Back to blog

TUTORIALS | SCRAPERS

Python Web Scraping Tutorial: Step-By-Step

Adomas Sulcas
2022-01-06

Share

that make building a tool for web scraping in Python an absolute breeze.

In this web scraping Python tutorial, we will outline everything needed to get started with a simple application. It will acquire text-based data from page sources, store it into a file and sort the output according to set parameters. Options for more advanced features when using Python for web scraping will be outlined at the very end with suggestions for implementation. By following the steps outlined below in this tutorial, you will be able to understand how to do web scraping.

What do we call web scraping?

Web scraping is an automated process of gathering public data. A webpage scraper automatically extracts large amounts of public data from target websites in seconds.

This Python web scraping tutorial will work for all operating systems. There will be slight differences when installing either Python or development environments but not in anything else.

Building a web scraper: Python prepwork

Throughout this entire web scraping tutorial, Python 3.4+ version will be used. Specifically, we used 3.8.3 but any 3.4+ version should work just fine.

For Windows installations, when installing Python make sure to check "PATH installation". PATH installation adds executables to the default Windows Command Prompt executable search. Windows will then recognize commands like "pip" or "python" without requiring users to point it to the directory of the executable (e.g. C:/tools/python/.../python.exe). If you have already installed Python but did not mark the checkbox, just rerun the installation and select modify. On the second screen select "Add to environment variables".

Getting to the libraries

Web scraping with Python is easy due to the many useful libraries available

which you can choose:

- Requests
- Beautiful Soup
- lxml
- Selenium

Requests library

Web scraping starts with sending HTTP requests, such as `POST` or `GET`, to a website's server, which returns a response containing the needed data. However, standard Python HTTP libraries are difficult to use and, for effectiveness, require bulky lines of code, further compounding an already problematic issue.

Unlike other HTTP libraries, the Requests library simplifies the process of making such requests by reducing the lines of code, in effect making the code easier to understand and debug without impacting its effectiveness. The library can be installed from within the terminal using the pip command:

```
pip install requests
```

Requests library provides easy methods for sending HTTP `GET` and `POST` requests. For example, the function to send an HTTP Get request is aptly named `get()`:

```
import requests
response = requests.get("https://oxylabs.io/")
print(response.text)
```

If there is a need for a form to be posted, it can be done easily using the post() method. The form

```
response = requests.post('https://oxylabs.io/', data=form_data)
print(response.text)
```

Requests library also makes it very easy to use proxies that require authentication.

```
proxies={'http': 'http://user:password@proxy.oxylabs.io'}
response = requests.get('http://httpbin.org/ip', proxies=proxies)
print(response.text)
```

But this library has a limitation in that it does not parse the extracted HTML data, i.e., it cannot convert the data into a more readable format for analysis. Also, it cannot be used to scrape websites that are written using purely JavaScript.

Beautiful Soup

Beautiful Soup is a Python library that works with a parser to extract data from HTML and can turn even invalid markup into a parse tree. However, this library is only designed for parsing and cannot request data from web servers in the form of HTML documents/files. For this reason, it is mostly used alongside the Python Requests Library. Note that Beautiful Soup makes it easy to query and navigate the HTML, but still requires a parser. The following example demonstrates the use of the html.parser module, which is part of the Python Standard Library.

#Part 1 – Get the HTML using Requests

```
import requests
url='https://oxylabs.io/blog'
response = requests.get(url)
```

#Part 2 – Find the element

```
print(soup.title)
```

This will print the title element as follows:

```
<h1 class="blog-header">Oxylabs Blog</h1>
```

Due to its simple ways of navigating, searching and modifying the parse tree, Beautiful Soup is ideal even for beginners and usually saves developers hours of work. For example, to print all the blog titles from this page, the `findAll()` method can be used. On this page, all the blog titles are in `h2` elements with class attribute set to `blog-card__content-title`. This information can be supplied to the `findAll` method as follows:

```
blog_titles = soup.findAll('h2', attrs={"class":"blog-card__content-ti
for title in blog_titles:
    print(title.text)
# Output:
# Prints all blog tiles on the page
```

BeautifulSoup also makes it easy to work with CSS selectors. If a developer knows a CSS selector, there is no need to learn `find()` or `find_all()` methods. The following is the same example, but uses CSS selectors:

```
blog_titles = soup.select('h2.blog-card__content-title')
for title in blog_titles:
    print(title.text)
```

What is more, it can be easily configured, with just a few lines of code, to extract any custom publicly available data or to identify specific data types. Our Beautiful Soup tutorial contains more on this and other configurations, as well as how this library works.

lxml

lxml is a parsing library. It is a fast, powerful, and easy-to-use library that works with both HTML and XML files. Additionally, lxml is ideal when extracting data from large datasets. However, unlike Beautiful Soup, this library is impacted by poorly designed HTML, making its parsing capabilities impeded.

The lxml library can be installed from the terminal using the `pip` command:

```
pip install lxml
```

This library contains a module html to work with HTML. However, the lxml library needs the HTML string first. This HTML string can be retrieved using the Requests library as discussed in the previous section. Once the HTML is available, the tree can be built using the `fromstring` method as follows:

```
# After response = requests.get()
from lxml import html
tree = html.fromstring(response.text)
```

This tree object can now be queried using XPath. Continuing the example discussed in the previous section, to get the title of the blogs, the XPath would be as follows:

```
//h2[@class="blog-card__content-title"]/text()
```

`h2` elements.

```
blog_titles = tree.xpath('//h2[@class="blog-card__content-title"]/text(
for title in blog_titles:
    print(title)
```

Suppose you are looking to learn how to use this library and integrate it into your web scraping efforts or even gain more knowledge on top of your existing expertise. In that case, our detailed lxml tutorial is an excellent place to start.

Selenium

As stated, some websites are written using JavaScript, a language that allows developers to populate fields and menus dynamically. This creates a problem for Python libraries that can only extract data from static web pages. In fact, as stated, the Requests library is not an option when it comes to JavaScript. This is where Selenium web scraping comes in and thrives.

This Python web library is an open-source browser automation tool (web driver) that allows you to automate processes such as logging into a social media platform. Selenium is widely used for the execution of test cases or test scripts on web applications. Its strength during web scraping derives from its ability to initiate rendering web pages, just like any browser, by running JavaScript – standard web crawlers cannot run this programming language. Yet, it is now extensively used by developers.

Selenium requires three components:

- Web Browser – Supported browsers are Chrome, Edge, Firefox and Safari
- Driver for the browser – See this page for links to the drivers
- The selenium package

The selenium package can be installed from the terminal:

After installation, the appropriate class for the browser can be imported. Once imported, the object of the class will have to be created. Note that this will require the path of the driver executable. Example for the Chrome browser as follows:

```
from selenium.webdriver import Chrome
driver = Chrome(executable_path='/path/to/driver')
```

Now any page can be loaded in the browser using the `get()` method.

```
driver.get('https://oxylabs.io/blog')
```

Selenium allows use of CSS selectors and XPath to extract elements. The following example prints all the blog titles using CSS selectors:

```
blog_titles = driver.get_elements_by_css_selector(' h2.blog-card__conte
for title in blog_tiles:
    print(title.text)
driver.quit() # closing the browser
```

Basically, by running JavaScript, Selenium deals with any content being displayed dynamically and subsequently makes the webpage's content available for parsing by built-in methods or even Beautiful Soup. Moreover, it can mimic human behavior.

The only downside to using Selenium in web scraping is that it slows the process because it must first execute the JavaScript code for each page before making it available for parsing. As a result, it is unideal for large-scale data extraction. But if you wish to extract data at a lower-scale or the lack of speed is not a drawback, Selenium is a great choice.

	Requests	Beautiful Soup	lxml	Selenium
Purpose	Simplify making HTTP requests	Parsing	Parsing	Simplify making H requests
Ease-of-use	High	High	Medium	Medium
Speed	Fast	Fast	Very fast	Slow
Learning Curve	Very easy (beginner-friendly)	Very easy (beginner-friendly)	Easy	Easy
Documentation	Excellent	Excellent	Good	Good
JavaScript Support	None	None	None	Yes
CPU and Memory Usage	Low	Low	Low	High
Size of Web Scraping Project Supported	Large and small	Large and small	Large and small	Small

failed.

WebDrivers and browsers

Every web scraper uses a browser as it needs to connect to the destination URL. For testing purposes we highly recommend using a regular browser (or not a headless one), especially for newcomers. Seeing how written code interacts with the application allows simple troubleshooting and debugging, and grants a better understanding of the entire process.

Headless browsers can be used later on as they are more efficient for complex tasks. Throughout this web scraping tutorial we will be using the Chrome web browser although the entire process is almost identical with Firefox.

To get started, use your preferred search engine to find the "webdriver for Chrome" (or Firefox). Take note of your browser's current version. Download the webdriver that matches your browser's version.

If applicable, select the requisite package, download and unzip it. Copy the driver's executable file to any easily accessible directory. Whether everything was done correctly, we will only be able to find out later on.

Finding a cozy place for our Python web scraper

One final step needs to be taken before we can get to the programming part of this web scraping tutorial: using a good coding environment. There are many options, from a simple text editor, with which simply creating a *.py file and writing the code down directly is enough, to a fully-featured IDE (Integrated Development Environment).

If you already have Visual Studio Code installed, picking this IDE would be the simplest option. Otherwise, I'd highly recommend PyCharm for any newcomer as it has very little barrier to entry and an intuitive UI. We will assume that PyCharm is used for the rest of the web scraping tutorial.

In PyCharm, right click on the project area and "New -> Python File". Give it a nice name!

Importing and using libraries

Time to put all those pips we installed previously to use:

```
from selenium import webdriver
```

PyCharm might display these imports in grey as it automatically marks unused libraries. Don't accept its suggestion to remove unused libs (at least yet).

We should begin by defining our browser. Depending on the webdriver we picked back in "WebDriver and browsers" we should type in:

```
driver = webdriver.Chrome(executable_path='c:\path\to\windows\webdriver

OR

driver = webdriver.Firefox(executable_path='/nix/path/to/webdriver/exec
```

Picking a URL

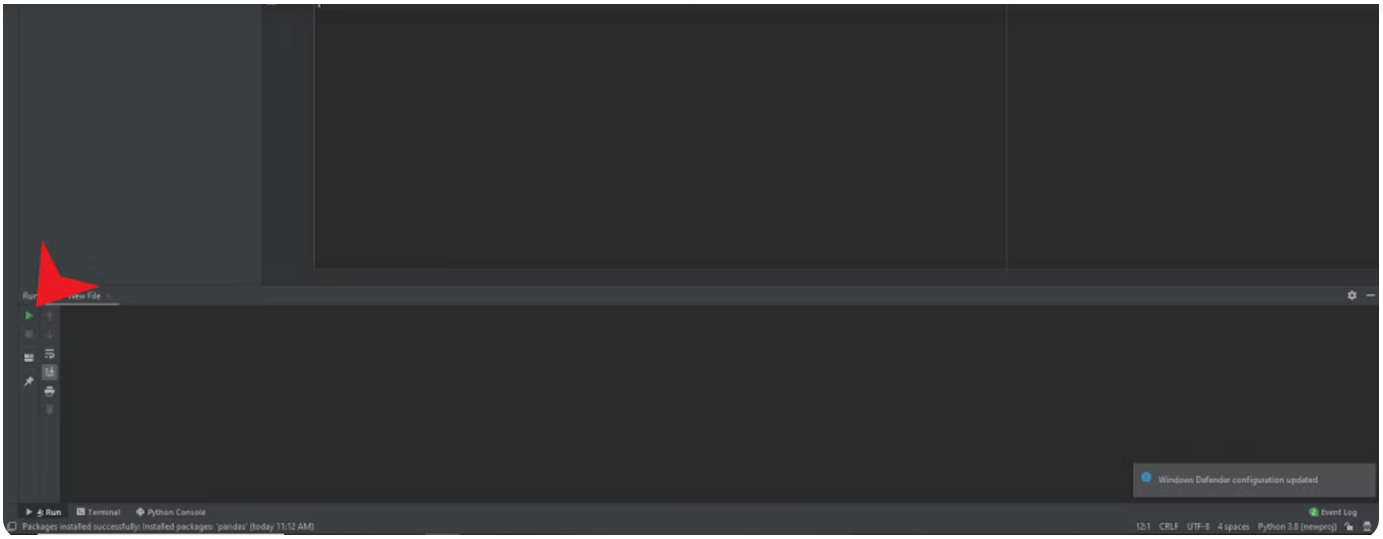
performing specific actions in order to display the required data. Scraping data from Javascript elements requires more sophisticated use of Python and its logic.

- Avoid image scraping. Images can be downloaded directly with Selenium.
- Before conducting any scraping activities ensure that you are scraping public data, and are in no way breaching third-party rights. Also, don't forget to check the robots.txt file for guidance.

Select the landing page you want to visit and input the URL into the driver.get('URL') parameter. Selenium requires that the connection protocol is provided. As such, it is always necessary to attach "http://" or "https://" to the URL.

```
driver.get('https://your.url/here?yes=brilliant')
```

Try doing a test run by clicking the green arrow at the bottom left or by right clicking the coding environment and selecting 'Run'.



Follow the red pointer

executable.

Defining objects and building lists

Python allows coders to design objects without assigning an exact type. An object can be created by simply typing its title and assigning a value.

```
# Object is “results”, brackets make the object an empty list.
# We will be storing our data here.
results = []
```

Lists in Python are ordered, mutable and allow duplicate members. Other collections, such as sets or dictionaries, can be used but lists are the easiest to use. Time to make more objects!

```
# Add the page source to the variable `content`.
content = driver.page_source
# Load the contents of the page, its source, into BeautifulSoup
# class, which analyzes the HTML as a nested data structure and allows
# its elements by using various selectors.
soup = BeautifulSoup(content)
```

Before we go on with, let’s recap on how our code should look so far:

```
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
driver = webdriver.Chrome(executable_path='/nix/path/to/webdriver/execu
driver.get('https://your.url/here?yes=brilliant')
results = []
content = driver.page_source
```

Try rerunning the application again. There should be no errors displayed. If any arise, a few possible troubleshooting options were outlined in earlier chapters.

Extracting data with our Python web scraper

We have finally arrived at the fun and difficult part – extracting data out of the HTML file. Since in almost all cases we are taking small sections out of many different parts of the page and we want to store it into a list, we should process every smaller section and then add it to the list:

```
# Loop over all elements returned by the `findAll` call. It has the fil
# to it in order to limit the data returned to those elements with a gi
for element in soup.findAll(attrs={'class': 'list-item'}):
    ...
```

"soup.findAll" accepts a wide array of arguments. For the purposes of this tutorial we only use "attrs" (attributes). It allows us to narrow down the search by setting up a statement "if attribute is equal to X is true then...". Classes are easy to find and use therefore we shall use those.

Let's visit the chosen URL in a real browser before continuing. Open the page source by using CTRL+U (Chrome) or right click and select "View Page Source". Find the "closest" class where the data is nested. Another option is to press F12 to open DevTools to select Element Picker. For example, it could be nested as:

```
<h4 class="title">
    <a href="...">This is a Title</a>
</h4>
```

Our attribute, "class", would then be "title". If you picked a simple target, in most cases data will be nested in a similar way to the example above. Complex targets might require more effort to get the data out. Let's get back to coding and add the class we found in the source:

Our loop will now go through all objects with the class "title" in the page source. We will process each of them:

```
name = element.find('a')
```

Let's take a look at how our loop goes through the HTML:

```
<h4 class="title">
    <a href="...">This is a Title</a>
</h4>
```

Our first statement (in the loop itself) finds all elements that match tags, whose "class" attribute contains "title". We then execute another search within that class. Our next search finds all the <a> tags in the document (<a> is included while partial matches like are not). Finally, the object is assigned to the variable "name".

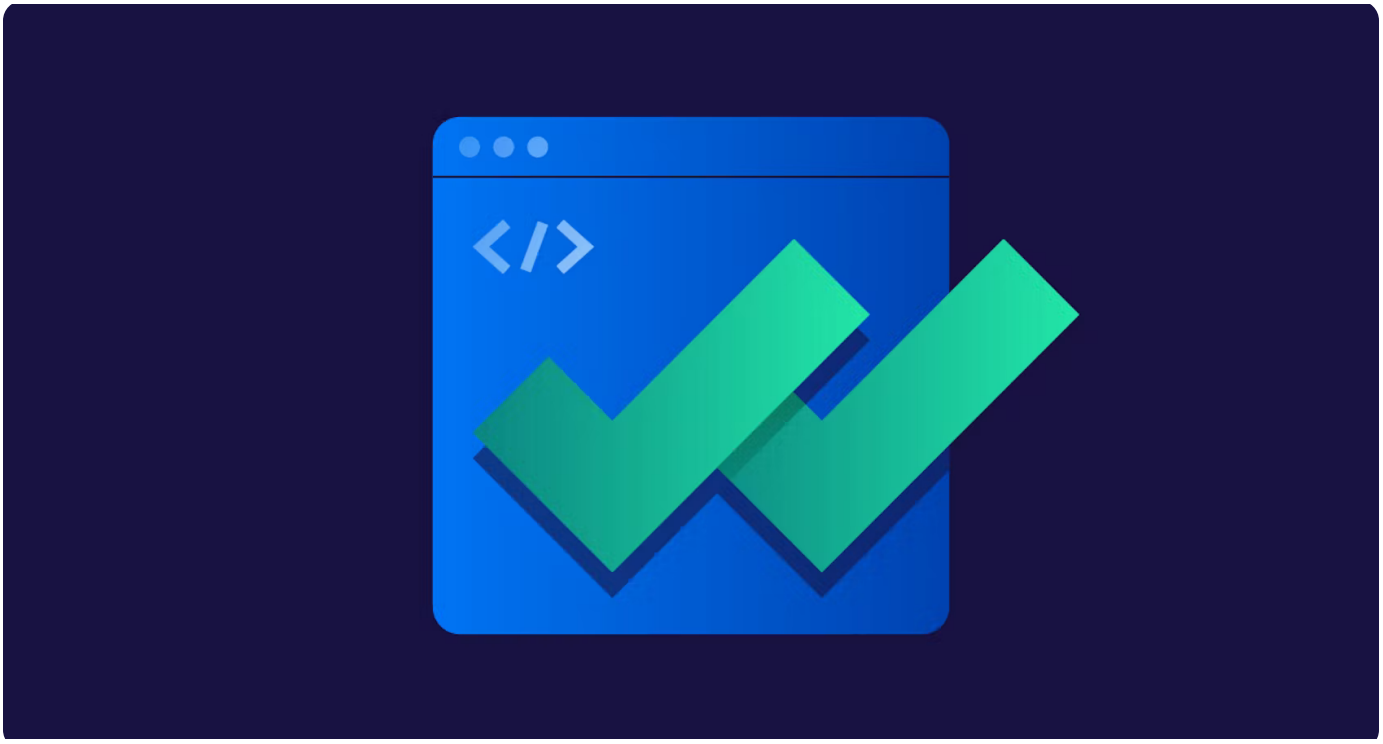
We could then assign the object name to our previously created list array "results" but doing this would bring the entire <a href...> tag with the text inside it into one element. In most cases, we would only need the text itself without any additional tags.

```
# Add the object of "name" to the list "results".
# `<element>.text` extracts the text in the element, omitting the HTML
results.append(name.text)
```

```
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
driver = webdriver.Chrome(executable_path='/nix/path/to/webdriver/execu
driver.get('https://your.url/here?yes=brilliant')
results = []
content = driver.page_source
soup = BeautifulSoup(content)
for element in soup.findAll(attrs={'class': 'title'}):
    name = element.find('a')
    results.append(name.text)
```

Note that the two statements after the loop are indented. Loops require indentation to denote nesting. Any consistent indentation will be considered legal. Loops without indentation will output an "IndentationError" with the offending statement pointed out with the "arrow".

Exporting the data to CSV



One of the simplest ways to check if the data you acquired during the previous steps is being collected correctly is to use "print". Since arrays have many different values, a simple loop is often used to separate each entry to a separate line in the output:

```
for x in results:
   print(x)
```

Both "print" and "for" should be self-explanatory at this point. We are only initiating this loop for quick testing and debugging purposes. It is completely viable to print the results directly:

```
print(results)
```

So far our code should look like this:

```
driver = webdriver.Chrome(executable_path='/nix/path/to/webdriver/execu
driver.get('https://your.url/here?yes=brilliant')
results = []
content = driver.page_source
soup = BeautifulSoup(content)
for a in soup.findAll(attrs={'class': 'class'}):
    name = a.find('a')
    if name not in results:
        results.append(name.text)
for x in results:
    print(x)
```

Running our program now should display no errors and display acquired data in the debugger

similar but moving our data to a csv file.

```
df = pd.DataFrame({'Names': results})
df.to_csv('names.csv', index=False, encoding='utf-8')
```

Our two new statements rely on the pandas library. Our first statement creates a variable "df" and turns its object into a two-dimensional data table. "Names" is the name of our column while "results" is our list to be printed out. Note that pandas can create multiple columns, we just don't have enough lists to utilize those parameters (yet).

Our second statement moves the data of variable "df" to a specific file type (in this case "csv"). Our first parameter assigns a name to our soon-to-be file and an extension. Adding an extension is necessary as "pandas" will otherwise output a file without one and it will have to be changed manually. "index" can be used to assign specific starting numbers to columns. "encoding" is used to save data in a specific format. UTF-8 will be enough in almost all cases.

```
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
driver = webdriver.Chrome(executable_path='/nix/path/to/webdriver/execu
driver.get('https://your.url/here?yes=brilliant')
results = []
content = driver.page_source
soup = BeautifulSoup(content)
for a in soup.findAll(attrs={'class': 'class'}):
    name = a.find('a')
    if name not in results:
        results.append(name.text)
df = pd.DataFrame({'Names': results})
df.to_csv('names.csv', index=False, encoding='utf-8')
```

Exporting the data to Excel

Pandas library features a function to export data to Excel. It makes it a lot easier to move data to an Excel file in one go.

```
df = pd.DataFrame({'Names': results})
df.to_excel('names.xlsx', index=False, encoding='utf-8')
```

The new statement creates a DataFrame - a two-dimensional tabular data structure. The column label is "Name," and the rows include data from the results array. Pandas can span more than one column, though that's not required here as we only have a single column of data.

The second statement transforms the DataFrame into an Excel file (".xlsx"). The first argument to the function specifies the filename - "names.xlsx". Followed by the index argument set to false to avoid numbering the rows. Finally, the encoding is set to "utf-8" to support a broader range of characters.

```
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver

driver = webdriver.Chrome(executable_path='/path/to/webdriver/executabl
driver.get('https://your.url/here?yes=brilliant')
content = driver.page_source
soup = BeautifulSoup(content)

results = []
for a in soup.findAll(attrs={'class': 'class'}):
    name = a.find('a')
    if name not in results:
        results.append(name.text)

df = pd.DataFrame({'Names': results})
```

To sum up, the code above creates a "names.xlsx" file with a "Names" column that includes all the data we have in the results array so far.

More lists. More!

Python web scraping often requires many data points

For the purposes of this tutorial, we will try something slightly different. Since acquiring data from the same class would just mean appending to an additional list, we should attempt to extract data from a different class but, at the same time, maintain the structure of our table.

Obviously, we will need another list to store our data in.

```
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
driver = webdriver.Chrome(executable_path='/nix/path/to/webdriver/execu
driver.get('https://your.url/here?yes=brilliant')
results = []
other_results = []
for b in soup.findAll(attrs={'class': 'otherclass'}):
# Assume that data is nested in ‘span’.
    name2 = b.find('span')
    other_results.append(name.text)
```

Since we will be extracting an additional data point from a different part of the HTML, we will need an additional loop. If needed we can also add another “if” conditional to control for duplicate entries:

Finally, we need to change how our data table is formed:

```
df = pd.DataFrame({'Names': results, 'Categories': other_results})
```

So far the newest iteration of our code should look something like this:

```
import pandas as pd
from bs4 import BeautifulSoup
```

```
content = driver.page_source
for a in soup.findAll(attrs={'class': 'class'}):
    name = a.find('a')
    if name not in results:
        results.append(name.text)
for b in soup.findAll(attrs={'class': 'otherclass'}):
    name2 = b.find('span')
    other_results.append(name.text)
df = pd.DataFrame({'Names': results, 'Categories': other_results})
df.to_csv('names.csv', index=False, encoding='utf-8')
```

If you are lucky, running this code will output no error. In some cases "pandas" will output an "ValueError: arrays must all be the same length" message. Simply put, the length of the lists "results" and "other_results" is unequal, therefore pandas cannot create a two-dimensional table.

There are dozens of ways to resolve that error message. From padding the shortest list with "empty" values, to creating dictionaries, to creating two series and listing them out. We shall do the third option:

```
series1 = pd.Series(results, name = 'Names')
series2 = pd.Series(other_results, name = 'Categories')
df = pd.DataFrame({'Names': series1, 'Categories': series2})
df.to_csv('names.csv', index=False, encoding='utf-8')
```

Note that data will not be matched as the lists are of uneven length but creating two series is the easiest fix *if* two data points are needed. Our final code should look something like this:

```
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
```

```
soup = BeautifulSoup(content)
for a in soup.findAll(attrs={'class': 'class'}):
    name = a.find('a')
    if name not in results:
        results.append(name.text)
for b in soup.findAll(attrs={'class': 'otherclass'}):
    name2 = b.find('span')
    other_results.append(name.text)
series1 = pd.Series(results, name = 'Names')
series2 = pd.Series(other_results, name = 'Categories')
df = pd.DataFrame({'Names': series1, 'Categories': series2})
df.to_csv('names.csv', index=False, encoding='utf-8')
```

Running it should create a csv file named "names" with two columns of data.

Web scraping with Python best practices

Our first web scraper should now be fully functional. Of course it is so basic and simplistic that performing any serious data acquisition would require significant upgrades. Before moving on to greener pastures, I highly recommend experimenting with some additional features:

- Create matched data extraction by creating a loop that would make lists of an even length.
- Scrape several URLs in one go. There are many ways to implement such a feature. One of the simplest options is to simply repeat the code above and change URLs each time. That would be quite boring. Build a loop and an array of URLs to visit.
- Another option is to create several arrays to store different sets of data and output it into one file with different rows. Scraping several different types of information at once is an important part of e-commerce data acquisition.
- Once a satisfactory web scraper is running, you no longer need to watch the browser perform its actions. Get headless versions of either Chrome or Firefox browsers and use those to reduce load times.
- Create a scraping pattern. Think of how a regular user would browse the internet and try to automate their actions. New libraries will definitely be needed. Use "import time" and "from

Try creating a long-lasting loop that rechecks certain URLs and scrapes data at set intervals. Ensure that your acquired data is always fresh.

- Make use of the Python Requests library. Requests is a powerful asset in any web scraping toolkit as it allows to optimize HTTP methods sent to servers.
- Finally, integrate proxies into your web scraper. Using location specific request sources allows you to acquire data that might otherwise be inaccessible.

If you enjoy video content more, watch our embedded, simplified version of the web scraping tutorial!

complicated process.

If you are interested in our in-house solution, check Web Scraper API for general purpose scraping applications.

If you want to find out more about how proxies or advanced data acquisition tools work, or about specific web scraping use cases, such as web scraping job postings or building a yellow page scraper, check out our blog. We have enough articles for everyone: a more detailed guide on how to avoid blocks when scraping and tackle pagination, is web scraping legal, an in-depth walkthrough on what is a proxy and many more!

About the author

Adomas Sulcas
PR Team Lead

Adomas Sulcas is a PR Team Lead at Oxylabs. Having grown up in a tech-minded household, he quickly developed an interest in everything IT and Internet related. When he is not nerding out online or immersed in reading, you will find him on an adventure or coming up with wicked business ideas.

Learn more about Adomas Sulcas

TUTORIALS | DATA ACQUISITION | SCRAPERS

Guide to Using Google Sheets for Basic Web Scraping

In this guide, you will learn how to scrape websites using Google Sheets, and you will find out about the functions in use and how to fix common errors.

Vytenis Kaubre
2022-11-08

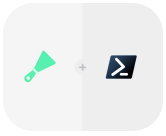
TUTORIALS | SCRAPERS

Guide to Extracting Website Data by Using Excel VBA

Follow a detailed tutorial of extracting website data by using Excel VBA as well as understand the main advantages and disadvantages of this scraping technique.

Yelyzaveta Nechytailo
2022-10-19

TUTORIALS | SCRAPERS



Web Scraping With PowerShell: The Ultimate Guide

Follow this PowerShell web scraping tutorial to build your own tool for public data acquisition.

Roberta Aukstikalnyte
2022-10-17

Get the latest news from
data gathering world

I'm interested

Scale up your business with Oxylabs®

Register

Contact sales

Our values

Affiliate program

Service partners

Press area

Residential Proxies sourcing

Careers

Our products

OxyCon

Project 4beta

Shared Datacenter Proxies

Dedicated Datacenter Proxies

Residential Proxies

Next-Gen Residential Proxies

Static Residential Proxies

SOCKS5 Proxies

Mobile Proxies

Rotating ISP Proxies

TOP LOCATIONS

United States

United Kingdom

Canada

Germany

India

All locations

RESOURCES

FAQ

Documentation

Blog

SCRAPER APIS

SERP Scraper API

E-Commerce Scraper API

Web Scraper API

INNOVATION HUB

Next-Gen Residential Proxies story

Adaptive Parser

Oxylabs' Patents

GET IN TOUCH
General: hello@oxylabs.io
Support: support@oxylabs.io
Career: career@oxylabs.io

Certified data centers and upstream providers

English

Connect with us