Webscraping Tutorial

Jonathan Jayes

2024-08-25

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction and Purpose

Welcome to the Web Scraping course, where we embark on a journey from the basics of web scraping to the development of a fully automated, interactive web application. This course is designed to provide you with a solid foundation in web scraping, combined with practical skills in Python programming, automation with GitHub Actions, and building dynamic web apps using Streamlit.

1.0.0.1 Course Objectives:

By the end of this course, you will be able to:

1. **Set Up a Python Environment:**
   Learn how to create and manage a virtual environment using `venv`, ensuring that your web scraping projects are isolated and dependencies are handled efficiently. This step is crucial for maintaining clean, reproducible, and conflict-free projects.
2. **Scrape Websites Efficiently:**
   Master the essential techniques for web scraping using Python libraries such as `requests` and `BeautifulSoup`. You'll start by scraping simple websites, like `example.com`, and gradually move to more complex tasks, such as extracting structured data from tables on websites like the SARS exchange rates page.
3. **Inspect and Analyze Web Pages:**
   Develop skills in inspecting web pages using tools like SelectorGadget, enabling you to identify the correct HTML elements and CSS selectors to target for scraping. Understanding the structure of a webpage is key to extracting the right data.
4. **Automate Web Scraping Tasks:**
   Automate the scraping process by setting up a GitHub repository and creating a GitHub Action with a CRON job. This will allow you to schedule your scraper to run at specific intervals, ensuring your data remains current without manual intervention.

5. **Build an Interactive Web Application:**
   Finally, you'll use Streamlit to build a web application that leverages your scraped data. In this course, you'll create a currency conversion tool that updates in real-time with data scraped from the SARS website, demonstrating how scraping can be integrated into a practical, user-friendly tool.

1.0.0.2 Course Structure:

This course is divided into the following sections:

1. **Getting Started:**
   - Set up your development environment with a virtual environment and install the necessary Python libraries (`requests`, `BeautifulSoup`).
   - Detailed instructions are provided for both Windows and MacOS users.
2. **Scraping `example.com`:**
   - A step-by-step guide to scraping a simple webpage.
   - Introduction to using SelectorGadget for identifying HTML elements to scrape.
3. **Scraping a Table from the SARS Website:**
   - Learn how to extract structured data from tables and save it locally.
   - Troubleshoot common issues in scraping table data.
4. **Automating Scraping with GitHub Actions:**
   - Automate your scraper to run on a schedule using GitHub Actions and CRON jobs.
   - Monitor and manage the automated process effectively.
5. **Building a Streamlit App:**
   - Develop an interactive Streamlit app for currency conversion, using the data you've scraped.
   - Focus on creating a user-friendly interface and ensuring app responsiveness.

> **Note**
>
> **Note:** This course is designed to take you from the basics of web scraping to building an interactive application that automates scraping tasks and leverages the data in a practical tool. Whether you are a beginner in web scraping or looking to enhance your skills, this course will guide you through every step of the process.

1.0.1 Getting the Most Out of This Course

- **Hands-On Learning:**
  Each section is designed with practical exercises. You'll be writing and running Python code, inspecting web pages, and ultimately building a real-world application.
- **Build as You Learn:**
  The course is structured to incrementally build up to a final project. By the end, you'll not only have the knowledge but also a complete, functional app to showcase your new skills.
- **Stay Curious:**
  Web scraping is a dynamic field with constantly evolving techniques and tools. Throughout the course, you'll find links to further reading and advanced topics. Use these resources to deepen your understanding and explore new possibilities.

With that introduction, you're now ready to dive into the first section: **Getting Started**. Let's set up your environment and get you ready for the exciting tasks ahead!

2 Getting Started: Setting Up a Virtual Environment and Installing Packages

---

In this section, we will cover the essential steps to set up your Python development environment. Setting up a virtual environment ensures that your projects are isolated, allowing you to manage dependencies effectively without affecting your global Python installation. This is particularly important for maintaining a clean and conflict-free workspace.

2.0.0.1 1. Setting Up a Virtual Environment with `venv`

A virtual environment in Python is an isolated environment that allows you to install packages and dependencies specific to your project without interfering with other projects or the global Python installation.

2.0.0.1.1 1.1 Setting Up on MacOS

Follow these steps to create and activate a virtual environment on MacOS:

1. **Open Terminal:**
   Start by opening the Terminal application on your Mac.
2. **Navigate to Your Project Directory:**
   Use the `cd` command to navigate to the directory where you want to create your project.

   ```
   cd path/to/your/project-directory
   ```

3. **Create a Virtual Environment:**
   Use the following command to create a virtual environment. Replace `myenv` with the name you want for your virtual environment.

   ```
   python3 -m venv myenv
   ```

4. **Activate the Virtual Environment:**
   After creating the virtual environment, you need to activate it. Run the following command:

   ```
   source myenv/bin/activate
   ```

   Once activated, you'll see the name of your virtual environment (e.g., `(myenv)`) appear at the beginning of your command line prompt, indicating that the environment is active.

5. **Deactivate the Virtual Environment:**
   When you are done working in the virtual environment, deactivate it by running:

   ```
   deactivate
   ```

2.0.0.1.2 1.2 Setting Up on Windows

For Windows users, the process is slightly different:

1. **Open Command Prompt or PowerShell:**
   You can use either Command Prompt or PowerShell to set up your virtual environment.

2. **Navigate to Your Project Directory:**
   Use the `cd` command to navigate to your project directory.

   ```
   cd path\to\your\project-directory
   ```

3. **Create a Virtual Environment:**
   Create a virtual environment using the following command. Replace `myenv` with your chosen environment name.

   ```
   python -m venv myenv
   ```

4. **Activate the Virtual Environment:**
   To activate the virtual environment, run:

   - **For Command Prompt:**

     ```
     myenv\Scripts\activate
     ```

   - **For PowerShell:**

     ```
     .\myenv\Scripts\Activate.ps1
     ```

After activation, you should see `(myenv)` at the beginning of your prompt, indicating that the virtual environment is active.

5. **Deactivate the Virtual Environment:**
   To deactivate the environment when you are done, simply run:

```
deactivate
```

Tip: Virtual Environment

Setting up a virtual environment ensures that your project dependencies are isolated and easy to manage. This isolation prevents conflicts between different projects and keeps your global Python environment clean.

Warning

Warning: Always remember to activate your virtual environment before installing any packages to ensure they are installed within the environment and not globally.

2.0.0.2 2. Installing Packages with `pip`

Once your virtual environment is activated, you can install the necessary packages using `pip`, Python's package manager. We'll start by installing two essential libraries for web scraping: `requests` and `BeautifulSoup4`.

2.0.0.2.1 2.1 Installing Packages

1. **Ensure Your Virtual Environment is Activated:**
   Before installing any packages, make sure your virtual environment is active. If you see `(myenv)` at the beginning of your command line, you're good to go.
2. **Install `requests` and `BeautifulSoup4`:**
   Use the following command to install both packages:

```
pip install requests beautifulsoup4
```

This command tells `pip` to download and install the `requests` library, which allows you to send HTTP requests, and `BeautifulSoup4`, a library for parsing HTML and XML documents.

3. **Verify the Installation:**
   After installation, you can verify that the packages were installed correctly by listing the installed packages:

   ```
   pip list
   ```

   You should see `requests` and `beautifulsoup4` in the list of installed packages.

2.0.0.2.2 2.2 Installing Specific Versions of Packages

If you need to install a specific version of a package, you can specify the version number:

```
pip install requests==2.26.0
```

This command will install version 2.26.0 of the `requests` package. Specifying versions can be useful when working on projects that require a particular version of a library.

2.0.0.2.3 2.3 Freezing Requirements

To make it easier to recreate the same environment later or share with others, you can "freeze" your environment's current state to a `requirements.txt` file:

```
pip freeze > requirements.txt
```

This file lists all the packages and their versions currently installed in your environment. Later, you or someone else can recreate the same environment by running:

```
pip install -r requirements.txt
```

> **Note**
>
> **Note:** Keeping your `requirements.txt` file up to date ensures that others can easily set up an identical environment for your project.

With your virtual environment set up and the necessary packages installed, you're ready to start working on your web scraping project. In the next section, we'll begin by scraping data from a simple webpage, `example.com`, and exploring how to use tools like SelectorGadget to identify the elements you need.

3 Scraping Basics and Scraping `example.com`

In this section, we will dive into the fundamentals of web scraping and provide a hands-on example by scraping a simple webpage, `example.com`. This exercise will help you understand how to send HTTP requests, parse HTML content, and extract specific data from a webpage.

3.0.0.1 1. What is Web Scraping?

Web scraping is the process of programmatically extracting information from websites. It involves fetching the content of a webpage, parsing the HTML, and selecting the data you need. Web scraping is a powerful tool for gathering data from the web for various applications, such as data analysis, machine learning, and automation.

However, it's important to note that not all websites allow scraping, and some have measures in place to prevent it. Always check a website's `robots.txt` file or terms of service to ensure you are scraping responsibly and within legal boundaries.

! Understanding `robots.txt`

Important: Always respect the terms of service of the website you are scraping. Be mindful of legal and ethical considerations when scraping data, and avoid overloading servers with frequent requests.
A `robots.txt` file is a simple text file placed at the root of a website to provide instructions to web crawlers and bots about which parts of the website they are allowed to access and index. When a web scraper or search engine bot visits a site, it typically checks the `robots.txt` file first to see what it is permitted to scrape.

3.0.1 Key Points:

- **Location:** The `robots.txt` file is located at the root of a website, usually accessible via `http://example.com/robots.txt`.

- **Syntax:** The file consists of directives that allow or disallow access to specific parts of the site. For example:

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

  This example tells all user agents (crawlers) that they are not allowed to access `/private-directory/`.

- **Respecting `robots.txt`:** While the directives in `robots.txt` are not legally binding, ethical web scraping practices dictate that you should respect them. Ignoring these directives could lead to legal repercussions or your IP address being blocked by the website.

- **Limitations:** Keep in mind that `robots.txt` is a voluntary protocol. Some bots might choose to ignore it, and it cannot prevent all forms of access.

Before scraping a website, always check the `robots.txt` file to ensure your activities align with the website's guidelines.

3.0.1.1 2. Overview of Scraping Process

The general process for web scraping involves the following steps:

1. **Sending an HTTP Request:**
   Use a Python library like `requests` to send an HTTP request to the server hosting the website.
2. **Retrieving the Response:**
   The server returns an HTTP response, typically containing the HTML of the webpage.
3. **Parsing the HTML:**
   Use a library like `BeautifulSoup` to parse the HTML and navigate the structure of the webpage.
4. **Identifying and Extracting Data:**
   Locate the specific HTML elements (e.g., `<div>`, `<p>`, `<a>`, etc.) that contain the data you want and extract it.
5. **Saving or Processing the Data:**
   Store the extracted data in a suitable format, such as a CSV file or a database, for further analysis or use.

3.0.1.2 3. Scraping `example.com`: A Hands-On Example

To get started with web scraping, we'll walk through a simple example using `example.com`, a placeholder domain provided by IANA (Internet Assigned Numbers Authority). While this website contains very basic content, it is perfect for demonstrating the fundamental concepts of web scraping.

3.0.1.2.1 3.1 Sending an HTTP Request

First, we'll send an HTTP GET request to `example.com` using the `requests` library.

```
import requests

# Send a GET request to the webpage
url = "http://example.com"
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    print("Successfully fetched the webpage!")
else:
    print(f"Failed to fetch the webpage. Status code: {response.status_code}")
```

In this code snippet: - We import the `requests` library. - We define the URL for `example.com`. - We send a GET request to fetch the webpage. - We check if the request was successful by inspecting the status code (200 indicates success).

3.0.1.2.2 3.2 Parsing the HTML

Next, we need to parse the HTML content of the page using `BeautifulSoup` to make it easier to navigate and extract specific elements.

```
from bs4 import BeautifulSoup

# Parse the HTML content
soup = BeautifulSoup(response.text, 'html.parser')

# Print the title of the webpage
title = soup.title.string
```

```
print(f"Page Title: {title}")
```

In this snippet: - We import `BeautifulSoup` from the `bs4` library. - We parse the HTML content of the response using `BeautifulSoup` with the HTML parser. - We extract and print the title of the webpage using the `<title>` tag.

3.0.1.2.3 3.3 Extracting Specific Data

Let's extract specific elements from the page. For example, we'll extract all the paragraphs (`<p>` tags) on the page.

```
# Extract and print all paragraphs
paragraphs = soup.find_all('p')
for idx, paragraph in enumerate(paragraphs, start=1):
    print(f"Paragraph {idx}: {paragraph.text}")
```

In this snippet: - We use `soup.find_all('p')` to find all paragraph elements on the page. - We loop through each paragraph and print its content.

3.0.1.2.4 3.4 Summary of Extracted Data

For `example.com`, the content is very simple. The title of the page is "Example Domain," and the body contains just a few paragraphs explaining the purpose of the domain. By following these steps, you've successfully scraped and extracted data from a webpage!

> Tip: Inspecting Web Pages
>
> Use your web browser's developer tools (usually accessible by right-clicking on the page and selecting "Inspect") to explore the HTML structure of a webpage. This helps in identifying the tags and classes associated with the data you want to scrape.

3.0.1.3 4. Using SelectorGadget to Identify HTML Elements

To scrape more complex websites, it's essential to accurately identify the HTML elements that contain the data you need. SelectorGadget is a browser extension that simplifies this process by allowing you to click on elements and automatically generating the correct CSS selector.

3.0.1.3.1 4.1 Installing SelectorGadget

1. **Chrome:**
   Install the SelectorGadget extension from the Chrome Web Store.
2. **Firefox:**
   Install the SelectorGadget bookmarklet by visiting the SelectorGadget website.

3.0.1.3.2 4.2 Using SelectorGadget

1. **Activate SelectorGadget:**
   Click the SelectorGadget icon in your browser to activate it.
2. **Select Elements:**
   Hover over elements on the webpage to highlight them. Click to select an element. SelectorGadget will generate a CSS selector that you can use in your scraping script.
3. **Refine Selection:**
   If the initial selection is too broad or too narrow, you can click on additional elements to refine the selector. The goal is to create a selector that precisely targets the data you want to extract.

3.0.1.3.3 4.3 Applying the Selector in Your Code

Once you have the CSS selector from SelectorGadget, you can use it in your `BeautifulSoup` code to target specific elements.

```
# Example: Using a CSS selector to find elements
selected_elements = soup.select('css-selector-from-selectorgadget')
for element in selected_elements:
    print(element.text)
```

Replace `'css-selector-from-selectorgadget'` with the actual selector provided by SelectorGadget.

3.0.2 Conclusion

In this section, you've learned the basics of web scraping, including how to send HTTP requests, parse HTML content, and extract specific data. You've also seen how to use SelectorGadget to simplify the process of identifying HTML elements on more complex webpages.

With these foundational skills, you are now ready to tackle more advanced scraping tasks. In the next section, we will explore scraping structured data from a table on the SARS website and saving it for further use.

---

4 Scraping a Table from the SARS Website

In this section, you will learn how to scrape data from a table on the SARS website, specifically the exchange rates table, and save this data to a pandas DataFrame. Once the data is in a DataFrame, we will also demonstrate how to save it as a JSON file for further use.

4.0.0.1 1. Inspecting the Webpage

The first step in web scraping is to inspect the webpage to identify the structure and the elements you want to extract. The SARS Rates of Exchange page contains a table with exchange rates for various currencies. This table is defined using standard HTML `<table>` tags, with each row (`<tr>`) containing data for a different currency.

To scrape this table, we'll focus on extracting the following data for each currency: - Country name - Abbreviation - Currency name - Exchange rate

The HTML structure of the table includes: - A table element with class `table table-bordered gvExt`. - Each row (`<tr>`) contains several cells (`<td>`), where: - The first cell contains the country flag (which we will ignore). - The second cell contains the country name. - The third cell contains the currency abbreviation. - The fourth cell contains the currency name. - The fifth cell contains the exchange rate.

4.0.0.2 2. Setting Up Your Environment

Before we start scraping, ensure that you have the necessary Python libraries installed in your virtual environment. We will use `requests` to fetch the webpage content, `BeautifulSoup` to parse the HTML, and `pandas` to store the data in a DataFrame.

```
pip install requests beautifulsoup4 pandas
```

4.0.0.3 3. Writing the Scraper

Below is the step-by-step guide to scraping the exchange rates table from the SARS website:

4.0.0.3.1 3.1 Importing the Required Libraries

Start by importing the necessary libraries.

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
```

4.0.0.3.2 3.2 Sending a GET Request to Fetch the Webpage

We'll send an HTTP GET request to retrieve the HTML content of the webpage.

```
url = "https://tools.sars.gov.za/rex/Rates/Default.aspx"
response = requests.get(url)

if response.status_code == 200:
    print("Successfully fetched the webpage!")
else:
    print(f"Failed to fetch the webpage. Status code: {response.status_code}")
```

4.0.0.3.3 3.3 Parsing the HTML with BeautifulSoup

Next, parse the HTML content using BeautifulSoup.

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

4.0.0.3.4 3.4 Locating the Table and Extracting Data

We will locate the table by its class name and then iterate over the rows to extract the relevant data.

```
# Locate the table
table = soup.find('table', {'class': 'table table-bordered gvExt'})

# Prepare lists to store the extracted data
countries = []
```

```
abbreviations = []
currencies = []
rates = []

# Loop through each row in the table
for row in table.find_all('tr')[1:]:  # Skip the header row
    cells = row.find_all('td')
    country = cells[1].text.strip()
    abbreviation = cells[2].text.strip()
    currency = cells[3].text.strip()
    rate = float(cells[4].text.strip())

    # Append the data to the lists
    countries.append(country)
    abbreviations.append(abbreviation)
    currencies.append(currency)
    rates.append(rate)
```

4.0.0.3.5 3.5 Creating a Pandas DataFrame

Now that we have the data, we can create a pandas DataFrame to organize it.

```
# Create a DataFrame
df = pd.DataFrame({
    'Country': countries,
    'Abbreviation': abbreviations,
    'Currency': currencies,
    'Rate': rates
})

# Display the DataFrame
print(df)
```

4.0.0.3.6 3.6 Saving the Data to a JSON File

Finally, let's save the DataFrame to a JSON file for further use.

```
# Save the DataFrame to a JSON file
df.to_json('exchange_rates.json', orient='records', indent=4)
```

```
print("Data has been saved to exchange_rates.json")
```

4.0.1 4. Summary

In this section, you learned how to: 1. Inspect a webpage to locate a table for scraping. 2. Write a Python script to scrape data from the table using `requests` and `BeautifulSoup`. 3. Store the scraped data in a pandas DataFrame. 4. Save the DataFrame to a JSON file.

This basic approach to scraping tables can be applied to many other websites, allowing you to automate the extraction of tabular data for analysis, reporting, or other applications.

> Tip: Handling Changes in Webpage Structure
>
> Websites can change their HTML structure over time, which may break your scraper. To maintain your scraping scripts, periodically check the structure of the pages you scrape and update your code as needed.

In the next section, we will explore how to automate this scraping process using GitHub Actions, enabling you to keep your data up-to-date without manual intervention.

References