NP'), ('PRP$', 'B-NP'),
 ('RB', 'O'), ('RBR', 'O'), ('RBS', 'B-NP'), ('RP', 'O'), ('SYM', 'O'),
 ('TO', 'O'), ('UH', 'O'), ('VB', 'O'), ('VBD', 'O'), ('VBG', 'O'),
 ('VBN', 'O'), ('VBP', 'O'), ('VBZ', 'O'), ('WDT', 'B-NP'),
 ('WP', 'B-NP'), ('WP$', 'B-NP'), ('WRB', 'O'), ('``', 'O')]
```

它已经发现大多数标点符号出现在 NP 块外，除了两种货币符号#和$。它也发现限定词（DT）和所有格（PRP$和 WP$）出现在 NP 块的开头，而名词类型(NN, NNP, NNPS, NNS)大多出现在 NP 的块内。

建立了一个 unigram 分块器，很容易建立一个 bigram 分块器：我们只需要改变类的名称为 BigramChunker，修改例 7-4 中行②构造一个 BigramTagger 而不是 UnigramTagger。由此产生的分块器的性能略高于 unigram 分块器：

```
>>> bigram_chunker = BigramChunker(train_sents)
>>> print bigram_chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  93.3%
    Precision:     82.3%
    Recall:        86.8%
    F-Measure:     84.5%
```

训练基于分类器的分块器

无论是基于正则表达式的分块器还是 n-gram 分块器，决定创建什么块完全基于词性标

记。然而，有时词性标记不足以确定一个句子应如何分块。例如：考虑下面的两个语句：

(3) a. Joey/NN sold/VBD the/DT farmer/NN rice/NN ./.

b. Nick/NN broke/VBD my/DT computer/NN monitor/NN ./.

这两句话的词性标记相同，但分块方式不同。在第一句中，the farmer 和 rice 都是单独的块，而在第二个句子中相应的部分，the computer monitor，是一个单独的块。显然，如果我们想最大限度地提升分块的性能，我们需要使用词的内容信息作为词性标记的补充。

我们包含词的内容信息的方法之一是使用基于分类器的标注器对句子分块。如在上一节使用的 n-gram 分块器，这个基于分类器的分块器分配 IOB 标记给句子中的词，然后将这些标记转换为块。对于基于分类器的标注器本身，我们将使用与我们在 6.1 节建立词性标注器相同的方法。

基于分类器的 NP 分块器的基础代码如例 7-5 所示。它包括两个类：第一个类①几乎与例 6-5 中 ConsecutivePosTagger 类相同。仅有的两个区别是它调用一个不同的特征提取器②，使用 MaxentClassifier 而不是 NaiveBayesClassifier③；第二个类④基本上是标注器类的一个包装器，将它变成一个分块器。训练期间，这第二个类映射训练语料中的块树到标记序列；在 parse()方法中，它将标注器提供的标记序列转换回一个块树。

例 7-5. 使用连续分类器对名词短语分块。

```
class ConsecutiveNPChunkTagger(nltk.TaggerI): ①
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = npchunk_features(untagged_sent, i, history) ②
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.MaxentClassifier.train( ③
            train_set, algorithm='megam', trace=0)
    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
class ConsecutiveNPChunker(nltk.ChunkParserI): ④
    def __init__(self, train_sents):
        tagged_sents = [[((w,t),c) for (w,t,c) in
                         nltk.chunk.tree2conlltags(sent)]
                        for sent in train_sents]
        self.tagger = ConsecutiveNPChunkTagger(tagged_sents)
    def parse(self, sentence):
        tagged_sents = self.tagger.tag(sentence)
        conlltags = [(w,t,c) for ((w,t),c) in tagged_sents]
```

```
        return nltk.chunk.conllstags2tree(conllstags)
```

留下来唯一需要填写的是特征提取器。首先，我们定义一个简单的特征提取器，它只是提供了当前标识符的词性标记。使用此特征提取器，我们的基于分类器的分块器的性能与 unigram 分块器非常类似：

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     return {"pos": pos}
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  92.9%
    Precision:     79.9%
    Recall:        86.7%
    F-Measure:     83.2%
```

我们还可以添加一个特征：前面词的词性标记。添加此特征允许分类器模拟相邻标记之间的相互作用，由此产生的分块器与 bigram 分块器非常接近。

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         prevword, prevpos = "<START>", "<START>"
...     else:
...         prevword, prevpos = sentence[i-1]
...     return {"pos": pos, "prevpos": prevpos}
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  93.6%
    Precision:     81.9%
    Recall:        87.1%
    F-Measure:     84.4%
```

下一步，我们将尝试把当前词增加为特征，因为我们假设这个词的内容应该对分块有用。我们发现这个特征确实提高了分块器的性能，大约 1.5 个百分点（相应的错误率减少大约 10%）。

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         prevword, prevpos = "<START>", "<START>"
...     else:
...         prevword, prevpos = sentence[i-1]
...     return {"pos": pos, "word": word, "prevpos": prevpos}
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:  94.2%
```

```
    Precision:      83.4%
    Recall:         88.6%
    F-Measure:      85.9%
```

最后，我们尝试用多种附加特征扩展特征提取器，例如：预取特征①、配对功能②和复杂的语境特征③。这最后一个特征，被称为 tags-since-dt，创建一个字符串，描述自最近的限定词以来遇到的所有词性标记。

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         prevword, prevpos = "<START>", "<START>"
...     else:
...         prevword, prevpos = sentence[i-1]
...     if i == len(sentence)-1:
...         nextword, nextpos = "<END>", "<END>"
...     else:
...         nextword, nextpos = sentence[i+1]
...     return {"pos": pos,
...             "word": word,
...             "prevpos": prevpos,
...             "nextpos": nextpos,
...             "prevpos+pos": "%s+%s" % (prevpos, pos),
...             "pos+nextpos": "%s+%s" % (pos, nextpos),
...             "tags-since-dt": tags_since_dt(sentence, i)}
>>> def tags_since_dt(sentence, i):
...     tags = set()
...     for word, pos in sentence[:i]:
...         if pos == 'DT':
...             tags = set()
...         else:
...             tags.add(pos)
...     return '+'.join(sorted(tags))
>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print chunker.evaluate(test_sents)
ChunkParse score:
    IOB Accuracy:    95.9%
    Precision:       88.3%
    Recall:          90.7%
    F-Measure:       89.5%
```

轮到你来：尝试为特征提取器函数 npchunk_features 增加不同的特征，看看是否可以进一步改善的 NP 分块器的性能。

7.4 语言结构中的递归

用级联分块器构建嵌套结构

到目前为止，我们的块结构一直是相对平的。已标注标识符组成的树在如 NP 这样的块节点下任意组合。然而，只需创建一个包含递归规则的多级的块语法，就可以建立任意深度的块结构。例 7-6 是名词短语、介词短语、动词短语和句子的模式。这是一个四级块语法器，可以用来创建深度最多为 4 的结构。

例 7-6. 一个分块器，处理 NP，PP，VP 和 S。

```
grammar = r"""
  NP: {<DT|JJ|NN.*>+}            # Chunk sequences of DT, JJ, NN
  PP: {<IN><NP>}                 # Chunk prepositions followed by NP
  VP: {<VB.*><NP|PP|CLAUSE>+$}   # Chunk verbs and their arguments
  CLAUSE: {<NP><VP>}             # Chunk NP, VP
  """
cp = nltk.RegexpParser(grammar)
sentence = [("Mary", "NN"), ("saw", "VBD"), ("the", "DT"), ("cat", "NN"),
    ("sit", "VB"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(sentence)
(S
  (NP Mary/NN)
  saw/VBD
  (CLAUSE
    (NP the/DT cat/NN)
    (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))
```

不幸的是，这一结果丢掉了 saw 为首的 VP。它还有其他缺陷。当我们将此分块器应用到一个有更深嵌套的句子时，让我们看看会发生什么。请注意，它无法识别①开始的 VP 块。

```
>>> sentence = [("John", "NNP"), ("thinks", "VBZ"), ("Mary", "NN"),
...     ("saw", "VBD"), ("the", "DT"), ("cat", "NN"), ("sit", "VB"),
...     ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> print cp.parse(sentence)
(S
  (NP John/NNP)
  thinks/VBZ
  (NP Mary/NN)
  saw/VBD  ①
  (CLAUSE
    (NP the/DT cat/NN)
    (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))
```

这些问题的解决方案是：让分块器在它的模式中循环：尝试完所有模式之后，重复此过程。我们添加一个可选的第二个参数 loop 指定这套模式应该循环的次数：

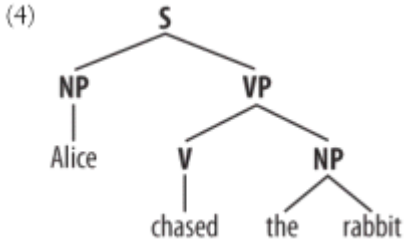
```
>>> cp = nltk.RegexpParser(grammar, loop=2)
>>> print cp.parse(sentence)
```

```
(S
  (NP John/NNP)
  thinks/VBZ
  (CLAUSE
    (NP Mary/NN)
    (VP
      saw/VBD
      (CLAUSE
        (NP the/DT cat/NN)
        (VP sit/VB (PP on/IN (NP the/DT mat/NN)))))))
```

这个级联过程使我们能创建深层结构。然而，创建和调试级联过程是困难的，关键点是它能更有效地做全面的分析（见第 8 章）。另外，级联过程只能产生固定深度的树（不超过级联级数），完整的句法分析这是不够的。

树

树是一组连接的加标签节点，从一个特殊的根节点沿一条唯一的路径到达每个节点。下面是一棵树的例子（注意：它们标准的画法是颠倒的）：



我们用“家庭”来比喻树中节点的关系：例如：S 是 VP 的**父母**；反之 VP 是 S 的一个**孩子**。此外，由于 NP 和 VP 同为 S 的两个孩子，它们也是**兄弟**。为方便起见，也有特定树的文本格式：

```
(S
  (NP Alice)
  (VP
    (V chased)
    (NP
      (Det the)
      (N rabbit))))
```

虽然我们将只集中关注语法树，树可以用来编码任何同构的超越语言形式序列的层次结构（如形态结构、篇章结构）。一般情况下，叶子和节点值不一定要是字符串。

在 NLTK 中，我们创建了一棵树，通过给一个节点添加标签和一个孩子链表：

```
>>> tree1 = nltk.Tree('NP', ['Alice'])
>>> print tree1
(NP Alice)
>>> tree2 = nltk.Tree('NP', ['the', 'rabbit'])
```

```
>>> print tree2
(NP the rabbit)
```

我们可以将这些不断合并成更大的树，如下所示：

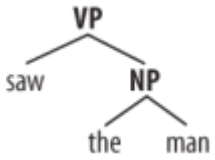
```
>>> tree3 = nltk.Tree('VP', ['chased', tree2])
>>> tree4 = nltk.Tree('S', [tree1, tree3])
>>> print tree4
(S (NP Alice) (VP chased (NP the rabbit)))
```

下面是树对象的一些的方法：

```
>>> print tree4[1]
(VP chased (NP the rabbit))
>>> tree4[1].node
'VP'
>>> tree4.leaves()
['Alice', 'chased', 'the', 'rabbit']
>>> tree4[1][1][1]
'rabbit'
```

复杂的树用括号表示难以阅读。在这些情况下，draw 方法是非常有用的。它会打开一个新窗口，包含树的一个图形表示。树显示窗口可以放大和缩小，子树可以折叠和展开，并将图形表示输出为一个 postscript 文件（包含在一个文档中）。

```
>>> tree3.draw()
```



树遍历

使用递归函数来遍历树是标准的做法。例 7-7 中的列表进行了演示。

例 7-7. 递归函数遍历树。

```
def traverse(t):
    try:
        t.node
    except AttributeError:
        print t,

    else:
        # Now we know that t.node is defined
        print '(', t.node,
        for child in t:
            traverse(child)
        print ')',
>>> t = nltk.Tree('(S (NP Alice) (VP chased (NP the rabbit)))')
>>> traverse(t)
( S ( NP Alice ) ( VP chased ( NP the rabbit ) ) )
```

我们已经使用了一种叫做动态类型的技术，检测 t 是一棵树（如：定义了 t.node）。

7.5 命名实体识别

在本章开头，我们简要介绍了命名实体（NEs）。命名实体是确切的名词短语，指示特定类型的个体，如组织、人、日期等。表 7-3 列出了一些较常用的 NEs 类型。这些应该是不言自明的，除了“FACILITY”：建筑和土木工程领域的人造产品；以及“GPE”：地缘政治实体，如城市、州/省、国家。

表 7-3. 常用命名实体类型

NE 类型	例子
组织	*Georgia-Pacific Corp., WHO*
人	*Eddy Bonte, President Obama*
地点	*Murray River, Mount Everest*
日期	*June, 2008-06-29*
时间	*two fifty a m, 1:30 p.m.*
货币	*175 million Canadian Dollars, GBP 10.40*
百分数	*twenty pct, 18.75 %*
设施	*Washington Monument, Stonehenge*
地缘政治实体	*South East Asia, Midlothian*

命名实体识别（NER）系统的目标是识别所有文字提及的命名实体。可以分解成两个子任务：确定 NE 的边界和确定其类型。命名实体识别经常是信息提取中关系识别的前奏，它也有助于其他任务。例如：在问答系统（QA）中，我们试图提高信息检索的精确度，不是返回整个页面而只是包含用户问题的答案的那些部分。大多数 QA 系统利用标准信息检索返回的文件，然后尝试分离文档中包含答案的最小的文本片段。现在假设问题是 Who was the first President of the US？被检索的一个文档中包含下面这段话：

(5) The Washington Monument is the most prominent structure in Washington, D.C. and one of the city’s early attractions. It was built in honor of George Washington, who led the country to independence and then became its first President.

分析问题时我们想到答案应该是 X was the first President of the US 的形式，其中 X 不仅是一个名词短语也是一个 PER 类型的命名实体。这应该使我们忽略段落中的第一句话。虽然它包含 Washington 的两个出现，命名实体识别应该告诉我们：它们都不是正确的类型。

我们如何识别命名实体呢？一个办法是查找一个适当的名称列表。例如：识别地点时，我们可以使用地名辞典，如亚历山大地名辞典或盖蒂地名辞典。然而，盲目这样做会出问题，如图 7-5 所示。

图 7-5：地点检测，通过在新闻故事中简单的查找：查找地名辞典中的每个词是容易出错的；案例区分可能有所帮助，但它们不是总会有的。

请看地名辞典很好的覆盖了很多国家的地点，却错误地认为 Sanchez 在多米尼加共和国而 On 在越南。当然，我们可以从地名辞典中忽略这些地名，但这样一来当它们出现在一个文档中时，我们将无法识别它们。

人或组织的名称的情况更加困难。任何这些名称的列表都肯定覆盖不全。每天都有新的组织出现，如果我们正在努力处理当代文本或博客条目，使用名称辞典查找来识别众多实体是不可能的。

困难的另一个原因是许多命名实体措辞有歧义。May 和 North 可能分别是日期和地点类型的命名实体的，但也可以都是人名；相反的，Chris-tian Dior 看上去像是一个人名，但更可能是组织类型。词 Yankee 在某些上下文中是普通的修饰语，但在短语 Yankee infielders 中会被标注为组织类型的一个实体。

更大的挑战来自如 Stanford University 这样的多词名称和包含其他名称的名称，如 Cecil H. Green Library 和 Escondido Village Conference Service Center。因此，在命名实体识别中，我们需要能够识别多标识符序列的开头和结尾。

命名实体识别是一个非常适合用基于分类器类型的方法来处理的任务，这些方法我们在名词短语分块时看到过。特别是，我们可以建立一个标注器，为使用 IOB 格式的每个块都加了适当类型标签的句子中的每个词加标签。这里是 CONLL 2002（conll2002）荷兰语训练数据的一部分：

```
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
```

在上面的表示中，每个标识符一行，与它的词性标记及命名实体标记一起。基于这个训练语料，我们可以构造一个可以用来标注新句子的标注器，使用 nltk.chunk.conlltags2tree()函数将标记序列转换成一个块树。

NLTK 提供了一个已经训练好的可以识别命名实体的分类器，使用函数 nltk.ne_chunk()访问。如果我们设置参数 binary=True①，那么命名实体只被标注为 NE；否则，分类器会添加类型标签，如 PERSON, ORGANIZATION, and GPE。

```
>>> sent = nltk.corpus.treebank.tagged_sents()[22]
```

```
>>> print nltk.ne_chunk(sent, binary=True) ①
(S
  The/DT
  (NE U.S./NNP)
  is/VBZ
  one/CD
  ...
  according/VBG
  to/TO
  (NE Brooke/NNP T./NNP Mossman/NNP)
  ...)
>>> print nltk.ne_chunk(sent)
(S
  The/DT
  (GPE U.S./NNP)
  is/VBZ
  one/CD
  ...
  according/VBG
  to/TO
  (PERSON Brooke/NNP T./NNP Mossman/NNP)
...)
```

7.6 关系抽取

一旦文本中的命名实体已被识别，我们就可以提取它们之间存在的关系。如前所述，我们通常会寻找指定类型的命名实体之间的关系。进行这一任务的方法之一是首先寻找所有(X, α, Y)形式的三元组，其中 X 和 Y 是指定类型的命名实体，α 表示 X 和 Y 之间关系的字符串。然后我们可以使用正则表达式从 α 的实体中抽出我们正在查找的关系。下面的例子搜索包含词 in 的字符串。特殊的正则表达式(?!\b.+ing\b)是一个否定预测先行断言，允许我们忽略如 success in supervising the transition of 中的字符串，其中 in 后面跟一个动名词。

```
>>> IN = re.compile(r'.*\bin\b(?!\b.+ing)')
>>> for doc in nltk.corpus.ieer.parsed_docs('NYT_19980315'):
...     for rel in nltk.sem.extract_rels('ORG', 'LOC', doc,
...                                      corpus='ieer', pattern = IN):
...         print nltk.sem.show_raw_rtuple(rel)
[ORG: 'WHYY'] 'in' [LOC: 'Philadelphia']
[ORG: 'McGlashan &AMP; 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']
```

搜索关键字 in 执行的相当不错，虽然它的检索结果也会误报，例如： [ORG: House Transportation Committee] , secured the most money in the [LOC: New York]；一种简单的基于字符串的方法排除这样的填充字符串似乎不太可能。

如前文所示，CoNLL 2002 命名实体语料库的荷兰语部分不只包含命名实体标注，也包含词性标注。这允许我们设计对这些标记敏感的模式，如下面的例子所示。show_clause() 方法以分条形式输出关系，其中二元关系符号作为参数 relsym 的值被指定①。

```
>>> from nltk.corpus import conll2002
>>> vnv = """
... (
... is/V|    # 3rd sing present and
... was/V|   # past forms of the verb zijn ('be')
... werd/V|  # and also present
... wordt/V  # past of worden ('become')
... )
... .*       # followed by anything
... van/Prep # followed by van ('of')
... """
>>> VAN = re.compile(vnv, re.VERBOSE)
>>> for doc in conll2002.chunked_sents('ned.train'):
...     for r in nltk.sem.extract_rels('PER', 'ORG', doc,
...                                    corpus='conll2002', pattern=VAN):
...         print nltk.sem.show_clause(r, relsym="VAN")  ①
VAN("cornet_d'elzius", 'buitenlandse_handel')
VAN('johan_rottiers', 'kardinaal_van_roey_instituut')
VAN('annie_lennox', 'eurythmics')
```

轮到你来：替换最后一行①为 print show_raw_rtuple(rel,lcon=True, rcon=True)。这将显示实际的词表示两个 NE 之间关系以及它们左右的默认 10 个词的窗口的上下文。在一本荷兰语词典的帮助下，你也许能够找出为什么结果 VAN('annie_lennox', 'eurythmics')是个误报。

7.7 小结

- 信息提取系统搜索大量非结构化文本，寻找特定类型的实体和关系，并用它们来填充有组织的数据库。这些数据库就可以用来寻找特定问题的答案。
- 信息提取系统的典型结构以断句开始，然后是分词和词性标注。接下来在产生的数据中搜索特定类型的实体。最后，信息提取系统着眼于文本中提到的相互临近的实体，并试图确定这些实体之间是否有指定的关系。
- 实体识别通常采用分块器，它分割多标识符序列，并用适当的实体类型给它们加标签。

常见的实体类型包括组织、人员、地点、日期、时间、货币、GPE（地缘政治实体）。

- 用基于规则的系统可以构建分块器，例如：NLTK 中提供的 RegexpParser 类；或使用机器学习技术，如本章介绍的 ConsecutiveNPChunker。在这两种情况中，词性标记往往是搜索块时的一个非常重要的特征。
- 虽然分块器专门用来建立相对平坦的数据结构，其中没有任何两个块允许重叠，但它们可以被串联在一起，建立嵌套结构。
- 关系抽取可以使用基于规则的系统，它通常查找文本中的连结实体和相关的词的特定模式；或使用机器学习系统，通常尝试从训练语料自动学习这种模式。

7.8 进一步阅读

本章额外的材料，包括网上免费提供的资源链接，发布在 http://www.nltk.org/。更多关于使用 NLTK 分块的例子，请参阅 http://www.nltk.org/howto 上的分块 HOWTO。

分块的普及很大一部分是由于 Abney 的开创性的工作，如(Abney, 1996a)。http://www.vinartus.net/spa/97a.pdf 中描述了 Abney 的 Cass 分块器。

根据 Ross 和 Tukey 1975 年的论文，词 **chink** 最初的意思是一个停用词序列(Abney, 1996a)。

IOB 格式（有时也称为 **BIO Format**）由(Ramshaw & Marcus, 1995)开发用来 NP 分块，并被由 Conference on Natural Language Learning（CoNLL）在 1999 年用于 NP 加括号共享任务。CoNLL2000 采用相同的格式标注了华尔街日报的文本作为一个 NP 分块共享任务的一部分。

(Jurafsky & Martin, 2008)的 13.5 节包含有关分块的一个讨论。第 22 章讲述信息提取，包括命名实体识别。有关生物学和医学中的文本挖掘的信息，请参阅(Ananiadou & McNaught, 2006)。

亚历山大地名辞典和盖蒂地名辞典的更多信息请参阅 http://en.wikipedia.org/wiki/Getty_Thesaurus_of_Geographic_Names 和 http://www.alexandria.ucsb.edu/gazetteer/。

7.9 练习

1. ○IOB 格式分类标注标识符为 I、O 和 B。三个标签为什么是必要的？如果我们只使用 I 和 O 标记会造成什么问题？
2. ○写一个标记模式匹配包含复数中心名词在内的名词短语，如 many/JJ researchers/NNS, two/CD weeks/NNS, both/DT new/JJ positions/NNS。通过泛化处理单数名词短语的标记模式，尝试做这个。
3. ○选择 CoNLL-2000 分块语料库中三种块类型之一。查看这些数据，并尝试观察组成这种类型的块的 POS 标记序列的任一模式。使用正则表达式分块器 nltk.RegexpParser 开发一个简单的分块器。讨论任何难以可靠分块的标记序列。
4. ○块的早期定义是出现在缝隙之间的材料。开发一个分块器以将完整的句子作为一个单独的块开始，然后其余的工作完全由加缝隙完成。在你自己的应用程序的帮助下，确定哪些标记（或标记序列）最有可能组成缝隙。相对于完全基于块规则的分块器比较这种方法的性能和易用性。
5. ◑写一个标记模式，涵盖包含动名词在内的名词短语，如 the/DT receiving/VBG end/NN, assistant/NN managing/VBG editor/NN。将这些模式加入到 grammar，

每行一个。用自己设计的一些已标注的句子，测试你的工作。

6. ◑写一个或多个标记模式处理有连接词的名词短语，如：July/NNP and/CC August /NNP，all/DT your/PRP$ managers/NNS and/CC supervisors/NNS，compa ny/NN courts/NNS and/CC adjudicators/NNS。
7. ◑用任何你之前已经开发的分块器执行下列评估任务。（请注意，大多数分块语料库包含一些内部的不一致，以至于任何合理的基于规则的方法都将产生错误。）
   a. 在来自分块语料库的 100 个句子上评估你的分块器，报告精度、召回率和 F 量度。
   b. 使用 chunkscore.missed()和 chunkscore.incorrect()方法识别你的分块器的错误，并讨论它。
   c. 与本章的评估部分讨论的基准分块器比较你的分块器的性能。
8. ◑使用基于正则表达式的块语法 RegexpChunk，为 CoNLL 分块语料库中块类型中的一个开发一个分块器。使用分块、加缝隙、合并或拆分规则的任意组合。
9. ◑有时一个词的标注不正确，例如 12/CD or/CC so/RB cases/VBZ 的中心名词。替代手工校正标注器的输出，好的分块器使用标注器的错误输出也能运作。查找使用不正确的标记正确为名词短语分块的其他例子。
10. ◑bigram 分块器的准确性得分约为 90%。研究它的错误，并试图找出它为什么不能获得 100%的准确率。实验 trigram 分块。你能够在提高性能吗？
11. ●在 IOB 块标注上应用 n-gram 和 Brill 标注方法。不是给词分配 POS 标记，在这里我们给 POS 标记分配 IOB 标记。例如：如果标记 DT（限定符）经常出现在一个块的开头，它会被标注为 B（begin）。相对于本章中讲到的正则表达式分块方法，评估这些分块方法的性能。
12. ●在第 5 章我们看到通过查找有歧义的 n-grams，即在训练数据中有多种可能的方式标注的 n-grams，可以得到标注性能的上限。应用同样的方法来确定一个 n-gram 分块器的上限。
13. ●挑选 CoNLL 分块语料库中三种块类型之一。写一个函数为你选择的类型做以下任务：
   a. 列出与此块类型的每个实例一起出现的所有标记序列。
   b. 计数每个标记序列的频率，并产生一个按频率减少的顺序排列的列表；每行要包含一个整数（频率）和一个标记序列。
   c. 检查高频标记序列。使用这些作为开发一个更好的分块器的基础。
14. ●在评估一节中提到的基准分块器往往会产生比它应该产生的块更大的块。例如：短语 [every/DT time/NN] [she/PRP] sees/VBZ [a/DT newspaper/NN]包含两个连续的块，我们的基准分块器不正确地将前两个结合： [every/DT time/NN she/P RP]。写一个程序，找出这些通常出现在一个块的开头的块内部的标记有哪些，然后设计一个或多个规则分裂这些块。将这些与现有的基准分块器组合，重新评估它，看看你是否已经发现了一个改进的基准。
15. ●开发一个 NP 分块器，转换 POS 标注文本为元组的一个链表，其中每个元组由一个后面跟一个名词短语和介词的动词组成，如：the little cat sat on the mat becomes ('s at', 'on', 'NP')...
16. ●宾州树库样例包含一部分已标注的《华尔街日报》文本，已经按名词短语分块。格式使用方括号，我们已经在本章遇到它了几次。语料可以使用 for sent in nltk.corpus. treebank_chunk.chunked_sents(fileid)来访问。这些都是平坦的树，正如我们使用 nltk.corpus.conll2000.chunked_sents()得到的一样。
   a. 函数 nltk.tree.pprint()和 nltk.chunk.tree2conllstr()可以用来从一棵树创建树库和 IOB 字符串。写函数 chunk2brackets()和 chunk2iob()，以一个单独的

块树为它们唯一的参数，返回所需的多行字符串表示。

b. 写命令行转换工具 bracket2iob.py 和 iob2bracket.py，（分别）读取树库或 CoNLL 格式的一个文件，将它转换为其他格式。（从 NLTK 语料库获得一些原始的树库或 CoNLL 数据，保存到一个文件，然后使用 for line in open(filename)从 Python 访问它。）

17. ●一个 n-gram 分块器可以使用除当前词性标记和 n-1 个前面的块的标记以外其他信息。调查其他的上下文模型，如 n-1 个前面的词性标记，或一个写前面块标记连同前面和后面的词性标记的组合。

18. ●思考一个 n-gram 标注器使用临近的标记的方式。现在观察一个分块器可能如何重新使用这个序列信息。例如：这两个任务将使用名词往往跟在形容词后面（英文中）的信息。这会出现相同的信息被保存在两个地方的情况。随着规则集规模增长，这会成为一个问题吗？如果是，推测可能会解决这个问题的任何方式。

第 8 章 分析句子结构

前面的章节重点关注词：如何识别它们，分析它们的结构，分配给他们词汇类别，以及获得它们的含义。我们还看到了如何识别词序列或 n-grams 的模式。然而，这些方法只触碰到管理句子的复杂约束的表面。我们需要一种方法处理自然语言中显著的歧义。我们还需要能够应对这样一个事实：句子有无限的可能，而我们只能写有限的程序来分析其结构和发现它们的含义。

本章的目的是要回答下列问题：

1. 我们如何使用形式化语法来描述无限的句子集合的结构？
2. 我们如何使用句法树来表示句子结构？
3. 语法分析器如何分析一个句子并自动构建语法树？

一路上，我们将覆盖英语语法的基础，并看到句子含义有系统化的一面，只要我们确定了句子结构，将更容易捕捉。

8.1 一些语法困境

语言数据和无限可能性

前面的章节中已经为你讲述了如何处理和分析的文本语料库，我们一直强调处理大量的每天都在增加的电子语言数据是 NLP 的挑战。让我们更加细致的思考这些数据，做一个思想上的实验：我们有一个巨大的语料库，包括在过去 50 年中英文表达或写成的一切。我们称这个语料库为“现代英语”合理吗？有许多原因，为什么我们的回答可能是否定的。回想一下，在第 3 章中，我们让你搜索网络查找 the of 模式的实例。虽然很容易在网上找到包含这个词序列的例子，例如：New man at the of IMG（见 http://www.telegraph.co.uk/sport/2387900/New-man-at-the-of-IMG.html），说英语的人会说大多数这样的例子是错误的，因此它们根本不是英语。

因此，我们可以说，“现代英语”并不等同于我们想象中的语料库中的非常大的词序列的集合。说英语的人可以判断这些序列，并将拒绝其中一些不合文法的。

同样，组成一个新的句子，并让说话者认为它是非常好的英语是很容易的。例如：句子有一个有趣的属性：它们可以嵌入更大的句子中。考虑下面的句子：

(1) 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.

如果我们用符号 S 表示整个句子，我们会看到像这样的模式：Andre said S and I think S。这些模板可以得到一个句子，并构建一个更大的句子。还有其他的我们可以使用的模板，如：S but S and S when S。稍微动点儿心思，我们就可以使用这些模板构建一些很

长的句子。这里有一个令人印象深刻的例子，来自 A.A. Milne 的《小熊维尼的故事》：《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…

这个长句子其实结构很简单，以 S but S when S 开始。从这个例子我们可以看到：语言提供给我们的结构，看上去可以无限扩展句子。这也是惊人的，我们能理解我们从来没有听说过的任意长度的句子：并不难假造一个全新的句子，一个在语言的历史上可能从来没有使用过的句子，而所有说这种语言的人都能理解它。

文法的目的是给出一个明确的语言描述。而我们思考文法的方式与我们认为这是一种语言紧密联系在一起。可以观察到的言语和书面文本是否是一个大却有限的集合？具有文法的句子是否有一些更抽象的东西，如隐性知识这样的，有能力的说话者能理解的？或者是两者的某种组合？我们不会在这个问题上采取立场，而是将介绍主要的方法。

在这一章中，我们将采取"生成文法"的形式化框架，其中一种"语言"被认为是所有合乎文法的句子的巨大集合，一个文法是一个形式化符号，可用于"产生"这个集合的成员。文法使用 S → S and S 形式的递归产生式，我们将在 8.3 节探讨。第 10 章中，我们会将其扩展到从它的组成部分的意思自动建立一个句子的意思。

普遍存在的歧义

一个众所周知的关于歧义例子如（2）所示，来自 Groucho Marx 的电影，Animal Crackers (1930)：

(2) While hunting in Africa, I shot an elephant in my pajamas. How an elephant got into my pajamas I'll never know.

让我们仔细看看短语 I shot an elephant in my pajamas 中的歧义。首先，我们需要定义一个简单的文法：

```
>>> 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'
```

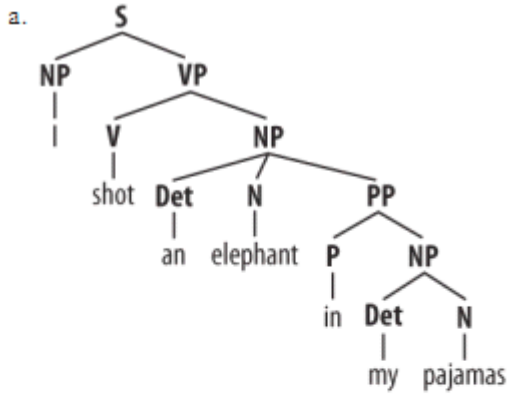
```
... P -> 'in'
... """)
```

这个文法允许以两种方式分析句子，取决于介词短语 in my pajamas 是描述大象还是枪击事件。

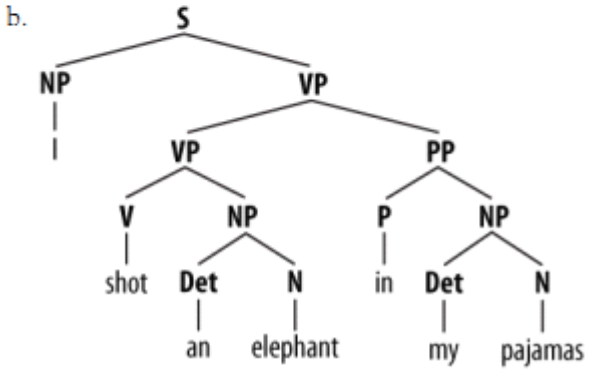
```
>>> sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
>>> parser = nltk.ChartParser(groucho_grammar)
>>> trees = parser.nbest_parse(sent)
>>> for tree in trees:
...     print tree
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant) (PP (P in) (NP (Det my) (N pajamas))))))
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))
```

程序产生两个括号括起的结构，我们可以用树来表示它们，如（3）所示：

(3) a.



b.



请注意，相关的所有词的含义是没有歧义的；例如：词 shot 不会在第一句话中表示“使用枪的动作”，而在第二句中表示“使用相机”。

轮到你来：思考下面的句子，看看你是否能想出两种完全不同的解释：Fighting animals could be dangerous. Visiting relatives can be tiresome. 个别词是否有歧义？如果没有，造成歧义的原因是什么？

本章介绍文法和分析，以形式化可计算的方法调查和建模我们一直在讨论的语言现象。正如我们所看到的，词序列中符合语法规则的和不符合语法规则的模式相对于短语结构和依赖是可以被理解的。我们可以开发使用文法和分析的这些结构的形式化模型。与以前一样，一个重要的动机是自然语言理解。当我们能可靠地识别一个文本所包含的语言结构时，我们从中可以获得多少文本的含义？一个程序通读了一个文本，它能否足够“理解”它，来回答一些简单的关于“发生了什么事”或“谁对谁做了什么”的问题？还像以前一样，我们将开发简单的程序来处理已注释的语料库，并执行有用的任务。

8.2 文法有什么用？

超越 n-grams

在第 2 章我们给出了一个如何使用 bigrams 中的频率信息生成文本的例子，生成短的词序列看上去似乎完全可以接受，但一长就迅速退化成无稽之谈。下面是另一对例子，我们通过计算儿童故事《The Adventures of Buster Brown》（包含在古登堡工程选集语料库中）的文本中的 bigrams：

(4) a. He roared with me the pail slip down his back

b. The worst part and clumsy looking for whoever heard light

你凭感觉能知道，这些序列是“词沙拉”，但你可能会发现很难确定它们错在哪里。学习文法的一个好处是，它提供了一个概念框架和词汇表能拼凑出这些直觉。让我们来仔细看看序列：the worst part and clumsy looking。这看起来像一个**并列结构**，两个短语通过并列连词如 and、but 或 or 连结在一起。下面是连词的语法功能的一个非正式（并且简单）的描述。

并列结构：如果 v1 和 v2 都是文法类型 X 的短语，那么 v1 and v2 也是 X 类型的短语。

这里有几个例子。首先，两个 NP（名词短语）连结在一起组成一个 NP，其次，两个 AP（形容词短语）连结在一起组成一个 AP。

(5) a. The book’s ending was (NP the worst part and the best part) for me.

b. On land they are (AP slow and clumsy looking).

我们不能做的是连结一个 NP 和一个 AP，这就是为什么 the worst part and clumsy looking 不合语法的原因。在我们形式化这些想法之前，我们需要了解**成分结构**的概念。

成分结构基于对词与其他词结合在一起形成单元的观察。一个词序列形成这样一个单元的证据是它是可替代的——也就是说，在一个符合语法规则的句子中的词序列可以被一个更小的序列替代而不会导致句子不符合语法规则。为了澄清这个想法，思考下面的句子：

(6) The little bear saw the fine fat trout in the brook.

事实上，我们可以代替 He 为 The little bear，这表明后者是一个单元。相比之下，我们不能以同样的方式代替 little bear saw。（我们在句子开头加注星号表示它不符合语法规则。）

(7) a. He saw the fine fat trout in the brook.

b. *The he the fine fat trout in the brook.

在图 8-1 中，我们系统地用较短的序列替代较长的序列，并使其依然符合语法规则。事实上，形成一个单元的每个序列可以被一个单独的词替换，我们最终只有两个元素。

the	little	bear	saw	the	fine	fat	trout	in	the	brook
the	bear		saw	the	trout			in	it	
He			saw	it				there		
He			ran					there		
He			ran							

图 8-1. 词序列的替代：从最上面一排开始，我们用单个的词（如：it）替换特别的词序列（如：the brook）；重复这一过程，我们得到符合语法规则的两个词的句子。

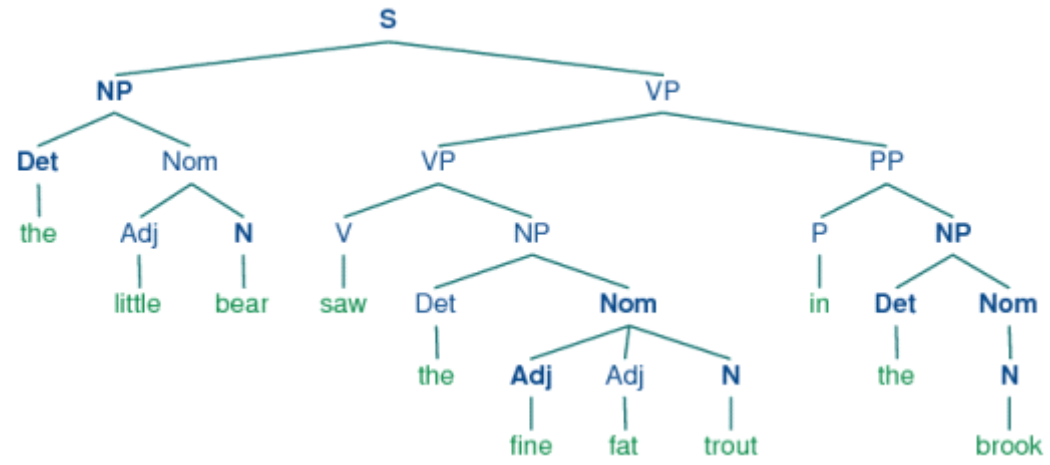
在图 8-2 中，我们为我们在前面的图上看到的词增加了文法类别标签。标签 NP，VP 和 PP 分别表示**名词短语**，**动词短语**和**介词短语**。

Det the	Adj little	N bear	V saw	Det the	Adj fine	Adj fat	N trout	P in	Det the	N brook
Det the	Nom bear		V saw	Det the	Nom trout			P in	NP it	
NP He			V saw	NP it				PP there		
NP He			VP ran					PP there		
NP He			VP ran							

图 8-2. 词序列的替换，附加文法分类：此图再现了图 8-1 并给名词短语（NP），动词短语（VP），介词短语（PP）以及名词性词（Nom）添加了相应的文法分类。

如果我们现在从最上面的行剥离出词汇，增加一个 S 节点，翻转图，最终我们得到一个标准的短语结构树，如（8）中所示。此树的每个节点（包括词）被称为一个**组成部分（成分，constituent）**。S 的**直接组成部分**是 NP 和 VP。

(8)



正如我们在 8.1 节中所看到的，句子可以有任意的长度。因此，短语结构树可以有任意深度。我们在 7.4 节中看到的级联块分析器只能产生有限深度的结构，使得分块方法在这里并不适用。

在下一节，我们将看到一个指定的文法如何将句子细分成它的直接成分，以及如何将这些进一步细分，直到到达单独词汇层面。

8.3 上下文无关文法

一种简单的文法

首先，让我们看一个简单的上下文无关文法（context-free grammar，CFG）。按照惯例，第一条生产式的左端是文法的**开始符号**，通常是 S，所有符合语法规则的树都必须有这个符号作为它们的根标签。NLTK 中，上下文无关文法定义在 nltk.grammar 模块。在例 8-1 中，我们定义了文法，并显示如何分析一个简单的符合文法的句子。

例 8-1. 上下文无关文法的例子。

```
grammar1 = nltk.parse_cfg("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked"
  NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park"
  P -> "in" | "on" | "by" | "with"
  """)
>>> sent = "Mary saw Bob".split()
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> for tree in rd_parser.nbest_parse(sent):
...         print tree
(S (NP Mary) (VP (V saw) (NP Bob)))
```

例 8-1 中的文法包含涉及各种句法类型的产生式，如表 8-1 中所列出的。在这里使用递归下降分析器也可以通过图形界面查看，如图 8-3 所示；我们在 8.4 节中将更详细讨论这个分析器。

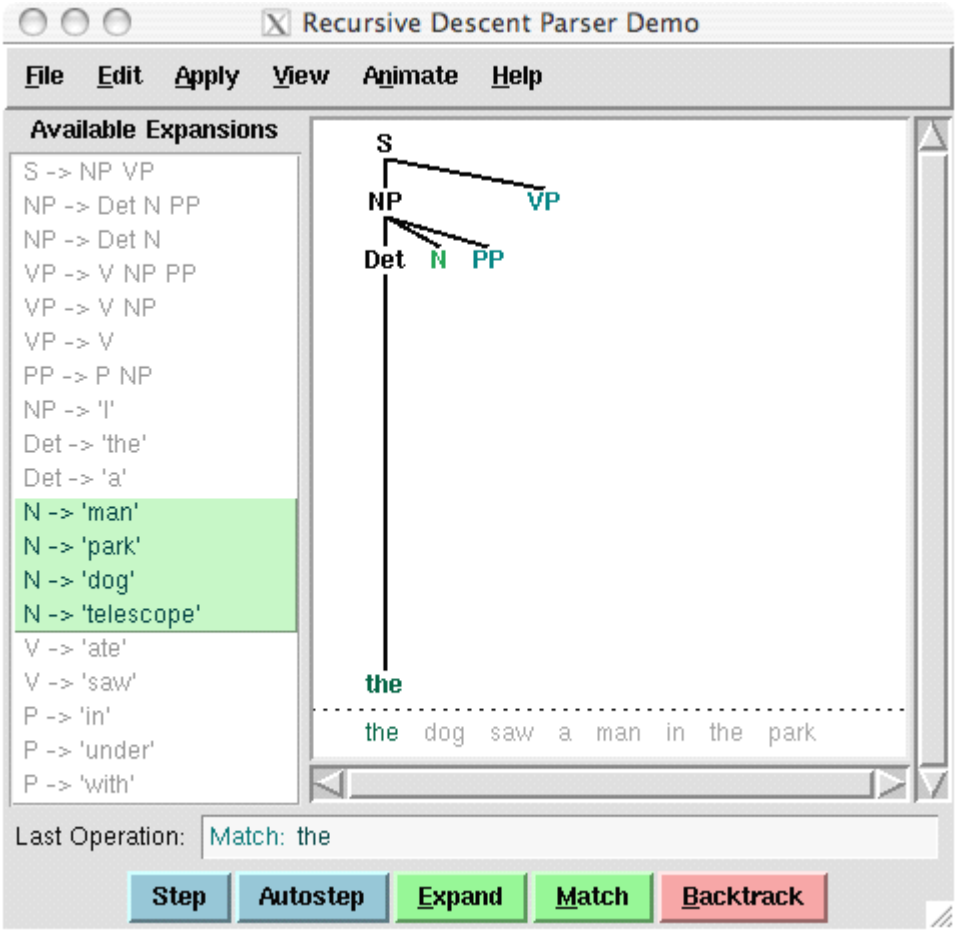


图 8-3. 递归下降分析器演示：在这个递归下降分析器生长出解析树和它匹配输入的词时，此工具可以让你观看它的操作，。

表 8-1. 句法类型

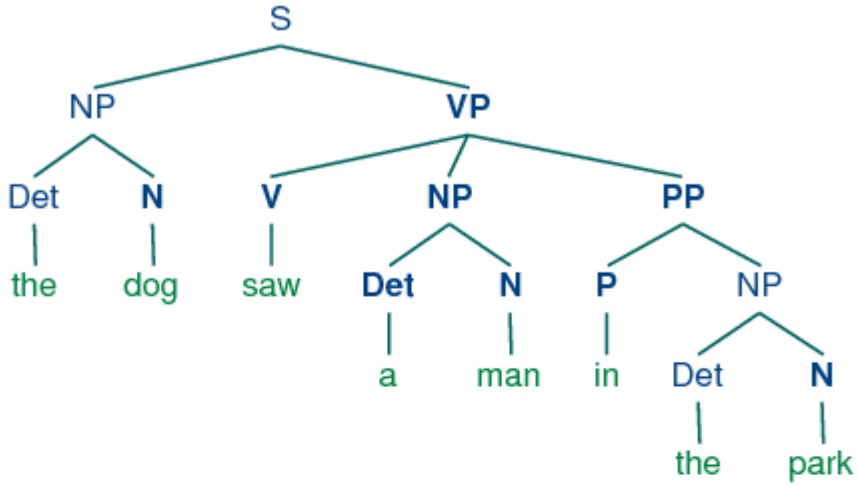
符号	意思	例子
S	句子	*the man walked*
NP	名词短语	*a dog*
VP	动词短语	*saw a park*
PP	介词短语	*with a telescope*
Det	限定词	*the*
N	名词	*dog*
V	动词	*walked*
P	介词	*in*

产生式如 VP -> V NP | V NP PP 右侧脱节了，中间显示|，这是两个产生式 VP -> V NP 和 VP -> V NP PP 的缩写。

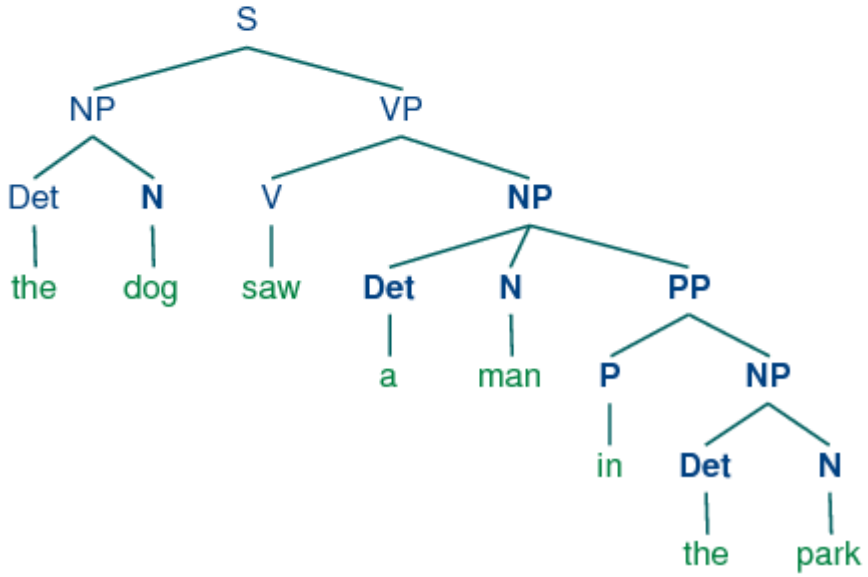
如果我们使用例 8-1 显示的文法分析句子 The dog saw a man in the park，我们会得到两棵树，类似与我们在(3)中看到的：

(9)

a.



b.



由于这句话的两棵树都符合我们的文法规则，这句话被称为结构上有歧义。这个歧义问题被称为介词短语附着歧义，正如我们在本章前面看到的。正如你可能还记得，这是一个附着歧义，因为 PP in the park 需要附着在树中两个位置中的一个：要么是 VP 的孩子要么是 NP 的孩子。当 PP 附着在 VP 上，合适的解释是：看到公园里发生的事情。然而，如果 PP 附着在 NP 上，意思就是：在公园里的一个人，看到（狗）的主语可能已经坐在公寓的阳台上俯瞰公园。

写你自己的文法

如果你有兴趣尝试写上下文无关文法 CFG，你会发现在一个文本文件 mygrammar.cfg 中创建和编辑你的语法会很有帮助。然后，你可以加载它到 NLTK 中，并按如下方式用它进行分析：

```
>>> grammar1 = nltk.data.load('file:mygrammar.cfg')
>>> sent = "Mary saw Bob".split()
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> for tree in rd_parser.nbest_parse(sent):
```

... print tree

确保你的文件名后缀为.cfg，并且字符串'file:mygrammar.cfg'中间没有空格符。如果命令 print tree 没有产生任何输出，这可能是因为你的句子 sent 并不符号你的文法。遇到这种情况可以将分析器的跟踪设置打开 rd_parser = nltk.RecursiveDescentParser(grammar1, trace=2)。你还可以查看当前使用的文法中的产生式，使用命令 for p in grammar1.productions(): print p。

当你写 CFG 在 NLTK 中分析时，你不能将文法类型与词汇项目一起写在同一个产生式的右侧。因此，产生式 PP -> 'of' NP 是不允许的。另外，你不得在产生式右侧仿制多个词的词汇项。因此，不能写成 NP -> 'New York'，而要写成类似 NP -> 'New_York'这样的。

句法结构中的递归

一个文法被认为是递归的，如果文法类型出现在产生式左侧也出现在右侧，如例 8-2 所示。产生式 Nom -> Adj Nom（其中 Nom 是名词性的类别）包含 Nom 类型的直接递归，而 S 上的间接递归来自于两个产生式的组合：S -> NP VP 与 VP -> V S。

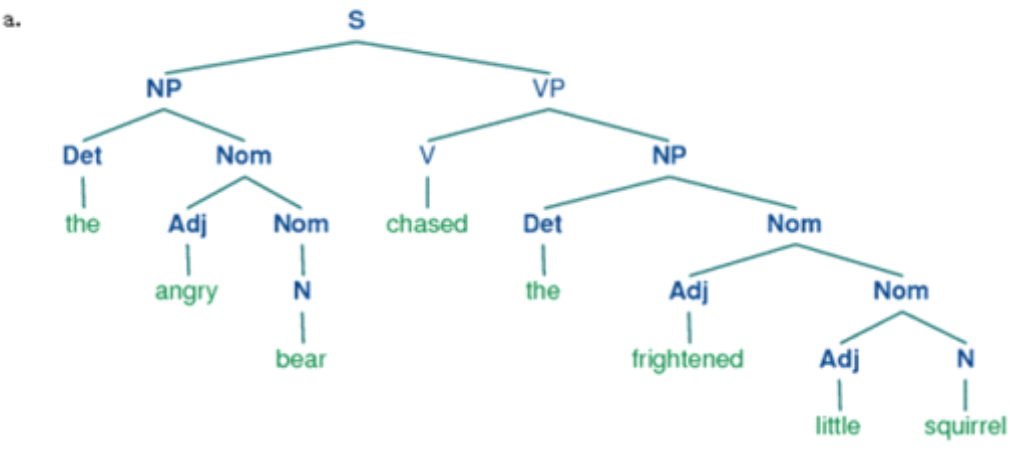
例 8-2. 递归的上下文无关文法。

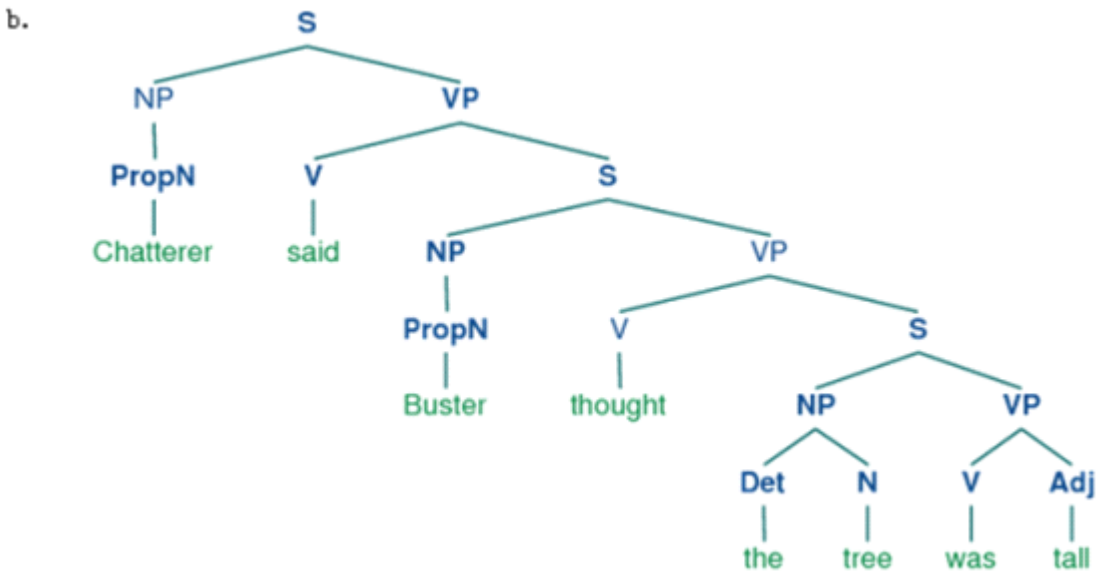
```
grammar2 = nltk.parse_cfg("""
  S  -> NP VP
  NP -> Det Nom | PropN
  Nom -> Adj Nom | N
  VP -> V Adj | V NP | V S | V NP PP
  PP -> P NP
  PropN -> 'Buster' | 'Chatterer' | 'Joe'
  Det -> 'the' | 'a'
  N -> 'bear' | 'squirrel' | 'tree' | 'fish' | 'log'
  Adj  -> 'angry' | 'frightened' |  'little' | 'tall'
  V ->  'chased'  | 'saw' | 'said' | 'thought' | 'was' | 'put'
  P -> 'on'
  """)
```

要看递归如何从这个语法产生，思考下面的树。（10a）包括嵌套的名词短语，而（10b）包含嵌套的句子。

(10)

a.





我们在这里只演示了两个层次的递归，递归深度是没有限制的。你可以尝试分析更深层次嵌套结构的句子。请注意，RecursiveDescentParser 是无法处理形如 X -> X Y 的**左递归**产生式；我们将在第 8.4 节谈及这个。

8.4 上下文无关文法分析

分析器根据文法产生式处理输入的句子，并建立一个或多个符合文法的组成结构。文法是一个格式良好的声明规范——它实际上只是一个字符串，而不是程序。分析器是文法的解释程序。它搜索符合文法的所有树的空间找出一棵边缘有所需句子的树。

分析器允许使用一组测试句子评估一个文法，帮助语言学家发现在他们的文法分析中存在的错误。分析器可以作为心理语言处理模型，帮助解释人类处理某些句法结构的困难。许多自然语言应用程序都在某种程度涉及文法分析；例如：我们会期望自然语言问答系统对提交的问题首先进行文法分析。

在本节中，我们将看到两个简单的分析算法，一种自上而下的方法称为递归下降分析，一种自下而上的方法称为移进-归约分析。我们也将看到一些更复杂的算法，一种带自下而上过滤的自上而下的方法称为左角落分析；一种动态规划技术称为图表分析。

递归下降分析

一种最简单的分析器将一个文法作为如何将一个高层次的目标分解成几个低层次的子目标的规范来解释。顶层的目标是找到一个 S。S→NP VP 生产式允许分析器替换这个目标为两个子目标：找到一个 NP，然后找到一个 VP。每个这些子目标都可以再次被子目标的子目标替代，使用左侧有 NP 和 VP 的产生式。最终，这种扩张过程达到子目标，如：找到词 telescope。这样的子目标可以直接与输入序列比较，如果下一个单词匹配就取得成功。如果没有匹配，分析器必须备份，并尝试其它选择。

递归下降分析器在上述过程中建立分析树。带着最初的目标（找到一个 S），创建 S 根节点。随着上述过程使用文法的产生式递归扩展，分析树不断向下延伸（故名为递归下降）。

我们可以在图形化示范 nltk.app.rdparser()中看到这个过程。执行此分析器的六个阶段，如图 8-4 所示。

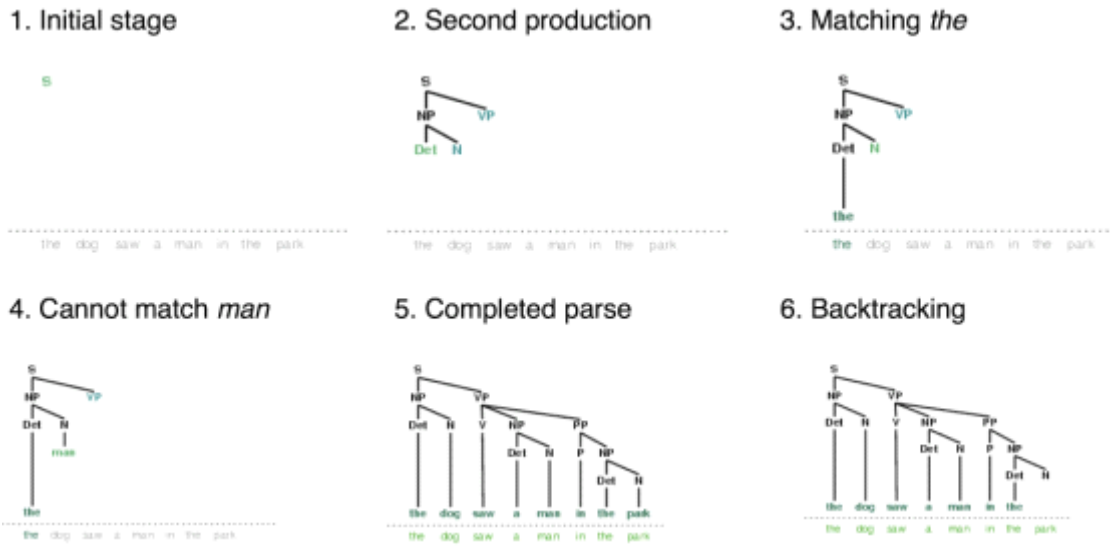


图 8-4. 递归下降分析器的六个阶段：分析器以一颗包含节点 S 的树开始；每个阶段它会查询文法来找到一个可以用于扩大树的产生式；当遇到一个词汇产生式时，将它的词与输入比较；发现一个完整的分析树后，分析器会回溯寻找更多的分析树。

在这个过程中，分析器往往被迫在多种可能的产生式中选择。例如：从第 3 步到第 4 步，它试图找到左侧有 N 的产生式。第一个是 N → man。当这不起作用时就回溯，按顺序尝试其他左侧有 N 的产生式，直到它得到 N → dog，与输入句子中的下一个词相匹配。一段时间以后，如第 5 步所示，它发现了一个完整的分析树。这是一个涵盖了整个句子的树，没有任何悬着的边。发现了分析树后我们可以让分析器寻找其它额外的分析树。它会再次回溯和探索选择其它产生式，以免漏掉任何一个产生分析树的情况。

NLTK 提供了一个递归下降分析器：

```
>>> rd_parser = nltk.RecursiveDescentParser(grammar1)
>>> 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))))
```

RecursiveDescentParser()接受一个可选的参数 trace。如果 trace 大于零，则分析器将报告它解析一个文本的步骤。

递归下降分析有三个主要的缺点。首先，左递归产生式，如：NP -> NP PP，会进入死循环。第二，分析器浪费了很多时间处理不符合输入句子的词和结构。第三，回溯过程中可能会丢弃分析过的成分，它们将需要在之后再次重建。例如：从 VP -> V NP 上回溯将放弃为 NP 创建的子树。如果分析器之后处理 VP -> V NP PP，那么 NP 子树必须重新创建。

递归下降分析是一种**自上而下分析**。自上而下分析器在检查输入之前先使用文法预测输入将是什么！然而，由于输入对分析器一直是可用的，从一开始就考虑输入的句子会是更明智的做法。这种方法被称为**自下而上分析**，在下一节中我们将看到一个例子。

移进-归约分析

一种简单的自下而上分析器是**移进-归约分析器**。与所有自下而上的分析器一样，移进-归约分析器尝试找到对应文法生产式右侧的词和短语的序列，用左侧的替换它们，直到整个句子归约为一个 S。

移位-规约分析器反复将下一个输入词推到堆栈（见 4.1 节），这是**移位**操作。如果堆栈上的前 n 项，匹配一些产生式的右侧的 n 个项目，那么就把它们弹出栈，并把产生式左边的项目压入栈。这种替换前 n 项为一项的操作就是**规约**操作。此操作只适用于堆栈的顶部；规约栈中的项目必须在后面的项目被压入栈之前做。当所有的输入都使用过，堆栈中只剩余一个项目，也就是一颗分析树作为它的根的 S 节点时，分析器完成。移位-规约分析器通过上述过程建立一颗分析树。每次弹出堆栈 n 个项目，它就将它们组合成部分的分析树，然后将这压回推栈。我们可以使用图形化示范 nltk.app.srparser()看到移位-规约分析算法步骤。执行此分析器的六个阶段如图 8-5 所示。

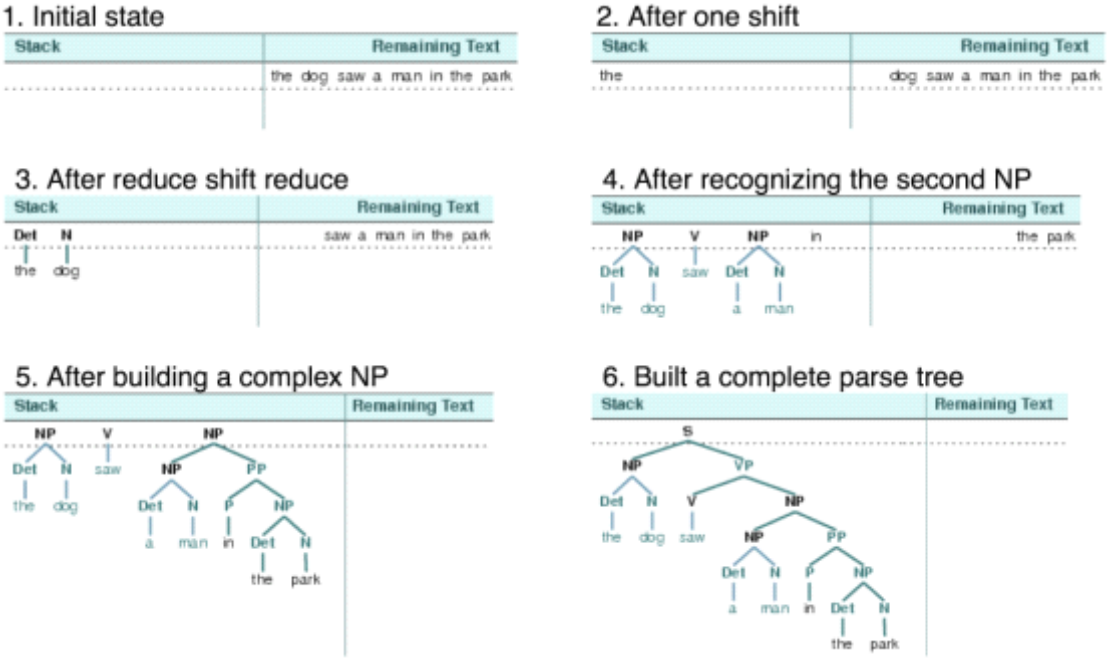


图 8-7. 移进-归约分析器的六个阶段：分析器一开始把输入的第一个词转移到堆栈；一旦堆栈顶端的项目与一个文法产生式的右侧匹配，就可以将它们用那个产生式的左侧替换；当所有输入都被使用过且堆栈中只有剩余一个项目 S 时，分析成功。

NLTK 中提供了 ShiftReduceParser()，移进-归约分析器的一个简单的实现。这个分析器不执行任何回溯，所以它不能保证一定能找到一个文本的解析，即使确实存在一个这样的解析。此外，它最多只会找到一个解析，即使有多个解析存在。我们可以提供一个可选的 trace 参数，控制分析器报告它分析一个文本的步骤的繁琐程度。

```
>>> sr_parse = nltk.ShiftReduceParser(grammar1)
>>> sent = 'Mary saw a dog'.split()
>>> print sr_parse.parse(sent)
  (S (NP Mary) (VP (V saw) (NP (Det a) (N dog))))
```

轮到你来：以跟踪模式运行上述分析器，看看序列的移进和规约操作，使用 sr_parse = nltk.ShiftReduceParser(grammar1, trace=2)。

移进-规约分析器可能会到达一个死胡同，而不能找到任何解析，即使输入的句子是符合语法的。这种情况发生时，没有剩余的输入，而堆栈包含不能被规约到一个 S 的项目。问题出现的原因是：较早前做出的选择不能被分析器撤销（虽然图形演示中用户可以撤消它们的选择）。分析器可以做两种选择：（a）当有多种规约可能时选择哪个规约，（b）当移进和规约都可以时选择哪个动作。

移进-规约分析器可以改进执行策略来解决这些冲突。例如：它可以通过只有在不能规约时才移进，解决移进-规约冲突；它可以通过优先执行规约操作，解决规约-规约冲突；它可以从堆栈移除更多的项目。（一个通用的移进-规约分析器，是一个“超前 LR 分析器”，普遍使用在编程语言编译器中。）

移进-规约分析器相比递归下降分析器的好处是，它们只建立与输入中的词对应的结构。此外，每个结构它们只建立一次。例如：NP(Det(the), N(man))只建立和压入栈一次，不管以后 VP -> V NP PP 规约或者 NP -> NP PP 规约会不会用到。

左角落分析器

递归下降分析器的问题之一是当它遇到一个左递归产生式时，会进入无限循环。这是因为它盲目应用文法产生式而不考虑实际输入的句子。左角落分析器是我们已经看到的自下而上与自上而下方法的混合体。

左角落分析器是一个带自下而上过滤的自上而下的分析器。不像普通的递归下降分析器，它不会陷入左递归产生式的陷井。在开始工作之前，左角落分析器预处理上下文无关文法建立一个表，其中每行包含两个单元，第一个存放非终结符，第二个存放那个非终结符可能的左角落的集合。表 8-2 用 grammar2 的文法演示了这一点。

表 8-2. grammar2 的左角落

类型	左角落（非终结符）
S	NP
NP	Det，PropN
VP	V
PP	P

分析器每次考虑产生式时，它会检查下一个输入词是否与左角落表格中至少一种非终结符的类别相容。

符合语句规则的子串表

上面讨论的简单的分析器在完整性和效率上都有限制。为了弥补这些，我们将运用**动态规划**算法设计技术分析问题。正如我们在 4.7 节中所看到的，动态规划存储中间结果，并在适当的时候重用它们，能显著提高效率。

这种技术可以应用到句法分析，使我们能够存储分析任务的部分解决方案，然后在必要的时候查找它们，直到达到最终解决方案。这种分析方法被称为**图表分析**。我们在本节中介绍它的主要思想；更多的实施细节请看网上提供的本章的材料。

动态规划使我们能够只建立一次 PP in my pajamas。第一次我们建立时就把它存入一个表格中，然后在我们需要作为对象 NP 或更高的 VP 的组成部分用到它时我们就查找表格。这个表格被称为**符合语法规则的子串表**或简称为 WFST。（术语“串”指一个句子中的连续

的词序列。）我们将展示如何自下而上的构建 WFST，以便系统地记录已经找到的句法成分。

让我们设置我们的输入为句子（2）。指定 WFST 的跨度的数值让人联想起 Python 的分片符号（3.2 节）。这种数据结构的另一种方式如图 8-6 所示，一个称为**图表**的数据结构。

图 8-6. 图表数据结构：词是线性图结构的边上的标签。

在 WFST 中，我们通过填充三角矩阵中的单元记录词的位置：纵轴表示一个子串的起始位置，而横轴表示结束位置（从而 shot 将出现在坐标(1, 2)的单元中）。为了简化这个演示，我们将假定每个词有一个独特的词汇类别，我们将在矩阵中存储词汇类别（不是词）。所以单元(1, 2)将包含条目 V。更一般的，如果我们输入的字符串是 $a_1 a_2 \ldots a_n$，我们的文法包含一个形为 $A \rightarrow a_i$ 的产生式，那么我们把 A 添加到单元(i-1, i)。

所以，对于 text 中的每个词，我们可以在我们的文法中查找它所属的类别。

```
>>> text = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
[V -> 'shot']
```

对于我们的 WFST，我们用 Python 中的链表的链表创建一个(n-1)×(n-1)的矩阵，在例 8-3 中的函数 init_wfst()中用每个标识符的词汇类型初始化它。我们还定义一个实用的函数 display()来为我们精美的输出 WFST。正如预期的那样，V 在(1, 2)单元中。

例 8-3. 使用符合语句规则的子串表的接收器。

```
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    .   .     .    .    .     .     N
>>> wfst1 = complete_wfst(wfst0, tokens, groucho_grammar)
>>> display(wfst1, tokens)
WFST 1   2     3    4    5     6     7
0    NP  .     .    S    .     .     S
1    .   V     .    VP   .     .     VP
2    .   .     Det  NP   .     .     .
3    .   .     .    N    .     .     .
4    .   .     .    .    P     .     PP
5    .   .     .    .    .     Det   NP
6    .   .     .    .    .     .     N
```

回到我们的表格表示，假设对于词 an 我们有 Det 在(2, 3)单元，对以词 elephant 有 N 在(3, 4)单元，对于 an elephant 我们应该在(2, 4)放入什么？我们需要找到一个形如 NP → Det N 的产生式。查询了文法，我们知道我们可以输入(0, 2)单元的 NP。

更一般的，我们可以在(i, j)输入 A，如果有一个产生式 A → B C，并且我们在(i, k)中找到非终结符 B，在(k, j)中找到非终结符 C。例 8-3 中的程序使用此规则完成 WFST。通过调用函数 complete_wfst()时设置 trace 为 True，我们看到了显示 WFST 正在被创建的跟踪输出：

```
>>> 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] ==> [1]   VP  [7]
[0]   NP  [1]   VP  [7] ==> [0]    S  [7]
```

例如：由于我们在 wfst[0][1]找到 Det，在 wfst[1][2]找到 N，我们可以添加 NP 到 wfst[0][2]。

为了帮助我们更简便地通过产生式的右侧检索产生式，我们为文法创建一个索引。这是空间-时间权衡的一个例子：我们对语法做反向查找，每次我们想要通过右侧查找产生式时不必遍历整个产生式链表。

我们得出结论：只要我们已经在(0, 7)单元构建了一个 S 节点，表明我们已经找到了一个涵盖了整个输入的句子，我们就为整个输入字符串找到了一个解析。最后的 WFST 状态如图 8-7 所示。

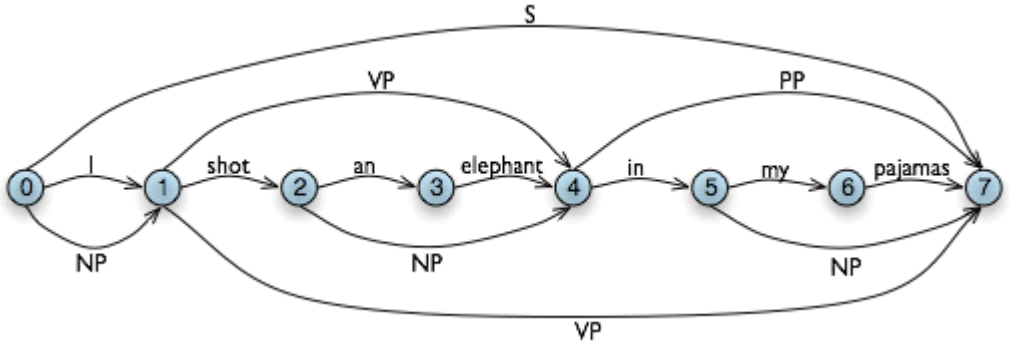


图 8-7. 图数据结构：图中额外的边表示非终结符。

请注意，在这里我们没有使用任何内置的分析函数。我们已经从零开始实现了一个完整的初级图表分析器！

WFST 有几个缺点。首先，正如你可以看到的，WFST 本身不是一个分析树，所以该技术严格地说是**认识**到一个句子被一个文法承认，而不是分析它。其次，它要求每个非词汇文法生产式是二元的。虽然可以将任意的 CFG 转换为这种形式，我们宁愿使用这种方法时没有这样的规定。第三，作为一个自下而上的方法，它潜在的存在浪费，它会在不符合文法的地方提出成分。

最后，WFST 并不能表示句子中的结构歧义（如两个动词短语的读取）。(2, 8)单元中的 VP 实际上被输入了两次，一次是读取 V NP，一次是读取 VP PP。这是不同的假设，第二个会覆盖第一个（虽然如此，这并不重要，因为左侧是相同的。）图表分析器使用稍微更丰富的数据结构和一些有趣的算法来解决这些问题（见 8.8 节）。

轮到你来：尝试交互式图表分析器应用程序 nltk.app.chartparser()。

8.5 依存关系和依存文法

短语结构文法是关于词和词序列如何结合起来形成句子成分的。一个独特的和互补的方式，**依存文法**，集中关注的是词与其他词之间的关系。依存关系是一个中心词与它的依赖之间的二元对称关系。一个句子的中心词通常是动词，所有其他词要么依赖于**中心词**，要么**依赖**路径与它联通。

依存关系表示是一个加标签的有向图，其中节点是词汇项，加标签的弧表示依赖关系，从中心词到依赖。图 8-8 显示了一个依存关系图，箭头从中心词指出它们的依赖。

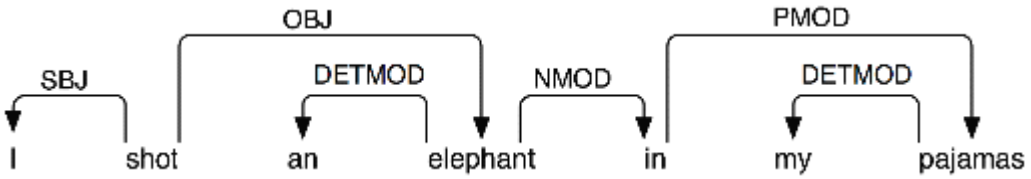


图 8-8. 依存结构：箭头从中心词指向它们的依赖；标签表示依赖的语法功能如：主语、宾语或修饰语。

图 8-8 中的弧加了依赖与它的中心词之间的语法功能标签。例如：I 是 shot（这是整个句子的中心词）的 SBJ（主语），in 是一个 NMOD（elephant 的名词修饰语）。与短语结构文法相比，依存文法可以作为一种依存关系直接用来表示语法功能。

下面是 NLTK 为依存文法编码的一种方式——注意它只能捕捉依存关系信息，不能指定依存关系类型：

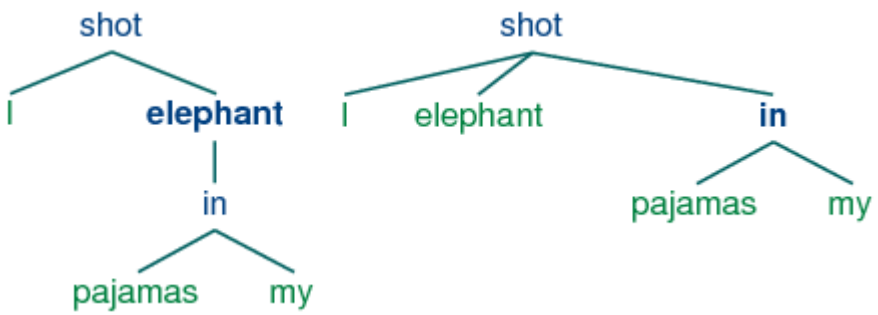
```
>>> groucho_dep_grammar = nltk.parse_dependency_grammar("""
... 'shot' -> 'I' | 'elephant' | 'in'
... 'elephant' -> 'an' | 'in'
... 'in' -> 'pajamas'
... 'pajamas' -> 'my'
... """)
>>> print groucho_dep_grammar
Dependency grammar with 7 productions
  'shot' -> 'I'
  'shot' -> 'elephant'
  'shot' -> 'in'
  'elephant' -> 'an'
  'elephant' -> 'in'
  'in' -> 'pajamas'
  'pajamas' -> 'my'
```

依存关系图是一个**投影**，当所有的词都按线性顺序书写，边可以在词上绘制而不会交叉。这等于是说一个词及其所有后代依赖（依赖及其依赖的依赖，等等）在句子中形成一个连续的词序列。图 8-8 是一个投影，我们可以使用投影依存关系分析器分析很多英语句子。下面的例子演示 groucho_dep_grammar 如何提供了一种替代的方法来捕捉附着歧义，我们之前在研究短语结构语法中遇到的。

```
>>> pdp = nltk.ProjectiveDependencyParser(groucho_dep_grammar)
>>> sent = 'I shot an elephant in my pajamas'.split()
>>> trees = pdp.parse(sent)
>>> for tree in trees:
...     print tree
(shot I (elephant an (in (pajamas my))))
(shot I (elephant an) (in (pajamas my)))
```

这些括号括起来的依存关系结构也可以显示为树，依赖被作为它们的中心词的孩子。

(14)



在比英语具有更多灵活的词序的语言中，非投影的依存关系也更常见。

在一个成分 C 中决定哪个是中心词 H，哪个是依赖 D，已经提出了很多不同的标准。最重要的有以下几种：

1. H 决定类型 C 的分布；或者，C 的外部句法属性取决于 H。
2. H 定义 C 的语义类型。
3. H 必须有而 D 是可选的。
4. H 选择 D 并且决定它是必须有的还是可选的。
5. D 的形态由 H 决定（如 agreement 或 case government）。

当我们在一个短语结构文法中说：一个 PP 的直接成分是 P 和 NP 时，我们隐含了中心词/依赖之间的区别。介词短语是一个短语，它的中心词是一个介词。此外，NP 是 P 的依赖。同样的区别在我们已经讨论过的其它类型的短语结构文法中也存在。这里要注意的关键点是：虽然短语结构文法看上去似乎与依存关系文法非常不同，它们隐含着依存关系。虽然 CFG 不会直接捕获依存关系，最近的语言框架已越来越多地采用这两种方法的结合的形式。

配价与词汇

让我们来仔细看看动词和它们的依赖。例 8-2 中的文法正确生成了类似（12）的例子。

(12) a. The squirrel was frightened.
b. Chatterer saw the bear.
c. Chatterer thought Buster was angry.
d. Joe put the fish on the log.

这些句子对应表 8-3 中的产生式。

表 8-3. VP 产生式和它们的中心词汇

产生式	中心词汇
VP -> V Adj	was
VP -> V NP	saw
VP -> V S	thought
VP -> V NP PP	put

也就是说，was 可以与跟在其后的形容词一起出现，saw 可以与跟在其后的 NP 一起出现，thought 可以与跟在其后的 S 一起出现，thought 可以与跟在其后的 NP 和 PP 一起出现。依赖 ADJ、NP、PP 和 S 通常被称为各自动词的**补语**，什么动词可以和什么补语一起出现具有很强的约束。对比（12），（13）中的词序列是不符合语法规则的：

(13) a. *The squirrel was Buster was angry.
b. *Chatterer saw frightened.
c. *Chatterer thought the bear.

d. *Joe put on the log.

稍微用点儿想象力就可以设计一个上下文，在那里动词和补语的不寻常的组合可以解释的通。然而，我们假设（13）中的例子在中性语境是可以解释的。

在依存文法的传统中，在表 8-3 中的动词被认为具有不同的**配价**。配价限制不仅适用于动词，也适用于其他类的中心词。

基于短语结构文法的框架内，已经提出了各种技术排除（13）中不合语法的例子。在 CFG 中，我们需要一些限制文法产生式的方式，使得扩展了 VP 后动词只与它们正确的补语一同出现。我们可以通过将动词划分成更多“子类别”做到这个，每个子类别与一组不同的补语关联。例如：**及物动词**，如：chased 和 saw，需要后面跟 NP 对象补语；它们是 NP 大类别的**子类别**。如果我们为及物动词引入一个新的类别标签，叫做 TV（transitive verb 的缩写），那么我们可以在下面的产生式中使用它：

```
VP -> TV NP
TV -> 'chased' | 'saw'
```

现在*Joe thought the bear 被排除了，因为我们没有列出 thought 是一个 TV，但 Chatterer saw the bear 仍然是允许的。表 8-4 提供了更多动词子类别标签的例子。

表 8-4. 动词子类别

符号	含义	例子
IV	Intransitive verb	barked
TV	Transitive verb	saw a man
DatV	Dative verb	gave a dog to a man
SV	Sentential verb	said that a dog barked

配价是一个词项的属性，我们将在第 9 章进一步讨论它。补语往往与修饰语对照，虽然两者都是依存关系的类别。

介词短语、形容词和副词通常充当修饰语。与补充不同修饰语是可选的，经常可以进行迭代，不会像补语那样被中心词选择。例如：副词 really 在所有（14）中的句子中都可以被添加为修饰语：

(14) a. The squirrel really was frightened.
b. Chatterer really saw the bear.
c. Chatterer really thought Buster was angry.
d. Joe really put the fish on the log.

PP 附着的结构歧义，我们已经在短语结构和依存文法中说过了，语义上对应修饰语的范围上的模糊。

扩大规模

到目前为止，我们只考虑了“玩具文法”，演示分析的关键环节的少量的文法。但有一个明显的问题就是这种做法是否可以扩大到覆盖自然语言的大型语料库。手工构建这样的一套产生式有多么困难？一般情况下，答案是：非常困难。即使我们允许自己使用各种形式化的工具，它们可以提供文法产生式更简洁的表示。保持对覆盖一种语言的主要成分所需要的众多产生式之间的复杂的相互作用的控制，仍然是极其困难的。换句话说，很难将文法模块化，每部分文法可以独立开发。反过来这意味着，在一个语言学家团队中分配编写文法的任务是很困难的。另一个困难是当文法扩展到包括更加广泛的成分时，适用于任何一个句子的

分析的数量也相应增加。换句话说，歧义随着覆盖而增加。

尽管存在这些问题，一些大的合作项目在为几种语言开发基于规则的文法上已取得了积极的和令人印象深刻的结果。例如：词汇功能语法（Lexical Functional Grammar，LFG）Pargram 项目、中心词驱动短语结构文法（Head-Driven Phrase Structure Grammar，HPSG）LinGO 矩阵框架、词汇化树邻接文法 XTAG 项目。

8.6 文法开发

分析器根据短语结构文法在句子上建立树。现在，我们上面给出的所有例子只涉及玩具文法包含少数的产生式。如果我们尝试扩大这种方法的规模来处理现实的语言语料库会发生什么？在本节中，我们将看到如何访问树库，并看看开发广泛覆盖的文法的挑战。

树库和文法

corpus 模块定义了树库语料的阅读器，其中包含了宾州树库语料的 10%的样本。

```
>>> from nltk.corpus import treebank
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> print t
(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))))
  (. .))
```

我们可以利用这些数据来帮助开发一个文法。例如：例 8-4 中的程序使用一个简单的过滤器找出带句子补语的动词。假设我们已经有一个形如 VP -> SV S 的产生式，这个信息使我们能够识别那些包括在 SV 的扩张中的特别的动词。

例 8-4. 搜索树库找出句子的补语。

```
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)
>>> from nltk.corpus import treebank
```

```
>>> [subtree for tree in treebank.parsed_sents()
...             for subtree in tree.subtrees(filter)]
 [Tree('VP', [Tree('VBN', ['named']), Tree('S', [Tree('NP-SBJ', ...]), ...]), ...]
```

PP 附着语料库，nltk.corpus.ppattach，是另一个有关特别动词配价的信息源。在这里，我们演示挖掘这个语料库的技术。它找出具有固定的介词和名词的介词短语对，其中介词短语附着到 VP 还是 NP，由选择的动词决定。

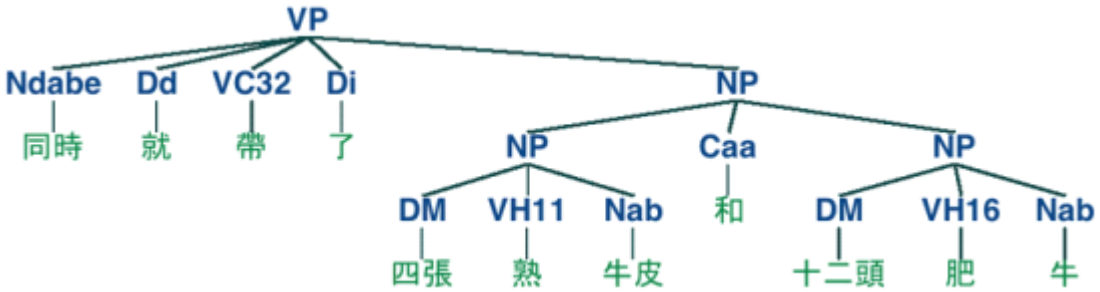
```
>>> entries = nltk.corpus.ppattach.attachments('training')
>>> table = nltk.defaultdict(lambda: nltk.defaultdict(set))
>>> for entry in entries:
...     key = entry.noun1 + '-' + entry.prep + '-' + entry.noun2
...     table[key][entry.attachment].add(entry.verb)
...
>>> for key in sorted(table):
...     if len(table[key]) > 1:
...         print key, 'N:', sorted(table[key]['N']), 'V:', sorted(table[key]['V'])
```

这个程序的输出行中我们发现 offer-from-group N: ['rejected'] V:['received']，这表示 received 期望一个单独的 PP 附着到 VP 而 rejected 不是的。和以前一样，我们可以使用此信息来帮助构建文法。

NLTK 语料库收集了来自 PE08 跨框架跨领域分析器评估共享任务的数据。一个更大的文法集合已准备好用于比较不同的分析器，它可以通过下载 large_grammars 包获得（如：python -m nltk.downloader large_grammars）。

NLTK 语料库也收集了中央研究院树库语料，包括 10000 句已分析的句子，来自现代汉语中央研究院平衡语料库。让我们加载并显示这个语料库中的一棵树。

```
>>> nltk.corpus.sinica_treebank.parsed_sents()[3450].draw()
```



有害的歧义

不幸的是，随着文法覆盖范围的增加和输入句子长度的增长，分析树的数量也迅速增长。事实上，它以天文数字的速度增长。

让我们在一个简单的例子帮助下来探讨这个问题。词 fish 既是名词又是动词。我们可以造这样的句子：fish fish fish，意思是 fish like to fish for other fish。（用 police 尝试一下，如果你喜欢更有意思的东西。）下面是“fish”句子的玩具文法。

```
>>> grammar = nltk.parse_cfg("""
... S -> NP V NP
... NP -> NP Sbar
... Sbar -> NP V
```

```
... NP -> 'fish'
... V -> 'fish'
... """)
```

现在，我们可以尝试分析一个较长的句子，fish fish fish fish fish，其中一个意思是：“fish that other fish fish are in the habit of fishing fish themselves。”我们使用 NLTK 的图表分析器，它在本章前面介绍过。这句话有两种读法。

```
>>> tokens = ["fish"] * 5
>>> cp = nltk.ChartParser(grammar)
>>> 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))))
```

随着句子长度增加到（3，5，7，...），我们得到的分析树的数量是：1; 2; 5; 14; 42; 132; 429; 1,430; 4,862; 16,796; 58,786; 208,012; ….（这是 Catalan 数，我们在第 4 章的练习中见过。）最后一个是句子长度为 23 的分析树的数目，这是宾州树库 WSJ 部分的句子的平均长度。对于一个长度为 50 的句子有超过 10^{12} 的解析，这只是 Piglet 句子长度的一般（8.1 节），这些句子小孩可以毫不费力的处理。没有实际的自然语言处理系统可以为一个句子构建数以百万计的树，并根据上下文选择一个合适的。很显然，人也不会这样做！

请注意，这个问题不是只在我们选择的例子中存在。(Church & Patil, 1982)指出 PP 附着句法歧义在像（15）这样的句子中也是按 Catalan 数的比例增长。

(15) Put the block in the box on the table.

这么多的结构歧义；词汇歧义有如何？只要我们试图建立一个广泛覆盖的文法，我们就被迫使词汇条目对它们的词性高度含糊。在玩具文法中，a 只是一个限定词，dog 只是一个名词，而 runs 只是一个动词。然而，在覆盖广泛的文法中，a 也是一个名词（如：part a），dog 也是一个动词（意思是密切跟踪），而 runs 也是一个名词（如：ski runs）。事实上，所有的词都可以作为名字被提及：例如：he verb 'ate' is spelled with three letters；在讲话中，我们不需要使用引号。此外，大多数名词都可以动词化。因此，一个覆盖广泛的文法分析器将对歧义不堪重负。即使是完全的胡言乱语往往也是符合语法的，例如：the a are of I。正如 (Abney, 1996)已经指出的，这不是一个“词沙拉”而是一个符号语法的名词短语，其中 are 是个名词，意思是一公顷（或 100 平方米）的百分之一，而 a 和 I 是名词指定坐标，如图 8-9 所示。

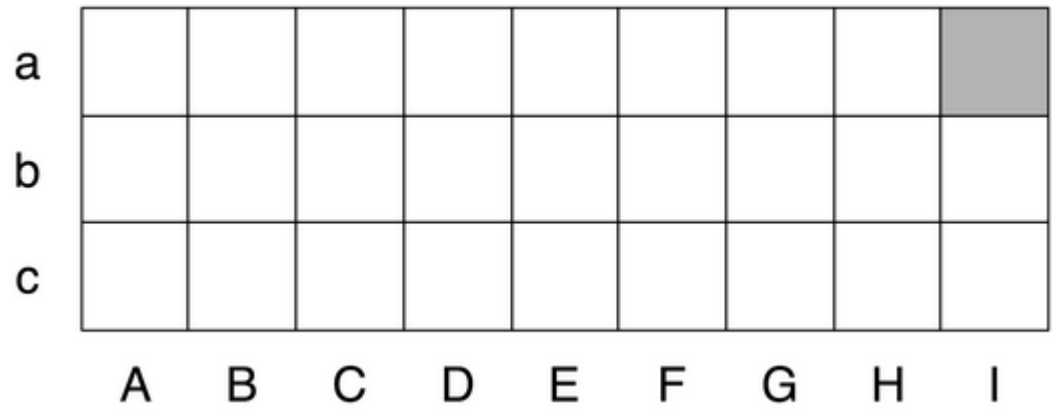


图 8-9. The a are of I：27 个围场的示意图，每一个的大小是一个 are，并用坐标确定；左上角的单元格是 a are of column A（after Abney）

即使这句话是不可能的，它仍是符合语法的，广泛覆盖的分析器应该能够为它建造一个分析树。同样，看上去没有歧义的句子，例如：John saw Mary，结果却有我们没有想到的其它的读法（正如 Abney 解释的）。这种歧义是不可避免的，导致在分析看似平淡无奇的句子时可怕的低效。概率分析提供了解决这些问题的方法，它使我们能够以来自语料库的证据为基础对歧义句的解析进行排名。

加权文法

正如我们刚才看到的，处理歧义是开发广泛覆盖的分析器的主要挑战。图表分析器提高了计算一个句子的多个分析的效率，但它们仍然因可能的分析的数量过多而不堪重负。加权文法和概率分析算法为这些问题提供了一个有效的解决方案。

在讲这些之前，我们需要理解为什么符合语法的概念可能是有倾向性的。思考动词 give。这个动词既需要一个直接宾语（被给予的东西）也需要一个间接宾语（收件人）。这些补语可以按任何顺序出现，如（16）所示。在“介词格”的形式（16a）中，直接宾语先出现，然后是包含间接宾语单独介词短语。

(16) a. Kim gave a bone to the dog.

b. Kim gave the dog a bone.

在“双宾语”的形式（16b）中，间接宾语先出现，然后是直接宾语。在这种情况下，两种顺序都是可以接受的。然而，如果间接宾语是代词，人们强烈偏好双宾语结构：

(17) a. Kim gives the heebie-jeebies to me (prepositional dative).

b. Kim gives me the heebie-jeebies (double object).

使用宾州树库样本，我们可以检查包含 give 的所有介词格和双宾语结构的实例，如例 8-5 所示。

例 8-5. 宾州树库样本中 give 和 gave 的用法。

```
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

我们可以观察到一种强烈的倾向就是最短的补语最先出现。然而，这并没有解释形如：give NP: federal judges / NP: a raise，其中生命度起了重要作用。事实上，根据(Bresnan & Hay, 2008)的调查，存在大量的影响因素。这些偏好可以用加权文法来表示。

概率上下文无关文法（probabilistic context-free grammar，PCFG）是一种上下文无关文法，它的每一个产生式关联一个概率。它会产生与相应的上下文无关文法相同的文本解析，并给每个解析分配一个概率。PCFG 产生的一个解析的概率仅仅是它用到的产生式的概率的乘积。

最简单的方法定义一个 PCFG 是从一个加权产生式序列组成的特殊格式的字符串加载它，其中权值出现在括号里，如例 8-6 所示。

例 8-6. 定义一个概率上下文无关文法（PCFG）。

```
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]
```

有时可以很方便的将多个产生式组合成一行，如：VP -> TV NP [0.4] | IV [0.3] | DatV NP NP [0.3]。为了确保由文法生成的树能形成概率分布，PCFG 文法强加了约束：产生式所有给定的左侧的概率之和必须为 1。例 8-6 中的文法符合这个约束：对 S 只有一个产生式，它的概率是 1.0；对于 VP，0.4+0.3+0.3=1.0；对于 NP，0.8+0.2=1.0。parse()返回的分析树包含概率：

```
>>> viterbi_parser = nltk.ViterbiParser(grammar)
>>> print viterbi_parser.parse(['Jack', 'saw', 'telescopes'])
(S (NP Jack) (VP (TV saw) (NP telescopes))) (p=0.064)
```

现在，分析树被分配了概率，一个给定的句子可能有数量庞大的可能的解析就不再是问题。分析器将负责寻找最有可能的解析。

8.7 小结

- 句子都有内部组织结构，可以用一棵树表示。组成结构的显著特点是：递归、中心词、补语和修饰语。
- 文法是一个潜在的无限的句子集合的一个紧凑的特性；我们说，一棵树是符合语法规则的或文法树授权一棵树。
- 文法是用于描述一个给定的短语是否可以被分配一个特定的成分或依赖结构的一种形式化模型。
- 给定一组句法类别，上下文无关文法使用一组生产式表示某类型 A 的短语如何能够被分析成较小的序列 α1 ... αn。
- 依存文法使用产生式指定给定的中心词的依赖是什么。
- 当一个句子有一个以上的文法分析就产生句法歧义（如介词短语附着歧义）。
- 分析器是一个过程，为符合语法规则的句子寻找一个或多个相应的树。
- 一个简单的自上而下分析器是递归下降分析器，在文法产生式的帮助下递归扩展开始符号（通常是 S），尝试匹配输入的句子。这个分析器并不能处理左递归产生式（如：NP -> NP PP）。它盲目扩充类别而不检查它们是否与输入字符串兼容的方式效率低下，而且会重复扩充同样的非终结符然后丢弃结果。
- 一个简单的自下而上的分析器是移位-规约分析器，它把输入移到一个堆栈中，并尝试匹配堆栈顶部的项目和文法产生式右边的部分。这个分析器不能保证为输入找到一个有效的解析，即使它确实存在，它建立子结构而不检查它是否与全部文法一致。

8.8 进一步阅读

本章的额外材料发布在 http://www.nltk.org/，包括网上免费提供的资源的链接。更多使用 NLTK 分析的例子，请看 http://www.nltk.org/howto 上的 Parsing HOWTO。

有许多关于句法的入门书籍：(O’Grady et al., 2004)是一个语言学概论，而（Radford, 1988）以容易接受的方式介绍转换文法，推荐其中的无限制依赖结构的转换文法。在形式语言学中最广泛使用的术语是生成文法，虽然它与 generation 并没有关系（Chomsky, 1965）。

(Burton-Roberts, 1997)是一本面向实践的关于如何分析英语成分的教科书，包含广泛的

例子和练习。(Huddleston & Pullum, 2002)提供了一份最新的英语句法现象的综合分析。

(Jurafsky & Martin, 2008)的第 12 章讲述英语的形式文法；13.1-3 节讲述简单的分析算法和歧义处理技术；第 14 章讲述统计分析；第 16 章讲述乔姆斯基层次和自然语言的形式复杂性。(Levin, 1993) 根据它们的句法属性，将英语动词划分成更细的类。

有几个正在进行的建立大规模的基于规则的文法的项目，如：LFG Pargram 项目（http://www2.parc.com/istl/groups/nltt/pargram/），HPSG LinGO 矩阵框架（http://www.delph-in.net/matrix/）以及 XTAG 项目（http://www.cis.upenn.edu/ xtag/）。

8.9 练习

1. ○你能想出符合语法的却可能之前从来没有被说出的句子吗？（与伙伴轮流进行。）这告诉你关于人类语言的什么？
2. ○回想一下 Strunk 和 White 的禁止在句子开头使用 however 表示“although”的意思。在网上搜索句子开头使用的 however。这个成分使用的有多广泛？
3. ○思考句子：Kim arrived or Dana left and everyone cheered。用括号的形式表示 and 和 or 的相对范围。产生这两种解释对应的树结构。
4. ○Tree 类实现了各种其他有用的方法。请看 Tree 帮助文档查阅更多细节（如：导入 Tree 类，然后输入 `help(Tree)`）。
5. ○在本练习中，你将手动构造一些分析树。
   a. 编写代码产生两棵树，对应短语 old men and women 的不同读法。
   b. 将本章中表示的任一一颗树编码为加标签的括号括起的形式，使用 `nltk.Tree()` 检查它是否符合语法。使用 `draw()` 显示树。
   c. 如（a）中那样，为 The woman saw a man last Thursday 画一棵树。
6. ○写一个递归函数，遍历一颗树，返回树的深度，一颗只有一个节点的树的深度应为 0。（提示：子树的深度是其子女的最大深度加 1。）
7. ○分析 A.A. Milne 关于 Piglet 的句子，为它包含的所有句子画下划线，然后用 S 替换这些（如：第一句话变为 S when S）。为这种“压缩”的句子画一个树形结构。用于建立这样一个长句的主要的句法结构是什么？
8. ○在递归下降分析器的演示中，通过选择“编辑”菜单上的“编辑文本”改变实验句子。
9. ○grammar1 中的文法（例 8-1）能被用来描述长度超过 20 词的句子吗？
10. ○使用图形化图表分析器接口，尝试不同的规则调用策略做实验。拿出你自己的可以使用图形界面手动执行的策略。描述步骤，并报告它的任何效率的提高（例如：用结果图表示大小）。这些改进取决于文法结构吗？你觉得一个更聪明的规则调用策略能显著提升性能吗？
11. ○对于一个你已经见过的或一个你自己设计的 CFG，用笔和纸手动跟踪递归下降分析器和移位-规约分析器的执行。
12. ○我们已经看到图表分析器增加边而从来不从图表中删除的边，为什么？
13. ○思考词序列：Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo。如 http://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo 解释的，这是一个语法正确的句子。思考此维基百科页面上表示的树形图，写一个合适的文法。正常情况下是小写，模拟听到这句话时听者会遇到的问题。你能为这句话找到其他的解析吗？当句子变长时分析树的数量如何增长？（这些句子的更多的例子可以在 http://en.wikipedia.org/wiki/List_of_homophonous_phrases 找到。）
14. ◑你可以通过选择“编辑”菜单上的“编辑文法”修改递归下降分析器演示程序。改变

第一次扩充产生式，即 NP -> Det N PP 为 NP -> NP PP。使用 Step 按钮，尝试建立一个分析树。发生了什么？

15. ◑扩展 grammar2 中的文法，将产生式中的介词扩展为不及物的，及物的和需要 PP 补语的。基于这些产生式，使用前面练习中的方法为句子 Lee ran away home 画一棵树。
16. ◑挑选一些常用动词，完成以下任务：
    a. 写一个程序在 PP 附着语料库 nltk.corpus.ppattach 找到那些动词。找出任何这样的情况：相同的动词有两种不同的附着，其中第一个是名词，或者第二个是名词，或者介词保持不变（正如我们在 8.2 节句法歧义中讨论过的）。
    b. 制定 CFG 文法产生式涵盖其中一些情况。
17. ◑写一个程序，比较自上而下的图表分析器与递归下降分析器的效率（8.4 节）。使用相同的文法和输入的句子。使用 timeit 模块比较它们的性能（见 4.7 节，如何做到这一点的一个例子）。
18. ◑比较自上而下、自下而上和左角落分析器的性能，使用相同的文法和 3 个符合语法的测试句子。使用 timeit 记录每个分析器在同一个句子上花费的时间。写一个函数，在这三句话上运行这三个分析器，输出 3×3 格的时间，以及行和列的总计。讨论你的结果。
19. ◑阅读“garden path”的句子。一个分析器的计算工作与人类处理这些句子的困难性有什么关系？（见 http://en.wikipedia.org/wiki/Garden_path_sentence。）
20. ◑在一个窗口比较多个树，我们可以使用 draw_trees()方法。定义一些树，尝试一下：

    ```
    >>> from nltk.draw.tree import draw_trees
    >>> draw_trees(tree1, tree2, tree3)
    ```

21. ◑使用树中的位置，列出宾州树库前 100 个句子的主语；为了使结果更容易查看，限制从高度最高为 2 的子树提取主语。
22. ◑查看 PP 附着语料库，尝试提出一些影响 PP 附着的因素。
23. ◑在 8-2 节中，我们说过简单的用术语 n-grams 不能描述所有语言学规律。思考下面的句子，尤其是短语 in his turn 的位置。这是基于 n-grams 的方法的一个问题吗？

    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)
24. ◑写一个递归函数产生嵌套的括号括起的形式的一棵树，显示去掉叶节点之后的子树的非终结符。于是 8.6 节中的关于 Pierre Vinken 的例子会产生：[[[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。连续的类别应用空格分隔。
25. ◑从古登堡工程下载一些电子图书。写一个程序扫描这些文本中任何极长的句子。你能找到的最长的句子是什么？这么长的句子的语法结构是什么？
26. ◑修改函数 init_wfst()和 complete_wfst()，使 WFST 中每个单元的内容是一组非终端符而不是一个单独的非终结符。
27. ◑思考例 8-3 中的算法。你能解释为什么分析上下文无关文法是与 n^3 成正比的，其中 n 是输入句子的长度。
28. ◑处理宾州树库语料库样本 nltk.corpus.treebank 中的每棵树，在 Tree.productions()的帮助下提取产生式。丢弃只出现一次的产生式。具有相同的左侧和类似的右侧的产生式可以被折叠，产生一个等价的却更紧凑的规则集。编写代码输出此紧凑的文法。
29. ●英语中定义句子 S 的主语的一种常见的方法是作为 S 的孩子和 VP 的兄弟的名词短

语。写一个函数，以一句话的树为参数，返回句子主语对应的子树。如果传递给这个函数的树的根节点不是 S 或它缺少一个主语，应该怎么做？

30. ●写一个函数，以一个文法（如例 8-1 定义的文法）为参数，返回由这个文法随机产生的一个句子。（使用 grammar.start()找出文法的开始符号；grammar.productions(lhs)得到具有指定左侧的文法的产生式的链表；production.rhs()得到一个产生式的右侧。）
31. ●使用回溯实现移位-规约分析器的一个版本，使它找出一个句子所有可能的解析，它可以被称为“递归上升分析器”。咨询维基百科关于回溯的条目 http://en.wikipedia.org/wiki/Backtracking。
32. ●正如我们在第 7 章中所看到的，可以将块表示成它们的块标签。当我们为包含 gave 的句子做这个的时候，我们发现如以下模式：

    gave NP
    gave up NP in NP
    gave NP up
    gave NP NP
    gave NP to NP

    a. 使用这种方法学习一个感兴趣的动词的互补模式，编写合适的文法产生式。（此任务有时也被称为词汇获取。）
    b. 识别一些英语动词的近同义词，如 dumped/filled/loaded，来自第 9 章中（64）的例子。使用分块方法研究这些动词的互补模式。创建一个文法覆盖这些情况。这些动词可以自由的取代对方，或者有什么限制吗？讨论你的结果。
33. ●开发一种基于递归下降分析器的左角落分析器，从 ParseI 继承。
34. ●扩展 NLTK 中的移位-规约分析器使其包含回溯，这样就能保障找到所有存在的解析（也就是说，它是完整的）。
35. ●修改函数 init_wfst()和 complete_wfst()，当一个非终结符添加到 WFST 中的单元时，它包括一个它所派生的单元的记录。实现一个函数，将一个分析树的 WFST 转换成这种形式。

第 9 章 建立基于特征的文法

自然语言具有范围广泛的文法结构，用第 8 章中所描述的简单的方法很难处理的如此广泛的文法结构。为了获得更大的灵活性，我们改变我们对待文法类别如 S、NP 和 V 的方式。我们将这些原子标签分解为类似字典的结构，以便可以提取一系列的值作为特征。

本章的目的是要回答下列问题：

1. 我们怎样用特征扩展上下文无关文法框架，以获得更细粒度的对文法类别和产生式的控制？
2. 特征结构的主要形式化属性是什么，我们如何使用它们来计算？
3. 我们现在用基于特征的文法能捕捉到什么语言模式和文法结构？

一路上，我们将介绍更多的英语句法主题，包括：约定、子类别和无限制依赖成分等现象。

9.1 文法特征

在第 6 章，我们描述了如何通过检测文本的特征建立分类器。那些特征可能非常简单，如提取一个单词的最后一个字母，或者更复杂一点儿，如分类器自己预测的词性标签。在本章中，我们将探讨在建立基于规则的文法中特征的作用。对比特征提取，记录已经自动检测到的特征，我们现在要声明词和短语的特征。我们以一个很简单的例子开始，使用字典存储特征和它们的值。

```
>>> kim = {'CAT': 'NP', 'ORTH': 'Kim', 'REF': 'k'}
>>> chase = {'CAT': 'V', 'ORTH': 'chased', 'REL': 'chase'}
```

对象 kim 和 chase 有几个共同的特征，CAT（文法类别）和 ORTH（正字法，即拼写）。此外，每一个还有更面向语义的特征：kim['REF']意在给出 kim 的指示物，而 chase['REL']给出 chase 表示的关系。在基于规则的文法上下文中，这样的特征和特征值对被称为特征结构，我们将很快看到它们的替代符号。

特征结构包含各种有关文法实体的信息。这些信息不需要详尽无遗，我们可能要进一步增加属性。例如：一个动词，知道它扮演的“语义角色”往往很有用。对于 chase，主语扮演“agent（施事）”的角色，而宾语扮演“patient（受事）”角色。让我们添加这些信息，使用'sbj'（主语）和'obj'（宾语）作为占位符，它会被填充，当动词和它的文法参数结合时：

```
>>> chase['AGT'] = 'sbj'
>>> chase['PAT'] = 'obj'
```

如果我们现在处理句子：Kim chased Lee，我们要“绑定”动词的施事角色和主语，受事角色和宾语。我们可以通过链接到相关的 NP 的 REF 特征做到这个。在下面的例子中，我们做一个简单的假设：在动词直接左侧和右侧的 NP 分别是主语和宾语。我们还在例子结尾为 Lee 添加了一个特征结构。

```
>>> sent = "Kim chased Lee"
>>> tokens = sent.split()
>>> lee = {'CAT': 'NP', 'ORTH': 'Lee', 'REF': 'l'}
>>> def lex2fs(word):
```

```
...     for fs in [kim, lee, chase]:
...         if fs['ORTH'] == word:
...             return fs
>>> subj, verb, obj = lex2fs(tokens[0]), lex2fs(tokens[1]), lex2fs(tokens[2])
>>> verb['AGT'] = subj['REF'] # agent of 'chase' is Kim
>>> verb['PAT'] = obj['REF']  # patient of 'chase' is Lee
>>> for k in ['ORTH', 'REL', 'AGT', 'PAT']: # check featstruct of 'chase'
...     print "%-5s => %s" % (k, verb[k])
ORTH  => chased
REL   => chase
AGT   => k
PAT   => l
```

同样的方法可以适用不同的动词——say，surprise——虽然在这种情况下，主语将扮演“source（源事）”（SRC）的角色，宾语扮演“experiencer（体验者）”（EXP）的角色：

```
>>> surprise = {'CAT': 'V', 'ORTH': 'surprised', 'REL': 'surprise',
...             'SRC': 'sbj', 'EXP': 'obj'}
```

特征结构是非常强大的，但我们操纵它们的方式是极其特别的。我们本章接下来的任务是，显示上下文无关文法和分析如何能扩展到合适的特征结构，使我们可以一种更通用的和有原则的方式建立像这样的分析。我们将通过查看句法协议的现象作为开始；我们将展示如何使用特征典雅的表示协议约束，并在一个简单的文法中说明它们的用法。

由于特征结构是表示任何形式的信息的通用的数据结构，我们将从更形式化的视点简要地看着它们，并演示 NLTK 提供的特征结构的支持。在本章的最后一部分，我们将表明，特征的额外表现力开辟了一个用于描述语言结构的复杂性的广泛的可能性。

句法协议

下面的例子展示词序列对，其中第一个是符合语法的而第二个不是。（我们在词序列的开头用星号表示它是不符合语法的。）

(1) a. this dog
    b. *these dog
(2) a. these dogs
    b. *this dogs

在英语中，名词通常被标记为单数或复数。范例中的形式也各不相同：this（单数），these（复数）。例（1）和（2）显示在名词短语中使用指示词和名词是有限制的。要么两个都是单数要么都是复数。主语和谓词间也存在类似的约束：

(3) a. the dog runs
    b. *the dog run
(4) a. the dogs run
    b. *the dogs runs

这里我们可以看到，动词的形态属性与主语名词短语的句法属性一起变化。这种一起变化被称为**协议（agreement）**。如果我们进一步看英语动词协议，我们将看到动词的现在时态通常有两种屈折形式：一为第三人称单数，另一个为人称和数量的所有其他组合，如表 9-1 所示。

表 9-1. 英语规则动词的协议范式

	单数	复数
第一人称	*I run*	*we run*
第二人称	*you run*	*you run*
第三人称	*he/she/it runs*	*they run*

我们可以让形态学特性的作用更加明确一点儿，如（5）和（6）所示。这些例子表明动词与它的主语在人称和数量上保持一致。（我们用 3 作为第三人称的缩写，SG 表示单数，PL 表示复数。）

(5) the dog run-s
dog.3.SG run-3.SG

(6) the dog-s run
dog.3.PL run-3.PL

让我们看看当我们在一个上下文无关文法中编码这些协议约束会发生什么。我们将以（7）中简单的 CFG 开始。

(7)
```
S   -> NP VP
NP  -> Det N
VP  -> V
Det -> 'this'
N   -> 'dog'
V   -> 'runs'
```

文法（7）使我们能够产生句子 this dog runs；然而，我们真正想要做的是也能产生 these dogs run，而同时阻止不必要的序列，如*this dogs run 和*these dog runs。最简单的方法是为文法添加新的非终结符和产生式：

(8)
```
S -> NP_SG VP_SG
S -> NP_PL VP_PL
NP_SG -> Det_SG N_SG
NP_PL -> Det_PL N_PL
VP_SG -> V_SG
VP_PL -> V_PL
Det_SG -> 'this'
Det_PL -> 'these'
N_SG -> 'dog'
N_PL -> 'dogs'
V_SG -> 'runs'
V_PL -> 'run'
```

在唯一的扩展 S 的产生式的地方，我们现在有两个产生式，一个覆盖单数主语 NP 和 VP 的句子，另一个覆盖复数主语 NP 和 VP 的句子。事实上，（7）中的每个产生式在（8）中都有两个与之对应。在小规模的文法中这不是什么真的问题，虽然它在审美上不那么吸引人。然而，在一个更大的涵盖了一定量的英语成分子集的文法中，这样将文法规模增加一倍的前景是非常缺乏吸引力。让我们假设现在我们用同样的方法处理第一，第二和第三人称，还有单数和复数。这将导致原有的文法被乘以因子 6，这是我们一定要避免的。我们可以做得比这更好吗？在下一节，我们将显示遵从数量和人称协议不必以“爆炸式”的产生式数目为代价。

使用属性和约束

我们说过非正式的语言类别具有属性，例如：名词具有复数的属性。让我们把这个弄的更明确：

(9) N[NUM=pl]

（9）中，我们介绍了一些新的符号，它的意思是类别 N 有一个（文法）**特征**叫做 NUM（“number 数字”的简写），此特征的值是 pl（“plural 复数”的简写）。我们可以添加类似的注解给其他类别，在词汇条目中使用它们：

(10) Det[NUM=sg] -> 'this'
Det[NUM=pl] -> 'these'
N[NUM=sg] -> 'dog'
N[NUM=pl] -> 'dogs'
V[NUM=sg] -> 'runs'
V[NUM=pl] -> 'run'

这确实有帮助吗？迄今为止，它看起来就像（8）中指定内容的一个稍微更详细的替代。当我们允许使用特征值变量，并利用这些表示限制时，事情变得更有趣：

(11) S -> NP[NUM=?n] VP[NUM=?n]
NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]
VP[NUM=?n] -> V[NUM=?n]

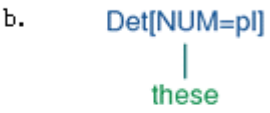
我们使用?n 作为 NUM 值上的变量；它可以在给定的产生式中被实例化为 sg 或 pl。我们可以读取第一条产生式就像再说不管 NP 为特征 NUM 取什么值，VP 必须取同样的值。

为了介绍这些特征限制如何工作，思考如何去建立一个树是有益的。词汇产生式将承认下面的树（树的深度为 1）：

(12)

a.

b.



(13)

a.

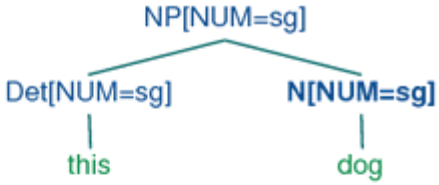
b.

现在 NP[NUM=?n] -> Det[NUM=?n] N[NUM=?n]表示无论 N 和 Det 的 NUM 的值是什么，它们都是相同的。因此，这个产生式允许(12a)和(13a)组合成一个 NP，如（14a）所示，它也将允许（12b）和（13b）组合，如（14b）中所示。相比之下，（15a）和（15b）被禁止因为它们子树的根的 NUM 值不同；这种值的不相容可以用顶端节点的值 FAIL 非正

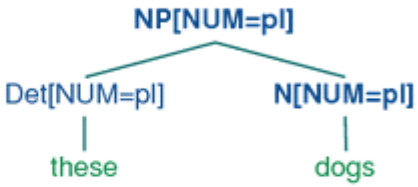
式的表示。

(14)

a.

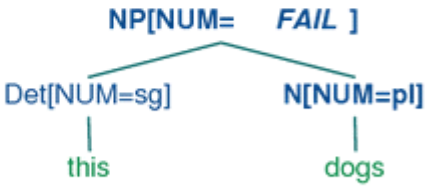


b.

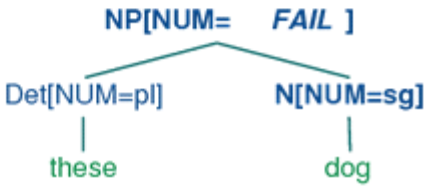


(15)

a.

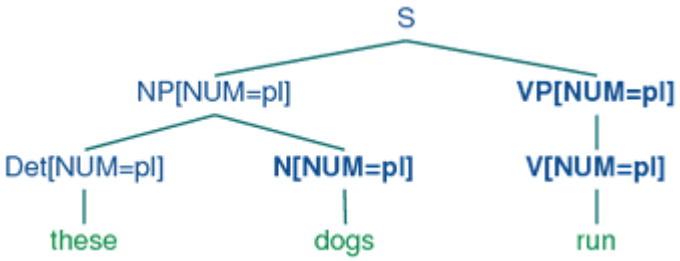


b.



产生式 VP[NUM=?n] -> V[NUM=?n]表示核心动词的 NUM 必须与它的 VP 父母的 NUM 值相同。结合扩展 S 的产生式，我们得出结论：如果主语核心名词的 NUM 值是 pl，那么 VP 核心动词的 NUM 值也是 pl。

(16)



文法（10）演示了如 this 和 these 这样的限定词的词汇产生式分别需要一个单数或复数的核心名词。然而，英语中其他限定词对与它们结合的名词的文法上的数量并不挑剔。一个描述这个的方法将是添加两个词汇条目到文法中，单数和复数版本的限定词如 the 一样一个。

Det[NUM=sg] -> 'the' | 'some' | 'several'

Det[NUM=pl] -> 'the' | 'some' | 'several'

然而，一个更优雅的解决办法是保留 NUM 的值为未指定，让它匹配与它结合的任何名词的数量。为 NUM 分配一个变化的值是解决的方法之一：

Det[NUM=?n] -> 'the' | 'some' | 'several'

但事实上我们可以更简单些，在这样的产生式中不给 NUM 任何指定。我们只需要在这个限制制约同一个产生式的其他地方的值时才明确地输入一个变量的值。

例 9-1 中的文法演示到目前为止我们已经在本章中介绍过的大多数想法，再加上几个新的想法。

例 9-1. 基于特征的文法的例子。

```
>>> nltk.data.show_cfg('grammars/book_grammars/feat0.fcfg')
% start S
# ###################
# Grammar Productions
# ###################
# S expansion productions
S -> NP[NUM=?n] VP[NUM=?n]
# NP expansion productions
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 productions
VP[TENSE=?t, NUM=?n] -> IV[TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> TV[TENSE=?t, NUM=?n] NP
# ###################
# Lexical Productions
# ###################
Det[NUM=sg] -> 'this' | 'every'
Det[NUM=pl] -> 'these' | 'all'
Det -> 'the' | 'some' | 'several'
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] -> 'disappeared' | 'walked'
TV[TENSE=past] -> 'saw' | 'liked'
```

注意一个句法类别可以有多个特征，例如：V[TENSE=pres, NUM=pl]。在一般情况下，我们喜欢多少特征就可以添加多少。

关于例 9-1 的最后的细节是语句%start S。这个“指令”告诉分析器以 S 作为文法的开始符号。

一般情况下，即使我们正在尝试开发很小的文法，把产生式放在一个文件中我们可以编辑、测试和修改是很方便的。我们将例 9-1 以 NLTK 的数据格式保存为文件 feat0.fcfg。你可以使用 nltk.data.load()制作你自己的副本进行进一步的实验。

NLTK 中使用 Earley 图表分析器分析基于特征的文法（关于这个的更多信息见 9.5 节），例 9-2 演示这个是如何实施的。为输入分词之后，我们导入 load_parser 函数①，以文法文件名为输入，返回一个图表分析器 cp②。调用分析器的 nbest_parse()方法将返回一个分析树的 trees 链表；如果文法无法分析输入，trees 将为空，否则会包含一个或多个分析

树，取决于输入是否有句法歧义。

例 9-2. 跟踪基于特征的图表分析器.

```
>>> tokens = 'Kim likes children'.split()
>>> from nltk import load_parser ①
>>> cp = load_parser('grammars/book_grammars/feat0.fcfg', trace=2) ②
>>> trees = cp.nbest_parse(tokens)
|.Kim .like.chil.|
|[----]    .    .| PropN[NUM='sg'] -> 'Kim' *
|[----]    .    .| NP[NUM='sg'] -> PropN[NUM='sg'] *
|[---->    .    .| S[] -> NP[NUM=?n] * VP[NUM=?n] {?n: 'sg'}
|.    [----]    .| TV[NUM='sg', TENSE='pres'] -> 'likes' *
|.    [---->    .| VP[NUM=?n, TENSE=?t] -> TV[NUM=?n, TENSE=?t] * NP[]
                     {?n: 'sg', ?t: 'pres'}
|.    .    [----]| N[NUM='pl'] -> 'children' *
|.    .    [----]| NP[NUM='pl'] -> N[NUM='pl'] *
|.    .    [---->| S[] -> NP[NUM=?n] * VP[NUM=?n] {?n: 'pl'}
|.    [--------]| VP[NUM='sg', TENSE='pres']
                     -> TV[NUM='sg', TENSE='pres'] NP[] *
|[==============]| S[] -> NP[NUM='sg'] VP[NUM='sg'] *
```

分析过程中的细节对于当前的目标并不重要。然而，有一个实施上的问题与我们前面的讨论文法的大小有关。分析包含特征限制的产生式的一种可行的方法是编译出问题中特征的所有可接受的值，是我们最终得到一个大的完全指定的（8）中那样的 CFG。相比之下，前面例子中显示的分析器过程直接与给定文法的未指定的产生式一起运作。特征值从词汇条目“向上流动”，变量值于是通过如{?n: 'sg', ?t: 'pres'}这样的绑定（即字典）与那些值关联起来。当分析器装配有关它正在建立的树的节点的信息时，这些变量绑定被用来实例化这些节点中的值；从而通过查找绑定中?n 和?t 的值，未指定的 VP[NUM=?n, TENSE=?t] -> TV[NUM=?n, TENSE=?t] NP[]实例化为 VP[NUM='sg', TENSE='pres'] -> TV [NUM='sg', TENSE='pres'] NP[]。

最后，我们可以检查生成的分析树（在这种情况下，只有一个）。

```
>>> 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))))
```

术语

到目前为止，我们只看到像 sg 和 pl 这样的特征值。这些简单的值通常被称为**原子**——也就是，它们不能被分解成更小的部分。原子值的一种特殊情况是**布尔值**，也就是说，值仅仅指定一个属性是真还是假。例如：我们可能要用布尔特征 AUX 区分**助动词**，如：can、may、will 和 do。那么产生式 V[TENSE=pres,aux=+] -> 'can'意味着 can 接受 TENSE 的值为 pres 并且 AUX 的值为+或 true。有一个广泛采用的约定用缩写表示布尔特征 f：不用 aux=+或 aux=-，我们分别用+aux 和-aux。这些都是缩写，然而，分析器就像+和-

是其他原子值一样解释它们。（17）显示了一些有代表性的产生式：

(17) V[TENSE=pres, +aux] -> 'can'
V[TENSE=pres, +aux] -> 'may'
V[TENSE=pres, -aux] -> 'walks'
V[TENSE=pres, -aux] -> 'likes'

我们说过分配“特征注释”给句法类别。表示整个类别的更激进的做法——也就是，非终结符加注释——作为一个特征的绑定。例如：N[NUM=sg]包含词性信息，可以表示为POS=N。因此，这个类别的替代符号是[POS=N, NUM=sg]。

除了原子值特征，特征可能需要本身就是特征结构的值。例如：我们可以将协议特征组合在一起（例如：人称、数量和性别）作为一个类别的不同部分，表示为 AGR 的值。这种情况中，我们说 AGR 是一个**复杂**值。（18）描述的结构，在格式上称为**属性值矩阵**（attribute value matrix，AVM）。

(18) [POS = N]
[ ]
[AGR = [PER = 3]]
[[NUM = pl]]
[[GND = fem]]

在传递中，我们应该指出有显示 AVM 的替代方法；图 9-1 显示了一个例子。虽然特征结构呈现的（18）中的风格不太悦目，我们将坚持用这种格式，因为它对应我们将会从 NLTK 得到的输出。

$$\begin{bmatrix} \text{POS} & N \\ \text{AGR} & \begin{bmatrix} \text{PER} & 3 \\ \text{NUM} & pl \\ \text{GND} & fem \end{bmatrix} \end{bmatrix}$$

图 9-1. 以属性值矩阵呈现一个特征结构。

关于表示，我们也注意到特征结构，像字典，对特征的顺序没有指定特别的意义。因此，（18）等价于：

(19) [AGR = [NUM = pl]]
[[PER = 3]]
[[GND = fem]]
[ ]
[POS = N]

当我们有可能使用像 AGR 这样的特征时，我们可以重构像例 9-1 这样的文法，使协议特征捆绑在一起。一个微小的文法演示了这个想法，如（20）所示。

(20) S -> NP[AGR=?n] VP[AGR=?n]
NP[AGR=?n] -> PropN[AGR=?n]
VP[TENSE=?t, AGR=?n] -> Cop[TENSE=?t, AGR=?n] Adj

Cop[TENSE=pres, AGR=[NUM=sg, PER=3]] -> 'is'
PropN[AGR=[NUM=sg, PER=3]] -> 'Kim'
Adj -> 'happy'

9.2 处理特征结构

在本节中，我们将展示如何构建特征结构，并在 NLTK 中操作。我们还将讨论统一的基本操作，这使我们能够结合两个不同的特征结构中的信息。

NLTK 中的特征结构使用构造函数 FeatStruct()声明。原子特征值可以是字符串或整数。

```
>>> fs1 = nltk.FeatStruct(TENSE='past', NUM='sg')
>>> print fs1
[ NUM   = 'sg'   ]
[ TENSE = 'past' ]
```

一个特征结构实际上只是一种字典，所以我们可以平常的方式通过索引访问它的值。我们可以用我们熟悉的方式指定值给特征：

```
>>> fs1 = nltk.FeatStruct(PER=3, NUM='pl', GND='fem')
>>> print fs1['GND']
fem
>>> fs1['CASE'] = 'acc'
```

我们还可以为特征结构定义更复杂的值，如前所述。

```
>>> 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
```

指定特征结构的另一种方法是使用包含 feature=value 格式的特征-值对的方括号括起的字符串，其中值本身可能是特征结构：

```
>>> print nltk.FeatStruct("[POS='N', AGR=[PER=3, NUM='pl', GND='fem']]")
[       [ PER = 3     ] ]
[ AGR = [ GND = 'fem' ] ]
[       [ NUM = 'pl'  ] ]
[                       ]
[ POS = 'N'             ]
```

特征结构本身并不依赖于语言对象；它们是表示知识的通用目的的结构。例如：我们可以将一个人的信息用特征结构编码：

```
>>> print nltk.FeatStruct(name='Lee', telno='01 27 86 42 96', age=33)
[ age   = 33                    ]
```

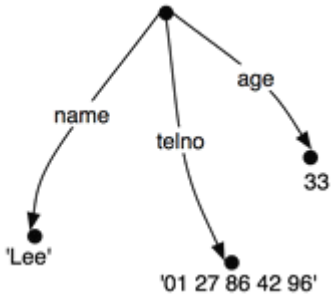
[name = 'Lee']

[telno = '01 27 86 42 96']

在接下来的几页中，我们会使用这样的例子来探讨特征结构的标准操作。这将使我们暂时从自然语言处理转移，因为在我们回来谈论文法之前需要打下基础。坚持！

将特征结构作为图来查看往往是有用的，更具体的，作为**有向无环图**（directed acyclic graphs，DAGs）。（21）相当于前面的 AVM。

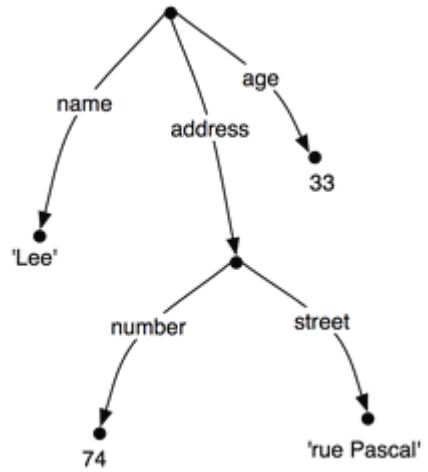
(21)



特征名称作为弧上的标签出现，特征值作为弧指向的节点的标签出现。

像以前一样，特征值可以是复杂的：

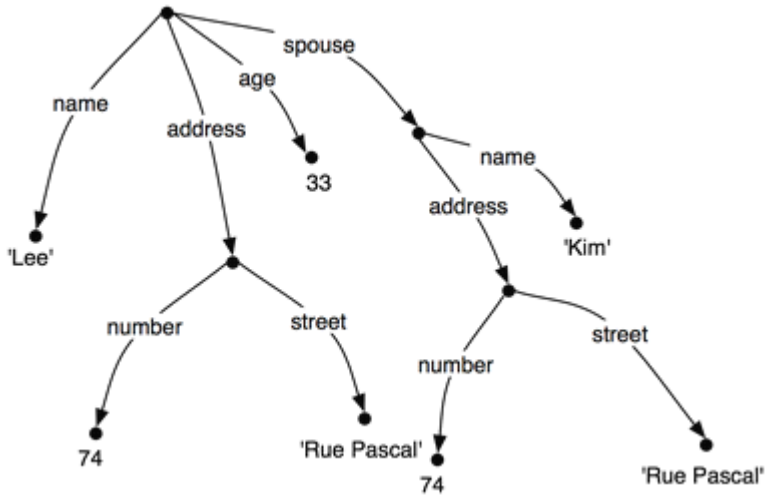
(22)



标签的元组表示路径。于是，('ADDRESS', 'STREET')就是一个特征路径，它在（22）中的值是标签为'rue Pascal'的节点。

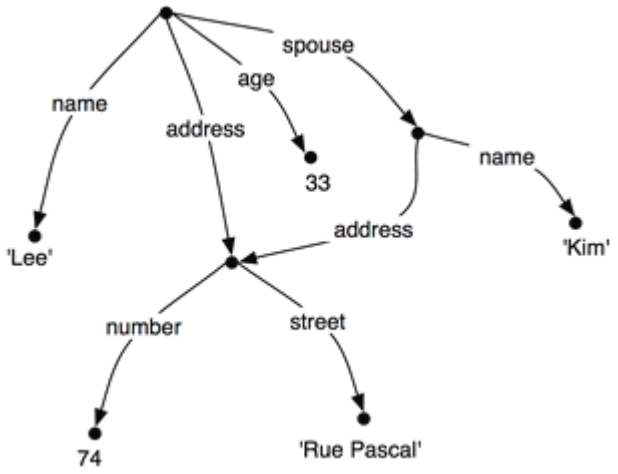
现在让我们思考一种情况：Lee 有一个配偶叫做 Kim，Kim 的地址与 Lee 的相同。我们可能会像（23）那样表示这个。

(23)



然而，不是重复特征结构中的地址信息，我们可以“共享”不同的弧之间的同一子图：

(24)



换句话说，(24)中路径('ADDRESS')的值与路径('SPOUSE', 'ADDRESS')的值相同。如（24）的 DAG 被称为包含**结构共享**或**重入**。当两条路径具有相同的值时，它们被称为是**等价**的。

为了在我们的矩阵式表示中表示重入，我们将在共享的特征结构第一次出现的地方加一个括号括起的数字前缀，例如（1）。以后任何对这个结构的引用将使用符号->(1)，如下所示。

```
>>> print nltk.FeatStruct("""[NAME='Lee', ADDRESS=(1)[NUMBER=74, STREET='rue Pascal'],
...                        SPOUSE=[NAME='Kim', ADDRESS->(1)]]""")
[ ADDRESS = (1) [ NUMBER = 74          ] ]
[               [ STREET = 'rue Pascal' ] ]
[                                         ]
[ NAME    = 'Lee'                         ]
[                                         ]
[ SPOUSE  = [ ADDRESS -> (1)  ]           ]
[           [ NAME    = 'Kim' ]           ]
```

括号内的整数有时也被称为**标记**或**同指标志**（coindex）。整数的选择并不重要。可以有任意数目的标记在一个单独的特征结构中。

```
>>> print nltk.FeatStruct("[A='a', B=(1)[C='c'], D->(1), E->(1)]")
[ A = 'a'               ]
[                       ]
[ B = (1) [ C = 'c' ] ]
[                       ]
[ D -> (1)              ]
[ E -> (1)              ]
```

包含和统一

认为特征结构提供一些对象的**部分信息**是很正常的，在这个意义上，我们可以根据它们通用的程度给特征结构排序。例如：（25a）比（25b）更一般（更少特征），（25b）比（25c）更一般。

(25) a. [NUMBER = 74]

b. [NUMBER = 74]
[STREET = 'rue Pascal']

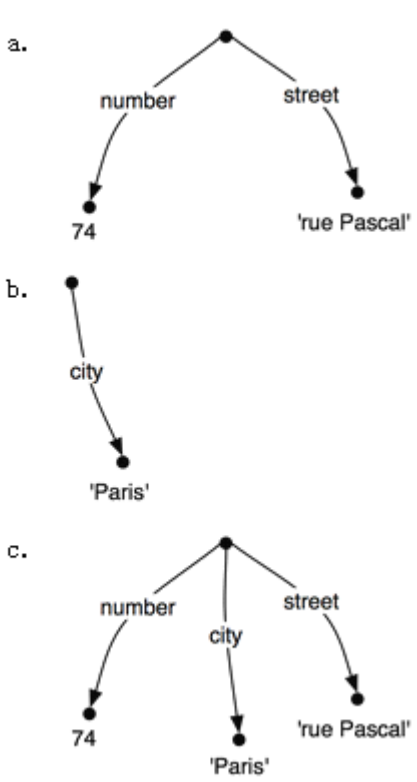
c. [NUMBER = 74]
[STREET = 'rue Pascal']
[CITY = 'Paris']

这个顺序被称为**包含（subsumption）**；一个更一般的特征结构**包含（ subsumes）**一个较少一般的。如果 FS0 包含 FS1（正式的，我们写成 FS0 FS1），那么 FS1 必须具备 FS0 所有路径和等价路径，也可能有额外的路径和等价路径。因此，（23）包括（24）因为后者有额外的等价路径。包含只提供了特征结构上的偏序，这应该是显而易见的，因为一些特征结构是不可比较的。例如：（26）既不包含（25a）也不被（25a）包含。

(26) [TELNO = 01 27 86 42 96]

所以，我们已经看到了，一些特征结构比其他的更具体。我们如何去具体化一个给定的特征结构？例如：我们可能决定地址应包括的不只是一个门牌号码和街道名称，还应该包括城市。也就是说，我们可能要用（27b）合并图（27a）去产生（27c）。

(27)



合并两个特征结构的信息被称为**统一**，由方法 unify()支持。

```
>>> fs1 = nltk.FeatStruct(NUMBER=74, STREET='rue Pascal')
>>> fs2 = nltk.FeatStruct(CITY='Paris')
>>> print fs1.unify(fs2)
[ CITY   = 'Paris'      ]
[ NUMBER = 74           ]
[ STREET = 'rue Pascal' ]
```

统一被正式定义为一个二元操作：FS0 FS1。统一是对称的，所以 FS0 FS1=FS1 FS0。Python 中也是如此：

```
>>> print fs2.unify(fs1)
```

```
[ CITY    = 'Paris'        ]
[ NUMBER  = 74             ]
[ STREET  = 'rue Pascal'   ]
```

如果我们统一两个具有包含关系的特征结构，那么统一的结果是两个中更具体的那个：

(28) If FS0 FS1, then FS0 FS1 = FS1

例子中统一（25b）和（25c）的结果是（25c）。

FS0 和 FS1 之间的统一将失败，如果这两个特征结构共享路径 π，但在 FS0 中的 π 值与在 FS1 中的 π 值是不同的原子值。这通过设置统一的结果为 None 来实现。

```
>>> fs0 = nltk.FeatStruct(A='a')
>>> fs1 = nltk.FeatStruct(A='b')
>>> fs2 = fs0.unify(fs1)
>>> print fs2
None
```

现在，如果我们看一下统一如何与结构共享相互作用，事情就变得很有趣。首先，让我们在 Python 中定义（23）：

```
>>> fs0 = nltk.FeatStruct("""[NAME=Lee,
...                           ADDRESS=[NUMBER=74,
...                                    STREET='rue Pascal'],
...                           SPOUSE= [NAME=Kim,
...                                    ADDRESS=[NUMBER=74,
...                                             STREET='rue Pascal']]]""")
>>> print fs0
[ ADDRESS = [ NUMBER = 74             ]                 ]
[           [ STREET = 'rue Pascal' ]                   ]
[                                                       ]
[ NAME    = 'Lee'                                       ]
[                                                       ]
[           [ ADDRESS = [ NUMBER = 74             ] ] ]
[ SPOUSE  = [           [ STREET = 'rue Pascal' ] ] ]
[           [                                     ] ]
[           [ NAME    = 'Kim'                     ] ]
```

我们为 Kim 的地址指定一个 CITY 作为参数会发生什么？请注意，fs1 需要包括从特征结构的根到 CITY 的整个路径。

```
>>> fs1 = nltk.FeatStruct("[SPOUSE = [ADDRESS = [CITY = Paris]]]")
>>> print fs1.unify(fs0)
[ ADDRESS = [ NUMBER = 74             ]                 ]
[           [ STREET = 'rue Pascal' ]                   ]
[                                                       ]
[ NAME    = 'Lee'                                       ]
[                                                       ]
[           [           [ CITY    = 'Paris'       ] ] ]
[           [ ADDRESS = [ NUMBER = 74             ] ] ]
[ SPOUSE  = [           [ STREET = 'rue Pascal' ] ] ]
[           [                                     ] ]
```

```
[              [ NAME      = 'Kim'                          ] ]
```

通过对比，如果 fs1 与 fs2 的结构共享版本统一，结果是非常不同的（如图（24）所示）：

```
>>> fs2 = nltk.FeatStruct("""[NAME=Lee, ADDRESS=(1)[NUMBER=74, STREET='rue Pascal'],
...                        SPOUSE=[NAME=Kim, ADDRESS->(1)]]""")
>>> print fs1.unify(fs2)
[                  [ CITY    = 'Paris'        ] ]
[ ADDRESS = (1) [ NUMBER = 74              ] ]
[                  [ STREET = 'rue Pascal' ] ]
[                                            ]
[ NAME      = 'Lee'                          ]
[                                            ]
[ SPOUSE    = [ ADDRESS -> (1)  ]            ]
[              [ NAME      = 'Kim' ]         ]
```

不是仅仅更新 Kim 的 Lee 的地址的“副本”，我们现在同时更新他们两个的地址。更一般的，如果统一包含指定一些路径 π 的值，那么统一同时指定与 π 等价的所有路径的值。

正如我们已经看到的，结构共享也可以使用变量表示，如?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)
```

9.3 扩展基于特征的文法

在本节中，我们回到基于特征的文法，探索各种语言问题，并展示将特征纳入文法的好处。

子类别

第 8 章中，我们增强了类别标签表示不同类别的动词，分别用标签 IV 和 TV 表示不及物动词和及物动词。这使我们能编写如下的产生式：

(29) VP -> IV
     VP -> TV NP

虽然我们知道 IV 和 TV 是两种 V，它们只是一个 CFG 中的原子非终结符，与任何其他符号对之间的区别是一样的。这个符号不会告诉我们任何有关一般动词的信息；例如：我们不能说“V 类的所有词汇都按时态标记”，因为 walk 是 IV 类的项目，不是 V 类的，所以，我们能替换如 TV 和 IV 类别标签为带着告诉我们这个动词是否与后面的 NP 对象结合或者它是否能不带补语等特征的 V？

一个简单的方法，最初为文法框架开发的称为广义短语结构文法（Generalized Phrase Structure Grammar，GPSG），通过允许词汇类别支持子类别特征尝试解决这个问题，它告诉我们该项目所属的子类别。相比 GPSG 使用的整数值表示 SUBCAT，下面的例子采用更容易记忆的值，即 intrans、trans 和 clause：

(30) VP[TENSE=?t, NUM=?n] -> V[SUBCAT=intrans, TENSE=?t, NUM=?n]
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=trans, TENSE=?t, NUM=?n] NP
VP[TENSE=?t, NUM=?n] -> V[SUBCAT=clause, TENSE=?t, NUM=?n] SBar
V[SUBCAT=intrans, TENSE=pres, NUM=sg] -> 'disappears' | 'walks'
V[SUBCAT=trans, TENSE=pres, NUM=sg] -> 'sees' | 'likes'
V[SUBCAT=clause, TENSE=pres, NUM=sg] -> 'says' | 'claims'
V[SUBCAT=intrans, TENSE=pres, NUM=pl] -> 'disappear' | 'walk'
V[SUBCAT=trans, TENSE=pres, NUM=pl] -> 'see' | 'like'
V[SUBCAT=clause, TENSE=pres, NUM=pl] -> 'say' | 'claim'
V[SUBCAT=intrans, TENSE=past] -> 'disappeared' | 'walked'
V[SUBCAT=trans, TENSE=past] -> 'saw' | 'liked'
V[SUBCAT=clause, TENSE=past] -> 'said' | 'claimed'

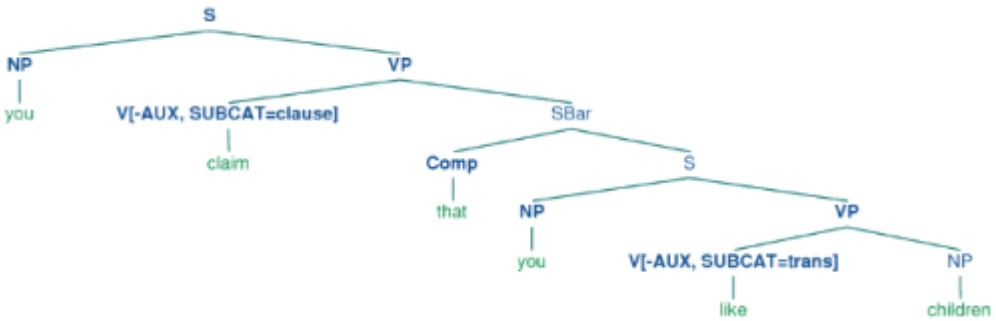
当我们看到一个词汇类别，如 V[SUBCAT=trans]，我们可以解释 SUBCAT 指向一个产生式，其中的 V[SUBCAT=trans]作为一个 VP 产生式中的核心词引入。按照约定，SUBCAT 值和引入核心词的产生式之间存在对应关系。对这种做法，SUBCAT 只能出现在词汇类别中，它是没有意义的，例如：要在 VP 上指定 SUBCAT。与要求的一样，walk 和 like 都属于类别 V，然而 walk 将只出现在被具有特征 SUBCAT=intrans 在右侧的产生式扩展的 VP 中，而 like 相反，它需要一个 SUBCAT=trans。

在（30）中的动词的第三个类别中，我们指定了一个类 SBar。这是一个从句标签，就像例子 You claim that you like children 中的 claim 的补语。我们需要两个进一步的产生式来分析这样的句子：

(31) SBar -> Comp S
Comp -> 'that'

产生的结构如下。

(32)



由于原本是用于称为文法类别的框架中的，子类别的一个替代方式被表示在基于特征的框架中，如 PATR 和核心驱动短语结构文法（Head-driven Phrase Structure Grammar）。不是用 SUBCAT 值作为索引产生式的方式，SUBCAT 值直接为中心词的配价（它能结合的参数的列表）编码。例如：动词 put 带 NP 和 PP 补语（put the book on the table）可以表示为（33）：

(33) V[SUBCAT=<NP, NP, PP>]

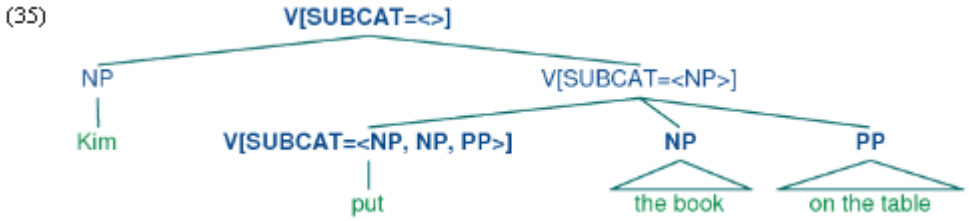
这是说动词可以结合三个参数。列表最左边元素是主语 NP，而其它所有的——这个例子中的后面跟着 PP 的 NP——包括了补语子类别。当动词如 put 与适当的补语结合时，SU

BCAT 中指定的要求被免除，只需要一个主语 NP。这个类别相当于传统上被认为的 VP，可以表示如下：

(34) V[SUBCAT=<NP>]

最后，一个句子成为一种不需要更多参数的动词类别，因此有一个值为空列表的 SUBCAT。树（35）显示了在短语 Kim put the book on the table 中这些类别是如何组合的。

(35)

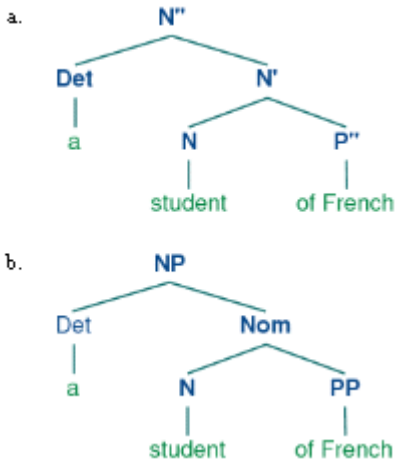


核心词回顾

我们注意到，在上一节中，通过从主类别标签分解出子类别信息，我们可以表达有关动词属性的更多概括。类似的另一个属性如下：V 类的表达式是 VP 类的短语的核心。同样，N 是 NP 的核心词，A（即形容词）是 AP 的核心词，P（即介词）是 PP 的核心词。并非所有的短语都有核心词——例如：一般认为连词短语（如：the book and the bel）缺乏核心词。然而，我们希望我们的文法形式能表达它所持有的父母/核心子女关系。现在，V 和 VP 只是原子符号，我们需要找到一种方法用特征将它们关联起来（就像我们以前关联 IV 和 TV 那样）。

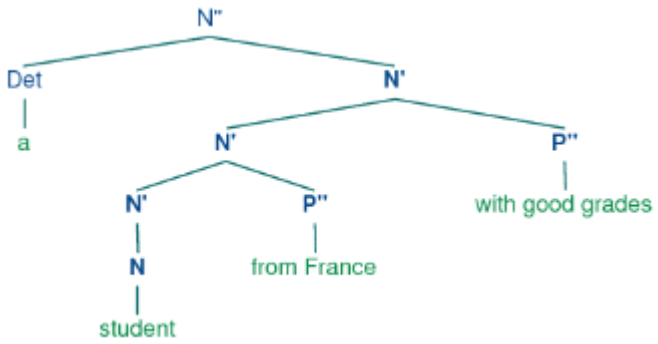
X-bar 句法通过抽象出**短语级别**的概念，解决了这个问题。它通常认为有三个这样的级别。如果 N 表示的词汇级别，那么 N'表示更高一层级别，对应较传统的级别 Nom，N''表示短语级别，对应类别 NP。（36a）演示了这种表示结构，而（36b）是更传统的对应。

(36)



结构（36a）的核心词是 N，而 N'和 N''被称为 N 的**（短语的）投影**。N''是**最大的投影**，N 有时也被称为**零投影**。X-bar 文法一个中心思想是所有成分都有结构的类似性。使用 X 作为 N、V、A 和 P 上的变量，我们说一个词汇核心 X 的直接补语子类别总是位于核心词的兄弟的位置，而修饰成分位于中间类别 X'的兄弟的位置。因此，（37）中两个 P''修饰成分的配置与（36a）中补语 P''的形成对比。

(37)



（38）中的产生式演示 X-bar 的级别用特征结构如何编码。（37）中嵌入结构通过扩展 N[BAR=1]的递归规则的两个应用来实现。

(38) S -> N[BAR=2] V[BAR=2]
N[BAR=2] -> Det N[BAR=1]
N[BAR=1] -> N[BAR=1] P[BAR=2]
N[BAR=1] -> N[BAR=0] P[BAR=2]

助动词与倒装

倒装从句——其中的主语和动词顺序互换——出现在英语疑问句，也出现在“否定”副词之后：

(39) a. Do you like children?
b. Can Jody walk?
(40) a. Rarely do you see Kim.
b. Never have I seen this dog.

然而，我们不能仅仅把任意动词放在主语前面：

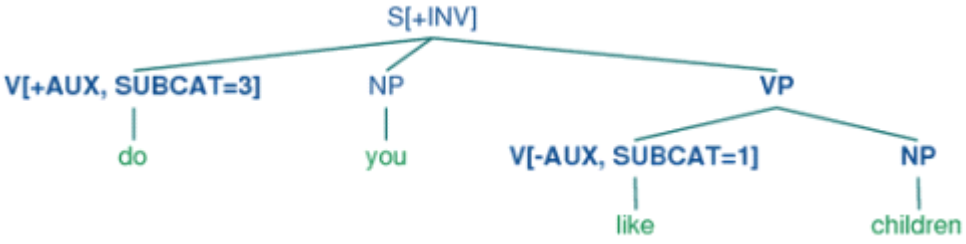
(41) a. *Like you children?
b. *Walks Jody?
(42) a. *Rarely see you Kim.
b. *Never saw I this dog.

可以放置在倒装的从句开头的动词术语叫做**助动词**的类别，如：do、can 和 have，也包括 be、will 和 shall。捕捉这种结构的方法之一是使用以下产生式：

(43) S[+INV] -> V[+AUX] NP VP

也就是说，标记有[+inv]的从句包含一个助动词，其后跟着一个 VP。（在更详细的文法中，我们需要在 VP 的形式上加一些限制，取决于选择的助动词。）（44）演示了一个倒装从句的结构。

(44)



无限制依赖成分

考虑下面的对比：

(45) a. You like Jody.

b. *You like.

(46) a. You put the card into the slot.

b. *You put into the slot.

c. *You put the card.

d. *You put.

动词 like 需要一个 NP 补语，而 put 需要一个跟随其后的 NP 和 PP。（45）和（46）表明，这些补语是必须的：省略它们会导致不符合语法。然而，有些上下文中强制性的补语可以省略，如（47）和（48）所示。

(47) a. Kim knows who you like.

b. This music, you really like.

(48) a. Which card do you put into the slot?

b. Which slot do you put the card into?

也就是说，一个强制性补语可以被省略，如果句子中有是适当的**填充**，例如(47a)中的疑问词 who，（47b）中的前置的主题 this music，或（48）中的 wh 短语 which card/slot。通常说类似(47)和(48)中的句子包含一个**缺口**，在那里强制性补语被省略了，这些缺口有时被用下划线标出：

(49) a. Which card do you put __ into the slot?

b. Which slot do you put the card into __?

所以，如果填充词许可，缺口就能出现。相反，填充词只会出现在句子希望有缺口的某个地方，如下面的例子所示：

(50) a. *Kim knows who you like Jody.

b. *This music, you really like hip-hop.

(51) a. *Which card do you put this into the slot?

b. *Which slot do you put the card into this one?

填充词和缺口之间的相互共同出现有时被称为“依赖”。在理论语言学中相当重视的一个问题一直是一个填充词和它许可的缺口之间可以相互作用的原因；特别的，我们能简单地列出一个分开这两个的序列的有限集合吗？答案是否定的：填充词和缺口之间的距离没有上界。这一事实可以很容易地使用包含句子补语的成分来说明，如（52）所示。

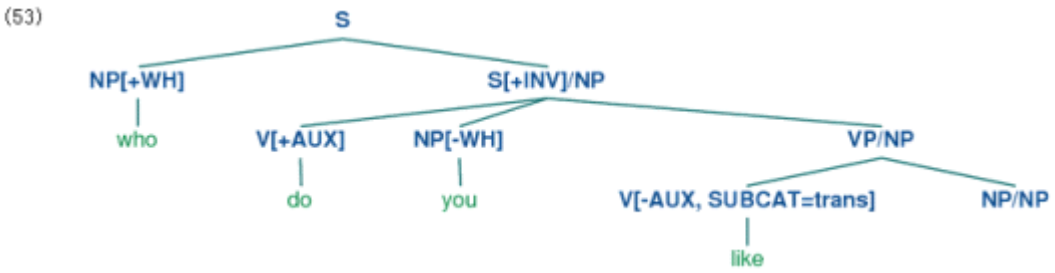
(52) a. Who do you like __?

b. Who do you claim that you like __?

c. Who do you claim that Jody says that you like __?

因为我们可以无限加深句子补语的递归，在整个句子中缺口可以无限远的被填充。这一属性导致无限依赖成分的概念，也就是填充词-缺口依赖，在那里填充词和缺口之间的距离没有上界

已经提出了各种各样的机制处理形式化文法中的无限依赖；在这里我们说明广义短语结构文法中使用的方法，其中包含斜线类别。一个斜线类别的形式是 Y/XP；我们解释为：类别 Y 的短语缺少一个类别 XP 的子成分。例如：S/NP 是缺少一个 NP 的 S。斜线类别的使用说明如（53）所示。



树的顶端部分引入了填充词 who（作为 NP[+wh]类表达式对待）和相应的包含成分 S/NP 的缺口一起。缺口信息于是被向着树的下方通过 VP/NP 类别“预填充”，直到到达类别 VP/NP。这时，由于意识到缺口信息为空字符串直接受控于 NP/NP，依赖被排除。

我们需要将斜线类别认为完全是一种新的对象吗？幸运的是，我们可以将它们纳入我们现有的基于特征的框架，通过将斜线作为一个特征，它右边的类别作为值；也就是说，S/NP 可变为 S[SLASH=NP]。在实践中，这也是分析器如何解释斜线类别的。

例 9-3 中的文法演示了斜线类别的主要原则，而且还包括倒装从句的产生式。为了简化演示，我们省略了动词上的任何时态规范。

例 9-3. 具有倒装从句和长距离依赖的产生式的文法，使用斜线类别。

```
>>> nltk.data.show_cfg('grammars/book_grammars/feat1.fcfg')
% start S
# ###################
# Grammar Productions
# ###################
S[-INV] -> NP VP
S[-INV]/?x -> NP VP/?x
S[-INV] -> NP S/NP
S[-INV] -> Adv[+NEG] S[+INV]
S[+INV] -> V[+AUX] NP VP
S[+INV]/?x -> V[+AUX] NP VP/?x
SBar -> Comp S[-INV]
SBar/?x -> Comp S[-INV]/?x
VP -> V[SUBCAT=intrans, -AUX]
VP -> V[SUBCAT=trans, -AUX] NP
VP/?x -> V[SUBCAT=trans, -AUX] NP/?x
VP -> V[SUBCAT=clause, -AUX] SBar
VP/?x -> V[SUBCAT=clause, -AUX] SBar/?x
VP -> V[+AUX] VP
VP/?x -> V[+AUX] VP/?x
# ###################
# Lexical Productions
# ###################
V[SUBCAT=intrans, -AUX] -> 'walk' | 'sing'
V[SUBCAT=trans, -AUX] -> 'see' | 'like'
V[SUBCAT=clause, -AUX] -> 'say' | 'claim'
V[+AUX] -> 'do' | 'can'
NP[-WH] -> 'you' | 'cats'
```

```
NP[+WH] -> 'who'
Adv[+NEG] -> 'rarely' | 'never'
NP/NP ->
Comp -> 'that'
```

例 9-3 中的文法包含一个“缺口引进”产生式，即 S[-INV] -> NP S/NP。为了正确的预填充斜线特征，我们需要为扩展 S、VP 和 NP 的产生式中箭头两侧的斜线添加变量值。例如：VP/?x -> V SBar/?x 是 VP -> V SBar 的斜线版本，也就是说，可以为一个成分的父母 VP 指定斜线值，只要也为孩子 SBar 指定同样的值。最后，NP/NP ->允许 NP 上的斜线信息为空字符串。使用例 9-3 中的文法，我们可以分析序列：who do you claim that you like：

```
>>> tokens = 'who do you claim that you like'.split()
>>> from nltk import load_parser
>>> cp = load_parser('grammars/book_grammars/feat1.fcfg')
>>> for tree in cp.nbest_parse(tokens):
...     print tree
(S[-INV]
  (NP[+WH] who)
  (S[+INV]/NP[]
    (V[+AUX] do)
    (NP[-WH] you)
    (VP[]/NP[]
      (V[-AUX, SUBCAT='clause'] claim)
      (SBar[]/NP[]
        (Comp[] that)
        (S[-INV]/NP[]
          (NP[-WH] you)
          (VP[]/NP[] (V[-AUX, SUBCAT='trans'] like) (NP[]/NP[] )))))))
```

这棵树的一个更易读的版本如（54）所示。

(54)

例 9-3 中的文法也可以分析没有缺口的句子：

```
>>> tokens = 'you claim that you like cats'.split()
>>> for tree in cp.nbest_parse(tokens):
...     print tree
(S[-INV]
  (NP[-WH] you)
  (VP[]
    (V[-AUX, SUBCAT='clause'] claim)
    (SBar[]
```

```
      (Comp[] that)
      (S[-INV]
        (NP[-WH] you)
        (VP[] (V[-AUX, SUBCAT='trans'] like) (NP[-WH] cats))))))
```

此外，它还允许没有 wh 结构的倒装句：

```
>>> tokens = 'rarely do you sing'.split()
>>> for tree in cp.nbest_parse(tokens):
...     print tree
(S[-INV]
  (Adv[+NEG] rarely)
  (S[+INV]
    (V[+AUX] do)
    (NP[-WH] you)
    (VP[] (V[-AUX, SUBCAT='intrans'] sing))))
```

德语中的格和性别

与英语相比，德语的协议具有相对丰富的形态。例如：在德语中定冠词根据格、性别和数量变化，如表 9-2 所示。

表 9-2. 德语定冠词的形态范式

格	男性	女性	中性	复数
主格	der	die	das	die
所有格	des	der	des	der
与格	dem	der	dem	den
宾格	den	die	das	die

德语中的主语采用主格，大多数动词采用宾格支配它们的宾语。不过，也有例外，例如: helfen 支配与格：

```
(55)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
```

例 9-4 中的文法演示带格的协议（包括人称、数量和性别）的相互作用。

例 9-4. 基于特征的文法的例子。

```
>>> nltk.data.show_cfg('grammars/book_grammars/german.fcfg')
% start S
 # Grammar Productions
```

```
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 Productions
# 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'
TV[OBJCASE=acc, AGR=[NUM=pl,PER=1]] -> 'sehen' | 'moegen'
TV[OBJCASE=acc, AGR=[NUM=pl,PER=2]] -> 'sieht' | 'moegt'
TV[OBJCASE=acc, AGR=[NUM=pl,PER=3]] -> 'sehen' | 'moegen'
TV[OBJCASE=dat, AGR=[NUM=pl,PER=1]] -> 'folgen' | 'helfen'
TV[OBJCASE=dat, AGR=[NUM=pl,PER=2]] -> 'folgt' | 'helft'
TV[OBJCASE=dat, AGR=[NUM=pl,PER=3]] -> 'folgen' | 'helfen'
```

正如你可以看到的，特征 objcase 被用来指定动词支配它的对象的格。下一个例子演示了包含支配与格的动词的句子的分析树：

```
>>> tokens = 'ich folge den Katzen'.split()
>>> cp = load_parser('grammars/book_grammars/german.fcfg')
>>> for tree in cp.nbest_parse(tokens):
...     print tree
(S[]
  (NP[AGR=[NUM='sg', PER=1], CASE='nom']
    (PRO[AGR=[NUM='sg', PER=1], CASE='nom'] ich))
  (VP[AGR=[NUM='sg', PER=1]]
    (TV[AGR=[NUM='sg', PER=1], OBJCASE='dat'] folge)
    (NP[AGR=[GND='fem', NUM='pl', PER=3], CASE='dat']
      (Det[AGR=[NUM='pl', PER=3], CASE='dat'] den)
      (N[AGR=[GND='fem', NUM='pl', PER=3]] Katzen))))
```

在开发文法时，排除不符合语法的词序列往往与分析符合语法的词序列一样具有挑战性。为了能知道在哪里和为什么序列分析失败，设置 load_parser()方法的 trace 参数可能是至关重要的。思考下面的分析故障：

```
>>> tokens = 'ich folge den Katze'.split()
>>> cp = load_parser('grammars/book_grammars/german.fcfg', trace=2)
>>> for tree in cp.nbest_parse(tokens):
...     print tree
|.ich.fol.den.Kat.|
|[---]   .   .   .| PRO[AGR=[NUM='sg', PER=1], CASE='nom'] -> 'ich' *
|[---]   .   .   .| NP[AGR=[NUM='sg', PER=1], CASE='nom']
                       -> PRO[AGR=[NUM='sg', PER=1], CASE='nom'] *
|[--->   .   .   .| S[] -> NP[AGR=?a, CASE='nom'] * VP[AGR=?a]
                           {?a: [NUM='sg', PER=1]}
|.   [---]   .   .| TV[AGR=[NUM='sg', PER=1], OBJCASE='dat'] -> 'folge' *
|.   [--->   .   .| VP[AGR=?a] -> TV[AGR=?a, OBJCASE=?c]
                               * NP[CASE=?c] {?a: [NUM='sg', PER=1], ?c: 'dat'}
|.   .   [---]   .| Det[AGR=[GND='masc', NUM='sg', PER=3], CASE='acc'] -> 'den' *
|.   .   [---]   .| Det[AGR=[NUM='pl', PER=3], CASE='dat'] -> 'den' *
```

```
|.   .   [--->   .| NP[AGR=?a, CASE=?c] -> Det[AGR=?a, CASE=?c]
                      * N[AGR=?a, CASE=?c] {?a: [NUM='pl', PER=3], ?c: 'dat'}
|.   .   [--->   .| NP[AGR=?a, CASE=?c] -> Det[AGR=?a, CASE=?c] * N[AGR=?a, CASE=?c]
                    {?a: [GND='masc', NUM='sg', PER=3], ?c: 'acc'}
|.   .   .   [---]| N[AGR=[GND='fem', NUM='sg', PER=3]] -> 'Katze' *
```

跟踪中的最后两个 Scanner 行显示 den 被识别为两个可能的类别：Det[AGR=[GND='masc', NUM='sg', PER=3], CASE='acc']和 Det[AGR=[NUM='pl', PER=3], CASE='dat']。我们从例 9-4 中的文法知道 Katze 的类别是 N[AGR=[GND=fem, NUM=sg, PER=3]]。然而，产生式中的变量?a 没有绑定：

```
NP[CASE=?c, AGR=?a] -> Det[CASE=?c, AGR=?a] N[CASE=?c, AGR=?a]
```

这将满足这些限制，因为 Katze 的 AGR 值将不与 den 的任何一个 AGR 值统一，也就是[GND='masc', NUM='sg', PER=3]或[NUM='pl', PER=3]。

9.4 小结

- 上下文无关文法的传统分类是原子符号。特征结构的一个重要的作用是捕捉精细的区分，否则将需要数量翻倍的原子类别。
- 通过使用特征值上的变量，我们可以表达文法产生式中的限制，允许不同的特征规格的实现可以相互依赖。
- 通常情况下，我们在词汇层面指定固定的特征值，限制短语中的特征值与它们的孩子中的对应值统一。
- 特征值可以是原子的或复杂的。原子值的一个特定类别是布尔值，按照惯例用[+/- feat]表示。
- 两个特征可以共享一个值（原子的或复杂的）。具有共享值的结构被称为重入。共享的值被表示为 AVM 中的数字索引（或标记）。
- 一个特征结构中的路径是一个特征的元组，对应从图的根开始的弧的序列上的标签。
- 两条路径是等价的，如果它们共享一个值。
- 包含的特征结构是偏序的。FS0 包含 FS1，当 FS0 比 FS1 更一般（较少信息）。
- 两种结构 FS0 和 FS1 的统一，如果成功，就是包含 FS0 和 FS1 的合并信息的特征结构 FS2。
- 如果统一在 FS 中指定一条路径 π，那么它也指定等效与 π 的每个路径 π'。
- 我们可以使用特征结构建立对大量广泛语言学现象的简洁的分析，包括动词子类别，倒装结构，无限制依赖结构和格支配。

9.5 进一步阅读

本章进一步的材料请参考 http://www.nltk.org/，包括特征结构的 HOWTO、特征文法、Earley 分析和文法测试套件。

协议现象的一个很好的介绍，请参阅(Corbett, 2006)。

理论语言学中最初使用特征的目的是捕捉语音的音素特性。例如：音/b/可能会被分解成结构[+labial, +voice]（[+唇，+语音]）。一个重要的动机是捕捉分割的类别之间的一般性，例如：/n/在任一+labial（唇）辅音前面被读作/m/。在乔姆斯基文法中，对一些现象，如：协议，使用原子特征是很标准的，原子特征也用来捕捉跨句法类别的概括，通过类比与

音韵。句法理论中使用特征的一个激进的扩展是广义短语结构语法（GPSG; [Gazdar et al., 1985]），特别是在使用带有复杂值的特征。

从计算语言学的角度来看，(Kay, 1985)提出语言的功能方面可以被属性-值结构的统一捕获，一个类似的方法由(Grosz & Stickel, 1983)在 PATR-II 形式体系中精心设计完成。词汇功能文法（Lexical-Functional grammar，LFG; [Kaplan & Bresnan, 1982]）的早期工作介绍了 f-structure 概念，它的主要目的是表示文法关系和与成分结构短语关联的谓词参数结构。（Shieber，1986）提供了研究基于特征文法方面的一个极好的介绍。

当研究人员试图为反面例子建模时，特征结构的代数方法的一个概念上的困难出现了。另一种观点，由(Kasper & Rounds, 1986) and (Johnson, 1988)开创，认为文法涉及结构功能的描述而不是结构本身。这些描述使用逻辑操作如合取相结合，而否定仅仅是特征描述上的普通的逻辑运算。这种面向描述的观点对 LFG 从一开始就是不可或缺的（Kaplan, 1989），也被中心词驱动短语结构文法的较高版本采用（HPSG; [Sag & Wasow, 1999]）。http://www.cl.uni-bremen.de/HPSG-Bib/上有 HPSG 文献的全面的参考书目。

本章介绍的特征结构无法捕捉语言信息中重要的限制。例如：有没有办法表达 NUM 的值只允许是 sg 和 pl，而指定[NUM=masc]是反常的。同样，我们不能说 AGR 的复合值必须包含特征 PER、NUM 和 GND 的指定，但不能包含如[SUBCAT=trans]这样的指定。**指定类型的特征结构**被开发出来弥补这方面的不足。一个早期的关于指定类型的特征结构的很好的总结是(Emele & Zajac, 1990)。一个形式化基础的更全面的检查可以在(Carpenter, 1992)中找到，(Copestake, 2002)重点关注为面向 HPSG 的方法实现指定类型的特征结构。

有很多著作是关于德语的基于特征文法框架上的分析的。(Nerbonne, Netter & Pollard, 1994)是这个主题的 HPSG 著作的一个好的起点，而(Müller, 2002)给出 HPSG 中的德语句法非常广泛和详细的分析。

(Jurafsky & Martin, 2008)的第 15 章讨论了特征结构、统一的算法和将统一整合到分析算法中。

9.6 练习

1. ○需要什么样的限制才能正确分析词序列，如 I am happy 和 she is happy 而不是*you is happy 或*they am happy？实现英语中动词 be 的现在时态范例的两个解决方案，首先以文法（8）作为起点，然后以文法（20）为起点。
2. ○开发例 9-1 中文法的变体，使用特种 COUNT 来区分下面显示的句子：

   (56) a. The boy sings.
   b. *Boy sings.

   (57) a. The boys sing.
   b. Boys sing.

   (58) a. The water is precious.
   b. Water is precious.
3. ○写函数 subsumes()判断两个特征结构 fs1 和 fs2 是否 fs1 包含 fs2。
4. ○修改（30）中所示的文法纳入特征 BAR 来处理短语投影。
5. ○修改例 9-4 中的德语文法，加入 9.3 节中介绍的子类别的处理。
6. ◑开发一个基于特征的文法，能够正确描述下面的西班牙语名词短语：

   (59) un cuadro hermos-o
   INDEF.SG.MASC picture beautiful-SG.MASC
   'a beautiful picture'

(60) un-os cuadro-s hermos-os
INDEF-PL.MASC picture-PL beautiful-PL.MASC
'beautiful pictures'

(61) un-a cortina hermos-a
INDEF-SG.FEM curtain beautiful-SG.FEM
'a beautiful curtain'

(62) un-as cortina-s hermos-as
INDEF-PL.FEM curtain beautiful-PL.FEM
'beautiful curtains'

7. ◑开发 earley_parser 的包装程序，只在输入序列分析出错时才输出跟踪。
8. ◑思考例 9-5 的特征结构。

例 9-5. 探索特征结构。

```
fs1 = nltk.FeatStruct("[A = ?x, B= [C = ?x]]")
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 = (1)[D = ?x], C = [E -> (1), F = ?x] ]")
fs6 = nltk.FeatStruct("[A = [D = d]]")
fs7 = nltk.FeatStruct("[A = [D = d], C = [F = [D = d]]]")
fs8 = nltk.FeatStruct("[A = (1)[D = ?x, G = ?x], C = [B = ?x, E -> (1)] ]")
fs9 = nltk.FeatStruct("[A = [B = b], C = [E = [G = e]]]")
fs10 = nltk.FeatStruct("[A = (1)[B = b], C -> (1)]")
```

在纸上计算下面的统一的结果是什么。（提示：你可能会发现绘制图结构很有用。）

a. fs1 and fs2
b. fs1 and fs3
c. fs4 and fs5
d. fs5 and fs6
e. fs5 and fs7
f. fs8 and fs9
g. fs8 and fs10

用 NLTK 检查你的答案。

9. ◑列出两个包含[A=?x, B=?x]的特征结构。
10. ◑忽略结构共享，给出一个统一两个特征结构的非正式算法。
11. ◑扩展例 9-4 中的德语语法，使它能处理所谓的动词第二顺位结构，如下所示：

(63) Heute sieht der Hund die Katze.

12. ◑同义动词的句法属性看上去略有不同(Levin,1993)。思考下面的动词 loaded、filled 和 dumped 的文法模式。你能写文法产生式处理这些数据吗？

(64) a. The farmer loaded the cart with sand
b. The farmer loaded sand into the cart
c. The farmer filled the cart with sand
d. *The farmer filled sand into the cart
e. *The farmer dumped the cart with sand
f. The farmer dumped sand into the cart

13. ●形态范例很少是完全正规的，矩阵中的每个单元的意义有不同的实现。例如：词位 w

alk 的现在时态词性变化只有两种不同形式：第三人称单数的 walks 和所有其他人称和数量的组合的 walk。一个成功的分析不应该额外要求 6 个可能的形态组合中有 5 个有相同的实现。设计和实施一个方法处理这个问题。

14. ●所谓的**核心特征**在父节点和核心孩子节点之间共享。例如：TENSE 是核心特征，在一个 VP 和它的核心孩子 V 之间共享。更多细节见(Gazdar et al., 1985)。我们看到的结构中大部分是核心结构——除了 SUBCAT 和 SLASH。由于核心特征的共享是可以预见的，它不需要在文法产生式中明确表示。开发一种方法自动计算核心结构的这种规则行为的比重。
15. ●扩展 NLTK 中特征结构的处理，允许统一值为链表的特征，使用这个来实现一个 HPSG 风格的子类别分析，核心类别的 SUBCAT 是它的补语的类别和它直接父母的 SUBCAT 值的连结。
16. ●扩展 NLTK 的特征结构处理，允许带未指定类别的产生式，例如：S[-INV] -> ?x S/?x。
17. ●扩展 NLTK 的特征结构处理，允许指定类型的特征结构。
18. ●挑选一些(Huddleston & Pullum,2002)中描述的文法结构，建立一个基于特征的文法计算它们的比例。

第 10 章 分析句子的意思

我们已经看到利用计算机的能力来处理大规模文本是多么有用。现在我们已经有了分析器和基于特征的文法，我们能否做一些类似分析句子的意思这样有用的事情？本章的目的是要回答下列问题：

1. 我们如何能表示自然语言的意思，使计算机能够处理这些表示？
2. 我们怎样才能将意思表示与无限的句子集合关联？
3. 我们怎样才能使用连接意思表示与句子的程序来存储知识？

一路上，我们将学习一些逻辑语义领域的形式化技术，看看如何用它们来查询存储了世间真知的数据库。

10.1 自然语言理解

查询数据库

假设有一个程序，让我们输入一个自然语言问题，返回给我们正确的答案：

(1) a. Which country is Athens in?

b. Greece.

写这样的一个程序有多难？我们可以使用到目前为止在这本书中遇到的技术吗？或者需要新的东西？在本节中，我们将看到解决特定领域的任务是相当简单的。但我们也将看到，要以一个更一般化的方式解决这个问题，就必须开辟一个全新的涉及意思的表示的理念和技术的框架。

因此，让我们开始先假设我们有关于城市和国家的结构化的数据。更具体的，我们将使用一个具有表 10-1 所示的前几行的数据库表。

表 10-1 所示的数据来自 80 聊天系统(Warren & Pereira, 1982)。人口数以千计算，注意在这些例子中使用的数据可以追溯到 1980 年代，在(Warren & Pereira, 1982)出版时，某些已经有些过时。

表 10-1. city_table：一个城市、国家和人口的表格

City	Country	Population
athens	greece	1368
bangkok	thailand	1178
barcelona	spain	1280
berlin	east_germany	3481
birmingham	united_kingdom	1112

从这个表格数据检索答案的最明显的方式是在一个如 SQL 这样的数据库查询语言中编写查询语句。

SQL（Structured Query Language，结构化查询语言）是为在关系数据库中检索和管理数据而设计的语言。如果你想了解更多有关 SQL 的信息，http://www.w3schools.com/sql/是一个方便的在线参考。

例如：执行查询（2）将得到值：'greece'：

(2) SELECT Country FROM city_table WHERE City = 'athens'

这条语句得到由所有 City 列的值是'athens'的数据行中 Country 列的所有值组成的结果的集合。

我们怎样才能使用英语得到与我们在查询系统中输入得到的相同的效果呢？第 9 章中描述的基于特征的文法形式可以很容易地从英语翻译到 SQL。文法 sql0.fcfg 说明如何将句子意思表示与句子分析串联组装。每个短语结构规则为特征 SEM 构建值作补充。你可以看到这些补充非常简单；在每一种情况下，我们对分割的子成分用字符串连接操作+来组成父成分的值。

```
>>> nltk.data.show_cfg('grammars/book_grammars/sql0.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[SEM=''] -> 'located'
P[SEM=''] -> 'in'
```

这使我们能够分析 SQL 查询：

```
>>> from nltk import load_parser
>>> cp = load_parser('grammars/book_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"
```

轮到你来：设置跟踪为最大，运行分析器，即 cp =load_parser('grammars/book_grammars/sql0.fcfg', trace=3)，研究当边被完整的加入到图表中时，如何建立 SEM 的值。

最后，我们在数据库 city.db 上执行查询，检索出一些结果：

```
>>> from nltk.sem import chat80
>>> rows = chat80.sql_query('corpora/city_database/city.db', q)
>>> for r in rows: print r[0],  ①
canton chungking dairen harbin kowloon mukden peking shanghai sian tientsin
```

由于每行 r 是一个单元素的元组，我们输出元组的成员，而不是元组本身①。

总结一下，我们已经定义了一个任务：计算机对自然语言查询做出反应，返回有用的数据。我们通过将英语的一个小的子集翻译成 SQL 来实现这个任务们可以说，我们的 NLTK 代码已经“理解”SQL，只要 Python 能够对数据库执行 SQL 查询，通过扩展，它也“理解”如“What cities are located in China”这样的查询。这相当于自然语言理解的例子能够从荷兰语翻译成英语。假设你是一个英语为母语的人，已经开始学习荷兰语。你的老师问你是否理解（3）的意思：

(3) Margrietje houdt van Brunoke.

如果你知道（3）中单个词的意思，并且知道如何将它们结合在一起组成整个句子的意思，你可能会说（3）的意思与 Margrietje loves Brunoke 相同。

一个观察者——让我们叫她 Olga——可能会将此作为你理解了（3）的意思的证据。但是，这将依赖于 Olga 自己懂英语。如果她不懂英语，那么你从荷兰语到英语的翻译就不能说明你理解了荷兰语。我们将很快回到这个问题。

文法 sql0.fcfg，连同 NLTK 的 Earley 分析器是实现从英语翻译到 SQL 的工具。这个文法够用吗？你已经看到整个句子的 SQL 翻译是由句子成分的翻译建立起来的。然而，这些成分的意思表示似乎没有很多的合理性。例如：如果我们看一下名词短语 Which cities 的分析，限定词和名词分别对应 SQL 片段 SELECT 和 City FROM city_table。但这两个都没有单独的符合语法的意思。

我们对语法还有另一种批评：我们“硬生生”的把一些数据库的细节加入其中。我们需要知道有关表（如：city_table）和字段的名称。但我们的数据库中可能确实存在相同的数据行但是用的是不同的表名和字段名，在这种情况下，SQL 查询就不能执行。同样，我们可以不同的格式存储我们的数据，如 XML，在这种情况下，检索相同的结果需要我们将我们的英语查询翻译成 XML 查询语言而不是 SQL。这些因素表明我们应该将英语翻译成比 SQL 更加抽象和通用的东西。

为了突出这一点，让我们考虑另一个英语查询及其翻译：

(4) 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

轮到你来：扩展文法 sql0.fcfg 使它能将（4a）转换为（4b），检查查询所返回的值。记得在聊天 80 数据库中数字以千为单位，因此（4b）中的 1000 代表一百万人口。

你可能会发现：最简单的方法是在一起处理之前扩展文法来处理查询 What cities have populations above 1,000,000。在你完成了这个任务之后，你可以比较你的方案与 NLTK 数据格式的 grammars/book_grammars/sql1.fcfg。

观察（4a）中 and 连接被翻译成（4b）中 SQL 对应的 AND。后者告诉我们从两个条件都为真的行中选择结果：Country 列的值是'china'，Population 列的值是大于 1000。and 的解释包含一个新的想法：它讲的是在某些特定情况下什么是真的，告诉我们在条件 s 中 Cond1 AND Cond2 为真，当且仅当条件 Cond1 在 s 中为真且 Cond2 也在 s 中为真。虽然这

并不是 and 在英语中的全部意思，但它具有很好的属性：独立于任何查询语言。事实上，我们已经给了它一个经典逻辑的标准解释。在下面的章节中，我们将探讨将自然语言的句子翻译成逻辑而不是如 SQL 这样的可执行查询语言的方法。逻辑形式的一个优势是它们更抽象，因此更通用。一旦我们翻译成了逻辑，只要我们想要，就可以再翻译成其他各种特殊用途的语言。事实上，大多说通过自然语言查询数据库的重要的尝试都是使用这种方法。

自然语言、语义和逻辑

我们一开始尝试捕捉（1a）的意思，通过将它翻译成计算机可以解释和执行的另一种语言，SQL，中的查询。但是，这仍然在回避问题的实质：翻译是否正确。从数据库查询走出来，我们注意到 *and* 的意思似乎可以用来指定在特定情况下语句是真或是假。不是将句子 S 从一种语言翻译到另一种，我们通过将 S 与现实的情况关联，尝试解释 *S* 是*关于*什么的，。让我们更进一步。假设有一种情况 s，其中有两个实体 Margrietje 和她最喜欢的娃娃 Brunoke。此外，两个实体之间有一种我们称之为*爱*的关系。如果你理解（3）的意思，那么你知道在情况 s 中它为真。你知道这个的部分原因是因为你知道 *Margrietje* 指的是 Margrietje，*Brunoke* 指的是 Brunoke，*houdt van* 指的是*爱*的关系。

我们引进了语义中的两个基本概念。第一个是*在某些情况下，陈述句非真即假*。第二个是名词短语和专有名词的定义*指的是世界上的东西*。所以（3）在 Margrietje 喜欢娃娃 Brunoke 这一情形中为真，如图 10-1 所示。

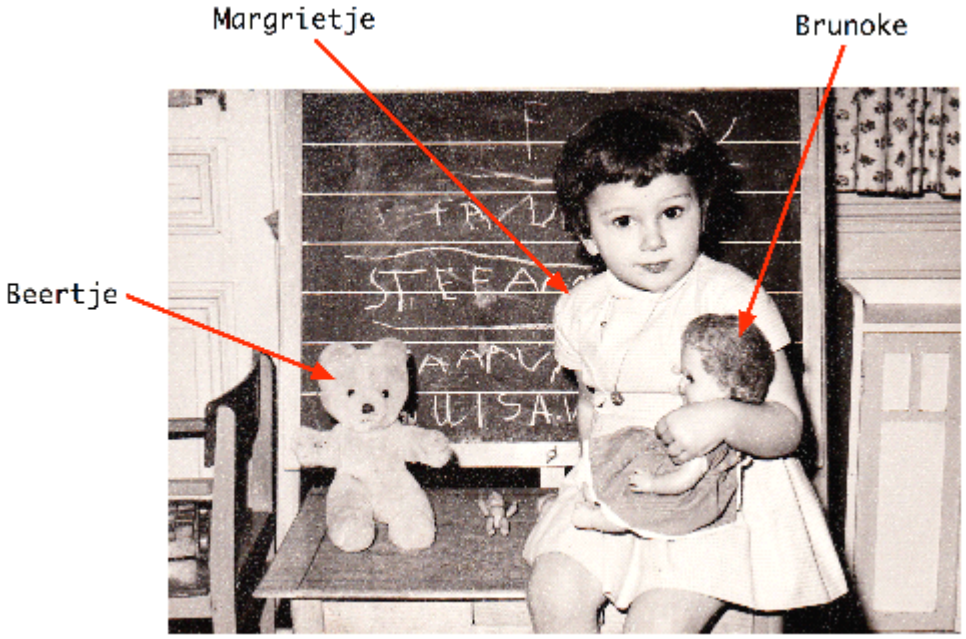


图 10.1：对 Margrietje 喜欢 Brunoke 的情形的描绘。

一旦我们采取了在一个情况下是真是假的概念，我们就有了一个强大的进行推理的工具。特别是，我们可以看到句子的集合，询问它们在某些情况下是否为真。例如：（5）中的句子可以都为真，而（6）和（7）中的则不能。换句话说，（5）中的句子是**一致的**，而（6）和（7）中的是**不一致的**。

(5) a. Sylvania is to the north of Freedonia.
b. Freedonia is a republic.

(6) a. The capital of Freedonia has a population of 9,000.
b. No city in Freedonia has a population of 9,000.

(7) a. Sylvania is to the north of Freedonia.

b. Freedonia is to the north of Sylvania.

我们选择有关 fictional countries（虚构的国家）（选自 Marx Brothers 1933 年的电影《Duck Soup》）的句子来强调你对这些例子的推理并不取决于真实世界中的真与假。如果你知道词 no 的意思，也知道一个国家的首都是一个位于该国的城市，那么你应该能够得出这样的结论：（6）中的两个句子是不一致的，不管 Freedonia 在哪里和它的首都有多少人口。也就是说，不存在使这两个句子同时都为真的情况。同样的，如果你知道 *to the north of* 所表达的关系是不对称的，那么你应该能够得出这样的结论：（7）中的两句话是不一致的。

从广义上讲，自然语言语义表示的基于逻辑的方法关注那些指导我们判断自然语言的一致性和不一致性的方面。设计一种逻辑语言的句法是为了使这些特征形式更明确。结果是如一致性这样的确定性属性往往可以简化成符号操作，也就是说，一种可以被计算机实施的任务。为了实现这种方法，我们首先要开发一种表示某种可能情况的技术。我们做的这些逻辑学家称之为模型。

一个句子集 W 的**模型**是某种情况的形式化表示，其中 W 中的所有句子都为真。表示模型通常的方式是集合论。段落的域 D（我们当前关心的所有实体）是个体的一个集合，而当集合从 D 建立，关系也被确立。让我们看一个具体的例子。我们的域 D 包括 3 个孩子，Stefan、Klaus 和 Evi，分别用 s、k 和 e 表示。记为 D = {s, k, e}。表达式 *boy* 是包含 Stefan 和 Klaus 的集合，表达式 *girl* 是包含 Evi 的集合，表达式 *is running* 是包含 Stefan 和 Evi 的集合。图 10-2 是这个模型的图形化描绘。

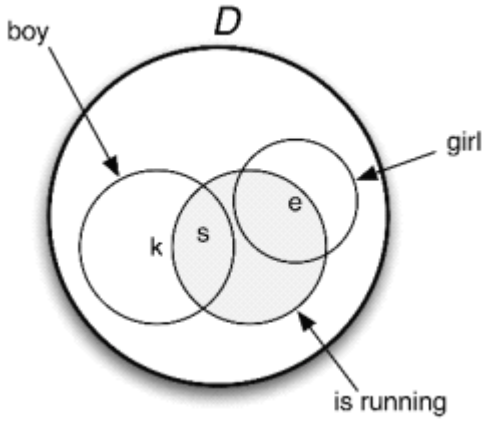


图 10-2. 一个模型图，包含一个域 D 及 D 的子集分别对应谓词 boy、girl 和 is running。

在本章后面，我们将使用模型来帮助评估英语句子的真假，并用这种方式来说明表示意思的一些方法。然而，在进入更多细节之前，让我们将从更广阔的角度进行讨论，回到我们在 1.5 节简要提到过的主题。一台计算机可以理解句子的意思吗？我们该如何判断它是否能理解？这类似与问：“计算机能思考吗？”阿兰·图灵提出的著名的回答是：通过检查计算机与人类进行理智的对话的能力（Turing, 1950）。假设你有一个与人聊天的会话和一个与计算机聊天的会话，但一开始你并不知道哪个是哪个。在与它们两个聊天后，如果你不能识别对方哪一个是计算机，那么计算机就成功地模仿了人类。如果一台计算机成功的被当做人类通过了这个“模仿游戏”（或“图灵测试”，这是它是俗称），那么对于图灵来说，就可以说计算机能思考，可以说它具有了智能。所以图灵从侧面回答了这个问题，不是检查计算机的内部状态，而是检查它的行为，作为具有智能的证据。同样的道理，我们认为要说一台计算机懂英语，只需要它的行为表现看上去它懂。这里最重要的是不是图灵模仿游戏的细节，而是以可观察的行为为依据来判断自然语言理解能力的想法。

10.2 命题逻辑

设计一种逻辑语言的目的是使推理形式更明确。因此，它可以在自然语言中确定一组句子是否是一致的。作为这种方法的一部分，我们需要开发一个句子φ的逻辑表示，它能形式化的捕捉φ为真的条件。我们先从一个简单的例子开始：

(8) [Klaus chased Evi] and [Evi ran away].

让我们分别用φ和ψ替代（8）中的两个子句，并用&替代对应的英语词 and 的逻辑操作：φ & ψ。这种结构是（80）的**逻辑形式**。

命题逻辑使我们能只表示语言结构的对应与句子的特定连接词的那些部分。刚才我们看了 and。其他的连接词还有 not、or 和 if...，then...。命题逻辑形式中，这些连接词的对应形式有时叫做**布尔运算符**。命题逻辑的基本表达式是**命题符号**，通常写作 P、Q、R 等。表示布尔运算符的约定很多。由于我们将重点探索 NLTK 中的逻辑表示方式，所以将使用下列 ASCII 版本的运算符：

```
>>> nltk.boolean_ops()
negation            -
conjunction         &
disjunction         |
implication         ->
equivalence         <->
```

从命题符号和布尔运算符，我们可以建立命题逻辑的**规范公式**的无限集合（或简称公式）。首先，每个命题字母是一个公式。然后，如果φ是一个公式，那么-φ也是一个公式。如果φ和ψ是公式，那么(φ & ψ)、(φ | ψ)、(φ -> ψ)以及(φ <-> ψ)也是公式。

表 10-2 指定了包含这些运算符的公式为真的条件。和以前一样，我们使用φ和ψ作为句子中的变量，iff 作为 if and only if（当且仅当）的缩写。

表 10-2. 命题逻辑的布尔运算符的真值条件

布尔运算符	真值条件		
非(*否定的情况*)	在 s 中-φ为真	iff	在 s 中φ为真
与(*and*)	在 s 中(φ & ψ)为真	iff	在 s 中φ为真并且在 s 中ψ为真
或(*or*)	在 s 中(φ | ψ)为真	iff	在 s 中φ为真或者在 s 中ψ为真
蕴含(*if ..., then ...*)	在 s 中(φ -> ψ)为真	iff	在 s 中φ为假或者在 s 中ψ为真
等价(*if and only if*)	在 s 中(φ <-> ψ)为真	iff	在 s 中φ和ψ同时为真或者同时为假

这些规则通常是简单的，虽然蕴含的真值条件违反了很多我们通常关于英语中条件句的直觉。形式(P -> Q)的公式是为假只有当 P 为真并且 Q 为假时。如果 P 为假（比如说，P 对应 The moon is made of green cheese）而 Q 为真（比如说，Q 对应 Two plus two equals four）那么 P -> Q 的结果为真。

NLTK 的 LogicParser()将逻辑表达式分析成**表达式**的各种子类：

```
>>> lp = nltk.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)>
```

从计算的角度来看，逻辑给了我们进行推理的一个重要工具。假设你表达：Freedonia is not to the north of Sylvania（Freedonia 不在 Sylvania 北边），而你给出理由：Sylvania is to the north of Freedonia（Sylvania 在 Freedonia 北边）。在这种情况下，你已经给出了一个**论证**。句子：Sylvania is to the north of Freedonia 是论证的**假设**，而 Freedonia is not to the north of Sylvania 是**结论**。从假设一步一步推到结论，被称为**推理**。通俗地说，就是我们以在结论前面写 therefore 这样的格式写一个论证。

(9) Sylvania is to the north of Freedonia.

Therefore, Freedonia is not to the north of Sylvania.

一个论证，如果没有它的所有的前提都为真而结论为假的情况，那么它是**有效**的。

现在，（9）的有效性关键取决于短语 to the north of 的含义，特别的，它实际上是一个非对称的关系：

(10) if x is to the north of y then y is not to the north of x.

不幸的是，在命题逻辑中，我们不能表达这样的规则：我们能用的最小的元素是原子命题，我们不能“向里看”来谈论个体 x 和 y 之间的关系。在这种情况下，我们可以做的最好的是捕捉不对称的一个特定案例。让我们使用命题符号 SnF 表示 Sylvania is to the north of Freedonia，用 FnS 表示 Freedonia is to the north of Sylvania。要说 Freedonia is not to the north of Sylvania，我们写成-FnS。也就是说，我们将 not 当做短语 it is not the case that 的等价，用一元布尔运算符-来翻译这个。分别替代（10）中的 x 和 y 为 Sylvania 和 Freedonia，于是我们得到一个蕴含，可以写作：

(11) SnF -> -FnS

给出一个完整的论证版本会怎样？我们将提到（9）的第一句话为两个命题逻辑公式：Snf 和（11）中的蕴含，它表示（相当贫乏的）我们的背景知识：to the north of 的意思。我们将写[A1, ..., An] / C 表示一个从假设[A1, ..., An]得出结论 C 的论证。这使得论证（9）可以表示为：

(12) [SnF, SnF -> -FnS] / -FnS

这是一个有效的论证：如果 SnF and SnF -> -FnS 在情况 s 中都为真，那么-FnS 也一定在 s 中为真。相反，如果 FnS 为真，这将与我们的理解冲突：在任何可能的情况下两个事物不能同时在对方的北边。同样的，链表[SnF, SnF -> -FnS, FnS] 是不一致的——这些句子不能全为真。

参数可以通过使用证明系统来测试“句法有效性”。在稍后的 10.3 节中我们会再多讲一些这个。NLTK 中的 inference 模块通过一个第三方定理证明器 Prover9 的接口，可以进行逻辑证明。推理机制的输入首先必须用 LogicParser()分析成逻辑表达式。

```
>>> lp = nltk.LogicParser()
>>> SnF = lp.parse('SnF')
>>> NotFnS = lp.parse('-FnS')
>>> R = lp.parse('SnF -> -FnS')
>>> prover = nltk.Prover9()
>>> prover.prove(NotFnS, [SnF, R])
True
```

这里有另一种方式可以看到结论如何得出。SnF -> -FnS 在语义上等价与-SnF | -FnS，其中|是对应与 or 的二元运算符。在一般情况下，ϕ | ψ在条件 s 中为真，要么ϕ在 s 中为真，要么ψ在 s 中为真。现在，假设 SnF 和-SnF | -FnS 都在 s 中为真。如果 SnF 为

真，那么-SnF不可能也为真。经典逻辑的一个基本假设是：一个句子在一种情况下不能同时为真和为假。因此，-FnS必须为真。

回想一下，我们解释相对于一个模型的一种逻辑语言的句子，它们是这个世界的一个非常简化的版本。一个命题逻辑的模型需要为每个可能的公式分配值True或False。我们一步步的来做这个：首先，为每个命题符号分配一个值，然后确定布尔运算符的含义（即表10-2）和运用它们到这些公式的组件的值，来计算复杂的公式的值。估值是从逻辑的基本符号映射到它们的值。下面是一个例子：

```
>>> val = nltk.Valuation([('P', True), ('Q', True), ('R', False)])
```

我们使用一个配对的链表初始化一个估值，每个配对由一个语义符号和一个语义值组成。所产生的对象基本上只是一个字典，映射逻辑符号（作为字符串处理）为适当的值。

```
>>> val['P']
True
```

正如我们稍后将看到的，我们的模型需要稍微更加复杂些，以便处理将在下一节中讨论的更复杂的逻辑形式；暂时的，在下面的声明中先忽略参数dom和g。

```
>>> dom = set([])
>>> g = nltk.Assignment(dom)
```

现在，让我们用val初始化模型m：

```
>>> m = nltk.Model(dom, val)
```

每一个模型都有一个evaluate()方法，可以确定逻辑表达式，如命题逻辑的公式，的语义值；当然，这些值取决于最初我们分配给命题符号如P、Q和R的真值。

```
>>> 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
```

轮到你来：做实验为不同的命题逻辑公式估值。模型是否给出你所期望的值？

到目前为止，我们已经将我们的英文句子翻译成命题逻辑。因为我们只限于用字母如P和Q表示原子句子，不能深入其内部结构。实际上，我们说将原子句子分成主语、宾语和谓词并没有语义上的好处。然而，这似乎是错误的：如果我们想形式化如（9）这样的论证，就必须要能“看到里面”基本的句子。因此，我们将超越命题逻辑到一个更有表现力的东西，也就是一阶逻辑。这正是我们下一节要讲的。

10.3 一阶逻辑

本章的剩余部分，我们将通过翻译自然语言表达式为一阶逻辑来表示它们的意思。并不是所有的自然语言语义都可以用一阶逻辑表示。但它是计算语义的一个不错的选择，因为它具有足够的表现力来表达语义的很多方面，而且另一方面，有出色的现成系统可用于开展一阶逻辑自动推理。

下一步我们将描述如何构造一阶逻辑公式，然后是这样的公式如何用来评估模型。

句法

一阶逻辑保留所有命题逻辑的布尔运算符，但它增加了一些重要的新机制。首先，命题被分析成谓词和参数，这将我们与自然语言的结构的距离拉近了一步。一阶逻辑的标准构造规则承认以下**术语**：独立变量和独立常量、带不同数量的**参数**的**谓词**。例如：Angus walks 可以被形式化为 walk(angus)，Angus sees Bertie 可以被形式化为 see(angus, bertie)。我们称 walk 为一元谓词，see 为二元谓词。作为谓词使用的符号不具有内在的含义，虽然很难记住这一点。回到我们前面的一个例子，（13a）和（13b）之间没有逻辑区别。

(13) a. love(margrietje, brunoke)

b. houden_van(margrietje, brunoke)

一阶逻辑本身没有什么实质性的关于词汇语义的表示——单个词的意思——虽然一些词汇语义理论可以用一阶逻辑编码。原子谓词如 *see(angus, bertie)*在情况 s 中是真还是假不是一个逻辑的问题，而是依赖于特定的估值，即我们为常量 *see*、*angus* 和 *bertie* 选择的值。出于这个原因，这些表达式被称为**非逻辑常量**。相比之下，**逻辑常量**（如布尔运算符）在一阶逻辑的每个模型中的解释总是相同的。

我们应该在这里提到：有一个二元谓词具有特殊的地位，它就是等号，如在 *angus = aj* 这样的公式中的等号。等号被视为一个逻辑常量，因为对于单独的术语 t1 和 t2，公式 t1 = t2 为真当且仅当 t1 和 t2 是指同一个实体。

检查一阶逻辑表达式的语法结构往往是有益的，这样做通常的方式是为表达式指定**类型**。下面是 Montague 文法的约定，我们将使用**基本类型**：e 是实体类型，而 t 是公式类型，即有真值的表达式的类型。给定这两种基本类型，我们可以形成函数表达式的**复杂类型**。也就是说，给定任何类型 σ 和 τ，<σ,τ>是一个对应与从'σ things'到'τ things'的函数的复杂类型。例如：<e, T>是从实体到真值，即一元谓词，的表达式的类型。可以调用 LogicParser 来进行类型检查。

```
>>> tlp = nltk.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,?>
```

为什么我们在这个例子的结尾看到<e,?>呢？虽然类型检查器会尝试推断出尽可能多的类型，在这种情况下，它并没有能够推断出 walk 的类型，所以其结果的类型是未知的。虽然我们期望 walk 的类型是<e, t>，迄今为止类型检查器知道的，在这个上下文中可能是一些其他类型，如<e, e>或<e, <e, t>>。要帮助类型检查器，我们需要指定一个**信号**，作为一个字典来实施，明确的与非逻辑常量类型关联：

```
>>> sig = {'walk': '<e, t>'}
>>> parsed = tlp.parse('walk(angus)', sig)
>>> parsed.function.type
<e,t>
```

一种二元谓词的类型<e, <e, t>>。虽然这是先组合类型 e 的一个参数成一个一元谓词的类型，我们可以用二元谓词的两个参数直接组合来表示二元谓词。例如：在 Angus sees Cyril 的翻译中谓词 see 会与它的参数结合得到结果 see(angus, cyril)。

在一阶逻辑中，谓词的参数也可以是独立变量，如 x、y 和 z。在 NLTK 中，我们采用的惯例：e 类型的变量都是小写。独立变量类似与人称代词，如 he、she 和 it，其中我们为了弄清楚它们的含义需要知道它们使用的上下文。解释（14）中的代名词的方法之一是指向上下文中相关的个体。

(14) He disappeared.

另一种方法是为代词 he 提供文本中的先行词，例如：通过指出（15a）在（14）之前。在这里，我们说 he 与名词短语 Cyril **指称相同**。在这个上下文中，(14)与(15b)是语义上是等价的。

(15) a. Cyril is Angus's dog.

b. Cyril disappeared.

相比之下，思考(16a)中出现的 he。在这种情况下，它受不确定的 NP ：a dog 的**约束**，这是一个与共指关系不同的关系。如果我们替换代词 he 为 a dog，结果(16b)就在语义上就不等效与（16a）。

(16) a. Angus had a dog but he disappeared.

b. Angus had a dog but a dog disappeared.

对应与（17a），我们可以构建一个**开放公式**（17b），变量 x 出现了两次（我们忽略了时态，以简化论述。）

(17) a. He is a dog and he disappeared.

b. dog(x) & disappear(x)

通过在（17b）前面指定一个**存在量词**∃x（“存在某些 x”），我们可以**绑定**这些变量，如在（18a）中，它的意思是（18b），或者更习惯写法（18c）。

(18) a. ∃x.(dog(x) & disappear(x))

b. At least one entity is a dog and disappeared.

c. A dog disappeared.

下面是(18a)在 NLTK 中的表示：

(19) exists x.(dog(x) & disappear(x))

除了存在量词，一阶逻辑为我们提供了**全称量词**∀X（“对所有 x”），如（20）所示。

(20) a. ?x.(dog(x) → disappear(x))

b. Everything has the property that if it is a dog, it disappears.

c. Every dog disappeared.

下面是(20a)在 NLTK 中的表示：

(21) all x.(dog(x) -> disappear(x))

虽然（20a）是（20c）的标准一阶逻辑翻译，其真值条件不一定是你所期望的。公式表示如果某些 x 是狗，那么 x 消失——但它并没有说有狗存在。在没有狗存在的情况下，（20a）仍然会为真。（请记住，当 P 为假时，(P -> Q)为真。）现在你可能会说，所有的狗都消失是以狗的存在为前提条件的，逻辑形式化表示的是完全错误的。但也可能找到其他的例子，没有这样一个前提。例如：我们也许可以解释 Python 表达式 astring.replace('ate', '8') 是替代 astring 中出现的所有'ate'为'8'，即使事实上'ate'可能没有出现（表 3-2）。

我们已经看到了一些量词约束变量的例子。下面的公式中会发生什么？

((exists x. dog(x)) -> bark(x))

量词 exists x 的范围是 dog(x)，所以 bark(x)中 x 的出现是不受限制的。因此，它可

以被其他一些量词约束，例如：

下面公式中的 all x：

all x.((exists x. dog(x)) -> bark(x))

在一般情况下，变量 x 在公式 φ 中出现是**自由的**，如果它在 φ 中没有出现在 all x 或 exists x 范围内。相反，如果 x 在公式 φ 中是**受限**的，那么它在 all x.φ 和 exists x.φ 限制范围内。如果公式中所有变量都是受限**的**，那么我们说这个公式是**封闭的**。

在此之前我们提到过，NLTK 中的 LogicParser 的 parse()方法返回 Expression 类的对象。这个类的每个实例 expr 都有 free()方法，返回一个在 expr 中自由的变量的集合。

```
>>> lp = nltk.LogicParser()
>>> lp.parse('dog(cyril)').free()
set([])
>>> lp.parse('dog(x)').free()
set([Variable('x')])
>>> lp.parse('own(angus, cyril)').free()
set([])
>>> lp.parse('exists x.dog(x)').free()
set([])
>>> lp.parse('((some x. walk(x)) -> sing(x))').free()
set([Variable('x')])
>>> lp.parse('exists x.own(y, x)').free()
set([Variable('y')])
```

一阶定理证明

回顾一下我们较早前在（10）中提出的 to the north of 上的限制，：

(22) if x is to the north of y then y is not to the north of x.

我们观察到命题逻辑不足以表示与二元谓词相关的概括，因此，我们不能正确的捕捉论证：Sylvania is to the north of Freedonia. Therefore, Freedonia is not to the north of Sylvania。

毫无疑问，用一阶逻辑形式化这些规则是很理想的：

all x. all y.(north_of(x, y) -> -north_of(y, x))

更妙的是，我们可以进行自动推理来证明论证的有效性。

定理证明在一般情况下是为了确定我们要证明的公式（**证明目标**）是否可以由一个有限序列的推理步骤从一个假设的公式列表派生出来。我们写作 A g，其中 A 是一个假设的列表（可能为空），g 是证明目标。我们将用 NLTK 中的定理证明接口 Prover9 来演示这个。首先，我们分析所需的证明目标①和两个②③。然后我们创建一个 Prover9 实例④，并在目标和给定的假设列表上调用它的 prove()⑤。

```
>>> NotFnS = lp.parse('-north_of(f, s)')  ①
>>> SnF = lp.parse('north_of(s, f)')  ②
>>> R = lp.parse('all x. all y. (north_of(x, y) -> -north_of(y, x))')  ③
>>> prover = nltk.Prover9()  ④
>>> prover.prove(NotFnS, [SnF, R])  ⑤
True
```

令人高兴的是，定理证明器证明我们的论证是有效的。相反，它得出结论：不能从我们

的假设推到出 north_of(f, s)：

```
>>> FnS = lp.parse('north_of(f, s)')
>>> prover.prove(FnS, [SnF, R])
False
```

一阶逻辑语言总结

我们将借此机会重新表述前面的命题逻辑的语法规则，并添加量词的形式化规则；所有这些一起组成一阶逻辑的句法。此外，我们会明确相关表达式的类型。我们将采取约定：<e^n, t>是一种由 n 个类型为 e 的参数组成产生一个类型为 t 的表达式的谓词的类型。在这种情况下，我们说 n 是谓词的**元数**。

1. 如果 P 是类型<en, t>的谓词，α1，… αn 是 e 类型的术语，那么 P（α1，… αn）的类型是 t。
2. 如果α和β都是 e 类型，那么（α=β）和（α!=β）是 t 类型。
3. 如果φ是 t 类型，那么是-φ也是 t 类型。
4. 如果φ和ψ是 t 类型，那么（φ&ψ）、（φ|ψ）、（φ -> ψ）和（φ<->ψ）也是 t 类型。
5. 如果φ是 t 类型，x 是类型为 e 的变量，那么 exists x.φ和 all x.φ也是 t 类型。

表 10-3 总结了 logic 模块的新的逻辑常量，以及 Expressions 模块的两个方法。

表 10-3. 一阶逻辑所需的新的逻辑关系和运算符总结

示例	描述
=	等于
!=	不等于
exists	存在量词
all	全称量词

真值模型

我们已经看到了一阶逻辑句法，在第 10.4 节我们将考察将英语翻译成一阶逻辑的任务。然而，正如我们在第 10.1 节中提到的，只有我们赋予一阶逻辑的句子以含义，才能进一步做下去。换句话说，我们需要给出一阶逻辑的真值条件的语义。从计算语义学的角度来看，采用这种方法能走多远是有明显的界限的。虽然我们要谈的是句子在某种情况下为真或为假，我们只能在电脑中以符号的方式表示这些情况。尽管有这些限制，通过在 NLTK 解码模块仍然可以获得较清晰的真值条件语义。

给定一阶逻辑语言 L，L 的模型 M 是一个<D, Val>对，其中 D 是一个非空集合，称为模型的域，Val 是一个函数，称为估值函数，它按如下方式从 D 中分配值给 L 的表达式：

1. 对于 L 中每一个独立常量 c，Val(c)是 D 中的元素。
2. 对于每一个元数 n≥0 的谓词符号 P，Val(P)是从 Dn 到{True, False}的函数。（如果 P 的元数为 0，则 Val(P)是一个简单的真值，P 被认为是一个命题符号。）

对于（2），如果 P 的元数是 2，然后 Val(P)将是一个从 D 的元素的配对到{True, False}的函数。我们将在 NLTK 中建立的模型中采取更方便的替代品，其中 Val(P)是一个配对的集合 S，定义如下：

(23) S = {s | f(s) = True}

这样的 f 被称为 S 的**特征函数**（如在进一步阅读中讨论过的）。

NLTK 的语义关系可以用标准的集合论方法表示：作为元组的集合。例如：假设我们有一个域包括 Bertie、Olive 和 Cyril，其中 Bertie 是男孩，Olive 是女孩，而 Cyril 是小狗。为了方便记述，我们用 b、o 和 c 作为模型中相应的标签。我们可以声明域如下：

```
>>> dom = set(['b', 'o', 'c'])
```

我们使用工具函数 parse_valuation()将“符号 => 值”形式的字符串序列转换成一个 Valuation 对象。

```
>>> v = """
... bertie => b
... olive => o
... cyril => c
... boy => {b}
... girl => {o}
... dog => {c}
... walk => {o, c}
... see => {(b, o), (c, b), (o, c)}
... """
>>> val = nltk.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',)])}
```

根据这一估值，see 的值是一个元组的集合，包含：Bertie 看到 Olive、Cyril 看到 Bertie 和 Olive 看到 Cyril。

轮到你来：模仿图 10-2 绘制一个图，描述域 dom 和相应的每个一元谓词的集合。

你可能已经注意到，我们的一元谓词（即 boy、girl、dog）也是以单个元组的集合而不是个体的集合出现的。这使我们能够方便的统一处理任何元数的关系。

一个形式为 P(T1, ... Tn)的谓词，其中 P 是 n 元的，为真的条件是对应于(T1, ... Tn)的值的元组属于 P 的值的元组的集合。

```
>>> ('o', 'c') in val['see']
True
>>> ('b',) in val['boy']
True
```

独立变量和赋值

在我们的模型，上下文的使用对应的是为变量**赋值**。这是一个从独立变量到域中实体的映射。赋值使用构造函数 Assignment，它也以论述的模型的域为参数。我们无需实际输入任何绑定，但如果我们要这样做，它们是以(变量,值)的形式来绑定，类似于我们前面看到的估值。

```
>>> g = nltk.Assignment(dom, [('x', 'o'), ('y', 'c')])
>>> g
{'y': 'c', 'x': 'o'}
```

此外，还可以使用 print()查看赋值，使用与逻辑教科书中经常出现的符号类似的符号：

```
>>> print g
g[c/y][o/x]
```

现在让我们看看如何为一阶逻辑的原子公式估值。首先，我们创建了一个模型，然后调用 evaluate()方法来计算真值。

```
>>> m = nltk.Model(dom, val)
>>> m.evaluate('see(olive, y)', g)
True
```

这里发生了什么？我们正在为一个公式估值，类似与我们前面的例子：see(olive, cyril)。然而，当解释函数遇到变量 y 时，不是检查 val 中的值，它在变量赋值 g 中查询这个变量的值：

```
>>> g['y']
'c'
```

由于我们已经知道 o 和 c 在 see 关系中表示的含义，所以 true 值是我们所期望的。在这种情况下，我们可以说赋值 g **满足**公式 see(olive, y)。相比之下，下面的公式相对 g 的评估结果为 False（检查为什么会是你看到的这样）。

```
>>> m.evaluate('see(y, x)', g)
False
```

在我们的方法中（虽然不是标准的一阶逻辑），变量赋值是部分的。例如：g 中除了 x 和 y 没有其它变量。方法 purge()从清除一个赋值中所有的绑定。

```
>>> g.purge()
>>> g
{}
```

如果我们现在尝试为公式，如 see(olive, y)，相对于 g 估值，就像试图解释一个包含一个 him 的句子，我们不知道 him 指什么。在这种情况下，估值函数未能提供一个真值。

```
>>> m.evaluate('see(olive, y)', g)
'Undefined'
```

由于我们的模型已经包含了解释布尔运算的规则，任意复杂的公式都可以组合和评估。

```
>>> m.evaluate('see(bertie, olive) & boy(bertie) & -walk(bertie)', g)
True
```

确定模型中公式的真假的一般过程称为**模型检查**。

量化

现代逻辑的关键特征之一就是变量满足的概念可以用来解释量化的公式。让我们用（24）作为一个例子。

(24) exists x.(girl(x) & walk(x))

什么时候为真？让我们想想我们的域，即在 dom 中的所有个体。我们要检查这些个体中是否有属性是女孩并且这种走路的。换句话说，我们想知道 dom 中是否有某个 u 使 g[u/x]满足开放的公式(25)。

(25) girl(x) & walk(x)

思考下面的：

```
>>> m.evaluate('exists x.(girl(x) & walk(x))', g)
True
```

在这里 evaluate()返回 True，因为 dom 中有某些 u 通过绑定 x 到 u 的赋值满足（25）。事实上，o 是这样一个 u：

```
>>> m.evaluate('girl(x) & walk(x)', g.add('x', 'o'))
True
```

NLTK 中提供了一个有用的工具：satisfiers()方法。它返回满足开放公式的所有个体的集合。该方法的参数是一个已分析的公式、一个变量和一个赋值。下面是几个例子：

```
>>> 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'])
```

想一想为什么 fmla2 和 fmla3 是那样的值，这是非常有用。 ->的真值条件的意思是 fmla2 等价于-girl(x) | walk(x)，要么不是女孩要么没有步行的个体满足条件。因为 b(Bertie)和 c(Cyril)都不是女孩，根据模型 m，它们都满足整个公式。当然 o 也满足公式，因为 o 两项都满足。现在，因为话题的域的每一个成员都满足 fmla2，相应的全称量化公式也为真。

```
>>> m.evaluate('all x.(girl(x) -> walk(x))', g)
True
```

换句话说，一个全称量化公式∀x.ϕ关于 g 为真，只有对每一个 u，ϕ关于 g[u/x]为真。

轮到你来：先用笔和纸，然后用 m.evaluate()，尝试弄清楚 all x.(girl(x) & walk(x))和 exists x.(boy(x) -> walk(x))的真值。确保你能理解为什么它们得到这些值。

量词范围歧义

当我们给一个句子的形式化表示两个量词时，会发生什么？

(26) Everybody admires someone.

用一阶逻辑有（至少）两种方法表示（26）：

(27) a. all x.(person(x) -> exists y.(person(y) & admire(x,y)))

b. exists y.(person(y) & all x.(person(x) -> admire(x,y)))

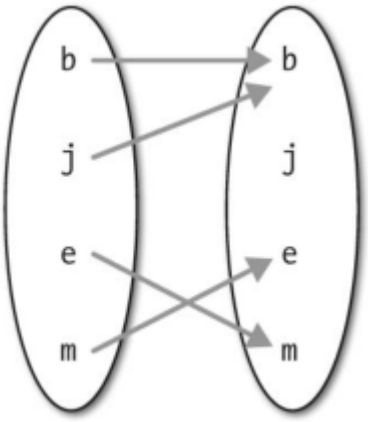
这两个我们都能用吗？答案是肯定的，但它们的含义不同。（27b）在逻辑上强于（27a）：它声称只有一个人，也就是 Bruce，被所有人钦佩。而（27a）只要其对于每一个 u 我们可以找到 u 钦佩的一些人 u'；但每次找到的人 u'可能不同。我们使用术语量化**范围**来区分（27a）和（27b）。首先，∀的范围比∃广，而在（27b）中范围顺序颠倒了，所以现在我们有两种方式表示（26）的意思，它们都相当合理。换句话说，我们称（26）关于量词范围有歧义，（27）中的公式给我们一种使这两个读法明确的方法。然而，我们不只是对与（26）相关联的两个不同的表示感兴趣；我们也想要显示模型中的两种表述是如何导致不同的真值条件的细节。

为了更仔细的检查歧义，让我们对估值做如下修正：

```
>>> 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 = nltk.parse_valuation(v2)
```

admire 关系可以使用（28）所示的映射图进行可视化。

(28)



（28）中，两个个体 x 和 y 之间的箭头表示 x 钦佩 y。因此，j 和 b 都钦佩 b（布鲁斯很虚荣），而 e 钦佩 m 且 m 钦佩 e。在这个模型中，公式（27a）为真而（27b）为假。探索这些结果的方法之一是使用 Model 对象的 satisfiers()的方法。

```
>>> dom2 = val2.domain
>>> m2 = nltk.Model(dom2, val2)
>>> g2 = nltk.Assignment(dom2)
```

```
>>> fmla4 = lp.parse('(person(x) -> exists y.(person(y) & admire(x, y)))')
>>> m2.satisfiers(fmla4, 'x', g2)
set(['a', 'c', 'b', 'e', 'j', 'm'])
```

这表明 fmla4 包含域中每一个个体。相反，思考下面的公式 fmla5；没有满足 y 的值。

```
>>> fmla5 = lp.parse('(person(y) & all x.(person(x) -> admire(x, y)))')
>>> m2.satisfiers(fmla5, 'y', g2)
set([])
```

也就是说，没有大家都钦佩的人。看看另一个开放的公式 fmla6，我们可以验证有一个人，即 Bruce，它被 Julia 和 Bruce 都钦佩。

```
>>> fmla6 = lp.parse('(person(y) & all x.((x = bruce | x = julia) -> admire(x, y)))')
>>> m2.satisfiers(fmla6, 'y', g2)
set(['b'])
```

轮到你来：基于 m2 设计一个新的模型，使（27a）在你的模型中为假；同样的，设计一个新的模型使（27b）为真。

模型的建立

我们一直假设我们已经有了一个模型，并要检查模型中的一个句子的真值。相比之下，模型的建立是给定一些句子的集合，尝试创造一种新的模型。如果成功，那么我们知道集合是一致的，因为我们有模型的存在作为证据。

我们通过创建 Mace()的一个实例并调用它的 build_model()方法来调用 Mace4 模式产生器，与调用 Prover9 定理证明器类似的方法。一种选择是将我们的候选的句子集合作为假设，保留目标为未指定。下面的交互显示了[a, c1]和[a, c2]都是一致的链表，因为 Mace 成功的为它们都建立了一个模型，而[c1,c2]不一致。

```
>>> a3 = lp.parse('exists x.(man(x) & walks(x))')
>>> c1 = lp.parse('mortal(socrates)')
>>> c2 = lp.parse('-mortal(socrates)')
>>> mb = nltk.Mace(5)
>>> print mb.build_model(None, [a3, c1])
True
>>> print mb.build_model(None, [a3, c2])
True
>>> print mb.build_model(None, [c1, c2])
False
```

我们也可以使用模型建立器作为定理证明器的辅助。假设我们正试图证明 A　g，即 g 是假设 A = [a1,a2, ..., an]的逻辑派生。我们可以同样的输入提供给 Mace4，模型建立器将尝试找出一个反例，就是要表明 g 不遵循从 A。因此，给定此输入，Mace4 将尝试为假设 A 连同 g 的否定找到一个模型，即链表 A' = [a1, a2, ..., an, -g]。如果 g 从 S 不能证明出来，那么 Mace4 会返回一个范例，比 Prover9 更快的得出结论：无法找到所需的证明。相反，如果 g 从 S 可以证明出来，Mace4 可能要花很长时间不能成功地找到一个反例模型，最终会放弃。

让我们思考一个具体的方案。我们的假设是链表[There is a woman that every man loves, Adam is a man, Eve is a woman]。我们的结论是 Adam loves Eve。M

ace4 能找到使假设为真而结论为假的模型吗？在下面的代码，我们使用 MaceCommand() 检查已建立的模型。

```
>>> a4 = lp.parse('exists y. (woman(y) & all x. (man(x) -> love(x,y)))')
>>> a5 = lp.parse('man(adam)')
>>> a6 = lp.parse('woman(eve)')
>>> g = lp.parse('love(adam,eve)')
>>> mc = nltk.MaceCommand(g, assumptions=[a4, a5, a6])
>>> mc.build_model()
True
```

因此答案是肯定的：Mace4 发现了一个反例模型，其中 Adam 爱某个女人而不是 Eve。但是，让我们细看 Mace4 的模型，转换成我们用来估值的格式：

```
>>> print mc.valuation
{'C1': 'b',
 'adam': 'a',
 'eve': 'a',
 'love': set([('a', 'b')]),
 'man': set([('a',)]),
 'woman': set([('a',), ('b',)])}
```

这个估值的一般形式应是你熟悉的：它包含了一些单独的常量和谓词，每一个都有适当类型的值。可能令人费解的是 C1。它是一个“Skolem 常量”，模型生成器作为存在量词的表示引入的。也就是说，模型生成器遇到 a4 里面的 exists y，它知道，域中有某个个体 b 满足 a4 中的开放公式。然而，它不知道 b 是否也是它的输入中的某个地方的一个独立常量的标志，所以它为 b 凭空创造了一个新名字，即 C1。现在，由于我们的假设中没有关于独立常量 adam 和 eve 的信息，模型生成器认为没有任何理由将它们当做表示不同的实体，于是它们都得到映射到 a。此外，我们并没有指定 man 和 woman 表示不相交的集合，因此，模型生成器让它们相互重叠。这个演示非常明显的隐含了我们用来解释我们的情境的知识，而模型生成器对此一无所知。因此，让我们添加一个新的假设，使 man 和 woman 不相交。模型生成器仍然产生一个反例模型，但这次更符合我们直觉的有关情况：

```
>>> a7 = lp.parse('all x. (man(x) -> -woman(x))')
>>> g = lp.parse('love(adam,eve)')
>>> mc = nltk.MaceCommand(g, assumptions=[a4, a5, a6, a7])
>>> mc.build_model()
True
>>> print mc.valuation
{'C1': 'c',
 'adam': 'a',
 'eve': 'b',
 'love': set([('a', 'c')]),
 'man': set([('a',)]),
 'woman': set([('b',), ('c',)])}
```

经再三考虑，我们可以看到我们的假设中没有说 Eve 是论域中唯一的女性，所以反例模型其实是可以接受的。如果想排除这种可能性，我们将不得不添加进一步的假设，如 exists y. all x. (woman(x) -> (x = y))，以确保模型中只有一个女性。

10.4 英语句子的语义

基于特征的文法中的合成语义学

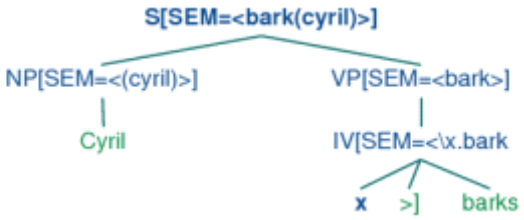
在本章开头，我们简要说明了一种在句法分析的基础上建立语义表示的方法，使用在第9章开发的文法框架。这一次，不是构建一个 SQL 查询，我们将建立一个逻辑形式。我们设计这样的文法的指导思想之一是**组合原则**。（也称为 Frege 原则，具体内容见[Partee, 1995] 。）

组合原则：整体的含义是部分的含义与它们的句法结合　　方式的函数。

我们将假设一个复杂的表达式的语义相关部分由句法分析理论给出。在本章中，我们将认为表达式已经用上下文无关文法分析过。然而，这不是组合原则的内容。

我们现在的目标是以一种可以与分析过程平滑对接的方式整合语义表达的构建。（29）说明了我们想建立的这种分析的第一个近似。

(29)



（29）中，根节点的 SEM 值显示了整个句子的语义表示，而较低节点处的 SEM 值显示句子成分的语义表示。由于 SEM 值必须以特殊的方式来对待，它们被括在尖括号里面用来与其他特征值区别。

到目前为止还好，但我们如何编写能给我们这样的结果的文法规则呢？我们的方法将与本章开始采用的文法 sql0.fcfg 类似，其中我们将为词汇节点指定语义表示，然后组合它们的子节点的每个部分的语义表示。然而，在目前情况下，我们将使用函数应用而不是字符串连接作为组成的模式。为了更具体，假设我们有 SEM 节点带有适当值的 NP 和 VP 成分。那么，一个 S 的 SEM 值由（30）这样的规则处理。（请看在 SEM 的值是一个变量的情况下，我们省略了尖括号）。

(30) S[SEM=<?vp(?np)>] -> NP[SEM=?subj] VP[SEM=?vp]

（30）告诉我们，给定某个 SEM 值?subj 表示主语 NP，某个 SEM 值?vp 表示 VP，父母 S 的 SEM 值通过将?vp 当做?np 的函数表达式来构建。由此，我们可以得出结论?vp 必须表示一个在它的域中有?np 的表示的函数。（30）是一个很好的使用组合原则建立语义的例子。

要完成文法是非常简单的；我们需要的规则全部显示如下：

```
VP[SEM=?v] -> IV[SEM=?v]
NP[SEM=<cyril>] -> 'Cyril'
IV[SEM=<\x.bark(x)>] -> 'barks'
```

VP 规则说的是父母的语义与核心孩子的语义相同。两个词法规则提供非逻辑常数分别作为 Cyril 和 barks 的语义值。barks 入口处有一块额外的符号，我们将简短的解释。

在讲述组合语义规则的细节之前，我们需要为我们的工具箱添加新的工具，称为 λ 演算。这是一个宝贵的工具，用于在我们组装一个英文句子的意思表示时组合一阶逻辑表达式。

λ演算

1.3 节中，我们指出数学集合符号对于制定我们想从文档中选择的词的属性 P 很有用。我们用（31）说明这个，它是“所有 w 的集合，其中 w 是 V（词汇表）的元素且 w 有属性 P”的表示。

(31) {w | w ∈ V & P(w)}

事实证明添加一些能达到同样的效果的东西到一阶逻辑中是非常有用的。我们用**λ运算符**（发音为“lambda”）做这个。（31）的λ对应是（32）。（由于我们不是要在这里讲述理论，我们只将 V 当作一元谓词。）

(32) λw. (V(w) & P(w))

λ表达式最初由 Alonzo Church 设计用来表示可计算函数，并提供数学和逻辑的基础。研究λ表达式的理论被称为λ演算。需要注意的是λ演算不是一阶逻辑的一部分——它们都可以单独使用。

λ是一个约束运算符，就像一阶逻辑量词。如果我们有一个开放公式，如（33a），那么我们可以将变量 x 与λ运算符绑定，如（33b）所示。（33c）给出相应的 NLTK 中的表示。

(33) a. (walk(x) & chew_gum(x))

b. λx.(walk(x) & chew_gum(x))

c. \x.(walk(x) & chew_gum(x))

请记住\是 Python 中的特殊字符。我们要么使用转义字符（另一个\）要么使用“原始字符串”（3.4 节），如下所示：

```
>>> lp = nltk.LogicParser()
>>> 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))
```

我们对绑定表达式中的变量的结果有一个特殊的名字：**λ-抽象**。当你第一次遇到λ-抽象时，很难对它们的意思得到一个直观的感觉。（33b）的一对英语表示是“是一个 x，其中 x 步行且 x 嚼口香糖”或“具有步行和嚼口香糖的属性。”通常认为λ-抽象可以很好的表示动词短语（或无主语从句），尤其是当它作为参数出现在它自己的右侧时。如（34a）和它的翻译（34b）中的演示。

(34) a. To walk and chew gum is hard

b. hard(\x.(walk(x) & chew_gum(x))

所以一般的描绘是这样的：一个开放公式φ有自由变量 x，x 抽象为一个属性表达式λx.φ——满足φ的 x 的属性。这里有一个如何建立抽象的官方版本：

(35)如果α是τ类型，x 是 e 类型的变量，那么\x.α是<e, τ>类型。

（34b）说明了一个情况：我们说某物的属性，即它是很难的。但我们通常所说的属性是为它们指定个体。而事实上，如果φ是一个开放公式，那么抽象λx.φ可以被用来作为一元谓词。在（36）中，术语 gerald 满足（33b）。

(36) \x.(walk(x) & chew_gum(x)) (gerald)

（36）说 Gerald 具有步行和嚼口香糖的属性，与（37）的意思相同。

(37) (walk(gerald) & chew_gum(gerald))

与（37）表示的相同。从（36）到（37）的约简是简化语义表示的非常有用的操作，我们将在本章的其余部分大量用到它。该操作通常被称为**β-约简**。为了使它在语义上合理，我们希望λx.α(β)能保持与α[β/x]有相同的语义值。这的确是真实的，稍微有些复杂，我们马上会遇到。为了在 NLTK 中实施表达式的β-约简，我们可以调用 simplify()方法①。

```
>>> e = lp.parse(r'\x.(walk(x) & chew_gum(x))(gerald)')
>>> print e
\x.(walk(x) & chew_gum(x))(gerald)
>>> print e.simplify()  ①
(walk(gerald) & chew_gum(gerald))
```

虽然我们迄今只考虑了λ-抽象的主体是一个某种类型 t 的开放公式，这不是必要的限制；主体可以是任何符合文法的表达式。下面是一个有两个λ的例子：

(38) \x.\y.(dog(x) & own(y, x))

正如（33b）中起到一元谓词的作用，（38）就像一个二元谓词：它可以直接应用到两个参数①。LogicParser 允许嵌套λ，如\x.\y.，写成缩写形式\x y. ①。

```
>>> print lp.parse(r'\x.\y.(dog(x) & own(y, x))(cyril)').simplify()
\y.(dog(cyril) & own(y,cyril))
>>> print lp.parse(r'\x y.(dog(x) & own(y, x))(cyril, angus)').simplify()  ①
(dog(cyril) & own(angus,cyril))
```

我们所有的λ-抽象到目前为止只涉及熟悉的一阶变量：x、y 等——类型 e 的变量。但假设我们要处理一个抽象，例如：\x.walk(x)，作为另一个λ-抽象的参数？我们不妨试试这个：

\y.y(angus)(\x.walk(x))

但由于自由变量 y 规定是 e 类型，\y.y(angus)只适用于 e 类型的参数，而\x.walk(x)是<e, t>类型的！相反，我们需要允许在更高级的类型的变量上抽象。让我们用 P 和 Q 作为<e, t>类型的变量，那么我们可以有一个抽象，如：\P.P(angus)。由于 P 是<e, t>类型，整个抽象是<<e, t>, t>类型。那么\P.P(angus)(\x.walk(x))是合法的，可以通过β-约简化简为\x.walk(x)(angus)，然后再次化简为 walk(angus)。

在进行β-约简时，对变量有些注意事项。例如，思考（39a）和（39b）的λ-术语，它只是一个自由变量的不同身份。

(39) a. \y.see(y, x)
     b. \y.see(y, z)

现在假设我们应用λ-术语\P.exists x.P(x)到这些术语中的每一个：

(40) a. \P.exists x.P(x)(\y.see(y, x))
     b. \P.exists x.P(x)(\y.see(y, z))

我们较早前指出过应用程序的结果应该是语义等价的。但是，如果我们让（39a）中的自由变量 x 落入（40a）中存在量词的范围内，那么约简之后的结果会不同：

(41) a. exists x.see(x, x)
     b. exists x.see(x, z)

（41a）是指有某个 x 能看到他/她自己，而（41b）的意思是有某个 x 能看到一个未指定的个体 z。出了什么错？显然，我们要禁止（41a）所示的那种变量“捕获”。

为了处理这个问题，让我们退后一会儿。我们使用的变量的特别的名字是否受到在（40a）的函数表达式的存在量词的约束呢？答案是否定的。事实上，给定任何绑定变量的表达

式（包含∀、∃或λ），为绑定的变量选择名字完全是任意的。例如：exists x.P(x)和exists y.P(y)是等价的；它们被称为**α-等价**，或**字母变体**。重新标记绑定的变量的过程被称为**α-转换**。当我们在logic模块中测试VariableBinderExpressions是否相等（即使用==）时，我们其实是测试α-等价：

```
>>> e1 = lp.parse('exists x.P(x)')
>>> print e1
exists x.P(x)
>>> e2 = e1.alpha_convert(nltk.Variable('z'))
>>> print e2
exists z.P(z)
>>> e1 == e2
True
```

当β-约减在一个应用f(a)中实施时，我们检查是否有自由变量在a中同时也作为f的子术语中绑定的变量出现。假设在上面讨论的例子中，x是a中的自由变量，f包括子术语exists x.P(x)。在这种情况下，我们产生一个exists x.P(x)的字母变体，也就是说，exists z1.P(z1)，然后再进行约减。这种重新标记由logic中的β-约减代码自动进行，可以在下面的例子中看到的结果:

```
>>> 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)
```

当你在下面的章节运行这些例子时，你可能会发现返回的逻辑表达式的变量名不同；例如：你可能在前面的公式的z1的位置看到z14。这种标签的变化是无害的——事实上，它仅仅是一个字母变体的例子。

在此附注之后，让我们回到英语句子的逻辑形式建立的任务。

量化的NP

在本节开始，我们简要介绍了如何为Cyril barks构建语义表示。你会以为这太容易了——肯定还有更多关于建立组合语义的。例如：量词？没错，这是一个至关重要的问题。例如，我们要给出（42a）的逻辑形式（42b）。如何才能实现呢？

(42) a. A dog barks.

b. exists x.(dog(x) & bark(x))

让我们作一个假设，我们建立复杂的语义表示的唯一操作是函数应用。那么我们的问题是这样的：我们如何给量化的NP：a dog一个语义表示，使它可以在（42b）中的结果中与bark结合？作为第一步，让我们将主语的SEM值作为函数表达式，而不是参数（这有时被称为类型提升。）。现在，我们寻找实例化?np的方式，使[SEM=<?np(\x.bark(x))>]等价于[SEM=<exists x.(dog(x) & bark(x))>]。这是否看上去有点让人联想到λ-演算中的β-约减？换句话说，我们想要一个λ-术语M来取代?np，从而应用M到\x.bark(x)产生（42b）。要做到这一点，我们替代（42b）中出现的\x.bark(x)为一个谓词变量P，并用λ绑定变量，如（43）中所示。

(43) \P.exists x.(dog(x) & P(x))

我们已经在（43）中使用了不同风格的变量——也就是，'P'而不是'x'或'y'——表明我们在不同类型的对象上抽象——不是单个个体，而是类型为<e,t>的函数表达式。因此，作为一个整体（43）的类型是<<e,t>,t>。我们将普遍把这个作为 NP 的类型。为了进一步说明，一个普遍的量化的 NP 看起来像（44）。

(44) \P.all x.(dog(x) -> P(x))

现在我们几乎完成了，除了我们还需要进行进一步的抽象，加上将限定词 a，即（45），和 dog 的语义组合的过程的应用。

(45) \Q P.exists x.(Q(x) & P(x))

将（46）作为一个函数表达式应用到\x.dog(x)产生(43)，应用到\x.bark(x)给我们\P.exists x.(dog(x) & P(x))(\x.bark(x))。最后，进行β-约简产生我们正想要的，即（42b）。

及物动词

我们的下一个挑战是处理含及物动词的句子，如（46）。

(46) Angus chases a dog.

我们所要建设的输出语义是 exists x.(dog(x) & chase(angus, x))。让我们看看我们如何能够利用λ-抽象得到这样的结果。可能的解决方案中一个重要制约因素是需要 *a dog* 的语义表示与 NP 是否充当句子的主语或宾语独立。换句话说，我们希望得到上述公式作为输出，同时坚持（43）作为 NP 的语义。第二个制约因素是：VP 应该有一个统一的解释的类型，不管它们是否只是一个不及物动词或及物动词加对象组成。更具体地说，我们规定 VP 的类型一直是<e, t>。鉴于这些制约因素，下面是 chases a dog 的成功的语义表示。

(47) \y.exists x.(dog(x) & chase(y, x))

将（47）作为一个 y 的属性，使得对于某只狗 x 有 y 追逐 x；或者更通俗的，作为一个追逐一只狗的 y。我们现在的任务是为 chases 设计语义表示，它可以与（43）结合从而派生出（47）。

让我们在（47）上进行β-约简的逆操作，得到（48）。

(48) \P.exists x.(dog(x) & P(x))(\z.chase(y, z))

（48）在一开始可能会稍微难读；你需要看到，它涉及从（43）到\z.chase(y,z)应用量化的 NP 表示。（48）通过β-约简与 exists x.(dog(x) & chase(y, x))等效。

现在，让我们用与 NP 的类型相同的变量 X，也就是<<e,t>,t>类型，替换（48）中的函数表达式。

(49) X(\z.chase(y, z))

及物动词的表示将必须使用 X 类型的参数来产生 VP 类型，也就是<e,t>类型，的函数表达式。我们可以确保这个，通过在（49）的 X 变量和主语变量 y 上的抽象。因此，完整的解决方案是给出（50）中所示的 chases 的语义表示。

(50) \X y.X(\x.chase(y, x))

如果（50）应用到（43），β-约简后的结果与（47）等效，这是我们一直都想要的东西：

```
>>> lp = nltk.LogicParser()
>>> tvp = lp.parse(r'\X x.X(\y.chase(x,y))')
>>> np = lp.parse(r'(\P.exists x.(dog(x) & P(x)))')
>>> vp = nltk.ApplicationExpression(tvp, np)
>>> print vp
```

```
(\X x.X(\y.chase(x,y)))(\P.exists x.(dog(x) & P(x)))
>>> print vp.simplify()
\x.exists z2.(dog(z2) & chase(x,z2))
```

为了建立一个句子的语义表示，我们也需要组合主语 NP 的语义。如果后者是一个量化的表达式，例如 every girl，一切都与我们前面讲过的 a dog barks 一样的处理方式；主语转换为函数表达式，这被用于 VP 的语义表示。然而，我们现在似乎已经用适当的名称为自己创造了另一个问题。到目前为止，这些已经作为单独的常量进行了语义的处理，这些不能作为像（47）那样的表达式的函数应用。因此，我们需要为它们提出不同的语义表示。我们在这种情况下所做的是重新解释适当的名称，使它们也成为如量化的 NP 那样的函数表达式。这里是 Angus 的 λ 表达式：

(51) \P.P(angus)

（51）表示相应与 Angus 为真的所有属性的集合的特征函数。从独立常量 angus 转换为\P.P(angus)是类型提升的另一个例子，前面简单的提到过，允许我们用等效的函数应用\P.P(angus)(\x.walk(x))替换一个布尔值的应用，如：\x.walk(x)(angus)。通过 β-约简，两个表达式都约简为 walk(angus)。

文法 sem.fcfg 包含一个用于分析和翻译简单的我们刚才一直找寻的这类例子的一个小的规则集合。下面是一个稍微复杂的例子。

```
>>> from nltk import load_parser
>>> parser = load_parser('grammars/book_grammars/simple-sem.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 提供了一些实用工具使获得和检查的语义解释更容易。函数 batch_interpret()用于批量解释输入句子的链表。它建立一个字典 d，其中对每个输入的句子 sent，d[sent]是包含 sent 的分析树和语义表示的(synrep, semrep)对的链表。该值是一个链表，因为 sent 可能有句法歧义；在下面的例子中，链表中的每个句子只有一个分析树。

```
(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 z1.(ankle(z1) & bite(cyril,z1))>]
  (NP[-LOC, NUM='sg', SEM=<\P.P(cyril)>]
    (PropN[-LOC, NUM='sg', SEM=<\P.P(cyril)>] Cyril))
  (VP[NUM='sg', SEM=<\x.exists z1.(ankle(z1) & bite(x,z1))>]
    (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=<\x.ankle(x)>]
        (N[NUM='sg', SEM=<\x.ankle(x)>] ankle)))))
```

现在我们已经看到了英文句子如何转换成逻辑形式，前面我们看到了在模型中如何检查

逻辑形式的真假。把这两个映射放在一起，我们可以检查一个给定的模型中的英语句子的真值。让我们看看前面定义的模型 m。工具 batch_evaluate()类似于 batch_interpret()，除了我们需要传递一个模型和一个变量赋值作为参数。输出是三元组(synrep, semrep, value)，其中 synrep, semrep 和以前一样，value 是真值。为简单起见，下面的例子只处理一个简单的句子。

```
>>> v = """
... bertie => b
... olive => o
... cyril => c
... boy => {b}
... girl => {o}
... dog => {c}
... walk => {o, c}
... see => {(b, o), (c, b), (o, c)}
... """
>>> val = nltk.parse_valuation(v)
>>> g = nltk.Assignment(val.domain)
>>> m = nltk.Model(val.domain, val)
>>> sent = 'Cyril sees every boy'
>>> grammar_file = 'grammars/book_grammars/simple-sem.fcfg'
>>> results = nltk.batch_evaluate([sent], grammar_file, m, g)[0]
>>> for (syntree, semrel, value) in results:
...     print semrep
...     print value
exists z3.(ankle(z3) & bite(cyril,z3))
True
```

再述量词歧义

上述方法的一个重要的限制是它们没有处理范围歧义。我们的翻译方法是句法驱动的，认为语义表示与句法分析紧密耦合，语义中量词的范围也因此反映句法分析树中相应的 NP 的相对范围。因此，像（26）这样的句子，在这里重复，总是会被翻译为（53a）而不是（53b）。

(52) Every girl chases a dog.

(53) a. all x.(girl(x) -> exists y.(dog(y) & chase(x,y)))

b. exists y.(dog(y) & all x.(girl(x) -> chase(x,y)))

有许多方法来处理范围歧义，我们将简要的看看最简单的一个。首先，让我们简要地考虑具有范围的公式的结构。图 10-3 描绘了（52）的这两种不同的读法。

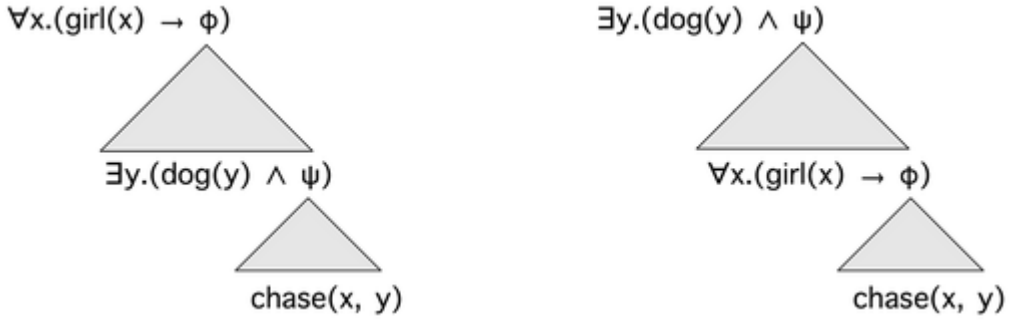


图 10-3. 量词范围

让我们先考虑左侧的结构。在上面，有对应与 every girl 的量词。ϕ可以被看作是量词范围内的所有东西的一个占位符。向下移动，我们看到可以插入相应与 a dog 的量词作为ϕ的实例。这提供了一种新的占位符ψ表示 a dog 的范围，这一点，我们可以堵塞语义的“核心”，即对应于 x chases y 的开放的句子。右侧的结构是相同的，除了两个量词的顺序交换了。

在被称为 **Cooper 存储**的方法中，语义表示不再是一阶逻辑表达式，而是一个由一个“核心”语义表示加一个绑定操作符链表组成的配对。就目前而言，可以认为一个**绑定操作符**是如（44）或（45）那样的量化 NP 的语义表示。沿图 10-3 所示的线向下，我们假设我们已经构建了一个 Cooper 存储风格的句子（52）的语义表示，让我们将开放公式 chase(x,y) 作为核心。给定有关（52）中两个 NP 的绑定操作符的链表，我们将一个绑定操作符从链表挑出来，与核心结合。

\P.exists y.(dog(y) & P(y))(\z2.chase(z1,z2))

然后，我们将链表中的另一个绑定操作符应用到结果中。

\P.all x.(girl(x) -> P(x))(\z1.exists x.(dog(x) & chase(z1,x)))

当链表为空时，我们就有了句子的传统的逻辑形式。将绑定操作符与核心以这种方式组合被称为 **S-检索**。如果我们仔细的允许绑定操作符的每个可能的顺序（例如：通过将链表进行全排列；见 4.5 节），那么我们将能够产生量词的每一个可能的范围排序。

接下来要解决的问题是我们如何建立一个核心+存储表示的组合。如前所述，文法中每个短语和词法规则将有一个 SEM 特征，但现在将有嵌入特征 CORE 和 STORE。要说明这里的机制，让我们考虑一个简单的例子：Cyril smiles。下面是动词 smiles 的词法规则（取自文法 storage.fcfg），看起来还可以：

```
IV[SEM=[CORE=<\x.smile(x)>, STORE=(/)]] -> 'smiles'
```

适当的名称 Cyril 的规则更为复杂。

```
NP[SEM=[CORE=<@x>, STORE=(<bo(\P.P(cyril),@x)>)]] -> 'Cyril'
```

谓词 bo 有两个子部分：一个适当的名称的标准（类型提升）表示，以及表达式@x，被称为绑定操作符的**地址**。（我们将简要解释为什么需要地址变量。）@x 是原变量，也就是，范围在逻辑的独立变量之上的变量，你会看到，它也提供了核心的值。VP 的规则只是向上渗透 IV 的语义，有趣的工作由 S 规则来做。

```
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]]
```

S 节点的核心值是应用 VP 的核心值，即\x.smile(x)，到主语 NP 的值的结果。后者不会是@x，而是一个@x 的实例，也就是 z3。β-约简后，<?vp(?subj)>将与<smile(z3)>统一。现在，当@x 实例化为分析过程的一部分时，它将会同样的被实例化。特别是主

语 NP 的 STORE 中出现的@x 也将被映射到 z3，产生元素 bo(\P.P(cyril),z3)。这些步骤可以在下面的分析树中看到。

```
(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)))
```

让我们回到更复杂的例子，（52），看看在用文法 storage.fcfg 分析后，存储风格的 SEM 值是什么。

```
CORE  = <chase(z1,z2)>
STORE = (bo(\P.all x.(girl(x) -> P(x)),z1), bo(\P.exists x.(dog(x) & P(x)),z2))
```

现在，应该很清楚为什么地址变量是绑定操作符的一个重要组成部分。记得在 S-检索过程中，我们将绑定操作符从 STORE 链表移出，先后将它们应用到 CORE。假设我们从我们想要与 chase(z1,z2)结合的 bo(\P.all x.(girl(x) -> P(x)),z1)开始。绑定操作符的量词部分是\P.all x.(girl(x) -> P(x))，将这与 chase(z1,z2)结合，后者需要先被转换成 λ-抽象。我们怎么知道在哪些变量上进行抽象？这是 z1 的地址告诉我们的，即 every girl 都有追求者而不是主动追求别人的人。

模块 nltk.sem.cooper_storage 处理将存储形式的语义表示转换成标准逻辑形式的任务。首先，我们构造一个 CooperStore 实例，并检查它的 STORE 和 CORE。

```
>>> from nltk.sem import cooper_storage as cs
>>> sentence = 'every girl chases a dog'
>>> trees = cs.parse_with_bindops(sentence, grammar='grammars/book_grammars/storage.fcfg')
>>> semrep = trees[0].node['sem']
>>> cs_semrep = cs.CooperStore(semrep)
>>> print cs_semrep.core
chase(z1,z2)
>>> for bo in cs_semrep.store:
...     print bo
bo(\P.all x.(girl(x) -> P(x)),z1)
bo(\P.exists x.(dog(x) & P(x)),z2)
```

最后，我们调用 s_retrieve()检查读法。

```
>>> cs_semrep.s_retrieve(trace=True)
Permutation 1
   (\P.all x.(girl(x) -> P(x)))(\z1.chase(z1,z2))
   (\P.exists x.(dog(x) & P(x)))(\z2.all x.(girl(x) -> chase(x,z2)))
Permutation 2
   (\P.exists x.(dog(x) & P(x)))(\z2.chase(z1,z2))
   (\P.all x.(girl(x) -> P(x)))(\z1.exists x.(dog(x) & chase(z1,x)))
>>> for reading in cs_semrep.readings:
...     print reading
exists x.(dog(x) & all z3.(girl(z3) -> chase(z3,x)))
all x.(girl(x) -> exists z4.(dog(z4) & chase(x,z4)))
```

10.5 段落语义层

段落是句子的序列。很多时候，段落中的一个句子的解释依赖它前面的句子。一个明显的例子来自照应代词，如 he、she 和 it。给定一个段落如：Angus used to have a dog.But he recently disappeared.你可能会解释 he 指的是 Angus 的狗。然而，在 Angus used to have a dog.He took him for walks in New Town.中，你更可能解释 he 指的是 Angus 自己。

(54) a. Angus owns a dog. It bit Irene.

b. ∃x.(dog(x) & own(Angus, x) & bite(x, Irene))

段落表示理论

一阶逻辑中的量化的标准方法仅限于单个句子。然而，似乎是有量词的范围可以扩大到两个或两个以上的句子的例子。我们之前看到过一个，下面是第二个例子，与它的翻译一起。

(54) a. Angus owns a dog. It bit Irene.

b. ∃x.(dog(x) & own(Angus, x) & bite(x, Irene))

也就是说，NP：a dog 的作用像一个绑定第二句话中的 it 的量词。段落表示理论（Discourse Representation Theory，DRT）的目标是提供一种方法处理这个和看上去是段落的特征的其它语义现象。一个**段落表示结构**（discourse representation structure，DRS）根据一个**段落指称**的列表和一个条件列表表示段落的意思。段落指称是段落中正在讨论的事情，它对应一阶逻辑的单个变量。**DRS 条件**应用于那些段落指称，对应于一阶逻辑的原子开放公式。图 10-4 演示了（54a）中第一句话的 DRS 如何增强为两个句子的 DRS。

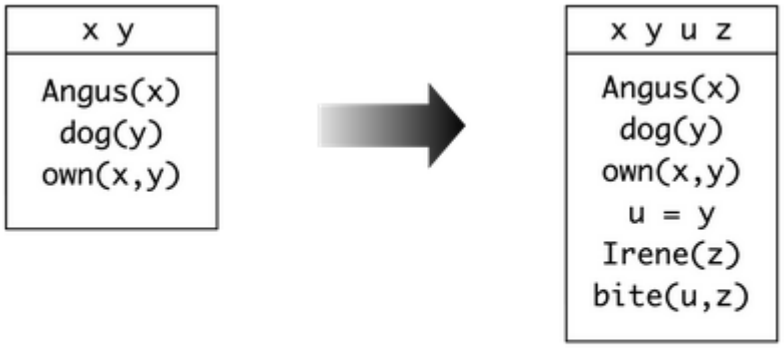


图 10-4. 建立一个 DRS：左侧的 DRS 表示段落中第一句话的处理结果，而右侧的 DRS 显示处理并整合第二句上下文的效果。

在处理（54a）的第二句时，以图 10-4 左侧的已经呈现的上下文背景进行解释。代词 it 触发另外一个新的段落指称，也就是 u，我们需要为它找一个**先行词**——也就是，我们想算出 it 指的是什么。在 DRT 中，为一个参照代词寻找先行词的任务包括将它连接到已经在当前 DRS 中的话题指称，y 是显而易见的选择。（我们会在不久讲述更多关于指代消解的内容。）处理步骤产生一个新的条件 u=y。第二句贡献的其余的内容也与第一个的内容合并，如图 10-4 右侧所示。

图 10-4 说明 DRS 如何表示多个句子。这种情况是一个两句话的段落，但原则上一个 DRS 可以对应整个文本的解释。我们可以查询图 10-4 中 DRS 右侧的真值。我们可以通俗地

说，它为真，如果在情况 s 中有实体 a、c 和 i，对应 DRS 中的段落指称使 s 中所有条件都为真；也就是说，a 是 Angus，c 是 a dog，a 拥有 c，i 是 Irene，c 咬了 i。

为了处理 DRS 计算，我们需要将其转换成线性格式。下面是一个例子，其中 DRS 是由一个段落指称链表和一个 DRS 条件链表组成的配对：

```
([x, y], [angus(x), dog(y), own(x,y)])
```

在 NLTK 建立 DRS 对象最简单的方法是通过解析一个字符串表示①。

```
>>> dp = nltk.DrtParser()
>>> drs1 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)])') ①
>>> print drs1
([x,y],[angus(x), dog(y), own(x,y)])
```

我们可以使用 draw()方法①可视化结果，如图 10-5 所示。

```
>>> drs1.draw() ①
```

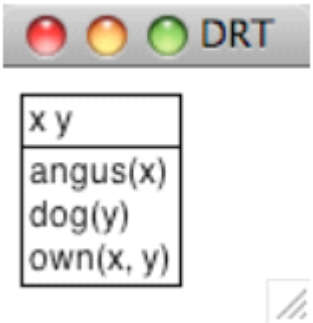


图 10-5. DRS 截图

我们讨论图 10-4 中 DRS 的真值条件时，假设最上面的段落指称被解释为存在量词，而条件也进行了解释，虽然它们是联合的。事实上，每一个 DRS 都可以转化为一阶逻辑公式，fol()方法实现这种转换。

```
>>> print drs1.fol()
exists x y.((angus(x) & dog(y)) & own(x,y))
```

作为一阶逻辑表达式功能补充，DRT 表达式有 DRS-连接运算符，用“+”符号表示。两个 DRS 的连接是一个单独的 DRS 包含合并的段落指称和来自两个论证的条件。DRS-连接自动进行 α -转换绑定变量避免名称冲突。

```
>>> 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)])
```

虽然迄今为止见到的所有条件都是原子的，一个 DRS 可以内嵌入另一个 DRS，这是全称量词被处理的方式。在 drs3 中，没有顶层的段落指称，唯一的条件是由两个子 DRS 组成，通过蕴含连接。再次，我们可以使用 fol()来获得真值条件的句柄。

```
>>> drs3 = dp.parse('([], [(([x], [dog(x)]) -> ([y],[ankle(y), bite(x, y)]))])')
>>> print drs3.fol()
all x.(dog(x) -> exists y.(ankle(y) & bite(x,y)))
```

我们较早前指出 DRT 旨在通过链接照应代词和现有的段落指称来解释照应代词。DRT 设置约束条件使段落指称可以像先行词那样“可访问”，但并不打算解释一个特殊的先行词如何被从候选集合中选出的。模块 nltk.sem.drt_resolve_anaphora 采用了类此的保守策略：如果 DRS 包含 PRO(x)形式的条件，方法 resolve_anaphora()将其替换为 x = [...]

形式的条件，其中[...]是一个可能的先行词链表。

```
>>> drs4 = dp.parse('([x, y], [angus(x), dog(y), own(x, y)])')
>>> drs5 = dp.parse('([u, z], [PRO(u), irene(z), bite(u, z)])')
>>> drs6 = drs4 + drs5
>>> print drs6.simplify()
([x,y,u,z],[angus(x), dog(y), own(x,y), PRO(u), irene(z), bite(u,z)])
>>> print drs6.simplify().resolve_anaphora()
([x,y,u,z],[angus(x), dog(y), own(x,y), (u = [x,y,z]), irene(z), bite(u,z)])
```

由于指代消解算法已分离到它自己的模块，这有利于在替代程序中交换，使对正确的先行词的猜测更加智能。

我们对 DRS 的处理与处理λ-抽象的现有机制是完全兼容的，因此可以直接基于 DRT 而不是一阶逻辑建立组合语义表示。这种技术在下面的不确定性规则（是文法 drt.fcfg 的一部分）中说明。为便于比较，我们已经从 simplesem.fcfg 增加了不确定性的平行规则。

```
Det[NUM=sg,SEM=<\P Q.([x],[]) + P(x) + Q(x)>] -> 'a'
Det[NUM=sg,SEM=<\P Q. exists x.(P(x) & Q(x))>] -> 'a'
```

为了对 DRT 规则如何运作有更好的了解，请看下面的 NP：a dog 的子树：

```
(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.([],[dog(x)])>]
    (N[NUM='sg', SEM=<\x.([],[dog(x)])>] dog)))))
```

作为一个函数表达式不确定性的λ-抽象被应用到\x.([],[dog(x)])，得到\Q.(([x],[]) + ([],[dog(x)]) + Q(x))；简化后，我们得到\Q.(([x],[dog(x)]) + Q(x))将 NP 作为一个整体表示。

为了解析文法 drt.fcfg，我们在 load_parser()调用中指定特征结构中的 SEM 值用 DrtParser 解析替代默认的 LogicParser。

```
>>> from nltk import load_parser
>>> parser = load_parser('grammars/book_grammars/drt.fcfg', logic_parser=nltk.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)])
```

段落处理

我们解释一句话时会使用丰富的上下文知识，一部分取决于前面的内容，一部分取决于我们的背景假设。DRT 提供了一个句子的含义如何集成到前面段落表示中的理论，但是在前面的讨论中明显缺少这两个部分。首先，一直没有尝试纳入任何一种推理；第二，我们只处理了个别句子。这些遗漏由模块 nltk.inference.discourse 纠正。

段落是一个句子的序列 s1, ... sn，段落线是读法的序列 s1-ri, ... sn-rj，每个序列对应段落中的一个句子。该模块按增量处理句子，当有歧义时保持追踪所有可能的线。为简单起见，下面的例子中忽略了范围歧义。

```
>>> dt = nltk.DiscourseTester(['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))
```

一个新句子添加到当前的段落时，设置参数 consistchk=True 会通过每条线，即每个可接受的读法的序列，的检查模块来检查一致性。在这种情况下，用户可以选择收回有问题的句子。

```
>>> 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', verbose=True)
Current sentences are
s0: A student dances
s1: Every student is a person
```

以类似的方式，我们使用 informchk= True 检查新的句子 φ 是否对当前的段落有信息量。定理证明器将段落线中现有的句子当做假设，尝试证明 φ；如果没有发现这样的证据，那么它是有信息量的。

```
>>> 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'
```

也可以传递另一套假设作为背景知识，并使用这些筛选出不一致的读法；详情请参阅 http://www.nltk.org/howto 上的段落 HOWTO。

discourse 模块可适应语义歧义，筛选出不可接受的读法。下面的例子调用 Glue 语义和 DRT。由于 Glue 语义模块被配置为使用的覆盖面广的 Malt 依存关系分析器，输入（Every dog chases a boy. He runs.）需要分词和标注。

```
>>> from nltk.tag import RegexpTagger
>>> tagger = RegexpTagger(
...     [('^(chases|runs)$', 'VB'),
...      ('^(a)$', 'ex_quant'),
...      ('^(every)$', 'univ_quant'),
...      ('^(dog|boy)$', 'NN'),
...      ('^(He)$', 'PRP')
... ])
>>> rc = nltk.DrtGlueReadingCommand(depparser=nltk.MaltParser(tagger=tagger))
>>> dt = nltk.DiscourseTester(['Every dog chases a boy', 'He runs'], rc)
>>> dt.readings()
s0 readings:
s0-r0: ([],[(([x],[dog(x)]) -> ([z3],[boy(z3), chases(x,z3)]))])
s0-r1: ([z4],[boy(z4), (([x],[dog(x)]) -> ([],[chases(x,z4)]))])
s1 readings:
s1-r0: ([x],[PRO(x), runs(x)])
```

段落的第一句有两种可能的读法，取决于量词的作用域。第二句的唯一的读法通过条件 PRO(x)表示代词 He。现在让我们看看段落线的结果：

```
>>> dt.readings(show_thread_readings=True)
d0: ['s0-r0', 's1-r0'] : INVALID: AnaphoraResolutionException
d1: ['s0-r1', 's1-r0'] : ([z6,z10],[boy(z6), (([x],[dog(x)]) ->
```

([],[chases(x,z6)])), (z10 = z6), runs(z10)])

当我们检查段落线 d0 和 d1 时，我们看到读法 s0-r0，其中 every dog 超出了 a boy 的范围，被认为是不可接受的，因为第二句的代词不能得到解释。相比之下，段落线 d1 中的代词（重写为 z10）通过等式(z10 = z6)绑定。

不可接受的读法可以通过传递参数 filter=True 过滤掉。

```
>>> dt.readings(show_thread_readings=True, filter=True)
d1: ['s0-r1', 's1-r0'] : ([z12,z15],[boy(z12), (([x],[dog(x)]) ->
([],[chases(x,z12)])), (z17 = z15), runs(z15)])
```

虽然这一小段是极其有限的，它应该能让你对于我们在超越单个句子后产生的语义处理问题，以及部署用来解决它们的技术有所了解。

10.6 小结

- 一阶逻辑是一种适合在计算环境中表示自然语言的含义的语言，因为它很灵活，足以表示自然语言含义的很多有用的方面，具有使用一阶逻辑推理的高效的定理证明器。（同样的，自然语言语义中也有各种各样的现象，需要更强大的逻辑机制。）
- 在将自然语言句子翻译成一阶逻辑的同时，我们可以通过检查一阶公式模型表述这些句子的真值条件。
- 为了构建成分组合的意思表示，我们为一阶逻辑补充了 λ-演算。
- λ-演算中的 β-约简在语义上与函数传递参数对应。句法上，它包括将被函数表达式中的 λ 绑定的变量替换为函数应用中表达式提供的参数。
- 构建模型的一个关键部分在于建立估值，为非逻辑常量分配解释。这些被解释为 n 元谓词或独立常量。
- 一个开放表达式是一个包含一个或多个自变量的表达式。开放表达式只在它的自变量被赋值时被解释。
- 量词的解释是对于具有变量 x 的公式 φ[x]，构建个体的集合，赋值 g 分配它们作为 x 的值使 φ[x]为真。然后量词对这个集合加以约束。
- 一个封闭的表达式是一个没有自由变量的表达式。也就是，变量都被绑定。一个封闭的表达式是真是假取决于所有变量赋值。
- 如果两个公式只是由绑定操作符（即 λ 或量词）绑定的变量的标签不同，那么它们是 α-等价。重新标记公式中的绑定变量的结果被称为 α-转换。
- 给定有两个嵌套量词 Q1 和 Q2 的公式，最外层的量词 Q1 有较广的范围（或范围超出 Q2）。英语句子往往由于它们包含的量词的范围而产生歧义。
- 在基于特征的文法中英语句子可以通过将 SEM 作为特征与语义表达关联。一个复杂的表达式的 SEM 值通常包括成分表达式的 SEM 值的函数应用。

10.7 进一步阅读

关于本章的进一步材料以及如何安装 Prover9 定理证明器和 Mace4 模型生成器的内容请查阅 http://www.nltk.org/。这两个推论工具一般信息见(McCune, 2008)。

用 NLTK 进行语义分析的更多例子，请参阅 http://www.nltk.org/howto 上的语义和逻辑 HOWTO。请注意，范围歧义还有其他两种解决方法，即(Blackburn & Bos, 2005)描述的 **Hole 语义**和(Dalrymple et al., 1999)描述的 **Glue 语义**。

自然语言语义中还有很多现象没有在本章中涉及到，主要有：

1. 事件、时态和体
2. 语义角色
3. 广义量词，如 most
4. 内涵结构，例如像 may 和 believe 这样的动词

（1）和（2）可以使用一阶逻辑处理，（3）和（4）需要不同的逻辑。下面的读物中很多都讲述了这些问题。

建立自然语言前端数据库方面的结果和技术的综合概述可以在(Androutsopoulos, Ritchie & Thanisch, 1995)中找到。

任何一本现代逻辑的入门书都将提出命题和一阶逻辑。强烈推荐(Hodges, 1977)，书中有很多有关自然语言的有趣且有洞察力的文字和插图。

要说范围广泛，参阅两卷本的关于逻辑教科书(Gamut, 1991a, 1991b)，也包含了有关自然语言的形式语义的当代材料，例如：Montague 文法和内涵逻辑。(Kamp & Reyle, 1993)提供段落表示理论的权威报告，包括涵盖大量且有趣的自然语言片段，包括时态、体和形态。另一个对许多自然语言结构的语义的全面研究是(Carpenter, 1997)。

有许多作品介绍语言学理论框架内的逻辑语义。(Chierchia & McConnell-Ginet, 1990)与句法相对无关，而(Heim & Kratzer, 1998)和(Larson & Segal, 1995)都更明确的倾向于将语义真值条件整合到乔姆斯基框架中。

(Blackburn & Bos, 2005)是致力于计算语义的第一本教科书，为该领域提供了极好的介绍。它扩展了许多本章涵盖的主题，包括量词范围歧义的未指定、一阶逻辑推理以及段落处理。

要获得更先进的当代语义方法的概述，包括处理时态和广义量词，尝试查阅(Lappin, 1996)或(van Benthem & ter Meulen, 1997)。

10.8 练习

1. ○将下列句子翻译成命题逻辑，并用 LogicParser 验证结果。提供显示你的翻译中命题变量如何对应英语表达的一个要点。
   a. If Angus sings, it is not the case that Bertie sulks.
   b. Cyril runs and barks.
   c. It will snow if it doesn't rain.
   d. It's not the case that Irene will be happy if Olive or Tofu comes.
   e. Pat didn't cough or sneeze.
   f. If you don't come if I call, I won't come if you call.
2. ○翻译下面的句子为一阶逻辑的谓词参数公式。
   a. Angus likes Cyril and Irene hates Cyril.
   b. Tofu is taller than Bertie.
   c. Bruce loves himself and Pat does too.
   d. Cyril saw Bertie, but Angus didn't.
   e. Cyril is a four-legged friend.
   f. Tofu and Olive are near each other.
3. ○翻译下列句子为成一阶逻辑的量化公式。
   a. Angus likes someone and someone likes Julia.
   b. Angus loves a dog who loves him.

c. Nobody smiles at Pat.
d. Somebody coughs and sneezes.
e. Nobody coughed or sneezed.
f. Bruce loves somebody other than Bruce.
g. Nobody other than Matthew loves Pat.
h. Cyril likes everyone except for Irene.
i. Exactly one person is asleep.

4. ○使用λ-抽象和一阶逻辑的量化公式，翻译下列动词短语。
   a. feed Cyril and give a capuccino to Angus
   b. be given ‘War and Peace’ by Pat
   c. be loved by everyone
   d. be loved or detested by everyone
   e. be loved by everyone and detested by no-one
5. ○思考下面的语句：

```
>>> lp = nltk.LogicParser()
>>> e2 = lp.parse('pat')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
exists y.love(pat, y)
```

显然这里缺少了什么东西，即 e1 值的声明。为了 ApplicationExpression(e1, e2) 被β-转换为 exists y.love(pat, y)，e1 必须是一个以 pat 为参数的λ-抽象。你的任务是构建这样的一个抽象，将它绑定到 e1，使上面的语句都是满足（上到字母方差）。此外，提供一个 e3.simplify() 的非正式的英文翻译。

现在根据 e3.simplify() 的进一步情况（如下所示）继续做同样的任务：

```
>>> print e3.simplify()
exists y.(love(pat,y) | love(y,pat))
>>> print e3.simplify()
exists y.(love(pat,y) | love(y,pat))
>>> print e3.simplify()
walk(fido)
```

6. ○如前面的练习中那样，找到一个λ-抽象 e1，产生与下面显示的等效的结果：

```
>>> e2 = lp.parse('chase')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
\x.all y.(dog(y) -> chase(x,pat))
>>> e2 = lp.parse('chase')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
\x.exists y.(dog(y) & chase(pat,x))
>>> e2 = lp.parse('give')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
\x0 x1.exists y.(present(y) & give(x1,y,x0))
```

7. ○如前面的练习中那样，找到一个λ-抽象 e1，产生与下面显示的等效的结果：

```
>>> e2 = lp.parse('bark')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
exists y.(dog(x) & bark(x))
>>> e2 = lp.parse('bark')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
bark(fido)
>>> e2 = lp.parse('\\P. all x. (dog(x) -> P(x))')
>>> e3 = nltk.ApplicationExpression(e1, e2)
>>> print e3.simplify()
all x.(dog(x) -> bark(x))
```

8. ◑开发一种方法，翻译英语句子为带有二元**广义量词**的公式。在此方法中，给定广义量词 Q，量化公式的形式为 Q(A, B)，其中 A 和 B 是<e, t>类型的表达式。那么，例如：all(A,B)为真当且仅当 A 表示的是 B 所表示的一个子集。
9. ◑扩展前面练习中的方法，使量词如 most 和 exactly three 的真值条件可以在模型中计算。
10. ◑修改 sem.evaluate 代码，使它能提供一个有用的错误消息，如果一个表达式不在模型的估值函数的域中。
11. ●从儿童读物中选择三个或四个连续的句子。一个例子是 nltk.corpus.gutenberg: bryantstories.txt，burgess-busterbrown.txt 和 edgeworth-parents.txt 中的故事集。开发一个文法，能将你的句子翻译成一阶逻辑，建立一个模型，使它能检查这些翻译为真或为假。
12. ●实施前面的练习，但使用 DRT 作为意思表示。
13. ●以(Warren & Pereira, 1982)为出发点，开发一种技术，转换一个自然语言查询为一种可以更加有效的在模型中评估的形式。例如：给定一个(P(x) & Q(x))形式的查询，将它转换为(Q(x) & P(x))，如果 Q 的范围比 P 小。

第 11 章 语言数据管理

已标注的语言数据的结构化集合在 NLP 的大部分领域都是至关重要的；但是，我们使用它们仍然面临着许多障碍。本章的目的是要回答下列问题：

1. 我们如何设计一种新的语言资源，并确保它的覆盖面、平衡以及支持广泛用途的文档？
2. 现有数据对某些分析工具格式不兼容，我们如何才能将其转换成合适的格式？
3. 有什么好的方法来记录我们已经创建的资源的存在，让其他人可以很容易地找到它？

一路上，我们将研究当前语料库的设计、创建一个语料库的典型工作流程，及语料库的生命周期。与在其他章节中一样，会有语言数据管理实际实验的很多例子，包括在语言学现场教学课程、实验室的工作和网络爬取中收集的数据。

11.1 语料库结构：一个案例研究

TIMIT 语料库是第一个广泛发布的已标注语音数据库，它有一个特别清晰的组织结构。TIMIT 由一个包括克萨斯仪器公司和麻省理工学院的财团开发，它也由此得名。它被设计用来为声学-语音知识的获取提供数据，并支持自动语音识别系统的开发和评估。

TIMIT 的结构

与布朗语料库显示文章风格和来源的平衡选集一样，TIMIT 包括方言、说话者和材料的平衡选集。对 8 个方言区中的每一种方言，具有一定年龄范围和教育背景的 50 个男性和女性的说话者每人读 10 个精心挑选的句子。设计中有两句话是所有说话者都读的，带来方言的变化：

(1) a. she had your dark suit in greasy wash water all year
    b. don’t ask me to carry an oily rag like that

选择的其余的句子是语音丰富的，包含所有单音（音）和全部的双音（二元音）。此外，设计在以下两方面取得平衡：多个说话者说相同的句子以便说话者之间进行比较，和语料库涵盖较大范围的句子以便获得最大的双音覆盖。每个说话者读的句子中的 5 个也会被 6 个其它说话者读（为了可比性）。每个说话者读的其余 3 个句子是不同的（为了覆盖面）。

NLTK 包括 TIMIT 语料库的一个样本。你可以使用常用的方式，也就是使用 help(nltk.corpus.timit)，访问它的文档。nltk.corpus.timit.fileids()可以看到语料库样本中 160 个录制的话语列表。每个文件名的内部结构如图 11-1 所示。

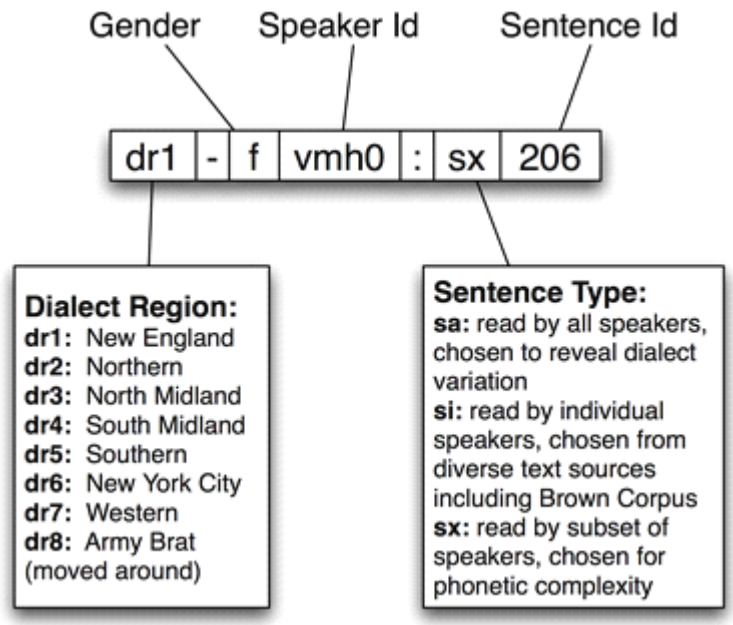


图 11-1. TIMIT 标识符的结构：每个记录使用说话者的方言区、性别、说话者标识符、句子类型、句子标识符组成的一个字符串作为标签。

每个项目都有音标，可以使用 phones()方法访问。我们可以按习惯的方式访问相应的词标识符。两种访问方法都允许一个可选的参数 offset=True，其中包括音频文件的相应跨度的开始和结尾偏移。

```
>>> phonetic = nltk.corpus.timit.phones('dr1-fvmh0/sa1')
>>> 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/sa1')
[('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)]
```

除了这种文本数据，TIMIT 还包括一个词典，提供每一个词的可与一个特定的话语比较的规范的发音：

```
>>> timitdict = nltk.corpus.timit.transcription_dict()
>>> timitdict['greasy'] + timitdict['wash'] + timitdict['water']
['g', 'r', 'iy1', 's', 'iy', 'w', 'ao1', 'sh', 'w', 'ao1', 't', 'axr']
>>> phonetic[17:30]
['g', 'r', 'iy', 's', 'iy', 'w', 'aa', 'sh', 'epi', 'w', 'aa', 'dx', 'ax']
```

这给了我们一点印象：语音处理系统在处理或识别这种特殊的方言（新英格兰）的语音中必须做什么。最后，TIMIT 包括说话人的人口学统计，允许细粒度的研究声音、社会和性别特征。

```
>>> 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')
```

主要设计特点

TIMIT 演示了语料库设计中的几个主要特点。首先，语料库包含语音和字形两个标注层。一般情况下，文字或语音语料库可能在多个不同的语言学层次标注，包括形态、句法和段落层次。此外，即使在给定的层次仍然有不同的标注策略，甚至标注者之间也会有分歧，因此我们要表示多个版本。TIMIT 的第二个特点是：它在多个维度的变化与方言地区和二元音覆盖范围之间取得平衡。人口学统计的加入带来了许多更独立的变量，这可能有助于解释数据中的变化，便于以后出于在建立语料库时没有想到的目的使用语料库，例如社会语言学。第三个特点是：将原始语言学事件作为录音来捕捉和作为标注来捕捉之间有明显的区分。两者一致表示文本语料库正确，原始文本通常有被认为是不可改变的作品的外部来源。那个作品的任何包含人的判断的转换——即使如分词一样简单——也是后来的修订版；因此以尽可能接近原始的形式保留源材料是十分重要的。

TIMIT 的第四个特点是语料库的层次结构。每个句子 4 个文件，500 个说话者每人 10 个句子，有 20,000 个文件。这些被组织成一个树状结构，示意图如图 11-2 所示。在顶层分成训练集和测试集，用于开发和评估统计模型。

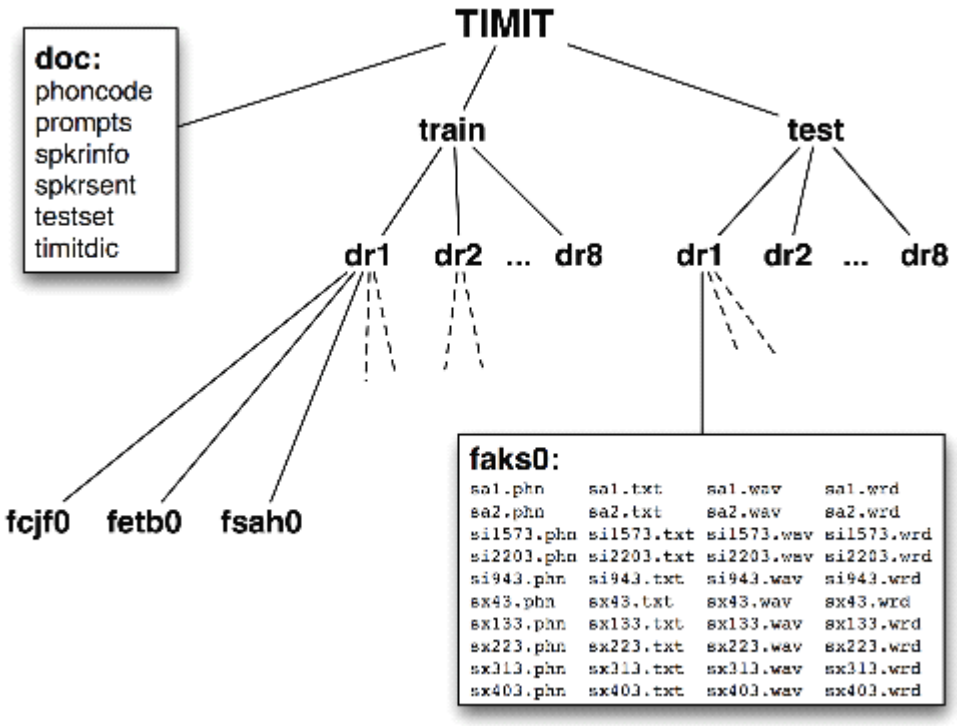


图 11-2. 发布的 TIMIT 语料库的结构；CD-ROM 包含文档、顶层的训练（train）和测试（test）目录；训练和测试目录都有 8 子目录，每个方言区一个；这些目录又包含更多子目录，每个说话者一个；列出的目录是女性说话者 aks0 *的目录的内容，显示 10 个* wav *文件配以一个录音文本文件、一个录音文本词对齐文件和一个音标文件。*

最后，请注意虽然 TIMIT 是语音语料库，它的录音文本和相关数据只是文本，可以使用程序处理了，就像任何其他的文本语料库那样。因此，许多在这本书中所描述的计算方法都适用。此外，注意 TIMIT 语料库包含的所有数据类型分为词汇和文字两个基本类别，我们将在下面讨论。说话者人口学统计数据只不过是词汇数据类型的另一个实例。

当我们考虑到文字和记录结构是计算机科学中关注数据管理的两个子领域首要内容，即

全文检索领域和数据库领域，这最后的观察就不太令人惊讶了。语言数据管理的一个显着特点是往往将这两种数据类型放在一起，可以利用这两个领域的成果和技术。

基本数据类型

不考虑它的复杂性，TIMIT 语料库只包含两种基本数据类型：词典和文本。正如我们在第 2 章中所看到的，大多数词典资源都可以使用记录结构表示，即一个关键字加一个或多个字段，如图 11-3 所示。词典资源可能是一个传统字典或比较词表，如下所示。它也可以是一个短语词典，其中的关键字是一个短语而不是一个词。词典还包括记录结构化的数据，我们可以通过对应主题的非关键字字段来查找条目。我们也可以构造特殊的表格（称为范例）来进行对比和说明系统性的变化，图 11-3 显示了三个动词。TIMIT 的说话者表也是一种词典资源。

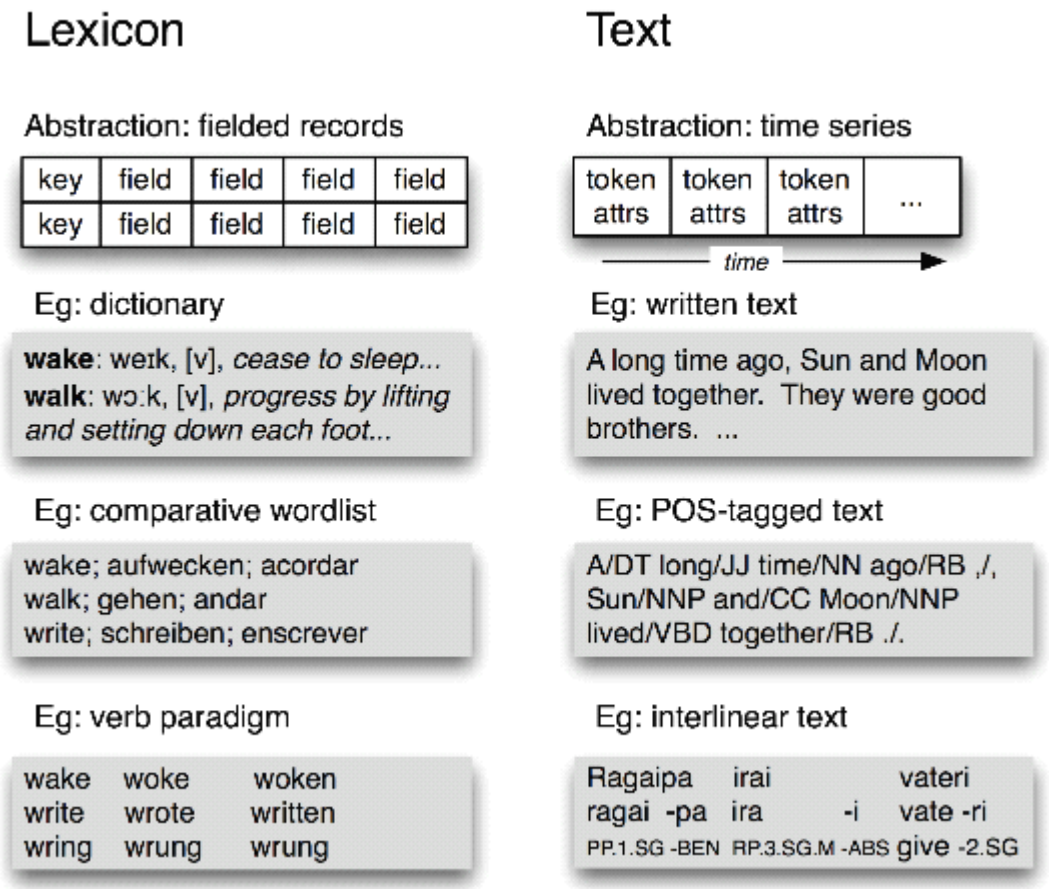


图 11-3. 基本语言数据类型——词汇和文本：它们的多样性中，词汇具有记录结构，而已标注文本具有时间组织。

在最抽象的层面上，文本是一个真实的或虚构的讲话事件的表示，该事件的时间过程也在文本本身存在。一个文本可以是一个小单位，如一个词或句子，也可以是一个完整的叙述或对话。它可能会有标注如词性标记、形态分析、话语结构等。正如我们在 IOB 标注技术（第 7 章）中所看到的，可以使用单个词的标记表示更高层次的成分。因此，图 11-3 所示的文本的抽象就足够了。

不考虑单独的语料库的复杂性和特质，最基本的，它们是带有记录结构化数据的文本集合。语料库的内容往往偏重于这些类型中的一种或多种。例如：布朗语料库包含 500 个文本文件，但我们仍然可以使用表将这些文件与 15 中不同风格关联。在事情的另一面，WordNe

t 包含 117659 个同义词集记录，也包含许多例子句子（小文本）来说明词的用法。TIMIT 处在中间，含有大量的独立的文本和词汇类型的材料。

11.2 语料库生命周期

语料库并不是从天而降的，需要精心的准备和许多人长时期的输入。原始数据需要进行收集、清理、记录并以系统化的结构存储。标注可分为各种层次，一些需要语言的形态或句法的专门知识。要在这个阶段成功取决于建立一个高效的工作流程，包括适当的工具和格式转换器。质量控制程序可以将寻找标注中的不一致落实到位，确保尽最大可能在标注者之间达成一致。由于任务的规模和复杂性，大型语料库可能需要几年的准备，包括几十或上百人多年的努力。在本节中，我们简要地回顾语料库生命周期的各个阶段。

语料库创建的三种方案

语料库的一种类型是设计在创作者的探索过程中逐步展现。这是典型的传统“领域语言学”模式，即来自会话的材料在它被收集的时候就被分析，明天的想法往往基于今天的分析中产生的问题。在随后几年的研究中产生的语料不断被使用，并可能用作不确定的档案资源。计算机化明显有利于这种类型的工作，以广受欢迎的程序 Shoebox 为例，它作为 Toolbox 重新发布，现在已有超过二十年的历史（见 2.4 节）。其他的软件工具，甚至是简单的文字处理器和电子表格，通常也可用于采集数据。在下一节，我们将着眼于如何从这些来源提取数据。

另一种语料库创建方案是典型的实验研究，其中一些精心设计的材料被从一定范围的人类受试者中收集，然后进行分析来评估一个假设或开发一种技术。此类数据库在实验室或公司内被共享和重用已很常见，经常被更广泛的发布。这种类型的语料库是“共同任务”的科研管理方法的基础，这在过去的二十年已成为政府资助的语言技术研究项目。在前面的章节中，我们已经遇到很多这样的语料库；我们将看到如何编写 Python 程序实践这些语料库发布前必要的一些任务。

最后，还有努力为一个特定的语言收集“参考语料”，如美国国家语料库（American National Corpus，ANC）和英国国家语料库（British National Corpus，BNC）。这里的目标已经成为产生各种形式、风格和语言的使用的一个全面的记录。除了规模庞大的挑战，还严重依赖自动标注工具和后期编辑共同修复错误。然而，我们可以编写程序来查找和修复错误，还可以分析语料库是否平衡。

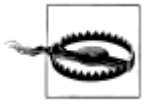
质量控制

自动和手动的数据准备的好的工具是必不可少的。然而，一个高质量的语料库的建立很大程度取决于文档、培训和工作流程等平凡的东西。标注指南确定任务并记录标记约定。它们可能会定期更新以覆盖不同的情况，同时制定实现更一致的标注的新规则。在此过程中标注者需要接受训练，包括指南中没有的情况的解决方法。需要建立工作流程，尽可能与支持软件一起，跟踪哪些文件已被初始化、标注、验证、手动检查等等。可能有多层标注，由不同的专家提供。不确定或不一致的情况可能需要裁决。

大的标注任务需要多个标注者，由此产生一致性的问题。一组标注者如何能一致的处理

呢？我们可以通过将一部分独立的原始材料由两个人分别标注，很容易地测量标注的一致性。这可以揭示指南中或标注任务的不同功能的不足。在对质量要求较高的情况下，整个语料库可以标注两次，由专家裁决不一致的地方。

报告标注者之间对语料库（如通过两次标注 10%的语料库）达成的一致性被认为是最佳实践。这个分数作为一个有用的在此语料库上训练的所有自动化系统的期望性能的上限。



注意！

应谨慎解释标注者之间一致性得分，因为标注任务的难度差异巨大。例如：90％的一致性得分对于词性标注是可怕的得分，但对语义角色标注是可以预期的得分。

Kappa 系数 k 测量两个人判断类别和修正预期的期望一致性的一致性。例如：假设要标注一个项目，四种编码选项可能性相同。这种情况下，两个人随机编码预计有 25％可能达成一致。因此，25％一致性将表示为 k = 0，相应的较好水平的一致性将依比例决定。对于一个 50％的一致性，我们将得到 k = 0.333，因为 50 是从 25 到 100 之间距离的三分之一。还有许多其他一致性测量方法；详情请参阅 help(nltk.metrics.agreement)。

我们还可以测量语言输入的两个独立分割的一致性，例如：分词、句子分割、命名实体识别。在图 11-4 中，我们看到三种可能的由标注者（或程序）产生的项目序列的分割。虽然没有一个完全一致，S1 和 S2 是接近一致的，我们想要一个合适的测量。windowdiff 是评估两个分割一致性的一个简单的算法，通过在数据上移动一个滑动窗口计算近似差错的部分得分。如果我们将标识符预处理成 0 和 1 的序列，当标识符后面跟着边界符号时记录下来，我们就可以用字符串表示分割，应用 windowdiff 打分器。

```
>>> s1 = "00000010000000001000000"
>>> s2 = "00000001000000010000000"
>>> s3 = "00010000000000000001000"
>>> nltk.windowdiff(s1, s1, 3)
0
>>> nltk.windowdiff(s1, s2, 3)
4
>>> nltk.windowdiff(s2, s3, 3)
16
```

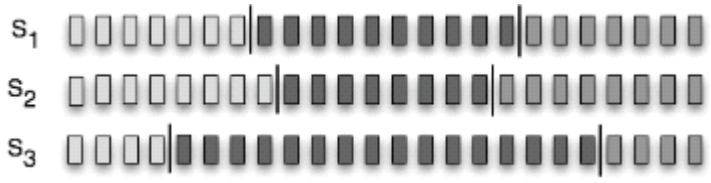


图 11-4. 一个序列的三种分割：小矩形代表字、词、句，总之，任何可能被分为语言单位的序列；S1 和 S2 是接近一致的，两者都与 S3 显著不同。

上面的例子中，窗口大小为 3。windowdiff 计算在一对字符串上滑动这个窗口。在每个位置它计算两个字符串在这个窗口内的边界的总数，然后计算差异，最后累加这些差异。我们可以增加或缩小窗口的大小来控制测量的敏感度。

维护与演变

随着大型语料库的发布，研究人员立足于均衡的从为完全不同的目的而创建的语料库中

派生出的子集进行调查的可能性越来越大。例如：Switchboard 数据库，最初是为识别说话人的研究而收集的，已被用作语音识别、单词发音、口吃、句法、语调和段落结构研究的基础。重用语言语料库的动机包括希望节省时间和精力，希望在别人可以复制的材料上工作，有时希望研究语言行为的更加自然的形式。为这样的研究选择子集的过程本身可视为一个不平凡的贡献。

除了选择语料库的适当的子集，这个新的工作可能包括重新格式化文本文件（如转换为 XML），重命名文件，重新为文本分词，选择数据的一个子集来充实等等。多个研究小组可以独立的做这项工作，如图 11-5 所示。在以后的日子，应该有人想要组合不同的版本的源数据，这项任务可能会非常繁重。

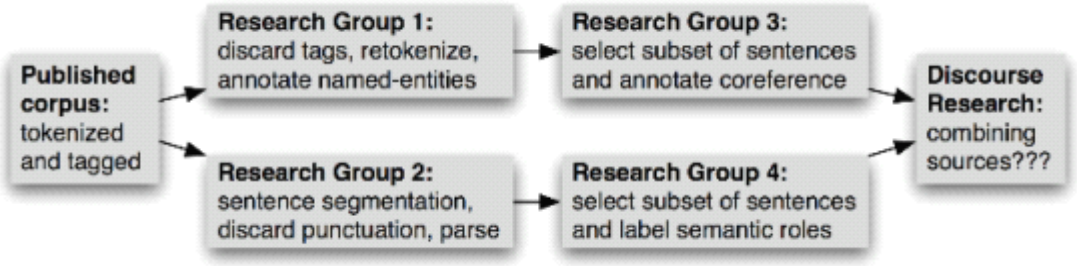
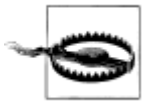


图 11-5. 语料库随着时间的推移而演变：语料库发布后，研究小组将独立的使用它，选择和丰富不同的部分；然后研究努力整合单独的标注，面临校准注释的艰巨的挑战。

由于缺乏有关派生的版本如何创建的，哪个版本才是最新的等记录，使用派生的语料库的任务变得更加困难。这种混乱情况的改进方法是集中维护语料库，专家委员会定期修订和扩充它，考虑第三方的意见，不时发布的新版本。出版字典和国家语料库可能以这种方式集中维护。然而，对于大多数的语料库，这种模式是完全不切实际的。

原始语料库的出版的一个中间过程是要有一个能识别其中任何一部分的规范。每个句子、树、或词条都有一个全局的唯一标识符，每个标识符、节点或字段（分别）都有一个相对偏移。标注，包括分割，可以使用规范的标识符（一个被称为**对峙注释**的方法）引用源材料。这样，新的标注可以与源材料独立分布，同一来源的多个独立标注可以对比和更新而不影响源材料。

如果语料库出版提供了多个版本，版本号或日期可以是识别规范的一部分。整个语料的版本标识符之间的对应表，将使任何对峙的注释更容易被更新。



注意！有时一个更新的语料包含对一直在外部标注的基本材料的修正。标识符可能会被分拆或合并，成分可能已被重新排列。新老标识符之间可能不会一一对应。使对峙标注打破新版本的这些组件比默默允许其标识符指向不正确的位置要好。

11.3 数据采集

从网上获取数据

网络是语言分析的一个丰富的数据源。我们已经讨论了访问单个文件，如 RSS 订阅、搜索引擎的结果（见 3.1 节）的方法。然而，在某些情况下，我们要获得大量的 Web 文本。

最简单的方法是获得出版的网页文本的文集。Web 语料库 ACL 特别兴趣组（The ACL Special Interest Group on Web as Corpus，SIGWAC）在 http://www.sigwac.org.uk/维护一个

资源列表。使用定义好的 Web 语料库的优点是它们有文档、稳定并允许重复性实验。

如果所需的内容在一个特定的网站，有许多实用程序能捕获网站的所有可访问内容，如 GNU Wget（http://www.gnu.org/software/wget/）。为了最大的灵活性和可控制，可以使用网络爬虫如 Heritrix（http://crawler.archive.org/）。爬虫允许细粒度的控制去看哪里，抓取哪个链接以及如何组织结果。例如：如果我们要编译双语文本集合，对应两种语言的文档对，爬虫需要检测站点的结构以提取文件之间的对应关系，它需要按照捕获的对应方式组织下载的页面。写你自己的网页爬虫可能使很有诱惑力的，但也有很多陷阱需要克服，如检测 MIME 类型、转换相对地址为绝对 URL、避免被困在循环链接结构、处理网络延迟、避免使站点超载或被禁止访问该网站等。

从字处理器文件获取数据

文字处理软件通常用来在具有有限的可计算基础设施的项目中手工编制文本和词汇。这些项目往往提供数据录入模板，通过字处理软件并不能保证数据结构正确。例如：每个文本可能需要有一个标题和日期。同样，每个词条可能有一些必须的字段。随着数据规模和复杂性的增长，用于维持其一致性的时间的比重也增大。

我们怎样才能提取这些文件的内容，使我们能够在外部程序中操作？此外，我们如何才能验证这些文件的内容，以帮助作者创造结构良好的数据，在原始的创作过程中最大限度提高数据的质量？

考虑一个字典，其中的每个条目都有一个词性字段，从一个 20 个可能值的集合选取，在发音字段显示，以 11 号黑体字呈现。传统的文字处理器没有能够验证所有的词性字段已正确输入和显示的搜索函数或宏。这个任务需要彻底的手动检查。如果字处理器允许保存文档为一种非专有的格式，如 text、HTML 或 XML，有时我们可以写程序自动做这个检查。

思考下面的一个词条的片段：“sleep [sli:p] v.i. condition of body and mind...”。我们可以在 MSWord 中输入这些词，然后“另存为网页”，然后检查生成的 HTML 文件：

```
<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'>v.i.</span></b>
<span style='mso-spacerun:yes'> </span>
<i>a condition of body and mind ...<o:p></o:p></i>
</p>
```

观察该条目的 HTML 段落表示，使用了<p>元素，词性出现在元素内。下面的程序定义了合法的词性的集合：legal_pos。然后从 dict.htm 文件提取所有 11 号字的内容，并存储在集合 used_pos 中。请看搜索模式包含一个括号括起来的子表达式；只有匹配该子表达式的材料才会被 re.findall 返回。最后，程序用 used_pos 和 legal_pos 的补集构建非法词性的集合：

```
>>> 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']

这个简单的程序只是冰山一角。我们可以开发复杂的工具来检查字处理器文件的一致性，并报告错误，使字典的维护者可以使用原来的文字处理器纠正的原始文件。

只要我们知道数据的正确格式，就可以编写其他程序将数据转换成不同格式。例 11-1 中的程序使用 nltk.clean_html()剥离 HTML 标记，提取词和它们的发音，以“逗号分隔值”（CSV）格式生成输出。

例 11-1. 将 Microsoft Word 创建的 HTML 转换成 CSV

```
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"))
```

注意！

更多 HTML 复杂的处理可以使用 http://www.crummy.com/software/BeautifulSoup/上的 Beautiful Soup 的包。

从电子表格和数据库中获取数据

电子表格通常用于获取词表或范式。例如：一个比较词表可以用电子表格创建，用一排表示每个同源组，每种语言一列（见 nltk.corpus.swadesh 和 www.rosettaproject.org）。大多数电子表格软件可以将数据导出为 CSV 格式。正如我们将在下面看到的，使用 csv 模块 Python 程序可以很容易的访问它们。

有时词典存储在一个完全成熟的关系数据库。经过适当的标准化，这些数据库可以确保数据的有效性。例如：我们可以要求所有词性都来自指定的词汇，通过声明词性字段为枚举类型或用一个外键引用一个单独的词性表。然而，关系模型需要提前定义好的数据（模式）结构，这与高度探索性的构造语言数据的主导方法相违背。被认为是强制性的和独特的字段往往需要是可选的、可重复。只有当数据类型提前全都知道时关系数据库才是适用的。如果不是，或者几乎所有的属性都是可选的或重复的，关系的做法就行不通了。

然而，当我们的目标只是简单的从数据库中提取内容时，完全可以将表格（或 SQL 查询结果）转换成 CSV 格式，并加载到我们的程序中。我们的程序可能会执行不太容易用 SQL 表示的语言学目的的查询，如：select all words that appear in example sentences for which no dictionary entry is provided。对于这个任务，我们需要从记录中提取足够的信息，使它连同词条和例句能被唯一的识别。让我们假设现在这个信息是在一个 CSV 文件 dict.csv 中：

```
"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"
```

现在，我们可以表示此查询，如下所示：

```
>>> import csv
>>> lexicon = csv.reader(open('dict.csv'))
>>> pairs = [(lexeme, defn) for (lexeme, _, _, defn) in lexicon]
>>> lexemes, defns = zip(*pairs)
>>> defn_words = set(w for defn in defns for w in defn.split())
>>> sorted(defn_words.difference(lexemes))
['...', 'a', 'and', 'body', 'by', 'cease', 'condition', 'down', 'each',
'foot', 'lifting', 'mind', 'of', 'progress', 'setting', 'to']
```

然后，这些信息将可以指导正在进行的工作来丰富词汇和更新关系数据库的内容。

转换数据格式

已标注语言数据很少以最方便的格式保存，往往需要进行各种格式转换。字符编码之间的转换已经讨论过（见 3.3 节）。在这里，我们专注于数据结构。

最简单的情况，输入和输出格式是同构的。例如：我们可能要将词汇数据从 Toolbox 格式转换为 XML，可以直接一次一个的转换词条（11.4 节）。数据结构反映在所需的程序的结构中：一个 for 循环，每次循环处理一个词条。

另一种常见的情况，输出是输入的摘要形式，如：一个倒置的文件索引。有必要在内存中建立索引结构（见例 4.8），然后把它以所需的格式写入一个文件。下面的例子构造一个索引，映射字典定义的词汇到相应的每个词条①的语意②，已经对定义文本分词③，并丢弃短词④。一旦该索引建成，我们打开一个文件，然后遍历索引项，以所需的格式输出行⑤。

```
>>> idx = nltk.Index((defn_word, lexeme)    ①
...                  for (lexeme, defn) in pairs    ②
...                  for defn_word in nltk.word_tokenize(defn)    ③
...                  if len(defn_word) > 3)    ④
>>> idx_file = open("dict.idx", "w")
>>> for word in sorted(idx):
...     idx_words = ', '.join(idx[word])
...     idx_line = "%s: %s\n" % (word, idx_words)    ⑤
...     idx_file.write(idx_line)
>>> idx_file.close()
```

由此产生的文件 dict.idx 包含下面的行。（如果有更大的字典，我们希望找到每个索引条目中列出的多个语意）。

```
body: sleep
cease: wake
condition: sleep
down: walk
each: walk
foot: walk
lifting: walk
mind: sleep
```

progress: walk
setting: walk
sleep: wake

在某些情况下，输入和输出数据都包括两个或两个以上的维度。例如：输入可能是一组文件，每个都包含单词频率数据的单独列。所需的输出可能是一个两维表，其中原来的列以行出现。在这种情况下，我们一次填补一列填充内部的数据结构，然后在写数据到输出文件中时一次读取一行。

最棘手的情况，源格式和目标格式覆盖的域略有不同，在它们之间的翻译过程中信息不可避免地会失去。例如：我们可以组合多个 Toolbox 文件创建一个 CSV 文件，其中包含了比较词表，失去了输入文件中除\lx 字段以外所有的。如果后来修改了 CSV 文件，将变化注入原有的 Toolbox 文件将是一个消耗体力的过程。这种“往返”问题的部分解决方法是为每个语言对象关联明确的标识符，使用这些对象的标识符。

决定要包含的标注层

发布的语料库中所包含的信息的丰富性差别很大。语料库最低限度通常会包含至少一个声音或字形符号的序列。事情的另一面，一个语料库可以包含大量的信息，如：句法结构、形态、韵律、每个句子的语义、加上段落关系或对话行为的标注。标注的这些额外的层可能正是有人执行一个特定的数据分析任务所需要的。例如：如果我们可以搜索特定的句法结构，找到一个给定的语言模式就更容易；如果每个词都标注了意义，为语言模式归类就更容易。这里提供一些常用的标注层：

分词：文本的书写形式不能明确地识别它的标识符。分词和规范化的版本作为常规的正式版本的补充可能是一个非常方便的资源。

断句：正如我们在第 3 章中看到的，断句比它看上去的似乎更加困难。因此，一些语料库使用明确的标注来断句。

分段：段和其他结构元素（标题，章节等）可能会明确注明。

词性：文档中的每个单词的词类。

句法结构：一个树状结构，显示一个句子的组成结构。

浅层语义：命名实体和共指标注，语义角色标签。

对话与段落：对话行为标记，修辞结构。

不幸的是，现有的语料库之间在如何表示标注上并没有多少一致性。然而，两个大类的标注表示应加以区别。**内联标注**通过插入带有标注信息的特殊符号或控制序列修改原始文档。例如：为文档标注词性时，字符串“fly”可能被替换为字符串“fly/NN”来表示词 fly 在文中是名词。相比之下，**对峙标注**不修改原始文档，而是创建一个新的文档，通过使用指针引用原始文档来增加标注信息。例如：这个新的文档可能包含字符串“<token id=8 pos='NN'/>”，表示 8 号标识符是一个名词。

标准和工具

一个用途广泛的语料库需要支持广泛的格式。然而，NLP 研究的前沿需要各种新定义的没有得到广泛支持的标注。一般情况下，并没有广泛使用的适当的创作、发布和使用语言数据的工具。大多数项目都必须制定它们自己的一套工具，供内部使用，这对缺乏必要的资源的其他人没有任何帮助。此外，我们还没有一个可以胜任的普遍接受的标准来表示语料库

的结构和内容。没有这样的标准，就不可能有通用的工具——同时，没有可用的工具，适当的标准也不太可能被开发、使用和接受。

针对这种情况的一个反应就是开拓未来开发一种通用的能充分表现捕获多种标注类型（见 11.8 节的例子）的格式。NLP 的挑战是编写程序处理这种格式的泛化。例如：如果编程任务涉及树数据，文件格式允许任意有向图，那么必须验证输入数据检查树的属性如：根、连通性、无环。如果输入文件包含其他层的标注，该程序将需要知道数据加载时如何忽略它们，将树数据保存到文件时不能否定或抹杀这些层。

另一种反应一直是写一个一次性的脚本来操纵语料格式；这样的脚本将许多 NLP 研究人员的文件夹弄得乱七八糟。在语料格式解析工作应该只进行一次（每编程语言）的前提下，NLTK 中的语料库阅读器是更系统的方法。

不是集中在一种共同的格式，我们认为更有希望开发一种共同的接口（参见 nltk.corpus）。思考 NLP 中的一个重要的语料类型 treebanks 的情况。将短语结构树存储在一个文件中的方法很多。我们可以使用嵌套的括号、或嵌套的 XML 元素、或每行带有一个(child-id, parent-id)对的依赖符号、或一个 XML 版本的依赖符号等。然而，每种情况中的逻辑结构几乎是相同的。很容易设计一种共同的接口，使应用程序员编写代码使用如：children()、leaves()、depth()等方法来访问树数据。注意这种做法来自计算机科学中已经接受的做法，即抽象数据类型、面向对象设计、三层结构（图 11-6）。其中的最后一个——来自关系数据库领域——允许终端用户应用程序使用通用的模型（“关系模型”）和通用的语言（SQL）抽象出文件存储的特质，并允许新的文件系统技术的出现，而不会干扰到终端用户的应用。以同样的方式，一个通用的语料库接口将应用程序从数据格式隔离。

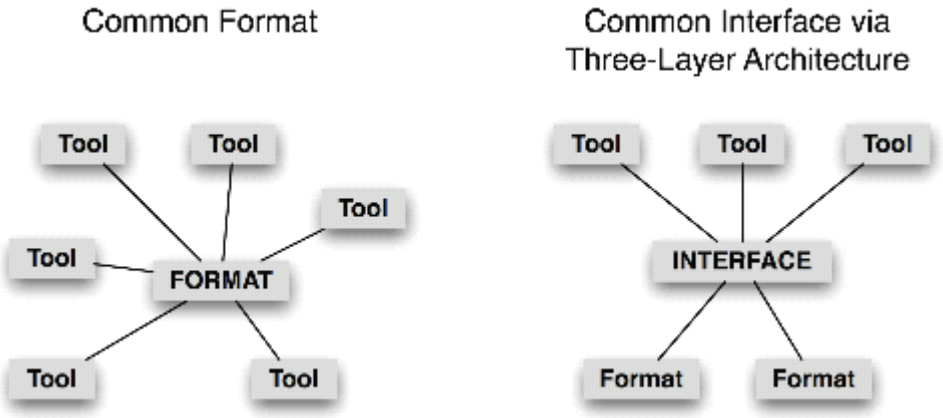


图 11-6. 通用格式对比通用接口

在此背景下，创建和发布一个新的语料库时，尽可能使用现有广泛使用的格式是权宜之计。如果这样不可能，语料库可以带有一些软件——如 nltk.corpus 模块——支持现有的接口方法。

处理濒危语言时特别注意事项

语言对科学和艺术的重要性体现在文化宝库包含在语言中。世界上大约 7000 种人类语言中的每一个都是丰富的，在它独特的方面，在它口述的历史和创造的传说，在它的文法结构和它的变化的词汇和它们含义中的细微差别。受威胁残余文化中的词能够区分具有科学家未知的治疗用途的植物亚种。当人们互相接触，每个人都为之前的语言提供一个独特的窗口，语言随着时间的推移而变化。世界许多地方，小的语言变化从一个镇都另一个镇，累加起来在一个半小时的车程的空间中成为一种完全不同的语言。对于其惊人的复杂性和多样性，人

类语言犹如丰富多彩的挂毯随着时间和空间而伸展。

然而，世界上大多数语言面临灭绝。对此，许多语言学家都在努力工作，记录语言，构建这个世界语言遗产的重要方面的丰富记录。在 NLP 的领域能为这方面的努力提供什么帮助吗？开发标注器、分析器、命名实体识别等不是最优先的，通常没有足够的数据来开发这样的工具。相反，最经常提出的是需要更好的工具来收集和维护数据，特别是文本和词汇。

从表面看，开始收集濒危语言的文本应该是一件简单的事情。即使我们忽略了棘手的问题，如：谁拥有文本，文本中包含的文化知识有关敏感性，转录仍然有很多明显的实际问题。大多数语言缺乏标准的书写形式。当一种语言没有文学传统时，拼写和标点符号的约定也没有得到很好的建立。因此，通常的做法是与文本收集一道创建一个词典。当在文本中出现新词时不断更新词典。可以使用文字处理器（用于文本）和电子表格（用于词典）来做这项工作。更妙的是，SIL 的自由软件 Toolbox 和 Fieldworks 对文本和词汇的创建集成提供了很好的支持。

当濒危语言的说话者学会自己输入文本时，一个共同的障碍就是对正确的拼写的极度关注。有一个词典大大有助于这一进程，但我们需要让查找的方法不要假设有人能确定任意一个词的引文形式。这个问题对具有复杂形态的包括前缀的语言可能是很急迫的。这种情况下，使用语义范畴标注词项，并允许通过语义范畴或注释查找是十分有益的。

允许通过相似的发音查找词项也是很有益的。下面是如何做到这一点的一个简单的演示。第一步是确定易混淆的字母序列，映射复杂的版本到更简单的版本。我们还可以注意到：辅音群中字母的相对顺序是拼写错误的一个来源，所以我们将辅音字母顺序规范化。

```
>>> mappings = [('ph', 'f'), ('ght', 't'), ('^kn', 'n'), ('qu', 'kw'),
...                 ('[aeiou]+', 'a'), (r'(.)\1', r'\1')]
>>> def signature(word):
...     for patt, repl in mappings:
...         word = re.sub(patt, repl, word)
...     pieces = re.findall('[âeiou]+', word)
...     return ''.join(char for piece in pieces for char in sorted(piece))[:8]
>>> signature('illefent')
'lfnt'
>>> signature('ebsekwieous')
'bskws'
>>> signature('nuculerr')
'nclr'
```

下一步，我们对词典中的所有词汇创建从特征到词汇的映射。我们可以用这为一个给定的输入词找到候选的修正（但我们必须先计算这个词的特征）。

```
>>> signatures = nltk.Index((signature(w), w) for w in nltk.corpus.words.words())
>>> signatures[signature('nuculerr')]
['anicular', 'inocular', 'nucellar', 'nuclear', 'unicolor', 'uniocular', 'unocular']
```

最后，我们应该按照与原词相似程度对结果排序。通过函数 rank()完成。唯一剩下的函数提供给用户一个简单的接口：

```
>>> def rank(word, wordlist):
...     ranked = sorted((nltk.edit_dist(word, w), w) for w in wordlist)
...     return [word for (_, word) in ranked]
>>> def fuzzy_spell(word):
...     sig = signature(word)
```

```
...         if sig in signatures:
...             return rank(word, signatures[sig])
...         else:
...             return []
>>> fuzzy_spell('illefent')
['olefiant', 'elephant', 'oliphant', 'elephanta']
>>> fuzzy_spell('ebsekwieous')
['obsequious']
>>> fuzzy_spell('nucular')
['nuclear', 'nucellar', 'anicular', 'inocular', 'unocular', 'unicolor', 'uniocular']
```

这仅仅是一个演示，其中一个简单的程序就可以方便的访问语言书写系统可能不规范或语言的使用者可能拼写的不是很好的上下文中的词汇数据。其他简单的 NLP 在这个领域的应用包括：建立索引以方便对数据的访问，从文本中拾取词汇表，构建词典时定位词语用法的例子，在知之甚少的数据中检测普遍或特殊模式，并在创建的数据上使用各种语言的软件工具执行专门的验证。我们将在 11.5 节返回到其中的最后一个。

11.4 使用 XML

可扩展标记语言（The Extensible Markup Language，XML）为设计特定领域的标记语言提供了一个框架。它有时被用于表示已标注的文本和词汇资源。不同于 HTML 的标签是预定义的，XML 允许我们组建自己的标签。不同于数据库，XML 允许我们创建的数据而不必事先指定其结构，它允许我们有可选的、可重复的元素。在本节中，我们简要回顾一下 XML 的一些与表示语言数据有关的特征，并说明如何使用 Python 程序访问 XML 文件中存储的数据。

语言结构中使用 XML

由于其灵活性和可扩展性，XML 是表示语言结构的自然选择。下面是一个简单的词汇条目的例子。

(2)
```
<entry>
    <headword>whale</headword>
    <pos>noun</pos>
    <gloss>any of the larger cetacean mammals having a streamlined
      body and breathing through a blowhole on the head</gloss>
</entry>
```

它由一系列括在尖括号中的 XML 标记组成。每个开始标记，如<gloss>，都匹配一个结束标记，如</gloss>；它们共同构成一个 XML 元素。上面的例子已经用空白符排版好了，它也可以放在长长的一行中。我们处理 XML 的方法通常不会注意空白符。在结构良好的 XML 中，在嵌套的同一级别中所有的开始标签必须有对应的结束标记（即 XML 文档必须是格式良好的树）。

XML 允许使用重复的元素，例如：添加另一个的 gloss 字段，如我们在下面看到的。我们将使用不同的空白符来强调：如何布局并不重要。

(3)
```
<entry><headword>whale</headword><pos>noun</pos><gloss>any of the
```

larger cetacean mammals having a streamlined body and breathing
through a blowhole on the head</gloss><gloss>a very large person;
impressive in size or qualities</gloss></entry>

再进一步可能是使用外部标识符链接我们的词汇到一些外部资源，如 WordNet。（4）中我们组合 gloss 和一个同义集标识符到一个我们称之为“sense”的新元素。

```
(4) <entry>
      <headword>whale</headword>
      <pos>noun</pos>
      <sense>
        <gloss>any of the larger cetacean mammals having a streamlined
          body and breathing through a blowhole on the head</gloss>
        <synset>whale.n.02</synset>
      </sense>
        <gloss>a very large person; impressive in size or qualities</gloss>
        <synset>giant.n.04</synset>
      </sense>
    </entry>
```

另外，我们可以使用 **XML 属性**表示同义集标识符，不必使用（5）中的嵌套结构。

```
(5)<entry>
      <headword>whale</headword>
      <pos>noun</pos>
      <gloss synset="whale.n.02">any of the larger cetacean mammals having
          a streamlined body and breathing through a blowhole on the head</gloss>
      <gloss synset="giant.n.04">a very large person; impressive in size or
          qualities</gloss>
    </entry>
```

这说明了 XML 的某些灵活性。如果它看上去似乎有点随意，这是因为它就是很随意！遵循 XML 的规则，我们可以指定新的属性名称，任意的嵌套它们。我们可以重复元素，省略它们，让它们每次的顺序都不同。我们可以指定字段，它的出现取决于其他一些字段的值，例如：如果词性是动词，那么条目可以有一个 past_tense 元素保存动词的过去式，但如果词性是名词，就不允许有 past_tense 元素。为了在这种自由上加一些秩序，我们用“架构（schema）”限制一个 XML 文件的格式，这是一种类似于上下文无关文法的声明。存在工具可以测试带有架构的 XML 的有效性。

XML 的作用

我们可以用 XML 来表示许多种语言信息。然而，灵活性是要付出代价的。每次我们增加复杂性，如：允许一个元素是可选的或重复的，我们对所有访问这些数据的程序都要做出更多的工作。我们也使它更难以检查数据的有效性，或使用一种 XML 查询语言来查询数据。

因此，使用 XML 来表示语言结构并不能神奇地解决数据建模问题。我们仍然需要解决如何结构化数据，然后用一个架构定义结构，并编写程序读取和写入格式，以及把它转换为其他格式。同样，我们仍然需要遵循一些有关数据规范化的标准原则。这是明智的，可以避免相同信息的重复复制，所以当只有一个副本变化时，不会导致数据不一致。例如：交叉引用表示为<xref>headword</xref>将重复存储一些其他词条的核心词，如果在其他位置

的字符串的副本被修改，链接就会被打断。信息类型之间存在的依赖关系需要建模，使我们不能创建没有根的元素。例如：如果 sense 的定义不能作为词条独立存在，那么 sense 就要嵌套在 entry 元素中。多对多关系需要从层次结构中抽象出来。例如：如果一个 word 可以有很多对应的 senses，一个 sense 可以有几个对应的 words，而 words 和 senses 都必须作为(word, sense)对的列表分别枚举。这种复杂的结构甚至可以分割成三个独立的 XML 文件。

正如我们看到的，虽然 XML 提供了一个格式方便和用途广泛的工具，但它不是能解决一切问题的灵丹妙药。

ElementTree 接口

Python 的 ElementTree 模块提供了一种方便的方式访问存储在 XML 文件中的数据。ElementTree 是 Python 标准库（自从 Python 2.5）的一部分，也作为 NLTK 的一部分提供，万一你在使用 Python 2.4。

我们将使用 XML 格式的莎士比亚戏剧集来说明 ElementTree 的使用方法。让我们加载 XML 文件并检查原始数据，首先在文件的顶部①，在那里我们看到一些 XML 头和一个名为 play.dtd 的架构，接着是根元素 PLAY。我们从 Act 1②再次获得数据。（输出中省略了一些空白行。）

```
>>> merchant_file = nltk.data.find('corpora/shakespeare/merchant.xml')
>>> raw = open(merchant_file).read()
>>> print raw[0:168]  ①
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="shakes.css"?>
<!-- <!DOCTYPE PLAY SYSTEM "play.dtd"> -->
<PLAY>
<TITLE>The Merchant of Venice</TITLE>
>>> print raw[1850:2075]  ②
<TITLE>ACT I</TITLE>
<SCENE><TITLE>SCENE I.  Venice. A street.</TITLE>
<STAGEDIR>Enter ANTONIO, SALARINO, and SALANIO</STAGEDIR>
<SPEECH>
<SPEAKER>ANTONIO</SPEAKER>
<LINE>In sooth, I know not why I am so sad:</LINE>
```

我们刚刚访问了作为一个字符串的 XML 数据。正如我们看到的，在 Act 1 开始处的字符串包含 XML 标记：title、scene、stage directions 等。

下一步是作为结构化的 XML 数据使用 ElementTree 处理文件的内容。我们正在处理一个文件（一个多行字符串），并建立一棵树，所以方法的名称是 parse①并不奇怪。变量 merchant 包含一个 XML 元素 PLAY②。此元素有内部结构；我们可以使用一个索引来得到它的第一个孩子，一个 TITLE 元素③。我们还可以看到该元素的文本内容：戏剧的标题④。要得到所有的子元素的列表，我们使用 getchildren()方法⑤。

```
>>> from nltk.etree.ElementTree import ElementTree
>>> merchant = ElementTree().parse(merchant_file)  ①
>>> merchant
<Element PLAY at 22fa800> ②
>>> merchant[0]
```

```
<Element TITLE at 22fa828> ③
>>> merchant[0].text
'The Merchant of Venice' ④
>>> merchant.getchildren()  ⑤
[<Element TITLE at 22fa828>, <Element PERSONAE at 22fa7b0>, <Element SCNDE
SCR at 2300170>,
<Element PLAYSUBT at 2300198>, <Element ACT at 23001e8>, <Element ACT at 2
34ec88>,
<Element ACT at 23c87d8>, <Element ACT at 2439198>, <Element ACT at 24923c8
>]
```

这部戏剧由标题、角色、一个场景的描述、字幕和五幕组成。每一幕都有一个标题和一些场景，每个场景由台词组成，台词由行组成，有四个层次嵌套的结构。让我们深入到第四幕：

```
>>> merchant[-2][0].text
'ACT IV'
>>> merchant[-2][1]
<Element SCENE at 224cf80>
>>> merchant[-2][1][0].text
'SCENE I.  Venice. A court of justice.'
>>> merchant[-2][1][54]
<Element SPEECH at 226ee40>
>>> merchant[-2][1][54][0]
<Element SPEAKER at 226ee90>
>>> merchant[-2][1][54][0].text
'PORTIA'
>>> merchant[-2][1][54][1]
<Element LINE at 226eee0>
>>> merchant[-2][1][54][1].text
"The quality of mercy is not strain'd,"
```

轮到你来：对语料库中包含的其他莎士比亚戏剧，如：《罗密欧与朱丽叶》或《麦克白》，重复上述的一些方法。方法列表请参阅 nltk.corpus.shakespeare.fileids()。

虽然我们可以通过这种方式访问整个树，使用特定名称查找子元素会更加方便。回想一下顶层的元素有几种类型。我们可以使用 merchant.findall('ACT')遍历我们感兴趣的类型（如：幕）。下面是一个做这种特定标记在每一个级别的嵌套搜索的例子：

```
>>> for i, act in enumerate(merchant.findall('ACT')):
...     for j, scene in enumerate(act.findall('SCENE')):
...         for k, speech in enumerate(scene.findall('SPEECH')):
...             for line in speech.findall('LINE'):
...                 if 'music' in str(line.text):
...                     print "Act %d Scene %d Speech %d: %s" % (i+1, j+1, k+1, line.text)
Act 3 Scene 2 Speech 9: Let music sound while he doth make his choice;
```

```
Act 3 Scene 2 Speech 9: Fading in music: that the comparison
Act 3 Scene 2 Speech 9: And what is music then? Then music is
Act 5 Scene 1 Speech 23: And bring your music forth into the air.
Act 5 Scene 1 Speech 23: Here will we sit and let the sounds of music
Act 5 Scene 1 Speech 23: And draw her home with music.
Act 5 Scene 1 Speech 24: I am never merry when I hear sweet music.
Act 5 Scene 1 Speech 25: Or any air of music touch their ears,
Act 5 Scene 1 Speech 25: By the sweet power of music: therefore the poet
Act 5 Scene 1 Speech 25: But music for the time doth change his nature.
Act 5 Scene 1 Speech 25: The man that hath no music in himself,
Act 5 Scene 1 Speech 25: Let no such man be trusted. Mark the music.
Act 5 Scene 1 Speech 29: It is your music, madam, of the house.
Act 5 Scene 1 Speech 32: No better a musician than the wren.
```

不是沿着层次结构向下遍历每一级，我们可以寻找特定的嵌入的元素。例如：让我们来看看演员的顺序。我们可以使用频率分布看看谁最能说：

```
>>> speaker_seq = [s.text for s in merchant.findall('ACT/SCENE/SPEECH/SPEAKER
')]
>>> speaker_freq = nltk.FreqDist(speaker_seq)
>>> top5 = speaker_freq.keys()[:5]
>>> top5
['PORTIA', 'SHYLOCK', 'BASSANIO', 'GRATIANO', 'ANTONIO']
```

我们也可以查看对话中谁跟着谁的模式。由于有 23 个演员，我们需要首先使用 5.3 节中描述的方法将“词汇”减少到可处理的大小。

```
>>> mapping = nltk.defaultdict(lambda: 'OTH')
>>> for s in top5:
...     mapping[s] = s[:4]
...
>>> speaker_seq2 = [mapping[s] for s in speaker_seq]
>>> cfd = nltk.ConditionalFreqDist(nltk.ibigrams(speaker_seq2))
>>> cfd.tabulate()
      ANTO BASS GRAT OTH PORT SHYL
ANTO     0   11    4  11    9   12
BASS    10    0   11  10   26   16
GRAT     6    8    0  19    9    5
 OTH     8   16   18 153   52   25
PORT     7   23   13  53    0   21
SHYL    15   15    2  26   21    0
```

忽略 153 的条目，因为是前五位角色之间相互对话，最大的值表示 Othello 和 Portia 的相互对话最多。

使用 ElementTree 访问 Toolbox 数据

在 2.4 节中，我们看到了一个访问 Toolbox 数据的简单的接口，Toolbox 数据是语言学家用来管理数据的一种流行和行之有效的格式。这一节中，我们将讨论以 Toolbox 软件所不

支持的方式操纵 Toolbox 数据的各种技术。我们讨论的方法也可以应用到其他记录结构化数据，不必管实际的文件格式。

我们可以用的 toolbox.xml()方法来访问 Toolbox 文件，将它加载到一个 ElementTree 对象中。此文件包含一个巴布亚新几内亚罗托卡特语的词典。

```
>>> from nltk.corpus import toolbox
>>> lexicon = toolbox.xml('rotokas.dic')
```

有两种方法可以访问 lexicon 对象的内容：通过索引和通过路径。索引使用熟悉的语法；lexicon[3]返回 3 号条目（实际上是从 0 算起的第 4 个条目），lexicon[3][0]返回它的第一个字段：

```
>>> lexicon[3][0]
<Element lx at 77bd28>
>>> lexicon[3][0].tag
'lx'
>>> lexicon[3][0].text
'kaa'
```

第二种方式访问 lexicon 对象的内容是使用路径。lexicon 是一系列 record 对象，其中每个都包含一系列字段对象，如 lx 和 ps。使用的路径 record/lx，我们可以很方便地解决所有的语意。在这里，我们使用 FindAll()函数来搜索路径 record/lx 的所有匹配，并且访问该元素的文本内容，将其规范化为小写：

```
>>> [lexeme.text.lower() for lexeme in lexicon.findall('record/lx')]
['kaa', 'kaa', 'kaa', 'kaakaaro', 'kaakaaviko', 'kaakaavo', 'kaakaoko',
'kaakasi', 'kaakau', 'kaakauko', 'kaakito', 'kaakuupato', ..., 'kuvuto']
```

让我们查看 XML 格式的 Toolbox 数据。ElementTree 的 write()方法需要一个文件对象。我们通常使用 Python 的内置在 open()函数创建。为了屏幕上显示输出，我们可以使用一个特殊的预定义的文件对象称为 stdout①（标准输出），在 Python 的 sys 模块中定义的。

```
>>> import sys
>>> from nltk.etree.ElementTree import ElementTree
>>> tree = ElementTree(lexicon[3])
>>> tree.write(sys.stdout) ①
<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 kovopaueva 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>
```

格式化条目

我们可以使用在前一节看到的同样的想法生成 HTML 表格而不是纯文本。这对于将 Toolbox 词汇发布到网络上非常有用。它产生 HTML 元素<table>、<tr>(表格的行)以及<td>(表格数据)。

```
>>> 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
<table>
  <tr><td>kakae</td><td>???</td><td>small</td></tr>
  <tr><td>kakae</td><td>CLASS</td><td>child</td></tr>
  <tr><td>kakaevira</td><td>ADV</td><td>small-like</td></tr>
  <tr><td>kakapikoa</td><td>???</td><td>small</td></tr>
  <tr><td>kakapikoto</td><td>N</td><td>newborn baby</td></tr>
  <tr><td>kakapu</td><td>V</td><td>place in sling for purpose of carrying</td></tr>
  <tr><td>kakapua</td><td>N</td><td>sling for lifting</td></tr>
  <tr><td>kakara</td><td>N</td><td>arm band</td></tr>
  <tr><td>Kakarapaia</td><td>N</td><td>village name</td></tr>
  <tr><td>kakarau</td><td>N</td><td>frog</td></tr>
</table>
```

11.5 使用 Toolbox 数据

鉴于 Toolbox 在语言学家中十分流行，我们将讨论一些使用 Toolbox 数据的进一步的方法。很多在前面的章节讲过的方法，如计数、建立频率分布、为同现制表，这些都可以应用到 Toolbox 条目的内容上。例如：我们可以为每个条目计算字段的平均个数：

```
>>> from nltk.corpus import toolbox
>>> lexicon = toolbox.xml('rotokas.dic')
>>> sum(len(entry) for entry in lexicon) / len(lexicon)
13.635955056179775
```

在本节中我们将讨论记录语言学的背景下出现的都不被 Toolbox 软件支持的两个任务。

为每个条目添加一个字段

添加一个自动从现有字段派生出的新的字段往往是方便的。这些字段经常使搜索和分析更加便捷。例如：例 11-2 中我们定义了一个函数 cv()，将辅音和元音的字符串映射到相应的 CV 序列，即：kakapua 将映射到 CVCVCVV。这种映射有四个步骤。首先，将字符串

转换为小写，那么将所有非字母字符[^a-z]用下划线代替。下一步，将所有元音替换为 V。最后，所有不是 V 或下划线的必定是一个辅音，所以我们将它替换为 C。现在，我们可以扫描词汇，在每个 lx 字段后面添加一个新的 cv 字段。例 11-2 显示了在一个特定条目上做这些；注意输出的最后一行表示新的 cv 字段。

例 11-2. 为词汇条目添加新的 cv 字段

```
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.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
```

如果一个 Toolbox 文件正在不断更新，例 11-2 将需要多次运行。可以将 add_cv_field()修改为修改现有条目的内容。使用这样的程序为数据分析创建一个附加的文件比替换手工维护的源文件要安全。

验证 Toolbox 词汇

Toolbox 格式的许多词汇不符合任何特定的模式。有些条目可能包括额外的字段，或以一种新的方式排序现有字段。手动检查成千上万的词汇条目是不可行的。我们可以在 Freq

Dist 的帮助下很容易地找出频率异常的字段序列：

```
>>> fd = nltk.FreqDist(':'.join(field.tag for field in entry) for entry in lexicon)
>>> fd.items()
[('lx:ps:pt:ge:tkp:dt:ex:xp:xe', 41), ('lx:rt:ps:pt:ge:tkp:dt:ex:xp:xe', 37),
('lx:rt:ps:pt:ge:tkp:dt:ex:xp:xe:ex:xp:xe', 27), ('lx:ps:pt:ge:tkp:nt:dt:ex:xp:xe', 20),
..., ('lx:alt:rt:ps:pt:ge:eng:eng:eng:tkp:tkp:dt:ex:xp:xe:ex:xp:xe:ex:xp:xe', 1)]
```

检查完高频字段序列后，我们可以设计一个词汇条目的上下文无关文法。例 11-3 中的文法使用我们在第 8 章看到的 CFG 的格式。这样的文法模型隐含 Toolbox 条目的嵌套结构，建立一个树状结构，树的叶子是单独的字段名。我们遍历条目并报告它们与文法的一致性，如例 11-3 所示。那些被文法接受的在前面加一个“+”①，那些被文法接受的在前面加一个“-”②。在开发这样一个文法的过程中，它可以帮助过滤掉一些标签③。

例 11-3. 使用上下文无关文法验证 Toolbox 中的条目。

```
grammar = nltk.parse_cfg("'
  S -> Head PS Glosses Comment Date Sem_Field Examples
  Head -> Lexeme Root
  Lexeme -> "lx"
  Root -> "rt" |
  PS -> "ps"
  Glosses -> Gloss Glosses |
  Gloss -> "ge" | "tkp" | "eng"
  Date -> "dt"
  Sem_Field -> "sf"
  Examples -> Example Ex_Pidgin Ex_English Examples |
  Example -> "ex"
  Ex_Pidgin -> "xp"
  Ex_English -> "xe"
  Comment -> "cmt" | "nt" |
  "')
def validate_lexicon(grammar, lexicon, ignored_tags):
    rd_parser = nltk.RecursiveDescentParser(grammar)
    for entry in lexicon:
        marker_list = [field.tag for field in entry if field.tag not in ignored_tags]
        if rd_parser.nbest_parse(marker_list):
            print "+", ':'.join(marker_list)  ①
        else:
            print "-", ':'.join(marker_list)  ②
>>> lexicon = toolbox.xml('rotokas.dic')[10:20]
>>> ignored_tags = ['arg', 'dcsv', 'pt', 'vx']  ③
>>> validate_lexicon(grammar, lexicon, ignored_tags)
- lx:ps:ge:tkp:sf:nt:dt:ex:xp:xe:ex:xp:xe:ex:xp:xe
- lx:rt:ps:ge:tkp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:tkp:nt:dt:ex:xp:xe:ex:xp:xe
- lx:ps:ge:tkp:nt:sf:dt
- lx:ps:ge:tkp:dt:cmt:ex:xp:xe:ex:xp:xe
```

- lx:ps:ge:ge:ge:tkp:cmt:dt:ex:xp:xe
- lx:rt:ps:ge:ge:tkp:dt
- lx:rt:ps:ge:eng:eng:eng:ge:tkp:tkp:dt:cmt:ex:xp:xe:ex:xp:xe:ex:xp:xe:ex:xp:xe:ex:xp:xe
- lx:rt:ps:ge:tkp:dt:ex:xp:xe
- lx:ps:ge:ge:tkp:dt:ex:xp:xe:ex:xp:xe

另一种方法是用一个块分析器（第 7 章），因为它能识别局部结构并报告已确定的局部结构，会更加有效。例 11-4 中我们为词汇条目建立一个块文法。这个程序的输出的一个示例如图 11-7 所示。

图 11-4. 为 Toolbox 词典分块：此块文法描述一种中国的语言勉方言的词汇条目结构。

```
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|rn|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.parse(grammar, encoding='utf8')
>>> toolbox.data.indent(lexicon)
>>> tree = ElementTree(lexicon)
>>> output = open("iu_mien_samp.xml", "w")
>>> tree.write(output, encoding='utf8')
>>> output.close()
```

```
<record>
  <lx>ceuv jiax</lx>
  <hm />
  <sense>
    <sn />
    <ps>vobj</ps>
    <dv>nzaeng jiax</dv>
    <ge>quarrel</ge>
    <de />
    <gn>吵架</gn>
    <gp>chao3 jia4</gp>
    <dn>争吵</dn>
    <example>
      <xv>Ninh mbuo i hmuangv mv - jiex jiax.</xv>
      <xe>That husband and wife have never quarrelled.</xe>
      <xn>他们夫妻俩从来不吵架。</xn>
    </example><example>
      <xv>Gorngv duh leiz mv duqv -.</xv>
      <xe>Have some common sense, don't quarrel.</xe>
      <xn>讲道理，别吵架。</xn>
    </example><lexfunc>
      <lf />
      <lv />
    </lexfunc>
  </sense><dt>18/Feb/2004</dt>
</record>
```

图 11-7. 一个词条的 XML 表示，对 Toolbox 记录的块分析的结果。

11.6 使用 OLAC 元数据描述语言资源

NLP 社区的成员的一个共同需要是发现具有很高精度和召回率的语言资源。数字图书馆社区目前已开发的解决方案包括元数据聚集。

元数据是什么？

元数据最简单的定义是“关于数据的结构化数据”。元数据是对象或资源的描述信息，无论是物理的还是电子的。而术语“元数据”本身是相对较新的，只要收集的信息被组织起来，元数据下面隐含的意义却一直在被使用。图书馆目录是一种行之有效的元数据类型；它们已经作为资源管理和发现工具有几十年了。元数据可以由“手工”产生也可以使用软件自动生成。

都柏林核心元数据倡议（Dublin Core Metadata Initiative）于 1995 年开始开发的信息发现、共享和管理的约定。都柏林核心元数据元素表示一个广泛的、跨学科一致的元素核心集合，这些元素核心集合有可能对资源发现有广泛作用。都柏林核心由 15 个元数据元素组成，其中每个元素都是可选的和可重复的，它们是：标题，创建者，主题，描述，发布者，参与者，日期，类型，格式，标识符，来源，语言，关系，覆盖范围和版权。此元数据集可以用来描述数字或传统的格式中存放的资源。

开放档案倡议（Open Archives Initiative，OAI）提供了一个跨越数字化的学术资料库的共同框架，不考虑资源的类型，包括文档，资料，软件，录音，实物制品，数码代替品等等。每个库由一个网络访问服务器提供归档项目的公共访问。每个项目都有一个唯一的标识符，并与都柏林核心元数据记录（也可以是其他格式的记录）关联。OAI 为元数据搜索服务定义了一个协议来“收获”资源库的内容。

OLAC：开放语言档案社区

开放语言档案社区（Open Language Archives Community，OLAC）是正在创建的一个世界性语言资源的虚拟图书馆的机构和个人的国际伙伴关系：（i）制订目前最好的关于语言资源的数字归档实施的共识，（ii）开发存储和访问这些资源的互操作信息库和服务的网络。OLAC 的主页在 http://www.language-archives.org/。

OLAC 元数据是描述语言资源的标准。通过限制某些元数据元素的值为使用受控词表中的术语，确保跨库描述的统一性。OLAC 元数据可用于描述物理和数字格式的数据和工具。OLAC 元数据扩展了都柏林核心元数据集（一个描述所有类型的资源被广泛接受的标准）。对这个核心集，OLAC 添加了语言资源的基本属性，如：主题语言和语言类型。下面是一个完整的 OLAC 记录的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<olac:olac xmlns:olac="http://www.language-archives.org/OLAC/1.1/"
           xmlns="http://purl.org/dc/elements/1.1/"
           xmlns:dcterms="http://purl.org/dc/terms/"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.language-archives.org/OLAC/1.1/
                http://www.language-archives.org/OLAC/1.1/olac.xsd">
```

```
    <title>A grammar of Kayardild. With comparative notes on Tangkic.</title>
    <creator>Evans, Nicholas D.</creator>
    <subject>Kayardild grammar</subject>
    <subject xsi:type="olac:language" olac:code="gyd">Kayardild</subject>
    <language xsi:type="olac:language" olac:code="en">English</language>
    <description>Kayardild Grammar (ISBN 3110127954)</description>
    <publisher>Berlin - Mouton de Gruyter</publisher>
    <contributor xsi:type="olac:role" olac:code="author">Nicholas Evans</contributor>
    <format>hardcover, 837 pages</format>
    <relation>related to ISBN 0646119966</relation>
    <coverage>Australia</coverage>
    <type xsi:type="olac:linguistic-type" olac:code="language_description"/>
    <type xsi:type="dcterms:DCMIType">Text</type>
</olac:olac>
```

以 XML 格式发布语言档案的目录，这些记录会定期被使用 OAI 协议的 OLAC 服务“收获”。除了这个软件基础设施外，OLAC 通过与语言资源社区广泛咨询的过程，已经证明了它是一系列描述语言资源的最佳实践（如：参见 http://www.language-archives.org/REC/bpr.html。）。

OLAC 网站上使用一个查询引擎可以搜索 OLAC 库。搜索“German lexicon”可以发现以下资源，其中包括：

- CALLHOME 德语词典 http://www.language-archives.org/item/oai:www.ldc.upenn.edu:LDC97L18
- MULTILEX 多语种词典 http://www.language-archives.org/item/oai:elra.icp.inpg.fr:M0001
- Slelex 西门子语音词汇 http://www.language-archives.org/item/oai:elra.icp.inpg.fr:S0048

搜索“Korean”会发现一个新闻语料库，一个树库，一个词库，一个儿童语言语料库和行间注释文本。它还可以发现软件，包括一个语法分析器和一个词法分析器。

可以看到上面的网址包括一个子字符串的形式：oai:www.ldc.upenn.edu:LDC97L18。这是一个的 OAI 标识符，使用在 ICANN（the Internet Corporation for Assigned Names and Numbers，互联网名称和编号分配公司）注册的 URI 方案。这些标识符格式是：oai:*archive*:*local_id*，其中 oai 是 URI 方案的名称，*archive* 是一个档案标识符，如：www.ldc.upenn.edu，*local_id* 是档案分配的资源标识符，如：LDC97L18。

给定一个 OLAC 资源的 OAI 标识符，可以使用以下形式的 URL 检索资源的完整的 XML 记录：http://www.language-archives.org/static-records/oai:archive:local_id。

11.7 小结

- 大多数语料库中基本数据类型是已标注的文本和词汇。文本有时间结构，而词汇有记录结构。
- 语料库的生命周期，包括数据收集、标注、质量控制以及发布。发布后生命周期仍然继续，因为语料库会在研究过程中被修改和丰富。
- 语料库开发包括捕捉语言使用的代表性的样本与使用任何一个来源或文体都有足够的材料之间的平衡；增加变量的维度通常由于资源的限制而不可行。
- XML 提供了一种有用的语言数据的存储和交换格式，但解决普遍存在的数据建模问题没有捷径。

- Toolbox 格式被广泛使用在语言记录项目中；我们可以编写程序来支持 Toolbox 文件的维护，将它们转换成 XML。
- 开放语言档案社区（OLAC）提供了一个用于记录和发现语言资源的基础设施。

11.8 进一步阅读

本章的其它材料发布在 http://www.nltk.org/，包括网上免费资源的链接。

语言学语料库的首要来源是语言数据联盟（Linguistic Data Consortium）和欧洲语言资源局（European Language Resources Agency），两者都有广泛的在线目录。本书中提到的主要语料库的细节也有介绍：美国国家语料库(Reppen, Ide & Suderman, 2005)、英国国家语料库(BNC,1999)，Thesaurus Linguae Graecae(TLG, 1999)、儿童语言数据交换系统(MacWhinney, 1995)和 TIMIT(Garofolo et al., 1986)。

计算语言学协会定期组织研讨会发布论文集，它的两个特别兴趣组：SIGWAC 和 SIGANN。前者推动使用网络作为语料，发起去除 HTML 标记的 CLEANEVAL 任务；后者鼓励对语言注解的互操作性的努力。(Croft,Metzler & Strohman, 2009)提供了网络爬虫的扩展讨论。

(Buseman, Buseman & Early, 1996)提供 Toolbox 数据格式的全部细节，最新的发布可以从 http://www.sil.org/computing/toolbox/ 免费下载。构建一个 Toolbox 词典的过程指南参见 http://www.sil.org/computing/ddp/。我们在 Toolbox 上努力的更多的例子记录在(Bird, 1999)和(Robinson, Aumann & Bird, 2007)。(Bird & Simons, 2003)调查了语言数据管理的几十个其他工具。也请参阅关于文化遗产数据的语言技术的 LaTeCH 研讨会的论文集。

有很多优秀的 XML 资源（如 http://zvon.org/）和编写 Python 程序处理 XML 的资源 http://www.python.org/doc/lib/markup.html。许多编辑器都有 XML 模式。XML 格式的词汇信息包括 OLIF（http://www.olif.net/）和 LIFT（http://code.google.com/p/lift-standard/）。

对于语言标注软件的调查，见 http://www.ldc.upenn.edu/annotation/ 的语言标注页。对峙注解最初的设计(Thompson & McKelvie, 1997)。语言标注的一个抽象的数据模型称为“标注图”在(Bird & Liberman, 2001)提出。语言描述的一个通用本体（GOLD）记录在 http://www.linguistics-ontology.org/中。

有关规划和建设语料库的指导，请参阅 (Meyer, 2002) 和(Farghaly,2003)。关于标注者之间一致性得分的方法的更多细节，见(Artstein & Poesio, 2008)和(Pevzner & Hearst, 2002)。

Rotokas 数据由 Stuart Robinson 提供，勉方言（Iu Mien）数据由 Greg Aumann 提供。

有关开放语言档案社区的更多信息，请访问 http://www.language-archives.org/，或参见(Simons & Bird, 2003)。

11.9 练习

1. ◑在例 11-2 中新字段出现在条目底部。修改这个程序使它就在 lx 字段后面插入新的子元素。（提示：使用 Element('cv')创建新的 cv 字段，分配给它一个文本值，然后使用父元素的 insert()方法。）
2. ◑编写一个函数，从一个词汇条目删除指定的字段。（我们可以在把数据给别人之前用它做些清洁，如：删除包含无关或不确定的内容的字段。）
3. ◑写一个程序，扫描一个 HTML 字典文件，找出具有非法词性字段的条目，并报告每

个条目的核心词。

4. ◑写一个程序，找出所有出现少于 10 次的词性（ps 字段）。或许有打字错误？
5. ◑我们看到一个增加 cv 字段的方法（11.5 节）。一件有趣的事情是当有人修改的 lx 字段的内容时，保持这个字段始终最新。为这个程序写一个版本，添加 cv 字段，取代所有现有的 cv 字段。
6. ◑写一个函数，添加一个新的字段 syl，计数一个词中的音节数。
7. ◑写一个函数，显示一个词位的完整条目。当词位拼写错误时，它应该显示拼写最相似的词位的条目。
8. ◑写一个函数，从一个词典中找出最频繁的连续字段对（如：ps 后面往往是 pt）。（这可以帮助我们发现一些词条的结构。）
9. ◑使用办公软件创建一个电子表格，每行包含一个词条，包括一个中心词，词性和注释。以 CSV 格式保存电子表格。写 Python 代码来读取 CSV 文件并以 Toolbox 格式输出。使用 lx 表示中心词，ps 表示词性，gl 表示注释。
10. ◑在 nltk.Index 帮助下，索引莎士比亚的戏剧中的词。产生的数据结构允许按单个词查找，如：music，返回演出，场景和台词的引用的链表，[(3, 2, 9),(5, 1, 23), ...] 的形式，其中(3, 2, 9)表示第 3 场演出场景 2 台词 9。
11. ◑构建一个条件频率分布记录《威尼斯商人》中每段台词的词长，以角色名字为条件；如：cfd['PORTIA'][12]会给我们 Portia 的 12 个词的台词的数目。
12. ◑写一个递归函数将任意 NLTK 树转换为对应的 XML，其中非终结符不能表示成 XML 元素，叶子表示文本内容，如：

```
<S>
  <NP type="SBJ">
    <NP>
      <NNP>Pierre</NNP>
      <NNP>Vinken</NNP>
    </NP>
    <COMMA>,</COMMA>
```

13. ●获取 CSV 格式的比较词表，写一个程序，输出相互之间至少有三个编辑距离的同源词。
14. ●建立一个出现在例句的词位的索引。假设对于一个给定条目的词位是 w。然后为这个条目添加一个单独的交叉引用字段 xrf，引用其它有例句包含 w 的条目的中心词。对所有条目做这个，结果保存为 Toolbox 格式文件。

后记：语言的挑战

自然语言带来了一些有趣的计算性挑战。我们已经在前面的章节探讨过许多这样的挑战，包括分词、标注、分类、信息提取和建立句法和语义表示。你现在应该已经准备好操作大型数据集来创建强健的语言现象的模型，并将它们扩展到实际语言技术的组件中。我们希望自然语言工具包（NLTK）已经开启的实用自然语言处理的令人振奋的努力能有更广泛的受众。

尽管有了前面的，语言带给我们的远远比临时的计算性挑战要大得多。考虑下面的句子，它们证明了语言的丰富性：

1. Overhead the day drives level and grey, hiding the sun by a flight of grey spears. (William Faulkner, As I Lay Dying, 1935)
2. When using the toaster please ensure that the exhaust fan is turned on. (sign in dormitory kitchen)
3. Amiodarone weakly inhibited CYP2C9, CYP2D6, and CYP3A4-mediated activities with Ki values of 45.1-271.6 μM (Medline, PMID: 10718780)
4. Iraqi Head Seeks Arms (spoof news headline)
5. The earnest prayer of a righteous man has great power and wonderful results. (James 5:16b)
6. Twas brillig, and the slithy toves did gyre and gimble in the wabe (Lewis Carroll, Jabberwocky, 1872)
7. There are two ways to do this, AFAIK :smile: (Internet discussion archive)

语言丰富性的其他证据是以语言为工作中心的学科繁多。一些明显的学科包括翻译、文学批评、哲学、人类学和心理学。许多不太明显的学科研究语言的使用，包括法律、诠释学、辩论术、电话学、教育学、考古学、密码分析学及言语病理学。它们分别应用不同的方法来收集现象、发展理论和测试假设。它们都有助于加深我们对语言和表现在语言上的智能的理解。

鉴于语言的复杂性和从不同的角度研究它的广泛的兴趣，很显然，我们仅仅触及了表面。此外，NLP 的本身也有许多我们没有提到的重要方法和应用。

在后记中，我们将以更宽广的视角看待 NLP，包括它的基础和你可能想要探索的进一步的方向。一些主题还没有得到 NLTK 很好的支持，你可能想通过为工具包贡献新的软件和数据，修正这些问题，。

语言处理与符号处理

以计算方式处理自然语言的观念脱胎于一个研究项目，可以追溯到 20 世纪初，使用逻辑重建数学推理，Frege、Russell、Wittgenstein、Tarski、Lambek 和 Carnap 的工作清楚的表明了这一点。这项工作导致语言作为可以自动处理的形式化系统的概念。3 个后来的开发奠定了自然语言处理的基础。第一个是**形式语言理论**。它定义一个语言为被一类自动机接受的字符串的集合，如：上下文无关语言和下推自动机，并提供了计算句法的基础。

第二个开发是**符号逻辑**。它提供了一个捕捉选定的自然语言的表达的逻辑证明的有关方面的形式化方法。符号逻辑中的形式化演算提供一种语言的句法和推理规则，并可能在一套

理论模型中对规则进行解释；例子是命题逻辑和一阶逻辑。给定这样的演算和一个明确的句法和语义，通过将自然语言的表达翻译成形式化演算的表达式，联系语义与自然语言的表达成为可能。例如：如果我们翻译 John saw Mary 为公式 saw(j, m)，我们（或明或暗地）将英语动词 saw 解释为一个二元关系，而 John 和 Mary 表示个体元素。更多的一般性的表达式如：All birds fly 需要量词，在这个例子中是∀，意思是"对所有的"：∀x (bird(x) → fly(x))。逻辑的使用提供了技术性的机制处理推理，而推理是语言理解的重要组成部分。

另一个密切相关的开发是**组合原则**，即一个复杂表达式的意思由它的各个部分的意思和它们的组合模式组成（第 10 章）。这一原则提供了句法和语义之间的有用的对应，即一个复杂的表达式的含义可以递归的计算。考虑句子：It is not true that p，其中 p 是一个命题。我们可以表示这个句子的意思为 not(p)。同样，我们可以表示 John saw Mary 的意思为 saw(j, m)。现在，我们可以使用上述信息递归的计算 It is not true that John saw Mary 的表示，得到 not(saw(j,m))。

刚刚简要介绍的方法都有一个前提：自然语言计算关键依赖于操纵符号表示的规则。NLP 发展的一个特定时期，特别是在 20 世纪 80 年代，这个前提为语言学家和 NLP 从业人员提供了一个共同的起点，导致一种被称为基于归一（基于特征）文法的形式化文法家族（见第 9 章），也导致了在 Prolog 编程语言上实现 NLP 应用。虽然基于文法的自然语言处理仍然是一个研究的重要领域，由于多种因素在过去的 15-20 年它已经有些黯然失色。一个显著的影响因素来自于自动语音识别。虽然早期的语音处理采用一个模拟一类基于规则的**音韵处理**的模型，典型的如：Sound Pattern of English（Chomsky & Halle, 1968），结果远远不能够解决实时的识别实际的讲话这样困难的问题。相比之下，包含从大量语音数据中学习的模式的系统明显更准确、高效和稳健的。此外，言语社区发现建立对常见的测试数据的性能的定量测量的共享资源对建立更好的系统的过程有巨大帮助。最终，大部分的 NLP 社区拥抱面向数据密集型的语言处理，配合机器学习技术和评价为主导的方法的越来越多地使用。

当代哲学划分

在上一节中描述的自然语言处理的两种方法的对比与在西方哲学的启蒙时期出现的关于**理性主义**与**经验主义**和**现实主义**与**理想主义**的早期形而上学的辩论有关。这些辩论出现在反对一切知识的来源被认为是神的启示的地方的正统思想的背景下，在十七和十八世纪期间，哲学家认为人类理性或感官经验优先了启示。笛卡尔和莱布尼兹以及其他人采取了理性的立场，声称所有的真理来源于人类思想，从出生起在我们的脑海中就植入的"天赋观念"的存在。例如：他们认为欧几里德几何原理是使用人的理性制定的，而不是超自然的启示或感官体验的结果。相比之下，洛克和其他人采取了经验主义的观点，认为我们的知识的主要来源是我们的感官经验，人类理性在翻译这些经验上起次要作用。这一立场经常引用的证据是伽利略的发现——基于对行星运动的仔细观察——太阳系是以太阳为中心，而不是地球为中心。在语言学的背景下，本次辩论导致以下问题：人类语言经验与我们先天的"语言能力"各自多大程度上作为我们的语言知识的基础？在 NLP 中这个问题表现为在计算模型构建中语料库数据与语言学反省之间的优先级。

还有一个问题，在现实主义和理想主义之间的辩论中被奉若神明的是理论结构的形而上学的地位。康德主张现象与我们可以体验的表现以及不能直接被认识的"事情本身"之间的相互区别。语言现实主义者会认为如名词短语这样的理论建构是一个现实世界的实体，是人类看法和理由的独立存在，它实际导致观测到的语言现象。另一方面，语言理想主义者会说名词短语以及如语义表示这样更抽象的结构本质上无法观察到，只是担任有用的虚构的角色。语言学家写理论的方式往往与现实主义的立场相违背，而 NLP 从业人员占据中立地位，

不然就倾向于理想主义立场。因此，在 NLP 中，如果一个理论的抽象导致一个有用的结果往往就足够了；不管这个结果是否揭示了任何人类语言处理。

这些问题今天仍然存在，表现为符号与统计方法、深层与浅层处理、二元与梯度分类以及科学与工程目标之间的区别。然而，这样的反差现在已经非常细微，辩论不再像从前那样是两极化。事实上，大多数的讨论——大部分的进展——都包含一个“平衡协调”。例如：一种中间立场是假设人类天生被赋予基于类比和记忆的学习方法（弱理性主义），并使用这些方法确定他们的感官语言经验（经验主义）的有意义的模式。

整本书中，我们已经看到了这种方法的很多例子。每次语料统计指导上下文无关文法产生式的选择，统计方法就会给出符号模型，即“文法工程”。每次使用基于规则的方法创建的一个语料被用来作为特征来源训练统计语言模型时，符号方法都会给出统计模型，即“文法推理”。圆圈是封闭的。

NLTK 的路线图

自然语言工具包是在不断发展的，随着人们贡献代码而不断扩大。NLP 和语言学的一些领域（还）没有得到 NLTK 很好的支持，特别欢迎在这些领域的贡献。有关这本书的出版之后的开发新闻，请查阅 http://www.nltk.org/。特别鼓励以下方面的贡献：

音韵学和形态学：

研究声音模式和文字结构的计算方法，通常用一个有限状态机工具包。如不规则词形变化和非拼接形态这样的现象使用我们一直在学习的字符串处理方法很难解决。该技术面临的挑战不仅仅是连接 NLTK 到一个高性能的有限状态机工具包，而且要避免词典数据的重复以及链接形态分析器和语法分析器所需形态学特征。

高性能模块：

一些 NLP 任务的计算量太大，使纯 Python 实现不可行。然而，在某些情况下，耗时只出现在训练模型期间，不是使用它们标注输入期间。NLTK 中的包系统提供了一个方便的方式来发布训练好的模型，即使那些使用不能随意分布的语料库训练的模型。替代方法是开发高性能的机器学习工具的 Python 接口，或通过使用类似与 MapReduce 的并行编程技术扩展 Python 的能力。

词汇语义学：

这是一个充满活力的领域，目前的研究大多围绕词典、本体、多词表达式等的继承模型，大都在现在的 NLTK 的范围之外。一个保守的目标是从丰富的外部存储获得词汇信息，以支持词义消歧、解析和语义解释等任务。

自然语言生成：

从含义的内在表示生产连贯的文本是 NLP 的重要组成部分；用于 NLP 的基于归一的方法已经在 NLTK 中开发，在这一领域做出更大的贡献还有限制。

语言实地调查：

语言学家面临的一个重大挑战是记录数以千计的濒危语言，这项工作产生大量异构且快速变化的数据。更多的实地调查的数据格式，包括行间的文本格式和词汇交换格式，在 NLTK 中得到支持，帮助语言学家维护和分析这些数据，解放他们，使他们能在数据提炼中花费尽可能多的时间。

其他语言：

对英语以外的语言的 NLP 改进支持包括两方面的工作：获准发布更多 NLTK 中的收集的语料库；写特定语言的 HOWTO 文件发布到 http://www.nltk.org/howto，说明 NLTK 中的使用，讨论语言相关的 NLP 问题，包括字符编码、分词、形态。一个特定语言专长的 NLP

研究人员可以安排翻译这本书，并在 NLTK 的网站上保存一个副本；这将不仅仅是翻译讨论的内容，而要使用目标语言的数据提供等效的可行的例子，一项不平凡的事业。

NLTK-Contrib：

许多 NLTK 中的核心组件都由 NLP 社区成员贡献，它们最初被安置在 NLTK 中的“Contrib”包，nltk_contrib。对添加到这个包中的软件的唯一要求是它必须用 Python 编写，与 NLP 有关，并给予与 NLTK 中其他软件一样的开源许可。不完善的软件也是值得欢迎的，随着时间的推移可能会被 NLP 社区的其他成员改进。

教材：

从 NLTK 开发的最初起，教材一直伴随着软件逐渐扩大填补这本书，也加上大量的网上材料。我们希望弄清楚提供这些材料包括：幻灯片、习题集、解答集、我们所覆盖的主题更详细的理解的教员的名字，并通知作者，我们可以为他们在 http://www.nltk.org/上做链接。具有特殊价值的材料，帮助 NLP 成为计算机科学和语言学系的本科主流课程，或者使 NLP 在二级本科课程中可以获得，在那里对语言、文学、计算机科学以及信息技术课程中的计算内容有明显的限制。

只是一个工具包：

在序言中已经指出，NLTK 是一个工具包，而不是一个系统。在 NLTK、Python、其他 Python 库、外部 NLP 的工具和格式的接口集成中会有很多问题需要解决。

Envoi ...

语言学家有时会被问到他们说多少种语言，不得不解释说这一领域实际上关注语言间共享的抽象结构的研究，一种比学说尽可能多的语言更深刻更难以捉摸的研究。同样的，计算机科学家有时会被问到他们懂多少种编程语言，不得不解释说计算机科学实际上关注能在任何编程语言中实施的数据结构和算法的研究，一种比争取学习尽可能多的编程语言更深刻更难以捉摸。

这本书涵盖了自然语言处理领域的许多主题。大多数的例子都使用 Python 和英语。不过，如果读者得出的结论是 NLP 是有关如何编写 Python 程序操纵英文文本，或者更广泛的，关于如何编写程序（以任何一种编程语言）处理（任何一种自然语言）文本的，这将是不幸的。我们选择 Python 和英语是权宜之计，仅此而已。即使我们关注编程本身也只是一种解决问题的手段：作为一种了解表示和操纵语言标注文本的集合的数据结构和算法的方式，作为一种方法来建立新的语言技术，更好地服务于信息社会的需求，并最终作为对人类语言极度丰富性的更深的理解的方法。

但是目前为止，happy hacking!

参考文献

NLTK 索引

一般索引

参考文献、NLTK 索引、一般索引见原书英文版。

关于作者

Steven Bird 是墨尔本大学计算机科学和软件工程系副教授，宾夕法尼亚大学的语言数据联盟高级副研究员。他是 1990 年英国爱丁堡大学计算音韵学博士，导师是 Ewan Klein。后来到喀麦隆开展夏季语言学研究所主持下的 Grassfields 班图语语言实地调查。最近，他作为语言数据联盟副主任花了几年时间领导研发队伍，创建已标注文本的大型数据库的模型和工具。在墨尔本大学，他建立了一个语言技术研究组，并在各级本科计算机科学课程任教。2009 年，史蒂芬成为计算语言学学会主席。

Ewan Klein 是英国爱丁堡大学信息学院语言技术教授。于 1978 年在剑桥大学完成形式语义学博士学位。在苏塞克斯和纽卡斯尔大学工作多年后，参加了在爱丁堡的教学岗位。他于 1993 年参与了爱丁堡语言科技集团的建立，并一直与之密切联系。从 2000 到 2002，他离开大学作为圣克拉拉的埃迪法公司的总部在爱丁堡的自然语言的研究小组的研发经理，负责处理口语对话。伊万是欧洲章计算语言学协会（European Chapter of the Association for Computational Linguistics）前任主席，并且是人类语言技术（ELSNET）欧洲卓越网络的创始成员和协调员。

Edward Loper 最近完成了宾夕法尼亚大学自然语言处理的机器学习博士学位。爱德华是史蒂芬在 2000 年秋季计算语言学研究生课程的学生，也是教师助手和 NLTK 开发的成员。除了 NLTK，他帮助开发了用于记录和测试 Python 软件的两个包：epydoc 和 doctest。

书的末页

《Python 自然语言处理》封面上的动物是露脊鲸，所有大鲸鱼中最稀有的。可以通过它的约占身体总长三分之的巨大的头来识别。它生活在两个半球的大洋表面温带和凉爽的海洋中。露脊鲸的名字被认为是得自捕鲸人，他们认为它“正是”他们要杀死取油的鲸鱼。虽然自从 20 世纪 30 年代以来它就被保护，露脊鲸仍然是所有大鲸中最濒危的。

大而笨重的露脊鲸通过其头部的老茧很容易区别于其他鲸鱼。它有一个广阔无背鳍的背部和很长的从眼睛上面开始的拱嘴。它的身体是黑色的，除了肚皮上的白色补丁。伤口和疤痕可能会呈现明亮的橙色，往往会被鲸虱子或 cyamids 感染。老茧——也在气孔附近出现，眼睛、下巴、上唇的上面——是黑色或灰色。它有大的形状像桨的鳍状肢和独特的 V 形的吹气，由其头部的顶端广泛分布的气孔发出，可以上升到海洋表面 16 英尺以上。

露脊鲸以浮游生物包括像磷虾虾和桡足类为食。作为须鲸，他们有一串从每一侧上颌骨悬挂下来的 225-250 个边缘折叠板，那里应该是牙齿的地方。板是黑色的，可有 7.2 英尺。露脊鲸是“海中食草动物”，经常张着自己的嘴巴慢慢的游泳。随着水流进嘴里通过须，猎物被困在舌头附近。

因为雌性要到 10 岁才能达到性成熟，在一年之久的妊娠后它们生下一个小鲸，所以露脊鲸数量增长缓慢。小露脊鲸会和母亲待一年。

露脊鲸遍布世界各地，但数量很少。一只露脊鲸通常单独或组成 1 至 3 只的小团体活动，但求偶时，它们可能形成 30 只的大组。与大多数须鲸一样，它们随季节性迁徙。在寒冷水域栖息觅食，然后迁移到温暖的水域繁殖和产犊。虽然它们在饲养季节可能游到远海，露脊鲸会在沿海地区的繁殖。有趣的是，许多雌性不会每年都回到这些沿海养殖区，而是只在繁殖年来到该地区。其他年份它们去哪里了这仍是一个谜。

露脊鲸唯一的天敌是逆戟鲸和人类。当危机出现时，一组露脊鲸可能在一起围成一个圈，用尾巴指向外面，以阻止捕食。这种防御并非总是成功的，小鲸偶尔会与它们的母亲分开而被杀害。

露脊鲸是游泳最慢的鲸之一，虽然它们短期爆发可能达到 10 英里每小时。它们可以下潜到至少 1000 英尺，并可以潜水长达 40 分钟。即使经过多年的保护状态，露脊鲸仍然极度濒危。只在过去的 15 年有证据表明它们在南半球的数量有所恢复。仍然不知道露脊鲸是否会在北半球存活。虽然目前没有捕杀，当前保护的问题包括船舶碰撞、捕鱼活动的冲突、栖息地的破坏、石油钻探和与其他鲸类可能的竞争冲突。露脊鲸没有牙齿，所以耳骨，在某些情况下，眼球晶体可以用来估计露脊鲸在死亡时的年龄。相信露脊鲸能活至少 50 年，但是有关它们寿命的数据很少。

封面图片来自多佛尔画报档案。封面字体是 Adobe ITC Garamond。文本的字体是 Linotype Birka；标题字体是 Adobe Myriad Condensed；而代码的字体是 LucasFont 的 TheSans MonoCondensed。