Web Scraping With Beautiful Soup in Python

Web Scraping is a technique employed to extract large amounts of data from websites whereby the data is retrieved and saved to a local file in your computer or to a database in a table (spreadsheet) format. This process of web scraping can be a time-consuming as it requires us to know the layout of the document and isolate the items. However, Python users have access to a robust library that makes the process of web scraping easy & quick. The name of this library is `BeautifulSoup` and comes preinstalled on the Anaconda platform.

In this tutorial, we will explore how to use this library to scrape a Wikipedia page for data.

Tip: If you do not have \`BeautifulSoup\` installed on Python simply run the follwing pip command \`pip install beautifulsoup4\`.

Section One: Getting the Web Page Content

The first thing we will do is import the libraries and modules. The most obvious one is the BeautifulSoup library as this is the library we will be using to scrape the page. However, to get the HTML content of the web page, we will need on more library; the `urllib.request` Library. This library is used to open the URL we want to scrape and then pass the HTML content of that page into the beautiful soup parser.

In [1]:
```
# beautiful soup libraries
from bs4 import BeautifulSoup
from urllib.request import Request, urlopen
import urllib.request
```

With our libraries & modules imported, let's get started by defining the webpage URL & then requesting it. Once we have our request object, we can pass it through our BeautifulSoup object, while making sure to specify the `html.parser` as the content of the response object is HTML. The loading process is the real beauty of the `BeautifulSoup` library, **as it handles all the content for use behind the scenes.**

In [2]:
```
# Define & request the url that we want to scrape
wiki_url = r"https://en.wikipedia.org/wiki/Eastern_Front_(World_War_II)"
html_content = urllib.request.urlopen(wiki_url)

# Pass the html_content(the webpage) through our beautiful soup object
soup = BeautifulSoup(html_content, 'html.parser')
```

Once the content is loaded, we have a new BeautifulSoup object that we can easily traverse. Here is a simple example where we use the `find` method to find the first a tag in our HTML code. Now, we can make our search a little more narrow by defining an attribute we want to identify. In this example, I want the `href` attribute to exist in my a tag.

In [3]:
```
# find the first `a` tag that has an `href` attribute.
link = soup.find('a', href=True)

display(link)
```

```
<a href="/wiki/Wikipedia:Protection_policy#semi" title="This article is semi-
protected until June 21, 2019 at 18:17 UTC."><img alt="Page semi-protected" d
ata-file-height="512" data-file-width="512" decoding="async" height="20" src
="//upload.wikimedia.org/wikipedia/en/thumb/1/1b/Semi-protection-shackle.svg/
20px-Semi-protection-shackle.svg.png" srcset="//upload.wikimedia.org/wikipedi
a/en/thumb/1/1b/Semi-protection-shackle.svg/30px-Semi-protection-shackle.svg.
png 1.5x, //upload.wikimedia.org/wikipedia/en/thumb/1/1b/Semi-protection-shac
kle.svg/40px-Semi-protection-shackle.svg.png 2x" width="20"/></a>
```

Section Two: Tag Objects

It's not apparent at first, but any object that we look for is returned to us as another object that has specific properties about it. To be more explicit, it's a beautiful soup tag object. This particular object has particular features that make retrieving different portions of it more manageable.

- **name:** This is the name of the tag.
- **text:** This is the readable text of the label if there is any.
- **attrs:** This returns a dictionary of each attribute behaving as the key and the corresponding info as the value.

To access any of the attributes, we call the object and then pass through the key which will return the value.

In [4]:
```
# what is the type.
display(type(link))

# define the name.
display(link.name)

# get the dictionary attribute.
display(link.attrs)

# lets get each value in that dictionary, luckily there is only 2.
display(link['href'])
display(link['title'])

# we can also get the text, if there is any. In this case there wasn't any tex
t to return so it just returns an empty string.
display(link.text)
```

```
bs4.element.Tag

'a'

{'href': '/wiki/Wikipedia:Protection_policy#semi',
 'title': 'This article is semi-protected until June 21, 2019 at 18:17 UTC.'}

'/wiki/Wikipedia:Protection_policy#semi'

'This article is semi-protected until June 21, 2019 at 18:17 UTC.'

''
```

Section Three: Finding Tags & Content

We've seen how to search for one tag, but what if want to search for multiple tags of the same type. Well, this is almost identical to searching for one tag we change the method we use to **`find_all`**. In this example, I use `find_all` to find all the a tags in the document, I still only want the ones that have a link, so I keep the `href` attribute set to the value `True`.

In [5]:
```
# find all the links in the document.
links = soup.find_all('a', href=True)

# what is the type of this links object, it should be a bs4.element.ResultSet.
display(type(links))

# we can iterate through this result set using a for loop. In this loop I grab
the href and limit the result sets to 10 items.
for link in links[0:10]:
    print(link['href'])
```

```
bs4.element.ResultSet
```

```
/wiki/Wikipedia:Protection_policy#semi
#mw-head
#p-search
/wiki/Great_Patriotic_War_(term)
/wiki/The_Great_Patriotic_War_(The_Americans)
/wiki/Patriotic_War_of_1812
/wiki/European_theatre_of_World_War_II
/wiki/World_War_II
/wiki/File:EasternFrontWWIIcolage.png
/wiki/Ilyushin_Il-2
```

We've seen how using `find_all` can make the process of getting the tags we want significantly easier. However, we just looked at some simple examples. This particular method is very flexible as it allows us to find some of the following:

> - Multiple Tags
> - Tags with specific attributes
> - Tags with specific attributes & attribute values.
> - Using functions to narrow our search.

Here are some more "advanced" examples of using `find_all`.

In [6]:
```
# return a list of all the <table> & and <a> tag
tables_and_links = soup.find_all(['a', 'table'])

# return a list of all the <table> tag with a 'content' attribute
tables_with_cont = soup.find_all('table','content')

# return a list of all the <div> tags that have a class attribute of 'navbox'
div_with_nav = soup.find_all('div', class_ = 'navbox')

# define a function to find the item
def list_with_links(tag):
    return tag.name == 'li' and len(tag.find_all('a'))> 7

# use the function with find_all
list_items_with_links = soup.find_all(list_with_links)
```

Section Four: The Beautiful Soup Family Tree

Navigating the HTML tree is pretty simple, once you understand it's relationship to other elements in the document. For example, multiple items can fall under the same tag. Take, for example, a simple document provided by BeautifulSoup

```
<html>
  <body>
   <a>
    <b>
     text1
    </b>
    <c>
     text2
    </c>
   </a>
  </body>
 </html>
```

We would consider that element `<b>` and element `<c>` are **siblings** because they both fall under the same a tag element. The a tag signifies that means we can navigate between the two using the `next_sibling` & `previous_sibling` properties in our BeautifulSoup library. Below, I provide some examples of using this on our Wikipedia webpage data.

In [7]:
```
# get my sibling
my_sib = soup.p.next_sibling
my_next_sib = my_sib.next_sibling


# I can also go backwards.
my_prev_sib = my_next_sib.previous_sibling

display(my_sib)
display(my_next_sib)
display(my_prev_sib)
```

```
'\n'

<div class="hatnote navigation-not-searchable" role="note">"Great Patriotic W
ar" redirects here. For a discussion of the term itself, see <a href="/wiki/G
reat_Patriotic_War_(term)" title="Great Patriotic War (term)">Great Patriotic
War (term)</a>. For the episode of The Americans, see <a href="/wiki/The_Grea
t_Patriotic_War_(The_Americans)" title="The Great Patriotic War (The American
s)">The Great Patriotic War (The Americans)</a>.</div>

'\n'
```

Tip: We could also grab all the siblings at once using the `previous_siblings` or the `next_siblings` property.

Okay, not too bad right? Well, guess what there are even more ways to navigate the document. For example, what if we wanted to get the children of the children? Well, in this case, we would want all the **descendants** of a particular tag. In BeautifulSoup we can use the **descendants** property to iterate through all of a tags descendants recursively. The `descendants` property works great for objects like tables, as tables have multiple descendants (i.e., rows) that we can access. Here is how it would look:

In [ ]:
```
# loop through all the descendants in a table and print the text if there is a
ny.
for descendant in soup.table.descendants:

    if descendant.text:
        print(descendant.text)
```

Tip: Again, if want to access just the one descendant it would be the `descendant` property vice `descendants`.

What goes up must come down. The same applies here, if we have a descendant we must have something above, we call the element above the parent. To access this parent, we merely name the `parent` property which will allow us to access the parent. Also, if we want all the parents, we would use the `parents` property.

In [ ]:
```
# access the parents text if there is any.
for parent in soup.table.parents:

    if parent.text:
        print(parent.text)
```