

# Day 1: Web Scraping w/ Python Requests & Beautiful Soup

## Activity 1: Scraping [BooksToScrape](#)

### Requirements:

[python 3.7+ installed locally.](#)

[pip installed](#)

[Full Code for Day 1](#)

```
pip install requests
```

```
pip install bs4
```

### References:

[Python Requests Library Documentation](#)

[Beautiful Soup Library Documentation](#)

[Status Codes](#)

[HTML Tag Reference Guide](#)

[CSS Selector Reference Guide](#)

```
import requests
r=requests.get()
r.text
r.status_code
```

Common status codes:

- 200 OK
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 429 Too Many Requests

```
if r.status_code == 200:
    with open('site.html', 'w') as file:
        file.write(r.text)
```

Ok, we have the front page's site html. **So what?**

What if we wanted to scrape all of the product titles and their prices?

In this case, there is no way to change the view on the site to show more than 20 at a time.

I will show you an example of a site where we can get around this so we can get it all at once.  
<https://juntasupervision.pr.gov/documents/>

In our site, we know all of the site urls (minus the front page) look like this:  
<https://books.toscrape.com/catalogue/page-50.html>

So we can write a short script to generate the URLs for us.

```
with open ('urls.txt', 'a') as file:
    for i in range(1,51):
        url = f'https://books.toscrape.com/catalogue/page-{i}.html\n'
        file.write(url)
```

Now, we have all of the URLs we need to capture the HTML files for all of the books. We can open up the file visually to make sure we are on track with our URLs. Of course, we don't have to store the URLs there and deal with file handling. We can use Python lists. Notice I use an f string for iteration where {i} is the iterative step. This is a common technique in generation of unique URLs.

```
url_list = ['https://books.toscrape.com/index.html']
for i in range(2,51):
    url_to_add = f'https://books.toscrape.com/catalogue/page-{i}.html'
    url_list.append(url_to_add)
```

You can of course now iterate through the url list and run requests.get() on each url, saving the requests to a separate file, or you can save the results in another list for easier access.

```
r_list = []
for url in url_list:
    r_list.append(requests.get(url))
```

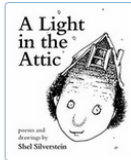
Ok. Now we have each page of the site scraped.

How do we make use of these pages and filter out only what we want?

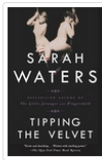
Let's go back to the site in our browser.

Warning! This is a demo website for web scraping purposes. Prices and ratings here v


- Sequential Art
- Classics
- Philosophy
- Romance
- Womens Fiction
- Fiction
- Childrens
- Religion
- Nonfiction
- Music
- Default
- Science Fiction
- Sports and Games
- Add a comment
- Fantasy
- New Adult
- Young Adult
- Science
- Poetry



★★★★☆  
A Light in the ...  
£51.77  
✓ In stock  
Add to basket



★★★★☆  
Tipping the Velvet  
£53.74  
✓ In stock  
Add to basket



★☆☆☆☆  
Sou  
£!  
✓ I  
Add

https://books.toscrape.com/catalogue/category/books/fiction\_10/index.html

Inspector Console Debugger Network Style Editor Performance Memory Storage Accessibility Application

Q Search HTML

```

<div>
  <ol class="row">
    ::before
    <li class="col-xs-6 col-sm-4 col-md-3 col-lg-3">
      <article class="product_pod">
        <div class="image_container"></div>
        <p class="star-rating Three"></p>
        <h3>
          <a href="catalogue/a-light-in-the-attic_1000/index.html" title="A Light in the Attic">A Light in the ...</a>
        </h3>
        <div class="product_price"></div>
      </article>
    </li>
  </ol>
</div>

```

We can see `<article>` tags with the class "product\_pod" that contains a unit of the page that describes a single book including its rating, price, title, and link to the book in the catalogue.

The title is tucked into the title of an `<a>` tag wrapped around an `<h3>` tag. We can use this knowledge of the structure of the page to isolate and filter for it.

The product price is wrapped in a `<div>` of class "product price" and it is inside the text of a `<p>` tag inside that div. We can also use this knowledge to isolate it.

We could use this knowledge to write Python code to isolate the elements all without outside libraries, but it is ***much more*** efficient and simple to do using [Beautiful Soup](#). Don't re-invent the wheel!

Let's start off with a single page and then we can scale up.

```

from bs4 import BeautifulSoup
soup = BeautifulSoup(r_list[0].text, 'html.parser')
book_articles = soup.find_all('article', class_='product_pod')

```

```

books_info = []

for article in book_articles:
    title = article.h3.a['title']

```

```
price = article.select_one('div.product_price p').text.strip()
book_info = {'title': title, "price": price}
books_info.append(book_info)
```

If we then print books\_info in our interpreter we can see that we have the dictionary populated.

```
>>> books_info
[{'title': 'A Light in the Attic', 'price': '£51.77'}, {'title': 'Tipping the Velvet', 'price': '£53.74'}, {'title': 'Soumission', 'price': '£50.10'}, {'title': 'Sharp Objects', 'price': '£47.82'}, {'title': 'Sapiens: A Brief History of Humankind', 'price': '£54.23'}, {'title': 'The Requiem Red', 'price': '£22.65'}, {'title': 'The Dirty Little Secrets of Getting Your Dream Job', 'price': '£33.34'}, {'title': 'The Coming Woman: A Novel Based on the Life of the Infamous Feminist, Victoria Woodhull', 'price': '£17.93'}, {'title': 'The Boys in the Boat: Nine Americans and Their Epic Quest for Gold at the 1936 Berlin Olympics', 'price': '£22.60'}, {'title': 'The Black Maria', 'price': '£52.15'}, {'title': 'Starving Hearts (Triangular Trade Trilogy, #1)', 'price': '£13.99'}, {'title': 'Shakespeare's Sonnets', 'price': '£20.66'}, {'title': 'Set Me Free', 'price': '£17.46'}, {'title': 'Scott Pilgrim's Precious Little Life (Scott Pilgrim #1)', 'price': '£52.29'}, {'title': 'Rip it Up and Start Again', 'price': '£35.02'}, {'title': 'Our Band Could Be Your Life: Scenes from the American Indie Underground, 1981-1991', 'price': '£57.25'}, {'title': 'Olio', 'price': '£23.88'}, {'title': 'Mesaerion: The Best Science Fiction Stories 1800-1849', 'price': '£37.59'}, {'title': 'Libertarianism for Beginners', 'price': '£51.33'}, {'title': 'It's Only the Himalayas', 'price': '£45.17'}]
```

We can now scale up with a for loop to go through each web page (there are 50 of them!)

```
books_info = []
for r in r_list:
    soup = BeautifulSoup(r.text, 'html.parser')
    book_articles = soup.find_all('article', class_='product_pod')
    for article in book_articles:
        title = article.h3.a['title']
        price = article.select_one('div.product_price p').text.strip()
        book_info = {'title': title, "price": price}
        books_info.append(book_info)
print(books_info)
```

If we now print books\_info into our interpreter, we can see the output of all of the prices.

Wrapping the code we have all together thus far:

```
import requests
from bs4 import BeautifulSoup
url_list = ['https://books.toscrape.com/index.html']
for i in range(1,51):
    url_to_add = f'https://books.toscrape.com/catalogue/page-{i}.html'
    url_list.append(url_to_add)
r_list = []
for url in url_list:
    r_list.append(requests.get(url))
books_info = []
for r in r_list:
    soup = BeautifulSoup(r.text, 'html.parser')
    book_articles = soup.find_all('article', class_='product_pod')
    for article in book_articles:
        title = article.h3.a['title']
        price = article.select_one('div.product_price p').text.strip()
        book_info = {'title': title, "price": price}
        books_info.append(book_info)
print(books_info)
```

**Challenge:** Can you write a line to pull the text for whether it is in stock or not?

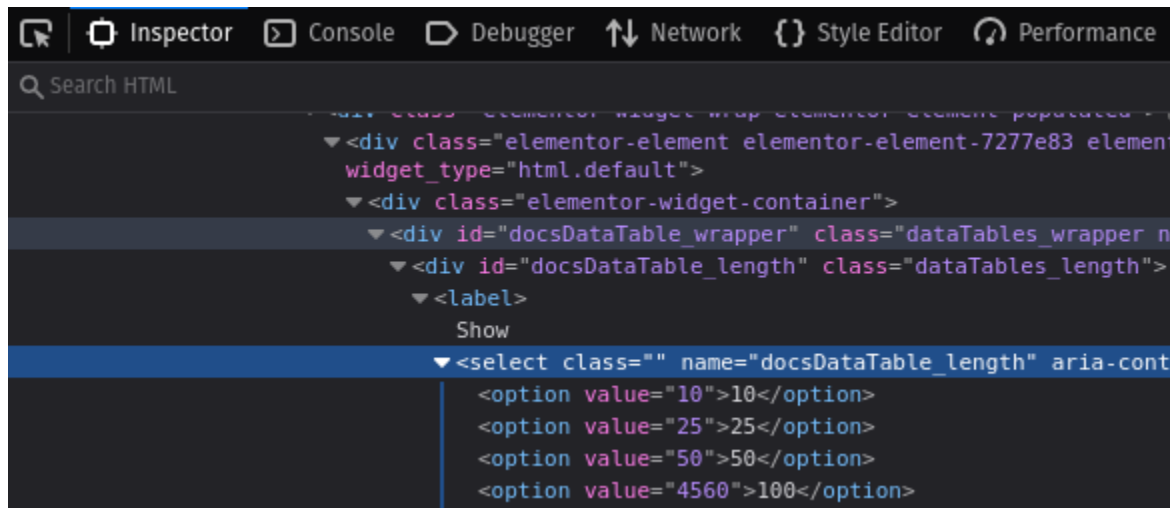
## Activity 2: Scraping the FOMB

Let's get back to the [Financial Oversight and Management Board of Puerto Rico's Site](#)

Let's open up the developer console over the "Show 25 Entries" dropdown menu by right click -> Inspect

Show  entries

We can click through and see that the options are hard-coded on the front end. We can manipulate that to make it view all of the entries- 4560.



Once we have modified the 4560 option, click on the dropdown and change to 100. You should see all 4,560 documents now populated on the page.

Since we modified the page in the browser, requests won't know about this if you just try to get() the html to then parse.

I'll have you download the modified/re-loaded page from within the browser manually in this case.

Open the browser inspector (usually CTRL-SHIFT-I), click on the top HTML or body tag, click Copy -> Inner HTML and then paste the HTML into your favorite text editor and save as **site.html** locally.

This site is a bit strange, it has table rows with the names "odd" and "even" and repeats those.

Nonetheless, we will be able to extract the document name and the link to the document by isolating the table, grabbing all of the rows, and grabbing each cell in each row. From each cell, we can get the content we want, namely the title and link value (href value).

```
from bs4 import BeautifulSoup

documents = []
with open('site.html', 'r') as file:
    soup = BeautifulSoup(file, 'html.parser')
    table = soup.find('table', {'id': 'docsDataTable'})
    rows = table.find_all('tr')
    for row in rows:
        cells = row.find_all('td')
        if len(cells) == 5: # Line added to filter out empty/malformed table rows
            document = {"title": cells[0].text, "link": cells[4].find('a')['href']}
            documents.append(document)
print(documents)
```

If I print the first entry, we will get something like so:

```
>>> print(documents[0])
{'title': 'JSAP - Carta - Legislatura - PC 1651 - Seguimiento carta a la
Legislatura', 'link':
'https://drive.google.com/file/d/1hls2PmdzgaTBYKkOgMkrr9MFJKC3-XN3/view?usp
=sharing'}
```

documents is a list that contains a dictionary with the title and link to each document on the page.

What if I wanted to download all of the documents on the site?

Normally, we could use requests to get the PDF from the link with a stream, and save it to a PDF file locally like so:

```
pdf_response = requests.get(link, stream=True)
pdf_filename = f'{your_title_here}.pdf'
with open(pdf_filename, 'wb') as pdf_file:
    for chunk in pdf_response.iter_content(chunk_size=1024):
        if chunk:
            pdf_file.write(chunk)
```

But, these are Google Drive links. Google Drive has **a lot** of anti-Scraping measures in place that prevent us from mass-downloading documents without using their API.

In this case, I write a copy of all of the links to a document called links.txt

```
output_file = 'links.txt'

for document in documents:
    doc_link = document.get('link')
    with open(output_file, 'a') as file:
        file.write(f'{doc_link}\n')
```

So the full program goes:

```
from bs4 import BeautifulSoup

documents = []
with open('site.html', 'r') as file:
    soup = BeautifulSoup(file, 'html.parser')
    table = soup.find('table', {'id': 'docsDataTable'})
    rows = table.find_all('tr')
    for row in rows:
        cells = row.find_all('td')
        if len(cells) == 5: # Line added to filter out empty/malformed table rows
            document = {"title": cells[0].text, "link": cells[4].find('a')['href']}
            documents.append(document)

output_file = 'links.txt'

for document in documents:
    doc_link = document.get('link')
    with open(output_file, 'a') as file:
        file.write(f'{doc_link}\n')
```

Now we have a full list of urls to documents in one place. We can then set up a [Google Drive API application](#) and use this [script](#) that uses the API to download all of the documents if desired.



## Not so easy

### Peeking at robots.txt

Many sites give you some insight into how aggressively they will be anti-scraping by going to the site/robots.txt. If you're lucky, it might give you some user agents for bots it does allow, and at what rates.

### Javascript-Heavy Sites

Thus far, we have been scraping mostly static sites that do not rely on a lot of Javascript to load their content. More and more of the web uses front-end Javascript and/or back-end PHP generated content to provide a dynamic user experience. Think of, for example, a Facebook feed, which varies depending on who is logged in. To scrape dynamic content like this, requests and BeautifulSoup isn't going to cut it. It requires the use of user-interaction based web scrapers like [Selenium](#) or [Playwright](#) along with the use of either a headless or headed browser.

### Tokenized Content

Sites like the [Georgia Department of Community Health](#) do not provide raw links to assets like PDFs, but instead generate a tokenized viewer for the PDF before download, and do not expose the location of the raw content. To make it harder, the GDCH also makes the tokenized links based on passing the correct cookie and header content, so even if you get the structure right, you need to get a legitimate cookie and rotate it every ten files or so. In these cases, you will have to do some research on the underlying structure of the page to find what are known as [Hidden APIs](#). I'll show you how I did so with the GDCH.

The [Macon-Bibb County Board Meetings Site is similar](#).

I will show you how I found the hidden API here.

### SSL Errors

Sites like the [Office of the State Inspector General for Virginia](#) have improperly setup SSL, complicating retrieving content using python requests. You will have to do things like pass `verify=False` with your get().

### Rate Limiting

#### 1. User Agent Blocking

Many sites deter scraping by imposing arbitrary limits like rejecting network requests from non-browsers (like the Python requests library). This is known as **user agent blocking**.

You can bypass user agent blocking by copying a legitimate user agent from your web browser or using a [fake user agent list](#) and rotating through them for each request

#### 2. IP Blocks

Requestors can also get IP blocked- meaning that you can only get a certain amount of content within 1 minute. Most APIs also have rate limits. You can slow down your requests to not reach the threshold, but it is likely that you still might get hit over time with a ban for behavior. In which case, you will want to create a pool of IP addresses also known as a [proxy pool](#).

### 3. Device blocks

Some sites make it harder for certain classes of IPs and devices in general to get the content. They analyze the behavior, rather than the content of the requests, to issue bans. So even if you say change user agents and IPs, it may not serve commercial IPs or desktop users certain types of content. In this case you will want to consider using mobile proxies. These proxies use IP addresses and user agent information that follow the behavior of mobile phones, which are much less likely to be detected as bots or scrapers. There are services that make this process easier, like [ScraperAPI](#). **All mobile proxies cost money and the vast majority of sites do not require their use to scrape.**

### 4. CAPTCHAs

Captchas suck. Sometimes they only occur once every ten requests, in which case we can rotate our IPs and user agent to defeat the captcha, but other times a CAPTCHA is generated every request.

If you are using a webdriver, you can implement more waits to back off to avoid CAPTCHAs, but it isn't always so simple. Or if you are using requests, you can time your requests to change pool when you expect the next request will be a CAPTCHA.

[Some CAPTCHAS](#) can be [defeated by machine learning](#) or other clever trickery, but there are classes of CAPTCHAS which do not have a way to be solved quickly by machines.

When you inevitably meet a site that has CAPTCHA every request, you will need either a machine-based CAPTCHA solver or to pay for a service ([example](#)) where a human solves the CAPTCHAs for you.

There are some ethical decisions to be made about using human based CAPTCHA solvers as the workers are paid nearly nothing to do the work. I advise against them and the sites using them probably are not worth it.