**NLP\_chapter3**

**第三章 加工原料文本**

从这章开始往后运行程序段前默认需要进行的准备工作：

>>> import nltk, re, pprint

>>> from nltk import word\_tokenize

**3.1 从网络和硬盘访问文本**

**电子书**

通过ASCII码文本文件的URL访问网络文本：

注意URL网址一定要输入正确，可以先在浏览器上访问然后再复制，否则会出现404not found，国外网站视情况挂VPN

>>> from urllib import request

>>> url = "http://www.gutenberg.org/files/2554/2554-0.txt"

>>> response = request.urlopen(url)

>>> raw = response.read().decode('utf8')

>>> type(raw)

如果使用的 Internet 代理 Python 不能正确检测出来，可以用下面的方法手动指定代理:

>>> proxies = {'http': 'http://www.someproxy.com:3128'}

>>> request.ProxyHandler(proxies)

通过这种方式获取的内容是以字符串的形式呈现的，对于语言处理，我们要将字符串分解为词和标点符号，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 = word\_tokenize(raw)

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。如果现在采取进一步的步骤从这个链表创建一个 NLTK 文本，则可以进行在第 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：

>>> text = nltk.Text(tokens)

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("PARTI")

>>> raw.rfind("End of Project Gutenberg's Crime")

>>> raw = raw[5303:1157681]

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。我们用这个切片重新给raw赋值，所以现在raw以“PARTI”开始一直到(但不包括)标记内容结尾的句子。

**处理HTML**

第一步是像以前一样使用 urlopen:

>>> url = "http://news.bbc.co.uk/2/hi/health/2284783.stm"

>>> html = request.urlopen(url).read().decode('utf8')

通过输入 print(html) 可以看到 HTML 的全部内容

为了提取文本，需要用到python库BeautifulSoup。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:

>>> from bs4 import BeautifulSoup

>>> raw = BeautifulSoup(html).get\_text()

>>> tokens = word\_tokenize(raw)

其中仍含有不需要的内容，根据需要做进一步处理

**处理搜索引擎的结果**

优势：规模庞大；可以用非常具体的模式匹配成千上万的例子

缺陷：搜索方式的范围受到严格限制；给出的结果受时间和位置影响大，容易不一致；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)

**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库的帮助下，可以访问一个博客的内容:

>>> import feedparser

>>> llog = feedparser.parse("http://languagelog.ldc.upenn.edu/nll/?feed=atom")

>>> llog['feed']['title']

>>> len(llog.entries)

>>> post = llog.entries[2]

>>> post.title #注意这里的方法是不加()的

>>> content = post.content[0].value

>>> content[:70]

>>> word\_tokenize(BeautifulSoup(content).get\_text())

**读取本地文件**

使用 Python 内置的 open()函数，然后是 read()方法:

>>> f = open('document.txt')

>>> f.read()

在python中检查当前目录：

>>> import os

>>> os.listdir('.')

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。

read()方法创建了一个包含整个文件内容的字符串

也可以使用一个 for 循环一次读文件中的一行:

>>> f = open('document.txt','rU') #一定要以通用方式打开

>>> for line in f:

print(line.strip())

strip()方法的作用是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。只需使用 nltk.data.find()来获取语料库项目的文件名。然后就可以使用刚才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**

第三方函数库如 pypdf 和 pywin32 提供了对这些格式的访问。

**捕获用户输入**

若想捕捉用户与程序交互时输入的文本，调用 Python 函数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 = input('Enter some text:')

>>> print('You typed',len(word\_tokenize(s)),'words')

**3.2 字符串:最底层的文本处理**

Print()函数和直接输入变量名称来检查的输出的区别：

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（有引号）。然而，当我们告诉解释器输出这个变量时（print()），我们没有看到引号字符，因为字符串的内容里面没有引号。

有用的字符串方法:所有的方法都**产生一个新的字符串或链表**

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(不输入t,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.title() 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

**字符串和链表之间不能连接**

链表可以改变其中的元素，字符串不可以：

>>> del list[-1]

链表支持修改原始值的操作，而不是产生一个新的值。

**3.3 使用 Unicode 进行文字处理**

**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。

翻译成 Unicode 叫做**解码**

将 Unicode 转化为其它编码的过程叫做**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**

使用nltk.data.find()函数定位文件：

>>> path = nltk.data.find('corpora/unicode\_samples/polish-lat2.txt')

Python 的open（）函数提供了将编码数据读入为 Unicode 字符串和将 Unicode 字符串以 编码形式写出的功能。open()函数有一个 encoding 参数来指定被读取或写入的文件的编码:

>>> f = open(path,encoding='latin2')

>>> for line in f:

line = line.strip()

print(line)

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。

>>> f = open(path,encoding='latin2')

>>> for line in f:

line = line.strip()

print(line.encode('unicode\_escape'))

在python3中，默认使用UTF-8作为源码的编码方式，可以使用 ord()查找一个字符的整数序数:

>>> ord('ń')

用适当的转义序列定义一个字符串：

>>> nacute = '\u0144'

>>> nacute

'ń'

查看某个字符在一个文本文件中的字节序列的表示方式：

>>> nacute.encode('utf8')

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 = open(path,encoding='latin2').readlines()

>>> line = lines[2]

>>> print(line.encode('unicode\_escape'))

b'Niemc\\xf3w pod koniec II wojny \\u015bwiatowej na Dolny \\u015al\\u0105sk, zosta\\u0142y\\n'

>>> for c in line:

if ord(c) > 127:

print('{} U+{:04x} {}'.format(c.encode('utf8'),ord(c),unicodedata.name(c)))

如果把上面的c.encode('utf8')替换为c，那么系统将会输出波兰语的字形

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

'Niemców pod koniec II wojny światowej na Dolny Śląsk, zostały\n'

>>> line.find('zosta\u0142y')

54

>>> line = line.lower()

>>> line

'niemców pod koniec ii wojny światowej na dolny śląsk, zostały\n'

>>> line.encode('unicode\_escape')

b'niemc\\xf3w pod koniec ii wojny \\u015bwiatowej na dolny \\u015bl\\u0105sk, zosta\\u0142y\\n'

>>> import re

>>> m = re.search('\u015b\w\*',line)

>>> m.group()

'światowej'

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

>>> word\_tokenize(line)

['niemców', 'pod', 'koniec', 'ii', 'wojny', 'światowej', 'na', 'dolny', 'śląsk', ',', 'zostały']

**3.4 使用正则表达式检测词组搭配**

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**

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)]

**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)]

^用来匹配单词的开头

如果不使用^,$这两个符号，那么正则表达式会找出所有包含正则表达式第一个参数字符串的词汇

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。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**。查找哪些词汇由相同的序列产生：

>>> [w for w in wordlist if re.search('^[ghi][mno][jkl][def]$',w)]

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:«[mn

o]»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][l jk][fed]$»匹配同样的词汇。

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?

"-”：按ASCII码的顺序以“-”左侧字符开头，右侧字符结尾的所有字符

"+”:表示前面的项目的一个或多个实例

“\*”表示前面的项目的零个或多个实例。

“+”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:«[^aei ouAEIOU]»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。注意其中包含非字母字符。

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。{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。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。

若省略圆括号，系统会自动判定操作符的范围：

>>> [w for w in wsj if re.search('(ed|ing)$',w)][:50]

>>> [w for w in wsj if re.search('ed|ing$',w)][:50]

上面第二个例子中的正则表达式«ed|ing$»相当于«(ed)|(ing$)»

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。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：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。

**3.5 正则表达式的有益应用**

**提取字符块**

通过re.findall() 方法找出所有(无重叠的)匹配指定正则表达式的字符：

>>> word = 'supercalifragilisticexpialidocious'

>>> re.findall(r'[aeiou]',word)

找出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.most\_common(12)

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(r'[0-9]{2,4}','2009-12-31')]

**在字符块上做更多事情**

下面这个例子中，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:

>>> 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]))

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()

寻找包含一个给定的辅音-元音对的单词的列表，例如:cv\_index['su']应该给出所有含有“su ” 的词汇：

>>> 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']

**查找词干**

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

使用正则表达式:

只提供后缀：因为括号有第二个功能:选择要提取的子字符串。

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

添加“?:”， 使用括号来指定连接的范围，但不选择要输出的字符串：

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

将词分成词干和后缀，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')

“\*”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，例如：'processes’会输出'processe'和's'。如果想使用“非贪婪”版本的“\*”操作符，写成“\*?”:

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

可以通过使第二个括号中的内容变成可选，来得到空后缀（当检测的词为单数时）:

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

定义一个函数来获取词干：

>>> def stem(word):

regexp = r'^(.\*?)(ing|ly|ed|ious|ies|ive|es|s|ment)?$'

stem,suffix = re.findall(regexp,word)[0]

return stem

注意这里[0]的使用，re.findall方法使用之后会获得包括一个在()内有两个字符串的字符串的列表,这时候需要使用[0]来进行切片，之后函数会将()内的内容进行拆分，否则会报错

这里的正则表达式因为其无差别删除作用，会产生一些非词(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**

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)。

“<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<.\*>，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)：

>>> from nltk.corpus import gutenberg,nps\_chat

>>> moby = nltk.Text(gutenberg.words('melville-moby\_dick.txt'))

>>> moby.findall(r'<a>(<.\*>)<man>')

找出以词“br o”结尾的三个词组成的短语:

>>> chat = nltk.Text(nps\_chat.words())

>>> chat.findall(r'<.\*><.\*><bro>')

找出以字母“l”开始的三个或更多词组成的序列:

>>> chat.findall(r'<l.\*>{3,}')

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:

>>> from nltk.corpus import brown

>>> hobbies\_learned = nltk.Text(brown.words(categories=['hobbies', 'learned']))

>>> hobbies\_learned.findall(r"<\w\*> <and> <other> <\w\*s>")

这里的\w的作用是去掉那些以标点符号开头的短语，如“ , and other microorganisms”

**3.6 规范化文本**

定义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，相对于自己制作的提取器，它们能处理的不规则的情况很广泛，如Porter 和 Lancaster 词干提取器：

>>> porter = nltk.PorterStemmer()

>>> lancaster = nltk.LancasterStemmer()

>>> [porter.stem(t) for t in tokens]

>>> [lancaster.stem(t) for t in tokens]

**词形归并**

即确保结果形式是字典中确定的词。

WordNet词形归并器删除词缀产生的词都是在它的字典中的词：

>>> wnl = nltk.WordNetLemmatizer()

>>> [wnl.lemmatize(t) for t in tokens]

**3.7 用正则表达式为文本分词**

**分词的简单方法**

在空格符处分割文本

原始文本：

>>> 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,'...'''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:

>>> re.split(r' ',raw)

>>> re.split(r'[ \t\n]+',raw)

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)。

为了避免出现如“(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)

可以看到，在开始和结尾都给了我们一个空字符串。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。

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)

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。再添加一个模式«[-.(]+»来匹配引号字符让它们与它们包括的文字分开，即使双连字符、省略号和左括号被单独分词：

>>> re.findall(r"\w+(?:[-']\w+)\*|'|[-.(]+|\S\w\*",raw)

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。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(pattern表达式内的而不是文本的)：

>>> nltk.regexp\_tokenize(text, pattern) #注意text和正则表达式的顺序跟re.findall相反

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。regex

p\_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。

**3.8 分割**

分词是一个更普遍的分割问题的一个实例。

**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。一些语料库已经提供在句子级别的访问。

NLTK通过包含 Punkt 句子分割器(Kiss & Strunk, 2006)简化了文本的分割步骤。这里是使用 它为一篇小说文本断句的例子：

>>> 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])

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。

**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。

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。

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:

>>> 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。

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。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。

最后一步是寻找最大化目标函数值的 0 和 1 的模式

**3.9 格式化:从链表到字符串**

**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。

' '.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。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**

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。直接输入变量名称的方法——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。

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。

避免不必要的空格符问题和输出包含变量和常量交替出现的表达式，使用**字符串格式化表达式**

即类似于'%s->%d;' % (word, fdist[word])的表达式

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**。

在python3中，转换说明符可以用{}来代替

例如：'{}->{};'.format(word, fdist[word])

**排列**

通过在大括号内使用:来指定字符的宽度，纯数字默认向右对齐，其他字符默认向左对齐，可以通过在:后添加<指定向左对齐，或添加>指定向右对齐：

>>> '{:6}'.format(41)

>>> '{:<6}'.format(41)

>>> '{:6}'.format('dog')

>>> '{:>6}'.format('dog')

也可以控制浮点数小数点后的位数：

>>> import math

>>> '{:.4f}'.format(math.pi)

Python3中如果输入%会自动转化为百分数，不需要乘以100：

>>> count,total=3205,9375

>>> 'accuracy for {} words: {:.4%}'.format(total,count/total)

可以使用width变量来指定一个字段的宽度：

>>> '{:{width}}'.format('Monty Python',width=15)

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。

**将结果写入文件**

>>> output\_file = open('output.txt','w')

>>> words = set(nltk.corpus.genesis.words('english-kjv.txt'))

>>> for word in sorted(words):

print(word,file=output\_file)

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:

>>> print(str(len(words)),file=output\_file)

**文本换行**

可以在 Python 的 textwrap 模块的帮助下采取换行:

>>> saying = ['After', 'all', 'is', 'said', 'and', 'done', ',','more', 'is', 'said', 'than', 'done', '.']

>>> 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),