

Be part of a better internet. [Get 20% off membership for a limited time](#)



From zero to hero: a basic guide into Web Scrapping!



Sofia Tello

10 min read · Sep 2, 2023



1



Yeah! I got that phrase from Hercules. If you'd like to sing along with the muses before we start, help yourself:

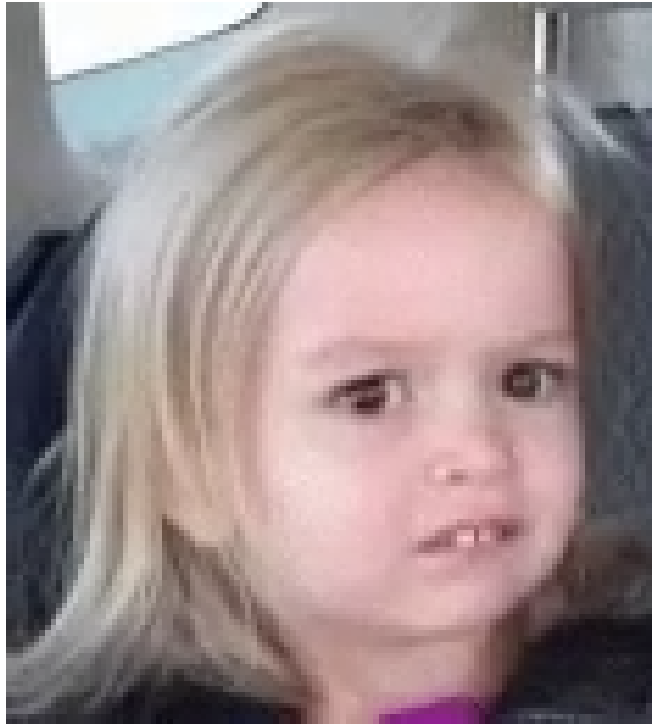
Hercules | Zero to Hero | Lyric Video | Disney Sing Along



When I started to write this post I was understanding literally ZERO about what I was doing. It all began with me trying to learn about Data Scraping and wanting to die. Thankfully, Hercules empowered me and now here I am! So, for you to not feel as lost as I did, I'll try to do a simple step-by-step guide. Let's do it!

Web Scraping

I'm not gonna lie, data scraping sucks, but we need it!!!



Yeah! I had the same face when my Data Science Bootcamp instructor named his presentation: “**Web Scraping. Find unique data through immense suffering**”. I did suffer a lot, but the truth is that it wasn’t as hard as it looked like. So let’s keep going!

As you can see on [Wikipedia](#):

Data scraping is a technique where a computer program extracts data from human-readable output coming from another program.

Particularly, **web scraping** is just collecting data from a website.

- **What for?** The data you gather can come from *anywhere* you want (as long as you ask for permission) and it can be used to obtain meaningful insights or simply to get info that you're interested in, but instead of doing copy-pastes, you can write a piece of code that would do this work for you.
- **How?** Well, that's what I'll try to show you in the following lines.

The fundamentals on HTML code

First, let's see how **html code** looks like; understanding its structure is essential since this is where we're extracting the data from.

```
html_doc = """
<!DOCTYPE html>
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>
```

```
<p class="story">Once upon a time there were three little sisters; and
their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
</html>
"""
```

Weren't we extracting data from a website? That's correct, this is a simplified example of HTML code that represents the structure of a web page. When you request its information, it will be typically delivered in this format, from which you're going to extract only what you need.

Let's review its structure:

- `< >`, `</>`: the items you see highlighted in pink, between the **angle brackets**, are directives to the browser, meaning they're telling something to it. So `<title>` defines the beginning of the element called 'title' and `</title>` the end of it. These elements could also tell the browser to show the element in **bold** letters or to make it a numbered list, but for our purpose, the **important thing to understand**, is that we're trying to

extract what's actually inside of these elements and identifying them will make it easier for us to know where to look and what to call in our code. In this way, these elements are going to function as tags.

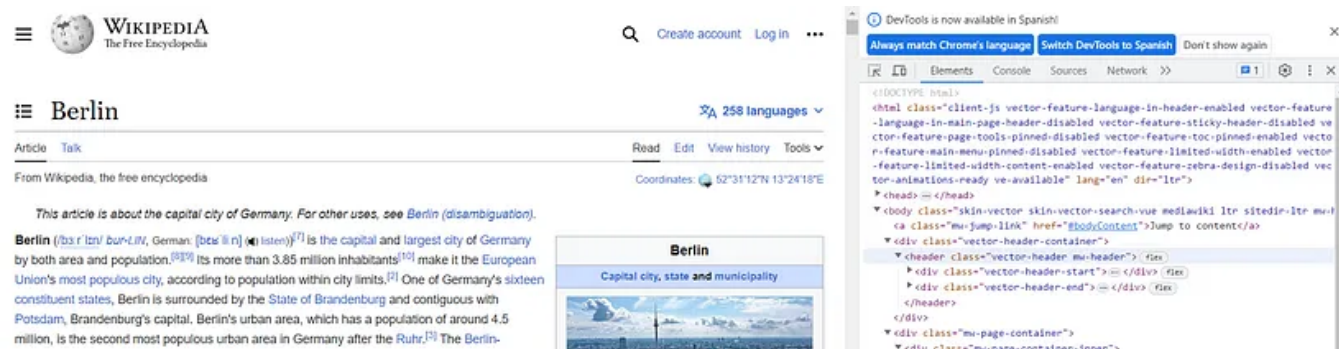
- **Tree structure:** as you can see, these `<>` **elements** are also contained inside of other ones. `<title>` is contained within `<head>` which is contained inside of `<html>`; we could then say that `<title>` is the *child* of `<head>`, who is the *child* of `<html>`. This is also important, since these parent-child relations can also work as tags in our code.
- **Attributes in tags:** inside `<p>` or `<a>` tags, there are some words highlighted in yellow, in this case, **class**, **href**, and **id**. They provide extra details about an element, such as its class, ID, or link destination. But most importantly, they can also be useful to look for information.
- **The data:** the strings that appear in black are the actual data that we're trying to extract. So, if I called for the tag 'title' I would get 'The Dormouse's story'.

→ **In summary** ← web scraping involves finding these specific tags, attributes and data within the HTML structure.

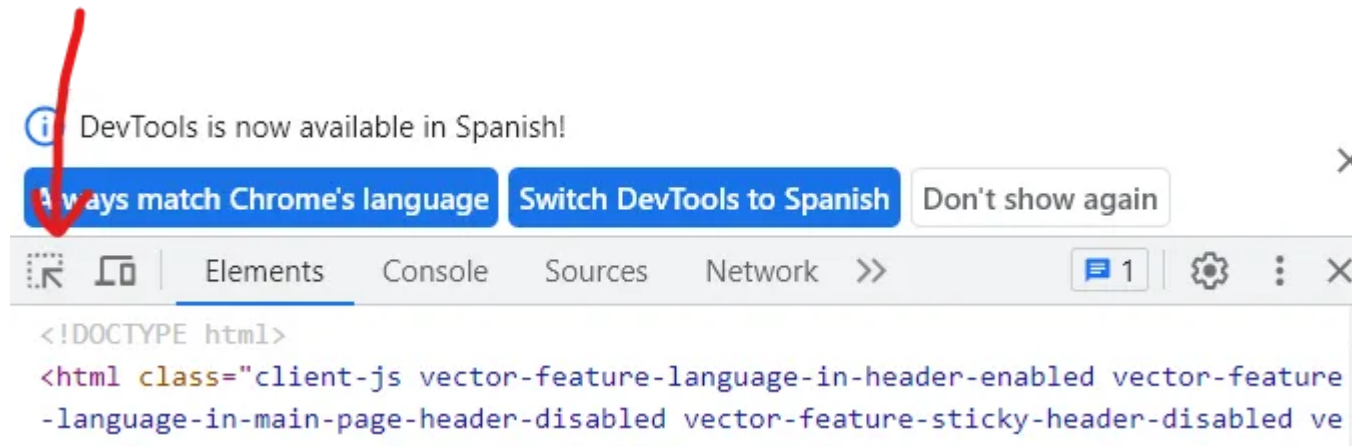
Other tools to understand the HTML Code

The example is very simple, but in real life (as usual), the process of extracting data can become tricky. Web pages are usually not just plain text, they can have complex structures that make it difficult to scrap through them. So, before getting into coding, let's see the HTML Code directly on a website.

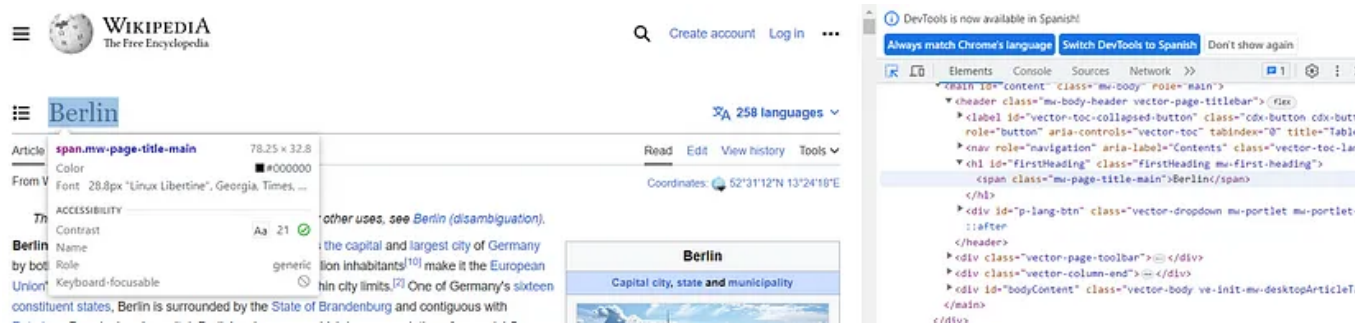
1. Open the website of your preference. In this case I'm using Wikipedia in Chrome.
 2. **Option 1:** press right click → select 'inspect' on the menu
Option 2: hit f12 button on your keyboard
- Something like this DevTools window should appear on the side of your screen:



- Notice that, when placing your cursor through each element of the code, its correspondent output on the webpage will be highlighted.
- Alternatively, you can click on the icon that appears on the top left of the DevTools window:



- And then, when you go through the web page, each elements correspondent piece of HTML code will be highlighted on the DevTools window.



This way you'll *always* know what tags to use in your code, the attributes, the tree structure and the parent-child relations!

Let's code!!

The time to **start coding** has come! In my case I used **Python** via Google Colab Notebooks.

1. Tools: BeautifulSoup, Pandas and Requests

Before we start scraping, we need to make sure we have the necessary tools.

- For starters, I used **BeautifulSoup**, which is a Python library that makes web scraping easier, provides a firm foundation to build upon and covers most of the web scraping tasks you'll encounter. If you're interested,

there are also other ones as Scrapy and Selenium. To install it, we have to run the following code on our sheet:

```
!pip install --upgrade beautifulsoup4
from bs4 import BeautifulSoup
import pandas as pd
import requests
```

- **Requests** is an essential library as well, since we'll need to submit an **http request** in order to **get** the content of a url (the HTML code of the webpage).
- Although we won't be using it in our example, **Pandas** is also a tool that we'd like to work with to be able to store the data, give it some structure and gain some insights.

2. Make sure you're allowed to extract data

The `robots.txt` file is like a guide for web robots (or "bots") visiting a website. It tells these bots which parts of the website they are allowed to visit

and which parts they should avoid. However, it's more like a suggestion rather than a rule. Still, it is important to understand it. So, do as follows:

- **Step 1:** Open a web browser of your choice (Google Chrome, Firefox, Safari, etc.).
- **Step 2:** In the URL bar, type the website's address, followed by “/robots.txt”. For example, for Wikipedia, you would type in “<https://www.wikipedia.com/robots.txt>” and press Enter.

```
# robots.txt for http://www.wikipedia.org/ and friends
#
# Please note: There are a lot of pages on this site, and there are
# some misbehaved spiders out there that go _way_ too fast. If you're
# irresponsible, your access to the site may be blocked.
#

# Observed spamming large amounts of https://en.wikipedia.org/?curid=NNNNNN
# and ignoring 429 ratelimit responses, claims to respect robots:
# http://mj12bot.com/
User-agent: MJ12bot
Disallow: /

# advertising-related bots:
User-agent: Mediapartners-Google*
Disallow: /

# Wikipedia work bots:
User-agent: IsraBot
Disallow:

User-agent: Orthogaffe
Disallow:

# Crawlers that are kind enough to obey, but which we'd rather not have
# unless they're feeding search engines.
User-agent: UbiCrawler
Disallow: /
```

What does this mean?:

- “User-agent:” mentions the web robot(s) for whom the rule(s) applies.
- “Disallow:” tells robots which directories not to access.
- “Allow:” tells robots which directories to access.

In the absence of “Allow” or “Disallow” directives, the default is to allow access. So, if a robots.txt file only specifies “Disallow” directives, robots can assume they’re allowed to access any part of the site not explicitly disallowed.

3. Make a Soup

So, now that we know we can access the website safely, we can make a soup!

- First, we need to download the html code with a **get request**. For this example, we’re going to get the code for the [Wikipedia page of Berlin](https://en.wikipedia.org/wiki/Berlin).
- Then, we’re going to check for the **status code**, it **MUST** be **200**, otherwise it means that there’s a problem! (You can check [this page](#) to look for the meaning of different errors).

```
url = 'https://en.wikipedia.org/wiki/Berlin'  
response = requests.get(url)  
response.status_code
```



200
OK

via [http.cat/status/200](http://cat/status/200)

- Now that you know everything works, you can make your soup. For this, we'll use **BeautifulSoup** plus the string that we got when requesting for the url, which in this case is saved in **response**.

- This soup object that we're creating will help us to look at the string in a human-readable way. And also, with the soup we can actually go through the HTML code using its tags.
- And, of course, we use *html.parser* because our source is HTML.

```
soup = BeautifulSoup(response.content, "html.parser")  
soup
```

4. Get the data!

This is meant to be a *super simple* example. So let's say we're trying to get Country, Population and Coordinates of Berlin. For this, the easiest way to go is to use the DevTools window in our Wikipedia page as a reference.

The image shows a web browser displaying the Wikipedia page for Berlin. On the right side, the Chrome DevTools 'Elements' panel is open, showing the DOM tree. A red arrow points from the 'Country' field in the Berlin infobox on the page to the corresponding HTML code in the DevTools. The HTML code shows a table with a row for 'Country' containing the value 'Germany'.

Country

Country	Germany
---------	---------

As we can see in the image, the country name is contained in a box that's displayed on the right of the page.

- Germany is contained in a `<td>` element.
- This `<td>` element is contained in a `<tr>` element.
- The `<tr>` element is contained in a `tbody` element.
- And all of that is inside of a table.
- This table has an attribute `class = "infobox ib-settlement vcard"`.

With this info we can establish a route to look at; for this we can use the **.select** method.

Ok, but not so fast. In this case, just using the route won't get us far.

- If we just do **soup.select('table.infobox tbody tr th')** we will get all of the code that corresponds to the infobox table.
- We could also use the **.text** or **.get_text()** functions to be able to look at the actual displayed text and not just the code.
- But, since there are different **th** elements, we'll need a **for loop** to iterate through each of them and print their text.

```
for s in soup.select('table.infobox tbody tr th'):
    print(s.text) # or .get_text()
```

Now, we can look at the text!

```
1 for s in soup.select('table.infobox tbody tr th'):
2     print(s)
```

```
<th class="infobox-above" colspan="2"><div class="fn org">Berlin</div></th>
<th class="infobox-label" scope="row">Country</th>
<th class="infobox-label" scope="row"><a href="/wiki/States_of_Germany" title="States of Germany">State</a></th>
<th class="infobox-header" colspan="2">Government<div class="ib-settlement-fn"></div></th>
<th class="infobox-label" scope="row">• Body</th>
```

```
1 for s in soup.select('table.infobox tbody tr th'):
2     print(s.get_text())
```

```
Berlin
Country
State
Government
• Body
```

If we look at the table at the Wikipedia page, we can see that Germany is contained next to Country, and looking at the code they seem to be siblings.

Now we can modify our **for loop** with a condition that extracts the Country's sibling info with the **.next_sibling** function; that'll finally give us the name of the country.

```
# Country

for s in soup.select('table.infobox tbody tr th'):
    if s.text == 'Country':
        print(s.next_sibling.get_text())
        # or print(s.next_sibling.text)
```

We could've also used the direct route to Germany, using its tag which is **td** as we figured out earlier, but the first method would be better if we wanted to somehow create a function to extract the countries for a lot of cities. Still, the 'direct' way to do this would be:

```
# Country

for s in soup.select('table.infobox tbody tr td'):
    if s.text == 'Germany':
        print(s.text)
```

Now we can do the same for the coordinates and the population!

→ **Coordinates**

```
# Coordinates

for s in soup.select('table.infobox tbody tr td'):
    if s.text.startswith('Coordinates'):
        print(s.text.split('/')[1].split(';'))
```

- For **coordinates**, we can use **.startswith** to search for “Coordinates”, since we know that the numbers that we’re looking for are contained in the same string. As we can see here:

```
1 for s in soup.select('table.infobox tbody tr td'):
2     print(s.text)
```

Capital city, state and municipality

Spree river, Museum Island and Berlin TV Tower in MitteVictory ColumnCharlottenburg P

FlagCoat of arms

Show BerlinShow Europe

Coordinates: 52°31′12″N 13°24′18″E﻿ / ﻿52.52000°N 13.40500°E﻿ / 52.52000; 13.40500

Germany

- We can also use the **.split** method to extract only the coordinates that appear in decimal format (or the ones we're interested in).

→ Population

Now the population is a tricky one because the info we're interested in is at another level.

Population (2021) ^[2]	
• City/State	
• Density	
• Urban ^[3]	
• Urban density	
• Metro ^[4]	6,144,600
• Metro density	201/km ² (520/sq mi)
Demonyms	Berliner(s) (English)

ACCESSIBILITY	
Contrast	Aa 19.92 ✓
Name	6,144,600
Role	gridcell
Keyboard-focusable	⊗

```
<tr class="mergedrow">...</tr>
<tr class="mergedrow">...</tr>
<tr class="mergedrow">...</tr>
<tr class="mergedrow">...</tr>
<tr class="mergedrow">
  <th scope="row" class="infobox-label">...</th>
  <td class="infobox-data">6,144,600</td>
</tr>
<tr class="mergedrow">...</tr>
```

The image shows how the metropolitan population is contained within 'Metro' and 'Population', which is at the same time inside of the infobox table.

```
1 for s in soup.select('table.infobox tbody tr th'):  
2     print(s.text)
```

```
Berlin  
Country  
State  
Government  
• Body  
• Governing Mayor  
• Bundesrat votes  
• Bundestag seats  
Area[1]  
• City/State  
• Urban  
• Metro  
Elevation  
Population (2021)[2]  
• City/State  
• Density  
• Urban[3]  
• Urban density  
• Metro[4]  
• Metro density
```

So how to get there? We can navigate through the same way as before: using **for loops** and **if statements**.

- In this case we're using **.parent** to access the parent element of the current `<th>` element. In this context, it refers to the `<tr>` (table row) element that contains the `<th>` element. This step is necessary because the goal is to navigate to the row that contains the "Population" header.

```
for s in soup.select('table.infobox tbody tr th'):
    if s.text.startswith('Population'):
        for sibling in s.parent.find_next_siblings():
            if 'Metro' in sibling.text:
                print(sibling.text)
```

```
Metro[4]6,144,600
Metro density201/km2 (520/sq mi)
```

And now we actually know where to extract the population data we need!

```
# Population

for s in soup.select('table.infobox tbody tr th'):
```

```
if s.text.startswith('Population'):
    for sibling in s.parent.find_next_siblings():
        if 'Metro' in sibling.text:
            print(sibling.select('td')[0].text)
```

And, we're done!

Awesome! Now we have the info we needed. This may seem as a very unpractical way to extract info, and it is true that if there's a change in the structure of the url you're scraping from, your code will be useless. Still, web scraping has amazing possibilities, such as:

- An online retailer might use web scraping to monitor competitor prices and adjust their own accordingly to stay competitive.
- In product comparison sites, data about product details, prices, and descriptions are extracted to provide consumers with the best deals.
- In marketing, web scraping can help identify trends, track user engagement, and understand consumer behavior by analysing data from social media platforms, online forums, and review sites.
- In the recruitment sector, HR companies can scrape job sites to find potential matches for their open positions.

- In finance, firms can use web scraping to pull financial data from various sources to make investment decisions.
- Businesses can use web scraping to track their online reputation by scraping data from online reviews and feedback.

Cool, right?!

Other resources

If you enjoyed web scraping (or you're completely lost), you may find this [Codecademy](#) course useful. You can also find more info on the possibilities of [BeautifulSoup](#) in [this link](#).

Thanks for reading ;)

Web Scraping

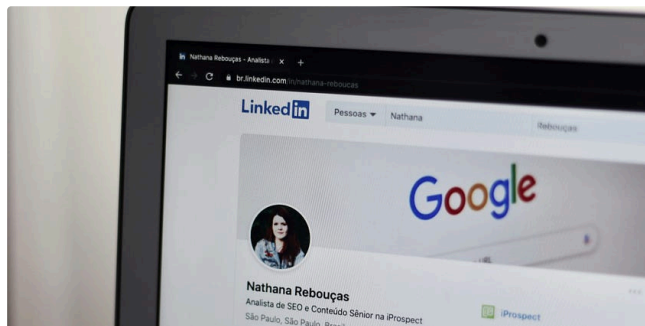


Written by Sofia Tello

1 Follower


Edit profile

Recommended from Medium



 Edward Jones



 Yennhi95zz in Data And Beyond