Supplementary Chapter 1 - Case Study: Web Scraper

1 Case Study: Web Scraper

In this case study, we will experience how a web scraper can be made using python and a few other 3rd party libraries. A reference Jupyter Notebook file is available for this case study.

Things you will learn:

- Requests library
- Beautiful Soup library
- Python's Datetime library
- Basic HTML

Let's say that you need to gather all the financial headlines from several news website to scan it for the latest financial news or to get stock data of a particular company over a period of 3 days. If you were to do it manually, you would have to go to each website rather often to scan for any updates. This task is both tedious and time consuming that is where Web scraping comes into the picture.

Web scraping the act of using a script to "scrape" a certain set of data that you want to extract from one or more websites. It involves sending a request to the website (URL) and getting a response back from the server with the HTML or XML page. The data is then extracted from this HTML or XML page via the script. Web scraping follows these basic steps

1. Find the website's URL from which you want to get information from
2. Inspect the page using your browser's `Inspect` function
3. Locate the data you need
4. Write the code
5. Execute your code to extract the data
6. Save the data into a file or some storage medium

1.1 Legalities

But before we start web scraping, let's go through the legal aspects if it. Web scraping is still a considered a grey area in terms of legalities as it is highly dependent on the laws of the country from which you are from, some forms of scraped data is still illegal. Websites will still use techniques to limit web scraping like CAPTCHA tests or setting password access or use of anti-scraping tools. Some websites also have a `robots.txt` file which is accessible from their main website (eg `http://www.flipkart.com/robots.txt`) from which it details whether or not web scrapers or crawlers are not allowed to scrape or index their website. For instance if the `robots.txt` contains lines like

```
User-agent: *
Disallow:/
```

it means that the website owner do not allow scraping of any data from their website. Let's view a portion of the `robots.txt` from the `flipkart.com`.

```
User-agent: Mediapartners-Google
Disallow:

User-agent: Adsbot-Google
Disallow:

User-agent: Googlebot-Image
Disallow:

# cart
User-agent: *
Disallow: /viewcart

# Something related to carousel and recommendation carousel
User-agent: *
Disallow: /dynamic/
...
```

It shows that it allows Google's web crawlers to index its website but all web scraping and indexing is not allow on their cart pages (line 12), etc. Lets now take a look at the `robots.txt` from Channel News Asia website that we will be using for this case study.

```
User-agent: *
Sitemap: https://www.channelnewsasia.com/googlenews/cna_news_sitemap.xml
Sitemap: https://www.channelnewsasia.com/sitemap.xml
Crawl-delay: 10
```

It allows all web scraping but web crawlers have a time delay on each page being indexed.

1.2 Setting up for Web Scraping

Now that we ascertained that the Channel News Asia website is okay for us to use, we have to make sure that we have the required libraries for web scraping. Install the following libraries if you do not have them

- **Requests: HTTP for Humans** - it is a simple HTTP library that makes HTTP requests for humans easier to handle. It is also a wrapper (hides the nasties) for the Python's built-in `urllib3` library under the hood.

  ```
  # to install
  pip install requests
  ```

- **Beautiful Soup** - is a Python library used for pulling data out of HTML and XML files.

  ```
  # to install
  pip install beautifulsoup4
  ```

Once you have completed installing those libraries, open up your favourite browser and navigate to `https://www.channelnewsasia.com/news/singapore`. We will be scraping all the titles of the articles and their elapsed time (in hours) from the time of publishing. Refer to the figure 1 below.

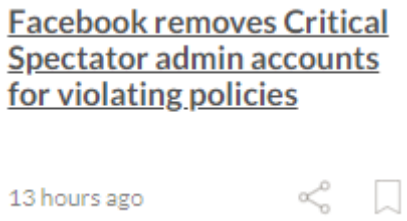

Figure 1: Article title and elapsed time.

To inspect the webpage, right click on the webpage and select the `Inspect` menu item from the context menu (refer to figure 2 below) or press `Ctrl + Shift + I` on the keyboard while the browser is active.

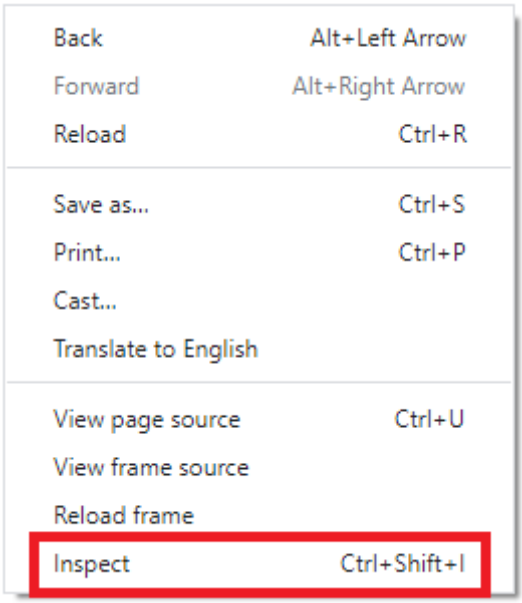

Figure 2: Location of the `Inspect` menu item on the context menu.

The `Inspect` menu item will bring up the inspection window in the browser with the HTML elements of the currently opened webpage. Figure 3 below shows the Inspector window from the Chrome web browser, it will look different on the other browsers.

Figure 3: Inspector window of the Chrome web browser.

Make sure that you are able to see the `Elements` of the webpage as that tab contains the HTML information of the loaded webpage. Hovering above the `div` elements in the Inspector window will highlight the elements within the webpage signifying that that `div` element is responsible for that part of the webpage.

Moving through and opening the `div` elements, we will find that the titles and time are encased a `a` and `time` HTML tag respectively. Refer to figure 4 below.

Figure 4: Identification of the HTML tags from which the data we are about to scrape.

Take note of the time element tag and its attribute `datetime`. We explain its significance in the next section.

1.3 Writing the code

At the start of even Python script, we need to import the required libraries. The libraries that we need are

```
# HTTP library for Python
import requests
# library for pulling data out of HTML and XML files
from bs4 import BeautifulSoup
# Python's datetime library
from datetime import datetime
```

Next we need to specify the website's URL.

```
url = "https://www.channelnewsasia.com/news/singapore"
```

To request a page from the server, we need to use a `GET` HTTP request. There are other HTTP request verbs but we will only be using the `GET` verb for this case study. With each `GET` request, the server will return with an appropriate `status code`. The status code that we want is a `200` status code which means `OK`. If you would like to read up on the other HTTP Status Codes, the documentation can be found here. Once we get an `OK` from the server, we can pass the received HTML page to `BeautifulSoup`.

```
# making the request for the webpage
page = requests.get(url)

# checking that the response is 200 (OK)
if page.status_code == 200:
    # passing the HTML page to BeautifulSoup using Python's html.parser
    soup = BeautifulSoup(page.text, 'html.parser')
else:
    # error getting the page
    print(page.status_code)
```

In the line 9, we see something called `html.parser`. This `html.parser` is a parser that provides `Beautiful Soup` idiomatic ways of navigating, searching and modifying the HTML parse tree. `Beautiful Soup` works with several different types of HTML parsers but we will be using the Python's built-in parser called `html.parser`. HTML pages uses tags to structure the content and those tags are placed in a tree structure. Each tag has an end tag, attributes and each attribute may or may not have a value. The format of a HTML tag is `<tag_name attribute=value attribute></tag_name>`.

As mention in the earlier section, we are looking for the tags `a` and `time`. The `a` tag is used to place a link on the webpage. Since there are many different `a` tags on that webpage, we need to narrow down the search by defining an attribute and value of the tag. Referring to that figure, lets pick the `class` attribute for both `a` and `time` tags. We can now use the `find_all()` function from `Beautiful Soup` to find all the `a` and `time` tags with the corresponding attribute `class` and values `teaser__title` and `teaser__time` respectively.

```
data = soup.find_all(['a', 'time'],
                     {'class':['teaser__title', 'teaser__time']})
```

The signature of the `find_all()` function is as follows `find_all(name, attrs, recursive, string, limit, **kwargs)`. The statement above uses the arguments `name` and `attrs`. The `name` argument tells `Beautiful Soup` which html tags to search for. That is then coupled with the other 5 attributes to narrow down the search. In this case, we have used the `attrs` argument to pass it a dictionary of values for the `class` attribute of the tags we are searching for. This `class` attribute of a HTML tag belongs to the Cascading Style Sheet (CSS) attribute called `class` which is used to determine the look and feel of the content on a webpage.

Because we are searching for 2 different tags with each having their different `class` values, we can either use a dictionary to group the `class` values list with the corresponding tag list or we can use 2 `find_all()` functions, one for each tag. The `find_all()` function returns a list of the results if it is able to find the tags otherwise it returns a `None`. More can be read about the `find_all()` function here. It is always good practice to check to see if the data being scraped was done properly by printing some of the first and last 2 to 3 elements of the list.

```
# check that the data we scraped is correct
print(len(data))
print(data[:2])
print(data[-2:])
```

You will notice that the elements in the list are the HTML tags and not the titles nor the time data that we want. In addition, the `time` tag does not have the same time information as on the element window in the Inspector. The `time` tag is actually a dynamic value and it gets recalculated each time we refresh the webpage. So how to we extract this information? The `time` tag has a `datetime` attribute that stores the epoch time that can be used to convert to human readable time with Python's `datetime` library. Epoch time is the a Unix based system for describing a point in time in seconds since 1 January 1970.

```
delimiter = '*'
header = 'article_name'+delimiter+'epoch_time\n'
data_to_write = [header]
temp = None

# extracting the data
for item in data:
    # checking for the 'a' tag
    if item.name == 'a':
        temp = item.get_text() + delimiter
    else:
        temp += item['datetime'] + '\n'
        data_to_write.append(temp)
        # reset the temporary variable
        temp = None
```

From lines 1 to 3, we can see that the data is going to be stored in a CSV file however, the delimiter (or separator) used is not the regular comma but an asterisk `*`. This delimiter is chosen because if you look at the article's titles, you can spot some of the titles have commas, semi-colons, colons, hyphens and apostrophes therefore the general delimiter have to be changed.

Now that the data have been processed, we can write it to file a `CSV` file. As this file is a computer generated file, we are going to append a timestamp to the file name.

```
1 # constructing the file name
2 timestamp = datetime.now().strftime("%d-%b-%Y_%H-%M-%S")
3 filename = 'scrape_'+timestamp+'.csv'
4
5 # writing the data to file
6 with open(filename, 'w') as fo:
7     for line in data_to_write:
8         fo.writelines(line)
```

Datetime is a Python built-in library for handling operations regarding dates and times. In line 2, we are getting the current date and time with the `now()` function then converting it to a string format using the `strftime()` function which accepts a formatting string `%d-%b-%Y_%H-%M-%S` as input and outputs a string of the date in accordance with the formatting string. The formatting string uses the C standard codes where

- `%d` - Day of the month as a zero-padded decimal number.
- `%b` - Month as locale's abbreviated name.
- `%Y` - Year with century as a decimal number.
- `%H` - Hour (24-hour clock) as a zero-padded decimal number.
- `%M` - Minute as a zero-padded decimal number.
- `%S` - Second as a zero-padded decimal number.

There are more string formatting codes and datetime information located here.

1.4 Bonus: How to work with Epoch time

We have seen earlier that the `datetime` attribute of the `time` tag has a string value that represents the epoch time and we also know that it is stored in seconds; so how do we convert it to normal date and time values if we need graph a time series chart that uses epoch time?

We use Python's `datetime` library! Epoch time is still considered a valid timestamp therefore we can use the function `datetime.fromtimestamp()` to convert it. Remember to convert the epoch string to an integer first.

```
1 timestamp = datetime.fromtimestamp(int(epoch))
```

Now that we have the converted timestamp, we can use it for our graphs.

1.5 References

1. How to scrape websites without getting blocked, https://www.scrapehero.com/how-to-prevent-getting-blacklisted-while-scraping/
2. Data: Engine for Growth – Implications for Competition Law, Personal Data Protection, and Intellectual Property Rights, https://www.cccs.gov.sg/resources/publications/occasional-research-papers/data-engine-for-growth
3. Unix time, https://en.wikipedia.org/wiki/Unix_time
4. Beautiful Soup Documentation, https://www.crummy.com/software/BeautifulSoup/bs4/doc/
5. Requests: HTTP for Humans, https://requests.readthedocs.io/en/master/
6. datetime — Basic date and time types, https://docs.python.org/3/library/datetime.html
7. Kho, Sept 2018, How to Web Scrape with Python in 4 Minutes, https://towardsdatascience.com/how-to-web-scrape-with-python-in-4-minutes-bc49186a8460
8. Hiremath, May 2020, A Beginner's Guide to learn web scraping with python!, https://www.edureka.co/blog/web-scraping-with-python/