# Modern Web Scraping With Python and Selenium

Web scraping has been around since the early days of the World Wide Web, but scraping modern sites that heavily rely on new technologies is anything but straightforward. In this article, Toptal Software Developer Neal Barnett demonstrates how you can use Python and Selenium to scrape sites that employ a lot of JavaScript, iframes, and certificates.

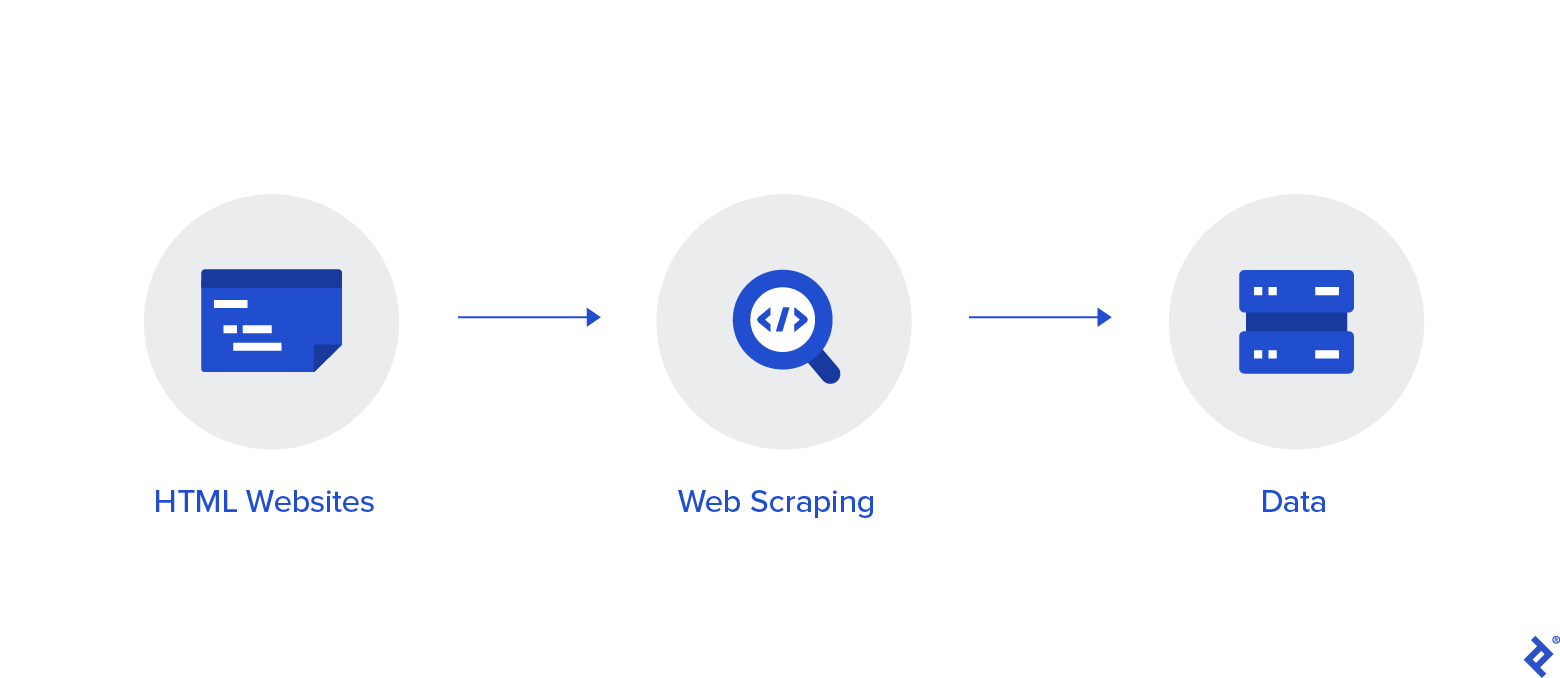
Neal is a senior consultant and database expert who brings a wealth of knowledge and more than two decades of experience to the table.

Web scraping has been used to extract data from websites almost from the time the World Wide Web was born. In the early days, scraping was mainly done on static pages – those with known elements, tags, and data.

More recently, however, advanced technologies in web development have made the task a bit more difficult. In this article, we’ll explore how we might go about scraping data in the case that new technology and other factors prevent standard scraping.

## Traditional Data Scraping

As most websites produce pages meant for human readability rather than automated reading, web scraping mainly consisted of programmatically digesting a web page’s mark-up data (think right-click, View Source), then detecting static patterns in that data that would allow the program to “read” various pieces of information and save it to a file or a database.



If report data were to be found, often, the data would be accessible by passing either form variables or parameters with the URL. For example:

https://www.myreportdata.com?month=12&year=2004&clientid=24823

Python has become one of the most popular web scraping languages due in part to the various web libraries that have been created for it. One popular library, [Beautiful Soup](https://www.crummy.com/software/BeautifulSoup/), is designed to pull data out of HTML and XML files by allowing searching, navigating, and modifying tags (i.e., the parse tree).

## Browser-based Scraping

Recently, I had a scraping project that seemed pretty straightforward and I was fully prepared to use traditional scraping to handle it. But as I got further into it, I found obstacles that could not be overcome with traditional methods.

Three main issues prevented me from my standard scraping methods:

1. **Certificate.** There was a certificate required to be installed to access the portion of the website where the data was. When accessing the initial page, a prompt appeared asking me to select the proper certificate of those installed on my computer, and click OK.
2. **Iframes.** The site used iframes, which messed up my normal scraping. Yes, I could try to find all iframe URLs, then build a sitemap, but that seemed like it could get unwieldy.
3. **JavaScript.** The data was accessed after filling in a form with parameters (e.g., customer ID, date range, etc.). Normally, I would bypass the form and simply pass the form variables (via URL or as hidden form variables) to the result page and see the results. But in this case, the form contained JavaScript, which didn’t allow me to access the form variables in a normal fashion.

So, I decided to abandon my traditional methods and look at a possible tool for browser-based scraping. This would work differently than normal – instead of going directly to a page, downloading the parse tree, and pulling out data elements, I would instead “act like a human” and use a browser to get to the page I needed, then scrape the data - thus, bypassing the need to deal with the barriers mentioned.

## Selenium

In general, [Selenium](https://www.selenium.dev) is well-known as an open-source testing framework for web applications – enabling [QA specialists](https://www.toptal.com/qa) to perform automated tests, execute playbacks, and implement remote control functionality (allowing many browser instances for load testing and multiple browser types). In my case, this seemed like it could be useful.

My go-to language for web scraping is Python, as it has well-integrated libraries that can generally handle all of the functionality required. And sure enough, a [Selenium library](https://selenium-python.readthedocs.io/) exists for Python. This would allow me to instantiate a “browser” – Chrome, Firefox, IE, etc. – then pretend I was using the browser myself to gain access to the data I was looking for. And if I didn’t want the browser to actually appear, I could create the browser in “headless” mode, making it invisible to any user.

## Project Setup

To start experimenting, I needed to set up my project and get everything I needed. I used a Windows 10 machine and made sure I had a relatively updated Python version (it was v. 3.7.3). I created a blank Python script, then loaded the libraries I thought might be required, using PIP (package installer for Python) if I didn’t already have the library loaded. These are the main libraries I started with:

1. [Requests](https://realpython.com/python-requests/) (for making HTTP requests)
2. [URLLib3](https://docs.python.org/3/library/urllib.request.html#module-urllib.request) (URL handling)
3. [Beautiful Soup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/) (in case Selenium couldn’t handle everything)
4. [Selenium](https://selenium-python.readthedocs.io/) (for browser-based navigation)

I also added some calling parameters to the script (using the argparse library) so that I could play around with various datasets, calling the script from the command line with different options. Those included Customer ID, from- month/year, and to-month/year.

## Problem 1 – The Certificate

The first choice I needed to make was which browser I was going to tell Selenium to use. As I generally use Chrome, and it’s built on the open-source Chromium project (also used by Edge, Opera, and Amazon Silk browsers), I figured I would try that first.

I was able to start up Chrome in the script by adding the library components I needed, then issuing a couple of simple commands:

# Load selenium components

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait, Select

from selenium.webdriver.support import expected\_conditions as EC

from selenium.common.exceptions import TimeoutException

# Establish chrome driver and go to report site URL

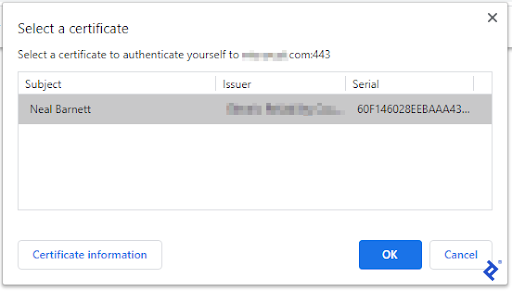
url = "https://reportdata.mytestsite.com/transactionSearch.jsp"

driver = webdriver.Chrome()

driver.get(url)

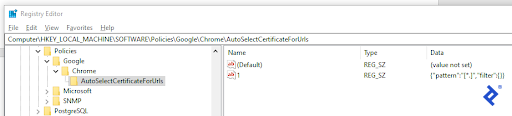
Since I didn’t launch the browser in headless mode, the browser actually appeared and I could see what it was doing. It immediately asked me to select a certificate (which I had installed earlier).

The first problem to tackle was the certificate. How to select the proper one and accept it in order to get into the website? In my first test of the script, I got this prompt:



This wasn’t good. I did not want to manually click the OK button each time I ran my script.

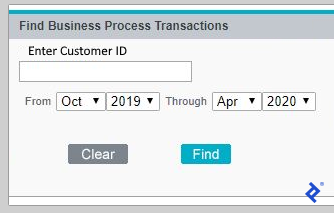
As it turns out, I was able to find a workaround for this - without programming. While I had hoped that Chrome had the ability to pass a certificate name on startup, that feature did not exist. However, Chrome does have the ability to autoselect a certificate if a certain entry exists in your Windows registry. You can set it to select the first certificate it sees, or else be more specific. Since I only had one certificate loaded, I used the generic format.



Thus, with that set, when I told Selenium to launch Chrome and a certificate prompt came up, Chrome would “AutoSelect” the certificate and continue on.

## Problem 2 – Iframes

Okay, so now I was in the site and a form appeared, prompting me to type in the customer ID and the date range of the report.



By examining the form in developer tools (F12), I noticed that the form was presented within an iframe. So, before I could start filling in the form, I needed to “switch” to the proper iframe where the form existed. To do this, I invoked Selenium’s switch-to feature, like so:

# Switch to iframe where form is

frame\_ref = driver.find\_elements\_by\_tag\_name("iframe")[0]

iframe = driver.switch\_to.frame(frame\_ref)

Good, so now in the right frame, I was able to determine the components, populate the customer ID field, and select the date drop-downs:

# Find the Customer ID field and populate it

element = driver.find\_element\_by\_name("custId")

element.send\_keys(custId) # send a test id

# Find and select the date drop-downs

select = Select(driver.find\_element\_by\_name("fromMonth"))

select.select\_by\_visible\_text(from\_month)

select = Select(driver.find\_element\_by\_name("fromYear"))

select.select\_by\_visible\_text(from\_year)

select = Select(driver.find\_element\_by\_name("toMonth"))

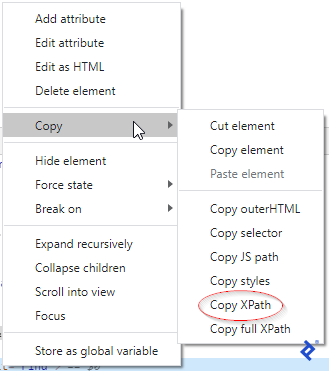
select.select\_by\_visible\_text(to\_month)

select = Select(driver.find\_element\_by\_name("toYear"))

select.select\_by\_visible\_text(to\_year)

## Problem 3 – JavaScript

The only thing left on the form was to “click” the Find button, so it would begin the search. This was a little tricky as the Find button seemed to be controlled by JavaScript and wasn’t a normal “Submit” type button. Inspecting it in developer tools, I found the button image and was able to get the XPath of it, by right-clicking.



Then, armed with this information, I found the element on the page, then clicked it.

# Find the ‘Find’ button, then click it

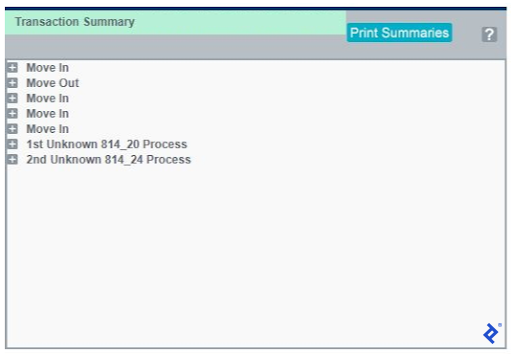
driver.find\_element\_by\_xpath("/html/body/table/tbody/tr[2]/td[1]/table[3]/tbody/tr[2]/td[2]/input").click()

And voilà, the form was submitted and the data appeared! Now, I could just scrape all of the data on the result page and save it as required. Or could I?

## Getting the Data

First, I had to handle the case where the search found nothing. That was pretty straightforward. It would display a message on the search form without leaving it, something like “No records found.” I simply searched for that string and stopped right there if I found it.

But if results did come, the data was presented in divs with a plus sign (+) to open a transaction and show all of its detail. An opened transaction showed a minus sign (-) which when clicked would close the div. Clicking a plus sign would call a URL to open its div and close any open one.



Thus, it was necessary to find any plus signs on the page, gather the URL next to each one, then loop through each to get all data for every transaction.

# Loop through transactions and count

links = driver.find\_elements\_by\_tag\_name('a')

link\_urls = [link.get\_attribute('href') for link in links]

thisCount = 0

isFirst = 1

for url in link\_urls:

if (url.find("GetXas.do?processId") >= 0): # URL to link to transactions

if isFirst == 1: # already expanded +

isFirst = 0

else:

driver.get(url) # collapsed +, so expand

# Find closest element to URL element with correct class to get tran type tran\_type=driver.find\_element\_by\_xpath("//\*[contains(@href,'/retail/transaction/results/GetXas.do?processId=-1')]/following::td[@class='txt\_75b\_lmnw\_T1R10B1']").text

# Get transaction status

status = driver.find\_element\_by\_class\_name('txt\_70b\_lmnw\_t1r10b1').text

# Add to count if transaction found

if (tran\_type in ['Move In','Move Out','Switch']) and

(status == "Complete"):

thisCount += 1

In the above code, the fields I retrieved were the transaction type and the status, then added to a count to determine how many transactions fit the rules that were specified. However, I could have retrieved other fields within the transaction detail, like date and time, subtype, etc.

For this project, the count was returned back to a calling application. However, it and other scraped data could have been stored in a flat file or a database as well.

## Additional Possible Roadblocks and Solutions

Numerous other obstacles might be presented while scraping modern websites with your own browser instance, but most can be resolved. Here are a few:

* **Trying to find something before it appears**

While browsing yourself, how often do you find that you are waiting for a page to come up, sometimes for many seconds? Well, the same can occur while navigating programmatically. You look for a class or other element – and it’s not there!

Luckily, Selenium has the ability to wait until it sees a certain element, and can timeout if the element doesn’t appear, like so:

element = WebDriverWait(driver, 10). until(EC.presence\_of\_element\_located((By.ID, "theFirstLabel")))

* **Getting through a Captcha**

Some sites employ Captcha or similar to prevent unwanted robots (which they might consider you). This can put a damper on web scraping and slow it way down.

For simple prompts (like “what’s 2 + 3?”), these can generally be read and figured out easily. However, for more advanced barriers, there are libraries that can help try to crack it. Some examples are [2Captcha](https://2captcha.com/software/2captcha-python-api), [Death by Captcha](https://deathbycaptcha.com/), and [Bypass Captcha](http://bypasscaptcha.com/).

* **Website structural changes**

Websites are meant to change – and they often do. That’s why when writing a scraping script, it’s best to keep this in mind. You’ll want to think about which methods you’ll use to find the data, and which not to use. Consider partial matching techniques, rather than trying to match a whole phrase. For example, a website might change a message from “No records found” to “No records located” – but if your match is on “No records,” you should be okay. Also, consider whether to match on XPATH, ID, name, link text, tag or class name, or CSS selector – and which is least likely to change.

## Summary: Python and Selenium

This was a brief demonstration to show that almost any website can be scraped, no matter what technologies are used and what complexities are involved. Basically, if you can browse the site yourself, it generally can be scraped.

Now, as a caveat, it does not mean that every website should be scraped. Some have legitimate restrictions in place, and there have been numerous [court cases](https://jaxenter.com/data-scraping-cases-165385.html) deciding the legality of scraping certain sites. On the other hand, some sites welcome and encourage data to be retrieved from their website and in some cases provide an API to make things easier.

Either way, it’s best to check with the terms and conditions before starting any project. But if you do go ahead, be assured that you can get the job done.

**Recommended Resources for Complex Web Scraping:**

* Advanced Python Web Scraping: [Best Practices & Workarounds](https://www.codementor.io/blog/python-web-scraping-63l2v9sf2q)
* Scalable do-it-yourself scraping: [How to build and run scrapers on a large scale](https://www.scrapehero.com/scalable-do-it-yourself-scraping-how-to-build-and-run-scrapers-on-a-large-scale/)