**Web Scraping Project using Scrapy**

Through this project I aim to explore the basics of webscraping and to try and make it easier for someone who wants to get started with scraping the website they want to scrape data from. This is not going to be ‘easy’ easy so feel free to contact me using the link at the bottom of this page if you have any doubts regarding webscraping and I will try to explain best I can. Also, there are not that many tutorials out there that really are beginner friendly.

If you're a beginner trying to understand webscraping and a powerful framework for it like scrapy,I hope this acts as a guide that walks you through its core concepts, practical examples, and essential tools to mask your identity while scraping like user agents, headers, and proxies.

We will be scraping data from the website below. Follow the link and explore a bit to get an idea of what the site we will be scraping data from is about.

LINK

With that being said, let us get started.

What is Webscraping?

Web scraping is the process of extracting data from websites. Imagine you're visiting a website to look up product prices, headlines, or reviews. Instead of manually copying this information, web scraping uses a program or script to fetch the data for you in an organised and efficient way.

How It Works:

 **Send a Request**:  
A **web scraper**, powered by the Scrapy framework, sends an HTTP request to a website asking for its content. This is usually done using a **Scrapy Spider** that you define to specify which URLs to scrape.

 **Receive a Response**:  
The website responds with its content, typically in HTML format. This response contains all the data and structure of the page.

 **Parse the Content**:  
The scraper **parses** (analyzes) the HTML to locate and extract specific information you need, such as product names, prices, or links. This step involves using **CSS selectors** or **XPath expressions** to target the desired data.

 **Store or Process the Data**:  
After extracting the data, Scrapy lets you store it in formats like JSON, CSV, or a database, or pass it through an **item pipeline** for further processing.

What is Scrapy?

Scrapy is a powerful Python framework for extracting data from websites.Scrapy is based on the concept of spiders. It handles tasks like sending requests, parsing responses, and storing data with minimal manual effort.

1. **Sending Requests**:  
   Spiders in Scrapy send requests to fetch web pages. For example:

yield scrapy.Request(url, callback=self.parse)

* + **url**: The web page to scrape.
  + **callback=self.parse**: Specifies the method to process the response.

1. **Processing Responses**:  
   The HTML content of a page is downloaded into a response object, which provides tools like CSS selectors and XPath to extract data.
2. **Yielding Data**:  
   Instead of returning data, Scrapy uses yield to handle multiple requests concurrently.
3. **Storing Data**:  
   Scraped data can be saved in JSON, CSV, or databases using Scrapy pipelines.

**Key Concepts in Scrapy**

**CSS Selectors vs. XPath**

Scrapy uses **CSS selectors** and **XPath** for parsing HTML.

**CSS Selectors**:

* Easy and readable for simple element selection.
* Example:

response.css('li.next a::attr(href)').get()

* + **li.next**: Selects <li> elements with the class next.
  + **a::attr(href)**: Extracts the href attribute of the anchor tag.
  + **.get()**: Retrieves the first matching result.

**XPath**:

* Powerful for complex queries involving element relationships.
* Example:

response.xpath("//ul[@class='breadcrumb']/li[@class='active']/preceding-sibling::li[1]/a/text()").get()

* + Selects the text of the link in the last <li> sibling before the active breadcrumb.

**Yield vs. Return**

* **yield**: Returns data or requests without ending the function, allowing Scrapy to handle multiple tasks concurrently.
* **return**: Ends the function, returning a single value.

**Following Links**

To scrape multiple pages, Scrapy uses response.follow:

yield response.follow(next\_page\_url, callback=self.parse)

* **next\_page\_url**: The URL of the next page to scrape.
* **callback=self.parse**: Specifies the function to process the response.

**Items and Item Pipelines**

**Items**

Items help define the structure of your data, acting like containers for scraped information. Think of this as a way to better organise the data you scraped. Example:

class BookItem(scrapy.Item):

title = scrapy.Field()

price = scrapy.Field()

**Item Pipelines**

Consider this as the data processing and cleaning tool. In addition, it is used to store data in databases. Example:

class PricePipeline:

def process\_item(self, item, spider):

item['price'] = item['price'].replace('$', '')

return item

**Steps to follow along**

1. Enter virtual environment (venv/Scripts/activate if you are on windows os)

2. Start new project: scrapy startproject project\_name

3. Change current directory to inside the project folder

4. Generate a new spider using: scrapy genspider spider\_name <https://link.to.scrape.com>

5. Install, assign, and enter ipython shell (we use ipython because it makes interaction between the spider and the site easier for us):

--- Install using pip install ipython

--- Assign shell as ipython inside scrapy.cfg file (just enter this line inside settings: shell = ipython)

--- Enter ipython shell: scrapy shell

6. Start fetching and sending requests to the site inside shell (think of this as a preliminary way to find out the perfect lines of code that will fetch the required data from the site)

7. Add the final lines of code to the spider file. After you are done, it should look something like this:

<p><img src="images/pf7/sub.jpg" alt="Image" class="img-fluid"></p>

8. Run scrapy in terminal and ask to return data in a csv/json file:

scrapy crawl spider\_name -o filename.csv

OR

scrapy crawl spider\_name -o filename.json

9. Voila! Your scraped data is ready!

Items and item pipelines are not necessary but to make the code more readable we will add it as well.

Make sure to use user agents, headers, proxies if you want to be extra sure that your ip address does not get blocked. This won’t be an issue if you follow step-by-step as this does not involve sending multiple requests and also the site we are scraping (books.to.scrape) is less hostile towards webscrapers. But be cautious if you are scraping other sites. Respect robots.txt! (You can check the rules using this format for any site by typing this in the search bar: www.link-to-site.com/robots.txt)

Let me know if you face any difficulty understanding the body of code, I would love to try my best to simplify things further.

**Conclusion**

Scrapy is an incredibly flexible and powerful framework for web scraping, but mastering it requires understanding its key components. By learning how to use CSS/XPath selectors, manage user agents and proxies, and handle dynamic websites, you can create efficient and stealthy scraping workflows to scrape any website you want.

If this peaks your interest, next up we will be diving into a project that will work through the complete data analytics process (remember the 6 steps in data analytics are: Ask>Prepare>Process>Analyse>Share>Act) from sourcing data using webscraping to sharing insights on the data using powerful visualisation tools like Power BI. For this we will be scraping the famous Wikipedia website.

Happy scraping!