



# Comprehensive Guide on Data Collection:

Web Scraping with Python  
From A to Z



# Introduction

In today's fast-changing business world, data gathering is essential for every data-driven business, so the concept of web scraping becomes more and more known to many. Data collection at scale manually is a time-consuming task, so by automating the whole process with web scraping, companies can focus on more vital tasks.

Getting started in web scraping is simple, except when it isn't, which is why you are here. It requires some time and effort to understand the main principles of web scraping with Python. If you're interested in starting web scraping, we can assure you that you're in the right place.

Python is the most popular programming language for web scraping because it can handle almost all data extraction processes smoothly. This is the reason why we prepared you two tutorials: how to scrape text-based data and how to scrape images with Python.

Over the years we spent in the web scraping industry, we've collected many technical insights for you to begin web scraping easily. By following the steps outlined in our articles, you will understand web scraping basics.

<b>Why is Python Used For Web Scraping</b>	<b>4</b>
Python advantages for web scraping	4
Python libraries used for web scraping	5
<b>Python Web Scraping Tutorial: Step-By-Step</b>	<b>7</b>
Building a web scraper: Python prepwork	7
Getting to the libraries	8
WebDrivers and browsers	8
Finding a cozy place for our Python web scraper	9
Importing and using libraries	9
Picking a URL	10
Defining objects and building lists	11
Extracting data with our Python web scraper	12
Exporting the data	15
More lists. More!	18
Web scraping with Python best practices	21
<b>Scrape Images From a Website with Python</b>	<b>24</b>
Libraries: new and old	24
Back to square one	25
Moving forward with defined functions	27
Time to extract images from the website	28
Putting it all together	29
Cleaning up	31
<b>Conclusion</b>	<b>34</b>

# Why is Python Used For Web Scraping

Python is an interpreted, general-purpose, and high-level programming language. Python is used for pretty much anything you would need, from building web apps to data analysis. Python's creators gave attention to its syntax and code readability, so now it allows developers to express concepts in fewer lines of code. This is the main reason why Python was created in the first place.

You can find comparisons that Python is like a chameleon of the programming world. Well, it's not a lie. If you ever wonder where Python is used for, it is everywhere, and you may not realize how widespread it is. The most common fields where Python is indispensable are web development, Machine Learning (ML) and Artificial Intelligence (AI) development, data science, video game development, and, of course, web scraping.

## Python advantages for web scraping

The best part is that Python, compared to other programming languages, is easy to learn, clear to read, and simple to write in.

**Diverse libraries.** Python has a fantastic collection of libraries such as *BeautifulSoup*, *Selenium*, *lxml*, and much more. These libraries are a perfect fit for web scraping and, also, for further work with extracted data. You'll find more information about these libraries below.

**Easy to use.** To put it simply, Python is easy to code. Of course, it's wrong to believe that you would easily write a code for web scraping without any programming knowledge. But, compared to other languages, it's much easier to use as you do not have to add semicolons like ";" or curly-brackets "{}" everywhere. Many developers agree that this is the reason why Python is less messy. Furthermore, Python syntax is clear and easy to read. Developers can simply navigate between different blocks in the code.

**Saves time.** As you probably know, web scraping was created to simplify time-consuming tasks like collecting vast amounts of data manually. Using Python for web scraping is similar because you are able to write a little bit of code that completes a large task. Python saves a bunch of developers' time.

**Community.** As Python is one of the most popular programming languages, it also has a very active community. Developers are sharing their knowledge on various questions, so if you are struggling while writing the code, you can always search for help.

## Python libraries used for web scraping

Powerful frameworks and libraries, explicitly built for web scraping, are the main reason why Python is a popular choice for data extraction. We'll take a closer look at all the essential libraries that makes every developer's web scraping tasks much easier.

**Selenium.** The primary purpose of Selenium is to test web applications. However, it's not limited to do just that as you can use Selenium for web scraping. It automates script processes because, for web scraping, the script needs to interact with a browser to perform repetitive tasks like clicking, scrolling, etc.

**BeautifulSoup.** BeautifulSoup is widely used for parsing the HTML files. According to their documentation, BeautifulSoup library is precisely built for pulling data out of HTML and XML files. It saves developers hours or even days of work.

**Pandas.** According to their official site, Pandas in web scraping is used for data manipulation and analysis. Pandas features include flexible reshaping and pivoting of data sets, reading and writing data between in-memory data structures and different formats, aggregating or transforming data, etc.

**Requests (HTTP for Humans).** This library is used for making various types of HTTP requests like GET, POST. Python Requests library

retrieves only static content of the page. This library doesn't parse the HTML data extracted from web sites. However, *requests* library can be used for basic web scraping tasks.

**lxml.** This library is similar to BeautifulSoup because developers use lxml for processing XML and HTML files in the Python language.

Now that we know what Python is good for, it should be easier to understand its appeal, especially for web scraping.

# Python Web Scraping Tutorial: Step-By-Step

Python is one of the easiest ways to get started as it is an object-oriented language. Python's classes and objects are significantly easier to use than in any other language. Additionally, many libraries exist that make building a tool for web scraping in Python an absolute breeze.

In this web scraping Python tutorial, we'll outline everything needed to get started with a simple application. It'll 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 you will be able to understand how to do web scraping.

This 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

A barebones installation isn't enough for web scraping. We'll be using three important libraries – BeautifulSoup v4, Pandas, and Selenium.

To install these libraries, start the terminal of your OS. Type in:

```
pip install BeautifulSoup4 pandas selenium
```

Each of these installations take anywhere from a few seconds to a few minutes to install. If your terminal freezes, gets stuck when downloading or extracting the package or any other issue outside of a total meltdown arises, use CTRL+C to abort any running installation.

Further steps in this web scraping with Python tutorial assume a successful installation of the previously listed libraries. If you receive a *"NameError: name \* is not defined"* it is likely that one of these installations has 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:

```
import pandas as pd
from bs4 import BeautifulSoup
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\executable.exe')
```

OR

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

## Picking a URL

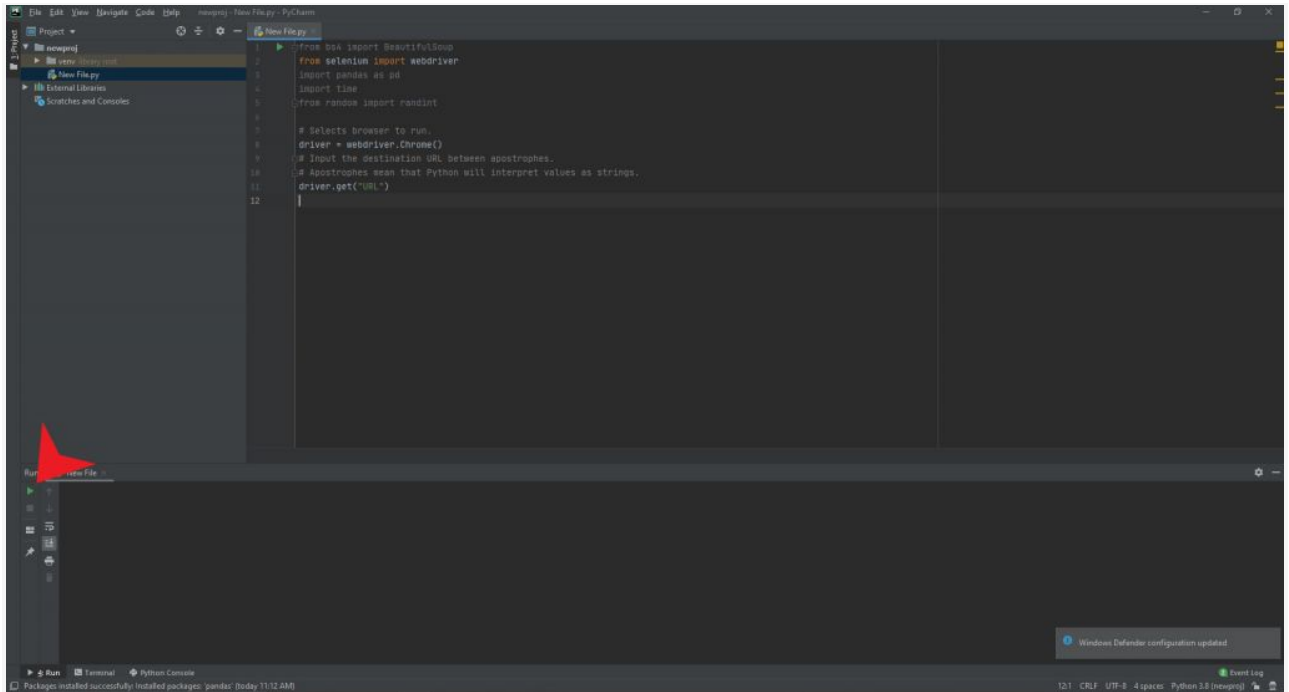
Before performing our first test run, choose a URL. As this web scraping tutorial is intended to create an elementary application, we highly recommended picking a simple target URL:

- Avoid data hidden in Javascript elements. These sometimes need to be triggered by performing specific actions in order to display 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 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

If you receive an error message stating that a file is missing then turn double check if the path provided in the driver “webdriver.\*” matches the location of the webdriver executable. If you receive a message that there is a version mismatch redownload the correct webdriver 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 to select

# 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/ex
ecutable')

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

results = []

content = driver.page_source
soup = BeautifulSoup(content)
```

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 filter `attrs` given

# to it in order to limit the data returned to those
elements with a given class only.

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:

```
# Change 'list-item' to 'title'.  
for element in soup.findAll(attrs={'class': 'title'}):  
    ...
```

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 <span> 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 tags.  
  
results.append(name.text)
```

Our loop will go through the entire page source, find all the occurrences of the classes listed above, then append the nested data to our list:

```
import pandas as pd  
  
from bs4 import BeautifulSoup  
  
from selenium import webdriver  
  
driver =  
webdriver.Chrome(executable_path='/nix/path/to/webdriver/ex  
ecutable')  
  
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

Even if no syntax or runtime errors appear when running our program, there still might be semantic errors. You should check

whether we actually get the data assigned to the right object and move to the array correctly.

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/ex
ecutable')

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 window. While “print” is great for testing purposes, it isn’t all that great for parsing and analyzing data.

You might have noticed that “import pandas” is still greyed out so far. We will finally get to put the library to good use. I recommend removing the “print” loop for now as we will be doing something 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/ex
ecutable')
```

```
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')
```

No imports should now be greyed out and running our application should output a “names.csv” into our project directory. Note that a “Guessed At Parser” warning remains. We could remove it by installing a third party parser but for the purposes of this Python web scraping tutorial the default HTML option will do just fine.

## More lists. More!

Many web scraping operations will need to acquire several sets of data. For example, extracting just the titles of items listed on an e-commerce website will rarely be useful. In order to gather meaningful information and to draw conclusions from it at least two data points are needed.

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/ex
ecutable')

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

from selenium import webdriver

```

```

driver =
webdriver.Chrome(executable_path='/nix/path/to/webdriver/ex
ecutable')

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

results = []
other_results = []

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

driver =
webdriver.Chrome(executable_path='/nix/path/to/webdriver/ex
ecutable')

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

results = []

other_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 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 random import randint” to create wait times between pages. Add “scrollto()” or use specific key inputs to move around the

browser. It's nearly impossible to list all of the possible options when it comes to creating a scraping pattern.

- Create a monitoring process. Data on certain websites might be time (or even user) sensitive. 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.

# Scrape Images From a Website with Python

Previously we outlined how to scrape text-based data with Python. Throughout the tutorial we went through the entire process: all the way from installing Python, getting the required libraries, setting everything up to coding a basic web scraper and outputting the acquired data into a .csv file. In the second installment, we will learn how to scrape images from a website and store them in a set location.

Before conducting image scraping please consult with legal professionals to be sure that you are not breaching third party rights, including but not limited to, intellectual property rights.

## Libraries: new and old

We will need quite a few libraries in order to extract images from a website. In the basic web scraper tutorial we used BeautifulSoup, Selenium and pandas to gather and output data into a .csv file. We will do all these previous steps to export scraped data (i.e. image URLs).

Of course, gathering image URLs into a list is not enough. We will use several other libraries to store the *content* of the URL into a variable, convert it into an image object and then save it to a specified location. Our newly acquired libraries are Pillow and Requests.

If you missed the previous installment:

```
pip install beautifulsoup4 selenium pandas
```

Install these libraries as well:



```
#install the Pillow library (used for image processing)
pip install Pillow

#install the requests library (used to send HTTP requests)
pip install requests
```

Additionally, we will use built-in libraries to **download images from a website**, mostly to store our acquired files in a specified folder.

## Back to square one

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

Our data extraction process begins almost exactly the same (we will import libraries as needed). We assign our preferred webdriver, select the URL from which we'll scrape image links and create a list to store them in. As our Chrome driver arrives at the URL, we use the variable 'content' to point to the page source and then "soupify" it with BeautifulSoup.

In the previous tutorial, we performed all actions by using built-in and library defined functions. While we could do another tutorial without defining any functions, it is an *extremely* useful tool for just about any project:

```
# Example on how to define a function and select custom
arguments for the

# code that goes into it.

def function_name(arguments):

    # Function body goes here.
```

We'll move our URL scraper into a defined function. Additionally, we'll reuse the same code as we used in the previous tutorial and repurpose it to scrape full URLs.

### Before

```
for a in soup.findAll(attrs={'class': 'class'}):

    name = a.find('a')

    if name not in results:

        results.append(name.text)
```

### After

```
#picking a name that represents the functions will be
useful later on.

def parse_image_urls(classes, location, source):

    for a in soup.findAll(attrs={'class': classes}):

        name = a.find(location)

        if name not in results:

            results.append(name.get(source))
```

Note that we now append in a different manner. Instead of appending the text, we use another function 'get()' and add a new

parameter 'source' to it. We use 'source' to indicate the field in the website where image links are stored . They will be nested in a 'src', 'data-src' or other similar HTML tags.

## Moving forward with defined functions

Let's assume that our target URL has image links nested in the classes 'blog-card\_\_link', 'img' and that the URL itself is in the 'src' attribute of the element. We would call our newly defined function as such:

```
parse_image_urls("blog-card__link", "img", "src")
```

Our code should now look something like this:

```
import pandas as pd

from bs4 import BeautifulSoup

from selenium import webdriver

driver =
webdriver.Chrome(executable_path='/nix/path/to/webdriver/ex
ecutable')

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

results = []

content = driver.page_source

soup = BeautifulSoup(content)

def parse_image_urls(classes, location, source):
    for a in soup.findAll(attrs={'class': classes}):
        name = a.find(location)
        if name not in results:
            results.append(name.get(source))
```

```
parse_image_urls("blog-card__link", "img", "src")
```

Since we sometimes want to export scraped data and we had already used pandas before, we can check by outputting everything into a “.csv” file. If needed, we can always check for any possible semantic errors this way.

```
df = pd.DataFrame("links": results)  
df.to_csv('links.csv', index=False, encoding='utf-8')
```

If we run our code right now, we should get a “links.csv” file outputted right into the running directory.

## Time to extract images from the website

Assuming that we didn’t run into any issues at the end of the previous section, we can continue to download images from websites.

```
#import library requests to send HTTP requests  
import requests  
for b in results:  
#add the content of the url to a variable  
    image_content = requests.get(b).content
```

We will use the requests library to acquire the content stored in the image URL. Our “for” loop above will iterate over our ‘results’ list.

```
#io manages file-related in/out operations  
import io  
  
#creates a byte object out of image_content and point the  
variable image_file to it  
image_file = io.BytesIO(image_content)
```

We are not done yet. So far the “image” we have above is just a Python object.

```
#we use Pillow to convert our object to an RGB image
from PIL import Image
image = Image.open(image_file).convert('RGB')
```

We are still not done as we need to find a place to save our images. Creating a folder “Test” for the purposes of this tutorial would be the easiest option.

```
#pathlib let's us point to specific locations. Will be used
to save our images.

import pathlib

#hashlib allows us to get hashes. We will be using sha1 to
name our images.

import hashlib

#sets a file_path variable which is pointed to
#our directory and creates a file based on #the sha1 hash
of 'image_content'

#and uses .hexdigest to convert it into a string.

file_path = pathlib.Path('nix/path/to/test',
hashlib.sha1(image_content).hexdigest()[:10] + '.png')

image.save(file_path, "PNG", quality=80)
```

## Putting it all together

Let's combine all of the previous steps without any comments and see how it works out. Note that pandas are greyed out as we are not extracting data into any tables. We kept it in for the sake of convenience. Use it if you need to see or double-check the outputs.

```
import hashlib
import io
from pathlib import Path
import pandas as pd
import requests
```

```

from bs4 import BeautifulSoup
from PIL import Image
from selenium import webdriver

driver =
webdriver.Chrome(executable_path='/nix/path/to/webdriver/ex
ecutable')
driver.get('https://your.url/here?yes=brilliant')
driver.execute_script("window.scrollTo(0,
document.body.scrollHeight);")
results = []
content = driver.page_source
soup = BeautifulSoup(content)

def gets_url(classes, location, source):
    results = []
    for a in soup.findAll(attrs={'class': classes}):
        name = a.find(location)
        if name not in results:
            results.append(name.get(source))
    return results

driver.quit()

if __name__ == "__main__":
    returned_results = gets_url("blog-card__link", "img",
"src")
    for b in returned_results::
        image_content = requests.get(b).content
        image_file = io.BytesIO(image_content)
        image = Image.open(image_file).convert('RGB')
        file_path = pathlib.Path('nix/path/to/test',
hashlib.shal(image_content).hexdigest()[:10] + '.png')
        image.save(file_path, "PNG", quality=80)

```

For efficiency, we quit our webdriver by using “driver.quit()” after retrieving the URL list we need. We no longer need that browser as everything is stored locally.

Running our application will output one of two results:

- Images are outputted into the folder we selected by defining the ‘file\_path’ variable.

- Python outputs a 403 Forbidden HTTP error.

Obviously, getting the first result means we are finished. We would receive the second outcome if we were to scrape our `/blog/` page. Fixing the second outcome will take a little bit of time in most cases, although, at times, there can be more difficult scenarios.

Whenever we use the `requests` library to send a request to the destination server, a default user-agent “Python-urllib/version.number” is assigned. Some web services might block these user-agents specifically as they are guaranteed to be bots. Fortunately, the `requests` library allows us to assign any user-agent (or an entire header) we want:

```
image_content = requests.get(b, headers={'User-agent':  
'Mozilla/5.0'}).content
```

Adding a user-agent will be enough for most cases. There are more complex cases where servers might try to check other parts of the HTTP header in order to confirm that it is a genuine user.

## Cleaning up

Our task is finished but the code is still messy. We can make our application more readable and reusable by putting everything under defined functions:

```
import io  
import pathlib  
import hashlib  
import pandas as pd  
import requests  
from bs4 import BeautifulSoup  
from PIL import Image  
from selenium import webdriver  
  
def get_content_from_url(url):  
    driver = webdriver.Chrome() # add "executable_path=" if  
    driver not in running directory
```

```

    driver.get(url)
    driver.execute_script("window.scrollTo(0,
document.body.scrollHeight);")
    page_content = driver.page_source
    driver.quit() # We do not need the browser instance for
further steps.
    return page_content

def parse_image_urls(content, classes, location, source):
    soup = BeautifulSoup(content)
    results = []
    for a in soup.findAll(attrs={"class": classes}):
        name = a.find(location)
        if name not in results:
            results.append(name.get(source))
    return results

def save_urls_to_csv(image_urls):
    df = pd.DataFrame({"links": image_urls})
    df.to_csv("links.csv", index=False, encoding="utf-8")

def get_and_save_image_to_file(image_url, output_dir):
    response = requests.get(image_url,
headers={"User-agent": "Mozilla/5.0"})
    image_content = response.content
    image_file = io.BytesIO(image_content)
    image = Image.open(image_file).convert("RGB")
    filename = hashlib.sha1(image_content).hexdigest()[:10]
+ ".png"
    file_path = output_dir / filename
    image.save(file_path, "PNG", quality=80)

def main():
    url = "https://your.url/here?yes=brilliant"
    content = get_content_from_url(url)
    image_urls = parse_image_urls(
        content=content, classes="blog-card__link",
location="img", source="src",
    )
    save_urls_to_csv(image_urls)

    for image_url in image_urls:
        get_and_save_image_to_file(

```



```
        image_url,  
output_dir=pathlib.Path("nix/path/to/test"),  
    )  
  
if __name__ == "__main__": #only executes if imported as  
main file  
    main()
```

Everything is now nested under clearly defined functions and can be called when imported. Otherwise it will run as it had previously.

By using the code outlined above, you should now be able to complete basic image scraping tasks such as to download all images from a website in one go.

# Conclusion

Python is a perfect fit for building web scrapers and extracting data as it has a large selection of libraries, and an active community to search for help if you have issues with coding. One of the most important parts why use Python for web scraping is that Python is easy to learn, clear to read, and simple to write in.

Building web scrapers, acquiring data, and drawing conclusions from large amounts of information is inherently an interesting and complicated process. We have provided several tutorials on how to start web scraping in Python. From here onwards, you are on your own.

By applying our advices on how to begin web scraping in Python, you'll be able to implement web scraping in your company's daily tasks to get the required data for making data-driven decisions. Don't forget that the web scraping industry is constantly evolving, so continuous learning is one of the successful web scraping elements.

If the whole web scraping process seems like a time-consuming task and you would rather spend time analyzing data rather than gathering it, you should contact us! Oxylabs are always ready to help businesses with their data-gathering processes.



# Want to Know More?

If you would like to know more about any of the topics mentioned in this white paper or [learn about our products](#), please get in touch! Our team is ready to answer any of your questions and offer you the best solution for your business needs.

[Get in touch with Oxylabs](#)

## Our Mission

Our mission is to share all the know-how that we collected over the years in the industry in order to create the future where big data is accessible to all businesses. We seek to create a healthy environment for everyone to grow and thrive in.

## Our Values

As a leading company in the proxy and web scraping industry, we ensure that the highest standards of business ethics lead all our operations. Our core values guide us toward achieving our mission. [Learn more](#)