Оглавление

[3 Обработка сырого текста 4](#_Toc514187735)

[3.1 Доступ к тексту из Web и на диске 5](#_Toc514187736)

[3.1.1 Электронные книги 5](#_Toc514187737)

[3.1.2 Обработка HTML 9](#_Toc514187738)

[3.1.3 Обработка результатов поисковых систем 11](#_Toc514187739)

[3.1.4 Обработка новостных потоков (RSS Feeds) 13](#_Toc514187740)

[3.1.5 Чтение локальных файлов 14](#_Toc514187741)

[3.1.6 Извлечение текста из PDF, MSWord и других двоичных форматов 17](#_Toc514187742)

[3.1.7 Перехват пользовательского ввода 18](#_Toc514187743)

[3.1.8 Трубопровод NLP 18](#_Toc514187744)

[3.2 Строки: Обработка текста на низком уровне 21](#_Toc514187745)

[3.2.1 Базовые операции со строками 22](#_Toc514187746)

[3.2.2 Печать строк 25](#_Toc514187747)

[3.2.3 Доступ к отдельным символам 26](#_Toc514187748)

[3.2.4 Доступ к подстрокам 29](#_Toc514187749)

[3.2.5 Другие операции над строками 31](#_Toc514187750)

[3.2.6 Различие между списками и строками 32](#_Toc514187751)

[3.3 Обработка текстов в кодировке Unicode 35](#_Toc514187752)

[3.3.1 Что такое Unicode? 35](#_Toc514187753)

[3.3.2 Извлечение кодированного текста из файлов 37](#_Toc514187754)

[3.3.3 Использование локализованной кодировки в Python 42](#_Toc514187755)

[3.4 Регулярные выражения для выявления шаблонов слов 44](#_Toc514187756)

[3.4.1 Использование мета-символов 45](#_Toc514187757)

[3.4.2 Диапазоны и замыкания 46](#_Toc514187758)

[3.5 Полезные применения регулярных выражений 51](#_Toc514187759)

[3.5.1 Извлечение фрагментов слов 52](#_Toc514187760)

[3.5.2 Больше действий с фрагментами слов 53](#_Toc514187761)

[3.5.3 Нахождение основы слова (стэмминг) 56](#_Toc514187762)

[3.5.4 Поиск токенизированного текста 59](#_Toc514187763)

[3.6 Нормализация текста 62](#_Toc514187764)

[3.6.1 Стеммеры 63](#_Toc514187765)

[3.6.2 Лемматизация 66](#_Toc514187766)

[3.7 Регулярные выражения для токенизации текста 67](#_Toc514187767)

[3.7.1 Простые подходы к токенизации 67](#_Toc514187768)

[3.7.2 NLTK's Regular Expression Tokenizer 72](#_Toc514187769)

[3.7.3 Дополнительные аспекты токенизации 73](#_Toc514187770)

[3.8 Сегментация 74](#_Toc514187771)

[3.8.1 Сегментация предложений 74](#_Toc514187772)

[3.8.2 Word Segmentation 76](#_Toc514187773)

[3.9 Форматирование: От списков к строкам 83](#_Toc514187774)

[3.9.1 От списков к строкам 84](#_Toc514187775)

[3.9.2 Строки и форматы 85](#_Toc514187776)

[3.9.3 Выравнивание информации 88](#_Toc514187777)

[3.9.4 Запись результатов в файл 92](#_Toc514187778)

[3.9.5 Перенос текста 93](#_Toc514187779)

[3.10 Summary 94](#_Toc514187780)

[3.11 Further Reading 96](#_Toc514187781)

[3.12 Упражнения 97](#_Toc514187782)

# 3 Обработка сырого текста

Наиболее важным источником текстов, несомненно, является Web. Удобно иметь готовую коллекцию текстов для исследований, вроде корпусов, которые мы рассматривали в предыдущих главах. Однако у Вас могут быть собственные тексты, к которым хотелось бы иметь доступ.

Целью данной главы являются ответы на следующие вопросы:

* как писать программы для доступа к текстам из локальной файловой системы или из Web, чтобы получить неограниченный объем языкового материала;
* как разделять документы на отдельные слова и знаки пунктуации, чтобы выполнять те же виды анализа, которые доступны для корпусов текстов;
* как писать программы, порождающие форматированный вывод и сохранять его в файле.

Для этого мы познакомимся с ключевыми концепциями NLP, включая **токенизацию** и **стэмминг**. Мы усовершенствуем навыки использования Python и узнаем о строках, файлах и **регулярных выражениях**. Поскольку Web содержит много текста, мы выясним, как иметь дело с гипертекстом.

**Замечание**: Начиная с этого места, подразумевается, что Вы начинаете каждую свою программу со следующих директив импорта:

>>> import nltk, re, pprint

>>> from nltk import word\_tokenize

## 3.1 Доступ к тексту из Web и на диске

### 3.1.1 Электронные книги

Мы рассматривали маленький пример текстов из Project Gutenberg, входящий в коллекцию корпусов NLTK. Однако Вам может понадобиться анализировать и другие тексты из Project Gutenberg. По адресу http://www.gutenberg.org/catalog/ можно просмотреть список из 25000 (уже 65941!) свободно распространяемых электронных книг и получить URL для загрузки текстового файла в кодировке ASCII. Хотя 90% текстов Project Gutenberg на английском, он также включает материалы на 50 других языках, включая Catalan, Chinese, Dutch, Finnish, French, German, Italian, Portuguese и Spanish (для каждого языка более 100 текстов).

Текст номер 2554 – это английский перевод «*Преступление и наказание»* – его можно получить следующим образом.

>>> 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)

<class 'str'>

>>> len(raw)

1176893

>>> raw[:75]

'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'

**Замечание.** Процессчтенияread() займет несколько секунд, поскольку загружается большая книга. Если Вы используете прокси-сервер, которые Python не распознает корректно, потребуется указать адрес прокси вручную перед использованием urlopen, как в следующем фрагменте:

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

>>> request.ProxyHandler(proxies)

Переменная raw содержит строку из 1,176,893 символа. (Это можно определить с помощью type(raw).) Это «сырое» линейное содержимое книги в виде строки символов, включая детали, которые нас не интересуют, такие как пробелы, переходы на новую строку, пустые строки. Заметим, что комбинация спецсимволов \r\n в Windows используется для перехода на новую строку. Для наших задач обработки текстов нужно разделять строку на слова и знаки пунктуации, как показано в [1.](http://www.nltk.org/book/ch01.html" \l "chap-introduction) Этот шаг называется **токенизацией**. Он порождает знакомую структуру в виде списка слов и знаков пунктуации.

>>> tokens = **nltk.word\_tokenize**(raw)

>>> type(tokens)

<class 'list'>

>>> len(tokens)

254354

>>> tokens[:10]

['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime', 'and', 'Punishment', ',', 'by']

Заметим, что NLTK был нужен для токенизации, но не для открытия URL и чтения из него строки. Если мы теперь проделаем стандартную процедуру создания объекта NLTK Text из множества токенов tokens (конструктор класса Text), то сможем выполнять всю лингвистическую обработку, так же как и для корпусов text1,…,text9 в [1](http://www.nltk.org/book/ch01.html" \l "chap-introduction), наряду с обычными списковыми операциями, например slicing (выделение фрагмента), определение коллокаций и т.д.:

>>> text = nltk.Text(tokens)

>>> type(text)

<class 'nltk.text.Text'>

>>> text[1024:1062]

['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; Pyotr Petrovitch; Pulcheria Alexandrovna; Avdotya

Romanovna; Rodion Romanovitch; Marfa Petrovna; Sofya Semyonovna; old

woman; Project Gutenberg-tm; Porfiry Petrovitch; Amalia Ivanovna;

great deal; Nikodim Fomitch; young man; Ilya Petrovitch; n't know;

Project Gutenberg; Dmitri Prokofitch; Andrey Semyonovitch; Hay Market

Заметим, что *Project Gutenberg* сам фигурирует в списке коллокаций. Дело в том, что каждый текст, загруженный из Project Gutenberg содержит заголовок с названием текста, автором, именами людей, которые сканировали и корректировали текст, лицензией и т.д. Иногда эта информация указывается в сноске в конце файла. Поэтому мы не можем надежно определить начало и конец содержимого текста. Приходится прибегать к ручному анализу файла, чтобы обнаружить уникальные строки, которые отмечают начало и конец текста. После этого можно «обрезать» текст в переменной raw, чтобы сохранилось только содержимое:

>>> raw.find("PART I")

5338

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

1157743

>>> raw = raw[5338:1157743]

>>> raw.find("PART I")

0

Методы find() и rfind() ("reverse find") позволяют находить индексы в строке. Мы заменяем содержимое raw его частью: теперь строка начинается с "PART I" и продолжается до фразы, отмечающей конец содержимого (не включая ее).

Это было первое «столкновение» с реальностью текстов в Web: эти тексты могут содержать нежелательное содержимое. При этом автоматический способ удаления этого содержимого неочевиден. Однако некоторые дополнительные усилия могут позволить удалить ненужный материал.

**(3.1.1\_1) Самостоятельно:** Фактически raw.rfind("End of Project Gutenberg's Crime") не работает (возвращает -1). Исправьте ошибку.

**(3.1.1\_2) Самостоятельно:** Определите три самых часто встречающихся слова в английском переводе «Ревизора».

**(3.1.1\_3)** **Самостоятельно:** Самостоятельно проведите небольшое исследование корпуса Гутенберга.

### 3.1.2 Обработка HTML

Большая часть текстов в Web имеют форму HTML-документов. С помощью Web-браузера можно сохранять тексты в локальных файлах, а затем работать с их содержимым с помощью описанных далее приемов. Однако если такую процедуру приходится осуществлять часто, удобнее автоматизировать ее с помощью Python. Первый шаг – такой же, как и ранее – использование функции urlopen. В качестве развлечения мы выбрали заметку из BBC News «*Blondes to die out in 200 years»* - научно-популярная статья о прогнозах исчезновения блондинок:

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

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

>>> html[:60]

'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN'

С помощью print(html) можно увидеть все HTML-содержимое, включая теги, изображения, JavaScript, формы и таблицы.

**(3.1.2\_1) Самостоятельно:** Определите, содержит ли некоторая web-страница таблицу, построенную с помощью тега <TABLE>.

Для извлечения из HTML только текста можно использовать Python-библиотеку BeautifulSoup, доступную по адресу http://www.crummy.com/software/BeautifulSoup/:

>>> from bs4 import BeautifulSoup

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

>>> tokens = word\_tokenize(raw)

>>> tokens

['BBC', 'NEWS', '|', 'Health', '|', 'Blondes', "'to", 'die', 'out', ...]

Текст по-прежнему содержит ненужный материал, имеющий отношение к навигации по сайту и связанным темам. Путем проб и ошибок можно найти начальный и конечный индекс содержимого и выбрать интересующие нас токены и принициализировать объект-текст.

>>> tokens = tokens[110:390]

>>> text = nltk.Text(tokens)

>>> text.concordance('gene')

Displaying 5 of 5 matches:

hey say too few people now carry the gene for blondes to last beyond the next

blonde hair is caused by a recessive gene . In order for a child to have blond

have blonde hair , it must have the gene on both sides of the family in the g

ere is a disadvantage of having that gene or by chance . They do n't disappear

des would disappear is if having the gene was a disadvantage and I do not thin

**(3.1.2\_2) Самостоятельно:** Почему повторные запуски последнего скрипта дают разные результаты?

**(3.1.2\_3) Самостоятельно:** Постарайтесь избавиться от «мусора» в содержимом web-документа, удалив из него специальные несодержательные токены.

**(3.1.2\_4) Самостоятельно:** Расскажите еще о 3-х полезных возможностях BeautifulSoup .

### 3.1.3 Обработка результатов поисковых систем

Web можно воспринимать как огромный корпус неаннотированного текста. Поисковые системы Web обеспечивают эффективные средства поиска на основании соответствия лингвистическим примерам. Основное преимущество поисковых систем это размер пространства поиска. Вследствие этого результатом поиска будет большое множество документов, каким-то образом соответствующих лингвистическому шаблону. Второе преимущество – простота использования.

***Таблица 3.1****:* Примеры коллокаций в Google: Количество попаданий в коллокацию для слов *absolutely* и *definitely*, с последующими *adore*, *love*, *like* или *prefer*. (Liberman, in *LanguageLog*, 2005).

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

**(3.1.3\_1) Самостоятельно:** Дайте содержательное объяснение результатам в Табл.3.1.

К сожалению, поисковые системы имеют существенные недостатки. Во-первых, допустимый диапазон поисковых запросов строго ограничен. В отличие от локальных корпусов, для которых можно писать программы поиска произвольно сложных шаблонов, поисковые системы позволяют формировать запрос в виде конкретных слов или нескольких слов иногда с использованием элементов регулярных выражений. Во-вторых, поисковые системы выдают недостаточно объективные результаты. При этом результаты могут сильно отличаться от времени и места осуществления запроса. Когда некоторое содержимое дублируется на нескольких сайтах, результаты поиска могут быть искажены. Наконец, разметка в результате поиска может измениться непредсказуемым образом, нарушая требования поиска, ориентированного на шаблоны (эту проблему можно уменьшить за счет использования API поисковой системы).

**(3.1.3\_2) Самостоятельно:** Проведите поиск в Web по запросу "the of" (в кавычках). Можно ли, исходя из большого количества элементов в результатах сделать вывод о частой коллокации *the of* в английском языке? Подтвердите Ваши соображения на некотором тексте из Web.

### 3.1.4 Обработка новостных потоков (RSS Feeds)

Блогосфера – важный источник текстов формального и неформального характера. С помощью Python-библиотеки *Universal Feed Parser*, доступной по адресу https://pypi.python.org/pypi/feedparser, мы можем получать доступ к содержимому блога следующим способом:

>>> import feedparser

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

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

'Language Log'

>>> len(llog.entries)

15

>>> post = llog.entries[2]

>>> post.title

"He's My BF"

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

>>> content[:70]

'<p>Today I was chatting with three of our visiting graduate students f'

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

>>> word\_tokenize(raw)

['Today', 'I', 'was', 'chatting', 'with', 'three', 'of', 'our', 'visiting',

'graduate', 'students', 'from', 'the', 'PRC', '.', 'Thinking', 'that', 'I',

'was', 'being', 'au', 'courant', ',', 'I', 'mentioned', 'the', 'expression',

'DUI4XIANG4', '\u5c0d\u8c61', '("', 'boy', '/', 'girl', 'friend', '"', ...]

Выполнив немного дополнительной работы, мы можем написать программу для создания небольшого корпуса блог-постов и использовать его как ресурс для решения задач NLP.

### 3.1.5 Чтение локальных файлов

Для чтения локального файла можно использовать встроенную в Python функцию open() и метод read(). Допустим, у нас имеется файл document.txt. Его содержимое можно загрузить следующим образом:

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

>>> raw = f.read()

**~~(3.1.5\_1) Самостоятельно:~~** ~~Создайте файл document.txt, используя текстовый редактор, введите в него несколько строк и сохраните его в виде простого текста. Далее в интерпретаторе Python откройте файл с помощью f = open('document.txt'), а затем ознакомьтесь с его содержимым с помощью print(f.read()).~~

При выполнении этих действий могут проявиться различные особенности. Если интерпретатор не найдет файл, Вы увидите следующую ошибку:

>>> 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'

Проверить наличие открываемого файла в текущем каталоге с помощью Python можно следующим образом:

>>> import os

>>> os.listdir('.')

Другая возможная проблема, с которой Вы можете столкнуться при работе с файлом – соглашения о способе завершения строки, которые могут различаться в разных операционных системах. Встроенная функция open() имеет второй параметр для управления способом открытия файла: open('document.txt', 'rU') - 'r' означает, что файл открывается для чтения, а 'U' – обозначает "Universal", что позволяет нам игнорировать различные правила завершения строк.

После успешного открытия файла чтение его содержимого можно выполнить несколькими способами. Метод read() создает строку, содержащую текст всего файла:

>>> f.read()

'Time flies like an arrow.\nFruit flies like a banana.\n'

Напомним, что символ '\n' является признаком завершения строки, который образуется при нажатии клавиши *Enter* при переходе на новую строку.

Также мы можем считывать содержимое файла построчно, используя цикл for:

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

>>> for line in f:

... print(line.strip())

Time flies like an arrow.

Fruit flies like a banana.

Здесь мы используем метод strip() для удаления символа перехода на новую строку в конце.

**(3.1.5\_1) Самостоятельно:** Создайте файл poem.txt, содержимым которого будет являться некоторое стихотворение. Далее в Python прочитайте этот файл и выведите все его содержимое на экран так, чтобы в каждой строке на экране было по 50 символов из стихотворения.

Файлы с корпусами NLTK также можно читать с помощью этих методов. Метод nltk.data.find() можно использовать для получения имени каждого элемента корпуса. Затем можно открывать и читать содержимое файла ранее описанным способом:

>>> path = nltk.data.find('corpora/gutenberg/melville-moby\_dick.txt')

>>> raw = open(path, 'r').read()

>>> print(raw([:1000])

**(3.1.5\_2) Самостоятельно**: Почему мы не видим начала текста Моби Дика? Как исправить?

### 3.1.6 Извлечение текста из PDF, MSWord и других двоичных форматов

Текст ASCII и HTML предназначены для чтения человеком. Другие виды файлов используют двоичные форматы (PDF, MSWord), которые можно открывать только с помощью специального программного обеспечения. Однако имеются библиотеки сторонних производителей – PyPDF2 и pywin32 – предоставляющие доступ к этим форматам. Особенно сложно извлекать текст из документов со многими колонками. Для одноразового преобразования небольшого количества документов проще открывать приложение с помощью соответствующего специализированного ПО, а затем сохранять его в текстовом формате в локальном файле. Если документ уже размещен в Web, его URL-адрес можно ввести в строке запроса в Google. Результат поиска часто содержит ссылку к HTML-версии документа, который Вы можете сохранить как текст.

### 3.1.7 Перехват пользовательского ввода

Иногда нужно получить текст, который пользователь вводит, когда взаимодействует с Вашей программой. Для того, чтобы пользователь получил приглашения для ввода текста используется функция Python input(). После сохранения введенной информации в переменной, с ней можно работать как и с другими строками.

>>> s = input("Enter some text: ")

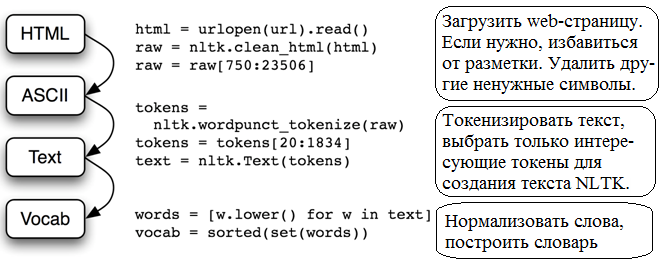
Enter some text: On an exceptionally hot evening early in July

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

You typed 8 words.

### 3.1.8 Трубопровод NLP

Рис. 3.1 суммирует приемы, рассмотренные в этой главе ранее, включая процесс построения словаря, рассмотренный в [1.](http://www.nltk.org/book/ch01.html" \l "chap-introduction) (Еще один шаг – нормализация – будет рассмотрен в [3.6](http://www.nltk.org/book/ch03.html" \l "sec-normalizing-text).)



**Рисунок 3.1**: Трубопровод NLP: Открыть URL, прочитать его HTML-содержимое, удалить разметку, выбрать подстроку, выполнить токенизацию и опционально преобразовать строку в объект nltk.Text. Кроме того можно перевести все слова в нижний регистр и извлечь из текста словарь.

Здесь происходит много процессов. Для понимания происходящего нужно хорошо представлять тип каждой используемой переменной. Тип любого объекта Python-объекта x можно узнать с помощью type(x), например, type(1) выдаст <int>, поскольку – целое число.

Когда мы загружаем содержимое URL или файла, а затем удаляем HTML-разметку, мы имеем дело со строками – типом Python <str>. (О строках подробнее в [3.2](http://www.nltk.org/book/ch03.html" \l "sec-strings)):

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

>>> type(raw)

<class 'str'>

Токенизация строки приводит к появлению списка (слов), имеющего тип <list>. Нормализация и сортировка списков порождает другие списки:

>>> tokens = word\_tokenize(raw)

>>> type(tokens)

<class 'list'>

>>> words = [w.lower() for w in tokens]

>>> type(words)

<class 'list'>

>>> vocab = sorted(set(words))

>>> type(vocab)

<class 'list'>

Тип объекта определяет операции, которые с ним можно выполнять. Например, операция добавления допустима со списками, но не со строками:

>>> vocab.append('blog')

>>> raw.append('blog')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'str' object has no attribute 'append'

Аналогично, мы можем конкатенировать строки со строками, а списки со списками, но нельзя конкатенировать строки со списками:

>>> 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

## 3.2 Строки: Обработка текста на низком уровне

Теперь наступил момент рассмотреть фундаментальный строковый тип данных, который до сих рассматривался только вскользь. В последних разделах мы фокусировались на тексте как на списке слов. Мы не обращали внимания на слова и способ их обработки в языке программирования. Использование интерфейс обработки корпусов из NLTK, мы можем не замечать файлы, в которых содержатся эти тексты. Содержимое слова или файла представлено фундаментальным типом данных в языке программирования, называемых **строковым типом**.

### 3.2.1 Базовые операции со строками

Строки определяются с помощью одинарных или двойных кавычек. Если внутри строки имеется одиночная или двойная кавычка, ее нужно снабдить символом \. Другой способ одинарная кавычка – окружается двойными или наоборот. В противном случае интерпретатор Python выявит ошибку:

>>> monty = 'Monty Python'

>>> monty

'Monty Python'

>>> circus = "Monty Python's Flying Circus"

>>> circus

"Monty Python's Flying Circus"

>>> circus = 'Monty Python\'s Flying Circus'

>>> circus

"Monty Python's Flying Circus"

>>> circus = 'Monty Python's Flying Circus'

File "<stdin>", line 1

circus = 'Monty Python's Flying Circus'

^

SyntaxError: invalid syntax

Иногда строки удобно располагать на нескольких строках экрана. Для этого у Python имеется несколько способов. В следующем примере для этого используется \ для объединения нескольких строк.

>>> 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:

К сожалению, эти методы не сохраняют перехода между строками сонета. Для этого можно использовать тройные кавычки:

>>> 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:

Теперь, умея задавать в тексте строки, испытаем простые операции с ними. Сначала рассмотрм операцию +, известную как **конкатенация**. Она создает новую строку, которая содержит две строки, соединенные в одну. Заметим, что конкатенация не выполняет никаких дополнительных действий, например вставки пробелов между словами. Как ни удивительно, в Python строки даже можно умножать:

>>> 'very' + 'very' + 'very'

'veryveryvery'

>>> 'very' \* 3

'veryveryvery'

**Самостоятельно 3.2.1\_1:** Выполните последующий код, а перед этим спрогнозируйте результат, для проверки вашего понимания операций + и \*. Следует понимать разницу между строкой ' ' (одиночный пробел) и '' (пустая строка).

>>> 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(line)

**Самостоятельно 3.2.1\_2:** Придумайте свой, столь же «удивительный» пример.

Мы видели, что сложение и умножение применяется в этих случаях к строкам, а не к числам. Однако операцию вычитания к строкам применять нельзя:

>>> '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'

Эти сообщения об ошибках – другой пример того, как Python сообщает о путанице в типах данных.

### 3.2.2 Печать строк

Чтобы увидеть содержимое строковой переменной, следует использовать функцию print:

>>> print(monty)

Monty Python

Заметим, что при выводе кавычки не показываются.

Функция print позволяет нам показывать более, чем одну строку в строке консоли различными способами:

>>> 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

### 3.2.3 Доступ к отдельным символам

Как было показано в [2](http://www.nltk.org/book/ch01.html" \l "sec-a-closer-look-at-python-texts-as-lists-of-words) для списков, строки индексируются, начиная с 0. Когда мы индексируем строку, то получаем доступ к ее отдельным символам. В одиночном символе нет ничего особенного – это просто строки длиной 1.

>>> monty[0]

'M'

>>> monty[3]

't'

>>> monty[5]

' '

Так же как и в случае списков, при попытке использовать индекс, выходящий за границу строки, получаем ошибку:

>>> monty[20]

Traceback (most recent call last):

File "<stdin>", line 1, in ?

IndexError: string index out of range

Также, по аналогии со строками, мы можем использовать для строк отрицательные индексы, где -1 – это индекс последнего символа. Положительные и отрицательные индексы дают нам два способа обращаться к любой позиции строки. В этом случае, когда строка имеет длину 12, с помощью индекса 5 и индекса -7 получаем доступ к тому же самому символу (пробелу). (Заметим, что 5 = len(monty) - 7.)

>>> monty[-1]

'n'

>>> monty[5]

' '

>>> monty[-7]

' '

Мы можем использовать циклы for для итерирования символов в строке. Функция print имеет дополнительный параметр end=' ', которым мы сообщаем Python выводить пробел вместо перехода на новую строку.

>>> sent = 'colorless green ideas sleep furiously'

>>> for char in sent:

... print(char, end=' ')

...

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

Также мы можем подсчитывать отдельные символы. При этом можно игнорировать различия в регистре, переводя символы в нижний регистр. Также можно фильтровать неалфавитные символы:

>>> 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.most\_common(5)

[('e', 117092), ('t', 87996), ('a', 77916), ('o', 69326), ('n', 65617)]

>>> [char for (char, count) in fdist.most\_common()]

['e', 't', 'a', 'o', 'n', 'i', 's', 'h', 'r', 'l', 'd', 'u', 'm', 'c', 'w',

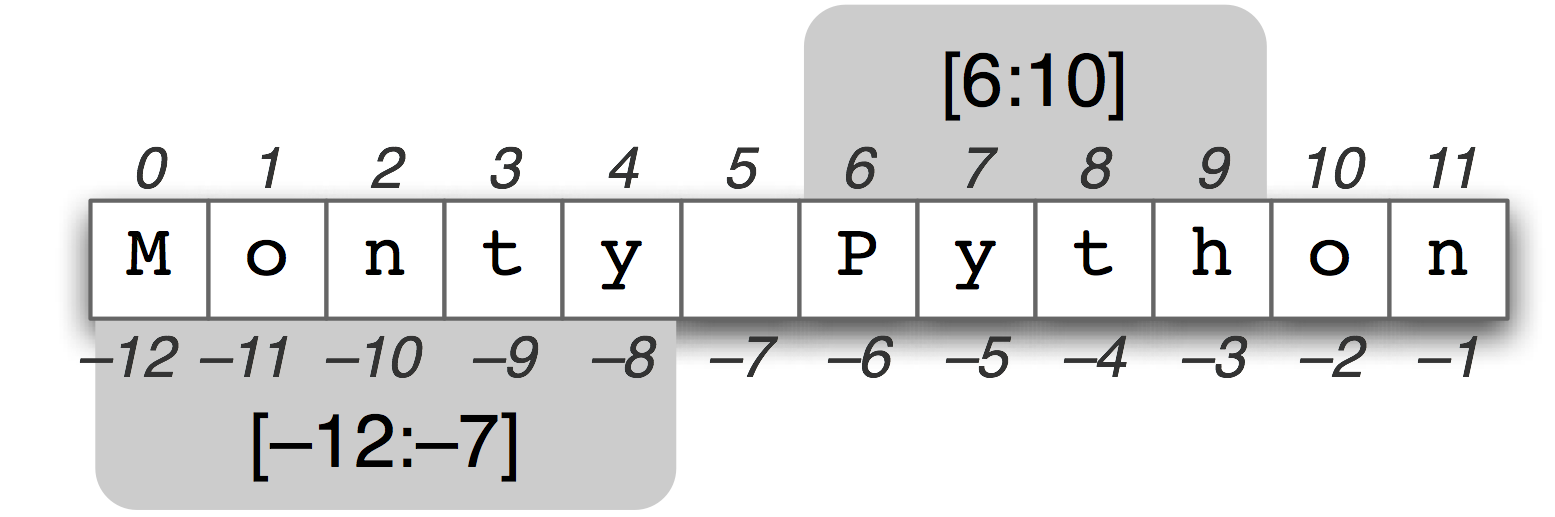
'f', 'g', 'p', 'b', 'y', 'v', 'k', 'q', 'j', 'x', 'z']

Здесь мы получили список символов алфавита, упорядоченный по частоте встречаемости в тексте. Частотное распределение можно визуализировать с помощью fdist.plot(). Относительная частота символов в тексте может быть использована для идентификации языка текста.

**Самостоятельно 3.2.3\_1:** Найдите 3 наиболее часто используемых согласных в английском переводе «Ревизора» из корпуса Гутенберга.

**Самостоятельно 3.2.3\_2:** Каких символов в «Ревизоре» больше – гласных или согласных.

### 3.2.4 Доступ к подстрокам



**Рисунок 3.2**: Выделение подстрок: Строка "Monty Python" показана со своими положительными и отрицательными индексами; две подстроки выделены с помощью нотации подстрок. Подстрока [m,n] содержит символы, начиная с позиции m, заканчивая символом n-1.

Подстрока – это последовательный фрагмент строки, который мы выделяем из строки для дальнейшей обработки. К подстрокам можно обращаться помощью той же нотации, что и для списков. (см. [3.2](http://www.nltk.org/book/ch03.html" \l "fig-string-slicing)). Например, следующий код получает доступ к подстроке, начиная с символа с индексом 6 до символа с индексом 10 (не включая):

>>> monty[6:10]

'Pyth'

Мы видим, что символы 'P', 'y', 't' и 'h' соответствуют monty[6]...monty[9], но не monty[10].

Подстроки можно выделять с помощью отрицательных индексов. И тут действует то же правило: начиная от стартовой точки и останавливаясь на символе, предшествующем завершающему индексу. В примере мы останавливаемся перед символом пробела.

>>> monty[-12:-7]

'Monty'

Так же, как с фрагментами списков, если мы пропустим первое значение, подстроки начинаются с начала строки. Если пропущено второе значение – подстрока продолжается до конца строки:

>>> monty[:5]

'Monty'

>>> monty[6:]

'Python'

Мы можем проверить, содержит ли строка некоторую подстроку с помощью оператора in:

>>> phrase = 'And now for something completely different'

>>> if 'thing' in phrase:

... print('found "thing"')

found "thing"

Мы также можем находить позицию подстроки в строке с помощью функции find():

>>> monty.find('Python')

6

**Самостоятельно 3.2.4\_1:** Составьте предложение и присвойте его переменной, например sent = 'my sentence...'. Затем запишите выражения, выделяющие из предложения отдельные слова. (Это, очевидно, не самый удобный метод обработки слов в тексте!)

### 3.2.5 Другие операции над строками

Python имеет всестороннюю поддержку обработки строк. В Таблице 3.2 показан перечень некоторых строковых функций. Для более подробной информации воспользуйтесь командой help(str).

***Таблица 3.2****:* Полезные методы типа String: приведенные методы возвращают строку или список с соответствующим результатом

|  |  |
| --- | --- |
| Метод | Описание |
| s.find(t) | индекс первого вхождения строки t в строке s (-1, если не найдено) |
| s.rfind(t) | индекс последнего вхождения строки t в строке s (-1, если не найдено) |
| s.index(t) | аналогично s.find(t), но вызывет ValueError, если строка не найдена |
| s.rindex(t) | аналогично s.rfind(t), но вызывет ValueError, если строка не найдена |
| s.join(text) | Объединяет слова текста в строку, используя строку s как соединитель |
| s.split(t) | Разбивает строку s на части, разделяемые с помощью t (по умолчанию – пробелы). Результат – список. |
| s.splitlines() | Разбивает строку s на список строк |
| s.lower() | копия строки в нижнем регистреs |
| s.upper() | копия строки в верхнем регистреs |
| s.title() | a titlecased version of the string s |
| s.strip() | копия строки без лидирующих и завершающих пробелов |
| s.replace(t, u) | замена в строке s подстрок t на u |

**Самостоятельно 3.2.5\_1**: Продемонстрируйте примеры использования всех функций в таблице выше.

### 3.2.6 Различие между списками и строками

Строки и списки являются частным случаем **последовательностей**. Мы можем разделять их на части с помощью индексации и выделения подстрок, и объединять с помощью конкатенации. Однако нельзя объединять строки и списки:

>>> 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']

Когда мы открываем в Python-программе файл для чтения, мы получаем строку со всем содержимым файла. Далее, используя цикл for, мы можем получить доступ только к элементарным составляющим строки – отдельным символам. Мы не можем выбирать уровень детализации. В отличие от строк, элементами списков могут быть разные по величине элементы – абзацы, предложения, фразы, слова, символы. Поэтому типичная задача после чтения файла в строку – токенизация строки в список строк ([3.7](http://www.nltk.org/book/ch03.html" \l "sec-tokenization)). И наоборот, записывая результат в файл, мы обычно конвертируем список в строку ([3.9](http://www.nltk.org/book/ch03.html" \l "sec-formatting)).

Списки и строки имеют разную функциональность. Преимущество списков состоит в возможности изменять его элементы:

>>> beatles[0] = "John Lennon"

>>> del beatles[-1]

>>> beatles

['John Lennon', 'Paul', 'George']

Если мы попытаемся это сделать со строкой, то получим ошибку:

>>> query[0] = 'F'

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: object does not support item assignment

Дело в том, что строки являются неизменяемыми объектами (**immutable**) — после создания строки ее нельзя изменить. В то же время списки являются изменяемыми (**mutable**) - их содержимое можно изменять в любой момент времени. Как следствие, операции со списками модифицируют исходное значение, а не создают новый объект.

**Самостоятельно:** Закрепите Ваши знания о строках, выполнив некоторые упражнения в конце главы.

## 3.3 Обработка текстов в кодировке Unicode

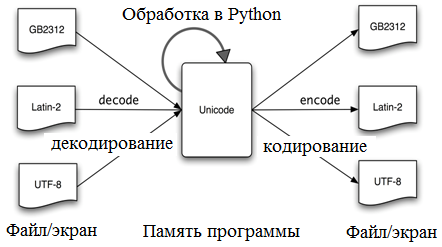
Наши программы часто должны иметь дело с различными языками и различными наборами символов. Концепция "простого текста" является фикцией. Если Вы живете в мире, использующем английский язык, то вероятно используете кодировку ASCII, даже не подозревая это. Если Вы живете в Европе, то можете использовать расширенное множество латинских символов, содержащее такие символы как "ø"(Дания и Норвегия), "ő" (Венгрия), "ñ" (Испания и Бретония), "ň" (Чехия и Словакия). Далее рассмотрим обзор средств использования кодировки Unicode для обработки текстов, использующих не-ASCII набор символов.

### 3.3.1 Что такое Unicode?

Unicode поддерживает более миллиона символов. Каждому символу присвоено число, называемое **кодом** (**code point)**. В Python коды записываются в форме \u*XXXX*, где *XXXX* – число из 4-х разрядов в 16-ичном виде.

В программе мы можем работать с Unicode-строками, как с обычными строками. Однако, когда символы Unicode сохраняются в файле или выводятся на экран, они должны быть представлены как поток байтов. Некоторые кодировки (такие как ASCII и Latin-2) используют один байт для кода, так что они способны поддерживать лишь небольшое подмножество Unicode, достаточное для некоторого одного языка. Другие кодировки (например, UTF-8) используют несколько байтов для кода и могут представлять широкий диапазон символов Unicode.

Текстовые файлы представлены в одном из форматов, поэтому нам нужен механизм преобразования их в Unicode — декодирование (**decoding**). И, наоборот, для вывода Unicode в файл или на экран, его нужно преобразовать в соответствующую кодировку (**encoding**). Это показано на Рис. [3.3](http://www.nltk.org/book/ch03.html" \l "fig-unicode).



**Рисунок 3.3**: Кодирование и декодирование Unicode

С точки зрения Unicode, символы являются абстрактными сущностями, которые могут быть реализованы как один или несколько глифов (изображений символов), которые могут быть показаны на экране или напечатаны на бумаге. Шрифт – это соответствие между символами и глифами.

### 3.3.2 Извлечение кодированного текста из файлов

Допустим, у нас есть небольшой файл с известной кодировкой. Например, polish-lat2.txt, как следует из имени, фрагмент польского текста (из польской Википедии - http://pl.wikipedia.org/wiki/Biblioteka\_Pruska). Кодировка Latin-2 также известна как ISO-8859-2. Найти этот файл можно функцией nltk.data.find().

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

Функция open() читает закодированный файл в строку Unicode. Она использует параметр, определяющий кодировку используемого файла:

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

>>> for line in f:

... line = line.**strip**()

... print(line)

Pruska Biblioteka Państwowa. Jej dawne zbiory znane pod nazwą

"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez

Niemców pod koniec II wojny światowej na Dolny Śląsk, zostały

odnalezione po 1945 r. na terytorium Polski. Trafiły do Biblioteki

Jagiellońskiej w Krakowie, obejmują ponad 500 tys. zabytkowych

archiwaliów, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.

Если строка не отображается корректно на экране или если Вы хотите видеть соответствующие числовые коды символов, Вы можете конвертировать не-ASCII символы в их 2-разрядное \x*XX* или 4-разрядное \u*XXXX* представление:

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

>>> for line in f:

... line = line.strip()

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

b'Pruska Biblioteka Pa\\u0144stwowa. Jej dawne zbiory znane pod nazw\\u0105'

b'"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez'

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

b'odnalezione po 1945 r. na terytorium Polski. Trafi\\u0142y do Biblioteki'

b'Jagiello\\u0144skiej w Krakowie, obejmuj\\u0105 ponad 500 tys. zabytkowych'

b'archiwali\\xf3w, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.'

Теперь в результирующей строке коды выводятся как escape-последовательности с префиксом \u, например, \u0144. Соответствующий Unicode-символ отображается в виде глифа на экране как ń. Также мы видим код \xf3, который соответствует глифу ó, и находится в диапазоне 128-255.

В Python 3 исходный код кодируется с использованием UTF-8 по умолчанию. Благодаря этому в средах программирования для Python можно использовать в строках Unicode-символы. В любом случае Unicode-символы можно включать в строки с помощью escape-последовательности \u*XXXX*. Целочисленный код символа можно определить с помощью функции ord():

>>> ord('ń')

324

4-х разрядная 16-ичная нотация для 324 - это 0144 (это можно узнать с помощью hex(324)), и мы можем задавать строки с помощью подходящих escape-последовательностей.

>>> nacute = '\u0144'

>>> nacute

'ń'

**Замечание.** Если Вы уверены в правильнойкодировке строки, но код Python не отображает правильные глифы, нужно проверить, установлены ли в системе необходимые шрифты. Возможно, придется сконфигурировать локальные настройки для показа UTF-8-символов. Затем используйте print(nacute.encode('utf8')), чтобы увидеть символ ń на экране.

**Самостоятельно 3.3.2\_1**: Определите, сколько в polish-lat2 не-ASCII символов.

Мы также можем увидеть, как этот символ представлен в последовательности байтов внутри текстового файла:

>>> nacute.encode('utf8')

b'\xc5\x84'

Модуль unicodedata позволяет нам исследовать свойства Unicode-символов. В следующем примере мы выберем все символы в третьей строке нашего польского текста, чьи коды выходят за границы ASCII-диапазона, и выведем их UTF-8 последовательность байтов, с последующим целочисленным кодом используя стандартные Unicode-соглашения (т.е. префикс U+ перед 16-ичными цифрами).

>>> 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)))

b'\xc3\xb3' U+00f3 LATIN SMALL LETTER O WITH ACUTE

b'\xc5\x9b' U+015b LATIN SMALL LETTER S WITH ACUTE

b'\xc5\x9a' U+015a LATIN CAPITAL LETTER S WITH ACUTE

b'\xc4\x85' U+0105 LATIN SMALL LETTER A WITH OGONEK

b'\xc5\x82' U+0142 LATIN SMALL LETTER L WITH STROKE

**Самостоятельно 3.3.2\_2**: Выведите информацию о 10 соседних не-ASCII символов.

**Самостоятельно 3.3.2\_3**: Сколько всего acute-символов?

Если заменить c.encode('utf8') в коде c, и если Ваша система поддерживает UTF-8, Вы получите следующий результат на экране:

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

В другом случае Вам может потребоваться заменить кодировку 'utf8' в примере на 'latin2', если это обусловлено требованиями к Вашей системе.

Следующие примеры иллюстрируют как методы строкового типа и модуль re могут работать с символами Unicode. (Подробнее о модуле re поговорим в следующей секции. Символ \w соответствует обычному символу в слове (см. [3.4](http://www.nltk.org/book/ch03.html" \l "tab-re-symbols)).

>>> 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()

'\u015bwiatowej'

Токенизаторы NLTK допускают использование строк Unicode в качестве входных данных, и, соответственно, возвращают строки Unicode в качестве результата.

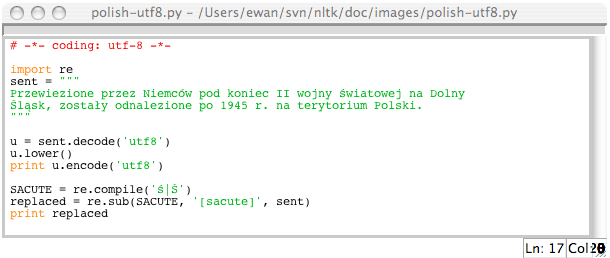
>>> word\_tokenize(line)

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

**Самостоятельно 3.3.2\_4**: Найдите в некотором тексте с кодировкой latin-2 символы, чьи глифы отличаются от глифов обычных символов английского алфавита. Выведите эти символы в виде их глифов и числовых кодов.

### 3.3.3 Использование локализованной кодировки в Python

Если Вы привыкли работать с символами в некоторой конкретной кодировке, то, вероятно, захотите использовать Ваши стандартные методы ввода и редактирования строк в файлах Python. Для этого включите строчку '# -\*- coding: <coding> -\*-' в начале Вашей программы. Заметьте, что *<coding>* должен быть правильной строкой, вроде 'latin-1', 'big5' или 'utf-8' (см. [3.4](http://www.nltk.org/book/ch03.html" \l "fig-polish-utf8)).



**Рисунок 3.4**: Unicode в среде IDLE: Строковые литералы в UTF-8 в редакторе IDLE; для того, чтобы это работало, в настройках IDLE укажем шрифт Courier CE.

Этот пример также иллюстрирует использование регулярных выражений для кодированных строк.

## 3.4 Регулярные выражения для выявления шаблонов слов

Многие задачи лингвистической обработки включают сопоставление шаблонам. Например, мы можем найти слова, завершающиеся строкой *ed*, используя endswith('ed'). В [4.2](http://www.nltk.org/book/ch01.html" \l "tab-word-tests) мы рассмотрим использование многих таких шаблонов слов. Регулярные выражения дают нам более мощные и гибкие методы описания полезных символьных шаблонов.

**Замечание:** Существует немало описаний регулярных выражений. Чтобы не повторяться, сосредоточим наше внимание на использование регулярных выражений на различных стадиях лингвистической обработки. Как обычно, мы адаптируем проблемно-ориентированный подход и представим новые особенности только при необходимости их использования для решения практических задач. В дальнейшем обсуждении будем обозначать регулярные выражения как pattern.

Для использования регулярных выражений в Python мы должны использовать библиотеку re, используя import re. Кроме того, нам также понадобится список слов для поиска, как в [4](http://www.nltk.org/book/ch02.html" \l "sec-lexical-resources). Мы выполним его предобработку и удалим из него имена собственные.

>>> import re

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

### 3.4.1 Использование мета-символов

Найдем слова, завершающиеся на «*ed»*, используя регулярное выражение «ed$». Будем использовать функцию re.search(p,s) для проверки факта вхождения шаблона p внутри строки s. Мы должны указать интересующие символы и знак $, который имеет специальный смысл в контексте регулярных выражений – в них он соответствует концу слова:

>>> [w for w in wordlist if re.search('ed$', w)]

['abaissed', 'abandoned', 'abased', 'abashed', 'abatised', 'abed', 'aborted', ...]

Символ ”.” соответствует любому одиночному символу. Предположим, у нас есть пространство для 8-символьного слова из кроссворда с символом ”*j*” в третьей позиции и с символом «*t»* в 6-й позиции. На месте каждой пустой ячейки мы используем символ ”.“:

>>> [w for w in wordlist if re.search('^..j..t..$', w)]

['abjectly', 'adjuster', 'dejected', 'dejectly', 'injector', 'majestic', ...]

**Самостоятельно 3.4.1\_1**: Символ ”^” соответствует началу строки, а символ «$” – концу строки. Каким будет результат предыдущего примера, если мы не укажем эти символы в шаблоне и оставим только «..j..t..»?

Наконец, символ «?» говорит о том, что предыдущий символ является необязательным. Таким образом, «^e-?mail$» будет соответствовать как «email» так и «e-mail». Мы можем подсчитать количество вхождений этого слова (в любом варианте), используя sum(1 for w in text if re.search('^e-?mail$', w)).

### 3.4.2 Диапазоны и замыкания



**Рисунок 3.5**: T9: Текст на 9 клавишах

Для ввода текста на мобильных устройствах используется система **T9** (см. [3.5](http://www.nltk.org/book/ch03.html" \l "fig-t9)). Два или более слов, которые вводятся с помощью одинаковой последовательности клавиш назовем **текстонимами**. Например, слова *hole* и *golf* вводятся с помощью одинаковой последовательности клавиш 4653. Какие еще слова можно получить с помощью данной последовательности клавиш? Для этого можно воспользоваться регулярным выражением «^[ghi][mno][jlk][def]$»:

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

['gold', 'golf', 'hold', 'hole']

Первая часть выражения - «^[ghi]» - соответствует символам *g*,*h* или *i* в начале слова*.* Следующая часть - «[mno]» - означает, что вторым символом может быть *m*, *n* или *o*. Таким же способом заданы ограничения для третьего и четвертого символа. Заметим, что порядок символов в квадратных скобках не имеет значения, поэтому шаблон «^[hig][nom][ljk][fed]$» соответствует тем же словам.

Например, шаблон «^[ghijklmno]+$», или более кратко «^[g-o]+$» соответствует словам, которые используют только клавиши 4, 5, 6 в центральной части клавиатуры, или «^[a-fj-o]+$», соответствующий словам, использующим правый верхний угол клавиатуры - 2, 3, 5, 6.

**Самостоятельно 3.4.2\_1**: Попробуйте найти другие примеры таких слов ("finger-twisters"), рассматривая слова, использующие только часть цифровой клавиатуры.

**Самостоятельно 3.4.2\_2**: Найдите в некотором тексте из корпуса Брауна все слова, состоящие только из символов, находящихся в двух нижних рядах клавиатуры T9.

А что же означают символы «-» и «+»? Рассмотрим подробнее роль символа «+». Он может применяться к отдельным символам и группам символов в квадратных скобках:

>>> 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)]

['miiiiiiiiiiiiinnnnnnnnnnneeeeeeeeee', 'miiiiiinnnnnnnnnneeeeeeee', 'mine',

'mmmmmmmmiiiiiiiiinnnnnnnnneeeeeeee']

>>> [w for w in chat\_words if re.search('^[ha]+$', w)]

['a', 'aaaaaaaaaaaaaaaaa', 'aaahhhh', 'ah', 'ahah', 'ahahah', 'ahh',

'ahhahahaha', 'ahhh', 'ahhhh', 'ahhhhhh', 'ahhhhhhhhhhhhhh', 'h', 'ha', 'haaa',

'hah', 'haha', 'hahaaa', 'hahah', 'hahaha', 'hahahaa', 'hahahah', 'hahahaha', ...]

Можно догадаться, что «+» означает "один или более экземпляров предшествующего элемента", который может быть отдельным символом вроде «m», множество типа [fed] или диапазон вроде [d-f]. Теперь заменим «+» на «\*», что означает "0 или более экземпляров предшествующего элемента ". Регулярное выражение «^m\*i\*n\*e\*$» будет соответствовать всем строкам, которым соответствует «^m+i+n+e+$», но кроме этого словам, в которых некоторые символы отсутствуют, например *me*, *min*, и *mmmmm*. Заметьте, что символы «+» и «\*» иногда называют **замыканиями Клини** или просто **замыканиями**.

Оператор «^» имеет другое назначение, когда встречается первым внутри [ ]. Например, «[^aeiouAEIOU]» соответствует любому символу, кроме гласных символов. Мы можем производить поиск внутри корпуса NPS Chat слов, которые составлены только из «не гласных» символов с помощью выражения «^[^aeiouAEIOU]+$». В результате будут найдены элементы вроде: «:):):)», «grrr», «cyb3r» «zzzzzzzz». Заметьте, что в строках содержатся не алфавитные символы.

Далее представлено еще несколько примеров регулярных выражений для поиска шаблонов, демонстрирующих использование некоторых новых символов: «\»,«{}»,«()» и «|»:

>>> 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', ...]

**Самостоятельно:** Изучите предыдущие примеры и постарайтесь усвоить, что означают символы «\», «{}», «()» и «|» перед тем, как продолжить чтение.

Вы, вероятно, знаете, что «\» означает, что следующий за ним символ теряет свой специальный смысл и должен буквально соответствовать символу в искомом слове. Таким образом, если «.» является специальным, то «\.» соответствует только точке. Выражения в скобках, вроде {3,5}, определяют количество повторений предыдущего элемента. Символ «|» обозначает возможность выбора между элементами слева и справа от него. Скобки определяют область действия оператора: они могут быть использованы совместно с «|», например следующим образом: «w(i|e|ai|oo)t». Этому шаблону соответствуют *wit*, *wet*, *wait* и *woot*.

**Самостоятельно 3.4.2\_3**: Выясните, что произойдет, когда Вы пропустите скобки в последнем выражении и произведете поиск по шаблону «ed|ing$».

Рассмотренные выше метасимволы приведены в Таблице [3.3](http://www.nltk.org/book/ch03.html" \l "tab-regexp-meta-characters1).

***Таблица 3.3****:*Основные метасимволы регулярных выражений

|  |  |
| --- | --- |
| Оператор | Поведение |
| . | Символ-заменитель, соответствует любому символу |
| ^abc | Соответствие шаблону *abc* в начале слова |
| abc$ | Соответствие шаблону *abc* в конце слова |
| [abc] | Соответствует одному из символов множества в скобках |
| [A-Z0-9] | Соответствует одному из символов диапазона |
| ed|ing|s | Соответствует одной из двух строк (дизюнкция) |
| \* | 0 или более экземпляров предшествующего элемента, a\*, [a-z]\* (также известно как замыкание Клини) |
| + | 1 или более экземпляров предшествующего элемента, a+, [a-z]+ |
| ? | 0 или 1 экземпляр предшествующего элемента (то есть, необязательный элемент), a?, [a-z]? |
| {n} | Точго *n* повторений (n – неотрицательное целое) |
| {n,} | Не менее *n* повторений |
| {,n} | Не более *n* повторений |
| {m,n} | Не менее *m* и не более *n* повторений |
| a(b|c)+ | Скобки, указывающие область действия оператора |

Для интерпретатора Python регулярное выражение – это обычная строка. Если в строке имеется «\» с последующим символом, эти символы будут интерпретироваться специальным образом. Например, «\b» интерпретируется как символ backspace. В общем случае, при использовании регулярных выражений, содержащих обратный слэш, нужно каким-то образом сообщать интерпретатору не заглядывать внутрь строки, а сразу передавать ее модулю re для обработки. Это делается путем размещения префикса в виде буквы r, перед строкой, что означает **raw string**. Например, строка r'\band\b' содержит два символа «\b », которые интерпретируются библиотекой re как отметки границ слова, а не символы backspace. Если Вы выработаете привычку использовать r'...' для регулярных выражений, как это будет делаться нами далее, то сможете избежать многих неожиданных затруднений.

## 3.5 Полезные применения регулярных выражений

Ранее использованные примеры посвящены поиску слов *w*, которые соответствуют некоторому регулярному выражению *regexp* с помощью re.search(regexp, w). Помимо проверки соответствия слов регулярному выражению, мы можем использовать регулярные выражения для извлечения полезного материала из слов или для некоторой модификации слов.

### 3.5.1 Извлечение фрагментов слов

Метод re.findall() ("find all") находит все (непересекающиеся) соответствия регулярному выражению в строке. Найдем все гласные в слове, а затем подсчитаем их количество:

>>> 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

Далее найдем все последовательности из двух или более гласных в некотором тексте и определим их относительную частоту:

>>> 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)

[('io', 549), ('ea', 476), ('ie', 331), ('ou', 329), ('ai', 261), ('ia', 253),

('ee', 217), ('oo', 174), ('ua', 109), ('au', 106), ('ue', 105), ('ui', 95)]

**Самостоятельно 3.5.1\_1:** В формате W3C Date Time Format даты представлены в виде 2009-12-31. Замените «?» в следующем коде Python регулярным выражением, чтобы преобразовать строку '2009-12-31' в список целых чисел [2009, 12, 31]:

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

### 3.5.2 Больше действий с фрагментами слов

Помимо использования re.findall() для извлечения материала из слов, имеются интересные возможности действий с фрагментами слов, например можно их заново собрать в строку.

Обычно английский текст в существенной степени избыточен и его по-прежнему легко читать после удаления гласных внутри слов. Например, слово *declaration* превращается в *dclrtn*, а *inalienable* - в *inlnble*, сохраняя стартовые и финальные гласные. Регулярное выражение в следующем примере ищет соответствия по стартовым и финальным гласным и всем согласным, игнорируя все остальные символы. Такая трех-ходовая дизъюнкция осуществляется слева направо. Если одна из трех составляющих соответствует слову, все остальные части регулярного выражения игнорируются. Мы используем re.findall() для нахождения всех подходящих фрагментов, а ''.join() – для объединения их в единое целое (см. [3.9](http://www.nltk.org/book/ch03.html" \l "sec-formatting) о подробностях операции join).

>>> 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

Далее попробуем скомбинировать регулярные выражения с условными частотными распределениями. Удалим из слов Rotokas все последовательности согласная-гласная, вроде *ka* и *si*. Далее вычислим и протабулируем частоту каждой пары:

>>> 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

Анализируя строки для *s* и *t*, заметим, что их частотные распределения частично дополняют друг друга, что выявляет определенную закономерность – фонемы с буквами *s* и *t* не являются полностью независимыми в языке. В качестве эксперимента мы можем удалить из алфавита Rotokas символ *s* и добавить правило произношения – буква *t* произносится как *s*, когда после нее следует *i*. (Заметьте, что единственное вхождение *su* в слове *kasuari*, позаимствовано из английского ('cassowary' – большая птица в Новой Зеландии).

Если мы хотим анализировать слова, которые представлены числами этой таблицы, полезно было бы иметь индекс, позволяющий быстро находить список слов , содержащих данную пару согласная-гласная, например, cv\_index['su'] должен дать все слова, содержащие *su*. Вот как это можно сделать:

>>> 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', ...]

Эта программа обрабатывает каждое слово w по очереди и для каждого из них находит каждую подстроку, соответствующую регулярному выражению «[ptksvr][aeiou]». В случае слова *kasuari*, она находит *ka*, *su* и *ri*. В результате список cv\_word\_pairs будет содержать ('ka', 'kasuari'), ('su', 'kasuari') и ('ri', 'kasuari'). Следующий шаг – использование nltk.Index(), преобразующего список в полезный индекс.

### 3.5.3 Нахождение основы слова (стэмминг)\*

Когда мы используем поисковый механизм Web, мы обычно не против того, что в найденных документах искомые термины имеют другие окончания. Запрос *laptops* найдет документы, содержащие *laptop* и наоборот. Действительно, *laptop* и *laptops* – всего лишь две формы одного слова в словаре (лемма). Для некоторых задач обработки текста мы хотим игнорировать окончания слов и иметь дело только с основой слова.

Существует несколько способов выделить основу из слова. Далее показано применение примитивного подхода, в котором из слов удаляется все, похожее на суффикс:

>>> 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

Несмотря на то, что для стемминга мы обычно используем встроенные в NLTK стэммеры, интересно рассмотреть, как для решения этой задачи можно использовать регулярные выражения. Первым нашим шагом будет построение дизъюнкции всех суффиксов. Эти суффиксы нужно ограничить скобками, чтобы ограничить диапазон действия дизъюнкции.

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

['ing']

Здесь re.findall() просто дает нам суффикс, даже если регулярное выражение сопоставлено целому слову. Это происходит потому, что скобки имеют еще одну функцию – определять извлекаемые подстроки. Если мы хотим использовать скобки только для определения диапазона действия дизъюнкции, в скобки нужно добавить ?:, что является очередной «тайной» особенностью регулярных выражений. Далее – пересмотренная версия:

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

['processing']

Однако часто нам нужно разделять слово на основу и суффикс. Поэтому нам нужно заключать в скобки обе части регулярного выражения:

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

[('process', 'ing')]

Выглядит многообещающе, однако еще остается проблема. Рассмотрим другое слово - *processes*:

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

[('processe', 's')]

Регулярное выражение некорректно нашло суффикс *-s* вместо *-es*. Это раскрывает еще одну тонкость: оператор «\*» является «жадным» и «.\*»-часть выражения пытается использовать как можно больше информации. Для использования «не жадной» версии оператора нужно использовать \*?:

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

[('process', 'es')]

Это работает, даже когда мы допускаем пустой суффикс, делая содержимое вторых скобок необязательным:

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

[('language', '')]

Этот подход все еще имеет много проблем, но мы все же определим функцию, выполняющую стемминг и применим ее ко всему тексту:

>>> 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 = 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', '.']

Заметим, что наше регулярное выражение удалило *s* не только из *ponds*, но также из *is* и *basis*. Это привело к выявлению не корректных слов, вроде *distribut* и *deriv*, но это допустимо в некоторых приложениях стемминга.

**Самостоятельно 3.5.3\_1:** На основе приведенного выше простого стеммера разработайте и примените аналогичный стеммер для русского или украинского языка.**3.12: Самостоятельно:**

**Самостоятельно 3.5.3\_2:** Добавьте в Ваш стеммер удаление префиксов слов.

### 3.5.4 Поиск токенизированного текста

Специальный тип регулярных выражений может использоваться для поиска в множестве слов текста (когда текст – это список слов). Например, "<a> <man>" находит все вхождения *a man* в тексте. Угловые скобки используются, чтобы отметить границы токенов, а любые пустые символы между угловыми скобками игнорируются (это поведение уникально для NLTK-метода findall() для текстов). В следующем примере мы добавим в выражение <.\*>, что соответствует любому одиночному токену, и окружим его скобками, благодаря чему получается только соответствующее слово (например, *monied*), но не соответствующая фраза (например, *monied man*). Второй пример находит фразы из трех слов, завершающихся словом *bro*. Последний пример находит последовательности из трех или более слов, начинающихся с буквы *l*.

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

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

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

monied; nervous; dangerous; white; white; white; pious; queer; good;

mature; white; Cape; great; wise; wise; butterless; white; fiendish;

pale; furious; better; certain; complete; dismasted; younger; brave;

brave; brave; brave

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

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

you rule bro; telling you bro; u twizted bro

>>> chat.findall(r"<l.\*>{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

**Самостоятельно:** Закрепите свое понимание регулярных выражений для поиска и замены с помощью nltk.re\_show(*p, s*), который помечает строку *s*, показывая все места, где был найден шаблон *p*, и nltk.app.nemo(), который предоставляет графический интерфейс для исследования регулярных выражений. Для большей практики выполните некоторые упражнения в конце главы.

Легко строить поисковые шаблоны, когда изучаемый лингвистический феномен связан с конкретными словами. В некоторых случаях небольшая доля креативности позволит продвинуться дальшею. Например, поиск в большом тексте выражений типа *x and other ys* позволяет выявить гипернимы (cf [5](http://www.nltk.org/book/ch02.html" \l "sec-wordnet)):

>>> 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

Имея достаточно текста, этот подход дает нам достаточно информации о таксономии объектов, не осуществляя ручной работы. Однако в результате наших поисковых запросов появляются ложные «положительные» элементы (false positives), которые нам хотелось бы исключить. Например, результат *demands and other factors* предлагает считать *demand* разновидностью *factor*, но это предложение связано с требованиями к зарплате. Тем не менее, мы можем конструировать нашу собственную онтологию понятий путем ручной корректировки результатов таких запросов.

**Замечание.** Такая комбинация автоматической и ручной обработки являются типичным способом конструирования нового корпуса. Мы вернемся к этому в главе [11](http://www.nltk.org/book/ch11.html" \l "chap-data).

Поиск в корпусе также отличается проблемой ложных «отрицательных» элементов (false negatives), то есть пропуск элементов, которые должны войти в результат. Рискованно делать вывод об отсутствии некоторого лингвистического явления в корпусе только из-за отсутствия некоторых экземпляров в результатах поиска. Возможно мы просто не достаточно продумали удачные шаблоны поиска.

**Самостоятельно:** Рассмотрите результаты применения шаблона *as x as y* для изучения информации о сущностях и их свойствах.

## 3.6 Нормализация текста

В более ранних примерах программ мы часто конвертировали текст в нижний регистр перед тем как делать что-либо со словами текста: set(w.lower() for w in text). Используя lower(), мы **нормализуем** текст к нижнему регистру таким образом, что различие между *The* и *the* игнорируется. Часто мы идем еще дальше и избавляемся от суффиксов и префиксов – эта задача называется **стеммингом**. Следующий шаг – убедиться, что в результате такого преобразования слово по-прежнему является словом из словаря, либо приведение к нормальной форме слова. Эта задача **лемматизации**. Рассмотрим эти задачи по порядку. Сначала определим данные, которые будем использовать в этом параграфе:

>>> 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 = **word\_tokenize**(raw)

### 3.6.1 Стеммеры

NLTK содержит несколько фирменных стеммеров и если Вам понадобится решать задачу стемминга, предпочтителнее использовать фирменные стеммеры, вместо разработки собственных регулярных выражений, которые, обычно, дают много неверных срабатываний. Стеммеры Porter и Lancaster используют собственные правила для удаления аффиксов. Посмотрите, как стеммер Porter корректно обрабатывает слово *lying*, приводя его к форме *lie*. Стеммер Lancaster с этой задачей не справляется.

>>> 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', '.']

Стемминг не является строго определенным процессом, и мы обычно выбираем стеммер, который лучше подходит для конкретного приложения. Стеммер Porter – хороший вариант, если Вы индексируете некоторые тексты и хотите поддерживать поиск с использованием альтернативных форм слов (показано в [3.6](http://www.nltk.org/book/ch03.html" \l "code-stemmer-indexing), где используется техника *объектно-ориентированного* программирования, выходящая за рамки данного пособия, форматирование строк будет рассмотрено в [3.9](http://www.nltk.org/book/ch03.html" \l "sec-formatting), а функция enumerate() - в [4.2](http://www.nltk.org/book/ch04.html" \l "sec-sequences)).

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 = int(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 = '{:>{width}}'.format(lcontext[-width:], width=width)

rdisplay = '{:{width}}'.format(rcontext[:width], width=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

**[Пример 3.6 (code\_stemmer\_indexing.py)](http://www.nltk.org/book/pylisting/code_stemmer_indexing.py)**: **Рисунок 3.6**: Индексация текста с использованием стеммера

**Самостоятельно 3.6.1\_1:** Для небольшого английского текста экспериментально определите, какой из двух стеммеров (Porter и Lancaster) дает меньшее количество ошибок.

**Самостоятельно 3.6.1\_2:** Найти и применить руський стемер. Сравнить со стандартными.

### 3.6.2 Лемматизация

Лемматизатор WordNetLemmatizer только удаляет аффиксы, если слово, образующееся в результате, имеется в его словаре. Этот процесс дополнительной проверки делает работу лемматизатора более медленной, чем у ранее описанных стеммеров. Заметьте, что он не справляется со словом *lying*, однако правильно превращает *women* в *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', '.']

Лемматизатор WordNet - хорошее средство, если Вам нужно составить словарь для некоторых текстов или список корректных лемм (основ слов).

**Самостоятельно 3.6.2\_1:** Для небольшого английского текста экспериментально оцените качество работы лемматизатора WordNetLemmatizer (например, % неправильно выявленных лемм).

**Замечание.** Другая задача лемматизации – определение нестандартных слов, включая числа, аббревиатуры, даты и сопоставление таких токенов на специальный словарь. Например, каждое десятичное число может быть сопоставлено токену 0.0, а каждый акроним - токену AAA. Это позволяет сохранить небольшой размер словаря и улучшает точность решения многих задач моделирования языка.

## 3.7 Регулярные выражения для токенизации текста

Токенизация – это задача разрезания строки на хорошо определенные лингвистические единицы, которые составляют основную языковую информацию. Хотя это и является фундаментальной задачей, мы можем отложить ее решение, поскольку многие корпуса уже токенизированы, и кроме того, NLTK имеет в своем составе несколько токенизаторов. Теперь, когда мы познакомились с регулярными выражениями, можно рассмотреть их использование для токенизации текста и получить более полный контроль над всем процессом.

### 3.7.1 Простые подходы к токенизации

Самый простой способ токенизации – разбиение его на основе пробельных символов. Рассмотрим следующий текст из *Alice's Adventures in Wonderland*:

>>> raw = """'When I'M a Duchess,' she said to herself, (not in a very hopeful tone

... though), 'I won't have any pepper in my kitchen AT ALL. Soup does very

... well without--Maybe it's always pepper that makes people hot-tempered,'..."""

Мы можем разделить плоский текст на основании пробелов, используя raw.split(). Чтобы это сделать с помощью регулярных выражений, недостаточно только находить символы-пробелы в строке, поскольку это даст токены, содержащие символ \n, хотя нам нужно было сопоставлять с любым количеством пробелов, табуляций и переходов на новую строку:

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

["'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)

["'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,'..."]

Регулярное выражение «[ \t\n]+» соответствует одному или более пробелам, символам табуляции (\t) и перехода на новую строку (\n). Также могут быть добавлены другие пустые символы (carriage-return, form-feed). Вместо этого, мы можем использовать в функции re сокращение \s, которое обозначает любой пустой символ. Поэтому предыдущий оператор может быть переписан следующим обпазом: re.split(r'\s+', raw).

**Важно:** Не забывайте начинать регулярное выражение с символа r ("raw"), что заставляет интерпретатор Python воспринимать строку буквально, не обрабатывая содержащиеся в строке символы «\».

Разбиение на основании пустых символов дает токены вроде '(not' и 'herself,'. В качестве альтернативы можно использовать символ \w, который для Python означает корректный символ слова (эквивалентно [a-zA-Z0-9\_]). Также имеется противоположный по смыслу специальный символ \W, сопоставляемый всем символам, кроме букв, цифр и подчеркивания. Мы можем использовать \W в простых регулярных выражениях для разбиения строк на основе любых символов, не являющихся символами слов:

>>> 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',

'']

Заметьте, что это дает нам пустые строки в начале и в конце (чтобы понять, почему это происходит, выполните 'xx'.split('x')). Мы получаем те же токены, но без пустых строк, с помощью re.findall(r'\w+', raw), используя шаблон, который сопоставляет слова вместо пробелов. Теперь, научившись сопосталять слова, мы можем расширить регулярное выражение, подходящее для многих случаев. Регулярное выражение «\w+|\S\w\*» сначала испытаем для сопоставления любым последовательностям словесных символов. Если соответствия не найдено, выражение попытается сопоставить любые непустые символы (\S противоположно \s), за которыми следуют словесные символы. Это означает, что знаки пунктуации объединяются с любой последующей буквой (например, *'s*), но эти последовательности из двух и более символов пунктуации разделяются.

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

Давайте обобщим функцию \w+ в предыдущем примере, чтобы допускать внутренние апострофы и тире: «\w+([-']\w+)\*». Это можно понимать как \w+, за которым следует 0 или более экземпляров [-']\w+; такому выражению соответствует, например *hot-tempered* или *it's*. (В это выражение нужно добавить ?:, по причинам, описанным ранее.) Также мы добавим в шаблон элементы для сопоставления кавычкам, так, чтобы они воспринимались как часть, отдельная от окружаемого ими текста.

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

В регулярное выражение также добавлено «[-.(]+», что позволяет токенизировать отдельно двойное тире, и открывающие скобки.

Таблица [3.4](http://www.nltk.org/book/ch03.html" \l "tab-re-symbols) содержит спецсимволы регулярных выражений, рассмотренные ранее и некоторые дополнительные.

***Таблица 3.4****:* Символы регулярных выражений

|  |  |
| --- | --- |
| Символ | Функция |
| \b | Граница слова (ширина 0) |
| \d | Любая десятичная цифра (эквивалент [0-9]) |
| \D | Любой нецифровой символ (эквивалент [^0-9]) |
| \s | Любой пустой символ (эквивалент [ \t\n\r\f\v]) |
| \S | Любой непустой символ (эквивалент [^ \t\n\r\f\v]) |
| \w | Любой буквенно-цифровой символ (эквивалент [a-zA-Z0-9\_]) |
| \W | Любой не буквенно-цифровой символ (эквивалент [^a-zA-Z0-9\_]) |
| \t | Символ табуляции |
| \n | Символ перехода на новую строку |

### 3.7.2 Токенизатор NLTK на регулярных выражениях

Функция nltk.regexp\_tokenize() похожа на re.findall() (мы использовали ее для токенизации). Однако nltk.regexp\_tokenize() более эффективна для токенизации и позволяет избежать необходимости специальной обработки скобок. Для читабельности мы разобъем регулярное выражение на несколько строк и добавим к ним комментарии. Специальный элемент (?x) ("verbose flag") говорит Python избавляться от внутренних пробелов и комментариев.

>>> 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; includes ], [

... '''

>>> nltk.regexp\_tokenize(text, pattern)

['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']

При использовании флага (?x) Вы уже не можете использовать ' ' для сопоставления с пробелом. Вместо этого следует использовать \s. Функция regexp\_tokenize() имеет дополнительный параметр gaps. Если он установлен в True, регулярное выражение определяет промежутки между токенами, так же как re.split().

**Замечание** Вы можете оценить работу токенизатора, сравнивая результирующие токены с исходным списком слов, и определить токены, которые отсутствуют в списке слов, используя set(tokens).difference(wordlist). Возможно, Вам захочется первоначально перевести все токены в нижний регистр.

### 3.7.3 Дополнительные аспекты токенизации

Токенизация оказывается намного более сложной задачей, чем Вы ожидали. В некоторых случаях может не оказаться однозначных ответов о том, что считать токеном в зависимости от предметной области.

Когда мы разрабатываем токенизатор, может оказаться полезным иметь доступ к плоскому тексту, который был токенизирован вручную, и сравнить результаты Вашей токенизации с эталонными токенами. Коллекция корпусов NLTK включает образец данных из Penn Treebank, включая плоский текст из Wall Street Journal (nltk.corpus.treebank\_raw.raw()) и его токенизированную версию (nltk.corpus.treebank.words()).

Завершающий аспект токенизации – присутствие сокращений, таких как *didn't*. Если мы анализируем смысл предложения, возможно будет полезно нормализовать сокращение до двух отдельных форм: *did* и *n't* (или *not*). Мы можем это сделать с помощью хэш-таблицы (lookup table).

## 3.8 Сегментация

В этой секции обсуждаются более продвинутые концепции.

Токенизация – это частный случай более общей задачи **сегментации**. В этом параграфе мы рассмотрим два других вида задачи, в которых используется радикально другая техника, в сравнении с рассмотренной выше.

### 3.8.1 Сегментация предложений

Работа с текстами на уровне отдельных слов часто предполагает возможность разбивать текст на отдельные предложения. Мы уже видели, что некоторые корпуса уже предоставляют доступ к предложениям. В следующем примере мы вычислим среднее количество слов в предложении для корпуса Brown:

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

20.250994070456922

В других случаях текст доступен только в форме потока символов. Перед токенизацией текста на слова, нужно сегментировать его на предложения. NLTK позволяет сделать это, прибегнув к помощи сегментатора предложений Punkt [(Kiss & Strunk, 2006)](http://www.nltk.org/book/bibliography.html" \l "kissstrunk2006). Далее – пример его использования на тексте романа Честертона «**[Человек, который был Четвергом](https://ru.wikipedia.org/wiki/%D0%A7%D0%B5%D0%BB%D0%BE%D0%B2%D0%B5%D0%BA,_%D0%BA%D0%BE%D1%82%D0%BE%D1%80%D1%8B%D0%B9_%D0%B1%D1%8B%D0%BB_%D0%A7%D0%B5%D1%82%D0%B2%D0%B5%D1%80%D0%B3%D0%BE%D0%BC)**». (Если к моменту использования Вами сегментатора, его внутренняя информация была изменена, Вы можете увидеть другие результаты):

>>> text = nltk.corpus.gutenberg.raw('chesterton-thursday.txt')

>>> sents = nltk.sent\_tokenize(text)

>>> pprint.pprint(sents[79:89])

['"Nonsense!"',

'said Gregory, who was very rational when anyone else\nattempted paradox.',

'"Why do all the clerks and navvies in the\n'

'railway trains look so sad and tired, so very sad and tired?',

'I will\ntell you.',

'It is because they know that the train is going right.',

'It\n'

'is because they know that whatever place they have taken a ticket\n'

'for that place they will reach.',

'It is because after they have\n'

'passed Sloane Square they know that the next station must be\n'

'Victoria, and nothing but Victoria.',

'Oh, their wild rapture!',

'oh,\n'

'their eyes like stars and their souls again in Eden, if the next\n'

'station were unaccountably Baker Street!"',

'"It is you who are unpoetical," replied the poet Syme.']

Заметьте, что этот пример основан на речи персонажа этого романа. Эта речь состоит из нескольких предложений, которые были разделены на строки. Это нормальное поведение большинства подобных приложений.

Разделение на предложения – довольно сложная задача, поскольку точка может использоваться как часть сокращений, а некоторые точки являются одновременно частью сокращения и концом предложения, например, *U.S.A.* в конце предложения.

Другой подход к сегментации предложений описан в [6.2](http://www.nltk.org/book/ch06.html" \l "sec-further-examples-of-supervised-classification).

### 3.8.2 Word Segmentation

Для некоторых систем письма токенизация текста еще более усложняется из-за того, отсутствуют визуальные границы слов. Например. В китайском языке трехсимвольная строка: 爱国人 (ai4 - "love" (глагол), guo2 - "country", ren2 - "person") может быть токенизирована как 爱国 / 人, "country-loving person" или как 爱 / 国人, "love country-person."

Аналогичная проблема возникает при обработке разговорной речи, когда слушатель должен сегментировать непрерывный поток речи в отдельные слова. Особенно сложным случаем этой проблемы является ситуация, когда мы не знаем последующих слов. Это бывает, когда слушатель учится, например, ребенок слушает взрослого. Рассмотрим следующий искусственный пример, в котором были удалены границы между словами:

(1a) doyouseethekitty

(1b) seethedoggy

(1c) doyoulikethekitty

(1d) likethedoggy

Нашим первым ходом будет формулировка проблемы: нам нужно найти способ отделить содержимое текста от сегментации. Мы можем сделать это, помечая каждый символ логической величиной, сообщающей, находится ли вслед за этим символом конец слова (эта идея в дальнейшем будет обозначаться термином "chunking" в [7.](http://www.nltk.org/book/ch07.html" \l "chap-chunk)). Будем считать, что слушатель получает информацию о границах, что обычно выражается в паузах между словами. Далее показан возможный пример, включая начальное состояние и результирующую сегментацию:

>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"

>>> seg1 = "0000000000000001000000000010000000000000000100000000000"

>>> seg2 = "0100100100100001001001000010100100010010000100010010000"

Заметьте, что строка сегментации состоит из нулей и единиц. В ней на один символ меньше, чем в исходном тексте, поскольку текст длиной *n* может быть разбит *n-1* разделителем. Функция segment() в [3.7](http://www.nltk.org/book/ch03.html" \l "code-segment) демонстрирует, что мы можем опять вернуться к тексту, но уже сегментированному.

def segment(text, segs):

words = []

last = 0

for i in range(len(segs)):

if segs[i] == '1':

words.append(text[last:i+1])

last = i+1

words.append(text[last:])

return words

>>> text = "doyouseethekittyseethedoggydoyoulikethekittylikethedoggy"

>>> seg1 = "0000000000000001000000000010000000000000000100000000000"

>>> seg2 = "0100100100100001001001000010100100010010000100010010000"

>>> segment(text, seg1)

['doyouseethekitty', 'seethedoggy', 'doyoulikethekitty', 'likethedoggy']

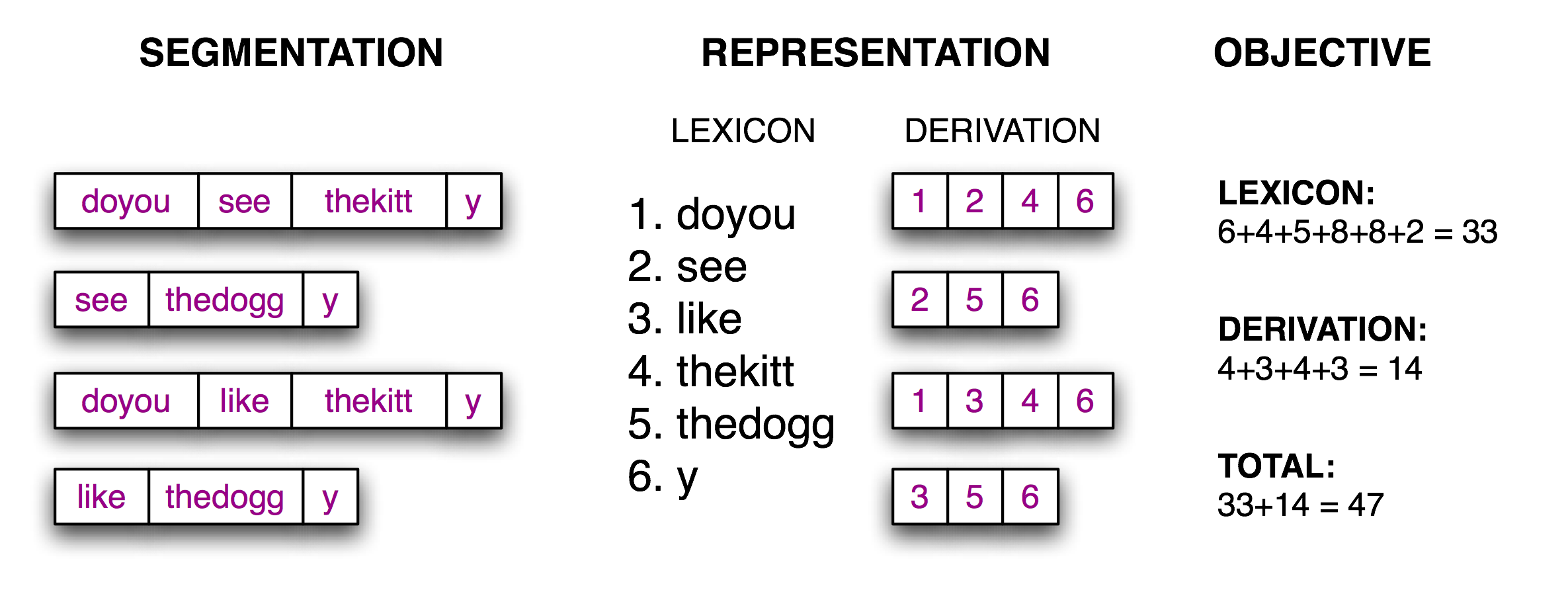
>>> segment(text, seg2)

['do', 'you', 'see', 'the', 'kitty', 'see', 'the', 'doggy', 'do', 'you',

'like', 'the', 'kitty', 'like', 'the', 'doggy']

**Пример 3.8.1**: Реконструкция сегментированного текста из строки:seg1 и seg2 представляет начальую и финальную сегментацию гипотетической речи, воспринимаемой ребенком; функция segment() может быть использована для реконструкции сегментированного текста.

Теперь задача сегментации становится поисковой задачей: найти битовую строку, которая позволит корректно сегментировать строку текста в слова. Будем считать, что обучаемый получает слова и хранит их во внутреннем лексиконе. Имея подходящий лексикон, становится возможным реконструировать исходный текст в виде последовательности лексических элементов. В соответствии с [(Brent, 1995)](http://www.nltk.org/book/bibliography.html" \l "brent1995), мы можем определить **целевую функцию** – оценочную функцию, значения которой мы попытаемся оптимизировать на основе размера лексикона (количество символов в словах плюс дополнительный символ-разделитель, отмечающий конец каждого слова) и объема информации, необходимого для реконструкции исходного текста из лексикона. Это проиллюстрировано на Рисунке [3.8](http://www.nltk.org/book/ch03.html" \l "fig-brent).



**Рисунок 3.8**: Вычисление целевой функции: Имея гипотетическую сегментацию исходного текста (слева), определить лексикон и таблицу (derivation), позволяющую реконструировать исходный текст, затем суммировать количество символов, используемых каждым лексическим элементом (включая маркер границы) и количество лексических элементов, используемых в каждом выводе, что будет служить рейтингом качества сегментации – меньшие значения рейтинга указывают на лучшее качество сегментации.

Такую целевую функцию нетрудно реализовать, как показано в Примере [3.3](http://www.nltk.org/book/ch03.html" \l "code-evaluate).

def evaluate(text, segs):

words = segment(text, segs)

text\_size = len(words)

lexicon\_size = sum(len(word) + 1 for word in 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)

47

>>> evaluate(text, seg2)

48

>>> evaluate(text, seg1)

64

**Пример 3.8.2**: Вычисление стоимости хранения лексикона и реконструирования исходного текста

Последний шаг – поиск шаблона из нулей и единиц, минимизирущего целевую функцию, определенну ранее. Заметим, что наилучшая сегментация включает «слова» вроде *thekitty*, поскольку в данных нет достаточно сведений, чтобы произвести дальнейшее разбиение.

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, 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)

61 ['doyouseetheki', 'tty', 'see', 'thedoggy', 'doyouliketh', 'ekittylike', 'thedoggy']

59 ['doy', 'ouseetheki', 'ttysee', 'thedoggy', 'doy', 'o', 'ulikethekittylike', 'thedoggy']

57 ['doyou', 'seetheki', 'ttysee', 'thedoggy', 'doyou', 'liketh', 'ekittylike', 'thedoggy']

55 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'likethekittylike', 'thedoggy']

54 ['doyou', 'seethekit', 'tysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']

52 ['doyou', 'seethekittysee', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']

43 ['doyou', 'see', 'thekitty', 'see', 'thedoggy', 'doyou', 'like', 'thekitty', 'like', 'thedoggy']

'0000100100000001001000000010000100010000000100010000000'

**[Пример 3.10](http://www.nltk.org/book/pylisting/code_anneal.py)**: Недетерминированный поиск с использованием имитации отжига: начать поиск только с фрагментации фраз; случайно изменяя нули и единицы, пропорционально «температуре»; с каждой итерацией температура снижается, а возмущение границ уменьшается. Поскольку этот алгоритм поиска не является детерминированным, вы можете увидеть несколько иной результат.

Имея достаточно данных, можно автоматически сегментировать текст на слова с достаточной степенью точности. Такие методы могут применяться к токенизации систем письма, которые не имеют визуального представления границ между словами.

**Самостоятельно 3.8.2\_1:** Выполните несколько экспериментов по сегментации слов и определите, как изменяется качество сегментатора в зависимости от входного текста.

## 3.9 Форматирование: От списков к строкам

Часто мы пишем программу для выявления одного элемента данных, такого как конкретный элемент в корпусе, который соответствует сложному критерию, или или одну обобщающую статистику, такую как количество слов или производительность тегера. Еще чаще мы пишем программы для создания структурированного результата, например табуляции чисел или лингвистических форм или переформатирование исходной информации. Когда требуемый результат является лингвистическим, результат в виде текста является наиболее естественным вариантом. Однако, если результаты имеют числову природу, может оказаться предпочтительнее графический результат. П этом параграфе мы узнаем о различных способах представления результата.

### 3.9.1 От списков к строкам

Простейший вид структурированного объекта, который мы используем для обработки текстов – это список слов. Когда мы хотим выводить слова на дисплей или в файл, мы должны конвертировать эти списки в строки. В Python это можно сделать методом join(), указав строку, используемую как соединитель.

>>> 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.'

Таким образом, ' '.join(silly) соединяет все элементы в списке silly в одну большую строку, используя символ ' ' в качестве разделителя между элементами. То есть join() – это метод строки, которую Вы используете в качестве «клея». (Многие считают такое использование join() противоречащим интуиции). Метод join() работает только для списков строк – то, что мы называем текстом — сложный тип, который имеет некоторые привилегии в Python.

### 3.9.2 Строки и форматы

Мы видели, что есть два способа показать содержимое объекта:

>>> word = 'cat'

>>> sentence = """hello

... world"""

>>> print(word)

cat

>>> print(sentence)

hello

world

>>> word

'cat'

>>> sentence

'hello\nworld'

Функция print заставляет Python воспроизвести наиболее читабельную форму объекта. Второй способ – указание имени объекта в командной строке – показывает некоторые детали устройства объекта. Однако оба способа не дают ключ к разгадке внутреннего устройства объекта.

Существует много других полезных способов отображать объект в виде строки символов. Это может быть полезно для человека, читающего текст, или потому, что мы хотим экспортировать наши данные в определенный формат файла для использования во внешней программе.

Форматированный вывод обычно содержит комбинацию переменных и предварительно заданных строк, например. с учетом частотного распределения fdist:

>>> fdist = nltk.FreqDist(['dog', 'cat', 'dog', 'cat', 'dog', 'snake', 'dog', 'cat'])

>>> for word in sorted(fdist):

... print(word, '->', fdist[word], end='; ')

cat -> 3; dog -> 4; snake -> 1;

Операторы print, содержащие переменные и константы, могут быть трудночитаемыми. В качестве альтернативы можно использовать **форматирование строк.**

>>> for word in sorted(fdist):

... print('{}->{};'.format(word, fdist[word]), end=' ')

cat->3; dog->4; snake->1;

Чтобы понять, что здесь происходит, проверим строку формата на собственном примере.

>>> '{}->{};'.format('cat', 3)

'cat->3;'

Фигурные скобки '{}' действуют как **место заполнения** строковыми значеними объектов, которые передаются методу str.format(). Мы внедряем '{}' в строку формата, а затем вызываем format() для осуществления подстановки в них соответствующих данных. Строка, содержащая поля замены, называется **строкой формата**.

В строке формата мы можем иметь любое количество заполнителей, но метод str.format должен иметь точно столько же параметров.

>>> '{} wants a {} {}'.format ('Lee', 'sandwich', 'for lunch')

'Lee wants a sandwich for lunch'

>>> '{} wants a {} {}'.format ('sandwich', 'for lunch')

Traceback (most recent call last):

...

'{} wants a {} {}'.format ('sandwich', 'for lunch')

IndexError: tuple index out of range

Параметры format() используются справа налево и любые избыточные параметры просто игнорируются.

>>> '{} wants a {}'.format ('Lee', 'sandwich', 'for lunch')

'Lee wants a sandwich'

Для заполнения в нестандартном порядке можно использовать места заполнения с индексами:

>>> 'from {1} to {0}'.format('A', 'B')

'from B to A'

Мы также можем указать значения для заполнителей косвенно. Вот пример с использованием цикла for:

>>> template = 'Lee wants a {} right now'

>>> menu = ['sandwich', 'spam fritter', 'pancake']

>>> for snack in menu:

... print(template.format(snack))

...

Lee wants a sandwich right now

Lee wants a spam fritter right now

Lee wants a pancake right now

### 3.9.3 Выравнивание информации

Пока наши строки формата генерировали вывод произвольной ширины на странице (или экране). Мы можем добавить отступы для получения вывода заданной ширины, вставив в заполнитель ':', за которым следует целое число. Таким образом, {: 6} указывает, что мы хотим, чтобы строка имела фиксированную ширину 6. По умолчанию содержимое прижимается вправо, но с помощью параметра выравнивания '<' перед спецификатором ширины можно выравнивать числа по левому краю.

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

' 41'

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

'41 '

Со строками все наоборот: по умолчанию они прижимаются к левому краю, но с помощью спецификатора '>' их можно выравнивать вправо.

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

'dog '

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

' dog'

Другие управляющие символы можно использовать для лпределения знака и точности чисел с плавающей запятой; например, {: .4f} указывает, что после десятичной точки в числе с плавающей запятой должно отображаться четыре цифры.

>>> import math

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

'3.1416'

Если вы укажете «%» в спецификации формата, то значение будет представлено в процентах без нет необходимости умножать на 100.

>>> count, total = 3205, 9375

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

'accuracy for 9375 words: 34.1867%'

Важное применение строк форматирования - это табуляция данных. Напомним, что в [1](http://www.nltk.org/book/ch02.html" \l "sec-extracting-text-from-corpora) мы видели данные, которые были образованы из условного распределения частот. Выполним табуляцию самостоятельно, полностью контролируя заголовки и ширину столбцов, как показано в Примере 3.11. Обратите внимание на четкое разделение между действиями по обработке языка и табулированием результатов.

def tabulate(cfdist, words, categories):

print('{:16}'.format('Category'), end=' ') # column headings

for word in words:

print('{:>6}'.format(word), end=' ')

print()

for category in categories:

print('{:16}'.format(category), end=' ') # row heading

for word in words: # for each word

print('{:6}'.format(cfdist[category][word]), end=' ') # 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

**[Пример 3.11 (code\_modal\_tabulate.py)](http://www.nltk.org/book/pylisting/code_modal_tabulate.py)**: Частота модальных глаголов в различных секциях корпуса Brown

Напомним, что в листинге 3.6 мы использовали строку формата '{:{width}}' и присваивали значение параметру width в функции format(). Это позволяет нам указать ширину поля с использованием переменной.

>>> '{:{width}}' % ("Monty Python", width=15)

'Monty Python '

Мы можем использовать это для автоматической настройки ширины столбца, достаточной для размещения всех слов, используя width = max(len(w) for w in words).

### 3.9.4 Запись результатов в файл

Мы видели, как читать текст из файлов ([3.1](http://www.nltk.org/book/ch03.html" \l "sec-accessing-text)). Часто полезно записывать результат в файлы. Следующий код открывает файл output.txt для записи и сохраняет результат программы в файл.

>>> 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)

Когда мы пишем нетекстовую информацию в файл, мы должны сначала преобразовать его в строку. Это преобразование можно сделать, используя форматирующие строки, как мы видели выше. Запишем в файл общее количество слов:

>>> len(words)

2789

>>> str(len(words))

'2789'

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

**Предостережение**! Нужно избегать использования имен файлов с пробелами, например, output file.txt, или файлов, имена которых различаются только регистром, например, Output.txt и output.TXT.

### 3.9.5 Перенос текста

Когда результат нашей программы является текстовым, его нужно выводить с переносом для удобного восприятия. Рассмотрим следующий результат, где строка переполняется и, поэтому, приходится использовать сложный оператор print:

>>> saying = ['After', 'all', 'is', 'said', 'and', 'done', ',',

... 'more', 'is', 'said', 'than', 'done', '.']

>>> for word in saying:

... print(word, '(' + str(len(word)) + '),', end=' ')

After (5), all (3), is (2), said (4), and (3), done (4), , (1), more (4), is (2), said (4), than (4), done (4), . (1),

Перенос текста в строке можно организовать с помощью модуля textwrap. Для максимальной ясности каждый этап действий будет представлен в отдельной строке:

>>> 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),

Заметим, что между словом more и последующим номером вставлен переход на новую строку. Если мы хотим этого избежать, можно переопределить строку форматирования так, чтобы она не содержала пробелов, например '%s\_(%d),'. Теперь, вместо печати значения wrapped, мы можем вывести wrapped.replace('\_', ' ').

## 3.10 Summary

In this book we view a text as a list of words. A "raw text" is a potentially long string containing words and whitespace formatting, and is how we typically store and visualize a text.

A string is specified in Python using single or double quotes: 'Monty Python', "Monty Python".

The characters of a string are accessed using indexes, counting from zero: 'Monty Python'[0] gives the value M. The length of a string is found using len().

Substrings are accessed using slice notation: 'Monty Python'[1:5] gives the value onty. If the start index is omitted, the substring begins at the start of the string; if the end index is omitted, the slice continues to the end of the string.

Strings can be split into lists: 'Monty Python'.split() gives ['Monty', 'Python']. Lists can be joined into strings: '/'.join(['Monty', 'Python']) gives 'Monty/Python'.

We can read text from a file input.txt using text = open('input.txt').read(). We can read text from urlusing text = request.urlopen(url).read().decode('utf8'). We can iterate over the lines of a text file using for line in open(f).

We can write text to a file by opening the file for writing output\_file = open('output.txt', 'w'), then adding content to the file print("Monty Python", file=output\_file).

Texts found on the web may contain unwanted material (such as headers, footers, markup), that need to be removed before we do any linguistic processing.

Tokenization is the segmentation of a text into basic units — or tokens — such as words and punctuation. Tokenization based on whitespace is inadequate for many applications because it bundles punctuation together with words. NLTK provides an off-the-shelf tokenizer nltk.word\_tokenize().

Lemmatization is a process that maps the various forms of a word (such as *appeared*, *appears*) to the canonical or citation form of the word, also known as the lexeme or lemma (e.g. appear).

Regular expressions are a powerful and flexible method of specifying patterns. Once we have imported the remodule, we can use re.findall() to find all substrings in a string that match a pattern.

If a regular expression string includes a backslash, you should tell Python not to preprocess the string, by using a raw string with an r prefix: r'regexp'.

When backslash is used before certain characters, e.g. \n, this takes on a special meaning (newline character); however, when backslash is used before regular expression wildcards and operators, e.g. \., \|, \$, these characters *lose* their special meaning and are matched literally.

A string formatting expression template % arg\_tuple consists of a format string template that contains conversion specifiers like %-6s and %0.2d.

## 3.11 Further Reading

For more examples of processing words with NLTK, see the tokenization, stemming and corpus HOWTOs at http://nltk.org/howto. Chapters 2 and 3 of [(Jurafsky & Martin, 2008)](http://www.nltk.org/book/bibliography.html" \l "jurafskymartin2008) contain more advanced material on regular expressions and morphology. For more extensive discussion of text processing with Python see [(Mertz, 2003)](http://www.nltk.org/book/bibliography.html" \l "mertz2003tpp). For information about normalizing non-standard words see [(Sproat et al, 2001)](http://www.nltk.org/book/bibliography.html" \l "sproat2001nor)

There are many references for regular expressions, both practical and theoretical. For an introductory tutorial to using regular expressions in Python, see Kuchling's *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/. For a comprehensive and detailed manual in using regular expressions, covering their syntax in most major programming languages, including Python, see [(Friedl, 2002)](http://www.nltk.org/book/bibliography.html" \l "friedl2002mre). Other presentations include Section 2.1 of [(Jurafsky & Martin, 2008)](http://www.nltk.org/book/bibliography.html" \l "jurafskymartin2008), and Chapter 3 of [(Mertz, 2003)](http://www.nltk.org/book/bibliography.html" \l "mertz2003tpp).

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

Ned Batchelder, *Pragmatic Unicode*, http://nedbatchelder.com/text/unipain.html

*Unicode HOWTO*, Python Documentation, http://docs.python.org/3/howto/unicode.html

David Beazley, Mastering Python 3 I/O, http://pyvideo.org/video/289/pycon-2010--mastering-python-3-i-o

Joel Spolsky, The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!), http://www.joelonsoftware.com/articles/Unicode.html

The problem of tokenizing Chinese text is a major focus of SIGHAN, the ACL Special Interest Group on Chinese Language Processing http://sighan.org/. Our method for segmenting English text follows [(Brent, 1995)](http://www.nltk.org/book/bibliography.html" \l "brent1995); this work falls in the area of language acquisition [(Niyogi, 2006)](http://www.nltk.org/book/bibliography.html" \l "niyogi2006).

Collocations are a special case of multiword expressions. A **multiword expression** is a small phrase whose meaning and other properties cannot be predicted from its words alone, e.g. *part of speech* [(Baldwin & Kim, 2010)](http://www.nltk.org/book/bibliography.html" \l "baldwinkim2010).

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

The approach to discovering hyponyms in text using search patterns like *x and other ys* is described by [(Hearst, 1992)](http://www.nltk.org/book/bibliography.html" \l "hearst1992hyp).

## 3.12 Упражнения

☼ Define a string s = 'colorless'. Write a Python statement that changes this to "colourless" using only the slice and concatenation operations.

☼ We can use the slice notation to remove morphological endings on words. For example, 'dogs'[:-1]removes the last character of dogs, leaving dog. Use slice notation to remove the affixes from these words (we've inserted a hyphen to indicate the affix boundary, but omit this from your strings): dish-es, run-ning, nation-ality, un-do, pre-heat.

☼ We saw how we can generate an IndexError by indexing beyond the end of a string. Is it possible to construct an index that goes too far to the left, before the start of the string?

☼ We can specify a "step" size for the slice. The following returns every second character within the slice: monty[6:11:2]. It also works in the reverse direction: monty[10:5:-2] Try these for yourself, then experiment with different step values.

☼ What happens if you ask the interpreter to evaluate monty[::-1]? Explain why this is a reasonable result.

☼ Describe the class of strings matched by the following regular expressions.

[a-zA-Z]+

[A-Z][a-z]\*

p[aeiou]{,2}t

\d+(\.\d+)?

([^aeiou][aeiou][^aeiou])\*

\w+|[^\w\s]+

Test your answers using nltk.re\_show().

☼ Write regular expressions to match the following classes of strings:

A single determiner (assume that *a*, *an*, and *the* are the only determiners).

An arithmetic expression using integers, addition, and multiplication, such as 2\*3+8.

☼ Write a utility function that takes a URL as its argument, and returns the contents of the URL, with all HTML markup removed. Use from urllib import request and then request.urlopen('http://nltk.org/').read().decode('utf8') to access the contents of the URL.

☼ Save some text into a file corpus.txt. Define a function load(f) that reads from the file named in its sole argument, and returns a string containing the text of the file.

Use nltk.regexp\_tokenize() to create a tokenizer that tokenizes the various kinds of punctuation in this text. Use one multi-line regular expression, with inline comments, using the verbose flag (?x).

Use nltk.regexp\_tokenize() to create a tokenizer that tokenizes the following kinds of expression: monetary amounts; dates; names of people and organizations.

☼ Rewrite the following loop as a list comprehension:

>>> sent = ['The', 'dog', 'gave', 'John', 'the', 'newspaper']

>>> result = []

>>> for word in sent:

... word\_len = (word, len(word))

... result.append(word\_len)

>>> result

[('The', 3), ('dog', 3), ('gave', 4), ('John', 4), ('the', 3), ('newspaper', 9)]

☼ Define a string raw containing a sentence of your own choosing. Now, split raw on some character other than space, such as 's'.

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

☼ What is the difference between calling split on a string with no argument or with ' ' as the argument, e.g. sent.split() versus sent.split(' ')? What happens when the string being split contains tab characters, consecutive space characters, or a sequence of tabs and spaces? (In IDLE you will need to use '\t' to enter a tab character.)

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

☼ Explore the difference between strings and integers by typing the following at a Python prompt: "3" \* 7and 3 \* 7. Try converting between strings and integers using int("3") and str(3).

☼ Use a text editor to create a file called prog.py containing the single line monty = 'Monty Python'. Next, start up a new session with the Python interpreter, and enter the expression monty at the prompt. You will get an error from the interpreter. Now, try the following (note that you have to leave off the .py part of the filename):

>>> from prog import monty

>>> monty

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

☼ What happens when the formatting strings %6s and %-6s are used to display strings that are longer than six characters?

◑ Прочитайте некоторый текст из корпуса, токенизируйте его и напечатайте список всех *wh*-слов текста *who*, *which*, *what* и т.д.) Выведите их в алфавитном порядке. Будут ли в этом списке повторяющиеся элементы, отличающиеся только величиной букв или по другим причинам?

◑ Создайте файл, состоящий из слов и их частот. В каждой строке – слово, пробел и количество вхождений слова, например fuzzy 53. Прочитайте этот файл в список Python, используя open(filename).readlines(). Затем с помощью split() разбейте каждую строку на две части, преобразовав строку с количеством вхождений в целое число с помощью int(). Результат – список в виде [['fuzzy', 53], ...].

◑ Напишите код для доступа к web-странице с прогнозом погоды и извлеките из него информацию о наиборльшей температуре на сегодня.

◑ Напишите функцию unknown(), которая принимает параметр - URL и возвращает список неизвестных слов на этой странице, то есть слов, отсутствующих в Words Corpus (nltk.corpus.words). Попытайтесь вручную классифицировать эти слова и проанализируйте Ваши результаты.

◑ Проанализируйте результаты обработки URL http://news.bbc.co.uk/ с помощью описанных ранее средств регулярных выражений. Вы найдете много нетекстовой информации, например, команды Javascript. Также обнаружатся нарушения в концовках предложений. Модифицируйте регулярные выражения для улучшения результатов.

◑ Можете ли Вы написать регулярное выражение, токенизирующее текст таким образом, что слово *don't* токенизируется на *do* и *n't*? Почему так не работает следующее регулярное выражение: «n't|\w+».

◑ Try to write code to convert text into *hAck3r*, using regular expressions and substitution, where e → 3, i → 1, o → 0, l → |, s → 5, . → 5w33t!, ate → 8. Normalize the text to lowercase before converting it. Add more substitutions of your own. Now try to map s to two different values: $ for word-initial s, and 5 for word-internal s.

◑ *Pig Latin* – это простая трансформация английского текста: каждое слово преобразуется следующим образом – одна или несколько соседних согласных переносятся из начала слова в конец, а затем в конце слова добаляется *ay*, например *string*→*ingstray*, *idle*→*idleay*. (http://en.wikipedia.org/wiki/Pig\_Latin)

Напишите фунцию, преобразующую слово в Pig Latin.

Напишите код, преобразующий таким образом весь текст.

Модифицируйте код, чтобы не разрывать qu (quiet становится ietquay), и определять, где y используется как согласная (yellow).

◑ Download some text from a language that has vowel harmony (e.g. Hungarian), extract the vowel sequences of words, and create a vowel bigram table.◑ Python's random module includes a function choice() which randomly chooses an item from a sequence, e.g. choice("aehh ") will produce one of four possible characters, with the letter h being twice as frequent as the others. Write a generator expression that produces a sequence of 500 randomly chosen letters drawn from the string "aehh ", and put this expression inside a call to the ''.join() function, to concatenate them into one long string. You should get a result that looks like uncontrolled sneezing or maniacal laughter: he haha ee heheeh eha. Use split() and join() again to normalize the whitespace in this string.

◑ Consider the numeric expressions in the following sentence from the MedLine Corpus: *The corresponding free cortisol fractions in these sera were 4.53 +/- 0.15% and 8.16 +/- 0.23%, respectively.* Should we say that the numeric expression *4.53 +/- 0.15%* is three words? Or should we say that it's a single compound word? Or should we say that it is actually *nine* words, since it's read "four point five three, plus or minus zero point fifteen percent"? Or should we say that it's not a "real" word at all, since it wouldn't appear in any dictionary? Discuss these different possibilities. Can you think of application domains that motivate at least two of these answers?

◑ Меры читабельности используются, чтобы оценить сложность прочтения текста . Это используется при выборе текста при изучении языка. Определим μw как среднее количество букв в слове, а μs – как среднее количество слов в предложении для данного текста. Индекс Automated Readability Index (ARI) определяется по формуле: 4.71μw + 0.5μs - 21.43. Вычислите показатель ARI для различных категорий Brown Corpus, включая секцию f (lore) и j (learned). Используйте nltk.corpus.brown.words() и nltk.corpus.brown.sents().

◑ Используйте Porter Stemmer для нормализации некоторого токенизированного текста, используя стеммер для каждого слова. Проделайте то же с Lancaster Stemmer и проанализируйте различия в результатах.

◑ Определите переменную saying, содержащую список ['After', 'all', 'is', 'said', 'and', 'done', ',', 'more','is', 'said', 'than', 'done', '.']. Обработайте этот список, используя цикл for, сохраняя длину каждого слова в новом списке lengths.

◑ Define a variable silly to contain the string: 'newly formed bland ideas are inexpressible in an infuriating way'. (This happens to be the legitimate interpretation that bilingual English-Spanish speakers can assign to Chomsky's famous nonsense phrase, *colorless green ideas sleep furiously* according to Wikipedia). Now write code to perform the following tasks:

Split silly into a list of strings, one per word, using Python's split() operation, and save this to a variable called bland.

Extract the second letter of each word in silly and join them into a string, to get 'eoldrnnnna'.

Combine the words in bland back into a single string, using join(). Make sure the words in the resulting string are separated with whitespace.

Print the words of silly in alphabetical order, one per line.

◑ The index() function can be used to look up items in sequences. For example, 'inexpressible'.index('e')tells us the index of the first position of the letter e.

What happens when you look up a substring, e.g. 'inexpressible'.index('re')?

Define a variable words containing a list of words. Now use words.index() to look up the position of an individual word.

Define a variable silly as in the exercise above. Use the index() function in combination with list slicing to build a list phrase consisting of all the words up to (but not including) in in silly.

◑ Напишите код, который преобразует прилагательные национальности (*Canadian*, *Australian*) в соответствующие существительные (*Canada*, *Australia*) (см. http://en.wikipedia.org/wiki/List\_of\_adjectival\_forms\_of\_place\_names).

◑ Проанализируйте содержимое LanguageLog блогов на наличие фраз вида *as best as p can* и *as best p can*, где *p* - местоимение. Исследуйте результаты на некоторм корпусе с помощью findall() в токенизированном тексте (http://itre.cis.upenn.edu/~myl/languagelog/archives/002733.html)

◑ Study the  *lolcat*  version of the book of Genesis, accessible as nltk.corpus.genesis.words('lolcat.txt'), and the rules for converting text into *lolspeak* athttp://www.lolcatbible.com/index.php?title=How\_to\_speak\_lolcat. Define regular expressions to convert English words into corresponding lolspeak words.

◑ Рассмотрите функцию re.sub() для реализации замены строк в контекте регулярных выражений (используйте help(re.sub) и другие источники). Используйте re.sub в коде для удаления HTML-тегов из HTML-файла и нормализации пробелов.

★ An interesting challenge for tokenization is words that have been split across a line-break. E.g. if *long-term*is split, then we have the string long-\nterm.

Write a regular expression that identifies words that are hyphenated at a line-break. The expression will need to include the \n character.

Use re.sub() to remove the \n character from these words.

How might you identify words that should not remain hyphenated once the newline is removed, e.g. 'encyclo-\npedia'?x

★ Read the Wikipedia entry on *Soundex*. Implement this algorithm in Python.

★ Obtain raw texts from two or more genres and compute their respective reading difficulty scores as in the earlier exercise on reading difficulty. E.g. compare ABC Rural News and ABC Science News (nltk.corpus.abc). Use Punkt to perform sentence segmentation.

★ Rewrite the following nested loop as a nested list comprehension:

>>> words = ['attribution', 'confabulation', 'elocution',

... 'sequoia', 'tenacious', 'unidirectional']

>>> vsequences = set()

>>> for word in words:

... vowels = []

... for char in word:

... if char in 'aeiou':

... vowels.append(char)

... vsequences.add(''.join(vowels))

>>> sorted(vsequences)

['aiuio', 'eaiou', 'eouio', 'euoia', 'oauaio', 'uiieioa']

★ Use WordNet to create a semantic index for a text collection. Extend the concordance search program in [3.6](http://www.nltk.org/book/ch03.html" \l "code-stemmer-indexing), indexing each word using the offset of its first synset, e.g. wn.synsets('dog')[0].offset (and optionally the offset of some of its ancestors in the hypernym hierarchy).

★ With the help of a multilingual corpus such as the Universal Declaration of Human Rights Corpus (nltk.corpus.udhr), and NLTK's frequency distribution and rank correlation functionality (nltk.FreqDist, nltk.spearman\_correlation), develop a system that guesses the language of a previously unseen text. For simplicity, work with a single character encoding and just a few languages.

★ Write a program that processes a text and discovers cases where a word has been used with a novel sense. For each word, compute the WordNet similarity between all synsets of the word and all synsets of the words in its context. (Note that this is a crude approach; doing it well is a difficult, open research problem.)

★ Read the article on normalization of non-standard words [(Sproat et al, 2001)](http://www.nltk.org/book/bibliography.html" \l "sproat2001nor), and implement a similar system for text normalization.