# WEB SCRAPPING

## Problem Statement

- The goal of this project is to collect and analyse motivational quotes from the "Quotes to Scrape" website.
- The website contains quotes from various authors, along with relevant metadata such as tags and links to further details.
- Manually extracting and organizing this data is time-consuming and inefficient.
- **Task:** An automated approach is needed to efficiently scrape the data from multiple pages, organize it, and save it in a structured format for further analysis.

## Solution

To address this problem, we will develop a web scraping script using Python. The script will:

1. Send HTTP requests to the "Quotes to Scrape" website.
2. Parse the HTML content of the webpage using BeautifulSoup to extract quotes, authors, and other relevant information.
3. Loop through multiple pages of the website to gather a comprehensive dataset.
4. Store the scraped data in a structured format using a pandas DataFrame.
5. Save the data to CSV files for easy access and analysis.

## Objective

The primary objectives of this project are:

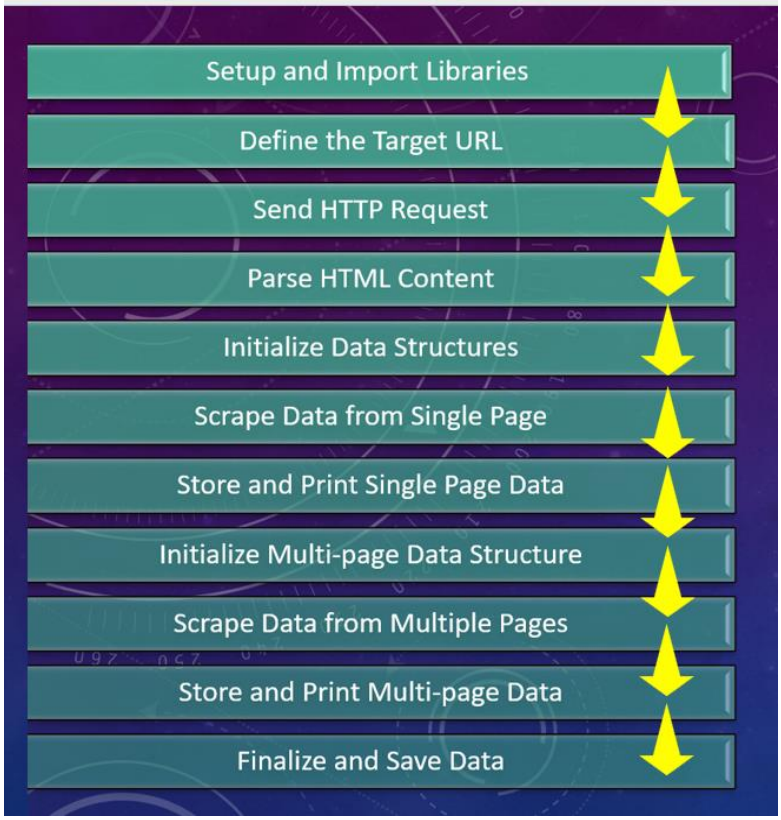
1. **Automate Data Collection:** Develop a script to automate the process of collecting quotes and related metadata from the "Quotes to Scrape" website.
2. **Extract and Organize Data:** Parse the HTML content to extract quotes, authors, tags, and details links, and organize this data into a structured format.
3. **Handle Multiple Pages:** Implement functionality to scrape data from multiple pages of website to ensure a comprehensive dataset.
4. **Save Data for Analysis:** Save collected data into CSV files for further analysis.

## Modules Of the Project

The project consists of the following modules:

1. **Requests**
   - Used for making HTTP requests to fetch the HTML content of the web pages
   - Utilized to send requests to the website to scrape quotes and related information.
2. **BeautifulSoup**
   - Used for parsing the HTML content retrieved from website
   - Allows extraction of specific data elements such as quotes, authors' names, details links, and tags from the HTML structure.
3. **Pandas**
   - Used for data manipulation and analysis.(scrapes data into DataFrames, making it easier to work and analyse the data.)
   - Facilitates the storage of the scraped data into CSV files.

## Workflow




1. Setup and Import Libraries:
   a. Import necessary libraries: requests for making HTTP requests, BeautifulSoup for parsing HTML, and pandas for data manipulation and storage.
2. Define the Target URL:
   a. Set the URL of the website to scrape (https://quotes.toscrape.com/).
3. Send HTTP Request:
   a. Use the requests library to send a GET request to the target URL and capture the response.
4. Parse HTML Content:
   a. Create a BeautifulSoup object to parse the HTML content of the webpage from the response.
5. Initialize Data Structures:
   a. Initialize empty lists to store quotes, authors, and additional data.
6. Scrape Data from Single Page:
   a. Loop through HTML elements with specified classes to extract quotes, authors, and other relevant information.
   b. Store the extracted data in the initialized lists.
7. Store and Print Single Page Data:
   a. Convert the scraped data from lists into a pandas DataFrame.
   b. Print the DataFrame to verify the extracted data.
   c. Save the DataFrame to a CSV file (scraped_data.csv).
8. Initialize Multi-page Data Structure:
   a. Initialize an empty list to store data from multiple pages.
9. Scrape Data from Multiple Pages:
   a. Loop through multiple pages by constructing URLs for each page.
   b. For each page, send a GET request and parse the HTML content.
   c. Extract quotes, authors, and other relevant information from each page and store it in the multi-page data list.

10. Store and Print Multi-page Data:
    a. Convert the multi-page scraped data into a pandas DataFrame.
    b. Print the DataFrame to verify the combined extracted data from all pages.
    c. Save the multi-page DataFrame to a CSV file (scraped_data_multiple_pages.csv).
11. Finalize and Save Data:
    a. Ensure all data is saved and printed correctly.
    b. The project is complete with the scraped data stored in CSV files for further analysis or use.

## Code

Python Code

```
# Import necessary libraries
import requests  # For making HTTP requests
from bs4 import BeautifulSoup  # For parsing HTML
import pandas as pd  # For data manipulation and analysis


# Define the URL of the website to scrape
link = 'https://quotes.toscrape.com/'


# Sending a request to the website and getting the response
res = requests.get(link)


# Creating a soup object to parse the HTML content
soup = BeautifulSoup(res.text, 'html.parser')


# Scraping quotes and authors' names
# Initialize empty lists to store quotes and authors
quotes = []
authors = []
```

Python Code

```
# Loop through all elements with the specified class and extract text
for quote in soup.find_all('span', class_='text'):
    quotes.append(quote.text[1:-1])  # Append the quote text
# Loop through all elements with the specified class and extract text
for i in soup.find_all('small', class_='author'):
    authors.append(i.text)  # Append the author's name
# Initialize a list to store data from multiple pages
data = []
# Loop through all elements with the specified class and extract desired information
for sp in soup.find_all('div', class_='quote'):
    quote = sp.find('span', class_='text').text[1:-1]  # Extract quote text
    author = sp.find('small', class_='author').text  # Extract author's name
    details = sp.find('a').get('href')  # Extract details link
    tags = [tag.text for tag in sp.find_all('a', class_='tag')]  # Extract tags
    tags = ','.join(tags)  # Convert tags list to a comma-separated string
    data.append([quote, author, details, tags])  # Append data to the list
# Initialize a list to store data from multiple pages
multiple_pages_data = []
# Loop through multiple pages and scrape data
for page in range(1, 11):
    # Construct the URL for each page
    page_link = f"http://quotes.toscrape.com/page/{page}"
    # Send a request to the website
    res = requests.get(page_link)
    # Create a soup object to parse the HTML content
    soup = BeautifulSoup(res.text, 'html.parser')
    # Loop through all elements with the specified class and extract desired information
    for sp in soup.find_all('div', class_='quote'):
        quote = sp.find('span', class_='text').text[1:-1]  # Extract quote text
        author = sp.find('small', class_='author').text  # Extract author's name
```

Python Code

```
    details = sp.find('a').get('href')  # Extract details link
    tags = [tag.text for tag in sp.find_all('a', class_='tag')]  # Extract tags
    tags = ','.join(tags)  # Convert tags list to a comma-separated string
    multiple_pages_data.append([quote, author, details, tags])  # Append data to the list

# Convert the list of data into a DataFrame
df = pd.DataFrame(data, columns=['Quote', 'Author', 'Details', 'Tags'])

# Print the DataFrame
print("Scraped Data from Single Page:")
print(df)

# Save the DataFrame to a CSV file
df.to_csv('scraped_data.csv', index=False)
print("Scraped data saved to 'scraped_data.csv'.")

# Convert the list of data from multiple pages into a DataFrame
df_multiple_pages = pd.DataFrame(multiple_pages_data, columns=['Quote', 'Author',
'Details', 'Tags'])

# Print the DataFrame from multiple pages
print("\nScraped Data from Multiple Pages:")
print(df_multiple_pages)

# Save the DataFrame from multiple pages to a CSV file
df_multiple_pages.to_csv('scraped_data_multiple_pages.csv', index=False)
print("Scraped data from multiple pages saved to 'scraped_data_multiple_pages.csv'.")
```

## Reason for Choosing BeautifulSoup

In this project, BeautifulSoup was chosen over other web scraping libraries for several reasons:

1. Simplicity and Ease of Use:
   - BeautifulSoup is very straightforward and easy to learn
   - excellent choice for beginners
   - quick implementation.
   - syntax is intuitive
   - integrates well with the requests library, which makes sending HTTP requests and parsing HTML responses simple and efficient.
2. Suitability for Static Pages: As target website (quotes.toscrape.com) is a static website, meaning the content does not require JavaScript execution to render. BeautifulSoup, when combined with requests, is perfect for such scenarios because it efficiently handles the parsing of static HTML content.
3. Project Requirements: As project involves basic HTML parsing and extraction of data (quotes, authors, details, tags). BeautifulSoup is highly effective for these tasks as it provides powerful tools for navigating, searching, and modifying the parse tree.

Alternatives and Their Use Cases

1. Scrapy:
   - Scrapy is a powerful and comprehensive web scraping framework designed for large-scale projects. It includes features for handling requests, processing data, and storing scraped content.
   - Use Case: Suitable for more complex scraping tasks that require managing requests, handling errors, following links, and processing large amounts of data. Ideal for projects where scalability and advanced scraping capabilities are necessary.
2. Selenium:
   - Selenium is primarily used for browser automation and can interact with JavaScript-heavy websites where content is dynamically loaded.
   - Use Case: Essential for scraping websites that rely on JavaScript to render content, performing actions like clicking buttons, filling forms, and navigating through multiple pages. Selenium controls a real browser, so it's also useful for testing web applications.
3. Scrapy-Selector:
   - Scrapy-Selector is part of the Scrapy framework and provides an API for selecting and extracting data from HTML and XML using CSS and XPath selectors.
   - Use Case: Typically used within the Scrapy ecosystem for enhanced data extraction capabilities. Ideal for projects where precise and flexible selection mechanisms are required.
4. Requests-HTML:
   - Requests-HTML is a library that combines the power of requests and parsing capabilities with the added ability to execute JavaScript.
   - Use Case: Suitable for projects that require both HTTP requests and JavaScript execution, bridging the gap between simple requests and more complex browser automation.

## Why BeautifulSoup Fits This Project

- Static Content: Since the target website is static, there is no need for JavaScript execution, making BeautifulSoup a more lightweight and efficient choice.

- Quick Setup: BeautifulSoup has a quick setup time with minimal boilerplate code, which aligns well with the project's simplicity and direct requirements.
- Focus on Parsing: The project focuses on parsing and extracting data from HTML, which BeautifulSoup handles exceptionally well with its powerful and flexible parsing capabilities.

In summary, BeautifulSoup, combined with the requests library, provides a simple, efficient, and effective solution for the requirements of this web scraping project.

## Key Learning

Key Learnings

1. Understanding Web Scraping Basics
2. Using requests Library
3. HTML Parsing with BeautifulSoup
4. Data Storage and Manipulation with pandas
5. Handling Single and Multi-Page Scraping
6. Efficiency and Suitability of Tools
7. Recognized efficiency of using BeautifulSoup for projects involving static websites and basic data extraction tasks.
8. Error Handling and Data Validation
9. Ethical Considerations and Best Practices

   By completing this project, we have built a solid foundation in web scraping and data extraction, equipped with practical skills and knowledge that can be applied to more advanced and varied web scraping tasks in the future.

## Conclusion

This web scraping project successfully demonstrates the extraction of data from a static website using Python libraries. By leveraging requests to handle HTTP requests and BeautifulSoup for parsing HTML content, we were able to scrape quotes, authors, and additional information from both single and multiple pages of the target website (quotes.toscrape.com). The scraped data was then stored in pandas DataFrames and saved to CSV files for further analysis or use. The project highlights the efficiency and simplicity of using BeautifulSoup for basic web scraping tasks, particularly when dealing with static web pages.