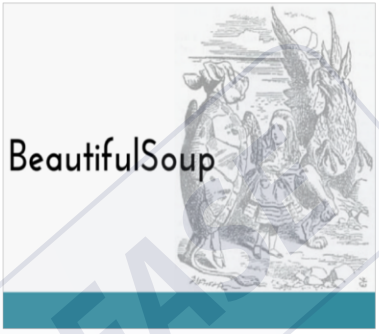
REVIEWED
By RISHABH YADAV at 3:43 pm, Aug 12, 2019

Rishabh
CONFIDENTIAL

Web scraping – REPORT BY RISHABH YADAV / Rishabhpranav

Web scraping, web harvesting, or web data extraction is data scraping used for extracting data from websites. Web scraping software may access the World Wide Web directly using the Hypertext Transfer Protocol, or through a web browser

Web Scraping Using Python and BeautifulSoup!



What is Web Scraping ?

Web Scraping (also termed Screen Scraping, Web Data Extraction, Web Harvesting etc.) is a technique employed to extract large amounts of data from websites whereby the data is extracted and saved to a local file in your computer or to a database in table (spreadsheet) format.

Data displayed by most websites can only be viewed using a web browser. They do not offer the functionality to save a copy of this data for personal use. The only option then is to manually copy and paste the data - a very tedious job which can take many hours or sometimes days to complete. Web Scraping is the technique of automating this process, so that instead of manually copying the data from websites, the Web_Scraping_software will perform the same task within a fraction of the time.

That being said, the actual code for webscraping is pretty simple.

Step 1: Find the URL you want to scrape.

One of my favorite things to scrape the web for, is to find speeches by famous politicians, scrape the text for the speech, and then analyze it for how often they approach certain topics, or use certain phrases. However, as with any sites, some of these speeches are protected, and scraping can be prohibited. Before you try to start scraping a site, it's a good idea to check the rules of the website first. The scraping rules can be found in the robots.txt file, which can be found by adding a /robots.txt path to the main domain of the site.

Step 2: Identify the structure of the sites HTML

Once you've found a site that you can scrape, you can use chrome's developer tools to inspect the site's HTML structure. This is important, because more than likely, you'll want to scrape data from certain HTML elements, or elements with specific classes or IDs. With the inspect tool, you can quickly identify which elements you need to target.

Step 3: Install Beautiful Soup and Requests

There are other packages and frameworks, like Scrapy. But Beautiful Soup allows you to parse the HTML in a a beautiful way,

so that's what I'm going to use. With Beautiful Soup, you'll also need to install a Request library, which will fetch the url content.

If you aren't familiar with it, the Beautiful Soup documentation has a lot of great examples to help get you started as well.

To install these two packages, you can simply use the pip installer.

```
$ pip install requests
```

and then...

```
$ pip install beautifulsoup4
```

Step 4: Web Scraping Code

Finally, we can start writing some code. Here's how I structured mine:

```
from bs4 import BeautifulSoup
import requests
# Here, we're just importing both Beautiful Soup and the Requests
librarypage_link =
'the url you want to scrape.scrape it real good.com'
# this is the url that we've already determined is safe and legal
to scrape from.page_response = requests.get(page_link, timeout=5)
# here, we fetch the content from the url, using the requests
librarypage_content = BeautifulSoup(page_response.content,
"html.parser")
#we use the html parser to parse the url content and store it in a
variable.textContent = []
for i in range(0, 20):
    paragraphs = page_content.find_all("p")[i].text
    textContent.append(paragraphs)
# In my use case, I want to store the speech data I mentioned
earlier.  so in this example, I loop through the paragraphs, and
push them into an array so that I can manipulate and do fun stuff
with the data.
```

Step 5: Isolating the results:

In the line of code above:

```
paragraphs = page_content.find_all("p")[i].text
```

This basically finds all of the <p> elements in the HTML. the `.text` allows us to select only the text from inside all the <p> elements. The difference is that without the `.text`, our return would probably look like this:

```
<p class="paragraph" id="7E33CH" >Lorem Ipsum is unattractive, both inside and out. I fully understand whyn. You know, it really doesn’t matter what you write as long as you’ve got a young, <a href="linktoacoolsite.com">and</a> beautiful, piece of text. I think my strongest asset maybe by far is my temperament. I have a placeholding temperament. Lorem Ipsum's father was with Lee Harvey Oswald prior to Oswald's being, you know, shot.</p>
```

This can be a little messy, and so filtering the results using the Beautiful Soup `.text`allows us to get a cleaner return, which might look more like this:

```
Lorem Ipsum is unattractive, both inside and out. I fully understand why it’s former users left it for something else. They made a good decision. You know, it really doesn’t matter what you write as long as you’ve got a young, and beautiful, piece of text. I think my strongest asset maybe by far is my temperament. I have a placeholding temperament. Lorem Ipsum's father was with Lee Harvey Oswald prior to Oswald's being, you know, shot.
```

Beautiful Soup also has a host of other ways to simplify and navigate the data structure:

```
soup.title
# <title>Returns title tags and the content between the tags</title>soup.title.string
# u'Returns the content inside a title tag as a string'soup.p
# <p class="title"><b>This returns everything inside the paragraph tag</b></p>soup.p['class']
# u'className' (this returns the class name of the element)soup.a
# <a class="link" href="http://example.com/example" id="link1">This would return the first matching anchor tag</a>// Or, we could use the find all, and return all the matching anchor tagssoup.find_all('a')
# [<a class="link" href="http://example.com/example1" id="link1">link2</a>,
```

```
#  <a class="link" href="http://example.com/example2"
id="link2">like3</a>,
#  <a class="link" href="http://example.com/example3"
id="link3">Link1</a>]soup.find(id="link3")
# <a class="link" href="http://example.com/example3"
id="link3">This returns just the matching element by ID</a>
```

There are a lot of other great ways to search, filter and isolate the results you want from the HTML. You can also be more specific, finding an element with a specific class or attribute:

```
soup.findAll('div',attrs={"class":"cool_paragraph"})
```

This would fine all the elements with the class "cool_paragraph".

Should I scrape the web in the first place?

An alternative to web scraping is using an API, if one is available. Obviously, in many cases, this isn't an option, but API's do provide faster and often more reliable data. Here are a few great APIs https://www.programmableweb.com/news/most-popular-apis-least-one-will-surprise-you/2014/01/23. Some APIs also provide more content than what would be available through web scraping.

Be Polite

Web scraping can also overload a server, if you are making a large amount of requests, and scraping large amounts of data. As I mentioned earlier, it's a good idea, before you start, to check the robots.txt before scraping.

Another good way to be polite when scraping is to be completely transparent, and even notify people to let them know you're going to crawl their site, why you are doing it, and what you are using the data for. One way to do this, and highly recommended, is to use a user agent. You can import a user agent library in python by pip installing the user_agent
library. https://pypi.org/project/user_agent/.

Your user agent can provide information like a link to more information about the scraper you're using. Your page about the scraper can and should include the information about what you're using it for, what IP address you are crawling from, and possibly a way to contact you, if your bot causes any problems.

The point is that web scraping can cause problems, and we don't want to cause problems. More than likely, at some point we will probably make mistakes that might affect a website. I think the gold rule is to just be open and honest in communicating to webmasters. If you respond to complaints quickly, you should be fine. Sometimes you may realize that your scraper has caused a problem even before the site you're scraping realizes it. In that case, it's an even better idea to make the first contact and basically say, "Hey, sorry about that. Here's what I did, and I fixed it."

In Conclusion:

I realize this really just scratches the surface of web scraping. And my intention isn't to go into a ton of detail here. Web scraping is actually pretty easy to get started. But doing it the right way takes a little more time and effort. Also, I'm still fairly new at this, and am by no means an expert on anything, and appreciate any feedback or tips!

Is Web Scraping legal?

Web scraping is an automated method used to extract large amounts of data from websites. The data on the websites are unstructured. Web scraping helps collect these unstructured data and store it in a structured form. There are different ways to scrape websites such as online Services, APIs or writing your own code. In this article, we'll see how to implement web scraping with python.

Talking about whether web scraping is legal or not, some websites allow web scraping and some don't. To know whether a website allows web scraping or not, you can look at the website's "robots.txt" file. You can find this file by appending "/robots.txt" to the URL that you want to scrape. For this example, I am scraping Flipkart website. So, to see the "robots.txt" file, the URL is www.flipkart.com/robots.txt.