Web Scraping Cheat Sheet (2022), Python for Web Scraping

The complete guide to web scraping: Beautiful Soup, Selenium, Scrapy, XPath, and more!

The PyCoach Follow 21 min read · Aug 5, 2021

Image by author

Web Scraping is the process of extracting data from a website. Although you only need the basics of Python to start learning web scraping, this might sometimes get complicated because web scraping goes beyond Python. It also involves learning the basics of HTML, XPath, and tons of new methods in Beautiful Soup, Selenium, or Scrapy.

This is why I decided to create this cheat sheet — to walk you through all the stuff you have to learn to successfully scrape a website with Python. This goes from the basic Python stuff you need to know before learning web scraping to the most complete web scraping framework, Scrapy.

Below, you'll find the topics covered. I'm leaving the basic Python stuff at the end of the article since most of you are probably familiar with it. In case you're an absolute beginner start with that section and then follow the order of the list below to easily learn web scraping.

Table of Contents

Important Note: A few months ago Selenium 4 was released. There are only a few changes between Selenium 3.x versions and Selenium 4. You can check out those changes in my web scraping cheat sheet in PDF format.

HTML for Web Scraping

A typical step in web scraping is inspecting an element within a website in order to obtain the HTML code behind it. This is why you should learn HTML basics before learning any Web Scraping library.

In this section, we will see the basic HTML stuff required for web scraping.

HTML Element Syntax

An HTML element usually uses tags and attributes for its syntax. The tags define how your web browser must format and display the content. Most HTML elements are written with an opening tag and closing tag, with content in between. The attributes define the additional properties of the element.

Let's imagine we're about to scrape a website that contains the transcript of movies, so you inspect a movie's title. The HTML element behind the title will look like the picture below.

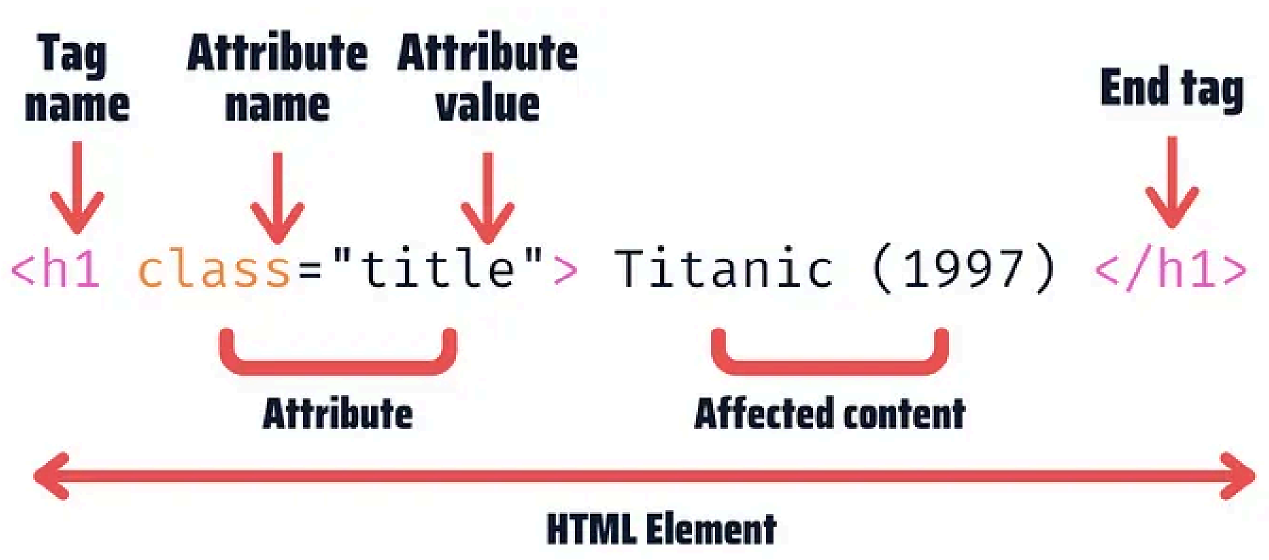


Image by author

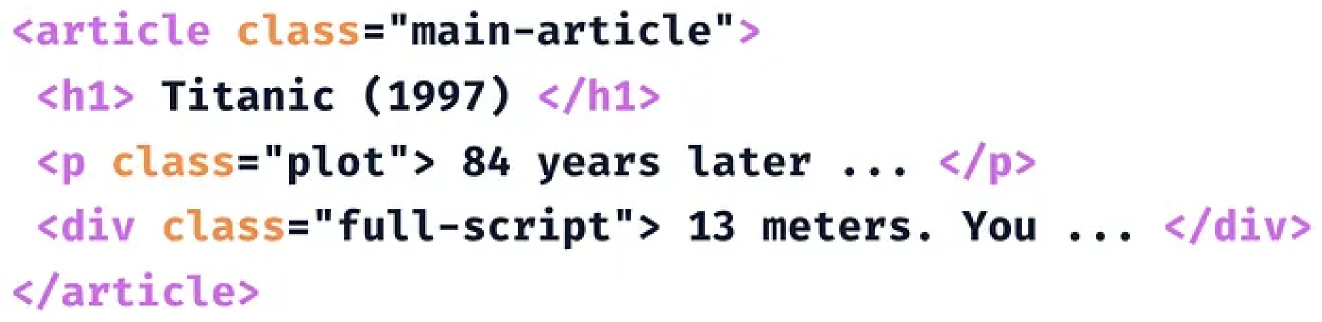
Let's see each element. First, the tag name is set to `h1`. This will give the word `Titanic (1997)` the biggest heading size of a page. Some other common tags are `a`, `p` and `div`. Then we have the attribute name set to `class`, which is one of the most common attributes. The last element is the content

`Titanic (1997)`. This is the only element you see on the page before inspecting.

HTML Code Example

So far we've seen just a single element HTML element, but the HTML code behind a website has hundreds of elements like the image above. Fortunately, when scraping a website we will only analyze the elements that contain the data we want to get.

To easily understand HTML code, let's check the following small code I wrote as an example.



```html
<article class="main-article">
 <h1> Titanic (1997) </h1>
 <p class="plot"> 84 years later ... </p>
 <div class="full-script"> 13 meters. You ... </div>
</article>
```

Image by author

This code represents a web that has the title, plot, and transcript of the movie Titanic. Each element has a tag name and most of them have an attribute. The HTML code is structured with "nodes". There are element nodes, attribute nodes, and text nodes. This might be hard to identify in plain code, so we'll see them much better in a tree structure.

HTML Tree Structure

The image below is the tree structure of the HTML code example we've seen before.

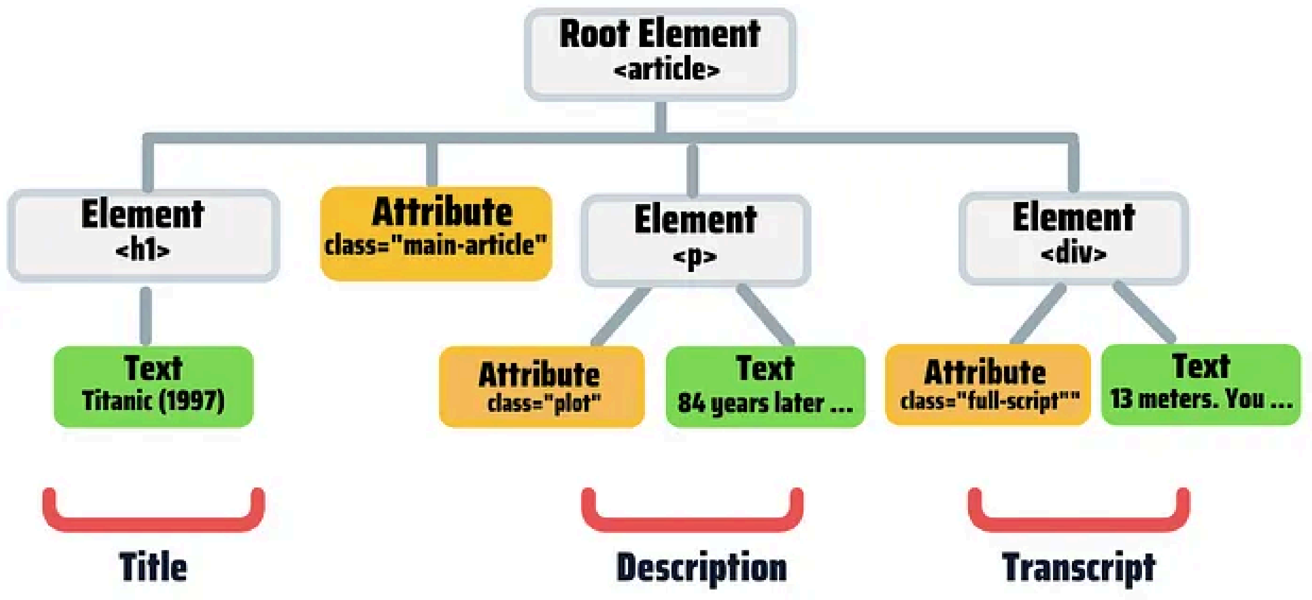


Image by author

In the tree above, each rectangle represents a node. The gray rectangles represent element nodes, the yellow rectangles represent attribute nodes and the green rectangles represent text nodes. Also, the tree shows hierarchical relationships between nodes (parent, child, and sibling nodes). Let's identify the relationships between nodes.

- The "root node" is the top node. In this example, `<article>` is the root.
- Every node has exactly one "parent", except the root. The `<h1>` node's parent is the `<article>` node.
- An element node can have zero, one, or several "children," but attributes and text nodes have no children. `<p>` has two child nodes, but no child elements.
- "Siblings" are nodes with the same parent (e.g., `h1`, `p` and `div`)
- A node's children and its children's children are called its "descendants". Similarly, a node's parent and its parent's parent are called its "ancestors".

When doing web scraping, locating children and parent nodes is sometimes vital when you can’t find a particular element but only its parent or child. We will see this in detail in the XPath section.

Beautiful Soup

Beautiful Soup is the easiest web scraping tool in Python. Although it has some limitations (e.g., it doesn’t scrape Javascript-driven websites), it should be the starting point for beginners.

Installing the libraries

To start working with Beautiful Soup we need 3 libraries. We use “beautifulsoup4” to scrape the website, “requests” to send requests to the website and “lxml” for parsing XML and HTML. Open up a terminal and run the following commands.

```
pip install beautifulsoup4
pip install requests
pip install lxml
```

Importing the libraries

After installing the necessary libraries, import `BeautifulSoup` and `requests` before scraping the website.

```python
from bs4 import BeautifulSoup
import requests
```

Creating the “soup”

In Beautiful Soup we use the "soup" object to find elements in a website. To create this object do the following.

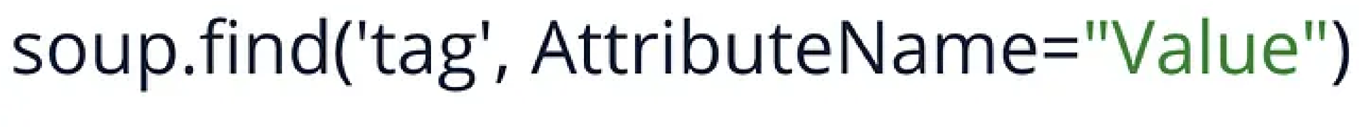
```
# 1.Fetch the pages (write the website you wish to scrape within
parentheses)
result = requests.get("www.google.com")

# 2.Get the page content
content = result.text

# 3. Create the soup
soup = BeautifulSoup(content, "lxml")
```

What we've done so far is what we always do regardless of the website you wish to scrape.

Finding elements: find() vs find_all()

There are two ways to get elements with Beautiful Soup: `find()` and `find_all()`. We use `find()` to get the first element that matches a specific tag name, class name, and id, while `find_all()` will get all the elements that matched and put them inside a list.

Both `find()` and `find_all()` have a similar syntax. Let's have a look.

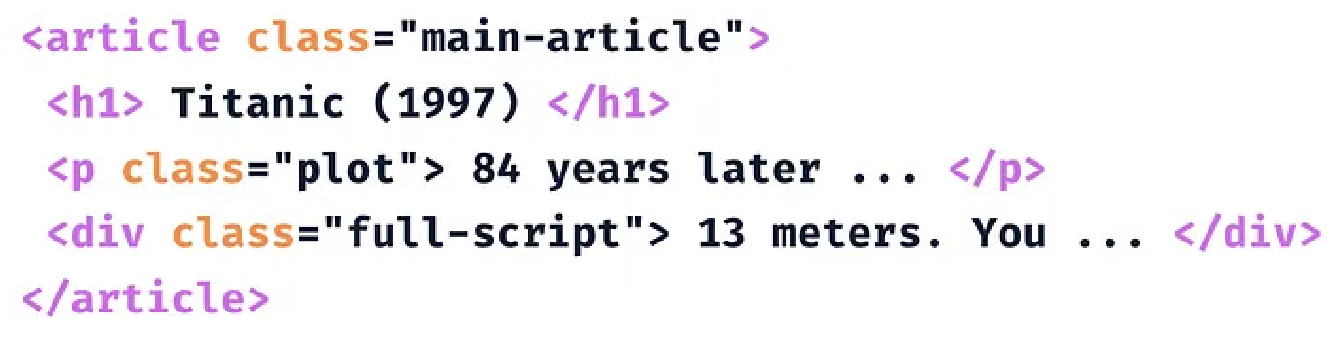


```
soup.find('tag', AttributeName="Value")
```

Image by author

The `find()` and `find_all()` usually takes 2 arguments, but we can omit any of them if necessary. Also, we should use "class_" whenever we want to locate an element by its class name. The "_" is only to make this argument different from Python's `class` keyword. Another common attribute used to locate elements is `id` because it represents unique elements.

Let's look at some examples of how to locate elements with Beautiful Soup. We'll be using the HTML code we've seen before.



```
<article class="main-article">
 <h1> Titanic (1997) </h1>
 <p class="plot"> 84 years later ... </p>
 <div class="full-script"> 13 meters. You ... </div>
</article>
```

Image by author

Let's locale the article element and title.

```
# Get the article element
element1 = soup.find('article', class_="main_article")

# Get the title element
element2 = soup.find('h1')
```

Let's imagine there are multiple h2 elements. We can get all of them with `find_all()`

```
# Get all h2 elements
elements = soup.find_all("h2")
```

We could also use `find_all()` in the examples we used for `find()` but we would've obtained a list with a single element.

Getting the values: text vs get_text()

Most of the time we want to get the text inside an element. There are 2 options to get the text in Beautiful Soup: `text` and `get_text()`. The first is a property while the second is a function. Both return the text of a tag as a string but with `get_text()` we can also add various keyword arguments to change how it behaves (e.g., `separator`, `strip`, `types`)

Let's look at some examples using the "element2" we got before.

```
data = element2.text
data = element2.get_text(strip=True, separator=' ')
```

In this particular example, `text` and `get_text()` will return the same text "Titanic (1997)." However, if we're scraping "dirty" data, the `strip` and `separator` arguments will come in handy. The first will get rid of leading and trailing spaces of the text, while the second adds a blank space as a separator (this will replace a newline '\n' for example)

We can also get a specific attribute of an element — like the `href` attribute within an `a` tag (the `href` will help us get the link of the element)

```
# Get the "a" tag
element = soup.find('a')

# Get the attribute value
data = element.get('href')
```

Below you can find a guide that will help to scrape your first website with Beautiful Soup.

How To Easily Scrape Multiple Pages of a Website Using Python

A simple guide to extracting data from websites with Python

betterprogramming.pub

XPath (Necessary for Selenium and Scrapy)

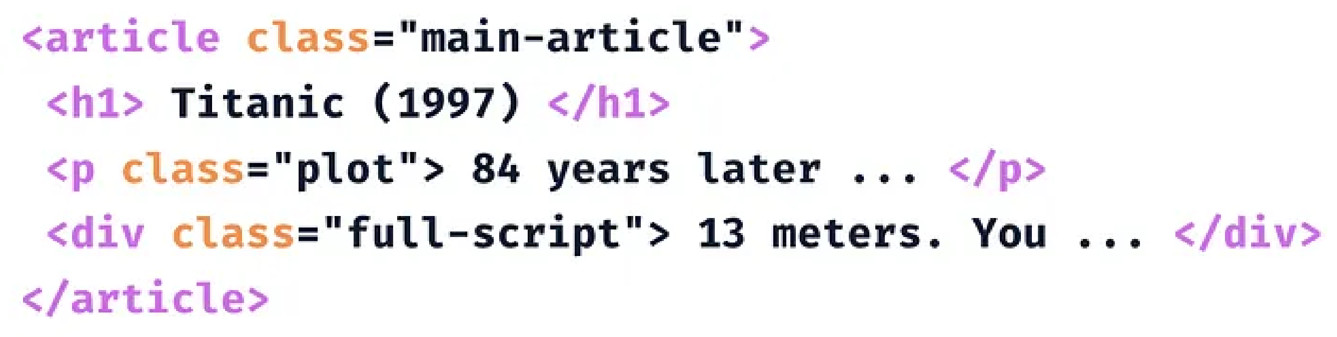
Before learning Selenium or Scrapy, we have to learn how to build an XPath. XPath is a query language for selecting nodes from an XML document. This will help us locate an element when the HTML code isn't simple.

XPath Syntax

An XPath usually contains a tag name, attribute name, and attribute value. Let's have a look at the XPath syntax.

```
//tagName[@AttributeName="Value"]
```

The `//` and `@` are special characters that we'll see later. Now let's check some examples to locate some elements of the HTML code we've been using so far.



```
<article class="main-article">
 <h1> Titanic (1997) </h1>
 <p class="plot"> 84 years later ... </p>
 <div class="full-script"> 13 meters. You ... </div>
</article>
```

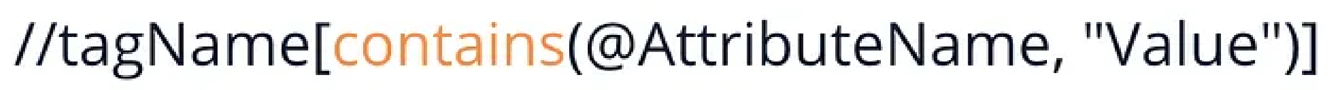
Image by author

Let's build the XPath of the article element, title, and transcript.

```
# Article element XPath
//article[@class="main-article"]

# Title element XPath
//h1

# Transcript element XPath
//div[@class="full-script"]
```

XPath Functions and Operators

Sometimes the HTML elements are complicated to locate with a simple XPath. This is when we need to use XPath functions. One of the most useful functions is `contains`. The `contains` function has the following syntax.



```
//tagName[contains(@AttributeName, "Value")]
```

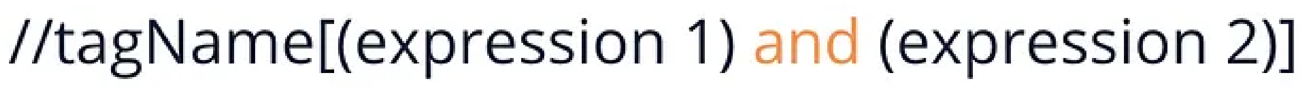
Image by author

Let's look at some examples with the HTML code we've used above.

```
# Article element XPath
//article[contains(@class, "main")]

# Transcript element XPath
//div[contains(@class, "script")]
```

As you can see, we don't need to write the whole value, but only a part of it. This is extremely useful when working with long values or attributes that have multiple value names.

On the other hand, XPath can also use `and` and `or` logical operators. Both have the same syntax.



```
//tagName[(expression 1) and (expression 2)]
```

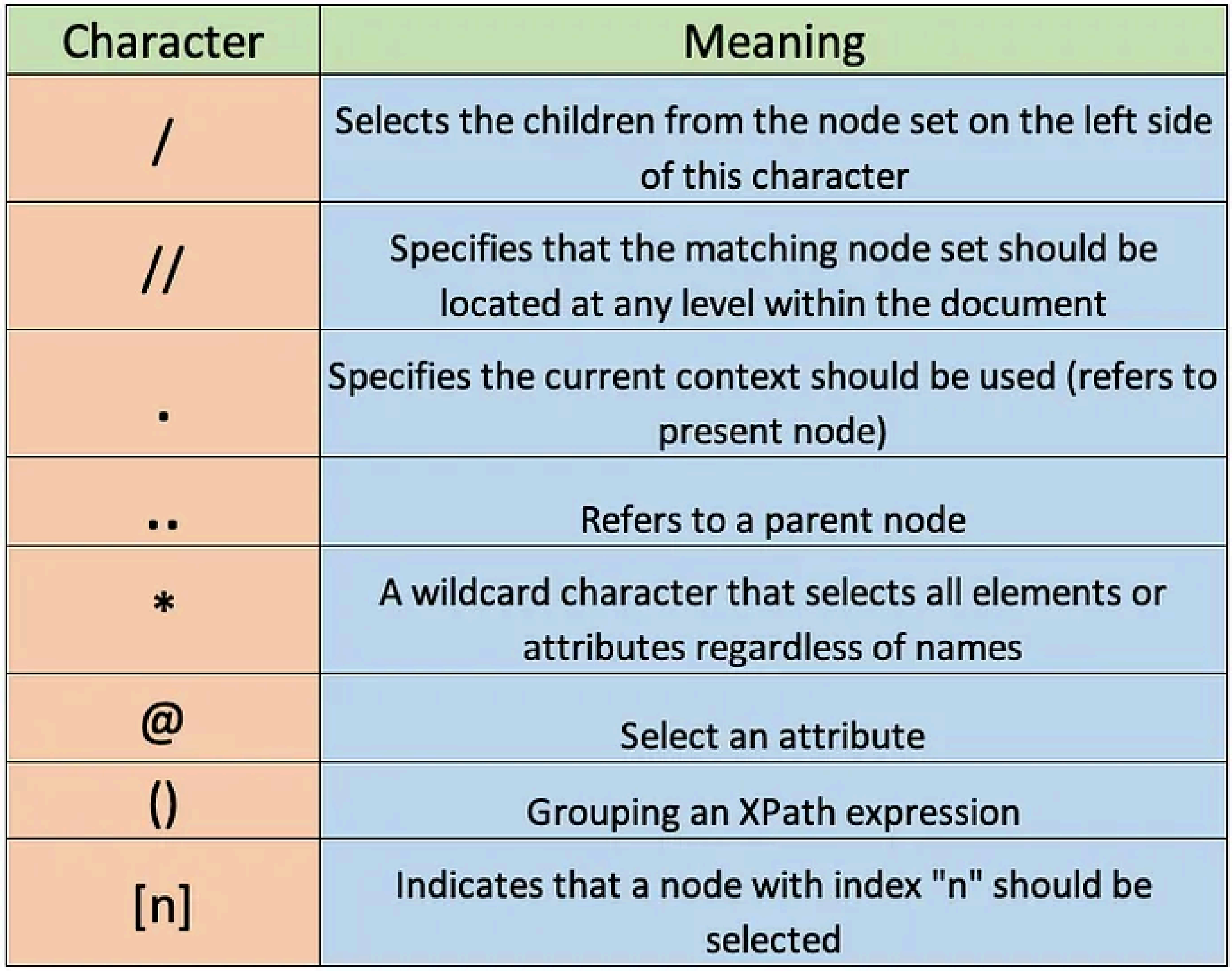
Image by author

To see an example, let's imagine we have an extra `p` element with attribute `class=”plot2"` in our HTML code.

```
# Locate elements that has either "plot" or "plot2" values
//p[(@class="plot") or (@class="plot2")]
```

XPath Special Characters

Building XPath might be a bit trickier in the beginning because there are many characters that we don't know their meaning. This is why I made the table below that contains the most common special characters.



Character	Meaning
/	Selects the children from the node set on the left side of this character
//	Specifies that the matching node set should be located at any level within the document
.	Specifies the current context should be used (refers to present node)
..	Refers to a parent node
*	A wildcard character that selects all elements or attributes regardless of names
@	Select an attribute
()	Grouping an XPath expression
[n]	Indicates that a node with index "n" should be selected

Image by author

Selenium

Selenium is most powerful than Beautiful Soup because it allows you to scrape JavaScript-driven pages.

I recommend you solve a project to memorize all the Selenium methods listed in this guide. Below there's my step-by-step tutorial on how to solve a Selenium project from scratch.

Tutorial: Python Selenium for Beginners — A Complete Web Scraping Project

Installing the libraries and Chromedriver

To start working with Selenium we need to install the `selenium` library and download chromedriver.

To download Chromedriver, go to this link. In the "Current Releases" section click on the Chromedriver version that corresponds to your Chrome browser (to check the version, click the 3 dot button on the upper right corner, click on "Help", then click on "About Google Chrome").

After you download the file, unzip it and remember where it's located.

To install Selenium, open up a terminal and run the following commands.

```
pip install selenium
```

Importing the libraries

After installing the necessary libraries, import `webdriver` before scraping the website.

```
from selenium import webdriver
```

Creating the "driver"

In Selenium we use the "driver" object to find elements in a website. To create this object, do the following.

```
# 1. Define the website you wish to scrape and path where
Chromedriver was downloaded
```

```
web = "www.google.com"
path = "introduce chromedriver path"

# 2. Create the driver
driver = webdriver.Chrome(path)
```

Once the driver is created we can open the website with `.get()`. Remember always to close the website after you scrape the content.

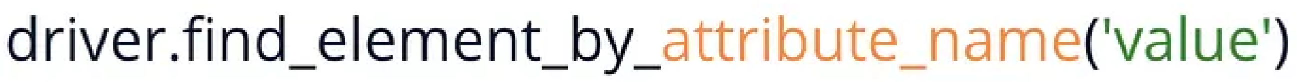
```
# 1. Open the website
driver.get(web)

# 2. Close the website
driver.quit()
```

What we've done so far is what we always do regardless of the website you wish to scrape.

Finding elements: find_element_by() vs find_elements_by()

There are two ways to get elements with Selenium: `find_element_by()` and `find_element`**`s`**`_by()`. We use the first to get the first element that matches a specific tag name, class name, id, and XPath, while the latter will get all the elements that matched and put them inside a list.

Let's have a look at the syntax of finding elements with Selenium.



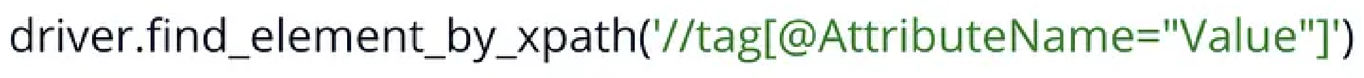
```
driver.find_element_by_attribute_name('value')
```

Image by author

You can replace the `attribute_name` with any attribute. Below you can find the most common attributes used.

```
# Finding a single element
driver.find_element_by_id('id_value')

# Finding multiple elements (returns a list)
driver.find_elements_by_class_name('value')
driver.find_elements_by_css_selector('value')
driver.find_elements_by_tag_name('value')
driver.find_elements_by_name('value')
driver.find_elements_by_xpath('value')
```

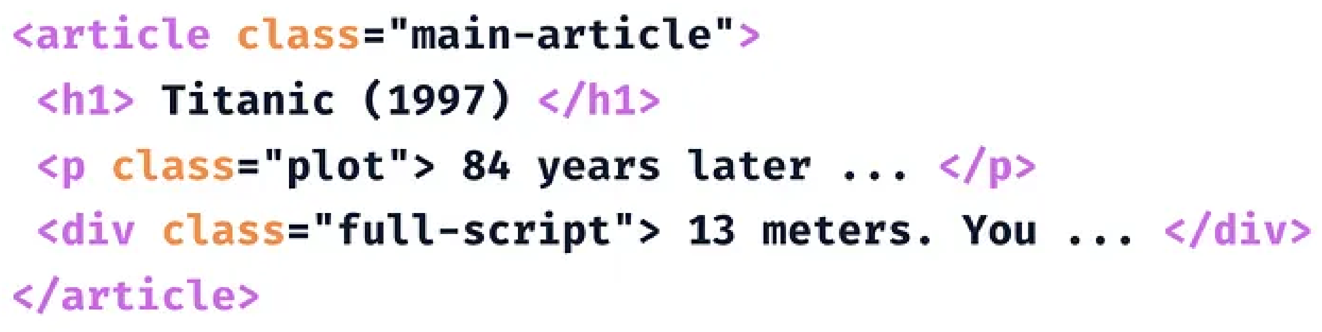
In the case of XPaths, there's a special syntax.



```
driver.find_element_by_xpath('//tag[@AttributeName="Value"]')
```

Image by author

Let's look at some examples. We'll be using the HTML code we've seen before.



```
<article class="main-article">
 <h1> Titanic (1997) </h1>
 <p class="plot"> 84 years later ... </p>
 <div class="full-script"> 13 meters. You ... </div>
</article>
```

Image by author

Let's locate the article element, title, and transcript.

```
# Get the article element
element1 = driver.find_element_by_class_name('main-article')

# Get the title element
element2 = driver.find_element_by_tag_name('h1')

# Get the transcript
element3=driver.find_element_by_xpath('//div[@class="full-script"]')
```

Getting the values: text

Most of the time we want to get the text inside an element. In Selenium we can use `.text` to get the text we want.

Let's look at some examples using the "element2" we got before.

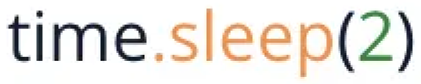
```
data = element2.text
```

In this particular example, `text` will return the text "Titanic (1997)." The `.text` does a good job formating the text we scrape, but if necessary use `strip` and the `separator` functions as additional operations after getting the data.

Waits: Implicit Waits vs Explicit Waits (Handling ElementNotVisibleException)

One of the problems of scraping Javascript-driven websites is that the data is loaded dynamically so it can take some seconds to display all the data correctly. As a result, an element might not be located in the DOM (Document Object Model) when scraping the website, so we'll get an "ElementNotVisibleException." This is why we have to make the driver wait until the data we wish to scrape is loaded completely.

There are 2 types of waits: implicit & explicit waits. An implicit wait is used to tell the web driver to wait for a certain amount of time when trying to locate an element. In Python, you can import the `time` library and then make an implicit wait with `time.sleep()` and specify the seconds to wait within parentheses. For example, if you want to make the driver stop for 2 seconds, write this.



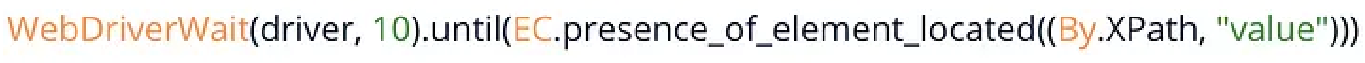
```
time.sleep(2)
```

Image by author

On the other hand, an explicit wait makes the web driver wait for a specific condition (Expected Conditions) to occur before proceeding further with the execution. First, you need to import a couple of libraries besides `webdriver`

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Chrome()
driver.get("www.google.com")

<explicit wait syntax>

driver.quit()
```

Explicit waits have the following syntax.



```
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.XPath, "value")))
```

Image by author

Let’s look at an example. We’ll locate the transcript element we’ve seen before, but now with expected conditions.

This will make the web driver wait for 10 seconds until an element with XPath `‘//div[@class=”full-script”]’` shows up in the DOM.

So far we've used `presence_of_element_located`, but there are other conditions. To name a few:

- `presence_of_all_elements_located`
- `element_to_be_clickable`
- `visibility_of_element_located`
- `element_located_to_be_selected`

Options: Headless mode, block ads, and more

Web scraping might get repetitive if we see the same browser opening over and over again or get trickier if the website we want to scrape has ads that interfere when scraping the data we want. Fortunately, you can easily deal with these and many other issues a browser has by using Selenium's Options.

First, you need to import the `Options` library.

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
```

Then to make Selenium silently scrape in the background, activate headless mode.

```
# 1. Create instance
options = Options()

# 2. Activate Headless mode
options.headless = True
driver = webdriver.Chrome(options=options)
...
```

In this example, we created an `options` instance and then set headless to `True` and finally, added the `options` instance to the `options` argument of webdriver.

To make Selenium block ads, we have to download a third-party Chrome extension. For example, you can download Adblock Plus and then add the path of the extension to `options`.

```
# 1. Create instance
options = Options()

# 2. Add the path of the extension
options.add_argument('path_of_extension')
driver = webdriver.Chrome(options=options)
```

Scrapy

Scrapy is the most powerful web scraping framework in Python and it's also a bit complicated to start with compare to Beautiful Soup and Selenium.

Installing the libraries

Although you can easily install Scrapy with `pip install Scrapy` it's not recommended to do it this way in the official documentation because you might get some installation issues (especially on Windows).

It's recommended to install Scrapy through Anaconda or Miniconda and use the package from the conda-forge channel. After you download either Ananconda or Miniconda, create a new environment, then open up a terminal (click on the play button of your environment and then click on "Open Terminal") and then install Scrapy with the following command.

```
conda install -c conda-forge scrapy
```

After this, you have to open this new environment with your IDE or text editor (in Pycharm I even had to install `protego` to start working with Scrapy using the command `conda install -c conda-forge protego`)

Creating a Project and Spider

To start working with Scrapy we need to create a project. Let's create a spider named `my_first_spider`.To do so, run the following command on the terminal.

```
scrapy startproject my_first_spider
```

Then we have to create a spider. Unlike Beautiful Soup or Selenium, we don't start with a blank script on Scrapy, but we use templates. Before we create a spider, we have to change the directory to the project's folder.

```
cd my_first_spider
```

To create a spider, run the following command on the terminal.

```
scrapy genspider example example.com
```

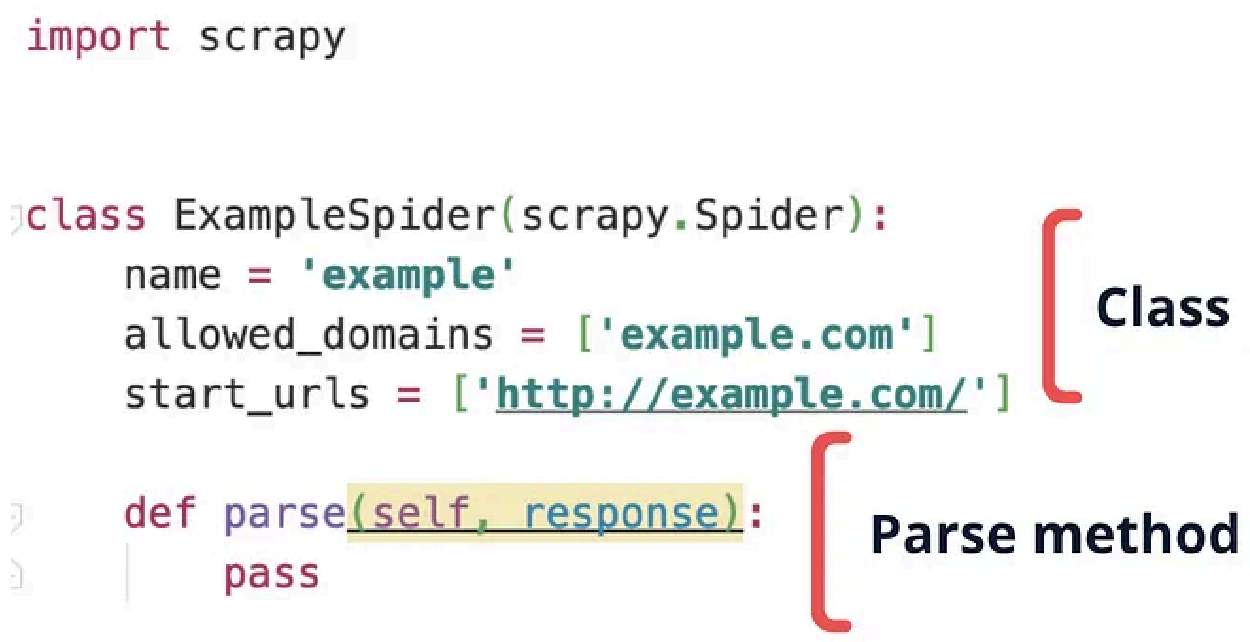
where `example` is the name of the spider and `example.com` is the website you wish to scrape.

After this, a new script named `example` should be created in the spider folder which is insider your project's folder. Open the script and you will see the basic template to scrape a website with Scrapy.

Now everything is set up to start scraping with Scrapy.

The Template

When you open the `example` script you should see that it already has some code written. It should look like the image below.



```python
import scrapy


class ExampleSpider(scrapy.Spider):
    name = 'example'
    allowed_domains = ['example.com']
    start_urls = ['http://example.com/']

    def parse(self, response):
        pass
```

Image by author

This is the basic template and is split into 2 parts: the class and parse method.

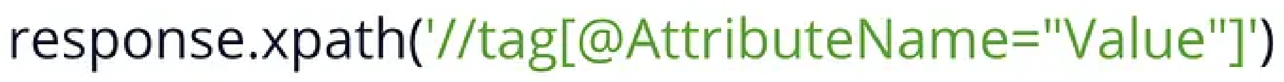
The class is built with the name of the spider and website we introduced in the previous command. You shouldn't edit the name of the variables, but you can change the values. Edit the `allowed_domains` value and write the main domain of the website you wish to scrape (aka root or homepage). You should also edit the `start_urls` value and write the exact link you want to scrape.

The parse method needs to be built by us. Here we use the response object to find elements and then extract the text data we want. We'll see how to build this in the next section.

Finding elements and getting the text value

In Scrapy, we use the `response` object to find elements. This is the equivalent of Beautiful Soup's `soup` or Selenium's `driver`.

Let's have a look at the syntax of finding elements with Scrapy.



```
response.xpath('//tag[@AttributeName="Value"]')
```

Image by author

Unlike Selenium, there's no `text` method that will help you get the text inside a node, so we have to include the text node in the XPath. In addition to that, you have to use either `.get()` or `.getall()` to get the text data. We use the first to get the first element that matches the XPath, while the latter will get all the elements that matched and put them inside a list.

Let's see some examples.

```
# Finding a single "h1" element and getting the text inside
title = response.xpath(‘//h1/text()’).get()

# Finding multiple elements (returns a list)
response.xpath(‘//tag[@AttributeName=”Value”]/text()’).getall()
```

Note: The `/text()` indicates that the text element is an immediate child node of the previous element (e.g., `text()` should be the immediate child node of `h1` in the first example). If the text is not an immediate child node, you should either include the element that is in between or use `//text()` to get all the children element.

Return data extracted

To see the data extracted we have to use the `yield` keyword. For example, let’s return the `title` we extracted in the previous section.

```
def parse(self, response):
    title = response.xpath(‘//h1/text()’).get()

    # Return data extracted
    yield {
        'titles': title,
    }
```

As you can see, we should use `yield` inside the parse method.

Run the spider and export data to CSV or JSON

To run the spider and get the data we want, run the following command on a terminal.

```
scrapy crawl example
```

where `example` is the name of our spider.

To export the data to a CSV or JSON file use the following command.

```
scrapy crawl example -o name_of_file.csv
scrapy crawl example -o name_of_file.json
```

Python Basics for Web Scraping

The following is the stuff you will frequently use when scraping a website regardless of the framework you use in Python (Beautiful Soup, Selenium, and Scrapy). I will also tell you why this stuff is important in web scraping.

Storing Data: List, Indexing

In Python, lists are used to store multiple items in a single variable. To create a list we have to place the elements inside square brackets `[]`, separated by commas. Consider the following list named `states` that contains the 4 most populated states in the US.

```
states = ["California", "Texas", "Florida", "New York"]
```

Each item in the list has an index (the position in the list). Python uses zero-based indexing. That means, the first element ("California") has an index 0, the second("Texas") has index 1, and so on.

To access an element by its index we need to use square brackets again (this is known as indexing)

```
states = ["California", "Texas", "Florida", "New York"]
>>> states[0]
"California"
>>> states[1]
"Texas"
>>> states[3]
"New York"
```

We can also use negative indexes to get elements starting on the last position of the list. In this case, instead of using indexes from zero and above, we can use indexes from -1 and below.

```
states = ["California", "Texas", "Florida", "New York"]
>>> states[-1]
"New York"
```

Why is this important for Web Scraping? Lists are necessary to store all the data scraped, so we can later export it to a CSV file. On the other hand, indexing is used when we locate multiple elements with the same name and you want to select one in particular (e.g., a specific page in a pagination bar)

Working with the data stored: For loops and If statement

A typical step after storing elements in a list is looping through it so you can work with the elements. To do so, we have to use the `for` and `in` keywords. Let's print each element of the `states` list we created before.

```
for state in states:
    print(state)
```

Breaking down the code:

- `states` is the name of the list of states we created
- `for state in states` is looping through each item of the list
- `print(state)` is executed once for each state of our `states` list

If we run this code, we obtain this:

```
California
Texas
Florida
New York
```

When we use the `if` statement. We're telling Python 'only continue when this condition is `True`' To do so, we have to use the `if` keyword:

```
for state in states:
    if state == "Florida":
        print(state)
```

If we run this code, we only obtain "Florida" because we told Python 'only print when the state is Florida'

```
Florida
```

Exporting the data: Python's "with open() as" and Pandas' dataframes

The simplest way to export the data scraped is using the built-in Python's "with open() as" pattern. Let's create a .txt file named `test`

```
with open('test.txt', 'w') as file:
    file.write('Data Exported!')
```

Breaking down the code:

- `open()` takes in a `file_name` and a `mode` and it opens the given filename in the current directory in the mode that you pass in.
- `'test.txt'` is the name of the file we created
- `‘w’` is the mode. There are write mode (‘w’), read mode (‘r’), and append mode (‘a’). Write mode (‘w’) deletes the existing file content, creating the file if it doesn’t already exist, and then allows you to write to the file.
- `file` is the name that represents the “with open() as” pattern
- `.write()` allows writing content inside the file created

If we run this code, we obtain a file named test.txt in our working directory. Open it and you will see the message “Data Exported!”

Another way to export the data extracted is using dataframes, but in this case, we have to install the Pandas library first.

```
pip install pandas
```

To easily create a dataframe, we will need to put the lists we created before into a dictionary. First, let's create a second list named `population` and then put the `population` and `states` list inside a dictionary we will name `dict_states`. To create a dictionary, we place items inside curly braces {} separated by commas. An item has a key and a corresponding value that is expressed as a pair (key: value).

```
# Second list
population = [39613493, 29730311, 21944577, 19299981]

# Storing lists within dictionary
dict_states = {'States': states, 'Population': population}
```

Above we created the keys 'States' and 'Population' and set our two lists as values. Now we can easily create a dataframe using the `.from_dict` method. Let's create a dataframe named `df_states`.

```
# Creating the dataframe
df_states = pd.DataFrame.from_dict(dict_states)
```

To export the data to a CSV file we use `.to_csv()` and inside parentheses, we write the name of the file we want to create.

```
df_states.to_csv('states.csv')
```

If we run the code above, we obtain a CSV file named states.csv in our working directory. Open it and you will the two lists in columns A and B.

Handling an exception error: Try-Except

When scraping a website you will come across different exceptions. The element might not be in the DOM, the element might not be visible yet, etc. The best way to handle an exception error is to use the try and except statement.

To show you how this works let's use the following list that contains integers and string elements.

```
new_list = [2, 4, 6, 'California']
```

Now let's see what happens if I loop through the list and divide each element by 2.

```
for element in new_list:
    print(element/2)
```

If you run the code above you will get this

```
1.0
2.0
3.0
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

So the code perfectly worked for the first three elements but then failed because the fourth element is a string and not an integer.

When scraping a website, if we leave the code as it is, we won't be able to store and then export the data because the code will break completely. This is why we have to use the `except` statement to let Python know how to handle an exception and then skip to the next iteration.

Python will execute the code following the `try` statement as a normal part of the script. However, Python will only execute the code following the `except` statement when the code runs into an error.

Let's print a message every time the code runs into an error.

```
for element in new_list:
    try:
        print(element/2)
    except:
        print('The element is not a number!')
```

If you run the code above you will get this.

```
1.0
2.0
3.0
The element is not a number!
```

Although the fourth element isn't a number, the code didn't break. Instead, we got a message that let us know what happened.

Workarounds: While-Break (Handling Beautiful Soup and Selenium limitations)

When a web scraping tool lacks some functionality, you have 2 options: either you learn from scratch how to use another scraping tool or find a workaround.

A good example of this is pagination. Scraping through multiple pages can be easily done on Scrapy. However, on Selenium and Beautiful Soup you might need to use the while loop. Below is the format of a while loop.

```
while <expr>:
    <statement(s)>
```

`<statement(s)>` represents the block to be repeatedly executed. The `<expr>` is evaluated in boolean context. If it's true, the body is executed and then `<expr>` checked again. If `<expr>` becomes false, the execution proceeds to the immediate code below the while loop

Let's see the following example.

```
n = 4
while n > 0:
    print(n)
    n -= 1
```

If you run this code, Python will print `n` until it gets a value greater than 0, then the condition becomes false. The output is the following.

```
4
3
```

```
2
1
```

We can also force the while loop to break using the `break` keyword.

```
n = 4
while n > 0:
    print(n)
    n -= 1
    if n == 2:
        break

print('Loop ended.')
```

This time Python will only print `n` before it gets the value of 2, then the while loop will break and the immediate code below the loop `print('Loop ended.')` will be executed. This is the output.

```
4
3
Loop ended.
```

Web Scraping Cheat Sheet [PDF]

That's it! Now you're ready to scrape a website with Python!

Join my email list with 10k+ people to get my Web Scraping Cheat Sheet I use in all my tutorials (Free PDF)