DAY_2

Web Scraping

Beautiful Soup

- Beautiful Soup is a Python library used to parse HTML and XML documents.
- It's especially useful for web scraping because it helps navigate, search, and modify the HTML (or XML) content fetched from a webpage.
- It transforms complex HTML documents into a tree structure, where each node corresponds to elements such as tags, text, attributes, etc.
- This makes it easy to locate and extract specific information.

Advantages of Beautiful Soup:

- Easy to Learn and Use: Has a user-friendly syntax that makes it easy for users to quickly locate and extract data from web pages.
- Flexible Parsing: Works with different parsers, such as the built-in Python parser or lxml, offering flexibility in terms of speed and error handling
- Handles Broken HTML: Automatically fixes errors in the HTML structure, allowing users to scrape data from pages that other parsers might struggle with.
- Efficient Navigation and Search Functions: Provides intuitive functions like find, find_all, and select to search and navigate through HTML tags and CSS selectors.
- Integration with Other Libraries: Integrates smoothly with libraries like Requests, to retrieve web pages before parsing them. Also works well with Pandas for data analysis or Selenium for JavaScript heavy pages, making it a versatile choice for a complete web scraping workflow.
- Well-Documented and Active Community: Has a comprehensive documentation and an active community that provides resources, tutorials, and troubleshooting support, making it accessible for new users.

Applications of Beautiful Soup:

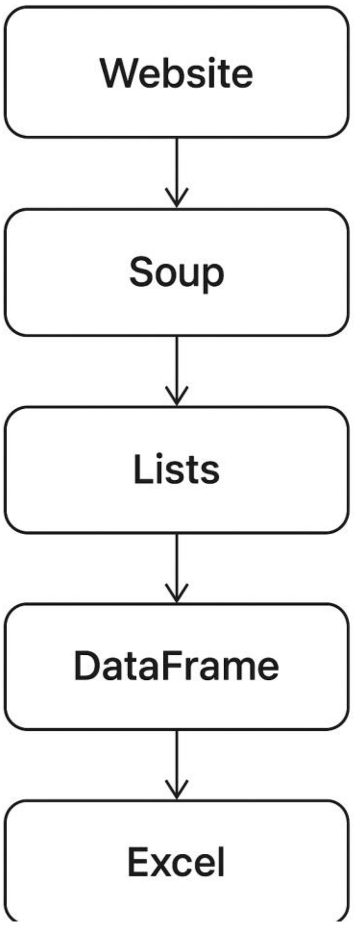
- **Price Comparison and Monitoring:** Widely used by e-commerce companies and consumers to scrape prices from various online stores.
- **Job Listings Aggregation**: Commonly used to scrape job listings from platforms like LinkedIn, indeed, or company career pages. This can help create job aggregators that compile positions from various sources.
- **Market Research and Sentiment Analysis:** Companies often use web scraping to collect data from forums, blogs, and review sites to analyze customer sentiment about their products or their competitors.

- **Real Estate Listings:** Useful for gathering real estate listings from sites like *Zillow* or *Realtor.com*. Data on prices, locations, features, and property availability can be scraped and analysed to identify trends, track prices, and help potential buyers and real estate investors make informed decisions.
- **Travel and Flight Price Tracking:** Used to monitor and compare prices across different airlines, hotels, and booking platforms. By gathering this data, users can develop apps to track flight and accommodation prices, helping travelers find the best deals.

Mini Project:

I've completed a mini project on web scraping. Specifically, I scraped data from the BBC website's sports section, focusing on the top scorers in the Premier League.

Flow Diagram



1. Importing libraries

```
import requests

import pandas as pd

from bs4 import BeautifulSoup
```

- **requests** → To send HTTP requests to the website.
- **pandas** → To create and manipulate a DataFrame and save it to Excel.
- **BeautifulSoup** → To parse HTML content and extract data.

2. Setting up the URL and making a request

```
url = "https://www.bbc.com/sport/football/premier-league/top-scorers"

response = requests.get(url)

response.raise_for_status()  # Raises an error if request fails

response.status_code        # Check if request was successful (200 = OK)

type(response.content)      # Checks the type of response (bytes)
```

- You are sending a **GET request** to the BBC Premier League top scorers page.
- raise_for_status() ensures the program stops if there's an error (like 404 or 500).
- response.content contains the raw HTML of the webpage.

3. Parsing the webpage

```
soup = BeautifulSoup(response.content, "html.parser")

print(soup.prettify())
```

- BeautifulSoup parses the HTML content and creates a **soup object** that is easy to navigate.
- prettify() prints the HTML in a readable format.

4. Creating lists to store data

```
player_names = []

team_names = []

goals = []

assists = []

num_matches = []

shots = []
```

- Empty lists are created to store data for each column: **player, team, goals, assists, matches, shots**.

5. Handling the request with a try-except block

```
try:
  response = requests.get(url)
  response.raise_for_status()
except Exception as e:
  print(e)
```

- Ensures that if there is any network or HTTP error, it prints the error instead of crashing.

6. Extracting data

```
soup = BeautifulSoup(response.content, 'html.parser')

players = soup.find('tbody').find_all('tr', class_='ssrcss-qqhdqi-TableRowBody e1icz100')
```

- Finds the **table body (tbody)** containing player rows.
- Each row <tr> represents a player, filtered by a specific **CSS class**.

7. Looping through each player row

```
for player in players:

  player_name = player.find('div', class_='ssrcss-m6ah29-PlayerName e1n8xy5b1').get_text(strip=True)

  team_name = player.find('div', class_='ssrcss-qvpga1-TeamsSummary e1n8xy5b0').get_text(strip=True)

  goals_scored = int(player.find('div', class_='ssrcss-18ap757-CellWrapper ef9ipf0').get_text(strip=True))


  stats = player.find_all('div', class_='ssrcss-1vo7v3r-CellWrapper ef9ipf0')

  assists_made = int(stats[0].get_text(strip=True))

  matches_played = int(stats[2].get_text(strip=True))

  shots_taken = int(stats[-3].get_text(strip=True))

  player_names.append(player_name)

  team_names.append(team_name)

  goals.append(goals_scored)

  assists.append(assists_made)

  num_matches.append(matches_played)

  shots.append(shots_taken)
```

- For each player row:
    - Extract **player name**, **team**, and **goals scored**.
    - Extract **assists**, **matches played**, and **shots** from a list of other stats.
- Data is appended to the respective lists.

8. Creating a DataFrame

```
data = {
   'player': player_names,
   'team': team_names,
   'matches': num_matches,
   'goals': goals,
   'assists': assists,
   'shots': shots
}
df_players = pd.DataFrame(data)
```

- Combines all lists into a **dictionary**.
- Converts the dictionary into a **pandas DataFrame** for easier handling and exporting.

9. Exporting data to Excel

```
df_players.to_excel('EPL Top Scorers.xlsx', index= False)
```

- Saves the DataFrame as an **Excel file** named EPL Top Scorers.xlsx.
- index=False prevents pandas from writing the row numbers in Excel.

Summary:
This code scrapes the **top scorers in the Premier League** from the BBC Sports website, collects **player stats** (name, team, goals, assists, matches, shots), and saves them into an **Excel file**.