## WEB SCRAPING USING SELENIUM

Web scraping involves two broad categories of web pages—pages that need rendering and pages that do not need rendering. Web scraping the pages that need rendering is often called web scraping dynamic web pages. This is where Selenium shines.

[Selenium](https://www.selenium.dev/) is one of the oldest and perhaps the most widely known tool. Selenium development began as early as 2004. This began as a tool for functional testing and the potential of web scraping was soon realized.

The biggest reason for Selenium’s is that it supports writing scripts in multiple programming languages, including Python. It means that you can write Python code to mimic human behaviour. The Python script will open the browser, visits web pages, enter text, click buttons, and copy text. This can be combined with other features in Python to save data in simple CSV or complex databases.

In this guide on how to web scrape with Selenium, we will be using Python 3. The code should work with any version of Python above 3.6.

**Setting up Selenium**

Setting up Selenium involves setting up two components—the selenium package for Python and the driver for the browser that you want to use.

**Selenium Package**

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

pip install selenium

**Selenium Drivers**

Depending on your Operating System and browser of choice, the source of downloads would differ. In this example, we are going to work with Chrome. Verify the version of Chrome installed by clicking the 3 dots on the top right of the Chrome, then point to About and select About Chrome. Take a note of the version number. You will need to know your version to get the appropriate driver.

You can find the links to download the drivers for Firefox, Chrome, and Edge [here](https://pypi.org/project/selenium/#drivers).

Once you download the executable appropriate for your operating system, extract it and place it in a folder. The next step would be to take a note of the folder path. You can add this path to your PATH environment variable or optionally, this folder path can also be provided in the code.

**Getting Started with Selenium**

To see Selenium in action, type in these lines in your favorite code editor and run it as a Python script. You can also run these statements from the Python console.

from selenium.webdriver import Chrome

driver = Chrome(executable\_path='C:/WebDrivers/chromedriver.exe')

driver.get("http://books.toscrape.com")

This will launch Chrome and load the web page. There will be notice below the address bar:

Chrome is being controlled by automated test software.

To close this browser, simply run this line:

driver.quit()

**Data Extraction with Selenium - Locating Elements**

The first step of extracting the data is to locate the elements. Selenium offers a variety of find\_element methods to help locate elements on a page:

* find\_element\_by\_id - Finds element by the id attribute
* find\_element\_by\_name - Finds element by element name attribute
* find\_element\_by\_xpath - Finds element by XPath (Recommended)
* find\_element\_by\_css\_selector - Find element by using a CSS selector( Recommended)
* find\_element\_by\_link\_text - Find <a> elements by matching its text
* find\_element\_by\_partial\_link\_text - Find <a> elements by matching its text *partially*
* find\_element\_by\_tag\_name - Finds element by the tag name
* find\_element\_by\_class\_name - Finds element by the class attribute

All these method return one instance of WebElement.

**XPath**

XPath is a syntax language that helps find a specific object in [DOM](https://www.w3schools.com/js/js_htmldom.asp). XPath syntax finds the elements from the root element either through an absolute path or by using a relative path. e.g.:

* **/** : Select child element. /html/body/div/p[1] will find the first p which is in a div tag, which in turn is a child of body element. This means that if a <span><div><p>something</p></div></span> will not be selected.
* **//**: Select all descendant element from the current element. //p will find all p elements, whether they are in a div or not.
* **[@attributename='value']**: It looks for a specific attribute with a specific value. This can also be used as [@attributename] to search for the presence of this attribute, irrespective of the value.
* XPath functions such as contains() can be used for a partial match

For example, on the web page [http://books.toscrape.com](http://books.toscrape.com/), if we want to locate the link to the Humor on the navigation pane, this can be done using the contains function. Note that the text() contains white space. That's why text()="Humor" will not work. This will need to contains functions.

# Page source

# <a href="catalogue/category/books/humor\_30/index.html">

# Humor

# </a>

driver.find\_element\_by\_xpath(f'//a[contains(text(),"Humor")]')

**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)

**Waiting for Elements to Appear**

When working with dynamic websites, some components take time to appear or become clickable. This means that there is a need of waiting in a controlled manner. Python's inbuilt time.sleep() is not a good option as the elements may take shorter or longer to appear.

**Implicit Waits**

An implicit wait tells WebDriver to poll the DOM for a certain amount of time when trying to find an element if not immediately available.

driver.implicitly\_wait(10) # seconds

This is usually good enough, but often more control is required.

**Explicit Waits**

Using explicit wait, finer control can be achieved. This allows for locating various events such as the presence of elements, change of the url, element to become clickable, and so on. Here is one example:

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

try:

books = WebDriverWait(driver, 10).until(

EC.presence\_of\_all\_elements\_located((By.CSS\_SELECTOR, '.product\_pod'))

)

finally:

driver.quit()

This will allow the a wait up to 10 seconds for this to return a web element. If the element is not found even after 10 seconds, an exception will be thrown . To dig deeper into this topic, go ahead and check out the official [Selenium documentation](https://selenium-python.readthedocs.io/waits.html).

In the file books\_selenium.py, you can see a complete example where selenium is used to get data from multiple categories of books. This file also has code to export the data to a CSV file.

**Selenium vs Puppeteer**

The biggest reason for Selenium’s popularity and complexity is that it supports writing tests in multiple programming languages. This includes C#, Groovy, Java, Perl, PHP, Python, Ruby, Scala, and even JavaScript. It supports multiple browsers, including Chrome, Firefox, Edge, Internet Explorer, Opera, and Safari.

However, for web scraping tasks, Selenium is perhaps more complex than it needs to be. Remember that Selenium’s real purpose is functional testing. For effective functional testing, it mimics what a human would do in a browser. Selenium thus needs three different components:

* A driver for each browser
* Installation of each browser
* The package/library depending on the programming language used

In the case of Puppeteer, though, the node package puppeteer includes Chromium. It means no browser or driver is needed. It makes it simpler. It also supports Chrome if that is what you need.

On the other hand, multiple browser support is missing. Firefox support is limited. Google announced [Puppeteer for Firefox](https://www.npmjs.com/package/puppeteer-firefox), but it was soon deprecated. As of writing this, [Firefox support is experimental](https://github.com/puppeteer/puppeteer#q-which-firefox-version-does-puppeteer-use). So, to sum up, if you need a lightweight and fast headless browser for web scraping, Puppeteer would be the best choice. You can check our [Puppeteer tutorial](https://oxy.yt/Rr4w) for more information.

**Selenium vs Scraping Tools**

Selenium is great if you want to learn web scraping. We recommend using it together with BeautifulSoup as well as focus on learning HTTP protocols, methods on how the server and browser exchange data, and how cookies and [headers](https://oxy.yt/nr7T) work.