Web Scraping with Python for Beginners - In Progress

Harness the full power of the web.

Mitchell Edwards

Web Scraping with Python for Beginners - In Progress

Harness the full power of the web.

Mitchell Edwards

This book is for sale at http://leanpub.com/beginner-python-web-scraping

This version was published on 2022-05-24

Contents

Python Web Scraping for Beginners

About

If you're reading this, it's because you have decided to embark upon a journey with me to write a whole series of books on Python web scraping in public. If you have no idea what I'm talking about, welcome! My name is Mitch Edwards, or @valhalla_dev if you follow me on Twitter[1], and I've been a developer for the better part of the last 10 years. In that time, I've developed countless projects for myself and a multitude of employers, from simple hacky scripts to large-scale enterprise data solutions. I spend my "free time" writing scrapers to research right wing extremism, economics, social media and more, developing malware to learn more about reverse engineering and developing startups. That's a lot to take in, so if you are interested in any of those subjects, here is where you can find my research.

Twitter[2] - where I post updates and links to all of my research and development, including the development of this book

My newsletter[3] - I have a weekly newsletter where I talk about software development, countering right wing extremism and, now, writing this book! This is a good way to get updates about what I'm working on and the different things I'm focusing on in a given week.

Valhalla Research[4] - where I post research regarding countering right wing extremism, software development, political philosophy and cyber security.

YouTube[5] - where I post video content related to cyber security and software development topics

---

Now, let's get talking about this book!

I previously developed a course on developing web scrapers and spiders using Python. I taught the course to over 250 students on Udemy[6] and released the course for free on YouTube[7] after a couple of months. Now I want to expand on a lot of the lessons I taught in that course in a series of books. This book is the first of that series.

Building a book in public

This book will be written entirely in public. I'm releasing this book as-is in chunks, which is why I chose Leanpub as a publisher and CMS. In the beginning, the book in its unfinished form will be

[1]https://twitter.com/valhalla_dev
[2]https://twitter.com/valhalla_dev
[3]https://www.getrevue.co/profile/valhalla_dev
[4]https://valhallaresearch.net
[5]https://youtube.com/viking_sec
[6]https://www.udemy.com/course/scrape-the-planet/learn/
[7]https://www.youtube.com/watch?v=8Xu-H2GM2AM&list=PL1jK3K11NINiOn4DdIDVdyQpcU3kaNxl0

free for people to follow along, but eventually I will put it behind a paywall once there is something worth paying for. People who have been following along from the beginning will continue getting it for free, so there's a lot of benefit in being an early fan!

I will be tweeting and blogging about the actual act of developing the book, and I'll keep Twitter in particular up to date about updates to the book. I'll also be uploading all of the code to GitHub publicly and for free, so you'll also benefit from open source software.

For now, though, that's all! Thank you all for the current and future support, and share this book with friends far and wide!

Introduction

What is web scraping
How to read this book
Why should you learn web scraping
## Theoretical outline

Basics of the web
What is web scraping (more in depth)
Why scrape the web?
Spiders versus scrapers
Ethics and legality of web scraping
Beginner web scraper design
## Setting up our environment

Virtual machines
Installing Python
Installing the required libraries
Installing MongoDB
## Quick Python Basics

Variables, math and concatenation
Loops
Functions
Libraries
## The Requests library

Our first request: making a GET request to Google
Understanding GET and POST requests
Introduction to API scraping
Making a simple web API
The Binance API
Using the Binance API to get stock price data
…
## The BeautifulSoup library

Introduction

A bit about me…

Hello! My name is Mitch Edwards. I'm a threat intelligence and cyber security researcher by day and a software developer and right wing extremism researcher by night. I'm also a YouTuber[8], blogger[9], newsletter author[10], cigar afficionado, powerlifter, dad and husband. I've been developing software for over 10 years now, with much of that being spent writing web scrapers and spiders.

I've written web scrapers to detect censorship on Chinese social media, to study darkweb markets where malware is bought and sold, to track and study right wing extremism online and more. Web scrapers have been a huge part of my career, and they continue to be one of my favorite problems to solve. This book is the start of a series of books on web scraping that will summarize and expand upon the last several years of experience in developing web scrapers both big and small. It also draws upon my experience teaching web scraper development through a course published in 2021 on Udemy that reached over 250 students, as well as coursework I have taught in person in my local community.

Enough about me…

What the hell are web scrapers??

That's the million-dollar question, isn't it?

Web scrapers are programs that collect information from the web in a more or less automated fashion. We will go further into the weeds on how web scrapers work fairly soon, but essentially web scrapers can do exactly what humans do when we surf the web. We read information, upload information, interact with other users, add content, delete content and make decisions. Web scrapers can do the exact same things that we do, but they can do it at the speed of computing, limited only by their design and implementation. Web scrapers take action based upon programmable logic, just like any other Python script or console program, but they are specifically built to automate web-based activity like collecting information from web-based resources or posting information to the web.

In the context of this book, we're specifically going to be talking about web scrapers built in the Python programming language. We're going to talk about all kinds of web scrapers, to include web

[8]https://youtube.com/viking_sec
[9]https://valhallaresearch.net
[10]https://www.getrevue.co/profile/valhalla_dev

scrapers that post information to the web, but there is a (hazy) line between bots, or programs that are specifically built to post information in an automated fashion like bots that automatically like or retweet tweets, and scrapers, whose typical function is more focused on collecting information, such as a scraper that pulls down stock price information. That separation is, as I said, hazy, and there isn't a perfect way to define what is a bot and what is a scraper, as they are often functionally the same thing. Instead of arguing definitions, we will operate on the one below:

"A scraper is a purposeful automated program whose primary function is to collect information from a web-based resource."

Now, what does the word "purposeful" mean in the above definition?

Well, the web scrapers we talk about in this book don't just pull down information for no reason. You could certainly develop purposeless scrapers, as I'm sure I have at some point in the last couple years, but this book will focus on scrapers that have a specific type of information to gather for a specific and existing purpose. This is not meant to be an insult to r/datahoarders[11] and similar communities, but we're not going to focus on the collection of data for its own sake.

A simple, one-line definition of web scraping isn't quite enough, though… it doesn't do the complexity of the problems of web scraping justice at all. Here is where we talk about what will be your most favorite or least favorite part of the book:

Theory.

I'll be honest with you, I'm a bit of a nerd about theory. I really love it. From programming frameworks and philosophies to theoretical and practical system design, I eat theory up. It's not for everyone, though, and some people let me know (fairly brutally, in at least one case) that the theory in my online course on web scraping was boring. The theory-based sections of that course were by far the most skipped portions according to Udemy's metric.

So, in the interest of free-will, I'm giving you the option of skipping the theory sections. I'm even making it easier for you by putting the bulk of the theory in the next chapter. You purchased the book, or, if you got in early, maybe you got it for free, but the express meaning of that trade is that you get to do with this book what you will. Skim it, read it back-to-front, copy/paste the code out of it and move on, I don't really care.

However, I'm not writing this book just for the sake of reading my own words. I have Twitter for endlessly pontificating on the subject of the day, I don't need to write thousands of words and hundreds of lines of code to do that. I'm writing this book because I want to *educate* people. I want to move the science forward. I want debate and I want to be proven wrong.

The only way for you to truly grasp the subject matter we will be covering in this book and the next ones in the series is to truly understand the theory of web scraper development. You could skip the theory and go right to the project sections, and you'll probably still learn a lot. One of the hardest lessons to learn as someone in software development (and cyber security, for that matter) is that you can go about learning in such a way that you have paper intelligence. Paper intelligence is all about knowing the facts: you can write each line of code from memory. Where you are lacking,

[11]https://www.reddit.com/r/DataHoarder/

though, is in not having any more depth than that: you don't know what the lines *mean* or why they are written in the way that they are. You can't innovate, you can't experiment, you can't push the industry or science forward.

I don't want to give you paper intelligence. I want you to be the one that pushes the industry and science forward. I want you to bend and break things, I want you to build and innovate things. So, I'm pleading with you, suspend your need for entertainment and easy code samples for just one chapter and bear with the theory. I promise you that you will look back on that decision fondly once you are a superhero dev.

Theory of Web Scraper Development

Woah… you actually started this chapter! You're one of the few who decided to tighten their belt and lean into the theory section. I promise you won't be disappointed. Your projects will be successful largely because you understand the underlying technology you're developing. So congratulations!

Let's break this chapter into chunks. First, we will have the theory of how the web works. Perfectly fine if you skip this bit, it's just covering how HTTP requests work from a fairly simple, theoretical level.

After that, we will discuss how scrapers work from a very basic level. This is where important stuff comes in: if you don't understand the foundational stuff, the next several chapters are going to be hard.

After we cover the basics of scrapers, we will cover what web spiders are from that same foundational high level. This is going to be a pretty cool section, especially if you want to really explore what web spiders are and how to take them to the next level.

After we touch on spiders, we will talk about API's: how to design simple ones, how to scrape enterprise API's, and what the difference is between page scrapers and API scrapers.

Then we will cover the dreaded **ethics** topic… People tend to skip this bit, and I'll say the same thing here that I did before the theory section began: you have the freedom to skip whatever you want, but I really hope you'll stick around for this section. It's incredibly important.

Finally, just before we start the practical stuff, we will talk effective design paradigms at a very high level. This will be a fairly small section because I'll save a lot of design theory for when we start putting code to screen, but there are a couple of things I want to cover beforehand.

Notably, we're going to touch on some stuff that isn't going to be covered in this book. There are some advanced topics I have to touch on but will fully cover in future books that are for a more advanced audience.

Theory of the Web

The web is made up of servers and clients. The end.

… Just kidding. That's a bit too basic to really understand how web scrapers interact with the internet. To understand that, you really have to understand how *people* interact with the web. We use our mobile devices, iPad tablets, desktops and laptops to connect to web servers that host web sites. Our device is the client, and the web server is… well… the server.

A server is really just a device that serves up data to a client, while a client is something that browses for or otherwise fetches and consumes that content. When we go to a web page, we're not technically connecting to a web **site**, we're really connecting to a web **server** and *requesting* some data from it. We use a suite of protocols in order to achieve this connection, including, notably, HTTP/HTTPS and DNS. I'm not going to explain the protocols in this book because that's a bit more out of scope than is really necessary, but you can start your research at the hyperlinks below:

An Overview of HTTP (Mozilla)[12]
What is the DNS Protocol (NS1)[13]

When developing scrapers and spiders, you generally don't have to deal with DNS much, but you have to have at least working knowledge of the HTTP protocol, so the bulk of this section will focus on HTTP requests and responses. I'm also not going to be specifying HTTP versus it's SSL-secured counterpart HTTPS, because you generally don't have to deal with specifics related to SSL when you're developing web scrapers, so there's not much of general import to say about HTTPS that isn't also true of developing with HTTP.

Basics of HTTP

HTTP is the HyperText Transfer Protocol and its primary job is the facilitation of the exchange of data between a web client (usually a web browser used by a human) and a server (usually a web server serving up HTML). Very notably, HTTP doesn't just facilitate the transfer of HTML data, as we will see in later chapters, but in the context of a web browser, pretty much all of the data a web browser is interested in and deals with is HTML data.

HTML (HyperText Markup Language) is the language that browsers can parse, or read, to render web pages. It's an incredibly robust language, but it's also messy as hell and incredibly difficult to read in its raw form. That's why, as a normal human, you won't have to actively deal with the HTML much: you type a website into the URL bar or click a link, some magic happens and then you end up with a bunch of HTML being rendered as a beautiful web site.

What we're interested in is the clicking of a link or typing the website into the URL bar as well as the magic happening before the page is rendered. Whenever you type an address into the address bar, or click a link, which, in this context, is basically the same thing, you're making an **HTTP request** to the web server hosting the content you're looking for. An HTTP request is simply a formatted method of informing the web server what you want to do. There are many types of HTTP requests, but the two we're going to focus on primarily are **POST** requests and **GET** requests.

The primary difference between the two requests is how they're used. A GET request is typically used to just fetch certain information from a web server. Typically, when you type a page into the address bar or click a link, you're almost always making a GET request (to start things off) that tells the web server what page and information you want to access. A POST request is typically used to send information, like you would a letter in the post. You typically attach this information to

[12]https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
[13]https://ns1.com/resources/dns-protocol

the POST request in the POST body, usually in the form of JSON data. This could include all kinds of information such as usernames and passwords, timestamps, tweet bodies, etc. GET requests can send information too, but it's typically in the URL request parameters, which are notably recorded in most HTTP server logs in an unencrypted form, meaning your confidential information sent as part of the URL will be stored in plain text in the URL of the web server, creating a fairly serious security vulnerability. POST request bodies aren't recorded in the logs by default in most cases, so they are notably more secure. In general you should always avoid sending private information like passwords via GET request, but it's fine using POST requests provided you're using some sort of encryption like SSL.

Below is an illustration of the difference:

<ILLUSTRATION/>

In general, you can usually depend on the definition mentioned above: GET requests are for requesting specific web pages in situations where you don't have to send much information along to gain access to that information, while POST requests are for receiving data that requires some input from the client to access an asset on the server.

### How *You* Browse the Web

So, now we understand how HTTP requests work and how most web traffic depends on GET and POST requests. Let's build on that a bit by talking about how **you** browse the web. We use web browsers, which take the HTML from a web server and renders it to the screen in nice pretty colors. Under the hood, though, it's much more complex…

You see, most web pages actually consist of HTML, JavaScript and CSS from *several* different web requests. We're talking sometimes dozens of them, especially for social media sites and other sites serving highly dynamic content. You can actually see this if you open up your developer tools in Firefox or Google Chrome on Twitter and go to the network tab.

<ILLUSTRATION/>

On my load, my browser counted over 120 different requests, just for one page! These requests were used to load individual tweets and images, advertisements, stylesheets and more. All I did was make the one request to twitter.com, though, so why were there so many other requests?

Well, HTML and JavaScript have certain tags and functionality that will cause a browser to dynamically load more content. The <link> tag causes a page to fetch things like scripts and stylesheets from other web pages, either ones on the same web server (local assets) or assets from other servers/sites (remote assets) and in this era of front-end web development, most big applications are built using JavaScript frameworks that have tons of functions that allow for dynamic content rendering. In React, for example, I can code a function that will dynamically fetch content from a server on a timer, so every 10 seconds the front-end will send another request to the server to load new assets. Every time you scroll to the "bottom" of an endless-scroll site like Twitter, you're going to create dozens or hundreds more requests to Twitter's servers to fetch new tweets.

## Basics of Web Scrapers

"Cool, Mitch, now that you're 3,000+ words into the book, can we get to scrapers now??"

Sure, overly sarcastic reader, but only if you drop the attitude first!

Web scrapers are essentially automated web browsers that work at 100x speed to fetch web pages and other assets from web servers using code. They make web requests, receive the responses, parse whatever data is in the response and do something with it. There are several big differences though.

First, web scrapers typically don't render dynamic content, they just grab static content. Now, for those of you who have experience with developing or researching web scraping, yes, you can build scrapers that will pull dynamic web pages. I've typically done it with Selenium, and I'll be covering that in one of the more advanced books. For now, though, let's just talk about your typical, run-of-the-mill web scraper. They will make an HTTP request, receive the response, and only render the static web page that's returned. If the target page is built with a JavaScript framework like React that handles dynamic content via state, dynamic asset fetching, etc., that's not going to be reflected in the result. If the page has <link> or <img> tags, they're not going to reach out to the secondary pages to pull that content automatically. We will talk about how that can be a **good** thing in a second.

Second, the mode of consumption is obviously very different. A web browser's entire purpose is to take a web page and render it into something viewable. Since we're using code instead of a browser, you're working with whatever the web page returns to you in its raw form, whether that's HTML code or JSON. This is where I divide web scrapers into two different categories: **API Scrapers** and **Page Scrapers**. API Scrapers are web scrapers that scrape API's (duh) and they typically deal mostly with JSON data, while Page Scrapers are web scrapers that scrape raw HTML pages and therefore have to parse HTML code. We're going to go into API's, what they are and how to build them later, and we will spend *most* of our time covering Page Scrapers in this book, so if you're not familiar with the terminology or the practical application, well… you're only a couple sections into the book, so you're not supposed to.

So, web scraping is just web browsing without the browser. You have to deal with some fairly complex code, lots of headaches parsing HTML code and a lot of fun edge cases that break or bork your scrapers…

So what's the point?

Why scrape the web, anyways?

Why build web scrapers? What are their purpose? If it's just automated web browsing, why not just browse the web?

Well, I alluded to it before, but I'll drive the point home here: web scraping is *wicked fast* when done correctly. You can scrape, parse and store gigabytes of data in just a couple of minutes with a web scraper built to handle that much data. I've cloned all of the data from fairly large social media sites with hundreds of thousands of posts in a day or so. When I said web scrapers are just automated web browsers, I was majorly oversimplifying.

The reason I love developing web scrapers is that it allows us to benefit from a human-machine symbiosis, a relationship where we get to combine what computers do best with what humans do best. Machines are very good at following very strict, codified routines, very, very fast, at parsing and storing complex data in an efficient manner and at "remembering" that data permanently through hard disk storage and databases. Humans are good at giving computers instructions (well, some humans are, I'm still fairly awful at coding) and coming up with complex, innovative analyses based on intuitive data analysis. Scrapers allow the computer to do the heavy lifting in data collection and free up our time to actually analyze the data that is collected and stored by the scraper, giving us the job of intuition and analysis and the computer the job of repeatable, routine tasks done at lightning fast speeds with perfect "memorization."

## Basics of Web Spiders

We're going to spend precious little time on web spiders, despite them being one of my favorite subjects in all of web scraping. I'll briefly go over why after I explain what web spiders are, but just know that web spiders are a wicked cool, and often necessary tool in the arsenal of any web scraper developer.

What is a web spider? Is it as creepy as the real thing?

It's not as creepy, but it can be just as powerful as the eight-legged devils that haunt your nightmares. Web spiders are web scrapers that have a specific focus on automated content discovery. To be clear, web spiders are a sub-species of web scrapers. They are just as much scrapers of the web as "normal" web scrapers are. Web scrapers, in the context we have discussed thus far, have a specific page or web-based asset in mind. I want the data on example.com hosted on the /foo/bar page, so I scrape that page, parse out the data I want and that's it. It's fairly linear, and it's often fairly simple. I could write a web scraper to pull, say, the temperature in my town on any given day in one or two lines of Python if I really wanted to.

Web spiders, on the other hand, are often a bit more complex. They are used in contexts where you don't know exactly what you want, but you know the general logic and layout of the web pages you need to access and where the assets are located. The classic example is a web forum: I want all posts and comments on a given web forum. The data I want in this case isn't located on one, or even several, individual pages, it's located on several pages. Web spiders also thrive in contexts where the content you want to scrape is dynamic, so for a web forum, a good web spider can be run daily or many times per day to grab all of the new posts and comments.

The input for a “normal” web scraper might be one or two pages that host the data I want. A web spider will hit the front page, or a user page that contains a ton of posts made by that user, and will scrape all of the data off of that page before moving on to the next one.

The key thing to understand between “normal” scrapers and spiders is that building spiders requires you to build models representing how an entire site is laid out, while “normal” scrapers really just require the understanding of one or two web pages.

Again, I’m doing web spiders dirty by oversimplifying them into a couple of paragraphs of high-level explanation, but this is why: to understand web spiders, you have to understand scrapers, and as of yet, we haven’t even built a simple scraper! Spiders also require a fair bit of effort in the system design and development. Because of this, I’m going to reserve discussions of web spiders for later, more advanced books, and instead will focus my efforts on giving you an understanding of how “normal” web scrapers work.

## API’s: How They’re Built and How to Scrape Them

API’s are a scraper’s best friend. We’re going to be spending the majority of our time on HTML web page scrapers instead of API scrapers for a variety of reasons, but one of those reasons is that scraping API’s is just so easy. I would recommend you become familiar with how API’s work, though, mainly because in a professional setting, most of what you’re going to be working on is scraping API’s instead of HTML web pages.

But let’s back up a second. What is an API?

API stands for Application Program Interface. Context clues will tell you a lot about what this means: an API is an **interface** between an **application** and another **program**. If you have a given web application, let’s say Twitter, the API is the thing that the program that you write actually talks to. It is the intermediary between the code that you write and the data that Twitter has. It handles all of the discussion, data fetching and formatting between the code that you wrote and the code and data belonging to Twitter.

How do API’s work?

API’s work through software functions called endpoints. An endpoint is a particular location on a server that corresponds to certain functionality within the API. An API call is when a user or consumer of the API accesses the API endpoint.

Let’s say we have an API for a social media website. Very generally you might have an API endpoint for creating a post, accessing a post by its post ID and deleting a post. Again, this is vastly oversimplifying what an actual API would look like for a company like Twitter, but it’s enough to illustrate the basics of what an API is and how to interact with it.

Let’s say we have an API endpoint located at **/api/create_post**. This API endpoint only accepts POST requests, so visiting it with a normal web browser isn’t going to work.If I were to visit the web page for the create_post API, it would send a GET request by default. Since we’re sending a post, it makes more sense to use a post request anyways: we're going to be sending the post text along with the API key and other metadata, so we will need to do that in the POST body.

So, we construct a POST body, in the form of a JSON object, and send it along to the endpoint, which would process the POST body, check the API key and actually create the post itself. We will get into the actual "how" of this very soon, it's just important that we understand how the endpoint works.

You might also have an **/api/get_post** endpoint to actually read a Tweet. Now, the way this often works is using a GET request and appending the ID of the post to the API call. So if the post ID is "01457936" I would send a GET request to /api/get_post/01457936" and the web server would fetch the relevant data, create a JSON response object and send it on.

That's the very basics of how an API works. We will talk a little about actual implementation later, so don't worry if you're still feeling lost.

## Ethics of Web Scraping

## Effective Web Scraper Design
