



Современный скрапинг веб-сайтов с помощью Python

АВТОМАТИЗАЦИЯ СБОРА ДАННЫХ С ИНТЕРНЕТ-САЙТОВ ПРИ ПОМОЩИ ЯЗЫКА PYTHON

Райан Митчелл
Современный скрапинг веб-сайтов с помощью Python. 2-е межд.
издание



2021

Научный редактор *С. Бычковский*

Переводчик *Е. Сандицкая*

Литературный редактор *Н. Хлебина*

Художник *В. Мостипан*

Корректоры *Н. Гринчик, Е. Павлович, Е. Рафалюк-Бузовская*

Райан Митчелл

Современный скрапинг веб-сайтов с помощью Python. 2-е
межд. издание . — СПб.: Питер, 2021.

ISBN 978-5-4461-1693-5

© [ООО Издательство "Питер"](#), 2021

*Все права защищены. Никакая часть данной книги не может
быть воспроизведена в какой бы то ни было форме без
письменного разрешения владельцев авторских прав.*

Введение

Если человек не слишком хорошо знаком с программированием, оно ему может показаться чем-то вроде волшебства. Но если программирование — волшебство, то *веб-скрапинг* — это очень сильное колдунство: написав простую автоматизированную программу, можно отправлять запросы на веб-серверы, запрашивать с них данные, а затем анализировать их и извлекать необходимую информацию.

Работая инженером-программистом, я обнаружила, что веб-скрапинг — одна из немногих областей программирования, восхищающая как разработчиков, так и обычных людей. Умение легко написать простой бот, который бы собирал данные и передавал их через терминал или сохранял в базе данных, не перестает повергать в некий трепет от осознания своих возможностей, независимо от того, сколько раз вам приходилось делать это раньше.

К сожалению, общаясь с другими программистами на тему веб-скрапинга, я обнаружила, что не все хорошо понимают суть метода. Одни считают его не вполне законным (и они ошибаются), другие не умеют обрабатывать страницы, содержащие много кода JavaScript или требующие регистрации. Многие не знают, как начать крупный проект по скрапину или даже где искать нужные данные. Книга призвана ответить на многие из этих вопросов, развеять ошибочные представления о веб-скрапинге, а также предоставить исчерпывающее руководство по решению его наиболее распространенных задач.

Веб-скрапинг — обширная и быстро развивающаяся область, поэтому я постаралась представить здесь не только общие принципы, но и конкретные примеры, охватывающие

практически все способы сбора данных, с которыми вы, вероятно, столкнетесь. В книге приводятся примеры кода, демонстрирующие эти принципы и позволяющие проверить их на практике. Сами примеры можно использовать и изменять как с указанием авторства, так и без него (хотя благодарности всегда приветствуются). Все примеры кода доступны на GitHub (<http://www.pythonscraping.com/code/>), где их можно просмотреть и скачать.

Что такое веб-скрапинг

Автоматизированный сбор данных в Интернете почти так же стар, как и сам Интернет. Несмотря на то что термин «*веб-скрапинг*» не является новым, еще несколько лет назад эту методику чаще называли *анализом интерфейсных данных*, *интеллектуальным анализом данных*, *сбором веб-данных* и т.п. Похоже, что наконец-то все пришли к единому мнению и предпочли называть это *веб-скрапингом*, поэтому я буду использовать данный термин на протяжении всей книги, хотя специализированные программы, которые просматривают несколько веб-страниц, я буду называть *веб-краулерами*, а программы, предназначенные для собственно веб-скрапинга, — *ботами*.

Теоретически веб-скрапинг — это сбор данных с использованием любых средств, за исключением программ, взаимодействующих с API. Обычно для этого пишут автоматизированную программу, которая обращается к веб-серверу, запрашивает данные (как правило, в формате HTML или в других форматах веб-страниц), а затем анализирует эти данные и извлекает оттуда полезную информацию.

На практике веб-скрапинг включает в себя широкий спектр методов и технологий программирования, таких как анализ

данных, синтаксический анализ естественных языков и информационная безопасность. Именно потому, что эта область столь широка, в части I данной книги будут рассмотрены фундаментальные основы веб-скрапинга и веб-краулинга, а в части II — более углубленные темы. Я рекомендую внимательно изучить первую часть и погружаться в более специализированные разделы второй части по мере необходимости.

Почему это называется веб-скрапингом

Получать доступ к Интернету только через браузер — значит упускать массу возможностей. Браузеры (кроме прочего) удобны для выполнения скриптов JavaScript, вывода изображений и представления объектов в понятной для человека форме, однако веб-скраперы гораздо лучше справляются с быстрым сбором и обработкой больших объемов данных. Вместо того чтобы просматривать страницу за страницей на экране монитора, можно читать сразу целые базы данных, в которых хранятся тысячи и даже миллионы страниц.

Кроме того, веб-скраперы позволяют заглядывать в места, недоступные обычным поисковым системам. Так, при поиске в Google «самых дешевых рейсов в Бостон» вы получите кучу ссылок на рекламные объявления и популярные сайты поиска авиабилетов. Google знает только то, что сообщается на страницах оглавлений этих сайтов, а вовсе не точные результаты различных запросов, введенных в приложение поиска рейсов. Однако правильно построенный веб-скрапер способен создать график изменения стоимости перелета в Бостон во времени на разных сайтах и определить даты, когда можно купить самый выгодный билет.

Вы спросите: «Разве API не создаются специально для сбора данных?» (О том, что такое API, см. в главе 12.) Действительно, возможности API бывают просто фантастическими, если удастся найти тот из них, который соответствует вашим целям. API предназначены для построения удобного потока хорошо отформатированных данных из одной компьютерной программы в другую. Для многих типов данных, которые вы, возможно, захотите использовать, существуют готовые API — например, для постов Twitter или страниц «Википедии». Как правило, если существует подходящий API, то предпочтительнее использовать его вместо создания бота для получения тех же данных. Однако нужного API может не оказаться, или же этот API может не соответствовать вашим целям по нескольким причинам:

- вам необходимо собирать относительно небольшие, ограниченные наборы данных с большого количества сайтов, у которых нет единого API;
- нужных данных сравнительно мало или они необычны и разработчик посчитал неоправданным создание для них специального API;
- источник не обладает инфраструктурой или техническими возможностями для создания API;
- это ценные и/или защищенные данные, не предназначенные для широкого распространения.

Даже если API *действительно* существует, его возможности по объему и скорости обрабатываемых запросов, а также по типам или формату предоставляемых данных могут оказаться недостаточными для ваших целей.

Именно в таких случаях в дело вступает веб-скрапинг. За редким исключением, если данные доступны в браузере, то доступны и через скрипт Python. Данные, доступные в скрипте, можно сохранить в базе данных. А с сохраненными данными можно делать практически все что угодно.

Разумеется, у доступа к почти любым данным есть множество чрезвычайно полезных вариантов применения: системы прогнозирования рынка, машинного перевода и даже медицинской диагностики получили огромное распространение благодаря возможности извлекать и анализировать данные с новостных сайтов, из переведенных текстов и с форумов по вопросам здоровья соответственно.

Даже в мире искусства веб-скрапинг расширяет возможности для творчества. В 2006 году проект We Feel Fine («Мы прекрасно себя чувствуем») (<http://wefeelfine.org/>) Джонатана Харриса (Jonathan Harris) и Сэпа Камвара (Sep Kamvar) собрал из нескольких англоязычных блогов фразы, начинающиеся со слов I feel или I am feeling («я чувствую, я ощущаю»). В итоге получилась визуализация большого количества данных, описывающих то, что чувствовал мир день за днем, минуту за минутой.

Независимо от того, чем вы занимаетесь, веб-скрапинг почти всегда дает возможность сделать это более эффективно, повысить продуктивность или даже перейти на совершенно новый уровень.

Об этой книге

Данная книга — не только начальное пособие по веб-скрапингу, но и всеобъемлющее руководство по сбору, преобразованию и использованию данных из несовместимых источников. Однако, несмотря на то что здесь применяется

язык программирования Python и изложены многие его основы, книгу не следует использовать для знакомства с этим языком.

Если вы вообще не знаете Python, то вам может быть сложно читать данную книгу. Пожалуйста, не используйте ее в качестве учебника по основам Python. Учитывая эту проблему, я постаралась представить все концепции и примеры кода с ориентиром на начальный и средний уровень программирования на Python, чтобы они были понятны широкому кругу читателей. Поэтому иногда здесь приводятся пояснения более сложных аспектов программирования на Python и общих вопросов информатики.

Если вы ищете более подробный учебник по Python, то рекомендую *Introducing Python* Билла Любановича (Bill Lubanovic)¹ — это хорошее, хоть и довольно объемное руководство. Тем, у кого не хватит на него времени, советую посмотреть видеоуроки *Introduction to Python* Джессики Маккеллар (Jessica McKellar) (издательство O'Reilly) (<http://oreil.ly/2HOqSNM>) — это отличный ресурс. Мне также понравилась книга *Think Python* моего бывшего профессора Аллена Дауни (Allen Downey) (издательство O'Reilly) (<http://oreil.ly/2fjbT2F>). Она особенно хороша для новичков в программировании. Это учебник не только по языку Python, но и по информатике вообще, а также по общим концепциям разработки ПО.

Технические книги часто посвящены какому-то одному языку или технологии. Однако веб-скрапинг — весьма разносторонняя тема, в которой задействованы базы данных, веб-серверы, HTTP, HTML, интернет-безопасность, обработка изображений, анализ данных и другие инструменты. В данной книге я постараюсь охватить все эти и другие темы с точки зрения сбора данных. Это не значит, что здесь они будут

раскрыты полностью, однако я намерена раскрыть их достаточно подробно, чтобы вы начали писать веб-скраперы!

В части I подробно рассматриваются веб-скрапинг и веб-краулинг. Особое внимание уделяется нескольким полезным библиотекам. Часть I вполне может служить подробным справочником по этим библиотекам и методикам (за некоторыми исключениями; по ним будут предоставлены дополнительные ссылки). Приемы, описанные в первой части книги, полезны всем, кто пишет веб-скраперы независимо от их конкретной цели и области приложения.

В части II раскрыты дополнительные темы, также полезные при написании веб-скраперов, но не всегда и не любых. К сожалению, данные темы слишком широки и их нельзя уместить в одной главе. Поэтому я буду часто ссылаться на другие ресурсы, где вы найдете дополнительную информацию.

Структура этой книги позволяет легко переходить от одной главы к другой, чтобы найти описание только веб-скрапинга или другую нужную вам информацию. Если концепция или фрагмент кода основывается на чем-то, о чем говорилось в предыдущей главе, то я явно ссылаюсь на раздел, в котором это было рассмотрено.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные слова.

Моноширинный шрифт

Используется для листингов программ, а также внутри абзацев, чтобы обратиться к элементам программы вроде переменных, функций, баз данных, типов данных, переменных

среды, инструкций и ключевых слов, имен и расширений файлов.

Моноширинный жирный шрифт

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

Моноширинный курсивный шрифт

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.



Этот рисунок указывает на совет или предложение.



Такой рисунок указывает на общее замечание.



Этот рисунок указывает на предупреждение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т.д.) можно скачать по адресу <https://github.com/REMitchell/python-scraping>.

Эта книга призвана помочь вам выполнять свою работу. Если какой-нибудь из приведенных примеров будет полезен для вас, то вы можете использовать его в своих программах и документации. Вам не нужно обращаться к нам за разрешением, если только вы не воспроизводите значительную часть кода. Так, для написания программы, в которой задействованы несколько фрагментов кода из данной книги, не требуется разрешения. А вот для продажи или распространения компакт-дисков с примерами из книг O'Reilly — требуется. Для ответа на вопрос с помощью этой книги и примера кода разрешение не нужно. Однако на включение значительного количества примеров кода из книги в документацию вашего продукта требуется разрешение.

Мы ценим ссылки на эту книгу, но не требуем их. Как правило, такая ссылка включает в себя название, автора, издателя и ISBN. Например: «Митчелл Райан. Современный скрапинг веб-сайтов с помощью Python. — СПб.: Питер, 2021. — 978-5-4461-1693-5».

Если вы считаете, что использование вами примеров кода выходит за рамки правомерного применения или предоставленных выше разрешений, то обратитесь к нам по адресу **permissions@oreilly.com**.

К сожалению, бумажные книги трудно поддерживать в актуальном состоянии. В случае веб-скрапинга это создает дополнительную проблему, так как многие библиотеки и сайты, на которые ссылается данная книга и от которых часто зависит код, изменяются, из-за чего примеры кода могут перестать работать или приводить к неожиданным результатам. Если вы захотите выполнить примеры кода, то не

копируйте их непосредственно из книги, а скачайте из репозитория GitHub. Мы — и я, и читатели этой книги, которые решили внести свой вклад и поделиться своими примерами (включая, возможно, вас!), — постараемся поддерживать хранилище в актуальном состоянии, вовремя внося необходимые изменения и примечания.

Помимо примеров кода, в этой книге часто приводятся команды терминала, демонстрирующие установку и запуск программного обеспечения. Как правило, эти команды предназначены для операционных систем на основе Linux, но большинство из них применимы и в Windows с правильно настроенной средой Python и установленным `pip`. В отношении тех случаев, когда это не так, я предоставила инструкции для всех основных операционных систем или внешние ссылки для пользователей Windows, чтобы облегчить задачу.

Благодарности

Лучшие продукты часто появляются благодаря многочисленным отзывам пользователей. Эта книга тоже никогда не появилась бы в сколько-нибудь полезном виде без помощи многих соавторов, руководителей и редакторов. Спасибо сотрудникам O'Reilly за их удивительную поддержку этого необычного начинания; моим друзьям и семье, которые давали советы и мирились с импровизированными чтениями; а также моим коллегам из HedgeServ, которым я, вероятно, теперь должна много часов работы.

В частности, спасибо Элисон Макдональд (Allyson MacDonald), Брайану Андерсону (Brian Anderson), Мигелю Гринбергу (Miguel Grinberg) и Эрику Ванвику (Eric VanWyk) за отзывы и советы — иногда резкие, но справедливые. Довольно

много разделов и примеров кода появились в книге именно в результате их вдохновляющих предложений.

Благодарю Йела Шпехта (Yale Specht) за его безграничное терпение в течение четырех лет и двух изданий, за его поддержку, которая помогала мне продолжать этот проект, и обратную связь по стилистике в процессе написания. Без него я написала бы книгу вдвое быстрее, но она вряд ли была бы настолько полезной.

Наконец, спасибо Джиму Уолдо (Jim Waldo) — именно с него все это началось много лет назад, когда он прислал впечатлительной девочке-подростку диск с Linux и книгу *The Art and Science of C*.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу **comp@piter.com** (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

¹ Любанович Б. Простой Python. Современный стиль программирования. 2-е изд. — СПб.: Питер, 2021.

Часть I. Разработка веб-скраперов

В первой части этой книги основное внимание будет уделено базовым механизмам веб-скрапинга: как на Python построить запрос к веб-серверу, выполнить базовую обработку полученного ответа и начать взаимодействие с сайтом в автоматическом режиме. В итоге вы сможете легко путешествовать по Интернету, создавая скраперы, способные переходить от одного домена к другому, собирать и сохранять информацию для последующего использования.

Честно говоря, веб-скрапинг — фантастическая отрасль: вложенные в нее относительно небольшие начальные инвестиции окупятся сторицей. Примерно 90 % проектов веб-скрапинга, которые вам встретятся, будут опираться на методы, описанные в следующих шести главах. Эта часть покажет, как обычные (хотя и технически подкованные) люди представляют себе работу веб-скраперов:

- извлечение HTML-данных из имени домена;
- анализ этих данных для получения требуемой информации;
- сохранение этой информации;
- возможен переход на другую страницу, чтобы повторить процедуру.

Прочитав эту часть, вы получите прочную основу, которая позволит вам перейти к более сложным проектам, описанным в части II. Не думайте, что первая половина книги менее важна, чем вторая, и более сложные проекты описаны во второй половине. При написании веб-скраперов вам придется

каждый день использовать почти всю информацию,
изложенную в первой половине данной книги!

Глава 1. Ваш первый веб-скрапер

С первых шагов веб-скрапинга вы сразу начнете ценить все те мелкие услуги, которые нам оказывают браузеры. Без HTML-форматирования, стилей CSS, скриптов JavaScript и рендеринга изображений Интернет с непривычки может показаться слегка пугающим. Но в этой и следующей главах мы посмотрим, как форматировать и интерпретировать данные, не прибегая к помощи браузера.

В этой главе мы начнем с основ отправки на веб-сервер GET-запросов — запросов на выборку или получение содержимого заданной веб-страницы, чтения полученного с нее HTML-кода и выполнения ряда простых операций по извлечению данных, чтобы выделить оттуда контент, который вы ищете.

Установка соединения

Если вам прежде не приходилось много работать с сетями или заниматься вопросами сетевой безопасности, то механика работы Интернета может показаться несколько загадочной. Вы же не хотите каждый раз задумываться о том, что именно делает сеть, когда вы открываете браузер и заходите на сайт **<http://google.com>**. Впрочем, в наше время это и не нужно. Компьютерные интерфейсы стали настолько совершенными, что большинство людей, пользующихся Интернетом, не имеют ни малейшего представления о том, как работает Сеть, и это, по-моему, здорово.

Однако веб-скрапинг требует сбросить покров с этого интерфейса — на уровне не только браузера (то, как он

интерпретирует весь HTML-, CSS- и JavaScript-код), но иногда и сетевого подключения.

Чтобы дать вам представление об инфраструктуре, необходимой для получения информации в браузере, рассмотрим следующий пример. У Алисы есть веб-сервер, а у Боба — ПК, с которого он хочет подключиться к серверу Алисы. Когда одна машина собирается пообщаться с другой, происходит примерно следующий обмен данными.

1. Компьютер Боба посылает поток битов — единиц и нулей, которым соответствует высокое и низкое напряжение в кабеле. Из этих битов складывается некая информация, делимая на заголовок и тело. В заголовке содержится MAC-адрес ближайшего локального маршрутизатора и IP-адрес конечной точки — сервера Алисы. В тело включен запрос Боба к серверному приложению Алисы.
2. Локальный маршрутизатор Боба получает все эти нули и единицы и интерпретирует их как пакет, который передается с MAC-адреса Боба на IP-адрес Алисы. Маршрутизатор Боба ставит свой IP-адрес на пакете в графе «отправитель» и отправляет пакет через Интернет.
3. Пакет Боба проходит через несколько промежуточных серверов, которые по соответствующим кабелям передают пакет на сервер Алисы.
4. Сервер Алисы получает пакет по своему IP-адресу.
5. Сервер Алисы считывает из заголовка пакета номер порта-приемника и передает пакет соответствующему приложению веб-сервера. (Портом-приемником пакета в случае веб-приложений почти всегда является порт номер 80; это словно номер квартиры в адресе для пакетных

данных, тогда как IP-адрес аналогичен названию улицы и номеру дома.)

6. Серверное приложение получает поток данных от серверного процессора. Эти данные содержат примерно такую информацию:

- вид запроса: GET;
- имя запрошенного файла: `index.html`.

7. Веб-сервер находит соответствующий HTML-файл, помещает его в новый пакет, созданный для Боба, и отправляет этот пакет на локальный маршрутизатор, откуда он уже знакомым нам способом передается на компьютер Боба.

Вуаля! Так работает Интернет.

В чем же при этом обмене данными участвует браузер? Ответ: ни в чем. В действительности браузеры — сравнительно недавнее изобретение в истории Интернета: Nexus появился всего в 1990 году.

Конечно, браузер — полезное приложение, которое создает эти информационные пакеты, сообщает операционной системе об их отправке и интерпретирует данные, превращая их в красивые картинки, звуки, видео и текст. Однако браузер — это всего лишь код, который можно разбить на части, выделить основные компоненты, переписать, использовать повторно и приспособить для выполнения чего угодно. Браузер может дать команду процессору отправлять данные в приложение, поддерживающее беспроводной (или проводной) сетевой интерфейс, но то же самое можно сделать и на Python с помощью всего трех строк кода:

```
from urllib.request import urlopen
html =
urlopen('http://pythonscraping.com/pages/page1.
html')
print(html.read())
```

Для выполнения этого кода можно использовать оболочку iPython, которая размещена в репозитории GitHub для главы 1 (https://github.com/REMitchell/python-scraping/blob/master/Chapter01_BeginningToScrape.ipynb), или же сохранить код на компьютере в файле `scrapetest.py` и запустить его в окне терминала с помощью следующей команды:

```
$ python scrapetest.py
```

Обратите внимание: если на вашем компьютере, кроме Python 3.x, также установлен Python 2.x, и вы используете обе версии Python параллельно, то может потребоваться явно вызвать Python 3.x, выполнив команду следующим образом:

```
$ python3 scrapetest.py
```

По этой команде выводится полный HTML-код страницы `page1`, расположенной по адресу **`http://pythonscraping.com/pages/page1.html`**. Точнее, выводится HTML-файл `page1.html`, размещенный в каталоге **<корневой веб-каталог>/pages** на сервере с доменным именем **`http://pythonscraping.com`**.

Почему так важно представлять себе эти адреса как «файлы», а не как «страницы»? Большинство современных веб-страниц связано с множеством файлов ресурсов. Ими могут быть файлы изображений, скриптов JavaScript, стилей CSS и любой другой контент, на который ссылается запрашиваемая

страница. Например, встретив тег `<imgsrc="cuteKitten.jpg">`, браузер знает: чтобы сгенерировать страницу для пользователя, нужно сделать еще один запрос к серверу и получить данные из файла `cuteKitten.jpg`.

Разумеется, у нашего скрипта на Python нет логики, позволяющей вернуться и запросить несколько файлов (пока что); он читает только тот HTML-файл, который мы запросили напрямую:

```
from urllib.request import urlopen
```

Эта строка делает именно то, что кажется на первый взгляд: находит модуль Python для запросов (в библиотеке `urllib`) и импортирует оттуда одну функцию — `urlopen`.

Библиотека `urllib` — это стандартная библиотека Python (другими словами, для запуска данного примера ничего не нужно устанавливать дополнительно), в которой содержатся функции для запроса данных через Интернет, обработки файлов `cookie` и даже изменения метаданных, таких как заголовки и пользовательский программный агент. Мы будем активно применять `urllib` в данной книге, так что я рекомендую вам прочитать раздел документации Python, касающийся этой библиотеки (<https://docs.python.org/3/library/urllib.html>).

Функция `urlopen` открывает удаленный объект по сети и читает его. Поскольку это практически универсальная функция (она одинаково легко читает HTML-файлы, файлы изображений и другие файловые потоки), мы будем довольно часто использовать ее в данной книге.

Знакомство с BeautifulSoup

*Прекрасный суп в столовой
ждет.*

Из миски жирный пар идет.

Не любит супа тот, кто глуп!

Прекрасный суп, вечерний суп!

Прекрасный суп, вечерний суп!

Алиса в Стране чудес. Издание 1958 г., пер. А. Оленича-Гнененко

Библиотека BeautifulSoup («Прекрасный суп») названа так в честь одноименного стихотворения Льюиса Кэрролла из книги «Алиса в Стране чудес». В книге это стихотворение поет Фальшивая Черепаха (пародия на популярное викторианское блюдо — фальшивый черепаховый суп, который варят не из черепахи, а из говядины).

Приложение BeautifulSoup стремится найти смысл в бессмысленном: помогает отформатировать и упорядочить «грязные» сетевые данные, исправляя ошибки в HTML-коде и создавая легко обходимые (traversable) объекты Python, являющиеся представлениями структур XML.

Установка BeautifulSoup

Поскольку BeautifulSoup не является стандартной библиотекой Python, ее необходимо установить. Если у вас уже есть опыт установки библиотек Python, то используйте любимый установщик и пропустите этот подраздел, сразу перейдя к следующему — «Работа с BeautifulSoup» на с. 29.

Для тех же, кому еще не приходилось устанавливать библиотеки Python (или кто подзабыл, как это делается), представлен общий подход, который мы будем использовать

для установки библиотек по всей книге, так что впоследствии вы можете обращаться к данному подразделу.

В этой книге мы будем использовать библиотеку BeautifulSoup 4 (также известную как BS4). Подробная инструкция по установке BeautifulSoup 4 размещена на сайте Crummy.com

(<http://www.crummy.com/software/BeautifulSoup/bs4/doc/>); здесь же описан самый простой способ для Linux:

```
$ sudo apt-get install python-bs4
```

И для Mac:

```
$ sudo easy_install pip
```

Эта команда устанавливает менеджер пакетов Python *pip*.

Затем выполните следующую команду для установки библиотеки:

```
$ pip install beautifulsoup4
```

Еще раз подчеркну: если на вашем компьютере установлены версии Python 2.x и 3.x, то лучше явно вызвать python3:

```
$ python3 myScript.py
```

То же самое касается установки пакетов, иначе пакеты могут установиться не для Python 3.x, а в Python 2.x:

```
$ sudo python3 setup.py install
```

При использовании *pip* с целью установки пакета для Python 3.x также можно вызвать *pip3*:

```
$ pip3 install beautifulsoup4
```

Установка пакетов для Windows практически не отличается от данного процесса для Mac и Linux. Скачайте последнюю версию BeautifulSoup 4 со страницы скачивания библиотеки (<http://www.crummy.com/software/BeautifulSoup/#Download>), перейдите в каталог, в котором вы ее распаковали, и выполните следующую команду:

```
> python setup.py install
```

Готово! Теперь компьютер будет распознавать BeautifulSoup как библиотеку Python. Чтобы в этом убедиться, откройте терминал Python и импортируйте ее:

```
$ python  
> from bs4 import BeautifulSoup
```

Импорт должен выполняться без ошибок.

Кроме того, для Windows есть программа-установщик менеджера пакетов pip (<https://pypi.python.org/pypi/setuptools>), с помощью которой можно легко устанавливать пакеты и управлять ими:

```
> pip install beautifulsoup4
```

Хранение библиотек непосредственно в виртуальных окружениях

Если вы собираетесь работать параллельно с несколькими проектами Python, или вам нужен удобный способ связать проекты с их библиотеками, или вы хотите исключить потенциальные конфликты между установленными библиотеками, то можете установить виртуальное окружение Python, где все разделено и легко управляется.

Если библиотека Python устанавливается без виртуального окружения, то она устанавливается *глобально*. Обычно такую установку должен выполнять администратор или пользователь **root**, и тогда библиотека Python станет доступна для всех пользователей и проектов на данном компьютере. К счастью, создать виртуальное окружение легко:

```
$ virtualenv scrapingEnv
```

Эта команда создает новое окружение с именем `scrapingEnv`. Чтобы использовать данное окружение, его необходимо активировать:

```
$ cd scrapingEnv/  
$ source bin/activate
```

После активации окружения его имя появится в командной строке, подсказывая, в каком именно окружении вы работаете. Все устанавливаемые библиотеки и запускаемые скрипты будут находиться только в этом виртуальном окружении.

В окружении `scrapingEnv` можно установить и использовать BeautifulSoup, например, так:

```
(scrapingEnv)ryan$ pip install  
beautifulsoup4  
(scrapingEnv)ryan$ python  
> from bs4 import BeautifulSoup  
>
```

Выйти из окружения можно с помощью команды

deactivate, после чего библиотеки, которые были установлены внутри виртуального окружения, станут недоступными:

```
(scrapingEnv)ryan$ deactivate
ryan$ python
> from bs4 import BeautifulSoup
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'bs4'
```

Хранение всей библиотеки с разделением по проектам позволит легко заархивировать всю папку окружения и отправить ее кому-нибудь. Если на компьютере получателя установлена та же версия Python, что и у вас, то ваш код будет работать в его виртуальном окружении, не требуя дополнительной установки каких-либо библиотек.

Я не буду настаивать, чтобы вы применяли виртуальное окружение при выполнении всех примеров этой книги, однако помните: вы можете воспользоваться им в любой момент, просто заранее его активировав.

Работа с BeautifulSoup

Из всех объектов библиотеки BeautifulSoup чаще всего используется собственно, сам BeautifulSoup. Посмотрим, как он работает, изменив пример, приведенный в начале главы:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
```

```
html =  
urlopen('http://www.pythonscraping.com/pages/page1.html')  
bs = BeautifulSoup(html.read(), 'html.parser')  
print(bs.h1)
```

Результат выглядит так:

```
<h1>An Interesting Title</h1>
```

Обратите внимание: этот код возвращает только первый попавшийся ему на странице экземпляр тега `h1`. По существующему соглашению на странице может быть только один тег `h1`, однако принятые в Интернете соглашения часто нарушаются. Поэтому следует помнить, что таким образом будет получен только первый экземпляр тега и не обязательно тот, который вы ищете.

Как и в предыдущих примерах веб-скрапинга, мы импортируем функцию `urlopen` и вызываем `html.read()`, чтобы получить контент страницы в формате HTML. Помимо текстовой строки, `BeautifulSoup` также может принимать файловый объект, непосредственно возвращаемый функцией `urlopen`. Чтобы получить этот объект, не нужно вызывать функцию `.read()`:

```
bs = BeautifulSoup(html, 'html.parser')
```

Здесь контент HTML-файла преобразуется в объект `BeautifulSoup`, имеющий следующую структуру:

```
html → <html><head>...</head><body>...</body>  
</html>
```

- **head**→ `<head><title>A Useful Page<title></head>`
- **title**→ `<title>A Useful Page</title>`
- **body**→ `<body><h1>An Int...</h1><div>Lorem ip...</div></body>`
- **h1**→ `<h1>An Interesting Title</h1>`
- **div**→ `<div>Lorem Ipsum dolor...</div>`

Обратите внимание: тег `h1`, извлеченный из кода страницы, находится на втором уровне структуры объекта `BeautifulSoup` (`html→body→h1`). Однако, извлекая `h1` из объекта, мы обращаемся к этому тегу напрямую:

```
bs.h1
```

На практике все следующие вызовы функций приведут к одинаковым результатам:

```
bs.html.body.h1
bs.body.h1
bs.html.h1
```

При создании объекта `BeautifulSoup` функции передаются два аргумента:

```
bs = BeautifulSoup(html.read(), 'html.parser')
```

Первый аргумент — это текст в формате HTML, на основе которого строится объект, а второй — синтаксический анализатор, который `BeautifulSoup` будет использовать для построения объекта. В большинстве случаев не имеет значения, какой именно синтаксический анализатор будет применяться.

Анализатор `html.parser` входит в состав Python 3 и не требует дополнительной настройки перед использованием. За редким исключением, в этой книге я буду применять именно его.

Еще один популярный анализатор — `lxml` (<http://lxml.de/parsing.html>). Он устанавливается через `pip`:

```
$ pip3 install lxml
```

Для того чтобы использовать `lxml` в `BeautifulSoup`, нужно изменить имя синтаксического анализатора в знакомой нам строке:

```
bs = BeautifulSoup(html.read(), 'lxml')
```

Преимущество `lxml`, по сравнению с `html.parser`, состоит в том, что `lxml` в целом лучше справляется с «грязным» или искаженным HTML-кодом. Анализатор `lxml` прощает неточности и исправляет такие проблемы, как незакрытые и неправильно вложенные теги, а также отсутствующие теги `head` или `body`. Кроме того, `lxml` работает несколько быстрее, чем `html.parser`, хотя при веб-скрапинге скорость анализатора не всегда является преимуществом, поскольку почти всегда главное узкое место — скорость самого сетевого соединения.

Недостатками анализатора `lxml` является то, что его необходимо специально устанавливать и он зависит от сторонних C-библиотек. Это может вызвать проблемы портируемости; кроме того, `html.parser` проще в использовании.

Еще один популярный синтаксический анализатор HTML называется `html5lib`. Подобно `lxml`, он чрезвычайно лоялен к ошибкам и прилагает еще больше усилий к исправлению

некорректного HTML-кода. Он также имеет внешние зависимости и работает медленнее, чем `lxml` и `html.parser`. Тем не менее выбор `html5lib` может быть оправданным при работе с «грязными» или написанными вручную HTML-страницами.

Чтобы использовать этот анализатор, нужно установить его и передать объекту `BeautifulSoup` строку `html5lib`:

```
bs = BeautifulSoup(html.read(), 'html5lib')
```

Надеюсь, благодаря этой краткой дегустации `BeautifulSoup` вы составили представление о возможностях и удобстве данной библиотеки. В сущности, она позволяет извлечь любую информацию из любого файла в формате HTML (или XML), если содержимое данного файла заключено в идентифицирующий тег или этот тег хотя бы присутствует в принципе. В главе 2 мы подробно рассмотрим более сложные вызовы функций библиотеки, а также регулярные выражения и способы их использования с помощью `BeautifulSoup` для извлечения информации с сайтов.

Надежное соединение и обработка исключений

Сеть — «грязное» место. Данные плохо отформатированы, сайты то и дело «падают», а разработчики страниц постоянно забывают ставить закрывающие теги. Один из самых неприятных моментов, связанных с веб-скрапингом, — уйти спать и оставить работающий скрапер, рассчитывая на завтра иметь все данные в вашей базе, а утром обнаружить, что скрапер столкнулся с ошибкой в каком-то непредсказуемом формате данных и почти сразу прекратил работу, стоило вам отвернуться от экрана. В подобных случаях возникает соблазн проклясть разработчика, который создал тот сайт (и выбрал

странный формат представления данных). Однако на самом деле если кому и стоит дать пинка, то это вам самим, поскольку именно вы не предусмотрели исключение!

Рассмотрим первую строку нашего скрапера, сразу после операторов импорта, и подумаем, как можно обрабатывать любые исключения, которые могли бы здесь возникнуть:

```
html =  
urlopen('http://www.pythonscraping.com/pages/page1.html')
```

Здесь могут случиться две основные неприятности:

- на сервере нет такой страницы (или при ее получении произошла ошибка);
- нет такого сервера.

В первой ситуации будет возвращена ошибка HTTP. Это может быть 404 Page Not Found, 500 Internal Server Error и т.п. Во всех таких случаях функция `urlopen` выдаст обобщенное исключение `HTTPError`. Его можно обработать следующим образом:

```
from urllib.request import urlopen  
from urllib.error import HTTPError  
  
try:  
    html =  
urlopen('http://www.pythonscraping.com/pages/page1.html')  
except HTTPError as e:  
    print(e)
```

```

        # Вернуть ноль, прекратить работу или
        # выполнить еще какой-нибудь "план Б".
else:
    # Продолжить выполнение программы.
    # Примечание: если при перехвате исключений
    # программа прерывает работу или происходит
    возврат
    # из функции, то оператор else не нужен.

```

Теперь в случае возвращения кода HTTP-ошибки выводится сообщение о ней и остальная часть программы, которая находится в ветви `else`, не выполняется.

Если не найден весь сервер (например, сервер по адресу **<http://www.pythonscraping.com>** отключен или URL указан с ошибкой), то функция `urlopen` возвращает `URLError`. Эта ошибка говорит о том, что ни один из указанных серверов не доступен. Поскольку именно удаленный сервер отвечает за возвращение кодов состояния HTTP, ошибка `HTTPError` не может быть выдана и вместо нее следует обрабатывать более серьезную ошибку `URLError`. Для этого можно добавить в программу такую проверку:

```

from urllib.request import urlopen
from urllib.error import HTTPError
from urllib.error import URLError

try:
    html =
urlopen('https://pythonscrapingthisurldoesnotexist.com')
except HTTPError as e:
    print(e)

```



```
except URLError as e:
    print('The server could not be found!')
else:
    print('It Worked!')
```

Конечно, даже если страница успешно получена с сервера, все равно остается проблема с ее контентом, который не всегда соответствует ожидаемому. Всякий раз, обращаясь к тегу в объекте BeautifulSoup, разумно добавить проверку того, существует ли этот тег. При попытке доступа к несуществующему тегу BeautifulSoup возвращает объект None. Проблема в том, что попытка обратиться к тегу самого объекта None приводит к возникновению ошибки AttributeError.

Следующая строка (в которой nonexistentTag — несуществующий тег, а не имя реальной функции BeautifulSoup) возвращает объект None:

```
print(bs.nonExistentTag)
```

Этот объект вполне доступен для обработки и проверки. Проблема возникает в том случае, если продолжать его использовать без проверки и попытаться вызвать для объекта None другую функцию:

```
print(bs.nonExistentTag.someTag)
```

Эта функция вернет исключение:

```
AttributeError: 'NoneType' object has no
attribute 'someTag'
```

Как же застраховаться от этих ситуаций? Проще всего — явно проверить обе ситуации:

```

try:
    badContent = bs.nonExistingTag.anotherTag
except AttributeError as e:
    print('Tag was not found')
else:
    if badContent == None:
        print ('Tag was not found')
    else:
        print(badContent)

```

Такие проверка и обработка каждой ошибки поначалу могут показаться трудоемкими, однако если немного упорядочить код, то его станет проще писать (и, что еще важнее, гораздо проще читать). Вот, например, все тот же наш скрапер, написанный немного по-другому:

```

from urllib.request import urlopen
from urllib.error import HTTPError
from bs4 import BeautifulSoup

def getTitle(url):
    try:
        html = urlopen(url)
    except HTTPError as e:
        return None
    try:
        bs = BeautifulSoup(html.read(),
                             'html.parser')
        title = bs.body.h1
    except AttributeError as e:
        return None
    return title

```

```
title = getTitle('http://www.pythonscraping.com/pages/p
age1.html')
if title == None:
    print('Title could not be found')
else:
    print(title)
```

В этом примере мы создаем функцию `getTitle`, которая возвращает либо заголовок страницы, либо, если получить его не удалось, — объект `None`. Внутри `getTitle` мы, как в предыдущем примере, проверяем наличие `HTTPError` и инкапсулируем две строки `BeautifulSoup` внутри оператора `try`. Ошибка `AttributeError` может возникнуть в любой из этих строк (если сервер не найден, то `html` вернет объект `None`, а `html.read()` выдаст `AttributeError`). Фактически внутри оператора `try` можно разместить любое количество строк или вообще вызвать другую функцию, которая будет генерировать `AttributeError` в любой момент.

При написании скраперов важно продумать общий шаблон кода, который бы обрабатывал исключения, но при этом был бы читабельным. Вы также, вероятно, захотите использовать код многократно. Наличие обобщенных функций, таких как `getSiteHTML` и `getTitle` (в сочетании с тщательной обработкой исключений), позволяет быстро — и надежно — собирать данные с веб-страниц в Сети.

Глава 2. Углубленный синтаксический анализ HTML-кода

Однажды Микеланджело спросили, как ему удалось создать такой шедевр, как «Давид». Известен его ответ: «Это легко. Вы просто срезаете ту часть камня, которая не похожа на Давида».

Большинство веб-скраперов мало напоминают мраморные статуи, однако при извлечении информации из сложных веб-страниц стоит придерживаться аналогичного подхода. Существует множество способов отбрасывать контент, не похожий на тот, что вы ищете, до тех пор, пока не доберетесь до нужной информации. В этой главе вы узнаете, как выполнять анализ сложных HTML-страниц, чтобы извлекать из них только необходимую вам информацию.

Иногда молоток не требуется

Столкнувшись с гордиевыми узлами тегов, многие испытывают острое желание углубиться в них с помощью многострочных операторов в расчете извлечь ценную информацию. Однако следует помнить: безрассудное наслаивание методов, описанных в этом разделе, может привести к тому, что код будет трудно отлаживать, или он станет постоянно сбоить, или же и то и другое. Прежде чем начать, рассмотрим несколько способов, которые позволяют вообще обойтись без углубленного синтаксического анализа HTML-кода!

Предположим, мы ищем некий контент: имя, статистические данные или блок текста. Возможно, этот контент похоронен под 20 тегами в каше из HTML-кода и не отличается никакими особыми HTML-тегами или атрибутами.

Допустим, мы решили отбросить осторожность и написать примерно следующее в попытке извлечь нужные данные:

```
bs.find_all('table')[4].find_all('tr')  
[2].find('td').find_all('div')[1].find('a')
```

Выглядит не слишком красиво. Но дело не только в эстетике: стоит администратору сайта внести малейшее изменение, и весь наш веб-скрапер сломается. А если разработчик сайта решит добавить еще одну таблицу или еще один столбец данных? Или разместит в верхней части страницы еще один компонент (с несколькими тегами `div`)? Показанная выше строка кода нестабильна: она опирается на то, что структура сайта никогда не изменится.

Что же делать?

- Найдите ссылку **Print This Page** (Распечатать эту страницу) или, возможно, мобильную версию сайта с более удачным HTML-форматированием (подробнее о том, как выдать себя за мобильное устройство и получить мобильную версию сайта, см. в главе 14).
- Поищите информацию в файле JavaScript. Учтите, что для этого вам может понадобиться исследовать импортированные файлы JavaScript. Например, однажды я получила с сайта адреса улиц (вместе с широтой и долготой) в виде аккуратно отформатированного массива, заглянув в JavaScript-код встроенной карты Google, на которой были точно отмечены все адреса.
- Это больше касается заголовков, однако информация может находиться в URL самой страницы.

- Если информация, которую вы ищете, по какой-либо причине является уникальной для данного сайта, — вам не повезло. В противном случае попробуйте найти другие источники, из которых можно было бы получить эту информацию. Нет ли другого сайта с теми же данными? Возможно, на нем приводятся данные, скопированные или агрегированные с другого сайта?

Когда речь идет о скрытых или плохо отформатированных данных, особенно важно не зарываться в код, не загонять себя в кроличью нору, из которой потом можно и не выбраться. Лучше сделайте глубокий вдох и подумайте: нет ли других вариантов?

На тот случай, если вы уверены, что альтернатив не существует, в остальной части этой главы описываются стандартные и нестандартные способы выбора тегов по их расположению, контексту, атрибутам и содержимому. Представленные здесь методы, при условии правильного использования, способны значительно облегчить написание более стабильных и надежных веб-краулеров.

Еще одна тарелка BeautifulSoup

В главе 1 мы кратко рассмотрели установку и запуск BeautifulSoup, а также выбор объектов по одному. В этом разделе обсудим поиск тегов по атрибутам, работу со списками тегов и навигацию по дереву синтаксического анализа.

Почти любой сайт, с которым вам придется иметь дело, содержит таблицы стилей. На первый взгляд может показаться, что стили на сайтах предназначены исключительно для показа сайта пользователю в браузере. Однако это неверно: появление CSS стало настоящим благом для веб-скраперов. Чтобы

назначать элементам разные стили, CSS опирается на дифференциацию HTML-элементов, которые в противном случае имели бы одинаковую разметку. Например, одни теги могут выглядеть так:

```
<span class="green"></span>
```

А другие — так:

```
<span class="red"></span>
```

Веб-скраперы легко различают эти два тега по их классам; например, с помощью BeautifulSoup веб-скрапер может собрать весь красный текст, игнорируя зеленый. Поскольку CSS использует эти идентифицирующие атрибуты для соответствующего оформления сайтов, мы можем быть практически уверены в том, что на большинстве современных сайтов будет много атрибутов `class` и `id`.

Создадим пример веб-скрапера, который сканирует страницу, расположенную по адресу <http://www.pythonscraping.com/pages/warandpeace.html>.

На этой странице строки, в которых содержатся реплики персонажей, выделены красным цветом, а имена персонажей — зеленым. В следующем примере исходного кода страницы показаны теги `span`, которым присвоены соответствующие классы CSS:

```
<span class="red">Heavens! what a virulent  
attack!</span> replied
```

```
<span class="green">the prince</span>, not in  
the least disconcerted
```

```
by this reception.
```

С помощью программы, аналогичной той, которая была рассмотрена в главе 1, можно прочитать всю страницу и создать на ее основе объект BeautifulSoup:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html =
urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
```

С помощью этого объекта BeautifulSoup можно вызвать функцию `find_all` и извлечь Python-список всех имен персонажей, полученных путем выбора текста из тегов `<spanclass="green">` (`find_all` — очень гибкая функция, которую мы будем широко использовать в этой книге):

```
nameList = bs.find_all('span',
{'class': 'green'})
for name in nameList:
    print(name.get_text())
```

Результатом выполнения этого кода должен стать список всех персонажей «Войны и мира» в порядке их появления в тексте. Так что же здесь происходит? Раньше мы вызывали функцию `bs.имяТега` и получали первое появление тега на странице. Теперь мы вызываем функцию `bs.find_all(имяТега, атрибутыТега)`, чтобы получить не только первый тег, а список всех тегов, присутствующих на странице.

Получив список персонажей, программа перебирает все имена в списке и использует функцию `name.get_text()`, чтобы очистить контент от тегов.



Когда использовать `get_text()`, а когда — сохранять теги

Функция `.get_text()` удаляет из документа, с которым вы работаете, все теги и возвращает строку, содержащую только текст в кодировке Unicode. Например, при работе с большим блоком текста, содержащим много гиперссылок, абзацев и других тегов, все эти теги будут удалены, останется только текст.

Учтите, что в объекте BeautifulSoup гораздо проще найти нужное, чем в текстовом фрагменте. Вызов `.get_text()` всегда должен быть последним, что вы делаете непосредственно перед выводом результата на экран, сохранением или манипулированием готовыми данными.

Как правило, следует как можно дольше сохранять структуру тегов документа.

Функции `find()` и `find_all()`

Функции BeautifulSoup `find()` и `find_all()` вы, скорее всего, будете использовать чаще других. С помощью этих функций можно легко фильтровать HTML-страницы, чтобы выделить списки нужных тегов или найти отдельный тег по всевозможным атрибутам.

Эти две функции очень похожи, о чем свидетельствуют их определения в документации BeautifulSoup:

```
find_all(tag, attributes, recursive, text,
limit, keywords)
find(tag, attributes, recursive, text,
keywords)
```

Скорее всего, в 95 % случаев вы будете использовать только первые два аргумента: `tag` и `attributes`. Однако мы подробно рассмотрим все аргументы.

Аргумент `tag` нам уже встречался; мы можем передать функции строку, содержащую имя тега, или даже Python-список имен тегов. Например, следующий код возвращает список всех тегов заголовков, встречающихся в документе²:

```
.find_all(['h1', 'h2', 'h3', 'h4', 'h5', 'h6'])
```

Аргумент `attribute` принимает Python-словарь атрибутов и ищет теги, которые содержат любой из этих атрибутов. Например, следующая функция ищет в HTML-документе теги `span` с классом `green` или `red`:

```
.find_all('span', {'class':{'green', 'red'}})
```

Аргумент `recursive` логический. Насколько глубоко вы хотите исследовать документ? Если `recursive` присвоено значение `True`, то функция `find_all` ищет теги, соответствующие заданным параметрам, в дочерних элементах и их потомках. Если же значение этого аргумента равно `False`, то функция будет просматривать только теги верхнего уровня документа. По умолчанию `find_all` работает рекурсивно (`recursive` имеет значение `True`); обычно лучше оставить все как есть, за исключением ситуаций, когда вы

точно знаете, что делаете, и нужно обеспечить высокую производительность.

Аргумент `text` необычен из-за отношения не к свойствам тегов, а к их текстовому контенту. Так, чтобы узнать, сколько раз на странице встречается слово `the prince`, заключенное в теги, можно заменить функцию `.find_all()` из предыдущего примера на следующие строки:

```
nameList = bs.find_all(text='the prince')
print(len(nameList))
```

Результатом будет число 7.

Аргумент `limit` по понятным причинам используется только в методе `find_all`; функция `find` эквивалентна вызову `find_all` со значением `limit`, равным 1. Этот аргумент можно использовать в тех случаях, когда вы хотите извлечь только первые `x` элементов, присутствующих на странице. Однако следует учитывать, что вы получите первые элементы в порядке их появления на странице, и это вовсе не обязательно будут те элементы, которые вам нужны.

Аргумент `keyword` позволяет выбрать теги, содержащие определенный атрибут или набор атрибутов. Например:

```
title = bs.find_all(id='title', class_='text')
```

Этот код возвращает первый тег со словом `text` в атрибуте `class_` и словом `title` в атрибуте `id`. Обратите внимание: по соглашению всем значениям атрибута `id` на странице следует быть уникальными. Поэтому на практике такая строка может быть не особенно полезна и должна быть эквивалентна следующей:

```
title = bs.find(id='title')
```

Аргумент keyword и атрибут class

В определенных ситуациях аргумент keyword может быть полезен. Однако технически, как свойство объекта BeautifulSoup, он избыточен. Помните: все, что можно сделать с помощью keyword, также можно выполнить методами, описанными далее в этой главе (см. разделы «Регулярные выражения» на с. 46 и «Лямбда-выражения» на с. 52).

Например, следующие две строки кода работают одинаково:

```
bs.find_all(id='text')
bs.find_all('', {'id':'text'})
```

Кроме того, используя keyword, вы периодически будете сталкиваться с проблемами, особенно при поиске элементов по атрибуту class, поскольку class в Python является защищенным ключевым словом. Другими словами, class – зарезервированное слово Python, которое нельзя применять в качестве имени переменной или аргумента (это не имеет никакого отношения к обсуждавшемуся ранее аргументу keyword функции BeautifulSoup.find_all()). Например, попытка выполнить следующий код повлечет синтаксическую ошибку из-за нестандартного использования слова class:

```
bs.find_all(class='green')
```

Вместо этого можно применить несколько неуклюжее решение BeautifulSoup с добавлением подчеркивания:

```
bs.find_all(class_='green')
```

Кроме того, можно заключить слово `class` в кавычки:

```
bs.find_all('', {'class': 'green'})
```

В этот момент у вас может возникнуть вопрос: «Постойте, но ведь я уже знаю, как получить тег со списком атрибутов: нужно передать в функцию атрибуты в виде словарного списка!»

Напомню: передача списка тегов в `.find_all()` в виде списка атрибутов действует как фильтр «или» (функция выбирает тег, присутствующий в списке: `тег1`, `тег2`, `тег3` и т.д.). Если список тегов достаточно длинный, то можно получить множество ненужных данных. Аргумент `keyword` позволяет добавить к этому списку дополнительный фильтр «и».

Другие объекты BeautifulSoup

До сих пор в этой книге нам встречались два типа объектов библиотеки BeautifulSoup:

- *объекты* BeautifulSoup — экземпляры, которые в предыдущих примерах кода встречались в виде переменной `bs`;
- *объекты* Tag — в виде списков или отдельных элементов, как результаты вызовов функций `find` и `find_all` для объекта BeautifulSoup или полученные при проходе по структуре объекта BeautifulSoup:

```
bs.div.h1
```

Однако в библиотеке есть еще два объекта, которые используются реже, но все же о них важно знать:

- *объекты* `NavigableString` — служат для представления не самих тегов, а текста внутри тегов (некоторые функции принимают и создают не объекты тегов, а объекты `NavigableString`);
- *объекты* `Comment` — применяются для поиска HTML-комментариев, заключенных в теги комментариев, `<!--` например, так `-->`.

Из всей библиотеки BeautifulSoup вам придется иметь дело только с этими четырьмя объектами (на момент написания данной книги).

Навигация по деревьям

Функция `find_all` выполняет поиск тегов по их именам и атрибутам. Но как быть, если нужно найти тег по его расположению в документе? Здесь нам пригодится навигация по дереву. В главе 1 мы рассмотрели навигацию по дереву BeautifulSoup только в одном направлении:

```
bs.tag.subTag.anotherSubTag
```

Теперь рассмотрим навигацию по деревьям HTML-кода во всех направлениях: вверх, по горизонтали и диагонали. В качестве образца для веб-скрапинга мы будем использовать наш весьма сомнительный интернет-магазин, размещенный по адресу <http://www.pythonscraping.com/pages/page3.html>, показанный на рис. 2.1.

 Totally Normal Gifts			
<p>Here is a collection of totally normal, totally reasonable gifts that your friends are sure to love! Our collection is hand-curated.</p> <p>We haven't figured out how to make online shopping carts yet, but you can send us a check to: 123 Main St. Abuja, Nigeria We will then send your totally amazing gift, pronto! Please include an extra \$5.00 for gift wrapping.</p>			
Item Title	Description	Cost	Image
Vegetable Basket	This vegetable basket is the perfect gift for your health conscious (or overweight) friends! <i>Now with super-colorful bell peppers!</i>	\$15.00	
Russian Nesting Dolls	Hand-painted by trained monkeys, these exquisite dolls are priceless! And by "priceless," we mean "extremely expensive"! <i>8 entire dolls per set! Octuple the presents!</i>	\$10,000.52	
Fish Painting	If something seems fishy about this painting, it's because it's a fish! <i>Also hand-painted by trained monkeys!</i>	\$10,005.00	
Dead Parrot	This is an ex-parrot! <i>Or maybe he's only resting?</i>	\$0.50	

Рис. 2.1. Снимок экрана с сайта <http://www.pythonscraping.com/pages/page3.html>

HTML-код этой страницы, представленный в виде дерева (некоторые теги для краткости опущены), выглядит так:

HTML

- body
 - div.wrapper
 - h1
 - div.content
 - table#giftList
 - tr
 - th
 - th
 - th
 - th

```
    - tr.gift#gift1
      - td
      - td
        - span.excitingNote
      - td
      - td
        - img
    - ...другие строки таблицы...
  - div.footer
```

Мы будем использовать эту HTML-структуру в качестве примера в нескольких следующих разделах.

Работа с детьми и другими потомками

В информатике и некоторых разделах математики часто приходится слышать о детях, с которыми проделывают ужасные вещи: перемещают, сохраняют, удаляют и даже убивают. К счастью, в этом разделе мы будем их всего лишь выбирать!

В BeautifulSoup, как и во многих других библиотеках, существует различие между *детьми* и *потомками*: как и в генеалогическом древе любого человека, дети всегда располагаются ровно на один уровень ниже родителей, тогда как потомки могут находиться на любом уровне дерева ниже родителя. Скажем, теги `tr` являются детьми тега `table`, а теги `tr`, `th`, `td`, `img` и `span` — потомками тега `table` (по крайней мере, в нашем примере). Все дети — потомки, но не все потомки — дети.

В целом функции BeautifulSoup всегда имеют дело с потомками тега, выбранного в данный момент. Например, функция `bs.body.h1` выбирает первый тег `h1`, который

является потомком тега `body`. Она не найдет теги, расположенные за пределами `body`.

Аналогично функция `bs.div.find_all('img')` найдет первый тег `div` в документе, а затем извлечет список всех тегов `img`, которые являются потомками этого тега `div`.

Получить только тех потомков, которые являются детьми, можно с помощью тега `.children`:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for child in bs.find('table', {'id': 'giftList'}).children:
    print(child)
```

Данный код выводит список всех строк таблицы `giftList`, в том числе начальную строку с заголовками столбцов. Если вместо функции `children()` в этом коде использовать функцию `desndants()`, то она найдет в таблице и выведет примерно два десятка тегов, включая `img`, `span` и отдельные теги `td`. Определенно имеет смысл различать детей и потомков!

Работа с братьями и сестрами

Функция `next_siblings()` библиотеки `BeautifulSoup` упрощает сбор данных из таблиц, особенно если в таблице есть

заголовки:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

for sibling in bs.find('table', {'id': 'giftList'}).tr.next_siblings():
    print(sibling)
```

Этот код должен выводить все строки таблицы, кроме первой с заголовком. Почему пропускается строка заголовка? Потому что объект не может быть сиблингом сам себе. Каждый раз, когда составляется список сиблингов (братьев и сестер) объекта, сам объект не включается в этот список. Как следует из названия, данная функция выбирает только *следующих* по списку сиблингов. Например, если выбрать строку, расположенную в середине таблицы, и вызвать для нее функцию `next_siblings`, то функция вернет только тех сиблингов, которые идут в списке после данной строки. Таким образом, выбрав строку заголовка и вызвав функцию `next_siblings`, мы получим все строки таблицы, кроме самой строки заголовка.



Конкретизируйте свой выбор

Приведенный выше код будет работать ничуть не хуже, если выбрать первую строку таблицы как `bs.table.tr` или даже просто `bs.tr`. Однако в коде я не поленилась выразить все это в более длинной форме:

```
bs.find('table',{'id':'giftList'}).tr
```

Даже если на странице только одна таблица (или другой интересующий нас тег), можно легко что-то упустить. Кроме того, макеты страниц постоянно меняются. Элемент определенного типа, который когда-то стоял на странице первым, может в любой момент стать вторым или третьим. Чтобы сделать скрапер более надежным, лучше делать выбор тегов как можно более точным. По возможности используйте атрибуты тегов.

У функции `next_siblings` есть парная функция `previous_siblings`. Она часто бывает полезна, если в конце списка одноуровневых тегов, который вы хотели бы получить, есть легко выбираемый тег.

И конечно же, существуют функции `next_sibling` и `previous_sibling`, которые выполняют почти то же, что и `next_siblings` и `previous_siblings`, но только возвращают не список тегов, а лишь один тег.

Работа с родителями

При сборе данных со страниц вы, скорее всего, быстро поймете, что выбирать родительский тег необходимо реже, чем детей или сиблингов. Как правило, просмотр HTML-страницы с целью поиска данных мы начинаем с тегов верхнего уровня,

после чего ищем способ углубиться в нужный фрагмент данных. Однако иногда встречаются странные ситуации, когда приходится использовать функции поиска родительских элементов `.parent` и `.parents` из библиотеки `BeautifulSoup`. Например:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html =
urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
print(bs.find('img',
{'src': '../img/gifts/img1.jpg'})
      .parent.previous_sibling.get_text())
```

Этот код будет выводить цену объекта, изображенного на картинке `../img/gifts/img1.jpg` (в данном случае цена составляет 15 долларов).

Как это работает? Ниже представлена древовидная структура фрагмента HTML-страницы, с которой мы работаем, и пошаговый алгоритм:

```
<tr>
- td
- td
- td ③
  - "$15.00" ③
- td ③
  -  ③
```

③ Выбираем тег изображения с атрибутом `src=" ../img/gifts/img1.jpg"`.

③ Выбираем родителя этого тега (в данном случае тег `td`).

③ С помощью функции `previous_sibling` выбираем предыдущего сиблинга этого тега `td` (в данном случае тег `td`, который содержит цену продукта в долларах).

③ Выбираем текст, содержащийся в этом теге, — `"$15.00"`.

Регулярные выражения

Как говорится в старой шутке по информатике, «допустим, есть некая проблема, которую мы хотим решить с помощью регулярных выражений. Теперь у нас две проблемы».

К сожалению, изучение регулярных выражений (часто сокращаемых до *regex*) часто сводится к пространным таблицам случайных символов, сочетания которых выглядят как абракадабра. Это отпугивает многих людей, а потом они решают рабочие задачи и пишут ненужные сложные функции поиска и фильтрации, хотя можно было бы обойтись всего одной строкой с регулярным выражением!

К счастью для вас, регулярные выражения не так уж трудно быстро освоить. Чтобы их изучить, достаточно рассмотреть всего несколько простых примеров и поэкспериментировать с ними.

Регулярные выражения получили свое название благодаря тому, что используются для идентификации регулярных строк; регулярное выражение позволяет сделать однозначный вывод о том, что данная строка соответствует определенным правилам, и вернуть ее; или же сделать вывод о том, что строка не соответствует правилам, и отбросить ее. Это невероятно удобно для быстрой проверки больших документов, в которых

нужно найти номера телефонов или адреса электронной почты.

Обратите внимание: я использовала выражение «*регулярная строка*». Что это такое? Это любая строка, которую можно построить с учетом последовательности линейных правил³ следующего вида.

1. Написать хотя бы одну букву *a*.
2. Добавить ровно пять букв *b*.
3. Добавить произвольное четное число букв *c*.
4. В конце поставить букву *d* или *e*.

Этим правилам соответствуют строки *aaaabbbbbccccc*, *aabbbbbbsse* и т.д. (количество вариантов бесконечно).

Регулярные выражения — всего лишь краткий способ представления этих наборов правил. Например, регулярное выражение для описанного выше набора правил выглядит так:

$aa^*bbbbb(cc)^*(d|e)$

На первый взгляд эта строка смотрится жутковато, но она станет понятнее, если разбить ее на составляющие:

- aa^* — буква *a*, после которой стоит символ $*$ (звездочка), означает «любое количество букв *a*, включая 0». Такая запись гарантирует, что буква *a* будет написана хотя бы один раз;
- $bbbbb$ — ничего особенного, просто пять букв *b* подряд;
- $(cc)^*$ — любое количество чего угодно можно заключить в скобки. Поэтому для реализации правила о четном

количестве букв `s` мы можем написать две буквы `s`, заключить их в скобки и поставить после них звездочку. Это значит, что в строке может присутствовать любое количество пар, состоящих из букв `s` (обратите внимание, что это также может означать 0 пар);

- `(d|e)` — вертикальная линия между двумя выражениями означает «то или это». В данном случае мы говорим «добавить `d` или `e`». Таким образом мы гарантируем, что в строку добавится ровно один из этих двух символов.



Эксперименты с регулярными выражениями

Осваивая регулярные выражения, очень важно поиграть с ними и понять, как они работают. Если вам не хочется ради пары строк открывать редактор кода и запускать программу, то проверить, работает ли регулярное выражение должным образом, можно на одном из специальных сайтов, например Regex Pal (<http://regexpal.com/>).

В табл. 2.1 приводятся наиболее часто используемые символы регулярных выражений с краткими пояснениями и примерами. Этот список ни в коем случае не претендует на полноту. Кроме того, как уже упоминалось, вы можете столкнуться с незначительными различиями, в зависимости от языка. Однако приведенные здесь 12 символов наиболее часто

встречаются в регулярных выражениях Python и могут служить для поиска и сбора практически любых строк.

Таблица 2.1. Часто используемые символы регулярных выражений

Символ (-ы)	Значение	Пример	Соответствующие строки
*	Предыдущий символ, подвыражение или символ в скобках повторяется ноль или более раз	a*b*	aaaaaaaa, aaabbbbb, bbbbbbb
+	Предыдущий символ, подвыражение или символ в скобках повторяется один раз или более	a+b+	aaaaaaaaab, aaabbbbb, abbbbbbb
[]	Любой символ в скобках (может читаться как «выберите любое количество этих предметов»)	[A-Z]*	APPLE, CAPITALS, QWERTY
()	Сгруппированное подвыражение (в «порядке операций» над регулярными выражениями выполняется в первую очередь)	(a*b)*	aaabaab, abaaab, ababaaaaab
{m, n}	Предыдущий символ, подвыражение или символ в скобках повторяется от m до n раз (включительно)	a{2,3}b{2,3}	aabbbb, aaabbbb, aabb
[^]	Любой одиночный символ, которого нет в скобках	[^A-Z]*	apple, lowercase, qwerty
	Любой символ, строка символов или подвыражение из тех, что разделены знаком (обратите внимание: это не заглавная буква i, а вертикальная черта, также называемая прямым слешем или символом конвейеризации)	b(al ile)d	bad, bid, bed
.	Любой одиночный символ (буква, цифра, пробел и т.д.)	b.d	bad, bzd, b\$d, b d
^	Указывает на то, что символ или подвыражение должны находиться в начале строки	^a	apple, asdf, a
\	Экранирующий символ (позволяет использовать специальные символы в их буквальном значении)	\^ \ \\	^ \
\$	Часто ставится в конце регулярного выражения и означает «до конца строки». Без этого в конце любого регулярного выражения де-факто стоит «.*», что позволяет принимать строки, в которых совпадает только первая часть. Данный знак можно считать аналогом символа ^	[A-Z]* [a-z]*\$	ABCabc, zzyzx, Bob

?!	«Не содержит». Эта странная пара символов, непосредственно предшествующая символу (или регулярному выражению), указывает на то, что данный символ не должен присутствовать в этом месте строки. Иногда его сложно использовать; в конце концов, символ может находиться в другой части строки. Если вы хотите совсем исключить данный символ, то задействуйте это выражение в сочетании с ^ и \$ на обоих концах строки	^((?! [A-Z]).)*\$	No-caps-here, \$ymb0ls a4e f!ne
----	---	----------------------	------------------------------------

Один из классических примеров регулярных выражений — распознавание адресов электронной почты. Точные правила составления адресов электронной почты слегка различаются на разных почтовых серверах, однако можно составить несколько общих правил. Регулярное выражение для каждого из этих правил показано во втором столбце табл. 2.2.

Таблица 2.2. Точные правила составления адресов электронной почты и регулярные выражения для каждого из них

Правило 1	[A-Za-z0-9._+]
Первая часть адреса электронной почты обязательно содержит хотя бы один элемент из следующего списка: заглавные буквы, строчные буквы, цифры от 0 до 9, точки (.), знаки «плюс» (+) или подчеркивания (_)	Регулярные выражения очень красиво сокращаются. Например, A-Z означает любую заглавную букву от A до Z. Поместив все возможные последовательности и символы в квадратные скобки (не путать с круглыми), мы как бы говорим: «Символ может удовлетворять любому из условий, перечисленных в скобках». Обратите также внимание: знак + означает, что символы могут встречаться произвольное количество раз, но не менее одного
Правило 2	@
После этого в адресе электронной почты должен стоять символ @	Все очень просто: в середине адреса должен стоять символ @ и встречаться ровно один раз
Правило 3	[A-Za-z]
Адрес электронной почты должен содержать хотя бы одну букву, заглавную или строчную	В первой части имени домена, после символа @, допустимы только буквы, и их должно быть не менее одной

Правило 4	.
Потом идет точка (.)	Перед доменом верхнего уровня должна стоять точка (.)
Правило 5	(com org edu net)
Наконец, адрес электронной почты заканчивается на com, org, edu или net (в действительности существует еще много доменов верхнего уровня, но для нашего примера этих четырех должно быть достаточно)	Здесь перечислены возможные последовательности букв, которые могут стоять после точки во второй части адреса электронной почты

Объединив все эти правила, получим следующее регулярное выражение:

`[A-Za-z0-9._+]+@[A-Za-z]+.(com|org|edu|net)`

При попытке написать регулярное выражение с нуля лучше сначала составить список шагов, которые бы четко описывали, какой должна быть ваша строка. Обратите внимание на граничные случаи. Например, если вы описываете номера телефонов, то учитываете ли коды стран и добавочные номера?



Регулярные — не значит неизменные!

Стандартная версия регулярных выражений (описанная в данной книге и используемая в Python и BeautifulSoup) основана на синтаксисе Perl. Этот или похожий синтаксис применяется в большинстве современных языков программирования. Однако следует помнить, что при

использовании регулярных выражений на другом языке вы можете столкнуться с проблемами.

Даже у некоторых современных языков, таких как Java, есть незначительные различия в способе обработки регулярных выражений. Если сомневаетесь, то читайте документацию!

Регулярные выражения и BeautifulSoup

Если предыдущий раздел, посвященный регулярным выражениям, показался вам несколько оторванным от темы этой книги, то теперь мы восстановим связь. В отношении веб-скрапинга BeautifulSoup и регулярные выражения идут рука об руку. В сущности, функция, принимающая строку в качестве аргумента (например, `find(id="идентификаторТега")`), скорее всего, будет принимать и регулярное выражение.

Рассмотрим несколько примеров, проверив страницу по адресу <http://www.pythonscraping.com/pages/page3.html>.

Обратите внимание: на этом сайте есть много изображений товаров, представленных в таком виде:

```

```

Если мы хотим собрать URL всех изображений товаров, то на первый взгляд решение может показаться довольно простым: достаточно выбрать все теги изображений с помощью функции `.find_all("img")`, верно? Не совсем. Кроме очевидных «лишних» изображений (например, логотипов), на современных сайтах часто встречаются скрытые и пустые изображения, используемые вместо пробелов и для выравнивания элементов, а также другие случайные теги изображений, о которых вы, возможно, не знаете. Определенно

нельзя рассчитывать на то, что все изображения на странице являются только изображениями товаров.

Предположим также, что макет страницы может изменяться или по какой-либо причине поиск правильных тегов не должен зависеть от *расположения* изображений на странице. Например, вы хотите собрать определенные элементы или фрагменты данных, разбросанные по всему сайту случайным образом. Так, для титульного изображения товара может быть предусмотрено специальное место наверху некоторых — но не всех — страниц.

Решение состоит в поиске чего-то идентифицирующего сам тег. В данном случае можно поискать путь к файлам изображений товаров:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html =
urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')
images = bs.find_all('img',
                     {'src':re.compile('..\img\gifts/img.*.jpg')})
for image in images:
    print(image['src'])
```

Этот код выводит только те относительные пути к изображениям, которые начинаются с `../img/gifts/img` и заканчиваются на `.jpg`:

```
../img/gifts/img1.jpg
```

```
../img/gifts/img2.jpg  
../img/gifts/img3.jpg  
../img/gifts/img4.jpg  
../img/gifts/img6.jpg
```

Регулярное выражение может использоваться как аргумент в выражении, написанном на BeautifulSoup, что обеспечивает большую гибкость при поиске нужных элементов.

Доступ к атрибутам

До сих пор мы исследовали способы доступа к тегам и их контенту и способы их фильтрации. Однако часто при веб-скрапинге нас интересует не содержимое тега, а его атрибуты. Это особенно полезно для таких тегов, как `a`, в атрибуте `href` содержащих URL, на которые ссылаются эти теги, или же тегов `img`, в атрибуте `src` содержащих ссылки на целевые изображения.

Python позволяет автоматически получить список атрибутов для объекта тега, вызвав следующую функцию:

```
myTag.attrs
```

Помните: эта функция возвращает в чистом виде словарь Python, благодаря чему получение атрибутов и управление ими становится тривиальной задачей. Например, для того, чтобы узнать, где находится файл с изображением, можно воспользоваться следующим кодом:

```
myImgTag.attrs['src']
```

Лямбда-выражения

Если вы окончили вуз по специальности, связанной с информатикой или вычислительной техникой, то, скорее всего, проходили лямбда-выражения. Это было давно, один раз, и потом вы их больше никогда не использовали. Если же нет, то лямбда-выражения могут быть вам незнакомы (или же вы о них где-то слышали и когда-то даже собирались изучить). В данном разделе мы не станем слишком углубляться в эти типы функций, я только покажу, как можно применять их в веб-скрапинге.

По сути, *лямбда-выражение* — это функция, которая передается в другую функцию как переменная; вместо того чтобы определять функцию как $f(x, y)$, мы можем определить ее как $f(g(x), y)$ или даже как $f(g(x), h(x))$.

BeautifulSoup позволяет передавать в функцию `find_all` функции определенных типов в качестве параметров.

Единственное ограничение состоит в том, что эти функции должны принимать в качестве аргумента объект тега и возвращать логическое значение. В этой функции BeautifulSoup оценивает каждый переданный ей объект тега; теги, имеющие значение `True`, возвращаются, а остальные отбрасываются.

Например, следующая функция возвращает все теги, имеющие ровно два атрибута:

```
bs.find_all(lambda tag: len(tag.attrs) == 2)
```

Здесь в качестве аргумента передается функция `len(tag.attrs)==2`. Когда она равна `True`, функция `find_all` станет возвращать соответствующий тег. Другими словами, будут найдены все теги с двумя атрибутами, например следующие:

```
<div class="body" id="content"></div>
```

```
<span style="color:red" class="title"></span>
```

Лямбда-функции настолько полезны, что ими даже можно заменять существующие функции BeautifulSoup:

```
bs.find_all(lambda tag: tag.get_text() ==  
              'Or maybe he\'s only resting?')
```

То же самое можно выполнить и без лямбда-функции:

```
bs.find_all('', text='Or maybe he\'s only  
resting?')
```

Однако если вы помните синтаксис лямбда-функции и знаете, как получить доступ к свойствам тега, то вам, возможно, больше никогда не понадобится вспоминать остальной синтаксис BeautifulSoup!

Поскольку лямбда-функция может быть любой функцией, которая возвращает значение `True` или `False`, эти функции даже можно комбинировать с регулярными выражениями, чтобы найти теги с атрибутом, соответствующим определенному строковому шаблону.

[2](#) Для получения списка всех тегов `h<уровень>`, имеющих в документе, существуют более лаконичные варианты кода. Есть и другие способы решения таких задач, которые мы рассмотрим в разделе, посвященном регулярным выражениям.

[3](#) Вы спросите: «А существуют ли “нерегулярные” выражения?» Да, существуют, но выходят за рамки этой книги. Нерегулярные выражения описывают такие строки, как «написать простое число букв `a`, а после них — ровно вдвое большее число букв `b`» или «написать палиндром». Строки этого типа невозможно описать с помощью регулярных выражений. К счастью, мне никогда не встречались ситуации, в которых веб-скрапер должен был бы находить подобные строки.

Глава 3. Разработка веб-краулеров

До сих пор нам встречались лишь отдельные статические страницы с несколько искусственно законсервированными примерами. В этой главе мы познакомимся с реальными задачами, для решения которых веб-скраперы перебирают несколько страниц и даже сайтов.

Веб-краулеры называются так потому, что они «ползают» (crawl) по Всемирной паутине и собирают данные с веб-страниц. В основе их работы лежит рекурсия. Веб-краулер получает контент страницы по ее URL, исследует эту страницу, находит URL другой страницы, извлекает содержимое *этой* другой страницы и далее до бесконечности.

Однако будьте осторожны: возможность собирать данные в Интернете еще не означает, что вы всегда должны это делать. Веб-скраперы, показанные в предыдущих примерах, отлично работают в ситуациях, когда все необходимые данные находятся на одной странице. Используя веб-краулеры, следует быть предельно внимательными к тому, какую часть пропускной способности сети вы задействуете, и всеми силами постараться определить, есть ли способ облегчить нагрузку на интересующий вас сервер.

Проход отдельного домена

Даже если вам еще не приходилось слышать об игре «Шесть шагов по “Википедии”», вы наверняка знаете о ее предшественнице — «Шесть шагов до Кевина Бейкона». В обеих играх цель состоит в том, чтобы установить взаимосвязь между двумя мало связанными элементами (в первом случае — между статьями «Википедии», а во втором — между актерами,

сыгравшими в одном фильме) и построить цепь, содержащую не более шести звеньев (включая начальный и конечный элементы).

Например, Эрик Айдл (Eric Idle) снялся в фильме «Дадли Справедливый» (Dudley Do-Right) с Бренданом Фрейзером (Brendan Fraser), который, в свою очередь, снялся с Кевином Бейконом (Kevin Bacon) в «Воздухе, которым я дышу» (The Air I Breathe)⁴. В этом случае цепь от Эрика Айдла до Кевина Бейкона состоит всего из трех элементов.

В этом разделе мы начнем разработку проекта, который позволит строить цепи для «Шести шагов по “Википедии”»: например, начав со страницы Эрика Айдла (https://en.wikipedia.org/wiki/Eric_Idle), вы сможете найти наименьшее количество переходов по ссылкам, которые приведут вас на страницу Кевина Бейкона (https://en.wikipedia.org/wiki/Kevin_Bacon).

А как насчет нагрузки на сервер «Википедии»?

По данным Фонда Викимедиа (вышестоящей организации, отвечающей в том числе за «Википедию»), за одну секунду происходит примерно 2500 обращений к веб-ресурсам сайта, причем более 99 % из них относятся к «Википедии» (см. раздел Traffic volume на странице Wikimedia in figures, https://meta.wikimedia.org/wiki/Wikimedia_in_figures_-_Wikipedia#Traffic_volume). Из-за большого объема трафика ваши веб-скраперы вряд ли сколько-нибудь заметно повлияют на нагрузку сервера «Википедии». Тем не менее, если вы намерены активно использовать примеры кода, приведенные в этой книге, или будете разрабатывать

собственные проекты для веб-скрапинга «Википедии», я призываю вас совершить необлагаемое налогом пожертвование в Фонд Викимедиа (https://wikimediafoundation.org/wiki/Ways_to_Give) — не только в качестве компенсации нагрузки на сервер, но и чтобы помочь сделать образовательные ресурсы более доступными для других пользователей.

Кроме того, имейте в виду: если вы планируете создать большой проект с применением данных из «Википедии», то стоит убедиться, что эти данные еще неоступны через API «Википедии»

«(https://www.mediawiki.org/wiki/API:Main_page).

«Википедия» часто используется в качестве сайта для демонстрации веб-скраперов и веб-краулеров, поскольку данный сайт имеет простую структуру HTML-кода и относительно стабилен. Однако эти же данные могут оказаться более доступными через API.

Вы уже, вероятно, знаете, как написать скрипт на Python, который бы получал произвольную страницу «Википедии» и создавал список ссылок, присутствующих на этой странице:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html =
urlopen('http://en.wikipedia.org/wiki/Kevin_Bacon')
bs = BeautifulSoup(html, 'html.parser')
```

```
for link in bs.find_all('a'):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

Если вы посмотрите на созданный список ссылок, то заметите, что в него вошли все ожидаемые статьи: Apollo 13, Philadelphia, Primetime Emmy Award и т.д. Однако есть и кое-что лишнее:

```
//wikimediafoundation.org/wiki/Privacy_policy
//en.wikipedia.org/wiki/Wikipedia:Contact_us
```

Дело в том, что на каждой странице «Википедии» есть боковая панель, нижний и верхний колонтитулы с множеством ссылок; также есть ссылки на страницы категорий, обсуждений и другие страницы, которые не содержат полезных статей:

```
/wiki/Category:Articles_with_unsourced_statemen
ts_from_April_2014
/wiki/Talk:Kevin_Bacon
```

Недавно мой друг, работая над аналогичным проектом по веб-скрапину «Википедии», заявил, что написал большую функцию фильтрации, насчитывающую более 100 строк кода, с целью определить, является ли внутренняя ссылка «Википедии» ссылкой на страницу статьи. К сожалению, он не уделил достаточно времени поиску закономерностей между «ссылками на статьи» и «другими ссылками», иначе бы нашел более красивое решение. Если вы внимательно посмотрите на ссылки, которые ведут на страницы статей (в отличие от других внутренних страниц), то заметите, что у всех таких ссылок есть три общие черты:

- они находятся внутри тега `div`, у которого атрибут `id` имеет значение `bodyContent`;
- в их URL нет двоеточий;
- их URL начинаются с `/wiki/`.

Мы можем немного изменить код, включив в него эти правила, и получить только нужные ссылки на статьи, воспользовавшись регулярным выражением `^(/wiki/)((?!:).)*$`):

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen('http://en.wikipedia.org/wiki/Kevin_Bacon')
bs = BeautifulSoup(html, 'html.parser')
for link in bs.find('div', {'id': 'bodyContent'}).find_all('a', href=re.compile('^(/wiki/)((?!:).)*$')):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

Запустив этот код, мы получим список всех URL статей, на которые ссылается статья «Википедии» о Кевине Бейконе.

Скрипт, который находит все ссылки на статьи для одной жестко заданной статьи «Википедии», конечно, интересен, однако с практической точки зрения довольно бесполезен. Вы

должны уметь, взяв этот код за основу, преобразовать его примерно в такую программу.

- Отдельная функция `getLinks` принимает URL статьи «Википедии» в формате `/wiki/<Название_статьи>` и возвращает список всех URL, на которые ссылается эта статья, в том же формате.
- Основная функция, которая вызывает `getLinks` с первоначальной статьей, выбирает из возвращенного списка случайную ссылку и снова вызывает `getLinks`, пока пользователь не остановит программу или пока не окажется, что на очередной странице нет ссылок.

Вот полный код программы, которая это делает:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org{}'.format(articleUrl))
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div', {'id': 'bodyContent'}).find_all('a', href=re.compile('^(/wiki/)((?!:).)*$'))
```

```
links = getLinks('/wiki/Kevin_Bacon')
while len(links) > 0:
    newArticle = links[random.randint(0,
len(links)-1)].attrs['href']
    print(newArticle)
    links = getLinks(newArticle)
```

Сразу после импорта необходимых библиотек программа присваивает генератору случайных чисел начальное значение, равное текущему системному времени, чем обеспечивает новый и интересный случайный путь по статьям «Википедии» практически при каждом запуске программы.

Псевдослучайные числа и случайные начальные значения

В предыдущем примере был задействован генератор случайных чисел Python для случайного выбора статьи на странице, по ссылке на которую будет продолжен обход «Википедии». Однако случайные числа следует использовать осторожно.

Компьютеры хорошо справляются с вычислением правильных решений, но откровенно слабы, когда нужно придумать что-то новое. По этой причине случайные числа могут стать проблемой. Большинство алгоритмов генерации этих чисел стремятся создать равномерно распределенную и труднопредсказуемую числовую последовательность, однако для запуска работы такого алгоритма необходимо «начальное» число. Если оно каждый раз будет одним и тем же, то алгоритм станет всякий раз генерировать одну и ту же

последовательность «случайных» чисел. Именно поэтому я использовала значение системных часов в качестве начального значения для создания новых последовательностей случайных чисел и, следовательно, генерации последовательностей случайных статей. Благодаря этому выполнять программу будет немного интереснее.

Любопытно, что генератор псевдослучайных чисел Python работает по *алгоритму Мерсенна Твистера* (Mersenne Twister). Он генерирует труднопредсказуемые и равномерно распределенные случайные числа, однако несколько загружает процессор. Случайные числа — это хорошо, но за все приходится платить!

Затем программа определяет функцию `getLinks`, которая принимает URL статьи в формате `/wiki/...`, добавляет в начало имя домена «Википедии» `http://en.wikipedia.org` и получает объект BeautifulSoup с HTML-кодом, который находится по этому адресу. Затем функция извлекает список тегов со ссылками на статьи, исходя из описанных ранее параметров, и возвращает их.

В основной части программы сначала создается список тегов со ссылками на статьи (переменная `links`), в котором содержатся ссылки, найденные на начальной странице **`https://en.wikipedia.org/wiki/Kevin_Bacon`**. Затем программа выполняет цикл и находит тег со случайной ссылкой на статью, извлекает оттуда атрибут `href`, выводит страницу и получает по извлеченному URL новый список ссылок.

Разумеется, решение задачи «Шесть шагов по “Википедии”» не ограничивается созданием веб-скрапера, переходящего с

одной страницы на другую. Нам также необходимо хранить и анализировать полученные данные. Продолжение решения этой задачи вы найдете в главе 6.



Обрабатывайте исключения!

По большей части обработка исключений в этих примерах пропущена для краткости, однако следует помнить: здесь может возникнуть множество потенциальных ловушек. Что, если, например, «Википедия» изменит имя тега `bodyContent`? Тогда при попытке извлечь текст из тега программа выдаст исключение `AttributeError`.

Таким образом, несмотря на то, что эти сценарии достаточно хороши в качестве тщательно контролируемых примеров, в автономном коде готового приложения требуется обрабатывать исключения гораздо лучше, чем позволяет описать объем данной книги. Для получения дополнительной информации об этом вернитесь к главе 1.

Сбор информации со всего сайта

В предыдущем разделе мы прошли по сайту, переходя от одной случайно выбранной ссылки к другой. Но как быть, если нужно систематизировать сайт и составить каталог или просмотреть все страницы? Сбор информации со всего сайта, особенно большого, — процесс, требующий интенсивного использования

памяти. Лучше всего с этим справляются приложения, имеющие быстрый доступ к базе данных для хранения результатов краулинга. Однако мы можем исследовать поведение таких приложений, не выполняя их в полном объеме. Подробнее об их выполнении с применением базы данных см. в главе 6.

Темный и глубокий Интернет

Вероятно, вам часто приходилось слышать о *глубоком*, *темном* или *скрытом Интернете*, особенно в последних новостях. Что имеется в виду?

Глубокий Интернет — это любая часть Сети, выходящая за пределы *видимого Интернета*. Видимой называют ту часть Интернета, которая индексируется поисковыми системами. Несмотря на большой разброс оценок, глубокий Интернет почти наверняка составляет около 90 % всего Интернета. Поскольку Google не способен отправлять формы или находить страницы, на которые не ссылается домен верхнего уровня, а также не выполняет поиск сайтов, для которых это запрещено в файле `robots.txt`, видимый Интернет продолжает составлять относительно небольшую часть Сети.

Темный Интернет, также известный как *Даркнет*, — нечто совершенно иное. Он работает на той же сетевой аппаратной инфраструктуре, но использует браузер Tor или другой клиент, у которого протокол приложения работает поверх HTTP, обеспечивая безопасный канал для обмена информацией. В темном Интернете, как и на обычном сайте,

тоже можно выполнять веб-скрапинг, однако эта тема выходит за пределы данной книги.

В отличие от темного, в глубоком Интернете веб-скрапинг выполняется относительно легко. Многие инструменты, описанные в этой книге, научат вас, как выполнять веб-скрапинг и сбор информации во многих местах, недоступных для ботов Google.

В каких случаях сбор данных со всего сайта полезен, а в каких — вреден? Веб-скраперы, которые перебирают весь сайт, хороши во многих случаях, включая следующие.

- *Формирование карты сайта.* Несколько лет назад мне встретилась задача: важный клиент хотел оценить затраты на редизайн сайта, однако не хотел предоставлять моей компании доступ ко внутренним компонентам существующей системы управления контентом и у него не было общедоступной карты сайта. Я воспользовалась веб-краулером, чтобы пройти по всему сайту, собрать все внутренние ссылки и разместить страницы в структуре папок, соответствующей той, что применялась на сайте. Это позволило мне быстро обнаружить разделы сайта, о которых я даже не подозревала, и точно подсчитать, сколько эскизов страниц потребуется создать и какой объем контента необходимо перенести.
- *Сбор данных.* Другой клиент хотел собрать статьи (истории, посты в блогах, новости и т.п.), чтобы построить рабочий прототип специализированной поисковой платформы. Это исследование сайтов должно было быть не всеобъемлющим, однако достаточно обширным (нам хотелось получать

данные лишь с нескольких сайтов). Мне удалось создать веб-краулеры, которые рекурсивно обходили каждый сайт и собирали данные только со страниц статей.

Общий подход к полному сбору данных с сайта заключается в том, чтобы начать со страницы верхнего уровня (например, с начальной страницы) и построить список всех ее внутренних ссылок. Затем обойти все страницы, на которые указывают эти ссылки, и на каждой из них собрать дополнительные списки ссылок, запускающие очередной этап сбора данных.

Конечно же, в такой ситуации количество ссылок стремительно растет. Если на каждой странице есть десять внутренних ссылок, а глубина сайта составляет пять страниц (что довольно типично для сайта среднего размера), то необходимо проверить 10^5 , то есть 100 000 страниц, чтобы с уверенностью утверждать: сайт пройден полностью. Как ни странно, хоть и «пять страниц в глубину и десять внутренних ссылок на каждой странице» — довольно типичный размер сайта, очень немногие сайты действительно насчитывают 100 000 и более страниц. Причина, конечно, состоит в том, что подавляющее большинство внутренних ссылок являются дубликатами.

Во избежание повторного сбора данных с одной и той же страницы крайне важно, чтобы все обнаруженные внутренние ссылки имели согласованный формат и сохранялись в общем множестве, что облегчило бы поиск во время работы программы. *Множество* похоже на список, но его элементы не располагаются в определенной последовательности. Кроме того, в множестве содержатся исключительно уникальные элементы, что идеально подходит для наших целей. Следует проверять лишь те ссылки, которые являются «новыми», и искать дополнительные ссылки только по ним:

```

from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    for link in bs.find_all('a', href=re.compile('^(/wiki/)')):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                # мы нашли новую страницу
                newPage = link.attrs['href']
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)

getLinks('')

```

Для демонстрации полной работы этого веб-краулера я ослабила требования к тому, как должны выглядеть внутренние ссылки (из предыдущих примеров). Веб-скрапер не ограничивается только страницами статей, а ищет все ссылки, начинающиеся с /wiki/, независимо от того, где они располагаются на странице и содержат ли двоеточия. Напомню: в URL страниц статей отсутствуют двоеточия, в отличие от адресов страниц загрузки файлов, обсуждений и т.п.

Изначально функция `getLinks` вызывается с пустым URL, то есть для начальной страницы «Википедии»: к пустому URL

внутри функции добавляется `http://en.wikipedia.org`. Затем функция просматривает каждую ссылку на первой странице и проверяет, есть ли она в *глобальном множестве* страниц (множестве страниц, с которыми скрипт уже встречался). Если нет, то ссылка добавляется в список, выводится на экран и функция `getLinks` вызывается для нее рекурсивно.



Осторожно с рекурсией

Это предупреждение редко встречается в книгах по программированию, но я считаю, что вы должны знать: если оставить программу работать достаточно долго, то предыдущая программа почти наверняка завершится аварийно.

В Python установлен по умолчанию предел рекурсии (количество рекурсивных вызовов программы), и он равен 1000. Поскольку сеть ссылок «Википедии» чрезвычайно обширна, данная программа в итоге достигнет предела рекурсии и остановится, если только не добавить счетчик рекурсии или что-то еще, призванное помешать этому.

Для «плоских» сайтов глубиной менее 1000 ссылок данный метод обычно — за редким исключением — работает хорошо. Например, однажды мне встретилась ошибка в динамически генерируемом URL, поскольку ссылка на следующую страницу зависела от адреса текущей страницы. Это привело к

бесконечно повторяющимся путем, таким как `/blogs/blogs.../blogs/blog-post.php`.

Однако, как правило, этот рекурсивный метод должен подходить для любого обычного сайта, с которым вы, скорее всего, столкнетесь.

Сбор данных со всего сайта. Веб-краулеры были бы довольно скучными программами, если бы только переходили с одной страницы на другую. Полезными они могут быть, что-то делая на странице, на которую перешли. Рассмотрим пример веб-скрапера, который бы выбирал заголовок, первый абзац контента страницы и ссылку на режим редактирования страницы (если таковая существует).

Как всегда, чтобы выбрать лучший способ сделать это, сначала нужно просмотреть несколько страниц сайта и определить его структуру. При просмотре нескольких страниц «Википедии» (как статей, так и страниц, не являющихся статьями, например страницы с политикой конфиденциальности) становится ясно следующее.

- У любой страницы (независимо от статуса: статья, история редактирования или любая другая страница) есть заголовок, заключенный в теги `h1→span`, и это единственный тег `h1` на странице.
- Как уже говорилось, весь основной текст находится внутри тега `div#bodyContent`. Но если вы хотите выделить конкретную часть текста — например, получить доступ только к первому абзацу, — то лучше использовать `div#mw-content-text→p` (выбрать только тег первого абзаца). Это подходит для всех контентных страниц, кроме страниц файлов (например,

https://en.wikipedia.org/wiki/File:Orbit_of_274301_Wikipedia.svg)

, на которых нет разделов основного текста.

- Ссылки для редактирования существуют только на страницах статей. Если такая ссылка есть, то она находится в теге `li#ca-edit`, в `li#ca-edit→span→a`.

Изменив исходный код нашего веб-краулера, мы можем создать комбинированную программу для краулинга и сбора данных (или по крайней мере для вывода данных на экран):

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen('http://en.wikipedia.org{}'.format(pageUrl))
    bs = BeautifulSoup(html, 'html.parser')
    try:
        print(bs.h1.get_text())
        print(bs.find(id='mw-content-text').find_all('p')[0])
        print(bs.find(id='ca-edit').find('span').find('a').attrs['href'])
    except AttributeError:
        print('This page is missing something! Continuing.')
```

```

        for link in bs.find_all('a',
href=re.compile('^(/wiki/)')):
            if 'href' in link.attrs:
                if link.attrs['href'] not in pages:
                    # мы нашли новую страницу
                    newPage = link.attrs['href']
                    print('- '*20)
                    print(newPage)
                    pages.add(newPage)
                    getLinks(newPage)

getLinks('')

```

Цикл `for` в этой программе, по сути, такой же, как и в первоначальной программе сбора данных (для ясности добавлен вывод дефисов в качестве разделителей контента).

Поскольку никогда нельзя быть полностью уверенными в том, что на каждой странице будут присутствовать все нужные данные, операторы `print` расположены в соответствии с вероятностью наличия этих данных на странице. Так, тег заголовка `h1` есть на каждой странице (во всяком случае, насколько я могу судить), вследствие чего мы сначала пытаемся получить эти данные. Текстовый контент присутствует на большинстве страниц (за исключением страниц файлов), так что этот фрагмент данных извлекается вторым. Кнопка редактирования есть только на страницах с заголовками и текстовым контентом, и то не на всех.



Разные схемы для различных потребностей

Очевидно, что, когда в блоке обработчика исключений заключено нескольких строк, возникает некая опасность. Прежде всего, вы не можете сказать, какая именно строка вызвала исключение. Кроме того, если по какой-либо причине на странице есть кнопка редактирования, но нет заголовка, то кнопка никогда не будет зарегистрирована. Однако во многих случаях при наличии определенной вероятности возникновения на сайте каждого из элементов этого достаточно и непредвиденное отсутствие нескольких точек данных или ведение подробных журналов не является проблемой.

Вы могли заметить: в этом и во всех предыдущих примерах мы не столько «собирали» данные, сколько «выводили» их. Очевидно, что данными, выводимыми в окно терминала, сложно манипулировать. Подробнее о хранении информации и создании баз данных вы узнаете в главе 5.

Обработка перенаправлений

Перенаправления позволяют веб-серверу использовать имя текущего домена или URL для контента, находящегося в другом месте. Существует два типа перенаправлений:

перенаправления на стороне сервера, когда URL изменяется до загрузки страницы;

перенаправления на стороне клиента, иногда с сообщением наподобие «через десять секунд вы перейдете на другой сайт», когда сначала загружается

страница сайта, а потом происходит переход на новую страницу.

С перенаправлениями на стороне сервера вам обычно ничего не нужно делать. Если вы используете библиотеку `urllib` для Python 3.x, то она обрабатывает перенаправления автоматически! В случае применения библиотеки запросов проследите, чтобы флаг `allow_redirects` имел значение `True`:

```
r = requests.get( 'http://github.com' ,  
                  allow_redirects=True)
```

Просто помните, что иногда URL сканируемой страницы может не совпадать с URL, по которому расположена эта страница.

Подробнее о перенаправлениях на стороне клиента, которые выполняются с помощью JavaScript или HTML, см. в главе 12.

Сбор информации с нескольких сайтов

Каждый раз, когда я делаю доклад о взломе веб-страниц, кто-нибудь обязательно спрашивает: «Как построить Google?» Мой ответ всегда состоит из двух частей: «Во-первых, вам нужно где-то взять много миллиардов долларов, чтобы купить крупнейшие в мире хранилища данных и разместить их в скрытых местах по всему миру. Во-вторых, вам нужно разработать веб-краулер».

В 1996 году, когда Google только начиналась, она состояла всего из двух аспирантов из Стэнфорда, у которых был старый сервер и веб-краулер, написанный на Python. Теперь, зная, как

работает веб-скрапинг, вы официально располагаете инструментами, необходимыми для того, чтобы стать следующим технологическим мультимиллиардером!

Заявляю совершенно серьезно: именно веб-краулеры — основа того, что движет многими современными веб-технологиями, и для их использования не обязательно иметь большое хранилище данных. Чтобы выполнить любой кросс-доменный анализ данных, необходимо разработать веб-краулеры, способные интерпретировать и хранить данные, собранные со множества интернет-страниц.

Как и в предыдущем примере, веб-краулеры, которые мы намерены создать, будут переходить по ссылкам от страницы к странице, формируя карту сети. Однако на этот раз они не станут игнорировать внешние ссылки, а, наоборот, будут переходить по ним.



Впереди — неведомые воды

Имейте в виду: код, представленный ниже, может переходить в любую точку Интернета. Если мы что-то и узнали из «Шести шагов по “Википедии”», так это то, что всего за несколько переходов можно полностью уйти с сайта, например, <http://www.sesamestreet.org>, и попасть в гораздо менее приятное место.

Дети, получите разрешение у родителей, прежде чем запускать этот код. Если у вас есть твердые убеждения или религиозные ограничения, которые не позволяют вам посещать порносайты,

то ознакомьтесь с данным кодом, однако будьте осторожны, запуская программу.

Прежде чем приступить к написанию веб-краулера, который, хотите вы этого или нет, будет переходить по всем внешним ссылкам, следует задать себе несколько вопросов.

- Какие данные я намереваюсь собрать? Достаточно ли для этого выполнить веб-скрапинг нескольких заранее определенных сайтов (что почти всегда проще сделать), или же мой краулер должен иметь возможность обнаруживать новые сайты, о которых я, вероятно, не подозреваю?
- Попад на определенный сайт, должен ли мой краулер сразу переходить по следующей внешней ссылке на другой сайт, или же ему следует задержаться на какое-то время на этом и изучить его более углубленно?
- Существуют ли какие-либо условия, при которых я бы не хотел выполнять веб-скрапинг текущего сайта? Интересует ли меня контент на иностранных языках?
- Как я буду защищаться от судебных исков, если мой веб-краулер привлечет внимание веб-мастера одного из сайтов, на которые попадет? (Подробнее об этом см. в главе 18.)

Менее чем в 60 строках кода можно без труда уместить гибкий набор функций Python, комбинация которых дает различные типы веб-скраперов:

```
from urllib.request import urlopen
from urllib.parse import urlparse
from bs4 import BeautifulSoup
```

```

import re
import datetime
import random

pages = set()
random.seed(datetime.datetime.now())

# Получить список всех внутренних ссылок,
# найденных на странице.
def getInternalLinks(bs, includeUrl):
    includeUrl =
'{}'.format(urlparse(includeUrl).scheme,
              urlparse(includeUrl).netloc)
    internalLinks = []
    # найти все ссылки, которые начинаются с
    "/"
    for link in bs.find_all('a',
                             href=re.compile('^(/|.*'+includeUrl+')'
    )):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in
internalLinks:
                if(link.attrs['href'].startswith(
h('/'))):
                    internalLinks.append(
                        includeUrl+link.attrs['
href'])
                else:
                    internalLinks.append(link.a
ttrs['href'])
    return internalLinks

```

```

# Получить список всех внешних ссылок,
найденных на странице.
def getExternalLinks(bs, excludeUrl):
    externalLinks = []
    # Найти все ссылки, которые начинаются с
"http" или "www",
    # не содержащие текущий URL.
    for link in bs.find_all('a',
                             href=re.compile('^(\http|www)
((?!'+excludeUrl+'.)*$'))):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in
externalLinks:
                externalLinks.append(link.attrs
['href'])
    return externalLinks

def getRandomExternalLink(startingPage):
    html = urlopen(startingPage)
    bs = BeautifulSoup(html, 'html.parser')
    externalLinks = getExternalLinks(bs,
urlparse(startingPage).netloc)
    if len(externalLinks) == 0:
        print('No external links, looking
around the site for one')
        domain =
'{}://{}'.format(urlparse(startingPage).scheme,
urlparse(startingPage).netloc)
        internalLinks =
getInternalLinks(bs, domain)
        return
getRandomExternalLink(internalLinks[random.rand

```

```

int(0,
                                len(int
ernalLinks)-1)])
        else:
                                return
externalLinks[random.randint(0,
len(externalLinks)-1)]

def followExternalOnly(startingSite):
                                externalLink =
getRandomExternalLink(startingSite)
        print('Random external link is:
{}'.format(externalLink))
        followExternalOnly(externalLink)
        followExternalOnly('http://oreilly.com')

```

Эта программа начинает с сайта **http://oreilly.com** и случайным образом переходит от одной внешней ссылки к другой. Вот пример результатов, которые она выводит:

```

http://igniteshow.com/
http://feeds.feedburner.com/oreilly/news
http://hire.jobvite.com/CompanyJobs/Careers.aspx?c=q319
http://makerfaire.com/

```

Не обязательно внешние ссылки обнаружатся на первой же странице сайта. В этом случае для поиска внешних ссылок используется метод, аналогичный тому, который применялся в предыдущем примере веб-краулинга для рекурсивного перехода в глубину сайта до тех пор, пока не будет найдена внешняя ссылка.

Принцип работы этой программы показан на рис. 3.1 в виде блок-схемы.

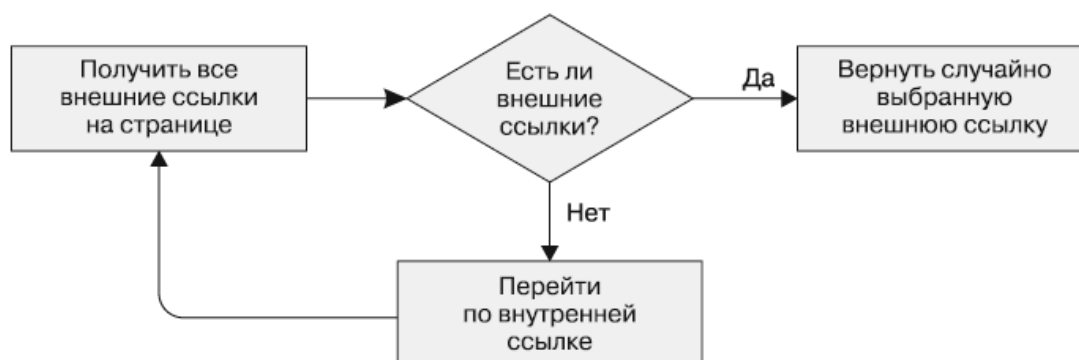


Рис. 3.1. Блок-схема нашего сценария краулинга сайтов



Не используйте учебные примеры программ в реальных приложениях

Еще раз напоминаю: ради экономии места и читабельности примеры программ, приведенные в этой книге, не всегда содержат необходимые проверки и обработки исключений, требуемые в готовом коде. Например, если веб-краулер не обнаружил на сайте ни одной внешней ссылки (что маловероятно, но в случае достаточно долгой работы программы в какой-то момент обязательно произойдет), то программа будет продолжать работу, пока не достигнет предела рекурсии Python.

Одним из простых способов повысить надежность этого веб-краулера было бы объединить его с кодом обработки

исключений при соединении, описанном в главе 1. Это позволило бы в случае возникновения ошибки HTTP или исключения на сервере при получении страницы выбрать другой URL и перейти по нему.

Прежде чем использовать этот код для каких-либо серьезных целей, обязательно добавьте в него проверки для устранения потенциальных ловушек.

Разделение задачи на простые функции, такие как «найти все внешние ссылки на странице», имеет преимущество: впоследствии код можно легко перестроить для выполнения другой задачи веб-краулинга. Например, если цель состоит в том, чтобы просмотреть сайт, найти все внешние ссылки и сохранить их, то можно добавить следующую функцию.

```
# Составляет список из всех внешних URL,
найденных на сайте.
allExtLinks = set()
allIntLinks = set()

def getAllExternalLinks(siteUrl):
    html = urlopen(siteUrl)

    domain =
    '{}://{}'.format(urlparse(siteUrl).scheme,
        urlparse(siteUrl).netloc)
    bs = BeautifulSoup(html, 'html.parser')
    internalLinks = getInternalLinks(bs,
domain)
    externalLinks = getExternalLinks(bs,
domain)
```

```

for link in externalLinks:
    if link not in allExtLinks:
        allExtLinks.add(link)
        print(link)
for link in internalLinks:
    if link not in allIntLinks:
        allIntLinks.add(link)
        getAllExternalLinks(link)

```

```

allIntLinks.add('http://oreilly.com')
getAllExternalLinks('http://oreilly.com')

```

Этот код можно представить как два совместно работающих цикла: один собирает внутренние ссылки, а второй — внешние. Блок-схема программы выглядит примерно так (рис. 3.2).

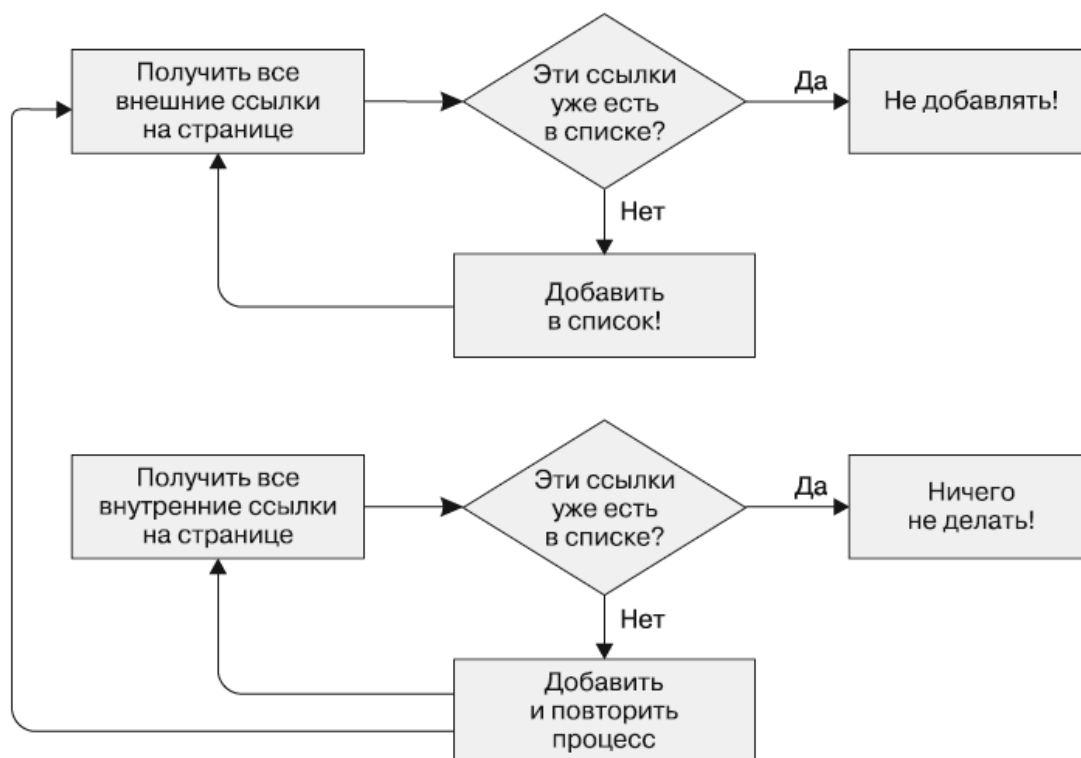


Рис. 3.2. Блок-схема веб-краулера сайтов, который собирает все внешние ссылки

Привычка составлять схемы и диаграммы того, что должен делать код, прежде чем писать его, фантастически полезна. По мере усложнения ваших веб-краулеров она сэкономит вам кучу времени и избавит от многих разочарований.

⁴ Благодарю сайт The Oracle of Bacon (<http://oracleofbacon.org/>), с помощью которого я удовлетворила свое любопытство относительно этой цепочки.

Глава 4. Модели веб-краулинга

Писать чистый и масштабируемый код довольно сложно, даже если у вас есть возможность контролировать входные и выходные данные. Написание кода веб-краулера, которому иногда приходится выполнять веб-скрапинг и сохранять различные данные с разных групп сайтов никак не контролируемых программистом часто представляет собой невероятно сложную организационную задачу.

Вам могут предложить собрать новостные статьи или публикации из блогов, размещенных на различных сайтах, у каждого из которых свои шаблоны и макеты. На одном сайте тег `h1` может содержать заголовок статьи, а на другом — заголовок самого сайта, а заголовок статьи будет заключен в тег `<spanid="title">`.

Возможно, вам понадобится гибко управлять тем, для каких сайтов нужно выполнить веб-скрапинг и как именно это делать, а также способ быстро добавлять новые сайты или изменять существующие — максимально быстро и без необходимости писать много строк кода.

Вас могут попросить собрать цены на товары с разных сайтов, чтобы в итоге можно было сравнивать цены на один и тот же товар. Возможно, они будут представлены в разных валютах. Вдобавок, вероятно, потребуется объединить их с внешними данными, полученными из какого-то другого источника, не имеющего отношения к Интернету.

Несмотря на то что вариантов применения веб-краулеров бесчисленное множество, большие масштабируемые веб-краулеры, как правило, относятся к одному из нескольких типов. Исследуя эти типы и распознавая ситуации, в которых

они используются, можно значительно повысить удобство сопровождения и надежность своих веб-краулеров.

В этой главе мы уделим основное внимание веб-краулерам, собирающим ограниченное количество «типов» данных (таких как обзоры ресторанов, новостные статьи, профили компаний) с большого количества сайтов и хранящим эти данные в виде объектов Python, которые читают и записывают в базу данных.

Планирование и определение объектов

Одна из распространенных ловушек веб-скрапинга — определение данных, которые разработчики намерены собирать, исключительно на основании того, что есть у них перед глазами. Например, желая собрать данные о товаре, можно было бы для начала заглянуть в магазин одежды и решить, что для каждого интересующего нас товара необходимо получить данные из следующих полей:

- наименование товара;
- цена;
- описание;
- размеры;
- цвета;
- тип ткани;
- рейтинг клиентов.

Посмотрев на другой сайт, вы обнаружите, что у товара есть SKU (stock keeping unit, единицы хранения, или артикул, используемый для отслеживания и заказа товаров). Вы наверняка захотите собирать и эти данные, даже если их не было на первом сайте! И вы добавите в список данное поле:

- артикул.

Для начала одежда вполне подойдет, однако необходимо убедиться, что этот веб-скрапер можно будет распространить и на другие типы товаров. Начав просматривать разделы товаров на других сайтах, вы поймете, что также необходимо собрать следующую информацию:

- твердая/мягкая обложка;
- матовая/глянцевая печать;
- количество отзывов клиентов;
- ссылка на сайт производителя.

Понятно, что такой подход неприемлем. Просто добавляя атрибуты к типу товара всякий раз, когда на сайте встречается новая информация, вы получите слишком большое количество полей, которые нужно отслеживать. Мало того, каждый раз при веб-скрапинге нового сайта придется подробно анализировать поля, уже имеющиеся на нем, и те, что уже были накоплены ранее, а также, возможно, добавлять новые (изменяя тип объекта Python и структуру базы данных). Это приведет к запутанному и трудночитаемому набору данных, а следовательно, к проблемам при его использовании.

Принимая решение о том, какие данные собирать, зачастую лучше игнорировать сайты. Нельзя запустить проект, рассчитанный на то, чтобы стать большим и масштабируемым, посмотрев только на один сайт и спросив себя: «Что здесь есть?» Вместо этого нужно задать другой вопрос: «Что мне нужно?» — а затем найти способы поиска необходимой информации.

Возможно, в действительности вам нужно всего лишь сравнивать цены на товары в нескольких магазинах и отслеживать изменения этих цен с течением времени. В этом случае вам необходима только та информация, которая бы позволила однозначно идентифицировать товар:

- название товара;
- производитель;
- идентификационный номер товара (если он есть и интересен вам).

Важно отметить, что ни одно из этих свойств товара не относится к конкретному магазину. Например, обзоры товаров, рейтинги, цены и даже описания одного и того же товара могут отличаться в разных магазинах, и их стоит хранить отдельно.

Другие данные (цветовые варианты товара и материал, из которого он сделан) относятся к товару, но могут быть разрезенными — существовать не для всех единиц товара. Важно сделать шаг назад, составить контрольный список для каждого свойства товара, которое вы считаете необходимым, и задать себе следующие вопросы.

- Поможет ли данная информация достичь целей проекта? Зайду ли я в тупик, если не получу ее, или это лишь данные

из разряда «пригодится», но, по большому счету, они ни на что не влияют?

- Если эти данные когда-нибудь *могут* пригодиться (а могут и нет) — насколько сложно будет вернуться и собрать их?
- Являются ли эти данные избыточными относительно уже собранных?
- Имеет ли логический смысл хранить данные именно в этом объекте? (Как уже упоминалось, сохранение описания в качестве свойства товара не имеет смысла, если описание одного и того же товара может быть разным на разных сайтах.)

Определившись, какие данные нужно собрать, важно задать себе еще несколько вопросов, чтобы затем решить, как хранить и обрабатывать эти данные в коде.

- Эти данные разреженные или плотные? Свойственны ли они любому товару, в любом списке или же существуют только для небольшого множества товаров?
- Насколько велик объем данных?
- Особенно это касается больших данных: нужно ли будет регулярно получать их каждый раз при выполнении анализа или только в отдельных случаях?
- Насколько переменчивы данные этого типа? Придется ли регулярно добавлять новые атрибуты, изменять типы (например, часто могут добавляться ткани с новыми узорами), или же эти данные никогда не изменяются (допустим, размеры обуви)?

Предположим, вы хотите выполнить некий метаанализ зависимости цены от свойств товара: например, от количества страниц в книге или типа ткани, из которой сшита одежда; также, возможно, в будущем вы захотите добавить другие атрибуты, от которых зависит цена. Просматривая подобные вопросы, вы можете обнаружить, что эти данные являются разреженными (сравнительно немногие товары имеют хотя бы один из указанных атрибутов), и решить, что придется часто добавлять или удалять атрибуты. В этом случае, вероятно, стоит создать тип товара, который выглядит следующим образом:

- название товара;
- производитель;
- идентификационный номер товара (если есть и нужен);
- атрибуты (необязательный параметр; список или словарь).

Тип атрибута выглядит так:

- имя атрибута;
- значение атрибута.

Со временем это позволит гибко добавлять новые атрибуты товара, не прибегая к необходимости изменять схему данных или переписывать код. Определяясь с тем, как хранить эти атрибуты в базе данных, вы можете решить записывать их в поле `attribute` в формате JSON или же хранить каждый атрибут в отдельной таблице, откуда извлекать их по

идентификатору товара. Подробнее о реализации этих типов моделей баз данных см. в главе 6.

Эти же вопросы можно задать и для другого типа информации, которую вам нужно будет хранить. Чтобы отслеживать цены, найденные для каждого товара, вам, вероятно, потребуется следующее:

- идентификатор товара;
- идентификатор магазина;
- цена;
- дата или метка времени для данной цены.

Что будет в случае, если атрибуты товара действительно влияют на его цену? Например, за рубашку большего размера магазин может выставить более высокую цену, чем за рубашку меньшего, так как пошив требует больше труда или материалов. В этом случае, возможно, стоит преобразовать товар «рубашка» в список товаров, по одному для каждого размера (чтобы у каждого вида рубашки была независимая цена), или же создать новый тип элемента для хранения информации об экземплярах товара, содержащий следующие поля:

- идентификатор товара;
- тип экземпляра (в данном случае размер рубашки).

Тогда каждая цена будет выглядеть так:

- идентификатор экземпляра товара;

- идентификатор магазина;
- цена;
- дата или метка времени для данной цены.

Тема «товары и цены» может показаться слишком узкой, однако основные вопросы, которые нужно себе задать, и логика, используемая при разработке этих объектов Python, применимы практически в любой ситуации.

При веб-скрапинге новостных публикаций может потребоваться следующая основная информация:

- заголовок;
- автор;
- дата;
- контент.

Но предположим, что некоторые статьи содержат «дату изменения», или «связанные публикации», или «количество перепостов в социальных сетях». Вам нужна эта информация? Она имеет отношение к вашему проекту? Как эффективно и гибко хранить количество перепостов в социальных сетях, если только некоторые новостные сайты используют все формы социальных сетей, а со временем популярность разных социальных сетей может увеличиваться или уменьшаться?

При разработке нового проекта может возникнуть соблазн сразу погрузиться в написание кода на Python, чтобы немедленно приступить к веб-скрапингу. Но если оставить

создание модели данных на потом, то доступность и формат данных часто будут определяться первым попавшимся сайтом.

Однако модель данных является основой всего кода, который ее использует. Неправильный выбор модели легко может привести к проблемам с написанием и сопровождением кода или к затруднениям при извлечении и эффективном применении полученных данных. Особенно это касается работы с разными сайтами: и известными, и неизвестными. Притом жизненно важно все хорошенько обдумать и спланировать, какие именно данные нужно собирать и как их хранить.

Работа с различными макетами сайтов

Одно из наиболее впечатляющих достижений поисковых систем, таких как Google, состоит в возможности извлекать релевантные и полезные данные с различных сайтов, не имея предварительных знаний об их структуре. Мы, люди, способны с первого взгляда определить, где у страницы заголовок, а где — основной контент (за исключением случаев крайне плохого веб-дизайна), однако заставить бот делать то же самое гораздо труднее.

К счастью, в большинстве случаев при веб-краулинге мы не намерены собирать данные с сайтов, которые никогда прежде не видели. Обычно речь идет максимум о нескольких десятках сайтов, предварительно отобранных человеком. Это значит, что нам не понадобятся сложные алгоритмы или машинное обучение для определения того, какой текст на странице «больше всего похож на заголовок», а какой, скорее всего, является «основным контентом». Все эти элементы можно задать вручную.

Наиболее очевидный подход — написать отдельный веб-краулер или парсер страниц для каждого сайта. Каждый такой краулер может принимать URL, строку или объект BeautifulSoup и возвращать результат веб-скрапинга в виде объекта Python.

Ниже приведены пример класса Content (представляющего собой фрагмент контента сайта, например новостную публикацию) и две функции веб-скрапинга, которые принимают объект BeautifulSoup и возвращают экземпляр Content:

```
import requests

class Content:
    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

def getPage(url):
    req = requests.get(url)
    return BeautifulSoup(req.text,
                          'html.parser')

def scrapeNYTimes(url):
    bs = getPage(url)
    title = bs.find('h1').text
    lines =
bs.select('div.StoryBodyCompanionColumn div p')
    body = '\n'.join([line.text for line in
lines])
    return Content(url, title, body)
```

```
def scrapeBrookings(url):
    bs = getPage(url)
    title = bs.find('h1').text
    body = bs.find('div', {'class', 'post-
body'}).text
    return Content(url, title, body)
```

```
url = 'https://www.brookings.edu/blog/future-
development/2018/01/26/'
'delivering-inclusive-urban-access-3-
uncomfortable-truths/'
content = scrapeBrookings(url)
print('Title: {}'.format(content.title))
print('URL: {}\n'.format(content.url))
print(content.body)
```

```
url =
'https://www.nytimes.com/2018/01/25/opinion/sun
day/'
'silicon-valley-immortality.html'
content = scrapeNYTimes(url)
print('Title: {}'.format(content.title))
print('URL: {}\n'.format(content.url))
print(content.body)
```

Добавляя функции веб-скрапинга для других новостных сайтов, вы, вероятно, заметите некоторую закономерность. В сущности, функция синтаксического анализа любого сайта делает одно и то же:

- находит элемент заголовка и извлекает оттуда текст заголовка;
- находит основной контент статьи;
- при необходимости находит другие элементы контента;
- возвращает объект Content, созданный с помощью ранее найденных строк.

Единственное, что здесь действительно зависит от сайта, — это CSS-селекторы, используемые для получения каждого элемента информации. Функции BeautifulSoup `find` и `find_all` принимают два аргумента: строку тега и словарь атрибутов в формате «ключ — значение», вследствие чего эти аргументы можно передавать как параметры, которые определяют структуру сайта и расположение нужных данных.

Чтобы было еще удобнее, вместо аргументов тегов и пар «ключ — значение» можно использовать функцию BeautifulSoup `select`, принимающую строку CSS-селектора для каждого элемента информации, который вы хотите получить, и разместить все эти селекторы в словарном объекте:

```
class Content:
    """
        Общий родительский класс для всех статей/
        страниц.
    """
    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body
```

```

def print(self):
    """
    Гибкая функция печати, управляющая выводом
    данных.
    """
    print('URL: {}'.format(self.url))
    print('TITLE: {}'.format(self.title))
    print('BODY:\n{}'.format(self.body))

class Website:
    """
    Содержит информацию о структуре сайта.
    """
    def __init__(self, name, url, titleTag,
bodyTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
        self.bodyTag = bodyTag

```

Обратите внимание: в классе `Website` хранится не информация, собранная с разных страниц, а инструкции о том, как ее собирать. Так, здесь хранится не заголовок «Название моей страницы», а лишь строка с тегом `h1`, который указывает на то, где содержатся заголовки. Именно поэтому класс называется `Website` (его информация относится ко всему сайту), а не `Content` (в котором содержится информация только с одной страницы).

С помощью классов `Content` и `Website` можно написать класс `Crawler` для веб-скрапинга заголовка и контента,

размещенных по любому URL веб-страницы, принадлежащей данному сайту:

```
import requests
from bs4 import BeautifulSoup

class Crawler:
    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text,
                              'html.parser')

    def safeGet(self, pageObj, selector):
        """
        Служебная функция, используемая для
        получения строки
        содержимого из объекта BeautifulSoup и
        селектора.
        Если объект для данного селектора не
        найден,
        то возвращает пустую строку.
        """
        selectedElems =
        pageObj.select(selector)
        if selectedElems is not None and
        len(selectedElems) > 0:
            return '\n'.join(
```

```

        [elem.get_text() for elem in
selectedElems])
        return ''

```

```

def parse(self, site, url):
    """
        Извлекает содержимое страницы с
заданным URL.
    """
    bs = self.getPage(url)
    if bs is not None:
        title = self.safeGet(bs,
site.titleTag)
        body = self.safeGet(bs,
site.bodyTag)
        if title != '' and body != '':
            content = Content(url, title,
body)
            content.print()

```

А следующий код определяет объекты сайтов и запускает весь процесс:

```

crawler = Crawler()

siteData = [
    ['0\Reilly Media', 'http://oreilly.com',
'h1', 'section#product-description'],
    ['Reuters', 'http://reuters.com', 'h1',
'div.StandardArticleBody_body_1gnLA'],
    ['Brookings', 'http://www.brookings.edu',
'h1', 'div.post-body'],

```

```

        ['New York Times', 'http://nytimes.com',
         'h1', 'div.StoryBodyCompanionColumn div p']
    ]
    websites = []
    for row in siteData:
        websites.append(Website(row[0], row[1],
                                row[2], row[3]))

    crawler.parse(websites[0],
                  'http://shop.oreilly.com/product/'\
                    '0636920028154.do')
    crawler.parse(websites[1],
                  'http://www.reuters.com/article/'\
                    'us-usa-epa-pruitt-idUSKBN19W2D0')
    crawler.parse(websites[2],
                  'https://www.brookings.edu/blog/'\
                    'techtank/2016/03/01/idea-to-retire-old-
                    methods-of-policy-education/')
    crawler.parse(websites[3],
                  'https://www.nytimes.com/2018/01/'\
                    '28/business/energy-environment/oil-
                    boom.html')

```

На первый взгляд этот новый способ может показаться удивительно простым, по сравнению с написанием отдельной функции Python для каждого сайта. Однако представьте, что произойдет при переходе от системы с четырьмя сайтами-источниками к системе с 20 или 200 источниками.

Написать список строк относительно легко, и он не займет много места. Такой список можно загрузить из базы данных или CSV-файла, импортировать из удаленного источника или передать кому-то, кто не является программистом, но умеет

заполнять формы и вводить новые сайты через пользовательский интерфейс, и этот человек никогда не увидит ни одной строки кода.

Конечно, здесь есть недостаток: мы отказываемся от определенной гибкости. В первом примере у каждого сайта есть собственная функция, написанная в свободной форме, для выбора и — если получение результата того требует — синтаксического анализа HTML-кода. Во втором примере все сайты должны иметь схожую структуру, в которой гарантированно существуют определенные поля. Полученные из них данные должны быть чистыми, а каждому интересующему нас полю следует иметь уникальный и надежный CSS-селектор.

Однако я считаю, что широкие возможности и относительная гибкость данного подхода более чем компенсируют его реальные или предполагаемые недостатки. В следующем разделе мы рассмотрим конкретные способы применения и варианты расширения этой базовой схемы, которые позволят, например, справляться с отсутствующими полями, собирать различные типы данных, проверять только определенные части сайта и хранить более сложную информацию о страницах.

Структурирование веб-краулеров

Создание гибких и изменяемых типов разметки сайтов не принесет особой пользы, если все равно придется вручную искать каждую ссылку, чтобы выполнить для нее веб-скрапинг. В предыдущей главе были показаны различные способы извлечения данных с сайтов и автоматического поиска новых страниц.

В этом разделе будет показано, как встроить эти методы в хорошо структурированный и расширяемый веб-краулер, который бы автоматически собирал ссылки и находил нужные данные. Здесь я представлю только три основные структуры веб-краулера, хотя считаю их пригодными к применению в большинстве ситуаций, которые вам, скорее всего, встретятся при извлечении данных с реальных сайтов, — разве что придется внести пару изменений. Я также надеюсь, что если вам попадется необычный случай, связанный с нестандартной задачей веб-краулинга, то эти структуры послужат для вас источником вдохновения и позволят построить веб-краулер с красивой и надежной структурой.

Веб-краулинг с помощью поиска

Один из самых простых способов сбора данных с сайта — тот же, что используют люди: с помощью панели поиска. Поиск на сайте по ключевому слову или теме и последующий сбор данных по списку результатов может показаться задачей, весьма зависящей от конкретного сайта, однако несколько ключевых моментов делают ее на удивление примитивной.

- Большинство сайтов получают список результатов поиска по определенной теме, передавая ее в виде строки через параметр URL, например: `http://example.com?search=мояТема`. Первую часть этого URL можно сохранить как свойство объекта `Website` и потом просто каждый раз добавлять к нему тему.
- Выполнив поиск, большинство сайтов формируют страницы результатов в виде легко идентифицируемого списка ссылок, обычно заключенных в удобный тег наподобие

`<spanclass="result">`, точный формат которого тоже можно сохранить в виде свойства объекта `Website`.

- Каждая *ссылка на результат поиска* представляет собой либо относительный URL (такой как `/articles/page.html`), либо абсолютный (например, `http://example.com/articles/page.html`). Независимо от того, какой вариант вы ожидаете — абсолютный или относительный, — URL можно сохранить как свойство объекта `Website`.
- Найдя и нормализовав URL на странице поиска, мы успешно сводим задачу к примеру, рассмотренному в предыдущем разделе, — извлечению данных со страницы сайта, имеющей заданный формат.

Рассмотрим реализацию этого алгоритма в виде кода. Класс `Content` здесь почти такой же, как в предыдущих примерах. Мы только добавили свойство `URL`, которое позволит отслеживать, откуда взят этот контент:

```
class Content:
    """Общий родительский класс для всех
    статей/страниц"""

    def __init__(self, topic, url, title,
body):
        self.topic = topic
        self.title = title
        self.body = body
        self.url = url
```

```

def print(self):
    """
        Гибкая функция печати, управляющая
        выводом данных
    """
    print('New article found for topic:
    {}'.format(self.topic))
    print('URL: {}'.format(self.url))
    print('TITLE: {}'.format(self.title))
    print('BODY:\n{}'.format(self.body))

```

Мы также добавили к классу Website несколько новых свойств. Свойство `searchUrl` определяет, по какому адресу следует обратиться, чтобы получить результаты поиска, если добавить к нему искомую тему. Свойство `resultListing` определяет блок, в котором заключена информация о каждом результате поиска, а `resultUrl` — тег внутри этого блока, содержащий точный URL результата. Свойство `absoluteUrl` представляет собой значение логического типа, указывающее на то, какими ссылками являются результаты поиска — абсолютными или относительными.

```

class Website:
    """Содержит информацию о структуре сайта"""

    def __init__(self, name, url, searchUrl,
resultListing,
resultUrl, absoluteUrl, titleTag,
bodyTag):
        self.name = name
        self.url = url
        self.searchUrl = searchUrl

```

```
self.resultListing = resultListing
self.resultUrl = resultUrl
self.absoluteUrl=absoluteUrl
self.titleTag = titleTag
self.bodyTag = bodyTag
```

Мы немного расширили файл `crawler.py` — теперь в нем содержатся данные `Website`, список тем для поиска и два цикла, в которых перебираются все темы и сайты. В этом файле также содержится функция `search`, переходящая на страницу поиска заданного сайта с заданной темой и извлекающая оттуда все найденные URL, перечисленные на странице результатов поиска.

```
import requests
from bs4 import BeautifulSoup

class Crawler:

    def getPage(self, url):
        try:
            req = requests.get(url)
        except requests.exceptions.RequestException:
            return None
        return BeautifulSoup(req.text,
                              'html.parser')

    def safeGet(self, pageObj, selector):
        childObj = pageObj.select(selector)
        if childObj is not None and len(childObj) > 0:
```



```

        return childObj[0].get_text()
    return ''

def search(self, topic, site):
    """
        Поиск на заданном сайте по заданной
теме
        и сохранение всех найденных страниц.
    """
    bs = self.getPage(site.searchUrl +
topic)
                                searchResults =
bs.select(site.resultListing)
    for result in searchResults:
        url = result.select(site.resultUrl)
[0].attrs['href']
        # Проверить, является ли URL
относительным или абсолютным.
        if(site.absoluteUrl):
            bs = self.getPage(url)
        else:
            bs = self.getPage(site.url +
url)
        if bs is None:
            print('Something was wrong with
that page or URL. Skipping!')
            return
            title = self.safeGet(bs,
site.titleTag)
            body = self.safeGet(bs,
site.bodyTag)
            if title != '' and body != '':

```

```

        content = Content(topic, title,
body, url)
        content.print()

crawler = Crawler()

siteData = [
    ['O'Reilly Media', 'http://oreilly.com',
'https://ssearch.oreilly.com/?q=',
    'article.product-result', 'p.title a',
True, 'h1',
    'section#product-description'],
    ['Reuters', 'http://reuters.com',
'http://www.reuters.com/search/news?blob=',
    'div.search-result-content', 'h3.search-
result-title a', False, 'h1',
    'div.StandardArticleBody_body_1gnLA'],
    ['Brookings', 'http://www.brookings.edu',
    'https://www.brookings.edu/search/?s=',
'div.list-content article',
    'h4.title a', True, 'h1', 'div.post-body']
]
sites = []
for row in siteData:
    sites.append(Website(row[0], row[1],
row[2],
row[3], row[4],
row[5], row[6], row[7]))

topics = ['python', 'data science']
for topic in topics:
    print('GETTING INFO ABOUT: ' + topic)

```

```
for targetSite in sites:
    crawler.search(topic, targetSite)
```

Этот скрипт перебирает все темы из списка `topics` и, прежде чем приступить к веб-скрапину по очередной теме, выводит предупреждение:

```
GETTING INFO ABOUT python
```

Затем скрипт просматривает все сайты из списка `sites` и сканирует каждый из них по каждой теме. Всякий раз, успешно находя информацию о странице, скрипт выводит ее в консоль:

```
New article found for topic: python
URL: http://example.com/examplepage.html
TITLE: Page Title Here
BODY: Body content is here
```

Обратите внимание: скрипт перебирает все темы, проходя по всем сайтам во внутреннем цикле. Почему бы не поступить наоборот, сначала перебрав все темы на одном сайте, а затем — на следующем? Цикл с перебором по темам позволяет более равномерно распределить нагрузку на веб-серверы. Это особенно важно, если наш список состоит из нескольких сотен тем и нескольких десятков сайтов. Не стоит направлять на один сайт сразу несколько десятков тысяч запросов; лучше сделать десять запросов, подождать несколько минут, затем сделать еще десять запросов, подождать еще несколько минут и т.д.

В итоге общее количество запросов не изменится, однако, как правило, лучше растянуть их на максимально возможное время. Обратите внимание: структура наших циклов обеспечивает простой способ сделать это.

Сбор данных с сайтов по ссылкам

В предыдущей главе было показано несколько способов, позволяющих обнаруживать на веб-страницах внутренние и внешние ссылки и затем использовать их для сбора данных с сайта. В данном подразделе мы объединим эти базовые методы и создадим более гибкий краулер сайтов, способный переходить по любой ссылке, соответствующей заданному URL-шаблону.

Веб-краулер такого типа хорошо работает для проектов, в которых нужно собрать данные со всего сайта, а не только из определенных результатов поиска или списка страниц. Этот метод хорошо работает и для страниц, имеющих между собой мало общего или совсем ничего.

В отличие от примера сбора данных со страниц с результатами поиска, рассмотренного выше, данные типы краулеров не требуют структурированного метода поиска ссылок, поэтому атрибуты, описывающие страницу поиска, в объекте `Website` не нужны. Однако, поскольку краулеру не даны конкретные инструкции о том, где и как расположены ссылки, которые он ищет, требуются некие правила, указывающие на то, какие страницы следует выбрать. Для этого мы предоставляем `targetPattern` — регулярное выражение, описывающее нужные URL — и создаем логическую переменную `absoluteUrl`:

```
class Website:
    def __init__(self, name, url,
targetPattern, absoluteUrl, titleTag, bodyTag):
        self.name = name
        self.url = url
        self.targetPattern = targetPattern
        self.absoluteUrl = absoluteUrl
```

```

        self.titleTag = titleTag
        self.bodyTag = bodyTag

class Content:

    def __init__(self, url, title, body):
        self.url = url
        self.title = title
        self.body = body

    def print(self):
        print('URL: {}'.format(self.url))
        print('TITLE: {}'.format(self.title))
        print('BODY:\n{}'.format(self.body))

```

Класс Content — тот же класс, что и в первом примере веб-краулера.

Класс Crawler стартует с начальной страницы сайта, находит внутренние ссылки и анализирует контент каждой из этих внутренних ссылок:

```

import re

class Crawler:
    def __init__(self, site):
        self.site = site
        self.visited = []

    def getPage(self, url):
        try:
            req = requests.get(url)

```

```

except
requests.exceptions.RequestException:
    return None
    return BeautifulSoup(req.text,
'html.parser')

    def safeGet(self, pageObj, selector):
        selectedElems =
pageObj.select(selector)
        if selectedElems is not None and
len(selectedElems) > 0:
            return '\n'.join([elem.get_text()
for
            elem in selectedElems])
        return ''

    def parse(self, url):
        bs = self.getPage(url)
        if bs is not None:
            title = self.safeGet(bs,
self.site.titleTag)
            body = self.safeGet(bs,
self.site.bodyTag)
            if title != '' and body != '':
                content = Content(url, title,
body)
                content.print()

    def crawl(self):
        """
        Получить ссылки с начальной страницы
сайта

```

```

        """
        bs = self.getPage(self.site.url)
        targetPages = bs.find_all('a',
                                   href=re.compile(self.site.targetPat
tern))
        for targetPage in targetPages:
            targetPage =
targetPage.attrs['href']
            if targetPage not in self.visited:
                self.visited.append(targetPage)
                if not self.site.absoluteUrl:
                    targetPage = '{
{}' .format(self.site.url, targetPage)
                self.parse(targetPage)

reuters = Website('Reuters',
                  'https://www.reuters.com', '^(/article/)',
                  False,
                  'h1', 'div.StandardArticleBody_body_1gnLA')
crawler = Crawler(reuters)
crawler.crawl()

```

Еще одно изменение, по сравнению с предыдущими примерами: объект `Website` (в данном случае переменная `reuters`), в свою очередь, является свойством объекта `Crawler`. Это позволяет удобно хранить в краулере посещенные страницы (`visited`), но также означает необходимость создания для каждого сайта нового краулера, вместо того чтобы многократно использовать один и тот же для проверки списка сайтов.

Безотносительно того, хотите ли вы, чтобы веб-краулер не зависел от конкретных сайтов, или желаете сделать его

атрибутом объекта `Crawler`, это структурное решение необходимо принимать в соответствии с конкретными потребностями. В общем случае годится любой из данных подходов.

Следует также отметить: веб-краулер станет получать ссылки с начальной страницы, но не продолжит сбор данных после того, как все эти страницы будут пройдены. Возможно, вы захотите написать веб-краулер, работающий по одной из схем, описанных в главе 3, который будет искать ссылки на каждой посещенной странице. Вы даже можете пройти по всем URL на всех страницах (а не только на имеющих заданную структуру), чтобы найти URL, соответствующие заданной структуре.

Сбор данных со страниц нескольких типов

В отличие от сбора данных с заданного множества страниц проверка всех внутренних ссылок на сайте может вызвать проблему, поскольку вы никогда точно не знаете, что получите. К счастью, есть несколько основных способов определения типа страницы.

- *По URL* — все публикации в блогах могут содержать URL (например, **`http://example.com/blog/title-ofpost`**).
- *По наличию или отсутствию определенных полей* — если на странице есть дата, но нет имени автора, то эту страницу можно классифицировать как пресс-релиз. При наличии у страницы заголовка, основного изображения и цены, но при отсутствии основного контента это может быть страница товара.

- По наличию на странице определенных тегов, идентифицирующих страницу, — теги можно использовать, даже если мы не собираем данные внутри них. Веб-краулер может искать элемент наподобие `<div id="relatedproducts">`, чтобы идентифицировать страницу как страницу товара, даже если вас не интересуют сопутствующие товары.

Чтобы отслеживать несколько типов страниц, вам понадобится создать на Python несколько типов объектов страниц. Это можно сделать следующими двумя способами.

Если страницы похожи (имеют в целом одинаковые типы контента), то можно добавить к существующему объекту веб-страницы атрибут `pageType`:

```
class Website:
    def __init__(self, name, url, titleTag,
bodyTag, pageType):
        self.name = name
        self.url = url
        self.titleTag = titleTag
        self.bodyTag = bodyTag
        self.pageType = pageType
```

Если страницы хранятся в SQL-подобной базе данных, то такой тип структуры страниц указывает на вероятное хранение этих страниц в одной таблице, в которую будет добавлено поле `pageType`.

Если же страницы или контент, который вы ищете, заметно различаются (содержат поля разных типов), то может понадобиться создать отдельные объекты для каждого типа страниц. Конечно, все веб-страницы будут иметь нечто общее:

URL и, вероятно, имя или заголовок. Это идеальная ситуация для использования подклассов:

```
class Webpage:
    def __init__(self, name, url, titleTag):
        self.name = name
        self.url = url
        self.titleTag = titleTag
```

Веб-краулер не будет использовать этот объект напрямую, однако на него будут ссылаться типы страниц:

```
class Product(Website):
    """Содержит информацию для веб-скрапинга
    страницы товара"""
    def __init__(self, name, url, titleTag,
productNumberTag, priceTag):
        Website.__init__(self, name, url,
TitleTag)
        self.productNumberTag =
productNumberTag
        self.priceTag = priceTag

class Article(Website):
    """Содержит информацию для веб-скрапинга
    страницы статьи"""
    def __init__(self, name, url, titleTag,
bodyTag, dateTag):
        Website.__init__(self, name, url,
titleTag)
        self.bodyTag = bodyTag
        self.dateTag = dateTag
```

Страница `Product` расширяет базовый класс `Website`, добавляя к нему атрибуты `productNumber` и `price`, относящиеся лишь к товарам, а класс `Article` добавляет атрибуты `body` и `date`, которые к товарам неприменимы.

Эти два класса можно использовать, например, для веб-скрапинга интернет-магазина, на сайте которого содержатся не только товары, но и публикации в блоге или пресс-релизы.

Размышления о моделях веб-краулеров

Собирать информацию из Интернета — словно пить из пожарного шланга. Ее слишком много, и не всегда понятно, что именно вам нужно и как это получить. Ответ на эти вопросы должен стать первым шагом в любом крупном (а иногда и небольшом) веб-проекте.

При сборе одних и тех же данных в разных областях или из разных источников ваша цель почти всегда должна состоять в том, чтобы попытаться нормализовать эти данные. Работать с данными, имеющими одинаковые или сопоставимые поля, гораздо проще, чем с данными, формат которых полностью определяется их первоначальным источником.

Во многих случаях приходится создавать веб-скраперы, исходя из вероятного появления новых источников данных в дальнейшем и с целью минимизировать издержки на программирование, которые потребуются для добавления этих новых источников. Даже если на первый взгляд сайт не соотносится с вашей моделью, возможно, есть более тонкие способы обеспечить данное соответствие. В долгосрочной перспективе умение замечать эти базовые закономерности позволит вам сэкономить время и деньги, а также избавит от множества неприятностей.

Кроме того, не следует игнорировать взаимосвязи между фрагментами данных. Предположим, вы ищете информацию, у которой в разных источниках есть свойства «тип», «размер» или «тема». Как вы будете хранить, извлекать и осмысливать эти атрибуты?

Архитектура ПО — обширная и важная тема, освоению которой можно посвятить всю карьеру. К счастью, программная архитектура для веб-скрапинга — гораздо более ограниченный и управляемый набор навыков, приобретаемых относительно легко. Занимаясь веб-скрапингом данных, вы, скорее всего, со временем заметите одни и те же постоянно повторяющиеся базовые закономерности. Чтобы создать хорошо структурированный веб-скрапер, не нужно много тайных знаний, но потребуется уделить время обдумыванию проекта в целом.

Глава 5. Scrapy

В предыдущей главе мы рассмотрели некоторые методы и схемы построения больших, масштабируемых и (что самое важное!) удобных в сопровождении веб-краулеров. Несмотря на то что их несложно разработать вручную, есть множество библиотек, фреймворков и даже инструментов с графическим интерфейсом, которые сделают это за вас или по крайней мере постараются немного упростить вам жизнь.

В этой главе вы познакомитесь с одной из лучших платформ для разработки веб-краулеров — Scrapy. Когда я работала над первым изданием данной книги, Scrapy для Python 3.x еще не была выпущена, поэтому ее упоминание в книге ограничилось одним разделом. С тех пор библиотека стала поддерживать Python 3.3+, в ней появились дополнительные функции, и я с удовольствием уделю ей не раздел, а целую главу.

Одна из проблем создания веб-краулеров состоит в том, что часто приходится выполнять одни и те же задачи: находить все ссылки на странице, оценивать разницу между внутренними и внешними ссылками, переходить на новые страницы. Эти основные стандартные операции полезно знать и уметь писать с нуля, но библиотека Scrapy способна многое из упомянутого сделать автоматически.

Конечно, Scrapy не читает мысли. Вам по-прежнему необходимо описать шаблоны страниц, указать точку, с которой следует начать работу, и построить правила для URL искомых страниц. Но и в этих случаях библиотека предоставляет чистую основу для построения четко структурированного кода.

Установка Scrapy

Scrapy предоставляет инструмент для скачивания библиотеки с ее сайта (<http://scrapy.org/download/>), а также инструкции по установке с помощью сторонних менеджеров, таких как `pip`.

Из-за сравнительно большого размера и сложности Scrapy обычно нельзя установить, как другие фреймворки, с помощью такой команды:

```
$ pip install Scrapy
```

Обратите внимание: я говорю «обычно», поскольку теоретически можно использовать и эту команду. Однако на практике я, как правило, сталкиваюсь с одной или несколькими сложными проблемами, связанными с зависимостями, несовпадением версий и неразрешимыми ошибками.

Если вы решили установить Scrapy с помощью `pip`, то настоятельно рекомендую использовать виртуальное окружение (подробнее о виртуальных окружениях см. во врезке «Хранение библиотек непосредственно в виртуальных окружениях» на с. 28).

Я предпочитаю другой способ установки — с помощью менеджера пакетов Anaconda (<https://docs.continuum.io/anaconda/>). Это программный продукт, производимый компанией Continuum и предназначенный для того, чтобы сглаживать острые углы при поиске и установке популярных пакетов Python для обработки данных. В следующих главах мы будем использовать многие другие пакеты, которыми управляет Anaconda, такие как NumPy и NLTK.

После установки Anaconda можно установить Scrapy с помощью следующей команды:

```
conda install -c conda-forge scrapy
```

Если у вас возникнут проблемы или потребуется свежая информация, то обратитесь к руководству по установке Scrapy (<https://doc.scrapy.org/en/latest/intro/install.html>).

Инициализация нового «паука». После установки платформы Scrapy необходимо выполнить небольшую настройку для каждого «паука» (spider) — проекта Scrapy, который, как и обычный паук, занимается обходом сети. В этой главе я буду называть «пауком» именно проект Scrapy, а краулером — любую программу, которая занимается сбором данных во Всемирной паутине, независимо от того, использует она Scrapy или нет.

Чтобы создать нового «паука» в текущем каталоге, нужно ввести в командной строке следующую команду:

```
$ scrapy startproject wikiSpider
```

В результате в каталоге будет создан новый подкаталог, а в нем — проект под названием wikiSpider. Внутри этого каталога находится следующая файловая структура:

```
scrapy.cfg
wikiSpider
- spiders
  - __init.py__
- items.py
- middlewares.py
- pipelines.py
- settings.py
- __init.py__
```

Поначалу в эти файлы Python записывается код-заглушка, что позволяет быстро создать новый проект «паука». В следующих разделах данной главы мы продолжим работать с проектом wikiSpider.

Пишем простой веб-скрапер

Чтобы создать веб-краулер, нужно добавить в дочерний каталог **wikiSpider** новый файл `wikiSpider/wikiSpider/article.py`. Затем в этом файле `article.py` нужно написать следующее:

```
import scrapy

class ArticleSpider(scrapy.Spider):
    name='article'

    def start_requests(self):
        urls = [
            'http://en.wikipedia.org/wiki/Python',
            '%28programming_language%29',
            'https://en.wikipedia.org/wiki/Functional_programming',
            'https://en.wikipedia.org/wiki/Monty_Python']
        return [scrapy.Request(url=url,
                                callback=self.parse)
                for url in urls]

    def parse(self, response):
        url = response.url
```



```
title =  
response.css('h1::text').extract_first()  
print('URL is: {}'.format(url))  
print('Title is: {}'.format(title))
```

Имя этого класса (**ArticleSpider**) отличается от имени каталога (**wikiSpider**), что указывает на следующее: данный класс отвечает за просмотр только страниц статей в рамках более широкой категории **wikiSpider**, которую мы впоследствии сможем использовать для поиска других типов страниц.

Для больших сайтов с разными типами контента можно создать отдельные элементы Scrapy для каждого типа (публикации в блогах, пресс-релизы, статьи и т.п.). У каждого из этих типов будут свои поля, но все они станут работать в одном проекте Scrapy. Имя каждого «паука» должно быть уникальным в рамках проекта.

Следует обратить внимание еще на две важные вещи: функции `start_requests` и `parse`.

Функция `start_requests` — это предопределенная Scrapy точка входа в программу, используемая для генерации объектов `Request`, которые в Scrapy применяются для сбора данных с сайта.

Функция `parse` — это функция обратного вызова, определяемая пользователем, которая передается в объект `Request` с помощью `callback=self.parse`. Позже мы рассмотрим более мощные трюки, которые возможны благодаря функции `parse`, но пока что она просто выводит заголовок страницы.

Для запуска «паука» `article` нужно перейти в каталог **wikiSpider/wikiSpider** и выполнить следующую команду:

```
$ scrapy runspider article.py
```

По умолчанию Scrapy выводит довольно подробные данные. Помимо отладочной информации, это будут примерно такие строки:

```
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG:
Crawled (200)
<GET      https://en.wikipedia.org/robots.txt>
(referer: None)
2018-01-21                                23:28:57
[scrapy.downloadermiddlewares.redirect]
DEBUG:      Redirecting      (301)      to      <GET
https://en.wikipedia.org/wiki/
Python_%28programming_language%29>      from      <GET
http://en.wikipedia.org/
wiki/Python_%28programming_language%29>
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG:
Crawled (200)
<GET
https://en.wikipedia.org/wiki/Functional_programming>
(referer: None)
URL                                              is:
https://en.wikipedia.org/wiki/Functional_programming
Title is: Functional programming
2018-01-21 23:28:57 [scrapy.core.engine] DEBUG:
Crawled (200)
<GET
https://en.wikipedia.org/wiki/Monty_Python>
(referer: None)
URL                                              is:
https://en.wikipedia.org/wiki/Monty_Python
Title is: Monty Python
```

Веб-скрапер проходит по трем страницам, указанным в списке `urls`, собирает с них информацию и завершает работу.

«Паук» с правилами

«Паук», созданный в предыдущем разделе, не очень-то похож на веб-краулер, так как ограничен лишь списком предоставленных ему URL. Он не умеет самостоятельно искать новые страницы. Чтобы превратить этого «паука» в полноценный веб-краулер, нужно задействовать предоставляемый Scrapy класс `CrawlSpider`.



Где искать этот код в репозитории GitHub

К сожалению, фреймворк Scrapy нельзя просто запустить из редактора Jupyter, что затрудняет фиксацию внесения изменений в код. Для представления примеров кода в тексте главы мы сохранили веб-скрапер из предыдущего раздела в файле `article.py`, а следующий пример создания «паука» Scrapy, перебирающего несколько страниц, — в файле `articles.py` (обратите внимание на использование буквы `s` в конце слова `articles`).

Последующие примеры также будут храниться в отдельных файлах со своими именами, которые будут приводиться в соответствующих разделах. Убедитесь, что используете правильные имена файлов при запуске этих примеров.

Следующий класс находится в файле `articles.py` в репозитории GitHub:

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class ArticleSpider(CrawlSpider):
    name = 'articles'
    allowed_domains = ['wikipedia.org']
    start_urls = [
        'https://en.wikipedia.org/wiki/'
        'Benevolent_dictator_for_life'
    ]
    rules = [Rule(LinkExtractor(allow=r'.*'),
        callback='parse_items',
        follow=True)]

    def parse_items(self, response):
        url = response.url

        title = response.css('h1::text').extract_first()
        text = response.xpath('//div[@id="mw-content-text"]//text()')
        ).extract()
        lastUpdated = response.css('li#footer-info-lastmod::text')
        ).extract_first()
        lastUpdated = lastUpdated.replace(
            'This page was last edited on ',
            ''
        )
        print('URL is: {}'.format(url))
```

```
print('title is: {}'.format(title))
print('text is: {}'.format(text))
print('Last updated:
{}'.format(lastUpdated))
```

Этот новый «паук» `ArticleSpider` является наследником класса `CrawlSpider`. Вместо функции `start_requests` он предоставляет списки `start_urls` и `allowed_domains`, которые сообщают «пауку», с чего начать обход сети и следует ли переходить по ссылке или игнорировать ее, в зависимости от имени домена.

Предоставляется и список `rules`, в котором содержатся дальнейшие инструкции: по каким ссылкам переходить, а какие — игнорировать (в данном случае с помощью регулярного выражения `.*` мы разрешаем переходить по всем URL).

Помимо заголовка и URL на каждой странице, здесь добавлено извлечение еще пары элементов. С помощью селектора `XPath` извлекается текстовый контент каждой страницы. `XPath` часто используется для извлечения текста, включая содержимое дочерних тегов (например, тега `<a>`, расположенного внутри текстового блока). Если для этого использовать CSS-селектор, то контент всех дочерних тегов будет игнорироваться.

Анализируется также строка с датой последнего изменения из нижнего колонтитула страницы, которая сохраняется в переменной `lastUpdated`.

Чтобы запустить этот пример, нужно перейти в каталог **wikiSpider/wikiSpider** и выполнить следующую команду:

```
$ scrapy runspider articles.py
```



Предупреждение: он будет работать вечно

Этот «паук», как и предыдущий, запускается из командной строки. Однако он не прекратит работу (по крайней мере, будет работать очень, очень долго), пока вы не остановите его выполнение, нажав Ctrl+C или закрыв окно терминала. Прошу, пожалейте сервер «Википедии», не выполняйте данную программу слишком долго.

При запуске этот «паук» проходит по сайту **wikipedia.org**, переходя по всем ссылкам домена **wikipedia.org**, выводя заголовки страниц и игнорируя все внешние ссылки (ведущие на другие сайты):

```
2018-01-21                                01:30:36
[scrapy.spidermiddlewares.offsite]
DEBUG:    Filtered    offsite    request    to
'www.chicagomag.com':
<GET      http://www.chicagomag.com/Chicago-
Magazine/June-2009/
Street-Wise/>
2018-01-21                                01:30:36
[scrapy.downloadermiddlewares.robotstxt]
DEBUG:    Forbidden    by    robots.txt:    <GET
https://en.wikipedia.org/w/
index.php?
title=Adrian_Holovaty&action=edit&section=3>
title is: Ruby on Rails
```

```
URL is:
https://en.wikipedia.org/wiki/Ruby_on_Rails
text is: ['Not to be confused with ', 'Ruby
(programming language)',
'. ', '\n', '\n', 'Ruby on Rails', ... ]
Last updated: 9 January 2018, at 10:32.
```

Пока данный веб-краулер работает неплохо, однако у него есть несколько ограничений. Вместо того чтобы посещать только страницы статей «Википедии», он вполне может переходить на страницы, не относящиеся к статьям, такие как эта:

```
title is: Wikipedia:General disclaimer
```

Рассмотрим внимательнее следующую строку кода, где используются объекты `Scrapy Rule` и `LinkExtractor`:

```
rules = [Rule(LinkExtractor(allow=r'.*'),
callback='parse_items',
follow=True)]
```

В данной строке содержится список объектов `Scrapy Rule`, определяющих правила, по которым фильтруются все найденные ссылки. При наличии нескольких правил каждая ссылка проверяется на соответствие им в порядке их перечисления. Первое подходящее правило — то, которое используется для определения способа обработки ссылки. Если ссылка не соответствует ни одному из правил, то игнорируется.

Правило может быть описано следующими четырьмя аргументами:

- `link_extractor` — единственный обязательный аргумент — объект `LinkExtractor`;

- `callback` — функция обратного вызова, которая должна использоваться для анализа контента страницы;
- `cb_kwargs` — словарь аргументов, передаваемых в функцию обратного вызова. Имеет вид `{arg_name1:arg_value1,arg_name2:arg_value2}` и может быть удобным инструментом для многократного использования одних и тех же функций синтаксического анализа в незначительно различающихся задачах;
- `follow` — указывает на то, хотите ли вы, чтобы веб-краулер обработал также страницы по ссылкам, найденным на данной странице. Если функция обратного вызова не указана, то по умолчанию используется значение `True` (в конце концов, при отсутствии действий с этой страницей логично предположить, что вы по крайней мере захотите применить ее для продолжения сбора данных с сайта). Если предусмотрена функция обратного вызова, то по умолчанию используется значение `False`.

Простой класс `LinkExtractor` предназначен исключительно для распознавания и возврата ссылок, обнаруженных в HTML-контенте страницы на основе предоставленных этому классу правил. Имеет ряд аргументов, которые можно использовать для того, чтобы принимать или отклонять ссылки на основе CSS-селекторов и XPath, тегов (можно искать ссылки не только в тегах `<a>!`), доменов и др.

От класса `LinkExtractor` можно унаследовать свой, в котором можно добавить нужные аргументы. Подробнее об извлечении ссылок см. в документации Scrapy (<https://doc.scrapy.org/en/latest/topics/link-extractors.html>).

Несмотря на всю гибкость свойств класса `LinkExtractor`, самыми распространенными аргументами, которые вы, скорее всего, будете использовать, являются следующие:

- `allow` — разрешить проверку всех ссылок, соответствующих заданному регулярному выражению;
- `deny` — запретить проверку всех ссылок, соответствующих заданному регулярному выражению.

Используя два отдельных класса `Rule` и `LinkExtractor` с общей функцией синтаксического анализа, можно создать «паука», который бы собирал данные в «Википедии», идентифицируя все страницы статей и помечая остальные страницы флагом (`articleMoreRules.py`):

```
from scrapy.contrib.linkextractors import LinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule

class ArticleSpider(CrawlSpider):
    name = 'articles'
    allowed_domains = ['wikipedia.org']
    start_urls = [
        'https://en.wikipedia.org/wiki/'
        'Benevolent_dictator_for_life'
    ]
    rules = [
        Rule(LinkExtractor(allow='^(/wiki/)'
            ((?!:).)*$'),
            callback='parse_items',
            follow=True,
```

```

        cb_kwargs={'is_article': True}),
        Rule(LinkExtractor(allow='.*'),
callback='parse_items',
        cb_kwargs={'is_article': False})
    ]

```

```

    def parse_items(self, response,
is_article):
        print(response.url)

        title =
response.css('h1::text').extract_first()
        if is_article:
            url = response.url

            text =
response.xpath('//div[@id="mw-content-text"]'
                '//text()').extract()

            lastUpdated =
response.css('li#footer-info-lastmod'
                '::text').extract_first()

            lastUpdated =
lastUpdated.replace('This page was '
                    'last edited on ', '')

            print('Title is: {}'.
'.format(title))

            print('title is: {}'.
'.format(title))

            print('text is: {}'.format(text))
        else:
            print('This is not an article:
{}'.format(title))

```

Напомню, что правила применяются к каждой ссылке в том порядке, в котором они представлены в списке. Все страницы статей (страницы, URL которых начинается с `/wiki/` и не содержит двоеточий) передаются в функцию `parse_items` с аргументом `is_article=True`. Все остальные ссылки (не на статьи) передаются в функцию `parse_items` с аргументом `is_article=False`.

Конечно, если вы хотите собирать информацию только со страниц статей и игнорировать все остальные, то такой подход был бы непрактичным. Намного проще игнорировать все страницы, которые не соответствуют структуре URL страниц со статьями, и вообще не определять второе правило (и переменную `is_article`). Однако бывают случаи, когда подобный подход может быть полезен — например, если информация из URL или данные, собранные при краулинге, влияют на способ синтаксического анализа страницы.

Создание объектов `Item`

Мы уже рассмотрели множество способов поиска, синтаксического анализа и краулинга сайтов с помощью `Scrapy`, однако эта платформа также предоставляет полезные инструменты для упорядочения собранных элементов и их хранения в созданных пользователем объектах с четко определенными полями.

Чтобы упорядочить всю собираемую информацию, нужно создать объект `Article`. Определим этот новый элемент с именем `Article` в файле `items.py`.

Когда мы впервые открываем файл `items.py`, он выглядит так:

```
# -*- coding: utf-8 -*-
```

```
# Здесь определяются модели объектов, для
которых выполняется веб-скрапинг.
#
# Подробнее см. документацию:
#
http://doc.scrapy.org/en/latest/topics/items.ht
ml
```

```
import scrapy
```

```
class WikispiderItem(scrapy.Item):
    # Определяем поля для ваших объектов, как
    здесь:
    # name = scrapy.Field()
    pass
```

Заменим этот предоставляемый по умолчанию код-заглушку Item на новый класс Article, который является расширением scrapy.Item:

```
import scrapy
```

```
class Article(scrapy.Item):
    url = scrapy.Field()
    title = scrapy.Field()
    text = scrapy.Field()
    lastUpdated = scrapy.Field()
```

Мы определили три поля, в которые будут собираться данные с каждой страницы: заголовок, URL и дату последнего редактирования страницы.

Если мы собираем данные для страниц разных типов, то нужно определить каждый тип как отдельный класс в `items.py`. Если собираемые объекты `Item` имеют очень большие размеры или если мы захотим перенести в их `Item` дополнительные функции синтаксического анализа, то можно разместить каждый такой объект в отдельном файле. Однако, поскольку наши объекты `Item` невелики, мне нравится хранить их в одном файле.

Обратите внимание на изменения, которые были внесены в класс `ArticleSpider` в файле `articleItems.py`, чтобы можно было создать новый объект `Article`:

```
from scrapy.contrib.linkextractors import
LinkExtractor
from scrapy.contrib.spiders import CrawlSpider,
Rule
from wikiSpider.items import Article

class ArticleSpider(CrawlSpider):
    name = 'articleItems'
    allowed_domains = ['wikipedia.org']
    start_urls =
['https://en.wikipedia.org/wiki/Benevolent'
'_dictator_for_life']
    rules = [
        Rule(LinkExtractor(allow='(/wiki/)
((?!:).)*$'),
            callback='parse_items', follow=True),
    ]

    def parse_items(self, response):
        article = Article()
```

```

        article['url'] = response.url
        article['title'] =
response.css('h1::text').extract_first()
        article['text'] =
response.xpath('//div[@id='
                                '"mw-content-
text"]//text()').extract()
        lastUpdated = response.css('li#footer-
info-lastmod::text'
                                ).extract_first()
        article['lastUpdated'] =
lastUpdated.replace('This page was '
                    'last edited on ', '')
    return article

```

При запуске этого файла с помощью команды:

```
$ scrapy runspider articleItems.py
```

получим обычные отладочные данные Scrapy, а также список всех объектов статей в виде словаря Python:

```

2018-01-21 22:52:38
[scrapy.spidermiddlewares.offsite] DEBUG:
Filtered offsite request to
'wikimediafoundation.org':
<GET
https://wikimediafoundation.org/wiki/Terms_of_U
se>
2018-01-21 22:52:38 [scrapy.core.engine] DEBUG:
Crawled (200)
<GET
https://en.wikipedia.org/wiki/Benevolent_dictat
or_for_life

```

```
#mw-head> (referer:
https://en.wikipedia.org/wiki/Benevolent_
dictator_for_life)
2018-01-21 22:52:38 [scrapy.core.scrapers]
DEBUG: Scraped from
<200
https://en.wikipedia.org/wiki/Benevolent_dictat
or_for_life>
{'lastUpdated': ' 13 December 2017, at 09:26.',
'text': ['For the political term, see ',
        'Benevolent dictatorship',
        '.'],
...}
```

Использование объектов Item в Scrapy не только обеспечивает хорошую упорядоченность кода и представление объектов в удобно читаемом виде. У объектов Item есть множество инструментов для вывода и обработки данных, о которых пойдет речь в следующих разделах.

Вывод объектов Item

В Scrapy объекты Item позволяют определить, какие фрагменты информации из посещенных страниц следует сохранять. Scrapy может сохранять эту информацию различными способами, такими как файлы форматов CSV, JSON или XML, с помощью следующих команд:

```
$ scrapy runspider articleItems.py -o
articles.csv -t csv
$ scrapy runspider articleItems.py -o
articles.json -t json
```

```
$ scrapy runspider articleItems.py -o
articles.xml -t xml
```

Каждая из этих команд запускает веб-скрапер `articleItems` и записывает полученные данные в заданном файле в указанном формате. Если такого файла еще не существует, то он будет создан.

Вы могли заметить, что в «пауке» `ArticleSpider`, созданном в предыдущих примерах, переменная текстовых данных представляет собой не одну строку, а их список. Каждая строка в нем соответствует тексту, расположенному в одном из элементов HTML, тогда как контент тега `<div id="mw-content-text">`, из которого мы собираем текстовые данные, представляет собой множество дочерних элементов.

`Scrapy` хорошо управляет этими усложненными значениями. Например, при сохранении данных в формате CSV списки преобразуются в строки, а запятые экранируются, так что список текстовых фрагментов в формате CSV занимает одну ячейку.

В XML каждый элемент этого списка сохраняется в отдельном дочернем теге:

```
<items>
<item>
  <url>https://en.wikipedia.org/wiki/Benevolent_dictator_for_life</url>
  <title>Benevolent dictator for life</title>
  <text>
    <value>For the political term, see
  </value>
    <value>Benevolent dictatorship</value>
  ...
```



```
        </text>
        <lastUpdated> 13 December 2017, at 09:26.
    </lastUpdated>
</item>
....
```

В формате JSON списки так и сохраняются в виде списков.

И конечно же, мы можем обрабатывать объекты `Item` самостоятельно, записывая их в файл или базу данных любым удобным способом, просто добавив соответствующий код в функцию синтаксического анализа в веб-краулере.

Динамический конвейер

Несмотря на то что `Scrapy` является однопоточной библиотекой, она способна выполнять и обрабатывать несколько запросов асинхронно, благодаря чему работает быстрее, чем веб-скраперы, рассмотренные ранее в этой книге. Впрочем, я продолжаю настаивать на своем мнении: когда речь идет о веб-скрапинге, быстрее не всегда значит лучше.

Веб-сервер сайта, на котором выполняется веб-скрапинг, должен обработать каждый ваш запрос. Важно иметь совесть и трезво оценивать, насколько приемлема создаваемая вами нагрузка на сервер (и не только совесть, но и мудрость — это в ваших собственных интересах, поскольку многие сайты имеют возможность и желание заблокировать то, что посчитают злостным веб-скрапингом). Подробнее об этике веб-скрапинга, а также о важности соответствующей настройки веб-скраперов см. в главе 18.

С учетом всего этого использование динамического конвейера в `Scrapy` позволяет еще более повысить скорость работы веб-скрапера, так как вся обработка данных будет

выполняться в то время, пока веб-скрапер ожидает ответа на запросы, вместо того чтобы дожидаться окончания обработки данных и только потом выполнять очередной запрос. Такой тип оптимизации иногда бывает просто необходим — если обработка данных занимает много времени или требуется выполнить вычисления, создающие значительную нагрузку на процессор.

Чтобы создать динамический конвейер, вернемся к файлу `settings.py` (см. начало главы). В нем содержатся следующие строки кода, заблокированные символами комментариев:

```
# Configure item pipelines
#
# See
http://scrapy.readthedocs.org/en/latest/topics/
item-pipeline.html
#ITEM_PIPELINES = {
#    'wikiSpider.pipelines.WikispiderPipeline':
#    300,
#}
```

Уберем символы комментариев из последних трех строк и получим следующее:

```
ITEM_PIPELINES = {
    'wikiSpider.pipelines.WikispiderPipeline':
    300,
}
```

Таким образом, мы создали класс Python `wikiSpider.pipelines.WikispiderPipeline`, который будет служить для обработки данных, а также указали целое число, соответствующее порядку запуска конвейера при наличии нескольких классов обработки. Здесь можно

использовать любое целое число, однако обычно это цифры от 0 до 1000; конвейер запускается в порядке возрастания.

Теперь нужно добавить класс конвейера в «паука» и переписать нашего исходного «паука» таким образом, чтобы он собирал данные, а конвейер выполнял нелегкую задачу по их обработке. Было бы заманчиво создать в исходном «пауке» метод `parse_items`, который бы возвращал ответ, и позволить конвейеру создавать объект `Article`:

```
def parse_items(self, response):  
    return response
```

Однако фреймворк Scrapy такого не допускает: должен возвращаться объект `Item` (или `Article`, который является расширением `Item`). Итак, сейчас назначение `parse_items` состоит в том, чтобы извлечь необработанные данные и обработать по минимуму — лишь бы можно было передать их в конвейер:

```
from scrapy.contrib.linkextractors import  
LinkExtractor  
from scrapy.contrib.spiders import CrawlSpider,  
Rule  
from wikiSpider.items import Article  
  
class ArticleSpider(CrawlSpider):  
    name = 'articlePipelines'  
    allowed_domains = ['wikipedia.org']  
    start_urls =  
    ['https://en.wikipedia.org/wiki/Benevolent_dict  
ator_for_life']  
    rules = [
```

```

        Rule(LinkExtractor(allow='(/wiki/)
((?!:).)*$'),
                callback='parse_items',
follow=True),
    ]

```

```

def parse_items(self, response):
    article = Article()
    article['url'] = response.url
    article['title'] =
response.css('h1::text').extract_first()
    article['text'] =
response.xpath('//div[@id='
                                '"mw-content-
text"]//text()').extract()
    article['lastUpdated'] =
response.css('li#'
                                'footer-info-
lastmod::text').extract_first()
    return article

```

Этот файл хранится в репозитории GitHub под именем `articlePipelines.py`.

Конечно, теперь нужно связать файл `pipelines.py` с измененным «пауком», добавив конвейер. При первоначальной инициализации проекта Scrapy был создан файл `wikiSpider/wikiSpider/pipelines.py`:

```

# -*- coding: utf-8 -*-

# Define your item pipelines here
#

```

```
# Don't forget to add your pipeline to the
ITEM_PIPELINES setting
```

```
# See:
http://doc.scrapy.org/en/latest/topics/item-
pipeline.html
```

```
class WikispiderPipeline(object):
    def process_item(self, item, spider):
        return item
```

Этот класс-заглушку следует заменить нашим новым кодом конвейера. В предыдущих разделах мы собирали два поля в необработанном формате — `lastUpdated` (плохо отформатированный строковый объект, представляющий дату) и `text` («грязный» массив строковых фрагментов), — но для них можно использовать дополнительную обработку.

Вместо кода-заглушки В `wikiSpider/wikiSpider/pipelines.py` поставим следующий код:

```
from datetime import datetime
from wikiSpider.items import Article
from string import whitespace

class WikispiderPipeline(object):
    def process_item(self, article, spider):
        dateStr = article['lastUpdated']
        article['lastUpdated'] =
        article['lastUpdated']
        .replace('This page was last edited
on', '')
```

```

        article['lastUpdated'] =
article['lastUpdated'].strip()
        article['lastUpdated'] =
datetime.strptime(
    article['lastUpdated'], '%d %B %Y, at
%H:%M.')
        article['text'] = [line for line in
article['text']
    if line not in whitespace]
        article['text'] =
''.join(article['text'])
    return article

```

У класса WikispiderPipeline есть метод `process_item`, который принимает объект `Article`, преобразует строку `lastUpdated` в объект `datetime` Python, очищает текст и трансформирует его из списка строк в одну строку.

Метод `process_item` является обязательным для любого класса конвейера. В Scrapy этот метод используется для асинхронной передачи объектов `Item`, собранных «пауком». Анализируемый объект `Article`, который возвращается в данном случае, будет сохранен или выведен Scrapy, если, например, мы решим выводить элементы в формате JSON или CSV, как это было сделано в предыдущем разделе.

Теперь можно выбрать, где обрабатывать данные: в методе `parse_items` в «пауке» или в методе `process_items` в конвейере.

В файле `settings.py` можно создать несколько конвейеров, каждый из которых будет выполнять свою задачу. Однако Scrapy передает все элементы, независимо от их типа, во все конвейеры по порядку. Синтаксический анализ элементов конкретных типов лучше реализовать в «пауке» и

только потом передавать данные в конвейер. Но если этот анализ занимает много времени, то можно переместить его в конвейер (где он будет выполняться асинхронно) и добавить проверку типа элемента:

```
def process_item(self, item, spider):  
    if isinstance(item, Article):  
        # обработка объектов Article
```

При написании проектов Scrapy, особенно крупных, важно учитывать, где и какую обработку вы собираетесь делать.

Ведение журнала Scrapy

Отладочная информация, генерируемая Scrapy, бывает полезна, однако, как вы могли заметить, часто чересчур многословна. Вы можете легко настроить уровень ведения журнала, добавив в файл `settings.py` проекта Scrapy следующую строку:

```
LOG_LEVEL = 'ERROR'
```

В Scrapy принята стандартная иерархия уровней ведения журнала:

- CRITICAL;
- ERROR;
- WARNING;
- DEBUG;
- INFO.

В случае уровня ведения журнала ERROR будут отображаться только события типов CRITICAL и ERROR; при ведении журнала на уровне INFO будут отображаться все события и т.д.

Кроме управления ведением журнала через файл `settings.py`, также можно управлять тем, куда попадают записи из командной строки. Чтобы они выводились не на терминал, а в отдельный файл журнала, нужно указать этот файл при запуске Scrapy из командной строки:

```
$ scrapy crawl articles -s LOG_FILE=wiki.log
```

Эта команда создает в текущем каталоге новый файл журнала (если такого еще нет) и делает в него все журнальные записи, оставляя терминал только для данных, выводимых операторами Python, которые вы сами добавили в программу.

Дополнительные ресурсы

Scrapy — мощный инструмент, способный решить многие проблемы веб-краулинга. Он автоматически собирает все URL и проверяет их на соответствие заданным правилам; гарантирует, что все адреса являются уникальными, при необходимости нормализует относительные URL и рекурсивно переходит на страницы более глубоких уровней.

В этой главе мы лишь поверхностно рассмотрели возможности Scrapy; я призываю вас заглянуть в документацию по Scrapy (<https://doc.scrapy.org/en/latest/news.html>), а также почитать книгу *Learning Scrapy* Димитриоса Кузис-Лукаса (Dimitrios Kouzis-Loukas) (издательство O'Reilly) (<http://shop.oreilly.com/product/9781784399788.do>), в которой содержится исчерпывающее описание данного фреймворка.

Scrapy — чрезвычайно обширная библиотека с множеством функций. Они хорошо сочетаются между собой, но во многом перекрывают друг друга, благодаря чему разработчик может легко создать собственный стиль. Если есть то, что вы хотели бы сделать с помощью Scrapy, но о чем не упоминается в этой главе, — скорее всего, такой способ — или даже несколько — существует!

Глава 6. Хранение данных

Несмотря на то что вывод данных на терминал доставляет массу удовольствия, он не слишком полезен для агрегирования и анализа данных. В большинстве случаев, чтобы работа удаленных веб-скраперов приносила пользу, необходимо иметь возможность сохранять получаемую от них информацию.

В данной главе мы рассмотрим три основных метода управления данными, которых вам будет достаточно практически для любого случая. Хотите запустить серверную часть сайта или создать собственный API? Тогда, возможно, веб-скраперы должны записывать информацию в базу данных. Нужен быстрый и простой способ собирать документы из Интернета и сохранять их на жестком диске? Вероятно, для этого стоит создать файловый поток. Требуется время от времени отправлять предупреждения или раз в день передавать сводные данные? Тогда отправьте себе письмо по электронной почте!

Возможность взаимодействовать с большими объемами данных и хранить их невероятно важна не только для веб-скрапинга, но и практически для любого современного программного обеспечения. В сущности, информация, представленная в данной главе, необходима для реализации многих примеров, рассматриваемых в следующих разделах книги. Если вы не знакомы с автоматизированным хранением данных, то я настоятельно рекомендую вам хотя бы просмотреть эту главу.

Медиафайлы

Есть два основных способа хранения медиафайлов: сохранить ссылку или скачать сам файл. Чтобы сохранить файл по ссылке, нужно сохранить URL, по которому находится этот файл.

Данный способ имеет следующие преимущества:

- веб-скраперы работают намного быстрее и требуют гораздо меньшей пропускной способности, если им не нужно скачивать файлы;
- сохраняя только URL, вы экономите место на своем компьютере;
- проще написать код, который не скачивает файлы, а лишь сохраняет их URL;
- избегая скачивания больших файлов, вы уменьшаете нагрузку на сервер, на котором размещен сканируемый ресурс.

Но есть и недостатки.

- Встраивание URL в ваш сайт или приложение называется *горячими ссылками*; это легкий способ нажать себе неприятности в Интернете.
- Вы вряд ли захотите размещать на чужих серверах мультимедийные ресурсы, которые используются в ваших приложениях.
- Файл, размещенный по определенному URL, может измениться. Это может привести к нелепой ситуации, если, к примеру, разместить картинку, расположенную по этому адресу, в публичном блоге. Если сохранить URL, рассчитывая позже скачать и изучить сам файл, то однажды данный файл

могут удалить или заменить чем-нибудь совершенно неподходящим.

- Настоящие браузеры не просто получают HTML-код страницы и двигаются дальше. Они также скачивают все ресурсы, необходимые для отображения данной страницы. Скачивая файлы, вы делаете действия веб-скрапера больше похожими на действия человека, который просматривает сайт, — это может стать преимуществом.

Принимая решение, сохранять ли сам файл или только его URL, следует спросить себя: намерены ли вы просматривать или читать этот файл чаще чем один-два раза? Или же со временем у вас накопится целая база данных с такими файлами, которая большую часть времени будет лежать на диске мертвым грузом? Если вероятнее второй вариант, то лучше просто сохранить URL. Если же первый, то читайте эту главу дальше!

В библиотеке `urllib`, используемой для извлечения контента веб-страниц, также содержатся функции для получения содержимого файлов. Следующая программа с помощью `urllib.request.urlretrieve` скачивает изображения с удаленного URL:

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
imageLocation = bs.find('a', {'id': 'logo'}).find('img')['src']
```

```
urlretrieve (imageLocation, 'logo.jpg')
```

Эта программа скачивает картинку с логотипом, размещенную по адресу **<http://pythonscraping.com>**, и сохраняет ее под именем `logo.jpg` в том же каталоге, из которого запускается скрипт.

Данный код отлично справляется с задачей, когда нужно скачать только один файл с заранее известным именем и расширением. Но большинство веб-скраперов не ограничиваются скачиванием одного файла. Следующая программа скачивает с главной страницы **<http://pythonscraping.com>** все файлы, на которые указывает ссылка в атрибуте `src` любого тега, если это внутренняя ссылка:

```
import os
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

downloadDirectory = 'downloaded'
baseUrl = 'http://pythonscraping.com'

def getAbsoluteURL(baseUrl, source):
    if source.startswith('http://www.'):
        url = 'http://{0}'.format(source[11:])
    elif source.startswith('http://'):
        url = source
    elif source.startswith('www.'):
        url = source[4:]
        url = 'http://{0}'.format(source)
    else:
```

```

        url = '{}/{ {}'.format(baseUrl, source)
    if baseUrl not in url:
        return None
    return url

def getDownloadPath(baseUrl, absoluteUrl,
downloadDirectory):
    path = absoluteUrl.replace('www.', '')
    path = path.replace(baseUrl, '')
    path = downloadDirectory+path
    directory = os.path.dirname(path)

    if not os.path.exists(directory):
        os.makedirs(directory)

    return path

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
downloadList = bs.find_all(src=True)

for download in downloadList:
    fileUrl = getAbsoluteURL(baseUrl,
download['src'])
    if fileUrl is not None:
        print(fileUrl)

urlretrieve(fileUrl, getDownloadPath(baseUrl,
fileUrl, downloadDirectory))

```



Будьте осторожны!

Помните, что бывает с теми, кто скачивает неизвестные файлы из Интернета? Представленный выше скрипт скачивает на жесткий диск вашего компьютера все подряд, включая случайные скрипты bash, .exe-файлы и другие потенциально вредоносные программы.

Считаете, что с вами не случится ничего плохого, поскольку вы никогда не запускаете ничего из того, что попало в папку Загрузки? Тогда вы просто напрашиваетесь на неприятности, особенно запуская программу с правами администратора. Что произойдет, если на каком-нибудь сайте вам попадется файл, который отправляет сам себя в папку `../../usr/bin/python`? В следующий раз, запустив скрипт Python из командной строки, вы можете случайно установить на свой компьютер вредоносную программу!

Рассмотренная выше программа написана исключительно для примера; ее нельзя устанавливать в рабочей среде без более тщательной проверки имен файлов и ее следует запускать только из учетной записи с ограниченными правами. И самое главное: помните, что нам всегда помогут резервное копирование файлов, хранение конфиденциальной информации вне жесткого диска и здравый смысл.

В этом скрипте с помощью лямбда-функции (см. главу 2) мы выбираем на главной странице сайта все теги, имеющие атрибут `src`, а затем очищаем и нормализуем URL, чтобы получить абсолютный путь для каждого скачиваемого файла (и гарантированно отсеять все внешние ссылки). Затем каждый файл скачивается по своей ссылке и помещается на компьютер в локальную папку `downloaded`.

Обратите внимание: для быстрой ссылки на каталог, в который помещаются результаты каждого скачивания, а также для создания при необходимости недостающих каталогов используется Python-модуль `os`. Он играет роль интерфейса между Python и операционной системой, позволяя ей управлять путями к файлам, создавать каталоги, получать информацию о работающих процессах и переменных среды и т.п.

Хранение данных в формате CSV

CSV (comma-separated values — «значения, разделенные запятыми») — один из самых популярных форматов файлов для хранения табличных данных. Благодаря своей простоте этот формат поддерживается Microsoft Excel и многими другими приложениями. Ниже показан пример совершенно правильного содержимого CSV-файла:

```
fruit,cost
apple,1.00
banana,0.30
pear,1.25
```

Как и в Python, здесь важны разделители: строки отделяются друг от друга символом новой строки, а столбцы внутри строки — запятыми (отсюда и название «разделенные

запятыми»). В других вариантах CSV-файлов (иногда называемых *character-separated values* — «файлы значений, разделенных символами») для разделения строк используются табуляции и другие символы, но эти форматы менее распространены и поддерживаются не так широко.

Если вы хотите скачивать CSV-файлы прямо из Интернета и хранить их локально, не анализируя и не изменяя, то данный раздел вам не нужен. Скачайте эти файлы, как любые другие, и сохраните их в формате CSV, используя методы, описанные в предыдущем разделе.

Python позволяет легко изменять и даже создавать CSV-файлы с нуля с помощью библиотеки `csv`:

```
import csv

csvFile = open('test.csv', 'w+')
try:
    writer = csv.writer(csvFile)
    writer.writerow(('number', 'number plus 2',
'number times 2'))
    for i in range(10):
        writer.writerow( (i, i+2, i*2))
finally:
    csvFile.close()
```

На всякий случай предупреждаю: создание файла в Python — практически безошибочная процедура. При отсутствии файла `test.csv` Python автоматически создаст его (но не каталог). Если же такой файл уже есть, то Python перезапишет `test.csv`, внося в него новые данные.

После выполнения этой программы должен получиться следующий CSV-файл:

```
number,number plus 2,number times 2
0,2,0
1,3,2
2,4,4
...
```

Одна из самых популярных задач веб-скрапинга — скачать HTML-таблицу и сохранить ее в виде CSV-файла. Сравнение текстовых редакторов в «Википедии» (https://en.wikipedia.org/wiki/Comparison_of_text_editors) представляет собой весьма сложную HTML-таблицу с выделением ячеек разными цветами, а также со ссылками, сортировкой и другим HTML-мусором, который необходимо будет вычистить, прежде чем записывать данные в формат CSV. Активно используя BeautifulSoup и функцию `get_text()`, мы можем сделать это менее чем за 20 строк кода:

```
import csv
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://en.wikipedia.org/wiki/'
               'Comparison_of_text_editors')
bs = BeautifulSoup(html, 'html.parser')
# Основная сравнительная таблица сейчас
# является первой на странице.
table = bs.find_all('table',
                    {'class': 'wikitable'})[0]
rows = table.find_all('tr')

csvFile = open('editors.csv', 'wt+')
```

```
writer = csv.writer(csvFile)
    try:
        for row in rows:
            csvRow = []
            for cell in row.find_all(['td',
'th']):
                csvRow.append(cell.get_text())
            writer.writerow(csvRow)
    finally:
        csvFile.close()
```



Более простой способ получить отдельную таблицу

Данный скрипт отлично интегрируется в веб-скрапер, если нужно преобразовать много HTML-таблиц в CSV-файлы или объединить таблицы в один CSV-файл. Но если нужно сделать это только один раз, то имеется средство получше: копирование и вставка. Выделив и скопировав весь контент HTML-таблицы и вставив его в файл Excel или Google Docs, мы как раз получим CSV-файл, не запуская скрипт!

В результате должен получиться правильно отформатированный CSV-файл, сохраненный на локальном компьютере под именем `editors.csv`.

MySQL

MySQL (официально произносится как «май эс-кью-эль», хотя многие говорят «май сиквел») — на сегодняшний день самая популярная система управления реляционными базами данных с открытым исходным кодом. Это несколько необычная ситуация: проект с открытым исходным кодом успешно конкурирует с большими программными продуктами. Однако популярность MySQL исторически была связана с двумя другими крупнейшими системами управления базами данных с закрытым исходным кодом: Microsoft SQL Server и СУБД Oracle.

MySQL популярна не без причин. Ее можно смело выбирать для большинства приложений. Это масштабируемая, надежная и полнофункциональная СУБД, используемая ведущими сайтами, в том числе YouTube⁵, Twitter⁶ и Facebook⁷.

Благодаря повсеместному распространению, цене («бесплатно» — очень хорошая цена) и удобству применения «как есть», без дополнительной настройки, MySQL — просто фантастическая база данных для веб-скраперов, и в этой книге мы будем использовать именно ее.

Что такое реляционная база данных

Реляционные данные — это данные, между которыми установлены некие отношения (от англ. relation — «отношение», «зависимость», «связь»). Я рада, что мы прояснили это!

Ладно, шучу. Говоря о реляционных данных, специалисты по информатике имеют в виду данные, которые не существуют сами по себе — у них есть свойства, связывающие их с

другими данными. Например, «пользователь А идет учиться в учебное заведение Б», причем пользователя А можно найти в таблице «Пользователи» базы данных, а учебное заведение Б – в таблице «Учебные заведения» этой же базы.

В данной главе вы познакомитесь с моделированием различных типов отношений и эффективным хранением данных в MySQL (или любой другой реляционной базе данных).

Установка MySQL

При отсутствии знаний о MySQL установка базы данных может показаться несколько страшноватой (если вы хорошо знакомы с MySQL, то можете спокойно пропустить данный раздел). На самом деле это не сложнее, чем установка практически любого другого программного обеспечения. По сути, MySQL работает с набором файлов, в которых содержатся данные. Эти файлы размещены на сервере или локальном компьютере и вмещают всю информацию, хранящуюся в базе данных. Кроме того, программный уровень MySQL обеспечивает удобный способ взаимодействия с данными через интерфейс командной строки. Например, следующая команда просматривает файлы данных и возвращает список всех пользователей базы данных по имени Ryan:

```
SELECT * FROM users WHERE firstname = "Ryan"
```

Если вы используете дистрибутив Linux на основе Debian (или другую операционную систему с командой `apt-get`), то

можете легко установить MySQL с помощью следующей команды:

```
$ sudo apt-get install mysql-server
```

Просто присматривайте за процессом установки, согласитесь со всеми требованиями к памяти и введите пароль для нового пользователя **root** при появлении соответствующей подсказки.

В macOS и Windows все немного сложнее. Перед скачиванием пакета необходимо создать учетную запись Oracle, если это не было сделано ранее.

Если вы используете macOS, то сначала нужно скачать дистрибутив (<http://dev.mysql.com/downloads/mysql/>).

Чтобы загрузить файл, выберите пакет .dmg и зарегистрируйтесь в системе под существующей учетной записью Oracle или создайте ее. Затем запустится довольно простой мастер установки (рис. 6.1).

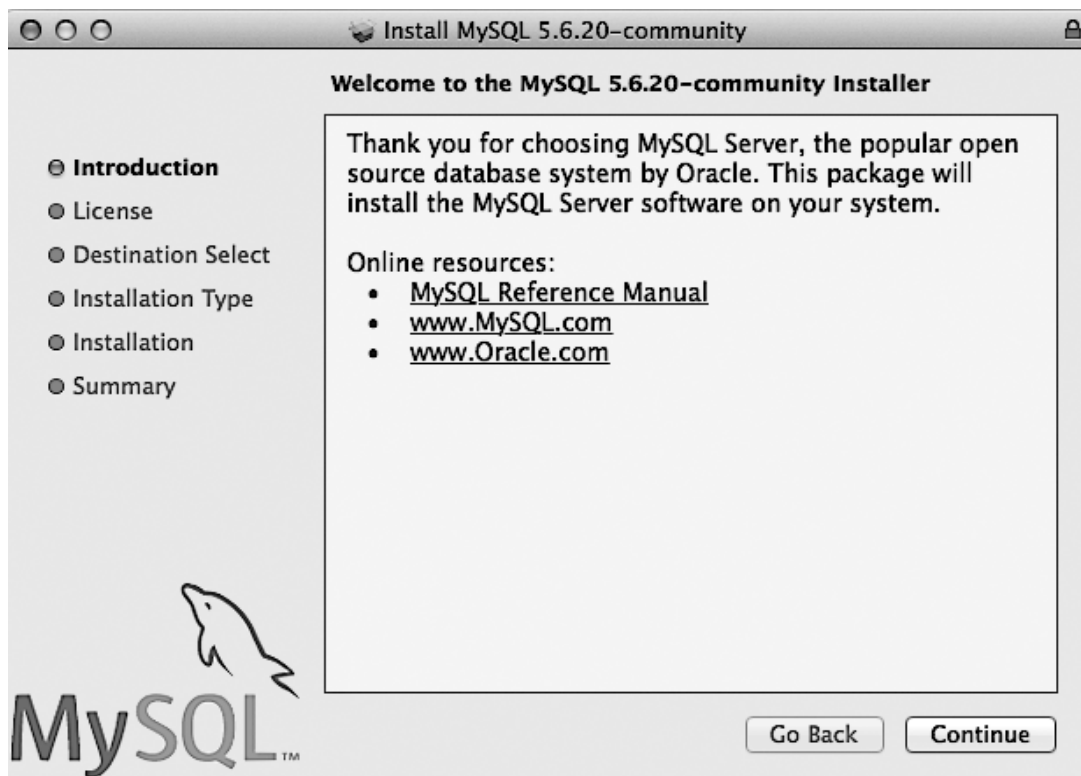


Рис. 6.1. Мастер установки MySQL в macOS

Вам должно быть достаточно варианта установки, предлагаемого по умолчанию. В этой книге предполагается, что вы установили MySQL именно так.

Если скачивание и запуск мастера установки кажется вам несколько утомительным занятием и вы используете Mac, то всегда можно обратиться к менеджеру пакетов Homebrew (<http://brew.sh/>). После установки Homebrew можно установить MySQL следующим образом:

```
$ brew install mysql
```

Homebrew — отличный проект с открытым исходным кодом и с хорошей интеграцией пакетов Python. Большинство сторонних модулей Python, используемых в данной книге, легко устанавливаются с помощью Homebrew. Если он у вас еще не установлен, то настоятельно рекомендую сделать это!

После установки MySQL на macOS можно запустить сервер MySQL следующим образом:

```
$ cd /usr/local/mysql
$ sudo ./bin/mysqld_safe
```

В Windows установка и запуск MySQL немного сложнее, однако, на наше счастье, удобный мастер установки (<http://dev.mysql.com/downloads/windows/installer/>) упрощает этот процесс. После загрузки он проведет вас по операциям, которые необходимо выполнить (рис. 6.2).

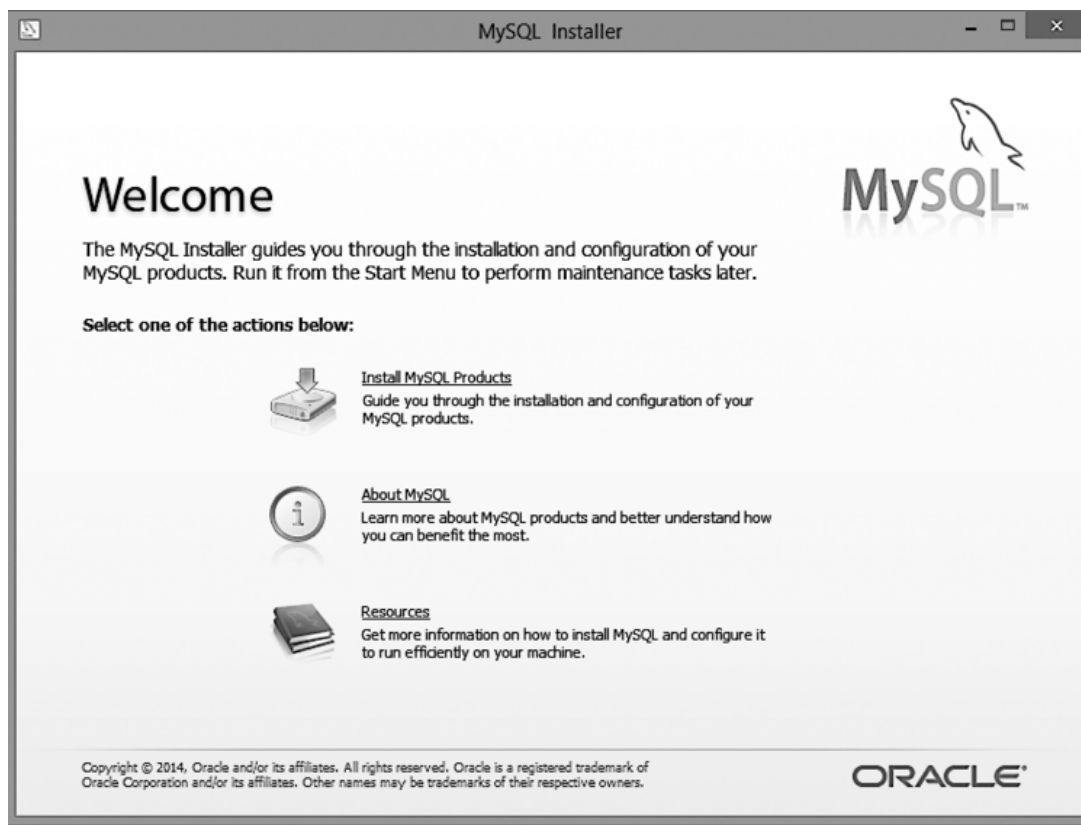


Рис. 6.2. Мастер установки MySQL в Windows

Здесь тоже можно установить MySQL, используя параметры по умолчанию, за единственным исключением: на странице **Setup Type** (Тип установки) я рекомендую выбрать вариант **Server Only** (Только сервер), чтобы не устанавливать множество

дополнительных программ и библиотек Microsoft. После этого можно использовать параметры установки по умолчанию и следовать инструкциям для запуска сервера MySQL.

Несколько основных команд

После запуска сервера MySQL появляется множество вариантов взаимодействия с базой данных. Есть много программных инструментов, играющих роль посредников, вследствие чего вам не обязательно иметь дело именно с командами MySQL (или по крайней мере не придется делать это часто). Такие инструменты, как phpMyAdmin и MySQL Workbench, позволяют без труда просматривать, сортировать и вставлять данные. Но все равно важно уметь использовать командную строку.

За исключением имен переменных, MySQL нечувствительна к регистру; например, `SELECT` — то же самое, что и `sElEcT`. Однако по соглашению все ключевые слова в MySQL принято писать заглавными буквами. И наоборот, большинство разработчиков предпочитают при именовании таблиц и баз данных использовать строчные буквы, хотя этот стандарт часто игнорируется.

Когда вы впервые войдете в MySQL, там не будет базы данных, в которую можно было бы вставить данные. Но ее можно создать:

```
> CREATE DATABASE scraping;
```

Поскольку в каждом экземпляре MySQL может содержаться несколько баз данных, прежде чем начать взаимодействовать с БД, необходимо указать MySQL, какую именно базу вы хотите использовать:

```
> USE scraping;
```

С этого момента (и пока вы не закроете соединение с MySQL или не переключитесь на другую базу) все вводимые команды будут выполняться для созданной нами базы данных `scraping`.

На первый взгляд все довольно просто. Наверное, так же легко можно создать и таблицу в базе данных? Попробуем создать таблицу для хранения коллекции веб-страниц, собранных веб-скрапером:

```
> CREATE TABLE pages;
```

И получим ошибку:

```
ERROR 1113 (42000): A table must have at least  
1 column
```

В отличие от базы данных, которая может существовать без таблиц, таблица MySQL не может существовать без столбцов. Чтобы определить столбцы в MySQL, необходимо перечислить их в виде списка, разделенного запятыми, в скобках после оператора `CREATETABLE<имя_таблицы>`:

```
> CREATE TABLE pages (id BIGINT(7) NOT NULL  
  AUTO_INCREMENT,  
  title VARCHAR(200), content VARCHAR(10000),  
  created TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY(id));
```

Каждое определение столбца состоит из трех частей:

- имя (`id`, `title`, `created` и т.д.);
- тип переменной (`BIGINT(7)`, `VARCHAR`, `TIMESTAMP`);

- также можно указать различные дополнительные атрибуты (NOTNULLAUTO_INCREMENT).

В конце списка столбцов нужно указать *ключ* таблицы. В MySQL ключи обеспечивают удобное представление контента таблицы для быстрого поиска. Далее в главе я покажу, как использовать эти ключи для ускорения работы базы данных. Пока просто запомните, что наилучшим вариантом ключа таблицы, как правило, является столбец идентификаторов (id).

После выполнения этого запроса можно воспользоваться командой DESCRIBE и посмотреть, как выглядит структура таблицы:

```
> DESCRIBE pages;
+-----+-----+-----+-----+-----+
| Field      | Type                | Null | Key |
| Default    | Extra               |      |     |
+-----+-----+-----+-----+-----+
| id         | bigint(7)          | NO   | PRI |
| NULL      | auto_increment     |      |     |
| title     | varchar(200)       | YES  |     |
| NULL      |                    |      |     |
| content   | varchar(10000)     | YES  |     |
| NULL      |                    |      |     |
| created   | timestamp          | NO   |     |
| CURRENT_TIMESTAMP |                |      |     |
+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

Конечно, эта таблица все еще пуста. Чтобы вставить в таблицу `pages` тестовые данные, можно воспользоваться следующей командой:

```
> INSERT INTO pages (title, content) VALUES  
("Test page title",  
"This is some test page content. It can be up  
to 10,000 characters  
long.");
```

Обратите внимание: хоть эта таблица и имеет четыре столбца (`id`, `title`, `content`, `created`), для вставки строки нужно определить только два из них (`title` и `content`). Дело в том, что столбец `id` получается автоматически, путем приращения (при вставке новой строки MySQL автоматически прибавляет 1 к идентификатору предыдущей строки), так что нам об этом заботиться практически не приходится. В столбец `timestamp` по умолчанию записывается текущее время.

Безусловно, мы *можем* задать эти значения явно:

```
> INSERT INTO pages (id, title, content,  
created) VALUES (3,  
"Test page title",  
"This is some test page content. It can be up  
to 10,000 characters  
long.", "2014-09-21 10:25:32");
```

При отсутствии в базе данных столбца, у которого `id` равен заданному целому числу, подобное переопределение вполне будет работать. Однако, как правило, поступать так — плохая идея; лучше доверить MySQL автоматическое заполнение столбцов `id` и `timestamp`, если только у вас нет веских причин поступить иначе.

Теперь, когда в нашей таблице появились данные, мы можем использовать различные методы их выбора. Вот несколько примеров операторов SELECT:

```
> SELECT * FROM pages WHERE id = 2;
```

Этот оператор дает MySQL такую команду: «Выбрать все страницы, у которых id равен 2». Звездочка (*) — подстановочный символ, он требует возвратить все строки таблицы, для которых условие (WHERE id=2) является истинным. Данный оператор возвращает вторую строку таблицы либо пустой результат, если в таблице нет строки, у которой id равен 2. Например, следующий запрос без учета регистра возвращает все строки, в которых поле title содержит фрагмент *test* (символ % в строках MySQL играет роль подстановочного):

```
> SELECT * FROM pages WHERE title LIKE  
"%test%";
```

А как быть, если в нашей таблице много столбцов и мы хотим, чтобы оператор возвращал только определенный набор данных? Тогда вместо выбора всех столбцов можно написать, например, так:

```
> SELECT id, title FROM pages WHERE content  
LIKE "%page content%";
```

Этот оператор возвращает только id и title для всех строк, в которых столбец content содержит фразу *page content*.

Синтаксис операторов DELETE почти такой же, как у SELECT:

```
> DELETE FROM pages WHERE id = 1;
```

Поэтому, особенно при работе с важными базами данных, которые нелегко восстановить, рекомендуется перед выполнением DELETE сначала выполнить аналогичный оператор SELECT (в данном случае SELECT*FROMpagesWHEREid=1), с целью убедиться, что выбираются только те строки, которые вы хотите удалить, а затем заменить SELECT* на DELETE. Многие программисты рассказывают ужасные истории о том, как они второпях ошиблись в условии или, что еще хуже, совершенно неправильно написали оператор DELETE и полностью разрушили клиентские данные. Не позволяйте этому случиться с вами!

Аналогичные меры предосторожности следует принимать и с операторами UPDATE:

```
> UPDATE pages SET title="A new title",  
content="Some new content" WHERE id=2;
```

В данной книге мы будем иметь дело только с основными операторами MySQL — простейшим выбором, вставкой и изменением. Если вы хотите узнать больше о командах и приемах этого мощного инструмента для работы с базами данных, то рекомендую прочитать книгу *MySQL Cookbook* Поля Дюбуа (Paul DuBois)⁸ (издательство O'Reilly) (<http://shop.oreilly.com/product/0636920032274.do>).

Интеграция с Python

К сожалению, в MySQL нет встроенной поддержки Python. Однако как для Python 2.x, так и для Python 3.x есть множество библиотек с открытым исходным кодом, позволяющих взаимодействовать с базами данных MySQL. Одной из самых

популярных таких библиотек является PyMySQL (<https://pypi.python.org/pypi/PyMySQL>).

Когда писалась эта книга, последней версией PyMySQL была 0.6.7, которая устанавливается с помощью `pip` следующим образом:

```
$ pip install PyMySQL
```

Вы также можете скачать и установить данную библиотеку из ее источника — это бывает удобно, если хотите использовать определенную версию библиотеки:

```
$ curl -L https://pypi.python.org/packages/source/P/PyMySQL/PyMySQL-0.6.7.tar.gz\ | tar xz
$ cd PyMySQL-PyMySQL-f953785/
$ python setup.py install
```

После установки библиотеки пакет PyMySQL должен стать доступным автоматически. При работающем локальном сервере MySQL должен успешно выполняться следующий скрипт (не забудьте указать пароль пользователя **root** для вашей базы данных):

```
import pymysql
conn = pymysql.connect(host='127.0.0.1',
unix_socket='/tmp/mysql.sock',
                                user='root',
passwd=None, db='mysql')
cur = conn.cursor()
cur.execute('USE scraping')
cur.execute('SELECT * FROM pages WHERE id=1')
```

```
print(cur.fetchone())  
cur.close()  
conn.close()
```

В этом примере появляются два новых типа объектов: объект-соединение (`conn`) и объект-курсор (`cur`).

Модели соединения и курсора широко применяются при программировании баз данных, хотя поначалу некоторые пользователи с трудом их различают. Понятно, что объект-соединение отвечает за само соединение с базой, а также за отправку информации из нее, обработку откатов (когда необходимо прервать выполнение одного или нескольких запросов и вернуть базу данных в исходное состояние) и создание новых объектов-курсоров.

У соединения может быть несколько курсоров. Курсор отслеживает определенную информацию *о состоянии*: например, то, какая база данных используется. Если у нас есть несколько баз и нужно записать информацию во все, то для обработки такого запроса можно создать ряд курсоров. Кроме того, курсор содержит результаты последнего выполненного запроса. Чтобы получить доступ к этой информации, можно воспользоваться функциями для курсора, такими как `cur.fetchone()`.

Важно закрывать и курсор, и соединение после окончания их использования. Игнорирование этого требования может привести к *утечкам соединений* — накоплению незакрытых неиспользуемых соединений, из-за которых программа не может закрыться, поскольку ей кажется, что эти соединения все еще задействованы. Именно подобные ошибки приводят к постоянному отключению баз данных (мне довелось допускать и исправлять много ошибок утечки соединений), поэтому не забывайте закрывать соединения!

Прежде всего вы, вероятно, захотите реализовать сохранение результатов веб-скрапинга в базе данных. Посмотрим, как это можно сделать, используя предыдущий пример — веб-скрапер «Википедии».

Работа с текстом в формате Unicode при просмотре веб-страниц может оказаться непростой задачей. По умолчанию MySQL не поддерживает Unicode. К счастью, мы можем активировать эту функцию (только помните, что тогда увеличится размер базы данных). Поскольку в «Википедии» нам встретится множество колоритных символов, самое время сообщить базе, что ей придется иметь дело с текстом в формате Unicode:

```
ALTER DATABASE scraping CHARACTER SET = utf8mb4
COLLATE = utf8mb4_unicode_ci;
ALTER TABLE pages CONVERT TO CHARACTER SET
utf8mb4 COLLATE utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE title title
VARCHAR(200) CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE content content
VARCHAR(10000) CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;
```

Эти четыре строки изменяют кодировку, используемую в базе данных по умолчанию, для таблицы и обоих ее столбцов, с utf8mb4 (технически это тоже Unicode, но с печально известной ужасной поддержкой большинства символов Unicode) на utf8mb4_unicode_ci.

Вы поймете, что у вас все получилось, если попыдаете вставить в поле title или content несколько умляутов или китайских иероглифов и это удастся сделать без ошибок.

Теперь, когда база данных готова принять все то, на что способна «Википедия», можно выполнить следующий код:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import pymysql
import re

conn = pymysql.connect(host='127.0.0.1',
                        unix_socket='/tmp/mysql.sock',
                        user='root',
                        passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute('USE scraping')

random.seed(datetime.datetime.now())

def store(title, content):
    cur.execute('INSERT INTO pages (title,
    content) VALUES '
                '("%s", "%s")', (title, content))
    cur.connection.commit()

def getLinks(articleUrl):
    html = urlopen('http://en.wikipedia.org'+articleUrl)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    content = bs.find('div', {'id':'mw-content-
    text'}).find('p')
```

```

        .get_text()
        store(title, content)
        return bs.find('div',
{'id': 'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))

links = getLinks('/wiki/Kevin_Bacon')
try:
    while len(links) > 0:
        newArticle = links[random.randint(0,
len(links)-1)].attrs['href']
        print(newArticle)
        links = getLinks(newArticle)
finally:
    cur.close()
    conn.close()

```

Здесь следует отметить несколько моментов. Во-первых, в строку соединения с базой данных добавляется условие `charset='utf8'`. Так мы сообщаем соединению, что всю информацию в базу следует отправлять в формате UTF-8 (само собой разумеется, что БД уже должна быть настроена соответствующим образом).

Во-вторых, обратите внимание на то, что появилась функция `store`. Она принимает две строковые переменные: `title` и `content` — и вставляет их в оператор `INSERT`, который выполняется объектом-курсором и затем принимается объектом-соединением этого курсора. Перед нами отличный пример разделения курсора и соединения: курсор хранит информацию о базе данных и ее контексте, однако использует соединение, чтобы через него отправлять информацию и вставлять данные в базу.

Наконец, мы видим, что в конце основного цикла программы добавился оператор `finally`. Он гарантирует, что независимо от того, завершится ли программа нормально, будет ли прервана или во время ее выполнения возникнут исключения (а с учетом вечного беспорядка в Интернете будьте уверены, что исключения станут возникать постоянно), курсор и соединение всегда будут закрываться непосредственно перед завершением программы. Всегда использовать оператор `try...finally` при веб-скрапинге и с открытым соединением с базой данных — хорошая идея.

PyMySQL — не слишком большой пакет, но у него так много полезных функций, что все они не поместятся в этой книге. Подробнее о них см. в документации на сайте PyMySQL (<https://pymysql.readthedocs.io/en/latest/>).

Приемы и рекомендуемые методики работы с базами данных

Есть люди, посвятившие всю свою карьеру изучению, настройке и изобретению новых баз данных. Я к ним не отношусь, и моя книга не об этом. Однако в данной области, как и во многих других разделах информатики, есть приемы, которые можно быстро освоить, чтобы ваши БД по крайней мере достаточно хорошо и быстро работали для большинства приложений.

Во-первых, за редким исключением, всегда добавляйте в свои таблицы столбцы идентификаторов. У любой таблицы MySQL должен быть хотя бы один первичный ключ (ключевой столбец, по которому сортируется таблица), подсказывающий MySQL, как ее упорядочить. Часто бывает сложно выбрать эти ключи рационально.

Споры о том, что лучше задействовать в качестве ключа: искусственно созданный столбец идентификатора или

уникальный атрибут, такой как имя пользователя, — не утихали в среде исследователей данных и разработчиков программного обеспечения в течение многих лет. Однако я склоняюсь к варианту с созданием столбцов идентификаторов. *Особенно* это касается веб-скрапинга и хранения чьих-то чужих данных. Вы не можете знать заранее, какой элемент на самом деле окажется уникальным или неуникальным. Несколько раз действительность меня немало удивляла.

Столбец идентификатора должен присутствовать во всех ваших таблицах, иметь автоматическое приращение и использоваться в качестве первичного ключа.

Во-вторых, используйте интеллектуальную индексацию. Словарь (тот, который книга, а не объект Python) представляет собой список слов, проиндексированных в алфавитном порядке. Это позволяет быстро найти нужное слово, если известно, как оно написано. Вдобавок можно представить словарь, статьи которого расположены в алфавитном порядке по определению слова. Он был бы не так полезен — разве что для викторины наподобие «Своей игры», где нужно угадывать слово по его определению. Но при поиске в базе данных подобные ситуации тоже случаются. Например, в базе может присутствовать поле, к которому вы часто обращаетесь:

```
>SELECT * FROM dictionary WHERE definition="A  
small furry animal that says meow";
```

```
+-----+-----+-----+-----+  
----+  
|          id          |          word          |  
definition              |  
+-----+-----+-----+-----+  
----+  
| 200 | cat  | A small furry animal that says  
meow |
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set (0.00 sec)
```

Возможно, в дополнение к предположительно уже имеющемуся индексу по `id` вы захотите добавить в эту таблицу индекс по столбцу `definition`, чтобы ускорить поиск по данному столбцу. Однако имейте в виду: при добавлении еще одной индексации потребуется место для нового индекса, а также увеличится время вставки новых строк. Особенно это касается ситуации, когда мы имеем дело с большими объемами данных. Поэтому следует тщательно рассмотреть все компромиссы использования индексов и то, что и сколько нужно индексировать. Чтобы индекс определений стал немного легче, можно индексировать только несколько первых символов из значения столбца `definition`. Следующая команда создает индекс для первых 16 символов из поля `definition`:

```
CREATE INDEX definition ON dictionary (id,
definition(16));
```

Благодаря данному индексу поиск слов по полному определению будет выполняться намного быстрее (особенно если первые 16 символов определений в большинстве записей существенно отличаются) и при этом не потребует слишком много дополнительного места в памяти и не очень замедлит обработку.

Что касается компромисса между временем выполнения запроса и размером базы данных (одна из фундаментальных операций балансировки при разработке БД), то одна из наиболее распространенных ошибок, особенно при веб-скрапинге больших объемов данных, написанных на

естественных языках, заключается в хранении большого количества повторяющихся данных. Предположим, вы хотите измерить частоту появления определенных фраз, которые присутствуют на разных сайтах. Эти фразы могут извлекаться из заданного списка или автоматически генерироваться с помощью алгоритма анализа текста. У вас может возникнуть соблазн сохранить данные примерно так:

```

+-----+-----+-----+-----+-----+
+-----+
| Field  | Type          | Null | Key | Default
| Extra          |
+-----+-----+-----+-----+-----+
+-----+
| id      | int(11)        | NO    | PRI |
NULL     | auto_increment |
| url     | varchar(200)   | YES   |     |
NULL     |                |
| phrase  | varchar(200)   | YES   |     |
NULL     |                |
+-----+-----+-----+-----+-----+
+-----+

```

Каждый раз при обнаружении фразы на сайте в эту таблицу базы данных будет добавляться строка, а в нее — записываться URL страницы, где была найдена эта фраза. Однако разделив данные на три таблицы, можно значительно сократить набор данных:

```

>DESCRIBE phrases
+-----+-----+-----+-----+-----+
+-----+
| Field  | Type          | Null | Key | Default
| Extra          |

```

```

+-----+-----+-----+-----+-----+
+-----+
| id          | int(11)          | NO      | PRI  |
NULL         | auto_increment  |
| phrase     | varchar(200)    | YES     |      |
NULL         |                  |

```

>DESCRIBE urls

```

+-----+-----+-----+-----+-----+
+-----+
| Field | Type          | Null | Key | Default |
Extra          |
+-----+-----+-----+-----+-----+
+-----+
| id     | int(11)       | NO   | PRI | NULL     |
auto_increment |
| url    | varchar(200)  | YES  |     |          |
NULL     |              |

```

>DESCRIBE foundInstances

```

+-----+-----+-----+-----+-----+
+-----+
| Field          | Type          | Null | Key | Default |
| Extra          |
+-----+-----+-----+-----+-----+
+-----+
| id              | int(11)       | NO   | PRI |          |
NULL             | auto_increment |
| urlId           | int(11)       | YES  |     |          |
NULL             |              |

```


	phraseId		int(11)		YES			
NULL								
	occurrences		int(11)		YES			
NULL								
+-----+-----+-----+-----+-----								
+-----+								

Хоть определения таблиц и занимают больше места, мы видим, что большинство столбцов — это просто поля для целочисленных `id`, которые занимают гораздо меньше места в базе. Кроме того, каждый полный URL и каждая текстовая фраза сохраняются ровно один раз.

Если не установить сторонний пакет или не вести подробный журнал, то будет невозможно определить, в какой момент был добавлен, обновлен или удален каждый элемент данных. В зависимости от наличия свободного места для хранения данных, частоты их изменения и важности определения того, когда оно произошло, можно рассмотреть вопрос о сохранении временных меток создания, изменения и удаления данных: `created`, `updated` и `deleted` соответственно.

Шесть шагов в MySQL

В главе 3 мы рассмотрели задачу «Шесть шагов по «Википедии»», цель которой состояла в том, чтобы найти связь между любыми двумя статьями «Википедии», построив цепь ссылок (то есть найти способ перемещаться от одной статьи к другой, переходя по ссылкам с одной страницы на другую). Чтобы решить данную задачу, необходимо создать боты, способные не только собирать данные с сайта (это мы уже сделали), но и сохранять информацию архитектурно надежным

способом, который бы позволил упростить анализ данных впоследствии.

Для этого нам понадобятся столбцы автоинкрементируемых идентификаторов, временные метки и несколько таблиц. Чтобы решить, как лучше хранить данную информацию, необходимо мыслить абстрактно. Ссылка — всего лишь то, что соединяет страницу А со страницей Б. Ссылка тоже вполне может соединять страницу Б со страницей А, но это будет уже другая ссылка. Для однозначной идентификации ссылки достаточно сказать: «На странице А существует ссылка, которая соединяет ее со страницей Б», другими словами, `INSERT INTO links (fromPageId, toPageId) VALUES (A, B);` (где А и В — уникальные идентификаторы этих двух страниц).

Систему, состоящую из двух таблиц и предназначенную для хранения страниц и ссылок, а также дат создания и уникальных идентификаторов записей, можно построить следующим образом:

```
CREATE TABLE `wikipedia`.`pages` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `url` VARCHAR(255) NOT NULL,  
  `created` TIMESTAMP NOT NULL DEFAULT  
CURRENT_TIMESTAMP,  
  PRIMARY KEY (`id` ));
```

```
CREATE TABLE `wikipedia`.`links` (  
  `id` INT NOT NULL AUTO_INCREMENT,  
  `fromPageId` INT NULL,  
  `toPageId` INT NULL,  
  `created` TIMESTAMP NOT NULL DEFAULT  
CURRENT_TIMESTAMP,  
  PRIMARY KEY (`id` ));
```

Обратите внимание: в отличие от предыдущих веб-краулеров, которые выводили заголовок страницы на экран, здесь мы даже не сохраняем заголовок в таблице `pages`. Почему? Прежде всего потому, что для сохранения заголовка страницы нужно зайти на нее и получить ее содержимое. Если мы хотим создать эффективный веб-краулер для заполнения этих таблиц, то нам нужна возможность сохранять страницу и ссылки на нее, не посещая ее саму.

Это верно не для всех сайтов, однако у ссылок и заголовков страниц «Википедии» есть одно хорошее свойство: ссылки легко преобразуются в заголовки и наоборот. Например, ссылка http://en.wikipedia.org/wiki/Monty_Python ведет на страницу с заголовком Monty Python.

Следующая программа сохраняет все страницы «Википедии», у которых число Бейкона (количество ссылок между этой страницей и страницей Кевина Бейкона включительно) равно 6 или меньше:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import pymysql
from random import shuffle

conn = pymysql.connect(host='127.0.0.1',
                        unix_socket='/tmp/mysql.sock',
                        user='root',
                        passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def insertPageIfNotExists(url):
```

```

        cur.execute('SELECT * FROM pages WHERE url
= %s', (url))
        if cur.rowcount == 0:
            cur.execute('INSERT INTO pages (url)
VALUES (%s)', (url))
            conn.commit()
            return cur.lastrowid
        else:
            return cur.fetchone()[0]

```

```

def loadPages():
    cur.execute('SELECT * FROM pages')
    pages = [row[1] for row in cur.fetchall()]
    return pages

```

```

def insertLink(fromPageId, toPageId):
    cur.execute('SELECT * FROM links WHERE
fromPageId = %s '
                'AND toPageId = %s', (int(fromPageId),
int(toPageId)))
    if cur.rowcount == 0:
        cur.execute('INSERT INTO links
(fromPageId, toPageId) VALUES (%s, %s)',
                    (int(fromPageId),
int(toPageId)))
        conn.commit()

```

```

def getLinks(pageUrl, recursionLevel, pages):
    if recursionLevel > 4:
        return

```

```

        pageId = insertPageIfNotExists(pageUrl)
        html =
urlopen('http://en.wikipedia.org{}'.format(page
Url))
        bs = BeautifulSoup(html, 'html.parser')
        links = bs.find_all('a',
href=re.compile('^(/wiki/)((?!:).)*$'))
        links = [link.attrs['href'] for link in
links]

        for link in links:
            insertLink(pageId,
insertPageIfNotExists(link))
            if link not in pages:
                # Мы обнаружили новую страницу,
добавляем ее
                # и ищем ссылки.
                pages.append(link)
                getLinks(link, recursionLevel+1,
pages)

getLinks('/wiki/Kevin_Bacon', 0, loadPages())
cur.close()
conn.close()

```

Три представленные ниже функции используются PyMySQL для взаимодействия с базой данных.

- Функция `insertPageIfNotExists`, как видно из названия, вставляет запись о новой странице, если таковая еще не существует. Это в сочетании с дополняемым списком всех собранных страниц, хранящихся в таблице `pages`, гарантирует, что записи страниц не будут дублироваться.

Данная функция также позволяет находить номера `pageId` для создания новых ссылок.

- Функция `insertLink` создает новую запись о ссылке в базе данных. При наличии такой ссылки запись не создается. Даже если на странице действительно есть *несколько* одинаковых ссылок, для нас это одна и та же ссылка, представляющая одни и те же отношения, и ей должна соответствовать только одна запись. Это также помогает сохранять целостность базы данных, если программа будет запускаться несколько раз для одних и тех же страниц.
- Функция `loadPages` загружает все текущие страницы из базы данных в список, чтобы можно было определить, следует ли посетить новую страницу. Список страниц также пополняется во время выполнения программы. Если веб-краулер запускается только один раз, начиная с пустой базы данных, то теоретически `loadPage` не требуется. Однако на практике вероятны проблемы: сбой в сети или же сбор ссылок за несколько приемов. Важно иметь возможность перезагрузить веб-краулер, при этом не потеряв результаты проделанной работы.

Следует учитывать одну потенциально проблемную особенность использования функции `loadPages` и создаваемого ею списка `pages`, по которому мы определяем, стоит ли посещать страницу. После загрузки страницы все ссылки, которые есть на ней, также сохраняются как страницы, хотя веб-краулер их еще не посещал, а лишь зафиксировал ссылки на них. Если его остановить и перезапустить, то все эти «зафиксированные, но не просмотренные» страницы так никогда и не будут просмотрены, а размещенные на них

ссылки — записаны. Чтобы это исправить, нужно добавить в каждой записи страницы логическую переменную `visited`, которая принимает значение `True`, только если эта страница была загружена, а ее исходящие ссылки — записаны.

Однако для наших целей такого решения достаточно. Если вы способны обеспечить длительное время выполнения (или только один запуск) программы и нет необходимости собрать полный набор ссылок (а нужен лишь довольно большой набор данных для экспериментов), то добавлять переменную `visited` не обязательно.

Продолжение и окончательное решение задачи перехода от страницы Кевина Бейкона (https://en.wikipedia.org/wiki/Kevin_Bacon) к странице Эрика Айidla (https://en.wikipedia.org/wiki/Eric_Idle) вы найдете в пункте «Шесть шагов по “Википедии”: заключение» на с. 172, посвященном решению задач направленных графов.

Электронная почта

Подобно тому как веб-страницы передаются по протоколу HTTP, электронная почта пересылается по протоколу SMTP (Simple Mail Transfer Protocol, простой протокол электронной почты). И точно так же, как для отправки веб-страниц по HTTP мы используем клиент веб-сервера, для отправки и получения электронной почты серверы задействуют различные почтовые клиенты, например Sendmail, Postfix или Mailman.

Несмотря на то что отправка электронной почты в Python осуществляется сравнительно просто, эта операция требует доступа к серверу, на котором работает SMTP. Настройка SMTP-клиента на сервере или локальном компьютере довольно сложна и выходит за рамки данной книги, однако есть

множество превосходных ресурсов, позволяющих решить эту задачу, особенно если вы работаете на Linux или в macOS.

В следующих примерах кода предполагается, что SMTP-клиент используется локально. (Чтобы модифицировать этот код для удаленного SMTP-клиента, замените `localhost` адресом удаленного сервера.)

Для отправки электронного письма с помощью Python достаточно всего девяти строк кода:

```
import smtplib
from email.mime.text import MIMEText

msg = MIMEText('The body of the email is here')

msg['Subject'] = 'An Email Alert'
msg['From'] = 'ryan@pythonscraping.com'
msg['To'] = 'webmaster@pythonscraping.com'

s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

В Python есть два важных пакета для отправки электронной почты: `smtplib` и `email`.

Python-модуль `email` содержит полезные функции форматирования для создания готовых к отправке пакетов электронной почты. Применяемый здесь объект `MIMEText` создает пустое электронное письмо, отформатированное для передачи по низкоуровневому протоколу MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения почтовой интернет-службы), через который устанавливаются SMTP-соединения более высокого уровня. Объект `MIMEText` содержит

адреса электронной почты, тело и заголовок, служащие в Python для создания правильно отформатированного электронного письма.

Пакет `smtplib` содержит информацию для обработки соединения с сервером. Как и при соединении с сервером MySQL, это соединение, будучи созданным и использованным, должно разрываться, чтобы не создавалось слишком большого количества соединений.

Эту простейшую операцию с электронной почтой можно расширить и сделать более полезной, представив ее в виде функции:

```
import smtplib
from email.mime.text import MIMEText
from bs4 import BeautifulSoup
from urllib.request import urlopen
import time

def sendMail(subject, body):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = 'christmas_alerts@pythonscraping.com'
    msg['To'] = 'ryan@pythonscraping.com'

    s = smtplib.SMTP('localhost')
    s.send_message(msg)
    s.quit()

bs = BeautifulSoup(urlopen('https://isitchristmas.com/'), 'html.parser')
```

```

while(bs.find('a',
{'id':'answer'}).attrs['title'] == 'NO'):
    print('It is not Christmas yet.')
    time.sleep(3600)

bs =
BeautifulSoup(urlopen('https://isitchristmas.com/'), 'html.parser')

sendMail('It\'s Christmas!',
'According to http://itischristmas.com, it
is Christmas!')

```

Этот скрипт раз в час проверяет сайт **<https://isitchristmas.com>**, главной функцией которого является вывод гигантского слова YES или NO, в зависимости от дня года. Если скрипт заметит на сайте что-либо отличное от NO, то отправит вам электронное письмо с предупреждением о наступлении Рождества.

Несмотря на то что эта конкретная программа выглядит не намного более полезной, чем обычный настенный календарь, достаточно слегка ее изменить — и можно будет делать множество чрезвычайно нужных вещей. Она может отправлять по электронной почте сообщения о сбоях в работе сайта, неполадках при тестировании или даже информировать о появлении в магазине Amazon товара, который вы ожидаете, — ничего из этого настенный календарь делать не может.

[5](http://bit.ly/1LWVmc8) *Joab Jackson*. YouTube Scales MySQL with Go Code (<http://bit.ly/1LWVmc8>), PCWorld, December 15, 2012.

[6](http://bit.ly/1KHDKns) *Jeremy Cole and Davi Arnaut*. MySQL at Twitter (<http://bit.ly/1KHDKns>), The Twitter Engineering Blog, April 9, 2012.

[7](http://on.fb.me/1RFMqvww) MySQL and Database Engineering: Mark Callaghan (<http://on.fb.me/1RFMqvww>), Facebook Engineering, March 4, 2012.

[8](#) *Дюбуа П.* MySQL. Сборник рецептов. — СПб.: Символ-Плюс, 2016.

Часть II. Углубленный веб-скрапинг

Итак, мы рассмотрели некоторые основы создания веб-скраперов; теперь начинается самое интересное. До сих пор наши веб-скраперы были довольно примитивными. Они не умели получать информацию, если сервер не предоставлял ее сразу, в удобном формате. Они принимали все за чистую монету и сохраняли, никак не анализируя. Они пасовали перед формами, необходимостью взаимодействия с сайтом и даже скриптами JavaScript. Короче говоря, эти веб-скраперы бесполезны для извлечения информации, если только она сама не хочет быть извлеченной.

В этой части книги вы научитесь анализировать необработанные данные и извлекать из них историю — ту самую, которую сайты часто скрывают под слоями JavaScript-кода, формами входа в систему и средствами защиты от веб-скрапинга. Вы узнаете, как использовать веб-скраперы для тестирования сайтов, автоматизации процессов и широкомасштабного доступа к Интернету. К тому времени, как вы закончите читать этот раздел, в вашем распоряжении появятся инструменты для сбора и обработки практически любого типа данных, представленных в любом виде и в любом месте Интернета.

Глава 7. Чтение документов

Заманчиво представлять себе Интернет главным образом как коллекцию текстовых сайтов, слегка разбавленных новомодным мультимедийным контентом web 2.0, на который при веб-скрапинге в большинстве случаев можно не обращать внимания. Однако при этом мы игнорируем то, чем в своей основе является Интернет: средство передачи файлов, независимо от их контента.

Хотя сам Интернет в той или иной форме существует еще с конца 1960-х, HTML появился лишь в 1992 году. До тех пор Интернет состоял главным образом из электронной почты и систем передачи файлов. Концепции веб-страниц в том виде, в каком мы их сегодня знаем, тогда не было. Другими словами, Интернет — это не коллекция HTML-файлов, а множество документов различных типов, причем HTML-файлы часто используются в качестве рамки, в которую вставляются эти документы. Не имея возможности читать документы разных типов, в том числе текст, PDF, изображения, видео, электронную почту и т.д., мы упускаем огромную часть доступных данных.

В этой главе рассматривается работа с документами, независимо от того, скачиваем мы их в локальную папку или читаем по сети и извлекаем данные. Вы также познакомитесь с различными текстовыми кодировками, что позволит читать HTML-страницы на иностранных языках.

Кодировка документов

Кодировка документа указывает приложению — будь то операционная система компьютера или написанный вами код

на Python, — как следует читать этот документ. Узнать кодировку обычно можно из расширения файла, хотя оно не обязательно соответствует кодировке. Я могу, например, сохранить файл `myImage.jpg` как `myImage.txt`, и проблем не возникнет — по крайней мере до тех пор, пока я не попытаюсь открыть его в текстовом редакторе. К счастью, подобные ситуации встречаются редко; как правило, достаточно знать расширение файла, в котором хранится документ, чтобы прочесть его правильно.

В своей основе любой документ содержит только нули и единицы. Затем вступают в действие алгоритмы кодирования. Они определяют такие вещи, как «сколько битов приходится на один символ» или «сколько битов занимает цвет пиксела» (в случае файлов с изображениями). Далее может подключаться уровень сжатия или некий алгоритм сокращения занимаемого места в памяти, как в случае с PNG-файлами.

Поначалу перспектива иметь дело с файлами, не относящимися к формату HTML, может выглядеть пугающе, однако будьте уверены: при подключении соответствующей библиотеки Python располагает всеми необходимыми средствами для работы с любым форматом информации, который вам попадется. Единственное различие между файлами с текстом, видео и изображениями состоит в том, как интерпретируются их нули и единицы. В этой главе мы рассмотрим следующие часто встречающиеся типы файлов: текст, CSV, PDF и документы Word.

Обратите внимание: во всех этих файлах, по сути, хранится текст. Если вы хотите узнать, как работать с изображениями, то я рекомендую прочесть данную главу, чтобы привыкнуть обрабатывать и хранить различные типы файлов, а затем перейти к главе 13, в которой вы получите дополнительную информацию об обработке изображений.

Текст

Хранить файлы в виде обычного текста в Интернете кажется несколько необычным, однако есть много простых сайтов и сайтов старого образца с обширными хранилищами текстовых файлов. Например, на сайте Инженерного совета Интернета (Internet Engineering Task Force, IETF) все опубликованные документы хранятся в формате HTML, PDF и текстовых файлов (см., например, <https://www.ietf.org/rfc/rfc1149.txt>). В большинстве браузеров эти текстовые файлы отлично отображаются, так что веб-скрапинг для них должен выполняться без проблем.

Для большинства простейших текстовых документов, таких как учебный файл, расположенный по адресу <http://www.pythonscraping.com/pages/warandpeace/chapter1.txt>, можно использовать следующий метод:

```
from urllib.request import urlopen
textPage =
urlopen('http://www.pythonscraping.com/'\
        'pages/warandpeace/chapter1.txt')
print(textPage.read())
```

Обычно, получая страницу с помощью `urlopen`, мы превращаем ее в объект `BeautifulSoup`, чтобы выполнить синтаксический анализ HTML-кода. В данном случае мы можем прочитать страницу напрямую. Мы могли бы превратить ее в объект `BeautifulSoup`, однако это было бы нерационально: здесь нет HTML-разметки, которую стоило бы анализировать, поэтому библиотека будет бесполезной. Прочитав текстовый файл как строку, мы можем лишь проанализировать его аналогично любой другой строке, прочитанной в Python. Правда, здесь не получится использовать HTML-теги в качестве

контекстных подсказок, указывающих на то, какой текст нам действительно нужен, а какой можно отбросить. Это может стать проблемой, если из текстовых файлов требуется извлечь лишь определенную информацию.

Текстовые кодировки и глобальный Интернет

Помните, раньше я говорила, что расширение в имени файла — это все, что нужно для правильного прочтения файла? Так вот, как ни странно, данное правило не распространяется на самый простой из всех видов документов: файлы с расширением `.txt`.

В девяти случаях из десяти описанные выше методы чтения текста будут работать. Однако иногда прочитать текст из Интернета бывает непросто. Далее я расскажу об основных кодировках для английского и других языков, от ASCII до Unicode и ISO, и о том, как с ними справляться.

История текстовых кодировок

Кодировка ASCII появилась еще в 1960-х, когда на счету был каждый бит и отсутствовали причины кодировать что-либо, кроме латинского алфавита и нескольких знаков препинания. Поэтому использовались всего 7 бит, позволявшие закодировать 128 символов, включая заглавные и строчные буквы, а также знаки препинания. При всем воображении разработчиков в любом случае оставались еще 33 непечатных символа, из которых одни использовались, а другие со временем, по мере развития технологий, были заменены и/или устарели. Места было предостаточно, не правда ли?

Как известно любому программисту, семерка — странное число. Это не степень двойки, но заманчиво близко к ней. В 1960-х специалисты по информатике спорили о том, следует ли

вводить еще один бит, чтобы получить удобное круглое число, или же оставить как есть, чтобы для хранения файлов требовалось меньше места. В итоге победили 7 бит. Однако в современных вычислениях каждая семибитная последовательность дополняется ведущим нулем⁹, поэтому мы получаем оба недостатка: и файлы на 14 % больше, и доступны всего 128 символов.

В начале 1990-х человечество наконец обнаружило, что, помимо английского, существуют и другие языки и было бы очень хорошо, если бы компьютеры могли отображать и их. Некоммерческая организация под названием «Консорциум Unicode» попыталась создать универсальную кодировку текста, присвоив код каждому символу, который может использоваться в любом текстовом документе на любом языке. Цель организации состояла в том, чтобы включить в кодировку все, от латинского алфавита, на котором была написана эта книга, до кириллицы (щ, ъ, ы), китайских иероглифов (象象), математических и логических символов (Σ , \geq) и даже смайликов и прочих символов, таких как знак биологической опасности (☠) и «пацифик» (☺).

Получившаяся кодировка, как вы, возможно, уже поняли, получила название UTF-8, что, как ни странно, расшифровывается следующим образом: Universal Character Set — Transformation Format 8 bit — «Универсальный набор символов — восьмиразрядная форма представления». Слово «восьмиразрядная» здесь относится не к размеру символа, а к наименьшему размеру, который требуется для отображения символа.

Реальное количество битов для представления символа в UTF-8 является переменной величиной. Оно может изменяться от 1 до 4 байт, в зависимости от положения символа в списке возможных символов (чем чаще используется символ, тем

меньше байтов он занимает, более редкие символы занимают больше байтов).

Как достигается эта гибкость кодировки? Использование 7 бит с возможно бесполезным ведущим нулем поначалу выглядело как недостаток структуры ASCII, но для UTF-8 оказалось огромным преимуществом. Поскольку кодировка ASCII была очень популярна, Консорциум Unicode решил задействовать преимущество этого ведущего нулевого бита, объявив: если байт начинается с нуля, то это значит, что символ занимает только этот один байт, благодаря чему схемы кодирования для ASCII и UTF-8 оказались идентичными. Таким образом, следующие символы допустимы как в UTF-8, так и в ASCII:

01000001 - A

01000010 - B

01000011 - C

А эти символы допустимы только в UTF-8. При интерпретации документа в кодировке ASCII они будут отображаться как непечатные:

11000011 10000000 - À

11000011 10011111 - ß

11000011 10100111 - ç

Кроме UTF-8, есть и другие стандарты UTF, такие как UTF-16, UTF-24 и UTF-32, хотя документы, закодированные в этих форматах, встречаются редко, за исключением необычных обстоятельств, которые выходят за рамки данной книги.

Несмотря на то что этот оригинальный «конструктивный недостаток» ASCII оказался большим преимуществом для UTF-8, полностью он не исчез. Первые 8 бит информации для

каждого символа по-прежнему позволяют кодировать только 128 символов, а не все 256. В символе UTF-8, требующем нескольких байтов, дополнительные начальные биты не расходуются на кодировку символов, а являются проверочными и служат для предотвращения повреждения кода. Из 32 (8×4) бит в четырехбайтовых символах только 21 бит используется для кодирования символов. В общей сложности это составляет 2 097 152 возможных символа, из которых в настоящее время применяются 1 114 112.

Очевидно, проблема всех универсальных стандартов кодирования языков заключается в том, что любой документ, написанный на каком-то одном иностранном языке, занимает намного больше места, чем мог бы. Даже если ваш язык насчитывает не более 100 символов, для представления каждого символа все равно потребуется 16 бит, а не 8, как в случае кодировки ASCII, привязанной к английскому языку. Из-за этого текстовые документы, написанные на иностранных языках в кодировке UTF-8, занимают примерно в два раза больше места, чем текстовые документы на английском, — по крайней мере это касается тех языков, в которых не используется латиница.

ISO решает эту проблему путем создания отдельной кодировки для каждого языка. Подобно Unicode, здесь используются те же кодировки, что и для ASCII, но дополнительный ведущий нулевой бит в начале каждого символа позволяет создать 128 специальных символов для всех языков, требующих этого. Данный вариант лучше всего подходит для европейских языков, чей алфавит построен на основе латиницы (символы которой в этой кодировке продолжают занимать позиции от 0 до 127), но имеет дополнительные специальные символы. Благодаря этому в кодировке ISO-8859-1 (разработанной для латинского

алфавита) появились такие символы, как дроби (например, 1/2) или символ авторского права (©).

В Интернете нередко встречаются и другие наборы символов ISO, такие как ISO-8859-9 (турецкий алфавит), ISO-8859-2 (в том числе немецкий), а также ISO-8859-15 (в том числе французский).

Несмотря на то что в последние годы популярность документов в кодировке ISO снижается, около 9 % сайтов в Интернете по-прежнему используют в качестве кодировки одну из разновидностей ISO¹⁰, поэтому нам все равно необходимо помнить о существовании разных кодировок и проверять кодировку перед веб-скрапингом сайта.

Использование кодировок на практике

В предыдущем разделе мы использовали стандартные параметры настройки `urlopen` для чтения текстовых документов, которые встречаются в Интернете. Эти параметры прекрасно подходят для большинства текстов на английском языке. Но если вам встретится документ на русском или арабском или даже всего лишь отдельное слово наподобие *re'sume'* — могут возникнуть проблемы.

Рассмотрим, к примеру, такой код:

```
from urllib.request import urlopen
textPage =
urlopen('http://www.pythonscraping.com/'\
        'pages/warandpeace/chapter1-ru.txt')
print(textPage.read())
```

Этот код читает первую главу романа Л. Толстого «Война и мир» (где встречается текст на русском и французском языках)

и выводит ее на экран. В частности, на экран будет выведено следующее:

```
b"\xd0\xa7\xd0\x90\xd0\xa1\xd0\xa2\xd0\xac
\xd0\x9f\xd0\x95\xd0\xa0\xd0\x92\xd0\
x90\xd0\xaf\n\nI\n\n\xe2\x80\x94 Eh bien, mon
prince.
```

Открыв данную страницу в большинстве браузеров, мы тоже увидим абракадабру (рис. 7.1).

Это не поймут даже те, для кого русский язык является родным. Проблема в том, что Python пытается прочитать документ в кодировке ASCII, а браузер — в кодировке ISO-8859-1. И ни тот ни другой, конечно же, не предполагают, что кодировка данного документа — UTF-8.

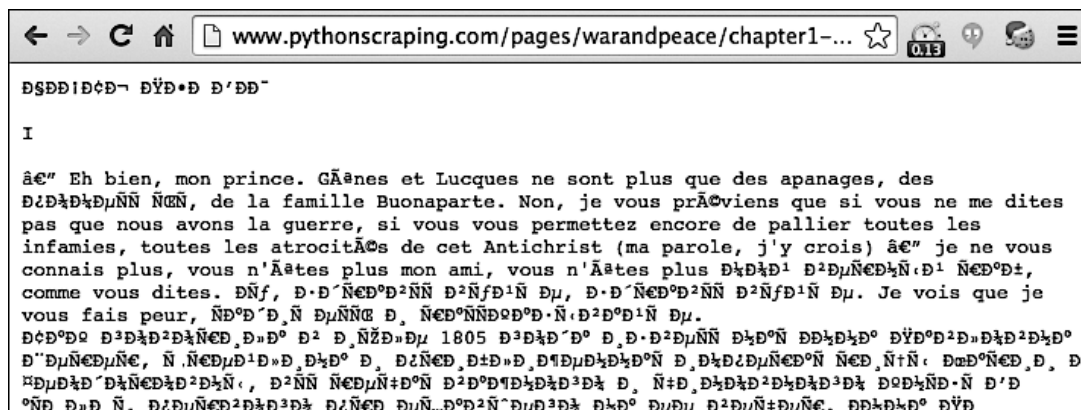


Рис. 7.1. Текст на русском и французском языках в кодировке в ISO-8859-1, которая во многих браузерах используется по умолчанию для текстовых документов

Но можно явно задать кодировку строки как UTF-8, и тогда кириллические символы будут выведены правильно:

```
from urllib.request import urlopen

textPage =
urlopen('http://www.pythonscraping.com/'\
```

```
'pages/warandpeace/chapter1-ru.txt')
print(str(textPage.read(), 'utf-8'))
```

Применение этой концепции в BeautifulSoup и Python 3.x выглядит так:

```
html = urlopen('http://en.wikipedia.org/wiki/Python_(p
rogramming_language)')
bs = BeautifulSoup(html, 'html.parser')
content = bs.find('div', {'id':'mw-content-
text'}).get_text()
content = bytes(content, 'UTF-8')
content = content.decode('UTF-8')
```

В Python 3.x все символы по умолчанию кодируются в UTF-8. У вас может возникнуть желание оставить все как есть и использовать кодировку UTF-8 во всех веб-скраперах, которые вам доведется писать. В конце концов, UTF-8 будет одинаково хорошо поддерживать и символы в кодировке ASCII, и текст на иностранных языках. Однако важно помнить о 9 % сайтов, использующих ту или иную версию кодировки ISO, из-за которых вам не удастся полностью избежать этой проблемы.

К сожалению, в случае с текстовыми документами невозможно точно определить кодировку документа. Есть библиотеки, которые позволяют исследовать документ и сделать правильное предположение (используя некую логику, способную сделать вывод, что Ñ^Ð°ÑÑÐoÐ°Ñ, вероятно, не является словом), но они часто ошибаются.

К счастью, в случае с HTML-страницами кодировка обычно обозначена в теге, расположенном в разделе <head> сайта. У большинства сайтов, особенно англоязычных, есть такой тег:

```
<meta charset="utf-8" />
```

А, например, на сайте ECMA International (<http://www.ecma-international.org/>) есть такой тег¹¹:

```
<META HTTP-EQUIV="Content-Type"  
CONTENT="text/html; charset=iso-8859-1">
```

Если вы планируете выполнять активный веб-скрапинг, особенно многоязычных сайтов, то, вероятно, имеет смысл найти этот метатег и использовать рекомендованную в нем кодировку для чтения контента страницы.

CSV

В процессе веб-скрапинга вам, скорее всего, либо встретятся CSV-файлы, либо придется работать с коллегой, которому нравится представлять данные в этом формате. К счастью, у Python есть отличная библиотека (<https://docs.python.org/3.4/library/csv.html>) для чтения и записи CSV-файлов. Она способна обрабатывать многие варианты CSV, но в этом разделе мы уделим основное внимание стандартному формату. Если вам попадется особый случай, с которым придется разбираться, то обратитесь к документации!

Чтение CSV-файлов. CSV-библиотека Python ориентирована в первую очередь на работу с локальными файлами, поскольку исходит из предположения, что необходимые CSV-данные уже хранятся на вашем компьютере. К сожалению, это не всегда так, особенно при веб-скрапинге. Есть несколько способов обойти данное условие.

- Скачайте файл на компьютер вручную, и пусть Python работает с локальным файлом.

- Напишите скрипт Python, который будет скачивать файл, читать его и (возможно) удалять после извлечения данных.
- Получите файл из Интернета в виде строки и оберните строку в объект StringIO, который ведет себя как файл.

Первые два варианта вполне работоспособны, однако занимать место на жестком диске файлами вместо того, чтобы хранить их в памяти, — плохая идея. Гораздо лучше прочитать файл как строку и обернуть его в объект, который Python будет обрабатывать как файл, притом не сохраняя данные в виде файла. Следующий скрипт получает CSV-файл из Интернета (в данном случае это список альбомов Monty Python, расположенный по адресу <http://pythonscraping.com/files/MontyPythonAlbums.csv>) и выводит его построчно в окно терминала:

```
from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen('http://pythonscraping.com/files/MontyPythonAlbums.csv').read().decode('ascii', 'ignore')
dataFile = StringIO(data)
csvReader = csv.reader(dataFile)

for row in csvReader:
    print(row)
```

Результат выглядит так:

```
['Name', 'Year']
```



```
["Monty Python's Flying Circus", '1970']  
['Another Monty Python Record', '1971']  
["Monty Python's Previous Record", '1972']  
...
```

Как видно из данного примера кода, объект `reader`, возвращаемый `csv.reader`, является итерируемым и состоит из объектов-списков Python. Поэтому доступ к строкам объекта `csvReader` осуществляется следующим образом:

```
for row in csvReader:  
    print('The album "' + row[0] + '" was released  
in ' + str(row[1]))
```

Результат выглядит так:

```
The album "Name" was released in Year  
The album "Monty Python's Flying Circus" was  
released in 1970  
The album "Another Monty Python Record" was  
released in 1971  
The album "Monty Python's Previous Record" was  
released in 1972  
...
```

Обратите внимание на первую строку: `Thealbum"Name"wasreleasedinYear`. При написании данного примера ее вполне можно было бы пропустить, но вы же не хотите, чтобы она попала в ваши данные в реальности! Менее опытный программист мог бы просто пропустить первую строку в объекте `csvReader` или написать специальный код для ее обработки. К счастью, у функции `csv.reader` есть альтернатива, которая позаботится обо всем этом автоматически. Знакомьтесь — `DictReader`:

```

from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen('http://pythonscraping.com/files/MontyPythonAlbums.csv').read().decode('ascii', 'ignore')
dataFile = StringIO(data)
dictReader = csv.DictReader(dataFile)

print(dictReader.fieldnames)
for row in dictReader:
    print(row)

```

Функция `csv.DictReader` возвращает значения всех строк CSV-файла в виде объектов не списка, а словаря, с именами полей, хранящимися в переменной `dictReader.fieldnames` и в качестве ключей каждого объекта словаря:

```

['Name', 'Year']
{'Name': 'Monty Python's Flying Circus',
'Year': '1970'}
{'Name': 'Another Monty Python Record', 'Year':
'1971'}
{'Name': 'Monty Python's Previous Record',
'Year': '1972'}

```

Здесь, конечно, есть недостаток: создание, обработка и вывод объектов `DictReader` занимает немного больше времени, чем объектов `csvReader`, однако удобство и простота использования часто того стоят. Учтите также, что при веб-скрапинге затраты на запрос и извлечение данных сайта с

внешнего сервера почти всегда будут главным ограничивающим фактором в любой создаваемой вами программе, поэтому едва ли стоит беспокоиться о том, что какая-то технология замедлит ваш компьютер еще на несколько микросекунд!

PDF

Мне как пользователю Linux до боли обидно видеть присланные файлы .docx, безбожно исковерканные в текстовом редакторе, не принадлежащем Microsoft, и мучительно подбирать кодеки для интерпретации очередного нового медиаформата Apple. В определенном смысле Adobe совершила прорыв, когда в 1993 году разработала переносимый формат документов (Portable Document Format, PDF). PDF-файлы обеспечили единый способ просмотра изображений и текстовых документов для всех пользователей независимо от платформы, на которой открывается документ.

Несмотря на то что принцип хранения PDF-файлов в Интернете несколько устарел (зачем хранить контент в статическом, медленно загружаемом формате, если его можно представить в виде HTML-кода?), формат PDF по-прежнему встречается повсеместно, особенно когда речь идет об официальных документах и анкетах.

В 2009 году британец Ник Иннес (Nick Innes) прославился в новостях, когда запросил у муниципального совета Бэкингемшира открытую информацию о результатах экзаменов, которая была доступна в соответствии с законом о свободе информации, принятом в Соединенном Королевстве. После нескольких запросов — и отказов — он наконец получил желаемую информацию в виде 184 PDF-документов.

Конечно, Иннес настоял на своем и в итоге получил более правильно отформатированную базу данных, однако, если бы на его месте был специалист по веб-скрапину, он, скорее всего, сэкономил бы много времени, потраченного на судебные разбирательства, вместо этого напрямую обработав PDF-документы с помощью одного из многочисленных модулей Python для синтаксического анализа PDF.

К сожалению, многие библиотеки синтаксического анализа PDF, созданные для Python 2.x, не получили обновлений после выхода Python 3.x. Однако, поскольку PDF является относительно простым и открытым форматом документов, есть много хороших библиотек Python, в том числе для Python 3.x, которые позволяют читать подобные документы.

К одной из таких относительно простых в использовании библиотек относится PDFMiner3K. Это гибкий инструмент, который можно применять из командной строки или интегрировать в код. Данная библиотека также поддерживает различные языковые кодировки, что опять же в Интернете часто бывает очень кстати.

Библиотеку PDFMiner3K можно установить обычным способом, через `pip`, или скачать этот модуль Python (<https://pypi.python.org/pypi/pdfminer3k>) и установить его, распаковав папку и выполнив следующую команду:

```
$ python setup.py install
```

Документация PDFMiner3K находится в извлеченной папке по адресу `/pdfminer3k-1.3.0/docs/index.html` — впрочем, существующая документация больше ориентирована на интерфейс командной строки, чем на интеграцию с кодом Python.

Ниже показан простейший способ использования библиотеки, позволяющий читать произвольные PDF-файлы

(исходный объект — локальный файл) и представлять их в виде строки:

```
from urllib.request import urlopen
from pdfminer.pdfinterp import PDFResourceManager, process_pdf
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from io import StringIO
from io import open

def readPDF(pdfFile):
    rsrcmgr = PDFResourceManager()
    retstr = StringIO()
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr,
        laparams=laparams)

    process_pdf(rsrcmgr, device, pdfFile)
    device.close()

    content = retstr.getvalue()
    retstr.close()
    return content

pdfFile = urlopen('http://pythonscraping.com/'
    'pages/warandpeace/chapter1.pdf')
outputString = readPDF(pdfFile)
print(outputString)
pdfFile.close()
```

В результате получим знакомый текст:

CHAPTER I

"Well, Prince, so Genoa and Lucca are now just family estates of the Buonapartes. But I warn you, if you don't tell me that this means war, if you still try to defend the infamies and horrors perpetrated by that Antichrist- I really believe he is Antichrist- I will

Преимущество этой программы для чтения PDF-документов состоит в том, что при работе с локальными файлами мы можем заменить обычный файловый объект Python объектом, возвращаемым `urlopen`, и использовать такую строку:

```
pdfFile =  
open('../pages/warandpeace/chapter1.pdf', 'rb')
```

Результат может быть неидеальным, особенно для PDF-файлов, содержащих изображения, необычно отформатированный текст, таблицы или диаграммы. Однако для большинства PDF-файлов, содержащих только текст, результат должен быть таким же, как если бы PDF был текстовым файлом.

Microsoft Word и файлы .docx

Рискуя обидеть моих друзей из Microsoft, все же скажу: я не люблю Microsoft Word. Не потому, что это плохая программа, а потому, что пользователи ею часто злоупотребляют. У Microsoft Word есть особый талант превращать простые текстовые документы или PDF-файлы в больших, медленных, трудно открываемых монстров, которые при переносе с одного

компьютера на другой часто теряют все форматирование и по непонятной причине оказываются доступными для редактирования, хотя подразумевается, что их контент должен быть статичным.

Файлы Word предназначены для создания контента, а не для обмена им. Тем не менее ряд сайтов постоянно используют эти файлы для представления важных документов, информации и даже диаграмм и мультимедиа — в общем, всего того, что можно и нужно представлять с помощью HTML-кода.

Примерно до 2008 года в продуктах Microsoft Office использовался собственный формат файлов .doc. Этот двоичный файловый формат было трудно читать, и он плохо поддерживался другими текстовыми процессорами. Стремясь идти в ногу со временем и создать стандарт, который бы применялся во многих других программах, компания Microsoft решила задействовать стандарт Open Office на основе XML, благодаря которому файлы стали совместимыми с другим ПО, в том числе построенным по принципу открытого исходного кода.

К сожалению, Python все еще не очень хорошо поддерживает этот формат файлов, используемый в Google Docs, Open Office и Microsoft Office. Существует библиотека `python-docx` (<http://python-docx.readthedocs.org/en/latest/>), однако она позволяет только создавать документы и читать лишь основные данные файла, такие как размер и заголовок, а не реальный контент. Чтобы прочитать содержимое файла Microsoft Office, вам потребуется разработать собственное решение.

Первым шагом к этому является чтение XML из файла:

```
from zipfile import ZipFile
from urllib.request import urlopen
```

```

from io import BytesIO

wordFile =
urlopen('http://pythonscraping.com/pages/AWordD
ocument.docx').read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content =
document.read('word/document.xml')
print(xml_content.decode('utf-8'))

```

Этот код считывает удаленный документ Word в виде двоичного файлового объекта (BytesIO аналогичен StringIO, который использовался ранее в этой главе), распаковывает его с помощью библиотеки ядра Python zipfile (по соображениям экономии места все файлы .docx упаковываются в архив), а затем читает распакованный файл, имеющий формат XML.

На рис. 7.2 показан документ Word, размещенный по адресу **<http://pythonscraping.com/pages/AWordDocument.docx>**.

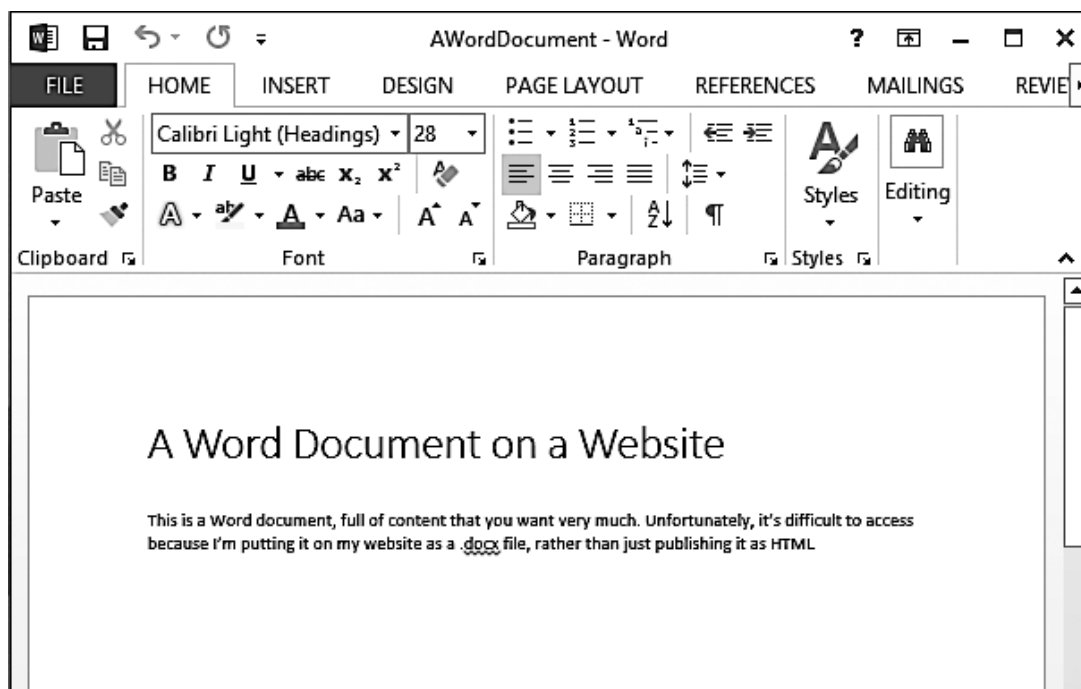


Рис. 7.2. Документ Word, полное содержимое которого вы, возможно, очень захотите прочитать, но к нему трудно получить доступ, поскольку я разместила его на своем сайте в виде файла .docx, вместо того чтобы опубликовать в формате HTML

Результат работы скрипта Python, прочитавшего мой простой документ Word, выглядит так:

```
<!--?xml      version="1.0"      encoding="UTF-8"
standalone="yes"?--><w:document
mc:ignorable="w14          w15          wp14"
xmlns:m="http://schemas.openxmlformats.org/offi
ceDocument/2006/math"
xmlns:mc="http://schemas.openxmlformats.org/mar
kup-compatibility/2006"      xmlns:o="urn:schemas-
microsoft-com:office:office"
xmlns:r="http://schemas.openxmlformats.org/offi
ceDocument/2006/relationships"
xmlns:v="urn:schemas-microsoft-
com:vml"xmlns:w="http://schemas.openxmlformats.
org/wordprocessingml/2006/main"
xmlns:w10="urn:schemas-microsoft-
```

com:office:word"
xmlns:w14="http://schemas.microsoft.com/office/
word/2010/wordml"
xmlns:w15="http://schemas.microsoft.com/office/
word/2012/wordml"
xmlns:wne="http://schemas.microsoft.com/office/
word/2006/wordml"
xmlns:wp="http://schemas.openxmlformats.org/dra
wingml/2006/wordprocessingDrawing"
xmlns:wp14="http://schemas.microsoft.com/office
/word/2010/wordprocessingDrawing"
xmlns:wpc="http://schemas.microsoft.com/office/
word/2010/wordprocessingCanvas"
xmlns:wpg="http://schemas.microsoft.com/

office/word/2010/wordprocessingGroup"
xmlns:wpi="http://schemas.microsoft.com/

office/word/2010/wordprocessingInk"
xmlns:wps="http://schemas.microsoft.com/office/
word/2010/wordprocessingShape"><w:body><w:p
w:rsidp="00764658" w:r sidr="00764658"
w:rsidrdefault="00764658"><w:ppr><w:pstyle
w:val="Title">

</w:pstyle></w:ppr><w:r><w:t>A Word Document on
a Website</w:t>

</w:r><w:bookmarkstart w:id="0"
w:name="_GoBack"></w:bookmarkstart>
<w:bookmarkend w:id="0"></w:bookmarkend></w:p>
<w:p w:rsidp="00764658" w:rsidr="00764658"
w:rsidrdefault="00764658"></w:p><w:p
w:rsidp="00764658" w:rsidr="00764658"
w:rsidrdefault="00764658" w:rsidrpr="00764658">
<w:r> <w:t>This is a Word document, full of

content that you want very much. Unfortunately,
it's difficult to access because I'm putting it
on my website as a .</w:t></w:r><w:prooferr
w:type="spellStart"></w:prooferr><w:r>
<w:t>docx</w:t></w:r><w:prooferr
w:type="spellEnd"></w:prooferr> <w:r> <w:t
xml:space="preserve"> file,

rather than just publishing it as HTML</w:t>
</w:r> </w:p> <w:sectpr w:rsidr=

"00764658" w:rsidrpr="00764658">
<w:pgszw:h="15840" w:w="12240"></w:pgsz>

<w:pgmar w:bottom="1440" w:footer="720"
w:gutter="0" w:header="720" w:left="1440"
w:right="1440" w:top="1440"></w:pgmar> <w:cols
w:space="720"></w:cols>&g; <w:docgrid
w:linepitch="360"></w:docgrid> </w:sectpr>
</w:body> </w:document>

Здесь явно слишком много метаданных, в которых утонул
реально полезный для нас текст. К счастью, весь текст этого
документа, включая расположенный сверху заголовок,
заключен в теги w:t, благодаря чему его легко извлечь:

```
from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO
from bs4 import BeautifulSoup
```

```
wordFile =
urlopen('http://pythonscraping.com/pages/AWordD
ocument.docx').read()
wordFile = BytesIO(wordFile)
```

```

document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')

wordObj = BeautifulSoup(xml_content.decode('utf-8'),
    'xml')
textStrings = wordObj.find_all('w:t')

for textElem in textStrings:
    print(textElem.text)

```

Обратите внимание: вместо синтаксического анализатора `html.parser`, который обычно передается в объект `BeautifulSoup`, мы используем синтаксический анализатор `xml`. Дело в том, что стандарт HTML не предусматривает двоеточия в именах тегов, и `html.parser` не распознает теги наподобие `w:t`.

Результат неидеален, но приемлем, а благодаря тому, что каждый тег `w:t` выводится в отдельной строке, мы можем легко отследить разделение текста в Word:

A Word Document on a Website

This is a Word document, full of content that
you want very much. Unfortunately,
it's difficult to access because I'm putting it
on my website as a .

docx

file, rather than just publishing it as HTML

Обратите внимание: слово `docx` оказалось в отдельной строке. В исходном XML оно заключено в тег `<w:proofErrw:type="spellStart"/>`. Этим способом Word

снабжает слово docx красным волнистым подчеркиванием, указывая на то, что в названии его собственного формата файла есть орфографическая ошибка.

Заголовку документа предшествует тег дескриптора стиля `<w:pstylew:val="Title">`. Несмотря на то что этот тег не позволяет однозначно идентифицировать заголовки (или другой стиль текста), использование средств навигации BeautifulSoup может оказаться полезным:

```
textStrings = wordObj.find_all('w:t')

for textElem in textStrings:
    style =
textElem.parent.parent.find('w:pStyle')
    if style is not None and style['w:val'] ==
'Title':
        print('Title is:
{}'.format(textElem.text))
    else:
        print(textElem.text)
```

Эту функцию можно легко дополнить, чтобы текст разных стилей заключался в теги или выделялся другим способом.

[9](#) Этот бит «отступа» еще вернется, чтобы преследовать нас в стандартах ISO.

[10](#) Согласно W3Techs (http://w3techs.com/technologies/history_overview/character_encodin), где для сбора статистики подобного рода используются веб-краулеры.

[11](#) Организация ECMA одной из первых присоединилась к стандарту ISO, поэтому неудивительно, что ее сайт написан в кодировке ISO.

Глава 8. Очистка «грязных» данных

До сих пор в книге мы игнорировали проблему плохо отформатированных данных, используя источники, в которых данные отформатированы в целом хорошо, и полностью отбрасывая данные, если они отличались от ожидаемых. Но часто при просмотре веб-страниц не приходится быть слишком разборчивыми в том, откуда берутся данные или как выглядят.

Ошибки в пунктуации, неправильно поставленные прописные буквы, разрывы строк и опечатки приводят к тому, что «грязные» данные в Интернете могут стать большой проблемой. В текущей главе мы рассмотрим несколько инструментов и методов, которые помогут пресечь эту проблему в корне, изменив способ написания кода и очищая данные после их попадания в базу.

Очистка данных в коде

Подобно тому как мы пишем код для обработки явных исключений, следует взять за правило защитное кодирование для обработки непредвиденных ситуаций.

В лингвистике есть понятие «*n*-грамма» — последовательность из *n* слов, используемых в тексте или речи. При анализе естественного языка часто бывает удобно разбить текст на общеупотребительные *n*-граммы или повторяющиеся наборы слов, которые часто применяются вместе.

В этом разделе я лишь покажу, как получать правильно отформатированные *n*-граммы, не затрагивая их использование для какого-либо анализа. Позже, в главе 9, мы рассмотрим применение 2- и 3-грамм для обобщения и анализа текста.

Следующий код возвращает список 2-грамм, найденных в статье «Википедии» о языке программирования Python:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

def getNgrams(content, n):
    content = content.split(' ')
    output = []
    for i in range(len(content)-n+1):
        output.append(content[i:i+n])
    return output

html = urlopen('http://en.wikipedia.org/wiki/Python_(programming_language)')
bs = BeautifulSoup(html, 'html.parser')
content = bs.find('div', {'id':'mw-content-text'}).get_text()
ngrams = getNgrams(content, 2)
print(ngrams)
print('2-grams count is: '+str(len(ngrams)))
```

Функция `getNgrams` принимает исходную строку, разбивает ее на последовательность слов (при условии, что все слова разделены пробелами) и заносит в массив все n -граммы (в данном случае 2-граммы), которые начинаются с каждого слова строки.

В результате из этого текста можно получить несколько действительно любопытных и полезных 2-грамм:

```
['of', 'free'], ['free', 'and'], ['and', 'open-source'], ['open-source', 'software']
```

Но, кроме этого, код также возвращает много мусора:

```
[ 'software\outline\SPDX\n\n\n\n\n\n\n\noperating',  
  'system\families\n\n\n\nAROS\nBSD\nDarwin\necos\nFreeDOS\nGNU\nHaiku\nInferno\nLinux\nMach\nMINIX\nOpenSolaris\nPlan' ],  
[ 'system\families\n\n\n\nAROS\nBSD\nDarwin\necos\nFreeDOS\nGNU\nHaiku\nInferno\nLinux\nMach\nMINIX\nOpenSolaris\nPlan',  
  '9\nReactOS\nTUD:OS\n\n\n\n\n\n\n\nDevelopment\n\n\n\nBasic' ],  
[ '9\nReactOS\nTUD:OS\n\n\n\n\n\n\n\nDevelopment\n\n\n\nBasic', 'For']
```

Вдобавок, поскольку 2-граммы создаются для каждого слова в тексте (кроме последнего), на момент написания этой книги из статьи получилось 7411 2-грамм. Не особенно хорошо управляемый набор данных!

Используя регулярные выражения для удаления управляющих символов (таких как \n) и фильтрацию для удаления любых символов Unicode, можно немного очистить этот результат:

```
import re

def getNgrams(content, n):
    content = re.sub('\n|[[\d+\\]]', ' ',
content)
    content = bytes(content, 'UTF-8')
    content = content.decode('ascii', 'ignore')
    content = content.split(' ')
    content = [word for word in content if word
!= '']]
```



```
output = []
for i in range(len(content)-n+1):
    output.append(content[i:i+n])
return output
```

В этом коде все символы новой строки заменяются пробелами, удаляются цитаты наподобие [123] и отфильтровываются все пустые строки, из-за которых в тексте появляется несколько пробелов подряд. Затем удаляются экранирующие символы путем представления контента в кодировке UTF-8.

Эти шаги значительно улучшают результат функции, но некоторые проблемы все равно остаются:

```
['years', 'ago('], ['ago(', '-'], ['- ', '-'],
['-', ')'], [')', 'Stable']
```

Ситуацию можно улучшить, удалив все знаки препинания, стоящие до и после каждого слова (вообще убрав всю пунктуацию). При этом остаются такие элементы, как дефисы, но удаляются не только пустые строки, но и строки, состоящие из единственного знака препинания.

Конечно, пунктуация тоже важна и, просто удалив ее, можно потерять некую ценную информацию. Например, можно предположить, что точка, после которой стоит пробел, означает конец предложения или утверждения. Мы можем запретить n -граммы, которые пересекают подобные точки, и рассматривать только n -граммы в пределах одного предложения.

Например, для следующего текста:

```
Python features a dynamic type system and
automatic memory management.
```

It supports multiple programming paradigms...

2-грамма ['memory', 'management'] корректна, а 2-грамма ['management', 'It'] — нет.

Теперь, после того как у нас появился более длинный список «задач очистки», мы ввели концепцию «предложений» и вся наша программа немного усложнилась, лучше разделить эти задачи на четыре самостоятельные функции:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string

def cleanSentence(sentence):
    sentence = sentence.split(' ')
    sentence = [word.strip(string.punctuation+string.whitespace)
                 for word in sentence]
    sentence = [word for word in sentence if
                 len(word) > 1
                 or (word.lower() == 'a' or word.lower()
                     == 'i')]
    return sentence

def cleanInput(content):
    content = re.sub('\n|[[\d+\]]', ' ',
                    content)
    content = bytes(content, 'UTF-8')
    content = content.decode('ascii', 'ignore')
    sentences = content.split('. ')
```

```
        return [cleanSentence(sentence) for  
sentence in sentences]
```

```
def getNgramsFromSentence(content, n):  
    output = []  
    for i in range(len(content)-n+1):  
        output.append(content[i:i+n])  
    return output
```

```
def getNgrams(content, n):  
    content = cleanInput(content)  
    ngrams = []  
    for sentence in content:  
        ngrams.extend(getNgramsFromSentence(sentence, n))  
    return(ngrams)
```

Функция `getNgrams` остается основной точкой входа в программу; `cleanInput`, как и прежде, удаляет переводы строк и цитаты, а также разбивает текст на «предложения» в зависимости от местоположения точек, за которыми следует пробел. Функция `cleanInput` также вызывает `cleanSentence`, которая разбивает предложение на слова, удаляет знаки препинания и пробелы, а также односимвольные слова наподобие `I` и `a`.

Главные строки кода, собственно, и создающие n -граммы, перемещаются в функцию `getNgramsFromSentence`, вызываемую из `getNgrams` для каждого предложения. Это гарантирует, что не будут создаваться n -граммы, части которых принадлежат разным предложениям.

Обратите внимание на использование функций Python `string.punctuation` и `string.whitespace`, позволяющих

получить список всех знаков препинания. Результат выполнения функции `string.punctuation` можно увидеть в терминале Python, введя следующий код:

```
>>> import string
>>> print(string.punctuation)
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

Результат выполнения функции `print(string.whitespace)` гораздо менее интересен (в конце концов, это же пробелы), но в нем присутствуют пробельные символы, включая неразрывные пробелы, табуляции и переводы строк.

Функция `item.strip(string.punctuation+string.whitespace)`, используемая в цикле, перебирающем все слова исходного текста, удаляет все знаки препинания по обе стороны от слова, однако слова с дефисами остаются нетронутыми (поскольку в них по обе стороны знака препинания стоят буквы).

Результат наших усилий приводит к созданию намного более чистых 2-грамм:

```
[['Python', 'Paradigm'], ['Paradigm', 'Object-
oriented'], ['Object-oriented', 'imperative'],
['imperative', 'functional'], ['functional',
'procedural'], ['procedural', 'reflective'],...
```

Нормализация данных. Все мы не раз сталкивались с плохо разработанными веб-формами наподобие такой: «Введите номер телефона в формате “xxx-xxx-xxxx”».

Вы, как хороший программист, могли бы спросить: «Почему бы им самим просто не удалять нецифровые символы, которые я ввожу?» *Нормализация данных* — это процесс,

гарантирующий, что лингвистически или логически эквивалентные строки, такие как телефонные номера (555) 123-4567 и 555.123.4567, будут отображаться эквивалентно или по крайней мере считаться эквивалентными при сравнении.

Взяв за основу код создания n -грамм из предыдущего раздела, мы можем добавить к нему функции нормализации данных.

Первая очевидная проблема данного кода состоит в том, что он выдает много повторяющихся 2-грамм. Все 2-граммы, генерируемые этим кодом, заносятся в список без учета частоты появления. Интересно было бы фиксировать их частоту, вместо того чтобы просто констатировать их существование, — это может оказаться полезным для составления графика последствий изменений, которые вносятся в алгоритмы очистки и нормализации данных. Если данные хорошо нормализованы, то общее количество уникальных n -грамм уменьшится, в то время как общее количество всех найденных n -грамм (то есть уникальных и неуникальных элементов, идентифицированных как n -граммы) не изменится. Другими словами, будет меньше «групп» при том же количестве n -грамм.

Для этого мы можем изменить код таким образом, чтобы n -граммы собирались не в список, а в объект Counter:

```
from collections import Counter

def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = Counter()
    for sentence in content:
        newNgrams = [' '.join(gram) for gram
in
```

```
        getNgramsFromSentence(sentence, 2)]
    ngrams.update(newNgrams)
    return(ngrams)
```

Есть много других способов сделать это — например, вносить n -граммы в словарный объект, в котором значение элемента списка показывает, сколько раз данная n -грамма встречается в тексте. Недостаток такого подхода заключается в том, что он требует немного больше управления и усложняет сортировку. Однако у использования объекта Counter тоже есть недостаток: данный объект не позволяет хранить списки (они не хешируются), поэтому необходимо сначала преобразовать их в строки, используя операцию `' '.join(ngram)` при генерации списка для каждой n -граммы.

Получим следующие результаты:

```
Counter({'Python Software': 37, 'Software Foundation': 37, 'of the': 34, 'of Python': 28, 'in Python': 24, 'in the': 23, 'van Rossum': 20, 'to the': 20, 'such as': 19, 'Retrieved February': 19, 'is a': 16, 'from the': 16, 'Python Enhancement': 15,...})
```

На момент написания этой книги в статье о Python насчитывалось 7275 2-грамм, из которых 5628 были уникальными, причем наиболее популярной 2-граммой была Software Foundation, второй по популярности — Python Software. Однако анализ результатов показывает, что Python Software еще два раза встречается в виде Python software. Аналогично van Rossum и Van Rossum являются разными элементами списка.

Если добавить в функцию `cleanInput` строку:

```
content = content.upper()
```

то общее количество найденных 2-грамм останется равным 7275, но количество уникальных сократится до 5479.

Кроме того, обычно бывает полезным на минуту остановиться и прикинуть, сколько вычислительной мощности мы готовы потратить на нормализацию данных. В некоторых случаях разные варианты написания слов эквивалентны, но для установления этой эквивалентности необходимо проверить каждое слово и определить, соответствует ли оно какой-либо из предварительно запрограммированных эквивалентностей.

Например, в списке 2-грамм присутствуют варианты Python 1st и Python first. Однако создание общего правила, гласящего, что «Все слова first, second, third и т.д. преобразуются в 1st, 2nd, 3rd и т.д. (или наоборот)», потребует еще приблизительно десяти проверок для каждого слова.

Аналогичным образом, непоследовательное использование дефисов (например, наличие в одном тексте слов «иррегулярный» и «иррегулярный»), орфографические ошибки и другие несовпадения, встречающиеся в естественных языках, повлияют на группировки n -грамм и в случае достаточно распространенных несовпадений могут привести к искажению результатов программы.

Одним из решений в случае разрыва слов при переносах может быть полное удаление дефисов и обработка каждого слова как отдельной строки, что потребует только одной операции. Однако это значит следующее: словосочетания, содержащие дефисы (что встречается слишком часто) тоже будут рассматриваться как одно слово. Возможно, лучше пойти по другому пути и рассматривать дефисы как пробелы. Просто будьте готовы к внезапному нашествию «иррегулярных» и «регулярных» слов!

Очистка задним числом

Код делает только то, что мы можем (или хотим) сделать. Кроме того, может возникнуть необходимость обрабатывать набор данных, созданный не нами, или даже набор данных, который вообще непонятно как очистить, сначала его не увидев.

В подобной ситуации многие программисты по привычке рвутся сразу писать скрипт, который решил бы все проблемы. Однако есть инструменты сторонних разработчиков, такие как OpenRefine, позволяющие не только легко и быстро очищать данные, но и удобно просматривать и использовать эти данные людям, далеким от программирования.

OpenRefine

OpenRefine (<http://openrefine.org/>) — это проект с открытым исходным кодом, основанный в 2009 году компанией Metaweb. В 2010 году Google приобрела Metaweb, сменив название проекта с Freebase Gridworks на Google Refine. В 2012 году Google прекратила поддержку Refine и снова изменила ее название на OpenRefine, и теперь любой желающий может внести свой вклад в развитие проекта.

Установка

Продукт OpenRefine необычен тем, что, хотя его интерфейс запускается из браузера, технически это самостоятельное приложение, которое необходимо скачать и установить. На веб-странице загрузки OpenRefine (<http://openrefine.org/download.html>) есть версии приложения для Linux, Windows и macOS.



Если вы пользователь Mac и не смогли открыть файл, то выберите команду System Preferences→Security & Privacy→General (Установки системы→Защищенность & Конфиденциальность→Общие). В разделе Allow apps downloaded from (Разрешить скачивание приложений) выберите вариант Anywhere (Везде). К сожалению, после перехода от Google к открытому исходному коду проект OpenRefine, похоже, утратил легитимность в глазах Apple.

Чтобы использовать OpenRefine, нужно сохранить данные в виде CSV-файла (при необходимости освежить знания о том, как это делается, см. раздел «Хранение данных в формате CSV» на с. 111). Кроме того, если данные хранятся в базе, то их можно экспортировать в CSV-файл.

Использование OpenRefine

В следующих примерах мы будем использовать данные, взятые из таблицы сравнения текстовых редакторов в «Википедии» (https://en.wikipedia.org/wiki/Comparison_of_text_editors) (рис. 8.1). Несмотря на то что данная таблица относительно хорошо отформатирована, в нее в течение длительного времени вносили правки разные люди, поэтому ее форматирование слегка отличается. Кроме того, поскольку данные таблицы предназначены для чтения людьми, а не машинами, некоторые варианты форматирования (например, **Free** вместо **\$0.00**) неприемлемы для использования в качестве входных данных программы.

75 rows										Extensions: <div>Freebase</div>	
Show as: rows records		Show: 5 10 25 50 rows			« first < previous 1 - 50 next > last »						
<div><div></div></div> All	<div><div></div></div> Name	<div><div></div></div> Creator	<div><div></div></div> First public release	<div><div></div></div> Latest stable version	<div><div></div></div> Programming language	<div><div></div></div> Cost (US\$)	<div><div></div></div> Software license	<div><div></div></div> Open source			
<div>☆</div>	<div>🔊</div> 1. Acme	Rob Pike	1993	Plan 9 and Inferno	C	\$0	LPL (OSI approved)	Yes			
<div>☆</div>	<div>🔊</div> 2. AkelPad	Alexey Kuznetsov, Alexander Shergalis	2003	4.9.0	C	\$0	BSD	Yes			
<div>☆</div>	<div>🔊</div> 3. Alphasoft	Vince Darley	1999	8.3.3		\$40	Proprietary, with BSD components	No			
<div>☆</div>	<div>🔊</div> 4. Aquamacs	David Reitter	2005	3.0a	C, Emacs Lisp	\$0	GPL	Yes			
<div>☆</div>	<div>🔊</div> 5. Atom	Github	2014	0.132.0	HTML, CSS, JavaScript, C++	\$0	MIT	Yes			
<div>☆</div>	<div>🔊</div> 6. BBEdit	Rich Siegel	1992-04	10.5.12	Objective-C, Objective-C++	\$49.99	Proprietary	No			
<div>☆</div>	<div>🔊</div> 7. Bluefish	Bluefish Development Team	1999	2.2.6	C	\$0	GPL	Yes			
<div>☆</div>	<div>🔊</div> 8. Code	Panic	2007	2.0.12	Objective-C	\$99	Proprietary	No			
<div>☆</div>	<div>🔊</div> 9. ConTEXT	ConTEXT Project Ltd	1999	0.98.6	Object Pascal (Delphi)	\$0	BSD	Yes			
<div>☆</div>	<div>🔊</div> 10. Crimson Editor	Ingyu Kang, Emerald Editor Team	1999	3.72	C++	\$0	GPL	Yes			
<div>☆</div>	<div>🔊</div> 11. DiaKonos	Pistos	2004	0.9.2	Ruby	\$0	MIT	Yes			
<div>☆</div>	<div>🔊</div> 12. E Text Editor	Alexander Stigsen	2005	2.0.2		\$46.95	Proprietary, with BSD components	No			
<div>☆</div>	<div>🔊</div> 13. ed	Ken Thompson	1970	unchanged from original	C	\$0	?	Yes			
<div>☆</div>	<div>🔊</div> 14. EditPlus	Sangil Kim	1998	3.5	C++	\$35	Shareware	No			
<div>☆</div>	<div>🔊</div> 15. Editra	Cody Precord	2007	0.6.77	Python	\$0	wxWindows license	Yes			

Рис. 8.1. Данные из таблицы сравнения текстовых редакторов в «Википедии», открытые в главном окне OpenRefine

Первое, на что следует обратить внимание в OpenRefine, — каждый заголовок столбца снабжен стрелкой. Она открывает меню инструментов, с помощью которых можно фильтровать, сортировать, преобразовывать или удалять данные этого столбца.

Фильтрация данных можно выполнить двумя способами: с помощью фильтров или фасетов. Фильтры хороши при фильтрации данных путем регулярных выражений: например, «Показать только те строки, у которых в столбце “Язык программирования” присутствует три и более названия языков программирования, разделенных запятыми» (рис. 8.2).

Facet / Filter

Undo / Redo 1

Refresh

Reset All

Remove All

✕ Programming language

.+,+,+

☐ case sensitive☒ regular expression

5 matching rows (75 total)

Show as: rows records

Show: 5 10 25 50 rows

<div>▼ All</div>	<div>▼ Name</div>	<div>▼ Creator</div>	<div>▼ First public release</div>
<div>☆</div> <div>🗨</div> <div>5.</div>	Atom	Github	2014
<div>☆</div> <div>🗨</div> <div>28.</div>	Komodo Edit	Activestate	open-sourced 2007
<div>☆</div> <div>🗨</div> <div>29.</div>	Komodo IDE	Activestate	2000
<div>☆</div> <div>🗨</div> <div>59.</div>	Sublime Text	Jon Skinner	2000
<div>☆</div> <div>🗨</div> <div>74.</div>	Zed	Zef Hemel	2010

Рис. 8.2. Регулярное выражение `.+,+,+` выбирает значения, которые содержат как минимум три элемента, разделенных запятыми

Фильтры легко комбинировать, редактировать и добавлять, манипулируя блоками, предлагаемыми в правом столбце, а также можно комбинировать с фасетами.

Фасеты — отличное средство для включения данных в выбранное множество или исключения из него на основании контента всего столбца. (Например, «Показать все продукты с лицензией GPL или MIT, которые появились после 2005 года» (рис. 8.3).) Фасеты располагают собственными инструментами фильтрации. Так, для фильтрации по числовому значению есть ползунки, позволяющие выбирать диапазон значений, который мы хотим включить в выборку.

The screenshot shows a web application interface for searching software. On the left, there are two facet panels. The top panel, titled 'Software license', shows a list of license types: GPL (5), MIT (2), Proprietary (5), Proprietary, with BSD components (1), and wxWindows license (1). It includes a 'Cluster' button and 'exclude' links for each license. The bottom panel, titled 'First public release', shows a histogram of release dates with a range slider set from 2,005.00 to 2,015.00. Below the histogram are checkboxes for 'Numeric' (checked), 'Non-numeric', 'Blank', and 'Error'. On the right, a table displays '7 matching rows (75 total)'. The table has columns for 'All', 'Name', 'Creator', and 'First public release'. The rows list software like Aquamacs, Atom, Geany, Gobby, Light Table, Yi, and Zed, along with their creators and release dates.

	All	Name	Creator	First public release
4.	Aquamacs	David Reitter	2005-01-01	
5.	Atom	Github	2005-01-01	
19.	Geany	Enrico Trivèrger	2005-01-01	
21.	Gobby	0x539 dev group	2005-01-01	
33.	Light Table	Chris Granger	2005-01-01	
73.	Yi	Don Stewart	2005-01-01	
74.	Zed	Zef Hemel	2005-01-01	

Рис. 8.3. При таком варианте выбора отображаются все текстовые редакторы с лицензиями GPL или MIT, впервые появившиеся после 2005 года

Как бы вы ни фильтровали данные, их всегда можно экспортировать в один из форматов, поддерживаемый OpenRefine. В число этих форматов входят CSV, HTML (HTML-таблица), Excel и некоторые другие.

Очистка. Фильтрация данных может быть успешной только при условии, что данные изначально сравнительно чисты. Например, в рассмотренном в предыдущем разделе примере фасета текстовый редактор с датой выпуска 01-01-2006 не был бы выбран, так как фасет «Первый официальный выпуск» искал значение 2006 и игнорировал все значения, которые от него отличались.

Преобразование данных в OpenRefine выполняется с помощью языка выражений OpenRefine, называемого GREL (Google Refine Expression Language, в аббревиатуре сохранилось старое название OpenRefine — Google Refine). Этот язык используется для создания коротких лямбда-функций, которые трансформируют значения ячеек на основе простых правил. Например:

```
if(value.length() != 4, "invalid", value)
```

Будучи примененной к столбцу **First stable release** (Первая стабильная версия), эта функция сохраняет все ячейки с датой, значения которых имеют формат YYYY, и помечает все остальные как `invalid` (рис. 8.4).

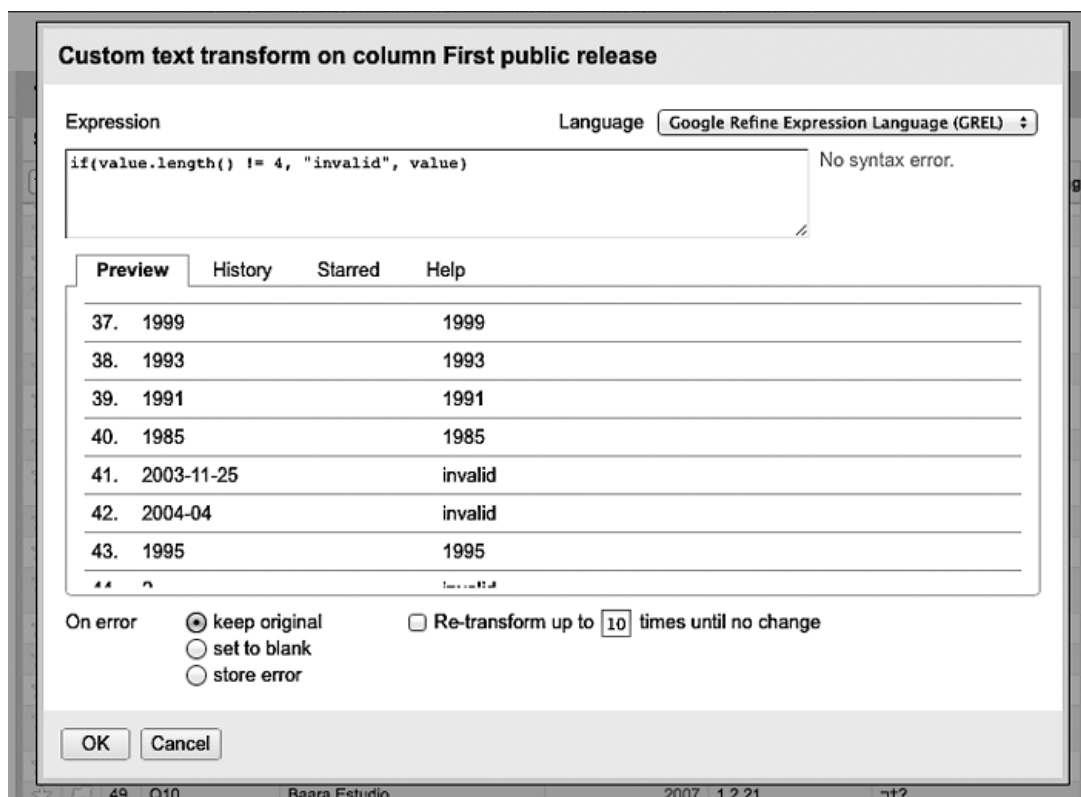


Рис. 8.4. Проект после вставки оператора GREL (предварительные результаты размещены под оператором)

Чтобы применить оператор GREL, следует нажать указывающую вниз стрелку, расположенную рядом с заголовком какого-либо столбца, и выбрать команду **Edit cells->Transform** (Редактирование ячеек->Преобразование).

Но если вы думаете, что маркировка всех неидеальных значений как недействительных облегчит их обнаружение, то сильно заблуждаетесь. Лучше попытаться по максимуму спасти информацию из плохо отформатированных значений. Это можно сделать с помощью функции GREL `match`:

```
value.match(".*([0-9]{4}).*").get(0)
```

Эта функция пытается сопоставить строковое значение с заданным регулярным выражением. Если данное выражение соответствует строке, то функция возвращает массив. Его

значениями являются все подстроки, которые соответствуют «группе захвата» в регулярном выражении (обозначены в нем круглыми скобками, в данном примере это `[0-9]{4}`).

Данный код, в сущности, находит все значения с четырьмя цифрами, идущими подряд, и возвращает первый такой фрагмент. Обычно этого достаточно, чтобы извлечь годы из текста или плохо отформатированной даты. Еще одним преимуществом описанного способа является то, что в случае несуществующей даты он возвращает `null`. (При выполнении операций с переменной `null` GREL не выдает исключение нулевого указателя.)

Редактируя ячейки и GREL, можно выполнять и многие другие преобразования данных. Полное руководство по этому языку доступно на странице OpenRefine на GitHub (<https://github.com/OpenRefine/OpenRefine/wiki/Documentation-For-Users>).

Глава 9. Чтение и запись текстов на естественных языках

До сих пор данные, с которыми мы работали, в основном представляли собой числа или перечислимые значения. В большинстве случаев мы просто сохраняли данные, никак их после этого не анализируя. В данной главе мы затронем сложную проблему английского языка¹².

Каким образом Google узнает, что вам нужно, когда вы вводите в строку поиска изображений слова `cutekitten` (симпатичный котенок)? По тексту, который окружает изображения этих самых котят. Каким образом YouTube узнает, что вам нужно показать определенный скетч «Монти Пайтона», если вы ввели в строку поиска слова `deadparrot` (мертвый попугай)? По заголовку и описанию, которые сопровождают каждое загруженное видео.

На самом деле, даже если ввести что-то вроде `deceasedbirdmontypython` (умершая птица монти пайтон), система все равно сразу же выдаст вам тот же скетч «Мертвый попугай», хотя на самой странице скетча нет слов `deceased` (умерший) или `bird` (птица). Google знает, что `hot dog` — это еда, а `boiling puppy` — вовсе нет. Каким образом? Все дело в статистике!

Вам может показаться, что анализ текста не имеет никакого отношения к вашему проекту, однако понимание концепций, лежащих в его основе, может быть чрезвычайно полезным для всех видов машинного обучения и вообще для возможности моделирования в виде алгоритмических и вероятностных задач.

Например, музыкальный сервис Shazam определяет, содержит ли данный аудиофрагмент конкретную песню, даже если в нем присутствуют фоновые помехи или искажения. Google разрабатывает автоматические подписи к изображениям, основываясь только на самом изображении¹³. Сравнивая известные изображения, скажем, хот-догов с другими их изображениями, поисковая система постепенно учится распознавать, как выглядит эта еда, и находить похожие детали на других изображениях, которые ей показывают.

Обобщение данных

В главе 8 было показано, как разделить текстовый контент на n -граммы — наборы фраз длиной n слов. В простейшем случае n -граммы можно использовать для определения того, какие наборы слов или фразы чаще всего встречаются в данном текстовом фрагменте. Вдобавок метод подходит для создания естественных на вид обобщений этих данных — если вернуться к исходному тексту и извлечь из него предложения, заключающие в себе некоторые из наиболее распространенных фраз.

Одним из примеров текста, который мы будем использовать для этого, является инаугурационная речь девятого президента Соединенных Штатов Америки Уильяма Генри Харрисона (William Henry Harrison). Он установил два рекорда в истории института президентов США: самая длинная инаугурационная речь и самое короткое время пребывания в должности — 32 дня.

Мы возьмем полный текст этой речи (<http://pythonscraping.com/files/inaugurationSpeech.txt>) в качестве источника данных для нескольких примеров в этой главе.

Слегка изменив код для поиска n -грамм, описанный в главе 8, можно создать код, который находил бы множества 2-грамм и возвращал объект Counter, содержащий все 2-граммы:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string
from collections import Counter

def cleanSentence(sentence):
    sentence = sentence.split(' ')
    sentence = [word.strip(string.punctuation+string.whitespace)
    for word in sentence]
    sentence = [word for word in sentence if len(word) > 1
    or (word.lower() == 'a' or word.lower() == 'i')]
    return sentence

def cleanInput(content):
    content = content.upper()
    content = re.sub('\n', ' ', content)
    content = bytes(content, "UTF-8")
    content = content.decode("ascii", "ignore")
    sentences = content.split('. ')
    return [cleanSentence(sentence) for sentence in sentences]

def getNgramsFromSentence(content, n):
```

```

        output = []
        for i in range(len(content)-n+1):
            output.append(content[i:i+n])
        return output

def getNgrams(content, n):
    content = cleanInput(content)
    ngrams = Counter()
    ngrams_list = []
    for sentence in content:
        newNgrams = [' '.join(gram) for gram
in
                        getNgramsFromSentence(sentence, 2)]
        ngrams_list.extend(newNgrams)
        ngrams.update(newNgrams)
    return(ngrams)

content = str(
    urlopen('http://pythonscraping.com/files/
inaugurationSpeech.txt')
        .read(), 'utf-8')
ngrams = getNgrams(content, 2)
print(ngrams)

```

На выходе получим, в частности, следующее:

```

Counter({'OF THE': 213, 'IN THE': 65, 'TO THE':
61, 'BY THE': 41,

'THE CONSTITUTION': 34, 'OF OUR': 29, 'TO BE':
26, 'THE PEOPLE': 24,

'FROM THE': 24, 'THAT THE': 23,...

```

Из этих 2-грамм *the constitution* представляется довольно частой темой речи, но 2-граммы *of the*, *in the* и *to the* едва ли заслуживают внимания. Можно ли каким-либо достаточно точным автоматическим способом избавиться от нежелательных слов?

К счастью, есть люди, которые тщательно изучают разницу между «интересными» и «неинтересными» словами, и их работа поможет нам решить эту задачу. Марк Дэвис (Mark Davies), профессор лингвистики в Университете Бригама Янга, собирает «Корпус современного американского английского языка» (<http://corpus.byu.edu/coca/>) — коллекцию из более 450 миллионов слов из популярных публикаций, выпущенных в США за последние десять лет.

Список из 5000 наиболее часто встречающихся слов доступен бесплатно. К счастью, этого более чем достаточно, чтобы их можно было использовать в качестве базового фильтра, позволяющего отсеять наиболее распространенные 2-граммы. Благодаря добавлению функции `isCommon` уже первые 100 слов значительно улучшили наш результат:

```
def isCommon(ngram):
    commonWords = ['THE', 'BE', 'AND', 'OF',
                   'A', 'IN', 'TO', 'HAVE', 'IT', 'I',
                   'THAT', 'FOR', 'YOU', 'HE', 'WITH',
                   'ON', 'DO', 'SAY', 'THIS', 'THEY',
                   'IS', 'AN', 'AT', 'BUT', 'WE', 'HIS',
                   'FROM', 'THAT', 'NOT', 'BY',
                   'SHE', 'OR', 'AS', 'WHAT', 'GO',
                   'THEIR', 'CAN', 'WHO', 'GET', 'IF',
                   'WOULD', 'HER', 'ALL', 'MY', 'MAKE',
                   'ABOUT', 'KNOW', 'WILL', 'AS',
                   'UP', 'ONE', 'TIME', 'HAS', 'BEEN',
                   'THERE', 'YEAR', 'SO', 'THINK',
```

```

        'WHEN', 'WHICH', 'THEM', 'SOME', 'ME',
        'PEOPLE', 'TAKE', 'OUT', 'INTO',
        'JUST', 'SEE', 'HIM', 'YOUR', 'COME',
        'COULD', 'NOW', 'THAN', 'LIKE',
        'OTHER', 'HOW', 'THEN', 'ITS', 'OUR',
        'TWO', 'MORE', 'THESE', 'WANT',
        'WAY', 'LOOK', 'FIRST', 'ALSO', 'NEW',
        'BECAUSE', 'DAY', 'MORE', 'USE',
        'NO', 'MAN', 'FIND', 'HERE', 'THING',
        'GIVE', 'MANY', 'WELL']
    for word in ngram:
        if word in commonWords:
            return True
    return False

```

В итоге мы получили следующие 2-граммы, которые встречаются в тексте более двух раз:

```

Counter({'UNITED STATES': 10, 'EXECUTIVE
DEPARTMENT': 4, 'GENERAL GOVERNMENT': 4,
'CALLED UPON': 3, 'CHIEF MAGISTRATE': 3,
'LEGISLATIVE BODY': 3, 'SAME CAUSES': 3,
'GOVERNMENT SHOULD': 3, 'WHOLE COUNTRY': 3,...

```

Неудивительно, что первыми в списке идут *United States* (Соединенные Штаты) и *executive department* (исполнительная власть), — этого следовало ожидать от инаугурационной речи президента.

Важно подчеркнуть: мы используем для фильтрации результатов сравнительно свежий список слов, распространенных в наше время. Это может оказаться неуместным, если учесть, что текст был написан в 1841 году. Однако, поскольку мы использовали примерно первые 100 слов данного списка (которые можно считать менее

подверженными временным изменениям, чем, скажем, последние 100 слов) и, похоже, получили удовлетворительные результаты, то, вероятно, можем сэкономить усилия и не создавать список слов, наиболее распространенных в 1841 году (хотя это могло бы быть интересным).

Теперь, когда мы извлекли из текста некоторые ключевые темы, как это поможет написать его резюме? Один из способов — найти первое предложение, содержащее каждую из «популярных» n -грамм. Теоретически, выбрав для каждой из них первый такой экземпляр, мы получим удовлетворительный обзор всего текста. Первые пять самых популярных 2-грамм дают следующие пункты.

- *The Constitution of the United States is the instrument containing this grant of power to the several departments composing the Government.*
- *Such a one was afforded by the executive department constituted by the Constitution. The General Government has seized upon none of the reserved rights of the States.*
- *The General Government has seized upon none of the reserved rights of the States.*
- *Called from a retirement which I had supposed was to continue for the residue of my life to fill the chief executive office of this great and free nation, I appear before you, fellow-citizens, to take the oaths which the constitution prescribes as a necessary qualification for the performance of its duties; and in obedience to a custom coeval with our government and what I believe to be your expectations I proceed to present to you a summary of the principles which will govern me in the discharge of the duties which I shall be called upon to perform.*

- *The presses in the necessary employment of the Government should never be used to “clear the guilty or to varnish crime”.*

Конечно, такое резюме вряд ли в обозримом будущем напечатают в CliffsNotes, но, учитывая, что размер исходного документа составляет 217 предложений, а четвертое (*Called from a retirement...*) довольно хорошо отражает основную тему, это не так уж плохо для первой попытки.

Для получения «самых важных» предложений из более длинного или разнообразного текста, возможно, стоит рассмотреть 3- или даже 4-граммы. В данном случае только одна 3-грамма используется несколько раз, и это *exclusive metallic currency* — едва ли характерная фраза для инаугурационной речи президента. Для более длинных оборотов имеет смысл использовать 3-граммы.

Другой подход заключается в поиске предложений, содержащих самые популярные n -граммы. Конечно, это будут более длинные предложения и в случае возникновения проблем можно искать предложения с наибольшим процентом слов, которые являются популярными n -граммами, или создать собственную метрику оценки, комбинируя несколько приемов.

Модели Маркова

Возможно, вам приходилось слышать о текстовых генераторах Маркова. Они стали популярными в развлекательных приложениях, таких как сайт That can be my next tweet! («Это мог бы быть мой следующий твит!») (<http://yes.thatcan.be/my/next/tweet/>), а также применяются для создания спам-писем, которые выглядят как настоящие и призваны обмануть системы обнаружения спама.

Все эти текстовые генераторы основаны на модели Маркова, которая часто используется для анализа больших множеств случайных событий, где каждое дискретное событие может сопровождаться другим дискретным событием с определенной вероятностью.

Например, можно построить модель Маркова для системы прогноза погоды, как показано на рис. 9.1.

В этой модели вероятность того, что следующий день после солнечного тоже будет солнечным, составляет 70 %, пасмурным — 20 %, а дождливым — 10 %. Если день дождливый, то вероятность того, что на следующий день будет дождь, составляет 50 %, вероятность ясной погоды — 25 %, пасмурной — также 25 %.

Вы могли заметить, что модель Маркова имеет следующие свойства.

- Все вероятности, исходящие из одного узла, в сумме всегда составляют ровно 100 %. Независимо от того, насколько сложна система, всегда должна быть 100-процентная вероятность того, что на следующем этапе она может привести еще куда-либо.
- Хотя в данной модели в любой момент существует лишь три возможных варианта погоды, ее можно расширить, создав бесконечный список состояний погоды.

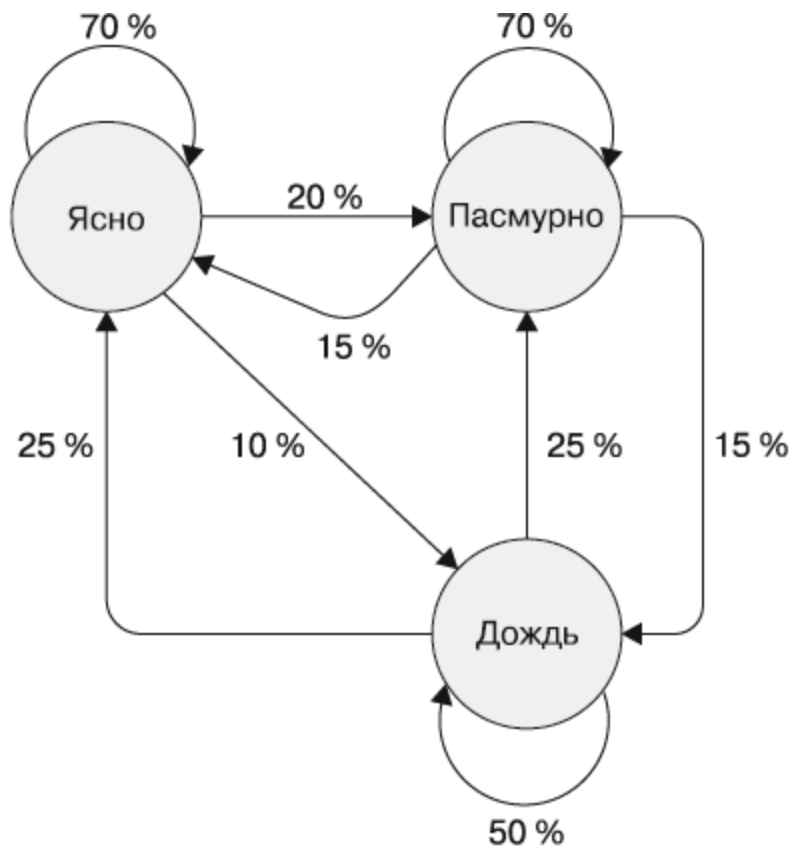


Рис. 9.1. Модель Маркова, описывающая теоретическую систему прогноза погоды

- Узел, в который вы перейдете, определяется только состоянием текущего узла. Если вы находитесь в узле «Ясно», то не имеет значения, были предыдущие 100 дней ясными или дождливыми, — вероятность ясной погоды на следующий день все равно составляет 70 %.
- Может оказаться так, что достичь одних узлов будет труднее, чем других. Стоящая за этим математика достаточно сложна, однако нетрудно заметить: в данной системе «Дождь» в любой момент времени является гораздо менее вероятным состоянием, чем «Ясно» или «Пасмурно» (поскольку сумма стрелок, указывающих на «Дождь», меньше 100 %).

Очевидно, что это простая система и модели Маркова могут быть сколь угодно большими. Алгоритм ранжирования страниц Google частично основан на данной модели: сайты представлены в виде узлов, а входящие и исходящие ссылки — в виде связей между ними. «Вероятность» попасть в определенный узел соответствует относительной популярности сайта. Другими словами, если бы наша метеорологическая система представляла собой чрезвычайно маленький Интернет, то «дождливый» сайт имел бы низкий рейтинг страниц, а «пасмурный» — высокий.

Учитывая это, вернемся к более конкретному примеру: анализу и написанию текста.

Снова используя инаугурационную речь Уильяма Генри Харрисона, проанализированную в предыдущем примере, мы можем написать следующий код, который будет генерировать на основе структуры этого текста цепи Маркова произвольной длины (сейчас длина цепи равна 100):

```
from urllib.request import urlopen
from random import randint

def wordListSum(wordList):
    sum = 0
    for word, value in wordList.items():
        sum += value
    return sum

def retrieveRandomWord(wordList):
    randIndex = randint(1,
wordListSum(wordList))
    for word, value in wordList.items():
        randIndex -= value
```

```

        if randIndex <= 0:
            return word

def buildWordDict(text):
    # удаляем разрывы строк и кавычки
    text = text.replace('\n', ' ');
    text = text.replace('"', '');

    # Убедимся, что знаки препинания
    рассматриваются как отдельные "слова",
    # чтобы они тоже включались в цепь Маркова.
    punctuation = [',', '.', ';', ':']
    for symbol in punctuation:
        text = text.replace(symbol, ' {}'.format(symbol));

    words = text.split(' ')
    # убираем пустышки
    words = [word for word in words if word != '']

    wordDict = {}
    for i in range(1, len(words)):
        if words[i-1] not in wordDict:
            # Создаем новый словарь для этого
            слова.
            wordDict[words[i-1]] = {}
        if words[i] not in wordDict[words[i-1]]:
            wordDict[words[i-1]][words[i]] = 0
            wordDict[words[i-1]][words[i]] += 1

```

```

        return wordDict

text =
str(urlopen('http://pythonscraping.com/files/in
augurationSpeech.txt')
        .read(), 'utf-8')
wordDict = buildWordDict(text)

# Генерируем цепь Маркова длиной 100.
length = 100
chain = ['I']
for i in range(0, length):
    newWord =
    retrieveRandomWord(wordDict[chain[-1]])
    chain.append(newWord)

print(' '.join(chain))

```

Результат выполнения этого кода меняется при каждом запуске программы. Вот пример совершенно бессмысленного текста, который он сгенерировал:

```

I sincerely believe in Chief Magistrate to make
all necessary sacrifices and oppression of the
remedies which we may have occurred to me in
the arrangement and disbursement of the
democratic claims them , consolatory to have
been best political power in fervently
commending every other addition of legislation
, by the interests which violate that the
Government would compare our aboriginal
neighbors the people to its accomplishment .
The latter also susceptible of the Constitution

```

not much mischief , disputes have left to betray . The maxim which may sometimes be an impartial and to prevent the adoption or

Что же делает этот код?

Функция `buildWordDict` принимает строку текста, полученную из Интернета. Выполняет некую очистку и форматирование: удаляет кавычки и ставит пробелы вокруг остальных знаков препинания, чтобы они фактически рассматривались как отдельные слова. Затем создает двумерный словарь (словарь словарей) такого вида:

```
{word_a : {word_b : 2, word_c : 1, word_d : 1},  
word_e : {word_b : 5, word_d : 2},...}
```

В этом примере словаря в тексте было обнаружено четыре вхождения слова `word_a`, после двух из которых следовало слово `word_b`, после одного — `word_c` и еще после одного — `word_d`. Слово `word_e` встретилось семь раз: пять раз после него стояло `word_b` и дважды — `word_d`.

Построив узловую модель этого результата, мы увидим, что из узла `word_a` выходит стрелка с пометкой «50 %», указывающая на `word_b` (которое в тексте встречается после `word_a` два раза из четырех), стрелка с пометкой «25 %», указывающая на `word_c`, и еще одна стрелка с пометкой «25 %», указывающая на `word_d`.

Создав такой словарь, его затем можно использовать в качестве справочной таблицы, чтобы увидеть, куда переходить дальше, независимо от того, на каком слове в тексте мы остановились¹⁴. В данном примере словаря словарей, если мы сейчас находимся на слове `word_e`, можем передать словарь `{word_b:5,word_d:2}` в функцию `retrieveRandomWord`. Эта

функция, в свою очередь, извлечет из словаря случайное слово, взвешенное по числу его вхождений в текст.

Начав со случайного исходного слова (в данном случае вездесущего I — «Я»), мы можем легко перемещаться по цепи Маркова, генерируя столько слов, сколько захотим.

Вдобавок цепи Маркова имеют тенденцию быть тем «реалистичнее», чем больше текста собирается, особенно из источников с похожим авторским стилем. В этом примере для создания цепи использовались 2-граммы (в которых следующее слово определяется по предыдущему), однако можно задействовать 3-граммы или n -граммы высшего порядка, где каждое следующее слово определяется двумя или более словами.

Несмотря на интересные результаты и большие выгоды, которые можно получить, накопив много мегабайтов текста в процессе веб-скрапинга, из-за подобного применения бывает трудно заметить практическую пользу цепей Маркова. Ранее в этом разделе отмечалось, что цепи Маркова моделируют то, как сайты переходят с одной страницы на другую. Большие коллекции подобных ссылок, представленных в виде указателей, позволяют строить веб-графы, полезные для хранения, отслеживания и анализа. То есть цепочки Маркова помогают сформировать основу для того, чтобы вы могли представить процесс веб-краулинга, а также для описания логики действия самих веб-краулеров.

Шесть шагов по «Википедии»: заключение. В главе 3 мы создали веб-скрапер, который собирает ссылки одной статьи «Википедии» на следующую, начиная со статьи о Кевине Бейконе, а в главе 6 добавили сохранение ссылок в базе данных. Почему я снова об этом вспоминаю? Потому что, оказывается, задача выбора пути по ссылкам от начальной до заданной конечной страницы (то есть поиск цепочки страниц

от https://en.wikipedia.org/wiki/Kevin_Bacon до https://en.wikipedia.org/wiki/Eric_Idle) — то же самое, что и построение цепочки Маркова, в которой заданы первое и последнее слово.

Задачи такого рода относятся к задачам *направленных графов*, где наличие связи $A \rightarrow B$ еще не означает наличие связи $B \rightarrow A$. После слова «футбол» часто стоит слово «игрок», однако вы обнаружите, что после слова «игрок» слово «футбол» встречается гораздо реже. Хотя в статье о Кевине Бейконе в «Википедии» есть ссылка на статью о его родном городе Филадельфии, статья о Филадельфии не отвечает взаимностью — в ней нет ссылки на страницу актера.

И наоборот, исходная игра «Шесть шагов до Кевина Бейкона» представляет собой задачу *ненаправленного графа*. Если известно, что Кевин Бейкон снимался в фильме «Коматозники» с Джулией Робертс (Julia Roberts), то Джулия Робертс тоже снималась в «Коматозниках» с Кевином Бейконом, поэтому здесь отношения являются двусторонними (у них нет «направления»). Задачи ненаправленных графов, как правило, встречаются в информатике реже, чем задачи направленных графов, но и те и другие достаточно сложны в вычислительном отношении.

Несмотря на большую проделанную работу по этим видам задач и множеству их разновидностей, одним из лучших и самых распространенных способов найти кратчайшие пути в направленном графе — и, таким образом, найти пути между статьей «Википедии» о Кевине Бейконе и любой другой статьей «Википедии» — остается поиск в ширину.

При *поиске в ширину* сначала выполняется поиск всех ссылок, ведущих непосредственно на начальную страницу. Если среди них нет конечной страницы (той, которую мы ищем), то выполняется поиск среди ссылок второго уровня — тех страниц, на которые ссылается страница, связанная

ссылкой с начальной. Этот процесс продолжается до тех пор, пока не будет достигнута предельная глубина (в данном случае 6) или найдена конечная страница.

Полная реализация поиска в ширину с помощью таблицы ссылок, описанной в главе 6, выглядит так:

```
import pymysql

conn      =      pymysql.connect(host='127.0.0.1',
unix_socket='/tmp/mysql.sock',
                                user='',    passwd='',    db='mysql',
charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def getUrl(pageId):
    cur.execute('SELECT url FROM pages WHERE id
= %s', (int(pageId)))
    return cur.fetchone()[0]

def getLinks(fromPageId):
    cur.execute('SELECT toPageId FROM links
WHERE fromPageId = %s',
                (int(fromPageId)))
    if cur.rowcount == 0:
        return []
    return [x[0] for x in cur.fetchall()]

def searchBreadth(targetPageId, paths=[[1]]):
    newPaths = []
    for path in paths:
        links = getLinks(path[-1])
```

```

        for link in links:
            if link == targetPageId:
                return path + [link]
            else:
                newPaths.append(path+[link])
        return searchBreadth(targetPageId,
newPaths)

nodes = getLinks(1)
targetPageId = 28624
pageIds = searchBreadth(targetPageId)
for pageId in pageIds:
    print(getUrl(pageId))

```

Здесь `getUrl` — вспомогательная функция, извлекающая URL из базы данных по идентификатору страницы. Функция `getLinks` работает аналогично, принимая `fromPageId` — целочисленный идентификатор текущей страницы — и извлекая список целочисленных ID всех страниц, на которые ссылается текущая страница.

Основная функция, `searchBreadth`, рекурсивно строит список всех возможных путей от страницы поиска и останавливается, когда находит путь, ведущий к конечной странице.

- Функция начинает работу с одного пути — `[1]`. Перейдя по нему, пользователь остается на конечной странице с идентификатором 1 (Кевин Бейкон) и не переходит по ссылкам.
- Для каждого пути в списке (при первом проходе есть только один путь, поэтому данный шаг короткий) функция находит

все ссылки, которые ведут со страницы, представляющей последнюю страницу пути.

- Для каждой из этих исходящих ссылок алгоритм проверяет, соответствуют ли они `targetPageId`. Если такое совпадение найдено, возвращается данный путь.
- При отсутствии совпадения новый путь добавляется в новый список (теперь более длинных) путей, состоящий из старого пути и новой исходящей ссылки с текущей страницы.
- Если `targetPageId` на этом уровне вообще не найден, то выполняется рекурсия и `searchBreadth` вызывается с тем же `targetPageId` и новым, более длинным списком путей.

После того как будет найден список идентификаторов страниц, содержащий путь между двумя страницами, все его ID преобразуются в соответствующие URL и выводятся на экран.

Ниже показан результат поиска ссылки между страницей Кевина Бейкона (которая в этой базе данных имеет идентификатор 1) и страницей Эрика Айбла (с идентификатором страницы 28 624):

```
/wiki/Kevin_Bacon  
/wiki/Primetime_Emma_Award_for_Outstanding_Lead  
_Actor_in_a_  
Miniseries_or_a_Movie  
/wiki/Gary_Gilmore  
/wiki/Eric_Idle
```

Если преобразовать это в отношения ссылок, то получим следующее: Кевин Бейкон→Прайм-таймовая премия «Эмми»→Гэри Гилмор→Эрик Айбл.

Помимо решения задач о шести шагах и построения моделей того, какие слова обычно следуют за другими в предложениях, направленные и ненаправленные графы могут применяться для моделирования различных ситуаций, возникающих при веб-скрапинге. Какие сайты ссылаются на другие характерные сайты? Какие исследовательские работы связаны ссылками? Какие продукты обычно предлагают с другими продуктами в интернет-магазинах? В чем сила той или иной ссылки? Взаимна ли эта связь?

Распознавание этих фундаментальных типов отношений может быть чрезвычайно полезным для построения моделей, визуализаций и прогнозов на основе данных, собранных в процессе веб-скрапинга.

Natural Language Toolkit

До сих пор в этой главе основное внимание уделялось статистическому анализу слов в тексте. Какие слова наиболее популярны, а какие не свойственны данному тексту? Какие слова обычно идут после тех или иных слов? Какие группы они образуют? Чего нам еще не хватает, так это возможности понять, что именно представляют собой данные слова.

Natural Language Toolkit (NLTK) — набор библиотек Python, предназначенных для идентификации и маркировки частей речи, встречающихся в тексте, написанном на естественном английском языке. NLTK начали создавать в 2000 году, и за последние 15 лет десятки разработчиков из разных стран внесли свой вклад в этот проект. У NLTK огромный функционал (чему посвящены целые книги), однако в текущем разделе мы рассмотрим лишь несколько способов применения данной библиотеки¹⁵.

Установка и настройка

Модуль `nltk` можно установить так же, как и другие модули Python: либо скачать пакет напрямую через сайт NLTK, либо воспользоваться одним из многочисленных сторонних инсталляторов с ключевым словом **nltk**. Подробные инструкции по установке модуля см. на сайте NLTK (<http://www.nltk.org/install.html>).

После установки NLTK рекомендуется скачать его предустановленные текстовые репозитории, чтобы было удобнее пользоваться данными функциями. Для этого нужно ввести в командной строке Python следующее:

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

В результате запустится NLTK Downloader (рис. 9.2).

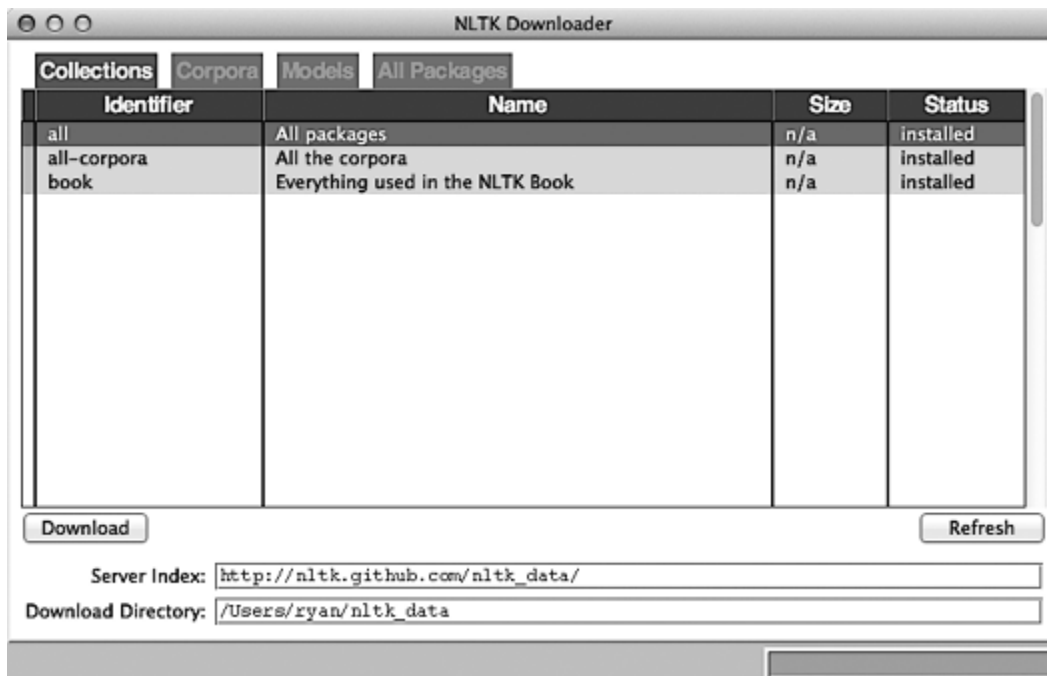


Рис. 9.2. NLTK Downloader позволяет просматривать и загружать дополнительные пакеты и текстовые библиотеки, связанные с модулем `nltk`

При первом знакомстве с корпусом NLTK я рекомендую установить все доступные пакеты. Вы сможете легко удалить лишние пакеты в любой момент.

Статистический анализ с помощью NLTK

NLTK отлично подходит для получения статистической информации о количестве слов, частоте и разнообразии слов в разделах текста. Если вам нужны лишь относительно простые вычисления (такие как количество уникальных слов, используемых в разделе текста), то импорт `nltk` может оказаться излишним — все же это большой модуль. Но если вам нужно провести сравнительно обширный анализ текста, то у вас под рукой будут функции, способные вычислить практически любую необходимую метрику.

Анализ с применением NLTK всегда начинается с объекта `Text`. Объекты `Text` создаются с помощью всего пары простых строк на Python:

```
from nltk import word_tokenize
from nltk import Text

tokens = word_tokenize('Here is some not very
interesting text')
text = Text(tokens)
```

На вход функции `word_tokenize` подается любая текстовая строка Python. На случай, если у вас нет подходящих длинных строк, но вы все равно хотели бы поэкспериментировать с этими функциями, в библиотеку NLTK встроено несколько книг, к которым можно получить доступ с помощью функции `import`:

```
from nltk.book import *
```

В результате будет импортировано девять книг:

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view
it.
```

```
Type: 'texts()' or 'sents()' to list the
materials.
```

```
text1: Moby Dick by Herman Melville 1851
```

```
text2: Sense and Sensibility by Jane Austen
1811
```

```
text3: The Book of Genesis
```

```
text4: Inaugural Address Corpus
```

```
text5: Chat Corpus
```

```
text6: Monty Python and the Holy Grail
```

```
text7: Wall Street Journal
```

```
text8: Personals Corpus
```

```
text9: The Man Who Was Thursday by G . K .
Chesterton 1908
```

Во всех следующих примерах мы будем работать с `text6`, `Monty Python and the Holy Grail` («Монти Пайтон и священный Грааль», киносценарий 1975 года).

К текстовым объектам применимы все те же манипуляции, что и к обычным массивам Python: мы рассматриваем текст как массив, состоящий из слов этого текста. Используя данное свойство, можно подсчитать количество уникальных слов в тексте и сравнить его с общим количеством слов (как мы помним, объекты `set` в Python содержат только уникальные значения):

```
>>> len(text6)/len(set(text6))
7.833333333333333
```

Как видим, каждое слово в этом скрипте использовалось в среднем около восьми раз. Мы также можем поместить текст в объект распределения частот, чтобы определить наиболее распространенные слова и частоту появления различных слов:

```
>>> from nltk import FreqDist
>>> fdist = FreqDist(text6)
>>> fdist.most_common(10)
[(':', 1197), ('.', 816), ('!', 801), (',', 731), ('"', 421), ('[', 319),
(']', 312), ('the', 299), ('I', 255),
('ARTHUR', 225)]
>>> fdist["Grail"]
34
```

Поскольку это скрипт, то в нем могут появляться отдельные специфические элементы, свойственные именно скриптам. Так, слово ARTHUR, написанное заглавными буквами, часто встречается потому, что ставится перед каждой репликой короля Артура. Кроме того, перед каждой новой репликой стоит двоеточие (:), которое служит разделителем между ней и именем персонажа. Используя данный факт, легко подсчитать, что в фильме 1197 реплик!

То, что в предыдущих главах мы называли 2-граммами, NLTK называет *биграммами* (иногда также 3-граммы называют *триграммами*, но я предпочитаю не биграммы и триграммы, а 2-граммы и 3-граммы). Создание, поиск и перечисление 2-грамм выполняются очень легко:

```
>>> from nltk import bigrams
```

```
>>> bigrams = bigrams(text6)
>>> bigramsDist = FreqDist(bigrams)
>>> bigramsDist[('Sir', 'Robin')]
18
```

Для нахождения всех 2-грамм *Sir Robin* нужно разбить эту 2-грамму на кортеж (*Sir*, *Robin*), чтобы она соответствовала представлению 2-грамм в распределении частот. Вдобавок существует модуль `trigrams`, который работает точно так же. Для общего случая можно импортировать модуль `ngrams`:

```
>>> from nltk import ngrams
>>> fourgrams = ngrams(text6, 4)
>>> fourgramsDist = FreqDist(fourgrams)
>>> fourgramsDist[('father', 'smelt', 'of',
'elderberries')]
1
```

Здесь функция `ngrams` вызывается для разбиения текстового объекта на n -граммы любого размера, который определяется вторым параметром функции. В данном случае мы разбиваем текст на 4-граммы. Затем можно продемонстрировать, что фраза *father smelt of elderberries* («отец пахнет бузиной») встречается в скрипте ровно один раз.

Распределения частот, текстовые объекты и n -граммы также могут перебираться и обрабатываться в цикле. Например, в следующем коде выводятся на экран все 4-граммы, которые начинаются со слова *coconut* (кокос):

```
from nltk.book import *
from nltk import ngrams
fourgrams = ngrams(text6, 4)
for fourgram in fourgrams:
```

```
if fourgram[0] == 'coconut':  
    print(fourgram)
```

В состав библиотеки NLTK входит обширный набор инструментов и объектов, предназначенных для упорядочения, подсчета, сортировки и измерения параметров больших объемов текста. Мы лишь едва затронули варианты их применения, однако большинство этих инструментов хорошо спроектированы и интуитивно вполне понятны для тех, кто знаком с Python.

Лексикографический анализ с помощью NLTK

До сих пор мы сравнивали и классифицировали все слова в тексте только по их собственному значению. Мы не делали различий между омонимами и не учитывали контекст, в котором использовались эти слова.

Некоторые считают, что омонимы не стоит учитывать, так как они редко представляют интерес. Однако вы будете удивлены, насколько часто они встречаются. Большинство носителей английского языка обычно даже не замечают, что то или иное слово является омонимом, и тем более не считают, будто в другом контексте это слово можно спутать с другим.

Предложение *He was objective in achieving his objective of writing an objective philosophy, primarily using verbs in the objective case*¹⁶ вполне понятно для человека, но веб-скрапер может посчитать, что одно и то же слово используется здесь четыре раза, и просто отбросить какую-либо информацию о значении каждого из этих слов.

Кроме определения частей речи, может быть полезно различать способы употребления слов. Например, можно находить названия компаний, составленные из обычных английских слов, или анализировать чьи-либо мнения о

компании. Смысл предложений «Продукты АСМЕ — это хорошо» и «Продукты АСМЕ — это неплохо», в сущности, один и тот же, несмотря на то что в одном из них используется слово «хорошо», а в другом — «плохо».

Теги Penn Treebank

В NLTK по умолчанию применяется популярная система обозначения частей речи, разработанная в Пенсильванском университете в рамках проекта Penn Treebank (<https://catalog.ldc.upenn.edu/LDC99T42>). Хотя одни обозначения в ней имеют определенный смысл (так, CC — это coordinating conjunction, соединительный союз), другие способны сбить с толку (например, RP означает particle — «частица»). Ниже приводится таблица обозначений, использованных в данном подразделе.

CC	Coordinating conjunction	Соединительный союз
CD	Cardinal number	Количественное числительное
DT	Determiner	Определяющее слово
EX	Existential “there”	Бытийное there
FW	Foreign word	Иностранное слово
IN	Preposition, subordinating conjunction	Предлог, подчинительный союз
JJ	Adjective	Прилагательное
JJR	Adjective, comparative	Прилагательное в сравнительной степени
JJS	Adjective, superlative	Прилагательное в превосходной степени
LS	List item marker	Обозначение элемента списка
MD	Modal	Модальный
NN	Noun, singular or mass	Существительное в единственном числе или неисчисляемое

NNS	Noun, plural	Существительное во множественном числе
NNP	Proper noun, singular	Имя собственное в единственном числе
NNPS	Proper noun, plural	Имя собственное во множественном числе
PDT	Predeterminer	Предетерминатив
POS	Possessive ending	Притяжательное окончание
PRP	Personal pronoun	Личное местоимение
PRP\$	Possessive pronoun	Притяжательное местоимение
RB	Adverb	Наречие
RBR	Adverb, comparative	Наречие, сравнительное
RBS	Adverb, superlative	Наречие, превосходная степень
RP	Particle	Частица
SYM	Symbol	Символ
TO	“to”	Слово «В»
UH	Interjection	Междометие
VB	Verb, base form	Глагол в неопределенной форме
VBD	Verb, past tense	Глагол в прошедшем времени
VBG	Verb, gerund or present participle	Глагол, герундий или причастие настоящего времени
VCN	Verb, past participle	Глагол, причастие прошедшего времени
VBP	Verb, non-third-person singular present	Глагол настоящего времени, единственного числа, не третьего лица
VBZ	Verb, third person singular present	Глагол настоящего времени, единственного числа, третьего лица
WDT	wh-determiner	wh-определяющее слово
WP	Wh-pronoun	Wh-местоимение
WP\$	Possessive wh-pronoun	Притяжательное wh-местоимение
WRB	Wh-adverb	Wh-наречие

Помимо измерения параметров языка, с помощью NLTK можно определять значения слов на основе контекста и

собственных обширных словарей. В простейшем случае NLTK позволяет идентифицировать части речи:

```
>>> from nltk.book import *
>>> from nltk import word_tokenize
>>> text = word_tokenize('Strange women lying
in ponds distributing swords' \
'is no basis for a system of government.')
>>> from nltk import pos_tag
>>> pos_tag(text)
[('Strange', 'NNP'), ('women', 'NNS'),
('lying', 'VBG'), ('in', 'IN')
, ('ponds', 'NNS'), ('distributing', 'VBG'),
('swords', 'NNS'), ('is'
, 'VBZ'), ('no', 'DT'), ('basis', 'NN'),
('for', 'IN'), ('a', 'DT'),
('system', 'NN'), ('of', 'IN'), ('government',
'NN'), ('.', '.')]

```

Каждое слово выделено в *кортеж*, содержащий само слово, и тег, идентифицирующий данную часть речи (дополнительную информацию об этих тегах см. в предыдущей врезке). На первый взгляд такой поиск может показаться простым, однако в следующем примере видно, как сложно правильно выполнить эту задачу:

```
>>> text = word_tokenize('The dust was thick so
he had to dust')
>>> pos_tag(text)
[('The', 'DT'), ('dust', 'NN'), ('was', 'VBD'),
('thick', 'JJ'), ('so
', 'RB'), ('he', 'PRP'), ('had', 'VBD'), ('to',
'TO'), ('dust', 'VB')]

```

Обратите внимание: слово *dust* использовано в предложении дважды: один раз как существительное (*dust*), другой раз — как глагол (*to dust*)¹⁷. NLTK правильно распознает оба варианта применения, основываясь на контексте этих слов в предложении. NLTK идентифицирует части речи с помощью контекстно-свободной грамматики, существующей в английском языке. *Контекстно-свободные грамматики* — наборы правил, по которым можно определить, какие элементы могут следовать за теми или иными элементами в упорядоченных списках. В данном случае эти правила определяют, какие части речи могут идти за другими частями речи. Всякий раз, когда в тексте встречается неоднозначное слово, такое как *dust*, применяются правила контекстно-свободной грамматики и выбирается часть речи, соответствующая этим правилам.

Виды машинного обучения

NLTK позволяет создавать совершенно новые контекстно-свободные грамматики — например, при обучении иностранному языку. Если вручную разметить соответствующими тегами Penn Treebank большие разделы текста на этом языке, то их можно передать обратно в NLTK и обучить программу правильно размечать другой текст, с которым ей предстоит столкнуться. Такой тип обучения — необходимый элемент любого машинного обучения, о котором мы еще поговорим в главе 14, когда будем обучать веб-скраперы распознавать символы капчи.

Зачем нам знать, является ли слово глаголом или существительным в данном контексте? Вероятно, это имеет смысл в научно-исследовательской лаборатории по информатике, но как может помочь при веб-скрапинге?

Одна из распространенных задач веб-скрапинга связана с поиском. Например, при веб-скрапинге текста с сайта вы можете захотеть найти слово *google*, но только в значении глагола «гуглить», а не как имя собственное. Или же, возможно, вам нужны только упоминания компании Google и при их поиске вы не хотите полагаться на правильное употребление заглавных букв. В таких случаях чрезвычайно полезна функция `pos_tag`:

```
from nltk import word_tokenize, sent_tokenize,
pos_tag
sentences = sent_tokenize('Google is one of the
best companies in the world.'\
' I constantly google myself to see what I\'m
up to.')
nouns = ['NN', 'NNS', 'NNP', 'NNPS']

for sentence in sentences:
    if 'google' in sentence.lower():
        taggedWords =
pos_tag(word_tokenize(sentence))
        for word in taggedWords:
            if word[0].lower() == 'google'
and word[1] in nouns:
                print(sentence)
```

Данная программа выводит только те предложения, в которых содержится слово *google* (или *Google*) в значении существительного, а не глагола. Конечно, мы могли бы

выразиться точнее и потребовать, чтобы выводились только экземпляры Google, помеченные как *NNP* (имя собственное). Но даже NLTK время от времени допускает ошибки, поэтому иногда имеет смысл оставить немного места для маневра, в зависимости от конкретного приложения.

Функция `pos_tag` из библиотеки NLTK позволяет устранить большую часть неоднозначностей естественного языка. Если искать в тексте не просто конкретное слово или фразу, а с учетом тега, то можно значительно повысить точность и эффективность поиска веб-скрапера.

Дополнительные ресурсы

Обработка, анализ и понимание машиной естественного языка — одна из самых сложных задач информатики. На эту тему написано бесчисленное множество книг и исследовательских работ. Я надеюсь, что материалы этой главы вдохновят вас на размышления, выходящие за рамки обычного веб-скрапинга, или по крайней мере дадут некое первичное направление, с которого следует начинать проект, требующий анализа естественного языка.

Есть множество отличных ресурсов, посвященных начальной обработке естественного языка и Python Natural Language Toolkit. В частности, в книге *Natural Language Processing with Python* Стивена Берда (Steven Bird), Эвана Кляйна (Ewan Klein) и Эдварда Лопера (Edward Loper) (издательство O'Reilly) (<http://oreil.ly/1HYt3vV>) эта тема изложена комплексно, начиная с основ.

Кроме того, есть книга *Natural Language Annotations for Machine Learning* Джеймса Пустейовского (James Pustejovsky) и Эмбер Стаббс (Amber Stubbs) (издательство O'Reilly) (<http://oreil.ly/S3BudT>) — несколько более сложное

теоретическое руководство. Чтобы выполнить представленные там уроки, вам понадобятся знания Python; для тем, рассмотренных в этой книге, прекрасно подходит Python Natural Language Toolkit.

[12](#) Многие методики, описанные в этой главе, применимы ко всем или к большинству языков, однако сейчас мы сосредоточимся на обработке только одного естественного языка — английского. Например, такие инструменты, как Python Natural Language Toolkit, ориентированы именно на английский язык; 56 % содержимого Интернета все еще написано по-английски (по данным W3Techs (http://w3techs.com/technologies/overview/content_language/all), немецкий язык отстает от английского всего на 6 %). Но кто знает? В будущем английский язык почти наверняка потеряет главенствующую роль в большей части Интернета, и в ближайшие несколько лет эти методы могут претерпеть значительные изменения.

[13](#) Vinyals O. et al. A Picture Is Worth a Thousand (Coherent) Words: Building a Natural Description of Images (<http://bit.ly/1HEJ8kX>), Google Research Blog, November 17, 2014.

[14](#) Исключением является последнее слово в тексте, поскольку за ним не стоит следующее. В нашем примере текста последнее слово представляет собой точку (.), что удобно, так как это слово встречается в тексте еще 215 раз и поэтому не является тупиком. Однако в реальных реализациях генератора Маркова может потребоваться специальная обработка случая, когда достигнуто последнее слово в тексте.

[15](#) Стоит еще раз отметить, что библиотека NLTK предназначена для работы с текстами на английском языке. Если вы хотите использовать ее для токенизации предложений на русском языке, необходимо добавить параметр "language=russian" при вызове токенизатора `tokens = word_tokenize("Текст на русском", lanugage=" russian")`. Более подробно о способах использования NLTK можно почитать по адресу <https://python-school.ru/nlp-text-preprocessing/>. Конкретно для русского языка в большинстве случаев используют специализированные библиотеки, такие как Natasha, Yargi-parser и т.д. Больше информации можно найти по адресу <https://natasha.github.io/>. — *Примеч. науч. ред.*

[16](#) На русский язык эту фразу можно перевести как «Он был объективен в достижении своей цели — написать объективную философию, в первую очередь используя глаголы со словами в косвенном падеже», что не представляет проблемы для понимания. В качестве русского аналога можно привести такой пример: «Косил косою косою косою». — *Примеч. пер.*

[17](#) Аналогичная русская фраза, в которой алгоритму пришлось бы отличать глагол от существительного, могла бы звучать так: «Слой пыли был таким толстым, что, пока он ее вытирал, его попросили: "Не пыли!"». — *Примеч. пер.*

Глава 10. Сбор данных из форм и проверка авторизации

Один из первых вопросов, который возникает, стоит лишь немного выйти за рамки простейшего анализа веб-страниц, — «Как получить доступ к информации, требующей входа в систему?». Сеть все больше стремится к взаимодействию, движется в сторону социальных сетей и пользовательского контента. Формы и проверка авторизации являются неотъемлемой частью таких сайтов, избежать их практически невозможно. К счастью, с ними легко справиться.

До сих пор взаимодействие наших веб-скраперов с веб-серверами в большинстве примеров сводилось к получению информации с помощью GET-запроса HTTP. Эта глава посвящена методу POST, который передает информацию на веб-сервер для хранения и анализа.

Формы — это, в сущности, способ отправить веб-серверу от пользователя POST-запрос, который веб-сервер может понять и задействовать впоследствии. Если размещенные на сайте теги ссылок позволяют пользователям форматировать GET-запросы, то HTML-формы дают возможность форматировать POST-запросы. Всего пара строк кода — и нам будет под силу самостоятельно создавать такие запросы и отправлять их на сервер с помощью веб-скрапера.

Библиотека Requests

По веб-формам вполне можно перемещаться, используя только основные библиотеки Python, но если добавить немного синтаксического сахара, то жизнь станет заметно слаще. Когда нужно сделать нечто большее, чем просто отправить GET-

запрос с помощью `urllib`, имеет смысл выйти за пределы основных библиотек Python.

Библиотека `Requests` (<http://www.python-requests.org>) отлично справляется со сложными HTTP-запросами, cookie-файлами, заголовками и др. Кеннет Рейтц (Kenneth Reitz), автор библиотеки `Requests`, говорит об основных инструментах Python следующее: «Стандартный модуль `urllib2` обеспечивает большую часть функциональных возможностей HTML, но его API никуда не годится. Он был создан для другого времени и другого Интернета. Он требует слишком много усилий (даже переопределения метода) для выполнения самых простых задач.

Так не должно быть. По крайней мере, не в Python».

Как и любую библиотеку Python, библиотеку *Requests* можно установить с помощью любого стороннего менеджера библиотек Python, такого как `pip`, или же скачав и установив исходный файл (<https://github.com/kennethreitz/requests/tarball/master>).

Отправка простейшей формы

Большинство веб-форм состоят из нескольких HTML-полей, кнопки отправки и страницы обработки, на которой, собственно, и выполняется обработка формы. Многие HTML-поля обычно текстовые, однако встречаются поля для загрузки файлов и передачи других нетекстовых данных.

Наиболее популярные сайты блокируют доступ к формам авторизации в файлах `robots.txt` (законность веб-скрапинга таких форм обсуждается в главе 18), поэтому на **`pythonscrapping.com`** я из соображений безопасности разработала несколько типов форм аутентификации, которые позволят вам испытать возможности ваших веб-скраперов. Самая простая из

этих форм размещена на странице <http://pythonscraping.com/pages/files/form.html>.

Полный HTML-код формы выглядит следующим образом:

```
<form method="post" action="processing.php">
First      name:      <input      type="text"
name="firstname"><br>
Last name: <input type="text" name="lastname">
<br>
<input type="submit" value="Submit">
</form>
```

Здесь следует отметить пару моментов: во-первых, имена двух полей ввода — `firstname` и `lastname`. Это важно. Имена полей соответствуют именам параметров, которые будут переданы на сервер при отправке формы. Если вы хотите имитировать действие, которое выполняет форма, и с помощью POST-запроса передать ваши собственные данные, то должны проследить, чтобы имена этих параметров совпадали.

Второе, на что следует обратить внимание, — это то, что обработка формы выполняется на странице **processing.php** (ее полный путь — <http://pythonscraping.com/pages/files/processing.php>). Все POST-запросы к форме должны выполняться *именно на этой* странице, а не на той, где размещена сама форма. Запомните: назначение HTML-форм состоит лишь в том, чтобы помочь посетителям сайта представить свои запросы в правильном формате для отправки на страницу, на которой выполняется сама обработка. Если вас не интересует формат самого запроса, то не имеет значения, на какой странице находится форма.

Для отправки формы с помощью библиотеки *Requests* достаточно четырех строк кода, включая импорт и инструкцию

вывода результатов на экран (да, это действительно очень просто):

```
import requests

params = {'firstname': 'Ryan', 'lastname':
'Mitchell'}

r =
requests.post("http://pythonscraping.com/pages/
files/processing.php", data=params)
print(r.text)
```

После отправки формы скрипт должен вернуть следующий контент страницы:

Hello there, Ryan Mitchell!

Этот скрипт применим ко многим простым формам, встречающимся в Интернете. Например, форма подписки на новостную рассылку O'Reilly Media выглядит так:

```
<form
action="http://post.oreilly.com/client/o/oreilly/forms/
quicksignup.cgi"
id="example_form2" method="POST">
    <input name="client_token" type="hidden"
value="oreilly" />
    <input name="subscribe" type="hidden"
value="optin" />
    <input name="success_url" type="hidden"
value="http://oreilly.com/store/
newsletter-thankyou.html" />
```

```

        <input name="error_url" type="hidden"
value="http://oreilly.com/store/
        newsletter-signup-error.html"
/>
        <input name="topic_or_dod" type="hidden"
value="1" />
        <input name="source" type="hidden"
value="orm-home-t1-dotd" />
        <fieldset>
            <input class="email_address long"
maxlength="200" name=
                "email_addr" size="25"
type="text" value=
                "Enter your email here" />
            <button alt="Join" class="skinny"
name="submit" onclick=
                "return
addClickTracking('orm','ebook','rightrail','dod
'

                );"value="submit">Join</button>
        </fieldset>
</form>

```

На первый взгляд это, возможно, смотрится пугающе, но учтите, что в большинстве случаев (исключения из данного правила мы рассмотрим позже) мы ищем только два элемента:

- имя поля (или полей), данные которого (которых) мы хотим передать (в данном случае это имя `email_addr`);
- атрибут `action` самой формы, то есть страница, на которую отправляется форма (в данном случае это

<http://post.oreilly.com/client/o/oreilly/forms/quicksignup.cgi>).

Просто добавьте в код необходимую информацию и выполните его:

```
import requests
params = {'email_addr':
'ryan.e.mitchell@gmail.com'}
r =
requests.post('http://post.oreilly.com/client/o
/oreilly/forms/
quicksignup.cgi',
data=params)
print(r.text)
```

В данном случае возвращается ссылка на страницу с еще одной формой, которую нужно заполнить, прежде чем вы попадете в список рассылки O'Reilly, но и к данной форме можно применить ту же концепцию. Однако если вы захотите попробовать сделать это сами, то я прошу вас использовать свои новые возможности во благо, а не заваливать издателя спамом с недействительными регистрационными данными.

Переключатели, флажки и другие поля ввода

Очевидно, что не любая веб-форма представляет собой набор текстовых полей, под которыми стоит кнопка **Submit** (Отправить). В стандарт HTML входит множество разнообразных полей ввода данных для форм, таких как переключатели, флажки и поля выбора. В HTML5 добавились ползунки (поля ввода диапазона), поля для электронной почты, даты и др. Если добавить сюда произвольные поля, создаваемые с помощью JavaScript: палитры выбора цвета,

календари и все остальное, что еще придумают разработчики, то возможности форм становятся просто безграничными.

Но каким бы сложным ни казалось поле формы, вас должны интересовать только две вещи: имя элемента и его значение. Имя элемента легко определить, посмотрев в исходный код и найдя атрибут `name`. Значение иногда бывает найти сложнее, поскольку оно может заполняться JavaScript непосредственно перед отправкой формы. В качестве примера приведу столь экзотическое поле формы, как палитра выбора цвета, — ее значением, скорее всего, будет что-то наподобие `#F03030`.

Если вы не уверены в формате значения поля ввода, то можно воспользоваться различными инструментами для отслеживания запросов GET и POST в браузере, которые отправляются на сайты и с сайтов. Как уже упоминалось, лучший и, пожалуй, самый очевидный способ отслеживания GET-запросов — это посмотреть на URL сайта. Если он имеет вид:

```
http://domainname.com?thing1=foo&thing2=bar
```

то это значит, что он соответствует форме следующего вида:

```
<form method="GET" action="someProcessor.php">
<input type="someCrazyInputType" name="thing1"
value="foo" />
<input type="anotherCrazyInputType"
name="thing2" value="bar" />
<input type="submit" value="Submit" />
</form>
```

и объекту параметров Python:

```
{ 'thing1': 'foo', 'thing2': 'bar' }
```

Если вы столкнулись со сложной формой POST и хотите узнать точно, какие параметры браузер отправляет на сервер, то самый простой способ — просмотреть их с помощью инспектора браузера или встроенных в браузер инструментов разработчика (рис. 10.1).

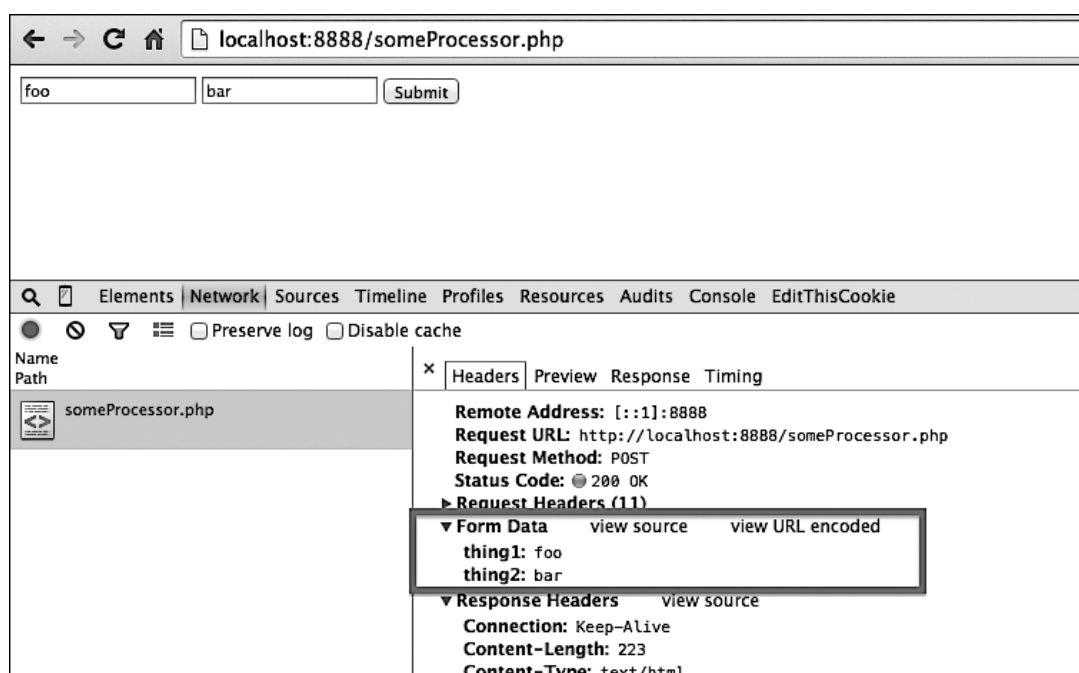


Рис. 10.1. В разделе Form Data, отмеченном рамкой, показаны POST-параметры thing1 и thing2 со значениями foo и bar

Чтобы открыть инструменты разработчика в Chrome, можно выбрать в меню пункты **View→Developer→Developer Tools** (Просмотр→Разработчик→Инструменты разработчика). Вы увидите список всех запросов, которые браузер создает при взаимодействии с текущим сайтом. Это может оказаться хорошим способом увидеть подробную структуру этих запросов.

Передача файлов и изображений

Несмотря на то что загрузка файлов широко распространена в Интернете, она не особенно часто применяется при веб-скрапинге. Однако вы, возможно, захотите написать тест для своего сайта, который включает в себя загрузку файла. В любом случае, полезно знать, как это делается.

Рассмотрим реальную форму загрузки файла, размещенную по адресу <http://www.pythonscraping.com/pages/files/form2.html>. Форма на этой странице имеет следующую разметку:

```
<form action="processing2.php" method="post"
enctype="multipart/form-data">
    Submit a jpg, png, or gif: <input
type="file" name="uploadFile"><br>
    <input type="submit" value="Upload File">
</form>
```

За исключением тега `<input>` с атрибутом `type`, имеющим значение `file`, эта форма мало чем отличается от форм с текстовыми полями, рассмотренных в предыдущих примерах. К счастью, способ обработки таких форм в Python-библиотеке `Requests` тоже похож:

```
import requests

files = {'uploadFile': open('files/python.png',
'rb')}

r = requests.post('http://pythonscraping.com/pages/
files/processing2.php',
                  files=files)

print(r.text)
```


Обратите внимание: значением, переданным в поле формы (с именем `uploadFile`), теперь является не обычная строка, а Python-объект `File`, возвращаемый функцией `open`. В данном примере мы отправляем файл с изображением, хранящийся на локальном компьютере, путь к которому относительно каталога, откуда запускается скрипт Python, выглядит так: `../files/Python-logo.png`.

Это действительно очень просто!

Обработка данных авторизации и параметров cookie

До сих пор мы по большей части обсуждали формы, позволяющие передавать информацию на сайт или просматривать необходимые сведения на странице, которая открывается сразу после отправки формы. Чем это отличается от формы авторизации, позволяющей сохранять статус авторизованного пользователя на протяжении всего посещения сайта?

Кто вошел в систему, а кто — нет, большинство современных сайтов отслеживают с помощью cookie-файлов. После аутентификации учетных данных пользователя сайт сохраняет их в cookie-файле браузера. В этом файле обычно содержатся сгенерированный сервером маркер, время ожидания и данные сопровождения. Затем сайт использует этот cookie-файл как своеобразное доказательство аутентификации, отображаемое на каждой странице, которую вы посетили во время своего пребывания на сайте. Cookie-файлы стали широко использоваться в середине 1990-х годов, а до тех пор безопасная аутентификация и отслеживание посетителей сайтов представляли огромную проблему.

Cookie-файлы — отличное решение для веб-разработчиков, однако для веб-скраперов могут быть проблемой. Вы можете

хоть весь день отправлять аутентификационные данные, но если не отслеживать cookie-файлы, возвращаемые формой аутентификации, то следующая же страница, на которую вы перейдете, будет работать так, словно вы вообще никогда не входили в систему.

Я создала простую форму аутентификации по адресу <http://pythonscraping.com/pages/cookies/login.html> (имя пользователя может быть любым, но паролем является слово password). Эта форма обрабатывается на странице <http://pythonscraping.com/pages/cookies/welcome.php>, где содержится ссылка на главную страницу <http://pythonscraping.com/pages/cookies/profile.php>.

Если вы попытаетесь открыть страницу **welcome.php** или страницу **profile.php** без аутентификации, то получите сообщение об ошибке и совет сначала ввести имя и пароль. На странице **profile.php** выполняется проверка cookie-файлов браузера, по результатам которой становится ясно, получил ли браузер cookie-файл на странице аутентификации.

Библиотека *Requests* позволяет легко отслеживать cookie-файлы:

```
import requests

params = {'username': 'Ryan', 'password':
'password'}

r =
requests.post('http://pythonscraping.com/pages/
cookies/welcome.php', params)
print('Cookie is set to:')
print(r.cookies.get_dict())
print('Going to profile page...')
```

```

r
requests.get('http://pythonscraping.com/pages/c
cookies/profile.php',
              cookies=r.cookies)
print(r.text)

```

Здесь мы отправляем параметры аутентификации на страницу **welcome.php**, которая обрабатывает данные формы входа в систему. Затем из результатов последнего запроса мы извлекаем данные cookie, выводим результат для проверки, а после передаем их на страницу **profile.php** в виде значения аргумента `cookies`.

Это хорошо работает в простых ситуациях, но как быть в случае более сложного сайта, который часто без предупреждения изменяет cookie-файлы, или если мы с самого начала не подумали о cookie-файлах? В таких ситуациях отлично работает функция `session` из библиотеки Requests:

```

import requests

session = requests.Session()

params = {'username': 'username', 'password':
          'password'}

s
session.post('http://pythonscraping.com/pages/c
cookies/welcome.php', params)
print('Cookie is set to:')
print(s.cookies.get_dict())
print('Going to profile page...')
s
session.get('http://pythonscraping.com/pages/co
okies/profile.php')

```

```
print(s.text)
```

В этом случае объект `session` (полученный путем вызова функции `request.Session()`) отслеживает информацию о сессии, такую как cookie-файлы, заголовки и даже информация о протоколах, которые могут использоваться поверх HTTP, например, HTTPAdapters.

Requests — просто фантастическая библиотека, по полноте поддерживаемых операций уступающая, пожалуй, только Selenium (см. главу 11), так что программисту нет нужды обдумывать их или писать код самостоятельно. Идея откинуться на спинку кресла и позволить библиотеке выполнить за вас всю работу может показаться заманчивой, однако при написании веб-скраперов крайне важно всегда знать, как выглядят cookie-файлы и чем именно они управляют. Это знание позволит вам сэкономить много часов мучительной отладки в попытках выяснить, почему сайт себя ведет так странно!

Базовая аутентификация доступа через HTTP. До появления cookie-файлов одним из популярных способов обработки аутентификационных данных была *базовая аутентификация доступа* через HTTP. Она до сих пор иногда еще встречается, особенно на сайтах с высоким уровнем безопасности или на корпоративных сайтах, а также в некоторых API. Я создала страницу с таким типом аутентификации по адресу <http://pythonscrapping.com/pages/auth/login.php> (рис. 10.2).

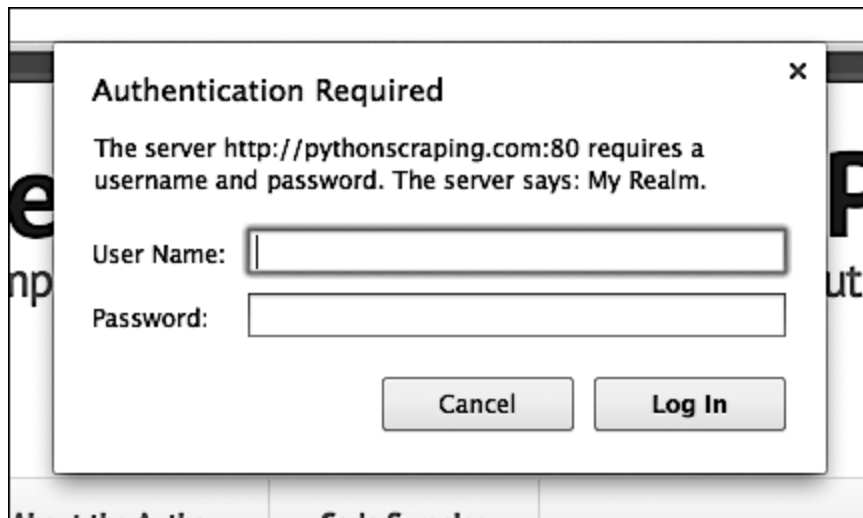


Рис. 10.2. Чтобы открыть страницу, защищенную базовой аутентификацией доступа, пользователь должен ввести свое имя и пароль

Как обычно в наших примерах, вы можете ввести любое имя пользователя и пароль **password**.

В пакет *Requests* входит модуль *auth*, специально созданный для обработки HTTP-аутентификации:

```
import requests
from requests.auth import AuthBase
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('ryan', 'password')
r = requests.post(url='http://pythonscraping.com/pages/auth/login.php',
                  auth=auth)
print(r.text)
```

Внешне это выглядит как обычный POST-запрос, однако объект *HTTPBasicAuth* передается в запросе в виде аргумента *auth*. Возвращаемый текст представляет собой страницу, защищенную именем пользователя и паролем (или страницу

Access Denied (Отказано в доступе), если запрос не будет выполнен).

Другие проблемы с формами

Веб-формы — излюбленная точка входа на сайт вредоносных ботов. Вы же не хотите, чтобы боты создавали пользовательские учетные записи, занимали драгоценное время сервера или отправляли спам-комментарии в блог! Именно поэтому на современных сайтах в HTML-формы часто встраиваются средства обеспечения безопасности, которые не всегда легко сразу заметить.



О том, как справляться с капчей, вы узнаете в главе 13, в которой рассматриваются обработка изображений и распознавание текста в Python.

Если вы столкнетесь с загадочной ошибкой или сервер отклонит отправку вашей формы по неизвестной причине, то ознакомьтесь с главой 14, в которой рассматриваются ловушки для хакеров (honeypots), скрытые поля и другие меры безопасности, которые сайты предпринимают для защиты форм.

Глава 11. Веб-скрапинг данных JavaScript

Скриптовые языки на стороне клиента — это языки, которые работают не на веб-сервере, а в самом браузере. Успешность клиентского языка зависит от способности браузера правильно интерпретировать и выполнять программы на этом языке. (Вот почему так легко отключить JavaScript в браузере.)

Клиентских языков гораздо меньше, чем серверных, — отчасти из-за того, что очень трудно заставить производителей всех браузеров выполнять требования стандарта. Для веб-скрапинга это хорошо: чем с меньшим количеством языков приходится иметь дело, тем лучше.

По большей части вы будете сталкиваться в Интернете только с двумя языками: ActionScript (который используется в приложениях Flash) и JavaScript. ActionScript сейчас встречается гораздо реже, чем десять лет назад, в основном для потоковой передачи мультимедийных файлов, в качестве платформы для онлайн-игр или отображения вводных страниц сайтов, владельцам которых еще не намекнули, что вводные страницы никто не любит. Как бы то ни было, поскольку большого спроса на веб-скрапинг Flash-страниц нет, в этой главе мы уделим основное внимание другому клиентскому языку, который на современных веб-страницах встречается повсеместно: JavaScript.

В настоящее время JavaScript — самый распространенный в Интернете и лучше всего поддерживаемый клиентский скриптовый язык. С его помощью можно собирать информацию для отслеживания пользователей, отправки форм без перезагрузки страницы, встраивания мультимедиа и запуска целых онлайн-игр. Даже обманчиво простые на вид страницы часто содержат несколько фрагментов JavaScript. Встроенные скрипты JavaScript можно найти в исходном коде страницы между тегами `script`:

```
<script>  
    alert("This creates a pop-up using  
    JavaScript");
```

</script>

Краткое введение в JavaScript

Очень полезно как минимум иметь представление о том, что происходит в коде, веб-скрапинг которого вы выполняете. Уже по одной лишь этой причине имеет смысл ближе познакомиться с JavaScript.

JavaScript — слабо типизированный язык, синтаксис которого часто сравнивают с C++ и Java. Некоторые элементы синтаксиса, такие как операторы, циклы и массивы, в этих языках, пожалуй, действительно похожи, однако слабая типизация и скриптовая природа JavaScript способны поставить в тупик некоторых программистов.

Например, следующий код рекурсивно вычисляет значения последовательности Фибоначчи и выводит их в консоль браузера, входящую в комплект инструментария разработчика:

```
<script>
function fibonacci(a, b){
    var nextNum = a + b;
    console.log(nextNum+" is in the Fibonacci
sequence");
    if(nextNum < 100){
        fibonacci(b, nextNum);
    }
}
fibonacci(1, 1);
</script>
```

Обратите внимание: все переменные идентифицируются путем добавления к ним ключевого слова `var`. Это как знак \$ в PHP или объявление типа (`int`, `String`, `List` и т.п.) в Java или C++. Python в этом смысле выделяется тем, что в нем нет столь явного объявления переменных.

В JavaScript также очень удобно то, что функции здесь тоже являются переменными:

```
<script>
var fibonacci = function() {
    var a = 1;
    var b = 1;
    return function () {
        var temp = b;
        b = a + b;
        a = temp;
        return b;
    }
}
var fibInstance = fibonacci();
console.log(fibInstance()+" is in the Fibonacci
sequence");
console.log(fibInstance()+" is in the Fibonacci
sequence");
console.log(fibInstance()+" is in the Fibonacci
sequence");
</script>
```

Поначалу это, возможно, выглядит жутковато, но все станет проще, если представить себе функции как лямбда-выражения (см. главу 2). Переменная `fibonacci` определена как функция. Ее значением является функция, которая выводит все числа последовательности Фибоначчи в порядке возрастания. При каждом вызове эта функция возвращает функцию вычисления очередного значения Фибоначчи, в результате выполнения которой увеличиваются значения переменных, используемых внутри функции `fibonacci`.

На первый взгляд это может показаться запутанным, однако некоторые задачи, такие как вычисление значений Фибоначчи, обычно решаются с помощью подобных схем. Передача функций в

качестве переменных также чрезвычайно полезна в качестве обратных вызовов при обработке действий пользователя; стоит освоить этот стиль программирования, если вы хотите научиться читать код JavaScript.

Популярные библиотеки JavaScript

Знать «чистый» JavaScript, безусловно, важно. Однако не стоит рассчитывать на многое в современном Интернете, если не использовать хотя бы одну из множества сторонних библиотек этого языка. При просмотре исходного кода страницы вам постоянно будет встречаться одна или несколько широко распространенных библиотек, которые мы рассмотрим далее.

Выполнение кода JavaScript с применением Python может потребовать очень больших затрат труда и вычислительных ресурсов, особенно в крупных масштабах. Знать, что к чему в JavaScript, и уметь анализировать код непосредственно (не прибегая к необходимости выполнять его для получения информации) может быть очень полезным навыком, который избавит вас от множества проблем.

jQuery

jQuery — чрезвычайно распространенная библиотека, которой пользуется 70 % самых популярных интернет-сайтов и примерно 30 % остальной части Интернета¹⁸. Понять, что на сайте используется jQuery, очень легко: где-то в его коде будет присутствовать импорт этой библиотеки:

```
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.
9.1/jquery.min.js">

</script>
```

Если вы нашли на сайте jQuery, то выполнять его веб-скрапинг следует очень осторожно: библиотека отлично умеет создавать

динамический HTML-контент, который образуется лишь после выполнения скрипта JavaScript. При традиционном веб-скрапинге вы получите только предварительно загруженную страницу в том виде, в каком она появляется прежде, чем JavaScript наполнит ее контентом (мы рассмотрим эту проблему веб-скрапинга более подробно в разделе «Ajax и динамический HTML» данной главы на с. 200).

Кроме того, на страницах с jQuery выше вероятность обнаружить анимацию, интерактивный контент и встроенные медиафайлы, что обычно усложняет веб-скрапинг.

Google Analytics

Библиотека *Google Analytics* используется примерно на 50 % сайтов^{[19](#)}, что делает ее, пожалуй, самой распространенной библиотекой JavaScript и самым популярным инструментом отслеживания пользователей в Интернете. Она используется также на сайтах <http://pythonscraping.com> и <http://www.oreilly.com/>.

Определить, применяется ли на веб-странице Google Analytics, легко. Внизу такой страницы будет размещаться примерно следующий код JavaScript (в данном случае код взят с сайта O'Reilly Media):

```
<!-- Google Analytics -->
<script type="text/javascript">

var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-4591498-1']);
_gaq.push(['_setDomainName', 'oreilly.com']);
_gaq.push(['_addIgnoredRef', 'oreilly.com']);
_gaq.push(['_setSiteSpeedSampleRate', 50]);
_gaq.push(['_trackPageview']);

(function() {
    var ga =
    document.createElement('script'); ga.type =
```

```
'text/javascript'; ga.async = true; ga.src =  
( 'https:' ==  
document.location.protocol ? 'https://ssl' :  
'http://www' ) +  
'./google-analytics.com/ga.js'; var s =  
document.getElementsByTagName( 'script' )[0];  
s.parentNode.insertBefore(ga, s); })(());  
  
</script>
```

Данный скрипт обрабатывает специальные cookie-файлы Google Analytics, с помощью которых отслеживаются посещения страницы пользователем. Иногда это представляет проблему для веб-скраперов, предназначенных для выполнения JavaScript и обработки cookie-файлов (например, для тех, которые используют описанную далее в этой главе библиотеку Selenium).

Если на сайте применяется Google Analytics или аналогичная система веб-аналитики и вы не хотите, чтобы сайт узнал, что на нем выполняется веб-краулинг или веб-скрапинг, то обязательно удалите все cookie-файлы, используемые для аналитики, или вообще все cookie-файлы.

Google Maps

Если вы провели хоть сколько-нибудь времени в Интернете, то почти наверняка встречали там встроенные в сайты карты *Google Maps*. API этой библиотеки позволяет очень легко встраивать в любой сайт карты с пользовательской информацией.

При веб-скрапинге любых данных о местоположении понимание принципов работы Google Maps позволяет легко получить координаты широты и долготы и даже адреса в удобном формате. Один из самых распространенных способов обозначить местоположение в Google Maps — использовать *маркер*, также известный как «булавка» (pin).

Маркеры можно вставить в любую карту Google с помощью следующего кода:

```
var marker = new google.maps.Marker({  
    position: new  
    google.maps.LatLng(-25.363882,131.044922),  
    map: map,  
    title: 'Some marker text'  
});
```

Python позволяет легко извлечь все координаты, которые находятся между `google.maps.LatLng(` и `)`, чтобы сформировать список широт и долгот.

Используя Google Reverse Geocoding API (<https://developers.google.com/maps/documentation/javascript/examples/geocoding-reverse>), можно преобразовать эти пары координат в адреса, представленные в формате, удобном для хранения и анализа.

Ajax и динамический HTML

До сих пор мы рассматривали только один способ коммуникации с веб-сервером — отправку ему того или иного HTTP-запроса при отображении новой страницы. Если вам когда-либо приходилось отправлять форму или извлекать информацию с веб-сервера без перезагрузки страницы, то вы, вероятно, имели дело с сайтом, на котором применялся Ajax.

Вопреки бытующему мнению, Ajax — это не язык программирования, а группа технологий, используемых для выполнения определенной задачи (очень похоже на веб-скрапинг, в сущности). Слово *Ajax* расшифровывается как *Asynchronous JavaScript and XML* (Асинхронный JavaScript и XML); эта технология служит для отправки и получения информации с веб-сервера без выполнения запроса на перезагрузку всей страницы.



Никогда не говорите: «Этот сайт будет написан на Ајах». Правильнее было бы сказать: «Данная форма будет использовать Ајах для связи с веб-сервером».

Подобно Ајах, *динамический HTML* (Dynamic HTML, DHTML) представляет собой набор технологий, применяемых для достижения общей цели. DHTML — это способность изменить HTML-код или язык CSS либо то и другое вместе так же, как клиентские сценарии изменяют HTML-элементы на странице. Например, кнопка появляется только после того, как пользователь переместит указатель мыши, при щелчке может измениться цвет фона, а по запросу Ајах — загрузиться новый блок контента.

Обратите внимание: слово «динамический» обычно ассоциируется с такими словами, как «перемещение» или «изменение», однако наличие интерактивных компонентов HTML, движущихся изображений или встроенных медиаэлементов не обязательно означает, что данная страница относится к DHTML, даже если выглядит динамично. Кроме того, за рядом скучнейших, совершенно неподвижных веб-страниц могут стоять DHTML-процессы, в ходе которых с помощью JavaScript изменяются HTML и CSS.

Выполнив веб-скрапинг многих сайтов, вы вскоре столкнетесь с такой ситуацией: контент, наблюдаемый в браузере, не соответствует контенту, который вы видите в исходном коде, полученном с данного сайта. Иногда, просматривая результаты работы веб-скрапера, можно мозги сломать, пытаясь выяснить, куда девалось все то, что вы видели на этой же самой странице в браузере.

Веб-страница также может представлять собой страницу загрузки, которая, по идее, перенаправляет пользователя на другую страницу, содержащую результаты. Однако вы заметите, что при таком перенаправлении URL страницы никогда не меняется.

Оба явления вызваны тем, что ваш веб-скрапер не способен выполнить скрипт JavaScript, от которого как раз и зависит все волшебство на этой странице. Без JavaScript HTML-код просто ничего не делает, и сайт может сильно отличаться от того, как выглядит в браузере, легко выполняемом JavaScript.

Есть несколько признаков того, что данная страница может использовать Ajax или DHTML для изменения или загрузки контента, но из подобных ситуаций есть только два выхода: выполнить веб-скрапинг содержимого непосредственно из JavaScript или задействовать пакеты Python, способные выполнять JavaScript, и провести веб-скрапинг сайта прямо в браузере.

Выполнение JavaScript в Python с помощью Selenium

Selenium (<http://www.seleniumhq.org/>) — это мощный инструмент веб-скрапинга, изначально разработанный для тестирования сайтов. В настоящее время он применяется и в тех случаях, когда требуется представить сайт в точности так, как он отображается в браузере. Selenium автоматизирует работу браузера, загружая сайт, получая необходимые данные и даже создавая снимки экрана или другие подтверждения того, что на сайте выполняются определенные действия.

У Selenium нет собственного браузера; его запуск требует интеграции со сторонними браузерами. Например, если запустить Selenium с Firefox, то, когда на экране откроется окно Firefox, следует перейти на нужный сайт и выполнить код. Кому-то это может показаться изящным, однако я предпочитаю, чтобы мои скрипты работали в фоновом режиме, для чего часто использую в Chrome *режим консоли*.

В *режиме консоли* браузер загружает сайты в память и выполняет JavaScript на странице без какой-либо графической визуализации сайта для пользователя. Сочетая Selenium с режимом консоли в Chrome, можно создавать чрезвычайно мощные веб-скраперы, способные легко обрабатывать cookie-файлы, JavaScript, заголовки и

все необходимое так, как если бы вы задействовали браузер с выводом страницы на экран.

Библиотеку Selenium можно скачать с ее сайта (<https://pypi.python.org/pypi/selenium>) или же установить ее из командной строки с помощью стороннего инсталлятора, такого как pip.

Веб-драйвер Chrome можно скачать с сайта ChromeDriver (<http://chromedriver.chromium.org/downloads>). ChromeDriver не является специализированной библиотекой Python — это независимое приложение, применяемое для управления Chrome, вследствие чего его нельзя установить с помощью pip. Чтобы использовать ChromeDriver, его необходимо скачать.

Ajax применяется для загрузки данных на многих страницах (в том числе, что характерно, в Google), однако я создала собственную страницу для запуска наших веб-скраперов по адресу <http://pythonscraping.com/pages/javascript/ajaxDemo.html>. На ней содержится некий текстовый фрагмент, жестко закодированный в HTML-коде страницы, который после двухсекундной задержки заменяется контентом, сгенерированным с помощью Ajax. Если бы мы решили выполнить веб-скрапинг этой страницы, используя традиционные методы, то получили бы только загрузочную страницу без нужных нам данных.

Библиотека Selenium — API, вызываемый для объекта WebDriver. Обратите внимание: это объект Python, представляющий или действующий как интерфейс предварительно загруженного приложения WebDriver. То и другое обозначается одним и тем же словом (и объект Python, и само приложение), однако концептуально важно различать их.

Объект WebDriver немного похож на браузер в том смысле, что способен загружать сайты, однако его также можно использовать как объект BeautifulSoup для поиска элементов страницы, взаимодействия с ними (отправки текста, нажатия кнопок и т.п.), а также для выполнения других действий, управляющих веб-скрапером.

Следующий код извлекает текст, стоящий «за стеной» Ajax на тестовой странице:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
import time

chrome_options = Options()
chrome_options.add_argument('--headless')
driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)
driver.get('http://pythonscraping.com/pages/javascript/ajaxDemo.html')
time.sleep(3)
print(driver.find_element_by_id('content').text)
driver.close()
```

Селекторы Selenium

В предыдущих главах мы выбирали элементы страницы с помощью селекторов BeautifulSoup, таких как `find` и `find_all`. В Selenium для поиска элементов DOM в WebDriver используется совершенно другой набор селекторов, хотя их имена довольно простые.

В этом примере мы использовали селектор `find_element_by_id`, хотя следующие селекторы также сработали бы:

```
driver.find_element_by_css_selector('#content')
driver.find_element_by_tag_name('div')
```

Разумеется, на тот случай, если нужно выбрать на странице несколько элементов, большинство этих селекторов позволяют

возвращать список элементов Python. Для этого в имени функций нужно заменить `element` на `elements` (то есть на множественное число):

```
driver.find_elements_by_css_selector('#content')
driver.find_elements_by_css_selector('div')
```

Если же вы все равно хотите использовать BeautifulSoup для анализа контента сайта, то можете применить функцию `WebDriver.page_source`. Она возвращает исходный код страницы (просматриваемый в момент вызова с помощью DOM) в виде строки:

```
pageSource = driver.page_source
bs = BeautifulSoup(pageSource, 'html.parser')
print(bs.find(id='content').get_text())
```

В результате получим новый Selenium WebDriver, использующий библиотеку Chrome, который дает WebDriver команду загрузить страницу, а затем приостанавливает выполнение на три секунды, после чего просматривает страницу и получает (как мы надеемся, загруженный к тому моменту) контент.

Создавая экземпляр нового Chrome WebDriver в Python, можно передать ему различные параметры через объект `Options`. В данном случае мы используем параметр `--headless`, чтобы WebDriver работал в фоновом режиме:

```
chrome_options = Options()
chrome_options.add_argument('--headless')
```

Кроме того, параметр `executable_path` должен указывать на местоположение скачанного приложения ChromeDriver:

```
driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)
```

Если все настроено правильно, то выполнение скрипта займет несколько секунд, а затем появится следующий текст:

```
Here is some important text you want to retrieve!  
A button to click!
```

Обратите внимание: хотя сама страница содержит HTML-кнопку, функция Selenium `.text` извлекает текстовое значение этой кнопки — так же, как и всего остального контента страницы.

Если вместо трехсекундной паузы заменить значение `time.sleep` на одну секунду, то возвращаемый текст изменится на исходный:

```
This is some content that will appear on the page  
while it's loading.  
You don't care about scraping this.
```

Это решение работает, однако оно несколько неэффективно, и его реализация в больших масштабах может вызвать проблемы. Время загрузки страниц непостоянно, оно зависит от нагрузки на сервер в любую конкретную миллисекунду. Скорость соединения также подвержена естественным изменениям. Загрузка страницы занимает чуть более двух секунд, но мы даем ей целых три, чтобы она наверняка успела загрузиться целиком. Более эффективное решение будет многократно проверять наличие определенного элемента, присущего полностью загруженной странице, и возвращать результат, только когда он есть.

В следующем коде показателем того, что страница полностью загружена, является наличие кнопки с идентификатором `loadedButton`:

```
page has been fully loaded:  
from selenium import webdriver  
from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import  
WebDriverWait
```

```

from selenium.webdriver.support import
expected_conditions as EC

chrome_options = Options()
chrome_options.add_argument('--headless')
driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)

driver.get('http://pythonscraping.com/pages/javascrip
t/ajaxDemo.html')
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_locat
ed((By.ID, 'loadedButton')))
finally:
    print(driver.find_element_by_id('content').text
)
    driver.close()

```

В этом скрипте есть несколько новых импортируемых модулей — в частности, `WebDriverWait` и `expected_conditions`. Их сочетание здесь нужно для формирования того, что в Selenium называется *неявным ожиданием*.

Неявное ожидание отличается от явного тем, что для продолжения скрипта ожидается определенное состояние DOM, в то время как при явном ожидании есть жестко заданное время — как в предыдущем примере, где длительность ожидания составляла три секунды. При неявном ожидании иницилирующее состояние DOM определяется с помощью ожидаемого условия `expected_condition` (обратите внимание: здесь импортируемый модуль обозначен как `EC` — для краткости, в соответствии с общим соглашением). Ожидаемые условия в библиотеке Selenium могут быть самыми разными, включая следующие:

- появление всплывающего окна с предупреждением;
- переход элемента (например, текстового поля) в состояние «выбран»;
- изменение заголовка страницы, отображение текста на странице или в определенном элементе;
- появление или исчезновение элемента DOM.

Большинство из этих ожидаемых условий требуют сначала указать элемент, за которым нужно проследить. Элементы указываются с помощью локаторов. Обратите внимание: локаторы — не то же самое, что селекторы (подробнее о селекторах см. выше, во врезке «Селекторы Selenium» на с. 203). *Локатор* — это абстрактный язык запросов, использующий объект `By`. Локаторы можно применять различными способами, в том числе для создания селекторов.

В следующем коде локатор используется для поиска элементов с идентификатором `loadedButton`:

```
EC.presence_of_element_located((By.ID,
    'loadedButton'))
```

Локаторы также можно использовать для создания селекторов с помощью функции `WebDriver.find_element`:

```
print(driver.find_element(By.ID, 'content').text)
```

Функционально это, конечно же, эквивалентно следующей строке из нашего примера:

```
print(driver.find_element_by_id('content').text)
```

Если можно не использовать локатор, то не делайте этого; так вы сэкономите один оператор импорта. Тем не менее этот удобный инструмент применим во множестве областей и обладает большой гибкостью.

Существуют следующие стратегии выбора локатора с объектом `By`:

- `ID` — используется в нашем примере; находит элементы по их HTML-атрибуту `id`;
- `CLASS_NAME` — служит для поиска элементов по их HTML-атрибуту `class`. Почему данная функция называется `CLASS_NAME`, а не просто `CLASS`? Дело в том, что использование записи `object.CLASS` создало бы проблемы для Java-библиотеки `Selenium`, где `.class` является зарезервированным методом. Чтобы сохранить синтаксис `Selenium` в разных языках, вместо этого применяется `CLASS_NAME`;
- `CSS_SELECTOR` — находит элементы по классу, идентификатору или имени тега, используя соглашения `#idName`, `.className` и `tagName`;
- `LINK_TEXT` — находит HTML-теги `<a>` по тексту, который в них содержится. Например, чтобы выбрать ссылку с надписью `Next`, нужно использовать запись `(By.LINK_TEXT, 'Next')`;
- `PARTIAL_LINK_TEXT` — работает аналогично `LINK_TEXT`, но для части строки;
- `NAME` — находит HTML-теги по их атрибуту `name`, что удобно для HTML-форм;
- `TAG_NAME` — находит HTML-теги по имени тега;
- `XPATH` — использует для выбора элементов выражение `XPath` (синтаксис которого описан в следующей врезке).

Синтаксис XPath

XPath (сокращение от XML Path) — язык запросов, используемый для навигации по XML-документам и выбора их частей. Основанный W3C в 1999 году, этот язык иногда применяется для работы с XML-документами в таких языках программирования, как Python, Java и C#.

BeautifulSoup не поддерживает XPath, в отличие от многих других библиотек, описанных в этой книге, например Scrapy и Selenium. XPath часто можно использовать аналогично селекторам CSS (наподобие `mytag#idname`), хотя он предназначен для работы с любыми XML-документами, а не только с их более частным случаем — документами HTML.

У синтаксиса XPath есть четыре основные концепции.

Узлы делятся на корневые и некорневые:

селектор `/div` выбирает узел `div`, только если тот находится в корне документа;

селектор `//div` выбирает все узлы `div`, независимо от того, в каком месте документа они находятся.

Выбор атрибута:

селектор `//@href` выбирает все узлы с атрибутом `href`;

селектор `//a[@href='http://google.com']` выбирает в документе все ссылки, которые указывают на Google.

Выбор узлов по положению:

селектор `//a[3]` выбирает третью ссылку в документе;

селектор `//table[last()]` выбирает последнюю таблицу в документе;

селектор `//a[position()<3]` выбирает первые две ссылки в документе.

Звездочка (*) соответствует любому набору символов или узлов и может использоваться в различных ситуациях:

селектор `//table/tr/*` выбирает всех потомков тега `tr` во всех таблицах (это удобно при выборе ячеек с помощью тегов `th` и `td`);

селектор `//div[@*]` выбирает все теги `div` с любыми атрибутами.

В синтаксисе XPath также есть много дополнительных свойств. За последние годы он превратился в довольно сложный язык запросов с булевой логикой, функциями (такими как `position()`) и множеством операторов, которые не рассматриваются в этой книге.

Если у вас возникнет проблема с выбором элементов в HTML или XML, которую не удастся решить с помощью представленных здесь функций, то обратитесь к странице Microsoft с описанием синтаксиса XPath (<https://msdn.microsoft.com/en-us/enus/library/ms256471>).

Дополнительные веб-драйверы Selenium

В предыдущем разделе мы задействовали Selenium в сочетании с Chrome WebDriver (ChromeDriver). В большинстве случаев нет особой причины открывать браузер и выполнять в нем веб-скрапинг, удобнее запустить его в режиме консоли. Однако работа без режима консоли и/или применение разных драйверов браузера могут оказаться полезными по следующим причинам.

- Исправление неполадок. Если код, работая в режиме консоли, дает сбой, то выполнить диагностику иногда бывает трудно, поскольку

вы не видите исходную страницу.

- Для диагностики проблем вы также можете приостановить выполнение кода и что-то сделать на веб-странице либо использовать инспектор кода во время работы веб-скрапера.
- Результаты тестов могут зависеть от конкретного браузера. Сбой в одном браузере при нормальной работе в другом может указывать на проблему, свойственную конкретному браузеру.

Сегодня в создании и обслуживании веб-драйверов Selenium для всех крупнейших браузеров принимает участие множество групп разработчиков, как официальных, так и неофициальных. Группа Selenium курирует коллекцию этих веб-драйверов (<http://www.seleniumhq.org/download/>), чтобы обеспечить удобство их использования.

```
firefox_driver = webdriver.Firefox('<path to  
Firefox webdriver>')  
safari_driver = webdriver.Safari('<path to Safari  
webdriver>')  
ie_driver = webdriver.Ie('<path to Internet  
Explorer webdriver>')
```

Обработка перенаправлений

Перенаправления на стороне клиента — это переходы на другие страницы, которые выполняются в вашем браузере с помощью JavaScript, а не те, что выполняются на сервере перед отправкой контента страницы. Иногда бывает сложно определить разницу между тем и другим, просто просматривая страницу в браузере. Перенаправление может произойти настолько быстро, что вы не заметите задержки при загрузке и предположите, что перенаправление на стороне клиента на самом деле является перенаправлением на стороне сервера.

Однако при веб-скрапинге разница очевидна. Перенаправление на стороне сервера, в зависимости от способа обработки, легко отслеживается Python-библиотекой `urllib` без помощи Selenium (подробнее это описано в главе 3). Перенаправления на стороне клиента вообще не будут обрабатываться, если в веб-скрапере не использовать нечто способное выполнять код JavaScript.

Selenium обрабатывает перенаправления JavaScript идентично другим скриптам JavaScript. Однако основная проблема с такими перенаправлениями состоит в том, когда следует остановить выполнение кода на странице — другими словами, как определить, что перенаправление завершено. На демонстрационной странице по адресу <http://pythonscraping.com/pages/javascript/redirectDemo1.html> приведен пример перенаправления этого типа с двухсекундной паузой.

Существует разумный способ обнаружить это перенаправление, «отследив» некий DOM-элемент в начале загрузки страницы, а затем многократно проверяя данный элемент, пока Selenium не выдаст исключение `StaleElementReferenceException`, говорящее о том, что данный элемент больше не привязан к DOM страницы и, следовательно, перенаправление завершилось:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.remote.webelement import WebElement
from selenium.common.exceptions import StaleElementReferenceException
import time

def waitForLoad(driver):
    elem = driver.find_element_by_tag_name('html')
    count = 0
    while True:
        count += 1
```

```

        if count > 20:
            print('Timing out after 10 seconds and
returning')
            return
        time.sleep(.5)
    try:
        elem ==
driver.find_element_by_tag_name('html')
    except StaleElementReferenceException:
        return
chrome_options = Options()
chrome_options.add_argument('--headless')
driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)
driver.get('http://pythonscraping.com/pages/javascrip
t/redirectDemo1.html')
waitForLoad(driver)
print(driver.page_source)
driver.close()

```

Данный скрипт проверяет страницу каждые полсекунды с предварительной задержкой десять секунд, хотя при необходимости время проверки и задержки легко изменить.

Вместо этого можно написать аналогичный цикл, проверяющий текущий URL страницы, пока он не изменится или пока не совпадет с искомым URL.

Ожидание появления и исчезновения элементов — обычная задача для Selenium. Для этого можно задействовать ту же функцию `WebDriverWait`, которую мы использовали в предыдущем примере с загрузкой кнопки. В следующем коде мы для выполнения той же задачи задаем время ожидания 15 секунд и определяем селектор XPath, который ищет контент тега `body` страницы:

```

from selenium.webdriver.common.by import By

```

```

from selenium.webdriver.support.ui import
WebDriverWait
from selenium.webdriver.chrome.options import
Options
from selenium.webdriver.support import
expected_conditions as EC
from selenium.common.exceptions import
TimeoutException

chrome_options = Options()
chrome_options.add_argument('--headless')
driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)

driver.get('http://pythonscraping.com/pages/javascrip
t/redirectDemo1.html')
try:
    bodyElement = WebDriverWait(driver,
15).until(EC.presence_of_element_
located(
    (By.XPATH, '//body[contains(text(),
    'This is the page you are looking
for!)]'))))
    print(bodyElement.text)
except TimeoutException:
    print('Did not find the element')

```

Последнее замечание о JavaScript

JavaScript используется на большинстве современных сайтов^{[20](#)}. На наше счастье, такое его применение по большей части не влияет на веб-скрапинг страницы. JavaScript может ограничиваться, например, активацией механизмов контроля, управлением небольшим разделом сайта или выпадающим меню. В тех случаях, когда это

влияет на веб-скрапинг сайта, код JavaScript легко выполняется с помощью таких инструментов, как Selenium, позволяющих создать простую HTML-страницу, которую вы уже научились читать в первой части этой книги.

Запомните: если на сайте задействован JavaScript, то это еще не значит, что все традиционные инструменты веб-скрапинга можно отбросить. Ведь главная цель JavaScript, в конце концов, формирование кода HTML и CSS для отображения в браузере или динамического взаимодействия с сервером с помощью HTTP-запросов и ответов на них. При использовании Selenium код HTML и CSS на странице можно прочитать и проанализировать так же, как любой другой код сайта, а HTTP-запросы и ответы веб-скрапер может отправить и обработать с помощью методов, описанных в предыдущих главах, даже без применения Selenium.

Кроме того, JavaScript может даже оказаться весьма уместным для веб-скрапинга, поскольку его применение в качестве «системы управления контентом на стороне браузера» может служить полезным API для связи с внешним миром, позволяя получать данные более непосредственно. Подробнее об этом вы узнаете в главе 12.

Если у вас все же возникнут проблемы с особенно запутанным JavaScript-кодом, то в главе 14 вы найдете информацию о Selenium и о непосредственном взаимодействии с динамическими сайтами, включая интерфейсы типа drag-and-drop.

¹⁸ Более подробную статистику см. в статье The State of jQuery 2014 (<https://blog.jquery.com/2014/01/13/the-state-of-jquery-2014/>), опубликованной в блоге Дэйва Метвина (Dave Methvin) 13 января 2014 года.

¹⁹ W3Techs, Usage Statistics and Market Share of Google Analytics for Websites (<http://w3techs.com/technologies/details/ta-googleanalytics/all/all>).

²⁰ W3Techs, Usage of JavaScript for Websites (<http://w3techs.com/technologies/details/cp-javascript/all/all>).

Глава 12. Веб-краулинг с помощью API

JavaScript традиционно принято считать вселенским проклятием веб-краулеров. Давным-давно были времена, когда вы могли быть уверены, что запрос, отправленный вами на веб-сервер, получит те же данные, которые пользователь увидит в своем браузере, сделав тот же запрос.

По мере распространения методов генерации и загрузки контента с помощью JavaScript и Ajax описанная выше ситуация становится все менее привычной. В главе 11 мы рассмотрели один из способов решения указанной проблемы: использование Selenium для автоматизации браузера и извлечения данных. Это легко сделать. Это работает почти всегда.

Проблема в том, что, когда у вас в руках есть столь мощный и эффективный «молоток», как Selenium, каждая задача веб-скрапинга начинает походить на гвоздь.

В текущей главе вы узнаете о том, как, минуя весь этот JavaScript (не выполняя и даже не загружая его!), получить прямой доступ к источнику данных — к интерфейсам API, которые генерируют эти данные.

Краткое введение в API

Существует бесчисленное количество книг, докладов и руководств о нюансах API REST, GraphQL²¹, JSON и XML, однако все они основаны на одной простой концепции. API определяет стандартизованный синтаксис, который позволяет одной программе взаимодействовать с другой, даже если они написаны на разных языках или имеют разную структуру.

Данный раздел посвящен веб-API (в особенности позволяющим веб-серверу взаимодействовать с браузером), и здесь мы будем понимать под API именно этот тип интерфейсов. Но вы можете учесть, что в других контекстах API также является обобщенным термином, который может обозначать, например, интерфейс, позволяющий программе на Java взаимодействовать с программой на Python, работающей на том же компьютере. API не всегда означает «интерфейс через Интернет» и не обязательно должен включать в себя какие-либо веб-технологии.

Веб-API чаще всего используются разработчиками для взаимодействия с широко разрекламированными и хорошо документированными открытыми сервисами. Например, американский кабельный спортивный телевизионный канал ESPN предоставляет API (<http://www.espn.com/apis/devcenter/docs/>) для получения информации о спортсменах, счетах в играх и др. У Google в разделе для разработчиков (<https://console.developers.google.com>) есть десятки API для языковых переводов, аналитики и геолокации.

В документации всех этих API обычно описываются маршруты или *конечные точки* в виде URL, которые можно запрашивать, с изменяемыми параметрами, либо входящими в состав этих URL, либо выступающими в роли GET-параметров.

Например, в следующем URL `pathparam` является параметром пути:

```
http://example.com/the-api-route/pathparam
```

А здесь `pathparam` является значением параметра `param1`:

```
http://example.com/the-api-route?  
param1=pathparam
```

Оба метода передачи данных в API используются достаточно широко, хотя, как и многие другие аспекты информатики, являются предметом жарких философских дискуссий о том, когда и где переменные следует передавать через путь, а когда — через параметры.

Ответ на API-запрос обычно возвращается в формате JSON или XML. В настоящее время JSON гораздо популярнее, чем XML, но последний иногда тоже встречается. Многие API позволяют выбирать тип ответа, обычно с помощью еще одного параметра, определяющего, какой тип ответа вы хотите получить.

Вот пример ответа на API-запрос в формате JSON:

```
{"user":{"id": 123, "name": "Ryan Mitchell",  
"city": "Boston"}}
```

А вот ответ на API-запрос в формате XML:

```
<user><id>123</id><name>Ryan Mitchell</name>  
<city>Boston</city></user>
```

Сайт **ip-api.com** (<http://ip-api.com/>) имеет понятный и удобный API, который преобразует IP-адреса в реальные физические адреса. Вы можете попробовать выполнить простой запрос API, введя в браузере следующее²²:

```
http://ip-api.com/json/50.78.253.58
```

В результате вы получите примерно такой ответ:

```
{"ip":"50.78.253.58","country_code":"US","count  
ry_name":"United States",  
"region_code":"MA","region_name":"Massachuset  
ts","city":"Boston",
```



```
"zip_code":"02116","time_zone":"America/New_York",  
"latitude":42.3496,  
"longitude":-71.0746,"metro_code":506}
```

Обратите внимание: в запросе есть параметр пути `json`. Чтобы получить ответ в формате XML или CSV, нужно заменить его на соответствующий формат:

```
http://ip-api.com/xml/50.78.253.58
```

```
http://ip-api.com/csv/50.78.253.58
```

API и HTTP-методы

В предыдущем разделе мы рассмотрели API, отправляющие на сервер GET-запрос для получения информации. Существует четыре основных способа (или метода) запроса информации с веб-сервера через HTTP:

- GET;
- POST;
- PUT;
- DELETE.

Технически типов запросов больше четырех (например, еще есть HEAD, OPTIONS и CONNECT), но они редко используются в API и маловероятно, что когда-либо встретятся вам. Подавляющее большинство API ограничиваются этими четырьмя методами, а иногда даже какой-то их частью. Постоянно встречаются API, которые используют только GET или только GET и POST.

GET — тот запрос, который вы используете, когда посещаете сайт, введя его адрес в адресной строке браузера. Обращаясь по адресу **http://ip-api.com/json/50.78.253.58**, вы применяете именно метод GET. Такой запрос можно представить как команду: «Эй, веб-сервер, будь добр, выдай мне эту информацию».

Запрос GET по определению не вносит изменений в содержимое базы данных сервера. Ничего не сохраняется и ничего не изменяется. Информация только считывается.

POST — запрос, который используется при заполнении формы или отправке информации, предположительно предназначенной для обработки серверным скриптом. Каждый раз, авторизуясь на сайте, вы делаете POST-запрос, передавая имя пользователя и (как мы надеемся) зашифрованный пароль. Делая POST-запрос через API, вы говорите серверу: «Будь любезен, сохрани эту информацию в базе данных».

Запрос PUT при взаимодействии с сайтами используется реже, но время от времени встречается в API. Этот запрос применяется для изменения объекта или информации. Например, в API можно задействовать запрос POST для создания пользователя и запрос PUT для изменения его адреса электронной почты²³.

Запросы DELETE, как нетрудно догадаться, служит для удаления объекта. Например, если отправить запрос DELETE по адресу **http://myapi.com/user/23**, то будет удален пользователь с идентификатором 23. Методы DELETE нечасто встречаются в открытых API, поскольку те в основном создаются для распространения информации или чтобы позволить пользователям создавать или публиковать информацию, но не удалять ее из баз данных.

В отличие от GET запросы POST, PUT и DELETE позволяют передавать информацию в теле запроса, в дополнение к URL или маршруту, с которого запрашиваются данные.

Как и ответ, получаемый от веб-сервера, эти данные в теле запроса обычно представляются в формате JSON или реже в формате XML. Конкретный формат данных определяется синтаксисом API. Например, при использовании API, который добавляет комментарии к сообщениям в блоге, можно создать следующий PUT-запрос:

```
http://example.com/comments?post=123
```

с таким телом запроса:

```
{"title": "Great post about APIs!", "body":  
"Very informative. Really helped me out with a  
tricky technical challenge I was facing. Thanks  
for taking the time to write such a detailed  
blog post about PUT requests!", "author":  
{"name": "Ryan Mitchell", "website":  
"http://pythonscraping.com", "company":  
"O'Reilly Media"}}
```

Обратите внимание: идентификатор сообщения в блоге (123) передается в качестве параметра в URL, а контент создаваемого нами комментария — в теле запроса. Параметры и данные могут передаваться и в параметре, и в теле запроса. Какие параметры обязательны и где передаются — опять-таки определяется синтаксисом API.

Подробнее об ответах на API-запросы

Как мы видели в примере с сайтом **ip-api.com** в начале данной главы, важной особенностью API является то, что эти

интерфейсы возвращают хорошо отформатированные ответы. Наиболее распространенные форматы ответов —XML (eXtensible Markup Language — расширяемый язык разметки) и JSON (JavaScript Object Notation — нотация объектов JavaScript).

В последние годы JSON стал намного популярнее, чем XML, по нескольким основным причинам. Во-первых, файлы JSON обычно меньше, чем хорошо проработанные файлы XML. Сравните, например, следующие данные в формате XML, занимающие 98 символов:

```
<user><firstname>Ryan</firstname>  
<lastname>Mitchell</lastname>  
<username>Kludgist</username></user>
```

А теперь посмотрите на те же данные в формате JSON:

```
{"user":  
{"firstname":"Ryan","lastname":"Mitchell","user  
name":"Kludgist"}}
```

Это всего 73 символа, на целых 36 % меньше, чем те же данные в формате XML.

Конечно, вероятен аргумент, что XML можно отформатировать так:

```
<user   firstname="ryan"   lastname="mitchell"  
username="Kludgist"></user>
```

Но это не рекомендуется, поскольку такое представление не поддерживает глубокое вложение данных. И все равно запись занимает 71 символ — примерно столько же, сколько эквивалентный JSON.

Другая причина, по которой JSON так быстро становится более популярным, чем XML, связана с изменением веб-

технологий. Раньше получателями API были по большей части серверные скрипты на PHP или .NET. Сейчас вполне может оказаться, что получать и отправлять вызовы API будет фреймворк наподобие Angular или Backbone. Серверным технологиям до определенной степени безразлично, в какой форме к ним поступают данные. Однако библиотекам JavaScript, таким как Backbone, проще обрабатывать JSON.

Принято считать, что API возвращают ответ либо в формате XML, либо в формате JSON, однако возможен любой другой вариант. Тип ответа API ограничен только воображением программиста, создавшего этот интерфейс. Еще один типичный формат ответа — CSV (как видно из примера с **ip-api.com**). Отдельные API даже позволяют создавать файлы. Можно отправить на сервер запрос, по которому будет сгенерировано изображение с наложенным на него заданным текстом, или же запросить определенный файл XLSX или PDF.

Некоторые API вообще не возвращают ответа. Например, если отправить на сервер запрос для создания комментария к записи в блоге, то он может вернуть только HTTP-код ответа 200, что означает: «Я опубликовал комментарий; все в порядке!» Другие запросы могут возвращать минимальный ответ наподобие такого:

```
{"success": true}
```

В случае ошибки вы можете получить такой ответ:

```
{"error": {"message": "Something super bad  
happened"}}
```

Или же, если API не очень хорошо сконфигурирован, вы можете получить не поддающуюся анализу трассировку стека или некий текст на английском. Отправляя запрос к API, как правило, имеет смысл сначала убедиться, что получаемый

ответ действительно имеет формат JSON (или XML, или CSV, или любой другой формат, который вы ожидаете получить).

Синтаксический анализ JSON

В данной главе мы рассмотрели различные типы API и их функционирование, а также примеры JSON-ответов, полученных от этих API. Теперь посмотрим, как можно анализировать и использовать эту информацию.

В начале главы мы рассмотрели пример API **ip-api.com**, который преобразует IP-адреса в физические:

```
http://ip-api.com/json/50.78.253.58
```

Мы можем взять результат этого запроса и использовать функции Python для синтаксического анализа JSON, чтобы его декодировать:

```
import json
from urllib.request import urlopen

def getCountry(ipAddress):
    response = urlopen('http://ip-
api.com/json/'+ipAddress).read()
    .decode('utf-8')
    responseJson = json.loads(response)
    return responseJson.get('country_code')

print(getCountry('50.78.253.58'))
```

Эта программа выводит код страны для IP-адреса 50.78.253.58.

Здесь используется библиотека синтаксического анализа JSON, которая является частью базовой библиотеки Python. Чтобы ее подключить, нужно всего лишь поставить вверху строку `import json`! В отличие от многих других языков, способных преобразовать строку JSON в специальный объект JSON или узел JSON, в Python задействован более гибкий подход: объекты JSON превращаются в словари, массивы JSON — в списки, строки JSON — в строки и т.д. Таким образом, получить доступ и манипулировать значениями, хранящимися в JSON, чрезвычайно легко.

Ниже представлена краткая демонстрация того, как Python-библиотека JSON обрабатывает значения, которые встречаются в строках JSON:

```
import json

jsonString = '{"arrayOfNums": [{"number": 0}, {"number": 1}, {"number": 2}], "arrayOfFruits": [{"fruit": "apple"}, {"fruit": "banana"}, {"fruit": "pear"}]}'
jsonObj = json.loads(jsonString)

print(jsonObj.get('arrayOfNums'))
print(jsonObj.get('arrayOfNums')[1])
print(jsonObj.get('arrayOfNums')[1].get('number') +
      jsonObj.get('arrayOfNums')[2].get('number'))
print(jsonObj.get('arrayOfFruits')[2].get('fruit'))
```

Результат выглядит так:

```
[{'number': 0}, {'number': 1}, {'number': 2}]  
{'number': 1}  
3  
pear
```

Первая строка — список словарных объектов, вторая — словарный объект, третья — целое число (сумма целых чисел, доступных в словарях), а четвертая — просто строка.

Недокументированные API

До сих пор в данной главе мы обсуждали только документированные API. Их разработчики предполагают, что эти API будут открытыми, публикуют информацию о них и рассчитывают на их использование другими разработчиками. Однако у подавляющего большинства API вообще нет никакой открытой документации.

Но зачем же создавать API без открытой документации? Как упоминалось в начале этой главы, все дело в JavaScript.

Традиционно веб-серверы для динамических сайтов при каждом запросе страницы пользователем выполняли следующие задачи:

- обрабатывали GET-запросы от пользователей, запрашивающих страницу сайта;
- получали из базы данные, которые должны появляться на этой странице;
- вставляли эти данные в HTML-шаблон страницы;

- отправляли этот отформатированный HTML-код пользователю.

По мере распространения фреймворков JavaScript многие из задач по созданию HTML-кода, выполняемых сервером, были перенесены в браузер. Сервер может отправить в браузер пользователя жестко закодированный HTML-шаблон, но для загрузки контента и размещения его в правильных местах этого шаблона будут выполняться отдельные запросы Ajax. Все это станет происходить на стороне браузера/клиента.

Поначалу описанная ситуация представляла проблему для веб-скраперов, привыкших делать запрос на HTML-страницу и возвращать именно ее со всем уже имеющимся на ней контентом. Вместо этого теперь веб-скраперы получают HTML-шаблон без какого-либо наполнения.

Для решения данной проблемы был создан Selenium. Теперь веб-скрапер может играть роль браузера для программиста, запрашивая HTML-шаблон, выполняя любой JavaScript, загружая все данные на их место и только *потом* производя веб-скрапинг этих данных. Поскольку весь HTML-код был загружен, задача, по сути, сводится к уже решенной ранее: к синтаксическому анализу и форматированию существующего HTML-кода.

Однако, поскольку вся система управления контентом (которая раньше размещалась исключительно на веб-сервере), по существу, переместилась в клиентский браузер, содержимое даже самых простых сайтов может раздуться до нескольких мегабайтов и дюжины HTTP-запросов.

Кроме того, в случае применения Selenium загружаются все «дополнения», которые не обязательно нужны пользователю: вызовы программ отслеживания, загрузка рекламы на боковых панелях, вызовы программ отслеживания для нее.

Изображения, CSS, внешние шрифты — все их необходимо загрузить. Это может показаться отличным решением при использовании браузера для просмотра веб-страниц, однако если вы пишете веб-скрапер, который должен быстро перемещаться по страницам, собирать конкретные данные и создавать как можно меньше нагрузки на веб-сервер, то может оказаться, что вы загружаете данных в 100 раз больше, чем требуется.

Но у всех этих JavaScript-, Ajax- и веб-модернизаций есть и преимущество: поскольку серверы больше не преобразуют данные в формат HTML, они часто играют роль тонкой оболочки вокруг базы данных. Она просто извлекает данные из базы и передает их на страницу через API.

Разумеется, эти API не предназначены для использования кем-то или чем-то, кроме самой веб-страницы, и, как следствие, разработчики оставляют их без документации и предполагают (или надеются), что никто не заметит этих API. Но они существуют.

Например, сайт New York Times (<http://nytimes.com>) загружает все результаты поиска через JSON. Если вы пройдете по ссылке:

```
https://query.nytimes.com/search/sitesearch/#/python
```

то получите последние новостные статьи по поисковому запросу python. Выполнив веб-скрапинг этой страницы с помощью urllib или библиотеки Requests, вы не найдете результатов поиска. Они загружаются отдельно через следующий вызов API:

```
https://query.nytimes.com/svc/add/v1/sitesearch.json?q=python&spotlight=
```

true&facet=true

Если бы мы загрузили эту страницу с помощью Selenium, то нам пришлось бы для каждого поиска сделать примерно 100 запросов и передать 600–700 Кбайт данных. Используя API напрямую, мы делаем только один запрос и передаем всего лишь около 60 Кбайт красиво отформатированных данных — именно тех, которые нам нужны.

Поиск недокументированных API

В предыдущих главах мы использовали инспектор Chrome для проверки контента HTML-страницы. Теперь применим этот инспектор для несколько иной цели: исследования запросов и ответов на вызовы функций, которые задействуются при формировании страницы.

Для этого откройте окно инспектора Chrome и перейдите на вкладку **Network** (Сеть), как показано на рис. 12.1.

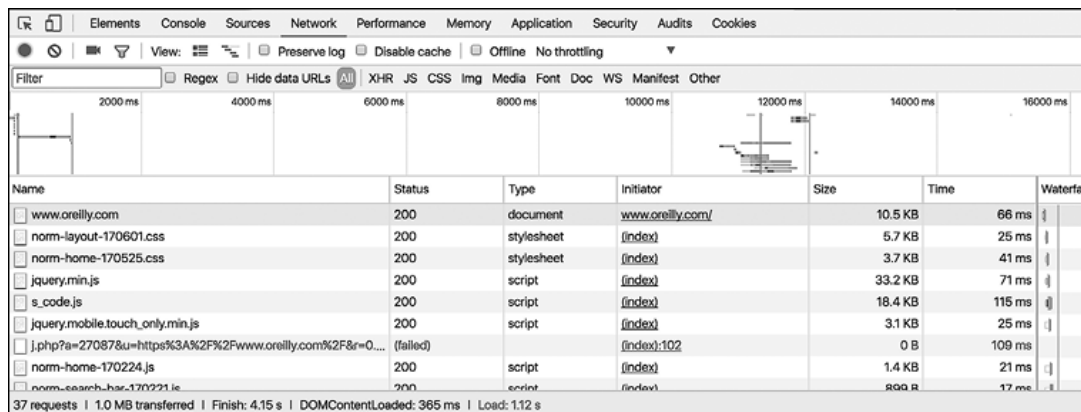


Рис. 12.1. Инструмент Chrome Network Inspector позволяет увидеть все вызовы функций, которые браузер делает и получает извне

Обратите внимание: это окно нужно открыть до загрузки страницы. Будучи закрытым, оно не отслеживает сетевые вызовы.

При загрузке страницы вы увидите строку, изменяющуюся в реальном времени, пока браузер обращается к веб-серверу с целью получить дополнительную информацию, необходимую для отображения страницы. Среди этих вызовов могут быть и вызовы API.

Поиск недокументированных API может потребовать небольшого детективного расследования (о том, как это сделать, см. в подразделе «Автоматический поиск и документирование API» на с. 223), особенно на крупных сайтах с большим количеством сетевых вызовов. Однако в большинстве случаев вы увидите это сразу.

Как правило, у вызовов API есть несколько свойств, отличающих их от других сетевых вызовов.

- Вызовы API часто содержат JSON или XML. Вы можете отфильтровать список таких запросов с помощью поля поиска и фильтрации.
- При GET-запросах URL обычно содержит значения передаваемых параметров. Это удобно, если, например, вы ищете вызов API, который возвращает результаты поиска или загружает данные для конкретной страницы. Просто отфильтруйте результаты по используемому поисковому запросу, идентификатору страницы или другой идентифицирующей информации.
- Обычно вызовы API относятся к типу XHR.

API не всегда заметны, особенно на больших сайтах с большим количеством функций, которые делают сотни вызовов при загрузке одной страницы. Однако со временем, приобретя немного практики, вы будете гораздо легче находить эту метафорическую иголку в стоге сена.

Документирование недокументированных API

После обнаружения вызова API часто бывает полезно хотя бы немного документировать его, особенно если ваши веб-скраперы будут в значительной степени опираться на данный вызов. Вероятно, вы захотите загрузить несколько страниц сайта, отфильтровывая нужный вызов API на вкладке **Network** (Сеть) в консоли инспектора. При этом вы можете заметить, как изменяется вызов от страницы к странице, и определить поля, которые он принимает и возвращает.

Каждый вызов API можно идентифицировать и задокументировать, обратив внимание на следующие поля:

- используемый метод HTTP;
- входные данные:
 - параметры пути;
 - заголовки (включая cookies);
 - контент тела (для вызовов PUT и POST);
- выходные данные:
 - заголовки ответа (включая набор cookie-файлов);
 - тип тела ответа;
 - поля тела ответа.

Автоматический поиск и документирование API

Работа по поиску и документированию API может показаться несколько утомительной и рутинной. Вероятно, потому, что по большей части так оно и есть. Некоторые сайты иногда пытаются скрыть, как браузер получает от них данные, и это несколько усложняет задачу, однако поиском и документированием API должны заниматься программы.

Я создала на GitHub репозиторий по адресу <https://github.com/REMitchell/apiscraper>, в котором попыталась выполнить некоторые базовые задачи из этой области.

Для загрузки страниц, сбора данных со страниц одного домена, анализа сетевого трафика, возникающего во время загрузки страниц, и преобразования этих запросов в читаемые вызовы API я использовала Selenium, ChromeDriver и библиотеку BrowserMob Proxy.

Для запуска данного проекта нам потребуется несколько важных частей, и прежде всего — само программное обеспечение.

Клонируйте проект GitHub `apiscraper` (<https://github.com/REMitchell/apiscraper>). Он должен включать следующие файлы:

- `sapicall.py` — содержит атрибуты, определяющие вызов API (путь, параметры и т.д.), а также логику, позволяющую установить, являются ли два вызова API одинаковыми;
- `apiFinder.py` — главный класс для веб-краулинга. Используются `webservice.py` и `consoleservice.py` для запуска процесса поиска API;
- `browser.py` — имеет только три метода: `initialize`, `get` и `close`, но выполняет весьма сложные действия, позволяющие связать прокси-сервер BrowserMob и Selenium. Прокручивает страницу с целью убедиться, что она

загружена целиком, сохраняет файлы HTTP Archive (HAR) в заданном месте для обработки;

- `console.service.py` — выполняет команды из консоли и запускает основной класс `APIFinder`;
- `harParser.py` — анализирует HAR-файлы и извлекает вызовы API;
- `html_template.html` — предоставляет шаблон для отображения вызовов API в браузере;
- `README.md` — страница readme для Git.

Скачайте с <https://bmp.lightbody.net/> двоичные файлы BrowserMob Proxy и поместите их в каталог проекта **apiscraper**.

На момент написания данной книги последней версией BrowserMob Proxy была 2.1.4, вследствие чего этот скрипт предполагает, что двоичные файлы находятся в каталоге **browsermob-proxy-2.1.4/bin/browsermob-proxy** относительно корневого каталога проекта. Если это не так, то вы можете указать другой каталог во время выполнения или (что, возможно, проще) внести изменения в код в `apiFinder.py`.

Скачайте `ChromeDriver` (<https://sites.google.com/a/chromium.org/chromedriver/downloads>) и поместите его в каталог проекта **apiscraper**.

Вам необходимо будет установить следующие библиотеки Python:

- `tldextract`;
- `selenium`;

- browsermob-proxy.

Когда эти операции по настройке будут завершены, можно начинать собирать вызовы API. Для начала введите команду:

```
$ python consoleservice.py -h
```

чтобы получить список параметров:

```
usage: consoleservice.py [-h] [-u [U]] [-d [D]]  
[-s [S]] [-c [C]] [-i [I]] [--p]
```

optional arguments:

```
-h, --help      show this help message and exit  
-u [U]          Target URL. If not provided,  
target directory will be scanned  
                  for har files.  
-d [D]          Target directory (default is  
"hars"). If URL is provided,  
                  directory will store har files.  
If URL is not provided,  
                  directory will be scanned.  
-s [S]          Search term  
-c [C]          File containing JSON formatted  
cookies to set in driver (with  
                  target URL only)  
-i [I]          Count of pages to crawl (with  
target URL only)  
--p             Flag, remove unnecessary  
parameters (may dramatically  
                  increase runtime)
```


Вызовы API, сделанные на одной странице, отличаются общим поисковым запросом. Например, на странице <http://target.com> можно найти API-запрос, возвращающий данные о продукте, которыми заполняется страница продукта:

```
$ python consoleservice.py -u
https://www.target.com/p/

rogue-one-a-star-wars-\story-blu-ray-dvd-
digital-3-disc/-/A-52030319 -s

"Rogue One: A Star Wars Story"
```

Эта команда возвращает информацию, включая URL, для API-запроса, который возвращает данные о продукте для этой страницы:

```
URL:
https://redsky.target.com/v2/pdp/tcin/52030319
METHOD: GET
AVG RESPONSE SIZE: 34834
SEARCH TERM CONTEXT:
c:"786936852318","product_description":
{"title":
"Rogue One: A Star Wars Story (Blu-ray + DVD +
Digital) 3 Disc",
"long_description":...
```

С помощью флага `-i` можно собирать данные с нескольких страниц (по умолчанию всего одна страница), начиная с предоставленного URL. Это может быть полезно для поиска по определенным ключевым словам во всем сетевом трафике или же, если опустить флаг ключевого слова `-s`, — для сбора всего трафика API, возникающего при загрузке каждой страницы.

Все собранные данные хранятся в виде HAR-файла, который по умолчанию размещается в каталоге **/har** в корне проекта, хотя этот каталог можно изменить с помощью флага **-d**.

Если URL не указан, то можно также выполнить поиск и анализ среди предварительно собранных HAR-файлов, размещенных в каталоге.

У этого проекта есть множество других функций, включая следующие:

- удаление ненужных параметров (параметров GET или POST, которые не влияют на возвращаемое значение вызова API);
- поддержку нескольких форматов вывода API (командная строка, HTML, JSON);
- возможность различать параметры пути, указывающие на определенный маршрут API, и те, которые просто играют роль параметров GET-запроса для одного и того же маршрута API.

Дальнейшее развитие проекта также планируется, поскольку я и мои коллеги продолжаем использовать его для веб-скрапинга и сбора данных API.

Объединение API с другими источниками данных

Смысл многих современных веб-приложений состоит в том, чтобы брать существующие данные и представлять их в более привлекательном виде, однако я бы поспорила с этим, сказав, что в большинстве случаев это неинтересно. Если вы используете API в качестве единственного источника данных, уж лучше просто скопировать чужую базу данных, которая уже

существует и, по сути, опубликована. Было бы гораздо интереснее взять два или более источника данных и по-новому объединить их или же использовать API в качестве инструмента, позволяющего взглянуть на извлеченные данные под другим углом.

Рассмотрим всего один пример того, как данные, полученные с помощью API, в сочетании с веб-скрапингом позволяют увидеть, какие части света вносят наибольший вклад в «Википедию».

Если вы потратили много времени, читая «Википедию», то вам, скорее всего, встречались страницы хронологии правок статей, на которых отображается список последних изменений. Если пользователь авторизовался в «Википедии» и внес изменения, то отображается его имя. Если же не авторизовался, то записывается его IP-адрес, как показано на рис. 12.2.

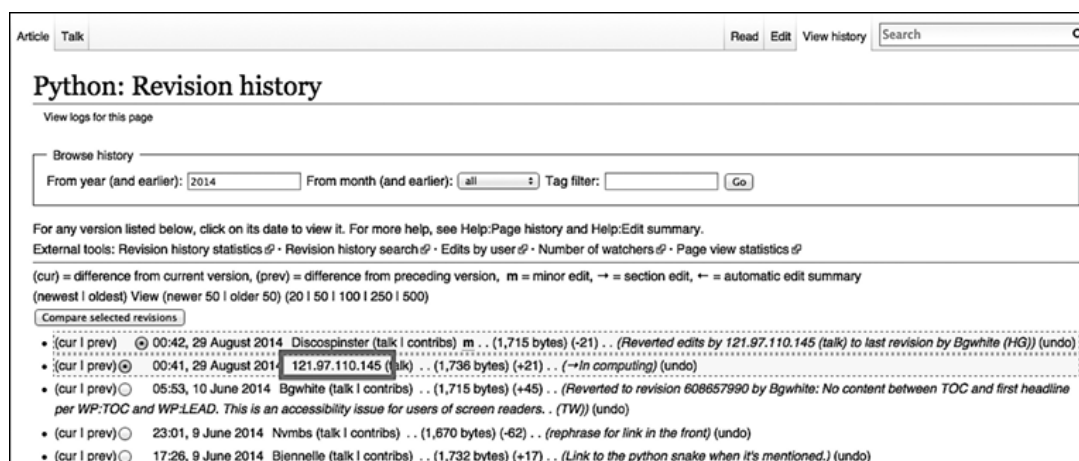


Рис. 12.2. IP-адрес анонимного редактора на странице хронологии правок статьи о Python в «Википедии»

На странице хронологии правок указан IP-адрес 121.97.110.145. Используя API `ip-api.com`, мы узнали, что на момент написания данной книги этот IP-адрес находился в

Кесон-Сити на Филиппинах (иногда географическое положение IP-адресов может изменяться).

Сама по себе эта информация не особенно интересна. А что, если собрать много таких точек с географическими координатами правок «Википедии» и данными о том, где они находятся? Несколько лет назад я так и сделала, используя библиотеку Google GeoChart (<https://developers.google.com/chart/interactive/docs/gallery/geochart>), и построила интересную диаграмму (<http://www.pythonscraping.com/pages/wikipedia.html>), показывающую, из каких точек земного шара вносятся изменения в англоязычную «Википедию», а также в «Википедию» на других языках (рис. 12.3).



Рис. 12.3. Визуализация правок «Википедии», построенная с помощью библиотеки Google GeoChart

Создать простейший скрипт, который бы собирал данные из «Википедии», находил страницы хронологии правок, а затем извлекал оттуда IP-адреса, несложно. Представленный ниже скрипт, использующий модифицированный код из главы 3, делает именно это:

```

from urllib.request import urlopen
from bs4 import BeautifulSoup
import json
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html =
urlopen('http://en.wikipedia.org{}'.format(articleUrl))
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div',
{'id': 'bodyContent'}).find_all('a',
href=re.compile('^(/wiki/)((?!:).)*$'))

def getHistoryIPs(pageUrl):
    # Страницы хронологии изменений имеют такой
    формат:
    # http://en.wikipedia.org/w/index.php?
    title=Title_in_URL&action=history.
    pageUrl = pageUrl.replace('/wiki/', '')
    historyUrl =
'http://en.wikipedia.org/w/index.php?title=
{}&action=history'
    .format(pageUrl)
    print('history url is:
{}'.format(historyUrl))
    html = urlopen(historyUrl)
    bs = BeautifulSoup(html, 'html.parser')

```

```

        # Находит только ссылки с классом "mw-
anonuserlink",
        # в которых указаны лишь IP-адреса, а не
имена пользователей.
        ipAddresses = bs.find_all('a',
{'class': 'mw-anonuserlink'})
        addressList = set()
        for ipAddress in ipAddresses:
            addressList.add(ipAddress.get_text())
        return addressList

links =
getLinks('/wiki/Python_(programming_language)')

while(len(links) > 0):
    for link in links:
        print('- '*20)

        historyIPs =
getHistoryIPs(link.attrs['href'])
        for historyIP in historyIPs:
            print(historyIP)

        newLink = links[random.randint(0,
len(links)-1)].attrs['href']
        links = getLinks(newLink)

```

В этой программе используются две основные функции: `getLinks` (уже знакомая нам по главе 3) и новая функция `getHistoryIPs`, которая ищет контент всех ссылок с классом `mw-anonuserlink` (указывающим на анонимного пользователя с IP-адресом, а не на пользователя с именем) и возвращает его в виде множества Python.

В этом коде также используется несколько произвольная (но тем не менее эффективная для данного примера) схема поиска статей, для которых можно извлечь хронологию правок. Согласно данной схеме сначала извлекаются хронологии правок всех статей «Википедии», на которые ссылается начальная страница (в нашем случае статья о языке программирования Python). Затем случайным образом выбирается новая начальная страница и извлекаются все страницы хронологии правок статей, на которые она ссылается. Так продолжается до тех пор, пока мы не попадем на страницу без ссылок.

Теперь, когда у нас есть код, извлекающий IP-адреса в виде строк, можно объединить его с функцией `getCountry` из предыдущего раздела, чтобы преобразовать эти IP-адреса в названия стран. Мы немного изменили код `getCountry`, чтобы учесть недействительные или искаженные IP-адреса, которые приведут к ошибке 404 (страница не найдена):

```
def getCountry(ipAddress):
    try:
        response = urlopen(
            'http://ip-api.com/json/{}'.format(ipAddress)).read().decode('utf-8')
    except HTTPError:
        return None
    responseJson = json.loads(response)
    return responseJson.get('country_code')

links =
getLinks('/wiki/Python_(programming_language)')
```

```

while(len(links) > 0):
    for link in links:
        print('- '*20)

                                historyIPs      =
getHistoryIPs(link.attrs["href"])
    for historyIP in historyIPs:
        country = getCountry(historyIP)
        if country is not None:
            print('{} is from
{}'.format(historyIP, country))

        newLink    =    links[random.randint(0,
len(links)-1)].attrs['href']
        links = getLinks(newLink)

```

Ниже представлен пример выполнения этой программы:

```

-----
history                                url                                is:
http://en.wikipedia.org/w/index.php?
title=Programming_
paradigm&action=history
68.183.108.13 is from US
86.155.0.186 is from GB
188.55.200.254 is from SA
108.221.18.208 is from US
141.117.232.168 is from CA
76.105.209.39 is from US
182.184.123.106 is from PK
212.219.47.52 is from GB
72.27.184.57 is from JM
49.147.183.43 is from PH

```


209.197.41.132 is from US
174.66.150.151 is from US

Дополнительные сведения об API

В данной главе показано несколько способов использования современных API для доступа к данным в Интернете для разработки более быстрых и мощных веб-скраперов. Если вы хотите не только задействовать, но и создавать API или же больше узнать о теории их построения и синтаксиса, то я рекомендую книгу *RESTful Web APIs* Леонарда Ричардсона (Leonard Richardson), Майка Амундсена (Mike Amundsen) и Сэма Руби (Sam Ruby) (издательство O'Reilly) (<http://bit.ly/RESTful-Web-APIs>). Она представляет собой подробный обзор теории и практики использования API в Интернете. Кроме того, у Майка Амундсена есть увлекательная серия видеороликов *Designing APIs for the Web* (издательство O'Reilly) (<http://oreil.ly/1GOXNhE>), где рассказывается о том, как разрабатывать собственные API. Это полезно уметь делать, если вы решите открыть доступ к собранным вами данным и представить их в удобном формате.

В то время как многие жалуются на вездесущий JavaScript и динамические сайты, из-за которых традиционные методы «захвата и анализа» HTML-страниц устаревают, я, со своей стороны, приветствую наших новых повелителей роботов. Поскольку динамические сайты меньше полагаются на HTML-страницы, рассчитанные на просмотр человеком, и в большей степени — на строго отформатированные файлы JSON, это полезно всем, кто стремится получить чистые, хорошо отформатированные данные.

Интернет больше не является коллекцией HTML-страниц со случайными украшениями в виде мультимедиа и CSS. Это

коллекция из файлов сотен типов и форматов, сотнями передаваемых по сети с целью сформировать страницы, которые вы просматриваете через браузер. Настоящая хитрость в том, чтобы чаще выглядывать за пределы страницы, которую вы видите, и получать данные непосредственно из ее источника.

[21](#) См.: Бэнкс А., Порселло Е. GraphQL: язык запросов для современных веб-приложений. — СПб.: Питер, 2019.

[22](#) Этот API преобразует IP-адреса в географические объекты, которые мы еще будем использовать в данной главе.

[23](#) В действительности во многих API вместо запросов PUT для обновления информации используются запросы POST. Независимо от того, создается новый объект или всего лишь изменяется старый, вопрос структуры API-запроса остается. Тем не менее все равно имеет смысл знать разницу между этими типами запросов, поскольку вы часто будете сталкиваться с запросами PUT в популярных API.

Глава 13. Обработка изображений и распознавание текста

Машинное зрение — обширнейшая область с далеко идущими планами: от самодвижущихся автомобилей Google до торговых автоматов, способных отличать фальшивые купюры от настоящих. В этой главе основное внимание будет уделено лишь одному небольшому аспекту данной области: распознаванию текста, а именно тому, как с помощью различных библиотек Python распознать и использовать текстовые изображения, найденные в Интернете²⁴.

Использование изображений вместо текста — обычная методика, применяемая в тех случаях, когда вы не хотите, чтобы этот текст находили и читали боты. Она часто встречается в контактных формах, когда адрес электронной почты частично или полностью заменяется изображением. В зависимости от того, насколько искусно это сделано, оно может быть даже незаметно для людей, просматривающих страницу. Но ботам такие изображения читать трудно, и данного приема бывает достаточно, чтобы большинство спамеров не смогли получить ваш адрес электронной почты.

В методиках капчи, конечно, применяется тот факт, что пользователи могут прочесть «секретные» изображения, а большинство ботов — нет. Одни изображения капчи сложнее, другие проще, и далее мы рассмотрим данную проблему.

Однако капча не единственный элемент Интернета, где необходимо помочь веб-скраперу преобразовать изображение в текст. Даже в наше время многие документы сканируются из печатных копий и в таком виде размещаются на сайтах, из-за чего часто являются недоступными, «спрятанными на видном месте». Если нет возможности преобразовать изображение в

текст, то остается единственный способ сделать эти документы доступными — ввести их вручную, и, конечно же, на него никогда не хватает времени.

Преобразование изображений в текст называется *оптическим распознаванием символов* (optical character recognition, OCR). Есть несколько крупных библиотек, выполняющих распознавание текста, и множество других библиотек, которые их поддерживают или построены на их основе. Временами система библиотек представляется довольно сложной, поэтому советую вам прочитать следующий раздел, прежде чем пытаться выполнить какое-либо из упражнений данной главы.

Обзор библиотек

Python — фантастически эффективный язык для обработки и чтения изображений, машинного обучения на основе изображений и даже создания изображений. Есть множество библиотек Python для обработки изображений, но мы сосредоточимся только на двух: Pillow и Tesseract.

В области обработки изображений и OCR в Интернете эти две библиотеки образуют мощный взаимодополняющий дуэт. *Pillow* выполняет первый проход, очищает и фильтрует изображения, а *Tesseract* пытается сопоставить фигуры, найденные на этих изображениях, со своей библиотекой известного текста.

В данной главе описываются установка и основы использования этих библиотек, а также приводится несколько примеров их совместного применения. Я также расскажу об углубленном обучении Tesseract, чтобы вы могли сами обучить Tesseract дополнительным шрифтам и языкам для решения

задач OCR (или даже распознавания капчи), с которыми можете столкнуться в Интернете.

Pillow

Возможно, Pillow не самая полнофункциональная библиотека для обработки изображений, но в ней есть все функции, которые вам, скорее всего, понадобятся, и даже больше — если только вы не планируете переписать Photoshop на Python, в таком случае вы читаете не ту книгу! У Pillow также есть преимущество: это одна из самых хорошо документированных сторонних библиотек, и она чрезвычайно проста в использовании без дополнительной настройки.

В дополнение к Python Imaging Library (PIL) для Python 2.x, на которой основана Pillow, эта библиотека включает в себя поддержку Python 3.x. Как и ее предшественница, Pillow позволяет легко импортировать изображения и обрабатывать их с помощью различных фильтров, масок и даже пиксельных преобразований:

```
from PIL import Image, ImageFilter

kitten = Image.open('kitten.jpg')
blurryKitten = kitten.filter(ImageFilter.GaussianBlur)
blurryKitten.save('kitten_blurred.jpg')
blurryKitten.show()
```

В этом примере изображение `kitten.jpg` открывается в используемой по умолчанию программе просмотра изображений, где к нему добавляется размытие, после чего размытое изображение сохраняется в файле `kitten_blurred.jpg` в том же каталоге.

Мы будем использовать Pillow для предварительной обработки изображений, чтобы упростить их машинное распознавание, однако, как уже упоминалось, помимо простых операций фильтрации, эта библиотека позволяет делать многое другое. Дополнительную информацию о ней см. в документации по Pillow (<http://pillow.readthedocs.org/>).

Tesseract

Tesseract — это библиотека для OCR. Благодаря поддержке Google (компания, известной своими технологиями оптического распознавания символов и машинного обучения), считается лучшей и наиболее точной из доступных систем распознавания текста с открытым исходным кодом.

Помимо точности, библиотека также чрезвычайно гибкая. Ее можно научить распознавать любое количество шрифтов (если они относительно непротиворечивы сами по себе, как вы скоро увидите). Кроме того, ее можно расширить для распознавания любого символа Unicode.

В данной главе мы будем использовать как утилиту командной строки Tesseract, так и ее стороннюю Python-оболочку pytesseract. Обе будут явно различаться по названию, поэтому знайте: когда вы видите в тексте слово Tesseract, речь идет об утилите командной строки, а когда pytesseract — о ее сторонней оболочке для Python.

Установка Tesseract

Для пользователей Windows есть удобная программа-инсталлятор Tesseract (<https://code.google.com/p/tesseract-ocr/downloads/list>). На момент написания этой книги ее последней версией была 3.02, хотя более новые версии тоже подойдут.

Пользователи Linux могут установить Tesseract с помощью apt-get:

```
$ sudo apt-get tesseract-ocr
```

Установить Tesseract на Mac немного сложнее, хотя эту задачу можно упростить с помощью многочисленных сторонних инсталляторов, таких как Homebrew (<http://brew.sh/>), который мы использовали в главе 6 для установки MySQL. Например, можно установить Homebrew и применять его для установки Tesseract с помощью следующих двух команд:

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/ \
install/master/install)"
$ brew install tesseract
```

Tesseract также можно установить непосредственно из источника, на странице скачивания проекта (<https://code.google.com/p/tesseract-ocr/downloads/list>).

Чтобы использовать некоторые функции Tesseract, такие как обучение программы распознаванию новых символов, чем мы займемся далее в этом разделе, необходимо установить переменную среды `$TESSDATA_PREFIX`, чтобы программа знала, где хранятся файлы данных.

В большинстве систем Linux и в macOS это можно сделать следующим образом:

```
$ export TESSDATA_PREFIX=/usr/local/share/
```

Обратите внимание: `/usr/local/share/` — каталог, в котором Tesseract размещает данные по умолчанию, однако стоит убедиться, что это верно для вашего варианта установки.

Аналогично в Windows вы можете создать переменную среды с помощью следующей команды:

```
#      setx      TESSDATA_PREFIX      C:\Program  
Files\Tesseract OCR\
```

pytesseract

После установки Tesseract мы готовы установить библиотеку-оболочку Python `pytesseract`, использующую уже установленную нами программу Tesseract для чтения файлов изображений, формируя на выходе строки и объекты, которые затем можно применять в скриптах Python.



Для выполнения примеров нужен pytesseract 0.1.9

Имейте в виду: между версиями `pytesseract` 0.1.8 и 0.1.9 есть ряд существенных изменений (в том числе появившихся при участии автора этой книги). В данном пункте описываются функции, которые есть только в версии 0.1.9 данной библиотеки. Пожалуйста, выполняя примеры кода, представленные в этой главе, убедитесь, что установили правильную версию.

Как обычно, вы можете установить `pytesseract` через `pip` или скачать библиотеку со страницы проекта `pytesseract` (<https://pypi.python.org/pypi/pytesseract>) и выполнить следующую команду:


```
$ python setup.py install
```

Библиотеку pytesseract можно использовать в сочетании с PIL для чтения текста из изображений:

```
from PIL import Image
import pytesseract

print(pytesseract.image_to_string(Image.open('files/test.png')))
```

Если вы установили библиотеку Tesseract в папке Python, то можете указать pytesseract это местоположение, добавив следующую строку:

```
pytesseract.pytesseract.tesseract_cmd =  
'/path/to/tesseract'
```

Кроме выдачи результатов OCR для изображений, как было показано в предыдущем примере, у pytesseract есть еще несколько полезных функций. Эта библиотека позволяет определять box-файлы (группы пикселей, заключенные в границах символа):

```
print(pytesseract.image_to_boxes(Image.open('files/test.png')))
```

Она также дает возможность выводить полные данные, в том числе степень уверенности, номера страниц и строк, информацию о границах и т.п.:

```
print(pytesseract.image_to_data(Image.open('files/test.png')))
```

По умолчанию для этих двух последних файлов выводятся строковые файлы с разделителями в виде пробелов или табуляций, но также можно получить результаты в виде словарей или (если декодирования в UTF-8 недостаточно) байтовых строк:

```
from PIL import Image
import pytesseract
from pytesseract import Output

print(pytesseract.image_to_data(Image.open('files/test.png'),
    output_type=Output.DICT))
print(pytesseract.image_to_string(Image.open('files/test.png'),
    output_type=Output.BYTES))
```

В этой главе библиотека `pytesseract` применяется в сочетании с утилитой командной строки `Tesseract`, которая запускается из Python через библиотеку `subprocess`. Библиотека `pytesseract` полезна и удобна, однако есть функции `Tesseract`, которые она не может выполнять, поэтому стоит ознакомиться и с другими методами.

NumPy

Для простого распознавания текста библиотека `NumPy` не нужна, однако понадобится далее в этой главе, когда мы будем обучать `Tesseract` распознавать дополнительные наборы символов и другие шрифты. Кроме того, применим `NumPy` для решения простых математических задач (таких как вычисление средневзвешенных значений) в ряде примеров, которые рассмотрим позже.

NumPy — это мощная библиотека, используемая для линейной алгебры и других серьезных математических приложений. Она хорошо сочетается с Tesseract благодаря способности математически представлять изображения и обрабатывать их в виде больших массивов пикселей.

NumPy можно установить с помощью любого стороннего инсталлятора Python, такого как pip, или скачав пакет (<https://pypi.python.org/pypi/numpy>) и установив его через команду `$pythonsetup.pyinstall`.

Даже если вы не планируете запускать какие-либо примеры кода с помощью этой библиотеки, я настоятельно рекомендую установить ее и добавить в свой арсенал Python. Она дополняет встроенную математическую библиотеку Python и имеет много полезных функций, особенно для операций со списками чисел.

По соглашению NumPy импортируется как `np` и может использоваться следующим образом:

```
import numpy as np

numbers = [100, 102, 98, 97, 103]
print(np.std(numbers))
print(np.mean(numbers))
```

В этом примере вычисляется стандартное отклонение и среднее арифметическое набора чисел, переданного программе.

Обработка хорошо отформатированного текста

Если повезет, то большая часть текста, который вам придется обрабатывать, будет относительно чистой и хорошо отформатированной. Такой текст обычно соответствует

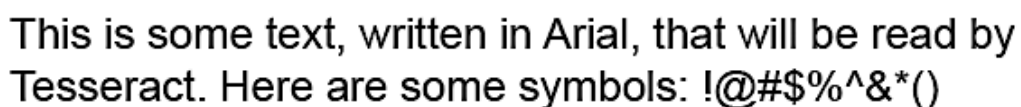
нескольким требованиям, хотя грань между «грязным» и «хорошо отформатированным» текстом может быть весьма субъективной.

В целом, хорошо отформатированный текст удовлетворяет следующим требованиям:

- набран одним стандартным шрифтом (без использования курсива, рукописных и чрезмерно декоративных шрифтов);
- при копировании или фотографировании имеет очень четкие края без артефактов копирования и темных пятен;
- хорошо выровнен, без перекошенных букв;
- не выходит за пределы изображения, не имеет обрезанных краев или полей по краям изображения.

Какие-то из этих моментов можно исправить в процессе предварительной обработки. Например, преобразовать цвета в оттенки серого, отрегулировать яркость и контрастность, а само изображение при необходимости обрезать и повернуть. Однако некоторые фундаментальные ограничения могут потребовать более глубокого обучения. Подробнее об этом см. в разделе «Чтение капчи и обучение Tesseract» данной главы на с. 247.

На рис. 13.1 показан идеальный пример хорошо отформатированного текста.



This is some text, written in Arial, that will be read by Tesseract. Here are some symbols: !@#\$%^&*()

Рис. 13.1. Образец текста, сохраненный в .tiff-файле и предназначенный для чтения с помощью Tesseract

Чтобы прочитать этот файл и записать результаты в текстовый файл, можно запустить Tesseract из командной строки:

```
$ tesseract text.tif textoutput | cat textoutput.txt
```

В результате получим строку с информацией о библиотеке Tesseract, где говорится, что она работает, после чего следует контент созданного файла `textoutput.txt`:

```
Tesseract Open Source OCR Engine v3.02.02 with
Leptonica
This is some text, written in Arial, that will
be read by
Tesseract. Here are some symbols: !@#$%^&'()
```

Результат в основном точен, за исключением символов `^` и `*`, которые были интерпретированы как двойные и одинарные кавычки соответственно. Однако в целом это делает текст более удобным для чтения.

Если размыть текст изображения, добавить несколько артефактов JPG-сжатия и небольшой градиент фона, то результаты станут намного хуже (рис. 13.2).

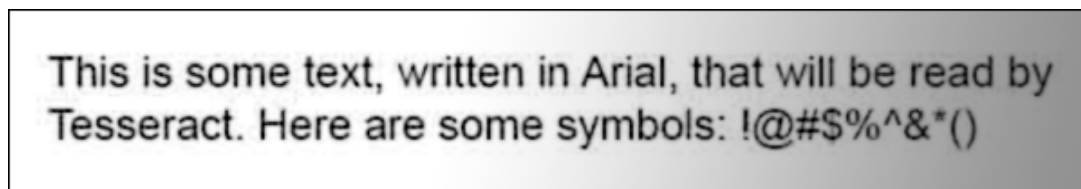


Рис. 13.2. К сожалению, многие документы, которые вам встретятся в Интернете, будут выглядеть приблизительно как этот пример

Tesseract справился с этим изображением далеко не так хорошо, прежде всего из-за градиента фона, и выдал следующий результат:

```
This is some text, written In Arlal, that"  
Tesseract. Here are some symbols: _
```

Обратите внимание: текст обрезается сразу, как только фоновый градиент затрудняет распознавание, и последний символ в каждой строке неверен, поскольку Tesseract безуспешно пытается его интерпретировать. Кроме того, JPG-артефакты и размытость мешают Tesseract отличать строчные *i* и прописные *I* от цифры 1.

Именно здесь нам пригодится скрипт Python, с помощью которого мы сначала очистим изображение. Используя библиотеку Pillow, мы можем создать пороговый фильтр, который позволит избавиться от серого фона, выделить текст и сделать изображение более четким для последующего чтения в Tesseract.

Далее вместо того, чтобы запускать Tesseract из командной строки, мы можем использовать библиотеку pytesseract для выполнения команд Tesseract и чтения полученного файла:

```
from PIL import Image  
import pytesseract  
  
def cleanFile(filePath, newFilePath):  
    image = Image.open(filePath)  
  
    # Устанавливаем пороговое значение для  
    изображения и сохраняем его.  
    image = image.point(lambda x: 0 if x < 143  
else 255)
```

```

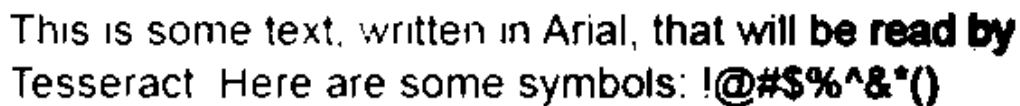
image.save(newFilePath)
return image

image = cleanFile('files/textBad.png',
'files/textCleaned.png')

# Вызываем tesseract, чтобы выполнить OCR
созданного изображения.
print(pyesseract.image_to_string(image))

```

Полученное изображение, автоматически сохраненное в файле textCleaned.png, показано на рис. 13.3.



This is some text, written in Arial, that will be read by
Tesseract Here are some symbols: !@#\$%^&*()

Рис. 13.3. Мы получили это изображение, пропустив исходное «грязное» изображение через пороговый фильтр

За исключением некоторых трудночитаемых или отсутствующих знаков пунктуации, текст читается — по крайней мере для нас. Теперь Tesseract сделает все, что от него зависит:

```

This us some text' written In Anal, that will
be read by
Tesseract Here are some symbols: !@#$%"&'()

```

Точки и запятые, будучи чрезвычайно мелкими, стали первыми жертвами этого изображения, походя друг на друга и почти исчезая как для нас, так и для Tesseract. Вдобавок здесь неудачное ошибочное распознавание слова Arial как Anal,

поскольку Tesseract интерпретировал буквы *г* и *і* как одну букву *п*.

Тем не менее это гораздо лучше, по сравнению с предыдущей версией, где была обрезана почти половина текста.

Главное слабое место Tesseract — разная яркость фона. Алгоритмы Tesseract пытаются автоматически отрегулировать контрастность изображения перед чтением текста, но вам, скорее всего, удастся добиться лучших результатов, выполнив это самостоятельно с помощью такого инструмента, как библиотека Pillow.

К изображениям, которые следует обязательно откорректировать перед передачей в Tesseract, относятся изображения с наклонным текстом, большими нетекстовыми областями или другими проблемами.

Автоматическая коррекция изображений

В предыдущем примере значение 143 было подобрано экспериментально в качестве «идеального» порога для преобразования всех пикселов изображения в черные или белые для того, чтобы Tesseract смог прочесть текст, содержащийся в изображении. Но как быть, если изображений много, у каждого свои проблемы с градациями серого и вы не можете откорректировать их все вручную в разумных пределах?

Один из способов найти наилучшее (или хотя бы достаточно хорошее) решение — запустить Tesseract для набора изображений, где каждое из них откорректировано со своим пороговым значением, и алгоритмически выбрать тот вариант, который демонстрирует наилучший результат, измеряемый неким сочетанием количества символов и/или строк, которые

Tesseract может прочесть, и «достоверностью», с которой были прочитаны эти символы.

Выбор алгоритма может немного различаться в разных приложениях, но вот один из вариантов перебора разных пороговых параметров при обработке изображения с целью найти «лучший» параметр:

```
import pytesseract
from pytesseract import Output
from PIL import Image
import numpy as np

def cleanFile(filePath, threshold):
    image = Image.open(filePath)
    # Выбираем пороговое значение для
    изображения и сохраняем его.
    image = image.point(lambda x: 0 if x <
threshold else 255)
    return image

def getConfidence(image):
    data = pytesseract.image_to_data(image,
output_type=Output.DICT)
    text = data['text']
    confidences = []
    numChars = []

    for i in range(len(text)):
        if data['conf'][i] > -1:
            confidences.append(data['conf'][i])
            numChars.append(len(text[i]))
```

```
        return np.average(confidences,
weights=numChars), sum(numChars)
```

```
filePath = 'files/textBad.png'
```

```
start = 80
```

```
step = 5
```

```
end = 200
```

```
for threshold in range(start, end, step):
    image = cleanFile(filePath, threshold)
    scores = getConfidence(image)
    print("threshold: " + str(threshold) + ",
confidence: "
        + str(scores[0]) + " numChars " +
str(scores[1]))
```

У этого скрипта есть две функции:

- `cleanFile` — принимает исходный «плохой» файл и значение порога, с которым запускает пороговый инструмент PIL. Обработывает файл и возвращает объект изображения PIL;
- `getConfidence` — принимает PIL-объект с очищенным изображением и пропускает его через Tesseract. Вычисляет среднюю достоверность каждой распознанной строки (взвешенной по количеству символов в этой строке), а также количество распознанных символов.

Изменяя пороговое значение и вычисляя каждый раз достоверность и количество распознанных символов, получим

следующий результат:

threshold:	80,	confidence:	61.8333333333	nu
mChars	18			
threshold:	85,	confidence:	64.9130434783	nu
mChars	23			
threshold:	90,	confidence:	62.2564102564	nu
mChars	39			
threshold:	95,	confidence:	64.5135135135	nu
mChars	37			
threshold:	100,	confidence:	60.7878787879	n
umChars	66			
threshold:	105,	confidence:	61.9078947368	n
umChars	76			
threshold:	110,	confidence:	64.6329113924	n
umChars	79			
threshold:	115,	confidence:	69.7397260274	n
umChars	73			
threshold:	120,	confidence:	72.9078947368	n
umChars	76			
threshold:	125,	confidence:	73.582278481	nu
mChars	79			
threshold:	130,	confidence:	75.6708860759	n
umChars	79			
threshold:	135,	confidence:	76.8292682927	n
umChars	82			
threshold:	140,	confidence:	72.1686746988	n
umChars	83			
threshold:	145,	confidence:	75.5662650602	n
umChars	83			
threshold:	150,	confidence:	77.5443037975	n
umChars	79			

threshold:	155,	confidence:	79.1066666667	n
umChars	75			
threshold:	160,	confidence:	78.4666666667	n
umChars	75			
threshold:	165,	confidence:	80.1428571429	n
umChars	70			
threshold:	170,	confidence:	78.4285714286	n
umChars	70			
threshold:	175,	confidence:	76.3731343284	n
umChars	67			
threshold:	180,	confidence:	76.7575757576	n
umChars	66			
threshold:	185,	confidence:	79.4920634921	n
umChars	63			
threshold:	190,	confidence:	76.0793650794	n
umChars	63			
threshold:	195,	confidence:	70.6153846154	n
umChars	65			

Существует четкая тенденция как по средней достоверности результата, так и по количеству распознанных символов. Оба показателя стремятся к максимуму в районе порогового значения 145, что близко к найденному вручную «идеальному» результату 143.

Пороговые значения от 140 до 145 дают максимальное количество распознанных символов (83), но пороговое значение 145 показывает наибольшую достоверность для этих символов, поэтому мы можем выбрать данный результат и вернуть текст, который был распознан с данным порогом как «наилучшее предположение» того, какой именно текст содержится в изображении.

Конечно, простое распознавание «большинства» символов еще не говорит о правильном распознавании всех этих

символов. При некоторых пороговых значениях Tesseract может разбивать отдельные символы на несколько или интерпретировать случайный шум в изображении как текстовый символ, которого на самом деле не существует. В такой ситуации имеет смысл больше полагаться на среднюю достоверность каждого результата.

Например, предположим, что среди результатов оказались (в том числе) следующие:

```
threshold: 145, confidence: 75.5662650602  
numChars 83
```

```
threshold: 150, confidence: 97.1234567890  
numChars 82
```

Тогда из этих двух вариантов, бесспорно, следует выбрать тот, который дает более чем 20-процентное повышение достоверности с потерей всего лишь одного символа, и предположить, что результат с порогом 145 оказался просто неверным, или, возможно, один символ разделился на два, или же алгоритм нашел нечто, чего не было в тексте.

Здесь имеет смысл предварительно немного поэкспериментировать, чтобы подобрать наилучший порог для алгоритма. Например, можно выбрать вариант, при котором достигается максимальное *произведение* достоверности и количества распознанных символов (в таком случае для порога 145 это значение равно 6272, и в нашем воображаемом примере побеждает пороговое значение 150, для которого это произведение равно 7964) или же какой-либо другой показатель.

Обратите внимание: такие алгоритмы выбора работают не только с обычными `threshold`, но и с произвольными значениями инструмента PII. Кроме того, PII позволяет делать

выбор на основе двух и более значений, варьируя их и выбирая наилучший результат аналогичным образом.

Очевидно, что такие алгоритмы выбора требуют больших вычислительных затрат. Мы многократно запускаем PIL и Tesseract для каждого изображения, в то время как, зная мы «идеальные» пороговые значения заранее, пришлось бы запускать эти программы только один раз.

Учтите: углубившись в обработку изображений, вы можете начать замечать закономерности в найденных «идеальных» значениях. Вместо проверки всех пороговых значений от 80 до 200 на практике может потребоваться проверить только пороги от 130 до 180.

Вы можете даже применить другой подход и при первом проходе устанавливать пороговые значения с шагом, например, 20, а затем использовать «жадный» алгоритм, чтобы усовершенствовать наилучший результат, уменьшив шаг для порогов между «лучшими» решениями, найденными на предыдущей итерации. Такой вариант может оказаться оптимальным и в случае выбора по нескольким переменным.

Веб-скрапинг текста, представленного в виде изображений на сайтах

Применение Tesseract для чтения текста из изображения, сохраненного на жестком диске, кому-то покажется не особенно интересным, но при использовании веб-скрапера может стать мощным инструментом. Иногда изображения, содержащие текст на сайтах, непреднамеренно вводят в заблуждение (как в случае меню на сайте ресторана в формате JPG), но часто их задействуют целенаправленно, чтобы скрывать текст, как я покажу на следующем примере.

Хоть в файле `robots.txt` сайта Amazon и стоит разрешение на просмотр страниц товаров, предварительный просмотр книг обычно недоступен для ботов. Это стало возможным потому, что предварительный просмотр книг загружается с помощью специальных скриптов Ajax, а изображения тщательно скрываются под слоями тегов `div`. Для обычного посетителя сайта эти изображения, пожалуй, больше похожи на Flash-презентации, чем на файлы изображений. Но даже если бы вы могли добраться до изображений, остается не такой уж простой вопрос: как прочитать содержащийся в них текст?

Следующий скрипт совершает именно этот подвиг: переходит к крупноформатному изданию²⁵ повести Л. Толстого «Смерть Ивана Ильича», открывает раздел чтения книги, собирает URL изображений, а затем последовательно скачивает, читает и выводит текст каждого из них.

Обратите внимание: корректная работа этого кода зависит от текущего списка товаров Amazon, а также от некоторых архитектурных особенностей данного сайта. Если список товаров исчезнет или будет изменен, то достаточно лишь поставить в коде страницы предварительного просмотра URL другой книги (я считаю, что лучше выбирать крупноформатные издания, набранные шрифтом без засечек).

Поскольку это относительно сложный код, который опирается на несколько концепций, изложенных в предыдущих главах, я добавила комментарии, чтобы вам было проще понять происходящее ниже:

```
import time
from urllib.request import urlretrieve
from PIL import Image
import tesseract
```

```

from selenium import webdriver

def getImageText(imageUrl):
    urlretrieve(image, 'page.jpg')
    p = subprocess.Popen(['tesseract',
        'page.jpg', 'page'],
        stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    p.wait()
    f = open('page.txt', 'r')
    print(f.read())

# создаем новый драйвер Selenium
driver = webdriver.Chrome(executable_path='<Path to chromedriver>')

driver.get('https://www.amazon.com/Death-Ivan-Ilyich\' \
    -Nikolayevich-Tolstoy/dp/1427027277')
time.sleep(2)

# Нажимаем кнопку предварительного просмотра
# книги.
driver.find_element_by_id('imgBlkFront').click()
imageList = []

# Ждем, пока загрузится страница.
time.sleep(5)

while 'pointer' in driver.find_element_by_id(

```



```

        'sitbReaderRightPageTurner').get_attribute(
'style'):
        # Листаем страницы, пока доступна стрелка
вправо.
        driver.find_element_by_id('sitbReaderRightP
ageTurner').click()
        time.sleep(2)
        # Получаем все загружаемые страницы (может
загружаться
        # несколько страниц одновременно,
        # но дубликаты не добавляются в множество).
        pages =
driver.find_elements_by_xpath('//div[@class=\'p
ageImage\']/div/img')
        if not len(pages):
            print('No pages found')
        for page in pages:
            image = page.get_attribute('src')
            print('Found image: {}'.format(image))
            if image not in imageList:
                imageList.append(image)
                getImageText(image)
driver.quit()

```

Теоретически этот скрипт можно выполнять с любым веб-драйвером Selenium, но я обнаружила, что в настоящее время он наиболее надежно работает с Chrome.

Как мы уже знаем по предыдущему опыту работы с функцией чтения Tesseract, она выводит длинные отрывки из книги в целом разборчиво, что видно из предварительного просмотра первой главы:

Chapter I

During an Interval In the Melvmskl trial In the large building of the Law Courts the members and public prosecutor met in [van Egorowch Shebek's private room, where the conversation turned on the celebrated Krasovski case. Fedor Vasillevich warmly maintained that it was not subject to their jurisdiction, Ivan Egorovich maintained the contrary, while Peter ivanowch, not havmg entered into the discussmn at the start, took no part in it but looked through the Gazette which had Just been handed in.

"Gentlemen," he said, "Ivan Ilych has died!"

Однако во многих словах здесь есть очевидные ошибки, такие как Melvmsl вместо фамилии Melvinski и discussmn вместо discussion. Многие подобные ошибки можно исправить, делая предположения на основе списка словарных слов (вероятно, с дополнениями, основанными на соответствующих именах собственных, таких как Melvinski).

Иногда ошибочным может оказаться целое слово, например:

it is he who is dead and not 1²⁶.

В данном случае слово I («я») заменено цифрой 1. Здесь, в дополнение к словарю слов, пригодится анализ цепей Маркова. Если какая-либо часть текста содержит крайне необычную фразу (and not 1), то можно предположить, что на самом деле это более распространенная фраза (and not I).

Конечно, здесь помогает тот факт, что такие замены символов выполняются по предсказуемым схемам: *v* заменяется на *w*, а *I* — на *1*. Если эти замены встречаются часто, то можно создать список и использовать его, чтобы «пробовать» новые слова и фразы, выбирая наиболее разумное решение. Подход может состоять в том, чтобы заменять символы, часто распознаваемые некорректно, и задействовать тот вариант, который соответствует слову в словаре или является общепризнанной (или наиболее распространенной) *n*-граммой.

Если вы решите воспользоваться этим методом, то обязательно прочитайте главу 9, чтобы получить дополнительную информацию о работе с текстом и обработке естественного языка.

Несмотря на то что текст в этом примере набран обычным шрифтом без засечек и Tesseract, по идее, должен относительно легко его распознавать, иногда небольшое дообучение помогает повысить точность. В следующем разделе обсуждается другой подход к решению задачи распознавания искаженного текста, который потребует небольших предварительных затрат времени.

Если предоставить Tesseract большой набор изображений с известными текстами, то можно «научить» программу гораздо точнее и достовернее распознавать тот же шрифт в будущем, даже несмотря на периодически возникающие проблемы с фоном и размещением текста на странице.

Чтение капчи и обучение Tesseract

Слово CAPTCHA знают все, но мало кому известно его значение: *Completely Automated Public Turing Test to Tell Computers and Humans Apart* («полностью автоматизированный публичный тест Тьюринга, позволяющий отличить компьютер от человека»). Эта неуклюжая аббревиатура намекает на примерно такую же неуклюжую роль тестов капчи, которые создают препятствия во вполне удобных веб-интерфейсах: и люди, и роботы часто испытывают трудности, пытаясь выполнить эти тесты.

Тест Тьюринга впервые был описан Аланом Тьюрингом в его работе 1950 года «Вычислительные машины и разум». В этой статье автор описал систему, в которой человек общался бы как с людьми, так и с программами искусственного интеллекта через компьютерный терминал. Если в процессе произвольной беседы человек не мог отличить другого человека от программы с ИИ, то считалось, что эта программа прошла тест Тьюринга, а искусственный интеллект, как рассуждал автор, можно было бы считать по-настоящему «мыслящим».

Ирония состоит в том, что за последние 60 лет мы перешли от использования этих тестов для проверки машин к их применению для тестирования самих себя и получили странные результаты. Недавно Google отказалась от известной своей сложностью системы reCAPTCHA, во многом из-за ее тенденции блокировать легальных пользователей сайта²⁷.

Другие капчи несколько проще. Например, в Drupal, распространенной системе управления контентом на основе PHP, есть популярный модуль капчи (<https://www.drupal.org/project/captcha>), способный генерировать тестовые изображения разной степени

сложности. По умолчанию изображение выглядит так, как показано на рис. 13.4.

Почему людям и машинам гораздо легче прочитать этот текст капчи, по сравнению с другими?

- Символы не перекрываются и не пересекают границ пространства друг друга по горизонтали. Другими словами, вокруг каждого символа можно нарисовать правильный прямоугольник, который не будет пересекать другие символы.
- Нет фоновых изображений, линий и другого отвлекающего мусора, который бы вводил в заблуждение OCR-программы.
- Для данного изображения это не очевидно, но в капче используется всего несколько шрифтов. В тексте чередуется чистый шрифт без засечек (на рисунке это символы 4 и M) и шрифт, похожий на рукописный (символы m, C и 3).
- Высокий контраст между белым фоном и темными символами.

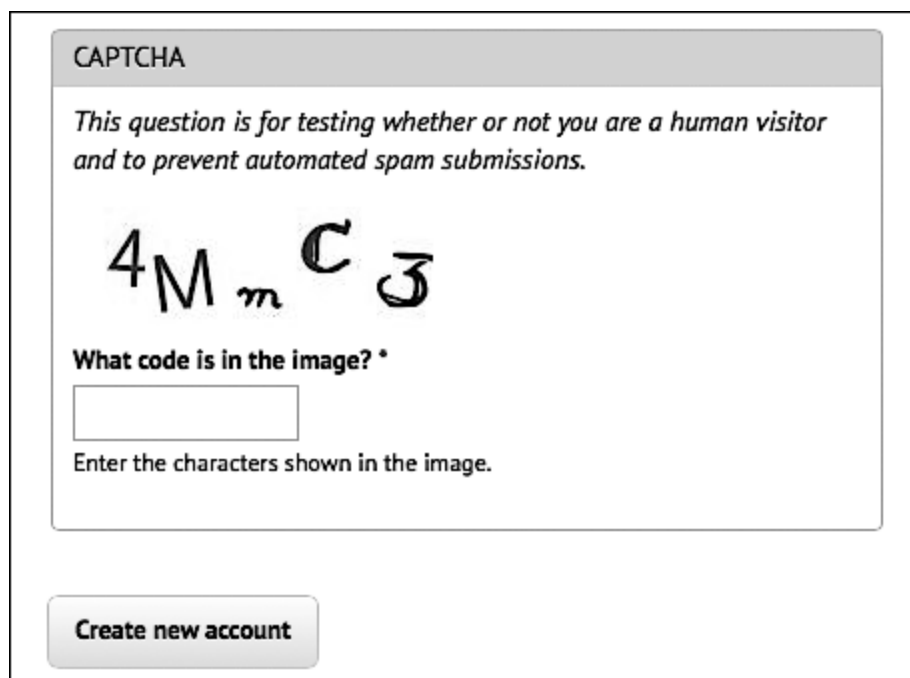


Рис. 13.4. Пример текстового изображения, используемого по умолчанию в проекте Drupal CAPTCHA

Тем не менее в этом изображении капчи есть несколько помех, которые затрудняют чтение текста OCR-программами:

- используются не только буквы, но и цифры, что увеличивает количество потенциально возможных символов;
- буквы со случайным наклоном легко читаются людьми, но могут сбить с толку OCR-программы;
- странноватый рукописный шрифт вызывает особые трудности, а пара дополнительных линий в С и 3 и необычно маленькая строчная буква m требуют дополнительного обучения компьютера.

Запустив Tesseract для этого изображения с помощью команды:

```
$ tesseract captchaExample.png output
```

получим следующий файл output.txt:

```
4N\ , , ,C<3
```

Программа правильно распознала символы 4, C и 3, но определенно не сможет в обозримом будущем заполнить поле формы, защищенное этим изображением капчи.

Обучение Tesseract. Чтобы обучить Tesseract распознавать написанное, будь то неясный и трудночитаемый шрифт или капча, нужно предоставить программе несколько примеров каждого символа.

Здесь самое время включить хороший подкаст или фильм, поскольку это пара часов довольно скучной работы. Первым шагом будет скачивание нескольких примеров ваших изображений капчи в один каталог. Количество примеров, которые вы подберете, будет зависеть от сложности капчи; я использовала 100 образцов файлов (всего 500 символов, или в среднем около восьми примеров на каждый символ); в моем случае для обучения на капче это, похоже, сработало неплохо.



Рекомендую присваивать изображениям имена по тому тексту капчи, который в них содержится (например, 4MmC3.jpg). Я обнаружила, что это помогает быстро оценить ошибки сразу для большого количества файлов; достаточно просмотреть все миниатюры файлов и сравнить изображения с именами. Это также очень помогает при проверке ошибок на последующих этапах.

Второй шаг — объяснить Tesseract, что именно означает каждый символ и где он находится в изображении. Сюда входит создание box-файлов, по одному на каждое изображение капчи. Box-файл выглядит так:

```
4 15 26 33 55 0
M 38 13 67 45 0
m 79 15 101 26 0
C 111 33 136 60 0
3 147 17 176 45 0
```

Первый символ — этот символ в изображении, следующие четыре числа — координаты прямоугольника, в который заключено изображение этого символа, а последнее число — номер страницы, используемой при обучении многостраничных документов (в нашем случае 0).

Очевидно, эти box-файлы не очень-то приятно создавать вручную, но здесь помогут различные инструменты. Мне нравится онлайн-инструмент Tesseract OCR Chopper, поскольку он не требует установки или дополнительных библиотек, работает на любом компьютере, где есть браузер, и относительно прост в использовании. Загрузите изображение, нажмите внизу кнопку **Add** (Добавить), если вам нужны дополнительные box-файлы, отрегулируйте их размер при необходимости и скопируйте и вставьте текст в новый box-файл.

Box-файлы нужно сохранять как обычный текст, в файлах с расширением .box. Аналогично файлам изображений их удобно именовать в соответствии с решениями капчи, которые они представляют (например, 4MmC3.box). Это также облегчает перепроверку и сравнение текстового контента в box-файлах с содержимым одноименного файла с изображением, если упорядочить файлы в каталоге по именам.

Аналогично случаям с изображениями вам придется создать примерно 100 таких файлов, чтобы получить достаточно данных. Кроме того, Tesseract может отбросить некоторые файлы как нечитаемые, поэтому может понадобиться папка с резервными копиями. Если вы обнаружите, что результаты распознавания не так хороши, как хотелось бы, или у Tesseract возникнут трудности с некоторыми символами — это хороший момент для отладки процесса, когда можно создать дополнительные обучающие данные и повторить попытку.

После того как вы создадите папку данных с box-файлами и файлами изображений, скопируйте эти данные в резервную папку, прежде чем делать с ней какие-либо дальнейшие манипуляции. Выполняя обучающие скрипты с этими данными, программа вряд ли что-нибудь удалит, однако лучше подстраховаться, чем потом сожалеть о многих часах, потраченных на создание потерянных box-файлов. Кроме того, полезно иметь возможность удалить «грязный» каталог со скомпилированными данными, чтобы повторить попытку.

Для полного анализа данных и создания обучающих файлов, необходимых для Tesseract, нужно выполнить еще полдюжины шагов. Есть инструменты, способные сделать это за вас, если предоставить им соответствующие исходные изображения и box-файлы, однако, к сожалению, на данный момент для Tesseract 3.02 таких инструментов не существует.

Я написала решение на Python (<https://github.com/REMitchell/tesseract-trainer>), которое обрабатывает файл, содержащий файлы изображений и box-файлы, и автоматически создает все файлы, необходимые для обучения.

Исходные параметры и операции, которые выполняет эта программа, содержатся в методах `__init__` и `runAll` класса:

```
def __init__(self):
    languageName = 'eng'
    fontName = 'captchaFont'
    directory = '<path to images>'

def runAll(self):
    self.createFontFile()
    self.cleanImages()
    self.renameFiles()
    self.extractUnicode()
    self.runShapeClustering()
    self.runMfTraining()
    self.runCnTraining()
    self.createTessData()
```

Здесь вам нужно определить значения всего трех простых переменных:

- **languageName** — трехбуквенный код языка, который используется в Tesseract, чтобы понять, к какому языку относится текст. В большинстве случаев вы, скорее всего, будете применять код `eng`, что означает «английский язык»;
- **fontName** — название выбранного шрифта. Это может быть что угодно, но оно должно быть одним словом без пробелов;
- **directory** — каталог, содержащий все изображения и box-файлы. Я рекомендую указать абсолютный путь; если же вы используете относительный, то он должен касаться того каталога, откуда вы запускаете код Python. В случае абсолютного пути можно запускать код из любого каталога на компьютере.

Рассмотрим используемые здесь функции.

Функция `createFontFile` создает необходимый файл `font_properties`, куда Tesseract заносит знания о новом шрифте, который мы создаем:

```
captchaFont 0 0 0 0 0
```

В этом файле содержится имя шрифта, после которого стоят единицы и нули, указывающие на то, следует ли рассматривать для данного шрифта курсив, полужирный или другие варианты начертания. (Обучение шрифтов с этими свойствами — интересное упражнение, но, к сожалению, выходит за рамки данной книги.)

Функция `cleanImages` создает высококонтрастные версии всех найденных файлов с изображениями, преобразует их в оттенки серого и выполняет другие операции, облегчающие чтение OCR-программами файлов с изображениями. Если мы имеем дело с изображениями капчи, содержащими визуальный «мусор», который легко отфильтровывается при постобработке, то ее можно добавить именно здесь.

Функция `renameFiles` присваивает всем box-файлам и соответствующим файлам изображений имена, необходимые для обработки в Tesseract (здесь числа в именах файлов — это последовательные номера, позволяющие отличать файлы друг от друга):

- `<languageName>.<fontName>.exp<fileNumber>.box;`
- `<languageName>.<fontName>.exp<fileNumber>.tiff.`

Функция `extractUnicode` просматривает все созданные box-файлы и формирует общее множество символов,

доступных для обучения. Получившийся файл в формате Unicode сообщит нам о том, сколько разных символов было найдено. Это может быть хорошим способом быстро определить, не упустили ли мы что-нибудь.

Следующие три функции, `runShapeClustering`, `runMfTraining` и `runCtTraining`, создают файлы `shapetable`, `pfhtable` и `normproto` соответственно. Все они предоставляют информацию о геометрии и форме каждого символа, а также статистическую информацию, которую Tesseract использует для вычисления вероятности того, что данный символ относится к тому или иному типу.

Наконец, Tesseract переименовывает каждый каталог со скомпилированными данными так, чтобы его имя начиналось с обозначения соответствующего языка (например, `shapetable` превратится в `eng.shapetable`), и компилирует все содержащиеся в них файлы в итоговый файл обучающих данных под названием `eng.traineddata`.

Единственное, что вам придется сделать вручную, — это переместить созданный файл `eng.traineddata` в корневой каталог `tessdata` с помощью следующих команд в Linux и Mac:

```
$cp /path/to/data/eng.traineddata  
$TESSDATA_PREFIX/tessdata
```

Если вы выполните эти операции, то у вас не должно возникнуть проблем с распознаванием капчи того типа, для которого вы обучили Tesseract. Теперь, когда вы дадите библиотеке задание прочитать изображение из нашего примера, то получите правильный ответ:

```
$ tesseract captchaExample.png output | cat  
output.txt
```

4MmC3

Это успех! Значительно лучше, по сравнению с предыдущей интерпретацией изображения как 4N\ , , , C<3.

Это был лишь краткий обзор всех возможностей обучения и распознавания шрифтов с помощью Tesseract. Если вы заинтересованы в более тщательном обучении Tesseract или, возможно, в создании собственной библиотеки обучающих файлов для распознавания капчи либо хотите подарить миру новые возможности распознавания шрифтов, то рекомендую изучить документацию по Tesseract <https://code.google.com/p/tesseract-ocr/wiki/TrainingTesseract3>.

Получение капчи и отправка решений

Многие популярные системы управления контентом часто страдают от спама, поскольку боты, запрограммированные на известное расположение их страниц регистрации пользователей, заваливают эти системы информацией о регистрации. Например, на сайте <http://pythonscraping.com> даже капча (по общему мнению, слабая) мало помогает сдержать наплыв регистрационных данных.

Как же ботам это удастся? Мы успешно справились с изображениями капчи, расположенными на локальном жестком диске, но как создать полностью функциональный бот? В этом разделе будут объединены многие приемы, описанные в предыдущих главах. Если вы еще не прочитали главу 10, то хотя бы бегло просмотрите ее.

Большинство изображений капчи обладают следующими свойствами.

- Это динамически генерируемые изображения, создаваемые программой на сервере. Их источники могут быть

непохожими на обычные изображения и иметь вид наподобие `<imgsrc="WebForm.aspx?id=8AP85CQKE9TJ">`, но их можно скачивать и обрабатывать как любые другие изображения.

- Правильный ответ на тестовое изображение хранится в базе данных на сервере.
- Многие изображения капчи имеют ограничение по времени и не позволяют думать слишком долго. Обычно для ботов это не проблема, но постановка решений капчи в очередь или другие методики, способные вызвать задержку между запросом капчи и отправкой решения, могут оказаться неудачными.

Обычно подход заключается в том, чтобы скачать файл с изображением капчи на локальный диск, очистить это изображение, использовать Tesseract для его анализа и вернуть решение в виде соответствующего параметра формы.

Я создала по адресу <http://pythonscraping.com/humans-only> страницу с формой комментариев, защищенной капчей, чтобы написанному мною боту было с чем сражаться. Он использует библиотеку Tesseract, запускаемую из командной строки, а не из оболочки pytesseract (хотя вполне можно использовать любой вариант), и выглядит так:

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup
import subprocess
import requests
from PIL import Image
```

```

from PIL import ImageOps

def cleanImage(imagePath):
    image = Image.open(imagePath)
    image = image.point(lambda x: 0 if x<143
else 255)

    borderImage =
ImageOps.expand(image,border=20,fill='white')
    borderImage.save(imagePath)

html =
urlopen('http://www.pythonscraping.com/humans-
only')
bs = BeautifulSoup(html, 'html.parser')
# Собираем заполненные перед этим значения
формы.
imageLocation = bs.find('img', {'title': 'Image
CAPTCHA'})['src']
formBuildId = bs.find('input',
{'name': 'form_build_id'})['value']
captchaSid = bs.find('input',
{'name': 'captcha_sid'})['value']
captchaToken = bs.find('input',
{'name': 'captcha_token'})['value']

captchaUrl =
'http://pythonscraping.com'+imageLocation
urlretrieve(captchaUrl, 'captcha.jpg')
cleanImage('captcha.jpg')
p = subprocess.Popen(['tesseract',
'captcha.jpg', 'captcha'], stdout=
subprocess.PIPE,stderr=subprocess.PIPE)

```

```

p.wait()
f = open('captcha.txt', 'r')

# Удаляем все пробельные символы.
captchaResponse = f.read().replace(' ', ''),
''.replace('\n', '')
print('Captcha solution attempt: '+captchaResponse)

if len(captchaResponse) == 5:
    params = {'captcha_token':captchaToken,
'captcha_sid':captchaSid,
'form_id':'comment_node_page_form',
'form_build_id': formBuildId,
'captcha_response':captchaResponse,
'name':'Ryan Mitchell',
'subject': 'I come to seek the Grail',
'comment_body[und][0][value]':
'...and I am definitely not a bot'}

r =
requests.post('http://www.pythonscraping.com/comment/reply/10',
data=params)
responseObj = BeautifulSoup(r.text,
'html.parser')
if responseObj.find('div',
{'class':'messages'}) is not None:
    print(responseObj.find('div',
{'class':'messages'}).get_text())
else:

```



```
print('There was a problem reading the  
CAPTCHA correctly!')
```

Обратите внимание: этот скрипт завершается ошибкой в двух случаях — если Tesseract не извлек из изображения ровно пять символов (поскольку мы знаем, что все допустимые решения для данной капчи должны состоять именно из пяти символов) или же если форма была отправлена, но ответ на капчу был неверным. Первое происходит примерно в 50 % случаев, и здесь не приходится заниматься отправкой формы — программа завершается с сообщением об ошибке. Второе происходит приблизительно в 20 % случаев при общей точности около 30 % (или около 80 % точности для каждого из пяти символов).

Такая точность может показаться низкой, однако имейте в виду: обычно пользователь не ограничен в количестве попыток распознать капчу и от большинства этих неправильных попыток можно отказаться, не отправляя саму форму. При отправке формы капча в большинстве случаев распознается точно. Если и это вас не убедило, то примите во внимание, что показатель точности для простого угадывания составляет 0,0000001 %. Выполнив вместо угадывания данную программу три или четыре раза, вы сэкономите время в 900 миллионов раз!

[24](#) См.: Хобсон Л., Ханнес Х., Коул Х. Обработка естественного языка в действии. — СПб.: Питер, 2020.

[25](#) При обработке текста, на котором он не обучался, Tesseract намного лучше справляется с крупноформатными изданиями книг, особенно если изображения небольшие. В следующем разделе будет показано, как обучить Tesseract распознавать различные шрифты, чтобы программа могла читать шрифты гораздо меньшего размера, в том числе при предварительном просмотре книг небольшого формата!

[26](#) Перевод на английский фразы: «умер он, а не я», где вместо «я» (I) по ошибке стоит цифра 1. — *Примеч. пер.*

[27](https://gizmodo.com/google-has-finally-killed-the-captcha-1793190374) См. <https://gizmodo.com/google-has-finally-killed-the-captcha-1793190374>.

Глава 14. Как избежать ловушек веб-скрапинга

Мало что огорчает больше, чем выполнить веб-скрапинг сайта, просмотреть результаты и обнаружить отсутствие в них данных, которые вы определенно видели в браузере. Или отправить абсолютно правильно заполненную форму, которую потом отклонит веб-сервер. Или получить сообщение о том, что ваш IP-адрес заблокирован сайтом по неизвестным причинам.

Это лишь часть самых трудных ошибок, которые приходится устранять, — не только потому, что они порой совершенно неожиданные (скрипт, отлично работавший на одном сайте, может вообще не действовать на другом, на вид практически таком же), но и потому, что эти ошибки намеренно не сопровождаются сколько-нибудь содержательными сообщениями об ошибках или данными трассировки стека, которыми можно было бы воспользоваться. Вас идентифицировали как бот, отклонили, и вы не знаете почему.

В этой книге я описала много способов, позволяющих выполнять различные сложные задачи на сайтах (отправлять формы, извлекать и очищать сложные данные, выполнять скрипты JavaScript и т.д.). Эта глава является чем-то вроде сборника в том смысле, что описанные здесь методы касаются самых разных аспектов (заголовки HTTP, стили CSS и HTML-формы — лишь некоторые из них). Однако у всех них есть кое-что общее: они призваны преодолеть препятствие, установленное с единственной целью — предотвратить автоматический веб-скрапинг сайта.

Независимо от того, насколько эта информация будет вам полезна в данный момент, я настоятельно рекомендую хотя бы

прочитать главу. Вы никогда не знаете, когда это поможет вам устранить сложную ошибку или вообще предотвратить проблему.

Этический момент

В первых нескольких главах мы затронули правовую «серую зону», где обитают веб-скраперы, а также ряд этических принципов, которыми следует руководствоваться. Честно говоря, эта глава с этической точки зрения является для меня, пожалуй, самой сложной. Мои сайты тоже страдают от ботов, спамеров, веб-скраперов и других всевозможных нежелательных виртуальных гостей — как, возможно, и ваши. Так зачем же учить людей, как сделать боты еще лучше?

Я считаю, что эту главу важно было включить в книгу по следующим причинам.

- Существуют в высшей степени этичные, юридически обоснованные причины для веб-скрапинга некоторых сайтов, не желающих, чтобы их обработал веб-скрапер. Моей предыдущей задачей веб-скрапинга был автоматический сбор информации с сайтов, которые публиковали в Интернете имена, адреса, номера телефонов и другую личную информацию клиентов без их согласия. Я использовала собранную информацию, чтобы делать официальные запросы на эти сайты с требованием удалить данную информацию. Во избежание конкуренции эти сайты бдительно охраняют свою информацию от веб-скраперов. Тем не менее моя работа по обеспечению анонимности клиентов моей компании (часть которых подвергалась преследованиям, являлась жертвой домашнего насилия или имела другие очень веские причины для того, чтобы не привлекать к себе внимания) убедительно доказала

необходимость веб-скрапинга, и я была благодарна судьбе за то, что у меня были навыки, требуемые для выполнения этой работы.

- Практически невозможно создать сайт, «защищенный от веб-скраперов» (который при этом был бы доступен для обычных пользователей), однако я все же надеюсь, что информация, изложенная в этой главе, поможет тем, кто хотел бы защитить свои сайты от вредоносных атак. Я постоянно буду указывать на слабые стороны каждого метода веб-скрапинга, которые вы сможете применять для защиты своего сайта. Имейте в виду: большинство сетевых ботов сегодня просто широко собирают информацию и находят уязвимости, так что использование всего нескольких простых методов, описанных в данной главе, скорее всего, станет помехой для 99 % из них. Тем не менее эти боты с каждым месяцем становятся все более изощренными, и лучше быть готовыми.
- Как и большинство программистов, я не верю, что отказ от изучения какой-либо информации — это хорошо.

Читая эту главу, имейте в виду: многие из описанных здесь скриптов вовсе не обязательно использовать для каждого сайта, который вам встретится. Поступать так не просто нехорошо — вы можете получить письмо-предупреждение о возможных санкциях или кое-что похуже (подробнее о том, что делать, если вы все же получите такое письмо, см. в главе 18). Но я не собираюсь тыкать вас в это носом всякий раз, когда мы будем обсуждать новую методику. Поэтому отложим данный вопрос до конца главы.

Выдать скрипт за человека

Фундаментальная проблема для сайтов, которые хотят избежать веб-скрапинга, состоит в том, чтобы выяснить, как отличать боты от людей. Несмотря на то что многие методы, используемые сайтами (например, капчи), сложно обмануть, все же есть несколько довольно простых вещей, позволяющих вашему боту стать более похожим на человека.

Настройте заголовки

На протяжении всей этой книги мы использовали Python-библиотеку Requests для создания, отправки и получения HTTP-запросов, таких как обработка форм на сайте в главе 10. Библиотека Requests отлично подходит и для определения заголовков. HTTP-заголовки — это списки атрибутов или предпочтений, которые передаются на веб-сервер вместе с каждым запросом. В протоколе HTTP определено несколько десятков странных типов заголовков, большинство из которых обычно не используется. Однако следующие семь полей постоянно применяются в большинстве основных браузеров при инициализации любого соединения (ниже показан пример данных из моего собственного браузера):

```
Host           https://www.google.com/
Connection     keep-alive
Accept         text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent     Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
```

```
39.0.2171.95 Safari/537.36
Referrer      https://www.google.com/
Accept-Encoding gzip, deflate, sdch
Accept-Language en-US,en;q=0.8
```

А вот заголовки, которые отправляет по умолчанию типичный веб-скрапер Python, написанный с помощью библиотеки `urllib`:

```
Accept-Encoding identity
User-Agent      Python-urllib/3.4
```

Если бы вы были администратором сайта и хотели бы заблокировать веб-скраперы, то какой из этих заголовков пропустили бы с большей вероятностью?

К счастью, заголовки можно полностью изменить с помощью библиотеки `Requests`. Для тестирования свойств браузера, доступных для просмотра веб-сервером, отлично подходит сайт **<https://www.whatismybrowser.com>**. Чтобы проверить настройки cookie, мы выполним веб-скрапинг этого сайта, применив следующий скрипт:

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()
headers = {'User-Agent': 'Mozilla/5.0
(Macintosh; Intel Mac OS X 10_9_5)
AppleWebKit 537.36 (KHTML, like
Gecko) Chrome',
          'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8'}
```

```
url = 'https://www.whatismybrowser.com/' \
      'developers/what-http-headers-is-my-' \
      'browser-sending'
req = session.get(url, headers=headers)
bs = BeautifulSoup(req.text, 'html.parser')
print(bs.find('table', {'class': 'table-' \
                        'striped'}).get_text)
```

Результат выполнения этого скрипта должен показать, что теперь заголовки соответствуют тем, которые заданы в коде, в словарном объекте `headers`.

Хотя сайты способны проверять, является ли посетитель человеком, на основе любых свойств в заголовках HTTP, я обнаружила, что, как правило, единственный параметр, который действительно имеет значение, — это `User-Agent`. Поэтому, над каким бы проектом вы ни работали, имеет смысл присвоить ему какое-нибудь менее подозрительное значение, чем `Python-urllib/3.4`. Кроме того, если вы когда-либо столкнетесь с чрезвычайно бдительным сайтом, то заполнение одного из часто используемых, но редко проверяемых заголовков, таких как `Accept-Language`, может оказаться решающим условием для того, чтобы вас приняли за человека.

То, как вы видите мир, зависит от заголовков

Предположим, вы хотите написать основанную на машинном обучении программу-переводчик для исследовательского проекта и для ее тестирования вам нужны большие объемы переведенных текстов. Многие крупные сайты предоставляют различные переводы одного и того же контента, в зависимости от языковых предпочтений, указанных в

заголовках. Просто изменив в заголовке `Accept-Language:en-US` на `AcceptLanguage:fr`, вы можете получить перевод на французском, если сайт поддерживает такую возможность (практически беспроблемным вариантом являются крупные международные компании).

Заголовки также могут подсказать сайту, что следует изменить формат представляемого ими контента. Например, при доступе в Интернет с мобильных устройств часто предоставляются урезанные версии сайтов, без рекламных баннеров, Flash-анимации и других отвлекающих элементов. Если вы попытаетесь заменить значение параметра `User-Agent` примерно на следующее, то, возможно, обнаружите, что задача веб-скрапера немного упростится!

```
User-Agent:Mozilla/5.0 (iPhone; CPU iPhone  
OS 7_1_2 like Mac OS X)  
AppleWebKit/537.51.2 (KHTML, like Gecko)  
Version/7.0 Mobile/11D257  
Safari/9537.53
```

Обработка данных cookie с помощью JavaScript

Правильная обработка данных cookie может смягчить многие проблемы веб-скрапинга, хотя у этой медали есть и обратная сторона. Сайты, которые отслеживают ваши перемещения по сайту с помощью данных cookie, могут попытаться выбросить с сайта веб-скрапер, если он будет демонстрировать ненормальное для человека поведение — например, слишком быстро заполнять формы или посещать слишком много страниц. Конечно, такое поведение можно замаскировать,

разорвав и снова установив соединение с сайтом или даже изменив свой IP-адрес (подробнее о том, как это сделать, см. в главе 17), однако если данные cookie будут выдавать вас с головой, то все попытки маскировки окажутся тщетными.

Данные cookie также могут оказаться необходимыми для веб-скрапинга сайта. Как показано в главе 10, чтобы оставаться на сайте, следует иметь возможность сохранять данные cookie и представлять их, переходя на новую страницу. Некоторые сайты даже не требуют явной аутентификации и получения каждый раз новых данных cookie — достаточно, посещая сайт, сохранить старую копию данных cookie, полученных после аутентификации.

При веб-скрапинге только одного или небольшого количества определенных сайтов я рекомендую изучить данные cookie, генерируемые этими сайтами, и подумать, какие из них должен поддерживать ваш веб-скрапер. Есть разные подключаемые модули браузеров, которые показывают, как устанавливаются данные cookie при посещении сайта и перемещении по нему. Один из моих любимых таких модулей — расширение Chrome EditThisCookie (<http://www.editthiscookie.com/>).

Для получения дополнительной информации об обработке данных cookie с помощью библиотеки Requests еще раз изучите примеры кода в разделе «Обработка данных авторизации и параметров cookie» в главе 10 на с. 191. Конечно, из-за невозможности выполнить JavaScript библиотека Requests не в состоянии обрабатывать многие данные cookie, создаваемые современными программами слежения наподобие Google Analytics. Такие данные cookie создаются только после выполнения скриптов на стороне клиента (или иногда на основании событий страницы, таких как нажатия кнопок при просмотре страницы). Чтобы справиться с этим, нужно

использовать пакеты Selenium и Chrome WebDriver (их установку и основы применения мы рассмотрели в главе 11).

Чтобы просмотреть данные cookie, можно посетить любой сайт (в данном примере <http://pythonscraping.com>) и вызвать в веб-драйвере функцию `get_cookies()`:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
chrome_options = Options()
chrome_options.add_argument('--headless')
driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)
driver.get('http://pythonscraping.com')
driver.implicitly_wait(1)
print(driver.get_cookies())
```

В результате получим вполне типичный для Google Analytics массив cookie:

```
[{'value': '1', 'httponly': False, 'name': '_gat', 'path': '/', 'expiry': 1422806785, 'expires': 'Sun, 01 Feb 2015 16:06:25 GMT', 'secure': False, 'domain': '.pythonscraping.com'}, {'value': 'GA1.2.1619525062.1422806186', 'httponly': False, 'name': '_ga', 'path': '/', 'expiry': 1485878185, 'expires': 'Tue, 31 Jan 2017 15:56:25 GMT', 'secure': False, 'domain': '.pythonscraping.com'}, {'value': '1', 'httponly': False, 'name': 'has_js', 'path': '/', 'expiry': 1485878185, 'expires': 'Tue, 31
```

```
Jan 2017 15:56:25 GMT', 'secure': False,  
'domain': 'pythonscraping.com'}]]
```

Для управления данными cookie можно воспользоваться функциями `delete_cookie()`, `add_cookie()` и `delete_all_cookies()`. Кроме того, можно сохранять данные cookie для применения в других веб-скраперах. Вот пример того, как совместно использовать эти функции:

```
from selenium import webdriver  
from selenium.webdriver.chrome.options import  
Options
```

```
chrome_options = Options()  
chrome_options.add_argument('--headless')
```

```
driver = webdriver.Chrome(  
    executable_path='drivers/chromedriver',  
    options=chrome_options)  
driver.get('http://pythonscraping.com')  
driver.implicitly_wait(1)
```

```
savedCookies = driver.get_cookies()  
print(savedCookies)
```

```
driver2 = webdriver.Chrome(  
    executable_path='drivers/chromedriver',  
    options=chrome_options)
```

```
driver2.get('http://pythonscraping.com')  
driver2.delete_all_cookies()  
for cookie in savedCookies:
```

```
driver2.add_cookie(cookie)

driver2.get('http://pythonscraping.com')
driver.implicitly_wait(1)
print(driver2.get_cookies())
```

В этом примере первый веб-драйвер получает сайт, выводит данные cookie, а затем сохраняет их в переменной savedCookies. Второй веб-драйвер загружает тот же сайт, удаляет собственные данные cookie и добавляет полученные от первого веб-драйвера.

Обратите внимание: второй веб-драйвер, прежде чем добавить данные cookie, должен сначала загрузить сайт. Таким образом, Selenium знает, к какому домену принадлежат данные cookie, даже если загрузка сайта ничего не дала веб-скраперу.

После этого второй веб-драйвер должен получить те же данные cookie, что и первый. Согласно Google Analytics теперь второй веб-драйвер идентичен первому и они будут отслеживаться одинаково. Если первый прошел аутентификацию на сайте, то второй тоже будет аутентифицирован.

Своевременность — наше все

Некоторые хорошо защищенные сайты могут препятствовать отправке форм или другому взаимодействию с сайтом, если пользователь делает это слишком быстро. Но даже при отсутствии подобных мер безопасности все равно скачивать с сайта слишком много информации и делать это значительно быстрее, чем может обычный человек, — хороший способ добиться того, чтобы вас заметили и заблокировали.

Многопоточное программирование — отличный способ загружать страницы быстрее, так что в одном потоке обрабатываются данные, а в другом потоке многократно загружаются страницы. Однако это кошмарный метод, если вы хотите написать хороший веб-скрапер. Всегда следует стремиться загружать как можно меньше отдельных страниц и делать минимум запросов данных. По возможности старайтесь растянуть этот процесс на несколько секунд, даже если придется добавить в код такие строки:

```
import time

time.sleep(3)
```

Нужны ли вам эти дополнительные несколько секунд между загрузками страниц, часто определяется экспериментально. Мне неоднократно приходилось с боем выцарапывать данные с сайта, каждые несколько минут доказывая, что «я не робот» (проходя тест капчи вручную и вставляя полученные данные cookie обратно в веб-скрапер, чтобы тот продолжал работу с сайтом как «доказавший, что он человек»). И лишь добавив `time.sleep`, я решила свои проблемы, и сайт позволил мне выполнять веб-скрапинг до бесконечности.

Иногда приходится замедляться, чтобы двигаться быстрее!

Основные средства защиты форм

Есть много проверенных тестов, вот уже многие годы более или менее надежно позволяющих отличить веб-скрапер от человека, использующего браузер. Не проблема, если бот скачивает статьи и публикации в блогах, которые все равно являются общедоступными. Проблема, когда бот создает тысячи пользовательских учетных записей и начинает

рассылать спам всем, кто зарегистрирован на вашем сайте. Веб-формы, особенно связанные с созданием учетных записей и аутентификацией, представляют значительную угрозу безопасности и вычислительным ресурсам, если уязвимы для произвольного использования ботами. Поэтому владельцы сайтов в высшей степени заинтересованы постараться ограничить доступ к их сайту (или по крайней мере им кажется, что доступ ограничен).

Эти меры защиты от ботов, основанные на формах и аутентификации, могут стать серьезной проблемой для веб-скраперов.

Примите во внимание, что в этом обзоре кратко рассмотрена лишь часть мер безопасности, с которыми вы можете столкнуться при создании автоматических ботов для этих форм. Подробнее о взаимодействии с хорошо защищенными формами см. в главе 13, посвященной капче и обработке изображений, а также в главе 17, где описывается работа с заголовками и IP-адресами.

Значения скрытых полей ввода

Скрытые поля в HTML-формах содержат значения, видимые для браузера, но не для пользователя (если только он не заглянет в исходный код сайта). По мере роста популярности данных cookie для хранения и передачи переменных между страницами сайта скрытые поля на какое-то время перестали использоваться, однако затем у них обнаружилось еще одно замечательное назначение: не позволять веб-скраперам отправлять формы.

На рис. 14.1 показан пример применения таких скрытых полей на странице входа в Facebook. У этой формы всего три видимых поля (**Username** (Имя пользователя), **Password** (Пароль)

и кнопка **Submit** (Отправить)), но много скрытых, через которые на сервер передается много информации.

Скрытые поля используются для предотвращения веб-скрапинга двумя основными способами: поле может заполняться значением переменной, случайно сгенерированным на странице формы, и его же сервер ожидает получить на странице обработки формы. Если в форме такое значение отсутствует, то сервер разумно предполагает, что данные не были отправлены естественным путем со страницы формы, а переданы ботом непосредственно на страницу обработки. Лучший способ обойти это препятствие — сначала выполнить веб-скрапинг страницы формы, получить случайно сгенерированную переменную и уже оттуда передать данные формы на страницу обработки.

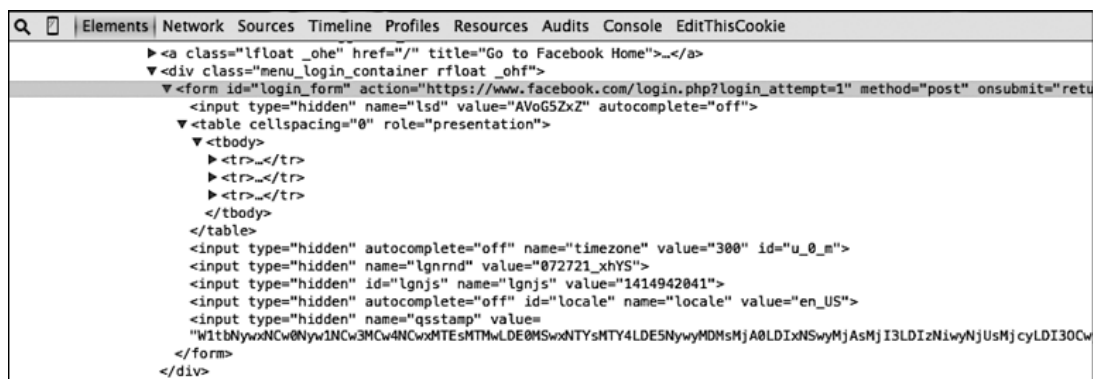


Рис. 14.1. Форма аутентификации в Facebook содержит несколько скрытых полей

Второй способ — своего рода поля-приманки. Если форма содержит скрытое поле с безобидным именем вроде «Имя пользователя» или «Адрес электронной почты», то плохо написанный бот может заполнить это поле и попытаться его отправить, независимо от того, является оно скрытым или нет. Любые скрытые поля с реальными значениями (отличными от предлагаемых по умолчанию на странице отправки формы)

следует игнорировать, иначе пользователя могут даже заблокировать на сайте.

Одним словом, иногда необходимо проверить страницу, на которой находится форма, чтобы увидеть, не пропустили ли вы что-нибудь такое, чего может ожидать сервер. Если вы видите несколько скрытых полей зачастую с длинными, случайно сгенерированными строковыми переменными, то, скорее всего, веб-сервер будет проверять их наличие при отправке формы. Или же может проверять, были ли эти переменные сгенерированы недавно с целью гарантировать однократное использование переменных формы (чтобы не позволить веб-скраперу просто сохранять эти значения в скрипте и время от времени снова их использовать). Или и то и другое.

Как справляться с полями-приманками

С одной стороны, CSS часто значительно упрощает жизнь (например, путем чтения тегов `id` и `class`), но, с другой стороны, когда нужно отличить полезную информацию от бесполезной, веб-скраперу бывает трудно это сделать. Если поле веб-формы скрыто от пользователя с помощью CSS, то разумно предположить, что обычный пользователь, посещающий сайт, не может заполнить данное поле, поскольку оно не отображается в браузере. Если же форма все-таки заполнена, то, скорее всего, это признак деятельности бота, и сообщение будет отклонено.

Это относится не только к формам, но и к ссылкам, изображениям, файлам и другим элементам сайта, которые могут быть прочитаны ботом, но скрыты от обычных пользователей, посещающих сайт через браузер. Посещение страницы сайта со скрытой ссылкой на нее вполне может запустить серверный скрипт, который заблокирует IP-адрес пользователя, аннулирует его аутентификацию на сайте или

предпримет какие-либо другие действия, чтобы предотвратить дальнейший доступ данного пользователя к сайту. В сущности, именно на этой концепции основаны многие бизнес-модели.

Возьмем, к примеру, страницу, расположенную по адресу **<http://pythonscraping.com/pages/itsatrap.html>**. Эта страница содержит две ссылки: одна из них скрыта средствами CSS, а вторая является видимой. Кроме того, здесь есть форма с двумя скрытыми полями:

```
<html>
<head>
    <title>A bot-proof form</title>
</head>
<style>
    body {
        overflow-x:hidden;
    }
    .customHidden {
        position:absolute;
        right:50000px;
    }
</style>
<body>
    <h2>A bot-proof form</h2>
    <a href=
        "http://pythonscraping.com/dontgohere"
        style="display:none;">Go here!</a>
    <a href="http://pythonscraping.com">Click
me!</a>
    <form>
        <input type="hidden" name="phone"
value="valueShouldNotBeModified"/><p/>
```

```
        <input type="text" name="email"
class="customHidden"
        value="intentionallyBlank"/>
</p>
        <input type="text" name="firstName"/>
</p>
        <input type="text" name="lastName"/>
</p>
        <input type="submit" value="Submit"/>
</p>
    </form>
</body>
</html>
```

Эти три элемента скрыты от пользователя следующими тремя способами:

- первая ссылка скрыта просто с помощью атрибута CSS `display:none`;
- поле `phone` является скрытым полем ввода;
- поле `email` невидимое, так как смещено на 50 000 пикселей вправо (предположительно за пределы экрана любого монитора); полоса прокрутки, благодаря которой это могло бы быть заметно, тоже скрыта.

К счастью, поскольку Selenium отображает помещаемые страницы, это позволяет отличать элементы, визуальное присутствие которых на странице, от тех, которые не видны. Наличие элемента на странице можно определить с помощью функции `is_displayed()`.

Например, следующий код извлекает описанную ранее страницу, находит скрытые ссылки и поля ввода форм:

```
from selenium import webdriver
from selenium.webdriver.remote.webelement
import WebElement
from selenium.webdriver.chrome.options import
Options

driver = webdriver.Chrome(
    executable_path='drivers/chromedriver',
    options=chrome_options)
driver.get('http://pythonscraping.com/pages/its
atrap.html')
links = driver.find_elements_by_tag_name('a')
for link in links:
    if not link.is_displayed():
        print('The link {} is a
trap'.format(link.get_attribute('href')))

fields =
driver.find_elements_by_tag_name('input')
for field in fields:
    if not field.is_displayed():
        print('Do not change value of
{}'.format(field.get_attribute('name')))
```

Selenium находит все скрытые поля и выводит следующие данные:

```
The link http://pythonscraping.com/dontgohere
is a trap
Do not change value of phone
```

Do not change value of email

Скорее всего, вы не захотите переходить по обнаруженным скрытым ссылкам, однако, отправляя форму, стоит убедиться в правильности заполнения скрытых полей (или в их заполнении Selenium). Резюмируем: просто игнорировать скрытые поля опасно, хотя следует быть осторожными при взаимодействии с ними.

Контрольный список: как выдать программу за человека

В данной главе, да и вообще во всей книге, много говорится о способах разработки веб-скрапера, который будет как можно меньше похож на него и как можно больше — на человека. Если, несмотря на это, вас все равно блокируют сайты и вы не знаете почему, то вот контрольный список, который можно использовать для устранения проблемы.

- Прежде всего, если страница, которую вы получаете с веб-сервера, пуста, на ней отсутствует информация или она каким-либо иным образом не соответствует вашим ожиданиям (или увиденному вами в браузере), — это, вероятно, вызвано тем, что для создания данной страницы выполняется скрипт JavaScript. Обратитесь к главе 11.
- Отправляя на сайт форму или POST-запрос, просмотрите страницу и убедитесь в отправке всего, что сайт ожидает от вас, причем в правильном формате. Чтобы увидеть, какой POST-запрос в действительности отправляется на сайт, и убедиться, что в нем есть все необходимое и «естественный» запрос выглядит идентично тому, который отправляет ваш бот, используйте соответствующий инструмент, такой как панель Chrome Inspector.

- Если при попытке аутентифицироваться на сайте вам не удастся там «закрепиться» или сайт себя ведет как-то странно, то проверьте данные cookie. Убедитесь, что они правильно сохраняются перед каждой загрузкой страницы и ваши данные cookie передаются на сайт при каждом запросе.
- Если вы получаете от клиента ошибки HTTP, особенно ошибку 403 Forbidden (доступ запрещен), то это может означать, что сайт идентифицировал ваш IP-адрес как адрес бота и не желает принимать какие-либо дополнительные запросы. Нужно либо подождать, пока ваш IP-адрес будет удален из списка, либо получить новый IP-адрес (например, сходить в ближайшее кафе или заглянуть в главу 17). Чтобы убедиться в отсутствии блокировок, попробуйте сделать следующее.
- Убедитесь, что веб-скраперы не перемещаются по сайту слишком быстро. Быстрый веб-скрапинг — порочная практика. Она ложится тяжелым бременем на серверы веб-администратора, может привести к юридическим проблемам и является главной причиной попадания веб-скраперов в черный список. Добавьте в ваши веб-скраперы задержки и оставьте их работать на ночь. Помните: писать программы или собирать данные в спешке — признак плохого управления проектами; стройте планы заранее, в первую очередь чтобы избежать подобной суматохи.
- Самое очевидное: смените заголовки! Некоторые сайты блокируют все, что объявляет себя веб-скрапером. Если вы не знаете точно, какие значения заголовков стоит использовать, то скопируйте заголовки из браузера.

- Убедитесь, что не нажимаете что-то и не получаете доступ к чему-либо, обычно недоступному человеку (подробнее об этом см. в подразделе «Как избежать ловушек» раздела «Основные средства защиты форм» на с. 265).
- Если обнаружится, что для получения доступа вам требуется слишком много «танцев с бубном», то посмотрите, нельзя ли связаться с администратором сайта, объяснить ему ваши действия и получить разрешение на использование веб-скраперов. Попробуйте отправить письмо по адресу `webmaster@<имядомена>` или `admin@<имядомена>`. Администраторы тоже люди, и вы будете удивлены тем, насколько они готовы делиться данными.

Глава 15. Тестирование сайтов с помощью веб-скраперов

При работе с веб-проектами, в которых используется большой стек технологий, как правило, регулярно тестируется только серверная часть стека. У большинства современных языков программирования (включая Python) есть тот или иной фреймворк для тестирования, но фронтенд сайтов часто остается за пределами этих автоматических тестов, хотя именно он обычно является единственной частью проекта, которую видит клиент.

Проблема отчасти состоит в том, что сайты, как правило, представляют собой смесь многих языков разметки и программирования. Можно написать юнит-тесты для разделов JavaScript, но эти тесты будут бесполезны, если HTML-код, с которым взаимодействует JavaScript, изменится таким образом, что JavaScript не будет выполнять предполагаемое действие на странице даже при правильной работе самого скрипта.

Задачу тестирования клиентской части сайтов часто оставляют напоследок или делегируют программистам более низкого уровня, вооруженным в лучшем случае контрольным списком того, что нужно проверить, и средством для отслеживания ошибок. Но если заблаговременно приложить немного больше усилий, то можно заменить этот контрольный список серией юнит-тестов, а человеческий глаз — веб-скрапером.

Вы только представьте: веб-разработка через тестирование! Ежедневные тесты, позволяющие убедиться, что все части веб-интерфейса работают должным образом. Каждый раз, когда кто-то добавляет на сайт новую функцию или меняет

положение элемента, запускается набор тестов. В этой главе мы рассмотрим основы тестирования и узнаем, как тестировать всевозможные виды сайтов, от простых до сложных, с помощью веб-скраперов на основе Python.

Основы тестирования

Если вам прежде не приходилось писать тесты для кода, то самое время начинать. Набор тестов, который можно запустить, чтобы убедиться в должной работе кода (по крайней мере, настолько, насколько вы написали тесты), экономит время и нервы, а также упрощает выпуск обновлений.

Что такое юнит-тесты

Термины «*тесты*» и «*юнит-тесты*» (иногда называют *модульными*) нередко считают взаимозаменяемыми. Часто, когда программисты говорят о «написании тестов», они действительно имеют в виду написание юнит-тестов. Но есть и такие, которые, говоря о написании юнит-тестов, на самом деле пишут какие-то другие.

Определения и методы юнит-тестирования часто меняются от компании к компании, однако такое тестирование обычно характеризуется следующими общими свойствами.

- Каждый юнит-тест проверяет один аспект функциональности компонента. Например, он может проверять, выдается ли соответствующее сообщение об ошибке при попытке снять с банковского счета отрицательное количество долларов.

Юнит-тесты часто группируются в один класс для того компонента, который тестируют. Например, после теста на

отрицательное значение суммы в долларах, снятых с банковского счета, может идти юнит-тест поведения банковского счета при попытке снять сумму, превышающую остаток.

- Каждый юнит-тест может проводиться совершенно независимо от других. Все настройки или отмены настройки, требуемые для юнит-теста, должен совершать он сам. Аналогично юнит-тесты не должны влиять на успешное или неудачное прохождение других тестов и должны иметь возможность успешно выполняться в любой последовательности.
- Каждый юнит-тест обычно содержит хотя бы одно *утверждение*. Например, юнит-тест может утверждать, что $2 + 2$ равно 4. Иногда такой тест содержит только состояние неудачи. Например, может завершиться неудачно при выдаче исключения, но выполниться по умолчанию, если все идет хорошо.
- Юнит-тесты отделены от основной части кода. Они обязательно должны импортировать и использовать тестируемый код, однако обычно тесты хранятся в отдельных классах и каталогах.

Есть много других типов тестов — например, комплексные и контрольные, — однако в этой главе основное внимание уделяется юнит-тестированию. Такие тесты не просто стали чрезвычайно популярными благодаря последним тенденциям разработки на основе тестирования; длина кода и гибкость этих тестов облегчают взаимодействие с ними в качестве примеров, а у Python есть ряд встроенных возможностей для

юнит-тестирования, о которых вы узнаете в следующем разделе.

Python-модуль unittest

Библиотека юнит-тестирования Python под названием `unittest` входит в комплект всех стандартных инсталляционных пакетов Python. Достаточно импортировать и расширить класс `unittest.TestCase`, и у вас появятся следующие возможности:

- функции `setUp` и `tearDown`, которые выполняются до и после каждого юнит-теста;
- несколько типов операторов утверждений, описывающих условия прохождения или непрохождения тестов;
- возможность выполнять любые функции, имена которых начинаются с `test_`, как у юнит-тестов, и игнорировать функции, не объявленные тестами.

Вот пример простого юнит-теста, позволяющего проверить, что в Python $2 + 2 = 4$:

```
import unittest

class TestAddition(unittest.TestCase):
    def setUp(self):
        print('Setting up the test')

    def tearDown(self):
        print('Tearing down the test')
```

```
def test_twoPlusTwo(self):
    total = 2+2
    self.assertEqual(4, total);

if __name__ == '__main__':
    unittest.main()
```

Несмотря на то что функции `setUp` и `tearDown` здесь не совершают никаких полезных действий, они все же включены в код в иллюстративных целях. Обратите внимание: эти функции выполняются до и после каждого теста, а не до и после всех тестов класса.

Результат выполнения тестовой функции при запуске из командной строки должен выглядеть так:

```
Setting up the test
Tearing down the test
.
-----
-----
Ran 1 test in 0.000s

OK
```

Это указывает на то, что тест пройден успешно и $2 + 2$ действительно равно 4.

Выполнение unittest в Jupyter Notebook

Все сценарии юнит-тестов в этой главе запускаются одинаково:

```
if __name__ == '__main__':  
    unittest.main()
```

Условие `if __name__ == '__main__'` истинно только в том случае, если данная строка выполняется непосредственно в Python, а не с помощью оператора `import`. Это позволяет запускать юнит-тест, используя расширение класса `unittest.TestCase`, непосредственно из командной строки.

В Jupyter Notebook все немного иначе. Параметры `argv`, создаваемые Jupyter, могут вызывать ошибки в юнит-тестах, и, поскольку фреймворк `unittest` по умолчанию завершает работу Python после выполнения теста (что вызывает проблемы в ядре Notebook), это необходимо предотвратить.

В Jupyter Notebook мы будем запускать юнит-тесты так:

```
if __name__ == '__main__':  
    unittest.main(argv=[''], exit=False)  
    %reset
```

Во второй строке всем переменным `argv` (аргументам командной строки) присваиваются значения в виде пустых строк, которые `unittest.main` игнорирует. Таким образом предотвращается еще и завершение работы `unittest` после выполнения теста.

Строка `%reset` нужна для того, чтобы освободить память и уничтожить все переменные, созданные пользователем в Jupyter Notebook. Без нее каждый юнит-тест, написанный в Notebook, будет содержать все методы всех ранее

выполненных тестов, которые тоже являются наследниками `unittest.TestCase`, включая методы `setUp` и `tearDown`. Это также означает, что каждый следующий юнит-тест будет запускать все методы из предыдущих аналогичных тестов!

Однако использование `%reset` говорит о том, что пользователь при выполнении тестов должен будет совершить еще одну операцию. При запуске теста Notebook выведет подсказку и спросит у пользователя, уверен ли он в своем желании освободить память. Чтобы это сделать, нужно просто ввести `y` и нажать **Enter**.

Тестирование «Википедии». Чтобы протестировать клиентский интерфейс вашего сайта (за исключением скриптов JavaScript, о которых я расскажу далее), надо всего лишь объединить Python-библиотеку `unittest` с веб-скрапером:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest

class TestWikipedia(unittest.TestCase):
    bs = None
    def setUpClass():
        url =
        'http://en.wikipedia.org/wiki/Monty_Python'
        TestWikipedia.bs =
        BeautifulSoup(urlopen(url), 'html.parser')

    def test_titleText(self):
```

```

        pageTitle =
TestWikipedia.bs.find('h1').get_text()
        self.assertEqual('Monty Python',
pageTitle);

    def test_contentExists(self):
        content = TestWikipedia.bs.find('div',
{'id': 'mw-content-text'})
        self.assertIsNotNone(content)

if __name__ == '__main__':
    unittest.main()

```

На этот раз у нас два теста: первый проверяет, соответствует ли заголовок страницы ожидаемому Monty Python, а второй — есть ли на странице элемент div с контентом.

Обратите внимание: контент страницы загружается только один раз, и глобальный объект bs используется тестами совместно. Это возможно благодаря тому, что в unittest определена функция setUpClass, которая выполняется только один раз в начале класса (в отличие от setUp, запускаемой перед каждым тестом в отдельности). Применяя setUpClass вместо setUp, мы исключаем лишние загрузки страницы; можно получить контент один раз и провести для него несколько тестов.

Помимо того, когда и как часто выполняются эти функции, между setUpClass и setUp есть еще одно фундаментальное архитектурное различие: setUpClass — статический метод, который «принадлежит» самому классу и использует глобальные переменные класса, а setUp — функция экземпляра класса, и она принадлежит конкретному экземпляру класса. Именно поэтому setUp может

устанавливать атрибуты для `self` — конкретного экземпляра класса, тогда как `setUpClass` может обращаться только к статическим атрибутам класса `TestWikipedia`.

Тестирование каждой страницы в отдельности может показаться не таким уж мощным или интересным, однако, как вы, вероятно, помните из главы 3, сравнительно легко построить веб-краулер, который бы итеративно перемещался по всем страницам сайта. Что произойдет, если объединить веб-краулер с юнит-тестом, проверяющим одну страницу?

Есть много способов запускать тесты многократно, но это следует делать осторожно, чтобы загружать каждую страницу только один раз для каждого набора тестов, которые вы хотите выполнить на этой странице. Кроме того, следует по возможности не хранить в памяти слишком много информации одновременно. Именно так работает следующая функция:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest
import re
import random
from urllib.parse import unquote

class TestWikipedia(unittest.TestCase):

    def test_PageProperties(self):
        self.url =
        'http://en.wikipedia.org/wiki/Monty_Python'
        # протестировать первые 10 попавшихся
        страниц
        for i in range(1, 10):
```



```

        self.bs =
BeautifulSoup(urlopen(self.url), 'html.parser')
        titles = self.titleMatchesURL()
        self.assertEqual(titles[0],
titles[1])
        self.assertTrue(self.contentExists(
))
        self.url = self.getNextLink()
        print('Done!')

    def titleMatchesURL(self):
        pageTitle =
self.bs.find('h1').get_text()
        urlTitle =
self.url[(self.url.index('/wiki/')+6):]
        urlTitle = urlTitle.replace('_', ' ')
        urlTitle = unquote(urlTitle)
        return [pageTitle.lower(),
urlTitle.lower()]

    def contentExists(self):
        content = self.bs.find('div',{'id':'mw-
content-text'})
        if content is not None:
            return True
        return False

    def getNextLink(self):
        # возвращает случайную ссылку,
найденную на странице,
        # используя методику из главы 3

```

```

        links = self.bs.find('div',
{'id': 'bodyContent'}).find_all(
        'a', href=re.compile('^(/wiki/)
((?!:).)*$'))

        randomLink =
random.SystemRandom().choice(links)

        return
'https://wikipedia.org{}'.format(randomLink.att
rs['href'])

if __name__ == '__main__':
    unittest.main()

```

Есть несколько моментов, на которые стоит обратить внимание. Во-первых, в этом классе содержится только один тест. Остальные функции с технической точки зрения являются лишь вспомогательными, даже если проделывают основную часть вычислительной работы с целью определить, был ли тест пройден без ошибок. Поскольку тестовая функция выполняет операторы утверждений, результаты теста возвращаются в тестовую функцию, в которой выполняются утверждения.

Кроме того, хоть `contentExists` и возвращает логическое значение, функция `titleMatchesURL` возвращает сами значения для оценки. Чтобы понять, зачем возвращать сами значения, а не только логику, сравним результаты следующего логической проверки:

```

=====
=====
FAIL:                                     test_PageProperties
(__main__.TestWikipedia)
-----
-----

```

```
Traceback (most recent call last):
  File "15-3.py", line 22, in
test_PageProperties
    self.assertTrue(self.titleMatchesURL())
AssertionError: False is not true
```

с результатами выполнения оператора assertEquals:

```
=====
=====
FAIL:                                     test_PageProperties
(__main__.TestWikipedia)
-----
-----
Traceback (most recent call last):
  File "15-3.py", line 23, in
test_PageProperties
    self.assertEqual(titles[0], titles[1])
AssertionError: 'lockheed u-2' != 'u-2 spy
plane'
```

Какой из них легче отлаживать? (В данном случае ошибка возникает из-за перенаправления, когда публикация <http://wikipedia.org/wiki/u-2%20spy%20plane> перенаправляет на статью под названием Lockheed U-2.)

Тестирование с помощью Selenium

JavaScript создает определенные проблемы не только при веб-скрапинге скриптов Ajax, описанном в главе 11, но и при тестировании сайтов. К счастью, у Selenium есть отличный фреймворк для работы с особенно сложными сайтами;

собственно говоря, эта библиотека изначально была создана именно для тестирования сайтов!

Несмотря на то что юнит-тесты на Python и на Selenium, безусловно, написаны на одном языке, в их синтаксисе на удивление мало общего. Selenium не требует представления юнит-тестов в виде функций внутри классов; здесь операторы `assert` не требуют круглых скобок, а тесты выполняются без каких-либо сообщений, за исключением сообщений о сбое:

```
driver = webdriver.Chrome()  
driver.get('http://en.wikipedia.org/wiki/Monty_  
Python')  
assert 'Monty Python' in driver.title  
driver.close()
```

При выполнении этого теста результаты выводиться не должны.

Таким образом, тесты на Selenium можно писать более небрежно, чем юнит-тесты на Python, и операторы `assert` можно интегрировать в обычный код, если вы хотите, чтобы действие кода прекращалось, когда не выполняется какое-либо условие.

Взаимодействие с сайтом

Недавно я захотела связаться с небольшой местной компанией через контактную форму на их сайте, но обнаружила повреждение HTML-формы; когда я нажимала кнопку отправки, ничего не происходило. Проведя небольшое расследование, я обнаружила, что на сайте применялась простая форма с отправкой данных по электронной почте — каждый раз при заполнении формы отправлялось электронное письмо с контентом формы. К счастью, зная это, я сумела

отправить представителям компании электронное письмо, в котором объяснила суть проблемы с их формой, и все же воспользовалась их услугами, несмотря на техническую проблему.

Если бы я захотела написать традиционный веб-скрапер, который бы использовал или тестировал эту форму, то он, скорее всего, просто скопировал бы ее разметку и отправил электронное письмо напрямую, вообще минуя данную форму. Как же проверить функциональность формы и убедиться, что она правильно работает через браузер?

В предыдущих главах мы уже обсуждали навигацию по ссылкам, отправку форм и другие виды интерактивных действий, однако все наши действия в своей основе были предназначены для того, чтобы *обойти* интерфейс браузера, а не использовать его. Но ведь Selenium позволяет именно вводить текст, нажимать кнопки и делать все остальное в браузере (в данном случае в Chrome в режиме консоли) и обнаруживать такие вещи, как неправильно написанные формы, плохой код JavaScript, опечатки в HTML-коде и другие проблемы, которые бы поставили в тупик обычных посетителей сайта.

Главное условие в тестах подобного рода — концепция `elements` в Selenium. Я уже упоминала этот объект в главе 11, он возвращается при вызовах функций такого типа:

```
usernameField                                     =  
driver.find_element_by_name('username')
```

Подобно тому как в браузере можно совершать всевозможные действия с различными элементами сайта, Selenium точно так же позволяет выполнять множество действий с любым элементом, в том числе следующие:

```
myElement.click()
myElement.click_and_hold()
myElement.release()
myElement.double_click()
myElement.send_keys_to_element('content      to
enter')
```

Элементы позволяют не только совершать одиночные действия — строки действий можно объединять в *цепи действий*, сохранять их и выполнять в программе один или несколько раз. Эти цепи полезны тем, что позволяют удобно объединять действия в длинные наборы, но при этом функционально идентичны явному вызову действия для одного элемента, как в предыдущих примерах.

Чтобы увидеть эту разницу, рассмотрим страницу формы, размещенную по адресу **<http://pythonscraping.com/pages/files/form.html>** (которая уже использовалась в качестве примера в главе 10). Мы можем заполнить и отправить эту форму следующим образом:

```
from selenium import webdriver
from selenium.webdriver.remote.webelement
import WebElement
from selenium.webdriver.common.keys import Keys
from selenium.webdriver import ActionChains
from selenium.webdriver.chrome.options import
Options

chrome_options = Options()
chrome_options.add_argument('--headless')

driver = webdriver.Chrome(
```

```
executable_path='drivers/chromedriver',
options=chrome_options)
driver.get('http://pythonscraping.com/pages/files/form.html')
```

```
firstnameField =
driver.find_element_by_name('firstname')
lastnameField =
driver.find_element_by_name('lastname')
submitButton =
driver.find_element_by_id('submit')
```

```
### METHOD 1 ###
```

```
#firstnameField.send_keys('Ryan')
lastnameField.send_keys('Mitchell')
submitButton.click()
#####
```

```
### METHOD 2 ###
```

```
actions =
ActionChains(driver).click(firstnameField)
    .send_keys('Ryan')
    .click(lastnameField)
    .send_keys('Mitchell')
    .send_keys(Keys.RETURN)
actions.perform()
#####
```

```
print(driver.find_element_by_tag_name('body').text)
```

```
driver.close()
```

Метод 1 состоит в вызове функции `send_keys` для двух полей, после чего нажимается кнопка отправки формы. В методе 2 используется общая цепь действий, в которой после вызова метода `perform` последовательно производится нажатие каждого поля формы, после чего туда вводится текст. Результат выполнения скрипта в обоих случаях один и тот же. Независимо от того, какой из двух методов был применен, выводится следующая строка:

```
Hello there, Ryan Mitchell!
```

Помимо этих двух методов, есть еще один вариант, в дополнение к объектам, которые используются для обработки команд: обратите внимание, что в первом методе для отправки формы мы нажимаем кнопку отправки, а во втором — используем клавишу **Enter** (Ввод) после заполнения текстового поля. Поскольку существует множество последовательностей событий для выполнения одного и того же действия, есть большое количество способов совершить его с помощью Selenium.

Метод **drag-and-drop**

Нажатие кнопок и ввод текста — само по себе уже неплохо, но в чем Selenium действительно незаменим, так это в работе с относительно новыми способами взаимодействия в Интернете. Selenium легко управляет интерфейсами, построенными на основе метода перетаскивания (`drag-and-drop`). Для использования функции перетаскивания необходимо указать *исходный* элемент (подлежащий перетаскиванию) и задать либо величину смещения, на которую он будет сдвинут, либо *конечный элемент*, на который нужно перетащить исходный элемент.

Пример интерфейса такого типа представлен на демонстрационной странице, расположенной по адресу <http://pythonscraping.com/pages/javascript/draggableDemo.html>:

```
from selenium import webdriver
from selenium.webdriver.remote.webelement
import WebElement
from selenium.webdriver import ActionChains
from selenium.webdriver.chrome.options import
Options
import unittest

class TestAddition(unittest.TestCase):
    driver = None

    def setUp(self):
        chrome_options = Options()
        chrome_options.add_argument('--
headless')
        self.driver = webdriver.Chrome(
            executable_path='drivers/chromedriv
er', options=chrome_options)
        url =
        'http://pythonscraping.com/pages/javascript/dra
ggableDemo.html'
        self.driver.get(url)

    def tearDown(self):
        driver.close()

    def test_drag(self):
```

```

element =
self.driver.find_element_by_id('draggable')
target =
self.driver.find_element_by_id('div2')
actions = ActionChains(self.driver)
actions.drag_and_drop(element,
target).perform()
self.assertEqual('You are definitely
not a bot!',
self.driver.find_element_by_id('mes
sage').text)
```

На демонстрационной странице в теге `div` с идентификатором `message` выводятся два сообщения. Первое гласит:

Prove you are not a bot, by dragging the square
from the blue area

to the red area!

(Докажите, что вы не бот, перетащив квадрат из синей области в красную!) Затем, сразу же после выполнения этой задачи, выводится другое сообщение:

You are definitely not a bot!

(Вы точно не бот!) Как следует из примера, показанного на демонстрационной странице, перетаскивание элементов с целью доказать, что вы не бот, является обычной практикой во многих тестах капчи. Боты уже давно успешно перетаскивают объекты (ведь это всего лишь последовательность действий: нажать, удерживать и перемещать), однако идея предложить

посетителю сайта перетащить тот или иной элемент с целью подтвердить, что он человек, еще долго не умрет.

Кроме того, библиотеки капч с перетаскиваниями редко используют какие-либо трудные для ботов задания, такие как «перетащить изображение котенка на изображение коровы» (что потребовало бы от программы синтаксического анализа умения различать изображения котенка и коровы); вместо этого они часто задействуют упорядочение чисел или какую-либо другую довольно тривиальную задачу, подобную приведенной в предыдущем примере.

Конечно, эффективность этих библиотек заключается в том, что вариантов слишком много, а такие тесты используются слишком редко; скорее всего, никто не озаботится созданием бота, способного проходить любые подобные тесты. В любом случае данного примера должно хватить для демонстрации того, почему никогда не следует применять описанный метод для крупных сайтов.

Создание снимков экрана

Помимо обычных возможностей тестирования, у Selenium есть интересная хитрость, которая позволит вам значительно упростить тестирование (или произвести впечатление на шефа): снимки экрана. Да, вы действительно можете получать фотоподтверждения прямо из юнит-тестов и вам не придется самолично нажимать **PrtScn**:

```
driver = webdriver.Chrome()  
driver.get('http://www.pythonscraping.com/')  
driver.get_screenshot_as_file('tmp/pythonscrapi  
ng.png')
```

Этот скрипт переходит на страницу <http://pythonscraping.com>, делает снимок экрана начальной страницы и сохраняет его в локальной папке tmp (для правильного сохранения файла она уже должна существовать). Снимки можно сохранять в различных графических форматах.

unittest или Selenium

Синтаксическая строгость и многословность Python-библиотеки unittest, возможно, желательна для большинства крупных наборов тестов. Но если нужно протестировать лишь несколько функций сайта, то единственным вариантом может оказаться гибкий и мощный Selenium. Что же выбрать?

Раскрою секрет: вам не нужно выбирать. Selenium вполне пригоден для получения информации о сайте, а unittest позволит оценить, соответствует ли эта информация критериям прохождения теста. Нет никаких причин, по которым вы не могли бы импортировать инструменты Selenium в Python unittest, объединив лучшие свойства обоих.

Например, в следующем скрипте создается юнит-тест для сайта с интерфейсом drag-and-drop с предположением, что при правильной работе после перетаскивания одного элемента на другой выводится сообщение **You are not a bot!** (Вы не бот!):

```
from selenium import webdriver
from selenium.webdriver import ActionChains
from selenium.webdriver.chrome.options import Options
import unittest

class TestDragAndDrop(unittest.TestCase):
```

```

driver = None
def setUp(self):
    chrome_options = Options()
    chrome_options.add_argument('--
headless')
    self.driver = webdriver.Chrome(
        executable_path='drivers/chromedriver', options=chrome_options)
    url =
'http://pythonscraping.com/pages/javascript/draggableDemo.html'
    self.driver.get(url)

def tearDown(self):
    self.driver.close()

def test_drag(self):
    element =
self.driver.find_element_by_id('draggable')
    target =
self.driver.find_element_by_id('div2')
    actions = ActionChains(self.driver)
    actions.drag_and_drop(element,
target).perform()
    self.assertEqual('You are definitely
not a bot!',
        self.driver.find_element_by_id('mes
sage').text)

```

Комбинируя Python unittest и Selenium, можно протестировать практически все, что есть на сайте. В сущности, присоединив некоторые библиотеки обработки изображений,

описанные в главе 13, можно даже сделать снимок экрана веб-страницы и попиксельно проверить, что должно находиться на ней.

Глава 16. Параллельный веб-краулинг

Веб-краулинг выполняется быстро. По крайней мере это, как правило, гораздо быстрее, чем если нанять дюжину стажеров, которые станут вручную копировать данные из Интернета! Конечно, развитие технологии и гедонистическое нетерпение в какой-то момент приведут к тому, что вам заявят, будто даже это «недостаточно быстро». Обычно в подобные моменты люди начинают задумываться о распределенных вычислениях.

В отличие от большинства других технологий веб-краулинг часто попросту невозможно улучшить, «бросая больше циклов на амбразуру». Быстрое выполнение одного процесса еще не означает, что с двумя процессами задача будет решена в два раза быстрее. А выполнение трех процессов может привести к блокировке вас на удаленном сервере, который вы уже заклевали своими запросами!

Однако есть случаи, когда параллельный веб-краулинг или запуск параллельных потоков/процессов могут быть полезны:

- сбор данных не из одного, а из нескольких источников (с нескольких удаленных серверов);
- выполнение длинных или сложных операций с собранными данными (например, анализ изображений или OCR), которые можно совершать параллельно с извлечением данных;
- сбор данных из крупного веб-сервиса, где вы платите за каждый запрос или где создание нескольких подключений к сервису не выходит за рамки пользовательского соглашения.

Процессы или потоки

Python поддерживает как многопроцессность, так и многопоточность. И та и другая обработка в итоге имеют одну и ту же цель: одновременное решение двух задач программирования вместо выполнения программы более традиционным линейным способом.

В информатике каждый процесс, работающий в операционной системе, может иметь несколько потоков. У него есть своя выделенная память — таким образом, несколько потоков могут обращаться к одной и той же памяти, а несколько процессов — не могут и должны передавать информацию явно.

Часто считается, что использовать многопоточное программирование и выполнять задачи в отдельных потоках с общей памятью проще, чем применять многопроцессное программирование. Но за это удобство приходится платить.

Глобальная блокировка интерпретатора Python (Global Interpreter Lock, GIL) предотвращает одновременное выполнение разными потоками одной и той же строки кода. GIL гарантирует отсутствие повреждений общей памяти, совместно используемой всеми процессами (например, не случится так, что в одни и те же байты памяти наполовину запишется одно значение, а наполовину — другое). Такая блокировка позволяет писать многопоточные программы, всегда точно зная, что именно вы получите в одной и той же строке, но может создавать и узкие места.

Многопоточный веб-краулинг

В Python 3.x используется модуль `thread`; модуль `thread` устарел.

В следующем примере показано использование нескольких потоков для выполнения задачи:

```
import _thread
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0, iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time())
- start)
        print ("{} {}".format(seconds_elapsed,
threadName))

try:
    _thread.start_new_thread(print_time,
('Fizz', 3, 33))
    _thread.start_new_thread(print_time,
('Buzz', 5, 20))
    _thread.start_new_thread(print_time,
('Counter', 1, 100))
except:
    print ('Error: unable to start thread')

while 1:
    pass
```

Это пример классического программного теста FizzBuzz (<http://wiki.c2.com/?FizzBuzzTest>) с несколько измененным выводом результатов:

1 Counter

```
2 Counter
3 Fizz
3 Counter
4 Counter
5 Buzz
5 Counter
6 Fizz
6 Counter
```

Скрипт запускает три потока, один из которых каждые три секунды выводит слово Fizz, другой каждые пять секунд — Buzz, а третий — каждую секунду слово Counter.

После запуска потоков основной поток выполнения запускает цикл `while1`, благодаря которому программа (и ее дочерние потоки) продолжает работать до тех пор, пока пользователь не нажмет **Ctrl+C**, чтобы остановить выполнение программы.

Вместо того чтобы выводить слова Fizz и Buzz, можно выполнять в потоках какую-либо полезную задачу, например, собрать данные с сайта:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random

import _thread
import time

def get_links(thread_name, bs):
    print('Getting links in
    {}'.format(thread_name))
```

```

            return bs.find('div',
{'id': 'bodyContent'}).find_all('a',
                                href=re.compile('^(/wiki/)((?!:).)*$'))

# Определяем функцию для потока
def scrape_article(thread_name, path):
    html =
urlopen('http://en.wikipedia.org{}'.format(path
))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Scraping {} in thread
{}'.format(title, thread_name))
    links = get_links(thread_name, bs)
    if len(links) > 0:
        newArticle = links[random.randint(0,
len(links)-1)].attrs['href']
        print(newArticle)
        scrape_article(thread_name, newArticle)

# Создаем следующие два потока:
try:
    _thread.start_new_thread(scrape_article,
('Thread 1', '/wiki/Kevin_Bacon',))
    _thread.start_new_thread(scrape_article,
('Thread 2', '/wiki/Monty_Python',))
except:
    print ('Error: unable to start threads')

while 1:
    pass

```

Обратите внимание, что мы включили в код следующую строку:

```
time.sleep(5)
```

Поскольку мы сканируем «Википедию» почти вдвое быстрее, чем если бы поток был один, эта строка предотвращает чрезмерную нагрузку, которую скрипт мог бы создать на серверы «Википедии». На практике при работе с сервером, где количество запросов не является проблемой, данную строку следует удалить.

А если немного переписать код, чтобы он отслеживал статьи, которые потоки уже встречали, с целью исключить повторное посещение статей? Для этого можно использовать список в многопоточной среде — точно так же, как и в однопоточной:

```
visited = []
def get_links(thread_name, bs):
    print('Getting links in
    {}'.format(thread_name))
    links = bs.find('div',
    {'id': 'bodyContent'}).find_all('a',
    href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not
    in visited]

def scrape_article(thread_name, path):
    visited.append(path)
```

Обратите внимание: первым действием, которое выполняет `scrape_article`, является добавление пути в список посещенных путей. Это уменьшает, но не исключает полностью

вероятность того, что веб-скрапер обработает данную страницу дважды.

Если вам особенно не повезет, то оба потока наткнутся на один и тот же путь в одно и то же время. Оба увидят, что этого пути нет в списке посещенных, и, соответственно, одновременно добавят его в список и проведут веб-скрапинг. Однако на практике такое вряд ли случится вследствие скорости выполнения и количества страниц, содержащихся в «Википедии».

Это пример *состояния гонки*. Такие состояния вызывают сложности при отладке даже у опытных программистов, так что важно определить, насколько ваш код способен создавать подобные ситуации, оценить их вероятность и предвидеть серьезность возможных последствий.

В случае конкретно этого состояния гонки, когда веб-скрапер дважды обрабатывает одну и ту же страницу, возможно, и не стоит переписывать код.

Состояния гонки и очереди

Мы могли бы наладить коммуникацию между потоками с помощью списков, однако те не предназначены специально для обмена данными между потоками и их неправильное использование вполне может привести к тому, что программа будет выполняться медленно или даже возникнут ошибки вследствие состояния гонки.

Списки отлично подходят для добавления или чтения элементов. Но с удалением элементов в произвольных точках, особенно в начале списка, дела обстоят далеко не так хорошо. Используя строку, подобную этой:

```
myList.pop(0)
```

мы фактически требуем, чтобы Python переписал весь список, а это замедлило бы выполнение программы.

Еще более опасно то, что при использовании списка легко сделать в строке случайную запись, которая не является поточно-ориентированной. Например, такая запись:

```
myList[len(myList)-1]
```

в многопоточной среде может в действительности получить не последний элемент списка или даже выдать исключение, если непосредственно после вычисления значения `len(myList)-1` другая операция изменит список.

Вы можете возразить, что было бы более в духе Python записать предыдущее выражение как `myList[-1]`. Ну да, конечно же, никто из нас в минуту слабости никогда не писал код не в стиле Python (особенно Java-разработчики не любят признаваться, что было время, когда они писали нечто вроде `myList[myList.length-1]`)! Но даже если ваш код безупречен, посмотрите на другие варианты потоконебезопасных строк, где используются списки:

```
my_list[i] = my_list[i] + 1
my_list.append(my_list[-1])
```

Обе эти записи могут привести к состоянию гонки, которое будет иметь неожиданные последствия. Так что откажемся от списков и станем передавать сообщения в потоки другими способами!

```
# Считываем входящее сообщение из глобального
списка
my_message = global_message
# Записываем сообщение обратно
```

```
global_message = 'I've retrieved the message'  
# делаем что-то с my_message
```

Это выглядит отлично, пока мы не обнаружим, что могли случайно стереть сообщение, поступившее из другого потока в момент между первой и второй строками, записав вместо него текст `I've retrieved the message`. И теперь нам придется для каждого потока построить сложную последовательность объектов личных сообщений с какой-то логикой с целью выяснять, кто что получает... или же использовать модуль `Queue`, созданный как раз для этого.

Очереди — это объекты, похожие на списки, которые работают либо по принципу «первым пришел — первым вышел» (`First In First Out, FIFO`), либо по принципу «последним пришел — первым вышел» (`Last In First Out, LIFO`). Очередь получает сообщения из любого потока с помощью функции `queue.put('Mymessage')` и может передать сообщение любому потоку, вызывающему функцию `queue.get()`.

Очереди предназначены не для хранения статических данных, а для их передачи потокобезопасным способом. После извлечения из очереди данные должны существовать только в том потоке, который их извлек. Поэтому очереди обычно используются для делегирования задач или отправки временных уведомлений.

Это может быть полезно при веб-краулинге. Например, мы хотим сохранить данные, собранные веб-скрапером, в базе, и чтобы при этом каждый поток сохранял свои данные быстро. Одно общее соединение для всех потоков может вызвать проблемы (одно соединение неспособно обрабатывать запросы параллельно), однако нет смысла присваивать каждому потоку веб-скрапера отдельное соединение с базой данных. По мере увеличения размера веб-скрапера (возможно, вы в итоге будете

собирать данные с сотен разных сайтов в сотне потоков) это может привести к большому количеству неиспользуемых соединений с базой данных, которые лишь время от времени выполняют запись после загрузки страницы.

Вместо этого можно создать меньшее количество потоков базы данных, у каждого из которых будет собственное соединение; каждый из них будет время от времени извлекать элементы из очереди и сохранять их в базе. Так мы получим гораздо более управляемый набор соединений с базой данных.

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random
import _thread
from queue import Queue
import time
import pymysql

def storage(queue):
    conn = pymysql.connect(host='127.0.0.1',
        unix_socket='/tmp/mysql.sock',
        user='root', passwd='', db='mysql',
        charset='utf8')
    cur = conn.cursor()
    cur.execute('USE wiki_threads')
    while 1:
        if not queue.empty():
            article = queue.get()
            cur.execute('SELECT * FROM pages
WHERE path = %s',
                (article["path"]))
```



```

        if cur.rowcount == 0:
            print("Storing article
{}").format(article["title"]))
            cur.execute('INSERT INTO pages
(title, path) VALUES (%s, %s)', \
                        (article["title"],
article["path"]))
            conn.commit()
        else:
            print("Article already exists:
{}").format(article['title']))

visited = []
def getLinks(thread_name, bs):
    print('Getting links in
{}'.format(thread_name))
    links = bs.find('div',
{'id': 'bodyContent'}).find_all('a',
    href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not
in visited]

def scrape_article(thread_name, path, queue):
    visited.append(path)
    html =
urlopen('http://en.wikipedia.org{}'.format(path
))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Added {} for storage in thread
{}'.format(title, thread_name))

```

```

        queue.put({"title":title, "path":path})
        links = getLinks(thread_name, bs)
        if len(links) > 0:
            newArticle = links[random.randint(0,
len(links)-1)].attrs['href']
            scrape_article(thread_name, newArticle,
queue)

queue = Queue()
try:
    _thread.start_new_thread(scrape_article,
('Thread 1',
    '/wiki/Kevin_Bacon', queue,))
    _thread.start_new_thread(scrape_article,
('Thread 2',
    '/wiki/Monty_Python', queue,))
    _thread.start_new_thread(storage, (queue,))
except:
    print ('Error: unable to start threads')

while 1:
    pass

```

В этом скрипте создаются три потока: два из них предназначены для веб-скрапинга страниц «Википедии», выбираемых случайным образом, а третий — для сохранения собранных данных в базе данных MySQL. Подробнее о MySQL и хранении данных см. в главе 6.

Модуль threading

Python-модуль `_thread` — довольно низкоуровневый, позволяющий управлять всеми нюансами потоков, однако высокоуровневых функций, которые могли бы облегчить жизнь разработчику, у этого модуля немного. Модуль `threading` является высокоуровневым интерфейсом, позволяющим аккуратно использовать потоки, одновременно реализуя все функции лежащего в его основе модуля `_thread`.

Например, мы можем использовать статические функции, такие как `enumerate`, чтобы получить список всех активных потоков, инициализированных через модуль `threading`, и для этого не придется отслеживать потоки самостоятельно. Аналогичным образом, функция `activeCount` возвращает общее количество потоков. Многие функции из модуля `_thread` получили в `threading` более удобные или запоминающиеся имена, например `currentThread` вместо `get_ident`, которая позволяет узнать имя текущего потока.

Вот простой пример использования модуля `threading`:

```
import threading
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0, iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time())
- start)
        print ('{} {}'.format(seconds_elapsed,
threadName))

threading.Thread(target=print_time,          args=
('Fizz', 3, 33)).start()
```

```
threading.Thread(target=print_time,          args=
('Buzz', 5, 20)).start()
threading.Thread(target=print_time,          args=
('Counter', 1, 100)).start()
```

Этот код выводит те же результаты алгоритма FizzBuzz, что и предыдущий простой пример с `_thread`.

Одна из приятных особенностей модуля `threading` — простота создания локальных данных потоков, недоступных для других потоков. Это может быть удобным, если у вас есть несколько потоков, каждый из которых выполняет веб-скрапинг своего сайта и ведет собственный локальный список посещенных страниц.

Эти локальные данные можно создавать в любой точке внутри функции потока, вызвав `threading.local()`:

```
import threading

def crawler(url):
    data = threading.local()
    data.visited = []
    # Сканируем сайт

threading.Thread(target=crawler,          args=
('http://brookings.edu')).start()
```

Это решает проблему состояния гонки, которое могло бы возникнуть между общими объектами потоков. Если объект не должен быть общедоступным, то его и не следует таким делать; подобный объект следует хранить в локальной памяти потока. Для безопасного общего доступа нескольких потоков к объекту можно использовать тот же модуль `Queue`, описанный в предыдущем подразделе.

Модуль `threading` играет роль своеобразной «няни» для потока, и такие его обязанности можно легко настроить. Функция `isAlive` по умолчанию проверяет, активен ли поток. Поток будет активным до тех пор, пока не завершит веб-краулинг (или не случится сбой).

Веб-краулеры часто рассчитаны на очень длительное время работы. Метод `isAlive` позволяет гарантировать, что в случае сбоя потока веб-краулер перезапустится:

```
threading.Thread(target=crawler)
t.start()

while True:
    time.sleep(1)
    if not t.isAlive():
        t = threading.Thread(target=crawler)
        t.start()
```

Чтобы добавить другие методы мониторинга, нужно расширить объект `threading.Thread`:

```
import threading
import time

class Crawler(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.done = False

    def isDone(self):
        return self.done
```

```
def run(self):
    time.sleep(5)
    self.done = True
    raise Exception('Something bad
happened!')

t = Crawler()
t.start()

while True:
    time.sleep(1)
    if t.isDone():
        print('Done')
        break
    if not t.isAlive():
        t = Crawler()
        t.start()
```

Этот новый класс `Crawler` содержит метод `isDone`, который годится для проверки того, завершил ли веб-краулер работу. Это может быть полезно, если есть какие-то дополнительные методы журналирования, которые необходимо завершить перед закрытием потока, при этом основная часть работы по веб-краулингу уже выполнена. Как правило, `isDone` можно заменить каким-либо статусом или показателем прогресса — например, показывающим текущую страницу или сколько страниц зарегистрировано.

Любые исключения, вызванные `Crawler.run`, приведут к перезапуску класса, пока `isDone` не станет равным `True`, после чего программа завершит работу.

Создавая классы веб-краулеров как расширения `threading.Thread`, можно сделать их более надежными и гибкими, а также одновременно контролировать все свойства нескольких веб-краулеров.

Многопроцессный веб-краулинг

Python-модуль `Processing` создает новые объекты процессов, которые можно запускать и присоединять из главного процесса. В следующем коде для демонстрации используется пример FizzBuzz из раздела, посвященного потокам и процессам.

```
from multiprocessing import Process
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0, iterations):
        time.sleep(delay)
        seconds_elapsed = str(int(time.time())
- start)
        print (threadName if threadName else
seconds_elapsed)

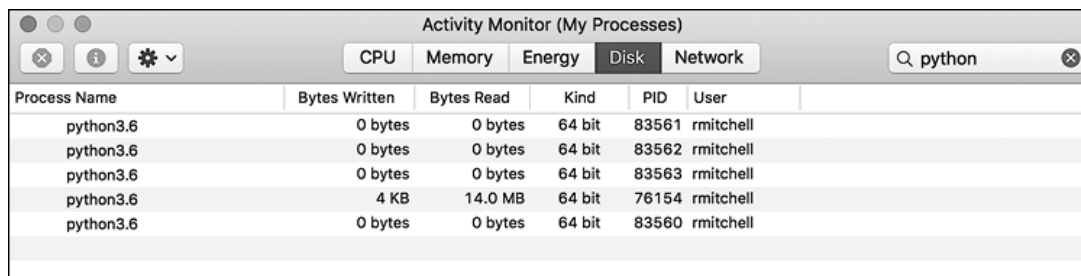
processes = []
processes.append(Process(target=print_time,
args=('Counter', 1, 100)))
processes.append(Process(target=print_time,
args=('Fizz', 3, 33)))
processes.append(Process(target=print_time,
args=('Buzz', 5, 20)))
```

```

for p in processes:
    p.start()
for p in processes:
    p.join()

```

Помните, что операционная система рассматривает каждый процесс как отдельную независимую программу. Если вы посмотрите на свои процессы через монитор активности или диспетчер задач ОС, то увидите картину, похожую на ту, что показана на рис. 16.1.



The screenshot shows the 'Activity Monitor (My Processes)' window with the 'CPU' tab selected. A search bar at the top right contains the text 'python'. The table below lists five processes, all named 'python3.6' and running under the user 'rmitchell'.

Process Name	Bytes Written	Bytes Read	Kind	PID	User
python3.6	0 bytes	0 bytes	64 bit	83561	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83562	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83563	rmitchell
python3.6	4 KB	14.0 MB	64 bit	76154	rmitchell
python3.6	0 bytes	0 bytes	64 bit	83560	rmitchell

Рис. 16.1. Пять процессов Python, выполняемых во время работы FizzBuzz

Четвертый процесс с PID 76154 — это действующий экземпляр Jupyter Notebook, который должен присутствовать, если вы работаете из редактора iPython. Пятый процесс, 83560, является основным потоком выполнения, запускающимся при первом запуске программы. PID присваиваются операционной системой последовательно. Если у вас нет другой программы, которая одновременно работает с FizzBuzz и быстро выделяет PID, то вы должны увидеть еще три последовательных PID — в данном случае это 83561, 83562 и 83563.

Эти PID также можно получить из кода с помощью модуля `os`:

```

import os
...

```



```
# Выводит дочерний PID
os.getpid()
# Выводит родительский PID
os.getppid()
```

Каждый процесс программы при выполнении `os.getpid()` должен выводить свой, отдельный PID, а в момент выполнения `os.getppid()` — один и тот же родительский PID.

Есть пара строк кода, которые в данной конкретной программе, строго говоря, не нужны. Если не добавить заключительный оператор `join`:

```
for p in processes:
    p.join()
```

то родительский процесс все равно завершится и его дочерние процессы автоматически подойдут к концу. Однако данное объединение необходимо, если после завершения этих дочерних процессов вы захотите выполнить еще какой-либо код.

Например:

```
for p in processes:
    p.start()
print('Program complete')
```

Если убрать оператор `join`, то результат будет следующим:

```
Program complete
1
2
```

Если включить оператор `join`, то программа сначала дождетсЯ завершения всех процессов и только потом

продолжит выполняться:

```
for p in processes:
    p.start()

for p in processes:
    p.join()
print('Program complete')

...
Fizz
99
Buzz
100
Program complete
```

Желая преждевременно остановить выполнение программы, можно, конечно, нажать **Ctrl+C**, чтобы завершить родительский процесс. При его завершении также подойдут к концу и все дочерние процессы, которые были им порождены, поэтому можно спокойно использовать **Ctrl+C**, не беспокоясь о том, что некоторые процессы случайно могут остаться работать в фоновом режиме.

Пример многопроцессного веб-краулинга

Пример многопоточного веб-краулинга «Википедии» можно изменить, чтобы использовать отдельные не потоки, а процессы:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
```

```

import random

from multiprocessing import Process
import os
import time

visited = []
def get_links(bs):
    print('Getting links in {}'.format(os.getpid()))
    links = bs.find('div', {'id': 'bodyContent'}).find_all('a', href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link for link in links if link not in visited]

def scrape_article(path):
    visited.append(path)
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    time.sleep(5)
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    print('Scraping {} in process {}'.format(title, os.getpid()))
    links = get_links(bs)
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        print(newArticle)
        scrape_article(newArticle)

```

```
processes = []
processes.append(Process(target=scrape_article,
args=( '/wiki/Kevin_Bacon', )))
processes.append(Process(target=scrape_article,
args=( '/wiki/Monty_Python', )))

for p in processes:
    p.start()
```

Мы снова искусственно замедляем процесс веб-скрапера, добавляя строку `time.sleep(5)`, чтобы его можно было использовать в качестве примера без чрезмерной нагрузки на серверы «Википедии».

Здесь мы заменяем определенную пользователем переменную `thread_name`, передаваемую в качестве аргумента, на результат вызова функции `os.getpid()`, который не нужно передавать как аргумент и к которому можно получить доступ в любой момент.

В результате получим примерно такой результат:

```
Scraping Kevin Bacon in process 84275
Getting links in 84275
/wiki/Philadelphia
Scraping Monty Python in process 84276
Getting links in 84276
/wiki/BBC
Scraping BBC in process 84276
Getting links in 84276
/wiki/Television_Centre,_Newcastle_upon_Tyne
Scraping Philadelphia in process 84275
```

Теоретически веб-краулинг в отдельных процессах должен выполняться немного быстрее, чем в отдельных потоках, по двум основным причинам.

- Процессы не подлежат блокировке GIL. Они могут выполнять одни и те же строки кода и изменять один и тот же объект (на самом деле это разные экземпляры одного и того же объекта) одновременно.
- Процессы могут выполняться на нескольких ядрах процессора, что позволит обеспечить выигрыш в скорости, если каждый из процессов или потоков интенсивно использует процессор.

Однако здесь, наряду с преимуществами, есть существенный недостаток. В рассмотренной программе все найденные URL хранятся в глобальном списке `visited`. Когда мы использовали несколько потоков, этот список был общим для всех потоков; при этом один поток, за редким исключением состояния гонки, не мог посетить страницу, уже посещенную другим потоком. Однако теперь каждый процесс получает собственную, независимую версию списка посещенных страниц и вполне может посещать страницы, уже просмотренные другими процессами.

Обмен данными между процессами

Каждый процесс работает в собственной, независимой памяти, что может вызвать проблемы, если мы хотим, чтобы процессы использовали общую информацию.

Изменив предыдущий пример так, чтобы выводилась текущая версия списка посещенных страниц, мы можем увидеть, как работает этот принцип:

```
def scrape_article(path):  
    visited.append(path)  
    print("Process {} list is now:  
{}".format(os.getpid(), visited))
```

В результате получим следующее:

```
Process      84552      list      is      now:  
['/wiki/Kevin_Bacon']  
Process      84553      list      is      now:  
['/wiki/Monty_Python']  
Scraping Kevin Bacon in process 84552  
Getting links in 84552  
/wiki/Desert_Storm  
Process      84552      list      is      now:  
['/wiki/Kevin_Bacon', '/wiki/Desert_Storm']  
Scraping Monty Python in process 84553  
Getting links in 84553  
/wiki/David_Jason  
Process      84553      list      is      now:  
['/wiki/Monty_Python', '/wiki/David_Jason']
```

Но есть способ обмена информацией между процессами, работающими на одной машине, через два типа объектов Python: очереди и каналы.

Очередь похожа на рассмотренную ранее очередь потоков. Один процесс может поместить в нее информацию, а другой — удалить ее оттуда. После удаления этой информации больше нет в очереди. Поскольку очереди спроектированы как способ временной передачи данных, они не особенно хорошо подходят для хранения статических ссылок, таких как список уже посещенных веб-страниц.

Но что если заменить этот статический список веб-страниц каким-либо делегатором веб-скрапинга? Веб-скраперы могут извлекать задачу из одной очереди в виде пути для продолжения веб-скрапинга (например, /wiki/Monty_Python), а затем помещать список «найденных URL» в другую очередь. А она будет обрабатываться делегатором веб-скрапинга, который следит, чтобы в первую очередь задач добавлялись только новые URL:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random
from multiprocessing import Process, Queue
import os
import time

def task_delegator(taskQueue, urlsQueue):
    # Инициализируем задачу для каждого
    # процесса.
    visited = ['/wiki/Kevin_Bacon',
               '/wiki/Monty_Python']
    taskQueue.put('/wiki/Kevin_Bacon')
    taskQueue.put('/wiki/Monty_Python')

    while 1:
        # Проверяем, есть ли в urlsQueue
        # новые ссылки, доступные для
        # обработки.
        if not urlsQueue.empty():
            links = [link for link in
                     urlsQueue.get() if link not in visited]
```

```

        for link in links:
            # Добавляем в taskQueue новую
ссылку.
            taskQueue.put(link)

def get_links(bs):
    links = bs.find('div',
{'id': 'bodyContent'}).find_all('a',
    href=re.compile('^(/wiki/)((?!:).)*$'))
    return [link.attrs['href'] for link in
links]

def scrape_article(taskQueue, urlsQueue):
    while 1:
        while taskQueue.empty():
            # Засыпаем на 100 мс, ожидая
очередь задач.
            # Это должно происходить редко.
            time.sleep(.1)
            path = taskQueue.get()
            html =
urlopen('http://en.wikipedia.org{}'.format(path
))
            time.sleep(5)
            bs = BeautifulSoup(html, 'html.parser')
            title = bs.find('h1').get_text()
            print('Scraping {} in process
{}'.format(title, os.getpid()))
            links = get_links(bs)
            # Отправляем это на обработку
делегатору.
            urlsQueue.put(links)

```



```
processes = []
taskQueue = Queue()
urlsQueue = Queue()
processes.append(Process(target=task_delegator,
args=(taskQueue, urlsQueue,)))
processes.append(Process(target=scrape_article,
args=(taskQueue, urlsQueue,)))
processes.append(Process(target=scrape_article,
args=(taskQueue, urlsQueue,)))

for p in processes:
    p.start()
```

У этого веб-скрапера есть ряд структурных отличий от скриптов, созданных нами ранее. Если раньше каждый процесс или поток выполнял случайный переход от назначенной ему начальной точки, то теперь они работают совместно, производя полный сбор данных с сайта. Каждый процесс может извлечь из очереди любую задачу, а не только те ссылки, которые он сам нашел.

Многопроцессный веб-краулинг: еще один подход

Все обсуждаемые здесь подходы многопоточного и многопроцессного веб-краулинга предполагают необходимость некоего родительского контроля для дочерних потоков и процессов. Мы можем их все одновременно запустить и завершить, передавать сообщения между ними или выделить для них общую память.

А если веб-скрапер спроектирован таким образом, что не нуждается в каких-либо указаниях или обмене данными? Тогда

у вас едва ли найдутся причины, чтобы озадачиваться импортом `_thread` на данном этапе.

Например, мы хотим параллельно просканировать два схожих сайта. У нас есть веб-краулер, который может просканировать любой из них, для чего потребуется небольшое изменение конфигурации или, вероятно, другие аргументы командной строки. Нет совершенно никаких причин, по которым мы не могли бы просто сделать следующее:

```
$ python my_crawler.py website1  
$ python my_crawler.py website2
```

И — вуаля! — мы только что запустили многопроцессный веб-краулер, сэкономив ресурсы процессора и избежав издержек на хранение родительского процесса для загрузки!

Конечно, представленный подход имеет свои недостатки. Если мы хотим запустить таким образом два веб-краулера на *одном* сайте, то нам понадобится какой-то способ, позволяющий убедиться, что эти веб-краулеры случайно не станут выполнять веб-скрапинг одних и тех же страниц. Для решения этой проблемы можно создать URL-правило («краулер 1 выполняет веб-скрапинг страниц блога, а краулер 2 — страниц товаров») или разделить сайт еще каким-либо способом.

Кроме того, можно организовать координацию через какую-либо промежуточную базу данных. Прежде чем перейти по новой ссылке, краулер может сделать запрос к ней и спросить: «Эта страница уже просканирована?» Веб-краулер использует базу в качестве системы межпроцессного взаимодействия. Разумеется, если метод тщательно не продумать, то он может привести к состоянию гонки или задержкам в случае медленного соединения с базой данных

(вероятно, такая проблема возможна только при подключении к удаленной БД).

Кроме того, вы можете обнаружить, что этот метод масштабируется хуже, чем предыдущие. Использование модуля `Process` позволяет динамически увеличивать или уменьшать количество процессов, сканирующих сайт или даже хранящих данные. Для их отключения вручную понадобится либо человек, физически выполняющий скрипт, либо отдельный управляющий скрипт (будь то скрипт `bash`, задание `cron` или что-либо еще).

Тем не менее я не раз с успехом использовала данный метод. Для небольших разовых проектов это отличный способ быстро получить много информации, особенно с нескольких сайтов.

Глава 17. Удаленный веб-скрапинг

В предыдущей главе мы рассмотрели запуск веб-скраперов в нескольких потоках и процессах, так что обмен данными между ними был несколько ограничен или же его приходилось тщательно планировать. В текущей главе мы доведем эту концепцию до ее логического завершения: будем запускать веб-краулеры не только в отдельных процессах, но и на разных машинах.

Этой теме не случайно посвящена последняя техническая глава книги. До сих пор мы запускали все приложения Python из командной строки на своем домашнем компьютере. Конечно, мы могли установить MySQL, постаравшись воспроизвести среду реального сервера. Но это не одно и то же. Как говорится, «если ты кого-то любишь — дай ему свободу».

В данной главе мы рассмотрим несколько способов запуска скриптов с разных компьютеров и даже с разных IP-адресов. У вас может возникнуть соблазн отложить этот шаг до лучших времен, поскольку сейчас он вам *не нужен*. Однако вы будете удивлены тем, как легко начать работу с уже имеющимися инструментами (такими как персональный сайт на платном хостинге) и насколько станет проще жить, стоит перестать пытаться запускать написанные на Python веб-скраперы с личного ноутбука.

Зачем использовать удаленные серверы

При запуске веб-приложения, предназначенного для широкой аудитории, использование удаленного сервера может показаться очевидным шагом, однако инструмент, созданный для наших целей, мы часто так и продолжаем запускать

локально. Те, кто решил перейти на удаленную платформу, обычно руководствуются двумя основными соображениями: потребностью в большей мощности и гибкости и необходимостью применять другой IP-адрес.

Как избежать блокировки IP-адреса

При создании веб-скраперов главное правило гласит: подделать можно практически все. Вы можете отправлять электронные письма с не принадлежащих вам адресов, передавать из командной строки автоматизированные данные о перемещениях мыши и даже, к ужасу веб-администраторов, передавать трафик их сайтов из Internet Explorer 5.0.

Единственное, что нельзя подделать, — это ваш IP-адрес. Кто угодно может прислать вам письмо с обратным адресом «Соединенные Штаты Америки, Вашингтон (округ Колумбия), Пенсильвания-авеню, 1600, Президент». Однако если письмо отправлено из Альбукерке, штат Нью-Мексико, вы можете быть уверены, что не состоите в переписке с президентом Соединенных Штатов²⁸.

Большинство усилий, направленных на то, чтобы не предоставить веб-скраперам доступ к сайтам, концентрируются на отличии людей от ботов. Дойти до блокировки IP-адресов — это примерно как фермеру отказаться от распыления пестицидов и взамен просто поджечь поле. Это крайняя, хотя и эффективная мера отбрасывания пакетов, отправленных с проблемных IP-адресов. Однако у такого решения есть свои нюансы.

- Списки доступа по IP-адресам трудно поддерживать. У крупных сайтов чаще всего есть собственные программы, автоматизирующие некоторые рутинные операции с такими списками (боты, блокирующие боты!). Однако все равно кто-

то должен время от времени проверять эти списки или хотя бы отслеживать их рост и устранять проблемы.

- Каждый адрес — это дополнительное время обработки, на которое увеличится время получения пакетов, поскольку сервер должен будет проверить полученные пакеты по списку и решить, следует ли их утверждать. Большое количество адресов, умноженное на большое количество пакетов, дает быстрый рост затрат времени. Чтобы сэкономить время и уменьшить сложность обработки, администраторы часто группируют IP-адреса в блоки и устанавливают правила вроде «заблокировать все 256 адресов из данного диапазона», если найдут несколько тесно сгруппированных нарушителей. Это приводит нас к третьему пункту.
- Блокировка IP-адреса также может привести к блокировке «хороших парней». Например, когда я училась в Инженерном колледже им. Франклина В. Олина, один студент написал программу, которая пыталась подделать голосование за широко известный контент на сайте **<http://digg.com>** (это было еще до пика популярности Reddit). Один заблокированный IP-адрес привел к тому, что все общежитие лишилось доступа к сайту. Тот студент просто перенес свою программу на другой сервер; тем временем страницы Digg потеряли многих посетителей из своей основной целевой аудитории.

Несмотря на все свои недостатки, блокировка IP-адресов остается чрезвычайно распространенным методом, которым пользуются администраторы, чтобы запретить подозрительным веб-скраперам доступ к серверам. Если IP-адрес будет заблокирован, то единственным действенным

решением остается веб-скрапинг с другого IP-адреса. Для этого можно либо переместить веб-скрапер на другой сервер, либо построить маршрутизацию трафика через другой сервер с помощью сервиса наподобие Tor.

Портируемость и расширяемость

Некоторые задачи слишком велики для домашнего компьютера и подключения к Интернету. И пусть вы не намерены создавать большую нагрузку на какой-либо сайт, но, возможно, собираете данные на разных сайтах, для чего потребуются гораздо большие пропускная способность и объем памяти, чем может обеспечить ваша текущая конфигурация.

Более того, избавившись от вычислительно интенсивной обработки данных, можно высвободить ресурсы домашнего компьютера для более важных задач (есть здесь любители World of Warcraft?). Вам не придется думать о том, как обеспечить бесперебойное электропитание и подключение к Интернету (можно зайти в Starbucks, запустить приложение, закрыть ноутбук и уйти, зная, что все продолжает надежно работать), и вы получите доступ к собранным данным в любом месте, где есть подключение к Интернету.

Если ваше приложение требует такой вычислительной мощности, что его не удовлетворит даже один большой вычислительный узел Amazon, то можно воспользоваться *распределенными вычислениями*. Тогда на достижение ваших целей будут работать параллельно несколько машин. Приведу простой пример: у вас есть один компьютер, который сканирует один набор сайтов, и второй компьютер, который сканирует второй набор сайтов; оба компьютера сохраняют собранные данные в общей базе.

Конечно, как отмечалось в предыдущих главах, кто угодно может скопировать систему поиска Google, но мало кто сможет повторить масштаб, в котором Google выполняет свой поиск. Распределенные вычисления — обширная область информатики, выходящая за рамки данной книги. Однако знание того, как запустить приложение на удаленном сервере, является необходимым первым шагом, и вы будете удивлены тем, на что способны современные компьютеры.

Tor

Сеть Onion Router, более известная под аббревиатурой *Tor*, представляет собой сеть волонтерских серверов, настроенных на маршрутизацию и перенаправление трафика через многие уровни (именно поэтому в названии присутствует слово Onion — «репчатый лук») разных серверов, чтобы скрыть происхождение этого трафика. Перед поступлением в сеть данные зашифровываются, и поэтому, если окажется, что какой-нибудь сервер прослушивается, природа коммуникации не будет раскрыта. Кроме того, хоть входящий и исходящий поток данных любого сервера и может быть скомпрометирован, чтобы расшифровать истинные начальные и конечные точки соединения, необходимо знать детали о входящих и исходящих данных для *всех* серверов коммуникационного пути — подвиг на грани невозможного.

Tor часто используют правозащитники и анонимные политические информаторы для общения с журналистами, вследствие чего значительную часть финансирования эта сеть получает от правительства США. Конечно же, она также широко используется для незаконной деятельности, из-за чего является постоянным объектом государственного надзора (хотя пока что он имел в лучшем случае переменный успех).



Ограничения анонимности Tor

Несмотря на то что причина, по которой мы будем использовать Tor в этой книге, состоит в изменении своего IP-адреса, а не в получении полной анонимности как таковой, все же стоит уделить время изучению некоторых сильных сторон и ограничений возможностей Tor по созданию анонимного трафика.

Используя Tor, можно предположить, что IP-адрес, с которого мы пришли на веб-сервер, не является для этого сервера тем IP-адресом, по которому нас можно проследить. Однако нас может выдать любая информация, предоставляемая нами веб-серверу. Например, если вы войдете в свою учетную запись Gmail и затем выполните подозрительный поиск в Google, то его можно будет привязать к вашей личности.

Однако, помимо очевидного, сама аутентификация в Tor способна раскрыть вашу анонимность. В декабре 2013 года студент Гарвардского университета, чтобы не сдавать выпускные экзамены, отправил через сеть Tor сообщение о том, что учебное заведение заминировано, используя анонимную учетную запись электронной почты. Когда сотрудники IT-отдела Гарварда просмотрели свои журналы, то обнаружили следующее: во время отправки сообщения об угрозе взрыва

трафик, поступивший в сеть Tor, попадал туда только с одной машины, зарегистрированной на студента, имя которого было известно. Несмотря на то что исходный пункт этого трафика определить не удалось (только его передачу через Tor), самого факта совпадения времени и того, что в это время был зарегистрирован вход в систему только с одной машины, было достаточно для того, чтобы на этого студента подали в суд.

Аутентификация в Tor — не автоматическая мантия-невидимка, которая предоставляла бы вам полную свободу делать в Интернете все что угодно. Да, это полезный инструмент, однако всегда применяйте его с умом, осторожностью и, конечно же, с благими намерениями.

Как вы узнаете в следующем подразделе, для применения Tor в Python необходимо установить и запустить Tor. К счастью, установить и запустить сервис Tor очень легко. Просто перейдите на страницу скачивания Tor (<https://www.torproject.org/download/download>), скачайте, установите, откройте и подключите его. И учтите, что при использовании Tor скорость соединения с Интернетом может показаться ниже обычной. Будьте терпеливы — возможно, данные, прежде чем попасть к вам, несколько раз обогнут земной шар!

PySocks. PySocks — удивительно простой модуль Python, способный маршрутизировать трафик через прокси-серверы и просто фантастически эффективный в сочетании с Tor. PySocks можно скачать с его сайта (<https://pypi.python.org/pypi/PySocks/1.5.0>) или использовать для его установки любой из многочисленных сторонних менеджеров модулей.

У этого модуля не очень много документации, однако пользоваться им чрезвычайно просто. Перед выполнением следующего кода необходимо запустить сервис Tor на порте 9150 (порте, применяемом по умолчанию):

```
import socks
import socket
from urllib.request import urlopen

socks.set_default_proxy(socks.SOCKS5,
                        "localhost", 9150)
socket.socket = socks.socksocket
print(urlopen('http://icanhazip.com').read())
```

Сайт **<http://icanhazip.com>** показывает только IP-адрес клиента, подключающегося к серверу, и может быть полезен для тестирования. При выполнении этот скрипт должен отображать IP-адрес, который не является вашим собственным.

Если вы хотите использовать Selenium и ChromeDriver в сочетании с Tor, то PySocks вообще не нужен — просто не забудьте сначала запустить Tor и укажите дополнительный параметр `проxy-server` для Chrome, согласно которому Selenium будет подключаться через порт 9150 по протоколу socks5:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

chrome_options = Options()
chrome_options.add_argument("--headless")
```

```

chrome_options.add_argument("--proxy-
server=socks5://127.0.0.1:9150")
driver =
webdriver.Chrome(executable_path='drivers/chrom
edriver',
options=chrome_option
s)

driver.get('http://icanhazip.com')
print(driver.page_source)
driver.close()

```

Этот скрипт также должен выводить не ваш IP-адрес — на сей раз тот, который в данный момент использует работающий у вас клиент Tor.

Удаленный хостинг

Хотя на полную анонимность в Интернете рассчитывать не стоит — она теряется, стоит вам лишь указать данные кредитной карты, — однако удаленное размещение веб-скраперов также способно значительно повысить скорость их работы. Это объясняется не только возможностью покупать машинное время на гораздо более мощных компьютерах, чем ваш, но и тем, что по дороге к точке назначения соединение больше не будет проходить через слои сети Tor.

Запуск веб-скрапера из учетной записи веб-хостинга

Если у вас есть личный или корпоративный сайт, то, вероятно, уже есть все, что нужно для запуска веб-скрапера с внешнего сервера. Даже на веб-серверах с относительно ограниченным доступом, где нет возможности выполнять команды из

командной строки, можно запускать и останавливать скрипты через веб-интерфейс.

В случае размещения вашего сайта на Linux-сервере тот, скорее всего, сам работает на Python. Если же вы используете хостинг на Windows-сервере, то вам повезло меньше: придется специально проверить, установлен ли там Python или согласится ли администратор сервера его установить.

Большинство мелких провайдеров веб-хостинга предоставляют программное обеспечение под названием cPanel, которое используется для выполнения простейших операций по администрированию, а также для предоставления информации о вашем сайте и связанных с ним сервисах. При наличии доступа к cPanel для проверки того, установлен ли на вашем сервере Python, нужно перейти в раздел Apache Handlers и добавить новый обработчик (если его там еще нет):

```
Handler: cgi-script  
Extension(s): .py
```

Эти строки дают указание вашему серверу выполнять все скрипты Python как скрипты CGI (Common Gateway Interface — общий шлюзовой интерфейс). Это любая программа, которая может выполняться на сервере и динамически генерировать контент для отображения на сайте. Явно определяя скрипты Python как скрипты CGI, мы разрешаем серверу их выполнять, а не просто отображать содержимое в браузере или позволять пользователю его скачивать.

Напишите скрипт Python, загрузите его на сервер и определите права доступа к файлу как 755, то есть разрешите выполнять этот файл как программу. Для выполнения скрипта перейдите в тот каталог, в который вы его загрузили, через браузер (или, что еще лучше, напишите веб-скрапер, который будет делать это вместо вас). Если вы не хотите, чтобы скрипт

попал в открытый доступ и его мог выполнить любой желающий, то есть два варианта этого избежать.

- Храните скрипт под нелогичным или скрытым URL и следите за тем, чтобы никогда не указывать ссылку на скрипт с какого-либо другого доступного URL во избежание его индексации поисковыми системами.
- Защитите скрипт паролем или потребуйте, чтобы перед выполнением скрипта ему передавался пароль или секретный маркер.

Конечно, запуск скрипта Python из сервиса, изначально предназначенного для отображения сайтов, напоминает хакерский прием. В частности, вы, вероятно, заметите, что ваш «веб-скрапер через сайт» загружается немного медленно. На самом деле страница вообще будет отображаться лишь после выполнения всего веб-скрапинга (и только тогда произойдет вывод данных, который вы запрограммировали с помощью операторов `print`). Это может занять несколько минут, часов или вообще не завершится никогда, в зависимости от того, как написан скрипт. Несмотря на то что он, безусловно, выполняет свою работу, вам может потребоваться более подробный вывод результатов в реальном времени. Для этого понадобится сервер, предназначенный не только для сайтов.

Запуск из облака

На заре компьютерных технологий программисты платили за машинное время или резервировали его для выполнения кода. С появлением персональных компьютеров такая необходимость отпала: вы просто пишете и выполняете код на своей машине. Но сейчас амбиции приложений опережают

разработку микропроцессоров до такой степени, что программисты снова стали возвращаться к почасовой плате за вычисления.

Однако на сей раз пользователи платят не за время одной физической машины, а за эквивалентную вычислительную мощность, которая часто распределяется между несколькими компьютерами. Облачная структура этой системы позволяет определять стоимость вычислительной мощности по времени пикового спроса. Например, Amazon позволяет делать наценку на «точечные вычислительные узлы», когда низкая цена важнее реагирования в реальном времени.

Виртуальные вычислительные узлы также являются более специализированными, их можно выбирать в зависимости от потребностей приложения — в большем объеме оперативной памяти или высокой скорости вычислений. Хотя веб-скраперы обычно не используют много памяти, возможно, при размещении приложения веб-скрапера стоит выбрать не универсальный вычислительный узел, а вариант с большим хранилищем для данных или более быстрыми вычислениями. Если вам предстоит обработка больших объемов текстов на естественном языке, распознавание текста или поиск путей в графах (например, при решении задачи «Шесть шагов по “Википедии”»), то вам может подойти вычислительный узел с высокой скоростью вычислений. При сборе больших объемов данных, сохранении файлов или выполнении обширной аналитики, возможно, стоит выбрать вариант с оптимизацией хранилища.

Хоть нас не ограничивает ничто, кроме денег, на момент написания этой книги стоимость аренды вычислительного узла начиналась всего с 1,3 цента в час (для микроузла Amazon EC2), а самый дешевый вычислительный узел Google стоил 4,5 цента в час с минимальным временем аренды всего десять минут.

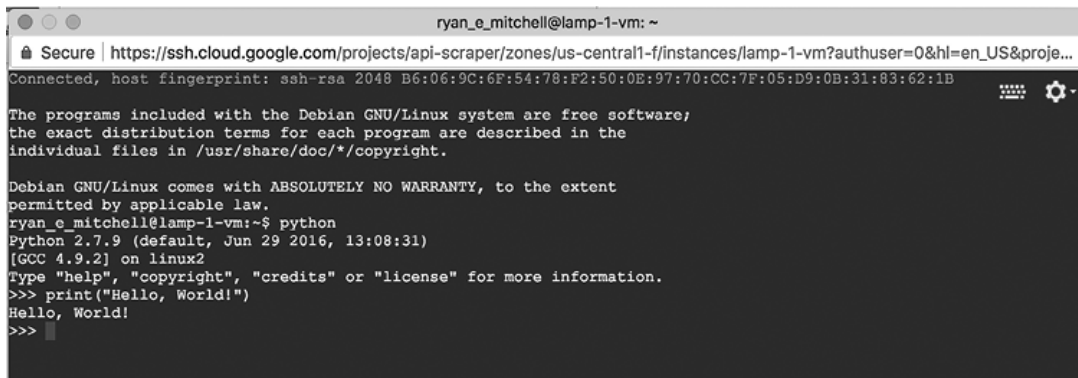
Благодаря экономии на масштабе арендовать небольшой вычислительный узел в крупной компании стоит почти столько же, сколько собственная физическая выделенная машина, за исключением того, что не придется нанимать IT-специалиста, который будет за ней присматривать.

Конечно, пошаговые инструкции по настройке и запуску вычислительных узлов для облачных вычислений несколько выходят за рамки этой книги, но вы, скорее всего, обнаружите, что пошаговые инструкции и не нужны. Поскольку Amazon и Google (не говоря уже о бесчисленных мелких компаниях в данной индустрии) борются за доллары облачных вычислений, они упростили настройку новых вычислительных узлов, так что вам остается лишь следовать простым подсказкам, придумать имя приложения и предоставить номер кредитной карты. Кроме того, на момент написания этой книги Amazon и Google предлагали бесплатное машинное время на сотни долларов для дальнейшего привлечения новых клиентов.

Настроив вычислительный узел, вы станете гордым владельцем IP-адреса, имени пользователя, а также открытого и закрытого ключей, которые можно будет применять для подключения к вашему вычислительному узлу через SSH. С этого момента все будет аналогично работе на сервере, которым вы физически владеете, за исключением того, что вам, естественно, больше не придется беспокоиться о техническом обслуживании оборудования или использовании многочисленных собственных инструментов мониторинга.

Я обнаружила, что если нужно что-то сделать «дешево и сердито», особенно при отсутствии большого опыта работы с SSH и парами ключей, то проще наладить и сразу запустить вычислительные узлы Google Cloud Platform. У них есть простой загрузчик и даже кнопка, доступная сразу после

запуска, которая позволяет просматривать терминал SSH прямо в браузере, как показано на рис. 17.1.



```
ryan_e_mitchell@lamp-1-vm: ~  
Secure | https://ssh.cloud.google.com/projects/api-scraper/zones/us-central1-f/instances/lamp-1-vm?authuser=0&hl=en_US&proje...  
Connected, host fingerprint: ssh-rsa 2048 B6:06:9C:6F:54:78:F2:50:0E:97:70:CC:7F:05:D9:0B:31:83:62:1B  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
ryan_e_mitchell@lamp-1-vm:~$ python  
Python 2.7.9 (default, Jun 29 2016, 13:08:31)  
[GCC 4.9.2] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello, World!")  
Hello, World!  
>>>
```

Рис. 17.1. Открытый в браузере терминал работающего вычислительного узла виртуальной машины Google Cloud Platform

Дополнительные ресурсы

Много лет назад в облаке работали главным образом те, кто захотел и смог пробраться сквозь дебри документации и уже имел опыт администрирования серверов. Однако сегодня благодаря возросшей популярности и конкуренции среди поставщиков облачных вычислений инструменты стали гораздо лучше.

Но все же для создания крупномасштабных или более сложных веб-скраперов и краулеров вам могут понадобиться дополнительные рекомендации по созданию платформы для сбора и хранения данных.

Книга *Google Compute Engine* Марка Коэна (Marc Cohen), Кэтрин Херли (Kathryn Hurley) и Пола Ньюсона (Paul Newson) (издательство O'Reilly) (<http://oreil.ly/1FVOW6y>) — это простой источник информации о применении облачных вычислений Google на Python и JavaScript. В ней рассматривается не только пользовательский интерфейс Google, но и инструменты

командной строки и скрипты, которые можно задействовать для повышения гибкости приложений.

Если вы предпочитаете работать с Amazon, то обратите внимание на книгу *Python and AWS Cookbook* Митча Гарната (Mitch Garnaat) (издательство O'Reilly) (<http://oreil.ly/VSctQP>) — краткое, но чрезвычайно полезное руководство, которое поможет начать работу с Amazon Web Services и покажет, как настроить и запустить масштабируемое приложение.

[28](#) Технически IP-адреса можно подделать в исходящих пакетах. Этот метод используется в распределенных атаках типа «отказ в обслуживании», когда злоумышленникам неважно получить ответные пакеты (которые если и будут отправлены, то доставятся по неправильному адресу). Но веб-скрапинг по определению является действием, в котором ожидается ответ от веб-сервера, поэтому для нас IP-адрес — то, что подделать невозможно.

Глава 18. Законность и этичность веб-скрапинга

В 2010 году инженер-программист Пит Уорден (Pete Warden) разработал веб-краулер для сбора данных из Facebook. Он собрал данные примерно 200 миллионов пользователей соцсети: имена, места жительства, сведения о друзьях и интересах. Конечно же, в Facebook это заметили и стали слать Уордену письма-предупреждения о нарушении прав интеллектуальной собственности, к которым он прислушался. Когда Уордена спросили, почему он подчинился требованиям прекратить нарушать правила, он ответил: «Большие данные дешевы, а вот адвокаты — нет».

В этой главе вы познакомитесь с законами США (и некоторыми международными законами), имеющими отношение к веб-скрапингу, и узнаете, как оценить степень законности и этичности веб-скрапинга в той или иной ситуации.

Прежде чем приступить к чтению раздела ниже, констатирую очевидный факт: я не юрист, я инженер-программист. Не интерпретируйте то, что вы прочтете здесь или в любой другой главе этой книги, как профессиональную юридическую консультацию. Я действительно считаю, что могу компетентно обсуждать законность и этичность веб-скрапинга. Однако прежде, чем начинать какие-либо юридически неоднозначные проекты по веб-копированию, вам следует проконсультироваться с юристом (а не с инженером-программистом).

Цель главы — дать вам основы, которые позволят изучать и обсуждать различные аспекты законности веб-скрапинга, такие как интеллектуальная собственность, несанкционированный доступ к компьютеру и использование

сервера. Однако это не заменит вам настоящую юридическую консультацию.

Торговые марки, авторские права, патенты... спасите-помогите!

Пора пройти краткий курс по основам интеллектуальной собственности! Есть три основных ее типа: торговые марки (обозначаемые символом ТМ или ®), авторские права (вездесущий ©) и патенты (иногда обозначаемые текстом, где написано, что данное изобретение защищено патентом, или номером патента, но часто не обозначаемые вообще никак).

Патенты используются только для объявления прав собственности на изобретения. Нельзя запатентовать изображение, текст или любую другую информацию. Впрочем, некоторые патенты, например на программное обеспечение, менее материальны, чем то, что мы привыкли считать изобретениями, однако имейте в виду: патентуется *вещь* (или методика), а не информация, содержащаяся в патенте. Если вы не создаете что-то по чертежам, собранным с помощью веб-скрапинга, и не пользуетесь методом веб-скрапинга, который запатентовал кто-то другой, то вряд ли случайно нарушите какой-либо патент, просто выполнив веб-скрапинг сайта.

Торговые марки также вряд ли будут проблемой, однако их уже необходимо учитывать. Патентное ведомство США утверждает следующее:

Торговая марка — это слово, фраза, символ и/или дизайн, которые идентифицируют и отличают источник товаров одной компании от товаров другой. Знак обслуживания — это слово, фраза, символ и/или дизайн, который идентифицирует и отличает источник услуг, а не товаров. Термином «торговая

марка» часто обозначают не только сами торговые марки, но и знаки обслуживания.

Помимо традиционного брендинга с помощью слов и символов, который мы имеем в виду, когда говорим о торговых марках, бывают и другие описательные атрибуты. Это может быть, например, форма контейнера (вспомните бутылки Coca-Cola) или даже цвет (сразу вспоминается розовый цвет стекловолоконной изоляции Owens Corning Pink Panther).

В отличие от патентов, право собственности на торговую марку сильно зависит от контекста, в котором она используется. Например, захотев опубликовать пост в блоге и сопроводить его изображением логотипа Coca-Cola, я могу это сделать (если только не намекаю на то, что спонсором или автором моего поста является Coca-Cola). Захоти я изготовить новый безалкогольный напиток и поставить на упаковке такой же логотип, как у Coca-Cola, — это было бы явно незаконным использованием торговой марки. Точно так же я могу изобразить на упаковке своего нового безалкогольного напитка Розовую Пантеру, однако не могу задействовать этот же цвет при создании утеплителя для стен.

Закон об авторском праве. У торговых марок и патентов есть общая черта: их признают только в том случае, если они официально зарегистрированы. Вопреки распространенному мнению, это не относится к материалам, защищенным авторским правом. Что делает изображения, текст, музыку и т.п. защищенными авторским правом? Не предупреждение «Все права защищены», написанное внизу страницы, и вообще ничего, что отличало бы опубликованные материалы от неопубликованных. Каждый созданный вами материал автоматически подпадает под действие закона об авторском праве сразу же после того, как вы его создали.

Международным стандартом авторского права является Бернская конвенция об охране литературных и художественных произведений, получившая свое название благодаря городу Берн в Швейцарии, где была принята в 1886 году. Эта конвенция, по сути, гласит: все страны, которые являются ее участницами, должны признавать охрану авторских прав произведений, созданных гражданами других стран — участниц конвенции, так, как если бы они были гражданами данной страны. На практике это значит, что, будучи гражданином США, вы можете быть привлечены к ответственности в Соединенных Штатах за нарушение авторских прав на материалы, написанные кем-либо, скажем, во Франции (и наоборот).

Очевидно, что для веб-скраперов авторские права являются проблемой. Если я соберу контент чьего-то блога и опубликую его в своем, то с уверенностью могу рассчитывать на судебный процесс. К счастью, у меня есть несколько уровней защиты, которые позволяют сделать мой проект веб-скрапинга блога оправданным, в зависимости от того, как он функционирует.

Во-первых, защита авторских прав распространяется только на творческие произведения, но не на статистику или факты. К счастью, большая часть того, за чем охотятся веб-скраперы, — *именно* статистика и факты. Веб-скрапер, собирающий стихи со всего Интернета и отображающий всю эту поэзию на вашем личном сайте, пожалуй, нарушал бы закон об авторском праве, но веб-скрапер, который собирает информацию о частоте стихотворных публикаций в разное время, не нарушает закон. Поэзия в чистом виде — творческое произведение. Однако среднее количество слов в стихах, опубликованных на сайте в разные месяцы, — не творческое произведение, а фактические данные.

Материалы, которые публикуются дословно (в отличие от агрегированных или вычисленных на основе необработанных данных веб-скрапинга), могут не нарушать закон об авторском праве, если эти данные представляют собой цены, имена руководителей компаний или какую-либо другую фактическую информацию.

Даже материал, защищенный авторским правом, может быть использован напрямую — в разумных пределах — в соответствии с Законом о защите авторских прав в цифровую эпоху (Digital Millennium Copyright Act, DMCA). В Законе изложены некоторые правила автоматической обработки материалов, защищенных авторским правом. Текст DMCA длинный, со множеством подробных правил, регулирующих все: от электронных книг до телефонов. Тем не менее в нем есть два основных момента, которые могут иметь особое отношение к веб-скрапину.

- Вы в «зоне безопасности», если извлекаете информацию из источника, который, по вашему мнению, содержит только материалы, не защищенные авторским правом, однако пользователь предоставил защищенные сведения. Все в порядке, если вы удалили данные, защищенные авторским правом, сразу после получения уведомления.
- Собирая контент, вы не имеете права обходить меры безопасности (такие как защита паролем).

Короче говоря, вы никогда не должны публиковать защищенные авторским правом материалы напрямую, без разрешения первоначального автора или правообладателя. Если вы храните защищенный авторским правом материал, к которому у вас есть свободный доступ, в своей непубличной базе данных с целью анализа — все в порядке. А вот

опубликовав эту базу на своем сайте для просмотра или скачивания, вы поступите нехорошо. Если вы анализируете эту БД и публикуете статистику по количеству слов, список авторов в порядке плодовитости или выводите другие результаты метаанализа данных — это нормально. Сопровождение этого метаанализа несколькими избранными цитатами или краткими образцами данных с целью подтвердить свою точку зрения — вероятно, тоже допустимо, но для надежности стоит свериться с положением о правомерном использовании в своде законов США.

Посягательство на движимое имущество

Посягательство на движимое имущество принципиально отличается от того, что мы называем «нарушением закона о владении собственностью», в том смысле, что оно распространяется не на недвижимость или землю, а на движимое имущество (такое как сервер). Этот закон применяется в тех случаях, когда ваш доступ к собственности нарушается неким образом: вы не можете либо получить к ней доступ, либо использовать ее.

В эпоху облачных вычислений заманчиво забыть, что веб-серверы — это реальные, материальные ресурсы. Однако серверы не только состоят из дорогих компонентов. Их еще нужно хранить, контролировать, охлаждать и снабжать огромным количеством электроэнергии. По ряду оценок, 10 % мирового потребления электроэнергии приходится на компьютеры²⁹. (Если вы подсчитали, сколько потребляет вся ваша электроника, и все еще в чем-то сомневаетесь, то посмотрите на необозримые серверные фермы Google, которые необходимо подключать к крупным электростанциям.)

Серверы являются дорогостоящими ресурсами, однако с юридической точки зрения интересны следующим: веб-мастера, как правило, действительно *хотят*, чтобы люди применяли их ресурсы (то есть получали доступ к их сайтам); они лишь не хотят, чтобы пользователи потребляли *слишком много* ресурсов. Одно дело — посмотреть сайт через браузер, и совсем другое — запустить полномасштабную DDoS-атаку.

Есть три критерия, определяющих, совершил ли веб-скрапер посягательство на движимое имущество.

- *Отсутствие согласия.* Поскольку веб-серверы являются общедоступными, они, как правило, «дают согласие» и на веб-скрапинг. Однако на многих сайтах соглашения о предоставлении услуг прямо запрещают использование веб-скраперов. Кроме того, любые доставленные вам предупреждения о нарушении прав интеллектуальной собственности явно аннулируют это согласие.
- *Фактический вред.* Серверы стоят дорого. Помимо стоимости самого сервера, если из-за ваших веб-скраперов сайт выйдет из строя или ограничится его способность обслуживать других пользователей, то это может быть расценено как вред, который вы причинили.
- *Преднамеренность.* Если вы написали код, то наверняка знали, что он делает!

Чтобы на вас подали заявление о нарушении правил использования, вы должны соблюсти все три эти критерия. Однако если вы нарушите соглашение об условиях обслуживания, но не причините реального вреда, то не думайте, что застрахованы от судебных исков. Вы вполне можете нарушить закон об авторском праве, DMCA, Акт о

компьютерном мошенничестве и злоупотреблении (подробнее о нем см. далее) или еще какой-либо из множества других законов, применимых к веб-скраперам.

Придержите ваши боты!

Были времена, когда веб-серверы были гораздо мощнее, чем персональные компьютеры. Фактически под *сервером* всегда подразумевался *большой компьютер*. Сейчас положение немного изменилось. Например, частота процессора моего персонального компьютера составляет 3,5 ГГц, а объем оперативной памяти — 8 Гбайт. И напротив, средний вычислительный узел Amazon (на момент написания этой книги) имел 4 Гбайт оперативной памяти и около 3 Ггерц вычислительной мощности.

При хорошей скорости подключения к Интернету и наличии выделенного компьютера достаточно одного персонального компьютера, чтобы создать большую нагрузку на многие сайты и даже причинить им вред или полностью вывести из строя. Если речь не идет о скорой медицинской помощи, при которой только сбор всех данных с сайта Васи Пупкина за две секунды спасет жизнь пациента, — нет причин бомбардировать сайт своими запросами.

Управляемый бот никогда не завершает работу. Иногда лучше включить веб-краулер на ночь, чем днем или вечером, и на то есть следующие причины.

Имея в распоряжении около восьми часов, даже при черепашьей скорости две секунды на страницу можно просмотреть более 14 000 страниц. Когда время не является проблемой, нет соблазна увеличивать скорость веб-краулеров.

Если предположить, что целевая аудитория сайта географически находится там же, где и вы (для удаленных целевых аудиторий внесите соответствующие коррективы), то в ночное время нагрузка на сайт должна значительно снижаться. Это значит, что ваш сбор данных не совпадет с временем пикового трафика.

Вы экономите время — спите вместо того, чтобы то и дело проверять журналы на наличие новой информации. Подумайте о том, какой сюрприз вас будет ждать утром в виде порции свежих данных!

Рассмотрим следующие сценарии:

у вас есть веб-краулер, просматривающий сайт Васи Пупкина, собирая с него некоторые или все данные;

у вас есть веб-краулер, просматривающий несколько сотен небольших сайтов, собирая с них некоторые или все данные;

у вас есть веб-краулер, просматривающий очень большой сайт, такой как «Википедия».

В первом случае лучше оставить бот работать на ночь и медленно.

Во втором лучше обращаться к каждому сайту по очереди — вместо того чтобы работать медленно, перебирать их по одному. В зависимости от количества сканируемых сайтов это значит, что вы можете собирать данные с той скоростью, которую способны обеспечить ваше интернет-соединение и компьютер, и при этом нагрузка на каждый из удаленных серверов останется в разумных пределах. Это можно сделать программно, используя несколько потоков (так что каждый из них будет проверять свой сайт и приостанавливать свое выполнение), или же задействовать списки Python для отслеживания сайтов.

В третьем варианте нагрузка, которую ваше интернет-соединение и домашний компьютер могут создать для такого сайта, как «Википедия», вряд ли будет замечена или вызовет беспокойство. Но вот если вы используете распределенную сеть машин — тогда, очевидно, другое дело. Соблюдайте осторожность и по возможности обратитесь к представителю компании.

Акт о компьютерном мошенничестве и злоупотреблении

В начале 1980-х компьютеры начали выходить из академических аудиторий и проникать в мир бизнеса. Впервые вирусы и «черви» стали восприниматься как нечто большее, чем просто неудобство (или даже забавное хобби), — серьезное уголовное преступление, способное причинить материальный ущерб. Наконец, в 1986 году появился Акт о компьютерном

мошенничестве и злоупотреблении (Computer Fraud and Abuse Act, CFAA).

На первый взгляд может показаться, что этот закон касается только обычных злостных хакеров, распространяющих вирусы, однако он имеет серьезные последствия и для веб-скраперов. Представьте скрапер, который сканирует сеть в поисках форм аутентификации с легко угадываемыми паролями или же собирает государственные секреты, случайно оказавшиеся в скрытом, но доступном месте. В соответствии с CFAA все описанные действия являются незаконными (и это справедливо).

Закон определяет семь основных уголовных преступлений, которые можно кратко описать следующим образом:

- намеренный несанкционированный доступ к компьютерам, принадлежащим правительству США, и получение с них информации;
- намеренный несанкционированный доступ к компьютеру и получение финансовой информации;
- намеренный несанкционированный доступ к компьютеру, принадлежащему правительству США, повлиявший на использование этого компьютера правительством;
- намеренный доступ к любому защищенному компьютеру с попыткой обмана;
- намеренный доступ к компьютеру без разрешения и причинение ущерба этому компьютеру;
- разглашение или передача паролей либо информации об авторизации на компьютерах, используемых

правительством США, или компьютерах, которые оказывают влияние на межгосударственную или внешнюю торговлю;

- попытки вымогательства денег или «чего-либо ценного» путем нанесения ущерба или угрозы причинить вред любому защищенному компьютеру.

Короче говоря: держитесь подальше от защищенных компьютеров, не заходите на компьютеры (в том числе на веб-серверы), к которым у вас нет доступа, и особенно держитесь подальше от компьютеров, принадлежащих правительству или финансовым организациям^{[30](#)}.

Файл robots.txt и условия использования

С юридической точки зрения условия использования сайта и файлы robots.txt находятся на интересной территории. Если сайт общедоступен, то право веб-мастера заявлять о том, что ПО может или не может получить к нему доступ, спорно. Было бы странным заявить, что «для просмотра данного сайта вы можете использовать браузер, но не программу, которую сами написали с этой целью».

На большинстве сайтов в нижней части каждой страницы есть ссылка на страницу «Условия использования» (Terms of Service, TOS). Это нечто большее, чем просто правила для веб-краулеров и автоматического доступа; здесь часто содержатся сведения о том, какая информация собрана на веб-сайте, что с ней здесь делается, и, как правило, юридическое уведомление о том, что услуги, предоставляемые сайтом, не подразумевают неких явных или скрытых гарантий.

Если вас интересуют оптимизация сайта для поисковых систем (Search Engine Optimization, SEO) или технологии

поиска, то вы, вероятно, слышали о файле `robots.txt`. Зайдя почти на любой крупный сайт и поискав там файл `robots.txt`, его можно обнаружить в корневой веб-папке: <http://website.com/robots.txt>.

Синтаксис файлов `robots.txt` был разработан в 1994 году, еще во время первой волны технологии веб-поиска. Примерно в это же время поисковые системы, обыскивающие весь Интернет, такие как AltaVista и DogPile, начали всерьез конкурировать с простыми списками сайтов, организованными по темам, которые велись, в частности, на Yahoo!. Развитие поиска в Интернете привело к взрывному росту не только количества веб-краулеров, но и доступности для обычных людей информации, собираемой этими краулерами.

Несмотря на то что сегодня мы привыкли считать такой вид доступности чем-то само собой разумеющимся, некоторые веб-мастера были шокированы, когда информация, опубликованная ими глубоко в файловой структуре своего сайта, стала появляться на первых страницах результатов поиска в основных поисковых системах. В ответ появился синтаксис файлов `robots.txt` под названием Robots Exclusion Standard — «стандарт исключения для роботов».

В отличие от TOS, где о веб-краулерах часто говорится в широком смысле и на вполне человеческом языке, файл `robots.txt` может быть очень легко проанализирован и использован автоматизированными программами. На первый взгляд может показаться, что это идеальное средство окончательного решения проблемы нежелательных ботов, однако учтите следующее.

- Никакого официального руководящего органа по синтаксису `robots.txt` нет. Это всего лишь широко распространенное

и в целом хорошо соблюдаемое соглашение, но ничто не мешает кому угодно создать собственную версию файла `robots.txt` (кроме того факта, что ни один бот не сможет его распознать или выполнить, пока данный формат не станет популярным). Другими словами, мы имеем дело с широко распространенным соглашением — главным образом потому, что оно относительно простое и у компаний нет причин изобретать собственный стандарт или пытаться улучшить существующий.

- Нет способа заставить кого-либо выполнять файл `robots.txt`. Файл — лишь знак, говорящий: «Пожалуйста, не заходите в эти части сайта». Есть множество библиотек веб-скрапинга, соблюдающие условия `robots.txt` (хотя это часто всего лишь настройка по умолчанию, которую можно изменить). Кроме того, часто встречаются и другие препятствия для выполнения условий `robots.txt` (в конце концов, вам нужно провести веб-скрапинг и синтаксический анализ содержимого страницы и применить к этому контенту логику вашего кода), которые не позволяют просто продвигаться вперед и выполнять веб-скрапинг всех нужных вам страниц.

Синтаксис Robot Exclusive Standard очень прост. Как и в Python (и во многих других языках), комментарии здесь начинаются с символа `#`, заканчиваются символом новой строки и могут использоваться в любом месте файла.

Первая строка файла после всех комментариев должна начинаться с `User-agent:`, после чего указывается пользователь, к которому применяются следующие правила. Далее идет набор правил: `Allow:` или `Disallow:` в зависимости от того, разрешается ли боту заходить в данный

раздел сайта. Звездочка (*) является подстановочным знаком и может использоваться для описания User-agent или URL.

Если последовательность правил кажется противоречивой, то последнее правило имеет приоритет. Например:

```
# Welcome to my robots.txt file!
```

```
User-agent: *
```

```
Disallow: *
```

```
User-agent: Googlebot
```

```
Allow: *
```

```
Disallow: /private
```

В этом случае запрещается доступ любых ботов к любой точке сайта, за исключением бота Google, которому разрешается доступ куда угодно, кроме каталога **/private**.

В Twitter файл robots.txt содержит подробные инструкции для ботов Google, Yahoo!, «Яндекса», Microsoft, а также других ботов и поисковых систем, не относящихся ни к одной из предыдущих категорий. Раздел Google (который практически не отличается от разрешений для остальных категорий ботов) выглядит следующим образом:

```
# Google Search Engine Robot
```

```
User-agent: Googlebot
```

```
Allow: /?_escaped_fragment_
```

```
Allow: /?lang=
```

```
Allow: /hashtag/*?src=
```

```
Allow: /search?q=%23
```

```
Disallow: /search/realtime
```

```
Disallow: /search/users
```

```
Disallow: /search/*/grid
```

```
Disallow: /*?
```

```
Disallow: /*/followers
```

```
Disallow: /*/following
```

Обратите внимание: Twitter ограничивает доступ к разделам своего сайта, имеющим API. Поскольку у Twitter есть хорошо отрегулированный API (на котором можно зарабатывать деньги за счет лицензирования), в интересах компании запретить использование любых «самодельных API», собирающих информацию, самостоятельно сканируя сайт.

На первый взгляд файл, сообщающий веб-краулеру, куда нельзя заходить, может показаться ограничительным, однако на самом деле это просто счастье разработчика. Обнаружение файла `robots.txt`, который запрещает сбор данных определенного раздела сайта, по сути, можно расценить как сообщение веб-мастера о том, что он разрешает доступ веб-краулеров ко всем остальным разделам сайта (в конце концов, если бы это было не так, то вам бы в первую очередь ограничили доступ при написании `robots.txt`).

Например, раздел файла `robots.txt` из «Википедии», который относится к обычным веб-скраперам (в отличие от поисковых систем), на удивление снисходителен. Он даже доходит до того, что содержит доступный человеку текст приветствия ботов (это для нас!) и блокирует доступ только к нескольким страницам, таким как страницы аутентификации, поиска и «случайной статьи»:

```
#
```

```
# Friendly, low-speed bots are welcome viewing  
article pages, but not
```

```
# dynamically generated pages please.
```

```
#
# Inktomi's "Slurp" can read a minimum delay
# between hits; if your bot supports
# such a thing using the 'Crawl-delay' or
# another instruction, please let us
# know.
#
# There is a special exception for API
# mobileview to allow dynamic mobile web
# & app views to load section content.
# These views aren't HTTP-cached but use parser
# cache aggressively and don't
# expose special: pages etc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Disallow: /w/
Disallow: /trap/
Disallow: /wiki/Especial:Search
Disallow: /wiki/Especial%3ASearch
Disallow: /wiki/Special:Collection
Disallow: /wiki/Spezial:Sammlung
Disallow: /wiki/Special:Random
Disallow: /wiki/Special%3ARandom
Disallow: /wiki/Special:Search
Disallow: /wiki/Special%3ASearch
Disallow: /wiki/Spesial:Search
Disallow: /wiki/Spesial%3ASearch
Disallow: /wiki/Spezial:Search
Disallow: /wiki/Spezial%3ASearch
Disallow: /wiki/Specjalna:Search
```

Disallow: /wiki/Specjalna%3ASearch
Disallow: /wiki/Speciaal:Search
Disallow: /wiki/Speciaal%3ASearch
Disallow: /wiki/Speciaal:Random
Disallow: /wiki/Speciaal%3ARandom
Disallow: /wiki/Speciel:Search
Disallow: /wiki/Speciel%3ASearch
Disallow: /wiki/Speciale:Search
Disallow: /wiki/Speciale%3ASearch
Disallow: /wiki/Istimewa:Search
Disallow: /wiki/Istimewa%3ASearch
Disallow: /wiki/Toiminnot:Search
Disallow: /wiki/Toiminnot%3ASearch

При написании веб-краулеров лишь вам решать, станут ли они подчиняться правилам robots.txt. Однако я настоятельно советую это делать, особенно если ваши веб-краулеры будут проверять все сайты подряд.

Три веб-скрапера

Поскольку вариации веб-скрапинга безграничны, есть множество способов попасть в юридические неприятности. В данном разделе мы рассмотрим три случая, которые так или иначе стали нарушением закона, обычно применяемого к веб-скраперам, и узнаем, как именно закон был задействован в каждом из этих случаев.

eBay против Bidder's Edge и посягательство на движимое имущество

В 1997 году игрушки Beanie Babies пользовались бешеным успехом, технологический сектор бурлил, а онлайн-аукционы

были самой популярной новинкой Интернета. Компания под названием Bidder's Edge основала и разработала новый вид сайта метааукционов. Вместо того чтобы заставлять пользователя переходить с одного аукционного сайта на другой, сравнивая цены, она объединила данные всех аукционов для определенного продукта (например, только что выпущенной куклы Ферби или нового экземпляра Spice World) и давала ссылку на сайт, который предлагал самую низкую цену.

Компания Bidder's Edge добилась этого с помощью целой армии веб-скраперов, которые постоянно делали запросы на веб-серверы различных аукционных сайтов с целью получить информацию о цене и продукте. Самым крупным из всех интернет-аукционов был eBay, и Bidder's Edge посещала его серверы примерно 100 000 раз в день. Даже по сегодняшним меркам это большой трафик. По данным eBay, в то время он составлял 1,53 % от общего интернет-трафика, что, безусловно, не доставляло радости владельцам аукциона.

eBay направил Bidder's Edge письмо-предупреждение о нарушении прав интеллектуальной собственности с предложением лицензировать свои данные. Однако переговоры по этому лицензированию успеха не имели, и Bidder's Edge продолжала собирать данные с сайта eBay.

Тогда eBay попытался заблокировать IP-адреса, используемые Bidder's Edge. Было заблокировано 169 IP-адресов, но Bidder's Edge удалось обойти блокировки с помощью прокси-серверов (серверов, которые пересылают запросы от имени другого компьютера, но применяя собственный IP-адрес прокси-сервера). Вообразите, какое разочарование и неуверенность испытывали от этого решения обе стороны: Bidder's Edge постоянно искала новые прокси-серверы и покупала новые IP-адреса, по мере того как eBay

блокировал старые, а eBay приходилось поддерживать большие списки блокировки для брандмауэров (и нести дополнительные большие затраты на сравнение IP-адресов при проверке каждого пакета).

Наконец, в декабре 1999 года eBay подал на компанию Bidder's Edge в суд по поводу посягательства на движимое имущество.

Поскольку серверы eBay были реальными, физическими ресурсами, которыми владела компания, и то, как компания Bidder's Edge их эксплуатировала, нельзя было назвать нормальным, посягательство на движимое имущество казалось идеальным законом для применения в суде. На самом деле в наше время посягательство на движимое имущество постоянно используется в судебных исках по веб-скрапину и его принято считать законом, касающимся IT.

Суд постановил: чтобы выиграть дело о посягательстве на движимое имущество, eBay должен был доказать две вещи:

- у Bidder's Edge не было разрешения на использование ресурсов eBay;
- в результате действий Bidder's Edge компания eBay понесла финансовые потери.

Поскольку имелись запись писем от eBay с предупреждением о нарушении прав интеллектуальной собственности и отчеты IT-отдела компании об использовании серверов и фактических расходах на серверы, eBay было относительно легко это сделать. Конечно, крупные судебные баталии никогда не заканчиваются легко и быстро: были поданы встречные иски, оплачены услуги многих адвокатов, и

в итоге в марте 2001-го дело решилось вне суда на сумму, размер которой не раскрывается.

Значит ли это, что любое несанкционированное использование чужого сервера автоматически является посягательством на движимое имущество? Не обязательно. История с Bidder's Edge была крайним случаем; компания задействовала так много ресурсов eBay, что тому пришлось покупать дополнительные серверы, увеличить расходы на электроэнергию и, возможно, нанять дополнительный персонал (может показаться, что 1,53 % — не так уж много, но в крупных компаниях это может составить значительную сумму).

В 2003 году Верховный суд Калифорнии вынес решение по другому делу — Корпорация Intel против Хамиди, в котором бывший сотрудник Intel (Хамиди) отправлял сотрудникам Intel нежелательные для компании электронные письма через все ее серверы. Суд постановил следующее: «Заявление Intel отклонено не потому, что электронная корреспонденция, передаваемая через Интернет, обладает уникальным иммунитетом, а потому, что посягательство на движимое имущество — в отличие от только что упомянутых оснований для предъявления претензий — не может быть доказано в Калифорнии без доказательств причинения вреда личной собственности истца или его законным интересам».

По сути, Intel не удалось доказать, будто затраты на передачу шести электронных писем, разосланных Хамиди всем сотрудникам (в каждом из которых, что характерно, предусматривалась возможность удаления из списка рассылки Хамиди — по крайней мере он был вежлив!), причинили компании какой-либо финансовый вред. Затраты не лишили Intel никакой собственности и не помешали использованию собственности компании.

Соединенные Штаты Америки против Ауэрнхаймера и Акт о компьютерном мошенничестве и злоупотреблении

Если информация легкодоступна в Интернете для человека, использующего браузер, то маловероятно, что за автоматическим доступом к той же самой информации сразу последуют неприятности с федералами. Однако стоит любопытному человеку найти в системе безопасности небольшую утечку, как она может быстро вырасти в огромную дыру и стать гораздо более опасной, когда до нее доберутся автоматические веб-скраперы.

В 2010 году Эндрю Ауэрнхаймер (Andrew Auernheimer) и Дэниел Спитлер (Daniel Spitler) обратили внимание на интересную функцию iPad: при посещении сайта AT&T с этого устройства вас перенаправляли на URL, содержащий уникальный идентификационный номер вашего iPad:

`https://dcp2.att.com/OEPClient/openPage?ICCID=
<idNumber>&IMEI=`

Данная страница содержала форму аутентификации с адресом электронной почты пользователя, чей идентификационный номер содержался в URL. Это позволяло пользователям получить доступ к их учетным записям, просто введя пароль.

Несмотря на то что потенциальных идентификационных номеров iPad было очень много, при наличии достаточного количества веб-скраперов оказалось возможным перебрать все вероятные номера, притом собирая адреса электронной почты. Предоставляя пользователям эту удобную функцию аутентификации, AT&T, по сути, опубликовала в Интернете адреса электронной почты своих клиентов.

Ауэрнхаймер и Спитлер создали веб-скрапер, который собрал 114 000 этих адресов, в том числе частные адреса электронной почты знаменитостей, генеральных директоров корпораций и правительственных чиновников. Затем Ауэрнхаймер (но не Спитлер) отправил данный список и информацию о том, как он был получен, в интернет-таблоид Gawker Media, опубликовавший историю (но не список) под заголовком: «Наихудшее нарушение системы безопасности Apple: обнародована информация о 114 000 владельцах iPad».

В июне 2011 года ФБР провело обыск в доме Ауэрнхаймера в связи со сбором адресов электронной почты, хотя в итоге его арестовали по обвинению в торговле наркотиками. В ноябре 2012 года Ауэрнхаймер был признан виновным в хищении персональных данных и заговоре с целью получить доступ к компьютеру без разрешения, а затем был приговорен к 41 месяцу лишения свободы в федеральной тюрьме и выплате 73 000 долларов в качестве компенсации.

Дело Ауэрнхаймера привлекло внимание адвоката по гражданским правам Орина Керра (Orin Kerr), который присоединился к его адвокатской группе и обжаловал дело Ауэрнхаймера в апелляционном суде третьего округа. В итоге 11 апреля 2014 года (эти юридические процессы обычно занимают довольно много времени) суд третьего округа удовлетворил апелляцию, заявив следующее: «Осуждение Ауэрнхаймера по пункту 1 должно быть отменено, поскольку в соответствии с Актом о компьютерном мошенничестве и злоупотреблении (18 U.S.C. § 1030(a)(2)(C)) посещение общедоступного сайта не является несанкционированным доступом. AT&T не использовала пароли или какие-либо иные средства защиты для контроля доступа к адресам электронной почты своих клиентов. То, что AT&T субъективно желала, чтобы посторонние не обнаружили эти данные, или что

Ауэрнхаймер, гиперболизируя, охарактеризовал свои действия как “воровство”, не имеет значения. Компания настроила свои серверы таким образом, что информация была доступна каждому, и тем самым разрешила всем желающим просматривать эту информацию. Доступ к адресам электронной почты через открытый сайт AT&T был разрешен в соответствии с CFAA и поэтому не является преступлением».

Таким образом, в правосудии восторжествовало здравомыслие, Ауэрнхаймер был в тот же день освобожден из тюрьмы, и все жили долго и счастливо.

Хоть в итоге суд и вынес решение, что Ауэрнхаймер не нарушил Акт о компьютерном мошенничестве и злоупотреблении, ФБР провело обыск в его доме, а сам он потратил многие тысячи долларов на оплату юридических услуг, провел три года в залах заседаний и тюрьмах. Какие уроки можем извлечь из этого мы, разработчики веб-скраперов, чтобы избежать подобных ситуаций?

Веб-скрапинг любой конфиденциальной информации, будь то персональные данные (в данном случае адреса электронной почты), коммерческие или правительственные секреты, — вероятно, не то, с чем стоит связываться, не имея в под рукой телефона адвоката. Даже если эта информация общедоступна, подумайте вот о чем: «Могли бы обычные пользователи компьютера легко получить доступ к данной информации при желании ее увидеть? Компания действительно хочет, чтобы пользователи видели это?»

Мне неоднократно приходилось звонить в компании и сообщать им об уязвимостях, найденных в системе безопасности их сайтов и веб-приложений. Такое поведение способно творить чудеса: «Привет, я специалист по безопасности, и я обнаружила потенциальную уязвимость на вашем сайте. Не могли бы вы направить меня к кому-нибудь,

кому я могла бы сообщить об этом и решить проблему?» В вас не только немедленно признают (доброго) хакерского гения — вы также можете получить бесплатные подписки, денежные вознаграждения и многие другие вкусности!

Ко всему прочему то, что Ауэрнхаймер передал информацию Gawker Media (прежде чем об этом узнала AT&T) и устроил себе рекламу за счет найденной уязвимости, сделало его еще более привлекательной целью для адвокатов AT&T.

Обнаружив уязвимости в системе безопасности сайта, лучше сообщите о них владельцам данного сайта, а не СМИ. У вас может возникнуть желание написать сообщение в блоге и объявить об этом всему миру, особенно если проблема не будет решена немедленно. Однако помните: это ответственность компании, а не ваша. Лучшее, что вы можете сделать, — убрать веб-скраперы (и, насколько возможно, вообще уйти) с сайта!

Филд против Google: авторские права и robots.txt

Однажды адвокат Блейк Филд (Blake Field) подал иск против Google на том основании, что функция кэширования сайта Google нарушала закон об авторском праве, поскольку показывала копию книги Филда после того, как он удалил ее со своего сайта. Закон об авторском праве позволяет создателю оригинального творческого произведения контролировать его распространение. Аргумент Филда состоял в том, что кэширование Google (после того как Филд удалил книгу со своего сайта) лишило его возможности контролировать распространение его книги.



Веб-кэш Google

Сканируя сайты, веб-скраперы Google (также известные как боты Google) делают копии этих сайтов и размещают в Интернете. Доступ к данному кэшу может получить любой желающий, используя URL следующего формата:

```
http://webcache.googleusercontent.com/search?q=cache:http://pythonscraping.com/
```

Если сайт, который вы ищете или для которого хотите выполнить веб-скрапинг, недоступен, то можете проверить, нет ли у Google его полезной копии!

То, что Филд знал о функции кэширования Google и не предпринял никаких действий, ему не помогло. В конце концов, он мог бы запретить ботам Google кэшировать свой сайт, просто создав файл `robots.txt` с простыми указаниями о том, какие страницы веб-скраперам следует обрабатывать, а какие — нет.

И главное, суд постановил, что положение DMCA о зоне безопасности позволило Google легально кэшировать и отображать сайты, такие как сайт Филда: «[a] поставщик услуг не несет ответственности за денежную компенсацию... за нарушение авторских прав по причине промежуточного или временного хранения материалов в системе или сети, контролируемой или управляемой поставщиком услуг либо для него».

[29](http://ti.me/2IFOF3F) Walsh B. The Surprisingly Large Energy Footprint of the Digital Economy [UPDATE], TIME.com, August 14, 2013 (<http://ti.me/2IFOF3F>).

[30](https://bit.ly/39R5TcV) В 2019 г. апелляционный суд 9-го округа США принял решение, что скрапинг публичных сайтов не противоречит закону CFAA (<https://bit.ly/39R5TcV>). Что касается России, автоматизированный сбор информации с сайтов является законным, если в процессе не нарушается законодательство РФ. — *Примеч. ред.*

Движемся дальше

Сеть продолжает развиваться. Технологии, которые открыли нам доступ к изображениям, видео, текстам и другим файлам данных, постоянно изменяются и обновляются. Чтобы идти в ногу со временем, должен развиваться и набор технологий, используемых для веб-скрапинга данных.

Кто знает, что будет дальше? Возможно, из последующих редакций данной книги полностью исчезнет JavaScript как устаревшая и редко используемая технология, а вместо этого много внимания получит анализ голограмм HTML8. Но сохранится главное: образ мышления и общий подход, необходимые для успешного веб-скрапинга любого сайта (или того, что мы станем понимать под «сайтами» в будущем).

Столкнувшись с любым веб-проектом, всегда следует задать себе такие вопросы.

- На какой вопрос я хочу получить ответ или какую задачу хочу решить?
- Какие данные помогут мне этого достичь и где они находятся?
- Каким образом сайт отображает эти данные? Могу ли я точно определить, в какой части кода сайта содержится нужная информация?
- Как изолировать данные и извлечь их?
- Какую обработку или анализ необходимо выполнить, чтобы получить от этих данных еще больше пользы?

- Как улучшить этот процесс, сделать его более быстрым и надежным?

Вам также необходимо понять не только то, как использовать представленные в книге инструменты по отдельности, но и то, как они могут работать совместно для решения более масштабной проблемы. Иногда данные легкодоступны и хорошо отформатированы, что позволяет решить задачу с помощью простого веб-скрапера. В других случаях приходится хорошенько подумать.

Например, в главе 11 мы объединили библиотеку Selenium, позволяющую идентифицировать Ajax-изображения, загруженные в Amazon, и Tesseract, чтобы прочесть их с помощью OCR. В задаче «Шесть шагов по “Википедии”» мы использовали регулярные выражения с целью написать веб-краулер, который хранил информацию о ссылках в базе данных, а затем — графовый алгоритм, чтобы ответить на вопрос: «Каков кратчайший путь по ссылкам между страницами Кевина Бейкона и Эрика Айдля?»

Когда речь идет об автоматическом сборе данных в Интернете, неразрешимые задачи встречаются редко. Просто запомните: Интернет — это один гигантский API с не очень хорошим пользовательским интерфейсом.

Об авторе

Райан Митчелл (Ryan Mitchell) — старший инженер-программист в бостонской компании HedgeServ, в которой она разрабатывает API и инструменты для анализа данных. Райан окончила Инженерно-технический колледж им. Франклина В. Олина, имеет степень магистра в области разработки программного обеспечения и сертификат по анализу и обработке данных, полученный на курсах повышения квалификации при Гарвардском университете. До прихода в HedgeServ Райан трудилась в компании Abine, где разрабатывала веб-скраперы и средства автоматизации на Python. Регулярно выступает консультантом проектов по веб-скрапину для розничной торговли, сферы финансов и фармацевтики. По совместительству работает консультантом и внештатным преподавателем в Северо-Восточном университете и Инженерно-техническом колледже им. Франклина В. Олина.

Об обложке

Животное, изображенное на обложке, — степной ящер, или саванный панголин (*Smutsia temminckii*). Это одиночное млекопитающее, ведущее ночной образ жизни. Панголины — близкие родственники броненосцев, ленивцев и муравьедов. Они встречаются в Южной и Восточной Африке. На континенте водятся еще три вида этих ящеров, и все они находятся под угрозой исчезновения.

Взрослые панголины достигают в среднем 30–100 сантиметров в длину и весят от 15 до 33 килограммов. Внешне они похожи на броненосцев и покрыты защитными чешуйками темного, светло-коричневого или оливкового цвета. У молодых ящеров чешуйки розоватые. Когда панголину что-то угрожает, его чешуйки на хвосте превращаются в оружие нападения, они могут порезать и поранить противника. У панголинов также есть стратегия защиты, похожая на имеющуюся у скунсов, — они выделяют неприятно пахнущую кислоту из желез, расположенных возле ануса. Это служит предупреждением для потенциальных врагов, а также помогает панголину пометить территорию. Брюхо панголина покрыто не чешуей, а коротким мехом.

Подобно своим родичам-муравьедам, панголины придерживаются диеты, состоящей из муравьев и термитов. Их невероятно длинные языки позволяют извлекать еду из бревен и муравейников. Язык панголина длиннее его тела; в расслабленном состоянии он втягивается в грудную полость.

Поскольку панголины — одиночные животные, после взросления они живут в норах, которые уходят глубоко под землю. Эти норы прежде часто принадлежали трубкозубам и бородавочникам, а панголины просто захватывают

заброшенные жилища. Однако с помощью трех длинных изогнутых когтей на передних лапах ящер при необходимости может легко выкопать собственную нору.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения; все они важны для нашего мира. Чтобы больше узнать о том, как вы можете им помочь, посетите сайт **animals.oreilly.com**.

Изображение на обложке этой книги взято из книги *Royal Natural History* Ричарда Лидеккера (Richard Lydekker).