

Data Using Python in Just 6 Steps



Zoltan Bettenbuk



March, 2022



Table of Contents

- Building a Stock Market Scraper With Requests and Beautiful Soup
 - 1. Setting Up Our Project

- 4. Integrating ScraperAPI to Handle IP Rotation and CAPCHAs
- 5. Store Data In a CSV File
- 6. Finished Code: Stock Market Data Script
- Wrapping Up: considerations when running your stock market data scraper

Whether you're an investor tracking your portfolio, or an investment firm looking for a way to stay up-to-date more efficiently, creating a script to scrape stock market data can save you both time and energy.

In this tutorial, we'll build a script to track multiple stock prices, organize them into an easy-to-read CSV file that will update itself with the push of a button, and collect hundreds of data points in a few seconds.

Building a Stock Market Scraper With Requests and Beautiful Soup

For this exercise, we'll be scraping [investing.com](https://www.investing.com) to extract up-to-date stock prices from Microsoft, Coca-Cola, and Nike, and storing it in a CSV file. We'll also show you how to protect your bot from being blocked by anti-scraping mechanisms and techniques using ScraperAPI.

Note: The script will work to scrape stock market data even without ScraperAPI, but will be crucial for scaling your project later.

need to know to scrape almost anything.

With that out of the way, let's jump into the code so you can learn how to scrape stock market data.

1. Setting Up Our Project

To begin, we'll create a folder named "scraper-stock-project", and open it from VScode (you can use any text editor you'd like). Next, we'll open a new terminal and install our two main dependencies for this project:

```
pip3 install bs4
pip3 install requests
```

After that, we'll create a new file named "stockData-scraper.py" and import our dependencies to it.

```
1 | <span class="hljs-keyword">import</span> requests          COPY
2 | <span class="hljs-keyword">from</span> bs4 <span class="hljs-keyword">import</span> requests
```

With Requests, we'll be able to send an HTTP request to download the HTML file which is then passed on to BeautifulSoup for parsing. So let's test it by sending a request to Nike's stock page:

```
1 | url = <span class="hljs-string">'https://www.investing.co COPY i
2 | page = requests.get(url)
3 | <span class="hljs-built_in">print</span>(page.status_code)
```

By printing the status code of the page variable (which is our request), we'll

200

[Done] exited with code=0 in 1.497 seconds

Success! Before moving on, we'll pass the response stored in `page` to BeautifulSoup for parsing:

```
1 | <span class="hljs-attr">soup</span> = BeautifulSoup(page. COPY
```

You can use any parser you want, but we're going with `html.parser` because it's the one we like.

Related Resource: [What is Data Parsing in Web Scraping? \[Code Snippets Inside\]](#)

2. Inspect the Website's HTML Structure (Investing.com)

Before we start scraping, let's open <https://www.investing.com/equities/nike> in our browser to get more familiar with the website.

↑ 142.95 +0.15 (+0.10%)

🕒 18/02 - Closed. Currency in USD ([Disclaimer](#))

After Hours ↓ 142.11 -0.84 (-0.59%) 18/02 - Real-time Data

Volume: 5,129,248

Day's Range: 141.81 - 143.94

52 wk Range: 125.44 - 179.10

As you can see in the screenshot above, the page displays the company's name, stock symbol, price, and price change. At this point, we have three questions to answer:

1. Is the data being injected with JavaScript?
2. What attribute can we use to select the elements?
3. Are these attributes consistent throughout all pages?

Check for JavaScript

There are several ways to verify if some script is injecting a piece of data, but the easiest way is to right-click, View Page Source.

After Hours ↓ 142.11 -0.84 (-0.59%) 18/02 - Re

Volume: 5,129,248

Day's Range: 141.81 - 143.94

52 wk Range: 125.44 - 179.10

Type: Equity Market: [United States](#) ISIN: US654106

InvestingPro

Fair Value P









XXX.XX



Company

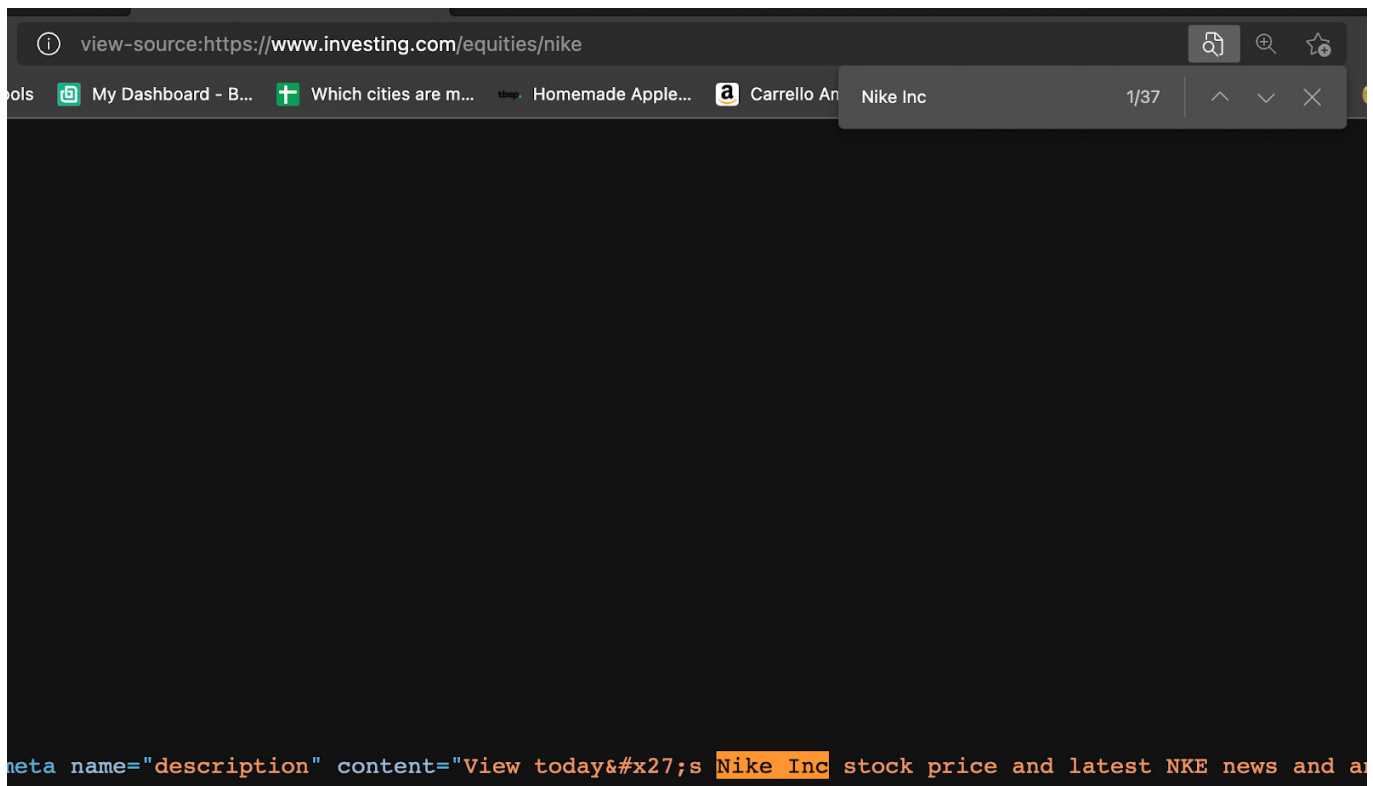
XXX



-  Send Page to Your Devices >
-  Create QR Code for this Page
-  Read Aloud ⇧⌘U
-  Translate to English
-  Add Page to Collections >
-  Share
-  Web Select ⇧⌘X
-  Web Capture ⇧⌘S
- Get image descriptions from Microsoft >
- View Page Source** ⇧⌘U

 Unl

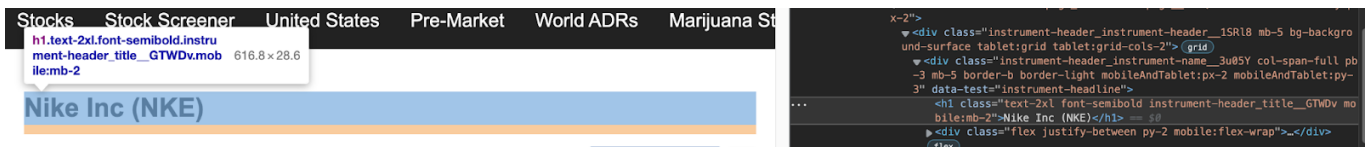
 Ca
Ho



Note: Checking for JavaScript is important because Requests can't execute JavaScript or interact with the website, so if the information is behind a script, we would have to use other tools to extract it, like Selenium.

Picking the CSS Selectors

Now let's inspect the HTML of the site to identify the attributes we can use to select the elements.



Extracting the company's name and the stock symbol will be a breeze. We just need to target the `H1` tag with class `'text-2xl font-semibold instrument-header_title__GTWDv mobile:mb-2'`.

However, the price, price change, and percentage change are separated into different spans.


```

</span>
<div class="text-xl flex items-end flex-wrap"> (flex)
  <span class="instrument-price_change-value__jkuml ml-2.5 text-p
    ositive-main" data-test="instrument-price-change">
    "+"
    <!-- -->
    "0.15"
  </span>
  <span class="instrument-price_change-percent__19cas ml-2.5 text-
    -positive-main" data-test="instrument-price-change-percent">
    "("
    <!-- -->
    "+"
    <!-- -->
    "0.10"
    <!-- -->
    "%)"
  </span>
</div>

```

What's more, depending on whether the change is positive or negative, the class of the element changes, so even if we select each span by using their class attribute, there will still be instances when it won't work.

The good news is that we have a little trick to get it out. Because Beautiful Soup returns a parsed tree, we can now navigate the tree and pick the element we want, even though we don't have the exact CSS class.

What we'll do in this scenario is go up in the hierarchy and find a parent div we can exploit. Then we can use `find_all('span')` to make a list of all the elements containing the span tag – which we know our target data uses. And because it's a list, we can now easily navigate it and pick those we need.

```
2 | <span class="hljs-string">change</span> = soup.find(<span class="
```

Now for a test run:

```
1 | <span class="hljs-built_in">print</span>('Loading: ', url COPY
2 | <span class="hljs-built_in">print</span>(company, price, change
```

And here's the result:

```
[Running] python -u "/Users/lyns/Desktop/scraper-stock-project/test-stockData-scraper.py"
Loading: https://www.investing.com/equities/nike
Nike Inc (NKE) 142.95 (+0.10%)

[Done] exited with code=0 in 2.639 seconds
```

3. Scrape Multiple Stocks

Now that our parser is working, let's scale this up and scrape several stocks. After all, a script for tracking just one stock data is likely not going to be very useful.

We can make our scraper parse and scrape several pages by creating a list of URLs and looping through them to output the data.

```
1 | urls = [ COPY
2 | <span class="hljs-string">'https://www.investing.com/equities/
3 | <span class="hljs-string">'https://www.investing.com/equities/
4 | <span class="hljs-string">'https://www.investing.com/equities/
5 | ]
6 | <span class="hljs-keyword">for</span> url <span class="hljs-ke
```

```
13 | <span class="hljs-built_in">print</span>(company, price, chang
```

Here's the result after running it:

```
[Running] python -u "/Users/lyns/Desktop/scraper-stock-project/test-stockData-scraper.py"
Loading: https://www.investing.com/equities/nike
Nike Inc (NKE) 142.95 (+0.10%)
Loading: https://www.investing.com/equities/coca-cola-co
Coca-Cola Co (KO) 62.54 (+0.68%)
Loading: https://www.investing.com/equities/microsoft-corp
Microsoft Corporation (MSFT) 287.93 (-0.96%)

[Done] exited with code=0 in 3.101 seconds
```

Awesome, it works across the board!

We can keep adding more and more pages to the list but eventually, we'll hit a big roadblock: anti-scraping techniques.

4. Integrating ScraperAPI to Handle IP Rotation and CAPCHAs

Not every website likes to be scraped, and for a good reason. When scraping a website, we need to have in mind that we are sending traffic to it, and if we're not careful, we could be limiting the bandwidth the website has for real visitors, or even increasing hosting costs for the owner. That said, as long as we respect web scraping best practices, we won't have any problems with our projects, and we won't cause the sites we're scraping any issues.

Browser behavior profiling

CAPTCHAs

Monitoring the number of requests from an IP address in a time period

These measures are designed to recognize bots, and block them from accessing the website for days, weeks, or even forever.

Instead of handling all of these scenarios individually, we'll just add two lines of code to make our requests go through ScraperAPI's servers and get everything automated for us.

First, let's [create a free ScraperAPI account](#) to access our API key and 5000 free API credits for our project.

API Key

51e43be283e4db2a5afb62660

Now we're ready to add to our loop a new params variable to store our key and target URL and use urlencode to construct the URL we'll use to send the request inside the page variable.

```
1 | <span class="hljs-keyword">params</span> = {<span class="hljs-keyword">COPY s
2 | page = requests.<span class="hljs-keyword">get</span>(<span cla
```

Every request will now be sent through ScraperAPI, which will automatically rotate our IP after every request, handle CAPCHAs, and use machine learning and statistical analysis to set the best headers to ensure success.

Quick Tip: ScraperAPI also allows us to scrape a dynamic site by setting 'render': true as a parameter in our params variable. ScraperAPI will render the page before sending back the response.

5. Store Data In a CSV File

To store your data in an easy-to-use CSV file, simply add these three lines between your URL list and your loop:

```
1 | file = open('data.csv', 'w')
2 | writer = csv.writer(file)
3 | writer.writerow(['Company', 'URL'])
```

This will create a new CSV file and pass it to our writer (set in the writer variable) to add the first row with our headers.

It's essential to add it outside of the loop, or it will rewrite the file after scraping each page, basically erasing previous data and giving us a CSV file with only the data from the last URL from our list.

In addition, we'll need to add another line to our loop to write the scraped data:

```
1 | writer.writerow([company, url])
```

6. Finished Code: Stock Market Data Script

You've made it! You can now use this script with your own API key and add as many stocks as you want to scrape:

```
1  <span class="hljs-comment">#dependencies</span>                                COPY
2  <span class="hljs-keyword">import</span> requests
3  <span class="hljs-keyword">from</span> bs4 <span class="hljs-keyword">import</span> csv
4  <span class="hljs-keyword">import</span> csv
5  <span class="hljs-keyword">from</span> urllib.parse <span class="hljs-keyword">import</span> urlparse
6  <span class="hljs-comment">#list of URLs</span>
7  urls = [
8  <span class="hljs-string">'https://www.investing.com/equities/
9  <span class="hljs-string">'https://www.investing.com/equities/
10 <span class="hljs-string">'https://www.investing.com/equities/
11 ]
12 <span class="hljs-comment">#starting our CSV file</span>
13 file = <span class="hljs-built_in">open</span>(<span class="hljs-string">'data.csv'</span>, <span class="hljs-keyword">mode</span>=<span class="hljs-string">'a'</span>)
14 writer = csv.writer(file)
15 writer.writerow(<span class="hljs-string">'Company'</span>, <span class="hljs-string">'Price'</span>, <span class="hljs-string">'Change'</span>)
16 <span class="hljs-comment">#looping through our list</span>
17 <span class="hljs-keyword">for</span> url <span class="hljs-keyword">in</span> urls:
18 <span class="hljs-comment">#sending our request through Scrape
19 params = {<span class="hljs-string">'api_key'</span>: <span class="hljs-string">'</span>
20 page = requests.get(<span class="hljs-string">'http://api.scrap
21 <span class="hljs-comment">#our parser</span>
22 soup = BeautifulSoup(page.text, <span class="hljs-string">'html
23 company = soup.find(<span class="hljs-string">'h1'</span>, <span class="hljs-string">'</span>
24 price = soup.find(<span class="hljs-string">'div'</span>, <span class="hljs-string">'</span>
25 change = soup.find(<span class="hljs-string">'div'</span>, <span class="hljs-string">'</span>
26 <span class="hljs-comment">#printing to have some visual feedb
27 <span class="hljs-built_in">print</span>(<span class="hljs-string">'Company, Price, Change'</span>)
28 <span class="hljs-built_in">print</span>(company, price, chang
29 <span class="hljs-comment">#writing the data into our CSV file
30 writer.writerow([company.encode(<span class="hljs-string">'utf
31 file.close()
```

scraper

You need to remember that the stock market isn't always open. For example, if you're scraping data from NYC's stock exchange, it closes at 5 pm EST on Fridays and opens on Monday at 9:30 am. So there's no point in running your scraper over the weekend. It also closes at 4 pm so that you won't see any changes in the price after that.

Trading Hours

PRE-OPENING SESSION: 3:30 A.M. ET

- Limit orders can be entered and will be queued until the Limit Order Auction at 4 a.m. ET

OPENING SESSION: 4:00 A.M. TO 9:30 A.M. ET

- 3:59 a.m. to 4:00 a.m. ET - Opening Auction Freeze Period
- 4:00 a.m. ET - Opening Auction
- 9:29 a.m. to 9:30 a.m. ET - Market Order Freeze Period
- 9:30 a.m. ET - Market Order Auction

Orders that are eligible for the Opening Auction may not be cancelled one minute prior to the Opening Session until the conclusion of the Opening Auction.

Market Orders and Auction-Only Limit Orders may not be canceled. Market Orders and Auction-Only Limit. Orders may not be entered on the same side as an imbalance.

CORE TRADING SESSION: 9:30 A.M. TO 4:00 P.M. ET

 1:5:00  10:00  10:00  Closing Auction Freeze Period
Market-on-Run and Closing Price Disseminated. All Day Orders entered for the core session will be canceled.

Market-on-Close (MOC) and Limit-on-Close (LOC) orders cannot be canceled. MOC and LOC orders may not be entered on the same side as an imbalance.

EXTENDED HOURS: 4:00 PM TO 8:00 P.M. ET

- 8:00 p.m. ET - Limit Orders entered after 4:00 p.m. ET are canceled.
- 8:01 p.m. ET - Portfolio Crossing System (PCS) Cross Orders are executed at the conclusion of extended trading hours.

Another variable to keep in mind is how often you need to update the data. The most volatile times for the stock exchange are opening and closing times. So it might be enough to run your script at 9:30 am, at 11 am, and at 4:30 pm to see how the stocks closed. Monday's opening is also crucial to monitor as many trades occur during this time.

Unlike other markets like Forex, the stock market typically doesn't make too

How to Use Web Scraping to Empower Marketing Decisions

Attracting the right consumers and converting them into paying customers has always required a balance of creativity, industry knowledge, and a clear understanding of consumer

[READ ARTICLE](#)

OCTOBER, 2022

Web Scraping in eCommerce: Use Cases and Tips For Scraping at Scale

Online shopping is nothing new, but we've seen exponential growth in eCommerce sales in recent years. Source Thanks to the pandemic, eCommerce adoption took a

[READ ARTICLE](#)

OCTOBER, 2022

Ready to start scraping?

Get started with 5,000 free API credits or contact sales

GET STARTED FOR FREE

No credit card required



Having built many web scrapers, we repeatedly went through the tiresome process of finding proxies, setting up headless browsers, and handling CAPTCHAs.

That's why we decided to start ScraperAPI, it handles all of this for you so you can scrape any page with a simple API call!



©ScraperAPI

Product

Pricing

Documentation

Developer Guides

FAQ

Blog

Affiliate Program

About

Support

[Free Proxies for Web Scraping](#)

[Top API, Proxies and Scraping Tools](#)

[Comparing Proxies For Web Scraping](#)

[What Are Residential Proxies?](#)

[What Are Rotating Proxies?](#)

[Best Ticket Proxies For Web Scraping](#)

[Extract HTML Table Using R](#)

[Build a Javascript Table Web Scraper](#)

[Scrape HTML Tables in JavaScript](#)

[Web Scraping vs Data Mining](#)

[How to Grab HTTP Headers and Cookies](#)

[Build a LinkedIn Scraper in Python](#)

[Build a Football Data Scraper](#)

[Build an eBay Web Scraper](#)

[Scrape HTML Table Using Python](#)

[Is Web Scraping Legal?](#)