O'REILLY®

Третье издание

Парсинг с помощью Python

Веб-скрапинг в действии



SPRINT book

Райан Митчелл

THIRD EDITION

Web Scraping with Python

Data Extraction from the Modern Web

Ryan Mitchell

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

ТРЕТЬЕ ИЗДАНИЕ

Парсинг с помощью Python

Веб-скрапинг в действии

Райан Митчелл

SPRINT book 2025

ББК 32.988.02-018
УДК 004.774+004.43
М67

Митчелл Райан

М67 Парсинг с помощью Python. Веб-скрапинг в действии. 3-е межд. изд. — Астана: «Спринт Бук», 2025. — 352 с.: ил.

ISBN 978-601-08-4570-1

Если программирование — магия, то веб-скрапинг — настоящее колдовство. Написав простую автоматизированную программу, можно отправлять запросы веб-серверам, получать ответы с данными, а затем анализировать эти данные и извлекать необходимую информацию. Это обновленное третье издание книги не только познакомит вас с веб-скрапингом, но и послужит исчерпывающим руководством по сбору практически любых видов данных в современном Интернете. В части I основное внимание уделено механике веб-скрапинга: как с помощью Python отправлять запросы веб-серверам, обрабатывать ответы и автоматизировать взаимодействие с сайтами. В части II исследуются более конкретные инструменты и приложения, которые пригодятся при любом сценарии веб-скрапинга.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.774+004.43

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098145354 англ.

ISBN 978-601-08-4570-1

Authorized Russian translation of the English edition of Web Scraping with Python, 3rd Edition ISBN 9781098145354 © 2024 Ryan Mitchell. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
© Перевод на русский язык ТОО «Спринт Бук», 2024
© Издание на русском языке, оформление ТОО «Спринт Бук», 2024

Краткое содержание

Введение 12

Часть I
РАЗРАБОТКА ВЕБ-СКРАПЕРОВ

Глава 1. Как работает Интернет 20
Глава 2. Законность и этичность веб-скрапинга 35
Глава 3. Применение веб-скрапинга 53
Глава 4. Ваш первый веб-скрапер 63
Глава 5. Углубленный парсинг HTML 75
Глава 6. Разработка веб-краулеров 93
Глава 7. Модели веб-краулинга 109
Глава 8. Scrapy 126
Глава 9. Хранение данных 142

ЧАСТЬ II
УГЛУБЛЕННЫЙ ВЕБ-СКРАПИНГ

Глава 10. Чтение документов 166
Глава 11. Обработка «грязных» данных 180
Глава 12. Чтение и запись текстов на естественных языках 195
Глава 13. Краулинг форм и авторизация 217
Глава 14. Веб-скрапинг данных JavaScript 227
Глава 15. Краулинг с помощью API 244
Глава 16. Обработка изображений и распознавание текста 259
Глава 17. Как избежать ловушек веб-скрапинга 284
Глава 18. Тестирование сайтов с помощью веб-скраперов 299
Глава 19. Параллельный веб-скрапинг 311
Глава 20. Прокси-сервисы для веб-скрапинга 327

Об авторе 349
Иллюстрация на обложке 350

Оглавление

Введение 12
Что такое веб-скрапинг 13
Почему это называется веб-скрапингом 13
Об этой книге 15
Условные обозначения 16
Использование примеров кода 17
Благодарности 18
От издательства 18

Часть I
РАЗРАБОТКА ВЕБ-СКРАПЕРОВ

Глава 1. Как работает Интернет 20
Передача данных в сети 21
Физический уровень 22
Канальный уровень 22
Сетевой уровень 23
Транспортный уровень 23
Сеансовый уровень 24
Уровень представления 24
Прикладной уровень 25
HTML 25
CSS 27
JavaScript 29
Просмотр сайтов с помощью инструментов разработчика 31

Глава 2. Законность и этичность веб-скрапинга 35
Торговые марки, авторские права, патенты… спасите-помогите! 36
Закон об авторском праве 37
Посягательство на движимое имущество 40
Закон о компьютерном мошенничестве и злоупотреблении 43
Файл robots.txt и условия его использования 44

Три веб-скрапера 47
eBay против Bidder's Edge и посягательство на движимое имущество 48
Соединенные Штаты Америки против Ауэрнхаймера и Закон о компьютерном мошенничестве и злоупотреблении 50
Филд против Google: авторские права и robots.txt 52

Глава 3. Применение веб-скрапинга 53
Классификация проектов 53
Электронная коммерция 54
Маркетинг 55
Академические исследования 57
Разработка продукта 58
Туризм 59
Продажи 60
Скрапинг результатов поиска 61

Глава 4. Ваш первый веб-скрапер 63
Установка и использование Jupyter 63
Установка соединения 64
Знакомство с BeautifulSoup 66
Установка BeautifulSoup 67
Работа с BeautifulSoup 69
Надежное соединение и обработка исключений 71

Глава 5. Углубленный парсинг HTML 75
Еще одна тарелка BeautifulSoup 75
Функции find() и find_all() объекта BeautifulSoup 77
Другие объекты BeautifulSoup 79
Навигация по деревьям 80
Регулярные выражения 84
Регулярные выражения и BeautifulSoup 88
Доступ к атрибутам 89
Лямбда-выражения 90
Иногда молоток не нужен 91

Глава 6. Разработка веб-краулеров 93
Обход отдельного домена 93
Краулинг всего сайта 98
Сбор данных со всего сайта 101
Краулинг нескольких сайтов 103

Глава 7. Модели веб-краулинга 109
Планирование и определение объектов 110
Работа с различными макетами сайтов 113
Структурирование веб-краулеров 118
Краулинг с помощью поиска 118
Краулинг сайтов по ссылкам 121
Краулинг страниц нескольких типов 123
Размышления о моделях веб-краулеров 125

Глава 8. Scrapy 126
Установка Scrapy 126
Инициализация нового «паука» 127
Пишем простой веб-скрапер 128
«Паук» с правилами 129
Создание объектов Item 134
Вывод объектов Item 136
Динамический конвейер 137
Ведение журнала в Scrapy 140
Дополнительные ресурсы 141

Глава 9. Хранение данных 142
Медиафайлы 142
Хранение данных в формате CSV 145
MySQL 147
Установка MySQL 148
Несколько основных команд 150
Интеграция с Python 153
Приемы и рекомендуемые методики работы с базами данных 156
«Шесть шагов» в MySQL 159
Электронная почта 162

ЧАСТЬ II
УГЛУБЛЕННЫЙ ВЕБ-СКРАПИНГ

Глава 10. Чтение документов 166
Кодировка документов 166
Текст 167
Кодирование текста и глобальный Интернет 168

CSV ... 172
Чтение CSV-файлов ... 172
PDF ... 174
Microsoft Word и файлы .docx ... 176

Глава 11. Обработка «грязных» данных ... 180
Очистка текста ... 181
Нормализация текста ... 185
Очистка данных с помощью Pandas ... 188
Очистка ... 189
Индексирование, сортировка и фильтрация ... 193
Дополнительные источники информации о Pandas ... 194

Глава 12. Чтение и запись текстов на естественных языках ... 195
Обобщение данных ... 196
Модели Маркова ... 200
Шесть шагов по «Википедии»: заключение ... 204
Natural Language Toolkit ... 207
Установка и настройка ... 207
Статистический анализ с помощью NLTK ... 209
Лексикографический анализ с помощью NLTK ... 211
Дополнительные ресурсы ... 216

Глава 13. Краулинг форм и авторизация ... 217
Библиотека Python Requests ... 217
Отправка простейшей формы ... 218
Переключатели, флажки и поля ввода ... 221
Передача файлов и изображений ... 222
Обработка данных авторизации и параметров cookie ... 223
Базовая аутентификация доступа через HTTP ... 225
Другие проблемы с формами ... 226

Глава 14. Веб-скрапинг данных JavaScript ... 227
Краткое введение в JavaScript ... 228
Популярные библиотеки JavaScript ... 229
Ajax и динамический HTML ... 232
Выполнение JavaScript в Python с помощью Selenium ... 233
Установка и использование Selenium ... 233
Селекторы Selenium ... 236

Ожидание окончания загрузки ... 237
Синтаксис XPath ... 239
Дополнительные веб-драйверы Selenium ... 240
Обработка перенаправлений ... 241
Последнее замечание о JavaScript ... 243

Глава 15. Краулинг с помощью API ... 244
Краткое введение в API ... 244
API и HTTP-методы ... 246
Подробнее об ответах на API-запросы ... 248
Парсинг JSON ... 249
Недокументированные API ... 250
Поиск недокументированных API ... 252
Документирование недокументированных API ... 253
Объединение API с другими источниками данных ... 254
Дополнительные сведения об API ... 257

Глава 16. Обработка изображений и распознавание текста ... 259
Обзор библиотек ... 260
Pillow ... 260
Tesseract ... 261
Обработка хорошо отформатированного текста ... 264
Автоматическая коррекция изображений ... 266
Веб-скрапинг текста, представленного в виде изображений на сайтах ... 270
Чтение капчи и обучение Tesseract ... 273
Обучение библиотеки Tesseract ... 274
Получение капчи и отправка решений ... 281

Глава 17. Как избежать ловушек веб-скрапинга ... 284
Этический момент ... 285
Выдать сценарий за человека ... 286
Настройте заголовки ... 286
Обработка данных cookie с помощью JavaScript ... 288
Создание цифровых отпечатков TLS ... 290
Своевременность — наше всё ... 293
Основные средства защиты форм ... 293
Значения скрытых полей ввода ... 294
Как избежать ловушек ... 295
Чек-лист: как выдать программу за человека ... 297

Глава 18. Тестирование сайтов с помощью веб-скраперов 299
Основы тестирования 300
Что такое модульные тесты 300
Модуль unittest 301
Тестирование «Википедии» 303
Тестирование с помощью Selenium 305
Взаимодействие с сайтом 306

Глава 19. Параллельный веб-скрапинг 311
Процессы или потоки 311
Многопоточный веб-краулинг 312
Состояния гонки и очереди 315
Дополнительные возможности модуля threading 318
Многопроцессный веб-краулинг 320
Пример 322
Обмен данными между процессами 323
Многопроцессный веб-краулинг: еще один подход 325

Глава 20. Прокси-сервисы для веб-скрапинга 327
Зачем использовать удаленные серверы 327
Как избежать блокировки IP-адреса 328
Переносимость и расширяемость 329
Tor 330
PySocks 331
Удаленный хостинг 332
Запуск веб-скрапера с учетной записи веб-хостинга 332
Запуск из облака 333
Движемся дальше 335
Прокси-сервисы для скрапинга 336
ScrapingBee 337
ScraperAPI 339
Oxylabs 342
Zyte 345
Дополнительные ресурсы 348

Об авторе 349

Иллюстрация на обложке 350

Введение

Если человек не слишком хорошо знаком с программированием, оно ему может показаться чем-то вроде волшебства. Но если программирование — волшебство, то *скрапинг* и *парсинг* — настоящее колдовство: применение магии для создания невероятно впечатляющих и полезных, но при этом удивительно простых чудес.

Работая инженером-программистом, я обнаружила, что веб-скрапинг — одна из немногих областей программирования, способных восхитить как разработчиков, так и обычных пользователей. Возможность легко создать простой бот, который будет собирать данные и передавать их через терминал или сохранять в базе данных, не перестает вызывать некий трепет от осознания своих способностей, независимо от того, сколько раз вам приходилось делать это раньше.

К сожалению, общаясь с другими программистами на тему скрапинга и парсинга, я обнаружила, что вокруг нее возникает много недопонимания и путаницы. Одни считают это не вполне законным (и они ошибаются), другие не умеют обрабатывать страницы, содержащие большое количество кода JavaScript или требующие регистрации. Многие не знают, как начать крупный проект по скрапингу или даже где искать нужные данные. Цель книги — дать ответы на многие из подобных часто встречающихся 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 рассмотреть не только общие принципы, но и конкретные примеры, охватывающие практически все способы сбора данных, с которыми вы можете столкнуться. 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 (**https://github.com/REMitchell/python-scraping**), где их можно просмотреть и скачать.

Что такое веб-скрапинг

Автоматизированный сбор данных в Интернете существует почти столько же, сколько и сам Интернет. Хотя термин *веб-скрапинг* не является новым, еще несколько лет назад эту методику чаще называли *анализом интерфейсных данных (screen scraping)*, *интеллектуальным анализом данных (data mining)*, *сбором веб-данных (web harvesting)* и т. п. Похоже, что наконец-то все пришли к единому мнению и предпочли называть это *веб-скрапингом*, поэтому я буду использовать данный термин в контексте всей книги, хотя специализированные программы, которые позволяют просматривать несколько веб-страниц, я буду называть *веб-краулерами*, а программы, предназначенные для собственно веб-скрапинга, — *ботами*.

Теоретически веб-скрапинг — это сбор данных с помощью любых средств, кроме программ, использующих API (или через браузер пользователя). Чаще всего для этого пишут программу, которая автоматически отправляет запрос на веб-сервер, получает данные (обычно в формате HTML или других форматах веб-страниц), а затем парсирует их для извлечения полезной информации.

На практике веб-скрапинг и парсинг включают в себя широкий спектр методов и технологий программирования, таких как анализ данных, синтаксический анализ естественного языка и информационная безопасность. Поскольку эта область очень широка, в первой части книги будут рассмотрены фундаментальные основы веб-скрапинга и краулинга, а во второй — более углубленные темы. Я рекомендую всем читателям внимательно изучить первую часть и по мере необходимости погружаться в более специализированные разделы второй части.

Почему это называется веб-скрапингом

Выходить в Интернет только через браузер — значит упускать массу возможностей. Несмотря на то что браузеры (кроме прочего) удобны для выполнения сценариев JavaScript, вывода изображений и представления объектов в понятной для пользователя форме, веб-скраперы гораздо быстрее справляются со сбором и обработкой больших объемов данных. Вместо того чтобы просматривать страницу за страницей в узком окне браузера, можно читать сразу целые базы данных, в которых хранятся тысячи или даже миллионы страниц.

Кроме того, веб-скраперы позволяют заглядывать в места, недоступные обычным поисковым системам. Так, если в Google ввести запрос «самые дешевые авиарейсы в Бостон», можно получить множество ссылок на рекламные объявления и популярные сайты поиска авиабилетов. Google знает только то, что содержится на страницах этих сайтов, а вовсе не точные результаты различных запросов,

введенных в приложение поиска рейсов. Однако правильно разработанный веб-скрапер способен построить график зависимости стоимости перелета в Бостон от времени с учетом данных разных сайтов и определить самые выгодные даты покупки билета.

Вы спросите: «Разве API не создаются специально для сбора данных?» (О том, что такое API, см. главу 15.) Действительно, возможности API бывают просто фантастическими, если удастся найти тот из них, который соответствует вашим запросам. Прежде всего они предназначены для построения удобного потока хорошо отформатированных данных из одной компьютерной программы в другую. Для многих типов данных, которые вам могут понадобиться, существуют готовые API, например для постов Twitter или страниц «Википедии». Как правило, предпочтительнее использовать уже существующий API вместо создания бота для получения тех же данных. Однако нужный API может отсутствовать или 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, к ним можно получить доступ с помощью сценария на Python и затем сохранить в базе данных. А с сохраненными данными можно делать практически все что угодно.

Очевидно, что доступ к практически неограниченным данным имеет множество чрезвычайно полезных применений: системы прогнозирования спроса, машинного перевода и даже медицинской диагностики развиваются благодаря возможности извлекать и анализировать данные с новостных сайтов, из переведенных текстов и форумов о здоровье соответственно.

Даже в мире искусства веб-скрапинг расширяет возможности для творчества. В 2006 году проект *We Feel Fine* («Мы чувствуем себя прекрасно») (**http://wefeelfine.org/**) Джонатана Харриса и Сэпа Камвара собрал из нескольких англоязычных блогов фразы, начинающиеся со слов I feel или I am feeling («я чувствую»,

«я ощущаю»). В итоге получилась популярная 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 позволяет делать 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 парсингу 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 книгу. Пожалуйста, не используйте ее в качестве учебника по основам Python. Учитывая это, я постаралась представить все концепции и примеры кода на начальном и среднем уровне программирования на Python, чтобы они были понятны широкому кругу читателей. Поэтому иногда здесь будут встречаться пояснения более сложных аспектов программирования на Python и общих вопросов информатики. Если вы более опытный читатель, то не стесняйтесь заглянуть в эти разделы!

Для тех, кто хотел бы изучить Python подробнее, рекомендую учебник *Introducing Python* Билла Любановича[1] — это хорошее, хотя и довольно объемное руководство. Если у вас не хватит на него времени, советую посмотреть видеоуроки *Introduction to Python* Джессики Маккеллар (O'Reilly) (http://oreil.ly/2HOqSNM) — это отличный ресурс. Мне также понравилась книга *Think Python* моего бывшего профессора Аллена Дауни[2] (O'Reilly) (http://oreil.ly/2fjbT2F). Она особенно хороша для новичков в программировании. Это учебник не только по языку Python, но и в целом по информатике, а также общим концепциям разработки ПО.

Технические книги часто посвящены какому-то одному языку или технологии. Однако веб-скрапинг — весьма обширная тема, требующая знаний в области баз данных, веб-серверов, HTTP, HTML, интернет-безопасности, обработки изображений, анализа данных и других инструментов. В книге я постараюсь охватить все эти и другие темы с точки зрения сбора данных. Это не значит, что здесь они

[1] *Любанович Б.* Простой Python. Современный стиль программирования. — 2-е изд. — СПб.: Питер, 2020.

[2] *Дауни А.* Основы Python. Научитесь думать как программист. 2021.

будут раскрыты полностью, однако я намерена рассмотреть их достаточно подробно, чтобы вы начали создавать веб-скраперы!

В части I детально рассматриваются веб-скрапинг и веб-краулинг. Особое внимание уделяется использованию нескольких полезных библиотек. Данная часть вполне может послужить подробным справочником по этим библиотекам и методикам (за некоторыми исключениями, по которым будут предоставлены дополнительные ссылки). Описанные здесь приемы окажутся полезны всем, кто пишет веб-скраперы, независимо от конкретной цели и области применения.

В части II раскрываются дополнительные темы, которые также могут пригодиться при написании веб-скраперов, но не всегда и не любых. К сожалению, данные темы слишком обширны, для того чтобы изучить их 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, например, только описание веб-скрапинга или другую нужную вам информацию. Если концепция или фрагмент кода основываются на данных, приведенных в предыдущей главе, то я даю ссылку на раздел, в котором это было рассмотрено.

Условные обозначения

В данной книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`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 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.



Этот 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**.

Настоящая книга предназначена для того, чтобы помочь вам выполнить свою работу. Все примеры кода из книги разрешается свободно использовать. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вы не обязаны обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не настаиваем на включении ссылки на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу **permissions@oreilly.com**.

К сожалению, печатные 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 привести к неожиданным результатам. Если вы захотите использовать примеры кода, то не копируйте их непосредственно из книги, а скачайте из репозитория GitHub. Я и читатели этой книги, которые решили внести свой вклад (включая, возможно, вас!), постараемся поддерживать хранилище в актуальном состоянии, вовремя внося необходимые изменения и дополнения.

Помимо примеров кода, в книге часто приводятся команды терминала, демонстрирующие установку и запуск программного обеспечения. Как правило, эти команды предназначены для операционных систем на базе Linux, но большинство из них применимы и для Windows с правильно настроенной средой Python и установленным pip. В книге также представлены инструкции для всех основных операционных систем или внешние ссылки для пользователей Windows, которые позволят выполнить задачу.

Благодарности

Лучшие продукты часто появляются благодаря многочисленным отзывам пользователей. Эта книга тоже никогда не появилась бы в каком-либо полезном виде без помощи многих сотрудников, руководителей и редакторов. Спасибо сотрудникам O'Reilly за их удивительную поддержку в этой немного необычной теме, моим друзьям и семье, которые давали советы и мирились с импровизированными чтениями, а также моим коллегам из Gerson Lehrman Group, которым я, вероятно, теперь должна много часов работы.

Спасибо моим редакторам — Саре Хантер, Джону Обеленусу и Трейси Ларвенц — за отзывы и советы, иногда жесткие, но справедливые. Довольно много разделов и примеров кода появились в книге именно благодаря их вдохновляющим предложениям.

На создание первых двух глав, а также внесение множества изменений и дополнений, включенных в третье издание, меня вдохновил Брайан Шпехт. Наследие, которое он оставил, гораздо обширнее, чем он сам мог бы предполагать, но область, которую он хотел заполнить, еще обширнее.

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

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

ЧАСТЬ I

Разработка веб-скраперов

Первая часть этой книги посвящена базовым механизмам веб-скрапинга: как использовать Python для запроса информации с веб-сервера, обработать полученный ответ и начать взаимодействовать с сайтом в автоматическом режиме. В итоге вы сможете легко перемещаться по Интернету, создавая скраперы, способные переходить из одного домена в другой, собирать и сохранять информацию для последующего использования.

Честно говоря, веб-скрапинг — фантастическая область, позволяющая получать огромные доходы при относительно небольших начальных инвестициях. Примерно 90 % проектов веб-скрапинга, с которыми вы столкнетесь, будут основаны на методах, описанных в следующих шести главах. В этой части показано, как обычные (хотя и технически подкованные) люди представляют себе работу веб-скраперов:

- извлечение HTML-данных из указанного домена;
- парсинг этих данных для получения требуемой информации;
- сохранение результатов;
- переход на другую страницу и повторение процедуры.

Прочитав первую часть, вы получите мощную основу, которая позволит вам перейти к более сложным проектам, описанным в части II. Однако это не означает, что первая половина книги менее важна, чем вторая. При создании веб-скраперов вам придется каждый день использовать почти всю информацию, изложенную в первой части данной книги!

ГЛАВА 1

Как работает Интернет

Мне встречалось не так много людей, которые по-настоящему знают, как работает Интернет, и я определенно не принадлежу к их числу.

Многим из нас достаточно простого набора ментальных абстракций, позволяющих пользоваться Интернетом ровно настолько, насколько это необходимо. Даже программисты могут пользоваться весьма ограниченными абстракциями, которых достаточно для решения их задач.

Из-за ограниченного объема книги и знаний автора эта глава также основывается на подобных абстракциях. В ней описывается механизм работы Интернета и веб-приложений в том объеме, который необходим для веб-скрапинга (и, возможно, немного больше).

Эта глава в некотором смысле откроет перед вами мир, где работают веб-скраперы: его обычаи, практики, протоколы и стандарты, к которым мы будем возвращаться на протяжении всей книги.

Если ввести URL в адресную строку браузера и нажать **Enter**, то как по волшебству появятся текст, изображения и мультимедийные объекты. И это волшебство наблюдают миллиарды разных людей каждый день. Они посещают те же сайты, используют те же приложения и часто получают медиаконтент и текст, созданные специально для них.

Все эти миллиарды людей используют различные типы устройств и приложений, написанных разными разработчиками из разных (часто конкурирующих!) компаний.

Удивительно, но не существует единого руководящего органа, который мог бы регулировать Интернет и координировать его развитие, обладая при этом какой-либо юридической силой. Вместо этого разные части Интернета управляются несколькими различными организациями, которые со временем развивались по принципах «от случая к случаю» и «по мере необходимости».

Конечно, если вы *откажетесь* соблюдать стандарты, принятые этими организациями, ваши веб-разработки могут просто... перестать работать. Например, ваш сайт не отображается в популярных браузерах, значит, люди не будут его посещать. Если данные, отправляемые вашим маршрутизатором, будут непонятны другим маршрутизаторам, то эти данные будут проигнорированы.

Веб-скрапинг — по сути, замена веб-браузера приложением собственной разработки. Поэтому так важно понимать стандарты и основы, на которых построены браузеры. Ваши будущие веб-скраперы должны поддерживать, а иногда даже нарушать обычаи и практики Интернета.

Передача данных в сети

На заре развития телефонии каждый телефонный аппарат был физически подключен проводом к центральному коммутатору. Решив позвонить другу, вы снимали трубку, просили оператора соединить вас, и тот физически (с помощью штекеров и разъемов) создавал прямое соединение между вашим телефоном и телефоном вашего друга.

Междугородние звонки стоили дорого, а установление соединения могло занять несколько минут. Чтобы позвонить из Бостона в Сиэтл, требовалось скоординировать действия операторов коммутаторов по всей территории США для создания единой огромной проводной цепи, напрямую соединяющей ваш телефон с телефоном другого абонента.

Сегодня нам не нужно временное выделенное соединение — мы можем совершать видеозвонки из дома в любую точку мира через устойчивую паутину проводов. Теперь не провод передает данные, они сами направляют себя, используя процесс, называемый *коммутацией пакетов*. Многие технологии внесли свой вклад в создание того, что мы называем Интернетом, но коммутация пакетов — это технология, положившая начало всему.

В сети с коммутацией пакетов отправляемое сообщение делится на упорядоченную последовательность пакетов, каждый из которых имеет адрес отправителя и получателя. Эти пакеты динамически пересылаются в тот или иной пункт назначения в сети на основе их адреса. Вместо того чтобы следовать по единственному выделенному соединению от получателя к отправителю вслепую, пакеты могут идти по любому пути, который выберет сеть. Фактически пакеты, составляющие одно сообщение, могут идти по разным маршрутам и прибывать в компьютер адресата не по порядку.

Если старые телефонные сети были подобны канатной дороге, доставляющей пассажиров из одного пункта на вершине холма в другой пункт внизу, то сети

с коммутацией пакетов подобны системе автомагистралей, по которым разные автомобили могут ехать в разные пункты назначения и обратно, пользуясь одними и теми же дорогами.

Современную сеть с коммутацией пакетов обычно описывают, используя модель взаимодействия открытых систем (Open Systems Interconnection, OSI), которая состоит из семи уровней маршрутизации, кодирования и обработки ошибок.

1. Физический уровень.
2. Канальный уровень.
3. Сетевой уровень.
4. Транспортный уровень.
5. Сеансовый уровень.
6. Уровень представления.
7. Прикладной уровень.

Большинство разработчиков веб-приложений почти не покидают прикладной уровень. Именно ему уделяется наибольшее внимание в данной книге. Однако, занимаясь веб-скрапингом, важно иметь хотя бы общее представление и о других уровнях. Например, сканирование отпечатков пальцев по протоколу TLS, которое будет рассмотрено в главе 17, представляет собой метод обнаружения веб-скрапинга, работающий на транспортном уровне.

Кроме того, знание особенностей организации и передачи данных на всех уровнях может помочь устранить ошибки в ваших веб-приложениях и веб-скраперах.

Физический уровень

На *физическом уровне* происходит передача информации по проводам Ethernet с помощью электрических сигналов. К важным особенностям относятся уровни напряжения, представленные кодами 1 и 0, частота их изменения, а также порядок интерпретации радиоволн, передающихся через устройства Bluetooth и Wi-Fi.

Этот уровень не требует программирования или цифровых инструкций и основан исключительно на физических и электрических стандартах.

Канальный уровень

На *канальном уровне* задается порядок передачи информации между двумя узлами локальной сети, например между компьютером и маршрутизатором. Определяются начало и конец передачи единого блока данных, и обеспечи-

вается исправление ошибок, если в процессе передачи данные были потеряны или искажены.

На этом уровне пакеты упаковываются в дополнительные «цифровые конверты», которые содержат информацию о маршрутизации и называются *фреймами*. Когда информация, содержащаяся во фрейме, становится не нужна, она разворачивается и отправляется по сети в виде пакета.

Важно отметить, что на канальном уровне все устройства в сети получают одни и те же данные — никакой коммутации или управления передачей пакетов адресатам на этом уровне не происходит. Однако устройства, не являющиеся получателями данных, обычно игнорируют их и ждут, пока не придет то, что предназначено для них.

Сетевой уровень

На *сетевом уровне* выполняется коммутация пакетов, и именно тут задействован Интернет. Происходит пересылка пакетов с вашего компьютера на маршрутизатор и далее на другие устройства, находящиеся за пределами локальной сети.

Сетевой уровень реализуется через интернет-протокол (Internet Protocol, IP) — часть широко известной связки протокола управления передачей и протокола Интернета (Transmission Control Protocol/Internet Protocol, TCP/IP). IP — это источник IP-адресов. Например, прямо сейчас мой IP-адрес в глобальной сети Интернет: 173.48.178.92. Это позволяет мне обмениваться данными с любыми другими имеющими IP-адрес компьютерами в мире.

Транспортный уровень

На *транспортном уровне* осуществляется подключение определенных сервисов или приложений, работающих на одном компьютере, к конкретным сервисам и приложениям, работающим на другом компьютере. Здесь также происходит исправление ошибок и выполняются повторные попытки передачи данных, если это необходимо.

TCP очень взыскателен и будет запрашивать недостающие пакеты в потоке до тех пор, пока все они не будут получены. TCP часто используется для передачи файлов, когда важно, чтобы все пакеты были доставлены корректно и из них получился правильно сформированный файл.

Протокол пользовательских дейтаграмм (User Datagram Protocol, UDP), наоборот, проигнорирует потерю пакетов, чтобы сохранить поток данных. Он часто

используется для видео- или аудиоконференций, где временное снижение качества передачи данных предпочтительнее, чем задержка в разговоре.

Поскольку у разных приложений на вашем компьютере могут быть разные требования к надежности данных (например, телефонный звонок во время загрузки файла), на транспортном уровне также используется номер порта. Операционная система назначает каждому приложению или сервису, работающему на компьютере, определенный номер порта, куда они отправляют и откуда получают данные.

Номер порта обычно записывается как число после IP-адреса, отделенное от последнего двоеточием. Например, запись 71.245.238.173:8080 говорит о том, что на компьютере с IP-адресом 71.245.238.173 работает приложение, которому операционная система назначила порт 8080.

Сеансовый уровень

На *сеансовом уровне* происходят открытие и закрытие сеанса между двумя приложениями. Поддержка сеансов позволяет получать информацию о том, какие данные были отправлены, а какие нет и с кем взаимодействует компьютер. Сеанс обычно остается открытым до тех пор, пока выполняются запросы, а затем закрывается.

Сеансовый уровень позволяет повторить передачу данных в случае кратковременного сбоя или отключения.



Сеансы на разных уровнях

Сеансы на сеансовом уровне модели OSI — это не то же, что сеансы веб-приложений, о которых обычно говорят веб-разработчики. Сеанс в веб-приложении — это понятие прикладного уровня, реализуемое программным обеспечением браузера.

Информация о сеансе прикладного уровня хранится в браузере, пока она востребована или пока пользователь не закроет окно браузера. На сеансовом уровне модели OSI сеанс обычно длится ровно столько, сколько необходимо для передачи одного файла!

Уровень представления

На *уровне представления* входящие данные преобразуются из строк символов в формат, понятный приложению. Здесь также происходят кодирование символов и сжатие данных. Уровень представления различает форматы входных данных, получаемых приложением (например, PNG или HTML), выполняет необходимые преобразования и передает преобразованные данные прикладному уровню.

Прикладной уровень

На *прикладном уровне* интерпретируются данные, закодированные на предыдущем уровне, и передаются приложению. Иными словами, уровень представления связан с преобразованием и идентификацией данных, а прикладной — с выполнением задач. Например, HTTP с его методами и статусами является протоколом прикладного уровня. Более банальные JSON и HTML (поскольку фактически являются типами файлов и определяют способ кодирования данных) — это протоколы уровня представления.

HTML

Основная функция веб-браузера — отображение документов HTML (HyperText Markup Language — язык разметки гипертекста). HTML-документы — это файлы с расширением `.html` или реже `.htm`.

Как и простые текстовые файлы, HTML-файлы содержат самые обычные текстовые символы, как правило, ASCII (см. раздел «Кодирование текста и глобальный Интернет» главы 10). Это означает, что их можно открыть и прочитать в любом текстовом редакторе.

Вот пример простого HTML-файла:

```
<html>
  <head>
    <title>A Simple Webpage</title>
  </head>
  <body>
    <!-- Это текст комментария, он не выводится браузером -->
    <h1>Hello, World!</h1>
  </body>
</html>
```

Файлы HTML — это особый тип файлов XML (Extensible Markup Language — расширяемый язык разметки). Строки, начинающиеся с `<` и заканчивающиеся `>`, называются *тегами*.

Стандарт XML определяет понятия открывающих, или *начальных*, тегов, таких как `<html>`, и закрывающих, или *конечных*, которые начинаются с пары символов `</`, например `</html>`. Между начальным и конечным тегами находится их *содержимое*.

Если тег не имеет содержимого, то он может быть записан в так называемой самозакрывающейся форме:

```
<p />
```

Тег может иметь атрибуты, которые записываются в форме `имяАтрибута="значение атрибута"`, например:

```
<div class="content">
  Lorem ipsum dolor sit amet, consectetur adipiscing elit
</div>
```

Здесь тег `div` имеет атрибут `class` со значением `main-content`.

HTML-*элемент* имеет начальный (открывающий) тег, иногда с дополнительными атрибутами, некоторое содержимое и конечный (закрывающий) тег. Элемент также может содержать другие элементы, которые в этом случае являются *вложенными*.

Стандарт XML определяет базовые понятия (тег, содержимое, атрибуты и их значения), а HTML — перечень допустимых тегов, их назначение, что они могут содержать и как должны интерпретироваться и отображаться браузером.

Например, стандарт HTML определяет использование атрибутов `class` и `id`, которые часто применяются для организации и управления отображением элементов HTML:

```
<h1 id="main-title">Some Title</h1>
<div class="content">
  Lorem ipsum dolor sit amet, consectetur adipiscing elit
</div>
```

Один и тот же класс может быть присвоен сразу нескольким элементам на странице, однако значения в атрибуте `id` должны быть уникальными для этой страницы. Таким образом, класс `content` могут иметь несколько элементов, но идентификатор `main-title` — только один.

Порядок отображения элементов HTML-документа на веб-странице полностью зависит от особенностей браузера. Если разные веб-браузеры будут отображать одни и те же элементы интерфейса по-разному, это приведет к несогласованности действий пользователей.

По этой причине HTML-теги должны быть согласованы и зафиксированы в едином стандарте. В настоящее время стандарт HTML контролируется Консорциумом Всемирной паутины (World Wide Web Consortium, W3C). Текущую спецификацию всех HTML-тегов можно найти по адресу **https://html.spec.whatwg.org/multipage/**.

Однако официальный стандарт W3C HTML — не лучший вариант для начинающих изучать HTML. Веб-скрапинг почти полностью заключается в чтении и интерпретации HTML-файлов, найденных в Интернете. Если вы никогда раньше не имели дела с HTML, я настоятельно рекомендую прочитать книгу *HTML & CSS: The Good Parts*[1], чтобы познакомиться с некоторыми наиболее распространенными тегами HTML.

[1] *Хеник Б.* HTML и CSS. Путь к совершенству. — СПб.: Питер, 2011.

CSS

Каскадные таблицы стилей (Cascading Style Sheets, CSS) определяют внешний вид HTML-элементов на веб-странице. Они задают, например, размещение, цвет, размер и другие свойства элементов, которые превращают скучную HTML-страницу со стилями, заданными браузером по умолчанию, в нечто более привлекательное для современного пользователя.

Для приведенного выше примера HTML:

```
<html>
  <head>
    <title>A Simple Webpage</title>
  </head>
  <body>
    <!-- Это текст комментария, он не выводится браузером -->
    <h1>Hello, World!</h1>
  </body>
</html>
```

вариант его оформления:

```
h1 {
  font-size: 20px;
  color: green;
}
```

Эта таблица стилей задает размер шрифта содержимого тега `h1` равным 20 пикселам и отображает его в зеленом цвете.

Часть `h1` в этой таблице стилей называется *селектором* или CSS-селектором. Он указывает, что стили, определяемые внутри фигурных скобок, будут применяться к содержимому любых тегов `h1`.

Селекторы CSS могут также применяться только к элементам с определенными атрибутами: `class` или `id`. Например, для разметки HTML:

```
<h1 id="main-title">Some Title</h1>
<div class="content">
  Lorem ipsum dolor sit amet, consectetur adipiscing elit
</div>
```

таблица стилей может быть следующей:

```
h1#main-title {
  font-size: 20px;
}

div.content {
  color: green;
}
```

Символ # уточняет, что определяемые стили должны применяться к элементу с атрибутом `id="main-title"`, а символ . указывает на значение атрибута `class`.

Если имя тега не имеет значения, то его можно полностью опустить. Например, следующая таблица стилей окрасит в зеленый цвет содержимое любого элемента с атрибутом `class="content"`:

```
.content {
  color: green;
}
```

Данные таблицы стилей могут содержаться либо в самом HTML, либо в отдельном CSS-файле с расширением `.css`. Таблица стилей в HTML-файле размещается внутри тегов `<style>` в заголовке HTML-документа:

```
<html>
  <head>
    <style>
      .content {
        color: green;
      }
    </style>
...
```

Чаще всего CSS импортируется в заголовок документа с помощью тега `link`:

```
<html>
  <head>
    <link rel="stylesheet" href="mystyle.css">
...
```

Занимаясь веб-скрапингом, вам нечасто придется писать таблицы стилей, чтобы сделать HTML красивым. Однако важно владеть синтаксисом и понимать, каким образом HTML-страница преобразуется с помощью CSS, чтобы видеть взаимосвязь между тем, что отображается в браузере, и тем, что написано в коде.

Например, вас может удивить элемент HTML, который не отображается на странице. Однако, прочитав код CSS, применяемый к элементу, вы можете увидеть определение стиля:

```
.mystery-element {
  display: none;
}
```

Это означает, что для атрибута элемента `display` установлено значение `none`, делая его невидимым.

Если вы никогда раньше не сталкивались с таблицами стилей, скорее всего, вам не потребуется глубоко их изучать, чтобы извлекать данные из Интернета, однако необходимо освоить их синтаксис и учитывать правила CSS, приведенные в этой книге.

JavaScript

Когда веб-сервер получает от клиента запрос на определенную веб-страницу, он выполняет некоторый код, чтобы создать ее и затем отправить пользователю. Этот *серверный код* может выполнять очень простые действия, такие как чтение статического HTML-файла с диска и его отправка, или очень сложные, написанные на Python (лучшем языке), Java, PHP или любом другом серверном языке программирования.

В итоге серверный код создает своеобразный поток данных, который отправляется браузеру и отображается на экране. Но что, если нужно выполнить некоторое действие, например изменить текст или перетащить элемент, без обращения к серверу и выполнения кода на его стороне? Для этого используется *клиентский код*.

Клиентский код — это любой код, который передается клиенту веб-сервером, но выполняется браузером. На заре Интернета (до середины 2000-х) клиентский код писался на нескольких языках. Возможно, вы помните Java-апплеты и Flash-приложения. Но со временем JavaScript стал единственным языком для реализации клиентского кода по одной простой причине: этот язык изначально поддерживался браузерами без дополнительного программного обеспечения, загружаемого и обновляемого отдельно (как, например, Adobe Flash Player).

JavaScript появился в середине 90-х как новая функция Netscape Navigator. Он был быстро адаптирован к Internet Explorer, что сделало его стандартом для обоих основных веб-браузеров того времени.

Несмотря на название, JavaScript не имеет почти ничего общего с Java — серверным языком программирования. Если не считать некоторого сходства в синтаксисе, это совершенно разные языки.

В 1996 году компании Netscape (где был создан JavaScript) и Sun Microsystems (создательница Java) заключили лицензионное соглашение, позволяющее Netscape использовать название JavaScript, что предполагало дальнейшее тесное сотрудничество в развитии двух языков (**https://www.infoworld.com/article/2653798/javascript-creator-ponders-past--future.html**). Однако это сотрудничество так и не состоялось, и с тех пор название языка JavaScript сбивает с толку.

Несмотря на то что изначально JavaScript был языком сценариев для уже несуществующего браузера, в настоящее время он превратился в один из самых популярных языков программирования в мире. Этой популярности поспособствовал тот факт, что на нем можно также писать серверный код, используя Node.js. Но основная причина его всеобщего признания, безусловно, заключается в том, что это единственный язык программирования, используемый на стороне клиента.

JavaScript встраивается в HTML-страницы с помощью тега `<script>`. Код на JavaScript можно вставить в форме содержимого:

```
<script>
  alert('Hello, world!');
</script>
```

Или же на него можно сослаться в отдельном файле, используя атрибут `src`:

```
<script src="someprogram.js"></script>
```

В отличие от HTML и CSS, вам, скорее всего, не нужно будет читать или писать код на JavaScript, занимаясь веб-скрапингом, но будет полезно хотя бы знать, как он выглядит. Иногда код может содержать полезные данные. Например:

```
<script>
  const data = '{"some": 1, "data": 2, "here": 3}';
</script>
```

Здесь объявляется JavaScript-переменная `data` с ключевым словом `const` (что означает «константа») и ей присваивается строка в формате JSON, содержащая данные, которые теоретически можно парсировать напрямую с помощью веб-скрапера.

JSON (JavaScript Object Notation — форма записи объектов JavaScript) — это текстовый формат, описывающий данные, который легко парсируется веб-скраперами и широко применяется в сети. Он будет рассмотрен подробнее в главе 15.

Иногда вы можете увидеть, что код на JavaScript запрашивает данные из стороннего источника:

```
<script>
  fetch('http://example.com/data.json')
    .then((response) => {
      console.log(response.json());
    });
</script>
```

Здесь код на JavaScript посылает запрос по адресу `http://example.com/data.json` и выводит полученный ответ в консоль (подробнее о консоли рассказывается в следующем разделе).

Изначально JavaScript создавался для реализации динамической интерактивности и анимации в статических веб-страницах. Однако сейчас динамическое поведение создается не только с помощью JavaScript. HTML и CSS тоже обладают некоторыми возможностями, позволяющими изменять содержимое страницы.

Например, поддержка анимации в CSS позволяет перемещать элементы, менять их цвет, размер и выполнять другие преобразования, когда пользователь выбирает элемент или наводит на него указатель мыши.

Понимание того, как связаны (часто в буквальном смысле) движущиеся части веб-сайта, может помочь вам избежать путаницы при поиске данных.

Просмотр сайтов с помощью инструментов разработчика

Подобно лупе ювелира или стетоскопу кардиолога, инструменты разработчика в браузере помогают исследовать содержимое веб-страниц. Чтобы извлечь данные с веб-сайта, вы должны знать, где и как они хранятся. Инструменты разработчика помогут вам ответить на эти вопросы.

В рамках данной книги будут рассмотрены инструменты разработчика, доступные в Google Chrome. Однако инструменты разработчика, имеющиеся в Firefox, Microsoft Edge и других браузерах, обладают похожими возможностями, и их можно использовать с не меньшим успехом.

Чтобы получить доступ к инструментам разработчика, воспользуйтесь следующими инструкциями.

Chrome

... ▸ **More Tools** ▸ **Developer Tools** (Дополнительные инструменты ▸ Инструменты разработчика).

Safari

Safari ▸ **Preferences** ▸ **Advanced** ▸ **Show Develop menu in menu bar** (Настройки ▸ Дополнительно ▸ Показывать меню «Разработка» в строке меню).

После этого останется только использовать меню разработчика: **Develop** ▸ **Show Web Inspector** (Разработка ▸ Показать веб-инспектор).

Microsoft Edge

... ▸ **Tools** ▸ **Developer Tools** (Дополнительные инструменты ▸ Инструменты разработчика).

Firefox

Tools ▸ **Browser Tools** ▸ **Web Developer Tools** (Инструменты ▸ Другие инструменты ▸ Инструменты веб-разработчика).

Во всех браузерах комбинация горячих клавиш для открытия инструментов разработчика одинаковая, но зависит от операционной системы.

Mac

Option+Command+I

Windows

CTRL+Shift+I

Занимаясь веб-скрапингом, чаще всего вы будете пользоваться вкладками **Network** (Сеть) (рис. 1.1) и **Elements** (Элементы).



Рис. 1.1. Инструменты разработчика Chrome, показывающие процесс загрузки страницы из «Википедии»

Вкладка **Network** (Сеть) отображает все запросы, выполняемые во время загрузки страницы. Если вы прежде никогда сюда не заглядывали, то вас ждет сюрприз! В процессе загрузки сложные страницы часто отправляют десятки и даже сотни запросов к ресурсам. В некоторых случаях страницы могут даже продолжать генерировать постоянный поток запросов на протяжении всего жизненного цикла в браузере. Например, они могут отправлять данные для трассировки действий или запрашивать обновления.



Ничего не отображается на вкладке Network (Сеть)?

Обратите внимание, что инструменты разработчика должны быть открыты к тому моменту, *когда* страница начнет выполнять запросы, иначе эти запросы не будут зафиксированы. Если вы сначала открыли страницу, а потом решили заглянуть в инструменты разработчика, то вам придется ее обновить, чтобы посмотреть, какие запросы она выполняет.

Открыв вкладку **Network** (Сеть) и щелкнув кнопкой мыши на том или ином сетевом запросе, вы увидите всю информацию, связанную с этим запросом. Компоновка инструмента исследования сетевых запросов немного отличается в разных браузерах, но в целом он позволяет увидеть:

- URL, на который был отправлен запрос;
- используемый метод HTTP;
- статус ответа;
- все заголовки и файлы cookie, связанные с запросом;
- полезную нагрузку;
- ответ.

Эта информация пригодится при разработке веб-скраперов, которые копируют запросы, чтобы получить те же данные, что и страница.

Вкладка **Elements** (Элементы) (рис. 1.2 и 1.3) помогает исследовать структуру и содержимое HTML-файлов. Этот инструмент чрезвычайно удобен для изучения конкретных фрагментов данных на странице, поиска HTML-тегов, включающих эти данные, и разработки парсеров для их извлечения.

Если вы откроете вкладку **Elements** (Элементы) и наведете указатель мыши на HTML-элемент, браузер автоматически выделит цветом соответствующий компонент на веб-странице. Этот инструмент помогает исследовать страницы и изучать их устройство (рис. 1.3).

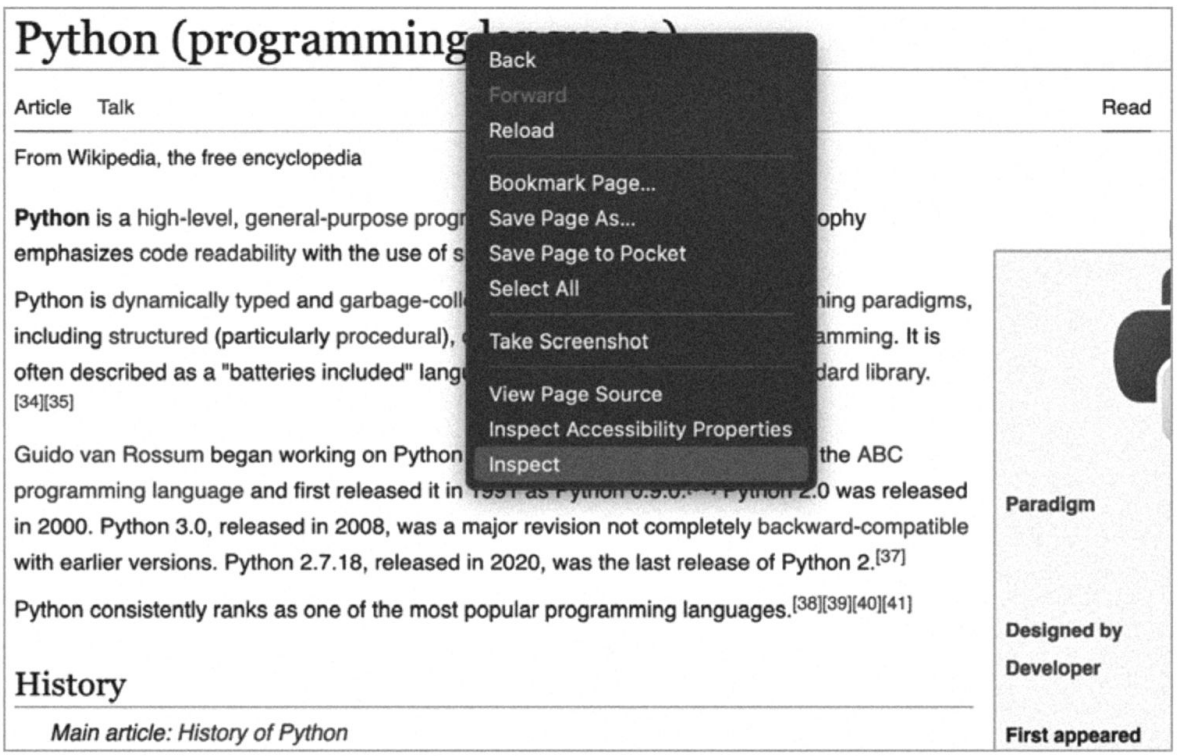


Рис. 1.2. Щелкните правой кнопкой мыши на любом фрагменте текста или данных и выберите Inspect (Просмотреть код), чтобы на вкладке Elements (Элементы) увидеть элементы, окружающие эти данные



Рис. 1.3. Если навести указатель мыши на HTML-элемент, то браузер выделит цветом соответствующий фрагмент на странице

Вам не нужно быть экспертом в интернет-технологиях, сетях или программировании, чтобы выполнить веб-скрапинг. Однако очень важно иметь хотя бы базовое понимание, как эти элементы сочетаются друг с другом и как инструменты разработчика в браузере отображают их.

ГЛАВА 2

Законность и этичность веб-скрапинга

В 2010 году инженер-программист Пит Уорден разработал веб-краулер для сбора данных с Facebook. Он собрал данные почти о 200 миллионах пользователей соцсети: имена, места жительства, сведения о друзьях и интересах. Конечно же, в Facebook это заметили и стали слать Уордену письма с просьбой прекратить свои действия, и он прислушался. Когда программиста спросили, почему он подчинился их требованиям, Пит ответил: «Большие данные — это дешево, а вот адвокаты — нет».

В этой главе вы познакомитесь с законами США (и некоторыми международными), имеющими отношение к веб-скрапингу, и узнаете, как оценить степень законности и этичности веб-скрапинга в той или иной ситуации.

Прежде чем перейти к следующему разделу, констатируем очевидный факт: я — инженер-программист, а не юрист. Не интерпретируйте все, что прочтете в этой книге, как профессиональную юридическую консультацию и тщательно обдумывайте свои действия. Я действительно считаю, что могу компетентно обсуждать правовые и этические аспекты веб-скрапинга. Однако, прежде чем начинать какие-либо юридически неоднозначные проекты, проконсультируйтесь 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 настоящую юридическую консультацию.

Торговые марки, авторские права, патенты... спасите-помогите!

Пришло время пройти краткий курс по основам интеллектуальной собственности! Существуют три основных ее типа: торговые марки (обозначаемые символами ТМ или ®), авторские права (часто встречающийся знак копирайта ©) и патенты (иногда сопровождаются указанием, что изобретение защищено патентом или номером патента, но обычно не обозначаются 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 технология), а не данные, из которых состоит программное обеспечение. Если вы не создаете что-то по чертежам, собранным с помощью веб-скрапинга, и не пользуетесь методом сбора данных, который запатентовал кто-то другой, то вряд ли случайно нарушите какой-либо патент, просто выполнив веб-скрапинг сайта.

Торговые марки также вряд ли могут вызвать проблемы, но все же их следует учитывать. Ведомство по патентам и товарным знакам США утверждает следующее.

> *Торговая марка* — это слово, фраза, символ и/или дизайн, которые идентифицируют и отличают товары одной компании от товаров другой. *Знак обслуживания* — это слово, фраза, символ и/или дизайн, которые идентифицируют и отличают 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.

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 будет явно незаконным использованием торговой марки. Точно так же я могу изобразить на упаковке своего нового безалкогольного напитка Розовую Пантеру, однако не могу использовать этот же цвет, если захочу производить утеплитель для стен.

Это подводит нас к теме добросовестного использования, которая часто обсуждается в контексте Закона об авторском праве, но также применима и к торговым маркам. Использовать или демонстрировать торговую марку как ссылку на бренд, который она представляет, — допустимо, но применять торговую марку способом, вводящим потребителя в заблуждение, — нет. Следует отметить, что понятие «добросовестное использование» неприменимо к патентам. Например, изобретение, запатентованное в одной отрасли, не может использоваться в другой без согласования с обладателем патента.

Закон об авторском праве

Торговые марки и патенты имеют общее свойство: 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 законом? Это не предупреждение о том, что 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, названная в честь города Берна в Швейцарии, где она 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).



Регистрация авторских прав

Несмотря на то что защита авторского права применяется автоматически и не требует какой-либо регистрации, все же есть возможность официально зарегистрировать авторское право в государственных органах США. Это часто делается в отношении ценных творческих произведений, таких как значимые кинофильмы, чтобы впоследствии облегчить любые судебные разбирательства и документально зафиксировать принадлежность произведения. Но пусть наличие этой возможности вас не смущает — все творческие работы, кроме случаев, когда они являются общественным достоянием, защищены авторским правом!

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 позволят сделать мой проект веб-скрапинга блога оправданным в зависимости от того, как он функционирует.

Защита авторских прав распространяется только на творческие произведения, а не на статистику или факты. К счастью, бóльшая часть того, за чем охотятся веб-скраперы, — *именно* статистика и факты.

Веб-скрапер, собирающий стихи со всего Интернета и отображающий всю эту поэзию на личном сайте, возможно, 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 информация о среднем количестве слов в стихах, опубликованных на сайте в разные месяцы, — это уже 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 об авторском праве в цифровую эпоху (Digital Millennium Copyright Act, DMCA). В нем изложены некоторые правила автоматической обработки материалов, защищенных авторским правом. DMCA — это обширный документ, который содержит множество специальных правил, регулирующих все: от электронных книг до телефонов. Тем не менее в нем есть два важных момента, которые могут иметь особое отношение к веб-скрапингу.

- Вы в «зоне безопасности», если извлекаете материал из источника, который, по вашему мнению, содержит только не защищенные авторским правом данные, однако пользователь предоставил защищенный материал, и вы удалили его сразу после получения уведомления.
- Собирая контент, вы не можете обойти меры безопасности (такие как защита паролем).

Кроме того, DMCA также признает, что использование материалов, являющееся правомерным в соответствии со сводом законов США, допустимо и уведомления о судебном разбирательстве не могут выдаваться в соответствии с защитой в «зоне

безопасности», если применение материала, защищенного авторским правом, классифицируется как допустимое.

Короче говоря, вы никогда не должны публиковать защищенные авторским правом материалы напрямую, без разрешения автора или правообладателя. Если вы храните защищенные авторским правом данные, к которым у вас есть свободный доступ, в своей личной базе данных с целью анализа — все в порядке. Однако вы поступите нехорошо, опубликовав эту базу на своем сайте для просмотра или скачивания. Если вы анализируете эти данные и публикуете статистику по количеству слов, список авторов по степени продуктивности 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.

Авторское право и искусственный интеллект

Генеративный искусственный интеллект (ИИ), или программы искусственного интеллекта, создающие новые «творческие» произведения на основе существующих творческих работ, представляют собой уникальную проблему для Закона об авторском праве.

Если результат работы программы генеративного ИИ похож на уже существующую работу, то может возникнуть проблема с авторскими правами. Известны многие прецеденты, которые помогают понять, что означает слово «похож». По данным Исследовательской службы Конгресса США (Congressional Research Service)[1]:

> Критерий существенного сходства сложно определить, и он различается в разных судах США, например, «существенное сходство концепции и восприятия», или «схожий внешний вид», или «обычный разумный человек не сможет заметить различия между двумя произведениями».

Проблема современных генеративных алгоритмов заключается в том, что невозможно автоматически определить, создал ваш ИИ захватывающую и новую работу или нечто производное от нее. ИИ может не иметь возможности обозначить свой результат как «существенно похожий» на конкретные входные данные или

[1] Полный анализ см. в разделе Generative Artificial Intelligence and Copyright Law (https://crsreports.congress.gov/product/pdf/LSB/LSB10922), Legal Sidebar, Congressional Research Service. 29 сентября 2023 г.

даже определить, какие из входных данных он использовал для создания своего творения! Первым признаком, когда что-то пошло не так, может стать письмо о прекращении противоправных действий или вызов в суд.

Помимо вопросов нарушения авторских прав, результатами работы генеративного ИИ в ходе судебных разбирательств также проверяют, не нарушил ли права правообладателя сам процесс обучения.

Для обучения этих систем почти всегда необходимо загружать, сохранять и воспроизводить защищенную авторским правом работу. Загрузка изображения или текста, защищенного авторским правом, может показаться простой задачей, но это мало чем отличается от загрузки фильма, защищенного авторским правом, ведь вы бы не стали скачивать такой фильм, не так ли?

Некоторые утверждают, что такой подход является добросовестным использованием, они не публикуют и не используют контент так, чтобы это могло повлиять на его продажи.

На момент написания этой книги компания OpenAI доказывала перед Ведомством по патентам и товарным знакам США (United States Patent and Trademark Office), что использование ею больших объемов материалов, защищенных авторским правом, является добросовестным использованием[1]. Этот аргумент в первую очередь относится к генеративным алгоритмам ИИ, но я подозреваю, что решение, которое будет вынесено по результатам разбирательства, будет применимо и к веб-скраперам, созданным для самых разных целей.

Посягательство на движимое имущество

Посягательство на движимое имущество принципиально отличается от того, что мы называем «нарушением закона о владении собственностью», поскольку 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 возникает соблазн не воспринимать веб-серверы как реальные, материальные ресурсы. Однако серверы не только состоят из дорогих компонентов. Их еще нужно хранить, контролировать, охлаждать и снабжать огромным количеством электроэнергии. По некоторым оценкам, 10 % мирового

[1] См.: Comment Regarding Request for Comments on Intellectual Property Protection for Artificial Intelligence Innovation (https://www.uspto.gov/sites/default/files/documents/OpenAI_RFC-84-FR-58141.pdf), docket No. PTO-C-2019-0038, U.S. Patents and Trademark Office.

потребления электроэнергии приходится на компьютеры[1]. (Если вы подсчитаете, сколько потребляет вся ваша электроника, и все еще будете в чем-то сомневаться, 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 потребляли 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.

Существуют 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 полученные вами 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, это может быть расценено как вред, который вы причинили.

Преднамеренность

Если вы написали код, то наверняка знали, что он делает! Слова об отсутствии намерений будут звучать неправдоподобно.

Чтобы на вас подали заявление о посягательстве на движимое имущество, ваши действия должны соответствовать всем трем критериям. Однако если вы нарушите соглашение об условиях эксплуатации, но не причините реального вреда, то не думайте, что застрахованы от судебных исков. Вы вполне можете нарушить Закон об авторском праве, DMCA, Закон о компьютерном мошенничестве и злоупотреблении (подробнее о нем см. далее в этой главе) или еще какой-либо из множества других нормативных документов, применимых к веб-скраперам.

---

[1] *Walsh B.* The Surprisingly Large Energy Footprint of the Digital Economy [UPDATE] // TIME.com. — August 14, 2013. — URL: http://ti.me/2IFOF3F.

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 — 32 Гбайт. И наоборот, средний экземпляр AWS имеет 4 Гбайт оперативной памяти и около 3 ГГц 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 неотложной медицинской помощи, когда пациента может спасти только срочный сбор всех данных с сайта Васи Пупкина, то нет причин атаковать сайт своими запросами.

Управляемый бот никогда не завершает работу. Иногда лучше включить краулер на ночь, чем днем или вечером, и на то есть следующие причины.

- Имея в распоряжении около восьми часов, даже при черепашьей скорости в две секунды на страницу можно просканировать 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 ваше сканирование не совпадет с временем пикового трафика.
- Вы экономите время — спите, вместо того чтобы постоянно проверять журналы на наличие новой информации. Подумайте о том, какой сюрприз вас будет ждать утром в виде порции свежих данных!

Рассмотрим следующие сценарии:

1) у вас есть веб-краулер, который просматривает сайт Васи Пупкина, собирая с него некоторые или все данные;
2) у вас есть веб-краулер, который просматривает несколько сотен небольших сайтов, собирая с них некоторые или все данные;
3) у вас есть веб-краулер, который просматривает очень большой сайт, такой как «Википедия».

В первом случае лучше оставить бот медленно работать на ночь.

Во втором — лучше обращаться к каждому сайту по очереди, вместо того чтобы работать, медленно перебирая 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 (когда каждый из них будет сканировать один сайт и приостанавливать свое выполнение) или списки Python для отслеживания сайтов.

В третьем варианте нагрузка, которую ваше интернет-соединение и домашний компьютер могут оказать на такой сайт, как «Википедия», вряд ли будет замечена или вызовет беспокойство. Но вот если вы используете распределенную сеть устройств — тогда, очевидно, другое дело. Соблюдайте осторожность и по возможности обратитесь к представителю компании.

Закон 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 не просто как неудобство (или даже забавное хобби), а как серьезное уголовное преступление, способное причинить материальный ущерб. В 1983 году фильм «Военные игры» с Мэттью Бродериком в главной роли также привлек внимание общественности и президента Рональда Рейгана к этой проблеме. В результате в 1986 году был принят Закон 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 для веб-скраперов он тоже имеет важное значение. 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, в целях извлечения информации;
- намеренный несанкционированный доступ к компьютеру и сбор финансовой информации;
- намеренный несанкционированный доступ к компьютеру, принадлежащему Правительству США, повлиявший на работу этого компьютера;
- намеренный доступ к любому защищенному компьютеру в мошеннических целях;

- намеренный доступ к компьютеру без разрешения и причинение ему ущерба;
- разглашение или передача паролей либо информации об авторизации на компьютерах, используемых Правительством США, или компьютерах, которые оказывают влияние на внутреннюю или внешнюю торговлю;
- попытки вымогательства денег или каких-то ценностей путем нанесения ущерба или угрозы причинения вреда любому защищенному компьютеру.

Иными словами, держитесь подальше от защищенных компьютеров, не заходите на компьютеры (в том числе на веб-серверы), к которым у вас нет доступа, и особенно остерегайтесь компьютеров, принадлежащих правительству или финансовым организациям.

Файл robots.txt и условия его использования

Условия использования сайта и файла `robots.txt` представляют интерес с юридической точки зрения. 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, а 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 происходит и, как правило, юридическое уведомление о том, что услуги, предоставляемые сайтом, не подразумевают неких явных или скрытых гарантий.

Если вас интересует оптимизация сайта для поисковых систем (Search Engine Optimization, SEO) или технологии поиска, то вы, скорее всего, слышали о файле `robots.txt`. Его можно обнаружить в корневой веб-папке `http://website.com/robots.txt` на любом крупном сайте, поискав там файл `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 собираемой ими информации.

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 парсирован 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` не существует. 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 следовать требованиям файла `robots.txt`. Файл — лишь знак, говорящий: «Пожалуйста, не заходите в эти части сайта». Есть множество библиотек веб-скрапинга, соблюдающих условия `robots.txt` (хотя обычно это 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 парсинг содержимого страницы и применить к этому контенту логику вашего кода), которые не позволяют просто продвигаться вперед и выполнять веб-скрапинг всех необходимых страниц.

Синтаксис Robot Exclusion Protocol 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 последующие 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 ботов к любой точке сайта, за исключением бота Google, которому разрешается доступ куда угодно, кроме каталога /private.

В Twitter (ныне переименованный в «X») 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 в интересах разработчика. Обнаружение файла robots.txt, который запрещает сканирование определенного раздела сайта, по сути, можно расценить как сообщение веб-мастера о том, что он разрешает доступ краулеров ко всем остальным разделам сайта (в конце концов, если бы это было не так, то вам в первую очередь ограничили бы доступ при написании robots.txt).

Например, раздел файла robots.txt из «Википедии», адресованный обычным веб-скраперам (в отличие от поисковых систем), на удивление дружелюбен. Он даже содержит удобочитаемый текст для приветствия ботов (это для нас!) и блокирует доступ только к нескольким страницам, таким как страницы аутентификации, поиска и «случайной статьи»:

```
#
# Мы приветствуем использование дружелюбных и неторопливых ботов
# для просмотра статических страниц, но не динамически генерируемых.
#
```

```
# Бот "Slurp" может сканировать с минимальной задержкой между попытками;
# если ваш бот поддерживает такие настройки, как 'Crawl-delay', то сообщите нам.
#
# Для MobileView API существует специальное исключение, позволяющее
# мобильным приложениям загружать содержимое разделов.
# Это содержимое не кэшируется протоколом HTTP, но активно кэшируется
# парсером, и потому к его загрузке не предъявляется специальных требований.
#
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 рекомендую это делать, особенно если ваши краулеры будут просматривать 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, что конкретно было противозаконным 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 тип сайта для проведения метааукционов. Вместо того чтобы заставлять пользователя переходить с одного аукционного сайта на другой, сравнивая цены, он объединял данные всех аукционов для определенного продукта (например, только что выпущенной куклы Ферби или нового экземпляра *Spice World*) и давал ссылку на тот, который предлагал самую низкую цену.

Компания Bidder's Edge добилась этого с помощью целой армии веб-скраперов, которые постоянно отправляли 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 обращалась к его серверам примерно 100 000 раз в день. Даже по сегодняшним меркам это большой трафик. По данным eBay, в то время он составлял 1,53 % общего интернет-трафика, что, безусловно, не доставляло радости владельцам аукциона.

Компания eBay направила Bidder's Edge письмо с просьбой прекратить свои действия и предложением лицензировать данные. Однако переговоры оказались безрезультатными, и Bidder's Edge продолжала сканировать сайт 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 используют 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 блокировки старых, а eBay приходилось поддерживать большие списки блокировки для брандмауэров (и нести дополнительные большие расходы на сравнение IP-адресов при проверке каждого пакета).

Наконец, в декабре 1999 года eBay подала 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 Bidder's Edge их эксплуатировала, нельзя было назвать нормальным, посягательство на движимое имущество казалось идеальным поводом для обращения в суд. На самом деле в наше время случаи посягательства

на движимое имущество постоянно фигурируют в судебных исках по веб-скрапингу, поэтому их принято относить к законам, касающимся IT.

Суд постановил: чтобы выиграть дело о посягательстве на движимое имущество, eBay должна была 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 копии писем от eBay с просьбой прекратить свои действия и отчеты IT-отдела компании об использовании серверов и фактических расходах на серверы, ей было относительно нетрудно выиграть дело. 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 было урегулировано во внесудебном порядке на сумму, размер которой не раскрывается.

Значит ли это, что любое несанкционированное использование чужого сервера автоматически является посягательством на движимое имущество? Необязательно. История с Bidder's Edge была чрезвычайным случаем; компания задействовала так много ресурсов eBay, что той 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 своим коллегам нежелательные для компании электронные письма через все ее серверы. Суд постановил следующее.

> Заявление Intel отклонено не потому, что электронная корреспонденция, передаваемая через Интернет, обладает уникальной неприкосновенностью, а потому, что посягательство на движимое имущество, в отличие от только что упомянутых оснований для предъявления претензий, не может быть доказано в Калифорнии без свидетельств причинения вреда личной собственности истца или его законным интересам.

По сути, Intel не удалось доказать, что передача 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!), причинила компании какой-либо финансовый вред. Затраты не лишили Intel никакой собственности и не помешали использованию собственности компании.

Соединенные Штаты Америки против Ауэрнхаймера и Закон о компьютерном мошенничестве и злоупотреблении

Если информация в Интернете легкодоступна для человека, использующего браузер, то маловероятно, что за автоматическим доступом к той же самой информации сразу последуют неприятности с законом. Однако стоит любопытному человеку найти в системе безопасности небольшую лазейку, 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 и Дэниел Спитлер обратили внимание на интересную функцию в iPad: при посещении сайта AT&T с этого устройства пользователя перенаправляли на URL, содержащий уникальный идентификационный номер его 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. Предоставив 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: обнародованы 114 000 владельцев 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 США в качестве компенсации.

Дело Ауэрнхаймера привлекло внимание адвоката по гражданским правам Орина Керра, 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 удовлетворил апелляцию, заявив следующее.

> «Осуждение Ауэрнхаймера по пункту 1 должно быть отменено, поскольку в соответствии с Законом 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. Тот факт, что AT&T субъективно желала, чтобы посторонние не обнаружили эти данные, или что Ауэрнхаймер преувеличенно 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».

Хотя приговор Ауэрнхаймеру был отменен только в апелляционном порядке, суд третьего округа, похоже, согласился с этим аргументом, о чем говорится в примечании к вынесенному решению.

> «Нам не пришлось решать, является ли поведение Ауэрнхаймера нарушением закона, но в ходе судебного разбирательства не было представлено никаких доказательств, что программа сбора учетных записей взломала какой-либо пароль или преодолела иной программный барьер. Программа просто обращалась к общедоступной части формы входа в систему и извлекала информацию, которую AT&T непреднамеренно опубликовала».

Хотя в итоге суд и вынес решение, что Ауэрнхаймер не нарушил Закон о компьютерном мошенничестве и злоупотреблении, однако ФБР провело обыск в его доме, а сам он потратил тысячи долларов на оплату юридических услуг, провел три года в залах заседаний и тюрьмах.

Какие уроки могут извлечь из этого разработчики веб-скраперов, чтобы избежать подобных ситуаций и не оказаться в дураках?

Сбор любой конфиденциальной информации, будь то персональные данные (в рассмотренном случае — адреса электронной почты), коммерческие или правительственные секреты, — вероятно, не то, с чем стоит связываться, не имея контактов адвоката. Даже если эти материалы общедоступны, подумайте вот о чем: «Могли бы обычные пользователи легко получить доступ к этим данным при желании их увидеть?» или «Компания действительно хочет, чтобы пользователи видели эту информацию?»

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 специалисту, с которым я могла бы обсудить эту проблему?»

В вас не только немедленно признают (доброго) хакерского гения, вы также можете получить бесплатные подписки, денежные вознаграждения и многие другие бонусы!

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 сообщить о них владельцам данного сайта, а не СМИ. У вас может возникнуть желание написать пост 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 (или по возможности вообще уйти) с сайта!

Филд против Google: авторские права и robots.txt

Однажды адвокат Блейк Филд 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.



Веб-кэш Google

Просматривая 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 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, создав файл `robots.txt` с простыми указаниями о том, какие страницы веб-скраперам следует обрабатывать, а какие нет.

Таким образом, суд постановил, что положение DMCA о зоне безопасности позволило Google легально кэшировать и отображать сайты, такие как сайт Филда: «[а] поставщик услуг не предоставляет денежную компенсацию... и не несет ответственности за нарушение авторских прав по причине промежуточного или временного хранения материалов в системе или сети, контролируемой или управляемой поставщиком услуг либо для него».

ГЛАВА 3

Применение веб-скрапинга

Веб-скраперы могут помочь практически любому бизнесу, но зачастую главная задача состоит в том, чтобы выяснить, *каким образом* это сделать. Как и в случае с искусственным интеллектом или программированием в целом, вы не можете просто взмахнуть волшебной палочкой и ожидать, что ваша прибыль увеличится.

Эффективное применение веб-скрапинга требует реальной стратегии и тщательного планирования. Вы должны определить конкретные задачи, выяснить, какие данные нужны для их решения, а затем наметить алгоритмы, которые позволят вашим веб-скраперам создавать эти данные.

Классификация проектов

Планируя проект по веб-скрапингу, подумайте о том, к какой категории или категориям его отнести.

Ваш скрапер будет узкоспециализированным или универсальным? В первом случае можно написать шаблоны, но для реализации второго потребуются другие методы.

- Если скрапинг будет применяться для сбора данных с одного веб-сайта или даже фиксированного перечня его страниц — это действительно узкоспециализированный проект веб-скрапинга.
- Если скрапинг будет применяться для сбора данных с фиксированного количества известных веб-сайтов — это тоже весьма специализированный скрапер, но при этом вам может понадобиться написать некоторый объем специального кода для каждого веб-сайта и потратить больше времени на разработку архитектуры скрапера.
- Если скрапинг будет применяться для сбора данных с большого количества неизвестных веб-сайтов, будут динамически определяться новые цели и ваш краулер начнет автоматически определять структуру веб-сайтов, возможно, вы пишете универсальный, или неспециализированный, скрапер.

Вы собираетесь запустить скрапер только один раз или он должен работать постоянно, загружая данные или отыскивая новые страницы?

- Одноразовые скраперы обычно создаются быстро и не требуют больших затрат. Код необязательно должен быть красивым! Конечным результатом такого проекта являются сами данные: вы можете передать файл Excel или CSV заказчику, и он будет доволен, а код после этого отправится в корзину.
- Любой проект, включающий мониторинг, повторное сканирование новых данных или обновление старых, потребует более надежного кода, который должен поддерживаться системой. Скраперу также может потребоваться собственная инфраструктура мониторинга, обнаруживающая ошибки, отказы или потребление чрезмерно большого количества ресурсов.

Являются ли собранные данные вашим конечным продуктом или предполагается, что они будут подвергнуты дальнейшему анализу и переработке?

- Когда скрапер просто собирает информацию, он может помещать ее в базу данных в том виде, в каком ее получил, или попутно выполнять некоторую очистку, например удалять символы доллара из цен на товары.
- Если полученная информация будет подвергаться углубленному анализу, то вы можете даже не знать, какие данные будут важны. В таких случаях желательно хорошо продумать архитектуру скрапера.

Подумайте, к каким категориям может относиться каждый из этих проектов и как, возможно, потребуется изменить масштаб проекта, чтобы он соответствовал интересам вашего бизнеса.

Электронная коммерция

Мною было написано множество веб-скраперов, собирающих самые разные данные из Интернета, но чаще всего ко мне обращались с просьбами реализовать сбор данных о товарах и ценах с сайтов электронной коммерции.

Как правило, такие запросы поступают от людей, владеющих конкурирующими сайтами электронной коммерции или проводящих исследования и планирующих выпустить новый продукт или открыть новый рынок. Первый показатель, который приходит на ум занимающимся электронной коммерцией, — это цены. Заказчик хочет знать, как цена на его товар соотносится с ценами конкурентов. Однако существует огромное количество других данных, которые он, возможно, захочет собрать.

Товары могут отличаться размерами, цветами и стилями. Эти различия могут быть связаны с разными стоимостью и доступностью. Нередко может оказаться полезным отслеживать все варианты каждого продукта, а также список основных продуктов. Обратите внимание, что для каждого наименования часто можно найти идентификационный код — артикул, уникальный для одного варианта продукта и веб-сайта электронной коммерции (в торговых сетях Target и Walmart каждая разновидность товара может иметь свой артикул, но сам артикул в пределах сайта останется прежним, если вернуться и проверить его позже). Даже если артикул не отображается на веб-сайте непосредственно, его нередко можно найти где-нибудь в HTML-коде страницы или JavaScript API, который заполняет данные о продукте на веб-сайте.

При скрапинге сайтов электронной коммерции также может быть важной информация о доступном количестве единиц товара. Как и артикул, это количество может не отображаться на сайте непосредственно, но и эту информацию обычно удается найти в HTML или API веб-страницы. Обязательно отслеживайте ситуации, когда товара нет в наличии! Это может пригодиться для оценки рыночного спроса и, возможно, даже для влияния на цены ваших собственных товаров, если они есть у вас на складе.

Когда товар выставлен на продажу, на веб-сайте, как правило, можно найти продажную и первоначальную цену. Обязательно зафиксируйте обе цены. Отслеживая продажи за период времени, вы сможете проанализировать стратегии продвижения и скидки на товары вашего конкурента.

Обзоры и рейтинги товаров — еще одна полезная информация, которую стоит собрать. Конечно, вы не сможете напрямую отобразить на своем сайте текст обзоров товаров с сайтов конкурентов. Однако анализ этих обзоров может помочь определить, какие товары особенно популярны.

Маркетинг

Управление брендом и маркетинг в Интернете часто связаны с обработкой больших объемов данных. Вместо того чтобы листать социальные сети или тратить часы на поиск упоминаний названия компании, вы можете позволить выполнять эту работу веб-скраперам!

Следует отметить, что скраперы могут использоваться злоумышленниками для так называемого копирования веб-сайта с целью продажи контрафактных товаров или обмана потенциальных клиентов. К счастью, веб-скраперы также могут помочь в борьбе с этим, сканируя результаты поисковых систем на предмет мошеннического или неправомерного использования товарных знаков

компании и другой интеллектуальной собственности. Некоторые компании, например MarqVision, продают услуги по веб-скрапингу, что позволяет передавать на аутсорсинг процесс скрапинга сайтов, выявления признаков мошенничества и отправки уведомлений об удалении.

В то же время не всякое использование торговых марок является нарушением авторских прав. Если ваша компания упоминается, например, в комментариях или обзорах, то вам наверняка захочется узнать об этом! Веб-скраперы могут собирать и отслеживать общественное мнение о компании и ее бренде.

Однако, изучая информацию о своем бренде в Интернете, не забывайте о конкурентах! Возможно, вы захотите собрать информацию о людях, которые делали обзоры продуктов конкурентов или рассказывали о них, с целью предложить им скидки по ознакомительным рекламным акциям.

Когда речь идет о маркетинге и Интернете, первое, что приходит на ум, — это социальные сети. Основное преимущество скрапинга социальных сетей заключается в том, что, как правило, существуют лишь несколько крупных сайтов, которые позволяют создавать специализированные скраперы. Эти сайты содержат миллионы хорошо отформатированных сообщений со схожими данными и атрибутами (например, количеством лайков, репостов и комментариев), которые можно легко сравнить между собой.

Недостаток социальных сетей — возможные препятствия на пути к получению данных. Некоторые сайты, например Twitter, предоставляют как платные программные интерфейсы, так и бесплатные. Другие сайты защищают свои данные, прибегая к помощи не только технологий, но и юристов. Я советую проконсультироваться с юридическим представителем вашей компании, прежде чем пытаться извлекать данные, особенно с таких сайтов, как Facebook и LinkedIn.

Отслеживание показателей публикаций (количества лайков, репостов и комментариев) на темы, имеющие отношение к вашему бренду, может помочь выявить актуальные вопросы, требующие решения. Анализ популярности таких атрибутов, как объем публикации, вставка изображений или видео, языковые предпочтения, тоже может помочь выяснить, что больше всего нравится вашей целевой аудитории.

Если оплата продвижения вашего продукта кем-то с сотнями миллионов подписчиков вам не по карману, можно обратиться к «микро-» и «нановлиятельным лицам» — пользователям с небольшим присутствием в социальных сетях, которые могут даже не считать себя влиятельными! В таких случаях имеет смысл создать веб-скрапер, выполняющий поиск целевых учетных записей, владельцы которых часто публикуют сообщения на темы, актуальные для вашего бренда.

Академические исследования

Большинство примеров в этой главе, по сути, представляют собой разные методы активации механизма капитализма, но вообще веб-скраперы можно применять и для поиска знаний. Довольно часто скраперы используются в медицинских, социологических и психологических исследованиях, а также во многих других областях.

Например, Ратгерский университет предлагает курс под названием «Вычислительная социология», слушатели которого изучают приемы сбора данных в Интернете для исследовательских проектов. Кроме того, эти материалы даже включены в некоторые академические программы, например курс «Сбор и анализ больших данных» в Университете Осло!

В 2017 году в рамках проекта, поддержанного организацией «Национальные институты здравоохранения», были собраны данные о заключенных в тюрьмах США, чтобы оценить число инфицированных ВИЧ среди них[1]. Запуск этого проекта вызвал необходимость проведения тщательного этического анализа, в ходе которого были сопоставлены преимущества этого исследования и возможные риски для неприкосновенности частной жизни заключенных. В итоге исследование продолжилось, но я советую вам помнить об этической стороне вашего проекта и учитывать это, прежде чем применять веб-скрапинг для анализа данных, особенно в медицинской сфере.

В рамках другого проекта в области здравоохранения были собраны и проанализированы сотни комментариев из новостных статей в *The Guardian* об ожирении[2]. Несмотря на небольшой масштаб этого исследования по сравнению с другими, следует учитывать, что веб-скраперы могут использоваться в проектах, требующих небольших данных и качественного анализа.

Вот еще один пример специализированного исследовательского проекта, в котором использовался скрапинг веб-страниц. В 2016 году было проведено комплексное исследование с целью сбора и анализа маркетинговых материалов всех канадских колледжей[3]. Исследователи определили, что наибольшей популярностью пользуются современные объекты и нетрадиционные организационные символы.

---

[1] *Rennie S., Buchbinder M. Juengst E.* Scraping the Web for Public Health Gains: Ethical Considerations from a 'Big Data' Research Project on HIV and Incarceration // National Library of Medicine. – April 2020. – No. 13 (1). – URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7392638/.

[2] Doing Stigma: Online Commentary Around Weight-Related News Media / Philip Brooker [et al.] // New Media & Society. – December 2017. – No. 20 (9). – P. 1–22.

[3] *Pizarro M. R.* Modern Campuses, Local Connections, and Unconventional Symbols: Promotional Practises in the Canadian Community College Sector // Tertiary Education and Management. – September 2016. – No. 22:218-30. – URL: https://link.springer.com/article/10.1080/13583883.2016.1193764.

В рамках экономических исследований банк Японии опубликовал статью[1] об использовании веб-скрапинга для получения альтернативных данных, то есть данных, выходящих за рамки того, что обычно используют банки, например статистика ВВП и корпоративные финансовые отчеты. В статье было показано, что одним из источников альтернативных данных являются веб-скраперы, которые можно использовать для корректировки индексов цен.

Разработка продукта

У вас есть бизнес-идея и для ее воплощения вам нужна доступная база данных с относительно общеизвестной информацией? У вас не получилось найти недорогой и удобный источник такой информации? Тогда, возможно, вам подойдет веб-скрапер.

Это реальный способ быстро получить данные, которые позволят создать минимально жизнеспособный продукт. Вот несколько ситуаций, когда веб-скрапер может оказаться удачным решением.

Туристический сайт со списком популярных туристических направлений и мероприятий

Для этого не годится база данных с простой географической информацией. Вам будет полезно знать, что туристы, посещающие Рио-де-Жанейро (Бразилия), обязательно захотят увидеть статую Христа-Искупителя. Справочник заведений тоже плохо подходит для этой цели. Если Британский музей может заинтересовать туристов, то магазин Сейнсбери, расположенный дальше по улице, не обладает такой привлекательностью. Однако существует множество сайтов, посвященных обзорам путешествий, которые уже содержат информацию о популярных у туристов достопримечательностях.

Блог с обзорами товаров

Соберите список названий товаров и ключевых слов или описаний, а остальное заполните с помощью генеративного искусственного интеллекта.

Поскольку речь зашла об искусственном интеллекте, следует отметить, что для обучения этих моделей требуются большие объемы данных! Независимо от того, собираетесь вы прогнозировать тенденции или создавать реалистичные описания

[1] *Kameda S.* Use of Alternative Data in the Bank of Japan's Research Activities // Bank of Japan Review. – January 2022. – No. 2022-E-1. – URL: https://www.boj.or.jp/en/research/wps_rev/rev_2022/data/rev22e01.pdf.

на естественном языке, веб-скрапинг поможет вам получить отличный обучающий набор данных для вашего продукта.

Для многих бизнес-проектов в сфере услуг часто необходимы тщательно охраняемые отраслевые знания, такие как списки поставщиков промышленных материалов, контактная информация экспертов в узкоспециализированных областях или открытые вакансии в компаниях. Веб-скрапер может собрать воедино фрагменты этой информации, найденные в различных местах, что позволит вам создать исчерпывающую базу данных с относительно небольшими первоначальными затратами.

Туризм

Туристическая индустрия заслуживает особого признания за множество разработанных в ней приложений для парсинга веб-страниц, которые помогают и тем, кто решил начать туристический бизнес, и тем, кто желает сэкономить на поездке в отпуск.

Отели, авиакомпании и компании по аренде автомобилей предлагают довольно узкий спектр услуг и имеют множество конкурентов. Это означает, что их цены, как правило, мало отличаются друг от друга и испытывают частые колебания во времени из-за необходимости реагировать на рыночные условия.

Такие веб-сайты, как Kayak и Trivago, могут быть достаточно большими и мощными, чтобы платить за использование их API или предоставлять им свои API, но любой компании приходится с чего-то начинать. Веб-скрапер может стать отличной основой для нового сайта-агрегатора в туристической отрасли, который отыскивает для пользователей лучшие предложения в Интернете.

Может быть, вы и не собираетесь начинать новый бизнес, но наверняка летали на самолете или планируете сделать это в будущем. Если вы ищете идеи для проверки навыков, полученных в этой книге, я настоятельно рекомендую в качестве первого проекта написать скрапер для сбора информации с туристических сайтов. Огромный объем и хронологические колебания данных, предлагаемых ими, создают некоторые инженерные сложности, которые будет интересно решить.

Туристические сайты также являются хорошим примером сферы для приложения сил по защите от скрапинга. Они стремятся делать свои сайты простыми для сканирования и индексирования поисковыми системами, а также удобными и доступными для всех пользователей. Но испытывают жесткую конкуренцию со стороны других туристических сайтов, что может потребовать применения продвинутых методов защиты, описанных далее в этой книге. Изучение

заголовков и файлов cookie, посылаемых вашим браузером, станет для вас хорошим первым шагом.

Если вас заблокировали на каком-то туристическом сайте и вы не знаете, как получить его содержимое с помощью сценария на Python, то не отчаивайтесь, вы наверняка сможете найти другой туристический сайт с точно такими же данными, на котором сможете опробовать свои идеи.

Продажи

Веб-скраперы — идеальный инструмент для привлечения потенциальных клиентов. Если вам известен веб-сайт, где можно найти контактную информацию лиц, составляющих ваш целевой рынок, то остальное несложно. Неважно, насколько специализированной является ваша область. При работе с клиентами, занимающимися продажами, мне приходилось составлять списки тренеров молодежных спортивных команд, владельцев фитнес-залов, поставщиков средств по уходу за кожей и представителей многих других целевых аудиторий.

Индустрия подбора персонала (которую я рассматриваю как разновидность продаж) часто использует веб-скраперы, чтобы собирать информацию как о кандидатах, так и о вакансиях. Из-за строгой политики LinkedIn по борьбе со скрапингом стали набирать популярность плагины, например Instant Data Scraper или Dux-Soup, собирающие информацию из профилей кандидатов при их просмотре вручную в браузере. Это позволяет рекрутерам быстро просмотреть кандидатов, убедиться, что они подходят под описание вакансии, и затем сохранить данные со страницы.

Каталоги, подобные Yelp, могут помочь в поиске заведений по таким параметрам, как дороговизна, прием оплаты кредитными картами, наличие доставки, продажа спиртных напитков. Несмотря на то что Yelp больше известен обзорами ресторанов, он также содержит подробную информацию о местных организациях, розничных магазинах, бухгалтерах, авторемонтных мастерских и многом другом.

Сайты вроде Yelp не просто рекламируют предприятия — контактную информацию можно использовать и для рассылки рекламных объявлений, а инструменты фильтрации помогут сузить целевой рынок.

Скрапинг сайтов предприятий со списками сотрудников тоже может стать ценным источником контактной информации для выработки персонализированных предложений. Проверка тегов структурированных данных, добавляемых поисковой системой Google (см. раздел «Скрапинг результатов поиска» ниже) — хорошая стратегия для создания универсального веб-скрапера, способного работать со многими сайтами и собирать надежную, хорошо отформатированную контактную информацию.

Почти все примеры в этой книге посвящены скрапингу содержимого веб-сайтов — подаче представленной на них информации в удобочитаемом формате. Однако полезной может оказаться даже сугубо техническая информация о сайте: применяемая система управления контентом, используемый на стороне сервера технологический стек, тип чат-бота или системы аналитики, присутствующей на сайте, и т. д.

Сведения о том, какие технологии уже имеются или могут понадобиться потенциальному клиенту, очень пригодятся для продаж и маркетинга.

Скрапинг результатов поиска

Скрапинг *страниц результатов поисковых систем* (Search Engine Results Page, SERP) — это практика извлечения полезных данных непосредственно из результатов, возвращаемых поисковой системой, без перехода на найденные страницы. Результаты поисковых систем обладают важным преимуществом: они имеют известный и согласованный формат. Страницы, ссылки на которые возвращают поисковые системы, имеют разные и неизвестные форматы. Работа с ними — трудная задача, которой лучше избегать.

В компаниях, занимающихся поисковыми системами, трудятся специалисты, работа которых заключается в анализе метаданных, грамотном программировании и использовании инструментов искусственного интеллекта для извлечения обобщенных статистических данных и ключевых слов из страниц. Используя результаты их труда вместо попыток сделать все самостоятельно, можно сэкономить много времени и денег.

Например, турнирные таблицы крупных американских спортивных лиг за последние 40 лет можно найти в различных источниках и, что самое неприятное, в разных форматах. На сайте **http://nhl.com**, например, турнирная таблица по хоккею доступна в одном формате, а на сайте **http://nfl.com** — в другом. Однако поиск в Google по запросу «результат НБА 2008» или «результат ГЛБ 2004» выдаст данные в едином формате с возможностью детализации результатов по отдельным играм и игрокам за указанный сезон.

Вам также может пригодиться информация о наличии и местоположении искомой информации в самих результатах поиска, например, какие отображаются веб-сайты и в каком порядке. Это может помочь следить за популярностью вашего бренда и ваших конкурентов.

Если вы проводите рекламную кампанию в поисковых системах или размышляете о ее запуске, то можете отслеживать только рекламу, а не все результаты. Опять же, вы можете изучать, какие объявления и в каком порядке появляются и, может быть, как эти результаты меняются с течением времени.

Кстати, не ограничивайтесь только основной страницей поиска. У Google, например, есть Google Maps (Карты Google), Google Images (Картинки Google), Google Shopping (Покупки Google), Google Flights (Рейсы Google), Google News (Новости Google) и т. д. Все это — поисковые системы для различных типов контента, которые могут иметь отношение к вашему проекту.

Даже если вы не собираете данные из самой поисковой системы, вам может быть полезно узнать больше о том, как поисковые системы отыскивают и распознают данные. Поисковые системы не играют в угадайку, выясняя назначение данных; они требуют, чтобы веб-разработчики оформляли контент особым образом для его отображения третьими лицами, такими как они сами.

Документацию по структурированным данным Google можно найти по адресу **https://developers.google.com/search/docs/appearance/structured-data**. Если вы встретите эти данные при скрапинге страниц из Интернета, то будете знать, как их использовать.

ГЛАВА 4

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, сценариев JavaScript и рендеринга изображений Интернет с непривычки может показаться слегка пугающим. В четвертой и пятой главах вы увидите, как форматировать и интерпретировать данные, не прибегая к помощи браузера.

Для начала рассмотрим основы отправки на веб-сервер `GET`-запросов — запросов на выборку или получение содержимого заданной веб-страницы, чтения ее HTML-кода и выполнения ряда простых операций по извлечению данных, чтобы выделить оттуда искомый контент.

Установка и использование Jupyter

Код примеров для этой книги можно найти по адресу **https://github.com/REMitchell/python-scraping**. Большинство примеров представлено в виде файлов Jupyter Notebook с расширением `.ipynb`.

Для тех, кто слышит это название впервые, сообщаю, что Jupyter Notebooks — отличный способ организации и использования множества небольших и взаимосвязанных фрагментов кода на Python, как показано на рис. 4.1.

Каждый фрагмент кода помещается в блок, называемый *ячейкой*. Код внутри ячейки можно запустить комбинацией клавиш **Shift + Enter** или нажав кнопку **Run** (Выполнить) вверху страницы.

Программа Jupyter была запущена в 2014 году в рамках проекта IPython (Interactive Python — интерактивный Python). Эти интерактивные блокноты были разработаны для запуска кода на Python в браузере, чтобы использовать его для обучения и презентаций.

Установить Jupyter Notebook можно командой:

```
$ pip install notebook
```

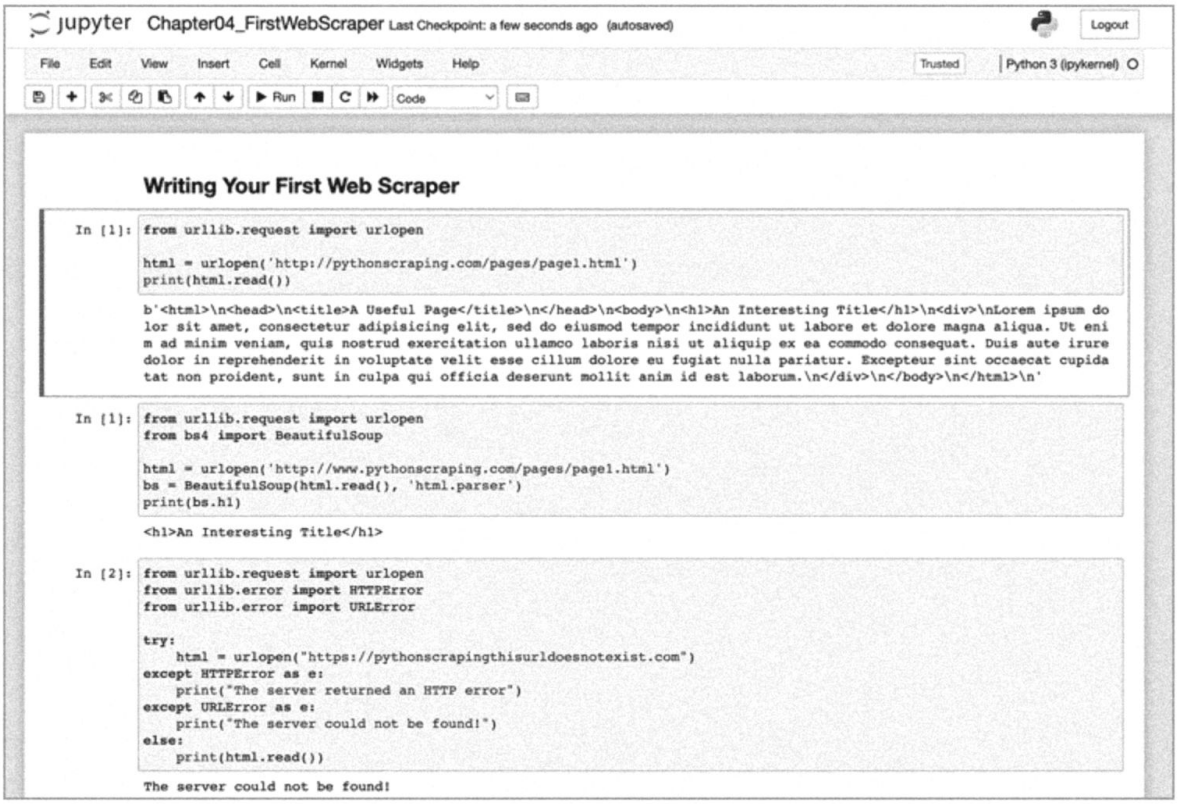


Рис. 4.1. Блокнот в формате Jupyter Notebook с кодом на Python, работающий в браузере

После установки вы сможете запустить веб-сервер командой `jupyter`. Перейдите в каталог с загруженными файлами упражнений для этой книги и выполните команду:

```
$ jupyter notebook
```

Она запустит веб-сервер на порту 8888. Если к этому моменту у вас уже был запущен веб-браузер, то он автоматически должен открыть новую вкладку. В противном случае скопируйте из окна терминала URL с токеном в адресную строку браузера.

Установка соединения

В первой главе этой книги вы узнали, как Интернет пересылает пакеты данных по проводам от браузера к веб-серверу и обратно. Когда вы открываете браузер, вводите `google.com` и нажимаете **Enter**, происходит следующее: данные в виде HTTP-запроса передаются с вашего компьютера веб-серверу Google, а тот отвечает HTML-файлом, который представляет начальную страницу *google.com*.

В чем же при этом обмене данными участвует браузер? Ответ: ни в чем. В действительности ARPANET (первая общедоступная сеть с коммутацией пакетов) появилась на 20 лет раньше первого браузера Nexus.

Конечно, браузер — полезное приложение, которое создает эти информационные пакеты, передает их операционной системе для отправки 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 отправить 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 вызвать именно Python 3.x, выполнив команду следующим образом:

```
$ python3 scrapetest.py
```

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

Почему важно воспринимать эти адреса как файлы, а не как страницы? Большинство современных веб-страниц связано со множеством файлов ресурсов. Это могут быть файлы изображений, сценариев JavaScript, стилей CSS и любой другой контент, на который ссылается запрашиваемая страница. Например, встретив тег `<img src="cuteKitten.jpg">`, браузер знает: чтобы сгенерировать страницу для пользователя, нужно послать еще один запрос серверу и получить данные из файла `cuteKitten.jpg`.

Разумеется, у нашего сценария 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 `request` (в библиотеке `urllib`) и импортирует оттуда одну функцию — `urlopen`.

Библиотека `urllib` — это стандартная библиотека Python (иными 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 использовать `urllib`, 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 достаточно универсальная функция (она одинаково легко читает HTML-файлы, файлы изображений и другие файловые потоки), мы будем весьма 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).

Суть приложения BeautifulSoup — найти смысл в бессмысленном: помочь отформатировать и упорядочить перемешанные сетевые данные, исправляя ошибки в HTML-коде и создавая легко прослеживаемые объекты Python, отражающие структуры 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».

Для тех, кому еще не приходилось устанавливать библиотеки Python (или кто подзабыл, как это делается), я покажу типичный подход, который мы будем использовать для установки библиотек в рамках всей книги, так что впоследствии вы сможете обращаться к данному подразделу.

Мы будем использовать библиотеку BeautifulSoup 4 (также известную как BS4). Подробная инструкция по ее установке размещена на сайте Crummy.com (**http://www.crummy.com/software/BeautifulSoup/bs4/doc/**).

Если вы уже давно программируете на Python, то, вероятно, знакомы с установщиком пакетов `pip` (package installer for Python: **https://pypi.org/project/pip/**). Если ранее вы не работали с `pip`, то установите его, чтобы затем загрузить BeautifulSoup и другие пакеты Python, используемые в этой книге.

В зависимости от того, каким способом был установлен Python, программа `pip` уже может присутствовать в вашей системе. Для проверки выполните команду:

```
$ pip
```

Если установщик `pip` уже имеется в системе, эта команда выведет текст справки. Если команда не будет распознана, то вам придется установить `pip`. Сделать это можно разными способами, например с помощью диспетчера пакетов `apt-get` в Linux или `brew` в macOS. Также в любой операционной системе можно загрузить файл начального развертывания `pip`, доступный по адресу **https://bootstrap.pypa.io/get-pip.py**, сохранить его с именем `get-pip.py` и запустить с помощью Python:

```
$ python get-pip.py
```

Подчеркну еще раз: если на вашем компьютере установлены Python 2.x и 3.x, вам следует явно вызвать `python3`:

```
$ python3 get-pip.py
```

После всех этих манипуляций можно, наконец, установить BeautifulSoup с помощью `pip`:

```
$ pip install bs4
```

Если у вас установлены две версии Python и две версии `pip`, вам может потребоваться вызвать `pip3`, чтобы установить пакеты для Python 3.x:

```
$ pip3 install bs4
```

Готово! Теперь компьютер будет распознавать BeautifulSoup как библиотеку Python. Вы можете в этом убедиться, если откроете терминал Python и импортируете ее:

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

Импорт должен выполниться без ошибок.

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 вне 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 создаст новое окружение с именем `scrapingEnv`. Чтобы использовать данное окружение, его необходимо активировать:

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

После активации окружения его имя появится в префиксе командной строки, подсказывая, в каком именно окружении вы работаете. Все устанавливаемые библиотеки и запускаемые сценарии 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 командой `deactivate`, после чего библиотеки, установленные внутри, 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'
```

Такой подход с установкой библиотек в каждый проект позволяет 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 на использовании виртуального окружения для выполнения примеров из этой книги, однако помните: вы можете воспользоваться виртуальным окружением 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 необязательно 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 — парсер, который `BeautifulSoup` будет использовать для создания объекта. В большинстве случаев не имеет значения, какой именно парсер будет использоваться.

Парсер `html.parser` входит в состав Python 3 и не требует дополнительной настройки перед использованием. В этой книге он будет применяться крайне редко.

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

```
$ pip install lxml
```

Чтобы использовать `lxml` в `BeautifulSoup`, нужно изменить имя парсера 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. Парсер `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 парсера не всегда является преимуществом, поскольку почти всегда главное слабое место — скорость самого сетевого соединения.



Избегайте чрезмерной оптимизации кода веб-скрапинга

Элегантные алгоритмы вызывают эстетическое удовольствие, но когда дело доходит до парсинга веб-страниц, все это может не иметь практического значения. Несколько микросекунд, выигранных на обработке, скорее всего, будут незаметны на фоне нескольких секунд сетевых задержек.

Хороший код скрапинга обычно ориентирован на надежность и удобочитаемость реализации, а не на оптимизацию обработки.

Одним из недостатков 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 парсер 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, что это небольшое знакомство с *BeautifulSoup* дало вам 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 5 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 назавтра иметь все данные в своей 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 хочется проклинать разработчика, который создал тот сайт (и выбрал странный формат представления данных). Однако на самом деле если кому и стоит дать пинка, то именно вам, поскольку именно вы не предусмотрели форс-мажор!

Рассмотрим первую строку нашего скрапера сразу после операторов импорта и подумаем, как можно обработать любые непредвиденные ситуации, которые могут 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.
```

Если теперь код столкнется с ошибкой HTTP, то он выведет сообщение о ней и остальная часть программы, которая находится в ветви `else`, не выполнится.

Если сервер вообще не будет найден (например, если сервер по адресу http://www.pythonscraping.com отключен или URL указан с ошибкой), то функция `urlopen` вернет `URLError`. Эта ошибка говорит о том, что ни один из указанных серверов недоступен. 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 следующую 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 в объекте None приведет к ошибке 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 пытаться 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)
```

Такая проверка и обработка каждой ошибки поначалу может показаться трудоемкой, 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/page1.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 мог бы обрабатывать исключения, но при этом быть читабельным. Возможно, вы захотите использовать код многократно. Наличие обобщенных функций, таких как **getSiteHTML** и **getTitle** (в сочетании с тщательной обработкой исключений), позволяет быстро и надежно сканировать сеть.

ГЛАВА 5

Углубленный парсинг HTML

Однажды Микеланджело спросили, как он создает свои шедевры. На что он ответил: «Я просто беру глыбу мрамора и отсекаю все лишнее».

При извлечении информации из сложных веб-страниц стоит придерживаться аналогичного подхода. Существует множество способов отсекать все лишнее, пока не останется только нужная информация. В этой главе вы узнаете, как выполнять анализ сложных HTML-страниц, чтобы извлекать из них только необходимую вам информацию.

Еще одна тарелка BeautifulSoup

В главе 4 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 HTML-кода веб-страницы.

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. Назначая 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.[1]
```

С помощью программы, аналогичной той, которая была рассмотрена в главе 4, 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 `<span class="green"></span>` (`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. Как же это работает? 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, но и 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.

---

[1] Цитата из романа Л. Толстого «Война и мир». В оригинале: «Dieu, quelle virulente sortie — отвечал, нисколько не смутясь такою встречей, вошедший князь...» — *Примеч. пер.*



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, и останется только текст.

Учтите, что в объекте BeautifulSoup гораздо проще найти нужное, чем в текстовом фрагменте. Вызов `.get_text()` всегда должен быть последним в последовательности выполняемых операций непосредственно перед выводом результата на экран, сохранением или манипулированием готовыми данными. Это значит, что вам следует сохранять структуру тегов документа как можно дольше.

Функции find() и find_all() объекта BeautifulSoup

Функции `find()` и `find_all()` объекта `BeautifulSoup` 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, attrs, recursive, text, limit, **kwargs)
find(tag, attrs, recursive, text, **kwargs)
```

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

Параметр `tag` нам уже встречался; мы можем передать в нем строку, содержащую имя тега, или даже Python-список имен тегов. Например, следующий код возвращает список всех тегов заголовков, встречающихся в документе[1]:

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

Параметр `attrs` принимает Python-словарь атрибутов и ищет теги, которые содержат любой из этих атрибутов. Например, следующий вызов отыщет в HTML-документе теги `span` с классом `green` *или* `red`:

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

[1] Для получения списка всех тегов *h<уровень>*, имеющихся в документе, существуют более лаконичные варианты кода. Есть и другие способы решения таких задач, которые мы рассмотрим в разделе, посвященном регулярным выражениям.

Параметр `recursive` — логический. Насколько глубоко вы хотите исследовать документ? Если в `recursive` передать значение `True`, то функция `find_all` будет искать теги, соответствующие заданным параметрам, в дочерних элементах и их потомках. Если в этом параметре передать `False`, то функция будет просматривать только теги верхнего уровня документа. По умолчанию `find_all` работает рекурсивно (`recurive` имеет значение `True`); обычно лучше оставить все как есть, за исключением ситуаций, когда вы точно знаете, что делаете, и нужно обеспечить высокую производительность.

Параметр `text` необычен тем, что определяет текстовое содержимое, а не свойства тегов. Так, чтобы узнать, сколько раз на странице встречаются слова *the prince*, заключенные 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.

Параметр `limit` по понятным причинам используется только в методе `find_all`; функция `find` эквивалентна вызову `find_all` со значением `limit`, равным 1. Этот параметр можно использовать в тех случаях, когда нужно 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.

Параметр `kwargs` позволяет передать в вызов метода любые дополнительные именованные аргументы. Именованные аргументы, которые `find` и `find_all` не смогут распознать, будут интерпретироваться как определения искомых атрибутов и их значений. Например:

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

Этот код вернет первый тег со словом `text` в атрибуте `class` и словом `title` в атрибуте `id`. Обратите внимание: по соглашению все атрибуты `id` на странице должны иметь уникальные значения. Поэтому на практике такой вызов может быть не особенно полезен и должен быть эквивалентен следующему:

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

Возможно, вы заметили, что в BeautifulSoup уже есть способ поиска тегов по их атрибутам и значениям: параметр `attr`. И действительно, следующие две строки идентичны:

```
bs.find(id='text')
bs.find(attrs={'id':'text'})
```

АРГУМЕНТ _CLASS И АТРИБУТ CLASS

В языке Python `class` — зарезервированное слово, которое нельзя использовать в качестве имени переменной или аргумента. Например, попытка выполнить следующий код приведет к синтаксической ошибке из-за нестандартного использования слова `class`:

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

По этой причине BeautifulSoup требует использовать именно аргумент `_class` вместо атрибута `class`.

Первый способ короче, быстрее и удобнее в случаях, когда нужно выбрать теги с определенными атрибутами. Но когда требуется нечто более сложное или нужно передать несколько вариантов значений атрибута в виде списка, то лучше использовать параметр **`attrs`**:

```
bs.find(attrs={'class':['red', 'blue', 'green']})
```

Другие объекты BeautifulSoup

До сих пор в 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-таки 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 **`NavigableStrings`**);
- *объекты* **`Comment`** — применяются для поиска HTML-комментариев, заключенных в теги комментариев, `<!-- например, так -->`.

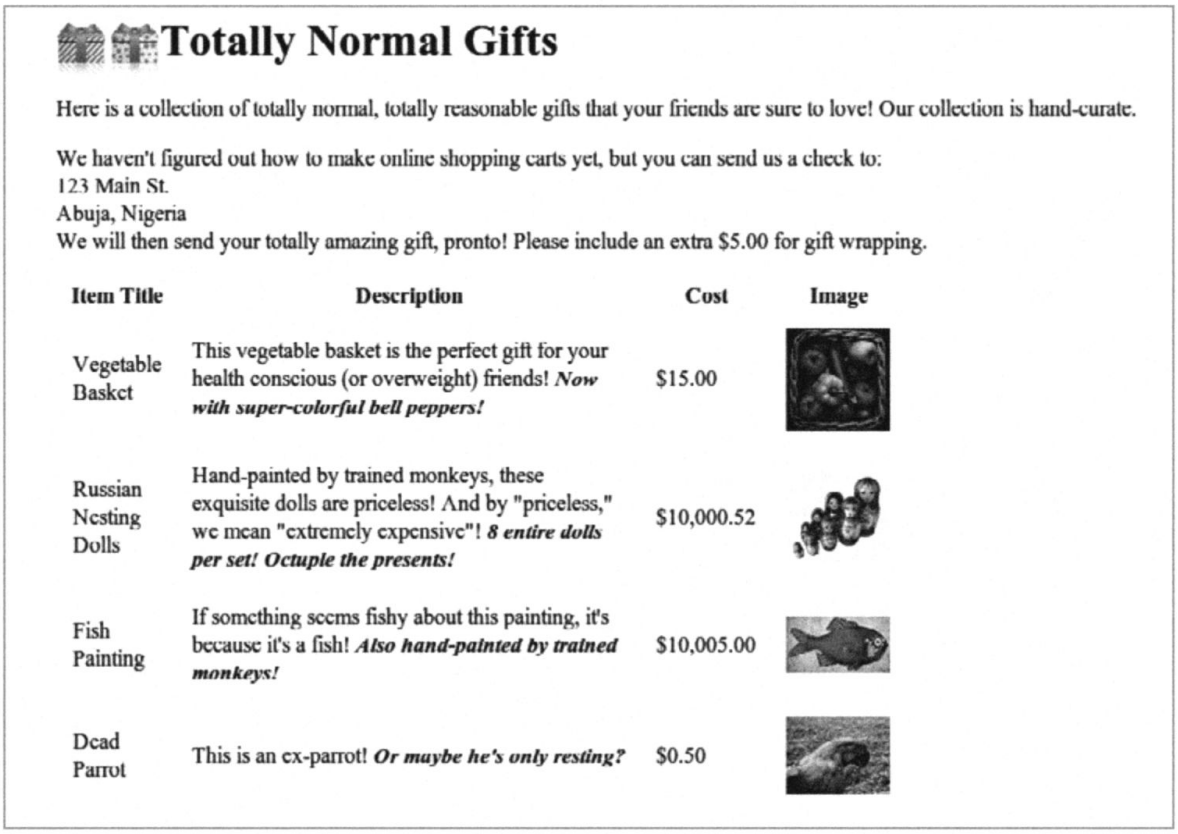
На момент написания этих строк в пакете BeautifulSoup имелись только эти четыре объекта. Они также были единственными объектами в пакете BeautifulSoup на момент его выпуска в 2004 году, поэтому количество доступных объектов вряд ли изменится в ближайшем будущем.

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 4 мы рассмотрели навигацию по дереву BeautifulSoup только в одном направлении:

```
bs.тег.вложенныйТег.ещеОдинВложенныйТег
```

Теперь рассмотрим навигацию по деревьям HTML во всех направлениях: по вертикали, горизонтали и диагонали. В качестве образца для веб-скрапинга будем использовать весьма сомнительный интернет-магазин, размещенный по адресу **http://www.pythonscraping.com/pages/page3.html**. Он показан на рис. 5.1.



Totally Normal Gifts

Here is a collection of totally normal, totally reasonable gifts that your friends are sure to love! Our collection is hand-curate.

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.

Item Title	Description	Cost	Image
Vegetable Basket	This vegetable basket is the perfect gift for your health conscious (or overweight) friends! ***Now with super-colorful bell peppers!***	$15.00	
Russian Nesting Dolls	Hand-painted by trained monkeys, these exquisite dolls are priceless! And by "priceless," we mean "extremely expensive"! ***8 entire dolls per set! Octuple the presents!***	$10,000.52	
Fish Painting	If something seems fishy about this painting, it's because it's a fish! ***Also hand-painted by trained monkeys!***	$10,005.00	
Dead Parrot	This is an ex-parrot! ***Or maybe he's only resting?***	$0.50	

Рис. 5.1. 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
   - ...остальные 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 тега `div`.

Найти 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)
```

Код выведет 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 выведет все строки таблицы, кроме первой с заголовком. Почему пропускается строка заголовка? Потому что объект не может быть братским самому себе. Каждый раз, когда составляется список братьев объекта, сам объект не включается в этот список. Как следует из названия, данная функция выбирает только *следующих* по списку братьев. Например, если выбрать строку из середины таблицы и вызвать для нее функцию `next_siblings`, то функция вернет только те строки, которые следуют в списке после выбранной. 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 выбирать теги как можно более точно. 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 возвращают не список тегов, а лишь один тег.

Работа с родителями

Занимаясь скрапингом веб-страниц, вы поймете, что выбирать родительский тег необходимо реже, чем детей или братьев/сестер. Как правило, просмотр HTML-страницы с целью поиска данных мы начинаем с тегов верхнего уровня, после чего углубляемся 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 выведет цену объекта, изображенного на картинке `../img/gifts/img1.jpg` (в данном случае цена составляет 15 долларов).

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

```
<tr>
- td
- td
- td ❸
  - "$15.00" ❹
- td ❷
  - <img src="../img/gifts/img1.jpg"> ❶
```

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

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

❸ С помощью функции `previous_sibling` выбираем братский тег, предшествующий этому тегу `td` (в данном случае — тег `td`, который содержит цену продукта в долларах).

❹ 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 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, вместо которых можно было бы обойтись всего одной строкой с регулярным выражением!

Регулярные выражения — бесценный инструмент для веб-скрапинга. К счастью для вас, освоить регулярные выражения не так уж сложно. Чтобы их изучить, достаточно рассмотреть всего несколько простых примеров и поэкспериментировать с ними.

Регулярные выражения получили свое название благодаря тому, что используются для идентификации строк, записанных с применением *регулярного языка*. Слово «язык» здесь означает нечто иное, нежели в словосочетаниях «язык программирования» или «естественный язык» (например, английский или французский). В данном случае это математическое понятие, означающее набор строк, подчиняющихся некоторым правилам.

Регулярный язык — это набор строк, которые можно сгенерировать с применением набора линейных правил и использовать, сопоставляя с исследуемой строкой и таким способом проверяя ее соответствие правилам[1]. Например.

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*.

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

[1] 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 был бы должен находить подобные строки.

удобно для быстрого сканирования больших документов, в которых нужно найти номера телефонов или адреса электронной почты.

Вышеописанным правилам соответствуют строки *aaaabbbbbccccd*, *aabbbbbcce* и т. д. С математической точки зрения существует бесконечное количество строк, соответствующих этому шаблону.

Регулярные выражения — всего лишь краткий способ представления этих наборов правил. Например, регулярное выражение, описывающее вышеприведенную последовательность шагов, 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`, за которой следует комбинация `a*` (*a-звездочка*), означающая «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 `c` мы можем написать две буквы `c`, заключить их в скобки и поставить после них звездочку. Это значит, что в строке может присутствовать любое количество пар, состоящих из букв `c` (обратите внимание, что это также может означать 0 пар);
- `(d|e)` — вертикальная линия между двумя выражениями означает «или». В данном случае мы говорим «добавить `d` или `e`». Таким образом, мы гарантируем, что в строку добавится только 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. 5.1 приведены наиболее часто используемые символы регулярных выражений с краткими пояснениями и примерами. Этот список далеко не полный. Кроме того, как уже упоминалось, в разных языках могут наблюдаться небольшие различия. Однако приведенные здесь 12 символов наиболее часто встречаются в регулярных выражениях Python и могут служить для поиска и сбора практически любых строк.

Таблица 5.1. Часто используемые символы регулярных выражений

Символ	Значение	Пример	Соответствующие строки
*	Предыдущий символ, подвыражение или символ в скобках повторяется 0 или более раз	a*b*	aaaaaaaa, aaabbbbb, bbbbbb
+	Предыдущий символ, подвыражение или символ в скобках повторяется 1 или более раз	a+b+	aaaaaaaab, aaabbbbb, abbbbbb
[]	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}	aabbb, aaabbb, 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 *символом конвейера*)	b(a|i|e)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, zzzyx, Bob
?!	«Не содержит». Эта странная пара символов, непосредственно предшествующая символу (или регулярному выражению), указывает на то, что данный символ не должен присутствовать в данном 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. 5.2.

Таблица 5.2. Точные правила составления адресов электронной почты и регулярные выражения для каждого из них

Правило 1 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 «подчеркивание» (_)	**[A–Za–z0–9._+]+** 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 будто 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 Адрес электронной почты должен содержать хотя бы одну букву: заглавную или строчную	**[A–Za–z]+** В первой части имени домена, после символа @, допустимы только буквы, и их должно быть не менее одной
Правило 4 Потом следует точка (.)	**.** Перед доменом верхнего уровня должна стоять точка (.)
Правило 5 Наконец, адрес электронной почты заканчивается доменом верхнего уровня: com, org, edu или net (в действительности их намного больше, но для нашего примера этих четырех должно быть достаточно)	**(com|org|edu|net)** 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 написании регулярного выражения с нуля лучше сначала составить список шагов, которые четко описывали бы, 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. На самом деле большинство функций, принимающих строки в качестве аргумента (например, `find(id="идентификаторТега")`), принимают и регулярные выражения.

Рассмотрим несколько примеров, просканировав страницу по адресу **http://www.pythonscraping.com/pages/page3.html**.

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

```
<img src="../img/gifts/img3.jpg">
```

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, 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 изображения рекомендуемого товара может быть предусмотрено специальное место вверху 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 выведет 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 с помощью 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 становятся тривиальной задачей. Например, для того чтобы узнать, где находится файл с изображением, можно воспользоваться следующим кодом:

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

Лямбда-выражения

Лямбда — это забавный академический термин, который в программировании означает всего лишь краткий способ записи функций. Вот как на Python можно написать функцию, возвращающую квадрат числа:

```
def square(n):
    return n**2
```

Ту же функцию можно определить в одной строке:

```
square = lambda n: n**2
```

Здесь переменной `square` присваивается функция, которая принимает один аргумент `n` и возвращает `n**2`. Но не существует правила, согласно которому функции вообще должны быть «именованными» или присваиваться переменным. Мы можем записывать их как простые значения:

```
>>> lambda r: r**2
<function <lambda> at 0x7f8f88223a60>
```

По сути, *лямбда-выражение* — это функция, которая существует сама по себе, без имени и без присваивания переменной. В Python лямбда-функция может располагаться только в одной строке кода (однако это вопрос стиля и хорошего вкуса, касающийся исключительно языка Python, а не какое-то фундаментальное правило информатики).

Чаще всего лямбда-выражения передаются как аргументы другим функциям. BeautifulSoup позволяет передавать функции определенных типов в качестве параметров в функцию `find_all`.

Единственное ограничение — 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`, то функция `find_all` возвращает 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.

Иногда молоток не нужен

Столкнувшись с гордиевыми узлами тегов, многие испытывают острое желание поработать с ними 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 сложным для отладки, или начнет постоянно сбоить, или же и то и другое. Прежде чем начать, рассмотрим несколько способов, которые позволяют вообще обойтись без углубленного парсинга HTML!

Предположим, мы ищем некий контент: имя, статистические данные или блок текста. Возможно, этот контент спрятан под 20 тегами в каше HTML 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 либо разместит в верхней части страницы еще один компонент (с несколькими тегами `div`)? Показанная выше строка кода ненадежна: она опирается на то, что структура сайта никогда не изменится.

Что же делать?

- Ищите любые ориентиры, которые можно использовать, чтобы перейти прямо в середину документа, как можно ближе к нужному вам контенту. Одним таким очевидным ориентиром могут служить атрибуты CSS, но также можно проявить немного смекалки и отобрать теги по их содержимому, используя `.find_all(text='искомое содержимое тега')`.
- Если нет простого способа отыскать нужный вам тег или любого из его родителей, то можно попробовать найти братский элемент, а затем через промежуточный вызов метода `.parent` перейти к целевому тегу.
- Найдите ссылку **Print This Page** (Распечатать эту страницу) или, как вариант, 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 17).
- Поищите информацию в файле JavaScript или в тегах `<script>`. Код на JavaScript часто содержит нужные вам данные и в более удобном формате! Например, однажды я получила с сайта адреса улиц (вместе с широтой и долготой) в виде аккуратно отформатированного массива, просто заглянув в код JavaScript для встроенной карты Google, на которой были точно отмечены все адреса. Дополнительную информацию об этом способе вы найдете в главе 11.
- Данные могут быть доступны в URL-адресе самой страницы. Например, там часто можно найти заголовки страниц и идентификаторы продуктов.
- 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. Может быть, существует другой сайт с теми же данными. Возможно, там приводится информация, скопированная или агрегированная с другого сайта.

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

Представленные здесь методы при условии правильного использования помогут значительно облегчить написание более стабильных и надежных веб-краулеров.

ГЛАВА 6

Разработка веб-краулеров

До сих пор нам встречались лишь отдельные статические страницы с немного искусственно подобранными примерами. В этой главе вы познакомитесь 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-адресу, исследует эту страницу, находит URL другой страницы, извлекает содержимое *этой другой* страницы и т. д. 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 внимательным к тому, какую часть пропускной способности сети вы задействуете, и всеми силами постараться определить, есть ли способ облегчить нагрузку на интересующий вас сервер.

Обход отдельного домена

Даже если вам еще не приходилось слышать об игре «Шесть шагов по "Википедии"», вы, возможно, слышали о ее одноименной версии «Шесть шагов до Кевина Бейкона»[1]. 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).

[1] Популярная игра, придуманная в 1990-х годах, https://ru.wikipedia.org/wiki/Шесть_шагов_до_Кевина_Бейкона.

Например, Эрик Айдл снялся в фильме «Дадли Справедливый» с Бренданом Фрейзером, который, в свою очередь, снялся с Кевином Бейконом в драме «Воздух, которым я дышу»[1]. В данном 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 приходятся на домен «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 окажут заметное влияние на нагрузку сервера «Википедии». Тем не менее если вы будете активно использовать примеры кода, приведенные в этой книге, или разрабатывать собственные проекты для веб-скрапинга «Википедии», то я призываю вас сделать необлагаемое налогом пожертвование в Фонд Викимедиа (https://wikimediafoundation.org/wiki/Ways_to_Give), чтобы не только компенсировать нагрузку на сервер, но и помочь сделать образовательные ресурсы более доступными для других пользователей.

Также имейте в виду: если вы планируете создать большой проект с применением данных из «Википедии», стоит убедиться, что эти данные еще недоступны через API «Википедии» (https://www.mediawiki.org/wiki/API:Main_page). «Википедия» часто используется в качестве сайта для демонстрации скраперов и краулеров, поскольку данный сайт имеет простую HTML-структуру и относительно стабилен. Однако эти же данные могут оказаться более доступными через API.

Вы уже, вероятно, знаете, как написать сценарий на Python, который получит произвольную страницу «Википедии» и создаст список ссылок на этой странице:

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

[1] Благодарю сайт The Oracle of Bacon (http://oracleofbacon.org/), который помог удовлетворить мое любопытство относительно этой цепочки.

```
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:

```
//foundation.wikimedia.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 со множеством ссылок, также есть ссылки на страницы категорий, обсуждений и другие страницы, которые не содержат полезных статей:

```
/wiki/Category:All_articles_with_unsourced_statements
/wiki/Talk:Kevin_Bacon
```

Недавно мой друг, работая над аналогичным проектом по веб-скрапингу «Википедии», заявил, что написал большую функцию фильтрации, состоящую из более чем 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 нашел бы 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/)((?!:).)*$')):
        print(link.attrs['href'])
```

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

Сценарий, который находит ссылки на все источники для одной строго заданной статьи «Википедии», конечно, интересен, однако с практической точки зрения довольно бесполезен. Вы должны уметь, взяв этот код за основу, преобразовать его примерно в следующую 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.

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.

ПСЕВДОСЛУЧАЙНЫЕ ЧИСЛА И СЛУЧАЙНЫЕ НАЧАЛЬНЫЕ ЗНАЧЕНИЯ

В предыдущем примере был задействован генератор случайных чисел Python для произвольного 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 трудно предсказуемую числовую последовательность, однако для запуска работы такого алгоритма необходимо начальное число. Если оно каждый раз будет одним и тем же, то алгоритм станет каждый 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*. Он генерирует трудно предсказуемые и равномерно распределенные случайные числа, однако несколько нагружает процессор. Случайные числа — это хорошо, но за все приходится платить!

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 будет, если, например, «Википедия» изменит имя тега `bodyContent`? Тогда при попытке извлечь текст из тега программа выдаст исключение `AttributeError`.

Таким образом, хотя эти сценарии достаточно хороши в качестве тщательно контролируемых примеров, автономный код готового приложения требует гораздо большей обработки исключений по сравнению с описанием в данной книге. Для получения дополнительной информации об этом вернитесь к главе 4.

Краулинг всего сайта

В предыдущем разделе мы прошли по сайту, перемещаясь от одной случайно выбранной ссылки к другой. Но как быть, если нужно систематизировать информацию на сайте и составить каталог или просмотреть все страницы? Для сбора информации со всего сайта, особенно крупного, требуется большой объем памяти, поэтому для решения такой задачи хорошо иметь доступ к быстрой базе данных, где можно хранить результаты 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 9.

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 глобальной сети, выходящая за пределы *видимого Интернета*[1]. Видимой называют ту часть Интернета, которая индексируется поисковыми системами. Несмотря на большой разброс оценок, глубокий Интернет почти наверняка составляет около 90 % всей Глобальной сети. 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 Всемирной паутины.

Темный Интернет, также известный как *Даркнет*, — нечто совершенно иное[2]. 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 Даркнете, 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 проводить веб-скрапинг и собирать информацию во многих местах, недоступных для ботов Google.

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

[1] См. статью: *Wright A.* Exploring a 'Deep Web' that Google Can't Grasp. http://nyti.ms/2pohZmu.

[2] См. статью: *Greenberg A.* Hacker Lexicon: What is the Dark Web? http://bit.ly/2psIw2M.

Формирование карты сайта

Несколько лет назад я столкнулась со следующей задачей: важный клиент хотел оценить затраты на редизайн сайта, однако не желал предоставлять моей компании доступ к 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 соответствии с фактической структурой папок, используемой 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 краулингу сайта заключается в том, чтобы начать со страницы верхнего уровня (например, с начальной страницы) и построить список всех ее внутренних ссылок. Затем просканировать все страницы, на которые указывают эти ссылки, и на каждой из них собрать дополнительные списки ссылок, запускающие очередной этап краулинга.

Конечно же, в такой ситуации количество ссылок стремительно растет. Если на каждой странице есть 10 внутренних ссылок, а глубина сайта составляет 5 страниц (что довольно типично для сайта среднего размера), то необходимо просканировать 10^5, то есть 100 000 страниц, чтобы с уверенностью утверждать: сайт пройден полностью. Как ни странно, хотя «5 страниц в глубину и 10 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 краулинга 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, во множестве содержатся исключительно уникальные элементы, что идеально подходит для наших целей. Следует сканировать 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):
    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, что 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 сценарий 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 слишком долго, то предыдущая программа, скорее всего, аварийно завершит работу.

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

Для «шаблонных» сайтов, содержащих менее 1000 ссылок, данный метод обычно работает хорошо, но бывают исключения. Например, однажды я столкнулась с ошибкой в динамически генерируемом URL, поскольку ссылка на следующую страницу зависела от адреса текущей. Это привело к бесконечно повторяющимся путям, таким как /blogs/blogs.../blogs/blog-post.php.

Тем не менее этот рекурсивный метод обычно должен подходить для любого типичного сайта, с которым вы, скорее всего, столкнетесь.

Сбор данных со всего сайта

Веб-краулеры были бы довольно скучными программами, если бы только и делали, что переходили с одной страницы на другую. Полезными они могут быть, именно что-то делая на странице, на которую перешли. Рассмотрим пример веб-скрапера, который бы выбирал заголовок, первый абзац содержимого 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 шаблон. 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**), в которых нет разделов с текстовым содержимым.
- 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):
    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 краулинга (для ясности добавлен вывод дефисов в качестве разделителей контента).

Поскольку никогда нельзя быть полностью уверенным 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 ситуаций

Очевидно, что, когда в блок обработчика исключений занесено несколько 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 9.

ОБРАБОТКА ПЕРЕНАПРАВЛЕНИЙ

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

1) на стороне сервера — URL изменяется до загрузки страницы;
2) на стороне клиента, иногда сопровождаются сообщением наподобие «через 10 секунд вы перейдете на другой сайт», — 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 `Requests` 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 14.

Краулинг нескольких сайтов

Каждый раз, когда я делаю доклад о веб-скрапинге, кто-нибудь обязательно спрашивает: «Как создать 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 начинал развиваться, команда его создателей состояла всего из двух аспирантов из Стэнфорда, у которых были 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 необязательно 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 переместить вас 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 вы — человек с высокими моральными принципами или религиозными убеждениями, которые не позволяют вам посещать порносайты, будьте осторожны, запуская программу.

Прежде чем приступать к написанию веб-краулера, который, независимо от ваших предпочтений, 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?
- Попав 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 подробно?
- 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 могу защитить себя от судебных исков, если мой веб-краулер привлечет внимание веб-мастера одного из сайтов, на которые попадет? (Подробнее об этом см. в главе 2.)

Менее чем в 60 строках кода можно без труда уместить гибкий набор функций Python, комбинация которых дает различные типы веб-скраперов. Здесь я для краткости опустила импорт библиотек и разбила код на несколько разделов для удобства обсуждения. Однако полную версию этого кода можно найти в репозитории GitHub (https://github.com/REMitchell/python-scraping):

```
# Получить список всех внутренних ссылок, найденных на странице
def getInternalLinks(bs, url):
    netloc = urlparse(url).netloc
    scheme = urlparse(url).scheme
    internalLinks = set()
    for link in bs.find_all('a'),
        if not link.attrs.get('href'):
            continue
        parsed = urlparse(link.attrs['href'])
        if parsed.netloc == '':
            l = f'{scheme}://{netloc}/{link.attrs["href"].strip("/")}'
            internalLinks.add(l)
        elif parsed.netloc == internal_netloc:
            internalLinks.add(link.attrs['href'])
    return list(internalLinks)
```

Первая функция — **getInternalLinks**. Она принимает объект **BeautifulSoup** и URL страницы. Этот URL используется только для идентификации **netloc** (network location — сетевого местоположения) и схемы **scheme** (обычно **http** или **https**) внутреннего сайта, поэтому важно отметить, что здесь можно использовать любой внутренний URL целевого сайта — нет необходимости передавать точный URL, полученный из объекта **BeautifulSoup**.

Эта функция создает множество с именем **InternalLinks** и использует его для отслеживания внутренних ссылок, найденных на странице. Она проверяет все теги **<a>** на наличие атрибута **href**, который либо не содержит **netloc** (это относительный URL, например **/careers/**), либо имеет **netloc**, соответствующий переданному URL:

```
# Получить список всех внешних ссылок, найденных на странице
def getExternalLinks(bs, url):
    internal_netloc = urlparse(url).netloc
    externalLinks = set()
    for link in bs.find_all('a'):
        if not link.attrs.get('href'):
            continue
        parsed = urlparse(link.attrs['href'])
        if parsed.netloc != '' and parsed.netloc != internal_netloc:
            externalLinks.add(link.attrs['href'])
    return list(externalLinks)
```

Функция **getExternalLinks** работает так же, как **getInternalLinks**. Она проверяет все теги **<a>** с атрибутом **href** и выбирает те, для которых значение **netloc** *не* совпадает с указанным URL:

```
def getRandomExternalLink(startingPage):
    bs = BeautifulSoup(urlopen(startingPage), 'html.parser')
    externalLinks = getExternalLinks(bs, startingPage)
    if not len(externalLinks):
```

```
        print('No external links, looking around the site for one')
        internalLinks = getInternalLinks(bs, startingPage)
        return getRandomExternalLink(random.choice(internalLinks))
    else:
        return random.choice(externalLinks)
```

Функция `getRandomExternalLink` вызывает `getExternalLinks`, чтобы получить список всех внешних ссылок на странице. Если список не пуст, то она выбирает из него случайную ссылку и возвращает:

```
def followExternalOnly(startingSite):
    externalLink = getRandomExternalLink(startingSite)
    print(f'Random external link is: {externalLink}')
    followExternalOnly(externalLink)
```

Функция `followExternalOnly` вызывает `getRandomExternalLink`, а затем выполняет рекурсивный обход найденных ссылок. Ее можно вызвать так:

```
followExternalOnly('https://www.oreilly.com/')
```

Эта программа загружает начальную страницу с сайта **http://oreilly.com** и случайным образом переходит от одной внешней ссылки к другой. Вот пример ее вывода:

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

Не всегда внешние ссылки могут быть обнаружены на первой странице сайта. В этом случае для поиска внешних ссылок используется метод, аналогичный тому, который применялся в предыдущем примере краулинга сайта в глубину до обнаружения внешней ссылки.

Принцип работы этой программы показан на рис. 6.1 в виде блок-схемы.



Рис. 6.1. Блок-схема нашего сценария краулинга сайтов



Не используйте учебные примеры программ в реальных приложениях

Еще раз напоминаю: в целях экономии места и удобства чтения примеры программ, приведенные в этой книге, не всегда содержат необходимые проверки и обработку 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 4. 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 возможных ошибок.

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 ваша цель — просканировать сайт, найти все внешние ссылки и сохранить их, то можно добавить следующую функцию:

```
# Составляет список из всех внешних URL, найденных на сайте
allExtLinks = []
allIntLinks = []

def getAllExternalLinks(url):
    bs = BeautifulSoup(urlopen(url), 'html.parser')
    internalLinks = getInternalLinks(bs, url)
    externalLinks = getExternalLinks(bs, url)
    for link in externalLinks:
        if link not in allExtLinks:
            allExtLinks.append(link)
            print(link)

    for link in internalLinks:
        if link not in allIntLinks:
            allIntLinks.append(link)
            getAllExternalLinks(link)

allIntLinks.append('http://oreilly.com')
getAllExternalLinks('http://www.oreilly.com/')
```

Этот код можно представить как два совместно работающих цикла: один собирает внутренние ссылки, а второй — внешние. Блок-схема программы показана на рис. 6.2.

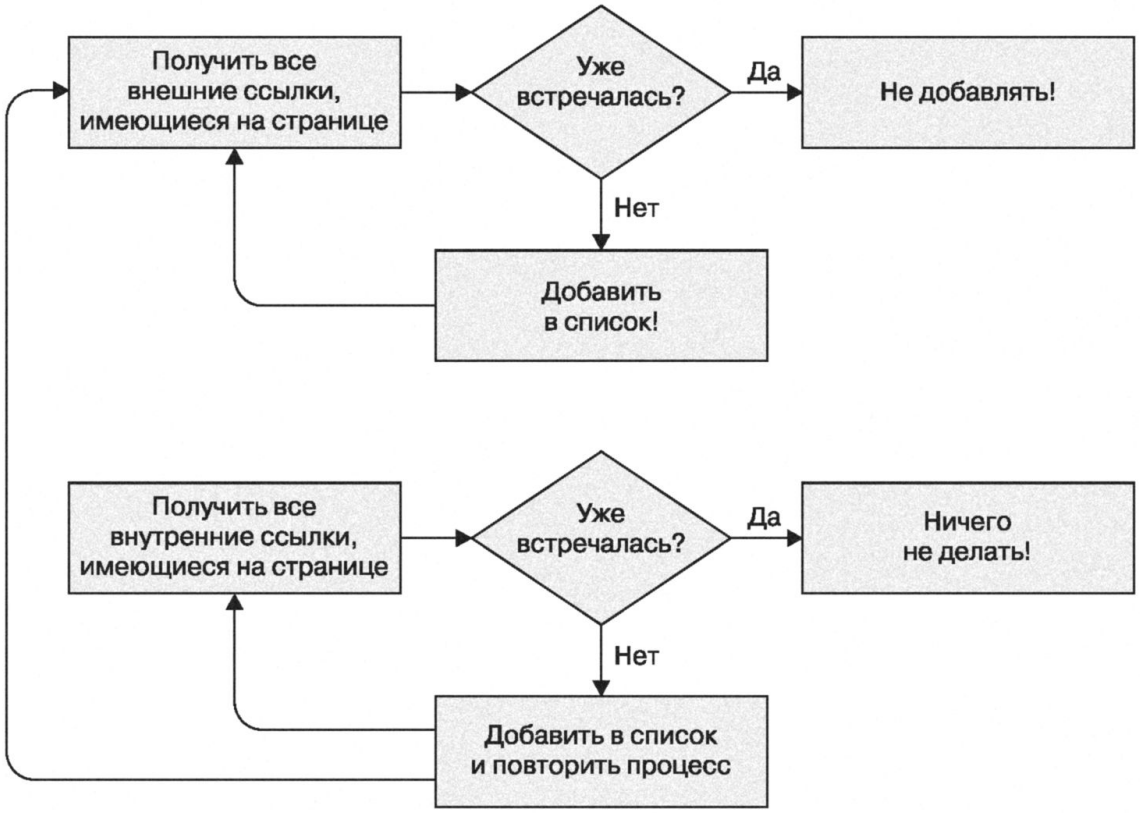


Рис. 6.2. Блок-схема веб-краулера сайтов, который собирает все внешние ссылки

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 массу времени и избавит от многих разочарований.

ГЛАВА 7

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, размещенные на различных сайтах, у каждого из которых свои шаблоны и макеты. На одном сайте тег `h1` может содержать заголовок статьи, однако на другом — заголовок самого сайта, а заголовок статьи будет заключен в тег `<span id="title">`.

Возможно, вам понадобится оперативный контроль над 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 написания многочисленных 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. Кроме того, 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 текущей главе основное внимание будет уделено 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 считывают 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 сначала заглянуть в магазин одежды и решить, что для каждого интересующего вас изделия необходимо заполнить следующие поля:

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

Просмотрев 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! Так, в список будет добавлено еще одно поле:

- артикул.

Для начала сбор данных об одежде вполне подойдет, однако необходимо убедиться, что этот веб-краулер можно будет использовать и для других типов товаров. Начав просматривать разделы продукции 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 трудно читаемому 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 позволила бы однозначно идентифицировать продукт:

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

Важно отметить, что вся эта информация не относится к конкретному магазину. Например, обзоры товаров, рейтинги, цены и даже описания одного и того же продукта могут отличаться в разных магазинах, и их стоит хранить отдельно.

Другие данные (цветовые варианты товара и материал, из которого он сделан) относятся к конкретному продукту, но могут быть разреженными — существовать не для всех единиц. Важно сделать шаг назад и составить чек-лист 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 *могут* когда-нибудь 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 на разных сайтах может быть разным.)

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 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 эти вопросы, вы можете обнаружить, что 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 9.

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, 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.

Предположим, что некоторые статьи содержат «дату изменения», или «связанные публикации», или «количество репостов в социальных сетях». Вам нужна эта информация? Она имеет отношение к вашему проекту? Как эффективно и гибко хранить количество репостов в социальных сетях, если только некоторые новостные сайты используют все формы соцсетей, 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. Однако если оставить создание модели данных на потом, то доступность и формат данных часто будут определяться первым попавшимся сайтом.

Тем не менее 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: как известными, так и неизвестными. Притом очень 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 сайтов с крайне плохим веб-дизайном), однако заставить бот делать то же самое гораздо труднее.

К счастью, в большинстве случаев при веб-краулинге мы не намерены собирать данные с сайтов, которых никогда прежде не видели. Обычно речь идет максимум о нескольких десятках сайтов, предварительно отобранных пользователем. 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 сканер 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`:

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

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

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

def scrapeCNN(url):
    bs = BeautifulSoup(urlopen(url))
    title = bs.find('h1').text
    body = bs.find('div', {'class': 'article__content'}).text
    print('body: ')
    print(body)
    return Content(url, title, body)

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

url = 'https://www.brookings.edu/research/robotic-rulemaking/'
content = scrapeBrookings(url)
content.print()

url = 'https://www.cnn.com/2023/04/03/investing/\
dogecoin-elon-musk-twitter/index.html'
content = scrapeCNN(url)
content.print()
```

Добавляя функции веб-скрапинга для других новостных сайтов, вы, вероятно, заметите, как начнет формироваться шаблон. В сущности, в процессе парсинга любого сайта делается одно и то же:

- находится элемент заголовка и оттуда извлекается текст заголовка;
- находится основной контент статьи;
- при необходимости находятся другие элементы контента;
- возвращается 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 облегчить задачу, 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 объекте словаря:

```
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 представлен 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).

Разрабатывая веб-скраперы, можно заметить, что часто приходится решать одни и те же задачи снова и снова. Например: получить содержимое страницы, попутно проверяя наличие ошибок, получить содержимое тега и корректно завершить работу, если тег пустой. Добавим их в класс `Crawler`:

```
class Crawler:
    def getPage(url):
        try:
            html = urlopen(url)
        except Exception:
            return None
        return BeautifulSoup(html, 'html.parser')

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

Обратите внимание, что на данный момент класс `Crawler` не имеет никакого состояния. Это просто набор статических методов. А еще ему дано немного неудачное имя — он вообще не сканирует! Но его можно сделать чуть более полезным, добавив метод `getContent`, который принимает экземпляр класса `Website` и URL и возвращает объект `Content`:

```
class Crawler:
    ...
    def getContent(website, path):
        """
        Извлекает контент указанной страницы
        """
        url = website.url+path
        bs = Crawler.getPage(url)
        if bs is not None:
            title = Crawler.safeGet(bs, website.titleTag)
            body = Crawler.safeGet(bs, website.bodyTag)
            return Content(url, title, body)
        return Content(url, '', '')
```

Далее показано, как можно использовать классы `Content`, `Website` и `Crawler` для анализа четырех разных веб-сайтов:

```
siteData = [
    ['O\'Reilly', 'http://oreilly.com', 'h1', 'div.title-description'],
    ['Reuters', 'http://reuters.com', 'h1', 'div.ArticleBodyWrapper'],
    ['Brookings', 'http://www.brookings.edu', 'h1', 'div.post-body'],
    ['CNN', 'https://www.cnn.com', 'h1', 'div.article__content']
]
websites = []
for name, url, title, body in siteData:
    websites.append(Website(name, url, title, body))

Crawler.getContent(
    websites[0],
    '/library/view/web-scraping-with/9781491910283'
    ).print()
Crawler.getContent(
    websites[1],
    '/article/us-usa-epa-pruitt-idUSKBN19W2D0'
    ).print()
Crawler.getContent(
    websites[2],
    '/blog/techtank/2016/03/01/idea-to-retire-old-methods-of-policy-education/'
    ).print()
Crawler.getContent(
    websites[3],
    '/2023/04/03/investing/dogecoin-elon-musk-twitter/index.html'
    ).print()
```

На первый взгляд этот новый метод 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, парсинга HTML. 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, сканировать 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 главе 6 были показаны различные способы краулинга сайтов и автоматического поиска новых страниц.

В этом разделе будет показано, как встроить эти методы в хорошо структурированный и расширяемый веб-краулер, который автоматически собирал бы ссылки и находил нужные данные. Здесь я представлю только три основные структуры веб-краулера, хотя считаю их пригодными к применению в большинстве ситуаций, с которыми вы, скорее всего, столкнетесь при краулинге 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.

Краулинг с помощью поиска

Один из самых простых способов краулинга сайта — тот же, что используют обычные пользователи: с помощью панели поиска. Поиск на сайте по ключевому слову или теме и последующий сбор данных по списку результатов могут показаться задачей, которая сильно зависит от конкретного сайта, однако несколько ключевых моментов делают ее на удивление простой.

- 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 `<span class="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, вы успешно сведете задачу к примеру, рассмотренному в предыдущем разделе, — извлечению данных со страницы сайта, имеющей заданный формат.

Рассмотрим реализацию этого алгоритма. 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))
```

В класс **Website** тоже было добавлено 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** хранит логическое значение, указывающее тип ссылок в результатах поиска: абсолютные или относительные.

```
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
```

Также был немного расширен класс **Crawler** — теперь в нем содержится объект **Website**, а также словарь с URL, ссылающимися на объекты **Content**, чтобы отличать адреса, просмотренные прежде. Обратите внимание, что методы **getPage** и **safeGet** не изменились и потому здесь не показаны:

```
class Crawler:
    def __init__(self, website):
        self.site = website
        self.found = {}

    def getContent(self, website, path):
        """
```

```
        Извлекает контент указанной страницы
        """
        bs = Crawler.getPage(url)
        if bs is not None:
            title = Crawler.safeGet(bs, self.site.titleTag)
            body = Crawler.safeGet(bs, self.site.bodyTag)
            return Content(topic, url, title, body)
        return Content(topic, url, '', '')

    def search(self, topic):
        """
        Поиск на заданном сайте по заданной теме
        и сохранение всех найденных страниц
        """
        bs = Crawler.getPage(self.site.searchUrl + topic)
        searchResults = bs.select(self.site.resultListing)
        for result in searchResults:
            url = result.select(self.site.resultUrl)[0].attrs['href']
            # Проверить, является URL относительным или абсолютным
            url = url if self.site.absoluteUrl else self.site.url + url
            if url not in self.found:
                self.found[url] = self.getContent(topic, url)
            self.found[url].print()
```

Вызвать этот класс `Crawler` можно так:

```
siteData = [
    ['Reuters', 'http://reuters.com',
     'http://www.reuters.com/search/news?blob=',
     'div.search-result-content', 'h3.search-result-title a',
      False, 'h1', 'div.ArticleBodyWrapper'],
    ['Brookings', 'http://www.brookings.edu',
     'https://www.brookings.edu/search/?s=',
     'div.article-info', 'h4.title a', True, 'h1', 'div.core-block']
]
sites = []
for name, url, search, rListing, rUrl, absUrl, tt, bt in siteData:
    sites.append(Website(name, url, search, rListing, rUrl, absUrl, tt, bt))

crawlers = [Crawler(site) for site in sites]
topics = ['python', 'data%20science']

for topic in topics:
    for crawler in crawlers:
        crawler.search(topic)
```

Как и раньше, для каждого веб-сайта создается массив с информацией о том, как выглядят теги, а также URL и имя сайта для трассировки. Затем эти данные загружаются в список объектов `Website` и преобразуются в объекты `Crawler`.

Затем сценарий перебирает список краулеров `crawlers` и сканирует каждый конкретный сайт по каждой теме. Если поиск завершен успешно, то результаты выводятся в консоль:

```
New article found for topic: python
URL: http://reuters.com/article/idUSKCN11S04G
TITLE: Python in India demonstrates huge appetite
BODY:
By 1 Min ReadA 20 feet rock python was caught on camera ...
```

Следует отметить, что сценарий перебирает все темы, проходя по всем сайтам во внутреннем цикле. Почему бы не поступить наоборот: сначала перебрать 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 эти запросы на максимально возможное время. Обратите внимание: структура наших циклов обеспечивает простой способ сделать это.

Краулинг сайтов по ссылкам

В главе 6 были рассмотрены несколько способов идентификации внешних и внутренних ссылок на веб-страницах с их последующим использованием для краулинга сайта. В данном подразделе мы объединим эти базовые методы и создадим более гибкий краулер сайтов, способный переходить по любой ссылке, соответствующей заданному шаблону URL.

Веб-краулер такого типа хорошо подходит 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 или совсем ничего общего.

В отличие от примера краулинга 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 повторно используем 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(f'URL: {self.url}')
        print(f'TITLE: {self.title}')
        print(f'BODY:\n{self.body}')
```

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

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

```
class Crawler:
    def __init__(self, site):
        self.site = site
        self.visited = {}

    def getPage(url):
        try:
            html = urlopen(url)
        except Exception as e:
            print(e)
            return None
        return BeautifulSoup(html, 'html.parser')

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

        def getContent(self, url):
        """
        Извлекает контент указанной страницы
        """
        bs = Crawler.getPage(url)
        if bs is not None:
            title = Crawler.safeGet(bs, self.site.titleTag)
            body = Crawler.safeGet(bs, self.site.bodyTag)
            return Content(url, title, body)
        return Content(url, '', '')

    def crawl(self):
        """
        Получить ссылки с начальной страницы сайта
        """
        bs = Crawler.getPage(self.site.url)
        targetPages = bs.find_all('a', href=re.compile(self.site.targetPattern))
        for targetPage in targetPages:
```

```
            url = targetPage.attrs['href']
            url = url if self.site.absoluteUrl else f'{self.site.url}{targetPage}'
            if url not in self.visited:
                self.visited[url] = self.getContent(url)
                self.visited[url].print()

brookings = Website(
    'Brookings', 'https://brookings.edu', '\/(research|blog)\/',
    True, 'h1', 'div.post-body')
crawler = Crawler(brookings)
crawler.crawl()
```

Как и в предыдущем примере, объект `Website` 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 сканирования списка сайтов.

Независимо от того, хотите ли вы, чтобы веб-краулер не зависел от конкретных сайтов, или же хотите 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, что веб-краулер станет получать ссылки с начальной страницы, но не продолжит краулинг после обхода всех этих страниц. Возможно, вы захотите написать веб-краулер, работающий по одной из схем, описанных в этой главе, 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 именно те из них, что соответствуют заданному шаблону.

Краулинг страниц нескольких типов

В отличие от краулинга заданного множества страниц, краулинг 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 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 этих тегов. Веб-краулер может искать элемент наподобие `<div id="related-products">`, 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 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 шаблона 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 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
```

Класс `Product` расширяет базовый класс `Website`, добавляя атрибуты `productNumber` и `price`, относящиеся лишь к товарам, а класс `Article` добавляет атрибуты `body` и `date`, которые к товарам не применимы.

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. Ответы на эти вопросы должны стать первым шагом в любом крупном (а иногда и небольшом) веб-проекте.

При сборе похожих данных в разных доменах или из разных источников ваша цель почти всегда должна состоять в том, чтобы попытаться упорядочить 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 следует создавать веб-скраперы, исходя из вероятного появления новых источников данных в дальнейшем, с целью минимизировать затраты на программирование, необходимые для добавления этих новых источников. Даже если на первый взгляд сайт не соотносится с вашей моделью, возможно, есть более изящные 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 описывать эти атрибуты?

Архитектура ПО — обширная и важная тема, на освоение которой может уйти целая карьера. 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 8

Scrapy

В главе 7 были рассмотрены 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.

С момента своего появления в 2008 году библиотека Scrapy быстро превратилась в один из наиболее популярных инструментов скрапинга на Python. В настоящее время ее поддерживает Zyte (ранее Scrapinghub).

Одна из сложностей создания веб-краулеров состоит в том, что часто приходится решать одни и те же задачи: отыскивать все ссылки на странице, идентифицировать их как внутренние и внешние, 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 со своего 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 команды

```
$ pip install Scrapy
```

Обратите внимание, что 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 неустранимыми ошибками.

Если вы решили установить Scrapy с помощью pip, то настоятельно рекомендую использовать виртуальную среду (подробнее о виртуальных средах см. во врезке «Хранение библиотек непосредственно в виртуальных окружениях» на с. 68).

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 фреймворка Scrapy необходимо выполнить небольшую настройку для каждого *«паука»* (spider) — проекта Scrapy, который, как и обычный паук, занимается обходом (краулингом) сети. В этой главе я буду называть «пауком» именно проект Scrapy, а краулером — любую программу, которая сканирует сеть, 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__
```

Эти файлы на Python запускаются с помощью кода-заглушки, что позволяет быстро создать новый проект «паука». В следующих разделах данной главы будет подробно описана работа с проектом `wikiSpider`.

Пишем простой веб-скрапер

Чтобы создать веб-краулер, нужно добавить в дочерний каталог **spiders** новый файл `wikiSpider/wikiSpider/spiders/article.py`. Именно в этот каталог будут отправляться все «пауки», расширяющие `scrapy.Spider`. Во вновь созданный файл `article.py` добавьте следующий код:

```
from scrapy import Spider, Request

class ArticleSpider(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 [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(f'URL is: {url}')
        print(f'Title is: {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 крупных сайтов с большим количеством различного контента можно создать отдельные элементы Scrapy для каждого типа содержимого (публикации в блогах, пресс-релизы, статьи и т. п.). Все они будут иметь свои поля и станут частью одного проекта Scrapy. Имя каждого «паука» должно быть уникальным в рамках проекта.

Следует обратить внимание еще на два важных момента: функции `start_requests` и `parse`.

- `start_requests` — это предопределенная Scrapy точка входа в программу, используемая для генерации объектов `Request`, которые в Scrapy применяются для краулинга сайта.
- `parse` — это функция обратного вызова, определяемая пользователем, которая передается в объект `Request` с помощью `callback = self.parse`. Позже будут рассмотрены более мощные приемы, которые возможны благодаря функции `parse`, но пока она просто выводит заголовок страницы.

Для запуска «паука» `article` нужно перейти во внешний каталог wikiSpider и выполнить следующую команду:

```
$ scrapy runspider wikiSpider/spiders/article.py
```

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

```
2023-02-11 21:43:13 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/robots.txt> (referer: None)
2023-02-11 21:43:14 [scrapy.downloadermiddlewares.redirect] DEBUG: Redirecting (3
01) to <GET https://en.wikipedia.org/wiki/Python_%28programming_language%29> from
<GET http://en.wikipedia.org/wiki/Python_%28programming_language%29>
2023-02-11 21:43:14 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/wiki/Functional_programming> (referer: None)
2023-02-11 21:43:14 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/wiki/Monty_Python> (referer: None)
URL is: https://en.wikipedia.org/wiki/Functional_programming
Title is: Functional programming
2023-02-11 21:43:14 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/wiki/Python_%28programming_language%29> (referer: None)
URL is: https://en.wikipedia.org/wiki/Monty_Python
Title is: Monty Python
URL is: https://en.wikipedia.org/wiki/Python_%28programming_language%29
Title is: Python (programming language)
```

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 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 блокнота Jupyter, что затрудняет демонстрацию последовательной разработки кода. Для представления примеров кода в тексте главы я сохранила 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.linkextractors import LinkExtractor
from scrapy.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('span.mw-page-title-main::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(f'URL is: {url}')
        print(f'Title is: {title} ')
        print(f'Text is: {text}')
        print(f'Last updated: {lastUpdated}')
```

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

Предоставляется также 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 применяется 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.

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

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

```
$ scrapy runspider wikiSpider/spiders/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, нажав Ctrl+C или закрыв окно терминала. Прошу, пожалейте сервер «Википедии», не запускайте данную программу надолго.

После запуска этот «паук» начинает обход сайта **wikipedia.org** и переходит по всем ссылкам в домене **wikipedia.org**, выводя заголовки страниц и игнорируя все внешние ссылки (ведущие на другие сайты):

```
2023-02-11 22:13:34 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite
 request to 'drupal.org': <GET https://drupal.org/node/769>
2023-02-11 22:13:34 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite
 request to 'groups.drupal.org': <GET https://groups.drupal.org/node/5434>
2023-02-11 22:13:34 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite
 request to 'www.techrepublic.com': <GET https://www.techrepublic.com/article/
open-source-shouldnt-mean-anti-commercial-says-drupal-creator-dries-buytaert/>
2023-02-11 22:13:34 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite
 request to 'www.acquia.com': <GET https://www.acquia.com/board-member/dries-b
uytaert>
```

Пока данный веб-краулер работает неплохо, однако у него есть несколько ограничений. Вместо того чтобы посещать только страницы статей «Википедии», он 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 `Rule`, определяющих правила, по которым фильтруются все найденные ссылки. При наличии нескольких правил каждая ссылка проверяется на соответствие им по порядку. Первое подходящее правило используется для определения способа обработки ссылки. Если ссылка не соответствует ни одному из правил, то она игнорируется.

Правило может быть описано следующими четырьмя аргументами:

`link_extractor`
: Единственный обязательный аргумент — объект `LinkExtractor`;

`callback`
: Функция обратного вызова, которая должна использоваться для парсинга контента страницы;

`cb_kwargs`
: Словарь аргументов, передаваемых в функцию обратного вызова. Имеет вид `{имя_аргумента1: значение_аргумента1, имя_аргумента2: значение_аргумента2}` и может быть удобным инструментом для многократного использования одних и тех же функций парсинга в незначительно различающихся задачах;

`follow`
: Указывает, должен ли веб-краулер также обрабатывать страницы, найденные по ссылкам на данной странице. Если функция обратного вызова не указана, то по умолчанию используется значение `True` (в конце концов, при отсутствии действий на этой странице логично предположить, что вы, по крайней мере, захотите применить ее для продолжения краулинга 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 селекторов CSS и XPath, тегов (можно искать ссылки не только в тегах `<a>`!), доменов и т. д.

Для класса `LinkExtractor` можно создавать расширения с дополнительными аргументами. 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 сканирование всех ссылок, соответствующих заданному регулярному выражению.

`deny`

Запретить сканирование всех ссылок, соответствующих заданному регулярному выражению.

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

```
from scrapy.linkextractors import LinkExtractor
from scrapy.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('span.mw-page-title-main::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(f'URL is: {url}')
            print(f'Title is: {title}')
            print(f'Text is: {text}')
        else:
            print(f'This is not an article: {title}')
```

Напомним, 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 шаблону 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 парсинга страницы.

Создание объектов Item

Уже было рассмотрено множество способов поиска, парсинга и краулинга сайтов с помощью Scrapy, однако этот фреймворк 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`. Определите этот новый элемент с именем `Article` в файле `items.py`.

Когда вы впервые открываете 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.html

import scrapy

class WikispiderItem(scrapy.Item):
    # определите здесь поля своего объекта, как показано ниже:
    # name = scrapy.Field()
    pass
```

Замените этот предоставляемый по умолчанию шаблон `Item` определением нового класса `Article`, который является расширением `scrapy.Item`:

```
import scrapy

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

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

Если вы собираете 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 вы захотите добавить в них дополнительные функции парсинга, то можно разместить каждый такой объект в отдельном файле. Однако, поскольку в нашем случае объекты `Item` невелики, я сохраню 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.linkextractors import LinkExtractor
from scrapy.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 wikiSpider/spiders/articleItems.py
```

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

```
2023-02-11 22:52:26 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/wiki/Benevolent_dictator_for_life#bodyContent> (referer: https://en.wi
kipedia.org/wiki/Benevolent_dictator_for_life)
2023-02-11 22:52:26 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/wiki/OCaml> (referer: https://en.wikipedia.org/wiki/Benevolent_dictato
r_for_life)
2023-02-11 22:52:26 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://en.wik
ipedia.org/wiki/Xavier_Leroy> (referer: https://en.wikipedia.org/wiki/Benevolent_
dictator_for_life)
2023-02-11 22:52:26 [scrapy.core.scraper] DEBUG: Scraped from <200 https://en.wik
ipedia.org/wiki/Benevolent_dictator_for_life>
{'lastUpdated': ' 7 February 2023, at 01:14',
'text': ['Title given to a small number of open-source software development'
         'leaders',
         ...
```

Использование объектов `Item` в Scrapy не только обеспечивает хорошую упорядоченность кода и представление объектов в удобочитаемом 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 информацию в файлах разных форматов, таких как 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 заданный файл 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 для хранения 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 вы собираете текстовые данные, представляет собой множество дочерних элементов.

Scrapy прекрасно справляется с такими сложными значениями. Например, при сохранении данных в формате CSV списки преобразуются в строки, а запятые экранируются, чтобы список текстовых фрагментов в формате CSV отображался в одной ячейке.

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> 7 February 2023, at 01:14.</lastUpdated>
</item>
....
```

В формате JSON списки так и сохраняются в виде списков.

И конечно же, вы можете обрабатывать объекты `Item` самостоятельно, записывая их в файл или базу данных любым удобным способом, просто добавив соответствующий код в функцию парсинга 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 (даже 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 19.

Таким образом, использование динамического конвейера в Scrapy позволяет еще больше повысить скорость работы веб-скрапера, так как вся обработка данных будет выполняться, пока веб-скрапер ожидает ответа на запросы, вместо того чтобы дожидаться окончания обработки данных и только потом выполнять очередной запрос. Данный 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 закомментированные строки кода:

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

Раскомментируем последние три строки 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» так, чтобы он собирал данные, а конвейер решал нелегкую задачу по их обработке. Было бы заманчиво создать в исходном «пауке» метод **parse_items**, который возвращал бы 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.linkextractors import LinkExtractor
from scrapy.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_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()
        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 -*-

# Определите здесь свои динамические конвейеры
#
# Не забудьте добавить свой конвейер в ITEM_PIPELINES
# См.: 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` (массив разнородных 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». Парсируемый объект `Article`, который возвращается в данном случае, будет сохранен или выведен Scrapy, если, например, вы решите 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. Парсинг элементов конкретных типов лучше реализовать в «пауке» и только потом передавать данные в конвейер. Но если этот парсинг 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 в Scrapy

Отладочная информация, генерируемая Scrapy, бывает полезна, однако, как вы могли заметить, часто слишком 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`.

При выборе уровня `ERROR` будут отображаться только события типов `CRITICAL` и `ERROR`; при выборе уровня `INFO` — все события и т. д.

Кроме управления уровнем ведения журнала через файл `settings.py` также можно указать, куда выводить записи. Чтобы они выводились не на терминал, а в отдельный файл журнала, нужно указать имя этого файла при запуске Scrapy из командной строки:

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

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

Дополнительные ресурсы

Scrapy — мощный инструмент, способный решить многие задачи 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 возможности Scrapy рассмотрены лишь поверхностно; я призываю вас заглянуть в документацию по Scrapy (**https://doc.scrapy.org/en/latest/news.html**), а также в официальное учебное руководство Scrapy (**https://docs.scrapy.org/en/latest/intro/tutorial.html**), где вы найдете подробное описание этого фреймворка.

Scrapy — чрезвычайно обширная библиотека со множеством функций. Они хорошо сочетаются между собой, но во многом перекрывают друг друга, благодаря чему разработчик может легко создать собственный стиль. Если у вас есть какие-то идеи, которые вы хотели бы реализовать с помощью Scrapy, но не нашли подходящую информацию в данной главе, скорее всего, существует способ (или даже несколько) сделать так, как вы задумали!

ГЛАВА 9

Хранение данных

Простой вывод данных в терминал 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 будут рассмотрены три основных метода управления данными, которых 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 периодически отправлять уведомления или раз в день передавать сводные данные? Тогда отправьте себе письмо по электронной почте!

Возможность хранить большие объемы данных и взаимодействовать с ними 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

Существуют два основных способа сохранения медиафайлов: по ссылке и скачивая 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, где размещен сканируемый ресурс.

Но есть и недостатки.

- Встраивание URL в ваш сайт или приложение называется *горячими ссылками*; это легкий способ нажить 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, например, разместить картинку, расположенную по этому адресу, в публичном блоге или 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. Благодаря скачиванию файлов действия веб-скрапера будут похожими на действия человека, который просматривает сайт, — это может стать преимуществом.

Принимая решение, сохранять 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 бóльшую 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, urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com')
bs = BeautifulSoup(html, 'html.parser')
imageLocation = bs.find('img', {'alt': 'python-logo'})['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 сценарий.

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, urlopen
from urllib.parse import urlparse
from bs4 import BeautifulSoup

downloadDir = 'downloaded'
baseUrl = 'http://pythonscraping.com/'
baseNetloc = urlparse(baseUrl).netloc

def getAbsoluteURL(source):
    if urlparse(baseUrl).netloc == '':
        return baseUrl + source
    return source

def getDownloadPath(fileUrl):
    parsed = urlparse(fileUrl)
    netloc = parsed.netloc.strip('/')
    path = parsed.path.strip('/')
    localfile = f'{downloadDir}/{netloc}/{path}'

    # Удалить имя файла из пути к нему, чтобы
    # получить структуру каталогов, где он хранится
    localpath = '/'.join(localfile.split('/')[:-1])
    if not os.path.exists(localpath):
        os.makedirs(localpath)
    return localfile

html = urlopen(baseUrl)
bs = BeautifulSoup(html, 'html.parser')
downloadList = bs.findAll(src=True)

for download in downloadList:
    fileUrl = getAbsoluteURL(download['src'])
    if fileUrl is not None:
        try:
        urlretrieve(fileUrl, getDownloadPath(fileUrl))
        print(fileUrl)
    except Exception as e:
        print(f'Could not retrieve {fileUrl} Error: {e}')
```

В этом сценарии с помощью лямбда-функции (см. главу 5) вы выбираете на главной странице сайта все теги, имеющие атрибут `src`, а затем очищаете и нормализуете 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`.



Будьте осторожны!

Помните, что бывает с теми, кто скачивает неизвестные файлы из Интернета? Представленный выше сценарий скачивает на жесткий диск вашего компьютера все подряд, включая случайные сценарии 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 сценарий 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 тщательной проверки имен файлов и 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.

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 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 парсируя и не изменяя, то информация из данного раздела вам

не понадобится. 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`, внеся 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. Список стран с ресторанами McDonald's в «Википедии» (https://en.wikipedia.org/wiki/List_of_countries_with_McDonald%27s_restaurants) представляет собой весьма сложную HTML-таблицу с выделением ячеек разными цветами, а также со ссылками, сортировкой и другим HTML-мусором, который необходимо вычистить, прежде чем записывать данные в формат CSV. Активно используя BeautifulSoup и функцию `get_text()`, мы можем сделать это менее чем в 20 строках кода:

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

html = urlopen('http://en.wikipedia.org/wiki/
       List_of_countries_with_McDonald%27s_restaurants')
bs = BeautifulSoup(html, 'html.parser')

# Основная таблица сейчас является первой на странице
table = bs.find('table',{'class':'wikitable'})
rows = table.findAll('tr')
csvFile = open('countries.csv', ,wt+')
```

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



Более простой способ получить отдельную таблицу

Данный сценарий отлично интегрируется в веб-скрапер, если нужно преобразовать много HTML-таблиц в CSV-файлы или объединить таблицы в один CSV-файл. Однако, если нужно сделать это только один раз, имеется способ получше: копирование и вставка. Выделив и скопировав все содержимое HTML-таблицы и вставив его в файл Excel или Google Docs, вы как раз получите CSV-файл, не запуская сценарий!

Результатом должен стать правильно отформатированный CSV-файл, сохраненный на локальном компьютере под именем `countries.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 YouTube[1], Twitter[2] и Facebook[3].

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.

---

[1] *Jackson J.* YouTube Scales MySQL with Go Code // PCWorld. — December 15, 2012. — URL: http://bit.ly/1LWVmc8.

[2] *Cole J., Arnaut D.* MySQL at Twitter // The Twitter Engineering Blog. — April 9, 2012. — URL: http://bit.ly/1KHDKns.

[3] *Callaghan M.* MySQL and Database Engineering // Facebook Engineering. — March 4, 2012. — URL: http://on.fb.me/1RFMqvw.

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 своей 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 отследите процесс 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. 9.1).

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

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, к счастью, удобный мастер установки (**http://dev.mysql.com/downloads/windows/installer/**) упрощает этот процесс. После загрузки он проведет вас по операциям, которые необходимо выполнить (рис. 9.2).

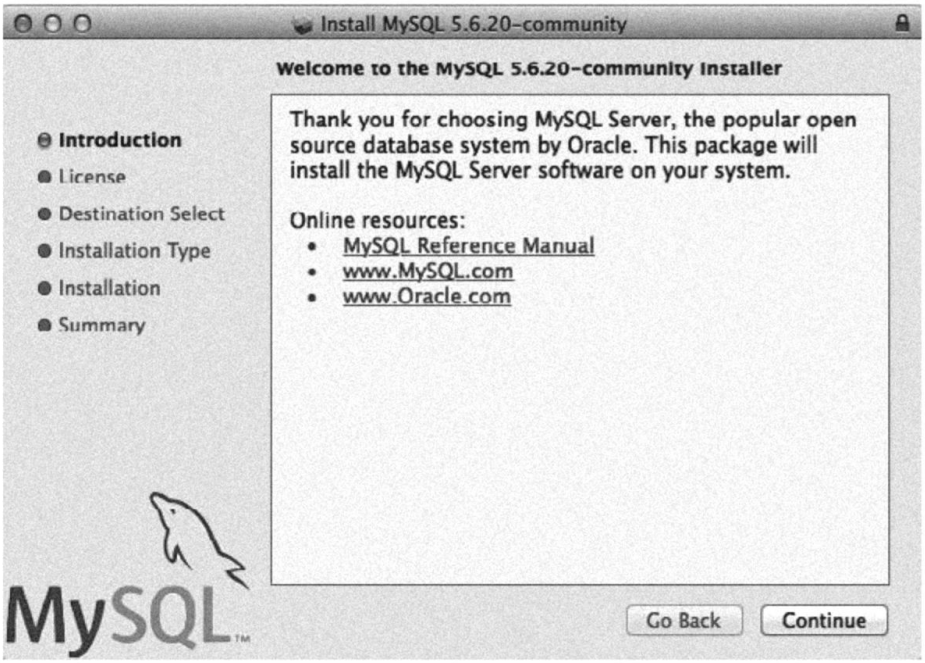


Рис. 9.1. Мастер установки MySQL в macOS

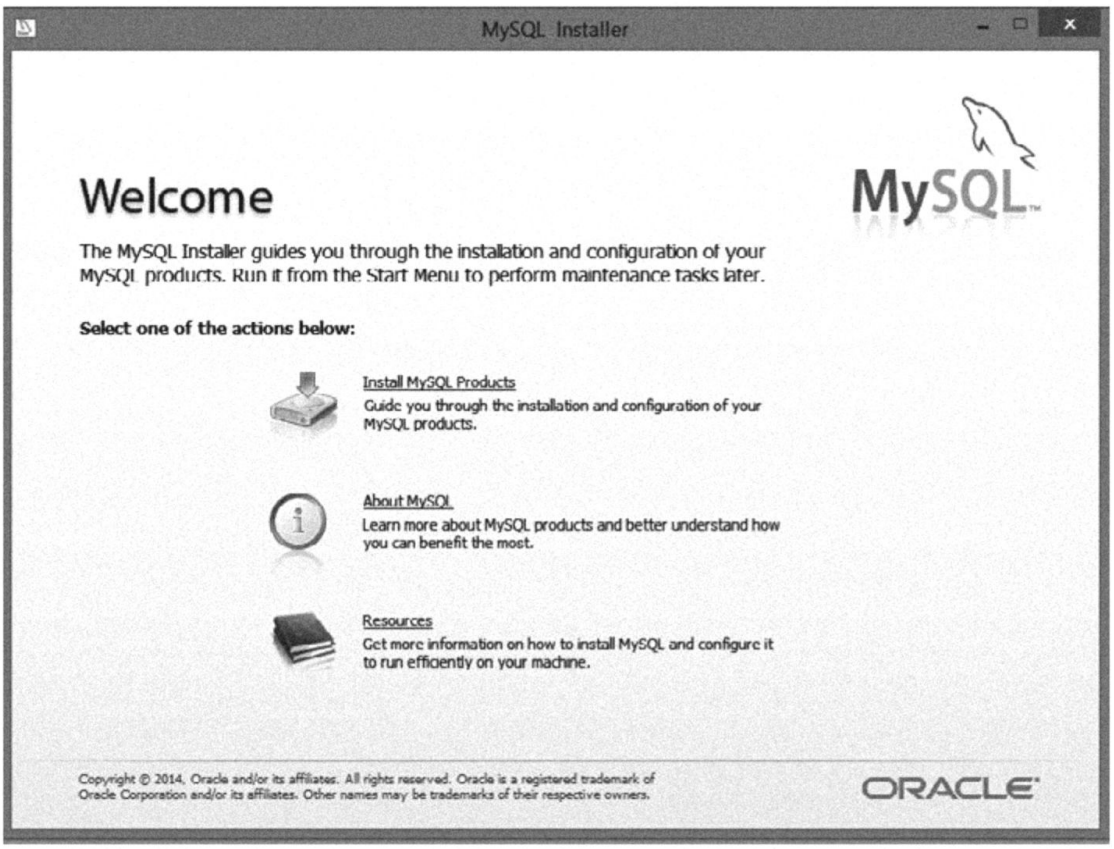


Рис. 9.2. Мастер установки MySQL в Windows

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.

После установки и запуска сервера MySQL у вас появится возможность взаимодействовать с ним из командной строки. В Windows также можно использовать инструмент MySQL Shell (**https://dev.mysql.com/downloads/shell/**). В macOS, я обычно устанавливаю инструменты командной строки с помощью диспетчера пакетов Homebrew (**https://brew.sh**):

```
$ brew install mysql
```

Установив инструменты командной строки, их можно использовать для подключения к серверу MySQL:

```
$ mysql -u root -p
```

Эта команда попросит вас ввести пароль root пользователя, которого вы создали в процессе установки.

Несколько основных команд

После запуска сервера MySQL появляется множество вариантов взаимодействия с базой данных. Есть много программных инструментов, играющих роль посредников, вследствие чего вам необязательно 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 операторе 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, не бывает таблицы MySQL без столбцов. Чтобы определить столбцы в MySQL, необходимо перечислить их в виде списка, разделенного запятыми, в скобках после оператора `CREATE TABLE <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 (`BIGINT(7)`, `VARCHAR`, `TIMESTAMP`);
- различные дополнительные атрибуты (`NOT NULL AUTO_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: хотя 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 значение для столбца `id` генерируется 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, вы *можете* 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 таблице появились данные, можно 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 вашей таблице много столбцов и вы хотите, 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 * FROM pages WHERE id = 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 предпринимать и при работе 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[1] (издательство O'Reilly) (http://shop.oreilly.com/product/0636920032274.do).

Интеграция с Python

К сожалению, Python не имеет встроенной поддержки MySQL. Однако 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 я работала над этой книгой, последней версией PyMySQL была 1.0.3, которая устанавливается с помощью pip следующим образом:

```
$ pip install PyMySQL
```

После установки библиотеки пакет PyMySQL станет доступным автоматически. При работающем локальном сервере MySQL должен успешно выполниться следующий сценарий (не забудьте указать пароль пользователя root для вашей базы данных):

```
import pymysql
conn = pymysql.connect(
    host='127.0.0.1',
```

[1] *Дюбуа П.* MySQL. Сборник рецептов. 2016.

```
    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. Объект-соединение отвечает не только за соединение с базой, но 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, какая база данных используется. Если у вас 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, вы можете активировать эту функцию (только помните, что тогда увеличится размер базы данных). Поскольку в «Википедии» вам 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 попробуете вставить в поле **title** или **content** несколько умляутов или китайских иероглифов и это удастся сделать без ошибок.

Теперь, когда база данных готова принять всю ту информацию, которую способна предоставить «Википедия», можно выполнить следующий код:

```
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 программа сообщает 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. Это 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, вы увидите, 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 в нем так много полезных функций, что описать их все не получится 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, 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 созданию столбцов идентификаторов. *Особенно* это касается веб-скрапинга и хранения чужих данных. Вы не можете знать заранее, какой элемент на самом деле окажется уникальным или неуникальным. Несколько раз действительность меня сильно удивляла.

Столбец идентификатора должен присутствовать во всех ваших таблицах, иметь автоматическое приращение и использоваться в качестве первичного ключа.

Во-вторых, используйте интеллектуальную индексацию. Словарь (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 вроде «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, а также 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    |                |
+-------------+---------+------+-----+---------+----------------+
```

Хотя 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 они произошли, можно рассмотреть вопрос о сохранении временных меток создания, изменения и удаления данных: `created`, `updated` и `deleted` соответственно.

«Шесть шагов» в MySQL

В главе 6 была рассмотрена задача «Шесть шагов по "Википедии"», цель которой состояла в поиске связи между любыми двумя статьями «Википедии» с помощью построения цепи ссылок (то есть нужно найти способ перейти от одной статьи к другой, переходя по ссылкам с одной страницы на другую). Чтобы решить данную задачу, необходимо создать боты, способные не только сканировать сайт (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 `A` и `B` — 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 DATABASE wikipedia;
USE wikipedia;

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 вы хотите создать эффективный веб-краулер для заполнения этих таблиц, то вам 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 Бэйкона» (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='password', db='mysql',
                       charset='utf8')
cur = conn.cursor()
cur.execute('USE wikipedia')

def insertPageIfNotExists(url):
    cur.execute('SELECT id FROM pages WHERE url = %s LIMIT 1', (url))
    page = cur.fetchone()
    if not page:
        cur.execute('INSERT INTO pages (url) VALUES (%s)', (url))
        conn.commit()
        return cur.lastrowid
    else:
        return page[0]

def loadPages():
    cur.execute('SELECT url FROM pages')
    return [row[0] for row in cur.fetchall()]

def insertLink(fromPageId, toPageId):
    cur.execute(
        'SELECT EXISTS(SELECT 1 FROM links WHERE fromPageId = %s\
        AND toPageId = %s)'
        , (int(fromPageId),
        int(toPageId))
    )
    if not cur.fetchone()[0]:
        cur.execute('INSERT INTO links (fromPageId, toPageId) VALUES (%s, %s)',
                    (int(fromPageId), int(toPageId)))
        conn.commit()

def pageHasLinks(pageId):
    cur.execute(
        'SELECT EXISTS(SELECT 1 FROM links WHERE fromPageId = %s)'
        , (int(pageId))
    )
    return cur.fetchone()[0]

def getLinks(pageUrl, recursionLevel, pages):
    if recursionLevel > 4:
        return
```

```
    pageId = insertPageIfNotExists(pageUrl)
    html = urlopen(f'http://en.wikipedia.org{pageUrl}')
    bs = BeautifulSoup(html, 'html.parser')
    links = bs.findAll('a', href=re.compile('^(/wiki/)((?!:).)*$'))
    links = [link.attrs['href'] for link in links]

    for link in links:
        linkId = insertPageIfNotExists(link)
        insertLink(pageId, linkId)
        if not pageHasLinks(linkId):
            print(f'Getting {link}')
            pages.append(link)
            getLinks(link, recursionLevel+1, pages)
        else:
            print(f'Already fetched {link}')

getLinks('/wiki/Kevin_Bacon', 0, loadPages())
cur.close()
conn.close()
```

Три представленные ниже функции используют PyMySQL для взаимодействия с базой данных.

`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.

`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.

`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 перезапустить 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 достаточно длительное время выполнения (или только один запуск) программы и нет необходимости собирать полный набор ссылок (а нужен лишь достаточно большой набор данных для экспериментов), то добавлять переменную `visited` необязательно.

Продолжение и окончательное решение задачи перехода от страницы Кевина Бейкона (https://en.wikipedia.org/wiki/Kevin_Bacon) к странице Эрика Айдла (https://en.wikipedia.org/wiki/Eric_Idle) вы найдете в подразделе «Шесть шагов по "Википедии": заключение» в главе 12, 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 используется 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 macOS.

В следующих примерах кода предполагается, что SMTP-клиент используется локально. (Чтобы модифицировать этот код для удаленного SMTP-клиента, замените `localhost` на адрес 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**, а также **msg** содержат адреса электронной почты, тело и заголовок, которые используются 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 создано и использовано, должно разрываться, чтобы избежать создания 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 сценарий раз в час проверяет сайт **https://isitchristmas.com** (главной функцией которого является вывод гигантского слова YES или NO, в зависимости от дня года). Если сценарий 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 ненамного более полезной, чем обычный настенный календарь, достаточно слегка ее изменить и можно будет делать множество чрезвычайно нужных вещей. Она способна 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.

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 сценариями 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, 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. Изучив этот раздел, вы получите инструменты для сбора и обработки практически любого типа данных, представленных в любом виде и в любом месте Интернета.

ГЛАВА 10

Чтение документов

Заманчиво представлять себе Интернет 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 по сути является Интернет: средством передачи файлов, независимо от их контента.

Хотя сама Глобальная сеть 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 будет рассмотрена 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 прочитать 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, имеющими формат, отличный от 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, каким образом интерпретируются их нули и единицы. В этой главе будут рассмотрены такие часто встречающиеся типы файлов, как 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 прочитать данную главу, чтобы привыкнуть обрабатывать и хранить различные типы файлов, а затем перейти к главе 16, в которой вы получите дополнительную информацию об обработке изображений!

Текст

Хранить файлы в виде обычного текста в Интернете кажется несколько необычным, однако есть много сайтов — простых или 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 в виде 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.

Почти для всех 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 парсинг HTML. В данном случае мы можем прочитать страницу напрямую. Можно превратить ее в объект BeautifulSoup, однако это было бы нерационально: здесь нет HTML-разметки, которую стоило бы

парсировать, поэтому библиотека будет бесполезной. Прочитав текстовый файл как строку, нам остается 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.

Кодирование текста 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 вы узнаете 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 прописные 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 нулем[1], 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 эти языки. 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,

[1] Этот бит «отступа» еще вернется, чтобы преследовать нас в стандартах ISO.

до кириллицы (щ, ъ, ы), китайских иероглифов (象形), математических и логических символов (Σ, ≥) и даже смайликов и прочих символов, таких как знак биологической опасности (☣) и «пацифик» (☮).

Получившаяся кодировка, как вы, возможно, уже поняли, названа 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? Применение 7 битов с возможно бесполезным ведущим нулем вначале 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 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 расходуются не на кодировку символов, а являются проверочными и служат для предотвращения повреждения кода. Из 32 (8 × 4) битов 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, ½) 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) и 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 ISO[1], поэтому нам по-прежнему 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 *résumé*, то 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())
```

[1] Согласно W3Techs (http://w3techs.com/technologies/history_overview/character_encoding), где для сбора статистики подобного рода используются веб-краулеры.

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. 10.1).

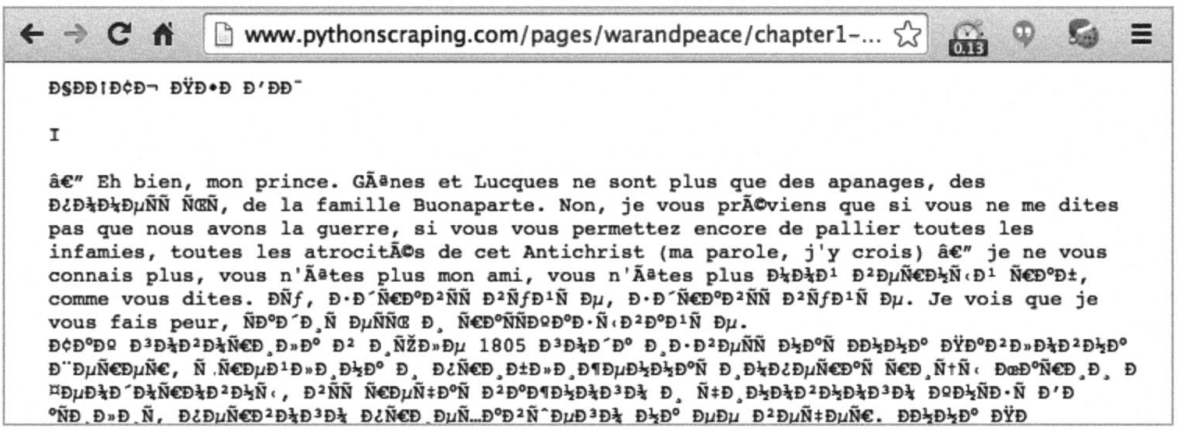


Рис. 10.1. Текст на русском и французском языках в кодировке в ISO-8859-1, которая во многих браузерах используется по умолчанию для текстовых документов

Этого 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.

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 выглядит так:

```
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()
content = bytes(content, 'UTF-8')
content = content.decode('UTF-8')
```

В Python 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 позволят исследовать документ и сделать правильное предположение (используя некую логику, позволяющую решить, что Ñ€Ð°ÑÑÐ¾Ð°Ð·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 на сайте ECMA International (**http://www.ecma-international.org/**), например, есть такой тег[1]:

```
<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, в 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 основное внимание будет уделено стандартному формату. Если вам попадется особый случай, с которым придется разбираться, то обратитесь к документации!

Чтение CSV-файлов

Библиотека `csv` 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.

[1] Организация ECMA одной из первых присоединилась к стандарту ISO, поэтому неудивительно, что ее сайт написан в кодировке ISO.

- Скачайте файл на компьютер вручную, и пусть Python работает с локальным файлом.
- Напишите сценарий 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 заполнять файлами место на жестком диске 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 сценарий получает CSV-файл из Интернета (в данном случае это список альбомов Monty Python, расположенный по адресу **http://pythonscraping.com/files/Monty-PythonAlbums.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: `The album "Name" was released in Year`. 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` занимают 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?), 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 Бакингемшира 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-скрапингу, 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 парсинга PDF.

К сожалению, поскольку PDF является относительно простым и открытым форматом документов, со временем появилось множество проектов библиотек для парсинга содержимого PDF-документов. Эти проекты обычно создаются, забрасываются, возрождаются и снова возобновляются. Самая популярная полнофункциональная и простая в использовании библиотека на данный момент — `pypdf` (**https://pypi.org/project/pypdf/**).

`pypdf` — бесплатная библиотека с открытым исходным кодом, позволяющая извлекать текст и изображения из PDF-файлов. С ее помощью можно также выполнять операции с PDF-файлами и генерировать их непосредственно из кода на Python.

Установить ее можно как обычно, используя pip:

```
$ pip install pypdf
```

Документация по библиотеке доступна по адресу **https://pypdf.readthedocs.io/en/latest/index.html**.

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 urlretrieve
from pypdf import PdfReader

urlretrieve(
    'http://pythonscraping.com/pages/warandpeace/chapter1.pdf',
    'chapter1.pdf'
)
reader = PdfReader('chapter1.pdf')

for page in reader.pages:
    print(page.extract_text())
```

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[1]
```

Обратите внимание, что PDF-файл, на который ссылается аргумент, должен быть реальным файлом. Поэтому сначала нужно загрузить файл на локальный компьютер и только потом передать его классу `PdfReader`. Если вы обрабатываете большое количество PDF-файлов и не хотите хранить их, то можете перезаписать предыдущий файл, передав то же имя файла в `urlretrieve` после извлечения текста.

Результат, полученный библиотекой `pypdf`, 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 содержимое должно быть статичным.

Файлы Word предназначены для создания контента, а не для обмена им. Тем не менее ряд сайтов постоянно использует эти файлы для представления важных документов, информации и даже диаграмм и мультимедиа — в общем, всего того, что можно и нужно представлять с помощью HTML.

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

[1] Глава 1
«Ну, князь, Генуя и Лукка — поместья фамилии Бонапарте. Нет, я вам вперед говорю, если вы мне не скажете, что у нас война, если вы еще позволите себе защищать все гадости, все ужасы этого Антихриста (право, я верю, что он Антихрист), — я вас больше не знаю...» — начало романа Л. Толстого «Война и мир». — *Примеч. пер.*

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/AWordDocument.docx').read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')
print(xml_content.decode('utf-8'))
```

Этот код читает удаленный документ Word как двоичный файловый объект (`BytesIO` аналогичен `StringIO`, который использовался ранее в этой главе), распаковывает его с помощью стандартной библиотеки `zipfile` (по соображениям экономии места все файлы `.docx` упаковываются в архив), а затем читает распакованный файл, имеющий формат XML.

На рис. 10.2 показан документ Word, размещенный по адресу http://python-scraping.com/pages/AWordDocument.docx.

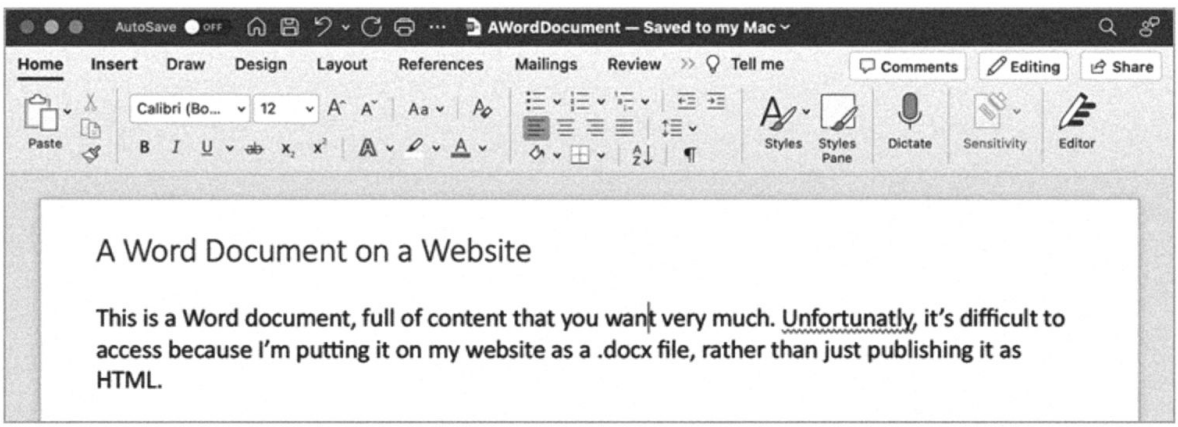


Рис. 10.2. Документ Word, полное содержимое которого вы, возможно, очень захотите прочитать, но к нему трудно получить доступ, поскольку я разместила его на своем сайте в виде файла .docx, вместо того чтобы опубликовать в формате HTML

Результат работы сценария Python, прочитавшего мой простой документ Word, выглядит так:

```
<w:document xmlns:wpc="http://schemas.microsoft.com/office/word/2010/
wordprocessingCanvas" xmlns:cx="http://schemas.microsoft.com/office/d
rawing/2014/chartex" xmlns:cx1="http://schemas.microsoft.com/office/d
rawing/2015/9/8/chartex" xmlns:cx2="http://schemas.microsoft.com/offi
ce/drawing/2015/10/21/chartex" xmlns:cx3="http://schemas.microsoft.co
m/office/drawing/2016/5/9/chartex" xmlns:cx4="http://schemas.microsof

...Прочие данные, описывающие схему...

<w:body><w:p w14:paraId="19A18025" w14:textId="54C8E458" w:rsidR="007
45992" w:rsidRDefault="00BF6C9C" w:rsidP="00BF6C9C"><w:pPr><w:pStyle
w:val="Heading1"/></w:pPr><w:r><w:t>A Word Document on a Website</w:t
></w:r></w:p><w:p w14:paraId="501E7A3A" w14:textId="77777777" w:rsidR
="00BF6C9C" w:rsidRDefault="00BF6C9C" w:rsidP="00BF6C9C"/><w:p w14:pa
raId="13929BE7" w14:textId="20FEDCDB" w:rsidR="00BF6C9C" w:rsidRPr="0
0BF6C9C" w:rsidRDefault="00BF6C9C" w:rsidP="00BF6C9C"><w:r><w:t xml:s
pace="preserve">This is a Word document, full of content that you wan
t very much. </w:t></w:r><w:proofErr w:type="spellStart"/><w:r><w:t>U
nfortuna</w:t></w:r><w:r w:rsidR="00BC14C7"><w:t>t</w:t></w:r><w:r><w
:t>ly</w:t></w:r><w:proofErr w:type="spellEnd"/><w:r><w:t xml:space="
preserve">, it's difficult to access because I'm putting it on my web
site as a .docx file, rather than just publishing it as HTML. </w:t><
/w:r></w:p><w:sectPr w:rsidR="00BF6C9C" w:rsidRPr="00BF6C9C"><w:pgSz
w:w="12240" w:h="15840"/><w:pgMar w:top="1440" w:right="1440" w:botto
m="1440" w:left="1440" w:header="720" w:footer="720" w:gutter="0"/><w
:cols w:space="720"/><w:docGrid w:linePitch="360"/></w:sectPr></w:bod
y></w:document>
```

Здесь явно слишком много метаданных, в которых утонул полезный 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/AWordDocument.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 парсера `html.parser`, который обычно передается в объект `BeautifulSoup`, мы используем парсер `xml`. Дело в том, что стандарт HTML не предусматривает двоеточия в именах тегов и `html.parser` не распознает теги, такие как `w:t`.

Результат не идеален, 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.
Unfortuna
t
ly
, 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 *unfortunatly* оказалось разбитым на несколько строк. В исходном XML оно заключено в тег `<w:proofErr w:type="spellStart"/>`. Так Word выделяет слова с орфографическими ошибками, которые нужно подчеркнуть красной волнистой линией.

Заголовку документа предшествует тег дескриптора стиля `<w:pstylew:val="Title">`. Несмотря на то что этот тег не позволяет однозначно идентифицировать заголовки (или другие стили 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.

ГЛАВА 11

Обработка «грязных» данных

До сих пор в этой книге я игнорировала 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 разборчивым в том, откуда берутся данные или как они 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 этой главе будут рассмотрены несколько инструментов и методов, которые помогут предотвратить подобные проблемы на начальном этапе, изменив способ написания кода и очищая данные после их попадания в базу.

ИЗВЛЕЧЬ, ПРЕОБРАЗОВАТЬ, ЗАГРУЗИТЬ

Извлечение, преобразование, загрузка (Extract, Transform, Load, ETL) — это часто используемая в программировании концепция, особенно в промышленности. Многие компании занимаются перемещением данных из одного места в другое, попутно преобразуя их. Данные извлекаются из источника, преобразуются с помощью некоторого бизнес-процесса и загружаются в другой источник.

Бóльшая часть этой книги посвящена букве «E» из аббревиатуры ETL, то есть извлечению данных из источников в Интернете. В главе 9 «Хранение данных» рассматривается буква «L» — загрузка данных в БД.

Эта глава, наряду со следующей «Чтение и запись текстов на естественных языках», посвящена букве «T» — преобразованию данных.

Условное разделение приложений для обработки данных на три этапа — извлечение, преобразование и загрузка — поможет вам организовать процессы и инструменты для создания чистых программных структур.

Теперь вы увидите, как скрапинг веб-страниц пересекается со своей близкой родственницей — наукой о данных. Название должности «специалист по данным» может наводить на мысли о передовых методах программирования и применении методов высшей математики, однако правда состоит в том, что бóльшую часть времени такие специалисты занимаются тяжелой и грязной работой. Кто-то должен очистить и нормализовать эти миллионы записей, прежде чем их можно будет использовать для построения модели машинного обучения, и этот человек — специалист по данным.

Очистка текста

Язык программирования Python очень хорошо подходит для обработки текста. Он позволяет с легкостью писать чистый, функциональный, модульный код, реализующий даже очень сложные проекты по обработке текста. Следующий код, например, извлекает текст из статьи в «Википедии» о Python, доступной по адресу **http://en.wikipedia.org/wiki/Python_(programming_language)**:

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

url = 'http://en.wikipedia.org/wiki/Python_(programming_language)'
html = urlopen(url)
bs = BeautifulSoup(html, 'html.parser')
content = bs.find('div', {'id':'mw-content-text'}).find_all('p')
content = [p.get_text() for p in content]
```

Эта статья начинается со строк:

```
Python is a high-level, general-purpose programming language. Its
design philosophy emphasizes code readability with the use of
significant indentation via the off-side rule.[33]
```

Выполним несколько операций с этим текстом:

- удалим ссылки вида «[123]»;
- удалим символы перевода строки;
- разобьем текст на предложения;
- удалим любой текст отступлений в середине предложения, заключенный в круглые скобки;
- удалим описания иллюстраций, не вошедших в текст;
- преобразуем все буквы в нижний регистр;
- удалим все знаки препинания.

Важно отметить, что эти операции должны применяться в определенном порядке. Например, удаление знаков препинания (включая квадратные скобки) затруднит идентификацию и удаление ссылок в дальнейшем. Удаление знаков препинания и преобразование всех символов в нижний регистр также помешают разделить текст на предложения.

Функции удаления символов перевода строки и преобразования букв в нижний регистр довольно просты:

```
def replace_newlines(text):
    return text.replace('\n', ' ')

def make_lowercase(text):
    return text.lower()
```

Вот пример текста, в котором заменим символы перевода строки пробелами, а не удалим их полностью:

```
It uses dynamic name resolution (late binding), which binds method
and variable names during program execution.
Its design offers some support for functional programming
in the Lisp tradition.
```

После замены получается такой текст:

```
It uses dynamic name resolution (late binding), which binds method
and variable names during program execution. Its design offers some
support for functional programming in the Lisp tradition.
```

Вставка пробелов гарантирует, что между всеми предложениями останутся пробелы.

Учитывая это, можно написать функцию для разделения предложений:

```
def split_sentences(text):
    return [s.strip() for s in text.split('. ')]
```

Разделяя предложения, мы ориентируемся не на одну только точку, а на точку с пробелом. Это предотвращает ошибочное разбиение десятичных дробей, например, в вездесущем «Python 2.5» или в примерах кода, таких как:

```
if (c = 1) { ...}
```

Кроме того, мы должны отыскать все лишние пробелы в начале и конце получившихся предложений и удалить их с помощью функции **strip**.

Однако функцию **split_sentences** нельзя вызвать без подготовки. Сразу после многих предложений следуют ссылки:

```
capable of exception handling and interfacing with the Amoeba
operating system.[13] Its implementation began in December 1989.[44]
```

Функцию удаления ссылок можно записать так:

```
import re

CITATION_REGEX = re.compile('\[[0-9]*\]')
def strip_citations(text):
    return re.sub(CITATION_REGEX, '', text)
```

Имя переменной `CITATION_REGEX` записано в верхнем регистре. Тем самым мы указываем, что она является константой и заранее компилируется вне самой функции. Функцию также можно было бы написать так:

```
def strip_citations(text):
    return re.sub(r'\[[0-9]*\]', '', text)
```

Однако в этом случае Python будет вынужден компилировать регулярное выражение при каждом вызове функции (а это может происходить и тысячи, и миллионы раз в зависимости от проекта). Поэтому было решено предварительно скомпилировать и подготовить его к работе только один раз. Несмотря на то что скорость работы программы не всегда является существенным препятствием при веб-скрапинге, тем не менее предварительная компиляция регулярных выражений вне функций проста и позволяет документировать код с помощью соответствующего имени переменной, хранящей регулярное выражение.

Удаление текста в скобках, такого как

```
all versions of Python (including 2.7[56]) had security issues
```

и

```
dynamic name resolution (late binding), which binds method
```

реализуется подобно удалению ссылок. Вот неплохая первая попытка:

```
PARENS_REGEX = re.compile('\(.*\)')
def remove_parentheses(text):
    return re.sub(PARENS_REGEX, '', text)
```

Эта функция действительно удалит текст в скобках в примерах выше, но также удалит все круглые скобки с их содержимым в примерах кода, например:

```
This has the advantage of avoiding a classic C error of mistaking
an assignment operator = for an equality operator == in conditions:
if (c = 1) { ...} is syntactically valid
```

Кроме того, есть опасность удалить лишнее, если в тексте есть непарные скобки. Встретив открывающую скобку, эта функция может удалить большие блоки, стремясь найти закрывающую скобку.

Чтобы решить эту проблему, можно посмотреть, какие типы символов обычно встречаются в тексте в скобках и искать только их, одновременно ограничив длину этого текста в скобках:

```
PARENS_REGEX = re.compile('\([a-z A-Z \+\.,\-]{0,100}\)')
def remove_parentheses(text):
    return re.sub(PARENS_REGEX, '', text)
```

Иногда в тексте могут присутствовать описания иллюстраций, которые не извлекаются. Например:

```
Hello world program:
```

Обычно эти описания короткие, начинаются с новой строки, содержат только буквы и заканчиваются двоеточием. Их можно удалить с помощью такого регулярного выражения:

```
DESCRIPTION_REGEX = re.compile('\n[a-z A-Z]*:')
def remove_descriptions(text):
    return re.sub(DESCRIPTION_REGEX, '', text)
```

Теперь можно убрать знаки препинания. Поскольку многие из предыдущих шагов используют знаки препинания, чтобы определить, какой текст удалить, а какой сохранить, удаление знаков препинания обычно является одним из последних шагов.

Модуль `string` (https://docs.python.org/3/library/string.html) содержит множество удобных наборов символов, один из которых — `string.punctuation`. Он содержит все знаки препинания ASCII:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Можно взять эту строку со всеми знаками препинания ASCII и превратить ее в регулярное выражение, использовав функцию `re.escape` (которая экранирует все зарезервированные символы регулярных выражений) и объединив все с помощью символа |:

```
puncts = [re.escape(c) for c in string.punctuation]
PUNCTUATION_REGEX = re.compile('|'.join(puncts))

def remove_punctuation(text):
    return re.sub(PUNCTUATION_REGEX, '', text)
```

При всех этих манипуляциях со строками можно заметить, что символы Unicode неправильно представляются в полученных нами строках. Одним из таких символов является неразрывный пробел, который в HTML представлен сущностью ` `. Его часто можно встретить в текстах из Интернета. В нашем фрагменте, полученном из «Википедии», его можно видеть как напечатанный `\xa0`:

```
python\xa020 was released...
```

Какие бы странные символы вам ни встретились, их всегда можно исправить с помощью пакета `unicodedata`. Нормализация символов Unicode станет нашим последним шагом в очистке текста:

```
def normalize(text):
    return unicodedata.normalize('NFKD', text)
```

Итак, мы создали набор коротких, хорошо организованных функций, выполняющих различные операции по очистке текста. Поскольку нам может потребоваться добавить или удалить функции или изменить порядок их вызова, мы можем поместить функции в список и вызывать их универсальным способом:

```
text_operations = [
    strip_citations,
    remove_parentheses,
    remove_descriptions,
    replace_newlines,
    split_sentences,
    make_lowercase,
    remove_punctuation,
    normalize
]

cleaned = content
for op in text_operations:
    if type(cleaned) == list:
        cleaned = [op(c) for c in cleaned]
    else:
        cleaned = op(cleaned)

print(cleaned)
```

Хотя Python обычно не рассматривается как функциональный язык, такой как JavaScript или даже Haskell, все же полезно помнить, что в подобных ситуациях функции можно передавать как переменные!

Нормализация текста

Что делать дальше с очищенным текстом? Часто после очистки текст разбивают на более мелкие фрагменты, которые проще оценивать и анализировать. Компьютерные лингвисты называют их *n-граммами*, где *n* — количество слов в каждом фрагменте текста. В этом примере мы будем работать конкретно с фрагментами текста из двух слов — 2-граммами, или биграммами.

N-граммы обычно не охватывают предложения. Таким образом, можно использовать текст, полученный в предыдущем разделе, разбить его на предложения и создать биграммы для каждого предложения в списке.

Следующая функция разбивает текст на *n*-граммы:

```
def getNgrams(text, n):
    text = text.split(' ')
    return [text[i:i+n] for i in range(len(text)-n+1)]

getNgrams('web scraping with python', 2)
```

Если передать этой функции текст *web scraping with python*, она выведет:

```
[['web', 'scraping'], ['scraping', 'with'], ['with', 'python']]
```

Первая очевидная проблема этой функции — она выдает много повторяющихся биграмм. Все биграммы, генерируемые этим кодом, заносятся в список без учета частоты появления. Интересно было бы фиксировать их частоту, а не просто констатировать их существование — это может оказаться полезным для оценки 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 можно изменить код так, чтобы *n*-граммы собирались не в список, а в объект `Counter`:

```
from collections import Counter

def getNgrams(text, n):
    text = text.split(' ')
    return [' '.join(text[i:i+n]) for i in range(len(text)-n+1)]

def countNGramsFromSentences(sentences, n):
    counts = Counter()
    for sentence in sentences:
        counts.update(getNgrams(sentence, n))
    return counts
```

Есть много других способов сделать это — например, вносить *n*-граммы в объект словаря, в котором значение элемента показывает, сколько раз данная *n*-грамма встречается в тексте. Недостаток такого подхода заключается в том, что он более трудоемкий и усложняет сортировку.

Однако использование объекта `Counter` тоже имеет свой недостаток: данный объект не позволяет хранить списки (они не хешируются), поэтому необходимо сначала преобразовать их в строки, используя операцию `' '.join(text[i:i+n])` при генерации списка для каждой *n*-граммы.

Можно вызвать функцию `countNGramsFromSentences` и передать ей наш очищенный текст, полученный в предыдущем разделе, а затем с помощью функции `most_common` получить список биграмм, отсортированный в порядке убывания частоты их встречаемости:

```
counts = countNGramsFromSentences(cleaned, 2)
print(counts.most_common())
```

Вот как выглядит получившийся результат:

```
[('in the', 19), ('of the', 19), ('such as', 18), ('as a', 14),
('in python', 12), ('python is', 9), ('of python', 9),
('the python', 9)...
```

На момент написания этой книги в статье о Python насчитывалось 2814 биграмм, причем самые популярные из них содержали словосочетания, часто встречающиеся в любом тексте на английском языке, например such as — «такой как». В зависимости от целей проекта вы можете удалить подобные *n*-граммы, которые не имеют особого отношения к фактическому содержимому. Подробнее об этом см. в главе 12.

Кроме того, обычно бывает полезным на минуту остановиться и подумать, сколько вычислительной мощности вы 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 какому-либо из предварительно запрограммированных эквивалентов.

Например, в списке биграмм присутствуют варианты Python 1st и Python first. Однако создание общего правила, которое гласит: «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.

Аналогично непоследовательное использование дефисов (например, наличие в одном тексте слов «ир-регулярный» 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 могут 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 «ир регулярных» и «регулярных» слов!

Очистка данных с помощью Pandas

Этот раздел посвящен не очаровательным медведям, обитающим в Китае, а пакету pandas, созданному специально для анализа данных на Python. Если прежде вам доводилось заниматься анализом данных и машинным обучением, то, скорее всего, вы уже сталкивались с этим пакетом, поскольку он широко применяется в этой области.

Pandas был создан программистом Уэсом Маккинни (Wes McKinney) в 2008 году как сольный проект. В 2009 году проект был обнародован и стал быстро завоевывать популярность. Этот пакет занял особую нишу в области обработки данных. В некотором смысле он функционировал как электронная таблица, поддерживал форматированный вывод и предлагал простые функции изменения формата. Он также использовал мощь и гибкость базового кода Python и библиотек обработки данных, на которых был построен.

Некоторые могут порекомендовать систему управления пакетами Anaconda (https://www.anaconda.com) для установки библиотек обработки данных, таких как numpy, pandas и scikit-learn. Но pandas, несмотря на превосходную поддержку в Anaconda, легко установить и с помощью pip:

```
pip install pandas
```

По соглашению пакет принято импортировать с сокращенным именем pd, а не с полным pandas:

```
import pandas as pd
```



Не импортируйте отдельные методы и классы из Pandas

Экосистема pandas — большая, сложная и часто перекрывает пространство имен встроенных функций и пакетов Python. По этой причине на функции pandas почти всегда следует ссылаться, начиная с pd, а не импортировать их напрямую, как в этом примере:

```
from pandas import array
from pandas.DataFrame import min
```

Такие инструкции импорта могут вызвать конфликт имен со встроенным модулем array и функцией min.

Одно принятое исключение можно сделать для класса DataFrame и импортировать его так:

```
from pandas import DataFrame
```

Класс DataFrame отсутствует в стандартной библиотеке Python, поэтому легко распознается как класс из пакета pandas. Однако это единственное исключение, которое вы, скорее всего, увидите, и многие по-прежнему предпочитают ссылаться на класс DataFrame как на pd.DataFrame. Поскольку на библиотеку часто ссылаются в коде, это одна из причин, почему принято импортировать pandas с кратким именем pd, а не с полным!

Объект из библиотеки pandas, с которым вы будете чаще всего работать, называется DataFrame. Он похож на электронные таблицы и может использоваться множеством способов. Например:

```
df = pd.DataFrame([['a', 1], ['b', 2], ['c', 3]])
df.head()
```

Метод head форматирует данные, хранящиеся в DataFrame, и выводит их в виде красиво оформленной таблицы с заголовками столбцов, как показано на рис. 11.1.

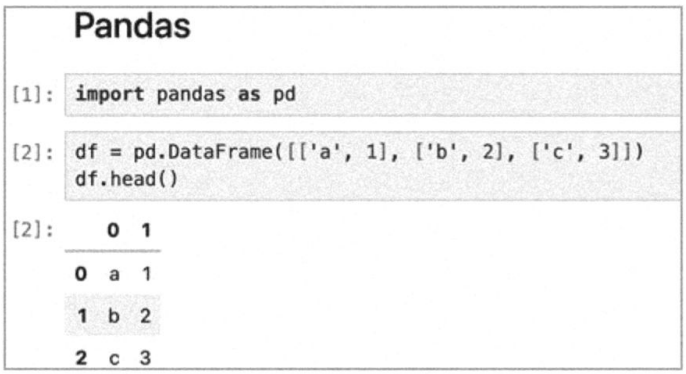


Рис. 11.1. Вывод содержимого DataFrame

Объекты DataFrame должны всегда иметь индекс и имена столбцов. Если они не заданы, как в этом случае, то он выводит простую таблицу с заголовками, сгенерированными автоматически. Индексы DataFrame — (0, 1, 2) — выделены полужирным шрифтом слева, а имена столбцов — (0, 1) — вверху.

DataFrame избавляет от необходимости работать с обычными списками и словарями Python и предоставляет огромное количество удобных вспомогательных функций для сортировки, очистки, обработки, упорядочения и отображения данных. При работе с большими наборами данных DataFrame также обеспечивает высокую скорость и экономию памяти, чем выгодно отличается от списков и словарей.

Очистка

В следующих примерах используются данные, полученные из списка стран, где есть рестораны McDonald's, доступного в «Википедии» (https://en.wikipedia.org/wiki/List_of_countries_with_McDonald%27s_restaurants). Для чтения данных непосредственно из файла CSV в DataFrame можно задействовать функцию pd.read_csv:

```
df = pd.read_csv('countries.csv')
df.head(10)
```

При желании методу head можно передать целое число, чтобы указать, сколько строк нужно вывести. По умолчанию выводятся пять строк. Этого достаточно, чтобы получить представление о данных CSV, извлеченных ранее, как показано на рис. 11.2.

```
[3]: df = pd.read_csv('countries.csv')
     df.head(10)
```

[3]:

	#	Country/territory	Date of first store	First outlet location	Max. no. ofoperatingoutlets	Source and date of source	People per outlet	Notes
1	1	United States	May 15, 1940Franchise: April 15, 1955	San Bernardino, CaliforniaDes Plaines, Illinoi...	13,515[10][failed verification][11]	(source: McDonald's United States March 29, 2023)	25,132	See McDonald's USA
2	2	Canada (details)	June 3, 1967	Richmond, British Columbia	1,400[10]	(source: McDonald's Canada 5 February 2022)	27,551	This is the first McDonald's outside of the Un...
3	3	Puerto Rico(territory of United States)	December 6, 1967	San Juan	108	(source: McDonald's 2013)	29,583	First McDonald's in Latin America and in the C...
4	4	United States Virgin Islands(territory of Unit	September 5, 1970	St. Croix	6	(source: McDonald's 2013)	17,878	NaN

Рис. 11.2. Вывод списка стран с дополнительными данными

Названия столбцов здесь получились слишком длинными и некорректно отформатированными. Их можно переименовать, используя метод `rename`:

```
df.rename(columns={
    '#': 'Order',
    'Country/territory': 'Country',
    'Date of first store': 'Date',
    'First outlet location': 'Location',
    'Max. no. ofoperatingoutlets': 'Outlets'
}, inplace=True)
```

Здесь в именованном аргументе `columns` передается словарь, где ключами являются исходные имена столбцов, а значениями — новые имена. Логический аргумент `inplace` указывает, что столбцы должны переименовываться на месте, в исходном объекте `DataFrame`, вместо возврата нового объекта `DataFrame`.

Далее вы можете отобрать только нужные вам столбцы, передав список имен этих столбцов в форме среза с использованием квадратных скобок `[]`:

```
df = df[['Order', 'Country', 'Date', 'Location', 'Outlets']]
```

Теперь, переименовав нужные нам столбцы, мы можем заняться данными в `DataFrame`. Кое-что в этих данных мы должны исправить. Во-первых, даты в столбцах `Date of first store` — «Дата открытия первого ресторана» и `Date` — «Дата» имеют правильный формат, но содержат также лишний текст или даже

другие даты. Простейшее решение — сохранить первое, что соответствует формату даты, и отбросить все остальное.

Функции `DataFrame` можно применять сразу ко всему столбцу, сначала выбрав этот столбец, используя тот же синтаксис «среза», что и выше. Выбранный столбец возвращается в виде экземпляра `Series`. Этот класс имеет метод `apply`, который применяет указанную функцию к каждому значению в `Series`:

```
import re

date_regex = re.compile('[A-Z][a-z]+ [0-9]{1,2}, [0-9]{4}')
df['Date'] = df['Date'].apply(lambda d: date_regex.findall(d)[0])
```

Здесь в роли функции я использовала лямбда-выражение, которое получает все совпадения с `date_regex` и возвращает первое из них в качестве даты.

После очистки эти даты можно преобразовать в фактические значения даты и времени `pandas` с помощью функции `to_datetime`:

```
df['Date'] = pd.to_datetime(df['Date'])
```

Часто существует тонкий баланс между быстрым и эффективным получением «чистых» данных и сохранением абсолютно точных и детальных данных. Например, очистка даты сократила текст *England: November 13, 1974[21] Wales: December 3, 1984 Scotland: November 23, 1987[22] Northern Ireland: October 12, 1991* в строке, соответствующей Соединенному Королевству (United Kingdom), до единственной даты: *1974-11-13*.

С технической точки зрения в этом нет ошибки. Если рассматривать Великобританию в целом, то первый ресторан «Макдоналдс» в этой стране действительно был открыт 13 ноября 1974 года. Однако это просто случайность, что даты оказались записаны в ячейке в хронологическом порядке и мы решили выбрать первую из них, хотя правильнее было бы выбрать самую раннюю. Нетрудно представить множество других обстоятельств, в которых нам может не повезти.

Иногда после обзора данных можно прийти к выводу, что выбранный метод очистки достаточно хорош, но изредка можно заметить, что в одной половине случаев он ошибается в одном, а во второй половине — в другом, достигая баланса при работе с большими наборами данных. Вы также можете решить, что нужен какой-то другой, более точный метод очистки или сбора данных.

В столбце Outlets — «Торговые точки» — набора данных представлены аналогичные проблемы. Он содержит следующий текст: *13,515[10][failed verification][11]* —«13,515[10][проверка не пройдена][11])» и *(excl. seasonal restaurants) 43 (including seasonal and mobile restaurants)* — «(исключая сезонные рестораны) 43 (включая сезонные и мобильные рестораны)». Эти данные не являются чистыми целыми числами, которые могут понадобиться нам для

дальнейшего анализа. И снова мы можем использовать простой подход, выбирая первое целое число, доступное в наборе данных:

```
int_regex = re.compile('[0-9,]+')

def str_to_int(s):
    s = int_regex.findall(s)[0]
    s = s.replace(',','')
    return int(s)

df['Outlets'] = df['Outlets'].apply(str_to_int)
```

То же самое можно реализовать в виде лямбда-функции, но если требуется выполнить несколько шагов, то можно создать отдельную именованную функцию. Преимущество обычных функций заключается в возможности вывести сообщения о любых исключениях, возникших в ходе подготовки данных к дальнейшим исследованиям:

```
def str_to_int(s):
    try:
        s = int_regex.findall(s)[0]
        s = s.replace(',','')
    except:
        print(f'Whoops: {s}')
    return int(s)
```

Теперь набор данных в `DataFrame` очищен и готов к дальнейшему анализу, как показано на рис. 11.3.

```
[96]: df.head(10)
```

[96]:

	Order	Country	Date	Location	Outlets
0	1	United States	1940-05-15	San Bernardino, CaliforniaDes Plaines, Illinoi...	13515
1	2	Canada (details)	1967-06-03	Richmond, British Columbia	1400
2	3	Puerto Rico(territory of United States)	1967-12-06	San Juan	108
3	4	United States Virgin Islands(territory of Unit...	1970-09-05	St. Croix	6
4	5	Costa Rica	1970-12-08	San José, 4th street, between 1st and Central ...	70
5	6	Australia	1971-05-30	Yagoona, New South Wales[12]	1000
6	7	Guam(territory of United States)	1971-06-10	Dededo	6
7	8	Japan	1971-07-21	Tokyo	2900
8	9	Netherlands	1971-08-21	Zaandam	263

Рис. 11.3. DataFrame с исправленными названиями столбцов, отформатированными датами и целочисленными значениями

Индексирование, сортировка и фильтрация

Выше я говорила, что все объекты `DataFrame` имеют индекс, независимо от того, определили вы его явно или нет. Сами данные о ресторанах «Макдоналдс» имеют удобный индекс: столбец `Order` — «Порядок». Он обозначает хронологический порядок, в котором в разных странах открывались эти рестораны. Вы можете назначить его индексом, используя метод `set_index`:

```
df.set_index(['Order'], inplace=True)
df.head()
```

Он удалит старый индекс и переместит в индекс столбец `Order`. Напомню, что именованный аргумент `inplace=True` означает, что операция выполняется на месте, в исходном объекте `DataFrame`, вместо возврата нового объекта `DataFrame`.

Для сортировки данных по одному или нескольким столбцам можно использовать метод `sort_values`. Он тоже принимает именованный аргумент `inplace`. Однако, поскольку сортировка обычно выполняется для анализа в ходе исследований и постоянная сортировка нежелательна, полезнее может оказаться возврат нового `DataFrame` для печати:

```
df.sort_values(by=['Outlets', 'Date'], ascending=False)
```

Итак, вы увидите, что наибольшее число ресторанов «Макдоналдс» открыто в США, за ними следует Китай, а затем Япония. Если говорить о Европе, то я думаю, французам будет приятно узнать, что Франция занимает четвертое место по количеству ресторанов «Макдоналдс» среди всех европейских стран!

Фильтровать данные в `DataFrame` можно с помощью метода `query`. В качестве аргумента он принимает строку запроса:

```
df.query('Outlets < 100')
```

Этот вызов вернет `DataFrame`, содержащий только записи, в которых столбец `Outlets` имеет значение меньше 100. В запросах к `DataFrame` можно использовать большинство операторов сравнения Python, однако сам язык запросов имеет синтаксис, отличный от синтаксиса Python. Например, такой запрос вызовет исключение:

```
df.query('Date is not None')
```

Чтобы проверить наличие или отсутствие каких-либо значений, в языке запросов `pandas` следует использовать функции `isnull` и `notnull`:

```
df.query('Date.isnull()')
df.query('Date.notnull()')
```

Как нетрудно догадаться, эти запросы выявляют и значения `None`, и объекты `NaN` из пакета `numpy`, на котором основана библиотека `pandas`.

Добавить еще одно логическое выражение можно через одиночный амперсанд:

```
df.query('Outlets < 100 & Date < "01-06-1990"')
```

Аналогом оператора `or` является вертикальная черта `|`:

```
df.query('Outlets < 100 | Date < "01-06-1990"')
```

Обратите внимание, что полная дата (`"1990-01-01"`) здесь не требуется, достаточно указать только год `"1990"`. Pandas довольно снисходительна к интерпретации дат в виде строк, хотя всегда старайтесь проверить и перепроверить соответствие возвращаемых данных вашим ожиданиям.

Дополнительные источники информации о Pandas

Я искренне надеюсь, что ваше знакомство с pandas на этом не закончится. Нам повезло, что Уэс Маккинни, создатель и «великодушный вечный повелитель» (https://en.wikipedia.org/wiki/Benevolent_dictator_for_life) проекта pandas, тоже написал книгу *Python for Data Analysis* (https://www.oreilly.com/library/view/python-for-data/9781098104023/)[1].

Если вы планируете всерьез заняться анализом данных или просто хотите иметь хороший инструмент для очистки и анализа данных в Python, то я рекомендую прочитать эту книгу.

[1] *Маккинни У.* Python и анализ данных. 2023.

ГЛАВА 12

Чтение и запись текстов на естественных языках

Данные, с которыми мы работали в рамках этой книги, до сих пор были представлены в виде чисел или исчисляемых величин. В большинстве случаев мы просто сохраняли информацию, никак ее после этого не анализируя. В данной главе будет затронута сложная проблема английского языка[1].

Каким образом Google узнает, что вам нужно, когда вы вводите в строку поиска изображений слова cute kitten — «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 dead parrot — «мертвый попугай»? По заголовку и описанию, которые сопровождают каждое загруженное видео.

На самом деле, даже если ввести что-то вроде deceased bird monty python — «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, a boiling puppy[2] — 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 реальных задач в терминах вероятности и алгоритмов.

---

[1] 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. Около 56 % содержимого Интернета по-прежнему 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.

[2] Можно перевести как «чертовски злой щенок». — *Примеч. пер.*

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 изображении[1]. 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.

Обобщение данных

В главе 11 было показано, как разделить текстовый контент на *n*-граммы — наборы фраз длиной в *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 Гаррисона (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/inauguration-Speech.txt**) в качестве источника данных для нескольких примеров в данной главе.

Слегка изменив код очистки текста, описанный в главе 11, его можно использовать для преобразования теста в список предложений, готовых к разбиению на *n*-граммы:

```
import re
import string

def replace_newlines(text):
    return text.replace('\n', ' ')

def make_lowercase(text):
    return text.lower()

def split_sentences(text):
    return [s.strip() for s in text.split('. ')]
```

[1] A Picture Is Worth a Thousand (Coherent) Words: Building a Natural Description of Images Oriol Vinyals [et al.] // Google Research Blog. — November 17, 2014. — URL: http://bit.ly/1HEJ8kX.

```
puncts = [re.escape(c) for c in string.punctuation]
PUNCTUATION_REGEX = re.compile('|'.join(puncts))
def remove_punctuation(text):
    return re.sub(PUNCTUATION_REGEX, '', text)
```

Затем извлечь текст и вызвать эти функции в определенном порядке:

```
content = str(
    urlopen('http://pythonscraping.com/files/inaugurationSpeech.txt').read(),
    'utf-8'
)

text_operations = [
    replace_newlines,
    split_sentences,
    make_lowercase,
    remove_punctuation
]
cleaned = content
for op in text_operations:
    if type(cleaned) == list:
        cleaned = [op(c) for c in cleaned]
    else:
        cleaned = op(cleaned)

print(cleaned)
```

И наконец, использовать очищенный текст, чтобы получить объект `Counter` всех биграмм и выбрать самые популярные:

```
def getNgrams(text, n):
    text = text.split(' ')
    return [' '.join(text[i:i+n]) for i in range(len(text)-n+1)]

def countNGramsFromSentences(sentences, n):
    counts = Counter()
    for sentence in sentences:
        counts.update(getNgrams(sentence, n))
    return counts

counts = countNGramsFromSentences(cleaned, 2)
print(counts.most_common())
```

Этот пример иллюстрирует удобство и мощь коллекций из стандартной библиотеки Python. Конечно, было бы не особенно сложно написать функцию, создающую счетчик на основе словаря, сортирующую данные по значениям и возвращающую наиболее популярные ключи. Однако знание встроенных коллекций и возможность выбрать ту, которая подходит для поставленной задачи, могут сэкономить много времени и сил!

На выходе получим, в частности, следующее:

```
[('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)...
```

Биграммы of the, in the и to the, как и самые обычные предлоги, едва ли заслуживают внимания. Из остальных чаще всего встречается the constitution (конституция). 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, 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 для создания простого фильтра, позволяющего отсеять наиболее распространенные биграммы. Благодаря добавлению функций `isCommon` и `filterCommon` уже первые 100 слов значительно улучшили наш результат:

```
COMMON_WORDS = ['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']

def isCommon(ngram):
    return any([w in COMMON_WORDS for w in ngram.split(' ')])

def filterCommon(counts):
    return Counter({key: val for key, val in counts.items() if not isCommon(key)})


filterCommon(counts).most_common()
```

В итоге вы получите следующие биграммы, которые встречаются в тексте более двух раз:

```
('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 текста инаугурационной речи президента.

Важно подчеркнуть: для фильтрации использованы результаты сравнительно свежего списка слов, распространенных в наше время. Это может оказаться неуместным, если учесть, что текст был написан в 1841 году. Однако поскольку мы задействовали 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 время и не создавать список слов, наиболее распространенных в 1841 году (хотя это было бы интересно).

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 биграмм дают следующие пункты.

- Конституция Соединенных Штатов является инструментом, содержащим в себе это разрешение на власть нескольким департаментам, которые составляют правительство.
- И такая возможность была предоставлена в виде института исполнительной власти, обусловленного Конституцией.
- Центральное правительство не присвоило себе ни одного из обеспеченных прав штатов.
- Вызванный из отставки — где я собирался провести остаток своей жизни, — чтобы занять наивысшую исполнительную должность в этой большой и свободной стране, я стою перед вами, соотечественники, чтобы принять присягу, которую Конституция прописывает обязательным условием для исполнения президентских обязанностей; кроме того, согласно давней традиции нашего правительства и, как мне кажется, согласно вашим ожиданиям, я собираюсь изложить вам основы тех принципов, которыми буду руководствоваться при исполнении обязанностей, к которым вскоре приступлю.
- Те органы печати, которые непосредственно финансируются правительством, никогда нельзя использовать для «оправдания виновных или отбеливания преступников»[1].

[1] Предложения взяты из перевода речи на русский язык: https://america-xix.ru/library/wh-harrison-inaugural/. — *Примеч. пер.*

Конечно, такое резюме вряд ли в обозримом будущем напечатают в CliffsNotes, но, учитывая, что размер исходного документа составляет 217 предложений, а четвертое («Вызванный из отставки...») довольно хорошо отражает основную тему, это не так уж плохо для первой попытки.

Для поиска «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 — «исключительно металлические деньги» — едва ли характерная фраза для инаугурационной речи президента. Для более длинных оборотов имеет смысл задействовать 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 поискать 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. 12.1.

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.

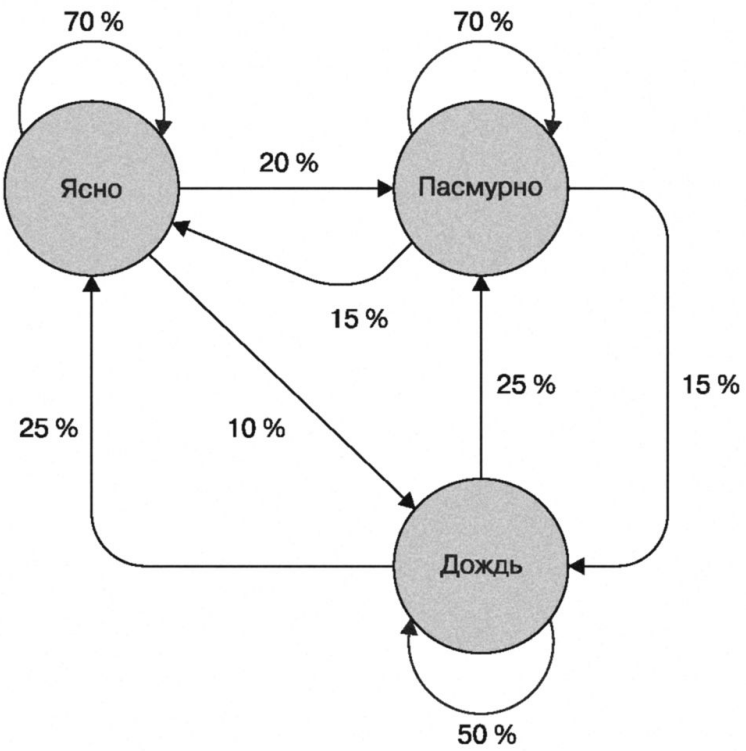


Рис. 12.1. Модель Маркова, описывающая теоретическую систему прогноза погоды

- Хотя в данной модели в любой момент существуют лишь три возможных варианта погоды, ее можно расширить, создав бесконечный список состояний погоды.
- Узел, в который вы перейдете, определяется только состоянием текущего узла. Если вы находитесь в узле «Ясно», то не имеет значения, были ли предыдущие 100 дней ясными или дождливыми, — вероятность ясной погоды на следующий день все равно составляет 70 %.
- Может оказаться так, что достичь одних узлов будет сложнее, 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 узлами. «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 использовав 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
from collections import defaultdict

def retrieveRandomWord(wordList):
    randIndex = randint(1, sum(wordlist.values()))
    for word, value in wordList.items():
        randIndex -= value
        if randIndex <= 0:
            return word

def cleanAndSplitText(text):
    # Удаляем разрывы строк и кавычки
    text = text.replace('\n', ' ').replace('"', '');

    # Знаки препинания тоже должны рассматриваться как "слова"
    # и включаться в цепь Маркова
    punctuation = [',','.',';',':']
    for symbol in punctuation:
        text = text.replace(symbol, f' {symbol} ');
    # Убираем пустышки
    return [word for word in text.split(' ') if word != '']

def buildWordDict(text):
    words = cleanAndSplitText(text)
    wordDict = defaultdict(dict)
    for i in range(1, len(words)):
        wordDict[words[i-1]][words[i]] = \
        wordDict[words[i-1]].get(words[i], 0) + 1
    return wordDict

text = str(urlopen('http://pythonscraping.com/files/inaugurationSpeech.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 добавляет пробелы вокруг остальных знаков препинания, чтобы они 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 слово `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 остановились[1]. В данном примере словаря словарей, если мы сейчас находимся на слове `word_e`, то 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 начального слова (в данном случае вездесущего «I» — «Я»), мы можем легко перемещаться по цепи Маркова, генерируя столько слов, сколько захотим.

---

[1] 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 на практике при реализации генератора Маркова может потребоваться специальная обработка случая, когда достигнуто последнее слово в тексте.

Вдобавок цепи Маркова имеют тенденцию становиться реалистичнее по мере увеличения количества текста, особенно из источников с похожим авторским стилем. В этом примере для создания цепи использовались биграммы (в которых следующее слово определяется по предыдущему), однако можно задействовать 3-граммы или *n*-граммы более высокого порядка, где каждое следующее слово определяется двумя или более предшествующими словами.

Несмотря на интересные результаты и большие выгоды, которые можно получить, накопив много мегабайтов текста в процессе веб-скрапинга, из-за подобных применений бывает трудно заметить практическую пользу цепей Маркова. Ранее в этом разделе отмечалось, что цепи Маркова моделируют связи между страницами на сайтах. С помощью больших коллекций подобных ссылок, представленных в виде указателей, можно строить веб-графы, полезные для хранения, отслеживания и анализа. То есть цепи Маркова позволяют 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 6 мы создали веб-скрапер, который собирает ссылки из одной статьи «Википедии» на следующую, начиная со статьи о Кевине Бейконе, а в главе 9 добавили сохранение ссылок в базе данных. Почему я снова об этом вспоминаю? Потому что, как 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 цепи Маркова, в которой заданы первое и последнее слова.

Задачи такого рода относятся к задачам *направленных графов*, где наличие связи А → Б еще не означает наличия связи Б → А. 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, то и она 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, соответственно, пути между статьей в «Википедии» о Кевине Бейконе и любой другой статьей на этом сайте 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 9, 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, то 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 Айдла (с идентификатором страницы 28624):

```
/wiki/Kevin_Bacon
/wiki/Primetime_Emmy_Award_for_Outstanding_Lead_Actor_in_a_
Miniseries_or_a_Movie
/wiki/Gary_Gilmore
/wiki/Eric_Idle
```

Если преобразовать это в соотношения ссылок, то получится следующее: Кевин Бейкон → премия «Эмми Прайм-тайм» → Гэри Гилмор → Эрик Айдл.

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? Теперь нужно 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. Функциональность NLTK огромна (ей посвящены целые книги), однако в текущем разделе мы рассмотрим лишь несколько способов применения данной библиотеки.

Установка и настройка

Модуль `nltk` можно установить так же, как и другие модули Python: либо скачать пакет напрямую с сайта NLTK, либо воспользоваться одним из многочисленных сторонних инсталляторов с именем, включающим слово nltk. Подробные инструкции по установке модуля см. на сайте NLTK (**http://www.nltk.org/install.html**).

После установки NLTK рекомендуется скачать готовый текстовый корпус, чтобы было удобнее пользоваться данными функциями. Для этого нужно ввести в командной строке Python следующее:

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

В результате запустится NLTK Downloader. Вы можете перемещаться по нему в терминале, используя перечисленные в меню команды:

```
NLTK Downloader
---------------------------------------------------------------------------
    d) Download   l) List    u) Update   c) Config   h) Help   q) Quit
---------------------------------------------------------------------------

Downloader> l

Packages:

  [*] abc..................... Australian Broadcasting Commission 2006
  [ ] alpino.................. Alpino Dutch Treebank
  [*] averaged_perceptron_tagger Averaged Perceptron Tagger
  [ ] averaged_perceptron_tagger_ru Averaged Perceptron Tagger (Russian)
  [ ] basque_grammars..... Grammars for Basque
  [ ] bcp47.............. BCP-47 Language Tags
  [ ] biocreative_ppi..... BioCreAtIvE (Critical Assessment of Information
                           Extraction Systems in Biology)
  [ ] bllip_wsj_no_aux.... BLLIP Parser: WSJ Model
  [*] book_grammars....... Grammars from NLTK Book
  [*] brown............... Brown Corpus
  [ ] brown_tei........... Brown Corpus (TEI XML Version)
  [ ] cess_cat............ CESS-CAT Treebank
  [ ] cess_esp............ CESS-ESP Treebank
  [*] chat80.............. Chat-80 Data Files
  [*] city_database....... City Database
  [*] cmudict............. The Carnegie Mellon Pronouncing Dictionary (0.6)
  [ ] comparative_sentences Comparative Sentence Dataset
  [ ] comtrans............ ComTrans Corpus Sample
  [*] conll2000........... CONLL 2000 Chunking Corpus

Hit Enter to continue:
```

Последняя страница содержания корпуса включает перечень коллекций:

```
Collections:
  [P] all-corpora......... All the corpora
  [P] all-nltk............ All packages available on nltk_data gh-pages
                           branch
  [P] all................. All packages
  [*] book................ Everything used in the NLTK Book
  [P] popular............. Popular packages
  [P] tests............... Packages for running tests
  [ ] third-party......... Third-party data packages

([*] marks installed packages; [P] marks partially installed collections)
```

В упражнениях этой главы мы будем использовать коллекцию book. Ее можно скачать через интерфейс NLTK Downloader или из кода на Python:

```
nltk.download('book')
```

Статистический анализ с помощью NLTK

NLTK отлично подходит для получения статистической информации о количестве, 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, которые можно импортировать из модуля `book`:

```
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 сценарии 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), ('[', 3
19), (']', 312), ('the', 299), ('I', 255), ('ARTHUR', 225)]
>>> fdist["Grail"]
34
```

Поскольку это сценарий, то в нем могут появляться отдельные специфические элементы, свойственные именно сценариям. 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!

Рассмотренные в предыдущих главах 2-граммы 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 сценарии 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)

[f for f in fourgrams if f[0] == 'coconut']
```

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”[1] вполне понятно человеку, но веб-скрапер может посчитать, что одно и то же слово используется здесь

[1] На русский язык эту фразу можно перевести как «Он был объективен в достижении своей цели — написать объективную философию, в первую очередь используя глаголы в косвенном падеже», что не представляет проблемы для понимания. В качестве русского аналога можно привести такой пример: «Косил косой косой косой». — *Примеч. пер.*

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 ACME — это хорошо» и «Продукты ACME — 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/docs/LDC95T7/cl93.html). 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, сравнительная степень
JJS	Adjective, superlative	Прилагательное, превосходная степень
LS	List item marker	Обозначение элемента списка
MD	Modal	Модальный
NN	Noun, singular or mass	Существительное, единственное число или неисчисляемое
NNS	Noun, plural	Существительное, множественное число

NNP	Proper noun, singular	Имя собственное, единственное число
NNPS	Proper noun, plural	Имя собственное, множественное число
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, неопределенная форма
VBD	Verb, past tense	Глагол, прошедшее время
VBG	Verb, gerund or present participle	Глагол, герундий или причастие настоящего времени
VBN	Verb, past participle	Глагол, причастие прошедшего времени
VBP	Verb, non-third-person singular present	Глагол, настоящее время, единственное число, не от третьего лица
VBZ	Verb, third person singular present	Глагол, настоящее время, единственное число, от 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), второй раз — как глагол (to dust)[1]. NLTK правильно распознает оба варианта применения, основываясь на контексте этих слов в предложении. NLTK идентифицирует части речи с помощью бесконтекстной грамматики, существующей в английском языке. *Бесконтекстные 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 бесконтекстной грамматики и выбирается часть речи, соответствующая этим правилам.

Зачем нам знать, является слово глаголом или существительным в данном контексте? Вероятно, это имеет смысл в научно-исследовательской лаборатории по информатике, но как может помочь при веб-скрапинге?

---

[1] Предложение "The dust was thick so he had to dust" можно перевести как «Слой пыли был толстый, и ему пришлось смахнуть пыль». — *Примеч. пер.*

ВИДЫ МАШИННОГО ОБУЧЕНИЯ

NLTK позволяет создавать совершенно новые бесконтекстные 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 16, когда будем обучать веб-скраперы распознавать символы капчи.

Одна из распространенных задач веб-скрапинга связана с поиском. Например, при веб-скрапинге текста с сайта вы можете захотеть найти слова *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 = [
    '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:
    for word, tag in pos_tag(word_tokenize(sentence)):
        if word.lower() == 'google' and tag in nouns:
            print(sentence)
```

Программа выводит только те предложения, в которых содержится слово google (или Google) в значении существительного, а не глагола. Конечно, можно было 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 бóльшую 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 — одни 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 начальное направление, с которого следует начинать осуществление проекта, требующего 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, Эвана Кляйна и Эдварда Лопера (O'Reilly) (http://oreil.ly/1HYt3vV) эта тема изложена комплексно, начиная с основ.

Кроме того, есть несколько более сложное теоретическое руководство — книга *Natural Language Annotations for Machine Learning* Джеймса Пустейовского и Эмбер Стаббс (O'Reilly) (http://oreil.ly/S3BudT). Чтобы выполнить представленные там уроки, вам понадобятся знания Python; для тем, рассмотренных в этой книге, прекрасно подходит Python Natural Language Toolkit.

ГЛАВА 13

Краулинг форм и авторизация

Один из первых вопросов, который появляется, 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 авторизация 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`-запросов по протоколу 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 Python 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 т. д. 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 2), поэтому на **pythonscraping.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.

Во-вторых, 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 сценарий должен вернуть следующий контент страницы:

```
Hello there, Ryan Mitchell!
```

Этот сценарий применим ко многим простым формам, встречающимся в Интернете. Например, форма подписки на новостную рассылку Web Scraping with Python выглядит так:

```
<form id="eclg-form">
  <div class="input-field">
    <label>First Name</label>
    <input type="text" name="first_name" class="eclg_firstname">
  </div>
  <div class="input-field">
    <label>Last Name</label>
    <input type="text" name="last_name" class="eclg_lastname">
  </div>
  <div class="input-field">
    <label>Email</label>
    <input type="text" name="email" class="eclg_email">
  </div>
  <div class="input-field input-submit">
    <button type="button" id="eclg-submit-btn">Send </button>
    <div class="eclg_ajax_loader" style="display: none;">
<img decoding="async" src="https://pythonscraping.com/wp-content/
plugins/email-capture-lead-generation//images/ajax_loader.gif">
</div>
  </div>
  <div class="eclg-message-container"></div>
</form>
```

На первый взгляд это, возможно, смотрится пугающе, но учтите, что в большинстве случаев (исключения из данного правила мы рассмотрим позже) вам нужны только два элемента:

- имя поля (или полей), данные которого (которых) вы хотите передать (в данном случае имя `first_name`, фамилию `last_name` и электронную почту `email`);
- атрибут `action` самой формы, то есть страница, на которую отправляется форма.

В данном случае атрибут `action` отсутствует. В отличие от традиционной HTML-формы, на этой странице используется программа, написанная на JavaScript, которая перехватывает событие отправки формы и посылает ее по указанному в программе URL.

В подобных ситуациях могут помочь инструменты разработчика в вашем браузере. Просто откройте вкладку **Network** (Сеть), заполните форму, нажмите кнопку **Submit** (Отправить) и наблюдайте за значениями, отправляемыми по сети (рис. 13.1).

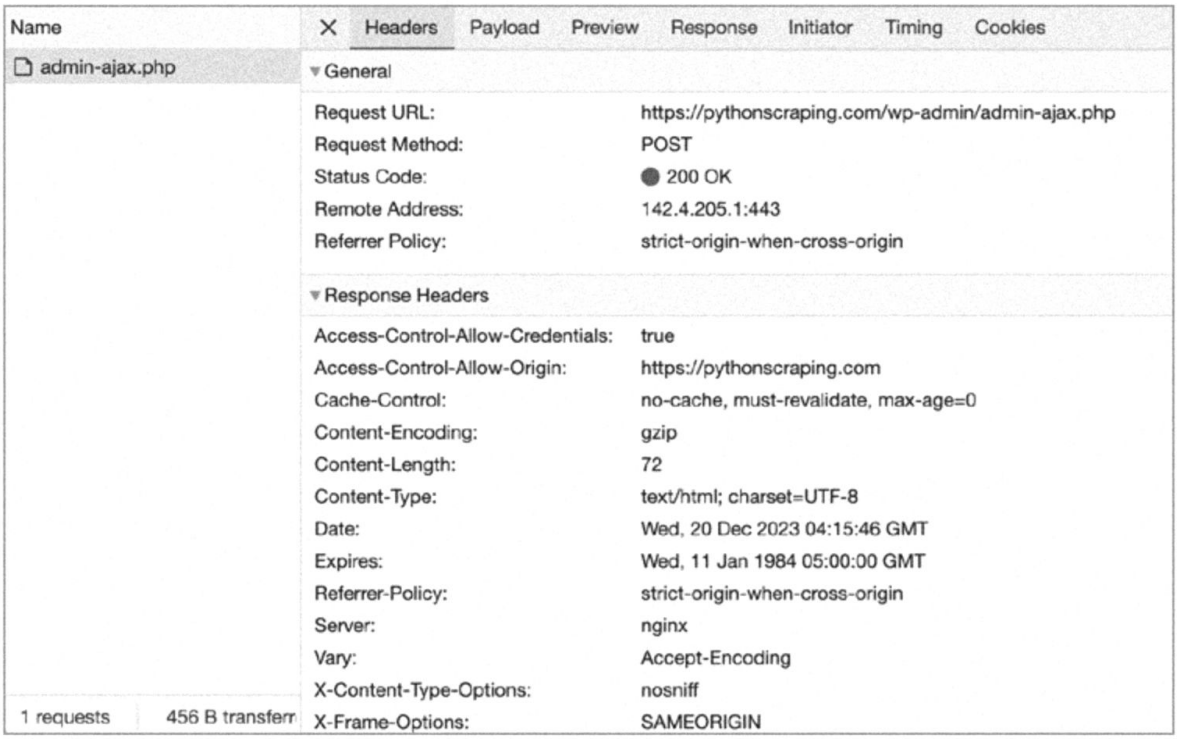


Рис. 13.1. Отправка запроса с формой подписки на pythonscraping.com

Не сомневаюсь, что вы сможете разобраться в запутанном коде JavaScript и получить тот же ответ, но использование вкладки **Network** (Сеть) в инструментах разработчика позволит легко увидеть, что содержимое формы отправляется на **https://pythonscraping.com/wp-admin/admin-ajax.php**.

Кроме того, на вкладке **Payload** (Данные форм) отображается четвертое значение формы, отправленное этой конечной точке: `action: eclg_add_newsletter`.

Благодаря этому вы можете воспроизвести отправку формы в коде на Python:

```
import requests
params = {
    'firstname': 'Ryan',
    'lastname': 'Mitchell',
```

```
    'email': 'ryanemitchell@gmail.com',
    'action': 'eclg_add_newsletter'
}
r = requests.post('https://pythonscraping.com/wp-admin/admin-ajax.php',
                  data=params)
print(r.text)
```

В данном случае возвращается ответ в формате JSON:

```
{"status":"1","errmsg":"You have subscribed successfully!."}
```

Переключатели, флажки и поля ввода

Очевидно, что не каждая 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 т. д. 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, если посмотреть в исходный код и найти атрибут `name`. Поиск значения иногда бывает сложным, поскольку оно может заполняться JavaScript непосредственно перед отправкой формы. В качестве примера можно привести такое 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
```

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 встроенного в браузер инструментов разработчика (рис. 13.2).

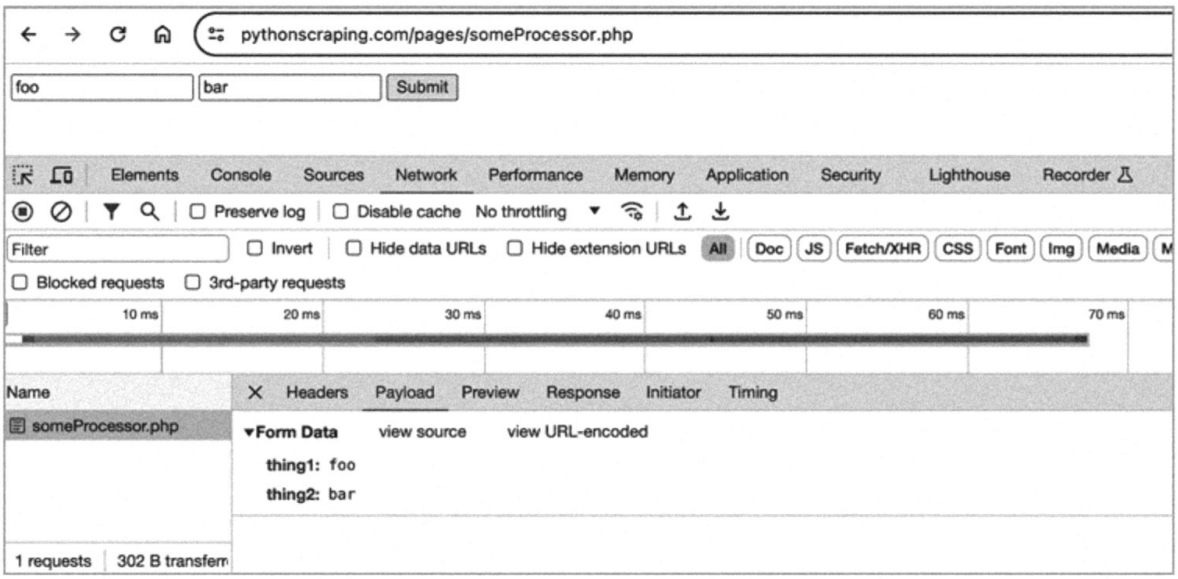


Рис. 13.2. В разделе Form Data, отмеченном рамкой, показаны POST-параметры thing1 и thing2 со значениями foo и bar

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 сценарий 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?

Большинство современных сайтов отслеживают, кто вошел в систему, а кто нет, 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 были огромной проблемой.

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(r.text)

print('Cookie is set to:')
print(r.cookies.get_dict())
print('Going to profile page...')
r = requests.get('http://pythonscraping.com/pages/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 затем 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*:

```
session = requests.Session()

params = {'username': 'Ryan', 'password': 'password'}
s = session.post('http://pythonscraping.com/pages/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/cookies/profile.php')
print(s.text)
```

В этом случае объект `session`, полученный вызовом функции `request.Session()`, отслеживает информацию о сессии, такую как cookie-файлы, заголовки и даже информацию о протоколах, которые могут использоваться поверх HTTP, например HTTPAdapters.

Requests — просто фантастическая библиотека, по полноте поддерживаемых операций уступающая, пожалуй, только Selenium (см. главу 14), 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://pythonscraping.com/pages/auth/login.php** (рис. 13.3).

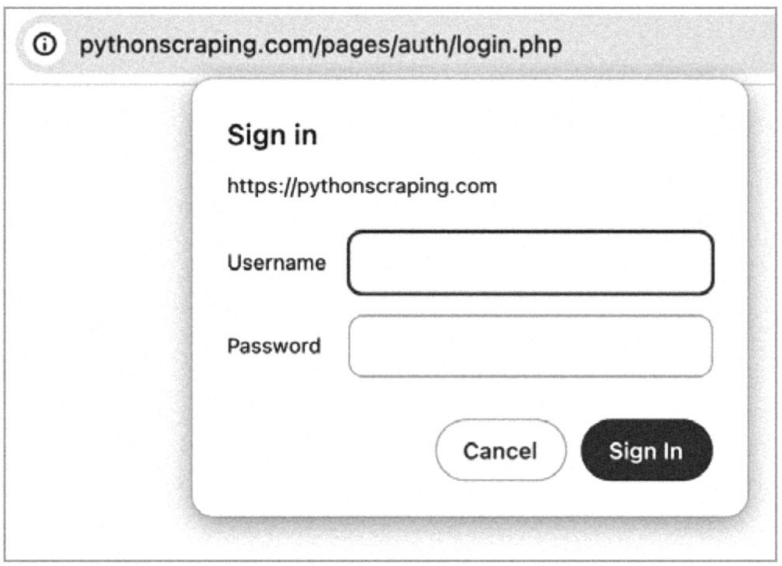


Рис. 13.3. Пользователь должен ввести свое имя и пароль, чтобы открыть страницу, защищенную базовой аутентификацией доступа

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 16, где 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 17, в которой рассматриваются ловушки для хакеров (honeypots), скрытые поля и другие меры безопасности, которые применяются на сайтах для защиты форм.

ГЛАВА 14

Веб-скрапинг данных JavaScript

Языки сценариев 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.)

В отличие от серверных языков, которых насчитываются сотни, клиентский язык остался только один — 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.



Другие клиентские языки программирования

Некоторые читатели могут не согласиться с моим заявлением «клиентский язык остался только один». С технической точки зрения они правы: помимо JavaScript существуют и такие языки, как ActionScript и VBScript, однако они больше не поддерживаются, а VBScript вообще всегда поддерживался только одним браузером. По этой причине их очень редко можно увидеть.

И если уж быть до конца честными, то следует также заметить, что абсолютно каждый может создать новый язык программирования на стороне клиента! И вероятно, их уже довольно много! Единственная проблема — получить широкую поддержку со стороны браузеров, чтобы этот язык эффективно всеми использовался.

Некоторые также утверждают, что CSS и HTML сами являются языками программирования. Я согласна с этим теоретически. У Лары Карки есть интересная статья в блоге на эту тему: https://notlaura.com/is-css-turing-complete/.

Однако на практике CSS и HTML чаще рассматриваются как языки разметки, отдельные от языков программирования, и, кроме того, они тоже довольно подробно описываются в этой книге.

В настоящее время JavaScript — самый распространенный в Интернете и лучше всего поддерживаемый клиентский язык сценариев. 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 сценарии 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 только поэтому 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 сценарная 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 типизированное объявление (`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 () {
        [b, a] = [a + b, b];
```

```
        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 5). 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`.

Также можно увидеть подобные функции, написанные с использованием стрелочного синтаксиса, появившегося в JavaScript ES6 (ECMAScript 6) в 2015 году:

```
<script>
const fibonacci = () => {
    var a = 1; var b = 1;
    return () => {
        [b, a] = [a + b, b];
        return b;
    }
}
const 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>
```

Здесь я использовала ключевое слово `const` и определила переменную-константу, которая не будет изменяться позже. Также иногда можно встретить ключевое слово `let`, обозначающее переменную, которой можно присвоить другое значение. Эти ключевые слова тоже появились в ES6.

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 чрезвычайно много времени и вычислительных ресурсов, особенно в крупных масштабах. Знание JavaScript и умение парсировать код непосредственно (не прибегая к необходимости выполнять его для получения информации) могут быть невероятно полезными навыками, которые избавят вас от множества проблем.

jQuery

jQuery — чрезвычайно распространенная библиотека, используемая более чем на 70 % всех веб-сайтов[1]. Понять, что на сайте применяется 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 сценария 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» ниже).

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[2], 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">
```

[1] См. анализ Web Technology Surveys по адресу https://w3techs.com/technologies/details/js-jquery. Для отслеживания изменения тенденций в использовании технологий с течением времени сервис W3Techs тоже использует веб-краулеры.

[2] Usage Statistics and Market Share of Google Analytics for Websites // W3Techs. — URL: http://w3techs.com/technologies/details/ta-googleanalytics/all/all.

```
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 сценарий 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 проведете хоть сколько-нибудь времени в Интернете, то почти наверняка встретите там встроенные в сайты карты *Google Maps*. Эта библиотека позволяет с легкостью встроить 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 координат широты и долготы.

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 распространенному 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 Ajax». Правильнее было бы сказать: «Данная форма будет использовать Ajax для связи с веб-сервером».

Подобно Ajax, *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, цвет фона может измениться при щелчке мыши, а по запросу Ajax — 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 необязательно означает, что данная страница относится к DHTML, даже если выглядит динамично. Кроме того, за рядом самых скучных, 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 сценарий 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, выполняющем 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.

Selenium не имеет собственного браузера и интегрируется 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 решением, однако я предпочитаю, чтобы мои сценарии 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.

Установка и использование Selenium

Библиотеку Selenium можно скачать с ее сайта (**https://pypi.python.org/pypi/selenium**) или установить из командной строки с помощью стороннего инсталлятора, такого как pip.

```
$ pip install selenium
```

Предыдущие версии Selenium требовали также вручную загружать веб-драйвер, позволяющий библиотеке взаимодействовать с вашим веб-браузером. Такое название веб-драйвер получил потому, что он управляет веб-браузером. Подобно *обычному драйверу* для аппаратного устройства, он позволяет пакету Selenium взаимодействовать с браузером и управлять им.

К сожалению, браузеры стали обновляться намного чаще, чем раньше, и при этом обновление происходит автоматически, из-за этого приходилось часто обновлять и веб-драйверы для Selenium. Переход на веб-сайт драйвера браузера (например, **http://chromedriver.chromium.org/downloads**), загрузка нового и замена старого файла стали рутиной. В версии Selenium 4, выпущенной в октябре 2021 года, появился пакет менеджера веб-драйверов.

Менеджер веб-драйверов можно установить с помощью pip:

```
$ pip install webdriver-manager
```

При вызове менеджер веб-драйверов загружает последнюю версию драйвера:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
driver.get("http://www.python.org")
time.sleep(2)
driver.close()
```

Конечно, было бы расточительством тратить ресурсы на установку нового файла каждый раз, когда запускается этот сценарий на случай, если браузер Chrome обновился с момента последнего запуска. Поэтому результатом работы диспетчера драйверов является просто путь в каталоге `driver`, где находится драйвер:

```
CHROMEDRIVER_PATH = ChromeDriverManager().install()
driver = webdriver.Chrome(service=Service(CHROMEDRIVER_PATH))
```

Однако, если вам нравится загружать файлы вручную, то можете передавать свой путь в объект `Service`:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service

CHROMEDRIVER_PATH = 'drivers/chromedriver_mac_arm64/chromedriver'
driver = webdriver.Chrome(service=Service(CHROMEDRIVER_PATH))
driver.get("http://www.python.org")
time.sleep(2)
driver.close()
```

Ajax применяется для загрузки данных на многих страницах, 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, который вызывается с помощью объекта `webdriver` (https://selenium-python.readthedocs.io/api.html). Обратите внимание: это объект Python, представляющий собой или действующий как интерфейс к предварительно загруженному приложению, которым управляет `webdriver`. Несмотря на то что термины «драйвер» и «веб-драйвер» часто используются взаимозаменяемо для обозначения обоих объектов (объекта Python и самого приложения), важно различать их концептуально.

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 элементами страницы (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.common.by import By
from selenium.webdriver.chrome.options import Options
import time

chrome_options = Options()
chrome_options.add_argument('--headless')
driver = webdriver.Chrome(
    service=Service(CHROMEDRIVER_PATH),
    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()
```

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 новый экземпляр Chrome WebDriver в Python, можно передать 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')
```

Независимо от того, используете вы пакет менеджера драйверов или загружаете драйвер самостоятельно, вы должны передать в конструктор объект `Service`, включающий этот путь, а также параметры веб-драйвера:

```
driver = webdriver.Chrome(
    service=Service(CHROMEDRIVER_PATH),
    options=chrome_options
)
```

Если все настроено правильно, то выполнение сценария займет несколько секунд, а затем появится следующий текст:

```
Here is some important text you want to retrieve!
A button to click!
```

Селекторы Selenium

В предыдущих главах мы выбирали элементы страницы с помощью селекторов `BeautifulSoup`, таких как `find` и `find_all`. В Selenium для поиска элементов используется похожий набор селекторов: `find_element` и `find_elements`.

Существует так много способов поиска и выбора элементов HTML, что можно подумать, будто Selenium поддерживает широкий спектр аргументов в этих методах. Однако в `find_element` и `find_elements` можно передать только два аргумента: объект `By` и строка селектора.

Объект `By` определяет, как должна интерпретироваться строка селектора, предлагая следующие варианты:

- `By.ID` — используется в примере; отыскивает элементы по HTML-атрибуту `id`;
- `By.NAME` — отыскивает элементы по HTML-атрибуту `name`; удобно использовать для поиска HTML-форм;
- `By.XPATH` — использует выражение XPath для выбора совпадающих элементов (синтаксис XPath будет рассматриваться далее в этой главе);
- `By.LINK_TEXT` — отыскивает элементы `<a>` по содержащемуся в них тексту, например, ссылку с надписью «Далее» можно найти, передав параметры `(By.LINK_TEXT, 'Далее')`;
- `By.PARTIAL_LINK_TEXT` — действует подобно `LINK_TEXT`, но ищет частичное совпадение со строкой;
- `By.TAG_NAME` — отыскивает элементы по имени тега;
- `By.CLASS_NAME` — отыскивает элементы по HTML-атрибуту `class`. Почему эта функция называется `CLASS_NAME`, а не просто `CLASS`? Использование формы `object.CLASS` создаст проблемы для библиотеки Selenium на Java, где `.class`

является зарезервированным методом. Чтобы обеспечить единообразие синтаксиса Selenium на разных языках, решено использовать имя `CLASS_NAME`;

- `By.CSS_SELECTOR` — отыскивает элементы по HTML-атрибуту `class`, `id` или имени тега, опираясь на соглашение `#idName`, `.className`, `tagName`.

В этом примере мы использовали селектор `driver.find_element(By.ID, 'content')`, 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.TAG_NAME, 'div')
```

Если же вы все равно хотите использовать BeautifulSoup для парсинга контента сайта, то можете применить функцию веб-драйвера `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())
```

Ожидание окончания загрузки

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`:

```
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(
    service=Service(CHROMEDRIVER_PATH),
    options=chrome_options)

driver.get('http://pythonscraping.com/pages/javascript/ajaxDemo.html')
try:
    element = WebDriverWait(driver, 10).until(
                        EC.presence_of_element_located((By.ID, 'loadedButton')))
finally:
    print(driver.find_element(By.ID, 'content').text)
    driver.close()
```

В этом сценарии 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 сценария ожидается определенное состояние DOM, в то время как при явном ожидании отсчитывается жестко заданное время, как в предыдущем примере, где длительность ожидания составляла три секунды. При неявном ожидании инициирующее состояние DOM определяется с помощью ожидаемого условия `expected_condition` (обратите внимание: здесь импортируемый модуль для краткости обозначен как `EC` по общему правилу). 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 в разделе «Селекторы Selenium»). *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 веб-драйвера `find_element`:

```
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.

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 в режиме консоли. Однако работа без режима консоли и/или применение разных драйверов браузера может оказаться полезной 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.

В большинстве случаев для получения драйверов браузеров предпочтительнее использовать менеджер веб-драйверов. Например, вы можете использовать менеджер веб-драйверов для Firefox и Microsoft Edge:

```
from webdriver_manager.firefox import GeckoDriverManager
from webdriver_manager.microsoft import EdgeChromiumDriverManager

print(GeckoDriverManager().install())
print(EdgeChromiumDriverManager().install())
```

Однако если вам нужна поддержка устаревшей версии браузера или драйвер, недоступный через менеджер веб-драйверов (например, Safari), то вам придется загрузить файлы драйвера вручную.

Сегодня в создании и обслуживании веб-драйверов Selenium для всех крупнейших браузеров принимает участие множество групп разработчиков, как официальных, так и неофициальных. Среди них есть и группа Selenium, курирующая коллекцию веб-драйверов (http://www.seleniumhq.org/download/), обеспечивающих удобство их использования с Selenium.

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 6). Перенаправления на стороне клиента вообще не будут обрабатываться, если в веб-скрапере не используется код JavaScript.

Selenium обрабатывает перенаправления JavaScript идентично другим сценариям 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, «запомнив» некий DOM-элемент сразу после загрузки страницы, а затем многократно проверяя его, 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.common.exceptions import StaleElementReferenceException
import time

def waitForLoad(driver):
    elem = driver.find_element(By.TAG_NAME, "html")
```

```
    count = 0
    for _ in range(0, 20):
        try:
            elem == driver.find_element(By.TAG_NAME, "html")
        except StaleElementReferenceException:
            return
        time.sleep(0.5)
    print("Timing out after 10 seconds and returning")

chrome_options = Options()
chrome_options.add_argument("--headless")
driver = webdriver.Chrome(
    service=Service(CHROMEDRIVER_PATH),
    options=chrome_options
)
driver.get('http://pythonscraping.com/pages/javascript/redirectDemo1.html')
waitForLoad(driver)
print(driver.page_source)
driver.close()
```

Данный сценарий проверяет страницу каждые полсекунды с предварительной задержкой в 10 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/javascript/redirectDemo1.html')
try:
    txt = 'This is the page you are looking for!'
    bodyElement = WebDriverWait(driver, 15).until(
        EC.presence_of_element_located((
            By.XPATH,
```

```
            f'//body[contains(text(), "{txt}")]'
        ))
    )
    print(bodyElement.text)
except TimeoutException:
    print('Did not find the element')
```

Последнее замечание о JavaScript

JavaScript используется на большинстве современных сайтов[1]. К счастью, его применение, как правило, 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, которая была изучена 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 парсировать 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 оперативно. Подробнее об этом вы узнаете в главе 15.

Если у вас все же возникнут проблемы с особенно запутанным JavaScript, то в главе 17 вы найдете информацию о Selenium и непосредственном взаимодействии с динамическими сайтами, включая интерфейсы типа drag-and-drop.

[1] W3Techs, «Usage Statistics of JavaScript as Client-Side Programming Language on Websites». http://w3techs.com/technologies/details/cp-javascript/all/all.

ГЛАВА 15

Краулинг с помощью API

JavaScript традиционно принято считать вселенским проклятием веб-краулеров. Давным-давно, в древней истории Интернета, 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 такой же запрос.

По мере распространения методов генерации и загрузки контента с помощью JavaScript и Ajax описанная выше ситуация становится все менее типичной. В главе 14 был рассмотрен один из способов решения указанной проблемы: использование Selenium для автоматизации браузера и извлечения данных. Оказывается, это легко сделать и работает почти всегда.

Суть 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 этой главе вы узнаете, 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 REST, GraphQL, JSON и XML API, однако все они основаны на одной простой концепции. *API* определяет стандартизованный синтаксис, который позволяет одной программе взаимодействовать с другой, даже если эти программы 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, Национальная метеорологическая служба США предоставляет API для получения данных о прогнозе погоды (**https://www.weather.gov/documentation/services-web-api**) на любой территории. У Google в разделе, адресованном разработчикам (**https://console.developers.google.com**), предлагаются 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** имеет понятный и удобный API, который преобразует IP-адреса в реальные физические адреса. Вы можете попробовать выполнить простой запрос API, введя в браузере следующее[1]:

```
http://ip-api.com/json/50.78.253.58
```

[1] Этот API преобразует IP-адреса в географические объекты, которые мы еще будем использовать в данной главе.

В результате вы получите примерно такой ответ:

```
{"as": "AS7922 Comcast Cable Communications, LLC","city": "Boston",
"country": "United States","countryCode": "US",
"isp": "Comcast Cable Communications","lat": 42.3584,"lon": -71.0598,
"org": "Boston Park Plaza Hotel","query": "50.78.253.58",
"region": "MA","regionName": "Massachusetts","status": "success",
"timezone": "America/New_York","zip": "02116"}
```

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. Существуют 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 отправки информации, предположительно предназначенной для обработки серверным сценарием. Каждый раз во время авторизации на сайте вы посылаете `POST`-запрос, передавая имя пользователя и (хочется верить) зашифрованный пароль.

Выполняя POST-запрос через API, вы будто 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 нового пользователя и запрос PUT для изменения адреса его электронной почты[1].

Запрос 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, которые 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.

---

[1] 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.

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","username":"Kludgist"}}
```

Всего 73 символа, на целых 36 % меньше, чем те же данные в формате XML.

Конечно, XML можно было бы отформатировать так:

```
<user firstname="ryan" lastname="mitchell" username="Kludgist"></user>
```

Но это считается плохой практикой, 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 в основном серверные сценарии на PHP или .NET. Сейчас вполне может оказаться, что получать и отправлять вызовы API будет фреймворк наподобие Angular или Backbone. Серверным технологиям отчасти неважно, 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 парсингу 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).

Парсинг JSON

Итак, вы рассмотрели различные типы API и их функционирование, а также примеры JSON-ответов, полученных от этих API. Теперь узнаете, как можно парсировать и использовать эту информацию.

В начале главы был рассмотрен пример API **ip-api.com**, который преобразует IP-адреса в физические:

```
http://ip-api.com/json/50.78.253.58
```

Вы можете взять результат этого запроса и использовать функции Python для парсинга 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('countryCode')

print(getCountry('50.78.253.58'))
```

Эта программа выводит код страны для IP-адреса **50.78.253.58**.

Здесь используется библиотека парсинга JSON, которая является частью стандартной библиотеки Python. Чтобы ее подключить, нужно всего лишь добавить строку **import json** в начале файла сценария! 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 объектов словарей, вторая — объект словаря, 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, поэтому 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 пользователю.

По мере распространения фреймворков JavaScript многие из задач по созданию HTML, 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.

Вначале 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 уже загружен, задача, по сути, сводится к парсингу 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 увеличиться до нескольких мегабайтов и дюжины HTTP-запросов.

Кроме того, в случае применения Selenium загружаются все «дополнения», которые необязательно нужны пользователю: вызовы программ отслеживания, реклама на боковых панелях, вызовы программ отслеживания для нее. Также необходимо загрузить изображения, CSS, внешние шрифты. 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 в сто раз больше данных, 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, американский гигант розничной торговли target.com загружает все результаты поиска через JSON. Вы можете найти продукт на их сайте, перейдя по ссылке **https://www.target.com/s?searchTerm=web%20scraping%20with%20python**.

Выполнив веб-скрапинг этой страницы с помощью `urllib` или библиотеки Requests, вы не найдете результатов поиска. Они загружаются отдельно через следующий вызов API:

```
https://redsky.target.com/redsky_aggregations/v1/web/plp_search_v2
```

Поскольку Target API требует передавать ключ с каждым запросом, а срок действия этих ключей ограничен, я рекомендую попробовать выполнить такой запрос самостоятельно и просмотреть полученные результаты в формате JSON.

Конечно, вы можете использовать Selenium для загрузки всех результатов поиска. Но тогда для каждого поиска вам придется сделать около 260 запросов и передать мегабайты данных. Используя API напрямую, вы делаете только один запрос и передаете лишь около 10 Кбайт красиво отформатированных данных — именно тех, которые нужны.

Поиск недокументированных API

В предыдущих главах мы использовали инспектор Chrome для проверки контента HTML-страницы. Теперь применим этот инструмент 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. 15.1.

Обратите внимание: это окно нужно открыть до загрузки страницы. Будучи закрытым, оно не отслеживает сетевые вызовы.

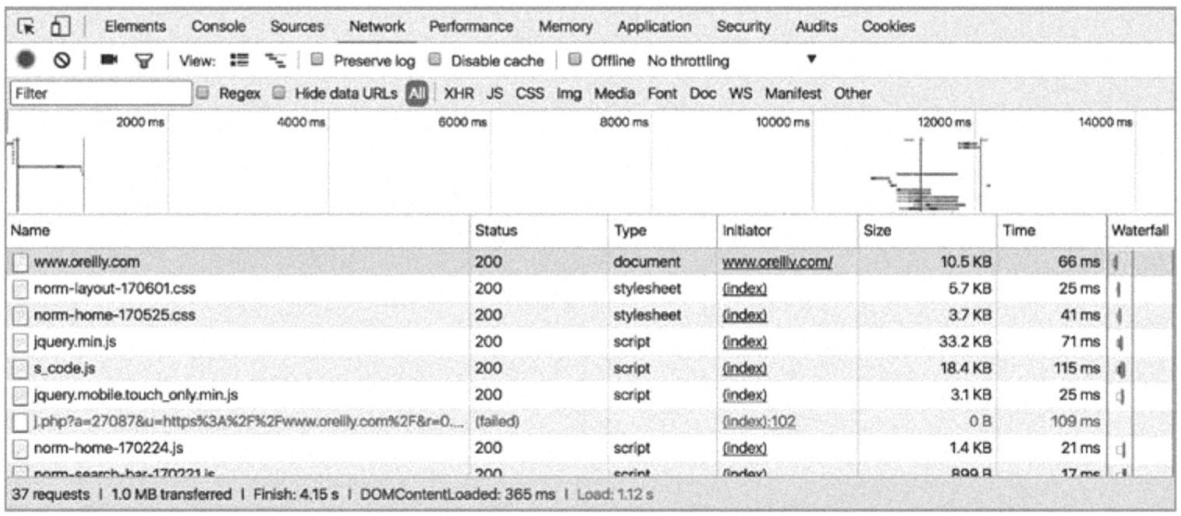


Рис. 15.1. Инструмент Chrome Network Inspector, позволяющий увидеть все вызовы функций, которые браузер делает и получает извне

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 «Документирование недокументированных API» ниже), 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 множеством функций, которые делают сотни вызовов при загрузке одной страницы. Однако со временем, немного потренировавшись, вы будете гораздо легче находить эту метафорическую иголку в стоге сена.

Документирование недокументированных API

После обнаружения вызова API часто бывает полезно хотя бы немного задокументировать 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.

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. 15.2.

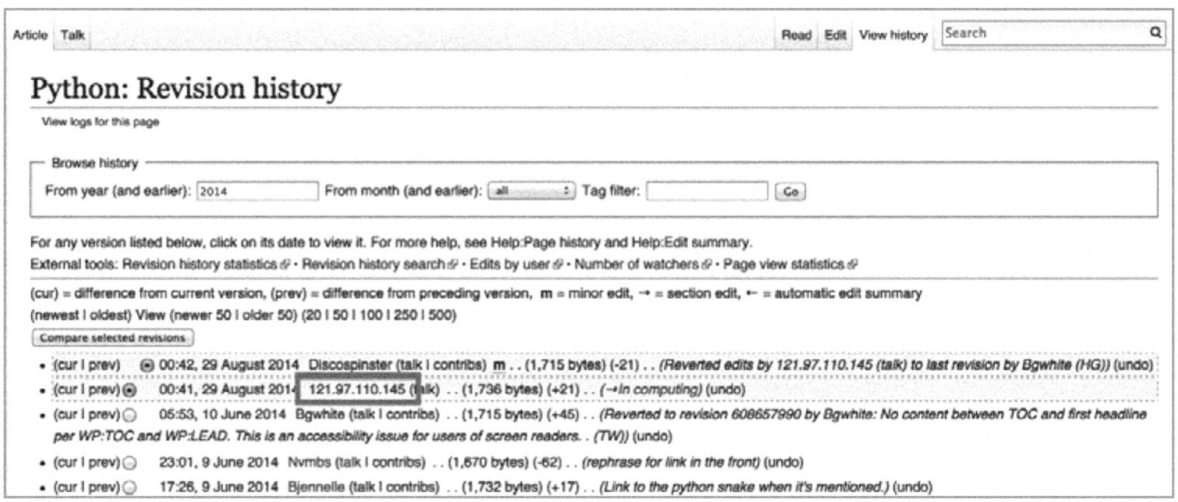


Рис. 15.2. IP-адрес анонимного редактора на странице хронологии правок статьи о Python в «Википедии»

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, использовав 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. 15.3).

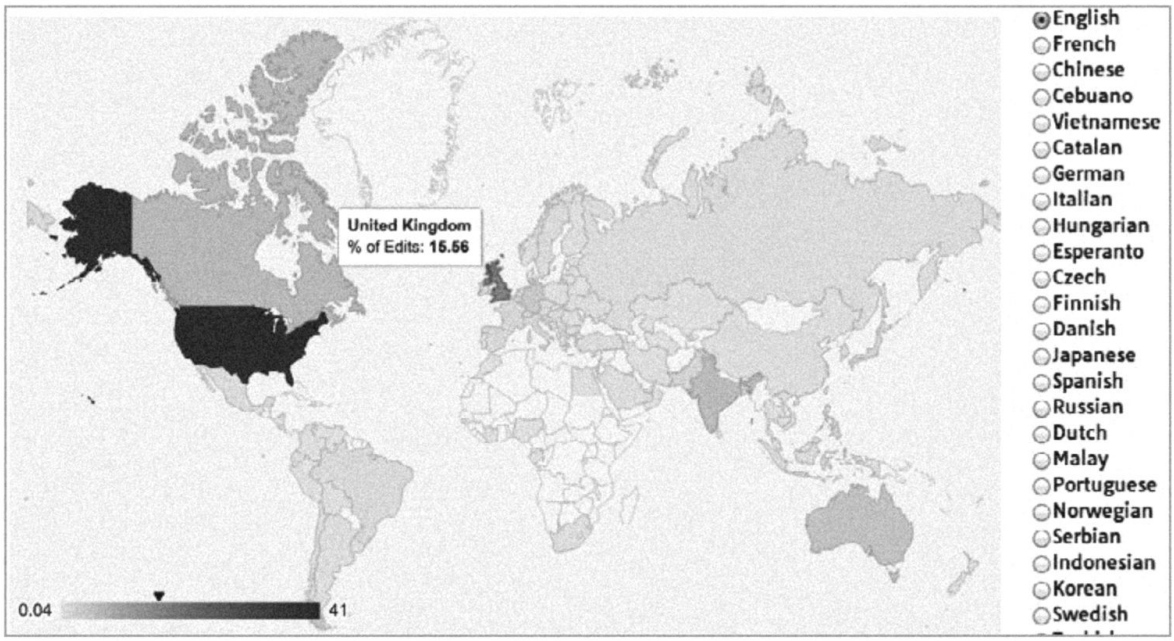


Рис. 15.3. Визуализация правок «Википедии», построенная с помощью библиотеки Google GeoChart

Создать простейший сценарий, который бы сканировал «Википедию», находил страницы хронологии правок, а затем извлекал оттуда IP-адреса, несложно. Представленный ниже сценарий, использующий модифицированный код из главы 6, делает именно это:

```
def getLinks(articleUrl):
    html = urlopen(f'http://en.wikipedia.org{articleUrl}')
    bs = BeautifulSoup(html, 'html.parser')
    return bs.find('div', {'id':'bodyContent'}).findAll('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 = f'http://en.wikipedia.org/w/index.php?title={pageUrl}\
    &action=history'
    print(f'history url is: {historyUrl}')
    bs = BeautifulSoup(urlopen(historyUrl), 'html.parser')
    # Находит только ссылки с классом "mw-anonuserlink",
    # в которых указаны лишь IP-адреса, а не имена пользователей
    ipAddresses = bs.findAll('a', {'class':'mw-anonuserlink'})
    return set([ip.get_text() for ip in ipAddresses])

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 6) 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, вначале 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. Немного изменим код **getCountry**, чтобы учесть недействительные или искаженные IP-адреса, которые приведут к ошибке 404 (страница не найдена):

```
def getCountry(ipAddress):
    try:
        response = urlopen(f'https://ipwho.is/{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:
            print(f'{historyIP} is from {getCountry(historyIP)}')

    newLink = links[random.randint(0, len(links)-1)].attrs['href']
    links = getLinks(newLink)
```

Ниже представлен результат выполнения этой программы:

```
--------------------
history url is: http://en.wikipedia.org/w/index.php?title=Programming_
paradigm&action=history
2405:201:2009:80b0:41bc:366f:a49c:52f2 is from IN
115.186.189.53 is from PK
103.252.145.68 is from IN
2405:201:400b:7058:b128:89fd:5248:f249 is from IN
172.115.220.47 is from US
2806:1016:d:54b6:8950:4501:c00b:507a is from MX
36.255.87.160 is from IN
2603:6011:1100:a1d0:31bd:8a11:a0c8:e4c3 is from US
2806:108e:d:bd2c:a577:db4f:2867:2b5c is from MX
2409:4042:e8f:8d39:b50c:f4ca:91b8:eb9d is from IN
107.190.108.84 is from CA
--------------------
history url is: http://en.wikipedia.org/w/index.php?title=Multi-paradigm_program
ming_language&action=history
98.197.198.46 is from US
75.139.254.117 is from US
```

Дополнительные сведения об API

В данной главе показано несколько способов использования современных API для доступа к данным в Интернете и того, как эти API можно применять 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*[1] Леонарда Ричардсона, Майка Амундсена и Сэма Руби (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

[1] *Амундсен М.* RESTful Web API паттерны и практики. – СПб.: Питер, 2025.

Designing APIs for the Web (O'Reilly) (http://oreil.ly/1GOXNhE), в которой 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 парсинга 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, предназначенные для потребления HTML, 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 сотен типов файлов и форматов данных, 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.

ГЛАВА 16

Обработка изображений и распознавание текста

Машинное зрение — обширнейшая область с далекоидущими целями и последствиями, 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 Интернете.

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 отображается в виде изображения. В зависимости от того, насколько искусно сделано, это 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, учитывается тот факт, что пользователи могут прочитать «секретные» изображения, а большинство ботов — нет. Одни изображения капчи сложнее, другие проще, и данная проблема будет рассмотрена далее в книге.

Но капча — не единственный элемент Интернета, где необходимо помочь вебскраперу 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 шаг — очищает и фильтрует изображения, а *Tesseract* пытается сопоставить фигуры, найденные на этих изображениях, со своей библиотекой известного текста.

В данной главе будут описаны установка и основы использования этих библиотек, а также приведены несколько примеров их совместного применения. Будет также подробно рассмотрено углубленное обучение 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 специальной 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 много другого. Дополнительную информацию о ней см. в документации по Pillow (http://pillow.readthedocs.org/).

Tesseract

Tesseract — это библиотека для OCR. Благодаря поддержке Google (компании, 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 будет рассмотрена как утилита командной строки `Tesseract`, так и ее сторонняя Python-оболочка `pytesseract`. Обе явно различаются 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 был использован в главе 9 для установки MySQL. Например, можно установить Homebrew и применять его для установки Tesseract с помощью следующих двух команд:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/\
HEAD/install.sh)"
$ brew install tesseract
```

Tesseract также можно установить непосредственно из исходных кодов, доступных на странице проекта (https://code.google.com/p/tesseract-ocr/downloads/list).

Для преобразования изображений в текст Tesseract использует модели машинного обучения, обученные на больших наборах данных (или наборах символов). Чтобы просмотреть доступные модели, поставляемые с Tesseract, используйте команду:

```
$ tesseract --list-langs
```

Она выведет содержимое каталога, в котором хранятся модели (`/usr/local/share` в Linux и `/opt/homebrew/share/tessdata/` на Mac), а также доступные модели.

После установки Tesseract можно приступать к установке библиотеки-оболочки `pytesseract`, использующей уже загруженную версию Tesseract для преобразования файлов изображений в строки и объекты, которые затем можно применять в сценариях Python.

Как обычно, установить `pytesseract` можно с помощью pip:

```
$ pip install pytesseract
```

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 `pytesseract` не обнаруживает Tesseract, то узнать каталог установки этой библиотеки можно с помощью команды:

```
$ which tesseract
```

и в сценарии на Python сообщить модулю `pytesseract` путь к этому каталогу, 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 пикселов, заключенные в границах символа):

```
print(pytesseract.image_to_boxes(Image.open('files/test.png')))
```

Эта библиотека также позволяет 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 в Tesseract есть функции, которые она не может использовать, 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 приложений с интенсивными математическими вычислениями. Она хорошо сочетается с Tesseract благодаря способности математически представлять изображения и обрабатывать их в виде больших массивов пикселов.

NumPy можно установить с помощью любого стороннего инсталлятора Python, такого как pip, или скачав пакет (https://pypi.python.org/pypi/numpy) и установив его командой `$ python setup.py install`.

Даже если вы не планируете запускать какие-либо примеры кода с помощью этой библиотеки, я настоятельно рекомендую установить ее и добавить в свой арсенал инструментов для 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 вычисляются стандартное отклонение и среднее арифметическое множества 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» далее в этой главе.

На рис. 16.1 показан идеальный пример хорошо отформатированного текста.



Рис. 16.1. Образец текста, сохраненный в .tiff-файле и предназначенный для чтения с помощью Tesseract

В каталоге *файлы* вы можете запустить Tesseract из командной строки, чтобы прочитать этот файл и записать результаты в текстовый файл:

```
$ tesseract text.png textoutput
$ cat textoutput.txt
```

В итоге получим содержимое вновь созданного файла **textoutput.txt**:

```
This is some text, written in Arial, that will be read by
Tesseract. Here are some symbols: !|@#$%&()
```

Результат в основном точен, за исключением символа вертикальной черты между ! и @. Однако в целом текст получился вполне читаемым.

Если размыть текст изображения, добавить несколько артефактов JPG-сжатия и небольшой градиент фона, то результаты станут намного хуже (рис. 16.2).

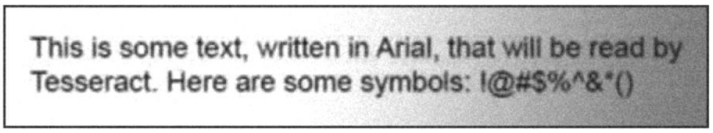


Рис. 16.2. К сожалению, многие документы, которые вам встретятся в Интернете, будут выглядеть приблизительно как этот пример

Вместо записи результатов в файл их можно вывести в терминал, для чего вместо имени файла следует передать символ дефиса (`-`):

```
$ tesseract text_bad.png -
```

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 вам пригодится сценарий на Python, с помощью которого вы сможете сначала очистить изображение. Используя библиотеку Pillow, можно создать пороговый фильтр, который позволит избавиться от серого фона, выделить текст и сделать изображение более четким для последующего чтения в Tesseract.

Кроме того, вместо того чтобы запускать Tesseract из командной строки, можно 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(pytesseract.image_to_string(image))
```

Полученное изображение, автоматически сохраненное в файле `text_cleaned.png`, показано на рис. 16.3.

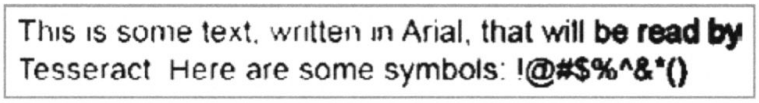


Рис. 16.3. Изображение, полученное после обработки исходного «грязного» изображения пороговым фильтром

За исключением некоторых трудночитаемых или отсутствующих знаков пунктуации, текст становится читаемым, 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 `r` и `i` как одну букву `n`.

Тем не менее этот вариант 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 откорректировать их все вручную не представляется возможным?

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]))
```

В этом сценарии есть две функции.

`cleanFile`

Принимает исходный «плохой» файл и значение порога, с которым запускает пороговый инструмент PIL. Обрабатывает 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 ней), 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  numChars  18
threshold:  85,  confidence:  64.9130434783  numChars  23
threshold:  90,  confidence:  62.2564102564  numChars  39
threshold:  95,  confidence:  64.5135135135  numChars  37
threshold:  100,  confidence:  60.7878787879  numChars  66
threshold:  105,  confidence:  61.9078947368  numChars  76
threshold:  110,  confidence:  64.6329113924  numChars  79
threshold:  115,  confidence:  69.7397260274  numChars  73
threshold:  120,  confidence:  72.9078947368  numChars  76
threshold:  125,  confidence:  73.582278481  numChars  79
threshold:  130,  confidence:  75.6708860759  numChars  79
threshold:  135,  confidence:  76.8292682927  numChars  82
threshold:  140,  confidence:  72.1686746988  numChars  83
threshold:  145,  confidence:  75.5662650602  numChars  83
threshold:  150,  confidence:  77.5443037975  numChars  79
threshold:  155,  confidence:  79.1066666667  numChars  75
threshold:  160,  confidence:  78.4666666667  numChars  75
threshold:  165,  confidence:  80.1428571429  numChars  70
threshold:  170,  confidence:  78.4285714286  numChars  70
threshold:  175,  confidence:  76.3731343284  numChars  67
threshold:  180,  confidence:  76.7575757576  numChars  66
threshold:  185,  confidence:  79.4920634921  numChars  63
threshold:  190,  confidence:  76.0793650794  numChars  63
threshold:  195,  confidence:  70.6153846154  numChars  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 этот результат и вернуть текст, который был распознан с данным порогом как наилучшее предположение того, какой именно текст содержится на 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 на 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 PIL. Кроме того, PIL 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, знай 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.

Хотя 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 сценариев 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 сценарий выполняет именно эту задачу: переходит к крупноформатному изданию повести Л. Толстого «Смерть Ивана Ильича», открывает раздел чтения книги, собирает URL изображений, а затем последовательно скачивает, читает и выводит текст каждого из них.



Выбор испытуемого объекта

При обработке текста, на котором Tesseract не обучался, он намного лучше справляется с крупноформатными изданиями книг, особенно если изображения небольшие. В следующем разделе будет показано, как обучить Tesseract распознавать различные шрифты, чтобы он мог читать шрифты гораздо меньшего размера, в том числе при предварительном просмотре книг небольшого формата!

Обратите внимание: корректная работа этого кода зависит от текущего списка товаров Amazon, а также 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:

```
# Извлекает URL изображения в виде текста
def image_to_text(image):
    urlretrieve(image, 'page.jpg')
    imageList.append(image)
    print(pytesseract.image_to_string(Image.open('page.jpg')))
```

```
# Создаем новый драйвер Selenium
driver = webdriver.Chrome(service=Service(CHROMEDRIVER_PATH))

driver.get(
    'https://www.amazon.com/Death-Ivan-Ilyich-Nikolayevich-Tolstoy/\
dp/1427027277')

# Нажимаем кнопку предварительного просмотра книги
driver.find_element(By.ID, 'litb-canvas-click-wrapper').click()
try:
    # Ждем, пока загрузится кадр iframe
    WebDriverWait(driver, 600).until(
        EC.presence_of_element_located((By.ID, 'litb-read-frame'))
    )
except TimeoutException:
    print('Did not find the iframe')

# Переключаемся на кадр iframe
frame = driver.find_element(By.ID, 'litb-read-frame')
driver.switch_to.frame(frame)

try:
    # Ждем, пока загрузится окно предварительного просмотра
    WebDriverWait(driver, 600).until(
        EC.presence_of_element_located((By.ID, 'kr-renderer'))
    )
except TimeoutException:
    print('Did not find the images')

# Собираем все изображения в элементах div с атрибутом "data-page"
images = driver.find_elements(By.XPATH, '//div[@data-page]/img')
for image in images:
    image_url = image.get_attribute('src')
    image_to_text(image_url)

driver.quit()
```

Теоретически этот сценарий 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 Melvinski trial in the large
building of the Law Courts the members and public
prosecutor met in Ivan Egorovich Shebek's private
room, where the conversation turned on the celebrated
Krasovski case. Fedor Vasilievich warmly maintained
that it was not subject to their jurisdiction, Ivan
Egorovich maintained the contrary, while Peter
```

```
Ivanovich, not having entered into the discussion 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!"[1]
```

Текст, набранный крупным шрифтом и шрифтом без засечек, распознается безупречно. Ошибки правописания 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 на с. 3:

```
it is he who is dead and not 1.[2]
```

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: "vi" 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 12, 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 дополнительное обучение помогает повысить точность. В следующем разделе обсуждается другой подход к решению задачи распознавания искаженного текста, который потребует небольших предварительных затрат времени.

---

[1] Источник:
«Глава 1.
В большом здании судебных учреждений во время перерыва заседания по делу Мельвинских члены и прокурор сошлись в кабинете Ивана Егоровича Шебек, и зашел разговор о знаменитом красовском деле. Федор Васильевич разгорячился, доказывая неподсудность, Иван Егорович стоял на своем, Петр же Иванович, не вступив сначала в спор, не принимал в нем участия и просматривал только что поданные "Ведомости".
— Господа! — сказал он, — Иван Ильич-то умер». — Л. Толстой, «Смерть Ивана Ильича». — *Примеч. пер.*

[2] В русском тексте: «умер он, а не я». — *Примеч. пер.*

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 изложен 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 70 лет мы перешли от использования этих тестов для проверки машин к их применению для тестирования самих себя и получили странные результаты. В 2017 году Google отказался от известной своей сложностью системы reCAPTCHA во многом благодаря ее тенденции блокировать легальных пользователей сайта[1] (см. пример на рис. 16.4). Многие другие компании последовали этому примеру, заменив традиционные текстовые CAPTCHA альтернативными блокировщиками ботов.

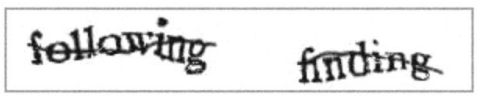


Рис. 16.4. Пример текста из системы Google reCAPTCHA, которая использовалась до 2017 года

Хотя популярность капч несколько упала, они по-прежнему широко используются, особенно на небольших сайтах. Они также полезны в качестве источника образцов текста, трудного для распознавания компьютером. Возможно, ваша цель — не разгадывание капчи, а чтение плохо отсканированных PDF-файлов или рукописных заметок. Принцип здесь тот же.

[1] См. статью: *Jones R.* Google Has Finally Killed the CAPTCHA // Gizmodo. — March 11, 2017. — URL: https://gizmodo.com/google-has-finally-killed-the-captcha-1793190374.

Учитывая это, я создала форму, которая запрещает роботам отправлять ее, требуя решения капчи: **https://pythonscraping.com/humans-only/**. В этом разделе вы научите библиотеку Tesseract, используя конкретные варианты шрифтов и текста, решать эту капчу с высокой надежностью.

Если вы не смогли прочитать это изображение, то подскажу, что на этой капче (рис. 16.5) изображен текст U8DG. Тессеракт, будучи роботом, определенно испытывает трудности с ее решением.



Рис. 16.5. Защита от ботов: https://pythonscraping.com/humans-only/

```
$ tesseract U8DG.png -
u& DS
```

В данном случае Tesseract возвращает пять символов (включая пробел) и правильно распознает только один символ — заглавную букву `D`.

Проблема не в том, что Tesseract плохо читает текст или что эта капча слишком сложна для компьютера, а в том, что этот конкретный рукописный шрифт отличается от обычных шрифтов, на которых обучалась библиотека. К счастью, ее можно научить распознавать дополнительные шрифты, символы и языки.

Обучение библиотеки Tesseract

Независимо от цели обучения — для решения капчи или распознавания любого другого текста — следует учитывать несколько факторов, влияющих на качество работы Tesseract и использовать возможность дополнительного обучения библиотеки.

- Перекрываются ли символы на изображении, то есть можно ли нарисовать вокруг каждого символа правильный прямоугольник, который не будет пересекать другие символы?
- Набран ли текст несколькими шрифтами или используется только один шрифт?

- Присутствуют ли фоновые изображения, линии и другой отвлекающий мусор?
- Имеет ли место высокий контраст с четкими границами между символами и фоном?
- Используется ли достаточно стандартный шрифт с засечками или без засечек или это необычный шрифт со случайными элементами и «завитушками»?

Если в некоторых образцах текста наблюдается перекрытие символов, то можно подумать об использовании только образцов, в которых таких перекрытий нет. Если перекрытия имеют место во всех образцах, то подумайте о возможности предварительной обработки для разделения символов перед обучением.

Скрапинг и подготовка изображений

На этапе предварительной обработки удаляется весь фоновый мусор и улучшаются цветность, контрастность и разделение символов на изображениях.



Сколько изображений потребуется?

Сколько изображений нужно получить? Я рекомендую отобрать с десяток образцов каждого символа или больше, если в вашем тексте много вариаций или случайностей. Tesseract иногда отбраковывает файлы как нечитаемые из-за перекрывающихся символов или по другим загадочным причинам, поэтому вам может понадобиться несколько образцов сверх этого числа. Если вы обнаружите, что результаты распознавания не так хороши, как хотелось бы, или Tesseract спотыкается на определенных символах, то стоит собрать дополнительные обучающие данные и повторить попытку.

Кроме того, если в одном и том же образце текста присутствует несколько вариантов шрифтов (произвольно наклоненный или запутанный текст), то вам может потребоваться больше обучающих данных.

Если шрифт достаточно стандартный и нет других серьезных осложняющих факторов, то попробуйте применить Tesseract без дополнительного обучения! Качество распознавания без обучения может быть вполне приемлемым для ваших нужд, а обучение может оказаться очень трудоемким процессом.

Для обучения необходимо предоставить Tesseract хотя бы несколько примеров каждого символа. Следующий код загружает 100 образцов изображений капчи, каждое из которых содержит четыре символа, всего 400 образцов символов:

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

if not os.path.exists('captchas'):
    os.mkdir('captchas')

for i in range(0, 100):
    bs = BeautifulSoup(urlopen('https://pythonscraping.com/humans-only/'))
    imgUrl = bs.find('img', {'class': 'wpcf7-captchac'})['src']
    urlretrieve(imgUrl, f'captchas/{imgUrl.split("/")[-1]}')
```

После просмотра загруженных обучающих изображений нужно выбрать вид предварительной обработки, если таковая необходима. На загруженных образцах изображен серый текст на черном фоне. Вы можете написать функцию `cleanImage`, которая преобразует их в изображения черного текста на белом фоне и добавляет белую рамку, чтобы отделить символы от края изображения:

```
def cleanImage(imagePath):
    image = Image.open(imagePath)
    image = image.point(lambda x: 255 if x<143 else 0)
    image = ImageOps.expand(image,border=20,fill='white')
    image.save(imagePath)

for filename in os.listdir('captchas'):
    if '.png' in filename:
        cleanImage(f'captchas/{filename}')
```

Создание .box-файлов с помощью проекта обучения Tesseract

Далее нужно на основе этих очищенных изображений создать `.box`-файлы, в каждой строке которых содержатся один символ с изображения и координаты углов прямоугольника, ограничивающего этот символ. Например, для изображения капчи с символами AK6F соответствующий `.box`-файл может иметь такое содержимое:

```
A 32 34 54 58
K 66 32 91 56
6 101 34 117 57
F 135 32 156 57
```

Я создала проект по адресу **https://github.com/REMitchell/tesseract-trainer**, который, среди прочего, содержит веб-приложение, помогающее создавать эти `.box`-файлы. Чтобы воспользоваться им, выполните следующие действия.

1. Переименуйте каждое изображение капчи в соответствующее ему решение. Например, файлу изображения, содержащему AK6F, дайте имя `AK6F.png`.
2. Откройте в браузере файл `createBoxes.html` из веб-приложения Tesseract trainer.
3. Щелкните на ссылке **Add a new file** (Добавить новый файл) и выберите несколько файлов изображений, переименованных на шаге 1.
4. Веб-приложение автоматически создаст прямоугольные рамки вокруг символов на изображении. Откорректируйте размеры и положение этих рамок, как показано на рис. 16.6.
5. Закончив корректировку рамок, нажмите кнопку **Download .box** (Загрузить файл .box), чтобы загрузить `.box`-файл. После этого должно появиться следующее изображение.

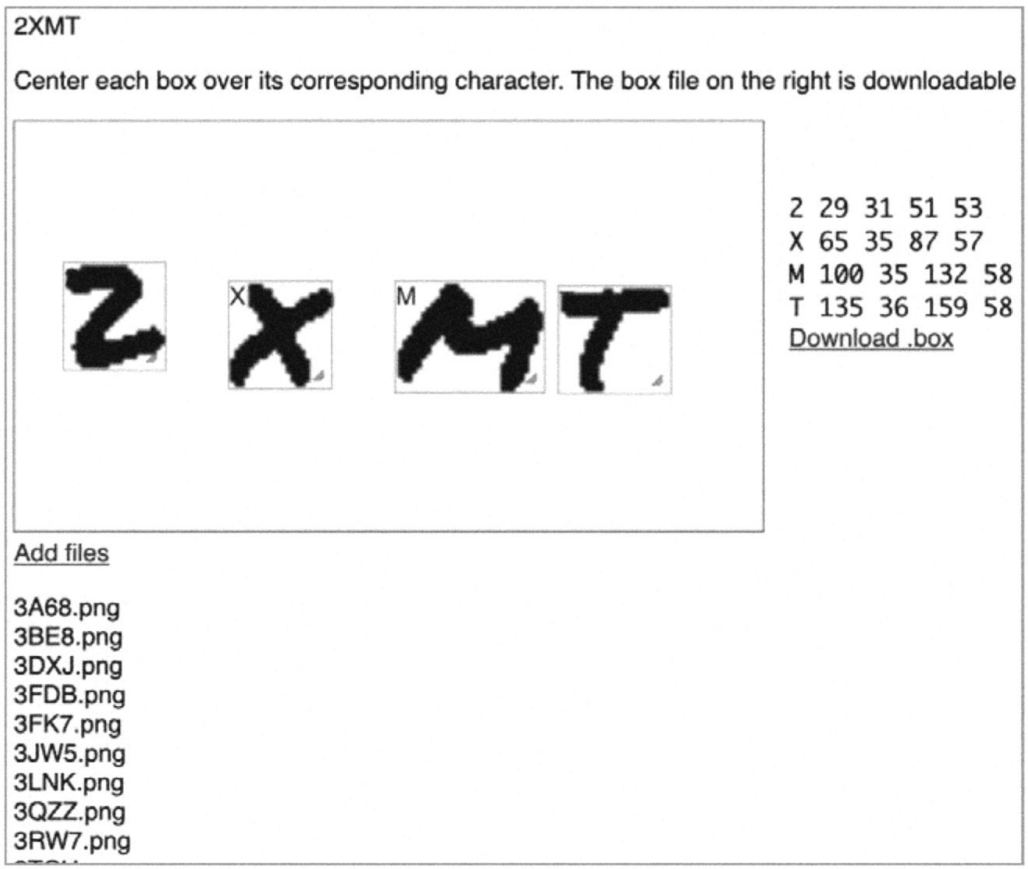


Рис. 16.6. Создание .box-файлов с помощью инструмента Tesseract trainer

Здесь самое время поставить хороший подкаст или фильм, поскольку далее последует пара часов довольно скучной работы. Сколько именно — зависит от количества создаваемых `.box`-файлов.

Следующий шаг — передать библиотеке Tesseract `.box`-файлы, созданные тяжким трудом, и дать ей возможность обучиться. Конечная цель процесса — создать файл *обученных данных*, который можно добавить в языковой каталог Tesseract.

В проект Tesseract trainer, доступный по адресу **https://github.com/REMitchell/tesseract-trainer**, я включила файл `trainer.py`. Этот сценарий ожидает найти каталог `data` в корне проекта с подкаталогами `cleaned` и `box` в нем.

`data`

— `cleaned`

  Содержит обработанные и очищенные изображения капчи в файлах с именами, соответствующими именам `.box`-файлов.

— `box`

  `.box`-файлы, загруженные из веб-приложения.

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 сценарии с этими данными, программа вряд ли что-нибудь удалит, однако лучше подстраховаться, чем потом сожалеть о многих часах, потраченных на создание потерянных `.box`-файлов.

Обучение Tesseract на .box-файлах

Чтобы завершить анализ данных и создать файлы с результатами обучения, необходимыми библиотеке Tesseract, нужно выполнить еще множество шагов. Сценарий `trainer.py` может выполнить их за вас.

Исходные параметры и операции, которые выполняет эта программа, содержатся в методах `__init__` и `runAll` класса:

```
CLEANED_DIR = 'cleaned'
BOX_DIR = 'box'
EXP_DIR = 'exp'
class TesseractTrainer():
    def __init__(self,languageName, fontName, directory='data'):
        self.languageName = languageName
        self.fontName = fontName
        self.directory = directory

    def runAll(self):
        os.chdir(self.directory)
        self.createDirectories()
        self.createFontProperties()
        prefixes = self.renameFiles()
        self.createTrainingFiles(prefixes)
        self.extractUnicode()
        self.runShapeClustering()
        self.runMfTraining()
        self.runCnTraining()
        self.createTessData()
```

В конце сценария `trainer.py` создается новый экземпляр `TesseractTrainer` и вызывается метод `runAll`:

```
trainer = TesseractTrainer('captcha', 'captchaFont')
trainer.runAll()
```

Конструктору `TesseractTrainer` передаются три параметра.

`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. Оно может быть любым, но состоять из одного слова без пробелов. Имя шрифта предназначено только для внутреннего использования во время обучения: вы вряд ли увидите его где-то еще, и вам не понадобится ссылаться на него.

`directory`

Каталог, содержащий подкаталоги с изображениями и `.box`-файлами. По умолчанию используется имя `data`. Если вы создаете несколько проектов, то можете в каждом предусмотреть передачу уникального имени каталога, чтобы не допустить беспорядка.

Рассмотрим некоторые из используемых здесь методов.

`createDirectories` создает подкаталоги, такие как `exp`, в которых позже будут храниться файлы обученных данных.

Функция `createFontProperties` создает необходимый файл `font_properties`, в который Tesseract заносит информацию о новом созданном шрифте:

```
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 этим свойствам — интересное упражнение, но, к сожалению, выходит за рамки данной книги.

`renameFiles` присваивает всем `.box`-файлам и соответствующим файлам изображений имена, необходимые для обработки в Tesseract (здесь числа `<fileNumber>` в именах файлов — это последовательные номера, позволяющие отличать файлы друг от друга):

- `<languageName>.<fontName>.exp<fileNumber>.box`;
- `<languageName>.<fontName>.exp<fileNumber>.tiff`.

`extractUnicode` просматривает все созданные `.box`-файлы и формирует общее множество символов, доступных для обучения. Получившийся файл в формате Unicode сообщит нам, сколько разных символов было найдено. При случае это поможет быстро определить, упустили мы что-нибудь или нет.

Следующие три функции, `runShapeClustering`, `runMfTraining` и `runCtTraining`, создают файлы `shapetable`, `pfftable` 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 с обученными данными `eng.traineddata`.

Использование обученных данных с Tesseract

Файл `traineddata` — главный результат всего процесса обучения. Этот файл сообщает библиотеке Tesseract, как идентифицировать символы в наборе обучающих данных. Чтобы использовать этот файл, его нужно переместить в корневой каталог `tessdata`.

Найти этот каталог можно с помощью команды:

```
$ tesseract --list-langs
```

Она выведет что-то вроде:

```
List of available languages in "/opt/homebrew/share/tessdata/" (3):
eng
osd
snum
```

Затем определите переменную окружения `TESSDATA_PREFIX` и запишите в нее путь к этому каталогу:

```
$ export TESSDATA_PREFIX=/opt/homebrew/share/tessdata/
```

Наконец, переместите новый файл `traineddata` в каталог `languages`:

```
$ cp data/exp/cap.traineddata $TESSDATA_PREFIX/cap.traineddata
```

После создания нового файла `traineddata` библиотека Tesseract должна автоматически распознать его как новый язык и уметь решать новые капчи, которые он содержит:

```
$ tesseract -l captcha U8DG.png -

U8DG
```

Это успех! Значительно лучше, по сравнению с предыдущей интерпретацией изображения как `u& DS`.

Это был лишь краткий обзор всех возможностей обучения и распознавания шрифтов с помощью Tesseract. Если вы заинтересованы в более углубленном 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://github.com/tesseract-ocr/tesseract**.

Получение капчи и отправка решений

Многие популярные системы управления контентом часто получают спам от ботов, которые предварительно запрограммированы на известное расположение страниц регистрации пользователей. Например, на сайте **http://pythonscraping.com** даже капча (по общему мнению, слабая) мало помогает сдержать наплыв регистрационных данных.

Как же ботам это удается? 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 13, то хотя бы бегло просмотрите ее.

Большинство изображений капчи обладают следующими свойствами.

- Это динамически генерируемые изображения, создаваемые программой на сервере. Их источники могут быть не похожими на обычные изображения и иметь вид наподобие `<img src="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 привести к неудаче.

Обычный подход заключается в том, чтобы скачать файл с изображением капчи на локальный диск, очистить это изображение, использовать Tesseract для его парсинга и вернуть решение в виде соответствующего параметра формы.

Я создала страницу с формой комментариев, защищенной капчей, по адресу **http://pythonscraping.com/humans-only**, 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.

Для начала загрузим страницу и найдем местоположение скрытого токена, который необходимо послать с остальными данными формы:

```
html = urlopen('http://www.pythonscraping.com/humans-only')
bs = BeautifulSoup(html, 'html.parser')
# Собираем заполненные перед этим значения формы
hiddenToken = bs.find(
    'input',
    {'name':'_wpcf7_captcha_challenge_captcha-170'}
)['value']
```

Этот скрытый токен также является именем файла изображения капчи на странице, что упрощает написание функции **getCaptchaSolution**:

```
def getCaptchaSolution(hiddenToken):
    imageLocation = f'https://pythonscraping.com/wp-content/uploads/wpcf7_
captcha/{hiddenToken}.png'
    urlretrieve(imageLocation, 'captcha.png')
    cleanImage('captcha.png')
    p = subprocess.Popen(
        ['tesseract','-l', 'captcha', 'captcha.png', 'output'],
        stdout=subprocess.PIPE,stderr=subprocess.PIPE
    )
    p.wait()
    f = open('output.txt', 'r')

    # Удалить пробельные символы
    captchaResponse = f.read().replace(' ', '').replace('\n', '')
    print('Captcha solution attempt: '+captchaResponse)
    return captchaResponse
```

Обратите внимание: этот сценарий завершается ошибкой в двух случаях: если Tesseract не извлекла из изображения ровно четыре символа (поскольку мы знаем, что все допустимые решения для данной капчи должны включать четыре символа) или же если форма была отправлена, но ответ на капчу был неверным.

В первом случае можно перезагрузить страницу и повторить попытку, скорее всего, без каких-либо санкций со стороны веб-сервера. Во втором случае сервер может заметить, что вы неправильно решаете капчу, и наказать вас. Многие серверы после нескольких неудачных попыток блокируют пользователя или подвергают его более строгой проверке.

Как владелец этого конкретного сервера могу засвидетельствовать, что он чрезвычайно снисходителен и вряд ли вас заблокирует!

Сами данные формы относительно объемны, и их можно просмотреть в репозитории GitHub или в инструментах разработчика браузера, выполнив отправку формы. Однако проверить длину решения капчи и отправить его с помощью библиотеки Requests довольно просто:

```
if len(captcha_solution) == 4:
    formSubmissionUrl = 'https://pythonscraping.com/wp-json/contact-form-7/v1/
      contact-forms/93/feedback'
    headers = {'Content-Type': 'multipart/
      form-data;boundary=----WebKitFormBoundaryBFvsPGsghJe0Esco'}
    r = requests.post(formSubmissionUrl, data=form_data, headers=headers)
    print(r.text)
else:
    print('There was a problem reading the CAPTCHA correctly!')
```

Если решение капчи было правильным (обычно так и есть), вы должны увидеть примерно такой ответ:

```
Captcha solution attempt: X9SU
{"contact_form_id":93,"status":"mail_sent","message":
"Thank you for your message. It has been sent.",
"posted_data_hash":"2bc8d1e0345bbfc281eac0410fc7b80d",
"into":"#wpcf7-f93-o1","invalid_fields":[],"captcha":
{"captcha-170":
"https:\/\/pythonscraping.com\/wp-content\/uploads
\/wpcf7_captcha\/3551342528.png"}}
```

В настоящее время капча не так распространена, как 10 или 20 лет назад, но она все еще используется на многих сайтах, поэтому важно знать, как с ней обращаться. Кроме того, навыки, полученные при реализации решения капчи, пригодятся вам и в других сценариях, где требуется преобразовывать изображения в текст.

ГЛАВА 17

Как избежать ловушек веб-скрапинга

Ничто так не огорчает, как после выполнения веб-скрапинга 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 (сценарий, 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 или данными трассировки стека, которыми можно было бы воспользоваться. Вас идентифицировали как бота, 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 сценарии 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 данную главу. Возможно, однажды 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 тому, как сделать боты еще лучше?

Я считаю, что эту главу важно включить в книгу по следующим причинам.

- Существуют в высшей степени этические, юридически обоснованные причины для веб-скрапинга некоторых сайтов, не желающих, чтобы их обработал веб-скрапер. Моей предыдущей задачей по веб-скрапингу был автоматический сбор информации с сайтов, которые публиковали в Интернете имена, адреса, номера телефонов и другие личные данные клиентов без их согласия. Я использовала собранную информацию, чтобы делать официальные запросы на эти сайты с требованием удалить данные сведения. Во избежание конкуренции сайты бдительно охраняют свою информацию от веб-скраперов. Тем не менее моя работа по обеспечению анонимности клиентов компании (некоторые из них подвергались преследованиям, или являлись жертвами домашнего насилия, или имели 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, необходимые 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 выполняют широкое сканирование информации и уязвимостей, так что использование всего нескольких простых методов, описанных в данной главе, скорее всего, станет помехой для 99 % из них. Тем не менее такие боты с каждым месяцем становятся все более изощренными, и лучше быть готовым.
- 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 сценариев 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 2). Но я не собираюсь тыкать вас в это носом всякий раз, когда мы будем обсуждать новую методику. Итак, отложим данный вопрос до конца главы или, как однажды философски заметил Форрест Гамп: «Это все, что я хотел сказать о Вьетнаме».

Выдать сценарий за человека

Фундаментальная проблема 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 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 используются. Однако следующие семь полей постоянно применяются в основных браузерах при инициализации любого соединения (ниже показан пример данных из моего браузера):

```
Host             https://www.google.com/
Connection       keep-alive
Accept           text/html,application/xhtml+xml,application/xml;q=0.9,image/
                 avif,image/webp,image/apng,*/*;q=0.8,application/
                 signed-exchange;v=b3;q=0.7
User-Agent       Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/
                 537.36 (KHTML, like Gecko) Chrome/113.0.0.0 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/3.9
```

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 сценарий:

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()

headers = {'User-Agent':'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
           AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0
Safari/537.36',
           'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,
            image/avif,image/webp,image/apng,*/*;q=0.8,application/
            signed-exchange;v=b3;q=0.7'}
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 сценария должен показать, что теперь заголовки соответствуют тем, которые заданы в коде, в словарном объекте headers.

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 приветствие Bonjour, если сайт поддерживает перевод (практически беспроигрышным 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
```

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/3.9`. 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.

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, отслеживающие ваши перемещения с помощью данных cookie, могут попытаться заблокировать 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 20), однако если данные cookie будут полностью выдавать вас, то все попытки маскировки окажутся тщетными.

Данные cookie также могут оказаться необходимыми для веб-скрапинга сайта. Как показано в главе 13, 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 модулей — расширение EditThisCookie для Chrome (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» главы 13. 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 сценариев 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 установка и основы применения были рассмотрены в главе 14).

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 данных cookie:

```
[{'domain': '.pythonscraping.com', 'expiry': 1722996491, 'httpOnly': False,
'name': '_ga', 'path': '/', 'sameSite': 'Lax', 'secure': False, 'value':
'GA1.1.285394841.1688436491'}, {'domain': '.pythonscraping.com', 'expiry':
1722996491, 'httpOnly': False, 'name': '_ga_G60J5CGY1N', 'path': '/',
'sameSite': 'Lax', 'secure': False, 'value':
'GS1.1.1688436491.1.0.1688436491.0.0.0'}]
```

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 эти функции будут работать вместе:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

chrome_options = Options()
chrome_options.add_argument("--headless")
driver = webdriver.Chrome(
    service=Service(CHROMEDRIVER_PATH),
    options=chrome_options
)

driver.get('http://pythonscraping.com')
driver.implicitly_wait(1)

savedCookies = driver.get_cookies()
print(savedCookies)

driver2 = webdriver.Chrome(
    service=Service(CHROMEDRIVER_PATH),
    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 страницу с сайта, выводит полученные данные cookie, а затем сохраняет их в переменной `savedCookies`. Второй веб-драйвер открывает ту же страницу, удаляет полученные им данные cookie и добавляет данные, полученные первым веб-драйвером.

Обратите внимание: второй веб-драйвер, прежде чем добавить данные cookie, должен сначала загрузить сайт. Благодаря этому шагу Selenium узнает, 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.

Создание цифровых отпечатков TLS

В начале 2000-х годов многие крупные технологические компании любили в ходе собеседований загадывать загадки потенциальным программистам. Сейчас это уже вышло из моды, потому что менеджеры по найму поняли две вещи: кандидаты делились решениями загадок и запоминали их, способность разгадывать загадки плохо коррелирует с продуктивностью человека.

Тем не менее одна из этих классических загадок на собеседовании по-прежнему имеет ценность как метафора протокола безопасности транспортного уровня (Transport Layer Security, TLS). Она описывается далее.

Вам нужно отправить другу совершенно секретное сообщение по опасному маршруту, где любые незапертые ящики с сообщениями перехватываются шпионами (но ящики, запертые на висячие замки, безопасны, если у шпионов нет ключей). Вы помещаете сообщение в ящик, который можно запереть несколькими замками. У вас есть навесные замки с соответствующими ключами, а у вашего друга есть свои замки с соответствующими ключами, но ни один из ключей вашего друга не подходит к вашим замкам, и наоборот. Как гарантировать, что ваш друг сможет открыть ящик на своей стороне и безопасно получить сообщение?

Обратите внимание, что пересылка ключа, открывающего ваш замок, даже в виде отдельной посылки невозможна. Шпионы перехватят эти посылки, сделают копии

ключей и сохранят их для дальнейшего использования. Кроме того, доставка ключа отдельной посылкой не гарантирует надежности (хотя здесь несколько нарушается стройность «загадки как метафоры»), потому что шпионы могут сделать копии *самого ящика* и, если ключ будет отправлен позже, с его помощью открыть свою копию ящика.

Вот одно из возможных решений: вы навешиваете замок на ящик и отправляете его другу. Ваш друг получает запертый ящик, вешает на него свой замок (теперь ящик заперт двумя замками) и отправляет его обратно. Вы снимаете свой замок и снова отправляете ящик своему другу, оставляя его замок нетронутым. Ваш друг получает ящик и открывает его.

По сути, именно так устанавливается безопасная связь в ненадежной сети. По защищенному протоколу связи, например HTTPS, все сообщения шифруются и расшифровываются с помощью ключа. Если злоумышленник получит ключ (представленный в загадке отдельной посылкой), он сможет прочитать любые отправляемые сообщения.

Так как же отправить своему другу ключ, который вы собираетесь использовать для шифрования и дешифрования будущих сообщений, чтобы этот ключ не был перехвачен и использован злоумышленниками? Зашифруйте его своим «замком», отправьте другу, тот добавит свой «замок» и вернет сообщение обратно, вы удалите свой «замок» и снова вернете сообщение другу, который теперь сможет «открыть» его. Такой способ обеспечивает безопасную передачу ключа.

Весь этот процесс «запирания», отправки, добавления еще одного «замка» и т. д. обрабатывается протоколом безопасности транспортного уровня TLS и называется *рукопожатием TLS*.

Помимо установки взаимно известного ключа, или *мастер-секрета*, во время рукопожатия настраивается много других параметров:

- выбирается наиболее высокая версия протокола TLS, поддерживаемая обеими сторонами (которая будет использоваться во время остальной части процедуры рукопожатия);
- определяется библиотека шифрования;
- выбирается метод сжатия;
- осуществляется идентификация сервера с использованием его общедоступного сертификата;
- проверяются работоспособность мастер-секрета на обеих сторонах и безопасность связи.

Эта процедура рукопожатия TLS выполняется каждый раз при подключении к новому веб-серверу, а также в том случае, когда создается новый HTTP-сеанс

с этим веб-сервером (дополнительную информацию о сеансах см. в главе 1). Точные сообщения, отправляемые вашим компьютером в процессе выполнения процедуры рукопожатия TLS, определяются приложением, устанавливающим соединение. Например, Chrome может поддерживать немного другие версии TLS или библиотеки шифрования, чем Microsoft Edge, поэтому сообщения, отправляемые им при рукопожатии TLS, будут отличаться.

Процедура рукопожатия TLS довольно продолжительная, а параметров, участвующих в согласовании, слишком много, и, как результат, умные администраторы серверов заметили, что сообщения, отправленные клиентами во время рукопожатия TLS, в некоторой степени уникальны для каждого приложения. Сообщения образуют своего рода *цифровые отпечатки TLS*, помогающие понять, были сообщения созданы Chrome, Microsoft Edge, Safari или библиотекой Requests языка Python.

Просмотреть информацию, сгенерированную во время рукопожатия TLS, можно, посетив **https://tools.scrapfly.io/api/fp/ja3?extended=1**. Чтобы сделать цифровые отпечатки TLS более управляемыми и простыми для сравнения, часто используется метод хеширования JA3, результаты которого можно видеть в ответе этого API. Цифровые отпечатки, хешированные методом JA3, каталогизируются в больших базах данных и используются для поиска, когда позже возникает необходимость идентифицировать приложение.

Отпечаток TLS немного похож на файл cookie `User-Agent`, поскольку представляет собой длинную строку, идентифицирующую приложение, которое вы используете для отправки данных. Но, в отличие от файла cookie, `User-Agent` отпечаток нелегко изменить. В Python операции по протоколу TLS контролируются библиотекой SSL (**https://github.com/python/cpython/blob/3.11/Lib/ssl.py**). Теоретически можно переписать библиотеку SSL. Упорным и кропотливым трудом можно изменить цифровой отпечаток TLS, который отправляет Python, настолько, чтобы сделать хеш JA3 нераспознаваемым для серверов, пытающихся заблокировать ботов на Python. Приложив еще больше усилий и усердия, можно выдать себя за безобидный браузер! Некоторые проекты, такие как **https://github.com/lwthiker/curl-impersonate**, стремятся добиться именно этого.

К сожалению, природа цифровых отпечатков TLS такова, что любые библиотеки, реализующие их подмену, требуют частого обслуживания добровольцами и склонны к быстрой деградации. Пока эти проекты не приобрели большую популярность и надежность, можно воспользоваться гораздо более простым способом обхода проверки цифровых отпечатков TLS и блокировки: Selenium.

На протяжении всей этой книги я предостерегала от использования автоматизации браузера для решения задач, если существуют альтернативные решения. Браузеры потребляют много памяти, часто загружают ненужные страницы и требуют дополнительных зависимостей, обеспечивающих работу вашего веб-краулера. Но когда дело доходит до снятия цифровых отпечатков TLS, имеет смысл использовать браузер и избежать ненужной головной боли.

Имейте в виду, что ваш цифровой отпечаток TLS будет одинаковым, независимо от того, используете вы версию браузера в обычном режиме или в режиме консоли. Так что смело отключайте графику и загружайте только те данные, которые вам нужны, — целевой сервер не узнает об этом (по крайней мере, на основе ваших данных TLS)!

Своевременность — наше всё

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, когда в одном потоке обрабатываются данные, а в другом — многократно загружаются страницы. Однако это плохой 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 добывать 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 для безопасности и вычислительных

ресурсов, если уязвимы для произвольного использования ботами. Поэтому владельцы сайтов в высшей степени заинтересованы в ограничении доступа 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 16, посвященной капче и обработке изображений, а также в главе 13, 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. 17.1 показан пример применения таких скрытых полей на странице входа в LinkedIn. 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.

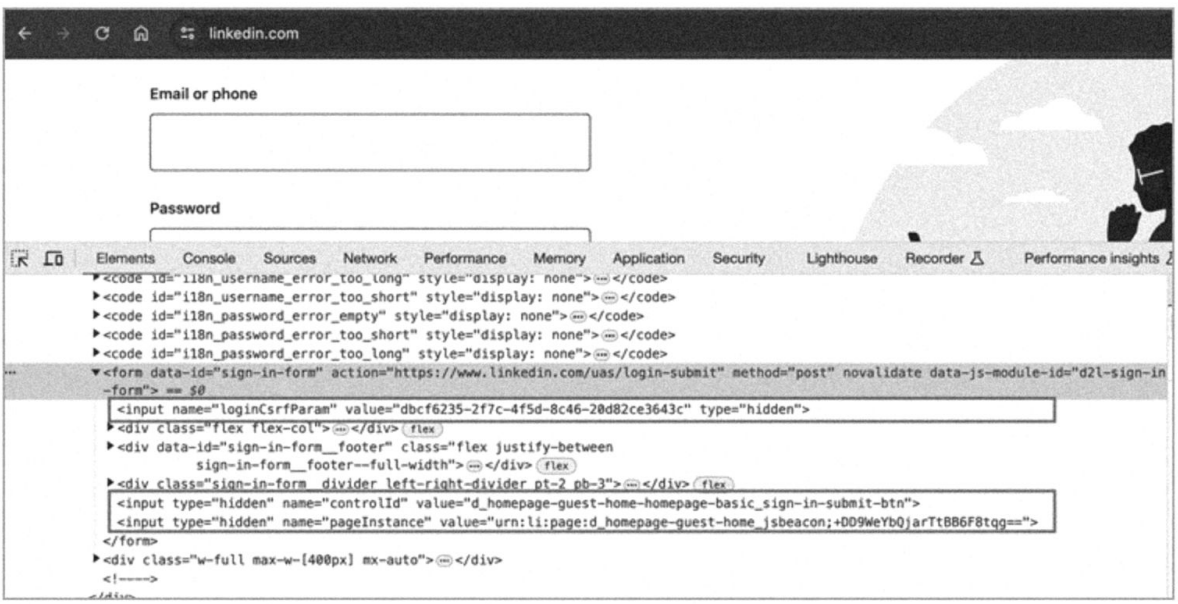


Рис. 17.1. Форма аутентификации в LinkedIn, содержащая несколько скрытых полей

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 форма не была отправлена естественным путем со страницы формы, а передана 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.

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.

Короче говоря, 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, также может проверять, были эти переменные сгенерированы недавно или нет, с целью гарантировать однократное использование переменных формы (чтобы не позволить веб-скраперу просто сохранять эти значения в сценарии и время от времени снова их использовать). Либо и то и другое.

Как избежать ловушек

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 сценарий, который заблокирует IP-адрес пользователя, аннулирует его аутентификацию на сайте или примет какие-либо другие меры, 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 пикселов 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, визуально присутствующие 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
from selenium.webdriver.common.by import By

driver = webdriver.Chrome(service=Service(CHROMEDRIVER_PATH))

driver.get('http://pythonscraping.com/pages/itsatrap.html')
links = driver.find_elements(By.TAG_NAME, 'a')
for link in links:
    if not link.is_displayed():
        print(f'The link {link.get_attribute("href")} is a trap')

fields = driver.find_elements(By.TAG_NAME, 'input')
for field in fields:
    if not field.is_displayed():
        print(f'Do not change value of {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 осторожным при взаимодействии с ними.

Чек-лист: 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 сам на себя и как можно больше — на человека. Если, несмотря на это, вас все равно блокируют сайты и вы не знаете почему, то вот чек-лист, 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 сценарий JavaScript. Обратитесь к главе 14.

- 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 удается 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 20). 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 данной главе в подразделе «Как избежать ловушек» раздела «Основные средства защиты форм»).
  - 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@<имя домена>` или `admin@<имя домена>`. Администраторы тоже люди, и вы будете удивлены их готовности делиться данными.

ГЛАВА 18

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 интерфейсы сайтов часто остаются за пределами этих автоматических тестов, хотя именно интерфейсы обычно являются 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 модульные тесты для разделов JavaScript, но они 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 сценария.

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 модульных 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 модульные тесты

Термины *«тесты»* и *«модульные тесты»* нередко считают взаимозаменяемыми. Часто, когда программисты говорят о написании тестов, они действительно имеют в виду написание модульных тестов. Но есть и такие, которые, говоря о написании модульных тестов, в действительности пишут какие-то другие.

Определения и методы модульного тестирования часто меняются от компании к компании, однако модульное тестирование обычно характеризуется следующими общими свойствами.

- Каждый модульный 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.

  Модульные тесты, проверяющие какой-то один компонент, часто объединяются в один класс. Например, после теста на отрицательное значение суммы в долларах, снятых с банковского счета, может идти модульный тест, проверяющий поведение банковского счета при попытке снять сумму, превышающую остаток.

- Каждый модульный тест может проводиться совершенно независимо от других. Все настройки или отмены настройки, требуемые для модульного теста, должен совершать он сам. Аналогично модульные тесты не должны влиять на успешное или неудачное прохождение других тестов и должны иметь возможность успешно выполняться в любой последовательности.
- Каждый модульный тест обычно содержит хотя бы одно *утверждение*. Например, модульный тест может утверждать, что 2 + 2 равно 4. Иногда такой тест может содержать только состояние неудачи. Например, может завершиться неудачно при выдаче исключения, но выполниться по умолчанию, если все идет хорошо.
- Модульные 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 модульному 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 модульного тестирования, о которых вы узнаете в следующем разделе.

Модуль unittest

Библиотека модульного тестирования `unittest` входит в состав стандартной библиотеки Python. Достаточно импортировать и расширить класс `unittest.TestCase` — и у вас появятся следующие возможности:

- функции `setUp` и `tearDown`, которые выполняются до и после каждого модульного теста;
- несколько типов операторов утверждений, описывающих условия прохождения или непрохождения тестов;
- возможность выполнять любые функции, имена которых начинаются с `test_`, как модульные тесты и игнорировать функции, не объявленные тестами.

Вот пример простого модульного 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 означает, что тест пройден успешно и 2 + 2 действительно равно 4.

ВЫПОЛНЕНИЕ UNITTEST В JUPYTER NOTEBOOK

Все сценарии модульных 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 модульный тест, используя расширение класса `unittest.TestCase`, непосредственно из командной строки.

В Jupyter Notebook все немного иначе. Параметры `argv`, создаваемые Jupyter, могут вызывать ошибки в модульных 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 модульные 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 `unnittest.main` игнорирует. Таким образом предотвращается еще и завершение работы `unittest` после выполнения теста.

Строка `%reset` нужна, чтобы освободить память и уничтожить все переменные, созданные пользователем в Jupyter Notebook. Без нее каждый модульный 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 модульный 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 (Ввод).

Тестирование «Википедии»

Чтобы протестировать клиентский интерфейс вашего сайта (за исключением сценариев JavaScript, о которых мы расскажем 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 6, сравнительно легко создать веб-краулер, который итеративно перемещался бы по всем страницам сайта. Что произойдет, если объединить веб-краулер с модульным 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.assertEquals(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 6
        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.attrs['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 выполняют 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.

Во-вторых, хотя `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 логического утверждения:

```
======================================================================
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.assertEquals(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 сценариев 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 модульные тесты на Python и Selenium, безусловно, написаны на одном языке, в их синтаксисе на удивление мало общего. Selenium не требует представления модульных 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 модульные 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 смогла воспользоваться 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 разметку формы и отправил бы 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 и другие проблемы, которые поставили бы в тупик обычных посетителей сайта.

Главное условие в тестах подобного рода — концепция elements в Selenium. Об этом объекте уже упоминалось в главе 14, 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 *цепочки действий*, сохранять их и выполнять в программе один или несколько раз. Эти цепочки 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 13). 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')

### МЕТОД 1 ###
#firstnameField.send_keys('Ryan')
lastnameField.send_keys('Mitchell')
submitButton.click()
#################

### МЕТОД 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 цепочка действий, в которой после вызова метода `perform` последовательно производится нажатие каждого поля формы, после чего туда вводится текст. Результат выполнения сценария 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 немало 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*, 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/chromedriver', options=chrome_options)
        url = 'http://pythonscraping.com/pages/javascript/draggableDemo.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('message').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 будет актуальна.

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 парсинга умения различать изображения котенка и коровы); вместо этого они часто используют 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 модульных тестов, и вам не придется самолично нажимать PrtScn:

```
driver = webdriver.Chrome()
driver.get('http://www.pythonscraping.com/')
driver.get_screenshot_as_file('tmp/pythonscraping.png')
```

Этот сценарий 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.

ГЛАВА 19

Параллельный веб-краулинг

Веб-краулинг выполняется быстро. По крайней мере гораздо быстрее, чем это сможет сделать дюжина стажеров, которые будут 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 иногда 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

Потоки и процессы не являются понятиями, специфическими для Python. Детали их реализации отличаются в разных операционных системах (и зависят от них), но в информатике принято считать, что процессы крупнее и имеют собственную память, тогда как потоки меньше и используют память внутри содержащего их процесса.

Обычно, запуская простую программу на Python, вы запускаете ее в отдельном процессе, содержащем один поток. Но Python поддерживает как многопроцессность, так и многопоточность. Обе имеют одну и ту же цель: одновременное решение двух задач программирования вместо выполнения программы более традиционным линейным способом.

Однако, выбирая между ними, следует знать плюсы и минусы каждого из них. Например, каждый процесс имеет собственную память, выделенную операционной системой. Это означает, что разные процессы не имеют доступа к памяти друг друга. Потоки 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 следующем примере показано использование нескольких потоков для выполнения задачи:

```
import threading
import time

def print_time(threadName, delay, iterations):
    start = int(time.time())
    for i in range(0,iterations):
        time.sleep(delay)
        print (f'{int(time.time() - start)} - {threadName}')

threads = [
    threading.Thread(target=print_time, args=('Fizz', 3, 33)),
    threading.Thread(target=print_time, args=('Buzz', 5, 20)),
    threading.Thread(target=print_time, args=('Counter', 1, 100))
]

[t.start() for t in threads]
[t.join() for t in threads]
```

Это вариант 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
```

Сценарий запускает три потока, один из которых каждые три секунды выводит слово "Fizz", другой каждые пять секунд — "Buzz", а третий каждую секунду — слово "Counter".

Вместо того чтобы выводить слова "Fizz" и "Buzz", можно выполнять в потоках какие-либо полезные действия, например сканировать сайт:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import random
import threading
import time

# Рекурсивно ищет ссылки на странице «Википедии»,
# затем переходит по случайной ссылке с искусственной задержкой 5 с
def scrape_article(thread_name, path):
    time.sleep(5)
    print(f'{thread_name}: Scraping {path}')
    html = urlopen('http://en.wikipedia.org{}'.format(path))
    bs = BeautifulSoup(html, 'html.parser')
    title = bs.find('h1').get_text()
    links = bs.find('div', {'id':'bodyContent'}).find_all('a',
        href=re.compile('^(/wiki/)((?!:).)*$'))
    if len(links) > 0:
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        scrape_article(thread_name, newArticle)

threads = [
    threading.Thread(
        target=scrape_article,
        args=('Thread 1', '/wiki/Kevin_Bacon',)
    ),
    threading.Thread(
        target=scrape_article,
        args=('Thread 2', '/wiki/Monty_Python',)
    ),
]

[t.start() for t in threads]
[t.join() for t in threads]
```

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 сценарий 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)
    ...
    links = get_links(thread_name, bs)
    ...
```

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 проходит одну и ту же страницу, возможно, и не стоит переписывать код.

Состояния гонки и очереди

Можно было 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 характерно для 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 потоко-небезопасных 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 вошел — первым вышел» (First In First Out, FIFO), либо по принципу «последним вошел — первым вышел» (Last In First Out, LIFO). Очередь получает сообщения из любого потока с помощью функции `queue.put('My message')` 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 его 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.

```
def storage(queue):
    conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
        user='root', passwd='password', db='mysql', charset='utf8')
    cur = conn.cursor()
    cur.execute('USE wikipedia')
    while 1:
        if not queue.empty():
            path = queue.get()
            cur.execute('SELECT * FROM pages WHERE url = %s', (path))
            if cur.rowcount == 0:
                print(f'Storing article {path}')
                cur.execute('INSERT INTO pages (url) VALUES (%s)', (path))
                conn.commit()
            else:
                print("Article already exists: {}".format(path))

visited = set()
```

```
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/)((?!:).)*$')
    )
    links = [link.get('href') for link in links]
    return [link for link in links if link and link not in visited ]

def scrape_article(thread_name, path):
    time.sleep(5)
    visited.add(path)
    print(f'{thread_name}: Scraping {path}')
    bs = BeautifulSoup(
        urlopen('http://en.wikipedia.org{}'.format(path)),
        'html.parser')
    links = get_links(thread_name, bs)
    if len(links) > 0:
        [queue.put(link) for link in links]
        newArticle = links[random.randint(0, len(links)-1)].attrs['href']
        scrape_article(thread_name, newArticle)

queue = Queue()

threads = [
    threading.Thread(
        target=scrape_article,
        args=('Thread 1', '/wiki/Kevin_Bacon',)
    ),
    threading.Thread(
        target=scrape_article,
        args=('Thread 2', '/wiki/Monty_Python',)
    ),
    threading.Thread(
        target=storage,
        args=(queue,)
    )
]

[t.start() for t in threads]
[t.join() for t in threads]
```

В этом сценарии 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 9.

Этот скрапер также несколько проще предыдущего. Вместо заголовка и URL страницы он исследует только URL. Кроме того, чтобы предотвратить добавление одного и того же URL в список **visited** посещенных страниц двумя потоками дважды, я превратила этот список во множество. Конечно, это не предотвращает сами попытки, но избавляет от их влияния на конечный результат.

Дополнительные возможности модуля threading

Модуль `threading` — это высокоуровневый интерфейс к более низкоуровневому модулю `_thread`. При желании вполне можно пользоваться и модулем `_thread`, но это потребует приложения дополнительных усилий и лишит вас мелочей, таких как удобные функции и изящные возможности, которые делают жизнь приятнее.

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. Аналогично 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.

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).

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**, позволяющий проверить, завершил ли веб-краулер работу. Это может быть полезно, когда есть какие-то дополнительные методы журналирования, которые необходимо завершить перед закрытием потока, при этом основная часть работы по 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 = [
    Process(target=print_time, args=('Counter', 1, 100)),
    Process(target=print_time, args=('Fizz', 3, 33)),
    Process(target=print_time, args=('Buzz', 5, 20))
]

[p.start() for p in processes]
[p.join() for p in processes]
```

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. 19.1.

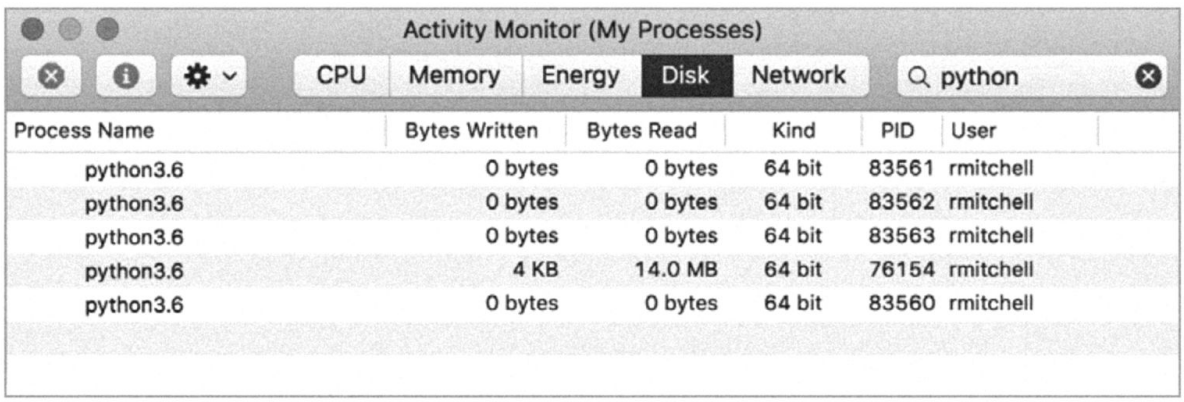


Рис. 19.1. Пять процессов Python, выполняемых во время работы FizzBuzz

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()
```

Вызов `os.getpid()` в каждом процессе должен вывести собственный PID процесса, а вызов `os.getppid()` — один и тот же PID родительского процесса.

Есть пара строк кода, которые в данной конкретной программе, строго говоря, не нужны. Если не включить заключительный оператор `join`

```
[p.join() for p in processes]
```

то родительский процесс завершится и автоматически остановит дочерние процессы. Однако если необходимо выполнить в родителе еще какие-то действия после завершения потомков, то следует вызвать метод `join`.

Например:

```
[p.start() for p in processes]
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 дочерних процессов и только потом продолжит выполнение:

```
[p.start() for p in processes]
[p.join() for p in processes]
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 завершатся 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

Пример многопоточного веб-краулинга «Википедии» можно изменить и вместо потоков использовать 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):
    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')
    print(f'Scraping {bs.find("h1").get_text()} in process {os.getpid()}')
    links = get_links(bs)
    if len(links) > 0:
        scrape_article(links[random.randint(0, len(links)-1)].attrs['href'])

processes = [
    Process(target=scrape_article, args=('/wiki/Kevin_Bacon',)),
    Process(target=scrape_article, args=('/wiki/Monty_Python',))
]
[p.start() for p in processes]
```

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 следующее:

```
Scraping Kevin Bacon in process 4067
Scraping Monty Python in process 4068
Scraping Ewan McGregor in process 4067
Scraping Charisma Records in process 4068
Scraping Renée Zellweger in process 4067
Scraping Genesis (band) in process 4068
Scraping Alana Haim in process 4067
Scraping Maroon 5 in process 4068
```

Теоретически веб-краулинг в отдельных процессах должен выполняться немного быстрее, чем в отдельных потоках, по двум основным причинам.

- Процессы не подвержены влиянию блокировки GIL. Они могут одновременно 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).
- 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 — извлечь ее оттуда. После извлечения информация исчезает из очереди. Поскольку очереди спроектированы как способ временной передачи данных, они не особенно хорошо подходят для хранения статических ссылок, таких как список посещенных веб-страниц.

Но что, если заменить этот статический список веб-страниц каким-либо диспетчером 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 диспетчером веб-скрапинга, который следит, чтобы в первую очередь задач добавлялись только новые URL:

```
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():
            # Приостановиться на 100 мс, ожидая появления заданий в очереди
            # 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(f'Scraping {bs.find('h1').get_text()} in process {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 сценариев, созданных нами ранее. Если раньше каждый процесс или поток выполнял случайный переход от назначенной ему начальной точки, то теперь они работают совместно, производя полное сканирование сайта. Каждый процесс может извлечь из очереди любую «задачу», а не только те ссылки, которые он сам нашел.

Ниже приводится пример работы этого скрапера, где процесс 97024 просматривает страницы **Monty Python** и **Philadelphia** (фильм, где снимался Кевин Бейкон):

```
Scraping Kevin Bacon in process 97023
Scraping Monty Python in process 97024
Scraping Kevin Bacon (disambiguation) in process 97023
Scraping Philadelphia in process 97024
Scraping Kevin Bacon filmography in process 97023
Scraping Kyra Sedgwick in process 97024
Scraping Sosie Bacon in process 97023
Scraping Edmund Bacon (architect) in process 97024
Scraping Michael Bacon (musician) in process 97023
Scraping Holly Near in process 97024
Scraping Leading actor in process 97023
```

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 так, что не нуждается в каких-либо указаниях или обмене данными? Тогда у вас едва ли найдутся причины, чтобы размышлять об импорте `_thread` на данном этапе.

Например, допустим, что нам нужно 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 запустить две команды:

```
$ 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 для обхода *одного* сайта, то нам понадобится какой-то способ, гарантирующий, 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, такую как *Redis* (https://redis.io/). Прежде чем перейти по новой ссылке, краулер может сделать запрос к ней и спросить: «Эта страница уже просканирована?» Краулер использует базу в качестве средства 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 сценарий, либо отдельный управляющий сценарий (будь то сценарий bash, задание cron или что-либо еще).

Тем не менее я не раз использовала данный метод с большим успехом. Для небольших одноразовых проектов это отличный способ быстро получить много информации, особенно с нескольких сайтов.

ГЛАВА 20

Прокси-сервисы для веб-скрапинга

Этой теме не случайно посвящена последняя глава книги. До сих пор вы запускали все приложения Python из командной строки на своем домашнем компьютере. Как говорится, «если ты кого-то любишь — дай ему свободу».

У вас может возникнуть подсознательное желание отложить этот шаг как нечто, что вам сейчас *не нужно*, однако вы можете быть удивлены, насколько проще станет ваша жизнь, когда вы перестанете запускать скраперы на своем ноутбуке.

Более того, с тех пор как в 2015 году было опубликовано первое издание этой книги, возникла целая индустрия прокси-компаний, занимающихся скрапингом веб-страниц. Раньше приходилось платить за аренду экземпляра облачного сервера и запускать на нем скрапер, как и любое другое программное обеспечение. Теперь же можно (тоже за определенную плату!) вызвать API и попросить «получить этот веб-сайт», а удаленная программа позаботится о деталях, решит любые проблемы безопасности и вернет вам данные.

В данной главе будут рассмотрены несколько способов, которые позволят вам направлять ваши запросы через удаленные IP-адреса, размещать и запускать программное обеспечение на других машинах и даже полностью переложить веб-скрапинг на прокси-сервер.

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. Зачем совершать лишние телодвижения, если нет мотивации для переноса программы в другое место? Те, кто решил перейти на удаленную платформу, обычно руководствуются двумя основными соображениями: потребностью в большей мощности и гибкости либо необходимостью использовать другой IP-адрес (потому что его адрес заблокирован или чтобы избежать его блокировки).

Как избежать блокировки IP-адреса

При создании веб-скраперов главное правило гласит: подделать можно практически все. Вы можете отправлять электронные письма с непринадлежащих вам адресов, передавать из командной строки автоматизированные данные о перемещениях мыши и даже, к ужасу веб-администраторов, передавать трафик их сайтов из Internet Explorer 9.0.

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 Штатов[1].

Большинство усилий, направленных на то, чтобы не дать 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 проблемы.

- 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 следующему 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

[1] 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.

популярный 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.

Переносимость и расширяемость

Некоторые задачи слишком велики для домашнего компьютера и подключения к Интернету. И хотя вы не намерены создавать большую нагрузку на какой-либо сайт, но, возможно, собираете данные на разных сайтах, для чего потребуется гораздо большая пропускная способность и объем памяти, чем способна 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. Приведем 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, хотя входящий и исходящий потоки данных любого сервера и могут быть скомпрометированы, 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.



Ограничения анонимности Tor

Хотя причина, по которой мы будем использовать Tor в книге, состоит в изменении своего IP-адреса, а не 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 вы пришли на веб-сервер, не является для этого сервера тем IP-адресом, по которому вас можно проследить. Однако вас может выдать любая информация, предоставляемая вами 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, пытаясь сдать выпускные экзамены, отправил через сеть Tor сообщение о том, что школа заминирована, 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[1].

Аутентификация в Tor — не автоматическая мантия-невидимка, которая предоставляет 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.

[1] См.: *Fandos N. P.* Harvard Sophomore Charged in Bomb Threat // The Harvard Crimson. — December 17, 2023. — URL: https://www.thecrimson.com/article/2013/12/17/student-charged-bomb-threat.

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, который, впрочем, имел пока что переменный успех.

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**), 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 (порт, применяемый по умолчанию):

```
import socks
import socket
from urllib.request import urlopen

socks.set_default_proxy(socks.PROXY_TYPE_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 этого сценария он 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 `proxy-server` для Chrome, согласно которому Selenium будет подключаться через порт 9150 по протоколу socks5:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
```

```
CHROMEDRIVER_PATH = ChromeDriverManager().install()
driver = webdriver.Chrome(service=Service(CHROMEDRIVER_PATH))
chrome_options = Options()
chrome_options.add_argument('--headless')
chrome_options.add_argument('--proxy-server=socks5://127.0.0.1:9150')
driver = webdriver.Chrome(
    service=Service(CHROMEDRIVER_PATH),
    options=chrome_options
)

driver.get('http://icanhazip.com')
print(driver.page_source)
driver.close()
```

При этом должен выводиться не ваш IP-адрес, а 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 может 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 с 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 сценарии через веб-интерфейс.

Если ваш сайт размещен на Linux-сервере, то на нем, скорее всего, уже установлен Python. Если же вы используете хостинг на Windows-сервере, то вам повезло меньше: придется специально проверить, есть ли там Python и 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 наличия 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 сценарии Python как *сценарии CGI* (Common Gateway Interface — общий шлюзовой интерфейс). CGI-сценариями называют программы, которые могут выполняться на сервере и динамически генерировать контент для отображения на сайте. Явно определяя сценарии Python как сценарии CGI, мы разрешаем серверу их выполнять, а не просто отображать содержимое в браузере или позволять пользователю его скачивать.

Напишите сценарий Python, загрузите его на сервер и определите права доступа к файлу как 755, то есть разрешите выполнять этот файл как программу. Для выполнения сценария 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 сценарий попал в открытый доступ и его мог выполнить любой желающий, то существуют два варианта этого избежать.

- Храните сценарий под нелогичным или скрытым URL и следите за тем, чтобы никогда не указывать ссылку на сценарий с какого-либо другого доступного URL во избежание его индексации поисковыми системами.
- Защитите сценарий паролем или потребуйте, чтобы перед выполнением сценария ему передавался пароль или секретный маркер.

Конечно, запуск сценария 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 завершиться никогда, в зависимости от того, как написан сценарий. 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 оплате 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 старших адресах памяти, высокой скорости вычислений или памяти большого объема. 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 текста 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.

Хотя нас не ограничивает ничто, кроме денег, на момент написания этой книги стоимость аренды самого дешевого вычислительного узла в Google (fi-micro) составляла всего 0,9 цента в час (меньше пенни), а сопоставимого по мощности микроэкземпляра Amazon EC2 — 0,8 цента в час. Благодаря экономии на масштабе аренда небольшого вычислительного узла в крупной компании обходится дешевле покупки и содержания собственной физической машины, потому что не требуется 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 они 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 в целях привлечения новых клиентов.

Если вы новичок в облачных вычислениях, то подскажу еще одного провайдера — DigitalOcean, который предоставляет вычислительные экземпляры (в их терминологии они называются капельками — droplets), начиная от 0,6 цента в час. У них невероятно простой пользовательский интерфейс, и они всего лишь отправляют вам по электронной почте IP-адрес и учетные данные для любого вновь созданного для вас экземпляра, чтобы вы могли войти в систему и начать работу. Они больше специализируются на хостинге веб-приложений, управлении DNS и балансировке нагрузки, но вы можете запускать на своих экземплярах все, что только захотите!

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, 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. 20.1.

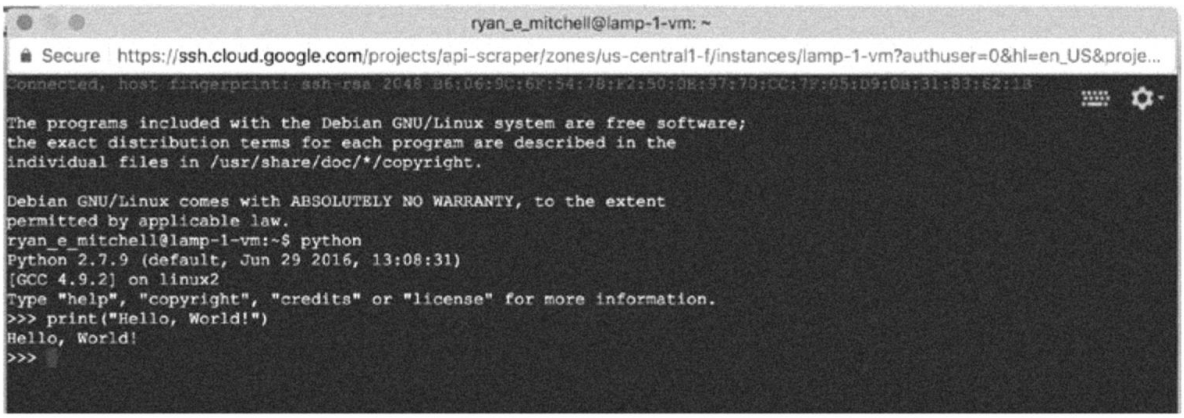


Рис. 20.1. Открытый в браузере терминал работающего вычислительного узла виртуальной машины Google Cloud Platform

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 парсинг 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, как использовать представленные в книге инструменты по отдельности, но и 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 16 мы объединили библиотеку Selenium, позволяющую идентифицировать Ajax-изображения, загруженные в Amazon, и Tesseract, чтобы прочитать их с помощью OCR. В задаче «Шесть шагов по "Википедии"» мы использовали регулярные выражения с целью написать веб-краулер, который хранил бы 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 Айдла?»

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.

Прокси-сервисы для скрапинга

В этой книге рассматривается множество продуктов и технологий, причем основное внимание уделяется бесплатному программному обеспечению с открытым исходным кодом. Платные продукты обсуждаются, лишь когда нет бесплатной альтернативы, она непрактична и/или платные продукты настолько распространены, что было бы упущением не упомянуть о них.

Индустрия прокси-серверов и API-сервисов для веб-скрапинга выглядит немного странно с точки зрения отрасли. Это новый, относительно узкий, но все же довольно многолюдный рынок с низким входным барьером. По этой причине пока не существует громких «имен», которые, по мнению всех программистов, *требовали* бы обсуждения. Да, некоторые имена известнее других, а некоторые услуги лучше, но пока что это все равно джунгли.

Кроме того, поскольку прокси-серверу, занимающемуся скрапингом веб-страниц, для работы требуется немалое количество оборудования и электроэнергии, то бесплатной альтернативы просто не существует и вряд ли она появится в будущем.

Это ставит меня в опасное положение, когда я пишу о множестве компаний, о которых вы, возможно, не слышали, но которые хотели бы получить ваши деньги. Спешу вас заверить, что ни одна из упомянутых мною компаний не заплатила мне.

Я пользовалась их услугами, разговаривала с их представителями, и в нескольких случаях мне были предоставлены бесплатные кредиты на исследовательские цели, но у меня нет никакого стимула продвигать какие-либо компании. Я не завишу ни от одной из них ни в финансовом, ни в эмоциональном отношении.

Я хочу, чтобы, читая этот раздел, вы размышляли об особенностях прокси-серверов веб-скрапинга и сервисов API, вашем бюджете и требованиях вашего проекта более обобщенно. Воспринимайте мои слова просто как результат тематического исследования и примеры того, «что там есть», а не как конкретные рекомендации. И если вам действительно захочется отдать деньги какой-либо из этих конкретных компаний, это будет касаться только вас и их!

ScrapingBee

ScrapingBee — самая маленькая из компаний в этом списке. Основное внимание в ней уделяется автоматизации JavaScript, консольным браузерам и безобидным на вид IP-адресам. ScrapingBee API хорошо документирован, но если вы предпочитаете не читать документацию, то на сайте ScrapingBee вы найдете инструмент, помогающий сгенерировать запросы API, который сведет вашу задачу к нажатию кнопок и копированию фрагментов текста.

Важной особенностью, которую следует учитывать при оценке прокси-сервисов, является количество времени, необходимое для получения ответа на запрос. Мало того, что ваш запрос должен быть перенаправлен с вашего компьютера на их сервер, затем на целевой сервер и обратно, прокси-служба может еще и буферизовать запросы на своей стороне, а не выполнять их немедленно. Нет ничего необычного в том, что ответ на запрос занимает минуту или больше. В ходе формальной оценки важно распределить эти запросы в течение дня и запланировать несколько типов запросов ко всем функциям, которые вы, возможно, захотите использовать.

Применяя ScrapingBee API напрямую, вы можете получить страницу с результатами и вывести не только ее, но и время, потребовавшееся для получения ответа:

```
import requests
import time

start = time.time()
params = {
    'api_key': SCRAPING_BEE_KEY,
    'url': 'https://www.target.com/p/-/A-83650487',
}
response = requests.get('https://app.scrapingbee.com/api/v1/', params=params)

print(f'Time: {time.time() - start}')
print(f'HTTP status: {response.status_code}')
print(f'Response body: {response.content}')
```

ScrapingBee также предлагает пакет для Python (https://pypi.org/project/scrapingbee/), который можно установить с помощью pip:

```
$ pip install scrapingbee
```

Это комплект инструментов для разработки программного обеспечения (Software Development Kit, SDK), который позволяет использовать различные функции API более удобным способом. Например, приведенный выше запрос можно записать так:

```
from scrapingbee import ScrapingBeeClient

start = time.time()
client = ScrapingBeeClient(api_key=SCRAPING_BEE_KEY)
response = client.get('https://www.target.com/p/-/A-83650487')

print(f'Time: {time.time() - start}')
print(f'HTTP status: {response.status_code}')
print(f'Response body: {response.content}')
```

Обратите внимание, что ответ, возвращаемый на этот запрос, можно использовать так же, как и в предыдущем примере.

Стоимость скрапинга обычно рассчитывается в единицах кредитов, где один базовый запрос API стоит один кредит. Такие функции, как выполнение JavaScript с помощью консольного браузера или домашнего IP-адреса, могут стоить от 5 до 75 кредитов. Каждый уровень платной подписки дает определенное количество кредитов в месяц.

Помимо бесплатной пробной версии с 1000 кредитов существуют платные подписки ScrapingBee, начиная с 50 долларов в месяц за 150 000 кредитов (3000 кредитов за доллар). По аналогии с большинством подобных услуг существуют большие скидки за объем — стоимость кредитов может опускаться до 13 000 за доллар и даже ниже при больших ежемесячных расходах.

Если вы хотите получить максимум от своих запросов, то имейте в виду, что ScrapingBee взимает 5 кредитов за выполнение JavaScript и включает эту плату по умолчанию. Это означает, что вышеуказанные запросы будут стоить не менее пяти кредитов каждый, а не один.

Это делает данную услугу очень удобной для клиентов, которые, возможно, не читали главу 14 этой книги и не понимают, почему данные, появляющиеся в их браузере, не отображаются в результатах скрапинга, возвращаемых из ScrapingBee. Если эти клиенты прочитают главу 15, они также поймут, как получить нужные им данные вообще без обработки JavaScript. Если вы прочитали обе эти главы, то можете отключить выполнение JavaScript и сократить затраты на запросы на 80 %, используя следующий код:

```
client = ScrapingBeeClient(api_key=SCRAPING_BEE_KEY)
params = {'render_js': 'false'}
response = client.get('https://www.target.com/p/-/A-83650487', params=params)
```

Как и многие другие подобные сервисы, ScrapingBee предлагает возможность использования премиальных IP-адресов, что может помочь предотвратить блокировку ваших скраперов веб-сайтами, которые опасаются IP-адресов, часто используемых ботами. Эти IP-адреса определяются как принадлежащие небольшим телекоммуникационным компаниям. Если этого недостаточно, ScrapingBee также предлагает скрытый IP-адрес по 75 кредитов за запрос. Скрытые IP-адреса, которые мне предоставили, были указаны как центры обработки данных и VPN-серверы, поэтому неясно, что именно представляют собой скрытые IP-адреса и какие реальные преимущества они дают по сравнению с адресами премиум-класса.

ScraperAPI

ScraperAPI, как следует из его названия, предлагает чистый REST-интерфейс со множеством возможностей. Он поддерживает асинхронные запросы, которые позволяют выполнить запрос на скрапинг и получить результаты позже отдельным вызовом API. Также можно предоставить конечную точку веб-обработчика, куда будут возвращены результаты по завершении запроса.

Простой вызов ScraperAPI за один кредит выглядит так:

```
import requests
import time

start = time.time()
params = {
    'api_key': SCRAPER_API_KEY,
    'url': 'https://www.target.com/p/-/A-83650487'
}
response = requests.get('http://api.scraperapi.com', params=params)
print(f'Time: {time.time() - start}')
print(f'HTTP status: {response.status_code}')
print(f'Response body: {response.content}')
```

ScraperAPI также имеет SDK, который можно установить с помощью pip:

```
$ pip install scraperapi-sdk
```

Этот SDK, как и многие другие, подобные ему, является лишь тонкой библиотекой-оболочкой, позволяющей посылать запросы из сценариев на Python. Аналогично ScrapingBee API на запросы возвращаются ответы в форме объектов библиотеки Requests:

```
from scraper_api import ScraperAPIClient

client = ScraperAPIClient(SCRAPER_API_KEY)
start = time.time()
result = client.get('https://www.target.com/p/-/A-83650487')
print(f'Time: {time.time() - start}')
print(f'HTTP status: {response.status_code}')
print(f'Response body: {response.content}')
```

При оценке сервисов скрапинга веб-страниц может возникнуть соблазн отдать предпочтение тем, которые имеют пакеты SDK для Python. Но не спешите и тщательно взвесьте, насколько наличие такого пакета уменьшит усилия по программированию и какие удобства даст. Технически можно с минимальными усилиями написать свой SDK для любого API скрапинга. Вот пример SDK для воображаемого API, состоящего всего из нескольких строк кода:

```
class RyansAPIClient:
    def __init__(self, key):
        self.key = key
        self.api_root = 'http://api.pythonscraping.com/ryansApiPath'

    def get(url):
        params = {'key': self.key, 'url': url}
        return requests.get(self.api_root, params=params)
```

Одна из уникальных особенностей ScraperAPI — инструменты автоматического парсинга продуктов Amazon и результатов поиска в Google. Большинство запросов на получение страницы со списком продуктов Amazon или страницы с результатами поиска в Amazon или Google стоит пять кредитов, а не один. Однако в документации упоминается явный вызов конечной точки Amazon Product Endpoint: **https://api.scraperapi.com/structured/amazon/product**, и похоже, что этот сервис включен по умолчанию:

```
from scraper_api import ScraperAPIClient

client = ScraperAPIClient(SCRAPER_API_KEY)
start = time.time()
result = client.get('https://www.amazon.com/Web-Scraping-Python-Collecting\
-Modern/dp/1491985577')
print(f'Time: {time.time() - start}')
print(f'HTTP status: {response.status_code}')
print(f'Response body: {response.text}')
```

В ответ на этот запрос возвращается примерно такой ответ:

```
Time: 4.672130823135376
HTTP status: 200
Response body: {"name":"Web Scraping with Python: Collecting More
Data from the Modern Web","product_information":{"publisher":
"O'Reilly Media; 2nd edition (May 8, 2018)","language":"English",
"paperback":"306 pages","isbn_10":"1491985577","isbn_13":
"978-1491985571","item_weight":"1.21 pounds" ...
```

Создание инструмента парсинга продуктов Amazon едва ли является непреодолимой задачей, но снятие с себя ответственности за тестирование и поддержку этого инструмента на протяжении многих лет может стоить затраченных средств.

Как упоминалось выше, ScraperAPI также позволяет выполнять асинхронные запросы и получать результаты позже. Для получения ответа на такой запрос требуется менее 100 мс:

```
start = time.time()
params = {
    'apiKey': SCRAPER_API_KEY,
    'url': 'https://www.target.com/p/-/A-83650487'
}
response = requests.post('https://async.scraperapi.com/jobs', json=params)
print(f'Time: {time.time() - start}')
print(f'HTTP status: {response.status_code}')
print(f'Response body: {response.content}')
```

Обратите внимание, что, в отличие от предыдущих примеров, это запрос `POST`, а не `GET`. В каком-то смысле вы публикуете данные для создания хранимой сущности на сервере ScraperAPI. Кроме того, имя атрибута, используемого для отправки ключа, меняется с `api_key` на `apiKey`.

Тело ответа содержит только URL, по которому можно получить результаты:

```
Time: 0.09664416313171387
HTTP status: 200
Response body: b'{"id":"728a365b-3a2a-4ed0-9209-cc4e7d88de96",
"attempts":0,"status":"running","statusUrl":"https://async.
scraperapi.com/jobs/728a365b-3a2a-4ed0-9209-cc4e7d88de96",
"url":"https://www.target.com/p/-/A-83650487"}'
```

Для обращения к этому URL не требуется ключ API, потому что достаточный уровень безопасности обеспечивает UUID. В случае, если выполнение запроса завершено, API вернет тело цели:

```
response = requests.get('https://async.scraperapi.com/jobs/\
728a365b-3a2a-4ed0-9209-cc4e7d88de96')
print(f'Response body: {response.content}')
```

Результаты таких асинхронных запросов хранятся до четырех часов или до тех пор, пока вы не получите данные. Такого же результата можно добиться и дома, используя многопоточный скрапер, но вам будет трудно добиться этого одновременно с чередованием домашних и мобильных IP-адресов, управлением данными сеанса, выполнением всего кода JavaScript (что быстро затормозит работу машины) и отслеживанием всех успехов и неудач на панели мониторинга.

Асинхронные запросы и веб-обработчики (когда прокси-сервис возвращает результаты на предоставленный вами URL) — прекрасные возможности, особенно для крупных и продолжительных проектов. ScraperAPI предоставляет все это без дополнительной платы за запрос, что особенно приятно.

Oxylabs

Oxylabs — крупная литовская компания, специализирующаяся на скрапинге страниц с результатами поисковых систем (Search Engine Results Page, SERP) и страниц продуктов. Ее экосистема продуктов и API требуют некоторого обучения. После создания учетной записи вы должны активировать (используя недельную пробную версию или оформив платную подписку) каждый «продукт» и создать отдельные учетные данные с именем пользователя и паролем, характерные для каждого продукта. Эти учетные данные работают как ключ API.

Продукт Web Scraper API позволяет выполнять вызовы, которые выглядят следующим образом:

```
import requests
import time

start = time.time()
data = {
    'url': 'https://www.target.com/p/-/A-83650487',
    'source': 'universal',
}

response = requests.post(
    'https://realtime.oxylabs.io/v1/queries',
    auth=(OXYLABS_USERNAME, OXYLABS_PASSWORD),
    json=data
)

response = response.json()['results'][0]

print(f'Time: {time.time() - start}')
print(f'HTTP status: {response["status_code"]}')
print(f'Response body: {response["content"]}')
```

Однако пользователя может ожидать сюрприз, если целевой URL будет переключен на URL из домена amazon.com:

```
data = {
    'url': 'https://www.amazon.com/Web-Scraping-Python-Collecting-Modern\
-dp-1491985577/dp/1491985577',
    'source': 'universal',
}

response = requests.post(
    'https://realtime.oxylabs.io/v1/queries',
    auth=(OXYLABS_USERNAME, OXYLABS_PASSWORD),
)
print(response.json())
```

Этот код выведет сообщение об ошибке:

```
{'message': 'provided url is not supported'}
```

Как и ScraperAPI, Oxylabs предлагает инструменты парсинга, специально разработанные для таких сайтов, как Amazon и Google. Однако для парсинга этих доменов — со специальными инструментами или без них — вы должны отдельно подписаться на продукт SERP Scraper API (для скрапинга Google, Bing, Baidu или Yandex) или продукт E-Commerce Scraper API (для скрапинга Amazon, Aliexpress, eBay и т. д.).

Подписавшись на продукт E-Commerce Scraper API, можно благополучно осуществить скрапинг домена Amazon, передав в атрибуте `source` значение `amazon` и свои учетные данные для этого продукта:

```
data = {
    'url': 'https://www.amazon.com/Web-Scraping-Python-Collecting-Modern\
-dp-1491985577/dp/1491985577',
    'source': 'amazon',
}

response = requests.post(
    'https://realtime.oxylabs.io/v1/queries',
    auth=(OXYLABS_USERNAME_ECOMMERCE, OXYLABS_PASSWORD),
    json=data
)
```

Этот запрос не делает ничего особенного. Он просто возвращает содержимое страницы, как обычно. Чтобы использовать шаблоны форматирования информации о продукте, также нужно установить атрибут `parse` в значение `True`:

```
data = {
    'url': 'https://www.amazon.com/Web-Scraping-Python-Collecting-Modern\
-dp-1491985577/dp/1491985577',
    'source': 'amazon',
    'parse': True
}
```

В таком случае сервис парсирует веб-сайт и вернет данные в формате JSON:

```
...
'page': 1,
'price': 32.59,
'stock': 'Only 7 left in stock - order soon',
'title': 'Web Scraping with Python: Collecting More Data from the Modern Web',
'buybox': [{'name': 'buy_new', 'price': 32.59, 'condition': 'new'},
...
```

Важно помнить, что сами инструменты парсинга не являются чем-то особенным для продукта E-Commerce Scraper API. Точно так же можно парсировать домен target.com с помощью обычного продукта Web Scraper API, установив атрибут `source` в значение `universal` и используя учетные данные Web Scraper API:

```
data = {
    'url': 'https://www.target.com/p/-/A-83650487',
    'source': 'universal',
```

```
    'parse': True
}

response = requests.post(
    'https://realtime.oxylabs.io/v1/queries',
    auth=(OXYLABS_USERNAME, OXYLABS_PASSWORD),
    json=data
)
```

Этот запрос вернет данные о продукте в формате JSON:

```
'url': 'https://www.target.com/p/-/A-83650487',
'price': 44.99,
'title': 'Web Scraping with Python - 2nd Edition by Ryan Mitchell (Paperback)',
'category': 'Target/Movies, Music & Books/Books/All Book Genres/Computers &
Technology Books',
'currency': 'USD',
'description': 'Error while parsing `description`: `(<class \'AttributeError\'>,
AttributeError("\'NoneType\' object has no attribute \'xpath\'"))`.',
'rating_score': 0, 'parse_status_code': 12004
```

При попытке автоматически парсировать страницы в домене target.com сервис может столкнуться с ошибками, как это произошло в примере выше в разделе `description`. К счастью, пользователи также могут писать собственные инструменты парсинга, совместимые с любым типом продукта API (Web Scraper API, SERP Scraper API, E-Commerce Scraper API и т. д.). Пользовательские парсеры принимают форму файлов JSON, оформленных согласно требованиям Oxylabs, которые определяют различные поля и селекторы XPath для сбора данных.

Пользовательские парсеры, по сути, являются бизнес-логикой веб-скрапера. Возможно, стоит учесть, что при переходе на использование другого сервиса веб-скрапинга или прокси-платформы эти инструменты окажутся бесполезными и их придется сильно модифицировать или переписывать заново. Разработка шаблонов парсинга веб-страниц на языке, специфичном для Oxylabs, может оказаться ограничивающим фактором для тех, кто пожелает перейти на использование другого сервиса.

Важно также подчеркнуть, что различные продукты API (которые, по сути, используют одну и ту же конечную точку API и имеют одинаковую структуру вызовов) определяются на основе не конкретных функций, а доменов, которым разрешено отправлять запросы, что может измениться в любой момент.

Домены, находящиеся в ведении конкретного продукта, также могут очень хорошо поддерживаться этим продуктом. Oxylab SERP Scraping API рекламирует поддержку таких сайтов, как Baidu и Bing, но для них не созданы шаблоны парсинга. Эта поддержка может ограничиваться простой возможностью поиска, например:

```
data = {
    'query': 'foo',
    'source': 'bing_search',
}
```

и не принимать полные URL:

```
data = {
    'url': 'https://bing.com?q=foo',
    'source': 'bing',
}
```

Обратите внимание, несмотря на то что я критикую некоторые аспекты продуктов Oxylab API, эта критика не направлена против компании как таковой и не должна интерпретироваться как результат всестороннего анализа или рекомендация. Я рассматриваю это только как тематическое исследование или пример для рассмотрения теми, кто, возможно, будет оценивать аналогичные продукты в будущем.

При оценке API и сервисов веб-скрапинга всегда важно учитывать, что рекламируется, что предоставляется и кто является целевой аудиторией. Структура вызовов API может раскрыть важную информацию о фактическом устройстве продукта, и даже документация может вводить в заблуждение.

Кроме перечисленных недостатков Oxylabs обладает также множеством превосходных качеств. Это один из лучших поставщиков прокси IP-адресов. Он постоянно имеет широкий спектр IP-адресов, которые определяются как принадлежащие мобильным операторам и центрам обработки данных. Как и у других прокси-сервисов, эти IP-адреса предлагаются по более высокой цене в зависимости от типа. Однако Oxylabs взимает плату за гигабайты, а не за запросы. В настоящее время стоимость варьируется от 22 долларов США за 1 Гбайт (мобильные IP-адреса с небольшим трафиком) до 8 долларов США за 1 Гбайт (абонентские IP-адреса с большим объемом трафика).

Zyte

Zyte, ранее известная как Scrapinghub, — еще одна крупная компания, предоставляющая услуги прокси-серверов и API для веб-скрапинга. Это также одна из старейших компаний, основанная в 2010 году. Я не имею особой привязанности ни к одной из этих компаний, но было неправдой, если бы я сказала, что Zyte как разработчик библиотеки Scrapy ничем не выделяется среди них. А кроме того начиная с 2019 года она также проводит саммит по веб-скрапингу (**https://www.extractsummit.io**).

Будучи крупной компанией, Zyte предлагает те же возможности, что и вышеупомянутые компании, и даже больше. Но, в отличие от большинства других компаний, она еще и продает данные. Если вам нужны, например, объявления

о вакансиях, данные о недвижимости или информация о продуктах, то компания может предоставить вам эти данные или предложить услуги консультантов, которые помогут вам собрать свои наборы данных.

Zyte поддерживает библиотеку Scrapy и включила его в свою линейку продуктов под названием Scrapy Cloud. Этот инструмент позволяет развертывать и запускать проекты Scrapy в облаке из репозитория GitHub или с локального компьютера с помощью клиента командной строки Scrapinghub (**https://pypi.org/project/shub/**). Это дает вам возможность сохранить независимость и переносимость вашей платформы веб-скраперов и при этом тесно взаимодействовать с экосистемой Zyte.

После развертывания проекта Scrapy сервис Zyte отыскивает все классы пауков в проекте и автоматически загружает их в вашу панель управления, которую можно использовать для запуска и мониторинга пауков во время их работы в облаке, а затем просматривать или загружать полученные данные.

Конечно, у Zyte есть и свой API. Он похож на другие API тем, что в значительной степени зависит от пакета Requests. Он также похож на API компании Oxylab тем, что полностью использует метод `POST` вместе с базовой аутентификацией HTTP. Однако в отличие от Oxylab базовая аутентификация используется только для отправки ключа Zyte, но не имени пользователя с паролем:

```
import time
from base64 import b64decode
import requests

json_data = {
    'url': 'https://www.target.com/p/-/A-83650487',
    'httpResponseBody': True,
}
start = time.time()
response = requests.post('https://api.zyte.com/v1/extract',
    auth=(ZYTE_KEY, ''), json=json_data)

response = response.json()
print(f'Time: {time.time() - start}')
print(f'HTTP status: {response["statusCode"]}')
body = b64decode(response["httpResponseBody"])
print(f'Response body: {body}')
```

Другое важное отличие — тела всех ответов возвращаются в виде строк в кодировке base64, а не в формате HTML или JSON. Эти строки легко обрабатывать с помощью пакета `base64`. Формат base64 позволяет также получать двоичные данные, изображения и другие файлы, для чего нужно лишь декодировать ответ как файл соответствующего типа.

Если вы не хотите использовать Scrapy и работаете над довольно простым проектом, то можете воспользоваться Zyte Automatic Extraction API. Этот сервис

использует искусственный интеллект для обнаружения различных полей на странице и возврата их в формате JSON. В настоящее время этот API работает как с товарами, так и с типами продуктов. Очевидно, что ему необязательно использовать кодировку base64, поскольку все парсируемые им страницы должны быть текстовыми:

```
json_data = [{
    'url': 'https://www.target.com/p/-/A-83650487',
    'pageType': 'product',
}]

response = requests.post(
    'https://autoextract.zyte.com/v1/extract',
    auth=(ZYTE_KEY, ''),
    json=json_data
)

print(response.json())
```

В документации к Zyte Automatic Extraction API указан URL **https://autoextract.scrapinghub.com/v1/extract**, содержащий прежнее название ScrapingHub, поэтому знайте, что обычно можно заменить `scrapinghub.com` на `zyte.com` и обеспечить обратную совместимость вашего кода, если Zyte решит закрыть старый домен.

Продукты Zyte в значительной степени ориентированы на разработчиков, работающих в корпоративной среде, которые хотят иметь полный контроль над своими скраперами. Однако Zyte предпочитает брать на себя управление IP-адресами с помощью Zyte Smart Proxy Manager. Zyte контролирует IP-адреса, через которые передается трафик. IP-адреса сохраняются между сеансами, но автоматически переключаются, если один из них оказывается заблокированным. Zyte пытается использовать переключение IP-адресов для создания естественного потока трафика к сайту и тем самым избежать подозрений.

Пользоваться Smart Proxy Manager довольно просто, однако установка сертификатов на ваш компьютер может усложнить задачу:

```
response = requests.get(
    'https://www.target.com/p/-/A-83650487',
    proxies={
        'http': f'http://{ZYTE_KEY}:@proxy.crawlera.com:8011/',
        'https': f'http://{ZYTE_KEY}:@proxy.crawlera.com:8011/',
    },
    verify='/path/to/zyte-proxy-ca.crt'
)
print(response.text)
```

Если вы не хотите использовать сертификат, можете отключить проверку с помощью модуля Requests (что, впрочем, не рекомендуется):

```
response = requests.get(
    ...
    verify=False
)
```

И конечно же, Zyte предлагает инструкции по интеграции своих прокси-сервисов со Scrapy (https://scrapy-zyte-smartproxy.readthedocs.io/en/latest/), которые затем можно запустить в Scrapy Cloud.

Стоимость прокси-запросов составляет около 1600 за доллар (или меньше на дорогих тарифах с ежемесячной оплатой). Стоимость запросов к API начинается примерно с 12 000 за доллар. Scrapy Cloud предлагает относительно недорогие тарифы, в числе которых бесплатный тариф и тариф Professional за 9 долларов в месяц. Вероятно, по их замыслам, это должно стимулировать использование Scrapy и интеграцию с платформой Zyte.

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, Кэтрин Херли и Пола Ньюсона (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 сценариев, которые можно задействовать для повышения гибкости приложений.

Если вы предпочитаете работать с Amazon, то обратите внимание на книгу *Python and AWS Cookbook* Митча Гарната (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.

Об авторе

Райан Митчелл пишет книги о веб-скрапинге и науке о данных с 2013 года. У нее есть шесть учебных курсов на LinkedIn, включая «Веб-скрапинг с помощью Python» и «Основы Python» — в настоящее время самый популярный курс по языку Python из представленных на платформе. Как эксперт в области веб-скрапинга, безопасности приложений и анализа данных, Райан проводила семинары и выступала на многих мероприятиях, включая Data Day и DEF CON.

Райан получила степень магистра в области разработки программного обеспечения в Школе повышения квалификации Гарвардского университета и степень бакалавра технических наук в Инженерном колледже Олина. В настоящее время работает главным инженером-программистом в Gerson Lehrman Group, где занимается машинным обучением и обработкой данных на Python. Она регулярно консультирует проекты веб-скрапинга в сферах розничной торговли, финансов и фармацевтики и некоторое время работала консультантом по учебным программам и внештатным преподавателем в Северо-Восточном университете и Инженерном колледже Олина.

Иллюстрация на 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 см в длину и весят от 15 до 33 кг. 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 степные ящеры живут в норах, которые уходят глубоко под землю. Эти норы часто прежде принадлежали трубкозубам и бородавочникам, а панголин просто захватил заброшенное жилище. Однако благодаря трем длинным изогнутым когтям 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.

Иллюстрацию для обложки нарисовала Карен Монтгомери на основе старинной гравюры из *Royal Natural History* Ричарда Лидеккера.

Райан Митчелл

Парсинг с помощью Python. Веб-скрапинг в действии

3-е международное издание

Перевел с английского А. Киселев

Изготовлено в России. Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес: 010000, Казахстан, город Астана, район Алматы, Проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.
Дата изготовления: 12.2024. Наименование: книжная продукция. Срок годности: не ограничен.
Подписано в печать 29.10.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 700. Заказ 0000.



Билл Любанович

FASTAPI: ВЕБ-РАЗРАБОТКА НА PYTHON

FastAPI — относительно новый, но надежный фреймворк с чистым дизайном, использующий преимущества актуальных возможностей Python. Как следует из названия, FastAPI отличается высоким быстродействием и способен конкурировать в этом с аналогичными фреймворками на таких языках, как Golang. Эта практическая книга расскажет разработчикам, знакомым с Python, как FastAPI позволяет достичь большего за меньшее время и с меньшим количеством кода.

Билл Любанович рассказывает о тонкостях разработки с применением FastAPI и предлагает множество рекомендаций по таким темам, как формы, доступ к базам данных, графика, карты и многое другое, что поможет освоить основы и даже пойти дальше. Кроме того, вы познакомитесь с RESTful API, приемами валидации данных, авторизации и повышения производительности. Благодаря сходству с такими фреймворками, как Flask и Django, вы легко начнете работу с FastAPI.

SPRINT book

O'REILLY®

Парсинг с помощью Python

Если программирование — магия, то веб-скрапинг — настоящее колдовство. Написав простую автоматизированную программу, можно отправлять запросы веб-серверам, получать ответы с данными, а затем анализировать эти данные и извлекать необходимую информацию. Это обновленное третье издание книги не только познакомит вас с веб-скрапингом, но и послужит исчерпывающим руководством по сбору практически любых видов данных в современном Интернете.

В части I основное внимание уделено механике веб-скрапинга: как с помощью Python отправлять запросы веб-серверам, обрабатывать ответы и автоматизировать взаимодействие с сайтами. В части II исследуются более конкретные инструменты и приложения, которые пригодятся при любом сценарии веб-скрапинга.

Книга покажет, как:

- анализировать сложные HTML-страницы;
- разрабатывать веб-сканеры с помощью фреймворка Scrapy;
- хранить данные, полученные с помощью скрапинга;
- читать и извлекать данные из документов;
- очищать и нормализовывать плохо отформатированные данные;
- читать и записывать информацию на естественных языках;
- выполнять поиск по формам и страницам входа;
- выполнять скрапинг JavaScript-кода и работать с API;
- писать и использовать программы для преобразования изображений в текст;
- обходить противоскрапинговые ловушки и блокаторы ботов;
- тестировать свои веб-сайты с помощью скрапинга.

«Эта превосходная и информативная книга учит осознанному и эффективному веб-скрапингу. Я рекомендую ее всем, кто хочет извлекать данные с веб-сайтов или планирует создать веб-сайт для обмена данными».

– Майкл Хопкинс, P.Geo., PMP

Райан Митчелл — старший разработчик в компании GLG, эксперт в области веб-скрапинга, веб-безопасности, data science. Проводит семинары и выступает на многих мероприятиях, включая Data Day и DEF CON, ведет курсы по веб-программированию и обработке данных, а также консультирует различные компании. Автор книги Instant Web Scraping with Java (Packt Publishing).

SPRINT book

ISBN:978-601-08-4570-1

9 786010 845701