Community Experience Distilled

Web Scraping with Python

Scrape data from any website with the power of Python

Richard Lawson

[PACKT] PUBLISHING open source* community experience distilled

Web Scraping with Python

Scrape data from any website with the power of Python

Richard Lawson

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Web Scraping with Python

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2015

Production reference: 1231015

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-436-4

www.packtpub.com

Credits

Author
Richard Lawson

Reviewers
Martin Burch
Christopher Davis
William Sankey
Ayush Tiwari

Acquisition Editor
Rebecca Youé

Content Development Editor
Akashdeep Kundu

Technical Editors
Novina Kewalramani
Shruti Rawool

Copy Editor
Sonia Cheema

Project Coordinator
Milton Dsouza

Proofreader
Safis Editing

Indexer
Mariammal Chettiar

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Richard Lawson is from Australia and studied Computer Science at the University of Melbourne. Since graduating, he built a business specializing at web scraping while traveling the world, working remotely from over 50 countries. He is a fluent Esperanto speaker, conversational at Mandarin and Korean, and active in contributing to and translating open source software. He is currently undertaking postgraduate studies at Oxford University and in his spare time enjoys developing autonomous drones.

I would like to thank Professor Timothy Baldwin for introducing me to this exciting field and Tharavy Douc for hosting me in Paris while I wrote this book.

About the Reviewers

Martin Burch is a data journalist based in New York City, where he makes interactive graphics for The Wall Street Journal. He holds a master of arts in journalism from the City University of New York's Graduate School of Journalism, and has a baccalaureate from New Mexico State University, where he studied journalism and information systems.

I would like to thank my wife, Lisa, who encouraged me to assist with this book; my uncle, Michael, who has always patiently answered my programming questions; and my father, Richard, who inspired my love of journalism and writing.

William Sankey is a data professional and hobbyist developer who lives in College Park, Maryland. He graduated in 2012 from Johns Hopkins University with a master's degree in public policy and specializes in quantitative analysis. He is currently a health services researcher at L&M Policy Research, LLC, working on projects for the Centers for Medicare and Medicaid Services (CMS). The scope of these projects range from evaluating Accountable Care Organizations to monitoring the Inpatient Psychiatric Facility Prospective Payment System.

I would like to thank my devoted wife, Julia, and rambunctious puppy, Ruby, for all their love and support.

Ayush Tiwari is a Python developer and undergraduate at IIT Roorkee. He has been working at Information Management Group, IIT Roorkee, since 2013, and has been actively working in the web development field. Reviewing this book has been a great experience for him. He did his part not only as a reviewer, but also as an avid learner of web scraping. He recommends this book to all Python enthusiasts so that they can enjoy the benefits of scraping.

He is enthusiastic about Python web scraping and has worked on projects such as live sports feeds, as well as a generalized Python e-commerce web scraper (at Miranj).

He has also been handling a placement portal with the help of a Django app to assist the placement process at IIT Roorkee.

Besides backend development, he loves to work on computational Python/data analysis using Python libraries, such as NumPy, SciPy, and is currently working in the CFD research field. You can visit his projects on GitHub. His username is `tiwariayush`.

He loves trekking through Himalayan valleys and participates in several treks every year, adding this to his list of interests, besides playing the guitar. Among his accomplishments, he is a part of the internationally acclaimed Super 30 group and has also been a rank holder in it. When he was in high school, he also qualified for the International Mathematical Olympiad.

I have been provided a lot of help by my family members (my sister, Aditi, my parents, and Anand sir), my friends at VI and IMG, and my professors. I would like to thank all of them for the support they have given me.

Last but not least, kudos to the respected author and the Packt Publishing team for publishing these fantastic tech books. I commend all the hard work involved in producing their books.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

The Internet contains the most useful set of data ever assembled, which is largely publicly accessible for free. However, this data is not easily reusable. It is embedded within the structure and style of websites and needs to be extracted to be useful. This process of extracting data from web pages is known as web scraping and is becoming increasingly useful as ever more information is available online.

What this book covers

Chapter 1, *Introduction to Web Scraping*, introduces web scraping and explains ways to crawl a website.

Chapter 2, *Scraping the Data*, shows you how to extract data from web pages.

Chapter 3, *Caching Downloads*, teaches you how to avoid redownloading by caching results.

Chapter 4, *Concurrent Downloading*, helps you to scrape data faster by downloading in parallel.

Chapter 5, *Dynamic Content*, shows you how to extract data from dynamic websites.

Chapter 6, *Interacting with Forms*, shows you how to work with forms to access the data you are after.

Chapter 7, *Solving CAPTCHA*, elaborates how to access data that is protected by CAPTCHA images.

Chapter 8, *Scrapy*, teaches you how to use the popular high-level Scrapy framework.

Chapter 9, *Overview*, is an overview of web scraping techniques that have been covered.

What you need for this book

All the code used in this book has been tested with Python 2.7, and is available for download at `http://bitbucket.org/wswp/code`. Ideally, in a future version of this book, the examples will be ported to Python 3. However, for now, many of the libraries required (such as Scrapy/Twisted, Mechanize, and Ghost) are only available for Python 2. To help illustrate the crawling examples, we created a sample website at `http://example.webscraping.com`. This website limits how fast you can download content, so if you prefer to host this yourself the source code and installation instructions are available at `http://bitbucket.org/wswp/places`.

We decided to build a custom website for many of the examples used in this book instead of scraping live websites, so that we have full control over the environment. This provides us stability—live websites are updated more often than books, and by the time you try a scraping example, it may no longer work. Also, a custom website allows us to craft examples that illustrate specific skills and avoid distractions. Finally, a live website might not appreciate us using them to learn about web scraping and try to block our scrapers. Using our own custom website avoids these risks; however, the skills learnt in these examples can certainly still be applied to live websites.

Who this book is for

This book requires prior programming experience and would not be suitable for absolute beginners. When practical we will implement our own version of web scraping techniques so that you understand how they work before introducing the popular existing module. These examples will assume competence with Python and installing modules with `pip`. If you need a brush up, there is an excellent free online book by Mark Pilgrim available at `http://www.diveintopython.net`. This is the resource I originally used to learn Python.

The examples also assume knowledge of how web pages are constructed with HTML and updated with JavaScript. Prior knowledge of HTTP, CSS, AJAX, WebKit, and MongoDB would also be useful, but not required, and will be introduced as and when each technology is needed. Detailed references for many of these topics are available at `http://www.w3schools.com`.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Most websites define a `robots.txt` file to let robots know any restrictions about crawling their website."

A block of code is set as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url><loc>http://example.webscraping.com/view/Afghanistan-1
    </loc></url>
  <url><loc>http://example.webscraping.com/view/Aland-Islands-2
    </loc></url>
  <url><loc>http://example.webscraping.com/view/Albania-3</loc>
    </url>
  ...
</urlset>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def link_crawler(..., scrape_callback=None):
          …
          links = []
    if scrape_callback:
        links.extend(scrape_callback(url, html) or [])
          ...
```

Any command-line input or output is written as follows:

```
$ python performance.py
Regular expressions: 5.50 seconds
BeautifulSoup: 42.84 seconds
Lxml: 7.06 seconds
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " When regular users open this web page in their browser, they will enter their e-mail and password, and click on the **Log In** button to submit the details to the server."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Introduction to Web Scraping

In this chapter, we will cover the following topics:

- Introduce the field of web scraping
- Explain the legal challenges
- Perform background research on our target website
- Progressively building our own advanced web crawler

When is web scraping useful?

Suppose I have a shop selling shoes and want to keep track of my competitor's prices. I could go to my competitor's website each day to compare each shoe's price with my own, however this would take a lot of time and would not scale if I sold thousands of shoes or needed to check price changes more frequently. Or maybe I just want to buy a shoe when it is on sale. I could come back and check the shoe website each day until I get lucky, but the shoe I want might not be on sale for months. Both of these repetitive manual processes could instead be replaced with an automated solution using the web scraping techniques covered in this book.

In an ideal world, web scraping would not be necessary and each website would provide an API to share their data in a structured format. Indeed, some websites do provide APIs, but they are typically restricted by what data is available and how frequently it can be accessed. Additionally, the main priority for a website developer will always be to maintain the frontend interface over the backend API. In short, we cannot rely on APIs to access the online data we may want and therefore, need to learn about web scraping techniques.

Is web scraping legal?

Web scraping is in the early Wild West stage, where what is permissible is still being established. If the scraped data is being used for personal use, in practice, there is no problem. However, if the data is going to be republished, then the type of data scraped is important.

Several court cases around the world have helped establish what is permissible when scraping a website. In *Feist Publications, Inc. v. Rural Telephone Service Co.*, the United States Supreme Court decided that scraping and republishing facts, such as telephone listings, is allowed. Then, a similar case in Australia, *Telstra Corporation Limited v. Phone Directories Company Pty Ltd*, demonstrated that only data with an identifiable author can be copyrighted. Also, the European Union case, *ofir.dk vs home.dk*, concluded that regular crawling and deep linking is permissible.

These cases suggest that when the scraped data constitutes facts (such as business locations and telephone listings), it can be republished. However, if the data is original (such as opinions and reviews), it most likely cannot be republished for copyright reasons.

In any case, when you are scraping data from a website, remember that you are their guest and need to behave politely or they may ban your IP address or proceed with legal action. This means that you should make download requests at a reasonable rate and define a user agent to identify you. The next section on crawling will cover these practices in detail.

You can read more about these legal cases at `http://caselaw.lp.findlaw.com/scripts/getcase.pl?court=US&vol=499&invol=340`, `http://www.austlii.edu.au/au/cases/cth/FCA/2010/44.html`, and `http://www.bvhd.dk/uploads/tx_mocarticles/S_-_og_Handelsrettens_afg_relse_i_Ofir-sagen.pdf`.

Background research

Before diving into crawling a website, we should develop an understanding about the scale and structure of our target website. The website itself can help us through their `robots.txt` and `Sitemap` files, and there are also external tools available to provide further details such as Google Search and `WHOIS`.

Checking robots.txt

Most websites define a `robots.txt` file to let crawlers know of any restrictions about crawling their website. These restrictions are just a suggestion but good web citizens will follow them. The `robots.txt` file is a valuable resource to check before crawling to minimize the chance of being blocked, and also to discover hints about a website's structure. More information about the `robots.txt` protocol is available at `http://www.robotstxt.org`. The following code is the content of our example `robots.txt`, which is available at `http://example.webscraping.com/robots.txt`:

```
# section 1
User-agent: BadCrawler
Disallow: /

# section 2
User-agent: *
Crawl-delay: 5
Disallow: /trap

# section 3
Sitemap: http://example.webscraping.com/sitemap.xml
```

In section 1, the `robots.txt` file asks a crawler with user agent `BadCrawler` not to crawl their website, but this is unlikely to help because a malicious crawler would not respect `robots.txt` anyway. A later example in this chapter will show you how to make your crawler follow `robots.txt` automatically.

Section 2 specifies a crawl delay of 5 seconds between download requests for all User-Agents, which should be respected to avoid overloading their server. There is also a `/trap` link to try to block malicious crawlers who follow disallowed links. If you visit this link, the server will block your IP for one minute! A real website would block your IP for much longer, perhaps permanently, but then we could not continue with this example.

Section 3 defines a `Sitemap` file, which will be examined in the next section.

Examining the Sitemap

`Sitemap` files are provided by websites to help crawlers locate their updated content without needing to crawl every web page. For further details, the sitemap standard is defined at `http://www.sitemaps.org/protocol.html`. Here is the content of the `Sitemap` file discovered in the `robots.txt` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url><loc>http://example.webscraping.com/view/Afghanistan-1
    </loc></url>
  <url><loc>http://example.webscraping.com/view/Aland-Islands-2
    </loc></url>
  <url><loc>http://example.webscraping.com/view/Albania-3</loc>
    </url>
  ...
</urlset>
```

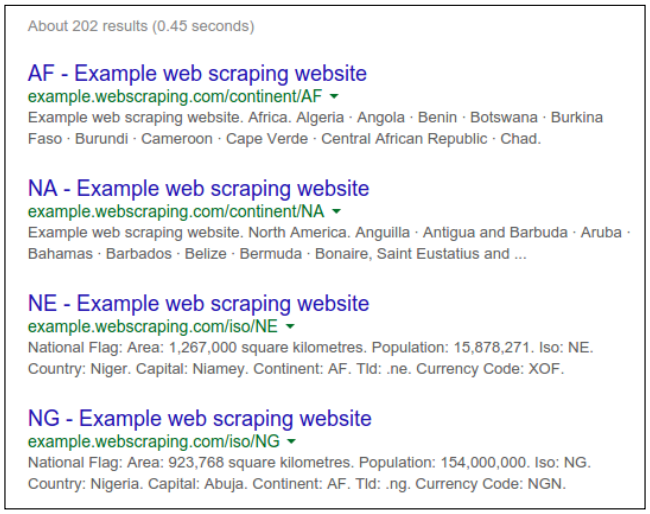
This sitemap provides links to all the web pages, which will be used in the next section to build our first crawler. `Sitemap` files provide an efficient way to crawl a website, but need to be treated carefully because they are often missing, out of date, or incomplete.

Estimating the size of a website

The size of the target website will affect how we crawl it. If the website is just a few hundred URLs, such as our example website, efficiency is not important. However, if the website has over a million web pages, downloading each sequentially would take months. This problem is addressed later in *Chapter 4*, *Concurrent Downloading*, on distributed downloading.

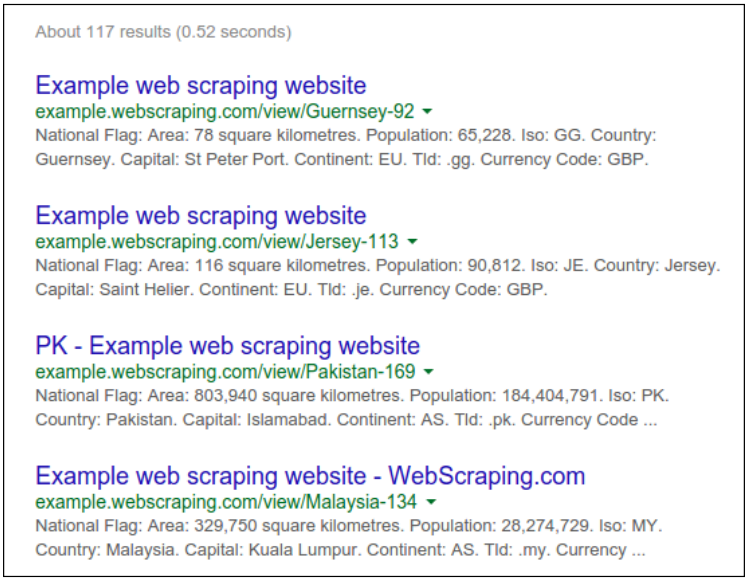
A quick way to estimate the size of a website is to check the results of Google's crawler, which has quite likely already crawled the website we are interested in. We can access this information through a Google search with the `site` keyword to filter the results to our domain. An interface to this and other advanced search parameters are available at `http://www.google.com/advanced_search`.

Here are the site search results for our example website when searching Google for `site:example.webscraping.com`:



As we can see, Google currently estimates 202 web pages, which is about as expected. For larger websites, I have found Google's estimates to be less accurate.

We can filter these results to certain parts of the website by adding a URL path to the domain. Here are the results for `site:example.webscraping.com/view`, which restricts the site search to the country web pages:



This additional filter is useful because ideally you will only want to crawl the part of a website containing useful data rather than every page of it.

Identifying the technology used by a website

The type of technology used to build a website will effect how we crawl it. A useful tool to check the kind of technologies a website is built with is the `builtwith` module, which can be installed with:

```
pip install builtwith
```

This module will take a URL, download and analyze it, and then return the technologies used by the website. Here is an example:

```
>>> import builtwith
>>> builtwith.parse('http://example.webscraping.com')
{u'javascript-frameworks': [u'jQuery', u'Modernizr', u'jQuery UI'],
 u'programming-languages': [u'Python'],
 u'web-frameworks': [u'Web2py', u'Twitter Bootstrap'],
 u'web-servers': [u'Nginx']}
```

We can see here that the example website uses the Web2py Python web framework alongside with some common JavaScript libraries, so its content is likely embedded in the HTML and be relatively straightforward to scrape. If the website was instead built with AngularJS, then its content would likely be loaded dynamically. Or, if the website used ASP.NET, then it would be necessary to use sessions and form submissions to crawl web pages. Working with these more difficult cases will be covered later in *Chapter 5*, *Dynamic Content* and *Chapter 6*, *Interacting with Forms*.

Finding the owner of a website

For some websites it may matter to us who is the owner. For example, if the owner is known to block web crawlers then it would be wise to be more conservative in our download rate. To find who owns a website we can use the `WHOIS` protocol to see who is the registered owner of the domain name. There is a Python wrapper to this protocol, documented at `https://pypi.python.org/pypi/python-whois`, which can be installed via `pip`:

```
pip install python-whois
```

Here is the key part of the `WHOIS` response when querying the appspot.com domain with this module:

```
>>> import whois
>>> print whois.whois('appspot.com')
```

```
{
  ...
  "name_servers": [
    "NS1.GOOGLE.COM",
    "NS2.GOOGLE.COM",
    "NS3.GOOGLE.COM",
    "NS4.GOOGLE.COM",
    "ns4.google.com",
    "ns2.google.com",
    "ns1.google.com",
    "ns3.google.com"
  ],
  "org": "Google Inc.",
  "emails": [
    "abusecomplaints@markmonitor.com",
    "dns-admin@google.com"
  ]
}
```

We can see here that this domain is owned by Google, which is correct—this domain is for the Google App Engine service. Google often blocks web crawlers despite being fundamentally a web crawling business themselves. We would need to be careful when crawling this domain because Google often blocks web crawlers, despite being fundamentally a web crawling business themselves.

Crawling your first website

In order to scrape a website, we first need to download its web pages containing the data of interest—a process known as **crawling**. There are a number of approaches that can be used to crawl a website, and the appropriate choice will depend on the structure of the target website. This chapter will explore how to download web pages safely, and then introduce the following three common approaches to crawling a website:

- Crawling a sitemap
- Iterating the database IDs of each web page
- Following web page links

Downloading a web page

To crawl web pages, we first need to download them. Here is a simple Python script that uses Python's `urllib2` module to download a URL:

```
import urllib2
def download(url):
    return urllib2.urlopen(url).read()
```

When a URL is passed, this function will download the web page and return the HTML. The problem with this snippet is that when downloading the web page, we might encounter errors that are beyond our control; for example, the requested page may no longer exist. In these cases, `urllib2` will raise an exception and exit the script. To be safer, here is a more robust version to catch these exceptions:

```
import urllib2

def download(url):
    print 'Downloading:', url
    try:
        html = urllib2.urlopen(url).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
    return html
```

Now, when a download error is encountered, the exception is caught and the function returns `None`.

Retrying downloads

Often, the errors encountered when downloading are temporary; for example, the web server is overloaded and returns a `503 Service Unavailable` error. For these errors, we can retry the download as the server problem may now be resolved. However, we do not want to retry downloading for all errors. If the server returns `404 Not Found`, then the web page does not currently exist and the same request is unlikely to produce a different result.

The full list of possible HTTP errors is defined by the Internet Engineering Task Force, and is available for viewing at `https://tools.ietf.org/html/rfc7231#section-6`. In this document, we can see that the `4xx` errors occur when there is something wrong with our request and the `5xx` errors occur when there is something wrong with the server. So, we will ensure our `download` function only retries the `5xx` errors. Here is the updated version to support this:

```
def download(url, num_retries=2):
    print 'Downloading:', url
    try:
        html = urllib2.urlopen(url).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
        if num_retries > 0:
            if hasattr(e, 'code') and 500 <= e.code < 600:
                # recursively retry 5xx HTTP errors
                return download(url, num_retries-1)
    return html
```

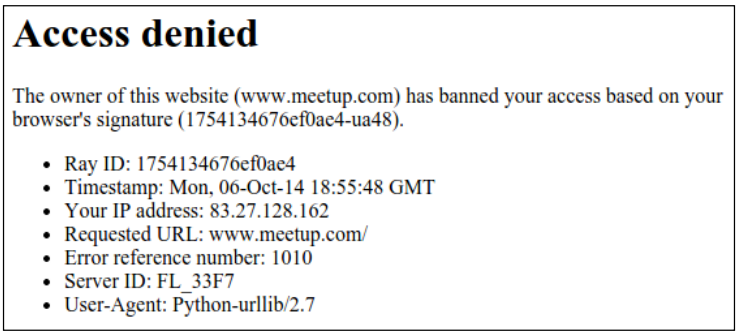
Now, when a `download` error is encountered with a `5xx` code, the `download` is retried by recursively calling itself. The function now also takes an additional argument for the number of times the download can be retried, which is set to two times by default. We limit the number of times we attempt to download a web page because the server error may not be resolvable. To test this functionality we can try downloading `http://httpstat.us/500`, which returns the 500 error code:

```
>>> download('http://httpstat.us/500')
Downloading: http://httpstat.us/500
Download error: Internal Server Error
Downloading: http://httpstat.us/500
Download error: Internal Server Error
Downloading: http://httpstat.us/500
Download error: Internal Server Error
```

As expected, the `download` function now tries downloading the web page, and then on receiving the 500 error, it retries the download twice before giving up.

Setting a user agent

By default, `urllib2` will download content with the `Python-urllib/2.7` user agent, where `2.7` is the version of `Python`. It would be preferable to use an identifiable user agent in case problems occur with our web crawler. Also, some websites block this default user agent, perhaps after they experienced a poorly made Python web crawler overloading their server. For example, this is what `http://www.meetup.com/` currently returns for Python's default user agent:



Access denied

The owner of this website (www.meetup.com) has banned your access based on your browser's signature (1754134676ef0ae4-ua48).

- Ray ID: 1754134676ef0ae4
- Timestamp: Mon, 06-Oct-14 18:55:48 GMT
- Your IP address: 83.27.128.162
- Requested URL: www.meetup.com/
- Error reference number: 1010
- Server ID: FL_33F7
- User-Agent: Python-urllib/2.7

So, to download reliably, we will need to have control over setting the user agent. Here is an updated version of our `download` function with the default user agent set to `'wswp'` (which stands for **Web Scraping with Python**):

```
def download(url, user_agent='wswp', num_retries=2):
    print 'Downloading:', url
    headers = {'User-agent': user_agent}
    request = urllib2.Request(url, headers=headers)
    try:
        html = urllib2.urlopen(request).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
        if num_retries > 0:
            if hasattr(e, 'code') and 500 <= e.code < 600:
                # retry 5XX HTTP errors
                return download(url, user_agent, num_retries-1)
    return html
```

Now we have a flexible download function that can be reused in later examples to catch errors, retry the download when possible, and set the user agent.

Sitemap crawler

For our first simple crawler, we will use the sitemap discovered in the example website's `robots.txt` to download all the web pages. To parse the sitemap, we will use a simple regular expression to extract URLs within the `<loc>` tags. Note that a more robust parsing approach called **CSS selectors** will be introduced in the next chapter. Here is our first example crawler:

```
def crawl_sitemap(url):
    # download the sitemap file
    sitemap = download(url)
    # extract the sitemap links
    links = re.findall('<loc>(.*?)</loc>', sitemap)
    # download each link
    for link in links:
        html = download(link)
        # scrape html here
        # ...
```

Now, we can run the sitemap crawler to download all countries from the example website:

```
>>> crawl_sitemap('http://example.webscraping.com/sitemap.xml')
Downloading: http://example.webscraping.com/sitemap.xml
Downloading: http://example.webscraping.com/view/Afghanistan-1
Downloading: http://example.webscraping.com/view/Aland-Islands-2
Downloading: http://example.webscraping.com/view/Albania-3
...
```

This works as expected, but as discussed earlier, `Sitemap` files often cannot be relied on to provide links to every web page. In the next section, another simple crawler will be introduced that does not depend on the `Sitemap` file.

ID iteration crawler

In this section, we will take advantage of weakness in the website structure to easily access all the content. Here are the URLs of some sample countries:

- `http://example.webscraping.com/view/Afghanistan-1`
- `http://example.webscraping.com/view/Australia-2`
- `http://example.webscraping.com/view/Brazil-3`

We can see that the URLs only differ at the end, with the country name (known as a slug) and ID. It is a common practice to include a slug in the URL to help with search engine optimization. Quite often, the web server will ignore the slug and only use the ID to match with relevant records in the database. Let us check whether this works with our example website by removing the slug and loading `http://example.webscraping.com/view/1`:

The web page still loads! This is useful to know because now we can ignore the slug and simply iterate database IDs to download all the countries. Here is an example code snippet that takes advantage of this trick:

```
import itertools
for page in itertools.count(1):
    url = 'http://example.webscraping.com/view/-%d' % page
    html = download(url)
    if html is None:
        break
    else:
        # success - can scrape the result
        pass
```

Here, we iterate the ID until we encounter a download error, which we assume means that the last country has been reached. A weakness in this implementation is that some records may have been deleted, leaving gaps in the database IDs. Then, when one of these gaps is reached, the crawler will immediately exit. Here is an improved version of the code that allows a number of consecutive download errors before exiting:

```
# maximum number of consecutive download errors allowed
max_errors = 5
# current number of consecutive download errors
num_errors = 0
for page in itertools.count(1):
    url = 'http://example.webscraping.com/view/-%d' % page
    html = download(url)
    if html is None:
        # received an error trying to download this webpage
        num_errors += 1
        if num_errors == max_errors:
            # reached maximum number of
            # consecutive errors so exit
            break
    else:
        # success - can scrape the result
        # ...
        num_errors = 0
```

The crawler in the preceding code now needs to encounter five consecutive `download` errors to stop iterating, which decreases the risk of stopping the iteration prematurely when some records have been deleted.

Iterating the IDs is a convenient approach to crawl a website, but is similar to the sitemap approach in that it will not always be available. For example, some websites will check whether the slug is as expected and if not return a `404 Not Found` error. Also, other websites use large nonsequential or nonnumeric IDs, so iterating is not practical. For example, Amazon uses ISBNs as the ID for their books, which have at least ten digits. Using an ID iteration with Amazon would require testing billions of IDs, which is certainly not the most efficient approach to scraping their content.

Link crawler

So far, we have implemented two simple crawlers that take advantage of the structure of our sample website to download all the countries. These techniques should be used when available, because they minimize the required amount of web pages to download. However, for other websites, we need to make our crawler act more like a typical user and follow links to reach the content of interest.

We could simply download the entire website by following all links. However, this would download a lot of web pages that we do not need. For example, to scrape user account details from an online forum, only account pages need to be downloaded and not discussion threads. The link crawler developed here will use a regular expression to decide which web pages to download. Here is an initial version of the code:

```
import re

def link_crawler(seed_url, link_regex):
    """Crawl from the given seed URL following links matched by link_regex
    """
    crawl_queue = [seed_url]
    while crawl_queue:
        url = crawl_queue.pop()
        html = download(url)
        # filter for links matching our regular expression
        for link in get_links(html):
            if re.match(link_regex, link):
                crawl_queue.append(link)

def get_links(html):
    """Return a list of links from html
    """
    # a regular expression to extract all links from the webpage
    webpage_regex = re.compile('<a[^>]+href=["\'](.*?)["\']',
        re.IGNORECASE)
    # list of all links from the webpage
    return webpage_regex.findall(html)
```

To run this code, simply call the `link_crawler` function with the URL of the website you want to crawl and a regular expression of the links that you need to follow. For the example website, we want to crawl the index with the list of countries and the countries themselves. The index links follow this format:

- `http://example.webscraping.com/index/1`
- `http://example.webscraping.com/index/2`

The country web pages will follow this format:

- `http://example.webscraping.com/view/Afghanistan-1`
- `http://example.webscraping.com/view/Aland-Islands-2`

So a simple regular expression to match both types of web pages is `/(index|view)/`. What happens when the crawler is run with these inputs? You would find that we get the following `download` error:

```
>>> link_crawler('http://example.webscraping.com',
    'example.webscraping.com/(index|view)/')
Downloading: http://example.webscraping.com
Downloading: /index/1
Traceback (most recent call last):
  ...
ValueError: unknown url type: /index/1
```

The problem with downloading `/index/1` is that it only includes the path of the web page and leaves out the protocol and server, which is known as a **relative link**. Relative links work when browsing because the web browser knows which web page you are currently viewing. However, `urllib2` is not aware of this context. To help `urllib2` locate the web page, we need to convert this link into an **absolute link**, which includes all the details to locate the web page. As might be expected, Python includes a module to do just this, called `urlparse`. Here is an improved version of `link_crawler` that uses the `urlparse` module to create the absolute links:

```
import urlparse
def link_crawler(seed_url, link_regex):
    """Crawl from the given seed URL following links matched by link_regex
    """
    crawl_queue = [seed_url]
    while crawl_queue:
        url = crawl_queue.pop()
        html = download(url)
        for link in get_links(html):
            if re.match(link_regex, link):
                link = urlparse.urljoin(seed_url, link)
                crawl_queue.append(link)
```

When this example is run, you will find that it downloads the web pages without errors; however, it keeps downloading the same locations over and over. The reason for this is that these locations have links to each other. For example, Australia links to Antarctica and Antarctica links right back, and the crawler will cycle between these forever. To prevent re-crawling the same links, we need to keep track of what has already been crawled. Here is the updated version of `link_crawler` that stores the URLs seen before, to avoid redownloading duplicates:

```
def link_crawler(seed_url, link_regex):
    crawl_queue = [seed_url]
    # keep track which URL's have seen before
    seen = set(crawl_queue)
    while crawl_queue:
        url = crawl_queue.pop()
        html = download(url)
        for link in get_links(html):
            # check if link matches expected regex
            if re.match(link_regex, link):
                # form absolute link
                link = urlparse.urljoin(seed_url, link)
                # check if have already seen this link
                if link not in seen:
                    seen.add(link)
                    crawl_queue.append(link)
```

When this script is run, it will crawl the locations and then stop as expected. We finally have a working crawler!

Advanced features

Now, let's add some features to make our link crawler more useful for crawling other websites.

Parsing robots.txt

Firstly, we need to interpret `robots.txt` to avoid downloading blocked URLs. Python comes with the `robotparser` module, which makes this straightforward, as follows:

```
>>> import robotparser
>>> rp = robotparser.RobotFileParser()
>>> rp.set_url('http://example.webscraping.com/robots.txt')
>>> rp.read()
>>> url = 'http://example.webscraping.com'
```

```
>>> user_agent = 'BadCrawler'
>>> rp.can_fetch(user_agent, url)
False
>>> user_agent = 'GoodCrawler'
>>> rp.can_fetch(user_agent, url)
True
```

The `robotparser` module loads a `robots.txt` file and then provides a `can_fetch()` function, which tells you whether a particular user agent is allowed to access a web page or not. Here, when the user agent is set to `'BadCrawler'`, the `robotparser` module says that this web page can not be fetched, as was defined in `robots.txt` of the example website.

To integrate this into the crawler, we add this check in the `crawl` loop:

```
...
while crawl_queue:
    url = crawl_queue.pop()
    # check url passes robots.txt restrictions
    if rp.can_fetch(user_agent, url):
        ...
    else:
        print 'Blocked by robots.txt:', url
```

Supporting proxies

Sometimes it is necessary to access a website through a proxy. For example, Netflix is blocked in most countries outside the United States. Supporting proxies with `urllib2` is not as easy as it could be (for a more user-friendly Python HTTP module, try `requests`, documented at `http://docs.python-requests.org/`). Here is how to support a proxy with `urllib2`:

```
proxy = ...
opener = urllib2.build_opener()
proxy_params = {urlparse.urlparse(url).scheme: proxy}
opener.add_handler(urllib2.ProxyHandler(proxy_params))
response = opener.open(request)
```

Here is an updated version of the `download` function to integrate this:

```
def download(url, user_agent='wswp', proxy=None, num_retries=2):
    print 'Downloading:', url
    headers = {'User-agent': user_agent}
    request = urllib2.Request(url, headers=headers)
```

```
    opener = urllib2.build_opener()
    if proxy:
        proxy_params = {urlparse.urlparse(url).scheme: proxy}
        opener.add_handler(urllib2.ProxyHandler(proxy_params))
    try:
        html = opener.open(request).read()
    except urllib2.URLError as e:
        print 'Download error:', e.reason
        html = None
        if num_retries > 0:
            if hasattr(e, 'code') and 500 <= e.code < 600:
                # retry 5XX HTTP errors
                html = download(url, user_agent, proxy,
                    num_retries-1)
    return html
```

Throttling downloads

If we crawl a website too fast, we risk being blocked or overloading the server. To minimize these risks, we can throttle our crawl by waiting for a delay between downloads. Here is a class to implement this:

```
class Throttle:
    """Add a delay between downloads to the same domain
    """
    def __init__(self, delay):
        # amount of delay between downloads for each domain
        self.delay = delay
        # timestamp of when a domain was last accessed
        self.domains = {}

    def wait(self, url):
        domain = urlparse.urlparse(url).netloc
        last_accessed = self.domains.get(domain)

        if self.delay > 0 and last_accessed is not None:
            sleep_secs = self.delay - (datetime.datetime.now() -
                last_accessed).seconds
            if sleep_secs > 0:
                # domain has been accessed recently
                # so need to sleep
                time.sleep(sleep_secs)
        # update the last accessed time
        self.domains[domain] = datetime.datetime.now()
```

This `Throttle` class keeps track of when each domain was last accessed and will sleep if the time since the last access is shorter than the specified delay. We can add throttling to the crawler by calling `throttle` before every download:

```
throttle = Throttle(delay)
...
throttle.wait(url)
result = download(url, headers, proxy=proxy,
    num_retries=num_retries)
```

Avoiding spider traps

Currently, our crawler will follow any link that it has not seen before. However, some websites dynamically generate their content and can have an infinite number of web pages. For example, if the website has an online calendar with links provided for the next month and year, then the next month will also have links to the next month, and so on for eternity. This situation is known as a **spider trap**.

A simple way to avoid getting stuck in a spider trap is to track how many links have been followed to reach the current web page, which we will refer to as `depth`. Then, when a maximum depth is reached, the crawler does not add links from this web page to the queue. To implement this, we will change the `seen` variable, which currently tracks the visited web pages, into a dictionary to also record the depth they were found at:

```
def link_crawler(..., max_depth=2):
    max_depth = 2
    seen = {}
    ...
    depth = seen[url]
    if depth != max_depth:
        for link in links:
            if link not in seen:
                seen[link] = depth + 1
                crawl_queue.append(link)
```

Now, with this feature, we can be confident that the crawl will always complete eventually. To disable this feature, `max_depth` can be set to a negative number so that the current depth is never equal to it.

Final version

The full source code for this advanced link crawler can be downloaded at `https://bitbucket.org/wswp/code/src/tip/chapter01/link_crawler3.py`. To test this, let us try setting the user agent to `BadCrawler`, which we saw earlier in this chapter was blocked by `robots.txt`. As expected, the crawl is blocked and finishes immediately:

```
>>> seed_url = 'http://example.webscraping.com/index'
>>> link_regex = '/(index|view)'
>>> link_crawler(seed_url, link_regex, user_agent='BadCrawler')
Blocked by robots.txt: http://example.webscraping.com/
```

Now, let's try using the default user agent and setting the maximum depth to `1` so that only the links from the home page are downloaded:

```
>>> link_crawler(seed_url, link_regex, max_depth=1)
Downloading: http://example.webscraping.com//index
Downloading: http://example.webscraping.com/index/1
Downloading: http://example.webscraping.com/view/Antigua-and-Barbuda-10
Downloading: http://example.webscraping.com/view/Antarctica-9
Downloading: http://example.webscraping.com/view/Anguilla-8
Downloading: http://example.webscraping.com/view/Angola-7
Downloading: http://example.webscraping.com/view/Andorra-6
Downloading: http://example.webscraping.com/view/American-Samoa-5
Downloading: http://example.webscraping.com/view/Algeria-4
Downloading: http://example.webscraping.com/view/Albania-3
Downloading: http://example.webscraping.com/view/Aland-Islands-2
Downloading: http://example.webscraping.com/view/Afghanistan-1
```

As expected, the crawl stopped after downloading the first page of countries.

Summary

This chapter introduced web scraping and developed a sophisticated crawler that will be reused in the following chapters. We covered the usage of external tools and modules to get an understanding of a website, user agents, sitemaps, crawl delays, and various crawling strategies.

In the next chapter, we will explore how to scrape data from the crawled web pages.

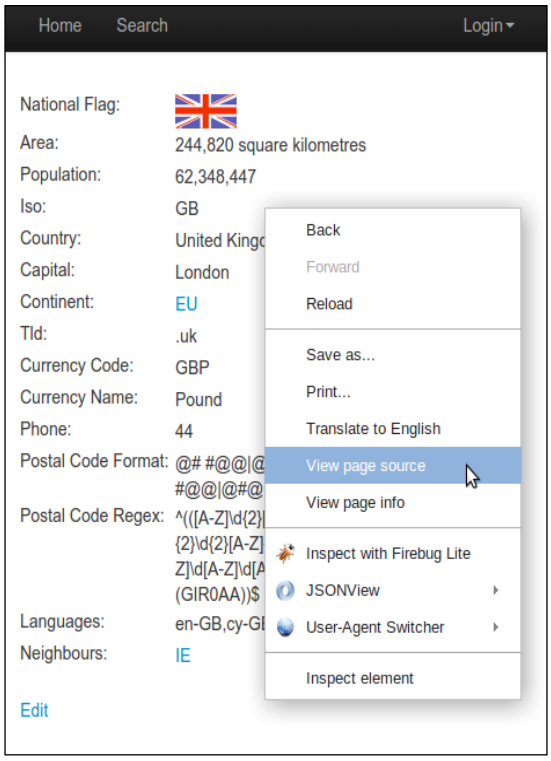
2
Scraping the Data

In the preceding chapter, we built a crawler that follows links to download the web pages we want. This is interesting but not useful—the crawler downloads a web page, and then discards the result. Now, we need to make this crawler achieve something by extracting data from each web page, which is known as **scraping**.

We will first cover a browser extension called **Firebug Lite** to examine a web page, which you may already be familiar with if you have a web development background. Then, we will walk through three approaches to extract data from a web page using regular expressions, Beautiful Soup and lxml. Finally, the chapter will conclude with a comparison of these three scraping alternatives.

Analyzing a web page

To understand how a web page is structured, we can try examining the source code. In most web browsers, the source code of a web page can be viewed by right-clicking on the page and selecting the **View page source** option:

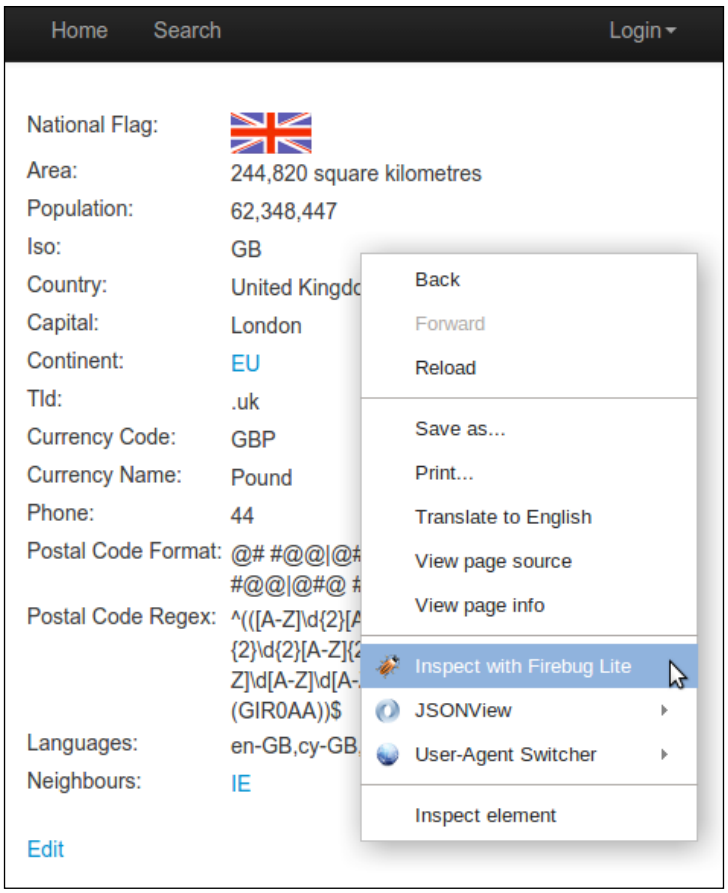


The data we are interested in is found in this part of the HTML:

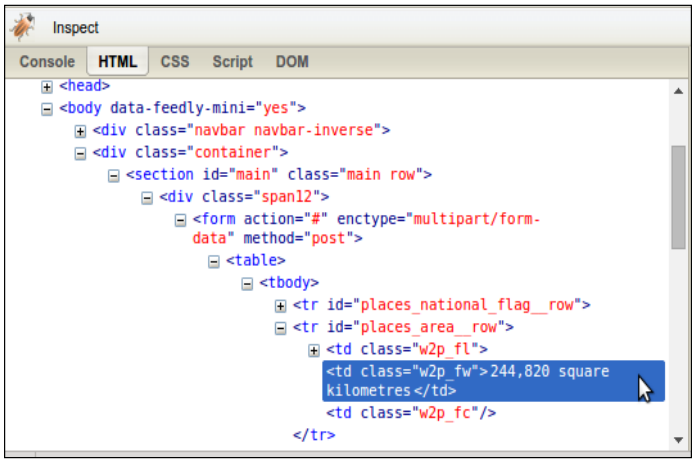
```
<table>
<tr id="places_national_flag__row"><td class="w2p_fl"><label
  for="places_national_flag"
    id="places_national_flag__label">National Flag:
      </label></td><td class="w2p_fw"><img
        src="/places/static/images/flags/gb.png" /></td><td
          class="w2p_fc"></td></tr>
...
<tr id="places_neighbours__row"><td class="w2p_fl"><label
  for="places_neighbours"
    id="places_neighbours__label">Neighbours: </label></td><td
      class="w2p_fw"><div><a href="/iso/IE">IE </a></div></td><td
        class="w2p_fc"></td></tr></table>
```

This lack of whitespace and formatting is not an issue for a web browser to interpret, but it is difficult for us. To help us interpret this table, we will use the Firebug Lite extension, which is available for all web browsers at `https://getfirebug.com/firebuglite`. Firefox users can install the full Firebug extension if preferred, but the features we will use here, and in *Chapter 6*, *Interacting with Forms* on dynamic content, are included in the Lite version.

Now, with Firebug Lite installed, we can right-click on the part of the web page we are interested in scraping and select **Inspect with Firebug Lite** from the context menu, as shown here:



This will open a panel showing the surrounding HTML hierarchy of the selected element:



In the preceding screenshot, the country attribute was clicked on and the Firebug panel makes it clear that the country area figure is included within a `<td>` element of class `w2p_fw`, which is the child of a `<tr>` element of ID `places_area__row`. We now have all the information needed to scrape the area data.

Three approaches to scrape a web page

Now that we understand the structure of this web page we will investigate three different approaches to scraping its data, firstly with regular expressions, then with the popular `BeautifulSoup` module, and finally with the powerful `lxml` module.

Regular expressions

If you are unfamiliar with regular expressions or need a reminder, there is a thorough overview available at `https://docs.python.org/2/howto/regex.html`.

To scrape the area using regular expressions, we will first try matching the contents of the `<td>` element, as follows:

```
>>> import re
>>> url = 'http://example.webscraping.com/view/United
        Kingdom-239'
>>> html = download(url)
>>> re.findall('<td class="w2p_fw">(.*?)</td>', html)
```

```
['<img src="/places/static/images/flags/gb.png" />',
  '244,820 square kilometres',
  '62,348,447',
  'GB',
  'United Kingdom',
  'London',
  '<a href="/continent/EU">EU</a>',
  '.uk',
  'GBP',
  'Pound',
  '44',
  '@# #@@|@## #@@|@@# #@@|@@## #@@|@#@ #@@|@@#@ #@@|GIR0AA',
  '^(([A-Z]\\d{2}[A-Z]{2})|([A-Z]\\d{3}[A-Z]{2})|([A-Z]{2}\\d{2}
    [A-Z]{2})|([A-Z]{2}\\d{3}[A-Z]{2})|([A-Z]\\d[A-Z]\\d[A-Z]{2})
      |([A-Z]{2}\\d[A-Z]\\d[A-Z]{2})|(GIR0AA))$',
  'en-GB,cy-GB,gd',
  '<div><a href="/iso/IE">IE </a></div>']
```

This result shows that the `<td class="w2p_fw">` tag is used for multiple country attributes. To isolate the area, we can select the second element, as follows:

```
>>> re.findall('<td class="w2p_fw">(.*?)</td>', html)[1]
'244,820 square kilometres'
```

This solution works but could easily fail if the web page is updated. Consider if this table is changed so that the population data is no longer available in the second row. If we just need to scrape the data now, future changes can be ignored. However, if we want to rescrape this data in future, we want our solution to be as robust against layout changes as possible. To make this regular expression more robust, we can include the parent `<tr>` element, which has an ID, so it ought to be unique:

```
>>> re.findall('<tr id="places_area__row"><td
  class="w2p_fl"><label for="places_area"
    id="places_area__label">Area: </label></td><td
      class="w2p_fw">(.*?)</td>', html)
['244,820 square kilometres']
```

This iteration is better; however, there are many other ways the web page could be updated in a way that still breaks the regular expression. For example, double quotation marks might be changed to single, extra space could be added between the `<td>` tags, or the `area_label` could be changed. Here is an improved version to try and support these various possibilities:

```
>>> re.findall('<tr
  id="places_area__row">.*?<td\s*class=["\']w2p_fw["\']>(.*?)
    </td>', html)
['244,820 square kilometres']
```

This regular expression is more future-proof but is difficult to construct, becoming unreadable. Also, there are still other minor layout changes that would break it, such as if a title attribute was added to the `<td>` tag.

From this example, it is clear that regular expressions provide a quick way to scrape data but are too brittle and will easily break when a web page is updated. Fortunately, there are better solutions.

Beautiful Soup

Beautiful Soup is a popular module that parses a web page and then provides a convenient interface to navigate content. If you do not already have this module, the latest version can be installed using this command:

```
pip install beautifulsoup4
```

The first step with Beautiful Soup is to parse the downloaded HTML into a soup document. Most web pages do not contain perfectly valid HTML and Beautiful Soup needs to decide what is intended. For example, consider this simple web page of a list with missing attribute quotes and closing tags:

```
<ul class=country>
    <li>Area
    <li>Population
</ul>
```

If the `Population` item is interpreted as a child of the `Area` item instead of the list, we could get unexpected results when scraping. Let us see how Beautiful Soup handles this:

```
>>> from bs4 import BeautifulSoup
>>> broken_html = '<ul class=country><li>Area<li>Population</ul>'
>>> # parse the HTML
>>> soup = BeautifulSoup(broken_html, 'html.parser')
>>> fixed_html = soup.prettify()
>>> print fixed_html
<html>
    <body>
        <ul class="country">
            <li>Area</li>
            <li>Population</li>
        </ul>
    </body>
</html>
```

Here, `BeautifulSoup` was able to correctly interpret the missing attribute quotes and closing tags, as well as add the `<html>` and `<body>` tags to form a complete HTML document. Now, we can navigate to the elements we want using the `find()` and `find_all()` methods:

```
>>> ul = soup.find('ul', attrs={'class':'country'})
>>> ul.find('li')  # returns just the first match
<li>Area</li>
>>> ul.find_all('li')  # returns all matches
[<li>Area</li>, <li>Population</li>]
```

For a full list of available methods and parameters, the official documentation is available at `http://www.crummy.com/software/BeautifulSoup/bs4/doc/`.

Now, using these techniques, here is a full example to extract the area from our example country:

```
>>> from bs4 import BeautifulSoup
>>> url = 'http://example.webscraping.com/places/view/
        United-Kingdom-239'
>>> html = download(url)
>>> soup = BeautifulSoup(html)
>>> # locate the area row
>>> tr = soup.find(attrs={'id':'places_area__row'})
>>> td = tr.find(attrs={'class':'w2p_fw'})  # locate the area tag
>>> area = td.text  # extract the text from this tag
>>> print area
244,820 square kilometres
```

This code is more verbose than regular expressions but easier to construct and understand. Also, we no longer need to worry about problems in minor layout changes, such as extra whitespace or tag attributes.

Lxml

Lxml is a Python wrapper on top of the `libxml2` XML parsing library written in C, which helps make it faster than Beautiful Soup but also harder to install on some computers. The latest installation instructions are available at `http://lxml.de/installation.html`.

As with Beautiful Soup, the first step is parsing the potentially invalid HTML into a consistent format. Here is an example of parsing the same broken HTML:

```
>>> import lxml.html
>>> broken_html = '<ul class=country><li>Area<li>Population</ul>'
>>> tree = lxml.html.fromstring(broken_html)  # parse the HTML
>>> fixed_html = lxml.html.tostring(tree, pretty_print=True)
>>> print fixed_html
<ul class="country">
    <li>Area</li>
    <li>Population</li>
</ul>
```

As with `BeautifulSoup`, `lxml` was able to correctly parse the missing attribute quotes and closing tags, although it did not add the `<html>` and `<body>` tags.

After parsing the input, `lxml` has a number of different options to select elements, such as XPath selectors and a `find()` method similar to Beautiful Soup. Instead, we will use CSS selectors here and in future examples, because they are more compact and can be reused later in *Chapter 5*, *Dynamic Content* when parsing dynamic content. Also, some readers will already be familiar with them from their experience with jQuery selectors.

Here is an example using the `lxml` CSS selectors to extract the area data:

```
>>> tree = lxml.html.fromstring(html)
>>> td = tree.cssselect('tr#places_area__row > td.w2p_fw')[0]
>>> area = td.text_content()
>>> print area
244,820 square kilometres
```

The key line with the CSS selector is highlighted. This line finds a table row element with the `places_area__row` ID, and then selects the child table data tag with the `w2p_fw` class.

CSS selectors

CSS selectors are patterns used for selecting elements. Here are some examples of common selectors you will need:

```
Select any tag: *
Select by tag <a>: a
Select by class of "link": .link
Select by tag <a> with class "link": a.link
Select by tag <a> with ID "home": a#home
Select by child <span> of tag <a>: a > span
```

```
Select by descendant <span> of tag <a>: a span
Select by tag <a> with attribute title of "Home": a[title=Home]
```

The CSS3 specification was produced by the W3C and is available for viewing at `http://www.w3.org/TR/2011/REC-css3-selectors-20110929/`.

Lxml implements most of CSS3, and details on unsupported features are available at `https://pythonhosted.org/cssselect/#supported-selectors`.

Note that, internally, `lxml` converts the CSS selectors into an equivalent XPath.

Comparing performance

To help evaluate the trade-offs of the three scraping approaches described in this chapter, it would help to compare their relative efficiency. Typically, a scraper would extract multiple fields from a web page. So, for a more realistic comparison, we will implement extended versions of each scraper that extract all the available data from a country's web page. To get started, we need to return to Firebug to check the format of the other country features, as shown here:

Firebug shows that each table row has an ID starting with `places_` and ending with `__row`. Then, the country data is contained within these rows in the same format as the earlier area example. Here are implementations that use this information to extract all of the available country data:

```
FIELDS = ('area', 'population', 'iso', 'country', 'capital',
  'continent', 'tld', 'currency_code', 'currency_name', 'phone',
    'postal_code_format', 'postal_code_regex', 'languages',
      'neighbours')

import re
def re_scraper(html):
    results = {}
    for field in FIELDS:
        results[field] = re.search('<tr id="places_%s__row">.*?<td
            class="w2p_fw">(.*?)</td>' % field, html).groups()[0]
    return results

from bs4 import BeautifulSoup
def bs_scraper(html):
    soup = BeautifulSoup(html, 'html.parser')
    results = {}
    for field in FIELDS:
        results[field] = soup.find('table').find('tr',
            id='places_%s__row' % field).find('td',
                class_='w2p_fw').text
    return results

import lxml.html
def lxml_scraper(html):
    tree = lxml.html.fromstring(html)
    results = {}
    for field in FIELDS:
        results[field] = tree.cssselect('table > tr#places_%s__row
            > td.w2p_fw' % field)[0].text_content()
    return results
```

Scraping results

Now that we have complete implementations for each scraper, we will test their relative performance with this snippet:

```
import time
NUM_ITERATIONS = 1000 # number of times to test each scraper
html = download('http://example.webscraping.com/places/view/
```

```
    United-Kingdom-239')
for name, scraper in [('Regular expressions', re_scraper),
    ('BeautifulSoup', bs_scraper),
    ('Lxml', lxml_scraper)]:
    # record start time of scrape
    start = time.time()
    for i in range(NUM_ITERATIONS):
        if scraper == re_scraper:
            re.purge()
        result = scraper(html)
        # check scraped result is as expected
        assert(result['area'] == '244,820 square kilometres')
    # record end time of scrape and output the total
    end = time.time()
    print '%s: %.2f seconds' % (name, end - start)
```

This example will run each scraper 1000 times, check whether the scraped results are as expected, and then print the total time taken. The `download` function used here is the one defined in the preceding chapter. Note the highlighted line calling `re.purge()`; by default, the regular expression module will cache searches and this cache needs to be cleared to make a fair comparison with the other scraping approaches.

Here are the results from running this script on my computer:

```
$ python performance.py
Regular expressions: 5.50 seconds
BeautifulSoup: 42.84 seconds
Lxml: 7.06 seconds
```

The results on your computer will quite likely be different because of the different hardware used. However, the relative difference between each approach should be equivalent. The results show that Beautiful Soup is over six times slower than the other two approaches when used to scrape our example web page. This result could be anticipated because `lxml` and the regular expression module were written in C, while `BeautifulSoup` is pure Python. An interesting fact is that `lxml` performed comparatively well with regular expressions, since `lxml` has the additional overhead of having to parse the input into its internal format before searching for elements. When scraping many features from a web page, this initial parsing overhead is reduced and `lxml` becomes even more competitive. It really is an amazing module!

Overview

The following table summarizes the advantages and disadvantages of each approach to scraping:

Scraping approach	Performance	Ease of use	Ease to install
Regular expressions	Fast	Hard	Easy (built-in module)
Beautiful Soup	Slow	Easy	Easy (pure Python)
Lxml	Fast	Easy	Moderately difficult

If the bottleneck to your scraper is downloading web pages rather than extracting data, it would not be a problem to use a slower approach, such as Beautiful Soup. Or, if you just need to scrape a small amount of data and want to avoid additional dependencies, regular expressions might be an appropriate choice. However, in general, `lxml` is the best choice for scraping, because it is fast and robust, while regular expressions and Beautiful Soup are only useful in certain niches.

Adding a scrape callback to the link crawler

Now that we know how to scrape the country data, we can integrate this into the link crawler built in the preceding chapter. To allow reusing the same crawling code to scrape multiple websites, we will add a `callback` parameter to handle the scraping. A `callback` is a function that will be called after certain events (in this case, after a web page has been downloaded). This scrape `callback` will take a `url` and `html` as parameters and optionally return a list of further URLs to crawl. Here is the implementation, which is simple in Python:

```
def link_crawler(..., scrape_callback=None):
    ...
    links = []
    if scrape_callback:
        links.extend(scrape_callback(url, html) or [])
        ...
```

The new code for the scraping `callback` function are highlighted in the preceding snippet, and the full source code for this version of the link crawler is available at `https://bitbucket.org/wswp/code/src/tip/chapter02/link_crawler.py`.

Now, this crawler can be used to scrape multiple websites by customizing the function passed to `scrape_callback`. Here is a modified version of the `lxml` example scraper that can be used for the `callback` function:

```
def scrape_callback(url, html):
    if re.search('/view/', url):
        tree = lxml.html.fromstring(html)
        row = [tree.cssselect('table > tr#places_%s__row >
            td.w2p_fw' % field)[0].text_content() for field in
                FIELDS]
        print url, row
```

This `callback` function would scrape the country data and print it out. Usually, when scraping a website, we want to reuse the data, so we will extend this example to save results to a CSV spreadsheet, as follows:

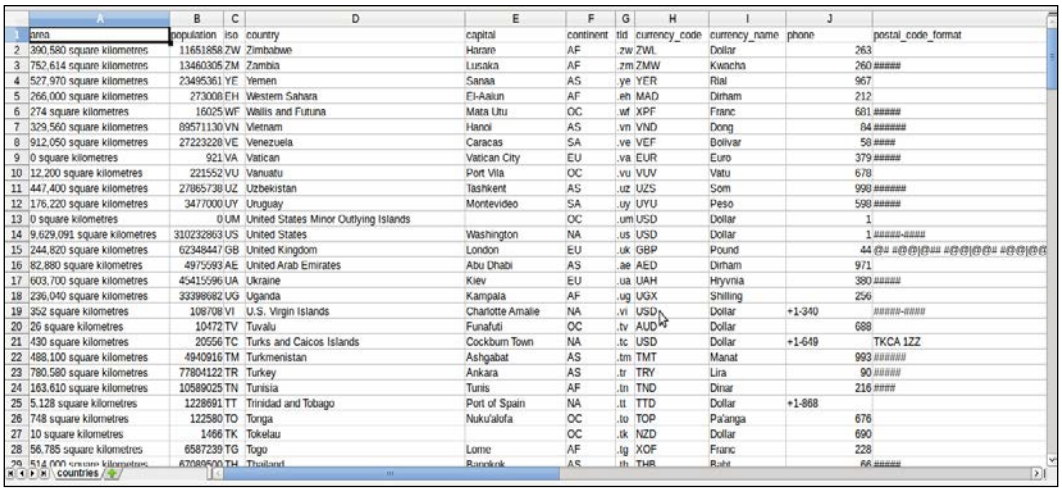
```
import csv
class ScrapeCallback:
    def __init__(self):
        self.writer = csv.writer(open('countries.csv', 'w'))
        self.fields = ('area', 'population', 'iso', 'country',
            'capital', 'continent', 'tld', 'currency_code',
                'currency_name', 'phone', 'postal_code_format',
                    'postal_code_regex', 'languages',
                        'neighbours')
        self.writer.writerow(self.fields)

    def __call__(self, url, html):
        if re.search('/view/', url):
            tree = lxml.html.fromstring(html)
            row = []
            for field in self.fields:
                row.append(tree.cssselect('table >
                    tr#places_{}__row >
                        td.w2p_fw'.format(field))
                            [0].text_content())
            self.writer.writerow(row)
```

To build this `callback`, a class was used instead of a function so that the state of the `csv` writer could be maintained. This `csv` writer is instantiated in the constructor, and then written to multiple times in the `__call__` method. Note that `__call__` is a special method that is invoked when an object is "called" as a function, which is how the `cache_callback` is used in the link crawler. This means that `scrape_callback(url, html)` is equivalent to calling `scrape_callback.__call__(url, html)`. For further details on Python's special class methods, refer to `https://docs.python.org/2/reference/datamodel.html#special-method-names`.

Here is how to pass this callback to the link crawler:

```
link_crawler('http://example.webscraping.com/', '/(index|view)',
    max_depth=-1, scrape_callback=ScrapeCallback())
```

Now, when the crawler is run with this `callback`, it will save results to a CSV file that can be viewed in an application such as Excel or LibreOffice:



	A	B	C	D	E	F	G	H	I	J	
1	area	population	iso	country	capital	continent	tld	currency_code	currency_name	phone	postal_code_format
2	390,580 square kilometres	11651858	ZW	Zimbabwe	Harare	AF	.zw	ZWL	Dollar	263	
3	752,614 square kilometres	13460305	ZM	Zambia	Lusaka	AF	.zm	ZMW	Kwacha	260	#####
4	527,970 square kilometres	23495361	YE	Yemen	Sanaa	AS	.ye	YER	Rial	967	
5	266,000 square kilometres	273008	EH	Western Sahara	El-Aaiun	AF	.eh	MAD	Dirham	212	
6	274 square kilometres	16025	WF	Wallis and Futuna	Mata Utu	OC	.wf	XPF	Franc	681	#####
7	329,560 square kilometres	89571130	VN	Vietnam	Hanoi	AS	.vn	VND	Dong	84	######
8	912,050 square kilometres	27223228	VE	Venezuela	Caracas	SA	.ve	VEF	Bolivar	58	####
9	0 square kilometres	921	VA	Vatican	Vatican City	EU	.va	EUR	Euro	379	#####
10	12,200 square kilometres	221552	VU	Vanuatu	Port Vila	OC	.vu	VUV	Vatu	678	
11	447,400 square kilometres	27865738	UZ	Uzbekistan	Tashkent	AS	.uz	UZS	Som	998	######
12	176,220 square kilometres	3477000	UY	Uruguay	Montevideo	SA	.uy	UYU	Peso	598	#####
13	0 square kilometres	0	UM	United States Minor Outlying Islands		OC	.um	USD	Dollar	1	
14	9,629,091 square kilometres	310232863	US	United States	Washington	NA	.us	USD	Dollar	1	#####-####
15	244,820 square kilometres	62348447	GB	United Kingdom	London	EU	.uk	GBP	Pound	44	@# #@@@## #@@@@# #@@@@#
16	82,880 square kilometres	4975593	AE	United Arab Emirates	Abu Dhabi	AS	.ae	AED	Dirham	971	
17	603,700 square kilometres	45415596	UA	Ukraine	Kiev	EU	.ua	UAH	Hryvnia	380	#####
18	236,040 square kilometres	33398682	UG	Uganda	Kampala	AF	.ug	UGX	Shilling	256	
19	352 square kilometres	108708	VI	U.S. Virgin Islands	Charlotte Amalie	NA	.vi	USD	Dollar	+1-340	#####-####
20	26 square kilometres	10472	TV	Tuvalu	Funafuti	OC	.tv	AUD	Dollar	688	
21	430 square kilometres	20556	TC	Turks and Caicos Islands	Cockburn Town	NA	.tc	USD	Dollar	+1-649	TKCA 1ZZ
22	488,100 square kilometres	4940916	TM	Turkmenistan	Ashgabat	AS	.tm	TMT	Manat	993	######
23	780,580 square kilometres	77804122	TR	Turkey	Ankara	AS	.tr	TRY	Lira	90	#####
24	163,610 square kilometres	10589025	TN	Tunisia	Tunis	AF	.tn	TND	Dinar	216	####
25	5,128 square kilometres	1228691	TT	Trinidad and Tobago	Port of Spain	NA	.tt	TTD	Dollar	+1-868	
26	748 square kilometres	122580	TO	Tonga	Nuku'alofa	OC	.to	TOP	Pa'anga	676	
27	10 square kilometres	1466	TK	Tokelau		OC	.tk	NZD	Dollar	690	
28	56,785 square kilometres	6587239	TG	Togo	Lome	AF	.tg	XOF	Franc	228	
29	514,000 square kilometres	67089500	TH	Thailand	Bangkok	AS	.th	THB	Baht	66	#####

countries

Success! We have completed our first working scraper.

Summary

In this chapter, we walked through a variety of ways to scrape data from a web page. Regular expressions can be useful for a one-off scrape or to avoid the overhead of parsing the entire web page, and `BeautifulSoup` provides a high-level interface while avoiding any difficult dependencies. However, in general, `lxml` will be the best choice because of its speed and extensive functionality, so we will use it in future examples.

In the next chapter we will introduce caching, which allows us to save web pages so that they only need be downloaded the first time a crawler is run.

3
Caching Downloads

In the preceding chapter, we learned how to scrape data from crawled web pages and save the results to a spreadsheet. What if we now want to scrape an additional field, such as the flag URL? To scrape additional fields, we would need to download the entire website again. This is not a significant obstacle for our small example website. However, other websites can have millions of web pages that would take weeks to recrawl. The solution presented in this chapter is to cache all the crawled web pages so that they only need to be downloaded once.

Adding cache support to the link crawler

To support caching, the `download` function developed in *Chapter 1*, *Introduction to Web Scraping*, needs to be modified to check the cache before downloading a URL. We also need to move throttling inside this function and only throttle when a download is made, and not when loading from a cache. To avoid the need to pass various parameters for every download, we will take this opportunity to refactor the `download` function into a class, so that parameters can be set once in the constructor and reused multiple times. Here is the updated implementation to support this:

```
class Downloader:
    def __init__(self, delay=5,
        user_agent='wswp', proxies=None,
            num_retries=1, cache=None):
        self.throttle = Throttle(delay)
        self.user_agent = user_agent
        self.proxies = proxies
        self.num_retries = num_retries
        self.cache = cache

    def __call__(self, url):
        result = None
```

```
        if self.cache:
            try:
                result = self.cache[url]
            except KeyError:
                # url is not available in cache
                pass
            else:
                if self.num_retries > 0 and \
                    500 <= result['code'] < 600:
                    # server error so ignore result from cache
                    # and re-download
                    result = None
        if result is None:
            # result was not loaded from cache
            # so still need to download
            self.throttle.wait(url)
            proxy = random.choice(self.proxies) if self.proxies
                else None
            headers = {'User-agent': self.user_agent}
            result = self.download(url, headers, proxy,
                self.num_retries)
            if self.cache:
                # save result to cache
                self.cache[url] = result
        return result['html']


    def download(self, url, headers, proxy, num_retries,
        data=None):
        ...
        return {'html': html, 'code': code}
```

The full source code for this class is available at `https://bitbucket.org/wswp/code/src/tip/chapter03/downloader.py`

The interesting part of the `Download` class used in the preceding code is in the `__call__` special method, where the cache is checked before downloading. This method first checks whether the cache is defined. If so, it checks whether this URL was previously cached. If it is cached, it checks whether a server error was encountered in the previous download. Finally, if no server error was encountered, the cached result can be used. If any of these checks fail, the URL needs to be downloaded as usual, and the result will be added to the cache. The `download` method of this class is the same as the previous `download` function, except now it returns the HTTP status code along with the downloaded HTML so that error codes can be stored in the cache. If you just want a simple download without throttling or caching, this method can be used instead of `__call__`.

The `cache` class is used here by calling `result = cache[url]` to load from `cache` and `cache[url] = result` to save to `cache`, which is a convenient interface that you should be familiar with from Python's builtin dictionary data type. To support this interface, our `cache` class will need to define the `__getitem__()` and `__setitem__()` special class methods.

The link crawler also needs to be slightly updated to support caching by adding the `cache` parameter, removing the throttle, and replacing the `download` function with the new class, as shown in the following code:

```
def link_crawler(..., cache=None):
    crawl_queue = [seed_url]
    seen = {seed_url: 0}
    num_urls = 0
    rp = get_robots(seed_url)
    D = Downloader(delay=delay, user_agent=user_agent,
        proxies=proxies, num_retries=num_retries, cache=cache)

    while crawl_queue:
        url = crawl_queue.pop()
        depth = seen[url]
        # check url passes robots.txt restrictions
        if rp.can_fetch(user_agent, url):
            html = D(url)
            links = []
            ...
```

Now, our web scraping infrastructure is prepared, and we can start building the actual cache.

Disk cache

To cache downloads, we will first try the obvious solution and save web pages to the filesystem. To do this, we will need a way to map URLs to a safe cross-platform filename. The following table lists the limitations for some popular filesystems:

Operating system	File system	Invalid filename characters	Maximum filename length
Linux	Ext3/Ext4	/ and \0	255 bytes
OS X	HFS Plus	: and \0	255 UTF-16 code units
Windows	NTFS	\, /, ?, :, *, ", >, <, and |	255 characters

To keep our file path safe across these filesystems, it needs to be restricted to numbers, letters, basic punctuation, and replace all other characters with an underscore, as shown in the following code:

```
>>> import re
>>> url = 'http://example.webscraping.com/default/view/
    Australia-1'
>>> re.sub('[^/0-9a-zA-Z\-.,;_ ]', '_', url)
'http_//example.webscraping.com/default/view/Australia-1'
```

Additionally, the filename and the parent directories need to be restricted to 255 characters (as shown in the following code) to meet the length limitations described in the preceding table:

```
>>> filename = '/'.join(segment[:255] for segment in
    filename.split('/'))
```

There is also an edge case that needs to be considered, where the URL path ends with a slash (/), and the empty string after this slash would be an invalid filename. However, removing this slash to use the parent for the filename would prevent saving other URLs. Consider the following URLs:

- `http://example.webscraping.com/index/`
- `http://example.webscraping.com/index/1`

If you need to save these, then index needs to be a directory to save the child path for 1. The solution our disk cache will use is appending `index.html` to the filename when the URL path ends with a slash. The same applies when the URL path is empty. To parse the URL, we will use the `urlparse.urlsplit()` function, which splits a URL into its components:

```
>>> import urlparse
>>> components =
    urlparse.urlsplit('http://example.webscraping.com/index/')
>>> print components
SplitResult(scheme='http', netloc='example.webscraping.com',
    path='/index/', query='', fragment='')
>>> print components.path
'/index/'
```

This function provides a convenient interface to parse and manipulate URLs. Here is an example using this module to append `index.html` for this edge case:

```
>>> path = components.path
>>> if not path:
>>>     path = '/index.html'
>>> elif path.endswith('/'):
>>>     path += 'index.html'
```

```
>>> filename = components.netloc + path + components.query
>>> filename
'example.webscraping.com/index/index.html'
```

Implementation

In the preceding section, we covered the limitations of the filesystem that need to be considered when building a disk-based cache, namely the restriction on which characters can be used, the filename length, and making sure a file and directory are not created in the same location. Together, using this logic to map a URL to a filename will form the main part of the disk cache. Here is an initial implementation of the DiskCache class:

```
import os
import re
import urlparse

class DiskCache:
    def __init__(self, cache_dir='cache'):
        self.cache_dir = cache_dir
        self.max_length = max_length

    def url_to_path(self, url):
        """Create file system path for this URL
        """
        components = urlparse.urlsplit(url)
        # append index.html to empty paths
        path = components.path
        if not path:
            path = '/index.html'
        elif path.endswith('/'):
            path += 'index.html'
        filename = components.netloc + path + components.query
        # replace invalid characters
        filename = re.sub('[^/0-9a-zA-Z\-.,;_ ]', '_', filename)
        # restrict maximum number of characters
        filename = '/'.join(segment[:250] for segment in
            filename.split('/'))
        return os.path.join(self.cache_dir, filename)
```

The class constructor shown in the preceding code takes a parameter to set the location of the cache, and then the `url_to_path` method applies the filename restrictions that have been discussed so far. Now we just need methods to load and save the data with this filename. Here is an implementation of these missing methods:

```
import pickle
class DiskCache:
    ...
```

```
    def __getitem__(self, url):
        """Load data from disk for this URL
        """
        path = self.url_to_path(url)
        if os.path.exists(path):
            with open(path, 'rb') as fp:
                return pickle.load(fp)
        else:
            # URL has not yet been cached
            raise KeyError(url + ' does not exist')

    def __setitem__(self, url, result):
        """Save data to disk for this url
        """
        path = self.url_to_path(url)
        folder = os.path.dirname(path)
        if not os.path.exists(folder):
            os.makedirs(folder)
        with open(path, 'wb') as fp:
            fp.write(pickle.dumps(result))
```

In `__setitem__()`, the URL is mapped to a safe filename using `url_to_path()`, and then the parent directory is created if necessary. The `pickle` module is used to convert the input to a string, which is then saved to disk. Also, in `__getitem__()`, the URL is mapped to a safe filename. Then, if the filename exists, the content is loaded and unpickled to restore the original data type. If the filename does not exist, that is, there is no data in the cache for this URL, a `KeyError` exception is raised.

Testing the cache

Now we are ready to try `DiskCache` with our crawler by passing it to the `cache` callback. The source code for this class is available at `https://bitbucket.org/wswp/code/src/tip/chapter03/disk_cache.py` and the cache can be tested with the link crawler by running this script:

```
$ time python disk_cache.py
Downloading: http://example.webscraping.com
Downloading: http://example.webscraping.com/view/Afghanistan-1
...
Downloading: http://example.webscraping.com/view/Zimbabwe-252
23m38.289s
```

The first time this command is run, the cache is empty so that all the web pages are downloaded normally. However, when we run this script a second time, the pages will be loaded from the cache so that the crawl should be completed more quickly, as shown here:

```
$ time python disk_cache.py
0m0.186s
```

As expected, this time the crawl completed much faster. While downloading with an empty cache on my computer, the crawler took over 23 minutes, while the second time with a full cache in just 0.186 seconds (over 7000 times faster!). The exact time on your computer will differ, depending on your hardware. However, the disk cache will undoubtedly be faster.

Saving disk space

To minimize the amount of disk space required for our cache, we can compress the downloaded HTML file. This is straightforward to implement by compressing the pickled string with `zlib` before saving to disk, as follows:

```
fp.write(zlib.compress(pickle.dumps(result)))
```

Then, decompress the data loaded from the disk, as follows:

```
return pickle.loads(zlib.decompress(fp.read()))
```

With this addition of compressing each web page, the cache is reduced from 4.4 MB to 2.3 MB and takes 0.212 seconds to crawl the cached example website on my computer. This is marginally longer than 0.186 seconds with the uncompressed cache. So, if speed is important for your project, you may want to disable compression.

Expiring stale data

Our current version of the disk cache will save a value to disk for a key, and then return it whenever this key is requested in future. This functionality may not be ideal when caching web pages because online content changes, so the data in our cache would become out of date. In this section, we will add an expiration time to our cached data so that the crawler knows when to redownload a web page. To support storing the timestamp of when each web page was cached is straightforward. Here is an implementation of this:

```
from datetime import datetime, timedelta

class DiskCacke:
    def __init__(self, ..., expires=timedelta(days=30)):
```

```
        ...
        self.expires = expires

    def __getitem__(self, url):
        """Load data from disk for this URL
        """
        ...
            with open(path, 'rb') as fp:
                result, timestamp =
                    pickle.loads(zlib.decompress(fp.read()))
                if self.has_expired(timestamp):
                    raise KeyError(url + ' has expired')
                return result
        else:
            # URL has not yet been cached
            raise KeyError(url + ' does not exist')


    def __setitem__(self, url, result):
        """Save data to disk for this url
        """
        ...
        timestamp = datetime.utcnow()
        data = pickle.dumps((result, timestamp))
        with open(path, 'wb') as fp:
            fp.write(zlib.compress(data))

    def has_expired(self, timestamp):
        """Return whether this timestamp has expired
        """
        return datetime.utcnow() > timestamp + self.expires
```

In the constructor, the default expiration time is set to 30 days with a `timedelta` object. Then, the `__set__` method saves the current timestamp in the pickled data and the `__get__` method compares this to the expiration time. To test this expiration, we can try a short timeout of 5 seconds, as shown here:

```
>>> cache = DiskCache(expires=timedelta(seconds=5))
>>> url = 'http://example.webscraping.com'
>>> result = {'html': '...'}
>>> cache[url] = result
>>> cache[url]
{'html': '...'}
>>> import time; time.sleep(5)
>>> cache[url]
Traceback (most recent call last):
...
KeyError: 'http://example.webscraping.com has expired'
```

As expected, the cached result is initially available, and then, after sleeping for 5 seconds, calling the same key raises a `KeyError` to show this cached download has expired.

Drawbacks

Our disk-based caching system was relatively simple to implement, does not depend on installing additional modules, and the results are viewable in our file manager. However, it has the drawback of depending on the limitations of the local filesystem. Earlier in this chapter, we applied various restrictions to map the URL to a safe filename, but an unfortunate consequence of this is that some URLs will map to the same filename. For example, replacing unsupported characters in the following URLs would map them all to the same filename:

- `http://example.com/?a+b`
- `http://example.com/?a*b`
- `http://example.com/?a=b`
- `http://example.com/?a!b`

This means that if one of these URLs were cached, it would look like the other three URLs were cached too, because they map to the same filename. Alternatively, if some long URLs only differed after the 255th character, the chomped versions would also map to the same filename. This is a particularly important problem since there is no defined limit on the maximum length of a URL. Although, in practice, URLs over 2000 characters are rare and older versions of Internet Explorer did not support over 2083 characters.

A potential solution to avoid these limitations is by taking the hash of the URL and using this as the filename. This may be an improvement - however, then we will eventually face a larger problem that many filesystems have; that is, a limit on the number of files allowed per volume and per directory. If this cache is used in a FAT32 filesystem, the maximum number of files allowed per directory is just 65,535. This limitation could be avoided by splitting the cache across multiple directories, however filesystems can also limit the total number of files. My current `ext4` partition supports a little over 15 million files, whereas a large website may have excess of 100 million web pages. Unfortunately the `DiskCache` approach has too many limitations to be of general use. What we need instead is to combine the multiple cached web pages into a single file and index them with a `B+ tree` or similar. Instead of implementing our own, we will use an existing database in the next section.

Database cache

To avoid the anticipated limitations to our disk-based cache, we will now build our cache on top of an existing database system. When crawling, we may need to cache massive amounts of data and will not need any complex joins, so we will use a NoSQL database, which is easier to scale than a traditional relational database. Specifically, our cache will use MongoDB, which is currently the most popular NoSQL database.

What is NoSQL?

NoSQL stands for **Not Only SQL** and is a relatively new approach to database design. The traditional relational model used a fixed schema and splits the data into tables. However, with large datasets, the data is too big for a single server and needs to be scaled across multiple servers. This does not fit well with the relational model because, when querying multiple tables, the data will not necessarily be available on the same server. NoSQL databases, on the other hand, are generally schemaless and designed from the start to shard seamlessly across servers. There have been multiple approaches to achieve this that fit under the NoSQL umbrella. There are column data stores, such as HBase; key-value stores, such as Redis; document-oriented databases, such as MongoDB; and graph databases, such as Neo4j.

Installing MongoDB

MongoDB can be downloaded from `https://www.mongodb.org/downloads`. Then, the Python wrapper needs to be installed separately using this command:

```
pip install pymongo
```

To test whether the installation is working, start MongoDB locally using this command:

```
$ mongod -dbpath .
```

Then, try connecting to MongoDB from Python using the default MongoDB port:

```
>>> from pymongo import MongoClient
>>> client = MongoClient('localhost', 27017)
```

Overview of MongoDB

Here is an example of how to save some data to MongoDB and then load it:

```
>>> url = 'http://example.webscraping.com/view/United-Kingdom-239'
>>> html = '...'
>>> db = client.cache
>>> db.webpage.insert({'url': url, 'html': html})
ObjectId('5518c0644e0c87444c12a577')
>>> db.webpage.find_one(url=url)
{u'_id': ObjectId('5518c0644e0c87444c12a577'),
u'html': u'...',
u'url': u'http://example.webscraping.com/view/United-Kingdom-239'}
```

A problem with the preceding example is that if we now insert another document with the same URL, MongoDB will happily insert it for us, as follows:

```
>>> db.webpage.insert({'url': url, 'html': html})
>>> db.webpage.find(url=url).count()
2
```

Now we have multiple records for the same URL when we are only interested in storing the latest data. To prevent duplicates, we can set the ID to the URL and perform `upsert`, which means updating the existing record if it exists; otherwise, insert a new one, as shown here:

```
>>> self.db.webpage.update({'_id': url}, {'$set': {'html': html}},
    upsert=True)
>>> db.webpage.update({'_id': url}, {'$set': {'html': ''}},
    upsert=True)
>>> db.webpage.find_one({'_id': url})
{u'_id': u'http://example.webscraping.com/view/
    United-Kingdom-239', u'html': u'...'}
```

Now, when we try inserting a record with the same URL as shown in the following code, the content will be updated instead of creating duplicates:

```
>>> new_html = '<html></html>'
>>> db.webpage.update({'_id': example_url}, {'$set': {'html': new_
html}}, upsert=True)
>>> db.webpage.find_one({'_id': url})
{u'_id': u'http://example.webscraping.com/view/United-Kingdom-239',
u'html': u'<html></html>'}
>>> db.webpage.find({'_id': url}).count()
1
```

We can see that after adding this record, the HTML has been updated and the number of records for this URL is still 1.

The official MongoDB documentation, which is available at `http://docs.mongodb.org/manual/`, covers these features and others in detail.

MongoDB cache implementation

Now we are ready to build our cache on MongoDB using the same class interface as the earlier `DiskCache` class:

```
from datetime import datetime, timedelta
from pymongo import MongoClient

class MongoCache:
    def __init__(self, client=None, expires=timedelta(days=30)):
        # if a client object is not passed then try
        # connecting to mongodb at the default localhost port
        self.client = MongoClient('localhost', 27017)
            if client is None else client
        # create collection to store cached webpages,
        # which is the equivalent of a table
        # in a relational database
        self.db = client.cache
        # create index to expire cached webpages
        self.db.webpage.create_index('timestamp',
            expireAfterSeconds=expires.total_seconds())

    def __getitem__(self, url):
        """Load value at this URL
        """
        record = self.db.webpage.find_one({'_id': url})
        if record:
            return record['result']
        else:
            raise KeyError(url + ' does not exist')

    def __setitem__(self, url, result):
        """Save value for this URL
        """
        record = {'result': result, 'timestamp':
            datetime.utcnow()}
        self.db.webpage.update({'_id': url}, {'$set': record},
            upsert=True)
```

The `__getitem__` and `__setitem__` methods here should be familiar to you from the discussion on how to prevent duplicates in the preceding section. You may have also noticed that a `timestamp` index was created in the constructor. This is a handy MongoDB feature that will automatically delete records in a specified number of seconds after the given timestamp. This means that we do not need to manually check whether a record is still valid, as in the `DiskCache` class. Let's try it out with an empty timedelta object so that the record should be deleted immediately:

```
>>> cache = MongoCache(expires=timedelta())
>>> cache[url] = result
>>> cache[url]
```

The record is still there; it seems that our cache expiration is not working. The reason for this is that MongoDB runs a background task to check for expired records every minute, so this record has not yet been deleted. If we wait for a minute, we would find that the cache expiration is working:

```
>>> import time; time.sleep(60)
>>> cache[url]
Traceback (most recent call last):
...
KeyError: 'http://example.webscraping.com/view/United-Kingdom-239
    does not exist'
```

This means that our MongoDB cache will not expire records at exactly the time given, and there will be up to a 1 minute delay. However, since typically a cache expiration of several weeks or months would be used, this relatively small additional delay should not be an issue.

Compression

To make this cache feature complete with the original disk cache, we need to add one final feature: **compression**. This can be achieved in a similar way as the disk cache by pickling the data and then compressing with `zlib`, as follows:

```
import pickle
import zlib
from bson.binary import Binary

class MongoCache:
    def __getitem__(self, url):
        record = self.db.webpage.find_one({'_id': url})
        if record:
            return pickle.loads(zlib.decompress(record['result']))
        else:
```

```
            raise KeyError(url + ' does not exist')

    def __setitem__(self, url, result):
        record = {
            'result': Binary(zlib.compress(pickle.dumps(result))),
            'timestamp': datetime.utcnow()
        }
        self.db.webpage.update(
            {'_id': url}, {'$set': record}, upsert=True)
```

Testing the cache

The source code for the `MongoCache` class is available at `https://bitbucket.org/wswp/code/src/tip/chapter03/mongo_cache.py` and as with `DiskCache`, the cache can be tested with the link crawler by running this script:

```
$ time python mongo_cache.py
http://example.webscraping.com
http://example.webscraping.com/view/Afghanistan-1
...
http://example.webscraping.com/view/Zimbabwe-252
23m40.302s

$ time python mongo_cache.py
0.378s
```

The time taken here is double that for the disk cache. However, MongoDB does not suffer from filesystem limitations and will allow us to make a more efficient crawler in the next chapter, which deals with concurrency.

Summary

In this chapter, we learned that caching downloaded web pages will save time and minimize bandwidth when recrawling a website. The main drawback of this is that the cache takes up disk space, which can be minimized through compression. Additionally, building on top of an existing database system, such as MongoDB, can be used to avoid any filesystem limitations.

In the next chapter, we will add further functionalities to our crawler so that we can download multiple web pages concurrently and crawl faster.

4
Concurrent Downloading

In previous chapters, our crawlers downloaded web pages sequentially, waiting for each download to complete before starting the next one. Sequential downloading is fine for the relatively small example website but quickly becomes impractical for larger crawls. To crawl a large website of 1 million web pages at an average of one web page per second would take over 11 days of continuous downloading all day and night. This time can be significantly improved by downloading multiple web pages simultaneously.

This chapter will cover downloading web pages with multiple threads and processes, and then compare the performance to sequential downloading.

One million web pages

To test the performance of concurrent downloading, it would be preferable to have a larger target website. For this reason, we will use the Alexa list in this chapter, which tracks the top 1 million most popular websites according to users who have installed the Alexa Toolbar. Only a small percentage of people use this browser plugin, so the data is not authoritative, but is fine for our purposes.

These top 1 million web pages can be browsed on the Alexa website at `http://www.alexa.com/topsites`. Additionally, a compressed spreadsheet of this list is available at `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`, so scraping Alexa is not necessary.

Parsing the Alexa list

The Alexa list is provided in a spreadsheet with columns for the rank and domain:

	A	B
1	1	google.com
2	2	facebook.com
3	3	youtube.com
4	4	yahoo.com
5	5	baidu.com
6	6	wikipedia.org
7	7	amazon.com
8	8	twitter.com
9	9	taobao.com
10	10	qq.com

Extracting this data requires a number of steps, as follows:

1. Download the `.zip` file.
2. Extract the CSV file from this `.zip` file.
3. Parse the CSV file.
4. Iterate each row of the CSV file to extract the domain.

Here is an implementation to achieve this:

```
import csv
from zipfile import ZipFile
from StringIO import StringIO
from downloader import Downloader

D = Downloader()
zipped_data = D('http://s3.amazonaws.com/alexa-static/top-1m.csv.zip')
urls = [] # top 1 million URL's will be stored in this list
with ZipFile(StringIO(zipped_data)) as zf:
    csv_filename = zf.namelist()[0]
    for _, website in csv.reader(zf.open(csv_filename)):
        urls.append('http://' + website)
```

You may have noticed the downloaded zipped data is wrapped with the `StringIO` class and passed to `ZipFile`. This is necessary because `ZipFile` expects a file-like interface rather than a string. Next, the CSV filename is extracted from the filename list. The `.zip` file only contains a single file, so the first filename is selected. Then, this CSV file is iterated and the domain in the second column is added to the URL list. The `http://` protocol is prepended to the domains to make them valid URLs.

To reuse this function with the crawlers developed earlier, it needs to be modified to use the `scrape_callback` interface:

```
class AlexaCallback:
    def __init__(self, max_urls=1000):
        self.max_urls = max_urls
        self.seed_url = 'http://s3.amazonaws.com/alexa-static/
            top-1m.csv.zip'

    def __call__(self, url, html):
        if url == self.seed_url:
            urls = []
            with ZipFile(StringIO(html)) as zf:
                csv_filename = zf.namelist()[0]
                for _, website in
                    csv.reader(zf.open(csv_filename)):
                    urls.append('http://' + website)
                    if len(urls) == self.max_urls:
                        break
            return urls
```

A new input argument was added here called `max_urls`, which sets the number of URLs to extract from the Alexa file. By default, this is set to 1000 URLs, because downloading a million web pages takes a long time (as mentioned in the chapter introduction, over 11 days when downloaded sequentially).

Sequential crawler

Here is the code to use `AlexaCallback` with the link crawler developed earlier to download sequentially:

```
scrape_callback = AlexaCallback()
link_crawler(seed_url=scrape_callback.seed_url,
    cache_callback=MongoCache(),
    scrape_callback=scrape_callback)
```

This code is available at `https://bitbucket.org/wswp/code/src/tip/chapter04/sequential_test.py` and can be run from the command line as follows:

```
$ time python sequential_test.py
...
26m41.141s
```

This time is as expected for sequential downloading with an average of ~1.6 seconds per URL.

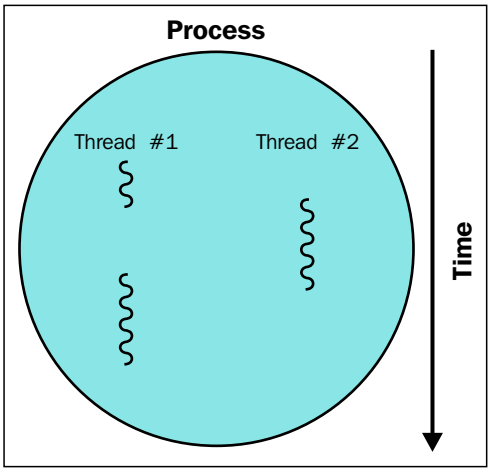
Threaded crawler

Now we will extend the sequential crawler to download the web pages in parallel. Note that if misused, a threaded crawler could request content too fast and overload a web server or cause your IP address to be blocked. To avoid this, our crawlers will have a `delay` flag to set the minimum number of seconds between requests to the same domain.

The Alexa list example used in this chapter covers 1 million separate domains, so this problem does not apply here. However, a delay of at least one second between downloads should be considered when crawling many web pages from a single domain in future.

How threads and processes work

Here is a diagram of a process containing multiple threads of execution:



When a Python script or other computer program is run, a process is created containing the code and state. These processes are executed by the CPU(s) of a computer. However, each CPU can only execute a single process at a time and will quickly switch between them to give the impression that multiple programs are running simultaneously. Similarly, within a process, the program execution can switch between multiple threads, with each thread executing different parts of the program. This means that when one thread is waiting for a web page to download, the process can switch and execute another thread to avoid wasting CPU time. So, using all the resources of our computer to download data as fast as possible requires distributing our downloads across multiple threads and processes.

Implementation

Fortunately, Python makes threading relatively straightforward. This means we can keep a similar queuing structure to the link crawler developed in *Chapter 1, Introduction to Web Scraping*, but start the crawl loop in multiple threads to download these links in parallel. Here is a modified version of the start of the link crawler with the `crawl` loop moved into a function:

```
import time
import threading
from downloader import Downloader
SLEEP_TIME = 1

def threaded_crawler(..., max_threads=10):
    # the queue of URL's that still need to be crawled
    crawl_queue = [seed_url]
    # the URL's that have been seen
    seen = set([seed_url])
    D = Downloader(cache=cache, delay=delay,
        user_agent=user_agent, proxies=proxies,
            num_retries=num_retries, timeout=timeout)

    def process_queue():
        while True:
            try:
                url = crawl_queue.pop()
            except IndexError:
                # crawl queue is empty
                break
            else:
                html = D(url)
                ...
```

Here is the remainder of the `threaded_crawler` function to start `process_queue` in multiple threads and wait until they have completed:

```
    threads = []
    while threads or crawl_queue:
        # the crawl is still active
        for thread in threads:
            if not thread.is_alive():
                # remove the stopped threads
                threads.remove(thread)
        while len(threads) < max_threads and crawl_queue:
            # can start some more threads
            thread = threading.Thread(target=process_queue)
            # set daemon so main thread can exit when receives ctrl-c
            thread.setDaemon(True)
            thread.start()
            threads.append(thread)

        # all threads have been processed
        # sleep temporarily so CPU can focus execution elsewhere
        time.sleep(SLEEP_TIME))
```

The loop in the preceding code will keep creating threads while there are URLs to crawl until it reaches the maximum number of threads set. During the crawl, threads may also prematurely shut down when there are currently no more URLs in the queue. For example, consider a situation when there are two threads and two URLs to download. When the first thread finishes its download, the crawl queue is empty so this thread exits. However, the second thread may then complete its download and discover additional URLs to download. The `thread` loop will then notice that there are still more URLs to download and the maximum number of threads has not been reached so create a new download thread.

The interface to process_link_crawler is still the same as the threaded crawler and is available at `https://bitbucket.org/wswp/code/src/tip/chapter04/process_test.py`. Now, let's test the performance of this multiprocess version of the link crawler with the following command:

```
$ time python threaded_test.py 5
...
4m50.465s
```

Since there are five threads, downloading is approximately five times faster! Further analysis of the threaded performance will be covered in the *Performance* section.

Cross-process crawler

To improve the performance further, the threaded example can be extended to support multiple processes. Currently, the crawl queue is held in local memory, which means other processes cannot contribute to the same crawl. To address this, the crawl queue will be transferred to MongoDB. Storing the queue independently means that even crawlers on separate servers could collaborate on the same crawl.

Note that for more robust queuing, a dedicated message passing tool such as Celery should be considered; however, MongoDB will be reused here to minimize the number of technologies introduced. Here is an implementation of the new MongoDB-backed queue:

```
from datetime import datetime, timedelta
from pymongo import MongoClient, errors

class MongoQueue:
    # possible states of a download
    OUTSTANDING, PROCESSING, COMPLETE = range(3)

    def __init__(self, client=None, timeout=300):
        self.client = MongoClient() if client is None else client
        self.db = self.client.cache
        self.timeout = timeout

    def __nonzero__(self):
        """Returns True if there are more jobs to process
        """
        record = self.db.crawl_queue.find_one(
            {'status': {'$ne': self.COMPLETE}}
        )
        return True if record else False

    def push(self, url):
        """Add new URL to queue if does not exist
        """
        try:
            self.db.crawl_queue.insert({'_id': url, 'status':
                self.OUTSTANDING})
        except errors.DuplicateKeyError as e:
            pass # this is already in the queue

    def pop(self):
        """Get an outstanding URL from the queue and set its
            status to processing. If the queue is empty a KeyError
                exception is raised.
        """
```

```
            record = self.db.crawl_queue.find_and_modify(
                query={'status': self.OUTSTANDING},
                update={'$set': {'status': self.PROCESSING,
                    'timestamp': datetime.now()}}
            )
            if record:
                return record['_id']
            else:
                self.repair()
                raise KeyError()

        def complete(self, url):
            self.db.crawl_queue.update({'_id': url}, {'$set':
                {'status': self.COMPLETE}})

        def repair(self):
            """Release stalled jobs
            """
            record = self.db.crawl_queue.find_and_modify(
                query={
                    'timestamp': {'$lt': datetime.now() -
                        timedelta(seconds=self.timeout)},
                    'status': {'$ne': self.COMPLETE}
                },
                update={'$set': {'status': self.OUTSTANDING}}
            )
            if record:
                print 'Released:', record['_id']
```

The queue in the preceding code defines three states: `OUTSTANDING`, `PROCESSING`, and `COMPLETE`. When a new URL is added, the state is `OUTSTANDING`, and when a URL is popped from the queue for downloading, the state is `PROCESSING`. Also, when the downloading is complete, the state is `COMPLETE`. Much of this implementation is concerned with how to handle when a URL is popped from the queue but the processing is never completed, for example, if the process that was handling the popped URL was terminated. To avoid losing the results of those URLs, this class takes a `timeout` argument, which is set to 300 seconds by default. In the `repair()` method, if the processing of a URL is found to take longer than this `timeout`, we assume that there has been an error and the URL state is returned to `OUTSTANDING` to be processed again.

Some minor changes are required to the threaded crawler to support this new queue type, which are highlighted here:

```
def threaded_crawler(...):
    ...
    # the queue of URL's that still need to be crawled
    crawl_queue = MongoQueue()
    crawl_queue.push(seed_url)

    def process_queue():
        while True:
            # keep track that are processing url
            try:
                url = crawl_queue.pop()
            except KeyError:
                # currently no urls to process
                break
           else:
                ...
                crawl_queue.complete(url)
```

The first change is replacing Python's built-in queue with the new MongoDB-based queue, named `MongoQueue`. This queue handles duplicate URLs internally, so the `seen` variable is no longer required. Finally, the `complete()` method is called after processing a URL to record that it has been successfully parsed.

This updated version of the threaded crawler can then be started in multiple processes with this snippet:

```
import multiprocessing

def process_link_crawler(args, **kwargs):
    num_cpus = multiprocessing.cpu_count()
    processes = []
    for i in range(num_cpus):
        p = multiprocessing.Process(target=threaded_crawler,
            args=[args], kwargs=kwargs)
        p.start()
        processes.append(p)
    # wait for processes to complete
    for p in processes:
        p.join()
```

This structure might look familiar now because the multiprocessing module follows a similar interface to the threading module used earlier. This code simply finds the number of CPUs available, starts the threaded crawler in a new process for each, and then waits for all the processes to complete execution.

Now, let's test the performance of this multiprocess version of the link crawler with using the following command. The interface to `process_link_crawler` is still the same as the threaded crawler and is available at `https://bitbucket.org/wswp/code/src/tip/chapter04/process_test.py`:

```
$ time python process_test.py 5
Starting 2 processes
...
2m5.405s
```

As detected by the script, the server on which this was tested has two CPUs, and the running time is approximately double that of the previous threaded crawler on a single process. In the next section, we will further investigate the relative performance of these three approaches.

Performance

To further understand how increasing the number of threads and processes affects the time required when downloading; here is a spreadsheet of results for crawling 1000 web pages:

Script	Number of threads	Number of processes	Time	Comparison with sequential
Sequential	1	1	28m59.966s	1
Threaded	5	1	7m11.634s	4.03
Threaded	10	1	3m50.455s	7.55
Threaded	20	1	2m45.412s	10.52
Processes	5	2	4m2.624s	7.17
Processes	10	2	2m1.445s	14.33
Processes	20	2	1m47.663s	16.16

The last column shows the proportion of time in comparison to the base case of sequential downloading. We can see that the increase in performance is not linearly proportional to the number of threads and processes, but appears logarithmic. For example, one process and five threads leads to 4X better performance, but 20 threads only leads to 10X better performance. Each extra thread helps, but is less effective than the previously added thread. This is to be expected, considering the process has to switch between more threads and can devote less time to each. Additionally, the amount of bandwidth available for downloading is limited so that eventually adding additional threads will not lead to a greater download rate. At this point, achieving greater performance would require distributing the crawl across multiple servers, all pointing to the same MongoDB queue instance.

Summary

This chapter covered why sequential downloading creates a bottleneck. We then looked at how to download large numbers of web pages efficiently across multiple threads and processes.

In the next chapter, we will cover how to scrape content from web pages that load their content dynamically using JavaScript.

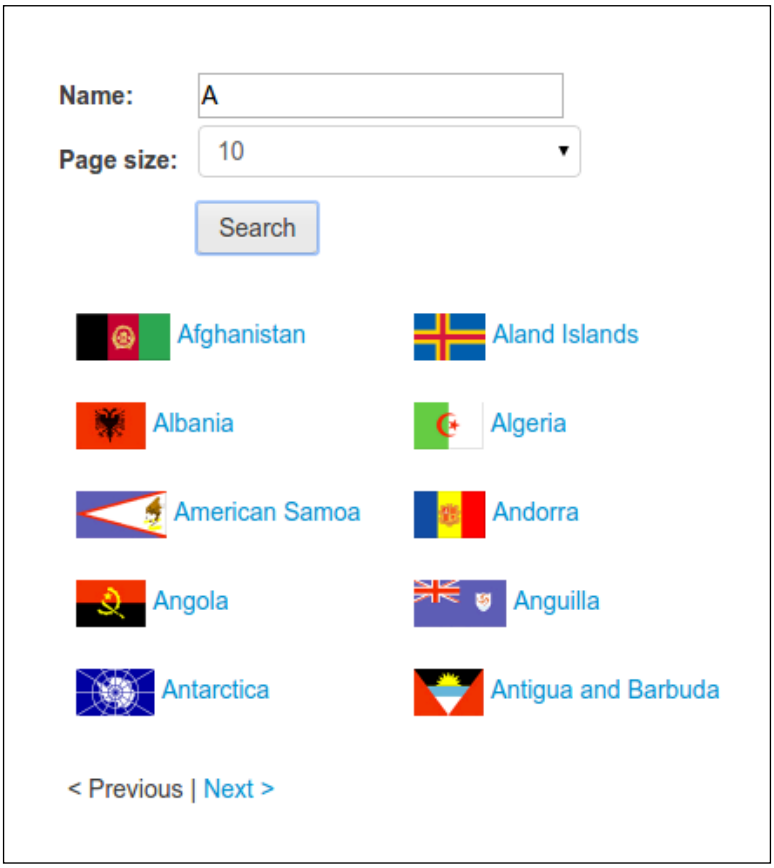
5
Dynamic Content

According to the United Nations Global Audit of Web Accessibility, 73 percent of leading websites rely on JavaScript for important functionalities (refer to `http://www.un.org/esa/socdev/enable/documents/execsumnomensa.doc`). The use of JavaScript can vary from simple form events to single page apps that download all their content after loading. The consequence of this is that for many web pages the content that is displayed in our web browser is not available in the original HTML, and the scraping techniques covered so far will not work. This chapter will cover two approaches to scraping data from dynamic JavaScript dependent websites. These are as follows:

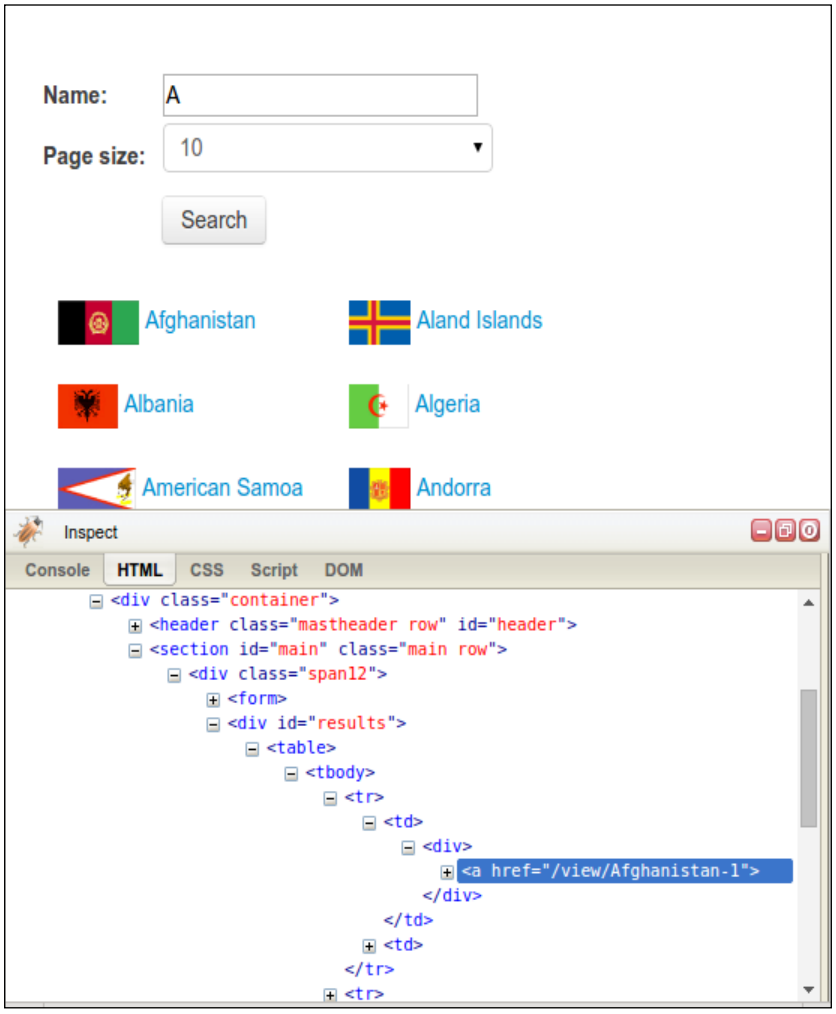
- Reverse engineering JavaScript
- Rendering JavaScript

An example dynamic web page

Let's look at an example dynamic web page. The example website has a search form, which is available at `http://example.webscraping.com/search`, that is used to locate countries. Let's say we want to find all the countries that begin with the letter A:



If we right-click on these results to inspect them with Firebug (as covered in *Chapter 2, Scraping the Data*), we would find that the results are stored within a `div` element of ID "results":



Let's try to extract these results using the `lxml` module, which was also covered in *Chapter 2, Scraping the Data*, and the `Downloader` class from *Chapter 3, Caching Downloads*:

```
>>> import lxml.html
>>> from downloader import Downloader
>>> D = Downloader()
>>> html = D('http://example.webscraping.com/search')
```

```
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('div#results a')
[]
```

The example scraper here has failed to extract results. Examining the source code of this web page can help you understand why. Here, we find that the `div` element we are trying to scrape is empty:

```
<div id="results">
</div>
```

Firebug gives us a view of the current state of the web page, which means that the web page has used JavaScript to load search results dynamically. In the next section, we will use another feature of Firebug to understand how these results are loaded.

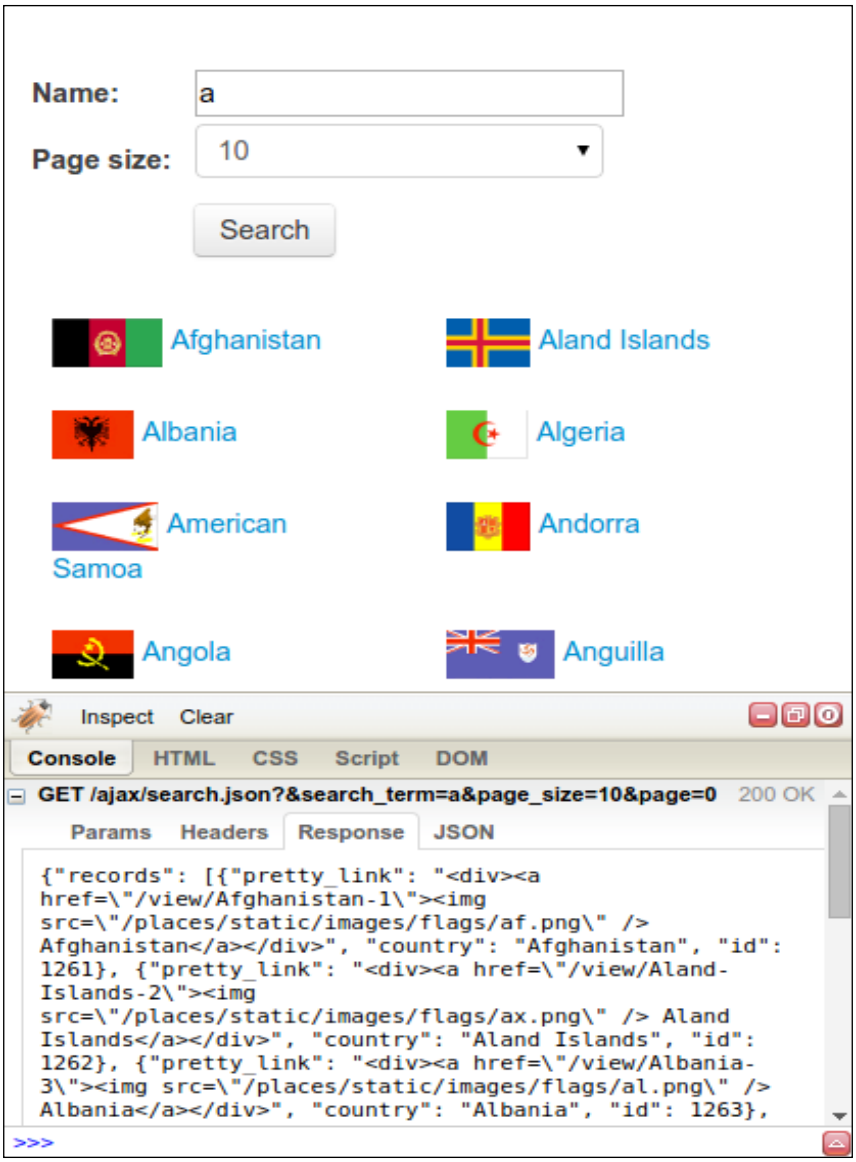
What is AJAX

AJAX stands for **Asynchronous JavaScript and XML** and was coined in 2005 to describe the features available across web browsers that made dynamic web applications possible. Most importantly, the JavaScript `XMLHttpRequest` object, which was originally implemented by Microsoft for ActiveX, was available in many common web browsers. This allowed JavaScript to make HTTP requests to a remote server and receive responses, which meant that a web application could send and receive data. The traditional way to communicate between client and server was to refresh the entire web page, which resulted in a poor user experience and wasted bandwidth when only a small amount of data needed to be transmitted.

Google's Gmail and Google Maps were early examples of the dynamic web applications and helped take AJAX to a mainstream audience.

Reverse engineering a dynamic web page

So far, we have tried to scrape data from a web page the same way as introduced in *Chapter 2*, *Scraping the Data*. However, it did not work because the data is loaded dynamically with JavaScript. To scrape this data, we need to understand how the web page loads this data, a process known as reverse engineering. Continuing the example from the preceding section, in Firebug, if we click on the **Console** tab and then perform a search, we will see that an AJAX request is made, as shown in this screenshot:



This AJAX data is not only accessible from within the search web page, but can also be downloaded directly, as follows:

```
>>> html =
  D('http://example.webscraping.com/ajax/
    search.json?page=0&page_size=10&search_term=a')
```

The AJAX response returns data in JSON format, which means Python's `json` module can be used to parse this into a dictionary, as follows:

```
>>> import json
>>> json.loads(html)
{u'error': u'',
  u'num_pages': 22,
  u'records': [{u'country': u'Afghanistan',
    u'id': 1261,
    u'pretty_link': u'<div><a href="/view/Afghanistan-1"><img
      src="/places/static/images/flags/af.png" />
        Afghanistan</a></div>'},
  ...]
}
```

Now we have a simple way to scrape countries containing the letter `A`. To find the details of all these countries then requires calling the AJAX search with each letter of the alphabet. For each letter, the search results are split into pages, and the number of pages is indicated by `page_size` in the response. An issue with saving results is that the same countries will be returned in multiple searches—for example, `Fiji` matches searches for `f`, `i`, and `j`. These duplicates are filtered here by storing results in a set before writing them to a spreadsheet—the set data structure will not store duplicate elements.

Here is an example implementation that scrapes all of the countries by searching for each letter of the alphabet and then iterating the resulting pages of the JSON responses. The results are then stored in a spreadsheet.

```
import json
import string

template_url =
  'http://example.webscraping.com/ajax/
    search.json?page={}&page_size=10&search_term={}'
countries = set()

for letter in string.lowercase:
    page = 0
    while True:
        html = D(template_url.format(page, letter))
        try:
            ajax = json.loads(html)
        except ValueError as e:
            print e
            ajax = None
```

```
            else:
                for record in ajax['records']:
                    countries.add(record['country'])
            page += 1
            if ajax is None or page >= ajax['num_pages']:
                break

    open('countries.txt', 'w').write('\n'.join(sorted(countries)))
```

This AJAX interface provides a simpler way to extract the country details than the scraping approach covered in *Chapter 2*, *Scraping the Data*. This is a common experience: the AJAX-dependent websites initially look more complex but their structure encourages separating the data and presentation layers, which can make our job of extracting this data much easier.

Edge cases

The AJAX search script is quite simple, but it can be simplified further by taking advantage of edge cases. So far, we have queried each letter, which means 26 separate queries, and there are duplicate results between these queries. It would be ideal if a single search query could be used to match all results. We will try experimenting with different characters to see if this is possible. This is what happens if the search term is left empty:

```
>>> url =
  'http://example.webscraping.com/ajax/
    search.json?page=0&page_size=10&search_term='
>>> json.loads(D(url))['num_pages']
0
```

Unfortunately, this did not work—there are no results. Next we will check if '*' will match all results:

```
>>> json.loads(D(url + '*'))['num_pages']
0
```

Still no luck. Now we will check '.', which is a regular expression to match any character:

```
>>> json.loads(D(url + '.'))['num_pages']
26
```

There we go: the server must be matching results with regular expressions. So, now searching each letter can be replaced with a single search for the dot character.

Further more, you may have noticed a parameter that is used to set the page size in the AJAX URLs. The search interface had options for setting this to 4, 10, and 20, with the default set to 10. So, the number of pages to download could be halved by increasing the page size to the maximum.

```
>>> url =
  'http://example.webscraping.com/ajax/
    search.json?page=0&page_size=20&search_term=.'
>>> json.loads(D(url))['num_pages']
13
```

Now, what if a much higher page size is used, a size higher than what the web interface select box supports?

```
>>> url =
  'http://example.webscraping.com/ajax/
    search.json?page=0&page_size=1000&search_term=.'
>>> json.loads(D(url))['num_pages']
1
```

Apparently, the server does not check whether the page size parameter matches the options allowed in the interface and now returns all the results in a single page. Many web applications do not check the page size parameter in their AJAX backend because they expect requests to only come through the web interface.

Now, we have crafted a URL to download the data for all the countries in a single request. Here is the updated and much simpler implementation to scrape all countries:

```
writer = csv.writer(open('countries.csv', 'w'))
writer.writerow(FIELDS)
html = D('http://example.webscraping.com/ajax/
    search.json?page=0&page_size=1000&search_term=.')
ajax = json.loads(html)
for record in ajax['records']:
    row = [record[field] for field in FIELDS]
    writer.writerow(row)
```

Rendering a dynamic web page

For the example search web page, we were able to easily reverse engineer how it works. However, some websites will be very complex and difficult to understand, even with a tool like Firebug. For example, if the website has been built with **Google Web Toolkit** (**GWT**), the resulting JavaScript code will be machine-generated and minified. This generated JavaScript code can be cleaned with a tool such as `JS beautifier`, but the result will be verbose and the original variable names will be lost, so it is difficult to work with. With enough effort, any website can be reverse engineered. However, this effort can be avoided by instead using a browser rendering engine, which is the part of the web browser that parses HTML, applies the CSS formatting, and executes JavaScript to display a web page as we expect. In this section, the WebKit rendering engine will be used, which has a convenient Python interface through the Qt framework.

What is WebKit?

The code for WebKit started life as the KHTML project in 1998, which was the rendering engine for the Konqueror web browser. It was then forked by Apple as WebKit in 2001 for use in their Safari web browser. Google used WebKit up to Chrome Version 27 before forking their version from WebKit called **Blink** in 2013. Opera originally used their internal rendering engine called **Presto** from 2003 to 2012 before briefly switching to WebKit, and then followed Chrome to Blink. Other popular browser rendering engines are **Trident,** used by Internet Explorer, and **Gecko** by Firefox.

PyQt or PySide

There are two available Python bindings to the Qt framework, `PyQt` and `PySide`. `PyQt` was first released in 1998 but requires a license for commercial projects. Due to this licensing problem, the company developing Qt, then Nokia and now Digia, later developed Python bindings in 2009 called `PySide` and released it under the more permissive LGPL license.

There are minor differences between the two bindings but the examples developed here will work with either. The following snippet can be used to import whichever Qt binding is installed:

```
try:
    from PySide.QtGui import *
    from PySide.QtCore import *
    from PySide.QtWebKit import *
except ImportError:
    from PyQt4.QtGui import *
    from PyQt4.QtCore import *
    from PyQt4.QtWebKit import *
```

Here, if `PySide` is not available, an `ImportError` exception will be raised and `PyQt` will be imported. If `PyQt` is also unavailable, another `ImportError` will be raised and the script will exit.

The instructions to download and install each Python binding are available at `http://qt-project.org/wiki/Setting_up_PySide` and `http://pyqt.sourceforge.net/Docs/PyQt4/installation.html`.

Executing JavaScript

To confirm that WebKit can execute JavaScript, there is a simple example available at `http://example.webscraping.com/dynamic`.

This web page simply uses JavaScript to write `Hello World` to a `div` element. Here is the source code:

```
<html>
    <body>
        <div id="result"></div>
        <script>
        document.getElementById("result").innerText = 'Hello World';
        </script>
    </body>
</html>
```

With the traditional approach of downloading the original HTML and parsing the result, the `div` element will be empty, as follows:

```
>>> url = 'http://example.webscraping.com/dynamic'
>>> html = D(url)
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('#result')[0].text_content()
''
```

Here is an initial example with WebKit, which needs to follow the `PyQt` or `PySide` imports shown in the preceding section:

```
>>> app = QApplication([])
>>> webview = QWebView()
>>> loop = QEventLoop()
>>> webview.loadFinished.connect(loop.quit)
>>> webview.load(QUrl(url))
>>> loop.exec_()
>>> html = webview.page().mainFrame().toHtml()
```

```
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('#result')[0].text_content()
'Hello World'
```

There is quite a lot going on here, so we will step through the code line by line:

- The first line instantiates the `QApplication` object that the Qt framework requires to be created before other Qt objects to perform various initializations.
- Next, a `QWebView` object is created, which is a container for the web documents.
- A `QEventLoop` object is created, which will be used to create a local event loop.
- The `loadFinished` callback of the `QwebView` object is connected to the `quit` method of `QEventLoop` so that when a web page finishes loading, the event loop will be stopped. The URL to load is then passed to `QWebView`. `PyQt` requires that this URL string is wrapped by a `QUrl` object, while for `PySide`, this is optional.
- The `QWebView` load method is asynchronous, so execution will immediately pass to the next line while the web page is loading—however, we want to wait until this web page is loaded, so `loop.exec_()` is called to start the event loop.
- When the web page completes loading, the event loop will exit and execution can move to the next line, where the resulting HTML of this loaded web page is extracted.
- The final line shows that JavaScript has been successfully executed and the `div` element contains `Hello World`, as expected.

The classes and methods used here are all excellently documented in the C++ Qt framework website at `http://qt-project.org/doc/qt-4.8/`. `PyQt` and `PySide` have their own documentation, however, the descriptions and formatting for the original C++ version is superior, and, generally Python developers use it instead.

Website interaction with WebKit

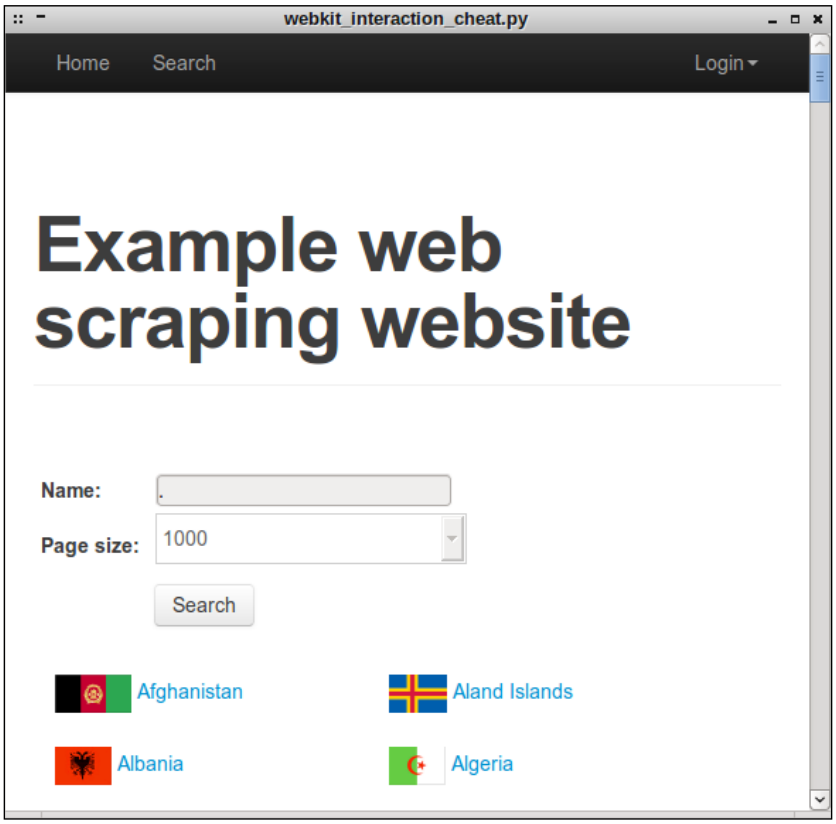
The search web page we have been examining requires the user to modify and submit a search form, and then click on the page links. However, so far, our browser renderer can only execute JavaScript and access the resulting HTML. To scrape the search page will require extending the browser renderer to support these interactions. Fortunately, Qt has an excellent API to select and manipulate the HTML elements, which makes this straightforward.

Here is an alternative version to the earlier AJAX search example, which also sets the search term to `'.'` and page size to `'1000'` to load all the results in a single query:

```
app = QApplication([])
webview = QWebView()
loop = QEventLoop()
webview.loadFinished.connect(loop.quit)
webview.load(QUrl('http://example.webscraping.com/search'))
loop.exec_()
webview.show()
frame = webview.page().mainFrame()
frame.findFirstElement('#search_term').
       setAttribute('value', '.')
frame.findFirstElement('#page_size option:checked').
       setPlainText('1000')
frame.findFirstElement('#search').
       evaluateJavaScript('this.click()')
app.exec_()
```

The first few lines instantiate the Qt objects required to render a web page, the same as in the previous `Hello World` example. Next, the `QWebView` GUI `show()` method is called so that the render window is displayed, which is useful for debugging. Then, a reference to the frame is created to make the following lines shorter. The `QWebFrame` class has many useful methods to interact with a web page. The following two lines use the CSS patterns to locate an element in the frame, and then set the search parameters. Then, the form is submitted with the `evaluateJavaScript()` method to simulate a click event. This method is very convenient because it allows for the insertion of any JavaScript code we want, including calling JavaScript methods defined in the web page directly. Then, the final line enters the application event loop so that we can review what has happened to the form. Without this, the script would end immediately.

This is what is displayed when this script is run:



Waiting for results

The final part of implementing our WebKit crawler is scraping the search results, which turns out to be the most difficult part because it is not obvious when the AJAX event is complete and the country data is ready. There are three possible approaches to deal with this:

- Wait a set amount of time and hope that the AJAX event is complete by then
- Override Qt's network manager to track when the URL requests are complete
- Poll the web page for the expected content to appear

The first option is the simplest to implement but is inefficient, because if a safe timeout is set, then usually a lot more time is spent waiting than necessary. Also, when the network is slower than usual, a fixed timeout could fail. The second option is more efficient but cannot be applied when the delay is from the client side rather than server side—for example, if the download is complete, but a button needs to be pressed before the content is displayed. The third option is more reliable and straightforward to implement, though there is the minor drawback of wasting CPU cycles when checking whether the content has loaded yet. Here is an implementation for the third option:

```
>>> elements = None
>>> while not elements:
...     app.processEvents()
...     elements = frame.findAllElements('#results a')
...
>>> countries = [e.toPlainText().strip() for e in elements]
>>> print countries
[u'Afghanistan', u'Aland Islands', ... , u'Zambia', u'Zimbabwe']
```

Here, the code keeps looping until the country links are present in the `results` div. For each loop, `app.processEvents()` is called to give the Qt event loop time to perform tasks, such as responding to click events and updating the GUI.

The Render class

To help make this functionality easier to use in future, here are the methods used and packaged into a class, whose source code is also available at `https://bitbucket.org/wswp/code/src/tip/chapter05/browser_render.py`:

```
import time

class BrowserRender(QWebView):
    def __init__(self, show=True):
        self.app = QApplication(sys.argv)
        QWebView.__init__(self)
        if show:
            self.show() # show the browser

    def download(self, url, timeout=60):
        """Wait for download to complete and return result"""
        loop = QEventLoop()
        timer = QTimer()
        timer.setSingleShot(True)
        timer.timeout.connect(loop.quit)
        self.loadFinished.connect(loop.quit)
        self.load(QUrl(url))
        timer.start(timeout * 1000)
```

```
        loop.exec_() # delay here until download finished
        if timer.isActive():
            # downloaded successfully
            timer.stop()
            return self.html()
        else:
            # timed out
            print 'Request timed out: ' + url

    def html(self):
        """Shortcut to return the current HTML"""
        return self.page().mainFrame().toHtml()

    def find(self, pattern):
        """Find all elements that match the pattern"""
        return self.page().mainFrame().findAllElements(pattern)

    def attr(self, pattern, name, value):
        """Set attribute for matching elements"""
        for e in self.find(pattern):
            e.setAttribute(name, value)

    def text(self, pattern, value):
        """Set attribute for matching elements"""
        for e in self.find(pattern):
            e.setPlainText(value)

    def click(self, pattern):
        """Click matching elements"""
        for e in self.find(pattern):
            e.evaluateJavaScript("this.click()")

    def wait_load(self, pattern, timeout=60):
        """Wait until pattern is found and return matches"""
        deadline = time.time() + timeout
        while time.time() < deadline:
            self.app.processEvents()
            matches = self.find(pattern)
            if matches:
                return matches
        print 'Wait load timed out'
```

You may have noticed the `download()` and `wait_load()` methods have some additional code involving a timer. This timer tracks how long has been spent waiting and cancels the event loop when the deadline is reached. Otherwise, when a network problem is encountered, the event loop would run indefinitely.

Here is how to scrape the search page using this new class:

```
>>> br = BrowserRender()
>>> br.download('http://example.webscraping.com/search')
>>> br.attr('#search_term', 'value', '.')
>>> br.text('#page_size option:checked', '1000')
>>> br.click('#search')
>>> elements = br.wait_load('#results a')
>>> countries = [e.toPlainText().strip() for e in elements]
>>> print countries
[u'Afghanistan', u'Aland Islands', ..., u'Zambia', u'Zimbabwe']
```

Selenium

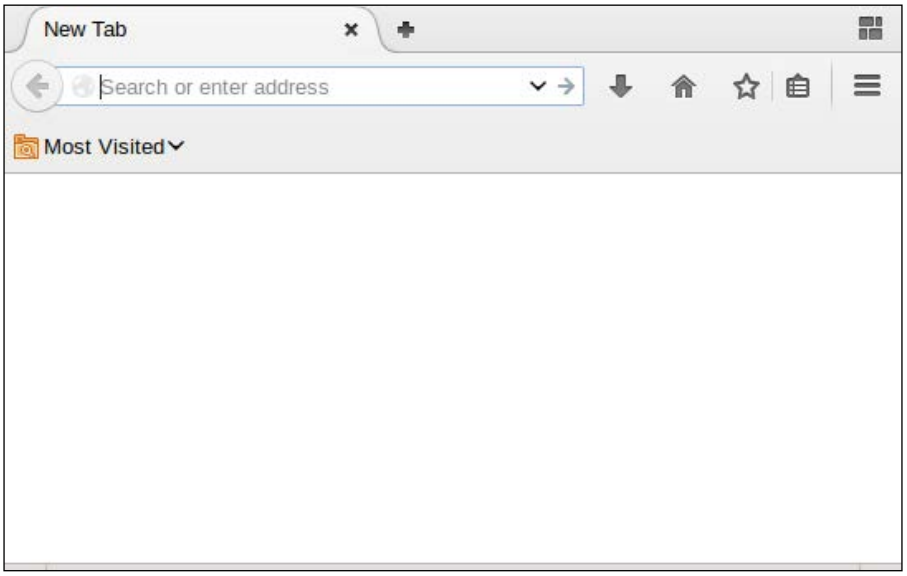
With the WebKit library used in the preceding example, we have full control to customize the browser renderer to behave as we need it to. If this level of flexibility is not needed, a good alternative is Selenium, which provides an API to automate the web browser. Selenium can be installed using `pip` with the following command:

```
pip install selenium
```

To demonstrate how Selenium works, we will rewrite the previous search example in Selenium. The first step is to create a connection to the web browser:

```
>>> from selenium import webdriver
>>> driver = webdriver.Firefox()
```

When this command is run, an empty browser window will pop up:



This is handy because with each command, the browser window can be checked to see if Selenium worked as expected. Here, Firefox was used, but Selenium also provides interfaces to other common web browsers, such as Chrome and Internet Explorer. Note that you can only use a Selenium interface for a web browser that is installed on your system.

To load a web page in the chosen web browser, the `get()` method is called:

```
>>> driver.get('http://example.webscraping.com/search')
```

Then, to set which element to select, the ID of the search textbox can be used. Selenium also supports selecting elements with a CSS selector or XPath. When the search textbox is found, we can enter content with the `send_keys()` method, which simulates typing:

```
>>> driver.find_element_by_id('search_term').send_keys('.')
```

To return all results in a single search, we want to set the page size to 1000. However, this is not straightforward because Selenium is designed to interact with the browser, rather than to modify the web page content. To get around this limitation, we can use JavaScript to set the select box content:

```
>>> js = "document.getElementById('page_size').options[1].text =
    '1000'"
>>> driver.execute_script(js);
```

Now the form inputs are all ready, so the search button can be clicked on to perform the search:

```
>>> driver.find_element_by_id('search').click()
```

Now we need to wait for the AJAX request to complete before loading the results, which was the hardest part of the script in the previous WebKit implementation. Fortunately, Selenium provides a simple solution to this problem by setting a timeout with the `implicitly_wait()` method:

```
>>> driver.implicitly_wait(30)
```

Here, a delay of 30 seconds was used. Now, if we search for elements and they are not yet available, Selenium will wait up to 30 seconds before raising an exception. To select the country links, we can use the same CSS selector that we used in the WebKit example:

```
>>> links = driver.find_elements_by_css_selector('#results a')
```

Then, the text of each link can be extracted to create a list of countries:

```
>>> countries = [link.text for link in links]
>>> print countries
[u'Afghanistan', u'Aland Islands', ..., u'Zambia', u'Zimbabwe']
```

Finally, the browser can be shut down by calling the `close()` method:

```
>>> driver.close()
```

The source code for this example is available at `https://bitbucket.org/wswp/code/src/tip/chapter05/selenium_search.py`. For further details about Selenium, the Python bindings are well documented at `https://selenium-python.readthedocs.org/`.

Summary

This chapter covered two approaches to scrape data from dynamic web pages. It started with reverse engineering a dynamic web page with the help of Firebug Lite, and then moved on to using a browser renderer to trigger JavaScript events for us. We first used WebKit to build our own custom browser, and then reimplemented this scraper with the high-level Selenium framework.

A browser renderer can save the time needed to understand how the backend of a website works, however, there are disadvantages. Rendering a web page adds overhead and so is much slower than just downloading the HTML. Additionally, solutions using a browser renderer often require polling the web page to check whether the resulting HTML from an event has occurred yet, which is brittle and can easily fail when the network is slow. I typically use a browser renderer for short term solutions where the long term performance and reliability is less important; then for long term solutions, I make the effort to reverse engineer the website.

In the next chapter, we will cover how to interact with forms and cookies to log into a website and edit content.

6
Interacting with Forms

In earlier chapters, we downloaded static web pages that always return the same content. Now, in this chapter, we will interact with web pages that depend on user input and state to return relevant content. This chapter will cover the following topics:

- Sending a `POST` request to submit a form
- Using cookies to log in to a website
- The high-level Mechanize module for easier form submissions

To interact with these forms, you will need a user account to log in to the website. You can register an account manually at `http://example.webscraping.com/user/register`. Unfortunately, we can not yet automate the registration form until the next chapter, which deals with `CAPTCHA`.

Form methods

HTML forms define two methods for submitting data to the server—`GET` and `POST`. With the `GET` method, data like `?name1=value1&name2=value2` is appended to the URL, which is known as a "query string". The browser sets a limit on the URL length, so this is only useful for small amounts of data. Additionally, this method is intended only to retrieve data from the server and not make changes to it, but sometimes this is ignored. With `POST` requests, the data is sent in the request body, which is separate from the URL. Sensitive data should only be sent in a `POST` request to avoid exposing it in the URL. How the `POST` data is represented in the body depends on the encoding type.

Servers can also support other HTTP methods, such as `PUT` and `DELETE`, however, these are not supported in forms.

The Login form

The first form that we will automate is the **Login** form, which is available at `http://example.webscraping.com/user/login`. To understand the form, we will use Firebug Lite. With the full version of Firebug or Chrome DevTools, it is possible to just submit the form and check what data was transmitted in the network tab. However, the Lite version is restricted to viewing the structure, as follows:

```
<form action="#" enctype="application/x-www-form-urlencoded" method="post">
  <table>
      <tbody>
          <tr id="auth_user_email__row">
              <td class="w2p_fl">
              <td class="w2p_fw">
                    <input class="string" id="auth_user_email" name="email" type="text" value=""/>
                </td>
                <td class="w2p_fc"/>
            </tr>
          <tr id="auth_user_password__row">
              <td class="w2p_fl">
              <td class="w2p_fw">
                    <input class="password" id="auth_user_password" name="password" type="password" value=""/>
                </td>
                <td class="w2p_fc"/>
            </tr>
          <tr id="auth_user_remember__row">
          <tr id="submit_record__row">
        </tbody>
    </table>
  <div style="display: none;">
        <input name="_next" type="hidden" value="/"/>
        <input name="_formkey" type="hidden" value="fd155d33-0b81-48de-86f8-6a2b62bcd701"/>
        <input name="_formname" type="hidden" value="login"/>
    </div>
</form>
```

The important parts here are the `action`, `enctype`, and `method` attributes of the `form` tag, and the two `input` fields. The `action` attribute sets the location where the form data will be submitted, in this case, `#`, which means the same URL as the **Login** form. The `enctype` attribute sets the encoding used for the submitted data, in this case, `application/x-www-form-urlencoded`. Also, the `method` attribute is set to `post` to submit form data in the body to the server. For the `input` tags, the important attribute is `name`, which sets the name of the field when submitted to the server.

Form encoding

When a form uses the `POST` method, there are two useful choices for how the form data is encoded before being submitted to the server. The default is `application/x-www-form-urlencoded`, which specifies that all non-alphanumeric characters must be converted to ASCII Hex values. However, this is inefficient for forms that contain a large amount of non-alphanumeric data, such as a binary file upload, so multipart/form-data encoding was defined. Here, the input is not encoded but sent as multiple parts using the MIME protocol, which is the same standard used for e-mail.

The official details of this standard can be viewed at `http://www.w3.org/TR/html5/forms.html#selecting-a-form-submission-encoding`.

When regular users open this web page in their browser, they will enter their e-mail and password, and click on the **Login** button to submit their details to the server. Then, if the login process on the server is successful, they will be redirected to the home page; otherwise, they will return to the **Login** page to try again. Here is an initial attempt to automate this process:

```
>>> import urllib, urllib2
>>> LOGIN_URL = 'http://example.webscraping.com/user/login'
>>> LOGIN_EMAIL = 'example@webscraping.com'
>>> LOGIN_PASSWORD = 'example'
>>> data = {'email': LOGIN_EMAIL, 'password': LOGIN_PASSWORD}
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(LOGIN_URL, encoded_data)
>>> response = urllib2.urlopen(request)
>>> response.geturl()
'http://example.webscraping.com/user/login'
```

This example sets the e-mail and password fields, encodes them with `urlencode`, and then submits them to the server. When the final print statement is executed, it will output the URL of the **Login** page, which means that the login process has failed.

This **Login** form is particularly strict and requires some additional fields to be submitted along with the e-mail and password. These additional fields can be found at the bottom of the preceding screenshot, but are set to `hidden` and so are not displayed in the browser. To access these hidden fields, here is a function using the `lxml` library covered in *Chapter 2*, *Scraping the Data*, to extract all the `input` tag details in a form:

```
import lxml.html
def parse_form(html):
```

```
    tree = lxml.html.fromstring(html)
    data = {}
    for e in tree.cssselect('form input'):
        if e.get('name'):
            data[e.get('name')] = e.get('value')
    return data
```

The function in the preceding code uses `lxml` CSS selectors to iterate all the `input` tags in a form and return their `name` and `value` attributes in a dictionary. Here is the result when the code is run on the **Login** page:

```
>>> import pprint
>>> html = urllib2.urlopen(LOGIN_URL).read()
>>> form = parse_form(html)
>>> pprint.pprint(form)
{
    '_next': '/',
    '_formkey': '0291ec65-9332-426e-b6a1-d97b3a2b12f8',
    '_formname': 'login',
    'email': '',
    'password': '',
    'remember': 'on'
}
```

The `_formkey` attribute is the crucial part here, which is a unique ID used by the server to prevent multiple form submissions. Each time the web page is loaded, a different ID is used, and then the server can tell whether a form with a given ID has already been submitted. Here is an updated version of the login process that submits `_formkey` and other hidden values:

```
>>> html = urllib2.urlopen(LOGIN_URL).read()
>>> data = parse_form(html)
>>> data['email'] = LOGIN_EMAIL
>>> data['password'] = LOGIN_PASSWORD
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(LOGIN_URL, encoded_data)
>>> response = urllib2.urlopen(request)
>>> response.geturl()
'http://example.webscraping.com/user/login'
```

Unfortunately, this version did not work either, and when run the login URL was printed again. We are missing a crucial component—cookies. When a regular user loads the **Login** form, this `_formkey` value will be stored in a cookie, which is then compared to the `_formkey` value in the submitted **Login** form data. Here is an updated version using the `urllib2.HTTPCookieProcessor` class to add support for cookies:

```
>>> import cookielib
>>> cj = cookielib.CookieJar()
>>> opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
>>> html = opener.open(LOGIN_URL).read()
>>> data = parse_form(html)
>>> data['email'] = LOGIN_EMAIL
>>> data['password'] = LOGIN_PASSWORD
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(LOGIN_URL, encoded_data)
>>> response = opener.open(request)
>>> response.geturl()
'http://example.webscraping.com'
```

What are cookies?

Cookies are small amounts of data sent by a website in the HTTP `response` headers, which look like this: `Set-Cookie: session_id=example;`. The web browser will store them, and then include them in the headers of subsequent requests to that website. This allows a website to identify and track users.

Success! The submitted form values have been accepted and the `response` URL is the home page. This snippet and the other login examples in this chapter are available for download at `https://bitbucket.org/wswp/code/src/tip/chapter06/login.py`.

Loading cookies from the web browser

Working out how to submit the login details as expected by the server can be quite complex, as the previous example demonstrated. Fortunately, there is a workaround for difficult websites—we can log in to the website manually in our web browser, and then have our Python script load and reuse these cookies to be automatically logged in. Each web browser stores their cookies in a different format, so we will focus on just the Firefox browser in this example.

FireFox stores its cookies in a `sqlite` database and its sessions in a JSON file, which can be connected to directly from Python. For the login, we only need the sessions, which are stored in this structure:

```
{"windows": [...
  "cookies": [
    {"host":"example.webscraping.com",
     "value":"514315085594624:e5e9a0db-5b1f-4c66-a864",
     "path":"/",
     "name":"session_id_places"}
  ...]
]}
```

Here is a function that can be used to parse these sessions into a `CookieJar` object:

```
def load_ff_sessions(session_filename):
    cj = cookielib.CookieJar()
    if os.path.exists(session_filename):
        json_data = json.loads(open(session_filename, 'rb').read())
        for window in json_data.get('windows', []):
            for cookie in window.get('cookies', []):
                c = cookielib.Cookie(0,
                    cookie.get('name', ''),
                    cookie.get('value', ''), None, False,
                    cookie.get('host', ''),
                    cookie.get('host', '').startswith('.'),
                    cookie.get('host', '').startswith('.'),
                    cookie.get('path', ''), False, False,
                    str(int(time.time()) + 3600 * 24 * 7),
                    False, None, None, {})
                cj.set_cookie(c)
    else:
        print 'Session filename does not exist:', session_filename
    return cj
```

One complexity is that the location of the FireFox sessions file will vary, depending on the operating system. On Linux, it should be located at this path:

```
~/.mozilla/firefox/*.default/sessionstore.js
```

In OS X, it should be located at:

```
~/Library/Application Support/Firefox/Profiles/*.default/
   sessionstore.js
```

Also, for Windows Vista and above, it should be located at:

```
%APPDATA%/Roaming/Mozilla/Firefox/Profiles/*.default/sessionstore.js
```

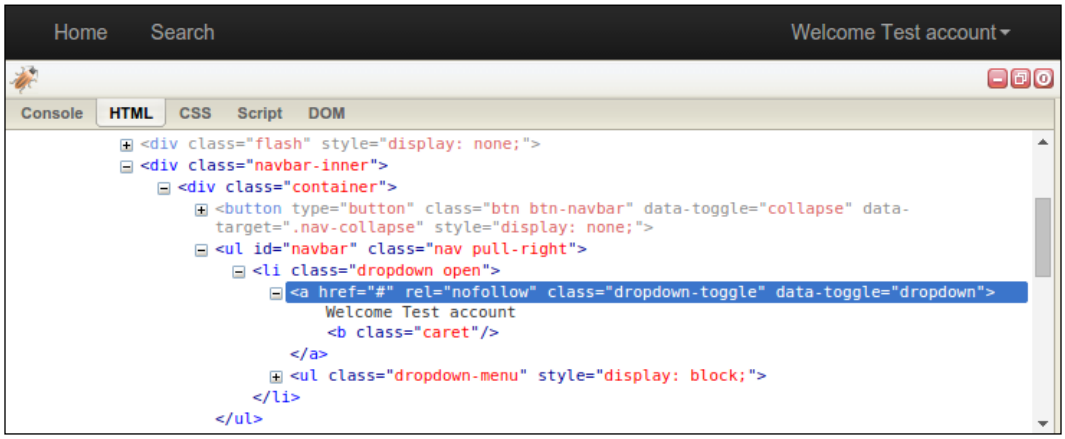
Here is a helper function to return the path to the session file:

```
import os, glob
def find_ff_sessions():
    paths = [
        '~/.mozilla/firefox/*.default',
        '~/Library/Application Support/Firefox/Profiles/*.default',
        '%APPDATA%/Roaming/Mozilla/Firefox/Profiles/*.default'
    ]
    for path in paths:
        filename = os.path.join(path, 'sessionstore.js')
        matches = glob.glob(os.path.expanduser(filename))
        if matches:
            return matches[0]
```

Note that the `glob` module used here will return all the matching files for the given path. Now here is an updated snippet using the browser cookies to log in:

```
>>> session_filename = find_ff_sessions()
>>> cj = load_ff_sessions(session_filename)
>>> processor = urllib2.HTTPCookieProcessor(cj)
>>> opener = urllib2.build_opener(processor)
>>> url = 'http://example.webscraping.com'
>>> html = opener.open(url).read()
```

To check whether the session was loaded successfully, we cannot rely on the login redirect this time. Instead, we will scrape the resulting HTML to check whether the logged in user label exists. If the result here is `Login`, the sessions have failed to load correctly. If this is the case, make sure you are already logged in to the example website in FireFox. Here is the structure of the user label in Firebug:



Firebug shows that this label is located within a `<ul>` tag of ID "navbar", which can easily be extracted with the `lxml` library used in *Chapter 2*, *Scraping the Data*:

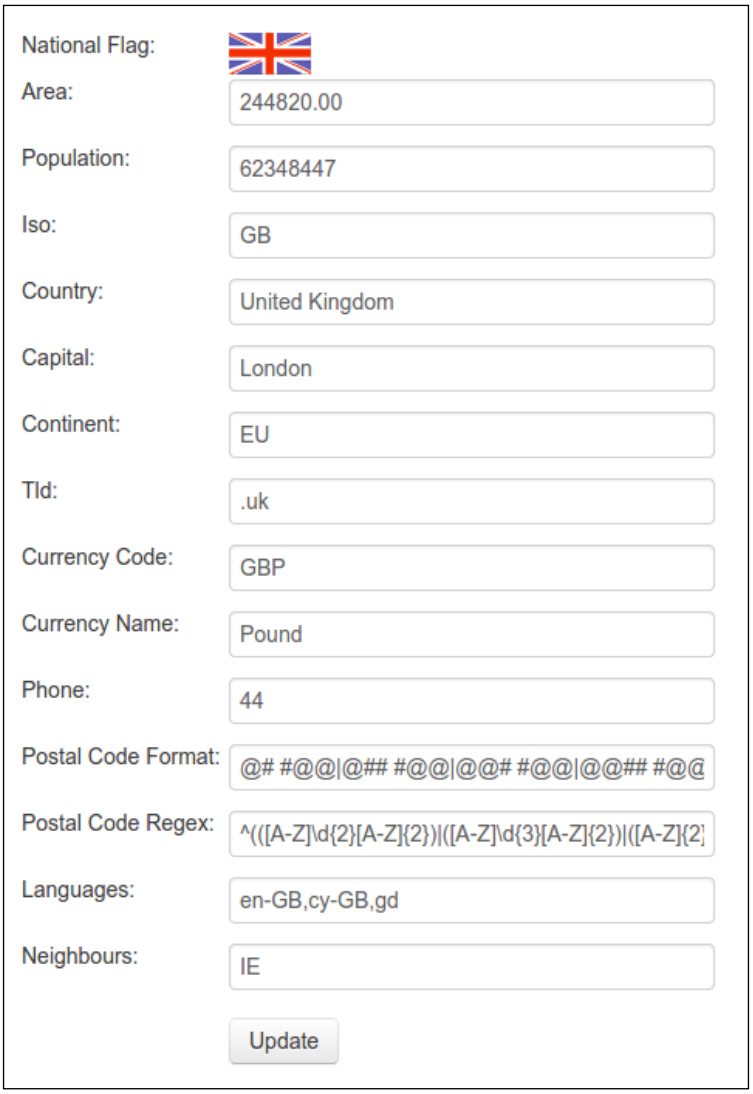
```
>>> tree = lxml.html.fromstring(html)
>>> tree.cssselect('ul#navbar li a')[0].text_content()
Welcome Test account
```

The code in this section was quite complex and only supports loading sessions from the Firefox browser. If you want support for all cookies as well as other web browsers try the `browsercookie` module, which can be installed with `pip install browsercookie` and is documented at `https://pypi.python.org/pypi/browsercookie`.

Extending the login script to update content

Now that the login automation is working, we can make the script more interesting by extending it to interact with the website and update the country data. The code used in this section is available at `https://bitbucket.org/wswp/code/src/tip/chapter06/edit.py`. You may have noticed an **Edit** link at the bottom of each country:

When logged in, this leads to another page where each property of a country can be edited:



We will make a script to increase the population of a country by one person each time it is run. The first step is to extract the current values of the country by reusing the `parse_form()` function:

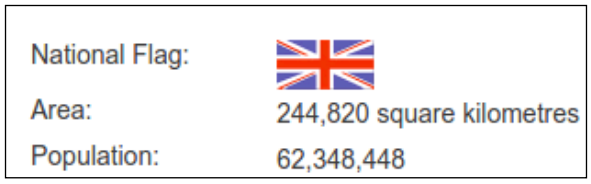
```
>>> import login
>>> COUNTRY_URL = 'http://example.webscraping.com/edit/
    United-Kingdom-239'
```

```
>>> opener = login.login_cookies()
>>> country_html = opener.open(COUNTRY_URL).read()
>>> data = parse_form(country_html)
>>> pprint.pprint(data)
{'_formkey': '4cf0294d-ea71-4cd8-ae2a-43d4ca0d46dd',
 '_formname': 'places/5402840151359488',
 'area': '244820.00',
 'capital': 'London',
 'continent': 'EU',
 'country': 'United Kingdom',
 'currency_code': 'GBP',
 'currency_name': 'Pound',
 'id': '5402840151359488',
 'iso': 'GB',
 'languages': 'en-GB,cy-GB,gd',
 'neighbours': 'IE',
 'phone': '44',
 'population': '62348447',
 'postal_code_format': '@# #@@|@## #@@|@@# #@@|@@## #@@|@#@
    #@@|@@#@ #@@|GIR0AA',
 'postal_code_regex': '^(([A-Z]\\d{2}[A-Z]{2})|([A-Z]\\d{3}
    [A-Z]{2})|([A-Z]{2}\\d{2}[A-Z]{2})|([A-Z]{2}\\d{3}
        [A-Z]{2})|([A-Z]\\d[A-Z]\\d[A-Z]{2})|([A-Z]{2}\\
            d[A-Z]\\d[A-Z]{2})|(GIR0AA))$',
 'tld': '.uk'}
```

Now we increase the population by one and submit the updated version to the server:

```
>>> data['population'] = int(data['population']) + 1
>>> encoded_data = urllib.urlencode(data)
>>> request = urllib2.Request(COUNTRY_URL, encoded_data)
>>> response = opener.open(request)
```

When we return to the country page, we can verify that the population has increased to **62,348,448**:



Feel free to test and modify the other fields too—the database is restored to the original country data each hour to keep the data sane. The code used in this section is available at `https://bitbucket.org/wswp/code/src/tip/chapter06/edit.py`. Note that the example covered here is not strictly web scraping, but falls under the wider scope of online bots. However, the form techniques used can be applied to interacting with complex forms when scraping.

Automating forms with the Mechanize module

The examples built so far work, but each form requires a fair amount of work and testing. This effort can be minimized by using Mechanize, which provides a high-level interface to interact with forms. Mechanize can be installed via `pip` using this command:

```
pip install mechanize
```

Here is how to implement the previous population increase example with Mechanize:

```
>>> import mechanize
>>> br = mechanize.Browser()
>>> br.open(LOGIN_URL)
>>> br.select_form(nr=0)
>>> br['email'] = LOGIN_EMAIL
>>> br['password'] = LOGIN_PASSWORD
>>> response = br.submit()
>>> br.open(COUNTRY_URL)
>>> br.select_form(nr=0)
>>> br['population'] = str(int(br['population']) + 1)
>>> br.submit()
```

This code is much simpler than the previous example because we no longer need to manage cookies and the form inputs are easily accessible. This script first creates the Mechanize browser object, and then we navigate to the login URL and select the login form. The selected form inputs are set by passing the name and values directly to the browser object. When debugging, `br.form` can be called to show the state of a form before it is submitted, as shown here:

```
>>> print br.form
<POST http://example.webscraping.com/user/login# application/
  x-www-form-urlencoded
```

```
<TextControl(email=)>
<PasswordControl(password=)>
<CheckboxControl(remember=[on])>
<SubmitControl(<None>=Login) (readonly)>
<SubmitButtonControl(<None>=) (readonly)>
<HiddenControl(_next=/) (readonly)>
<HiddenControl(_formkey=5fa268b4-0dfd-4e3f-a274-e73c6b7ce584)
  (readonly)>
<HiddenControl(_formname=login) (readonly)>>
```

Once the login form parameters are set, `br.submit()` is called to submit the selected login form, and the script will be logged in. Now, we can navigate to the edit country page, and use the form interface again to increment the population. Note that the population input expects a string, otherwise Mechanize will raise a `TypeError` exception.

To check whether this has worked, we can use Mechanize to return to the country form and query the current population, as follows:

```
>>> br.open(COUNTRY_URL)
>>> br.select_form(nr=0)
>>> br['population']
62348449
```

As expected, the population has increased again and is now `62,348,449`.

Further documentation and examples for the Mechanize module are available from the project website at `http://wwwsearch.sourceforge.net/mechanize/`.

Summary

Interacting with forms is a necessary skill when scraping web pages. This chapter covered two approaches: first, analyzing the form to generate the expected `POST` request manually, and second, using the high-level Mechanize module.

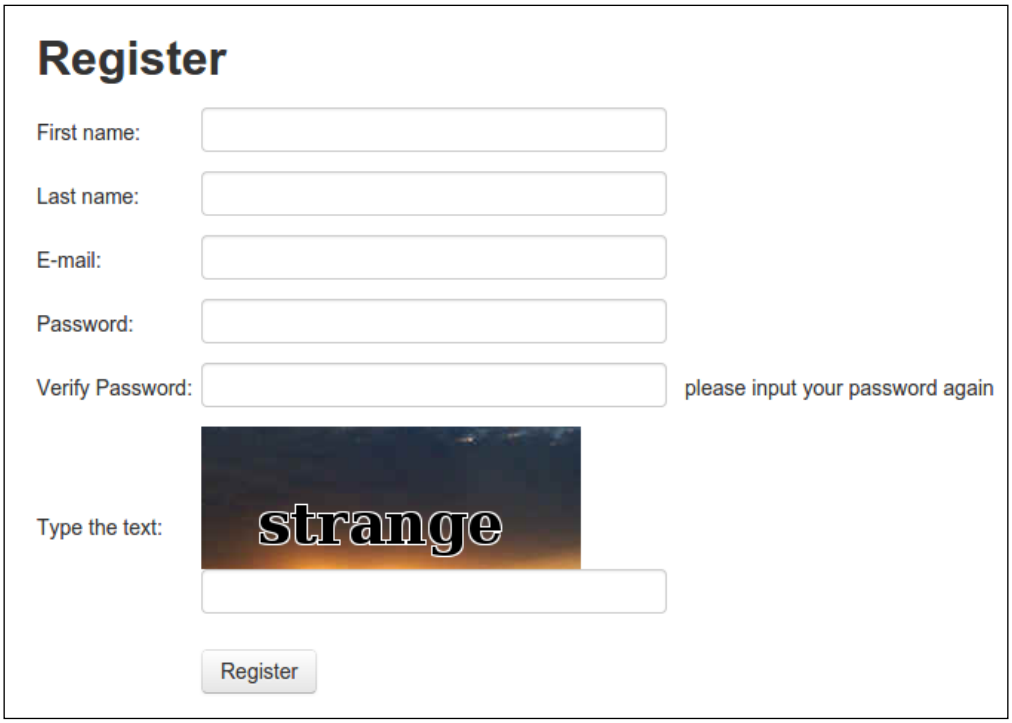
In the following chapter, we will expand our form skillset and learn how to submit forms that require passing `CAPTCHA`.

7
Solving CAPTCHA

CAPTCHA stands for **Completely Automated Public Turing test to tell Computers and Humans Apart**. As the acronym suggests, it is a test to determine whether the user is human or not. A typical CAPTCHA consists of distorted text, which a computer program will find difficult to interpret but a human can (hopefully) still read. Many websites use CAPTCHA to try and prevent bots from interacting with their website. For example, my bank website forces me to pass a CAPTCHA every time I log in, which is a pain. This chapter will cover how to solve a CAPTCHA automatically, first through **Optical Character Recognition** (**OCR**) and then with a CAPTCHA solving API.

Registering an account

In the preceding chapter on forms, we logged in to the example website using a manually created account and skipped automating the account creation part, because the registration form requires passing a CAPTCHA. This is how the registration page at `http://example.webscraping.com/user/register` looks:



Note that each time this form is loaded, a different CAPTCHA image will be shown. To understand what the form requires, we can reuse the `parse_form()` function developed in the preceding chapter.

```
>>> import cookielib, urllib2, pprint
>>> REGISTER_URL = 'http://example.webscraping.com/user/register'
>>> cj = cookielib.CookieJar()
>>> opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
>>> html = opener.open(REGISTER_URL).read()
>>> form = parse_form(html)
>>> pprint.pprint(form)
{'_formkey': '1ed4e4c4-fbc6-4d82-a0d3-771d289f8661',
 '_formname': 'register',
 '_next': '/',
```

```
    'email': '',
    'first_name': '',
    'last_name': '',
    'password': '',
    'password_two': None,
    'recaptcha_response_field': None}
```

All of the fields in the preceding code are straightforward, except for `recaptcha_response_field`, which in this case requires extracting **strange** from the image.

Loading the CAPTCHA image

Before the CAPTCHA image can be analyzed, it needs to be extracted from the form. FireBug shows that the data for this image is embedded in the web page, rather than being loaded from a separate URL:

To work with images in Python, we will use the `Pillow` package, which can be installed via `pip` using this command:

```
pip install Pillow
```

Alternative ways to install `Pillow` are covered at `http://pillow.readthedocs.org/installation.html`.

`Pillow` provides a convenient `Image` class with a number of high-level methods that will be used to manipulate the CAPTCHA images. Here is a function that takes the HTML of the registration page, and returns the CAPTCHA image in an `Image` object:

```
from io import BytesIO
import lxml.html
from PIL import Image

def get_captcha(html):
    tree = lxml.html.fromstring(html)
    img_data = tree.cssselect('div#recaptcha img')[0].get('src')
    img_data = img_data.partition(',')[-1]
    binary_img_data = img_data.decode('base64')
```

```
    file_like = BytesIO(binary_img_data)
    img = Image.open(file_like)
    return img
```

The first few lines here use `lxml` to extract the image data from the form. This image data is prepended with a header that defines the type of data. In this case, it is a PNG image encoded in Base64, which is a format used to represent binary data in ASCII. This header is removed by partitioning on the first comma. Then, this image data needs to be decoded from Base64 into the original binary format. To load an image, `PIL` expects a file-like interface, so this binary data is wrapped with `BytesIO` and then passed to the `Image` class.

Now that we have the CAPTCHA image in a more useful format, we are ready to attempt extracting the text.

Pillow vs PIL

`Pillow` is a fork of the better known **Python Image Library** (**PIL**), which currently has not been updated since 2009. It uses the same interface as the original `PIL` package and is well documented at `http://pillow.readthedocs.org`. The examples in this chapter will work with both `Pillow` and `PIL`.

Optical Character Recognition

Optical Character Recognition (**OCR**) is a process to extract text from images. In this section, we will use the open source Tesseract OCR engine, which was originally developed at HP and now primarily at Google. Installation instructions for Tesseract are available at `https://code.google.com/p/tesseract-ocr/wiki/ReadMe`. Then, the `pytesseract` Python wrapper can be installed with `pip`:

```
pip install pytesseract
```

If the original CAPTCHA image is passed to `pytesseract`, the results are terrible:

```
>>> import pytesseract
>>> img = get_captcha(html)
>>> pytesseract.image_to_string(img)
''
```

An empty string was returned, which means Tesseract failed to extract any characters from the input image. Tesseract was designed to extract more typical types of text, such as book pages with a consistent background. If we want to use Tesseract effectively, we will need to first modify the CAPTCHA images to remove the background noise and isolate the text. To better understand the CAPTCHA system we are dealing with, here are some more samples:

The samples in the preceding screenshot show that the CAPTCHA text is always black while the background is lighter, so this text can be isolated by checking each pixel and only keeping the black ones, a process known as **thresholding**. This process is straightforward to achieve with `Pillow`:

```
>>> img.save('captcha_original.png')
>>> gray = img.convert('L')
>>> gray.save('captcha_gray.png')
>>> bw = gray.point(lambda x: 0 if x < 1 else 255, '1')
>>> bw.save('captcha_thresholded.png')
```

Here, a threshold of less than `1` was used, which means only keeping pixels that are completely black. This snippet saved three images—the original CAPTCHA image, when converted to grayscale, and after thresholding. Here are the images saved at each stage:

The text in the final thresholded image is much clearer and is ready to be passed to Tesseract:

```
>>> pytesseract.image_to_string(bw)
'strange'
```

Success! The CAPTCHA text has been successfully extracted. In my test of 100 images, this approach correctly interpreted the CAPTCHA image 84 times. Since the sample text is always lowercase ASCII, the performance can be improved further by restricting the result to these characters:

```
>>> import string
>>> word = pytesseract.image_to_string(bw)
>>> ascii_word = ''.join(c for c in word if c in
    string.letters).lower()
```

In my test on the same sample images, this improved performance to 88 times out of 100. Here is the full code of the registration script so far:

```
import cookielib
import urllib
import urllib2
import string
import pytesseract
REGISTER_URL = 'http://example.webscraping.com/user/register'

def register(first_name, last_name, email, password):
    cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    html = opener.open(REGISTER_URL).read()
    form = parse_form(html)
    form['first_name'] = first_name
    form['last_name'] = last_name
    form['email'] = email
    form['password'] = form['password_two'] = password
    img = extract_image(html)
    captcha = ocr(img)
    form['recaptcha_response_field'] = captcha
    encoded_data = urllib.urlencode(form)
    request = urllib2.Request(REGISTER_URL, encoded_data)
    response = opener.open(request)
    success = '/user/register' not in response.geturl()
    return success


def ocr(img):
    gray = img.convert('L')
    bw = gray.point(lambda x: 0 if x < 1 else 255, '1')
    word = pytesseract.image_to_string(bw)
    ascii_word = ''.join(c for c in word if c in
        string.letters).lower()
    return ascii_word
```

The `register()` function downloads the registration page and scrapes the form as usual, where the desired name, e-mail, and password for the new account are set. The CAPTCHA image is then extracted, passed to the OCR function, and the result is added to the form. This form data is then submitted and the response URL is checked to see if registration was successful. If it fails, this would still be the registration page, either because the CAPTCHA image was solved incorrectly or an account with this e-mail already existed. Now, to register an account, we simply need to call the `register()` function with the new account details:

```
>>> register(first_name, last_name, email, password)
True
```

Further improvements

To improve the CAPTCHA OCR performance further, there are a number of possibilities, as follows:

- Experimenting with different threshold levels
- Eroding the thresholded text to emphasize the shape of characters
- Resizing the image—sometimes increasing the size helps
- Training the OCR tool on the CAPCHA font
- Restricting results to dictionary words

If you are interested in experimenting to improve performance, the sample data used is available at `https://bitbucket.org/wswp/code/src/tip/chapter07/samples/`. However the current 88 percent accuracy is sufficient for our purposes of registering an account, because actual users will also make mistakes when entering the CAPTCHA text. Even 1 percent accuracy would be sufficient, because the script could be run many times until successful, though this would be rather impolite to the server and may lead to your IP being blocked.

Solving complex CAPTCHAs

The CAPTCHA system tested so far was relatively straightforward to solve—the black font color meant the text could easily be distinguished from the background, and additionally, the text was level and did not need to be rotated for Tesseract to interpret it accurately. Often, you will find websites using simple custom CAPTCHA systems similar to this, and in these cases, an OCR solution is practical. However, if a website uses a more complex system, such as Google's reCAPTCHA, OCR will take a lot more effort and may not be practical. Here are some more complex CAPTCHA images from around the web:

In these examples, the text is placed at different angles and with different fonts and colors, so a lot more work needs to be done to clean the image before OCR is practical. They are also somewhat difficult for people to interpret, particularly for those with vision disabilities.

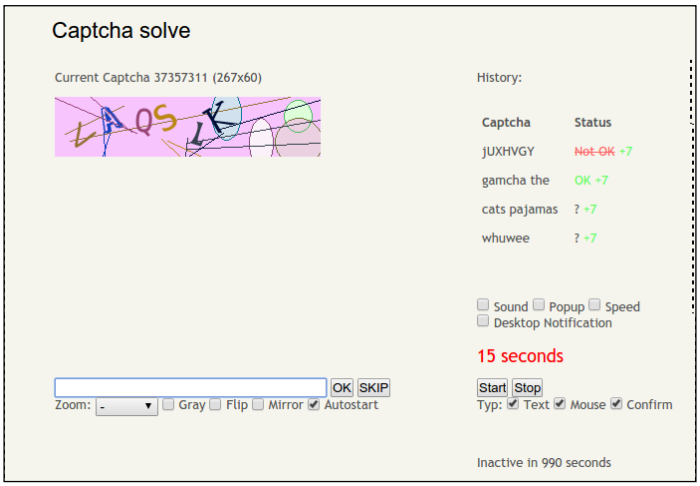
Using a CAPTCHA solving service

To solve these more complex images, we will make use of a CAPTCHA solving service. There are many CAPTCHA solving services available, such as `2captcha.com` and `deathbycaptcha.com`, and they all offer a similar rate of around 1000 CAPTCHAs for $1. When a CAPTCHA image is passed to their API, a person will then manually examine the image and provide the parsed text in an HTTP response, typically within 30 seconds. For the examples in this section, we will use the service at `9kw.eu`, which does not provide the cheapest per CAPTCHA rate or the best designed API. However, on the positive side, it is possible to use the API without spending money. This is because `9kw.eu` allows users to manually solve CAPTCHAs to build up credit, which can then be spent on testing the API with our own CAPTCHAs.

Getting started with 9kw

To start using **9kw**, you will need to first create an account at `https://www.9kw.eu/register.html`:

Then, follow the account confirmation instructions, and when logged in, navigate to `https://www.9kw.eu/usercaptcha.html`:



On this page, you can solve other people's CAPTCHAs to build up credit to later use with the API. After solving a few CAPTCHAs, navigate to `https://www.9kw.eu/index.cgi?action=userapinew&source=api` to create an API key.

9kw CAPTCHA API

The 9kw API is documented at `https://www.9kw.eu/api.html#apisubmit-tab`. The important parts for our purposes to submit a CAPTCHA and check the result are summarized here:

Submit captcha

URL: `https://www.9kw.eu/index.cgi` (POST)

apikey: your API key

action: must be set to "usercaptchaupload"

file-upload-01: the image to solve (either a file, url or string)

base64: set to "1" if the input is Base64 encoded

maxtimeout: the maximum time to wait for a solution
(must be between 60 - 3999 seconds)

selfsolve: set to "1" to solve this CAPTCHA ourself

Return value: ID of this CAPTCHA

Request result of submitted captcha

URL: `https://www.9kw.eu/index.cgi` (GET)

apikey: your API key

action: must be set to "usercaptchacorrectdata"

id: ID of CAPTCHA to check

info: set to "1" to return "NO DATA" when there is not yet a solution
(by default returns nothing)

Return value: Text of the solved CAPTCHA or an error code

Error codes

0001 API key doesn't exist

0002 API key not found

0003 Active API key not found

...

0031 An account is not yet 24 hours in the system.

0032 An account does not have the full rights.

0033 Plugin needs an update.

Here is an initial implementation to send a CAPTCHA image to this API:

```
import urllib
import urllib2
API_URL = 'https://www.9kw.eu/index.cgi'

def send_captcha(api_key, img_data):
    data = {
        'action': 'usercaptchaupload',
        'apikey': api_key,
        'file-upload-01': img_data.encode('base64'),
        'base64': '1',
        'selfsolve': '1',
        'maxtimeout': '60'
    }
    encoded_data = urllib.urlencode(data)
    request = urllib2.Request(API_URL, encoded_data)
    response = urllib2.urlopen(request)
    return response.read()
```

This structure should hopefully be looking familiar by now—first, build a dictionary of the required parameters, encode them, and then submit this in the body of your request. Note that the `selfsolve` option is set to `'1'`: this means that if we are currently solving CAPTCHAs at the 9kv web interface, this CAPTCHA image will be passed to us to solve, which saves us credit. If not logged in, the CAPTCHA image is passed to another user to solve as usual.

Here is the code to fetch the result of a solved CAPTCHA image:

```
def get_captcha(api_key, captcha_id):
    data = {
        'action': 'usercaptchacorrectdata',
        'id': captcha_id,
        'apikey': api_key
    }
    encoded_data = urllib.urlencode(data)
    # note this is a GET request
    # so the data is appended to the URL
    response = urllib2.urlopen(API_URL + '?' + encoded_data)
    return response.read()
```

A drawback with the 9kw API is that the response is a plain string rather than a structured format, such as JSON, which makes distinguishing the error messages more complex. For example, if no user is available to solve the CAPTCHA image in time, the `ERROR NO USER` string is returned. Hopefully, the CAPTCHA image we submit never includes this text!

Another difficulty is that the `get_captcha()` function will return error messages until another user has had time to manually examine the CAPTCHA image, as mentioned earlier, typically 30 seconds later. To make our implementation friendlier, we will add a wrapper function to submit the CAPTCHA image and wait until the result is ready. Here is an expanded version that wraps this functionality in a reusable class, as well as checking for error messages:

```
import time
import urllib
import urllib2
import re
from io import BytesIO

class CaptchaAPI:
    def __init__(self, api_key, timeout=60):
        self.api_key = api_key
        self.timeout = timeout
        self.url = 'https://www.9kw.eu/index.cgi'

    def solve(self, img):
        """Submit CAPTCHA and return result when ready
        """
        img_buffer = BytesIO()
        img.save(img_buffer, format="PNG")
        img_data = img_buffer.getvalue()
```

```
            captcha_id = self.send(img_data)
            start_time = time.time()
            while time.time() < start_time + self.timeout:
                try:
                    text = self.get(captcha_id)
                except CaptchaError:
                    pass # CAPTCHA still not ready
                else:
                    if text != 'NO DATA':
                        if text == 'ERROR NO USER':
                            raise CaptchaError('Error: no user
                                available to solve CAPTCHA')
                        else:
                            print 'CAPTCHA solved!'
                            return text
                print 'Waiting for CAPTCHA ...'

            raise CaptchaError('Error: API timeout')

        def send(self, img_data):
            """Send CAPTCHA for solving
            """
            print'Submitting CAPTCHA'
            data = {
                'action': 'usercaptchaupload',
                'apikey': self.api_key,
                'file-upload-01': img_data.encode('base64'),
                'base64': '1',
                'selfsolve': '1',
                'maxtimeout': str(self.timeout)
            }
            encoded_data = urllib.urlencode(data)
            request = urllib2.Request(self.url, encoded_data)
            response = urllib2.urlopen(request)
            result = response.read()
            self.check(result)
            return result

        def get(self, captcha_id):
            """Get result of solved CAPTCHA
            """
            data = {
                'action': 'usercaptchacorrectdata',
                'id': captcha_id,
```

```
            'apikey': self.api_key,
            'info': '1'
        }
        encoded_data = urllib.urlencode(data)
        response = urllib2.urlopen(self.url + '?' + encoded_data)
        result = response.read()
        self.check(result)
        return result

    def check(self, result):
        """Check result of API and raise error if error code
        """
        if re.match('00\d\d \w+', result):
            raise CaptchaError('API error: ' + result)


class CaptchaError(Exception):
    pass
```

The source for the `CaptchaAPI` class is also available at `https://bitbucket.org/wswp/code/src/tip/chapter07/api.py`, which will be kept updated if 9kw.eu modifies their API. This class is instantiated with your API key and a timeout, by default, set to 60 seconds. Then, the `solve()` method submits a CAPTCHA image to the API and keeps requesting the solution until either the CAPTCHA image is solved or a timeout is reached. To check for error messages in the API response, the `check()` method merely examines whether the initial characters follow the expected format of four digits for the error code before the error message. For more robust use of this API, this method could be expanded to cover each of the 34 error types.

Here is an example of solving a CAPTCHA image with the `CaptchaAPI` class:

```
>>> API_KEY = ...
>>> captcha = CaptchaAPI(API_KEY)
>>> img = Image.open('captcha.png')
>>> text = captcha.solve(img)
Submitting CAPTCHA
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
```

```
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
CAPTCHA solved!
>>> text
juxhvgy
```

This is the correct solution for the first complex CAPTCHA image shown earlier in this chapter. If the same CAPTCHA image is submitted again soon after, the cached result is returned immediately and no additional credit is used:

```
>>> text = captcha.solve(img_data)
Submitting CAPTCHA
>>> text
juxhvgy
```

Integrating with registration

Now that we have a working CAPTCHA API solution, we can integrate it with the previous form. Here is a modified version of the `register` function that now takes a function to solve the CAPTCHA image as an argument so that it can be used with either the OCR or API solutions:

```
def register(first_name, last_name, email, password, captcha_fn):
    cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    html = opener.open(REGISTER_URL).read()
    form = parse_form(html)
    form['first_name'] = first_name
    form['last_name'] = last_name
    form['email'] = email
    form['password'] = form['password_two'] = password
    img = extract_image(html)
    form['recaptcha_response_field'] =  captcha_fn(img)
    encoded_data = urllib.urlencode(form)
    request = urllib2.Request(REGISTER_URL, encoded_data)
    response = opener.open(request)
    success = '/user/register' not in response.geturl()
    return success
```

Here is an example of how to use it:

```
>>> captcha = CaptchaAPI(API_KEY)
>>> register(first_name, last_name, email, password, captcha.solve)
Submitting CAPTCHA
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
```

```
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
Waiting for CAPTCHA ...
True
```

It worked! The CAPTCHA image was successfully extracted from the form, submitted to the 9kv API, solved manually by another user, and then the result was submitted to the web server to register a new account.

Summary

This chapter showed how to solve CAPTCHAs, first by using OCR, and then with an external API. For simple CAPTCHAs, or for when you need to solve a large amount of CAPTCHAs, investing time in an OCR solution can be worthwhile. Otherwise, using a CAPTCHA solving API can prove to be a cost effective alternative.

In the next chapter, we will introduce Scrapy, which is a popular high-level framework used to build scraping applications.

8
Scrapy

Scrapy is a popular web scraping framework that comes with many high-level functions to make scraping websites easier. In this chapter, we will get to know Scrapy by using it to scrape the example website, just as we did in *Chapter 2*, *Scraping the Data*. Then, we will cover **Portia**, which is an application based on Scrapy that allows you to scrape a website through a point and click interface

Installation

Scrapy can be installed with the `pip` command, as follows:

```
pip install Scrapy
```

Scrapy relies on some external libraries so if you have trouble installing it there is additional information available on the official website at: `http://doc.scrapy.org/en/latest/intro/install.html`.

Currently, Scrapy only supports Python 2.7, which is more restrictive than other packages introduced in this book. Previously, Python 2.6 was also supported, but this was dropped in Scrapy 0.20. Also due to the dependency on Twisted, support for Python 3 is not yet possible, though the Scrapy team assures me they are working to solve this.

If Scrapy is installed correctly, a `scrapy` command will now be available in the terminal:

```
$ scrapy -h
Scrapy 0.24.4 - no active project

Usage:
  scrapy <command> [options] [args]

Available commands:
  bench         Run quick benchmark test
  check         Check spider contracts
  crawl         Run a spider
...
```

We will use the following commands in this chapter:

- `startproject`: Creates a new project
- `genspider`: Generates a new spider from a template
- `crawl`: Runs a spider
- `shell`: Starts the interactive scraping console

For detailed information about these and the other commands available, refer to `http://doc.scrapy.org/en/latest/topics/commands.html`.

Starting a project

Now that Scrapy is installed, we can run the `startproject` command to generate the default structure for this project. To do this, open the terminal and navigate to the directory where you want to store your Scrapy project, and then run `scrapy startproject <project name>`. Here, we will use `example` for the project name:

```
$ scrapy startproject example
$ cd example
```

Here are the files generated by the `scrapy` command:

```
scrapy.cfg
example/
    __init__.py
    items.py
    pipelines.py
    settings.py
    spiders/
        __init__.py
```

The important files for this chapter are as follows:

- `items.py`: This file defines a model of the fields that will be scraped
- `settings.py`: This file defines settings, such as the user agent and crawl delay
- `spiders/`: The actual scraping and crawling code are stored in this directory

Additionally, Scrapy uses `scrapy.cfg` for project configuration and `pipelines.py` to process the scraped fields, but they will not need to be modified in this example.

Defining a model

By default, `example/items.py` contains the following code:

```
import scrapy

class ExampleItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    pass
```

The `ExampleItem` class is a template that needs to be replaced with how we want to store the scraped country details when the spider is run. To help focus on what Scrapy does, we will just scrape the country name and population, rather than all the country details. Here is an updated model to support this:

```
class ExampleItem(scrapy.Item):
    name = scrapy.Field()
    population = scrapy.Field()
```

Full documentation about defining models is available at `http://doc.scrapy.org/en/latest/topics/items.html`.

Creating a spider

Now, we will build the actual crawling and scraping code, known as a **spider** in Scrapy. An initial template can be generated with the `genspider` command, which takes the name you want to call the spider, the domain, and optionally, a template:

```
$ scrapy genspider country example.webscraping.com --template=crawl
```

The built-in `crawl` template was used here to generate an initial version closer to what we need to crawl the countries. After running the `genspider` command, the following code will have been generated in `example/spiders/country.py`:

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

from example.items import ExampleItem

class CountrySpider(CrawlSpider):
    name = 'country'
    start_urls = ['http://www.example.webscraping.com/']
    allowed_domains = ['example.webscraping.com']

    rules = (
        Rule(LinkExtractor(allow=r'Items/'),
            callback='parse_item', follow=True),
    )

    def parse_item(self, response):
        i = ExampleItem()
        #i['domain_id'] =
            response.xpath('//input[@id="sid"]/@value').extract()
        #i['name'] = response.xpath('//div[@id="name"]').extract()
        #i['description'] =
            response.xpath('//div[@id="description"]').extract()
        return i
```

The initial lines import the required Scrapy libraries, including the `ExampleItem` model defined in the *Defining a model* section. Then, a class is created for the spider, which contains a number of class attributes, as follows:

- `name`: This attribute is a string to identify the spider
- `start_urls`: This attribute is a list of URLs to start the crawl. However, the `start_urls` default attribute is not what we want because `www` has been prepended to the `example.webscraping.com` domain
- `allowed_domains`: This attribute is a list of the domains that can be crawled—if this is not defined, any domain can be crawled
- `rules`: This attribute is a set of regular expressions to tell the crawler which links to follow

The `rules` attribute also has a `callback` function to parse the responses of these downloads, and the `parse_item()` example method gives an example of how to scrape data from the response.

Scrapy is a high-level framework, so there is a lot going on here in these few lines of code. The official documentation has further details about building spiders, and can be found at `http://doc.scrapy.org/en/latest/topics/spiders.html`.

Tuning settings

Before running the generated spider, the Scrapy settings should be updated to avoid the spider being blocked. By default, Scrapy allows up to eight concurrent downloads for a domain with no delay between downloads, which is much faster than a real user would browse, and so is straightforward for a server to detect. As mentioned in the Preface, the example website that we are scraping is configured to temporarily block crawlers that consistently download at faster than one request a second, so the default settings would cause our spider to be blocked. Unless you are running the example website locally then I recommend adding these lines to `example/settings.py` so that the crawler only downloads a single request at a time per domain with a delay between downloads:

```
CONCURRENT_REQUESTS_PER_DOMAIN = 1
DOWNLOAD_DELAY = 5
```

Note that Scrapy will not use this exact delay between requests, because this would also make a crawler easier to detect and block. Instead, it adds a random offset to this delay between requests. For details about these settings and the many others available, refer to `http://doc.scrapy.org/en/latest/topics/settings.html`.

Testing the spider

To run a spider from the command line, the `crawl` command is used along with the name of the spider:

```
$ scrapy crawl country -s LOG_LEVEL=ERROR
[country] ERROR: Error downloading <GET http://www.example.webscraping.
com/>: DNS lookup failed: address 'www.example.webscraping.com' not
found: [Errno -5] No address associated with hostname.
```

As expected, the crawl failed on this default spider because `http://www.example.webscraping.com` does not exist. Take note of the `-s LOG_LEVEL=ERROR` flag—this is a Scrapy setting and is equivalent to defining `LOG_LEVEL = 'ERROR'` in the `settings.py` file. By default, Scrapy will output all log messages to the terminal, so here the log level was raised to isolate the error messages.

Here is an updated version of the spider to correct the starting URL and set which web pages to crawl:

```
    start_urls = ['http://example.webscraping.com/']

    rules = (
        Rule(LinkExtractor(allow='/index/'), follow=True),
        Rule(LinkExtractor(allow='/view/'), callback='parse_item')
    )
```

The first rule will crawl the index pages and follow their links, and then the second rule will crawl the country pages and pass the downloaded response to the `callback` function for scraping. Let us see what happens when this spider is run with the log level set to `DEBUG` to show all messages:

```
$ scrapy crawl country -s LOG_LEVEL=DEBUG
...
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/>
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/
index/1>
[country] DEBUG: Filtered duplicate request: <GET http://example.
webscraping.com/index/1> - no more duplicates will be shown (see
DUPEFILTER_DEBUG to show all duplicates)
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/view/
Antigua-and-Barbuda-10>
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/user/
login?_next=%2Findex%2F1>
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/user/
register?_next=%2Findex%2F1>
...
```

This log output shows that the index pages and countries are being crawled and duplicate links are filtered, which is handy. However, the spider is wasting resources by also crawling the login and register forms linked from each web page, because they match the `rules` regular expressions. The login URL in the preceding command ends with `_next=%2Findex%2F1`, which is a URL encoding equivalent to `_next=/index/1`, and would let the server know where to redirect to after the user logged in. To prevent crawling these URLs, we can use the `deny` parameter of the rules, which also expects a regular expression and will prevent crawling all matching URLs. Here is an updated version of code to prevent crawling the user login and registration forms by avoiding the URLs containing `/user/`:

```
rules = (
    Rule(LinkExtractor(allow='/index/', deny='/user/'),
        follow=True),
    Rule(LinkExtractor(allow='/view/', deny='/user/'),
        callback='parse_item')
)
```

Further documentation about how to use this class is available at `http://doc.scrapy.org/en/latest/topics/link-extractors.html`.

Scraping with the shell command

Now that Scrapy can crawl the countries, we need to define what data should be scraped. To help test how to extract data from a web page, Scrapy comes with a handy command called `shell` that will download a URL and provide the resulting state in the Python interpreter. Here are the results for a sample country:

```
$ scrapy shell http://example.webscraping.com/view/United-Kingdom-239
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x7f1475da5390>
[s]   item       {}
[s]   request    <GET http://example.webscraping.com/view/United-
Kingdom-239>
[s]   response   <200 http://example.webscraping.com/view/United-
Kingdom-239>
[s]   settings   <scrapy.settings.Settings object at 0x7f147c1fb490>
[s]   spider     <Spider 'default' at 0x7f147552eb90>
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser
```

We can now query these objects to check what data is available.

```
In [1]: response.url
'http://example.webscraping.com/view/United-Kingdom-239'
In [2]: response.status
200
```

Scrapy uses `lxml` to scrape data, so we can use the same CSS selectors as those in *Chapter 2*, *Scraping the Data*:

```
In [3]: response.css('tr#places_country__row td.w2p_fw::text')
[<Selector xpath=u"descendant-or-self::
    tr[@id = 'places_country__row']/descendant-or-self::
    */td[@class and contains(
    concat(' ', normalize-space(@class), ' '),
    ' w2p_fw ')]/text()" data=u'United Kingdom'>]
```

This method returns an `lxml` selector; to apply it, the `extract()` method needs to be called:

```
In [4]: name_css = 'tr#places_country__row td.w2p_fw::text'
In [5]: response.css(name_css).extract()
[u'United Kingdom']
In [6]: pop_css = 'tr#places_population__row td.w2p_fw::text'
In [7]: response.css(pop_css).extract()
[u'62,348,447']
```

These CSS selectors can then be used in the `parse_item()` method generated earlier in `example/spiders/country.py`:

```
def parse_item(self, response):
    item = ExampleItem()
    name_css = 'tr#places_country__row td.w2p_fw::text'
    item['name'] = response.css(name_css).extract()
    pop_css = 'tr#places_population__row td.w2p_fw::text'
    item['population'] = response.css(pop_css).extract()
    return item
```

Checking results

Here is the completed version of our spider:

```
class CountrySpider(CrawlSpider):
    name = 'country'
    start_urls = ['http://example.webscraping.com/']
    allowed_domains = ['example.webscraping.com']
    rules = (
```

```
            Rule(LinkExtractor(allow='/index/', deny='/user/'),
                follow=True),
            Rule(LinkExtractor(allow='/view/', deny='/user/'),
                callback='parse_item')
        )

        def parse_item(self, response):
            item = ExampleItem()
            name_css = 'tr#places_country__row td.w2p_fw::text'
            item['name'] = response.css(name_css).extract()
            pop_css = 'tr#places_population__row td.w2p_fw::text'
            item['population'] = response.css(pop_css).extract()
            return item
```

To save the results, we could add extra code to the `parse_item()` method to write the scraped country data, or perhaps define a pipeline. However, this work is not necessary because Scrapy provides a handy `--output` flag to save scraped items automatically in CSV, JSON, or XML format. Here are the results when the final version of the spider is run with the results output to a CSV file and the log level is set to `INFO`, to filter out less important messages:

```
$ scrapy crawl country --output=countries.csv -s LOG_LEVEL=INFO
[scrapy] INFO: Scrapy 0.24.4 started (bot: example)
[country] INFO: Spider opened
[country] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0
items/min)
[country] INFO: Crawled 10 pages (at 10 pages/min), scraped 9 items (at 9
items/min)
...
[country] INFO: Crawled 264 pages (at 10 pages/min), scraped 238 items
(at 9 items/min)
[country] INFO: Crawled 274 pages (at 10 pages/min), scraped 248 items
(at 10 items/min)
[country] INFO: Closing spider (finished)
[country] INFO: Stored csv feed (252 items) in: countries.csv
[country] INFO: Dumping Scrapy stats:
  {'downloader/request_bytes': 155001,
   'downloader/request_count': 279,
   'downloader/request_method_count/GET': 279,
   'downloader/response_bytes': 943190,
```

```
    'downloader/response_count': 279,
    'downloader/response_status_count/200': 279,
    'dupefilter/filtered': 61,
    'finish_reason': 'finished',
    'item_scraped_count': 252,
    'log_count/INFO': 36,
    'request_depth_max': 26,
    'response_received_count': 279,
    'scheduler/dequeued': 279,
    'scheduler/dequeued/memory': 279,
    'scheduler/enqueued': 279,
    'scheduler/enqueued/memory': 279}
[country] INFO: Spider closed (finished)
```

At the end of the crawl, Scrapy outputs some statistics to give an indication of how the crawl performed. From these statistics, we know that 279 web pages were crawled and 252 items were scraped, which is the expected number of countries in the database, so we know that the crawler was able to find them all.

To verify these countries were scraped correctly we can check the contents of `countries.csv`:

```
    name,population
    Afghanistan,"29,121,286"
    Antigua and Barbuda,"86,754"
    Antarctica,0
    Anguilla,"13,254"
    Angola,"13,068,161"
    Andorra,"84,000"
    American Samoa,"57,881"
    Algeria,"34,586,184"
    Albania,"2,986,952"
    Aland Islands,"26,711"
    ...
```

As expected this spreadsheet contains the name and population for each country. Scraping this data required writing less code than the original crawler built in *Chapter 2, Scraping the Data* because Scrapy provides a lot of high-level functionalities. In the following section on Portia we will re-implement this scraper writing even less code.

Interrupting and resuming a crawl

Sometimes when scraping a website, it can be useful to pause the crawl and resume it later without needing to start over from the beginning. For example, you may need to interrupt the crawl to reset your computer after a software update, or perhaps, the website you are crawling is returning errors and you want to continue the crawl later. Conveniently, Scrapy comes built-in with support to pause and resume crawls without needing to modify our example spider. To enable this feature, we just need to define the `JOBDIR` setting for the directory where the current state of a crawl is saved. Note that separate directories must be used to save the state of multiple crawls. Here is an example using this feature with our spider:

```
$ scrapy crawl country -s LOG_LEVEL=DEBUG -s JOBDIR=crawls/country
...
[country] DEBUG: Scraped from <200 http://example.webscraping.com/view/
Afghanistan-1>
  {'name': [u'Afghanistan'], 'population': [u'29,121,286']}
^C  [scrapy] INFO: Received SIGINT, shutting down gracefully. Send again
to force
[country] INFO: Closing spider (shutdown)
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/view/
Antigua-and-Barbuda-10> (referer: http://example.webscraping.com/)
[country] DEBUG: Scraped from <200 http://example.webscraping.com/view/
Antigua-and-Barbuda-10>
  {'name': [u'Antigua and Barbuda'], 'population': [u'86,754']}
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/view/
Antarctica-9> (referer: http://example.webscraping.com/)
[country] DEBUG: Scraped from <200 http://example.webscraping.com/view/
Antarctica-9>
  {'name': [u'Antarctica'], 'population': [u'0']}
...
[country] INFO: Spider closed (shutdown)
```

Here, we see that `^C` (*Ctrl* + *C*) was used to send the terminate signal, and that the spider finished processing a few items before terminating. To have Scrapy save the crawl state, you must wait here for the crawl to shut down gracefully and resist the temptation to enter *Ctrl* + *C* again to force immediate termination! The state of the crawl will now be saved in `crawls/country`, and the crawl can be resumed by running the same command:

```
$ scrapy crawl country -s LOG_LEVEL=DEBUG -s JOBDIR=crawls/country
...
```

```
[country] INFO: Resuming crawl (12 requests scheduled)
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/view/
Anguilla-8> (referer: http://example.webscraping.com/)
[country] DEBUG: Scraped from <200 http://example.webscraping.com/view/
Anguilla-8>
  {'name': [u'Anguilla'], 'population': [u'13,254']}
[country] DEBUG: Crawled (200) <GET http://example.webscraping.com/view/
Angola-7> (referer: http://example.webscraping.com/)
[country] DEBUG: Scraped from <200 http://example.webscraping.com/view/
Angola-7>
  {'name': [u'Angola'], 'population': [u'13,068,161']}
...
```

The crawl now resumes from where it was paused and continues as normal. This feature is not particularly useful for our example website because the number of pages to download is very small. However, for larger websites that take months to crawl, being able to pause and resume crawls is very convenient.

Note that there are some edge cases not covered here that can cause problems when resuming a crawl, such as expiring cookies, which are mentioned in the Scrapy documentation available at `http://doc.scrapy.org/en/latest/topics/jobs.html`.

Visual scraping with Portia

Portia is a an open-source tool built on top of Scrapy that supports building a spider by clicking on the parts of a website that need to be scraped, which can be more convenient than creating the CSS selectors manually.

Installation

Portia is a powerful tool, and it depends on multiple external libraries for its functionality. It is also relatively new, so currently, the installation steps are somewhat involved. In case the installation is simplified in future, the latest documentation can be found at `https://github.com/scrapinghub/portia#running-portia`.

The recommended first step is to create a virtual Python environment with `virtualenv`. Here, we name our environment `portia_example`, which can be replaced with whatever name you choose:

```
$ pip install virtualenv
$ virtualenv portia_example --no-site-packages
$ source portia_example/bin/activate
(portia_example)$ cd portia_example
```

Why use virtualenv?

Imagine if your project was developed with an earlier version of `lxml`, and then in a later version, `lxml` introduced some backwards incompatible changes that would break your project. However, other projects are going to depend on the newer version of `lxml`. If your project uses the system-installed `lxml`, it is eventually going to break when `lxml` is updated to support other projects.

Ian Bicking's `virtualenv` provides a clever hack to this problem by copying the system Python executable and its dependencies into a local directory to create an isolated Python environment. This allows a project to install specific versions of Python libraries locally and independent of the wider system. Further details are available in the documentation at `https://virtualenv.pypa.io`.

Then, from within `virtualenv`, Portia and its dependencies can be installed:

```
(portia_example)$ git clone https://github.com/scrapinghub/portia
(portia_example)$ cd portia
(portia_example)$ pip install -r requirements.txt
(portia_example)$ pip install -e ./slybot
```

Portia is under active development, so the interface may have changed by the time you read this. If you want to use the same version as the one this tutorial was developed with, run this `git` command:

```
(portia_example)$ git checkout 8210941
```

If you do not have `git` installed, the latest version of Portia can be downloaded directly from `https://github.com/scrapinghub/portia/archive/master.zip`.

Once the installation steps are completed, Portia can be started by changing to the slyd directory and starting the server:

```
(portia_example)$ cd slyd
(portia_example)$ twistd -n slyd
```

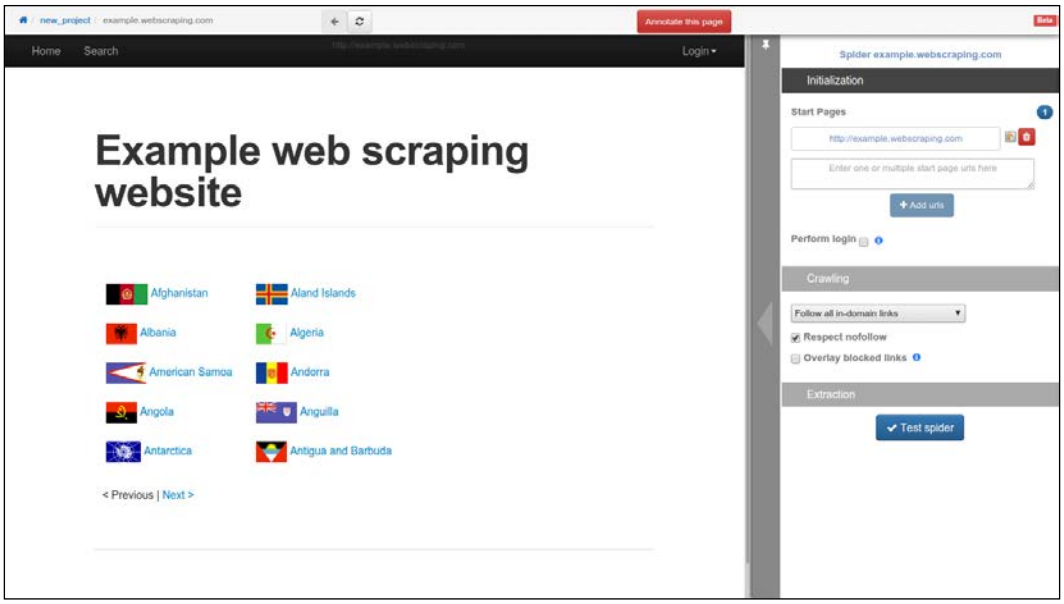
If installed correctly, the Portia tool will now be accessible in your web browser at `http://localhost:9001/static/main.html`.

This is how the initial screen looks:

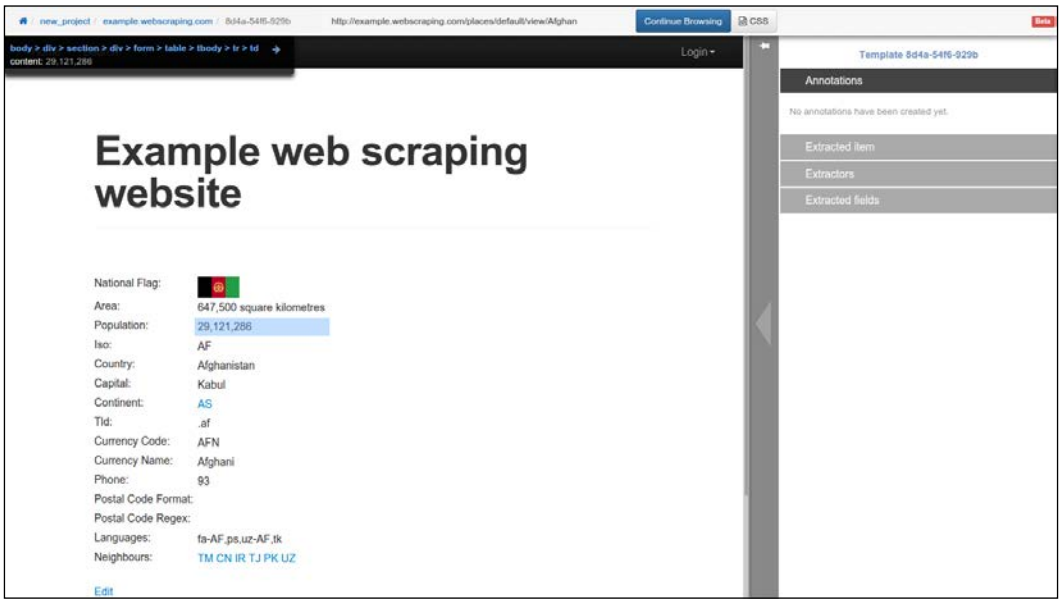
If you have problems during installation it is worth checking the Portia Issues page at `https://github.com/scrapinghub/portia/issues`, in case someone else has experienced the same problem and found a solution.

Annotation

At the Portia start page, there is a textbox to enter the URL of the website you want to scrape, such as `http://example.webscraping.com`. Portia will then load this web page in the main panel:

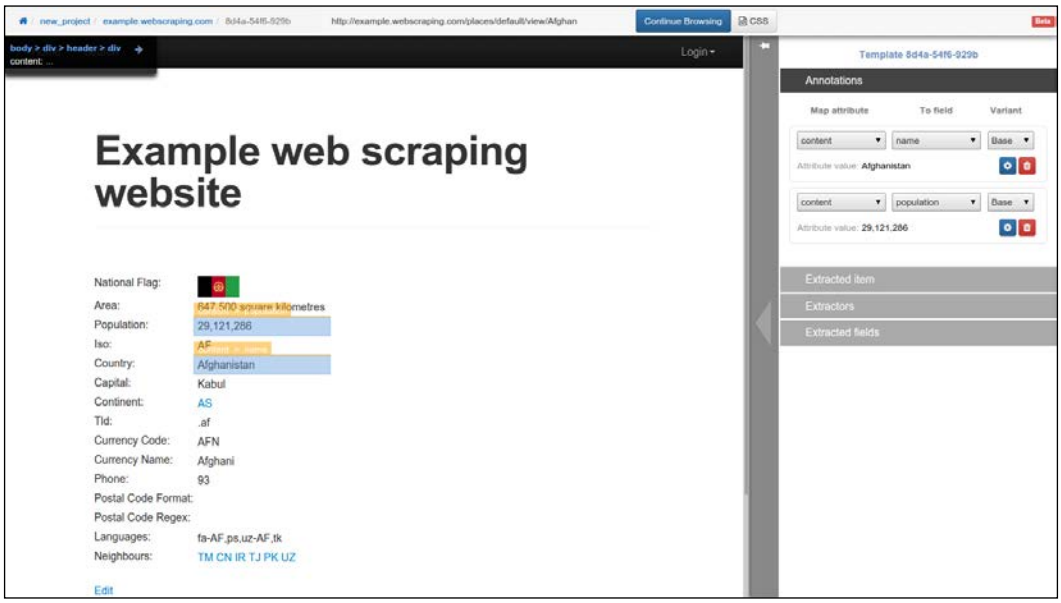


By default, the project name is set to **new_project**, and the spider name is set to the domain (**example.webscraping.com**), which can both be modified by clicking on these labels. Next, navigate to a sample country web page to annotate what data is of interest to you:



Click on the **Annotate this page** button, and then when the country population is clicked on, this dialogue box will pop-up:

Click on the **+ field** button to create a new field called `population`, and click on **Done** to save the new field. Then, do the same for the country name and any other fields you are interested in. The annotated fields will be highlighted in the web page and can be edited in the panel on the right-hand side:



When the annotations are complete, click on the blue **Continue Browsing** button at the top.

Tuning a spider

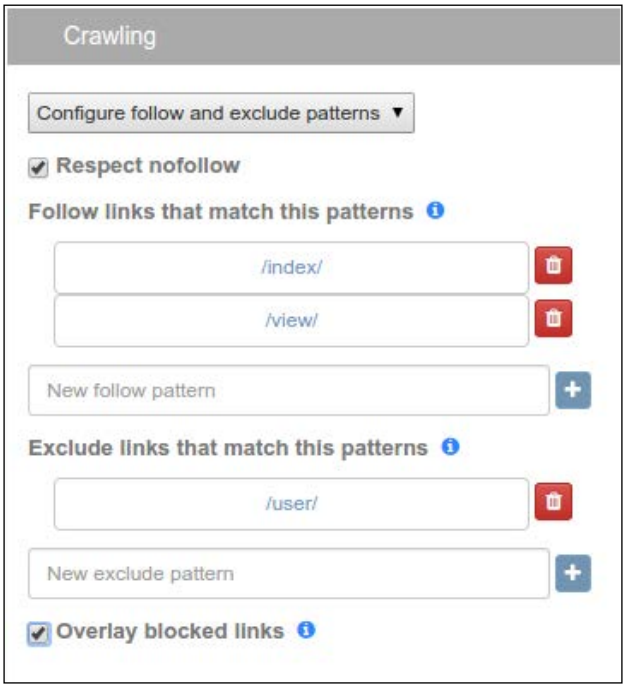
After completing the annotations, Portia will generate a Scrapy project and store the resulting files in `data/projects`. To run this spider, the `portiacrawl` command is used along with the project and spider name. However, if this spider is run using the default settings, it will quickly encounter server errors:

```
(portia_example)$ portiacrawl portia/slyd/data/projects/new_project
example.webscraping.com [example.webscraping.com] DEBUG: Crawled (200)
<GET http://example.webscraping.com/view/Antarctica-9>

[example.webscraping.com] DEBUG: Scraped from <200 http://example.
webscraping.com/view/Antarctica-9>

  {'_template': '9300cdc044d4b75151044340118ccf4efd976922',

   '_type': u'default',

   u'name': [u'Antarctica'],

   u'population': [u'0'],

   'url': 'http://example.webscraping.com/view/Antarctica-9'}

...

[example.webscraping.com] DEBUG: Retrying <GET http://example.
webscraping.com/edit/Antarctica-9> (failed 1 times): 500 Internal Server
Error
```

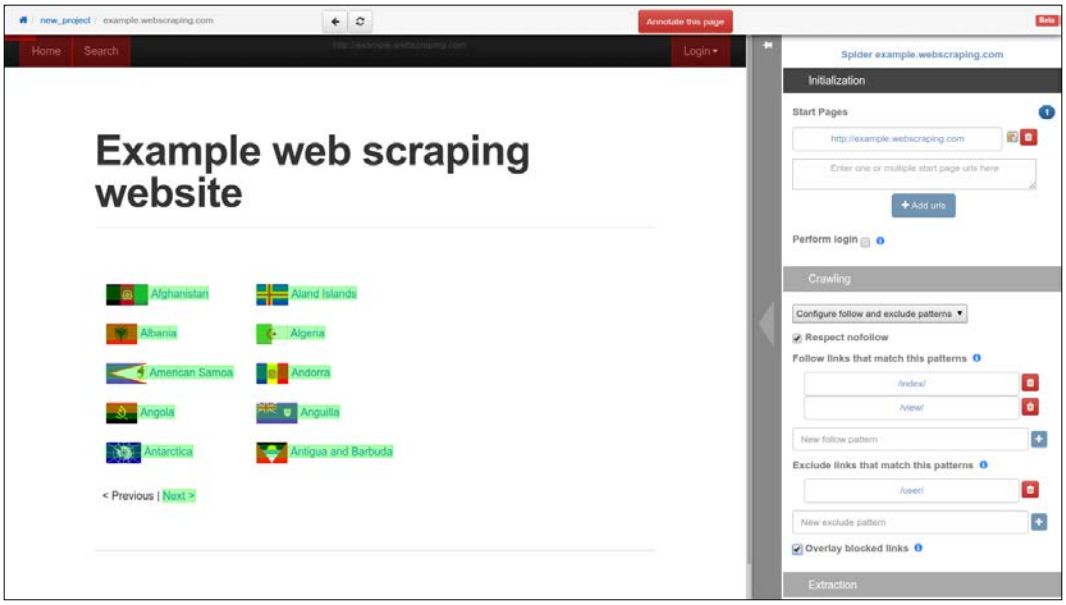
This is the same problem that was touched on in the *Tuning the settings* section because this Portia-generated project uses the default Scrapy crawl settings, which download too fast. These settings can again be modified in the settings file (located at `data/projects/new_project/spiders/settings.py`). However, to demonstrate something new this time, we will set them from the command line:

```
(portia_example)$ portiacrawl portia/slyd/data/projects/new_project
example.webscraping.com -s CONCURRENT_REQUESTS_PER_DOMAIN=1 -s DOWNLOAD_
DELAY=5

...

[example.webscraping.com] DEBUG: Crawled (200) <GET http://example.
webscraping.com/user/login?_next=%2Findex%2F1>

[example.webscraping.com] DEBUG: Crawled (200) <GET http://example.
webscraping.com/user/register?_next=%2Findex%2F1>
```

If this slower crawl is run, it will avoid being blocked. However, it will then encounter the same inefficiency of downloading unnecessary web pages, such as login and registration. By default, Portia will generate a spider that crawls all the URLs of a given domain. To instead crawl only specific URLs, the **Crawling** tab in the right-hand side panel can be configured:



Here, we added `/index/` and `/view/` for the spider follow patterns, and `/user/` for the exclude pattern, similar to what we used in the previous Scrapy project. If the **Overlay blocked links** box at the bottom is checked, Portia will highlight green and red the links that will be followed and excluded respectively:



Checking results

The generated spider is now ready for execution, and, as before, the output CSV file can be specified using the `--output` flag:

```
(portia_example)$ portiacrawl portia/slyd/data/projects/new_project
example.webscraping.com --output=countries.csv -s CONCURRENT_REQUESTS_
PER_DOMAIN=1 -s DOWNLOAD_DELAY=5
```

When this command is run, this spider will produce the exact output as the manually created Scrapy version.

Portia is a handy tool to use in conjunction with Scrapy. For straightforward websites, it will typically be faster to develop the crawler with Portia. On the other hand, for more complex websites—for example, if the interface is JavaScript dependent—there is the option to develop the Scrapy crawler directly in Python.

Automated scraping with Scrapely

For scraping the annotated fields Portia uses a library called **Scrapely**, which is a useful open-source tool developed independently of Portia and is available at `https://github.com/scrapy/scrapely`. Scrapely uses training data to build a model of what to scrape from a web page, and then this model can be applied to scrape other web pages with the same structure in future. Here is an example to show how it works:

```
(portia_example)$ python
>>> from scrapely import Scraper
>>> s = Scraper()
>>> train_url = 'http://example.webscraping.com/view/Afghanistan-1'
>>> s.train(train_url, {'name': 'Afghanistan', 'population':
'29,121,286'})
>>> test_url = 'http://example.webscraping.com/view/United-Kingdom-239'
>>> s.scrape(test_url)
[{u'name': [u'United Kingdom'], u'population': [u'62,348,447']}]
```

First, Scrapely is given the data we want to scrape from the `Afghanistan` web page to train the model, being the country name and population. Then, this model is applied to another country page and Scrapely uses this model to correctly return the country name and population here too.

This workflow allows scraping web pages without needing to know their structure, only the desired content to extract in a training case. This approach can be particularly useful if the content of a web page is static, while the layout is changing. For example, with a news website, the text of the published article will most likely not change, though the layout may be updated. In this case, Scrapely can then be retrained using the same data to generate a model for the new website structure.

The example web page used here to test Scrapely is well structured with separate tags and attributes for each data type so that Scrapely was able to correctly train a model. However, for more complex web pages, Scrapely can fail to locate the content correctly, and so their documentation warns that you should "train with caution". Perhaps, in future, a more robust automated web scraping library will be released, but, for now, it is still necessary to know how to scrape a website directly using the techniques covered in *Chapter 2, Scraping the Data*.

Summary

This chapter introduced Scrapy, a web scraping framework with many high-level features to improve efficiency at scraping websites. Additionally, this chapter covered Portia, which provides a visual interface to generate Scrapy spiders. Finally, we tested Scrapely, the library used by Portia to scrape web pages automatically for a given model.

In the next chapter, we will apply the skills learned so far to some real-world websites.

9
Overview

This book has so far introduced scraping techniques using a custom website, which helped us focus on learning particular skills. Now, in this chapter, we will analyze a variety of real-world websites to show how these techniques can be applied. Firstly, we will use Google to show a real-world search form, then Facebook for a JavaScript-dependent website, Gap for a typical online store, and finally, BMW for a map interface. Since these are live websites, there is a risk that they will have changed by the time you read this. However, this is fine because the purpose of these examples is to show you how the techniques learned so far can be applied, rather than to show you how to scrape a particular website. If you choose to run an example, first check whether the website structure has changed since these examples were made and whether their current terms and conditions prohibit scraping.

Google search engine

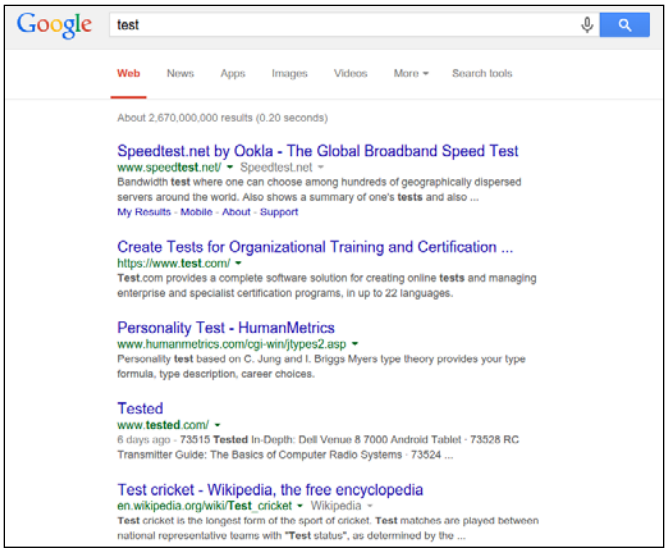
According to the Alexa data used in *Chapter 4*, *Concurrent Downloading*, google.com is the world's most popular website, and conveniently, its structure is simple and straightforward to scrape.

International Google

Google may redirect to a country-specific version, depending on your location. To use a consistent Google search wherever you are in the world, the international English version of Google can be loaded at `http://www.google.com/ncr`. Here, **ncr** stands for **no country redirect**.

Here is the Google search homepage loaded with Firebug to inspect the form:

We can see here that the search query is stored in an input with name `q`, and then the form is submitted to the path `/search` set by the `action` attribute. We can test this by doing a test search to submit the form, which would then be redirected to a URL like `https://www.google.com/searchq=test&oq=test&es_sm=93&ie=UTF-8`. The exact URL will depend on your browser and location. Also note that if you have Google Instant enabled, AJAX will be used to load the search results dynamically rather than submitting the form. This URL has many parameters but the only one required is `q` for the query. The URL `https://www.google.com/search?q=test` will produce the same result, as shown in this screenshot:



The structure of the search results can be examined with Firebug, as shown here:

Here, we see that the search results are structured as links whose parent element is a `<h3>` tag of class "`r`". To scrape the search results we will use a CSS selector, which were introduced in *Chapter 2*, *Scraping the Data*:

```
>>> import lxml.html
>>> from downloader import Downloader
>>> D = Downloader()
>>> html = D('https://www.google.com/search?q=test')
>>> tree = lxml.html.fromstring(html)
>>> results = tree.cssselect('h3.r a')
>>> results
[<Element a at 0x7f3d9affeaf8>,
 <Element a at 0x7f3d9affe890>,
 <Element a at 0x7f3d9affe8e8>,
 <Element a at 0x7f3d9affeaa0>,
 <Element a at 0x7f3d9b1a9e68>,
 <Element a at 0x7f3d9b1a9c58>,
 <Element a at 0x7f3d9b1a9ec0>,
 <Element a at 0x7f3d9b1a9f18>,
 <Element a at 0x7f3d9b1a9f70>,
 <Element a at 0x7f3d9b1a9fc8>]
```

So far, we downloaded the Google search results and used `lxml` to extract the links. In the preceding screenshot, the link includes a bunch of extra parameters alongside the actual website URL, which are used for tracking clicks. Here is the first link:

```
>>> link = results[0].get('href')
>>> link
'/url?q=http://www.speedtest.net/
    &sa=U&ei=nmgqVbgCw&ved=0CB&usg=ACA_cA'
```

The content we want here is `http://www.speedtest.net/`, which can be parsed from the query string using the `urlparse` module:

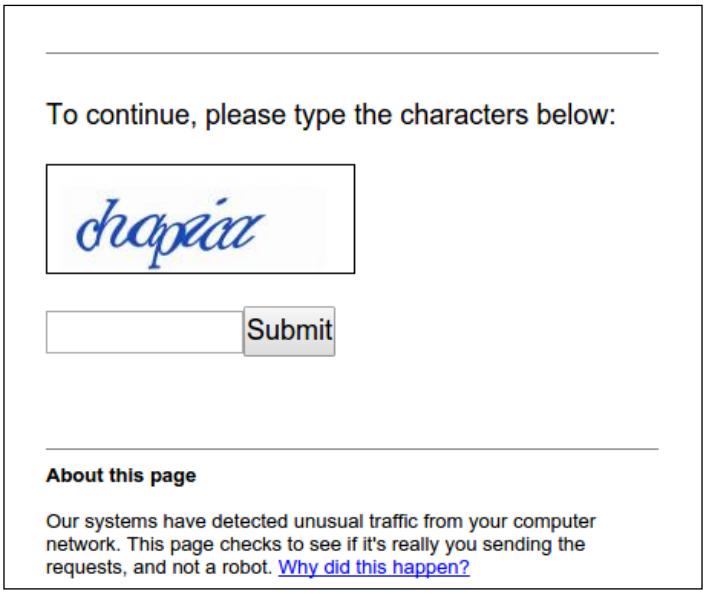
```
>>> import urlparse
>>> qs = urlparse.urlparse(link).query
>>> urlparse.parse_qs(qs)
{'q': ['http://www.speedtest.net/'],
 'ei': ['nmgqVbgCw'],
 'sa': ['U'],
 'usg': ['ACA_cA'],
 'ved': ['0CB']}
>>> urlparse.parse_qs(qs).get('q', [])
['http://www.speedtest.net/']
```

This query string parsing can be applied to extract all links.

```
>>> links = []
>>> for result in results:
...     link = result.get('href')
...     qs = urlparse.urlparse(link).query
...     links.extend(urlparse.parse_qs(qs).get('q', []))
...
>>> links
['http://www.speedtest.net/',
 'https://www.test.com/',
 'http://www.tested.com/',
 'http://www.speakeasy.net/speedtest/',
 'http://www.humanmetrics.com/cgi-win/jtypes2.asp',
 'http://en.wikipedia.org/wiki/Test_cricket',
 'https://html5test.com/',
 'http://www.16personalities.com/free-personality-test',
 'https://www.google.com/webmasters/tools/mobile-friendly/',
 'http://speedtest.comcast.net/']
```

Success! The links from this Google search have been successfully scraped. The full source for this example is available at `https://bitbucket.org/wswp/code/src/tip/chapter09/google.py`.

One difficulty with Google is that a CAPTCHA image will be shown if your IP appears suspicious, for example, when downloading too fast.



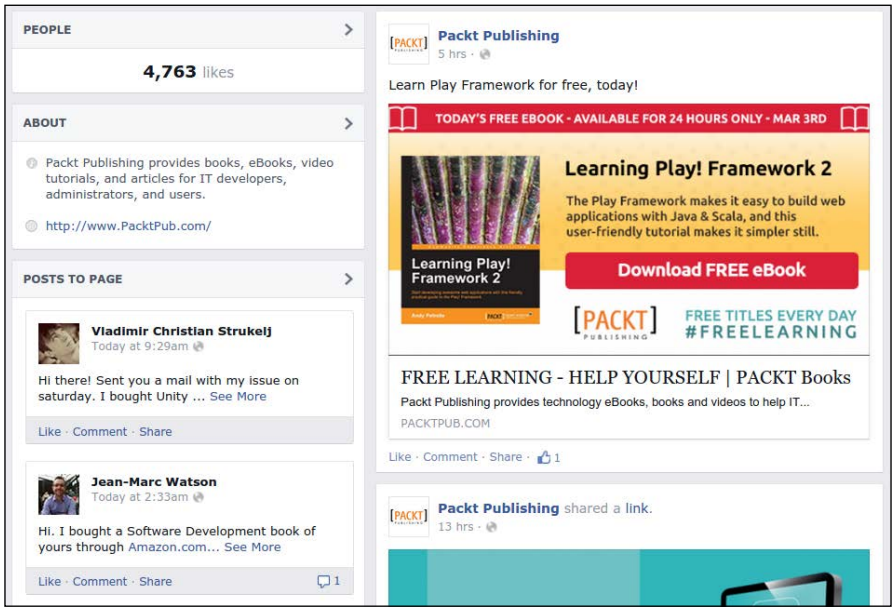
This CAPTCHA image could be solved using the techniques covered in *Chapter 7*, *Solving CAPTCHA*, though it would be preferable to avoid suspicion and download slowly, or use proxies if a faster download rate is required.

Facebook

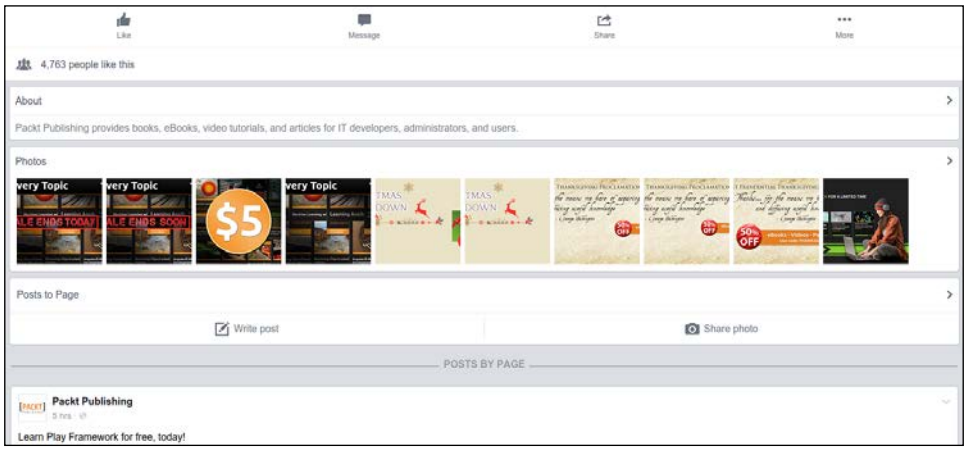
Currently, Facebook is the world's largest social network in terms of monthly active users, and therefore, its user data is extremely valuable.

The website

Here is an example Facebook page for Packt Publishing at `https://www.facebook.com/PacktPub`:



Viewing the source of this page, you would find that the first few posts are available, and that later posts are loaded with AJAX when the browser scrolls. Facebook also has a mobile interface, which, as mentioned in *Chapter 1*, *Introduction to Web Scraping*, is often easier to scrape. The same page using the mobile interface is available at `https://m.facebook.com/PacktPub`:



If we interacted with the mobile website and then checked Firebug we would find that this interface uses a similar structure for the AJAX events, so it is not actually easier to scrape. These AJAX events can be reverse engineered; however, different types of Facebook pages use different AJAX calls, and from my past experience, Facebook often changes the structure of these calls so scraping them will require ongoing maintenance. Therefore, as discussed in *Chapter 5*, *Dynamic Content*, unless performance is crucial, it would be preferable to use a browser rendering engine to execute the JavaScript events for us and give us access to the resulting HTML.

Here is an example snippet using Selenium to automate logging in to Facebook and then redirecting to the given page URL:

```
from selenium import webdriver

def facebook(username, password, url):
    driver = webdriver.Firefox()
    driver.get('https://www.facebook.com')
    driver.find_element_by_id('email').send_keys(username)
    driver.find_element_by_id('pass').send_keys(password)
    driver.find_element_by_id('login_form').submit()
    driver.implicitly_wait(30)
    # wait until the search box is available,
    # which means have successfully logged in
    search = driver.find_element_by_id('q')
    # now logged in so can go to the page of interest
    driver.get(url)
    # add code to scrape data of interest here ...
```

This function can then be called to load the Facebook page of interest and scrape the resulting generated HTML.

The API

As mentioned in *Chapter 1*, *Introduction to Web Scraping*, scraping a website is a last resort when their data is not available in a structured format. Facebook does offer API's for some of their data, so we should check whether these provide access to what we are after before scraping. Here is an example of using Facebook's Graph API to extract data from the Packt Publishing page:

```
>>> import json, pprint
>>> html = D('http://graph.facebook.com/PacktPub')
>>> pprint.pprint(json.loads(html))
{u'about': u'Packt Publishing provides books, eBooks, video
    tutorials, and articles for IT developers, administrators, and
        users.',
 u'category': u'Product/service',
```

```
 u'founded': u'2004',
 u'id': u'204603129458',
 u'likes': 4817,
 u'link': u'https://www.facebook.com/PacktPub',
 u'mission': u'We help the world put software to work in new ways,
    through the delivery of effective learning and information
        services to IT professionals.',
 u'name': u'Packt Publishing',
 u'talking_about_count': 55,
 u'username': u'PacktPub',
 u'website': u'http://www.PacktPub.com'}
```

This API call returns the data in JSON format, which was parsed into a Python `dict` using the `json` module. Then, some useful features, such as the company name, description, and website can be extracted.

The Graph API provides many other calls to access user data and is documented on Facebook's developer page at `https://developers.facebook.com/docs/graph-api`. However, most of these API calls are designed for a Facebook app interacting with an authenticated Facebook user, and are, therefore, not useful for extracting other people's data. To extract additional details, such as user posts, would require scraping.

Gap

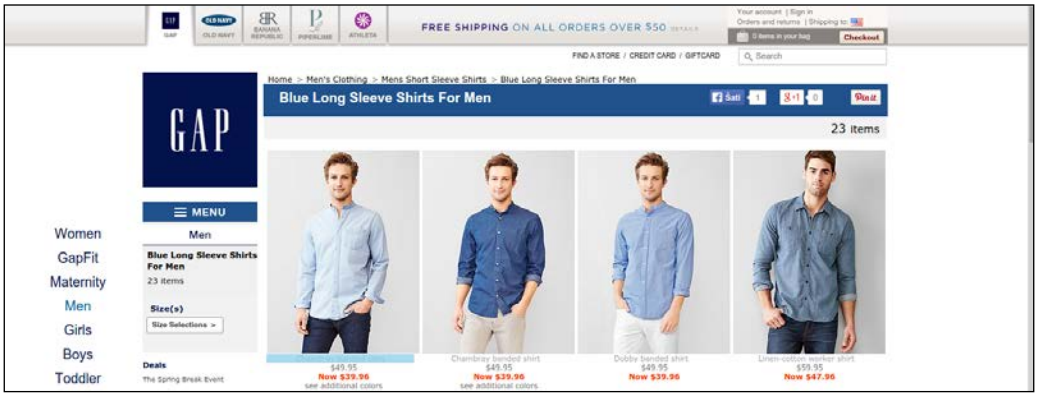
Gap has a well structured website with a `Sitemap` to help web crawlers locate their updated content. If we use the techniques from *Chapter 1*, *Introduction to Web Scraping*, to investigate a website, we would find their `robots.txt` file at `http://www.gap.com/robots.txt`, which contains a link to this Sitemap:

```
Sitemap: http://www.gap.com/products/sitemap_index.xml
```

Here are the contents of the linked `Sitemap` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <sitemap>
        <loc>http://www.gap.com/products/sitemap_1.xml</loc>
        <lastmod>2015-03-03</lastmod>
    </sitemap>
    <sitemap>
        <loc>http://www.gap.com/products/sitemap_2.xml</loc>
        <lastmod>2015-03-03</lastmod>
    </sitemap>
</sitemapindex>
```

As shown here, this `Sitemap` link is just an index and contains links to other `Sitemap` files. These other `Sitemap` files then contain links to thousands of product categories, such as `http://www.gap.com/products/blue-long-sleeve-shirts-for-men.jsp`:



There is a lot of content to crawl here, so we will use the threaded crawler developed in *Chapter 4, Concurrent Downloading*. You may recall that this crawler supports an optional callback for defining how to parse the downloaded web page. Here is a callback to crawl the Gap `Sitemap` link:

```
from lxml import etree
from threaded_crawler import threaded_crawler

def scrape_callback(url, html):
    if url.endswith('.xml'):
        # Parse the sitemap XML file
        tree = etree.fromstring(html)
        links = [e[0].text for e in tree]
        return links
    else:
        # Add scraping code here
        pass
```

This callback first checks the downloaded URL extension. If the extension is `.xml`, the downloaded URL is for a `Sitemap` file, and the `lxml etree` module is used to parse the XML and extract the links from it. Otherwise, this is a category URL, although this example does not implement scraping the category. Now we can use this callback with the threaded crawler to crawl `gap.com`:

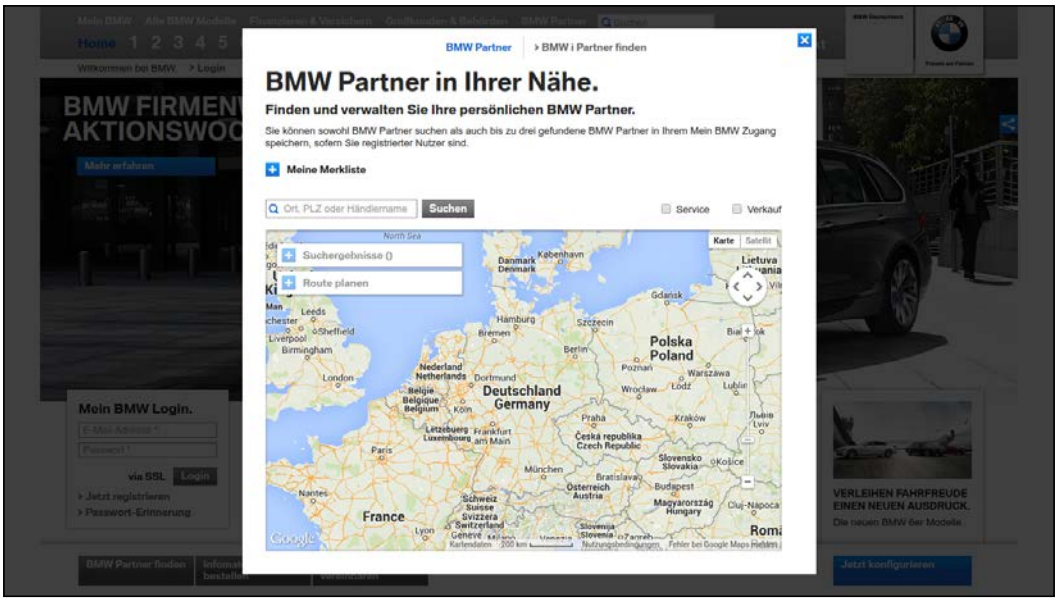
```
>>> from threaded_crawler import threaded_crawler
>>> sitemap = 'http://www.gap.com/products/sitemap_index.xml'
>>> threaded_crawler(sitemap, scrape_callback=scrape_callback)
Downloading: http://www.gap.com/products/sitemap_1.xml
```

```
Downloading: http://www.gap.com/products/sitemap_2.xml
Downloading: http://www.gap.com/products/
    cable-knit-beanie-P987537.jsp
Downloading: http://www.gap.com/products/
    2-in-1-stripe-tee-P987544.jsp
Downloading: http://www.gap.com/products/boyfriend-jeans-2.jsp
...
```

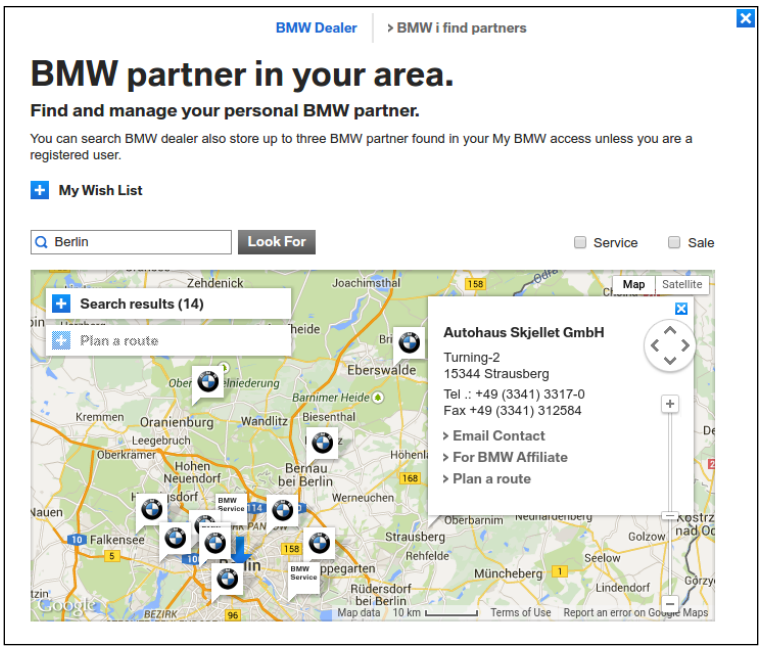
As expected the `Sitemap` files were first downloaded and then the clothing categories.

BMW

The BMW website has a search tool to find local dealerships, available at `https://www.bmw.de/de/home.html?entryType=dlo`:



This tool takes a location, and then displays the points near it on a map, such as this search for `Berlin`:



Using Firebug, we find that the search triggers this AJAX request:

```
https://c2b-services.bmw.com/c2b-localsearch/services/api/v3/
    clients/BMWDIGITAL_DLO/DE/
        pois?country=DE&category=BM&maxResults=99&language=en&
            lat=52.507537768880056&lng=13.425269635701511
```

Here, the `maxResults` parameter is set to `99`. However, we can increase this to download all locations in a single query, a technique covered in *Chapter 1*, *Introduction to Web Scraping*. Here is the result when `maxResults` is increased to `1000`:

```
>>> url = 'https://c2b-services.bmw.com/
    c2b-localsearch/services/api/v3/clients/BMWDIGITAL_DLO/DE/
    pois?country=DE&category=BM&maxResults=%d&language=en&
        lat=52.507537768880056&lng=13.425269635701511'
>>> jsonp = D(url % 1000)
>>> jsonp
'callback({"status":{
...
})'
```

This AJAX request provides the data in **JSONP** format, which stands for **JSON with padding**. The padding is usually a function to call, with the pure JSON data as an argument, in this case the `callback` function call. To parse this data with Python's `json` module, we need to first strip this padding:

```
>>> import json
>>> pure_json = jsonp[jsonp.index('(') + 1 : jsonp.rindex(')')]
>>> dealers = json.loads(pure_json)
>>> dealers.keys()
[u'status', u'count', u'translation', u'data', u'metadata']
>>> dealers['count']
731
```

We now have all the German BMW dealers loaded in a JSON object—currently, 731 of them. Here is the data for the first dealer:

```
>>> dealers['data']['pois'][0]
{u'attributes': {u'businessTypeCodes': [u'NO', u'PR'],
 u'distributionBranches': [u'T', u'F', u'G'],
 u'distributionCode': u'NL',
 u'distributionPartnerId': u'00081',
 u'fax': u'+49 (30) 20099-2110',
 u'homepage': u'http://bmw-partner.bmw.de/
 niederlassung-berlin-weissensee',
 u'mail': u'nl.berlin@bmw.de',
 u'outletId': u'3',
 u'outletTypes': [u'FU'],
 u'phone': u'+49 (30) 20099-0',
 u'requestServices': [u'RFO', u'RID', u'TDA'],
 u'services': []},
 u'category': u'BMW',
 u'city': u'Berlin',
 u'country': u'Germany',
 u'countryCode': u'DE',
 u'dist': 6.65291036632401,
 u'key': u'00081_3',
 u'lat': 52.562568863415,
 u'lng': 13.463589476607,
 u'name': u'BMW AG Niederlassung Berlin Filiale Wei\xdfensee',
 u'postalCode': u'13088',
 u'street': u'Gehringstr. 20'}
```

We can now save the data of interest. Here is a snippet to write the name and latitude and longitude of these dealers to a spreadsheet:

```
with open('bmw.csv', 'w') as fp:
    writer = csv.writer(fp)
    writer.writerow(['Name', 'Latitude', 'Longitude'])
    for dealer in dealers['data']['pois']:
        name = dealer['name'].encode('utf-8')
        lat, lng = dealer['lat'], dealer['lng']
        writer.writerow([name, lat, lng])
```

After running this example, the contents of the `bmw.csv` spreadsheet will look similar to this:

```
Name,Latitude,Longitude
BMW AG Niederlassung Berlin Filiale
Weißensee,52.562568863415,13.463589476607
Autohaus Graubaum GmbH,52.4528925,13.521265
Autohaus Reier GmbH & Co. KG,52.56473,13.32521
...
```

The full source code for scraping this data from BMW is available at `https://bitbucket.org/wswp/code/src/tip/chapter09/bmw.py`.

Translating foreign content

You may have noticed that the first screenshot for BMW was in German, but the second in English. This is because the text for the second was translated using the Google Translate browser extension. This is a useful technique when trying to understand how to navigate a website in a foreign language. When the BMW website is translated, the website still works as usual. Be aware, though, as Google Translate will break some websites, for example, if the content of a select box is translated and a form depends on the original value.

Google Translate is available as the `Google Translate` extension for Chrome, the `Google Translator` addon for Firefox, and can be installed as the `Google Toolbar` for Internet Explorer. Alternatively, `http://translate.google.com` can be used for translations—however, this often breaks functionality because Google is hosting the content.

Summary

This chapter analyzed a variety of prominent websites and demonstrated how the techniques covered in this book can be applied to them. We applied CSS selectors to scrape Google results, tested a browser renderer and an API to scrape Facebook pages, used a `Sitemap` to crawl Gap, and took advantage of an AJAX call to scrape all BMW dealers from a map.

You can now apply the techniques covered in this book to scrape websites that contain data of interest to you. I hope you enjoy this power as much as I have!

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

Q

R

Thank you for buying
Web Scraping with Python

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

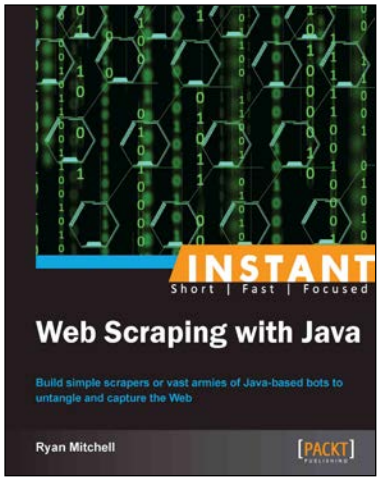
[PACKT] open source
community experience distilled
PUBLISHING

Instant PHP Web Scraping

ISBN: 978-1-78216-476-0 Paperback: 60 pages

Get up and running with the basic techniques of web scraping using PHP

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Build a re-usable scraping class to expand on for future projects.
3. Scrape, parse, and save data from any website with ease.
4. Build a solid foundation for future web scraping topics.



Instant Web Scraping with Java

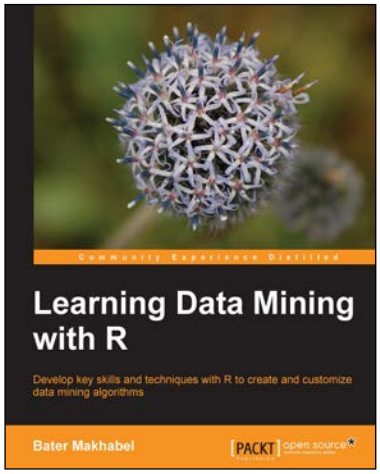
ISBN: 978-1-84969-688-3 Paperback: 72 pages

Build simple scrapers or vast armies of Java-based bots to untangle and capture the Web

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Get your Java environment up and running.
3. Gather clean, formatted web data into your own database.
4. Learn how to work around crawler-resistant websites and legally subvert security measures.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING



Learning Data Mining with R

ISBN: 978-1-78398-210-3 Paperback: 314 pages

Develop key skills and techniques with R to create and customize data mining algorithms

1. Develop a sound strategy for solving predictive modeling problems using the most popular data mining algorithms.
2. Gain understanding of the major methods of predictive modeling.
3. Packed with practical advice and tips to help you get to grips with data mining.

Expert Python Programming

ISBN: 978-1-84719-494-7 Paperback: 372 pages

Best practices for designing, coding, and distributing your Python software

1. Learn Python development best practices from an expert, with detailed coverage of naming and coding conventions.
2. Apply object-oriented principles, design patterns, and advanced syntax tricks.
3. Manage your code with distributed version control.

Please check **www.PacktPub.com** for information on our titles