GESIS Guides to Digital Behavioral Data
#11

# Collecting Data from Dynamic Websites with Selenium

FELIX SOLDNER

GESIS – Leibniz Institute for the Social Sciences, Cologne, Germany

Publication date: December 3, 2024; Version 1.0

*In this guide, we will show you how to use Selenium, a browser automation software, with Python to collect content from websites where content only appears after interacting with them (scrolling, clicking, etc.). Readers of this guide should be familiar with Python, HTML, and the scraping of static websites (see this web scraping guide for an introduction to that topic).*

**Keywords:** web scraping, infinite scrolling, pagination, Python, Selenium

## 1 Dynamic websites

Dynamic websites change as users interact with them, presenting different information to different users. The content of such pages changes due to various reasons, including:

- clicking, scrolling, mouse hovering,
- screen sizes, languages (IP-based), devices,
- previous visits (user's browsing history),
- location and local time.

Changes can occur on the client side (i.e., user), such as JavaScript interactions, that do not necessarily change the website's source code but rather its appearance, such as expanding text boxes or drop-down menus. In such a case, classical scraping methods are sufficient for extracting the entire text because the displayed truncated version of a text is already stored in the loaded website but visibly hidden.

However, changes can also occur on the server side, leading to changes in both the website's appearance and contents. For example, more content might be provided when scrolling to the bottom of a website, which can often be observed in social media feeds (e.g., Facebook, X/Twitter, or Quora) or online shopping platforms, enabling users to scroll endlessly. Website interactions that lead to content changes are often challenging or not obtainable through classic scraping approaches since they require a JavaScript execution initiated by a user interaction. For such cases, we can use browser automation tools, such as Selenium, to help us imitate user interactions and make data collection possible.

## 2 Selenium

Selenium is a browser automation software that can interface with many different browser types and programming languages. Thus, we can write programming scripts that control the browser and imitate our behavior, such as clicking or scrolling. Before writing a programming script, we need to set up Selenium by downloading a driver. Depending on the browser we want to use (e.g., Firefox or Chrome), we need a different driver, which you can find here. For this guide, we will use Google Chrome, for which you can find the driver here. For downloading the correct driver, you need to know which operating system (e.g., Windows, Linux, Mac) your machine runs on and which browser version you have. For Chrome, you can find the browser version under Settings > About Chrome (see Figure 1 below).

Download the correct driver and unpack the zip folder. In our case, we will end up with the file "chromedriver.exe", which needs to be in the same folder we are running our script in (set directory for script environment).

The Selenium webpage contains documentation for all the programming languages, which you can find here.

Since we are using Python, we can also find a separate documentation here. Selenium can also be used in R via the package RSelenium. There are many tutorials on this package available online. A good (though somewhat older) can be found here.

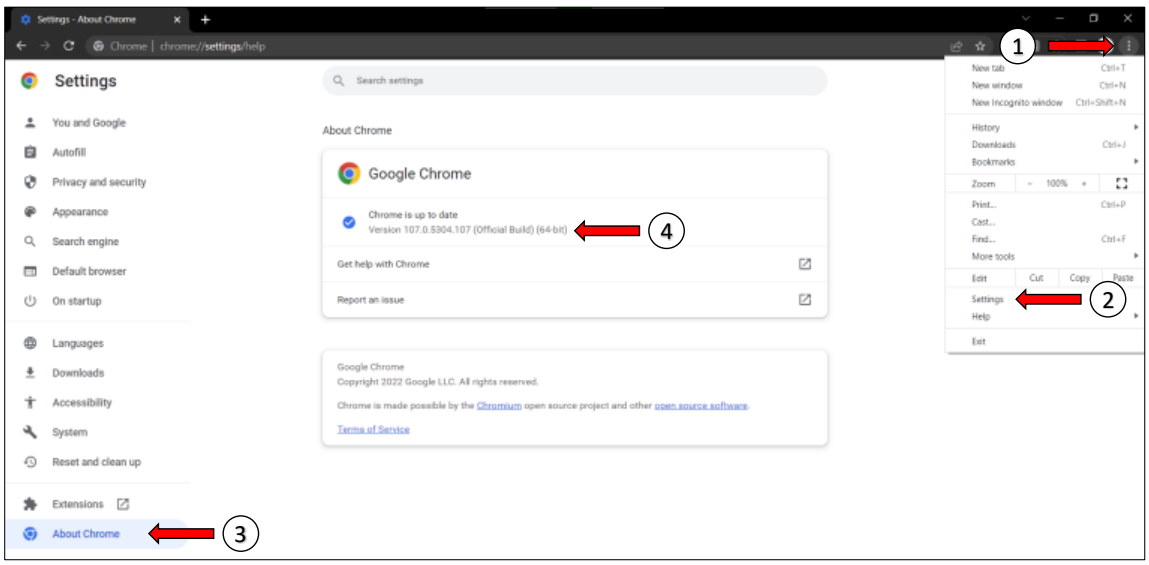


**Figure 1.** Screenshot of Google Chrome settings to determine its version.

Both documentations are very useful and should be kept close when working with Selenium. When you inspect the documentation, you will recognize that besides sending specific behavioral commands to the browser, accessing web elements is very similar to other tools like, e.g., Beautiful Soup. You will need XPATH, CSS selectors, and other properties of web elements to interact with them.

Next, we will show you how to utilize Selenium with Chrome to imitate user interactions on a website, allowing you to collect data from it. We use practice websites in our guide, but for any other website, you must verify whether scraping data from the webpage is allowed by reading the website's Terms of Service (ToS). If scraping is permitted, the ToS should be saved and archived for transparency and as supporting documentation that scraping was allowed. However, implementing a scraper might still not be favorable due to the increased computational burden on the website server by the scrapers' information requests. We can artificially slow down the scraper and reduce the server's computational load to prevent such a burden. More information and resources regarding ethical and legal issues around scraping can be found in this overview of online data collection approaches.

# 3 Collecting information from dynamic websites with Selenium

## 3.1 Header

This guide will show you how to use Selenium (a browser automation software) with Python to collect information from dynamic websites.

We will start by importing the necessary libraries.

```
import pandas as pd
from time import sleep # to slow down our scraper

## Selenium specific packages
# to load the browser
from selenium import webdriver

# To automate typing (e.g., filling out forms)
from selenium.webdriver.common.keys import Keys

# To search for web elements
from selenium.webdriver.common.by import By

# To change our user agent
from selenium.webdriver.chrome.options import Options
```

We can now start the driver (i.e., the browser), which should appear as a separate window. The driver can only work if the `chromedriver.exe` file is in the working directory of the current Notebook.

```
driver = webdriver.Chrome()
```

We will use the "driver" instance to navigate the website and find web elements. With the following code, we can obtain our current user agent.

```
agent = driver.execute_script("return navigator.userAgent")
print(agent)
```

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36
```

We can change the user-agent information to make ourselves identifiable. Many websites that permit scraping will require you to make yourself visible and contactable through the user-agent. We need to restart our browser, before we can change the user-agent.

```
driver.quit()

# add our project name and e-mail address to the user-agent

opts = Options()
opts.add_argument("user-agent=practicing dynamic web scraping;
contact me through: [e-mail address]")

driver = webdriver.Chrome(options=opts)
```

Let's check if we changed our user agent.

```
agent = driver.execute_script("return navigator.userAgent")
agent
```

```
'practicing dynamic web scraping; contact me through: [e-mail address]'
```

Looks good! Now, we can start with our project.

## 3.2 Infinite scrolls

Website content is often hidden behind scroll actions, such as for endless news feeds or web searches (e.g., Google). We will show you how to scroll with Selenium, triggering content to be loaded. We will use the website scrapingclub.com, which has integrated exercises we can borrow from.

First, we need to load the website with Selenium.

```
url_search =
"https://scrapingclub.com/exercise/list_infinite_scroll/"

driver.get(url_search)
```

Let's find the elements that store the product information (name, price, etc.). Similarly, when working with other scraping approaches (e.g., static webscraping with `BeautifoulSoup`), we need to find the web elements by inspecting the HTML structure of the website and locating them through their paths, class, or names. Ideally, the elements have an ID we can identify them with. With Selenium, we can use commands such as `find_element` or `find_elements` and combine it with `By` to specify how we want to find it. For example, we can use `CLASS_NAME`, `CSS_SELECTOR`, or `ID`. Look at the documentation for locating elements for more options.

Here, we use the class name of the advertisements, "post". If you are new to scraping and unsure how to find class names, tag names, etc., have a look at this guide on static web scraping.

```
products = driver.find_elements(By.CLASS_NAME, "post")
```

Let's check how many results we can find and how the information is stored.

```
len(products)
```

```
10
```

```
products[1].text
```

```
'Patterned Slacks\n$29.99'
```

Currently, we can only locate ten products, but when we visit the website and scroll down, we see that more products are loaded.

We can scroll down with Selenium by implementing the `execute_script()` function and define a scroll script such as "window.scrollTo(0, Y)" where Y is the height of the monitor (this Stackoverflow post provides further solutions). You can specify any number for Y or determine it dynamically for the length of the website using `document.body.scrollHeight` to scroll to the (current) end bottom of the website.

Let's test it out and see how many elements we can find.

```
driver.execute_script("window.scrollTo(0,
document.body.scrollHeight);")

products = driver.find_elements(By.CLASS_NAME, "post")
len(products)
```

```
10
```

We still can find only ten products because we did not wait long enough for the website to load. We can implement artificial pauses with `sleep()` to wait for the website to load all content.

```
sleep(4) # wait for 4 seconds.

products = driver.find_elements(By.CLASS_NAME, "post")
len(products)
```

```
60
```

Now, we can find 60 products. For an infinite feed, we can implement the method we used in a while loop to keep scrolling.

We can now collect some information about the loaded products, such as the names, prices, and image links. We can start by extracting the names from `products` and store them in a list with `text`. Since the string also contains the price, we need to split it.

```
names = [name.text.split("\n")[0] for name in products]
names[23:26]
```

```
['Crinkled Flounced Blouse',
 'Bib Overall Dress',
 'Loose-knit Sweater']
```

Similarly, we can extract the prices, but we need to remove the "$" characters and convert them to numbers (float).

```
prices = [float(price.text.split("\n")[1].replace("$","")) for
price in products]
prices[23:26]
```

```
[24.99, 29.99, 17.99]
```

Lastly, we want to save the image links. We can find the elements by the class name.

```
images = driver.find_elements(By.CLASS_NAME, "card-img-top")
```

The elements are saved in a list, and we need to extract the links, which are saved as a value of the attribute "src". By using the function `get_attribute()`, we can extract the links from "src" or other attributes (e.g., "id", "href").

```
images[0]. get_attribute("src")
```

```
'https://scrapingclub.com/static/img/90008-E.jpg'
```

As previously implemented, we can wrap the extraction into a list comprehension.

```
image_links = [image.get_attribute("src") for image in images]
image_links[23:26]
```

```
['https://scrapingclub.com/static/img/00959-A.jpg',
 'https://scrapingclub.com/static/img/94323-B.jpg',
 'https://scrapingclub.com/static/img/71342-J.jpg']
```

Let's save the data into a Data Frame.

```
products_df = pd.DataFrame({"name": names, "price": prices,
"image_link": image_links})
```

Lastly, we need to close the browser.

```
driver.quit()
```

## 3.3 Filling out forms (log-ins)

Selenium also allows us to fill out entry fields, such as log-in credentials or search queries, adding a layer of interaction and allowing you to automate advanced scraping processes.

We will show you how to log in to a website using this practice website. First, we need to start the browser.

```
opts = Options()
opts.add_argument("user-agent=practicing dynamic web scraping;
contact me through: [e-mail address]")

driver = webdriver.Chrome(options=opts)
```

Next, we can load the website.

```
url_search = "https://scrapingclub.com/exercise/basic_login/"
driver.get(url_search)
```

Then, we need to find the element containing the entry field for our username and password. Often, and as in this case, they have specific IDs, we can find them by. Here, we can find the entry field for the username with its id “id_name”.

```
login_name = driver.find_element(By.ID,'id_name')
```

Next, we can use the `send_keys()` function to send keystrokes to the field specified by a string.

```
username = "scrapingclub" # specify username
login_name.send_keys(username)

sleep(1.5)
```

We can repeat the same process with the password.

```
PW_field = driver.find_element(By.ID, 'id_password')
my_password = "scrapingclub"
PW_field.send_keys(my_password)

sleep(1.5)
```

Lastly, we need to log in by clicking the “Log in” button. To do so, we locate the button by its css selector and use the `click()` function.

```
driver.find_element(By.CSS_SELECTOR, "button.btn-
primary").click()

sleep(2)
```

Looking at the website, we can see that we successfully logged in. The same approach can also accept or decline cookies on a given website.

```
driver.quit()
```

## 3.4 Navigation and pagination

When scraping static webpages, navigation, and pagination is handled through altering the URL of the webpage. However, when scraping a dynamic website, changing the URL will often not suffice to load the content you want to collect. Thus, we have to imitate the clicks we would do as a user when navigating through the page. Thus, we will now show you how to use the `click()` function to navigate through a website, place items in a shopping basket, and retrieve the total price of all products. We will use the practice website to shop Pokémons.

Let's find the following Pokémons on the first three pages and place them in our basket:

- Ivisaur
- Beedrill
- Pikachu
- Diglett

First, we load the website.

```
opts = Options()
opts.add_argument("user-agent=practicing dynamic web scraping;
contact me through: [e-mail address]")

driver = webdriver.Chrome(options=opts)

url_search = "https://scrapeme.live/shop/"driver.get(url_search)
```

As we can see, the first two Pokémons are on the first page, and we can add them to our basket by clicking on the "Add to basket" button, which we can identify through the css selector "data-product_id".

```
# identifying button
Ivisaur = driver.find_element(By.CSS_SELECTOR, '[data-
product_id="729"]')
Ivisaur.click() # click on button
```

We can repeat the same procedure with the next Pokémon Beedrill.

```
Beedrill = driver.find_element(By.CSS_SELECTOR, '[data-
product_id="742"]')
Beedrill.click()

sleep(1)
```

Our next two Pokémons are on pages 2 and 3. Similar to clicking on "add to basket", we can also click on the next page button. First, we need to find the button. Here, we can use the class name "next".

```
next_page = driver.find_element(By.CLASS_NAME, "next")
next_page.click()

sleep(1)
```

We implemented a pause for 1 second, allowing the page to load the content. We can now repeat the process of adding the Pokémon to our shopping basket.

```
Pikachu = driver.find_element(By.CSS_SELECTOR, '[data-
product_id="752"]')
Pikachu.click()

sleep(1)
```

The last Pokémon is on page 3.

```
next_page = driver.find_element(By.CLASS_NAME, "next")
next_page.click()

sleep(1)
```

Lastly, we add Diglett to the shopping basket.

```
Diglett = driver.find_element(By.CSS_SELECTOR, '[data-
product_id="801"]')
Diglett.click()

sleep(1)
```

After adding all items, how much do all of the Pokémons cost? Hovering with the mouse over the shopping basket shows us that the total price at the top is not correctly updated, but the "Subtotal" at the bottom contains all prices correctly. Reloading the webpage would update the prices, but we try to avoid new requests as much as possible for any scraping project, limiting the server load. Thus, we will collect the price visible to us when hovering over the basket.

While inspecting the HTML structure, we see that the subtotal price element is not uniquely findable. However, we can uniquely find the parent element through its class name and then retrieve the element children through `get_attribute()` specified with "`innerHTML`".

```
basket_price = driver.find_element(By.CLASS_NAME, "woocommerce-
mini-cart__total").get_attribute("innerHTML")
```

The basket_price element contains all children elements as a string.

```
basket_price
```

```
'<strong>Subtotal:</strong> <span class="woocommerce-Price-
amount amount"><span class="woocommerce-Price-
currencySymbol">£</span>414.00</span>'
```

We can extract the price through a regex search through the pattern  surrounding the price.

```
import re

result = re.search('</span>(.*)</span>', basket_price)
print(result.group(1))
```

```
414.00
```

```
driver.quit()
```

## 4 Conclusion

We showed you how to utilize Selenium to imitate user interactions, allowing you to load and collect website content. With these approaches, you will be able to collect data from most websites. For more advanced methods, check out the Selenium and the associated Python documentation.

All links were retrieved on August 5, 2024.

Acknowledgements

N. Gizem Bacaksizlar Turbic provided helpful comments on the manuscript.

About the author

**Felix Soldner** is a post-doctoral researcher at GESIS – Leibniz Institute for the Social Sciences in Cologne, Germany at the Computational Social Science department. He works with data science methods, such as Machine Learning (ML) and Natural Language Processing (NLP), to research subjects, such as deception detection, fraud prevention, dark web markets, and data biases impacting ML performances. For his projects he uses various data collection approaches, including custom scrapers and APIs.

Suggested citation

Soldner, F. (2024). *Collecting Data from Dynamic Websites with Selenium*. (GESIS Guides to Digital Behavioral Data, 11). Cologne: GESIS – Leibniz Institute for the Social Sciences.

Series editors

Danica Radovanović, Maria Zens, Johannes Breuer, Katrin Weller, Claudia Wagner

Publisher

GESIS Leibniz Institute for the Social Sciences