Web Scraping at Scale: Practice and Theory

Introduction

Web scraping and large-scale data acquisition are quickly becoming the primary drivers of the business decision-making process. Instead of relying on intuition or previous experience, companies can now base their decisions on large swaths of fresh, high-quality public data.

As an increasing number of businesses are getting involved in large scale data acquisition, staying competitive and profitable means gaining access to new data around the clock. These businesses build next-generation tools, create actionable insights and deliver a better service or product to their customers. Those who do not utilize web scraping will quickly be outshined by competitors in several areas.

Web scraping, as a process, can seem relatively simple. An application goes through a list of relevant websites and collects data. It is then stored, parsed, prepared for analysis, and the entire process is repeated continuously.

Yet, most websites aren't too keen on letting any automated script run amok. Anti-bot measures are often implemented to reduce server strain, and the negative impact web scraping might have on user experience. These issues compound and multiply as projects become greater in scale.

On the other hand, those who run web scraping projects would like to acquire as much data as possible while still doing it reliably. As such, web scraping becomes a balancing act between avoiding anti-bot measures, efficiently browsing the same website numerous times, and circumventing any possible negative user experience.

Understanding and getting involved in large-scale data acquisition can reveal previously undetected insights, drive business growth, and create new opportunities. Yesterday was the best time to get started in web scraping. The second best time is now.

Building web scrapers **3**
Python Web Scraping Tutorial: Step-By-Step 4
Scraping Images From a Website with Python 15
JavaScript Web Scraping Tutorial 22
Web scraping libraries **32**
Python Requests Library 33
XML Processing and Web Scraping With lxml 39
Using Python and Beautiful Soup to Parse Data: Intro Tutorial 47
Web Scraping With Selenium: DIY or Buy? 53
Optimization strategies **58**
5 Key HTTP Headers for Web Scraping 59
Web Scraping Best Practices 62
Setting the Right Approach to Web Scraping 68
Industry trends **70**
5 Great Web Scraping Use Cases to Gain a Competitive Advantage 71
Using Web Scraping for Lead Generation 74
The Legal Framework of Data Scraping 76

Building web scrapers

Python Web Scraping Tutorial: Step-By-Step

Getting started with web scraping is simple except when it isn't which is why you are here. Python web scraping 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 Python web scraping tool an absolute breeze.

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 for Python 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.

Prepwork

Throughout this entire web scraping tutorial we will be using any 3.4+ version of Python. 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 Python web scraping. We'll be using three important libraries – BeautifulSoup v4, Pandas, and Selenium.

- BeautifulSoup is widely used to parse the HTML files
- Pandas is used to create structured data
- Selenium provides browser automation

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 assumes 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 to code in

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(r'c:\path\to\windows\webdriver\executable.exe')
```

Or

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

Picking a URL

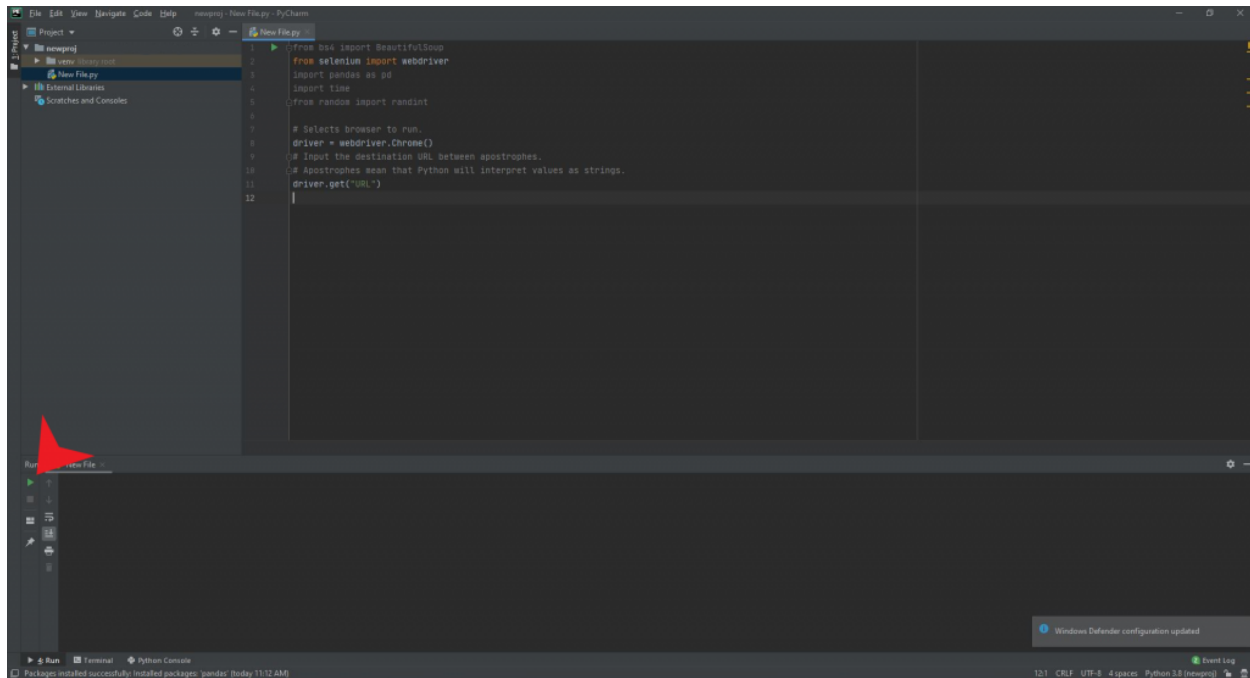
Before performing our first test run, choose a URL. As this web scraping tutorial is intended to create a very basic application, it's highly recommended that the target URL isn't too complicated:

- 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 right.



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('/nix/path/to/webdriver/executable')
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 the data

We have finally arrived at the fun and difficult part - extract 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('/nix/path/to/webdriver/executable')
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. We 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 we 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('/nix/path/to/webdriver/executable')
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('/nix/path/to/webdriver/executable')
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('/nix/path/to/webdriver/executable')
driver.get('https://your.url/here?yes=brilliant')
```

```
results = []
other_results = []
```

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:

```
for b in soup.findAll(attrs={'class': 'otherclass'}):
# Assume that data is nested in 'span'.
    name2 = b.find('span')
    other_results.append(name.text)
```

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

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

So far our 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('/nix/path/to/webdriver/executable')
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)
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('/nix/path/to/webdriver/executable')
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.

To infinity and beyond

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.

- Finally, integrate proxies into your web scraper. Using location specific request sources allows you to acquire data that might otherwise be inaccessible.

Scraping Images From a Website with Python

We highly recommend reading "Python Web Scraping Tutorial: Step-By-Step" before moving forward. Understanding how to build a basic data extraction tool will make creating a Python image scraper significantly easier. Additionally, we will use parts of code we had written previously as a foundation to download image links. Finally, we will use both Selenium and the requests library for learning purposes.

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/executable')
```

```
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 will 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 will reuse the same code we used in the "Python Web Scraping Tutorial: Step-by-Step" article 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 is None:
                print("Error: No matching values found")
                quit()
        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/executable')
driver.get('https://your.url/here?yes=brilliant')
results = []
content = driver.page_source
soup = BeautifulSoup(content)

def g_url(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))

g_url("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/executable')
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.sha1(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")
```

```python
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__":

  main()
```

Wrapping up

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. Upgrading an image scraper can be done in a variety of ways, most of which we outlined in the previous installment.

JavaScript Web Scraping Tutorial

With the arrival of Node.js, JavaScript has evolved into a very powerful language for web scraping. Node.js, sometimes written as Node js or even nodejs, is the engine that runs the JavaScript code without a browser. Additionally, npm, or Node.js Package Manager has a massive collection of libraries, which make web scraping in node.js very easy. Web scraping with JavaScript and Node.js is not only easy, it is fast, and the learning curve is very low for those who are already familiar with JavaScript.

This guide assumes at least a basic understanding of JavaScript. Familiarity with Chrome or Firefox Developer tools would also help and some knowledge of jQuery or CSS Selectors is essential.

Required software

There are only two pieces of software that will be needed:

1. Node.js (which comes with npm—the package manager for Node.js)
2. Any code editor

The only thing that you need to know about Node.js is that it is a runtime framework. This simply means that JavaScript code, which typically runs in a browser, can run without a browser. Node.js is available for Windows, Mac OS, and Linux. It can be downloaded at the official download page.

Set up Node.js project

Before writing any code to web scrape using node js, create a folder where JavaScript files will be stored. These files will contain all the code required for web scraping.

Once the folder is created, navigate to this folder and run the initialization command:

```
npm init -y
```

This will create a package.json file in the directory. This file will contain information about the packages that are installed in this folder. The next step is to install the Node.js Packages that will be discussed in the next section.

Node.js packages

For node.js web scraping, we need to use certain packages, also known as libraries. These libraries are prepackaged code that can be reused. The packages can be downloaded and installed using the npm install command, which is the Node.js Package Manager.

To install any package, simply run npm install <package-name>. For example, to install the package axios, run this on your terminal:

```
npm install axios
```

This also supports installing multiple packages. Run the following command to install all the packages used in this tutorial:

```
npm install axios cheerio json2csv
```

This command will download the packages in node_modules directory and update the package.json file.

Basics steps of web scraping with JavaScript

Almost every web scraping project using Node.js or JavaScript would involve three basics steps:

1. Sending the HTTP request
2. Parsing the HTTP response and extracting desired data
3. Saving the data in some persistent storage, e.g. file, database and similar

The following sections will demonstrate how Axios can be used to send HTTP requests, cheerio to parse the response and extract the specific information that is needed, and finally, save the extracted data to CSV using json2csv.

Sending HTTP request

The first step of web scraping with JavaScript is to find a package that can send HTTP requests and return the response. Even though request and request-promise have been quite popular in the past, these are now deprecated. You will still find many examples and old code using these packages. With millions of downloads every day, Axios is a good alternative. It fully supports Promise syntax as well as async-await syntax.

Parsing the response – Cheerio

Node.js provides another useful package, Cheerio. This package is useful because it converts the raw HTML captured by Axios into something that can be queried using a jQuery-like syntax.

JavaScript developers are usually familiar with jQuery. This makes Cheerio a very good choice to extract the information from HTML.

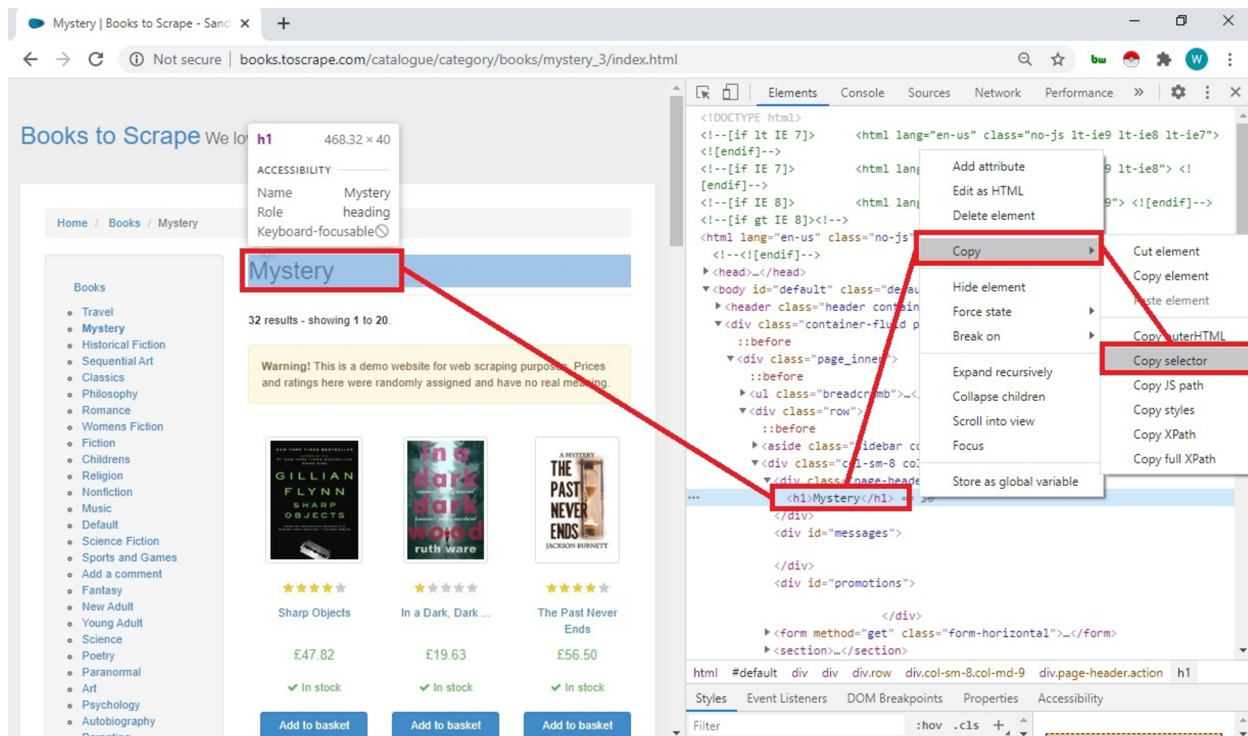
JavaScript web scraping – a practical example

One of the most common scenarios of web scraping with JavaScript is to scrape ecommerce stores. A good place to start is a fictional book store http://books.toscrape.com/. This site is very much like a real store, except that this is fictional and is made to learn web scraping.

Creating selectors

The first step before beginning JavaScript web scraping is creating selectors. The purpose of selectors is to identify the specific element to be queried.

Begin by opening the URL http://books.toscrape.com/catalogue/category/books/mystery_3/index.html in Chrome or Firefox. Once the page loads, right-click on the title of the genre, Mystery, and select Inspect. This should open the Developer Tools with <h1>Mystery</h1> selected in the Elements tab.



The simplest way to create a selector is to right-click this h1 tag in the Developer Tools, point to Copy, and then click Copy Selector. This will create a selector like this:

```
#content_inner > article > div.row > div.col-sm-6.product_main > h1
```

This selector is valid and works well. The only problem is that this method creates a long selector. This makes it difficult to understand and maintain the code.

After spending some time with the page, it becomes clear that there is only one h1 tag on the page. This makes it very easy to create a very short selector:

```
h1
```

Alternatively, a third-party tool like Selector Gadget extension for Chrome can be used to create selectors very quickly. This is a useful tool for web scraping in JavaScript.

Note that while this works most of the time, there will be cases where it doesn't work. Understanding how CSS selectors work is always a good idea. W3Schools has a good CSS Reference page.

Scraping the genre

The first step is to define the constants that will hold a reference to Axios and Cheerio.

```
const cheerio = require("cheerio");
const axios = require("axios");
```

The address of the page that is being scraped is saved in the variable URL for readability

```
const url =
"http://books.toscrape.com/catalogue/category/books/mystery_3/index.html";
```

Axios has a method get() that will send an HTTP GET request. Note that this is asynchronous method and thus needs await prefix:

```
const response = await axios.get(url);
```

If there is a need to pass additional headers, for example, User-Agent, this can be sent as the second parameter:

```
const response = await axios.get(url, {

  headers:

  {

    "User-Agent": "custom-user-agent string",

  }

});
```

This particular site does not need any special header, which makes it easier to learn.

Axios supports both the Promise pattern and the async-await pattern. This tutorial focuses on the async-await pattern. The response has a few attributes like headers, data, etc. The HTML that we

want is in the data attribute. This HTML can be loaded into an object that can be queried, using cheerio.load() method.

```
const $ = cheerio.load(response.data);
```

Cheerio's load () method returns a reference to the document, which can be stored in a constant. This can have any name. To make our web scraping code look and feel more like jQuery a $ can be used instead of a name.

Finding this specific element within the document is as easy as writing $("<selector here>"). In this particular case, it would be $("h1").

The method text() will be used everywhere when writing web scraping code with JavaScript, as it can be used to get the text inside any element. This can be extracted and saved in a local variable.

```
const genre = $("h1").text();
```

Finally, console.log() will simply print the variable value on the console.

```
console.log(genre);
```

To handle errors, the code will be surrounded by a try-catch block. Note that it is a good practice to use console.error for errors and console.log for other messages.

Here is the complete code put together. Save it as genre.js in the folder created earlier, where the command npm init was run.

```
const cheerio = require("cheerio");
const axios = require("axios");
const url =
"http://books.toscrape.com/catalogue/category/books/mystery_3/index.html";

async function getGenre() {

  try {

  const response = await axios.get(url);
  const document = cheerio.load(response.data);
  const genre = document("h1").text();
  console.log(genre);

  } catch (error) {

  console.error(error);
```

```
  }

}

getGenre();
```

The final step to run this web scraping in JavaScript is to run it using Node.js. Open the terminal and run this command:

```
node genre.js
```

The output of this code is going to be the genre name:

```
Mystery
```

Congratulations! This was the first program that uses JavaScript and Node.js for web scraping. Time to do more complex things!

Scraping book listings

Let's try scraping listings. Here is the same page that has a book listing of the Mystery genre – http://books.toscrape.com/catalogue/category/books/mystery_3/index.html

First step is to analyze the page and understand the HTML structure. Load this page in Chrome, press F12, and examine the elements.

Each book is wrapped in <article> tag. It means that all these books can be extracted and a loop can be run to extract individual book details. If the HTML is parsed with Cheerio, jQuery function each() can be used to run a loop. Let's start with extracting the title of all the books. Here is the code:

```
const books = $("article"); //Selector to get all books
books.each(function ()

           { //running a loop

    title = $(this).find("h3 a").text(); //extracting book title
    console.log(title);//print the book title

      });
```

As it is evident from the above code that the extracted details need to be saved somewhere else inside the loop. The best idea would be to store these values in an array. In fact, other attributes of the books can be extracted and stored as a JSON in an array.

Here is the complete code. Create a new file, paste this code and save it as books.js in the same folder that where npm init was run:

```
const cheerio = require("cheerio");
const axios = require("axios");
const mystery =
"http://books.toscrape.com/catalogue/category/books/mystery_3/index.html";
const books_data = [];

async function getBooks(url) {

  try {

    const response = await axios.get(url);
    const $ = cheerio.load(response.data);
    const books = $("article");

    books.each(function () {

      title = $(this).find("h3 a").text();
      price = $(this).find(".price_color").text();
      stock = $(this).find(".availability").text().trim();
      books_data.push({ title, price, stock }); //store in array

    });

    console.log(books_data);//print the array

  } catch (err) {

    console.error(err);

  }

}

getBooks(mystery);
```

Run this file using Node.js from the terminal:

```
node books.js
```

This should print the array of books on the console. The only limitation of this JavaScript code is that it is scraping only one page. The next section will cover how pagination can be handled.

Handling pagination

The listings like this are usually spread over multiple pages. While every site may have its own way of paginating, the most common one is having a next button on every page. The exception is the last, which will not have a next page link.

The pagination logic for these situations is rather simple. Create a selector for the next page link. If the selector results in a value, take the href attribute value and call getBooks function with this new URL recursively.

Immediate after the books.each() loop, add these lines:

```
if ($(".next a").length > 0) {

  next_page = baseUrl + $(".next a").attr("href"); //converting to absolute URL
  getBooks(next_page); //recursive call to the same function with new URL

}
```

Note that the href returned above is a relative URL. To convert it into an absolute URL, the simplest way is to concatenate a fixed part to it. This fixed part of the URL is stored in the baseUrl variable

```
const baseUrl ="http://books.toscrape.com/catalogue/category/books/mystery_3/"
```

Once the scraper reaches the last page, the Next button will not be there and the recursive call will stop. At this point, the array will have book information from all the pages. The final step of web scraping with Node.js is to save the data.

Saving scraped data to CSV

If web scraping with JavaScript is easy, saving data into a CSV file is even easier. It can be done using these two packages —fs and json2csv. The file system is represented by the package fs, which is in-built. json2csv would need to be installed using npm install json2csv command

```
npm install json2csv
```

after the installation, create a constant that will store this package's Parser.

```
const j2cp = require("json2csv").Parser;
```

The access to the file system is needed to write the file on disk. For this, initialize the fs package.

```
const fs = require("fs");
```

Find the line in the code where an array with all the scraped is available, and then insert the following lines of code to create the CSV file.

```
const parser = new j2cp();
const csv = parser.parse(books_data); // json to CSV in memory
fs.writeFileSync("./books.csv", csv); // CSV is now written to disk
```

Here is the complete script put together. This can be saved as a .js file in the node.js project folder. Once it is run using the node command on the terminal, data from all the pages will be available in books.csv file.

```
const fs = require("fs");
const j2cp = require("json2csv").Parser;
const axios = require("axios");
const cheerio = require("cheerio");
const mystery =
"http://books.toscrape.com/catalogue/category/books/mystery_3/index.html";
const books_data = [];


async function getBooks(url) {

  try {

  const response = await axios.get(url);
  const $ = cheerio.load(response.data);
  const books = $("article");

  books.each(function () {

    title = $(this).find("h3 a").text();
    price = $(this).find(".price_color").text();
    stock = $(this).find(".availability").text().trim();
    books_data.push({ title, price, stock });

  });

  // console.log(books_data);

  const baseUrl = "http://books.toscrape.com/catalogue/category/books/mystery_3/";

  if ($(".next a").length > 0) {
```

```
    next = baseUrl + $(".next a").attr("href");
    getBooks(next);

  } else {

    const parser = new j2cp();
    const csv = parser.parse(books_data);
    fs.writeFileSync("./books.csv", csv);

  }

  } catch (err) {

  console.error(err);

  }

}

getBooks(mystery);
```

Run this file using Node.js from the terminal:

```
node books.js
```

We now have a new file books.csv, which will contain all the desired data. This can be viewed using any spreadsheet program such as Microsoft Excel.

Summary

This whole exercise of web scraping using JavaScript and Node.js can be broken down into three steps — send the request, parse and query the response, and save the data. For all three steps, there are many packages available.

Web scraping libraries

Python Requests Library

Requests library is widely used in Python to send HTTP requests. It's employed as a third-party alternative to the standard "urllib", "urllib2", and "urllib3" as they can be confusing and often need to be used together. Requests in Python greatly simplifies the process of sending HTTP requests to their destination.

Learning to send requests in Python is a part of any budding developer's journey. In this Python requests tutorial, we will outline the grounding principles, the basic and some advanced uses of the Requests Python library.

Requests development philosophy

Requests is a library that strives to be as easy to use and parse as possible. Standard Python HTTP libraries are difficult to use, parse and often require significantly more statements to do the same thing. Let's take a look at a Urllib3 and a Python requests example:

Urllib3:

```
#!/usr/bin/env python# -*- coding: utf-8 -*-
import urllib3

http = urllib3.PoolManager()
gh_url = 'https://api.github.com'
headers = urllib3.util.make_headers(user_agent= 'my-agent/1.0.1',
basic_auth='abc:xyz')
requ = http.request('GET', gh_url, headers=headers)
print (requ.headers)
print(requ.data)
# ------# 200# 'application/json'
```

Requests:

```
#!/usr/bin/env python# -*- coding: utf-8 -*-
import requests
r = requests.get('https://api.github.com', auth=('user', 'pass'))
print r.status_codeprint r.headers['content-type']
# ------# 200# 'application/json'
```

Not only does Python Requests library reduce the amount of statements needed but it also makes the code significantly easier to understand and debug even for the untrained eye.

As it can be seen, the Requests library is notably more efficient than any standard Python library and that is no accident. Requests have been and are being developed with several PEP 20 (The Zen of Python) idioms in mind:

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Readability counts.

These five idioms form the foundation of the ongoing Requests library development and any new contribution should conform to the principles listed above.

Getting started with Requests

The Python library "Requests" isn't a part of the Python Standard Library, therefore it needs to be downloaded and installed. Installing Requests is simple as it can be done through a terminal.

```
$ pip install requests
```

We recommend using the terminal provided in the coding environment (e.g. PyCharm) as it will ensure that the library will be installed without any issues.

Finally, before beginning to use Requests in any project, the library needs to be imported:

```
import requests
```

GET requests

Out of all the possible HTTP requests, GET is the most commonly used. GET, as the name indicates, is an attempt to acquire data from a specified source (usually, a website). In order to send a GET request, invoke requests.get() and add a destination URL, e.g.:

```
import requests

requests.get('http://httpbin.org/')
```

Sending the GET request through Python's console will return a <Response [200]> message. A 200 response is 'OK' showing that the request has been successful. Response messages can also be viewed by creating an object and print(object.status_code). There are many more status codes and several of the most commonly encountered are:

- 200 – 'OK'
- 400 – 'Bad request' is sent when the server cannot understand the request sent by the client. Generally, this indicates a malformed request syntax, invalid request message framing, etc.
- 401 – 'Unauthorized' is sent whenever fulfilling the requests requires supplying valid credentials.

- 403 – 'Forbidden' means that the server understood the request but will not fulfill it. In cases where credentials were provided, 403 would mean that the account in question does not have sufficient permissions to view the content.
- 404 – 'Not found' means that the server found no content matching the Request-URI. Sometimes 404 is used to mask 403 responses when the server does not want to reveal reasons for refusing the request.

GET requests can be sent with specific parameters if required. Parameters follow the same logic as if one were to construct a URL by hand. Each parameter is sent after a question mark added to the original URL and pairs are split by the ampersand (&) sign:

```
payload = {'key1': 'value1', 'key2': 'value2'}
requests.get('http://httpbin.org/', params=payload)
```

Our URL would now be formed as:

```
https://httpbin.org/get?key2=value2&key1=value1
```

Yet while useful, status codes by themselves do not reveal much about the content acquired. So far, we only know if the acquisition was successful or not, and if not, for what possible reason.

Reading responses

In order to view what has been acquired by a GET request in Python, we should create an object. For the sake of simplicity, let's name it 'response':

```
response = requests.get('http://httpbin.org/')
```

We can now access the status code without using the console. In order to do so we will need to print out a specific section (status_code):

```
print(response.status_code)
```

So far the output will be identical to the one received before – <Response [200]>. Note that status codes in the Request library in Python have boolean values assigned to them (200 up to 400 is True, 400 and above is False). Using response codes as boolean values can be useful for several reasons such as checking whether the response was successful in general before continuing to perform other actions on the response.

In order to read the content of the response, we need to access the text part by using response.text. Printing the output will provide the entire response into the Python debugger window.

```
print(response.text)
```

Requests automatically attempts to make an educated guess about the encoding based on the HTTP header, therefore providing a value is unnecessary. In rare cases, changing the encoding may be needed and it can be done by specifying a value to response.encoding. Our specified value will then be used whenever we make a call.

Responses can also be decoded to the JSON format. HTTPbin doesn't send a request that can be decoded into JSON. Attempting to do so will raise an exception. For explanatory purposes, let's use Github's API:

```
response = requests.get('http://api.github.com')
print(response.json())
```

Using .json() returns a dictionary object that can be accessed and searched.

Using Python Requests headers

Response headers are another important part of the request. While they do not contain any content of the original message, headers hold many important details of the response such as information about the server, the date, encoding, etc. Every detail can be acquired from the initial response by making a call:

```
print(response.headers)
```

As with the .json() call, headers create a dictionary type object which can then be accessed. Adding parameters to the call will list out a part of the response, e.g.:

```
print(response.headers['Date'])
```

Our function will now print the date stored in the response header. Values are considered case-insensitive, therefore Requests will output the same result regardless of whether the parameter was formed as 'date' or 'Date'.

Requests can also send custom headers. Dictionary-type objects are used yet again, although this time they have to be created. Headers are passed in an identical manner to parameters. To check whether our request header has been sent successfully we will need to make the call response.request.headers:

```
import requests

headers = {'user-agent': 'my-agent/1.0.1'}
response = requests.get('http://httpbin.org/', headers=headers)
print(response.request.headers)
```

Running our code should output the request header in the debugger window with the user agent stated as 'my-agent/1.0.1'. As a general rule, sending well-known user agents is recommended as otherwise some websites could return a 403 'Forbidden' response.

Custom HTTP headers are usually used for troubleshooting or informational purposes. User agents are often utilized in web scraping projects in order to change the perceived source of incoming requests.

POSTing

POST requests are the second most common HTTP method. They are used to create a resource on a server with specified data. Sending a POST request is almost as simple as sending a GET:

```
response = requests.post('https://httpbin.org/post', data = {'key':'value'})
```

Of course, all HTTP methods (HEAD is an exception) return a response body which can be read. Responses to POST requests can be read in the same manner as GET (or any other method):

```
print(response.text)
```

Responses, rather obviously, in the relation to the types of requests made. For example, a POST request response contains information regarding the data sent to the server.

In most cases, specifying the data in the POST request might not be enough. Requests library accepts arguments from dictionary objects which can be utilized to send more advanced data:

```
payload = {'key1': 'value1', 'key2': 'value2'}
response = requests.post('https://httpbin.org/post', data = payload)
```

Our new request would send the payload object to the destination server. At times, sending JSON POST requests can be necessary. Requests have an added feature that automatically converts the POST request data into JSON.

```
import requests

payload = {'key1': 'value1', 'key2': 'value2'}
response = requests.post('https://httpbin.org/post', json = payload)
print(response.json())
```

Alternatively, the json library might be used to convert dictionaries into JSON objects. A new import will be required to change the object type:

```
import json
import requests

payload = {
   'key1': 'value1',
   'key2': 'value2'}
```

```
jsonData = json.dumps(payload)
response = requests.post('https://httpbin.org/post', data = jsonData)
print(response.json())
```

Note that the "json" argument is overridden if either "data" or "files" is used. Requests will only accept one of the three in a single POST.

Other HTTP methods

POST and GET are the two most common methods used by the average user. For example, Real-Time Crawler users utilize only these two HTTP methods in order to send job requests (POST) and receive data (GET). Yet, there are many more ways to interact with servers over HTTP.

- PUT – replaces all the current representations of the target resource with the uploaded content.
- DELETE – removes all the current representations of the target resource given by URI.
- HEAD – similar to GET, but it transfers the status and header section only.
- OPTIONS – describes the communication options for the target resource.
- TRACE – echoes the original request message back to its source.
- PATCH – applies modifications to a specified resource.

All the HTTP methods listed above are rarely used outside of server administration, web development and debugging. An average internet user will not have the required permissions to perform actions such as DELETE or PUT on nearly any website. Other HTTP methods are mostly useful for testing websites, something that is quite often outside the field of interest of the average internet user.

Conclusion

Python Requests library is both an incredibly powerful and easy to use tool that can be utilized to send HTTP requests. Understanding the basics is often enough to create simple applications or scripts.

XML Processing and Web Scraping With lxml

In this Python lxml tutorial, we will explore the lxml library. We will go through the basics of creating XML documents and then jump onto processing XML and HTML documents. Finally, we will put together all the pieces and see examples of web scraping with lxml. Each step of this tutorial is complete with practical Python lxml examples.

Prerequisites

This tutorial is aimed at developers who have at least a basic understanding of Python. A basic understanding of XML and HTML is also required. Simply put, if you know what an attribute is in XML, that is enough to understand this article.

This tutorial uses Python 3 code snippets but everything works on Python 2 with minimal changes as well.

What is lxml in Python

lxml is one of the fastest and feature-rich libraries for processing XML and HTML in Python. This library is essentially a wrapper over C libraries libxml2 and libxslt. This combines the speed of the native C library and the simplicity of Python.

Using Python's lxml library, XML and HTML documents can be created, parsed, and queried. It is a dependency on many of the other complex packages like Scrapy.

Installation

The best way to download and install the lxml library is from Python Package Index (PyPI). If you are on Linux (debian-based), simply run:

```
sudo apt-get install python3-lxml
```

Another way is to use the pip package manager. This works on Windows, Mac, and Linux:

```
pip3 install lxml
```

On windows, just use pip install lxml, assuming you are running Python 3.

Creating a simple XML document

Any XML or any XML compliant HTML can be visualized as a tree. A tree has a root and branches. Each branch optionally may have further branches. All these branches and the root are represented as an Element.

A very simple XML document would look like this:

```
<root>

    <branch>

        <branch_one>
        </branch_one>
        <branch_one>
        </branch_one>

    </branch>

</root>
```

If an HTML is XML compliant, it will follow the same concept.

Note that HTML may or may not be XML compliant. For example, if an HTML has <br> without a corresponding closing tag, it is still valid HTML, but it will not be a valid XML. In the later part of this tutorial, we will see how these cases can be handled. For now, let's focus on XML compliant HTML.

The Element class

To create an XML document using python lxml, the first step is to import the etree module of lxml:

```
>>> from lxml import etree
```

Every XML document begins with the root element. This can be created using the Element type. The Element type is a flexible container object which can store hierarchical data. This can be described as a cross between a dictionary and a list.

In this python lxml example, the objective is to create an HTML, which is XML compliant. It means that the root element will have its name as html:

```
>>> root = etree.Element("html")
```

Similarly, every html will have a head and a body:

```
>>> head = etree.Element("head")

>>> body = etree.Element("body")
```

To create parent-child relationships, we can simply use the append() method.

```
>>> root.append(head)

>>> root.append(body)
```

This document can be serialized and printed to the terminal with the help of tostring() function. This function expects one mandatory argument, which is the root of the document. We can optionally set pretty_print to True to make the output more readable. Note that tostring() serializer actually returns bytes. This can be converted to string by calling decode():

```
>>> print(etree.tostring(root, pretty_print=True).decode())
```

The SubElement class

Creating an Element object and calling the append() function can make the code messy and unreadable. The easiest way is to use the SubElement type. Its constructor takes two arguments – the parent node and the element name. Using SubElement, the following two lines of code can be replaced by just one.

```
body = etree.Element("body")
root.append(body)
# is same as
body = etree.SubElement(root,"body")
```

Setting text and attributes

Setting text is very easy with the lxml library. Every instance of the Element and SubElement exposes two methods – text and set, the former is used to specify the text and later is used to set the attributes. Here are the examples:

```
para = etree.SubElement(body, "p")
para.text="Hello World!"
```

Similarly, attributes can be set using key-value convention:

```
para.set("style", "font-size:20pt")
```

One thing to note here is that the attribute can be passed in the constructor of SubElement:

```
para = etree.SubElement(body, "p", style="font-size:20pt", id="firstPara")
para.text = "Hello World!"
```

The benefit of this approach is saving lines of code and clarity. Here is the complete code. Save it in a python file and run it. It will print an HTML which is also a well-formed XML.

```
from lxml import etree
```

```
root = etree.Element("html")
head = etree.SubElement(root, "head")
title = etree.SubElement(head, "title")
title.text = "This is Page Title"
body = etree.SubElement(root, "body")
heading = etree.SubElement(body, "h1", style="font-size:20pt", id="head")
heading.text = "Hello World!"
para = etree.SubElement(body, "p", id="firstPara")
para.text = "This HTML is XML Compliant!"
para = etree.SubElement(body, "p", id="secondPara")
para.text = "This is the second paragraph."

etree.dump(root)  # prints everything to console. Use for debug only
```

Note that here we used etree.dump() instead of calling etree.tostring(). The difference is that dump() simply writes everything to the console and doesn't return anything, tostring() is used for serialization and returns a string which you can store in a variable or write to a file. dump() is good for debug only and should not be used for any other purpose.

Add the following lines at the bottom of the snippet and run it again:

```
with open('input.html', 'wb') as f:
   f.write(etree.tostring(root, pretty_print=True))
```

This will save the contents to input.html in the same folder you were running the script. Again, this is a well-formed XML, which can be interpreted as XML or HTML.

Parsing XML files

The previous section was a Python lxml tutorial on creating XML files. In this section, we will look at traversing and manipulating an existing XML document using the lxml library.

Before we move on, save the following snippet as input.html.

```
<html>

 <head>

   <title>This is Page Title</title>

 </head>

 <body>

   <h1 style="font-size:20pt" id="head">Hello World!</h1>

   <p id="firstPara">This HTML is XML Compliant!</p>
```

```
   <p id="secondPara">This is the second paragraph.</p>

 </body>

</html>
```

When an XML document is parsed, the result is an in-memory ElementTree object.

The raw XML contents can be in a file system or a string. If it is in a file system, it can be loaded using the parse method. Note that the parse method will return an object of type ElementTree. To get the root element, simply call the getroot() method.

```
from lxml import etree

tree = etree.parse('input.html')
elem = tree.getroot()
etree.dump(elem)  # prints file contents to console
```

The lxml.etree module exposes another method that can be used to parse contents from a valid xml string — fromstring()

```
xml = '<html><body>Hello</body></html>'
root = etree.fromstring(xml)
etree.dump(root)
```

One important difference to note here is that fromstring() method returns an object of element. There is no need to call getroot().

Finding elements in XML

Broadly, there are two ways of finding elements using the Python lxml library. The first is by using the Python lxml querying languages: XPath and ElementPath. For example, the following code will return the first paragraph element.

Note that the selector is very similar to XPath. Also note that the root element name was not used because elem contains the root of the XML tree.

```
tree = etree.parse('input.html')
elem = tree.getroot()
para = elem.find('body/p')
etree.dump(para)
# Output
# <p id="firstPara">This HTML is XML Compliant!</p>
```

Similarly, findall() will return a list of all the elements matching the selector.

```
elem = tree.getroot()
para = elem.findall('body/p')
for e in para:
   etree.dump(e)
# Outputs
# <p id="firstPara">This HTML is XML Compliant!</p>
# <p id="secondPara">This is the second paragraph.</p>
```

The second way of selecting the elements is by using XPath directly. This approach is easier to follow by developers who are familiar with XPath. Furthermore, XPath can be used to return the instance of the element, the text, or the value of any attribute using standard XPath syntax.

```
para = elem.xpath('//p/text()')
for e in para:
   print(e)


# Output
# This HTML is XML Compliant!
# This is the second paragraph.
```

Handling HTML with lxml.html

Throughout this article, we have been working with a well-formed HTML which is XML compliant. This will not be the case a lot of the time. For these scenarios, you can simply use lxml.html instead of lxml.etree.

Note that reading directly from a file is not supported. The file contents should be read in a string first. Here is the code to print all paragraphs from the same HTML file.

```
from lxml import html

with open('input.html') as f:
   html_string = f.read()
tree = html.fromstring(html_string)
para = tree.xpath('//p/text()')
for e in para:
   print(e)

# Output
# This HTML is XML Compliant!
# This is the second paragraph
```

Web scraping with lxml

Now that we know how to parse and find elements in XML and HTML, the only missing piece is getting the HTML of a web page.

For this, the 'requests' library is a great choice. It can be installed using the pip package manager:

```
pip install requests
```

Once the requests library is installed, HTML of any web page can be retrieved using a simple get() method. Here is an example.

```
import requests

response = requests.get('http://books.toscrape.com/')
print(response.text)
# prints source HTML
```

This can be combined with lxml to retrieve any data that is required.

Here is a quick example that prints a list of countries from Wikipedia:

```
import requests
from lxml import html

response =
requests.get('https://en.wikipedia.org/wiki/List_of_countries_by_population_in_2010')

tree = html.fromstring(response.text)
countries = tree.xpath('//span[@class="flagicon"]')
for country in countries:
   print(country.xpath('./following-sibling::a/text()')[0])
```

In this code, the HTML returned by response.text is parsed into the variable tree. This can be queried using standard XPath syntax. The XPaths can be concatenated. Note that the xpath() method returns a list and thus only the first item is taken in this code snippet.

This can easily be extended to read any attribute from the HTML. For example, the following modified code prints the country name and image URL of the flag.

```
for country in countries:
   flag = country.xpath('./img/@src')[0]
   country = country.xpath('./following-sibling::a/text()')[0]
   print(country, flag)
```

Conclusion

In this Python lxml tutorial, various aspects of XML and HTML handling using the lxml library have been introduced. Python lxml library is a light-weight, fast, and feature-rich library. This can be used to create XML documents, read existing documents, and find specific elements. This makes this library equally powerful for both XML and HTML documents. Combined with requests library, it can also be easily used for web scraping.

Using Python and Beautiful Soup to Parse Data: Intro Tutorial

Although web scraping in its totality is a complex and nuanced field of knowledge, building your own basic web scraper is not all that difficult. And that's mostly due to coding languages such as Python. This language makes the process much more straightforward thanks to its relative ease of use and the many useful libraries that it offers. One of these wildly popular libraries is Beautiful Soup, a Python package used for parsing HTML and XML documents. And that's exactly what we'll be focusing on in this tutorial.

This tutorial is useful for those seeking to quickly grasp the value that Python and Beautiful Soup v4 offers. After following the provided examples you should be able to understand the basic principles of how to parse HTML data. The examples will demonstrate traversing a document for HTML tags, printing the full content of the tags, finding elements by ID, extracting text from specified tags and exporting it to a .csv file.

Before getting to the matter at hand, let's first take a look at some of the fundamentals.

What is data parsing?

Data parsing is a process during which a piece of data gets converted into a different type of data according to specified criteria. It is an important part of web scraping since it helps transform raw HTML data into a more easily readable format that can be understood and analyzed.

A well-built parser will identify the needed HTML string and the relevant information within it. Based on predefined criteria and the rules of the parser, it will filter and combine the needed information into CSV or JSON files.

What is Beautiful Soup?

Beautiful Soup is a Python package for parsing HTML and XML documents. It creates a parse tree for parsed pages based on specific criteria that can be used to extract, navigate, search and modify data from HTML, which is mostly used for web scraping. It is available for Python 2.7 and Python 3. A useful library, it can save programmers loads of time.

Installing Beautiful Soup

Before working on this tutorial, you should have a Python programming environment set up on your machine. For this tutorial we will assume that PyCharm is used since it's a convenient choice even for the less experienced with Python and is a great starting point. Otherwise, simply use your go-to IDE.

On Windows, when installing Python make sure to tick the "PATH installation" checkbox. PATH installation adds executables to the default Windows Command Prompt executable search. Windows will then recognize commands like "pip" or "python" without having to point to the directory of the executable which makes things more convenient.

You should also have Beautiful Soup installed on your system. No matter the OS, you can easily do it by using this command on the terminal to install the current latest version of Beautiful Soup:

```
pip install BeautifulSoup4
```

If you are using Windows, it is recommended to run terminal as administrator to ensure that everything works out smoothly.

Finally, since we will be working with a sample file written in HTML, you should be at least somewhat familiar with HTML structure.

Getting started

A sample HTML file will help demonstrate the main methods of how Beautiful Soup parses data. This file is much more simple than your average modern website, however, it will be sufficient for the scope of this tutorial.

```
<!DOCTYPE html>
<html>
   <head>
       <title>What is a Proxy?</title>
       <meta charset="utf-8">
   </head>

   <body>
       <h2>Proxy types</h2>

       <p>
         There are many different ways to categorize proxies. However, two of
the most popular types are residential and data center proxies. Here is a list of the
most common types.
       </p>

       <ul id="proxytypes">
           <li>Residential proxies</li>
           <li>Datacenter proxies</li>
           <li>Shared proxies</li>
           <li>Semi-dedicated proxies</li>
           <li>Private proxies</li>
       </ul>
```

```
    </body>
</html>
```

For PyCharm to use this file, simply copy it to any text editor and save it with the .html extension to the directory of your PyCharm project.

Going further, open PyCharm and after a right click on the project area navigate to New -> Python File. Congratulations and welcome to your new playground!

Traversing for HTML tags

First, we can use Beautiful Soup to extract a list of all the tags used in our sample HTML file. For this, we will use the soup.descendants generator.

```
from bs4 import BeautifulSoup

with open('index.html', 'r') as f:
    contents = f.read()
    soup = BeautifulSoup(contents, features="html.parser")
    for child in soup.descendants:
        if child.name:
            print(child.name)
```

After running this code (right click on code and click "Run") you should get the below output:

```
html
head
title
meta
body
h2
p
ul
li
li
li
li
li
```

What just happened? Beautiful Soup traversed our HTML file and printed all the HTML tags that it has found sequentially. Let's take a quick look at what each line did.

```
from bs4 import BeautifulSoup
```

This tells Python to use the Beautiful Soup library.

```
with open('index.html', 'r') as f:
   contents = f.read()
```

And this code, as you could probably guess, gives an instruction to open our sample HTML file and read its contents.

```
soup = BeautifulSoup(contents, features="html.parser")
```

This line creates a BeautifulSoup object and passes it to Python's built in HTML parser. Other parsers, such as lxml, might also be used, but it is a separate external library and for the purpose of this tutorial the built-in parser will do just fine.

```
for child in soup.descendants:
       if child.name:
       print(child.name)
```

The final pieces of code, namely the soup.descendants generator, instruct Beautiful Soup to look for HTML tags and print them in the PyCharm console. The results can also easily be exported to a .csv file but we will get to this later.

Getting the full content of tags

To get the content of tags, this is what we can do:

```
from bs4 import BeautifulSoup

with open('index.html', 'r') as f:
   contents = f.read()
   soup = BeautifulSoup(contents, features="html.parser")
   print(soup.h2)
   print(soup.p)
   print(soup.li)
```

This is a simple instruction that outputs the HTML tag with its full content in the specified order. Here's what the output should look like:

```
<h2>Proxy types</h2>
<p>
         There are many different ways to categorize proxies.  However, two of the
most popular types are residential and data center proxies. Here is a list of the
most common types.
       </p>
<li>Residential proxies</li>
```

You could also remove the HTML tags and print text only, by using, for example:

```
print(soup.li.text)
```

Which in our case will give the following output: "Residential proxies"

Note that this only prints the first instance of the specified tag. Let's continue to see how to find elements by ID or using the find_all method to filter elements by specific criteria.

Using Beautiful Soup to find elements by ID

We can use two similar ways to find elements by ID:

```
print(soup.find('ul', attrs={'id': 'proxytypes'}))
```

or

```
print(soup.find('ul', id='proxytypes'))
```

Both of these will output the same result in the Python Console:

```
<ul id="proxytypes">
<li>Residential proxies</li>
<li>Datacenter proxies</li>
<li>Shared proxies</li>
<li>Semi-dedicated proxies</li>
<li>Private proxies</li>
</ul>
```

Finding all specified tags and extracting text

The find_all method is a great way to extract specific data from an HTML file. It accepts many criteria that make it a flexible tool allowing us to filter data in convenient ways. Yet for this tutorial we do not need anything more complex. Let's find all items of our list and print them as text only:

```
for tag in soup.find_all('li'):
   print(tag.text)
```

This is how the full code should look like:

```
from bs4 import BeautifulSoup

with open('index.html', 'r') as f:
   contents = f.read()
```

```
    soup = BeautifulSoup(contents, features="html.parser")
    for tag in soup.find_all('li'):
        print(tag.text)
```

And here's the output:

```
Residential proxies
Datacenter proxies
Shared proxies
Semi-dedicated proxies
Private proxies
```

Congratulations, you should now have a basic understanding of how Beautiful Soup might be used to parse data. It should be noted that the information presented in this article is useful as introductory material yet real-world web scraping with BeautifulSoup and the consequent parsing of data is usually much more complicated than this. For a more in-depth look at Beautiful Soup you will hardly find a better source than its documentation, so be sure to check it out too.

Exporting data to a .csv file

A very common real-world application would be exporting data to a .csv file for later analysis. Although this is outside the scope of this tutorial, let's take a quick look at how this might be achieved.

First, you would need to install the pandas library that helps Python create structured data. This can be easily done by using:

```
pip install pandas
```

You should also add this line to the beginning of your code to import the library:

```
import pandas as pd
```

Going further, let's add some lines that will export the list we extracted earlier to a .csv file. This is how our full code should look like:

```
from bs4 import BeautifulSoup
import pandas as pd

with open('index.html', 'r') as f:
    contents = f.read()

    soup = BeautifulSoup(contents, features="html.parser")
    results = soup.find_all('li')
```

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

What happened here? Let's take a look:

```
results = soup.find_all('li')
```

This line finds all instances of the <li> tag and stores it in the results object.

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

And here we see the pandas library at work, storing our results into a table (DataFrame) and exporting it to .csv.

If all went well, a new file titled names.csv should appear in the directory of your Python project and inside you should see a table with the proxy types list. That's it! You now not only know how extracting data from an HTML works but can also programmatically export it to a new file.

Conclusion

As you can see, Beautiful Soup is a greatly useful HTML parser. With a relatively low learning curve, you can quickly grasp how to navigate, search, and modify the parse tree. With the addition of libraries such as pandas you can further manipulate and analyze the data which offers a powerful package for a near infinite amount of data collection and analysis use cases.

Web Scraping With Selenium: DIY or Buy?

In order to understand the fundamentals of web scraping, it's important to learn how to leverage different frameworks and request libraries. By developing an understanding for various HTTP methods (mainly GET and POST) web scraping can become a lot easier.

For instance, Selenium is one of the better known and often used tools that help automate web browser interactions. By using it together with other technologies (e.g., BeautifulSoup), you can get a better grasp on web scraping basics.

How does Selenium work? It automates your written script processes, as the script needs to interact with a browser to perform repetitive tasks like clicking, scrolling, etc. As described on Selenium's official webpage, it is "primarily for automating web applications for testing purposes, but is certainly not limited to just that."

In this guide, on how to web scrape with Selenium, we will be using Python 3.x. as our main input language (as it is not only the most common scraping language but the one we closely work with as well).

Setting up Selenium

Firstly, to download the Selenium package, execute the pip command in your terminal:

```
pip install selenium
```

You will also need to install Selenium drivers, as it enables python to control the browser on OS-level interactions. This should be accessible via the PATH variable if doing a manual installation.

You can download the drivers for Firefox, Chrome, and Edge from here.

Quick starting Selenium

Let's begin the automatization by starting up your browser:

- Open up a new browser window (in this instance, Firefox)
- Load the page of your choice (our provided URL)

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('http://oxylabs.io/')
```

This will launch it in the headful mode. In order to run your browser in headless mode and run it on a server, it should look something like this:

```
from selenium import webdriver
from selenium.webdriver.firefox.options import Options

options = Options()
options.headless = True
options.add_argument("--window-size=1920,1200")

driver = webdriver.firefox(options=options, executable_path=DRIVER_PATH)
driver.get("https://www.oxylabs.io/")
print(driver.page_source)
driver.quit()
```

Data extraction with Selenium by locating elements

find_element

Selenium offers a variety of functions to help locate elements on a page:

- find_element_by_id
- find_element_by_name
- find_element_by_xpath
- find_element_by_link_text (find element by using text value)
- find_element_by_partial_link_text (find element by matching some part of a hyperlink text(anchor tag))
- find_element_by_tag_name
- find_element_by_class_name
- find_element_by_css_selector (find element by using a CSS selector for id class)

As an example, let's try and locate the H1 tag on oxylabs.io homepage with Selenium:

```
<html>
   <head>
       ... something
   </head>
   <body>
       <h1 class="someclass" id="greatID"> Partner Up With Proxy Experts</h1>
   </body>
</html>

h1 = driver.find_element_by_name('h1')
h1 = driver.find_element_by_class_name('someclass')
h1 = driver.find_element_by_xpath('//h1')
```

```
h1 = driver.find_element_by_id('greatID')
```

You can also use the find_elements (plural form) to return a list of elements. E.g.:

```
all_links = driver.find_elements_by_tag_name('a')
```

This way, you'll get all anchors in the page.

However, some elements are not easily accessible with an ID or a simple class. This is why you will need XPath.

XPath

XPath is a syntax language that helps find a specific object in DOM. XPath syntax finds the node from the root element either through an absolute path or by using a relative path. e.g.:

- / : Select node from the root. /html/body/div(1) will find the first div
- //: Select node from the current node no matter where they are. //form(1) will find the first form element
- (@attributename='value'): a predicate. It looks for a specific node or a node with a specific value.

Example:

```
//input[@name='email'] will find the first input element with the name "email".

<html>
<body>
  <div class = "content-login">
    <form id="loginForm">
        <div>
           <input type="text" name="email" value="Email Address:">
           <input type="password" name="password"value="Password:">
        </div>
       <button type="submit">Submit</button>
    </form>
  </div>
</body>
</html>
```

WebElement

WebElement in Selenium represents an HTML element. Here are the most commonly used actions:

- element.text (accessing text element)
- element.click() (clicking on the element)
- element.get_attribute(`class’) (accessing attribute)
- element.send_keys(`mypassword’) (sending text to an input)

Slow website render solutions

Some websites use a lot of JavaScript to render content, and they can be tricky to deal with as they use a lot of AJAX calls. There are a few ways to solve this:

- time.sleep(ARBITRARY_TIME)
- WebDriverWait()

Example:

```
try:
   element = WebDriverWait(driver, 10).until(
       EC.presence_of_element_located((By.ID, "mySuperId"))
   )
finally:
   driver.quit()
```

This will allow the located element to be loaded after 10 seconds. To dig deeper into this topic, go ahead and check out the official Selenium documentation.

Conclusion

Selenium is a great tool for web scraping, especially when learning the basics. But, depending on your goals, it is sometimes easier to choose an already-built tool that does web scraping for you. Building your own scraper is a long and resource-costly procedure that might not be worth the time and effort.

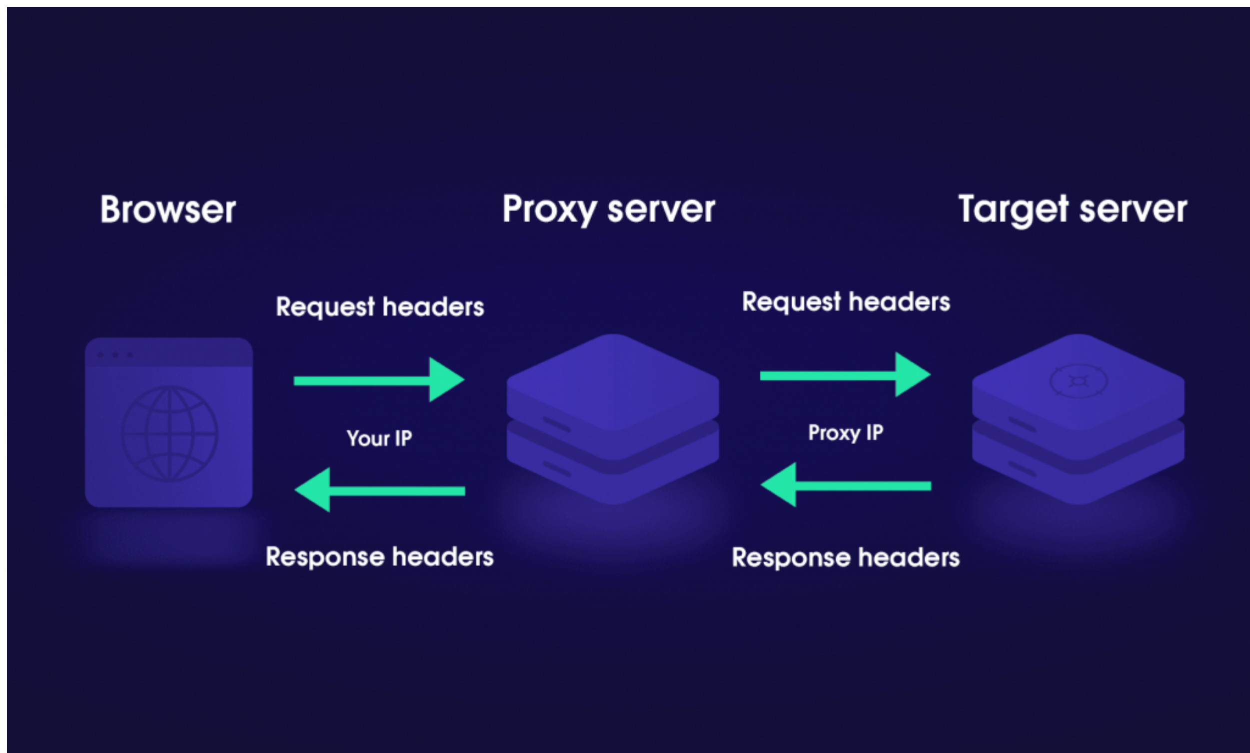
Optimization strategies

5 Key HTTP Headers for Web Scraping

One of the most important challenges of web scraping is to stay unblocked for as long as possible. Of course, there are proven resources and techniques, such as the use of a proxy or practicing IP rotation that will help your web scraper to avoid blocks.

However, another sometimes overlooked technique is to use and optimize HTTP headers. This practice will allow you to significantly decrease your web scraper's chances of getting blocked by various data sources, and also ensure that the retrieved data is of high quality.

In this article, we are revealing 5 most essential HTTP headers that need to be used and optimized, and provide you with the reasoning behind it.



User-Agent

The User-Agent request header passes information related to the identification of application type, operating system, software and its version, and allows for data target to decide what type of HTML layout to use in response i.e. mobile, tablet or pc, e.g.:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5)
AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/12.1.1 Safari/605.1.15
```

Authenticating User-Agent request header is a common practice by web servers, and it is the first check that allows data sources to identify suspicious requests. For instance, when web scraping is in process, numerous requests are traveling to the web server, and if User-Agent request headers are identical, it will seem as if it is a bot-like activity. Hence, experienced web scraping punters will manipulate and differentiate User-Agent header strings, which consequently allow portraying multiple organic users' sessions.

So, when it comes to User-Agent request header, remember to frequently alter the information this header carries, which will substantially reduce your odds of getting blocked.

Accept-Language

The Accept-Language request header passes information indicating to a web server which languages the client understands, and which particular language is preferred when the web server sends the response back, e.g. "en-gb".

It's worth mentioning that this particular header usually comes into play when web servers are unable to identify the preferred language e.g. via URL.

That said, the key with the Accept-Language request header is relevant. It is essential to ensure that set languages are in accordance with the data-target domain and client's IP location. Simply because, if requests from the same client would appear in multiple languages this would raise suspicions to the web server of bot-like behavior (non-organic request approach), and consequently, they might block the web scraping process.

Accept-Encoding

The Accept-Encoding request header notifies the web server what compression algorithm to use when the request is handled. In other words, it states that the required information can be compressed (if web server can handle it) when being sent out from the web server to the client, e.g. "br, gzip, deflate".

However, when optimized it allows saving traffic volume, which is a win-win situation for both the client and the web server from the traffic load perspective. The client still gets the required information (just compressed), and the web server isn't wasting its resources by transferring a huge load of traffic.

Accept

The Accept request header falls into a content negotiation category, and its purpose is to notify the web server on what type of data format can be returned to the client, e.g:

```
test/html,application/xhtml+xml,application/x
```

```
ml;q=0.9,*/*;q=0.8
```

It's as simple as it sounds, but a common hiccup with web scraping is overlooking or forgetting to configure the request header accordingly to the web server's accepted format. If the Accept request header is configured suitably, it will result in a more organic communication between the client and the server, and consequently, decrease web scraper's chances of getting blocked.

Referer

The Referer request header provides the previous web page's address before the request is sent to the web server, e.g.: "http://www.google.com/"

It might seem that the Referer request header has very little impact when it comes to blocking the scraping process, when in fact, it actually does. Think of a random organic user's internet usage patterns. This user is quite likely surfing the mighty internet and losing track of hours in a day. Hence, if you want to portray the web scraper's traffic to seem more organic, simply specify a random website before starting a web scraping session.

The key is not to jump the gun and instead take this rather straightforward step. Hence, remember to always set up the Referer request header, and boost your chances of slipping under anti-scraping measures implemented by web servers.

Wrapping it up

Now you know which HTTP headers for web scraping to configure, and by doing so, it will allow increasing your web scraper's chances of the successful and efficient data extraction operation.

It's safe to state that the more you know about the technical side of web scraping, the more fruitful your web scraping results will be. Use this knowledge wisely, and it's a given that your web scraper will work more effectively and efficiently.

Web Scraping Best Practices

Haphazardly acquiring, storing and analyzing data from a few pages will only work for small, short-term projects. Whenever consistent, long-term data acquisition is required web scraping best practices have to be employed.

Best web scraping practices can be separated into two large categories: ethical and technical. Optimizing the technical side of a web scraper isn't enough as there are legal and ethical considerations to take into account.

Ensuring that data acquisition activities do not disrupt the regular operations of web servers or break any laws associated with the content being scraped is vital. Whenever in doubt, consult your lawyers and keep in mind Google's old motto: "Do no evil".

Ethical best practices for web scraping

All of the web scraping tips outlined below will apply to any data acquisition case. While Python based web scrapers are the most common way of extracting data from websites, there are general guidelines that apply regardless of the foundation used.

Remember that some websites license their data. Sending a request for it to the owners might be an easier way over attempting to scrape it as some websites will be willing to provide their datasets.

Before engaging in any web scraping activity, the first step is to ensure that the soon-to-be-acquired data is public. Then, legal evaluation needs to be done for each case. While there is currently no industry-wide regulation, there is plenty of US and EU web scraping case law at. These should be used to evaluate the limits of a web scraping project. We believe that over the years some landmark cases have appeared:

- Feist Publications v. Rural Telephone Service (1991)
- eBay v. Bidder's Edge (2000)
- Power Ventures v. Facebook (2009)
- Craigslist v. 3taps Inc. (2013)
- QVC v. Resultly (2014)
- Ryanair cases (2014)
- HiQ labs v. LinkedIn (2019)

Read, respect and utilize robots.txt

Robots.txt files are used by web servers as directives for crawlers and scrapers (mostly Googlebot) on what they can and cannot access. Robots.txt is generally located in the root directory of a website (e.g. https://oxylabs.io/robots.txt).

Websites might have detailed rules on what is allowed, ranging from the frequency of scraping to disallowing access to specific pages. Luckily, most websites provide a reasonable set of rules that will grant an understanding of their web scraping guidelines.

In cases where such rules are absent, the responsibility lies upon the one attempting to scrape websites. Therefore, each case should be carefully assessed in order to minimize the impact on servers and remain as close as possible to the guidelines provided in robots.txt.

Scrape during off-peak hours

Outside of possible copyright infringements or otherwise illegally obtaining data, one of the worst things that can happen is server slowdown. Most scrapers and crawlers move through pages significantly faster than an average user as they don't actually read the content provided, only crawl the page and index it or download and save the data locally.

Thus, a single unrestrained web scraping tool will affect server load to a much greater extent than any regular internet user. In turn, scraping during high-load times might negatively impact user experience due to service slowdowns.

Scraping during off-peak hours will greatly reduce the negative impact a web scraper might have on the service. Finding the best time to crawl the website and acquire data will vary on a case-by-case basis but picking the hours just after midnight (localized to the service) is a good starting point.

Limit requests per second (and per day)

Continuing with the concern of overloading the servers, limiting both requests per second and the total amount per day is highly recommended. As mentioned previously, web scrapers can achieve incredible speeds that are much higher than the average internet user. Limiting how fast they move from page to page and download the content will help server administrators handle the load put on to their servers.

Limiting the amount of requests per day might be more useful for websites with more static information. As they are not prone to change, scanning them a few times every 24 hours should be more than enough. For websites with a lot of dynamic information (e.g. search engines or e-commerce pages), finding the sweet spot will be harder.

Respect intellectual property

Before engaging in any data acquisition activities, be sure that what you are downloading and storing isn't someone's intellectual property. If the acquired data will then be used in other sources or republished, intellectual property laws become even more important.

Thankfully, most websites store factual information which not always can be considered intellectual property. Although even in these cases checking in with your lawyers is highly recommended. Laws on data acquisition will differ between regions.

For example, database protection (consisting only of factual information) can be understood differently in EU and US law, therefore some scraping approaches might need to be changed.

Be transparent and responsible

Remember that data is a tool that can be used by well-meaning and ill-willed individuals. Do not share data sets freely as the information might be used by third parties with an impaired moral compass.

Additionally, being transparent will also help in resolving any possible issues regarding scraping. Always check whether the Terms of Service are applicable to you and whether a breach of conduct would happen due to web scraping activities.

Technical web scraping best practices

Technical best practices will be somewhat related to the way a web scraper is built. While the general principles remain, their implementation will slightly differ. As such, these guidelines will be written with Python web scraping best practices in mind. Some adjustments might need to be made for different languages or tools.

Caching

Caching web pages means saving bandwidth for both parties. Larger web scraping projects generally run many requests per second which can put exceptional strain on servers. If the web scraper in question runs through many pages which are not updated frequently, caching at least some of them is a worthwhile option as it will shrink the scraping path for some time.

Certain pages can be cached automatically and revisited later on. Revisiting rarely updated pages after a set amount of time will reduce the total amount of URLs visited per day. Thus, it becomes useful to both parties as a web scraper can perform its routine tasks faster while at the same time reduce the negative impact it has on servers.

Avoid image scraping

Image scraping might be required for some web scraping projects but it should generally be avoided. Images are data-heavy objects that can often be copyright protected meaning that not only it will take additional bandwidth and storage space but there's also a higher risk of infringing on someone else's rights.

Additionally, since images are data-heavy, they are often hidden in JavaScript elements (e.g. behind Lazy loading) which will significantly increase the complexity of the data acquisition process and slow down the web scraper itself. To get images out of JS elements, a more complicated scraping procedure (something that would force the website to load all content) would have to be written and employed.

Avoid JavaScript

Data nested in JavaScript elements is notoriously hard to acquire. Websites use many different JavaScript features to display content based on specific user actions. A common practice is to only display product images in search bars after the user has provided some input.

JavaScript can also cause a host of other issues – memory leaks, application instability or, at times, complete crashes. Dynamic features can often become a burden. Avoid JavaScript unless absolutely necessary.

Add timeouts

Adding timeouts is one of the most important python web scraping tips. For example, the Requests library has no default timeout time (or more specifically, they are infinite). A web scraper might get stuck forever if the server doesn't respond.

Setting the default timeout time to somewhere between 10 to 60 seconds will generally be enough to prevent any infinite hangs. Exact timing parameters will differ based on many factors, so experimenting with them is worthwhile.

Avoiding blocked IP addresses

A large part of web scraping is centered around avoiding IP address blocks. For some projects, data needs to be scraped often which can quickly trigger anti-bot algorithms and result in incomplete data sets or complete loss of access.

Regaining access to data generally requires the use of proxies or other workarounds. Therefore, maintaining an unblocked IP address for as long as possible is vital to efficient web scraping. Employing certain strategies can greatly reduce the likelihood of losing access to important data.

User agent rotation

User agents are a part of every HTTP request sent over. They are, essentially, a device identifying information that is provided to a website so that it can display its content correctly. Consistently changing user agents is a common practice in web scraping.

HTTP libraries send their own user agents that show the source of the requests (e.g. from Python's request library). Many websites automatically return a 403 Unauthorized response if they detect the default Requests library header. Therefore, changing user agents is one of the few vital web scraping Python tips out there.

Depending on the complexity of anti-bot algorithms, websites might be able to detect incomplete headers or faked user agents. Having a small collection of real user agents will help retain IP addresses for longer periods of time.

Crawling patterns

Regular internet users are somewhat random in their browsing activities – they might scroll certain distances, click on random (non)interactive objects, spend differing amounts of time reading the websites content etc. A web scraper without any crawling pattern will only go to necessary locations, download the content, and move on. Websites can easily detect such activity and flag the scraper as a bot.

Developing different crawling patterns for web scrapers requires thinking about user behavior and some practical data. A great starting point is to think how a regular user would browse the website and try to apply the same pattern to a web scraper. Specific crawling patterns may be applied to one user agent in order to maximize the resemblance to regular browsing behavior.

Before trying to create the best crawling pattern, remember that users browse certain categories of websites differently. For example, how an average internet user interacts with content on social media, e-commerce websites and search engines will vary wildly. Having a crawling pattern for each category instead of trying to apply a one-size-fits-all rule will greatly reduce the chances of getting an IP address blocked.

Curate the URL library

Web scrapers rely, in one way or another, on a URL library. Advanced scrapers might automatically acquire new links for more complex targets but they still need some pointers on where to begin their journey. Maintaining a clean URL library will help with both efficiency and blocked IP addresses.

Some websites host honeypot traps. They are generally links that are invisible to a human but are easily acquired by an automated crawler. Honeypots might instantly block the IP address,

display incorrect information, send the bot on an endless loop or flag the browser as a bot and perform other actions.

Additionally, removing dead URLs will greatly increase the speed of data acquisition. As the web scraper won't differentiate between live and dead URLs beforehand, visiting a 404 page might accidentally provide muddy data or just make the tool go to useless pages.

Use proxies

Proxies are an important line of defense as they can be used in a variety of ways to reduce the amount of blocked IP addresses. They can help by reducing the total amount of requests coming from one IP or by changing the address.

Most websites accept a set amount of requests per day or hour before they block the user. Often, these bans do not last too long due to the fact that most IP addresses can belong to different users over time but blocks can still disrupt web scraping operations. Proxies can reduce the total amount of outgoing HTTP requests by employing IP address rotation. Setting a maximum amount of requests before switching proxies is a common practice in web scraping.

Yet, even with all the best practices in the world, the possibility of getting an IP address blocked still remains. There is simply no way to completely and accurately predict all of the factors that might trigger a ban.

Therefore, using proxies as a way to change IPs is a vital part of any web scraping operation. Scraping without the use of proxies may mean stopping the entire data acquisition process for an extended period of time as the IP address used may get blocked.

Putting it all together

Before engaging in upgrades to your web scraper, remember that each best practice takes a lot of time and effort to develop into a smooth feature. Add each best practice one at a time and keep testing.

Even with the best developers, intentions and performance, new features will inevitably break. Finally, remember that neither efficiency nor block avoidance trump web scraping ethics. Always consult your lawyers before engaging in web scraping and don't sacrifice someone's work for a small increase in data acquisition.

Setting the Right Approach to Web Scraping

Recently, we hosted the first webinar about residential proxies usage mistakes and how to solve them. We shared our knowledge on how to start web scraping. In this article, we will specify our tips on setting the right approach to web scraping and what are the key elements for the best web scraping practice.

Successful web scraping

Just as with most data gathering tasks, getting started is the hardest part. To make it easier, follow these steps: set a preferred session, see if it works with a test query, and then start scraping your target website. Testing is an essential part because you can check if your web scraping will be successful, and make sure you will get the best results.

Sessions and their importance

Sessions are an essential part of the residential proxy network. They enable you to use the same IP address for multiple requests. By default, every new request that goes through the residential network is carried out by a new proxy and this can cause issues. For example, if you are using a full browser, bot, or a headless browser to download assets from your target websites, all of them must be downloaded using the same IP address. In this case, assets mean everything that comes with the HTML – CSS, JavaScript files, images, and so on.

Reliable proxy providers will offer you flexible and adjustable session control features, so you can be sure that this part will be managed easily.

HTTP headers for web scraping

HTTP abbreviation stands for HyperText Transfer Protocol, which manages how communication is transferred and structured on the internet. Also, HTTP is responsible for how web servers and browsers should respond to different requests. There are different types of HTTP headers: request header, response header, general HTTP header, entity header, and so on.

When web scraping, sending the HTTP headers, and preferably in the right order is the minimum these days. All the requests without specific HTTP headers are likely to be blocked very quickly. For successful web scraping, you should think of every possible way to avoid blocks. Optimizing HTTP headers reduces the chances of being blocked by data sources.

To start optimizing HTTP headers, we advise you to see how the browser works by itself. In Firefox or Chrome, hit the F12 button and open developer tools. Go to the Network tab and refresh a page you are on. You will see all requests that the browser had to make in order to fully render

the page. Find where the HTML content was loaded, and you will see what headers and in which order were sent. Try to make this happen on your scraper too.

“Fingerprinting” and its relevance

”Fingerprinting” is all the information that your browser gives websites about you and your computer, such as mouse input, resolution, installed plugins, and much more. Having all this information, you can make a single hash, a fingerprint. It makes it easier to identify if requests come from a browser or not. Fingerprinting is becoming the primary weapon to identify web scraping bots and increases the chances of being blocked.

Some websites already have anti-scraping solutions that check ”fingerprints”, but it is not very common yet. The major problems are that it still brings a lot of false positives, which might have converted to sales. More importantly, it requires tremendous hardware resources to process all the data. Overall, chances to run into such issues are quite slim, but if you do, the best way is to use a headless browser, preferably with stealth addons.

More practical tips on web scraping

1. Visit the home page before accessing the inner content. Regular users rarely have full links to products or articles, first they land on the home page, and then browse further.

2. Data that is under authentication or protected with the password could be considered as private, and scraping such data in some cases can be illegal. Before starting web scraping of any kind, we suggest you consult your legal advisors and carefully read the particular website’s terms of service, or even receive a scraping license if possible.

3. Choose the right proxy type for your web scraping tasks. Two of the main proxy types are residential and datacenter proxies. Usually, they are used for different targets.

Conclusion

Figuring out how to start web scraping can be a complicated task. To make it easier, follow this workflow: set a preferred session, see if it works with a test query, and then start scraping your target public data source. Do not forget to discuss with your legal advisors that you would not encounter any legal issues when web scraping.

The most difficult part is to avoid being blocked by targeted servers. Sessions, HTTP headers, headless browsers, and ”fingerprinting” are the essential things you should note to make your web scraping session successful.

Industry trends

5 Great Web Scraping Use Cases to Gain a Competitive Advantage

For many years, numerous companies have struggled to get ahead in their market for one reason or another. An unfortunate reality is that there are many stumbling blocks to gaining a competitive advantage, whether it's because of a lack of time, resources, or otherwise.

To make matters worse, valuable data used to be complicated to access, as the only option was to gather it manually. But when the digital age arrived, things began to change drastically. Before we knew it, the amount of data available publicly reached astronomical levels, with data eventually overtaking oil to become the world's most valuable resource.

The perfect way to access this data is by using a web scraping service, which automates the data collection process and allows your business to extract data that is freely available in the public domain. Once this information is in your hands, your company is then free to use it to your advantage. However, just be aware that most web scraping projects require a proxy, as this helps to ensure anonymity while preventing other websites from blocking your IP address.

But if you're asking yourself the question – what is web scraping used for? You've come to the right place. In the following article, we'll delve deeper into the most prominent web scraping uses and how each one can benefit your business. So, without any further ado, here's everything that you need to know.

Stock market research

As anyone in the financial world will tell you, the stock market tends to be incredibly unpredictable and vulnerable, and it's been this way for many years. Given that this won't change any time soon, you should always take precautions and keep up with any market changes by any means necessary.

If you do just that, this will ensure that you're prepared to take immediate action should things take a turn for the worse. After all, it only takes one minor change in the stock market, and a company can disappear right before your very eyes.

For many years, manually performing stock market research was almost impossible for most businesses, as it would prove to be costly and inefficient. Luckily, you can use web scraping services to automatically keep track of current stocks in the market and access incredibly valuable financial data nowadays.

As far as web scraping uses go, stock market research is certainly right up there, as it enables you to make the data scraping process in this department fast and straightforward. Plus, you'll

also be able to access plenty of high-quality financial data that will allow you to focus on other areas of your business as well as save time, money, and resources.

Business intelligence

Gathering internal data is an essential consideration for any business nowadays, as it can help you to identify how well your campaigns are performing and allow you to make well-informed business decisions. You can gather such information from different areas of your organisation, including sales, marketing, and HR, with each department being vital to the everyday running of your business.

However, if your business is looking to go a step further, you'll also need to gather external data to remain one step ahead of the competition. Web scraping technology helps significantly with this process, as it will enable your company to gather data on your competitors, such as which marketing approach they are taking. With this information at your disposal, you'll then be free to replicate your competition's marketing efforts and hopefully generate more business in the process.

Dynamic pricing

With so much competition to contend with nowadays, setting optimal prices is one of the most effective ways to entice customers to choose your business. For many years, pricing has been a key consideration for customers. So, if you're able to provide them with a better deal than your competitors, it's likely that customers will flock to your website.

But since most businesses set their prices based on the laws of supply and demand, it's crucial to be flexible with your pricing. That way, you can remain competitive in your market while bringing in as much revenue as possible. If you fail to price your products appropriately, you'll either miss out on potential revenue due to your prices being too low or perhaps you'll drive customers away with rates that are too high.

Luckily, it doesn't have to be this way, as you can use a web scraping service to set an effective dynamic pricing strategy. Web scraping technology will automatically gather the latest pricing information which saves you doing it manually. Once you have this data at your disposal, you can then set the price for your product or service appropriately depending on the conditions of the market.

Enhanced lead generation

If you're looking to attract new customers frequently, lead generation is vital. This is because leads are a necessity for any business, and they form a fundamental part of the sales process. Your sales and marketing departments are probably already working hard to generate leads for

your company consistently. But it's at this stage of the customer journey you can make a huge impact.

A web scraping service can be incredibly useful in terms of lead generation, as you will be able to access a considerable amount of valuable data from your competitors. Once you've structured it into an understandable format, you'll have a massive database of potential clients at your disposal. Then all that's left is to target your campaigns so that they will be more likely to reach your ideal customers.

Competitor analysis

Before the internet came along, most businesses only had to worry about competitors in the vicinity of their company premises. But as an increasing number of companies grew an online presence, it wasn't long before competition became increasingly fierce, with many companies now having to compete with competitors from all four corners of the globe.

Of course, it's essential to continually improve the processes of your business and adapt to changes in the market. But accessing publicly available data on your competitors is the perfect way to get ahead of them. This information can include such things as which marketing strategies they are using, what prices they are charging, and so much more.

Either way, having access to this information has the potential to be incredibly useful, and it gives you the perfect opportunity to outmuscle your competitors. A bonus is that web scraping services automate the whole process, so you won't even need to lift a finger to gain access to highly valuable data that would be extremely difficult and time-consuming to access manually.

Final thoughts

Now that we're in the age of Big Data, the immense value that data provides is clear for all to see. It's now so valuable that the Big Data market will supposedly be worth a staggering $229.4 billion by 2025, just to put things into perspective.

As outlined throughout this article, there are many uses for web scraping services, and the purposes listed above are some of the best ways to use web scraping within your business. So, if you're looking to get ahead in your market and remain one step ahead of your competitors, investing in a web scraping service can most definitely make all the difference.

Using Web Scraping for Lead Generation

Lead generation is what makes your business happen. And using web scraping by applying residential proxies is one of the most optimal ways to generate leads for your business. Lead generation will attract and convert anyone interested in your products or services. Based on Ringlead statistics, 85% of B2B marketers say that lead generation is their most important content marketing goal. This is the main reason why you should consider web scraping for lead generation.

As a business, for you to reach out to your prospective customers and generate more sales, you need qualified leads. That's getting all the details such as the name of a company, street address, contact number, emails, and other necessary information.

Now it is quite apparent that you'll be looking for such information on the internet. Publicly available data with the information mentioned above is readily accessible on many platforms, from social media to featured articles.

Now, gathering social data by hand, especially if you're looking for leads, will take you an unholy amount of time. There are a lot of lead generation tools for this matter, but according to Martech Today, spending on marketing automation tools is expected to reach $25.1 billion annually by 2023.

Building your lead scraper to generate leads can be a lot more cost-efficient, and using web scraping is said to be one of the most effective ways to generate leads. In this article, we'll go through the process of web scraping for lead generation and where to start.

Identifying sources

The first step that should be taken for gathering data for lead generation is identifying what sources you'll be using. You need to figure out where your target customer is located on the internet. Do you want customers or influencers? This will help you clarify which sources you'll need to scrape and find high-quality leads.

If your competitors' customer information is publicly available, you could scrape their websites for their customer demographics. This would give you a good visualization of where to begin with and where your potential customers are located.

Extracting data

Having figured out the sources where your potential clients are located, you'll need to extract the data for your business to be able to use it.

There are a few ways you can extract personal data:

- Buying a lead generation tool like business leads scraper from reliable providers.
- Using readily available scraping tools.
- Writing the code yourself and implementing proxies.

As we already mentioned at the beginning of the article, buying a lead scraper can be costly, and building your infrastructure of data retrieval if you have the right people resources can be a lot cheaper and easier. If you need to achieve your business goals and you can't imagine doing this job without required data, it's worth it to invest in a lead scraper or to spare some time and build your own business lead scraper.

Data extraction simplifies the whole process as well. Usually, when gathering data, it comes unstructured and requires further processing. According to Forrester, data analysts spend up to 80% of their time collecting and preparing data for analysis. However, when building your infrastructure, you'll be able to remove incomplete, duplicate, or incorrect data points.

Wrapping up

Lead generation is the most critical sales and marketing business goal, and according to Strategic IC, companies that automate lead management see a 10% or more bump in revenue in 6-9 months. So having the business leads scraper or other right tools to build your infrastructure to gather data for leads generation is where you should be concentrating on.

The Legal Framework of Data Scraping

In this article, we'll be concentrating over the landmark scraping cases that set the tone for future scraping legal claims such as copyright, CFAA, etc. and how it affects scraping today and what web scraping legal issues can one encounter when scraping certain websites.

Feist Publications v. Rural Telephone Service Co (1991)

The situation concerns two U.S. telephone service companies Feist and Rural. Feist was making compilations of telephone listings and in doing so, copied entries from the Rural's directory, leading to the latter suing for copyright breach.

In courts, it was decided that an element of creativity is needed for sets of information to be copyrightable. The court found no creativity in Rural's alphabetical list of phone numbers and denied its copyright protection.

The decision set the tone for future scraping copyright claims, as it established that compilations of factual information were not protectable by copyright. Still, the creative parts of it (e.g., author's comments, order or style of presentation, etc.) might be.

eBay vs. Bidder's Edge (2000)

Bidder's Edge, an online auction listing aggregator, was scraping eBay's auction data and continued to do so after receiving a C&D (Cease and Desist) letter as well as an IP address block. eBay sued Bidder's Edge under U.S. legal rule of trespass to chattels, which forbids intentional interference with another person's movable personal property.

Bidder's Edge activities only amounted to approximately 100,000 hits per day (1,5% of eBay's total daily traffic). Despite Bidder's Edge activities being minor in scale, the court found them sufficient for trespass to chattels to apply and ordered Bidder's Edge to stop scraping eBay.

The decision was criticized and deconstructed by other courts in future cases, with some of them stating that actual harm would need to be shown to prove "interference" within the context of the trespass to chattels rule.

Facebook v. Power Ventures (2009)

Power Ventures was an operator of a website, aggregating different social network information on a single page. Because of its scraping activities, it was sued by Facebook for allegedly breaching U.S. Computer Fraud and Abuse Act (the CFAA).

The CFAA forbids obtaining information from a protected computer (or network) after intentionally accessing it without or by excess authorization. The court decided that continuing to access a network after receiving a C&D letter referencing the CFAA can lead to a violation of the said act.

The decision was heavily scrutinized because the question of whether Facebook's user data should be considered publicly available was not analyzed. Critics also feared that the CFAA could be used as a tool to shut down the competition by big internet companies. Thankfully for scraping companies, a case with similar circumstances yielded a different outcome in a future landmark decision (see below: hiQ labs v. LinkedIn).

Craigslist vs 3Taps (2013)

3taps was scraping Craigslist to aggregate user-submitted Craigslist advertisements. After issuing a C&D letter and an IP address block, Craigslist sued 3taps for breaching the CFAA as well as for infringement of its copyright.

The court applied a similar theory as in Facebook v. Power Ventures case, saying there is a breach of CFAA as 3Taps authorization to Craigslist's network was revoked via a C&D letter and IP address block. Concerning the copyright claim, the court found that Craigslist owned the intellectual property rights on the advertisements, as for two weeks, Craigslist's Terms of Use (ToU) attributed them the full ownership of those rights. It was also found that these advertisements were protected by copyright, as Craigslist fulfilled the required creativity condition by categorizing the advertisements.

The decision started active discussions with regards to the legal "weight" of C&D letters, which are unilateral lists of demands issued by the sender, as well as to the fact that Craigslist was able to claim exclusive intellectual property rights to its advertisers' ad copy, even if temporarily.

QVC v. Resultly (2014)

A dispute between QVC, an online and TV retailer that got scraped by Resultly provided a few interesting insights as far as scraping goes.

QVC tried to invoke a different CFAA ground, which prohibits intentionally causing damage. Resultly's scraping activities (500-600 requests/s) did overload QVC's servers, but this argument was rebuked as Resultly's business directly benefited from QVC's website running without interruption. Further, QVC's ToU did not prohibit scraping, while its robots.txt file did not put a limit on crawl rates.

No infringement of the CFAA was found in this case by the courts.

Ryanair v. PR Aviation (2015)

Ryanair's argument with a flight price comparison company PR Aviation provided a glimpse of how scraping could be interpreted in European courts. Ryanair's website subjects its visitors to ToU, which explicitly prohibits scraping. PR Aviation was scraping Ryanair, who took them to court in the Netherlands for breach of contract.

Ryanair came out second best from the dispute, as the Dutch court said that there was no valid contract formed between the companies. It made an interesting allegory, stating that anyone putting up a poster in a shop window visible from the public road, which reads: "Whoever reads further, must pay € 5," cannot accept that the person reading this wants to commit to such a condition.

Still, this does not mean that ToU would not be applicable in a different scenario, as there were a lot of circumstances unfavorable to Ryanair here. Namely, the facts that at the time of the scraping, Ryanair was presenting its ToU in a browsewrap, which is not generally accepted as legally binding by courts, as well as the fact that the scraped data was free and accessible to everyone.

Ryanair v. Expedia (2019)

Expedia, a U.S. flight comparison company, was scraping Ryanair's data and continued doing so after receiving a C&D letter. Consequently, it was sued by Ryanair for breaching the CFAA. Expedia argued that Ryanair is an Irish company, therefore the CFAA, a U.S. statute should not be applicable.

The courts established that the CFAA might indeed apply to U.S. companies acting internationally. After this, Ryanair and Expedia settled the case, with the details being confidential. With that being said, as of this day, there are no Ryanair flights being offered via Expedia's website.

HiQ labs v. LinkedIn (2019)

HiQ labs is a company that scrapes data from public LinkedIn profiles to provide tools and insights on employees to businesses. After allowing HiQ scrape for several years, in 2017, LinkedIn issued a C&D letter to HiQ and themselves launched a tool similar to HiQ's functionality. HiQ sought an injunction in court, which was granted, leading to LinkedIn being asked to withdraw the C&D letter and stop applying any blocking measures against HiQ.

LinkedIn appealed the decision, arguing that HiQ's scraping was breaching the CFAA. The court decided that HiQ was not acting in breach of the CFAA, as the data scraped from LinkedIn was public (profiles containing user-generated content; not put behind a password wall). The court said that companies should not be able to revoke authorization where one is not needed in the

first place, as well as that allowing companies like LinkedIn to decide who can collect and use public data would be contrary to the public interest.

The decision was favorable to scraping companies and reconsidered some of the much-criticized previous court practice with regards to the applicability of the CFAA, narrowing the applicability of this act with regards to public data (e.g., Facebook v. Power Ventures, Craigslist v. 3Taps). With that being said, if not done with caution, scraping activities might still be subject to potential breaches of the CFAA (e.g., under different case's circumstances) as well as other grounds such as, among others, trespass to chattels, copyright or breach of contract.

Wrapping up

Now, having gone through this, is it legal to scrape data from websites? As our legal counsels would put it themselves – it depends. There is no simple answer to this question as one must answer whether the scraping done does not breach any laws surrounding the said data.

So please, take this article as informational and educational only. It does not replace independent professional advice and judgement. Statements of fact and opinions expressed are those of the presenters only, and unless expressly stated to the contrary, are not the opinion or position of Oxylabs.

If you would like to know more about any of the topic mentioned here 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