**Introduction to Web Scraping**

**Author:** Apurva Sachin Kulkarni

**Date:** 14th August, 2025

**Introduction**

In today’s data-driven world, the ability to automate the collection and processing of online information has become an invaluable skill for developers, analysts, and researchers alike. This document explores the fundamentals of web scraping and browser automation, focusing on the use of Selenium—a versatile and widely adopted tool for interacting with dynamic, JavaScript-heavy websites. It also provides practical insights into how Selenium works with WebDrivers, and demonstrates these concepts through a complete Python-based example that extracts structured data from Wikipedia. By combining clear explanations, best practices, and real-world examples, this guide equips readers with both the conceptual understanding and the hands-on techniques needed to implement efficient, reliable, and maintainable scraping solutions.

**1. What is Web Scraping?**

Web scraping is the process of automatically extracting information from websites. Instead of copying and pasting data manually, a program (called a scraper or bot) navigates web pages, finds the required elements, and collects their data into a structured format—like a CSV, JSON, or database. Web scraping is the automated process of collecting data from websites. Instead of reading a web page yourself and manually copying the information, you create a program. Here’s how it works step by step:

1. **Send a request to a webpage** (just like your browser does when you visit a site).
2. **Get the HTML content** of the page.
3. **Parse the HTML** to locate the specific elements (tables, text, links, images) you need.
4. **Extract the data** and store it in your desired format.

**Common tools and libraries:**

* **Python:** BeautifulSoup, Selenium, Scrapy, requests
* **JavaScript:** Puppeteer, Playwright, Cheerio
* **Other:** Browser extensions and APIs

**Why is Web Scraping Needed?**

Web scraping solves these problems by automating data collection. The internet is full of valuable information, but most of it is:

* **Unstructured** (plain HTML pages, not downloadable tables)
* **Dynamic** (changing frequently)
* **Not available via API** (websites don't always provide a way to fetch their data directly)

**Common use cases:**

* **Price comparison**: Scraping e-commerce sites to track product prices
* **Market research**: Gathering data about competitors, products, or trends
* **News aggregation**: Collecting articles and headlines automatically
* **Lead generation**: Extracting contact information from business directories

**2. What is Selenium?**

Selenium is a popular open-source framework primarily used for automating web browsers. It allows developers and testers to write scripts that interact with web applications the same way a human would—clicking buttons, filling forms, navigating pages, and verifying content.

**Key Features of Selenium:**

* **Browser Automation:** Works with Chrome, Firefox, Edge, Safari, etc.
* **Supports Multiple Languages:** You can write tests in Python, Java, C#, Ruby, JavaScript, and more.
* **Cross-Platform:** Works on Windows, macOS, and Linux.
* **Open Source:** Free to use with a large community.
* **Framework Integration:** Can be integrated with tools like TestNG, JUnit, and CI/CD pipelines.

**Components of Selenium:**

1. **Selenium WebDriver**
   * A tool to control browsers programmatically.
   * Interacts directly with the browser without needing an intermediary.
2. **Selenium IDE**
   * A browser extension for recording and replaying tests.
   * Good for beginners or quick test creation.
3. **Selenium Grid**
   * Allows running tests in parallel across multiple machines and browsers.
   * Useful for large-scale automated testing.
4. **Selenium RC (Remote Control)** – **Deprecated**
   * Older version of Selenium that required a server to interact with browsers.

**Use Cases:**

* Automated testing of web applications.
* Repetitive tasks on websites (like form submissions, scraping dynamic content).
* Regression testing to verify new code changes don’t break existing features.

**3. How is Selenium used in Web Scraping?**

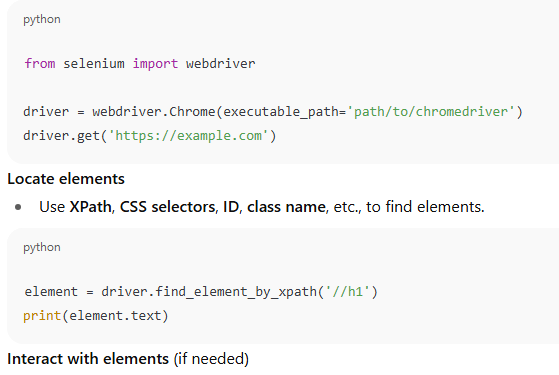
Selenium is widely used for web scraping, especially when dealing with dynamic websites—sites that load content using JavaScript or AJAX, which traditional tools like requests and BeautifulSoup can’t handle directly. Unlike static scraping, Selenium simulates a real browser, so it can interact with all elements on a page.

**Steps for Web Scraping with Selenium:**

**1. Set up Selenium and WebDriver**

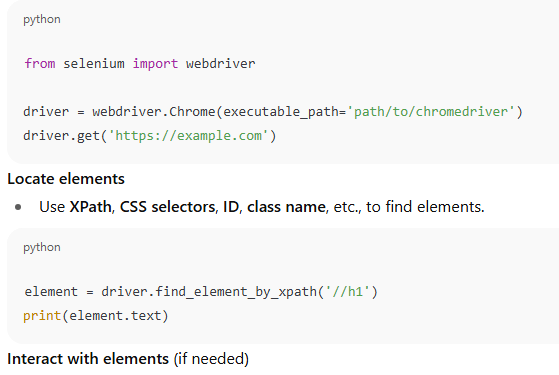
* Install Selenium (e.g., pip install selenium)
* Download a browser driver like ChromeDriver for Chrome or geckodriver for Firefox.

**2. Open the target website**



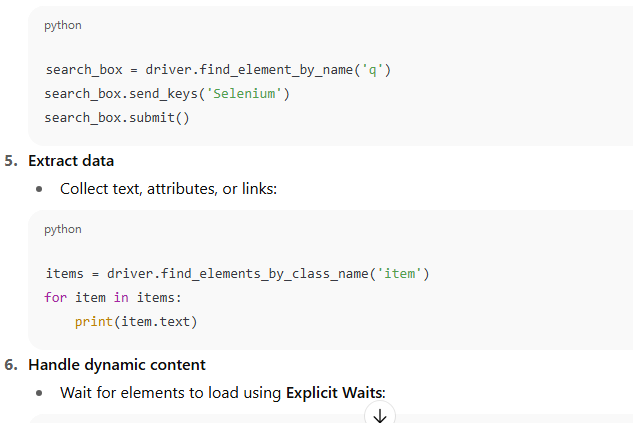
3. **Locate elements**

* Use **XPath**, **CSS selectors**, **ID**, **class name**, etc., to find elements.



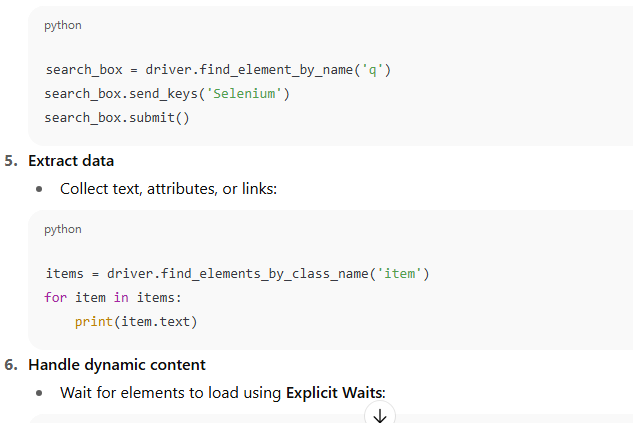
4. **Interact with elements** (if needed)

* Click buttons, fill forms, scroll, or navigate multiple pages:



5. **Extract data**

* Collect text, attributes, or links:



6. **Handle dynamic content**

* Wait for elements to load using **Explicit Waits**:



7. **Close the browser**



**Why Selenium for Web Scraping**

* Handles JavaScript-heavy websites.
* Can interact with forms, buttons, and dropdowns.
* Can scroll or paginate dynamically loaded pages.
* Simulates real user behavior, reducing the risk of being blocked (if combined with delays).

**4. What is webdriver?**

A WebDriver is a small program that acts as a bridge between Selenium and your web browser. Think of it as a personal driver for your browser—Selenium tells the WebDriver what to do, and the WebDriver makes the browser perform those actions. It’s a browser-specific executable provided by the browser vendor. It implements the WebDriver protocol (W3C standard), which Selenium uses to communicate with browsers

Examples:

* **ChromeDriver** → for Google Chrome
* **geckodriver** → for Mozilla Firefox
* **msedgedriver** → for Microsoft Edge

**Why is it used?**

**1. To control browsers automatically**

WebDriver lets Selenium perform actions just like a human user would:

* Open a webpage
* Click buttons, links, dropdowns
* Fill and submit forms
* Scroll pages or take screenshots

**2. To ensure cross-browser compatibility**

Each browser behaves slightly differently. WebDrivers are tailored to their specific browser but expose a common Selenium API, so your test scripts stay the same regardless of the browser.

**3. To handle dynamic content and full page rendering**

Since WebDriver drives a real browser, you can scrape and test JavaScript-heavy pages that wouldn’t work with just requests or BeautifulSoup.

**How Selenium uses a WebDriver**

1. Your Python (or Java, etc.) script sends commands through Selenium.
2. Selenium talks to the WebDriver executable.
3. The WebDriver talks to the browser using its internal automation protocol.
4. The browser performs the requested action and returns results to Selenium.

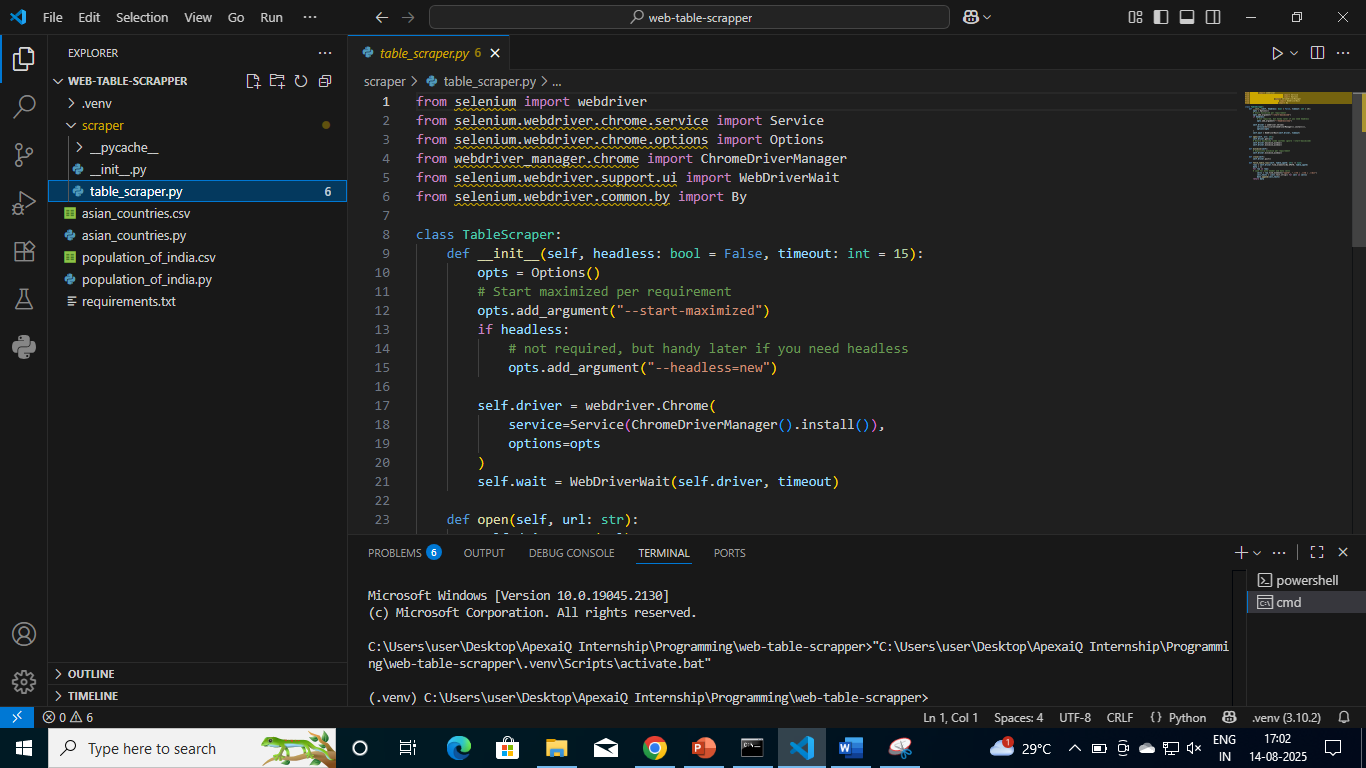
**5. Web Scraping Example – Extracting Table from a Wikipedia Page**

**1. table\_scraper.py — TableScraper Class**

This file defines a reusable class to handle Selenium browser automation for extracting table data from a webpage.

**Imports**

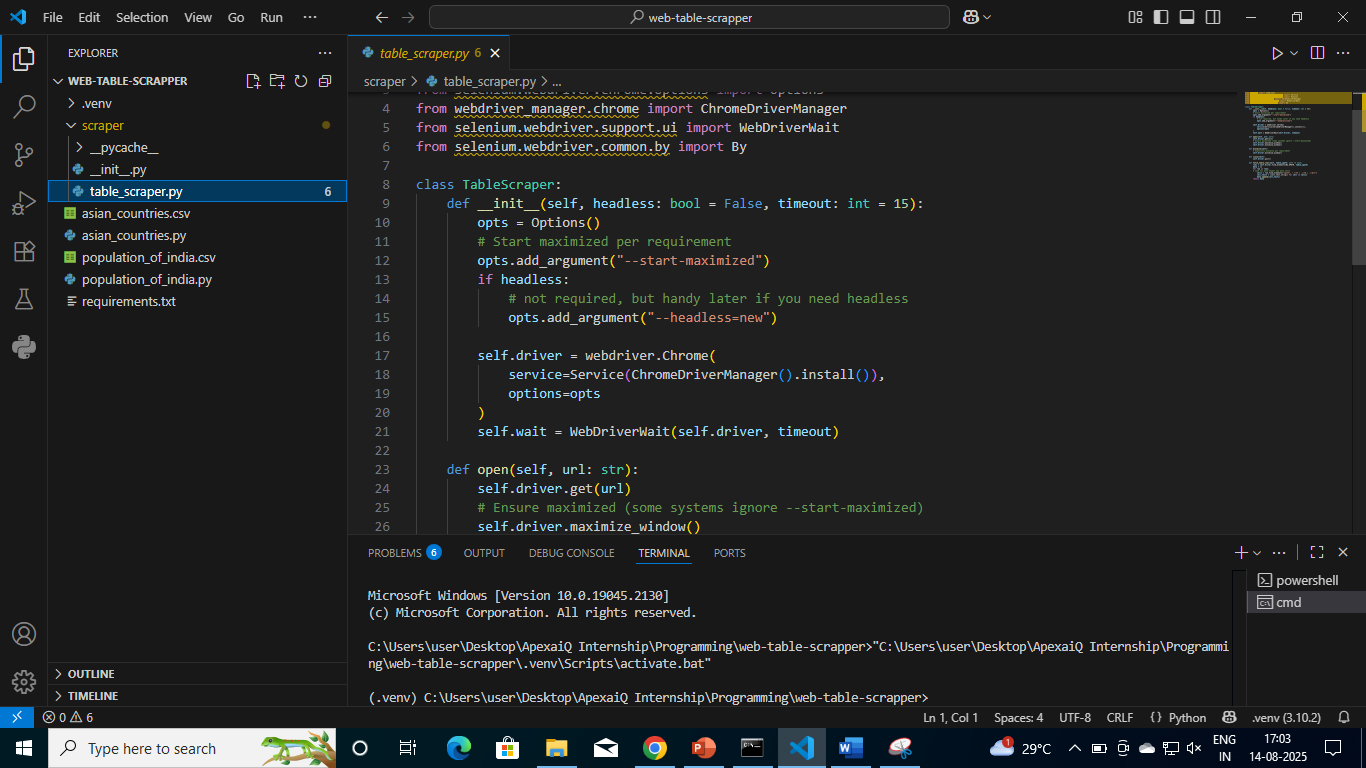
* selenium.webdriver and related modules — For controlling Chrome browser.
* webdriver\_manager.chrome — Automatically downloads and manages ChromeDriver.
* WebDriverWait — Waits until certain conditions are met before interacting with elements.
* By — Locates elements by XPath, CSS selectors, etc.



**Methods in TableScraper**

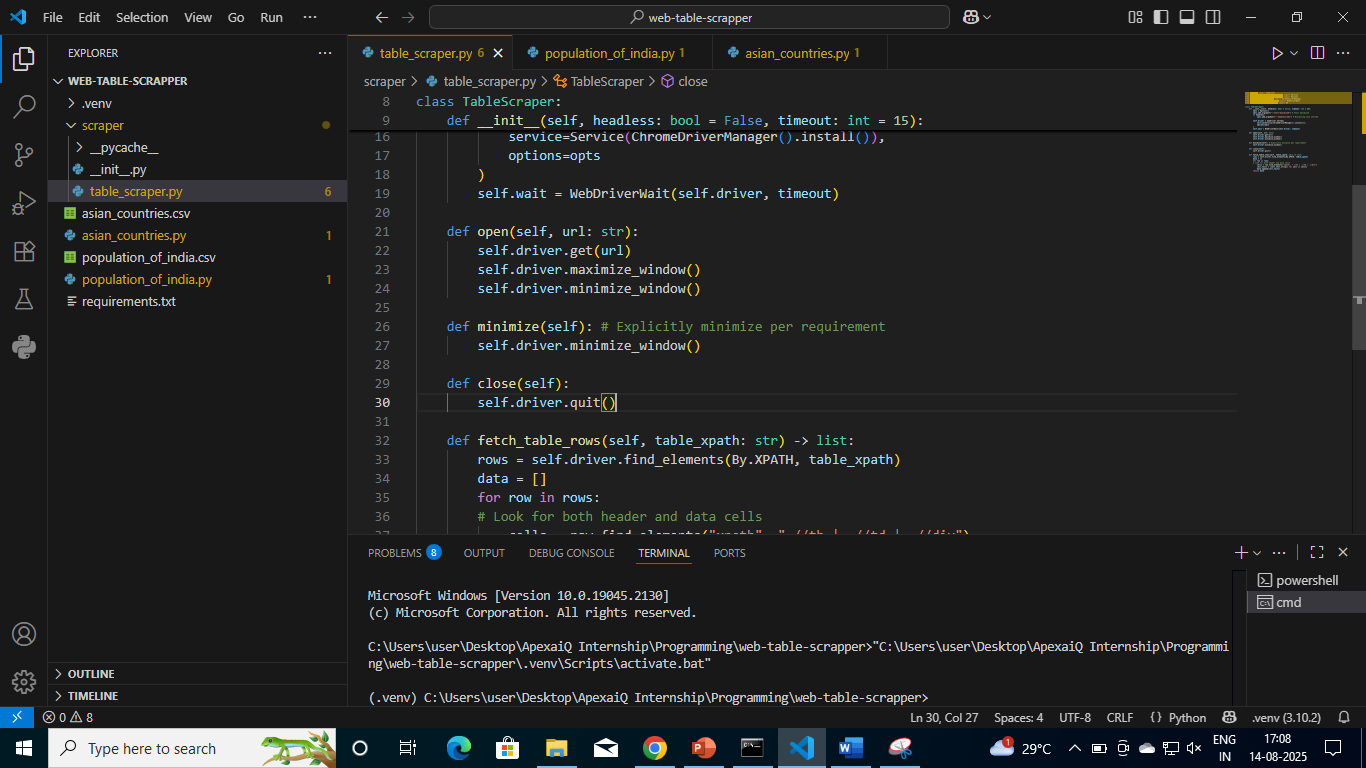
**\_\_init\_\_(self, headless=False, timeout=15)**

* **Purpose**: Initializes the Chrome WebDriver with options.
* **Key Points**:
  + --start-maximized — Starts browser in maximized mode.
  + If headless=True, runs without opening a visible browser window.
  + Uses ChromeDriverManager().install() to avoid manual driver downloads.
  + Creates a WebDriverWait object for waiting on elements.



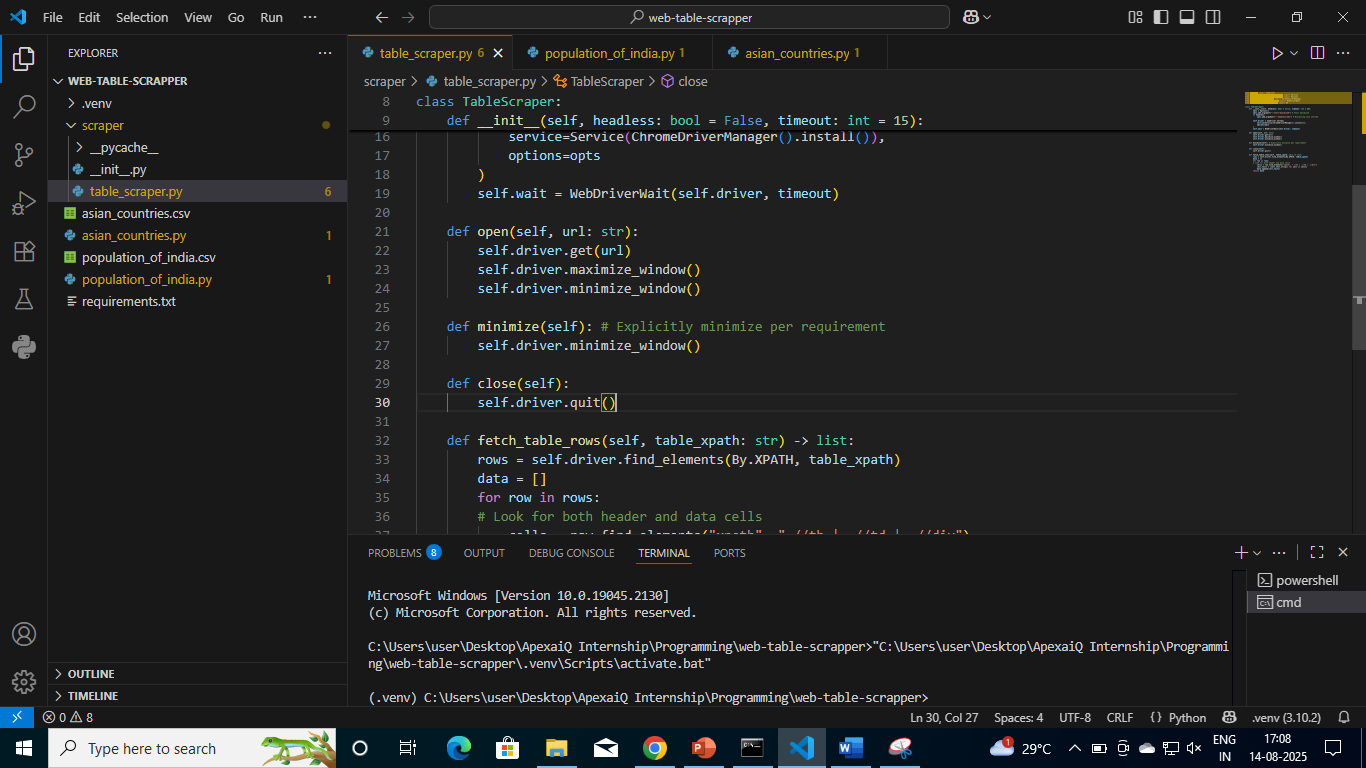
**open(self, url)**

* **Purpose**: Opens the given URL.
* **Steps**:
  + Navigates to the specified URL.
  + Maximizes the browser window (extra check).
  + Minimizes it immediately (likely to reduce screen clutter while running).



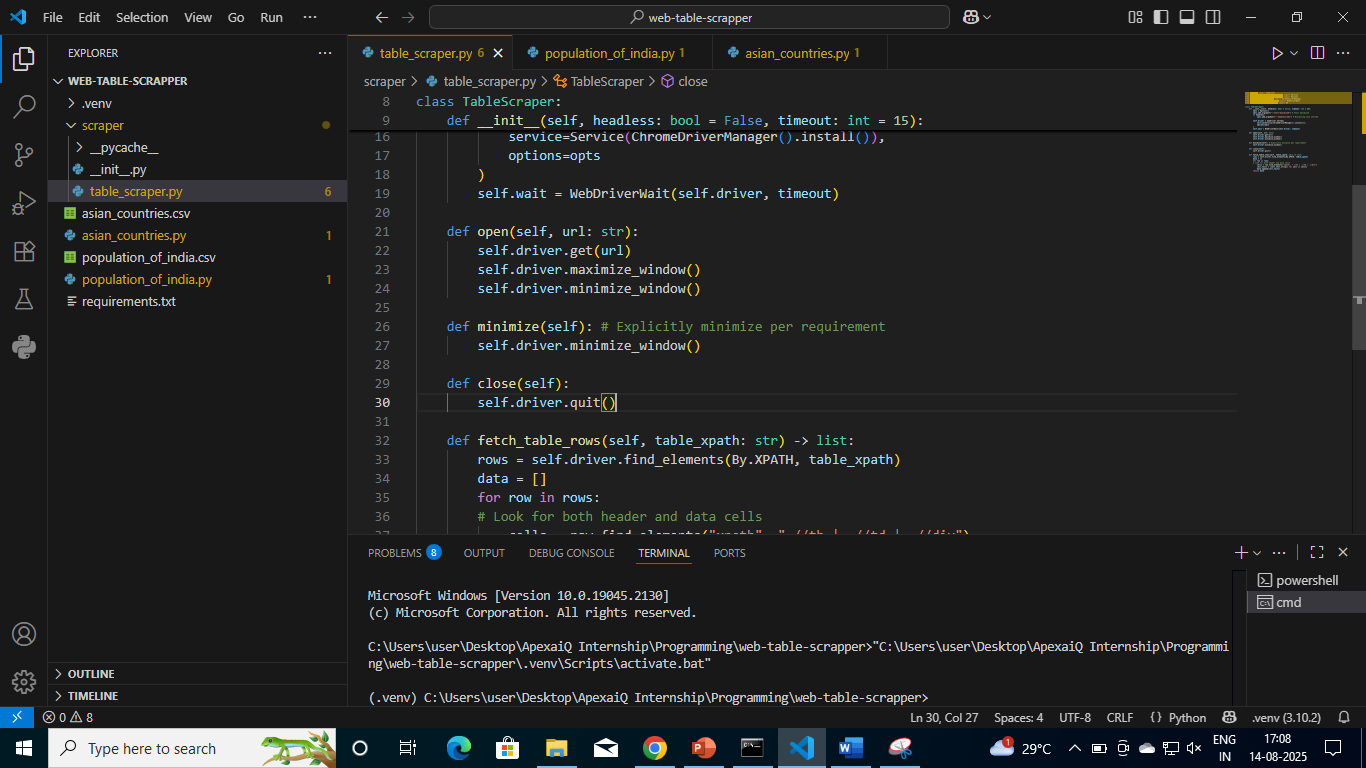
**minimize(self)**

* **Purpose**: Minimizes the browser window manually.



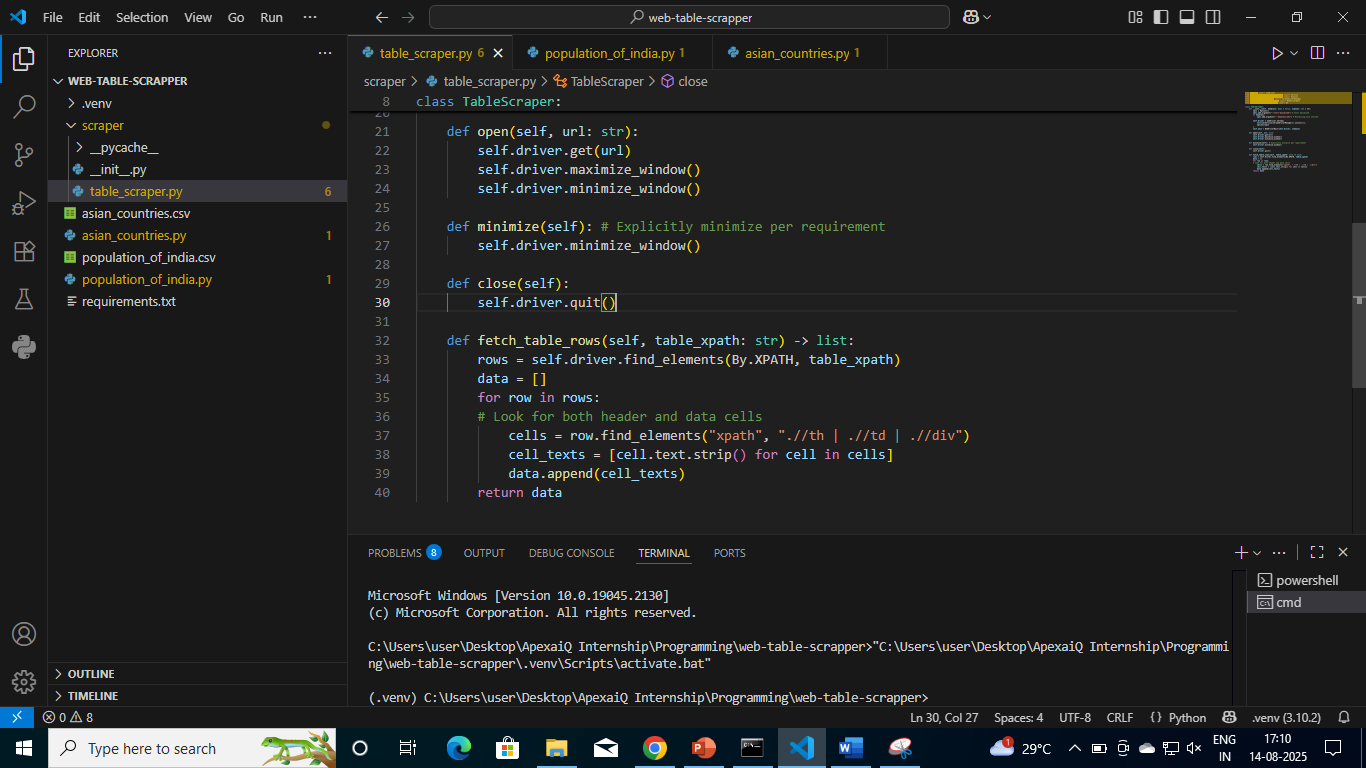
**close(self)**

* **Purpose**: Closes the browser and ends the WebDriver session.



**fetch\_table\_rows(self, table\_xpath)**

* **Purpose**: Extracts data from a table on the webpage.
* **Steps**:
  + Finds all rows using table\_xpath.
  + For each row, finds all cells (th, td, or div).
  + Extracts and strips the text content.
  + Returns a list of lists (each inner list is one row).

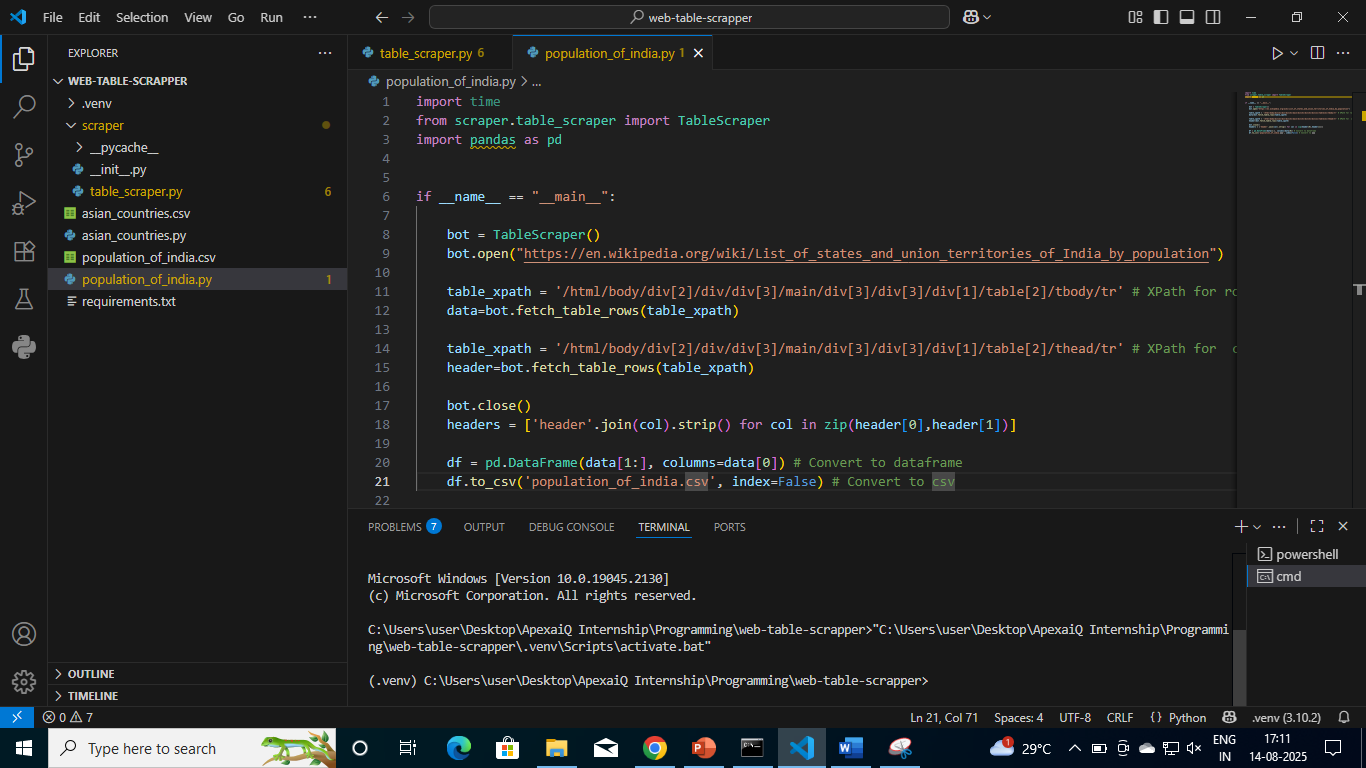


**2. population\_of\_india.py — Script to Scrape Wikipedia Data**

This is the **main script** that uses the TableScraper class to scrape population data of Indian states and union territories from Wikipedia.

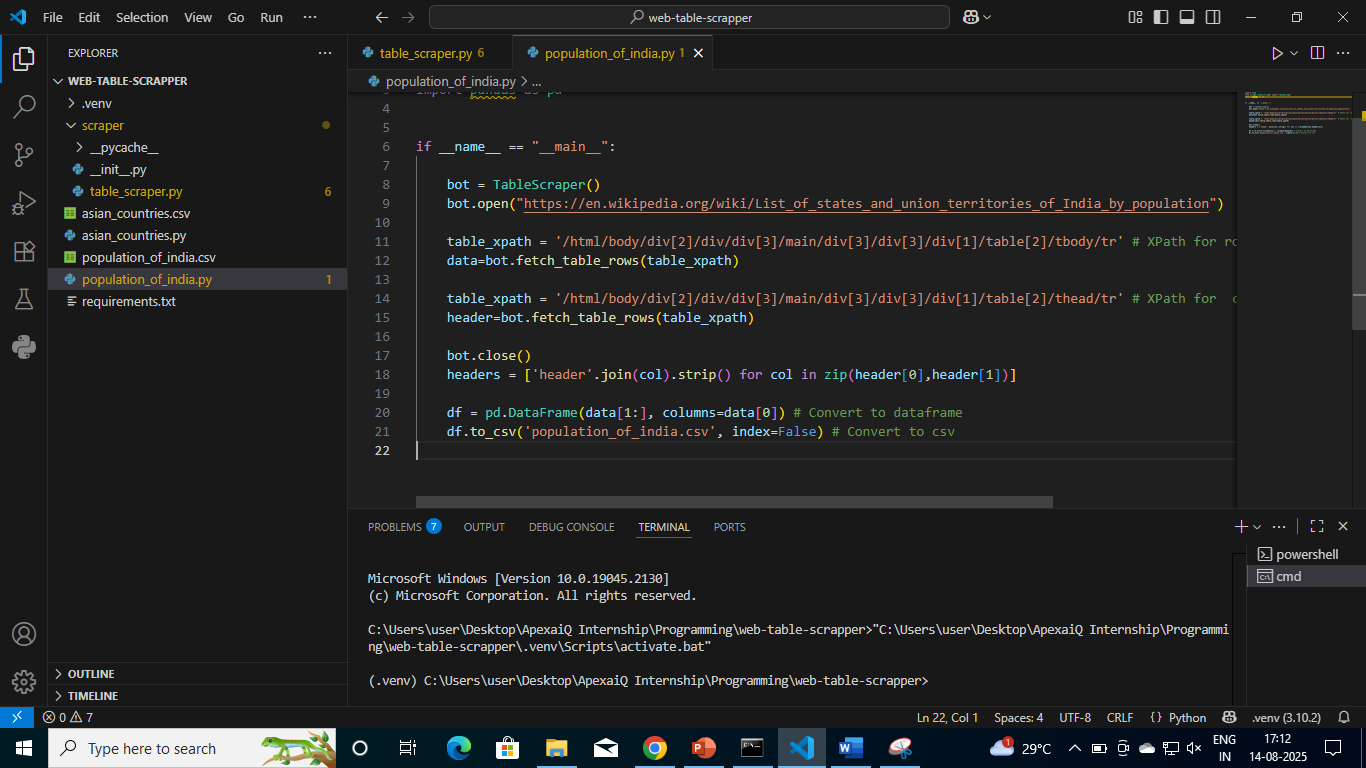
**Imports**

* time — Not actively used in the provided script.
* TableScraper — Imported from your scraper package.
* pandas — Used to store and export data as CSV.



**Flow in population\_of\_india.py**

1. **Create Bot Instance**
2. bot = TableScraper()
   * Starts Chrome WebDriver.
3. **Open Wikipedia Page**
4. bot.open("https://en.wikipedia.org/wiki/List\_of\_states\_and\_union\_territories\_of\_India\_by\_population")
5. **Scrape Table Body Data**
6. table\_xpath = '/html/body/.../table[2]/tbody/tr'
7. data = bot.fetch\_table\_rows(table\_xpath)
   * Extracts each row of the table body.
8. **Scrape Table Header Data**
9. table\_xpath = '/html/body/.../table[2]/thead/tr'
10. header = bot.fetch\_table\_rows(table\_xpath)
11. **Close Browser**
12. bot.close()
13. **Prepare Headers** *(Note: Your current code constructs headers but doesn’t use them — instead, it uses data[0] as column names)*
14. headers = ['header'.join(col).strip() for col in zip(header[0], header[1])]
15. **Create DataFrame & Save CSV**
16. df = pd.DataFrame(data[1:], columns=data[0])
17. df.to\_csv('population\_of\_india.csv', index=False)
    * Converts scraped data into a Pandas DataFrame.
    * Saves it as population\_of\_india.csv.



**Execution Flow Summary**

1. Initialize browser using TableScraper.
2. Open Wikipedia page for **List of states and union territories of India by population**.
3. Extract table **body rows**.
4. Extract table **headers**.
5. Close the browser.
6. Store scraped data into a DataFrame.
7. Export to CSV for later use.

**Conclusion**

Web scraping, when done responsibly and ethically, opens the door to vast possibilities—transforming unstructured online content into meaningful, actionable data. Selenium stands out as a powerful choice for handling complex, interactive websites where traditional scraping methods fall short. Through the example of extracting population data from Wikipedia, this document has illustrated how to set up a browser automation workflow, navigate pages, locate elements, and store results for further analysis. The techniques and best practices discussed here form a strong foundation for tackling more advanced automation tasks, integrating scraping into larger data pipelines, and ensuring code quality and maintainability. With these skills, developers can bridge the gap between the vast information on the web and the structured datasets needed for research, business intelligence, and innovation.

\*\*\*