Webscraping Tutorial

Jonathan Jayes

2024-08-25

Table of contents

Preface

Welcome to this tutorial on web scraping using Python, Lisa, Jacques and Almaro! Whether you're new to python programming or looking to enhance your skills in data extraction, this course is designed to guide you through the essential techniques of web scraping. Throughout the tutorial, we will cover everything from setting up your Python environment to scraping simple websites, extracting structured data from tables, automating your scraping tasks, and even building an interactive web application.

One of the additional benefits of this course is the opportunity to deepen your understanding of Python itself. Python is a versatile scripting language that's not only powerful but also widely supported. This means that there's a wealth of information available online, and chatbots like ChatGPT are particularly skilled at generating Python code that works.

To enhance your learning experience, I've included videos from Vincent at Calmcode for almost every chapter. Vincent's tutorials are high-quality, well-explained, and can provide you with deeper insights into both Python and web scraping. If you find yourself needing more information or a different perspective on a topic, I highly recommend checking out his Calmcode courses.

To get started with Python, here's a brief introductory video that explains the basics:

In addition to the Calmcode videos and other resources, remember that the Python community is vast and supportive. Whether you're troubleshooting an issue or looking for guidance on a specific task, there's a good chance you'll find help online.

I hope you find this course both informative and enjoyable.

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.

1.0.1 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 Virtual Environments

Figure 2.1: See this video on calmcode.io - Setting Up a Virtual Environment.

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. Additionally, you should add your virtual environment to your `.gitignore` file to prevent it from being uploaded to GitHub. A `.gitignore` file is a text file that tells Git which files or directories to ignore in a project. This is useful for excluding files that are not necessary for others to see or use, such as

your virtual environment, which can be recreated using the `requirements.txt` file.

Tip

Task: To practice setting up a virtual environment and installing packages, create a new virtual environment named `.venv` and install the `requests` and `BeautifulSoup4` libraries. Verify that the packages are installed correctly by listing them using `pip list`.

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.

2.0.1 Further Reading and Resources

To learn more about the importance and usage of virtual environments in Python, you can explore the following resources:

- **Python Virtual Environments: A Primer** – A comprehensive guide on why and how to use virtual environments in Python.
- **The Hitchhiker's Guide to Python: Virtual Environments** – This guide covers the essentials of virtual environments and offers tips for managing them effectively.
- **Python Documentation: venv** – The official Python documentation for the `venv` module, including advanced usage and customization options.

Understanding and effectively using virtual environments will significantly improve your workflow as a Python developer, ensuring that your projects remain organized, consistent, and free of dependency conflicts.

3 Scraping Basics

Figure 3.1: See this video on calmcode.io - an alternative to beautifulsoup.

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.

Here is an example of using SelectorGadget to identify the CSS selector for a specific element on a webpage:

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.

Tip

Task: To practice scraping a webpage, try scraping `example.com` using the code snippets provided. Inspect the HTML content of the page, extract specific elements like paragraphs, and experiment with different CSS selectors using 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

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.

> **Tip**
>
> **Task:** To practice scraping the SARS exchange rates table, run the provided code in your Python environment and examine the extracted data in the DataFrame. You can also open the `exchange_rates.json` file to view the saved data.
> Try to scrape a table on Wikipedia or another website of your choice if you want to extend yourself further.

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.

5 Automating Scraping

Figure 5.1: See this video on calmcode.io - an explanation of github actions.

In this section, we will explore how to automate the web scraping process using GitHub Actions. This includes understanding what GitHub Actions are, how to configure a workflow using YAML files, the role of CRON jobs in automation, and the benefits of running a scraping script on a server (or "someone else's computer").

5.0.0.1 1. What are GitHub Actions?

GitHub Actions is a powerful feature provided by GitHub that allows you to automate tasks directly from your GitHub repository. These tasks, known as **workflows**, can be triggered by specific events, such as pushing code to a repository, opening a pull request, or on a scheduled basis using CRON jobs.

With GitHub Actions, you can automate a wide range of tasks, such as running tests, deploying applications, or in our case, executing a web scraping script regularly to ensure your data is always up-to-date.

Key Features of GitHub Actions: - **CI/CD Automation:** Automate your continuous integration and continuous deployment (CI/CD) pipelines. - **Custom Workflows:** Define custom workflows to perform tasks automatically in response to GitHub events. - **Extensible:** Use pre-built actions from the GitHub Marketplace or write your own custom actions.

5.0.0.2 2. Understanding YAML Files

To configure GitHub Actions, you use **YAML** files. YAML (YAML Ain't Markup Language) is a human-readable data format that's commonly used for configuration files. In the context of GitHub Actions, YAML files define the steps of your workflow, including when and how your tasks should be run.

Example YAML File Structure:

```yaml
name: Web Scraping Automation

on:
  schedule:
    - cron: '0 0 * * *'  # This runs the script daily at midnight

jobs:
  scrape-data:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.x'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install requests beautifulsoup4 pandas

    - name: Run scraper
      run: python scrape_sars.py
```

Breaking Down the YAML File: - **`name:`** This defines the name of the workflow (e.g., "Web Scraping Automation"). - **`on:`** Specifies the event that triggers the workflow. In this case, `schedule` is used to run the workflow on a CRON schedule. - **`jobs:`** Defines one or more jobs that run as part of the workflow. Each job runs on a specified environment (e.g., `ubuntu-latest`). - **`steps:`** Lists the steps to be executed within the job, such as checking out the code, setting up Python, installing dependencies, and running the scraping script.

5.0.0.3 3. What are CRON Jobs?

CRON is a time-based job scheduler in Unix-like operating systems. It allows you to run scripts or commands automatically at specified intervals. In the context of GitHub Actions, CRON jobs enable you to schedule workflows to run at specific times or intervals.

CRON Syntax: - CRON syntax consists of five fields: `minute`, `hour`, `day of month`, `month`, and `day of week`. - For example, `0 0 * * *` means "run the script daily at midnight".

Common CRON Schedules: - `0 0 * * *`: Every day at midnight. - `*/30 * * * *`: Every 30 minutes. - `0 12 * * 1-5`: Every weekday at 12:00 PM.

Using CRON jobs with GitHub Actions allows you to automate tasks like scraping data at regular intervals without manual intervention.

5.0.0.4 4. The Purpose of Running a Scraping Script on a Server

Running a scraping script on a server—essentially "someone else's computer"—has several advantages, especially when it comes to automation and reliability.

Why Use a Server for Scraping? - 24/7 Availability: Servers are typically always on and connected to the internet, meaning your scraping script can run continuously at scheduled intervals without needing your local machine to be on. - **Resource Efficiency:** By running the script on a server, you offload the computational work and resource consumption from your own machine. - **Automation:** Automating the scraping process on a server ensures that the script runs regularly without the need for manual intervention, keeping your data up-to-date. - **Centralized Management:** Managing the script from a centralized server (like GitHub Actions) allows for better version control, easier collaboration, and consistent execution environments. - **Scalability:** As your scraping tasks grow, running them on a server allows you to scale more efficiently, handling larger datasets and more frequent scraping without burdening your local resources.

Example Use Case: Imagine you have a scraper that needs to gather exchange rate data daily from the SARS website. Instead of running this script manually every day on your computer, you set it up to run automatically using GitHub Actions. With CRON scheduling, the script will execute at the same time every day, fetch the data, and store it in your repository, ready for analysis or further processing.

5.0.1 Automating Data Addition to Your Repository with GitHub Actions

---

To fully automate the process of scraping data and adding it to your GitHub repository, there are several important steps to follow. This section will guide you through configuring your GitHub repository settings and modifying your YAML workflow file to ensure that your scraping script not only runs on a schedule but also updates your repository with the latest data.

5.0.1.1 1. Configuring GitHub Repository Settings

Before your GitHub Actions workflow can push changes to your repository, you need to adjust the repository settings to allow the actions to write data.

5.0.1.1.1 Steps to Configure Repository Settings:

1. **Navigate to Your Repository Settings:**
   - Go to your GitHub repository.
   - Click on the "Settings" tab.
2. **Enable GitHub Actions Permissions:**
   - In the left sidebar, select "Actions."
   - Under "General" settings, find the "Workflow permissions" section.
   - Select **"Read and write permissions."** This allows GitHub Actions to commit changes to your repository.
   - Ensure the option **"Allow GitHub Actions to create and approve pull requests"** is also enabled if you plan to use this feature.
3. **Save Changes:**
   - Click "Save" to apply these settings.

5.0.1.2 2. Modifying the YAML Workflow File

Once your repository is configured, you need to adjust your YAML workflow file to automate the process of pulling the latest data, adding the new scraped data, and pushing the changes back to the repository.

5.0.1.2.1 Steps in the YAML Workflow File:

Here's an example of how you can modify your YAML file:

```yaml
name: Web Scraping Automation

on:
  schedule:
    - cron: '0 0 * * *'  # Run daily at midnight

jobs:
  scrape-and-update:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout the repository
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.x'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install requests beautifulsoup4 pandas

    - name: Run scraper
      run: python scrape_sars.py

    - name: Pull latest changes
      run: git pull origin main

    - name: Commit and push changes
      run: |
        git config --global user.name "github-actions[bot]"
        git config --global user.email "github-actions[bot]@users.noreply.github.com"
        git add .
        git commit -m "Updated exchange rates on $(date)"
        git push origin main
```

Explanation of the YAML File:

- **`name: Web Scraping Automation`**: This is the name of the workflow.
- **`on: schedule:`**: This specifies that the workflow runs on a CRON schedule, set to run daily at midnight.

- `scrape-and-update:`: The job defined here will run on the latest Ubuntu environment.
- `Checkout the repository:`: This step checks out the code from your repository so that the action can make changes to it.
- `Set up Python:`: This sets up the Python environment to run your scraping script.
- `Install dependencies:`: Installs the necessary Python libraries for the scraping script.
- `Run scraper:`: This runs the `scrape_sars.py` script, which scrapes the data and saves it locally.
- `Pull latest changes:`: This step ensures that the workflow pulls the latest version of the repository before making any changes. This prevents conflicts if other changes have been made to the repository since the last run.
- `Commit and push changes:`: This step adds the new scraped data, commits it with a custom message (including the date), and pushes it back to the main branch of the repository.

5.0.1.2.2 Custom Commit Message:

- The commit message `git commit -m "Updated exchange rates on $(date)"` includes a timestamp, making it clear when the data was last updated. You can customize this message to include a salutation or any other information you find relevant.

5.0.1.3 3. Reusing and Repurposing YAML Files

While YAML files are powerful and allow for a high degree of customization, they can be complex and verbose. It's important to understand that:

- **You Don't Need to Master YAML Syntax:** Most users don't need to write YAML files from scratch. Instead, you can reuse and repurpose existing YAML files, modifying them to suit your specific needs.
- **Leverage Existing Templates:** GitHub provides many examples and templates in the GitHub Actions marketplace. These can be adapted to automate tasks like running tests, deploying code, or, as we've seen, scraping and updating data.
- **Focus on Understanding Key Concepts:** The key is to understand the core concepts, such as how to trigger actions (`on:`), what jobs and steps do, and how to manage dependencies. With this understanding, you'll be able to modify YAML files effectively without needing to write them from scratch.

5.0.2 Summary

By configuring your GitHub repository and setting up a properly structured YAML workflow file, you can fully automate the process of scraping data and updating your repository. This approach ensures that your data is always current, with minimal manual intervention.

Most importantly, remember that while YAML files are a powerful tool for automation, the focus should be on reusing and adapting existing templates rather than writing them from scratch. This allows you to automate tasks efficiently without needing deep expertise in YAML syntax.

In the next section, we'll guide you through building a Streamlit app that leverages this automated data to create a dynamic, interactive user interface.

6 Streamlit App

Stretch Goal: Building a Streamlit Currency Conversion App

Figure 6.1: See this video on calmcode.io - an explanation the Streamlit library.

As a stretch goal, you can take your web scraping project to the next level by building a dynamic and interactive web application using Streamlit. This app will read in the latest exchange rate data from your public GitHub repository and allow users to convert various amounts of foreign currencies into South African Rands (ZAR) using the most recent exchange rates.

6.0.0.1 1. Project Overview

The goal of this task is to create a user-friendly web app where users can: - Select a foreign currency from a dropdown list. - Enter an amount in the selected foreign currency. - Instantly see the equivalent amount in South African Rands based on the latest exchange rates.

This app will pull the exchange rate data directly from your GitHub repository, ensuring that it always uses the most up-to-date information scraped and stored by your automated GitHub Actions workflow.

6.0.0.2 2. Steps to Build the Streamlit App

6.0.0.2.1 2.1 Set Up the Streamlit Environment

First, you need to set up your Python environment for Streamlit:

1. **Install Streamlit:** If you haven't already installed Streamlit, do so by running:

   ```
   pip install streamlit
   ```

2. **Start a New Streamlit App:** Create a new Python file, for example, `app.py`, where you will write your Streamlit code.

6.0.0.2.2 2.2 Fetch the Data from GitHub

Your app will need to fetch the latest exchange rate data stored in your public GitHub repository. Here's how you can do that:

1. **Import Necessary Libraries:**

```
import streamlit as st
import pandas as pd
import requests
import json
```

2. **Fetch the JSON Data from GitHub:**

   Use the `requests` library to fetch the JSON file from your repository:

```
url = "https://raw.githubusercontent.com/your-username/your-repo/main/exchange_rates.
response = requests.get(url)
data = response.json()
```

3. **Load the Data into a DataFrame:**

   Convert the JSON data into a pandas DataFrame for easy manipulation:

```
df = pd.DataFrame(data)
```

6.0.0.2.3 2.3 Build the Streamlit User Interface

Next, you'll build the interface where users can interact with the app:

1. **Create the Dropdown for Currency Selection:**

```
st.title("Currency to ZAR Converter")

currencies = df['Currency'].tolist()
selected_currency = st.selectbox("Select a currency", currencies)
```

2. **Input Field for the Amount:**

   Allow the user to enter the amount they want to convert:

```
amount = st.number_input(f"Enter amount in {selected_currency}", min_value=0.0)
```

3. **Calculate the Equivalent in Rands:**

   Fetch the exchange rate for the selected currency and calculate the equivalent amount in Rands:

```
if selected_currency:
    rate = df.loc[df['Currency'] == selected_currency, 'Rate'].values[0]
    equivalent_in_rands = amount / rate
    st.write(f"{amount} {selected_currency} is equivalent to {equivalent_in_rands:.2f
```

6.0.0.2.4 2.4 Running the Streamlit App

To run the app locally:

```
streamlit run app.py
```

This will open the app in your browser, where you can test the currency conversion functionality.

6.0.0.3 3. Leveraging ChatGPT for Assistance

Building a Streamlit app can be a complex task, especially if you're new to web development or Streamlit. Here's how ChatGPT can assist you:

1. **Guidance on Streamlit Components:**
   ChatGPT can provide you with explanations, examples, and best practices for using various Streamlit components like buttons, sliders, and charts.
2. **Debugging and Troubleshooting:**
   If you encounter errors or bugs while building your app, ChatGPT can help you troubleshoot and debug your code.
3. **Enhancing the App:**
   ChatGPT can suggest features to enhance your app, such as adding additional functionalities like a currency exchange history chart, or customizing the app's appearance.
4. **Learning More About APIs:**
   If you want to expand your app to include more features, such as fetching real-time data from a currency exchange API, ChatGPT can guide you through the process of integrating external APIs into your app.

6.0.1 Conclusion

This stretch goal allows you to take full advantage of the data you've scraped and automated through GitHub Actions. By building a Streamlit app, you create a practical tool that can convert currencies to South African Rands, using the most recent exchange rates available.

Not only does this task solidify your understanding of web scraping, automation, and data handling, but it also introduces you to web app development with Streamlit. And remember, ChatGPT is here to support you throughout the process, offering guidance, solutions, and enhancements to make your app even better. Good luck, and enjoy the creative process of building your currency converter!

References