# Import Splinter and BeautifulSoup

from splinter import Browser

from bs4 import BeautifulSoup as bs

from webdriver\_manager.chrome import ChromeDriverManager

Importing these libraries to utilize their functionalities for web scraping. The splinter module provides the browser automation capabilities, while beautifulsoup4 allows you to parse and extract data from HTML or XML documents. Additionally, webdriver\_manager is used to manage the web driver required by Splinter, such as the ChromeDriver.

executable\_path = {'executable\_path': ChromeDriverManager().install()}

browser = Browser('chrome', \*\*executable\_path, headless=False)

executable\_path with a key-value pair. The key is 'executable\_path', and the value is obtained by calling ChromeDriverManager().install(). The ChromeDriverManager() function from webdriver\_manager.chrome automatically fetches and installs the appropriate ChromeDriver executable based on your system and the installed Chrome browser.

Browser instance named browser using the Splinter library. The first argument 'chrome' specifies that you want to use the Chrome browser for automation.

The \*\*executable\_path syntax unpacks the executable\_path dictionary as keyword arguments. It passes the ChromeDriver executable path to the Browser instance, allowing it to locate and use the ChromeDriver.

The headless=False argument specifies that the browser should run in non-headless mode, which means that the browser window will be visible and actions performed by the script will be visible to the user. If you set headless=True, the browser will run in the background without a visible window.

After executing this code, I can use the browser object to perform various browser automation tasks, such as visiting URLs, interacting with web elements, and extracting data from web pages using Splinter's methods.

url = 'https://static.bc-edx.com/data/web/mars\_news/index.html'

browser.visit(url)

html=browser.html

The url variable to the desired web page URL that you want to visit. Then, the browser.visit(url) method is called to instruct the browser object to open the specified URL in the Chrome browser.

After visiting the URL, the browser.html attribute is used to retrieve the HTML content of the web page. The HTML content is then stored in the html variable for further processing.

Now, I have the HTML content of the web page stored in the html variable. I can use BeautifulSoup or other parsing libraries to extract specific data or perform any necessary web scraping operations on the HTML content.

articles = soup.find\_all('div', class\_='list\_text')

The find\_all() method in BeautifulSoup searches the HTML document for all occurrences of a specified tag and class. In this case, I am looking for <div> elements that have the class attribute set to 'list\_text'.

The result of find\_all() is stored in the articles variable. It will be a list containing all the matching <div> elements that meet the specified criteria.

for article in articles:

    # Extract the title and preview text from the elements.

    title = article.find('div', class\_='content\_title').text

    preview = article.find('div', class\_='article\_teaser\_body').text

    # Store each title and preview pair in a dictionary.

    article\_dict = {}

    article\_dict['title'] = title

    article\_dict['preview'] = preview

    # Add the dictionary to the list.

    articles\_lst.append(article\_dict)

Within the for loop, I iterate over each article in the articles list. For each article, I use the find() method to locate the title and preview text within the corresponding <div> element using their respective classes ('content\_title' and 'article\_teaser\_body').

I extract the text content of the title and preview elements using the .text attribute. Then, creating a dictionary named article\_dict and store the title and preview as key-value pairs within the dictionary.

Finally, I append the article\_dict dictionary to the articles\_lst list. This results in a list of dictionaries, with each dictionary containing the title and preview text for a single article.

Make sure to initialize the articles\_lst list before the for loop to avoid any potential issues.

with io.open('Mars\_news\_results.json', 'w', encoding='utf8') as outfile:

    str= json.dumps(articles\_lst,

                    indent=4, sort\_keys=True,

                    separators=(',', ': '), ensure\_ascii=False,

                    default=str)

    outfile.write(str)

First, the code imports the necessary modules, io and json, for file operations and JSON serialization.

Then, a with statement is used to open the file "Mars\_news\_results.json" in write mode ('w') with UTF-8 encoding (encoding='utf8'). The with statement ensures that the file is properly closed after writing.

Inside the with block, the json.dumps() function is used to convert the articles\_lst list to a JSON-formatted string. Several optional arguments are provided to customize the serialization process:

* indent=4: Specifies the indentation level in the output JSON for better readability.
* sort\_keys=True: Sorts the keys in the output JSON alphabetically.
* separators=(',', ': '): Configures the separators used in the output JSON.
* ensure\_ascii=False: Ensures that non-ASCII characters are properly encoded in the output JSON.
* default=str: Specifies a custom function (str in this case) to handle non-serializable objects.

The resulting JSON-formatted string is then written to the file using the outfile.write() method.

After executing this code, you should find a file named "Mars\_news\_results.json" containing the extracted article information in JSON format.

#1\_Extract table body.

temps\_tbl = soup.find('tbody')

#2\_Extract header row.

temps\_hdr = temps\_tbl.find\_all('th')

#3\_Extract all rows of data.

temps = temps\_tbl.find\_all('tr', class\_='data-row')

# Extract the header row in a list.

hdr\_lst = []

for each in temps\_hdr:

    hdr\_lst.append(each.text)

1. Extracting the table body: temps\_tbl = soup.find('tbody') This line of code finds the <tbody> element within the HTML document using BeautifulSoup's find method.
2. Extracting the header row: temps\_hdr = temps\_tbl.find\_all('th') Here, finding all the <th> elements within the temps\_tbl (table body) using the find\_all method.
3. Extracting all rows of data: temps = temps\_tbl.find\_all('tr', class\_='data-row') This line retrieves all the <tr> elements with the class name "data-row" from the temps\_tbl.

Finally, extract the text from each <th> element and store it in a list called hdr\_lst using a loop:

python

hdr\_lst = []

for each in temps\_hdr:

hdr\_lst.append(each.text)

This loop iterates over each element in temps\_hdr and extracts the text using the text attribute of BeautifulSoup. The extracted text is then appended to the hdr\_lst.

# Create an empty list

temps\_lst = []

# Loop through the scraped data to create a list of rows

for temp in temps:

    temp\_row = []

    for data in temp.find\_all('td'):

        temp\_row.append(data.text)

    temps\_lst.append(temp\_row)

1. I create an empty list called temps\_lst to store the rows extracted from the HTML table.
2. The outer loop iterates over each element in the temps list, which contains the rows of data extracted from the HTML table.
3. Inside the outer loop, I create an empty list called temp\_row to store the data for the current row.
4. The inner loop, using temp.find\_all('td'), finds all the <td> elements within the current row (temp).
5. For each <td> element found, extract the text content using the .text attribute and append it to the temp\_row list.
6. After iterating through all the <td> elements in the current row, append the temp\_row list to the temps\_lst list.
7. The process continues for each row in the temps list, populating temps\_lst with a nested list structure containing the extracted data from each row.
8. df['id'] = pd.to\_numeric(df['id'], downcast='integer')
9. df['terrestrial\_date'] = pd.to\_datetime(df['terrestrial\_date'])
10. df['sol'] = pd.to\_numeric(df['sol'], downcast='integer')
11. df['ls'] = pd.to\_numeric(df['ls'], downcast='integer')
12. df['month'] = pd.to\_numeric(df['month'], downcast='integer')
13. df['min\_temp'] = pd.to\_numeric(df['min\_temp'], downcast='float')
14. df['pressure'] = pd.to\_numeric(df['pressure'], downcast='float')
15. df['id'] = pd.to\_numeric(df['id'], downcast='integer'): This line converts the 'id' column in the DataFrame df to the integer data type using pd.to\_numeric(). The downcast='integer' parameter is used to downcast the data type to the smallest possible integer type.
16. df['terrestrial\_date'] = pd.to\_datetime(df['terrestrial\_date']): Here, the 'terrestrial\_date' column is converted to a datetime data type using pd.to\_datetime(). This function interprets the values in the column as dates and converts them accordingly.
17. df['sol'] = pd.to\_numeric(df['sol'], downcast='integer'): The 'sol' column is converted to the integer data type in a similar manner as shown in the first line.
18. df['ls'] = pd.to\_numeric(df['ls'], downcast='integer'): The 'ls' column is converted to the integer data type using the pd.to\_numeric() function with downcast='integer'.
19. df['month'] = pd.to\_numeric(df['month'], downcast='integer'): Similarly, the 'month' column is converted to the integer data type using pd.to\_numeric() with downcast='integer'.
20. df['min\_temp'] = pd.to\_numeric(df['min\_temp'], downcast='float'): The 'min\_temp' column is converted to the float data type using pd.to\_numeric() with downcast='float'. This allows for decimal values in the column.
21. df['pressure'] = pd.to\_numeric(df['pressure'], downcast='float'): Lastly, the 'pressure' column is converted to the float data type in the same way as the previous line.