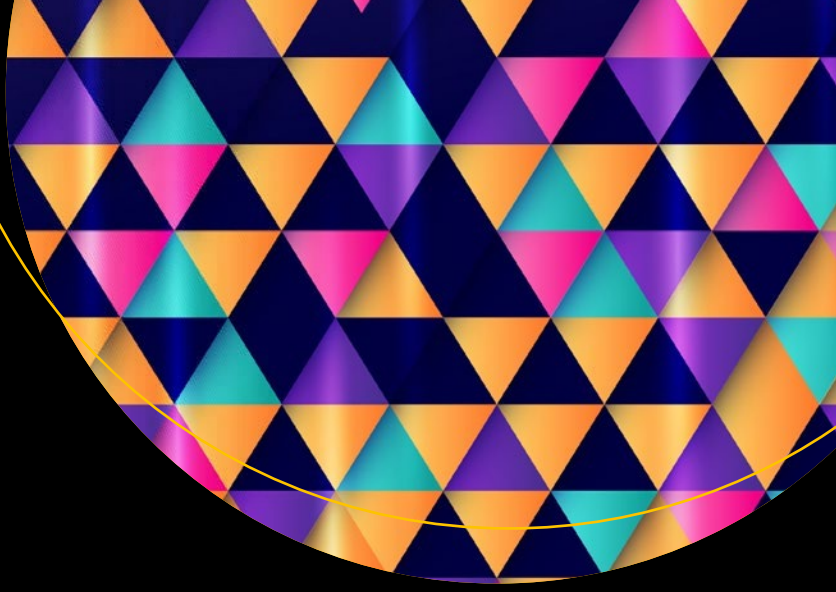

Website Scraping with Python

Using BeautifulSoup and Scrapy

Gábor László Hajba

Apress®

Website Scraping with Python

Using BeautifulSoup and Scrapy

Gábor László Hajba

Apress®

Website Scraping with Python

Gábor László Hajba
Sopron, Hungary

ISBN-13 (pbk): 978-1-4842-3924-7 ISBN-13 (electronic): 978-1-4842-3925-4
https://doi.org/10.1007/978-1-4842-3925-4

Library of Congress Control Number: 2018957273

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Todd Green
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484239247. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To those who are restless, like me,
and always want to learn something new.

Table of Contents

About the Author

Gábor László Hajba is a Senior Consultant at EBCONT enterprise technologies, who specializes in Java, Python, and Crystal. He is responsible for designing and developing customer needs in the enterprise software world. He has also held roles as an Advanced Software Engineer with Zühlke Engineering, and as a freelance developer with Porsche Informatik. He considers himself a workaholic, (hard)core and well-grounded developer, pragmatic minded, and freak of portable apps and functional code.

He currently resides in Sopron, Hungary with his loving wife, Ágnes.

About the Technical Reviewer

Chaim Krause is an expert computer programmer with over thirty years of experience to prove it. He has worked as a lead tech support engineer for ISPs as early as 1995, as a senior developer support engineer with Borland for Delphi, and has worked in Silicon Valley for over a decade in various roles, including technical support engineer and developer support engineer. He is currently a military simulation specialist for the US Army's Command and General Staff College, working on projects such as developing serious games for use in training exercises.

He has also authored several video training courses on Linux topics and has been a technical reviewer for over twenty books, including iOS Code Testing, Android Apps for Absolute Beginners (4ed), and XML Essentials for C# and .NET Development (all Apress). It seems only natural then that he would be an avid gamer and have his own electronics lab and server room in his basement. He currently resides in Leavenworth, Kansas with his loving partner, Ivana, and a menagerie of four-legged companions: their two dogs, Dasher and Minnie, and their three cats, Pudems, Talyn, and Alaska.

Acknowledgments

Many people have contributed to what is good in this book. Remaining errors and problems are the author's alone.

Thanks to Apress for making this book happen. Without them, I'd have never considered approaching a publisher with my book idea.

Thanks to the editors, especially Jill Balzano and James Markham. Their advices made this book much better.

Thanks to Chaim Krause, who pointed out missing technical information that may be obvious to me but not for the readers.

Last but not least, a big thank you to my wife, Ágnes, for enduring the time invested in this book.

I hope this book will be a good resource to get your own website scraping projects started!

Introduction

Welcome to our journey together exploring website scraping solutions using the Python programming language!

As the title already tells you, this book is about website scraping with Python. I distilled my knowledge into this book to give you a useful manual if you want to start data gathering from websites.

Website scraping is (in my opinion) an emerging topic.

I expect you have Python programming knowledge. This means I won't clarify every code block I write or constructs I use. But because of this, you're allowed to differ: every programmer has his/her own unique coding style, and your coding results can be different than mine.

This book is split into six chapters:

1. **Getting Started** is to get you started with this book: you can learn what website scraping is and why it worth writing a book about this topic.
2. **Enter the Requirements** introduces the requirements we will use to implement website scrapers in the follow-up chapters.
3. **Using `Beautiful Soup`** introduces you to Beautiful Soup, an HTML content parser that you can use to write website scraper scripts. We will implement a scraper to gather the requirements of Chapter 2 using Beautiful Soup.

4. **Using Scrapy** introduces you to Scrapy, the (in my opinion) best website scraping toolbox available for the Python programming language. We will use Scrapy to implement a website scraper to gather the requirements of Chapter 2.

5. **Handling JavaScript** shows you options for how you can deal with websites that utilize JavaScript to load data dynamically and through this, give users a better experience. Unfortunately, this makes basic website scraping a torture but there are options that you can rely on.

6. **Website Scraping in the Cloud** moves your scrapers from running on your computer locally to remote computers in the Cloud. I'll show you free and paid providers where you can deploy your spiders and automate the scraping schedules.

You can read this book from cover to cover if you want to learn the different approaches of website scraping with Python. If you're interested only in a specific topic, like `Scrapy` for example, you can jump straight to Chapter 4, although I recommend reading Chapter 2 because it contains the description of the data gathering task we will implement in the vast part of the book.

CHAPTER 1

Getting Started

Instead of installation instructions, which follow later for each library, we will dive right into deep water: this chapter introduces website scraping in general and the requirements we will implement throughout this book.

You may expect a thorough introduction into website scraping, but because you are reading this book I expect you already know what website scraping is and you want to learn how to do it with Python.

Therefore, I'll just give you a glance at the topic and jump right into the depths of creating a script that scrapes websites!

Website Scraping

The need to scrape websites came with the popularity of the Internet, where you share your content and a lot of data. The first widely known scrapers were invented by search engine developers (like Google or AltaVista). These scrapers go through (almost) the whole Internet, scan every web page, extract information from it, and build an index that you can search.

Everyone can create a scraper. Few of us will try to implement such a big application, which could be new competition to Google or Bing. But we can narrow the scope to one or two web pages and extract information in a structured manner–and get the results exported to a database or structured file (JSON, CSV, XML, Excel sheets).

G. L. Hajba, *Website Scraping with Python*, https://doi.org/10.1007/978-1-4842-3925-4_1

Nowadays, *digital transformation* is the new buzzword companies use and want to engage. One component of this transformation is providing data access points to everyone (or at least to other companies interested in that data) through APIs. With those APIs available, you do not need to invest time and other resources to create a website scraper.

Even though providing APIs is something scraper developers won't benefit from, the process is slow, and many companies don't bother creating those access points because they have a website and it is enough to maintain.

Projects for Website Scraping

There are a lot of use cases where you can leverage your knowledge of website scraping. Some might be common sense, while others are extreme cases. In this section you will find some use cases where you can leverage your knowledge.

The main reason to create a scraper is to extract information from a website. This information can be a list of products sold by a company, nutrition details of groceries, or NFL results from the last 15 years. Most of these projects are the groundwork for further data analysis: gathering all this data manually is a long and error-prone process.

Sometimes you encounter projects where you need to extract data from one website to load it into another–a migration. I recently had a project where my customer moved his website to WordPress and the old blog engine's export functionality wasn't meant to import it into WordPress. I created a scraper that extracted all the posts (around 35,000) with their images, did some formatting on the contents to use WordPress short codes, and then imported all those posts into the new website.

A weird project could be to download the whole Internet! Theoretically it is not impossible: you start at a website, download it, extract and follow all the links on this page, and download the new sites too. If the websites

you scrape all have links to each other, you can browse (and download) the whole Internet. I don't suggest you start this project because you won't have enough disk space to contain the entire Internet, but the idea is interesting. Let me know how far you reached if you implement a scraper like this.

Websites Are the Bottleneck

One of the most difficult parts of gathering data through websites is that websites differ. I mean not only the data but the layout too. It is hard to create a good-fit-for-all scraper because every website has a different layout, uses different (or no) HTML IDs to identify fields, and so on.

And if this is not enough, many websites change their layout frequently. If this happens, your scraper is not working as it did previously. In these cases, the only option is to revisit your code and adapt it to the changes of the target website.

Unfortunately, you won't learn secret tricks that will help you create a scraper that always works–if you want to write specialized data extractors. I will show some examples in this book that will always work if the HTML standard is in use.

Tools in This Book

In this book you will learn the basic tools you can use in Python to do your website scraping. You will soon realize how hard it is to create every single piece of a scraper from scratch.

But Python has a great community, and a lot of projects are available to help you focus on the important part of your scraper: data extraction. I will introduce you to tools like the `requests` library, `Beautiful Soup`, and `Scrapy`.

The `requests` library is a lightweight wrapper over the tedious task of handling HTTP, and it emerged as the recommended way:

> *The Requests package is recommended for a higher level HTTP client interface.*
>
> — Python 3 documentation

`Beautiful Soup` is a content parser. It is not a tool for website scraping because it doesn't navigate pages automatically and it is hard to scale. But it aids in parsing content, and gives you options to extract the required information from XML and HTML structures in a friendly manner.

`Scrapy` is a website scraping framework/library. It is much more powerful than `Beautiful Soup`, and it can be scaled. Therefore, you can create more complex scrapers easier with `Scrapy`. But on the other side, you have more options to configure. Fine-tuning `Scrapy` can be a problem, and you can mess up a lot if you do something wrong. But with great power comes great responsibility: you must use `Scrapy` with care.

Even though `Scrapy` is **the Python library** created for website scraping, sometimes I just prefer a combination of `requests` and `Beautiful Soup` because it is lightweight, and I can write my scraper in a short period–and I do not need scaling or parallel execution.

Preparation

When starting a website scraper, even if it is a small script, you must prepare yourself for the task. There are some legal and technical considerations for you right at the beginning.

In this section I will give you a short list of what you should do to be prepared for a website scraping job or task:

1. Do the website's owners allow scraping? To find out, read the *Terms & Conditions* and the *Privacy Policy* of the website.

2. Can you scrape the parts you are interested in? See the `robots.txt` file for more information and use a tool that can handle this information.
3. What technology does the website use? There are free tools available that can help you with this task, but you can look at the website's HTML code to find out.
4. What tools should I use? Depending on your task and the website's structure, there are different paths you can choose from.

Now let's see a detailed description for each item mentioned.

Terms and Robots

Scraping currently has barely any limitations; there are no laws defining what can be scraped and what cannot.

However, there are guidelines that define what you should respect. There is no enforcing; you can completely ignore these recommendations, but you shouldn't.

Before you start any scraping task, look at the *Terms & Conditions* and *Privacy Policy* of the website you want to gather data from. If there is no limitation on scraping, then you should look at the `robots.txt` file for the given website(s).

When reading the terms and conditions of a website, you can search for following keywords to find restrictions:

- scraper/scraping
- crawler/crawling
- bot
- spider
- program

Most of the time these keywords can be found, and this makes your search easier. If you have no luck, you need to read through the whole legal content and it is not as easy–at least I think legal stuff is always dry to read.

In the European Union there's a data protection right that has been live for some years but strictly enforced from 2018: GDPR. Keep the private data of private persons out of your scraping—you can be held liable if some of it slips out into public because of your scraper.

---

robots.txt

Most websites provide a file called `robots.txt`, which is used to tell web crawlers what they can scrape and what they should not touch. Naturally, it is up to the developer to respect these recommendations, but I advise you to **always obey** the contents of the `robots.txt` file.

Let's see one example of such a file:

```
User-agent: *
Disallow: /covers/
Disallow: /api/
Disallow: /*checkval
Disallow: /*wicket:interface
Disallow: ?print_view=true
Disallow: /*/search
Disallow: /*/product-search
Allow: /*/product-search/discipline
Disallow: /*/product-search/discipline?*facet-subj=
Disallow: /*/product-search/discipline?*facet-pdate=
Disallow: /*/product-search/discipline?*facet-type=category
```

The preceding code block is from `www.apress.com/robots.txt`. As you can see, most content tells what is disallowed. For example, scrapers shouldn't scrape `www.apress.com/covers/`.

Besides the Allow and Disallow entries, the User-agent can be interesting. Every scraper should have an identification, which is provided through the user agent parameter. Bigger bots, created by Google and Bing, have their unique identifier. And because they are scrapers that add your pages to the search results, you can define *excludes* for these bots to leave you alone. Later in this chapter, you will create a script which will examine and follow the guidelines of the `robots.txt` file with a custom user agent.

There can be other entries in a `robots.txt file`, but they are not standard. To find out more about those entries, visit https://en.wikipedia.org/wiki/Robots_exclusion_standard.

Technology of the Website

Another useful preparation step is to look at the technologies the targeted website uses.

There is a Python library called `builtwith`, which aims to detect the technologies a website utilizes. The problem with this library is that the last version *1.3.2* was released in 2015, and it is not compatible with Python 3. Therefore, you cannot use it as you do with libraries available from the PyPI.[1]

However, in May 2017, Python 3 support has been added to the sources, but the new version was not released (yet, I'm writing this in November 2017). This doesn't mean we cannot use the tool; we must manually install it.

First, download the sources from https://bitbucket.org/richardpenman/builtwith/downloads/. If you prefer, you can clone the repository with Mercurial to stay up to date if new changes occur.

After downloading the sources, navigate to the folder where you downloaded the sources and execute the following command:

```
pip install .
```

[1]PyPI – the Python Package Index

The command installs `builtwith` to your Python environment and you can use it.

Now if you open a Python CLI, you can look at your target site to see what technologies it uses.

```
>>> from builtwith import builtwith
>>> builtwith('http://www.apress.com')
{'javascript-frameworks': ['AngularJS', 'jQuery'],
'font-scripts': ['Font Awesome'], 'tag-managers':
['Google Tag Manager'], 'analytics': ['Optimizely']}
```

The preceding code block shows which technologies Apress uses for its website. You can learn from AngularJS that if you plan to write a scraper, you should be prepared to handle dynamic content that is rendered with JavaScript.

`builtwith` is not a magic tool, it is a website scraper that downloads the given URL; parses its contents; and based on its knowledge base, it tells you which technologies the website uses. This tool uses basic Python features, which means sometimes you cannot get information in the website you are interested in, but most of the time you get enough information.

Using Chrome Developer Tools

To walk through the website and identify the fields of the requirements, we will use Google Chrome's built-in *DevTools*. If you do not know what this tool can do for you, here is a quick introduction.

> *The Chrome Developer Tools (DevTools for short), are a set of web authoring and debugging tools built into Google Chrome. The DevTools provide web developers deep access into the internals of the browser and their web application. Use the DevTools to efficiently track down layout issues, set JavaScript breakpoints, and get insights for code optimization.*

As you can see, DevTools give you tools to see inside the workings of the browser. We don't need anything special; we will use DevTools to see where the information resides.

In this section I will guide us with screenshots through the steps I usually do when I start (or just evaluate) a scraping project.

Set-up

First, you must prepare to get the information. Even though we know which website to scrape and what kind of data to extract, we need some preparation.

Basic website scrapers are simple tools that download the contents of the website into memory and then do extraction on this data. This means they are not capable of running dynamic content just like JavaScript, and therefore we have to make our browser similar to a simple scraper by disabling JavaScript rendering.

First, right-click with your mouse on the web page and from the menu select "Inspect," as shown in Figure 1-1.

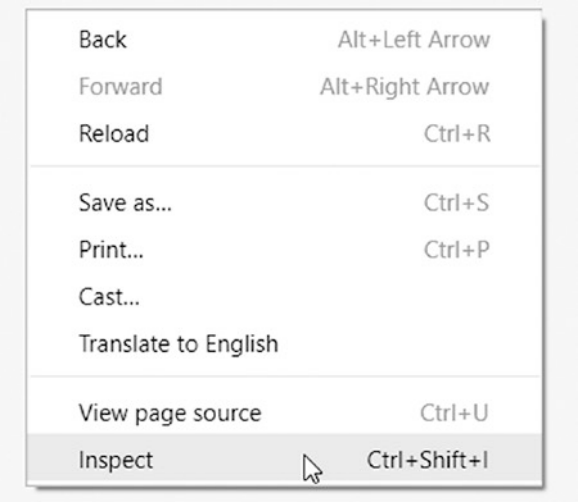

Figure 1-1. *Starting Chrome's DevTools*