GESIS Guides to Digital Behavioral Data
#10

How to collect online data through web scraping

FELIX SOLDNER, N. GIZEM BACAKSIZLAR TURBIC & POURIA MIRELMI

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

Publication date: September 2, 2024; Version 1.0

This guide will show you how to collect online data through web scraping. We will introduce the basic principles of web scraping and then demonstrate how to build a scraper through two examples of collecting information about books and poems from two different websites. Readers of this guide should be somewhat familiar with HTML because most webpages use it as a basic structure with which we will need to interact to implement a scraper.

Keywords: data collection, web scraping, how-to, static web pages, web harvesting

1 Introduction to web scraping

Broadly, web scraping can describe all methods and techniques used for extracting data and information from the web (Dewi et al., 2019). What we refer to as web scraping here is sometimes also known as **screen scraping**. In essence, researchers download the HTML files used by the web browser to display content and then extract the parts they are interested in with XML parsers. This method of data collection offers the advantage of being able to extract virtually any visible information from a website. Content commonly accessed through mobile apps (e.g., Instagram, Facebook, TikTok) but also viewable on a web browser can be collected as well. Although manually collecting such content is possible (e.g., by right-clicking and selecting “save page as”), it is inefficient and can be automated with the help of programming languages or standalone software. Additionally, web scraping is an alternative for collecting data when no APIs are available. Visible

content on a website can be generated either statically or dynamically. Content generated statically is readily available when the requested URL is loaded (i.e., the content is displayed when visiting the website without interacting). Content that is generated dynamically means that the website displays content due to user interactions with it (e.g., clicking, scrolling). For example, a social media newsfeed that allows endless scrolling is dynamically generated. A reloading of the website is not required. Parsing statically generated content is called **static web scraping**, which we will demonstrate in this guide. Scraping a static website is more straightforward, as no interactions are required to load the content we are interested in. While static websites were the norm, dynamically generating web content is increasingly used by service providers.

Before implementing a scraper, the website's Terms of Service (ToS) should be read and saved to ensure transparency and determine whether scraping is allowed. The legality of web scraping can vary depending on several factors, including the researcher's country of residence, the type of data being scraped, the purpose of the scraping, and how the scraped data is shared with third parties (Whittaker, 2022). If scraping is permitted, implementing a scraper might still not be favorable due to the increased computational burden on the website server by the scrapers' information requests. However, artificially slowing down the scraper can help reduce the server's computational load. More information and resources regarding ethical and legal issues around scraping can be found in this overview of online data collection approaches in Guide #8.

2 Building a scraper

A scraper might take two approaches:

- Visit a website, locate the content we want (parsing), and save it locally in a structured format (e.g., data frame), or
- visit a website and save the entire website locally in its raw state (e.g., HTML files) before parsing and saving it structurally.

Saving the raw HTML files will become more favorable the longer the scraper runs, ensuring that any parsing issues can be corrected retroactively without the risk of missing information completely. Locating the content on the live website or the saved HTML files requires an understanding of the webpage structure (i.e., HTML structure), which we can inspect by right-clicking with the mouse anywhere on a webpage and selecting "inspect". Thus, to parse the websites' content, we need to identify which HTML element holds the information we want (Figure 1).

We can have a look at http://dataquestio.github.io/web-scraping-pages/ids_and_classes.html and inspect the website (right-click and "inspect"). The inspector function helps us identify the precise location of the element we click on in the HTML structure. Here, we can identify a tree structure in which we have several elements, such as `<body>`,

`<div>`, or `<p>`. Some elements have attributes, such as `id` or `class`, that can further help us identify elements.

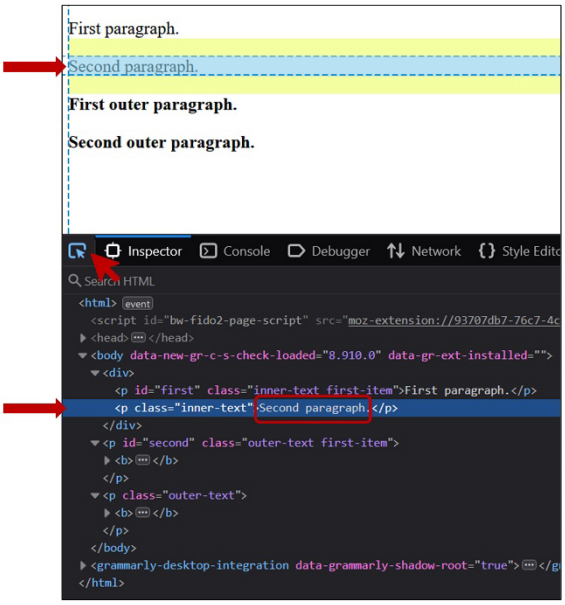

Figure 1. Example of inspecting a website.

The programming of a web scraper changes slightly depending on whether parsing is performed before or after downloading the web content. However, a few steps are essential, such as

- creating a list of websites that are intended to be scraped,
- looping through and downloading the websites' content (with or without parsing), and
- saving the downloaded content in a database or data frame (e.g., SQL, CSV) if you want to analyze the data further.

2.1 Retrieving a website's content with "requests"

We can use a Python package `requests`, a useful HTTP library, to load a website's content into our programming environment. Let's import the package and fetch the website. (Make sure the `requests` package is installed in your environment)

```
import requests
```

With the `get()` function, we can fetch the website through the URL.

```
page = requests.get("http://dataquestio.github.io/web-scraping-
pages/ids_and_classes.html")
```

We can retrieve a response status from `page` with `status_code`. The different codes (numbers) give us information on how the server reacted to our request. A number between 200 and 299 is a success, while numbers such as 400 or 500 are types of server errors. For more information on the meaning of status codes, see: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
page.status_code
```

```
200
```

Success! Let's retrieve some information from the website with content.

```
content = page.content
content
```

```
b'<html>\n    <head>\n        <title>A simple example
page</title>\n    </head>\n    <body>\n        <div>\n
<p class="inner-text first-item" id="first">\n
First paragraph.\n            </p>\n            <p
class="inner-text">\n                Second paragraph.\n
</p>\n        </div>\n        <p class="outer-text first-item"
id="second">\n            <b>\n                First outer
paragraph.\n            </b>\n        </p>\n        <p
class="outer-text">\n            <b>\n                Second
outer paragraph.\n            </b>\n        </p>\n
</body>\n</html>'
```

We can see the entire website content with the HTML layout, tags, elements, etc. We need to parse the website to retrieve the content we want using the package `BeautifulSoup`.

2.2 Parsing the websites' content with "BeautifulSoup"

Let's import the package and load the retrieved content into the package. We can specify the parser type matching the content we are working with. In this case, HTML. (see https://tedboy.github.io/bs4_doc/ for the documentation).

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(content, 'html.parser')
soup
```

```
<html>
<head>
<title>A simple example page</title>
</head>
<body>
<div>
```

```
<p class="inner-text first-item" id="first">
                First paragraph.
            </p>
<p class="inner-text">
                Second paragraph.
            </p>
</div>
[…]
</body>
</html>
```

Using `prettify()`, we can format the output and see its tree structure, making the information more accessible to read.

```
print(soup.prettify())
```

```
<html>
 <head>
  <title>
   A simple example page
  </title>
 </head>
 <body>
  <div>
   <p class="inner-text first-item" id="first">
    First paragraph.
   </p>
   <p class="inner-text">
    Second paragraph.
   </p>
  </div>
[…]
 </body>
</html>
```

The nested tree structure of the tags is now more easily visible.

Finding elements. By using the functions `find()` and `find_all()`, we can find the first, or all, occurrences of an element. For example, we can search for `p` or `div`.

```
soup.find('p')
```

```
<p class="inner-text first-item" id="first">
                First paragraph.
            </p>
```

The same operation is also possible by using a dot.

```
soup.p
```

```
<p class="inner-text first-item" id="first">
                First paragraph.
            </p>
```

`find_all()` works in the same manner, but will return a list.

We can also specify attributes and pass them to the methods. For example, a list of all `p` elements for which the `class` attribute is `"outer-text"`.

```
soup.find_all('p', {'class': "outer-text"})
```

```
[<p class="outer-text first-item" id="second">
 <b>
                First outer paragraph.
            </b>
 </p>,
 <p class="outer-text">
 <b>
                Second outer paragraph.
            </b>
 </p>]
```

We can also search by the attribute `id`. Here, specified as `"first"`.

```
soup.find_all(id="first")
```

```
[<p class="inner-text first-item" id="first">
                First paragraph.
            </p>]
```

Other method of finding elements. BeautifulSoup has the `select()` method which utilizes CSS selectors and will return all matching elements in a list.

See the documentation for all supported CSS selectors, but here are some of the basics.

You can find tags.

```
soup.select("p")
```

```
[<p class="inner-text first-item" id="first">
                First paragraph.
            </p>,
 <p class="inner-text">
                Second paragraph.
            </p>,
 […]
```

You can find tags within a tag.

```
soup.select("div p")
```

```
[<p class="inner-text first-item" id="first">
                First paragraph.
             </p>,
 <p class="inner-text">
                Second paragraph.
             </p>]
```

You can find tags with specific classes.

```
soup.select("p.first-item")
```

```
[<p class="inner-text first-item" id="first">
                First paragraph.
             </p>,
 <p class="outer-text first-item" id="second">
 <b>
                First outer paragraph.
             </b>
 </p>]
```

You can find tags by id.

```
soup.select("#second")
```

```
[<p class="outer-text first-item" id="second">
 <b>
                First outer paragraph.
             </b>
 </p>]
```

You can also find tags through a combination of criteria.

```
soup.select("div p.first-item#first")
```

```
[<p class="inner-text first-item" id="first">
                First paragraph.
             </p>]
```

Retrieving human-readable text. If we want to retrieve human-readable text within a website and elements, we can use the `get_text()` method. It returns all the text in a document or within the specified element as a string.

```
soup.get_text()
```

```
'\n\nA simple example page\n\n\n\n\n                First
paragraph.\n            \n\n                Second paragraph.\n
\n\n\n\n                First outer paragraph.\n
\n\n\n\n                Second outer paragraph.\n
\n\n\n'
```

With BeautifulSoup we can strip whitespaces (including new lines, indicated with \n) from the beginning and end of each bit of text.

```
soup.get_text(strip = True)
```

```
'A simple example pageFirst paragraph.Second paragraph.First
outer paragraph.Second outer paragraph.'
```

We can also specify a string, which will be used to join the texts together.

```
soup.get_text("|", strip=True)
```

```
'A simple example page|First paragraph.|Second paragraph.|First
outer paragraph.|Second outer paragraph.'
```

We now know how to read out and parse the content of a webpage. Next, we will combine the previous methods to scrape more information from websites.

2.3 Scraping book information

In this section, we will scrape information about books from the website https://books.toscrape.com/, a scrape-practicing website. We are interested in scraping information about the books (e.g., title, price, rating) that are presented on the starting page. To understand how we can scrape the information, we need to inspect the website and look at the HTML structure.

We will parse the content while visiting the page before saving information on the local machine. However, saving the HTML website and parsing the content from the file offline would also be possible.

After inspecting the website, we can start by importing the required packages.

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
```

Next, we retrieve the content from the website with `requests` and inspect the response.

```
url = "https://books.toscrape.com/"
page = requests.get(url)
page.status_code
```

```
200
```

We can now load the HTML content into `BeautifulSoup`.

```
soup = BeautifulSoup(page.content,'html')
```

Through the inspection, we know that the titles are stored under the element “h3”. Let’s find all of them and save them in a list.

```
titles = soup.find_all("h3")
titles[0] # inspecting the first list item
```

```
<h3><a href="catalogue/a-light-in-the-attic_1000/index.html"
title="A Light in the Attic">A Light in the ...</a></h3>
```

Our first list item contains the title and unnecessary content and formatting from the website. We can retrieve the raw title through the command `text`. We can neatly wrap the extraction process in a list comprehension, which will iterate through the list, extract the title, and save it in a new list.

```
titles_clean = [title.text for title in titles]
titles_clean[0]
```

```
'A Light in the ...'
```

We now have a list of all the book titles from the starting page. We can repeat the same process for the prices and the ratings. When we inspect the website, we see that the class name of the HTML, that contains the price is “price_color”.

```
prices = soup.find_all("p", {"class": "price_color"})
prices[0]
```

```
<p class="price_color">£51.77</p>
```

Here, we want the text and convert the price into a number, allowing us to make operations on them later. We can achieve that by retrieving the text of the list elements, removing the currency sign (£) through `replace()`, and converting it with `float()`. All are integrated in a list comprehension.

```
prices_clean = [float(price.text.replace("£", "")) for price in
prices]
prices_clean[0]
```

```
51.77
```

We repeat the process with the ratings. However, they are saved not as a text but as a class value.

```
ratings = soup.find_all("p", {"class": "star-rating"})
ratings[0]
```

```
<p class="star-rating Three">
<i class="icon-star"></i>
<i class="icon-star"></i>
[…]
</p>
```

As we can see, the "p" element also contains child elements "I" and has two class values, "star-rating", and "Three". We can retrieve the "Three" as we would access the values of a dictionary and then specify the second value. We also want the rating as a number for which we can install and import the `text2num` package, which can convert text to numbers.

As before, we can wrap our extraction process of each list element in a list comprehension to create a new list with clean values.

```
from text_to_num import text2num

ratings_clean = [text2num(rating["class"][1].lower(), "en") for
rating in ratings]
ratings_clean[0]
```

```
3
```

Great! We can now save our lists into a data frame.

```
results = pd.DataFrame({"Title": titles_clean, "Price":
prices_clean, "Rating": ratings_clean})
results.head()
```

The output is shown in **Table 1**.

Title	Price	Rating
A Light in the ...	51.77	3
Tipping the Velvet	53.74	1
Soumission	50.10	1
Sharp Objects	47.82	4
Sapiens: A Brief History ...	54.23	5

Table 1. Book information

2.4 Finding elements more efficiently

Finding the precise elements for accurate parsing is very time-consuming. Fortunately, selectorlib, a browser plugin with an associated Python package exists that can speed up that process. The browser plugin acts as an overlay that allows us to select any element on a webpage through point-and-click and records the location within the HTML structure as a template. We can load the template into Python and interpret it with the `selectorlib` package, which will help us locate and parse the content we are interested in.

We will use the plugin and show you how to scrape quotes from the scraping practice website https://quotes.toscrape.com with the help of the selectorlib package. Since most of the content we are interested in is scattered across a webpage, we will also show you how to integrate page navigation into the scraping process.

Follow these steps:

- Visit https://selectorlib.com/chrome.html and install the plugin in Chrome.
- Watch the videos on https://selectorlib.com/chrome.html to understand how to use the plugin.
- Create a template that saves the authors, quotes, and tags.
- export the template as a .yml file and name it “Quotes”

Note that the plugin needs permission to see all the content in the browser, and it should be disabled or uninstalled when not used.

Let's install the `selectorlib` package (https://pypi.org/project/selectorlib/) and import the `Extractor` from it.

```
from selectorlib import Extractor
```

Next, we will load the “Quotes” template into our environment with the `Extractor` function and load the websites with `requests.get()`.

```
path = "SPECIFY PATH"
quotes_extract = Extractor.from_yaml_file(path + 'Quotes.yml')

base_url = "https://quotes.toscrape.com"
r = requests.get(base_url)
```

We can now parse the content of our request with our template and extract the information we want.

```
extract = quotes_extract.extract(r.text)
extract
```

```
{'quote': ['“The world as we have created it is a process of
our thinking. It cannot be changed without changing our
thinking.”',
  '“It is our choices, Harry, that show what we truly are, far
more than our abilities.”',
  […]
],
 'author': ['Albert Einstein',
  'J.K. Rowling',
  […]
],
 'tags': ['Tags: change deep-thoughts thinking world',
  'Tags: abilities choices',
  […]
]}
```

The information is stored in a dictionary, which we can save in a data frame.

```
quotes_data = pd.DataFrame({"Quote": extract["quote"], "Author":
extract["author"], "Tags": extract["tags"]})
quotes_data.head()
```

The output is shown in **Table 2**.

With selectorlib, we quickly located and extracted the information. Let’s save it into a CSV. Some of the characters in our text data are incorrectly stored when saved to a CSV. We can set the encoding method to correct it.

```
path = "SPECIFY PATH"
quotes_data.to_csv(path+'Quotes.csv', encoding="utf-8-sig")
```

Using selectorlib will greatly speed up the parsing process. However, the plugin may not always work, depending on the website and if the content is created dynamically, which should be tested.

Quote	Author	Tags
“The world as we have created it is a process ...	Albert Einstein	Tags: change deep-thoughts thinking world
“It is our choices, Harry, that show what we t...	J.K. Rowling	Tags: abilities choices
“There are only two ways to live your life. On...	Albert Einstein	Tags: inspirational life live miracle mir-acles
“The person, be it gentleman or lady, who has ...	Jane Austen	Tags: aliteracy books classic humor
“Imperfection is beauty, madness is genius and...	Marilyn Monroe	Tags: be-yourself inspirational

Table 2. Parsed quotes with authors and tags.

2.5 Integrating page navigation

For our project, we are interested in all quotes on the website. Unfortunately, they are stored on different pages. We need to navigate to all the pages containing quotes and make new requests for each of them to retrieve the quotes. In our case, we need to scrape ten pages to retrieve all quotes.

Pagination. We could manually create a list of all page URLs by copy-pasting the website’s URLs into our programming script. However, such an approach is time-consuming and should be automated if possible. Alternatively, we can take the base URL (e.g., https://quotes.toscrape.com) and generate the remaining URLs automatically. Similar to inspecting the website’s HTML structure, understanding the URL structure is essential. Usually, the URL follows a clear pattern for pagination and other common operations (e.g., selecting the size and color of products on a shopping platform). Thus, after some testing (i.e., clicking through the webpage), a pattern typically emerges that we can utilize to generate the URLs automatically (e.g., looping through pages of search results). Another practice is to automatically find all links on the website of the current URL and add them to the list of URLs to be scraped. That way, the entire webpage can be collected, but managing duplicates becomes important to avoid unnecessary scraping.

For the website of quotes, we found that pages are easily selected by adding “/page/PAGENUMBER/” to the base URL. For example: https://quotes.toscrape.com/page/2/ for the second page. We can exploit that pattern and exchange the page number in a loop.

To alleviate the server load we also implement artificial pauses after each request with the `sleep` function.

Continuously saving data. After determining how to obtain the necessary URLs, the scraper should save the website’s content (which will be parsed later) or parsed information from each URL request. In either case, the data should be saved in a local folder structure, allowing easy (automated) access to the data in later stages of the project.

Locally saving the data may include unparsed (e.g., HTML files) or parsed (e.g., CSV or Excel files) data. The data should be continuously saved while scraping the content from the website to prevent unnecessary computer memory usage and data loss if the scraping procedure unexpectedly stops (e.g., due to errors in the programming scripts, internet, or server issues). Once the web data collection is completed, the data can be parsed into a data frame or merged if already parsed.

Here, we will save the parsed content when collecting quotes from the website.

We start by importing the required packages and defining the URL we want to work with and the path where we want to save the data.

```
from time import sleep
base_url = "https://quotes.toscrape.com/page/"
save_path = "SPECIFY PATH"
```

We take the same approach as before, extracting the information with selectorlib and saving the data as a CSV file. The only difference is that we wrap these steps into a loop that starts at 1 and continues until 10, representing the pages. The increasing numbers can be integrated into the URL for our request and the saved filename. We add a sleep timer of 1 second after each loop.

```
for page in range(1,11):
    URL = base_url + str(page)
    r = requests.get(URL)
    extract = quotes_extract.extract(r.text)
    quotes_data = pd.DataFrame({"Quote": extract["quote"],
"Author": extract["author"], "Tags": extract["tags"]})
    quotes_data.to_csv(path+'Quotes_' + str(page) + '.csv',
encoding="utf-8-sig")
    print(str(page) + ". page is processed")
    sleep(1)
```

Lastly, we need to merge the data from our files. The os package helps us to locate and list files in a folder. To ensure we only merge relevant files, we specify that only files that start with “Quotes_” and end with “.csv” are used.

```
import os
data_frames = []
csv_files = [file for file in os.listdir(save_path) if
file.startswith('Quotes_') and file.endswith('.csv')]
csv_files
```

```
['Quotes_1.csv',
 'Quotes_10.csv',
 […]
 'Quotes_9.csv']
```

After listing the correct filenames, we need to load each file into a data frame and append them into a list. Then, we can merge the list of data frames with `concat`. Since each file has its index, which was integrated into the "Unnamed: 0" column, we drop it.

```
for file in csv_files:
    file_path = os.path.join(save_path, file)
    quote_data = pd.read_csv(file_path)
    data_frames.append(quote_data)

all_quotes = pd.concat(data_frames,
ignore_index=True).drop("Unnamed: 0", axis=1)
all_quotes.head()
```

The output is structurally the same as in **Table 2** but contains poem data from all ten pages.

We now have our final data. We can also save it into a single CSV file.

```
all_quotes.to_csv(save_path + "all_quotes_.csv", index=False,
encoding="utf-8-sig")
```

3 Conclusion

In this guide, we have illustrated several approaches to scraping information from a static webpage and discussed what to consider when navigating the website and saving its content. The methods we presented, especially how to create URLs dynamically and how to best save the data, will allow us to scale up for bigger scraping projects. However, we need to verify that scraping the website is allowed and that we implement pauses to avoid overburdening the servers.

Before implementing larger scraping projects, small test runs should be conducted to estimate the time required for the entire scraping process and the required data storage capacity. We now have the tools and resources to build a scraper for static websites, including pagination and how to store data, avoiding unnecessary information loss in case of unforeseen errors.

References

Dewi, L. C., Meiliana, & Chandra, A. (2019). Social media web scraping using social media developers API and Regex. *Procedia Computer Science*, *157*, 444–449. https://doi.org/10.1016/j.procs.2019.08.237

Whittaker, Z. (2022, April 18). *Web scraping is legal, US appeals court reaffirms*. TechCrunch. https://techcrunch.com/2022/04/18/web-scraping-legal-court/

All references and links were retrieved on June 15, 2024.

About the author(s)

Felix Soldner is a post-doctoral researcher at GESIS – Leibniz Institute for the Social Sciences in Cologne, Germany in 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.

N. Gizem Bacaksizlar Turbic is a post-doctoral researcher at GESIS – Leibniz Institute for the Social Sciences in Cologne, Germany in the Computational Social Science department. Her research areas include social and political networks, and social media analysis.

Pouria Mirelmi is a master's student in Computational Social Science at RWTH Aachen University, and a student assistant at GESIS – Leibniz Institute for the Social Sciences. His current research focus is Social Network Analysis and Large Language Models.

Suggested citation

Soldner, F., Bacaksizlar Turbic, N. G., & Mirelmi, P. (2024). *How to collect online data through web-scraping* (GESIS Guides to Digital Behavioral Data, 10). 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