16 Amazing Articles in our student magazine - Dataquest Direct

Your peers at Dataquest have written **16 AMAZING ARTICLES** in the past month to participate in the **March Writers Contest**. They've written personal unguided projects, tutorials and personal success stories. Check them out and mark your favourites!

A Guide to Scraping HTML Tables with Pandas and BeautifulSoup

python web-scraping

---

otavios.s Community Moderator

15d

It’s very common to run into HTML tables while scraping a webpage, and without the right approach, it can be a little tricky to extract useful, consistent data from them.

In this article, you’ll see how to perform a quick, efficient scraping of these elements with two main different approaches: using only the Pandas library and using the traditional scraping library BeautifulSoup.

As an example, I scraped the Premier League classification table. This is good because it’s a common table that can be found on basically any sports website. Although it makes sense to inform you this, the table being is scraped won’t make much difference while you read as I tried to make this article as generalistic as possible.

pandas.read_html(): The Shortcut

If all you want is to get some tables from a page and nothing else, you don’t even need to set up a whole scraper to do it as *Pandas* can get this job done by itself. The `pandas.read_html()` function uses some scraping libraries such as *BeautifulSoup* and *Urllib* to return a list containing all the tables in a page as DataFrames. You just need to pass the URL of the page.

```
dfs = pd.read_html(url)
```

All you need to do now is to select the DataFrame you want from this list:

```
df = dfs[4]
```

If you’re not sure about the order of the frames in the list or if you don’t want your code to rely on this order (websites can change), you can always search the DataFrames to find the one you’re looking for by its length…

```
for df in dfs:
    if len(df) == 20:
        the_one = df
        break
```

… or by the name of its columns, for example.

```
for df in dfs:
    if df.columns == ['#', 'Team', 'MP', 'W', 'D', 'L', 'Point
        the_one = df
        break
```

But Pandas isn't done making our lives easier. This function accepts some helpful arguments to help you get the right table. You can use `match` to specify a string o regex that the table should match; `header` to get the table with the specific headers you pass; the `attrs` parameter allows you to identify the table by its class or id, for example.

However, if you're not scraping only the tables and are using, let's say, Requests to get the page, you're encouraged to pass `page.text` to the function instead of the URL:

```
page = requests.get(url)
soup = BeautifulSoup(page.text, 'html.parser')

dfs = pd.read_html(page.text)
```

The same goes if you're using Selenium's web driver to get the page:

```
dfs = pd.read_html(driver.page_source)
```

That's because by doing this you'll significantly reduce the time your code takes to run since the `read_html()` function does not need to get the page anymore. Check the average time elapsed for one hundred repetitions in each scenario:

```
Using the URL:
Average time elapsed: 0.2345 seconds
Using page.text:
Average time elapsed: 0.0774 seconds
```

Using the URL made the code about three times slower. So it only makes sense to use it if you're not going to get the page first using other libraries.

Getting the Table's Elements with BeautifulSoup

Although Pandas is really great, it does not solve all of our problems. There will be times when you'll need to scrape a table element-wise, maybe because you don't want the entire table or because the table's structure is not consistent or for whatever other reason.

To cover that, we first need to understand the standard structure of an HTML table:

```
<table>
    <tr>
        <th>
        <th>
        <th>
        <th>
        <th>
        <th>
        <th>
    </tr>
    <tr>
        <td>
        <td>
        <td>
        <td>
        <td>
        <td>
        <td>
    </tr>
    <tr>
        <td>
        <td>
        <td>
        <td>
        <td>
        <td>
        <td>
    </tr>
```

.

```
.
.
</table>
```

Where `tr` stands for “table row”, `th` stands for “table header” and `td` stands for “table data”, which is where the data is stored as text.

The pattern is usually helpful, so all we have left to do is select the correct elements using BeautifulSoup.

The first thing to do is to find the table. The `find_all()` method returns a list of all elements that satisfied the requirements we pass to it. We then must select the table we need in that list:

```
table = soup.find_all('table')[4]
```

Depending on the website, it will be necessary to specify the table class or id, for instance.

The rest of the process is now almost intuitive, right? We just need to select all the `tr` tags and the text in the `th` and `td` tags inside them. We could just use `find_all()` again to find all the `tr` tags, yes, but we can also iterate over these tags in a more straightforward manner.

The `children` attribute returns an iterable object with all the tags right beneath the parent tag, which is `table`, therefore it returns all the `tr` tags. As it’s an iterable object, we need to use it as such.

After that, each `child` is `tr` tag. We just need to extract the text of each `td` tag inside it. Here’s the code for all this:

```
for child in soup.find_all('table')[4].children:
    for td in child:
        print(td.text)
```

And the process is done! You then have the data you were looking for and you can manipulate it the way it best suits you.

Other Possibilities

Let's say you're not interested in the table's header, for instance. Instead of using `children`, you could select the first `tr` tag, which contains the header data, and use the `next_siblings`attribute. This, just like the `children` attribute, will return an iterable, but with all the other `tr` tags, which are the **siblings** of the first one we selected. You'd be then skipping the header of the table.

```
for sibling in soup.find_all('table')[4].tr.next_siblings:
    for td in sibling:
        print(td.text)
```

Just like children and the next siblings, you can also look for the previous siblings, parents, descendants, and way more. The possibilities are endless, so make sure to check the **BeautifulSoup documentation** to find the best option for your scraper.

A Real-Life Example

We've so far written some very straightforward code to extract HTML tables using Python. However, when doing this for real you'll, of course, have some other issues to consider.

For instance, you need to know how you're going to store your data. Will you directly write it in a text file? Or will you store it in a list or in a dictionary and then creating the *.csv* file? Or will you create an empty DataFrame and fill it with the data? There certainly are lots of possibilities. My choice was to store everything in a big list of lists that will be later transformed into a DataFrame and exported as a *.csv* file. Read more about storing scraped data **here**.

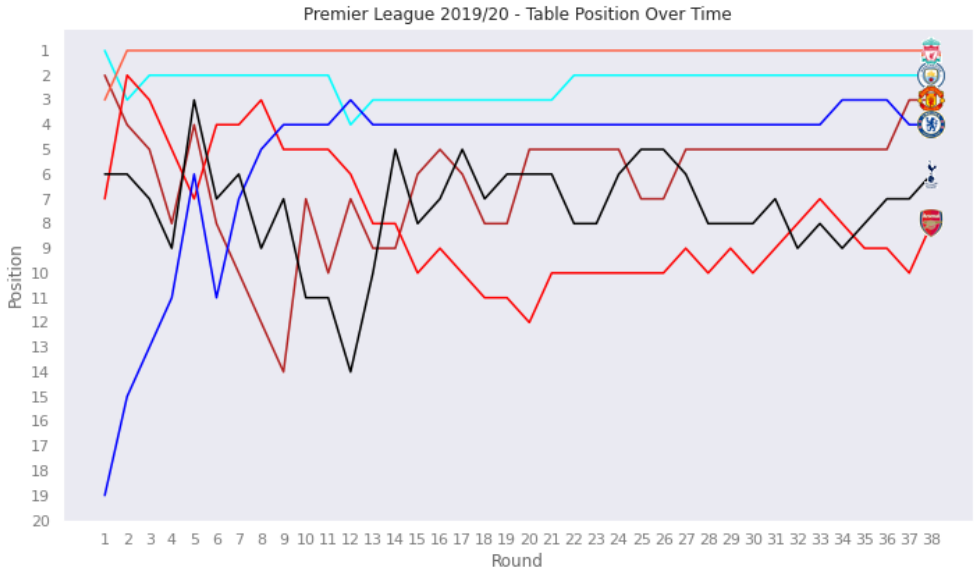
In another subject, you might want to use some `try` and `except` clauses in your code to make it prepared to handle some exceptions it may find along the way. You never know what unexcepted error can crash your code.

In this example, I scraped the Premier League table after every round in the entire 2019/20 season using most of what I've covered in this article. This is the entire code for it:

```
import pandas as pd
import numpy as np
import requests
```

```
from bs4 import BeautifulSoup
from time import sleep


def get_table(round, url=url):
    round_url = f'{url}/{round}'
    page = requests.get(round_url)
    soup = BeautifulSoup(page.text, 'html.parser')

    rows = []
    for child in soup.find_all('table')[4].children:
        row = []
        for td in child:
            try:
                row.append(td.text.replace('\n', ''))
            except:
                continue
        if len(row) > 0:
            rows.append(row)

    df = pd.DataFrame(rows[1:], columns=rows[0])
    return df


for round in range(1, 39):
    table = get_table(round)
    table.to_csv(f'PL_table_matchweek_{round}.csv', index=Fals
    sleep(np.random.randint(1, 10))
```

Everything is there: gathering all the elements in the table using the `children` attribute, handling exceptions, transforming the data into a DataFrame, exporting a .csv file, and pausing the code for a random number of seconds. After all this, all the data gathered by this code produced this interesting chart:



You're not going to find the data needed to plot a chart like that waiting for you on the internet. But that's the beauty of scraping: you can go get the data yourself!

As a wrap this up I hope was somehow useful and that you never have problems when scraping an HTML table again. If you have a question, a suggestion, or just want to be in touch, feel free to send a message, connect through **LinkedIn**, follow on **Twitter**, or leave a comment down below! If you're interested in more articles like this, check my **Medium profile**!

Thanks for reading!

Cover Image Designed by vectorjuice / Freepik

🔗 Announcing the Community Champions for this week!