**Web Scraping - Example**

**Author:** Apurva Sachin Kulkarni

**Date:** 18th August, 2025

**Introduction**

This project demonstrates an object-oriented approach to web scraping using Selenium. The scraper is designed to extract structured product data from the Troemner website, handling challenges such as infinite scrolling and inconsistent data formats. By encapsulating the scraping logic inside a class, the code remains modular, reusable, and easy to maintain. It leverages tools like webdriver\_manager for automatic driver setup, pandas for efficient data storage, and re for parsing model numbers through flexible regular expressions. The project illustrates how browser automation can be combined with data processing techniques to create a reliable and extensible scraping solution.

**Code Overview:**

The code is an object-oriented web scraper built using Selenium to extract product data from the Troemner website. It:

1. Opens the product listing page.
2. Scrolls repeatedly to load all products (handling infinite scrolling).
3. Extracts product details such as name, URL, model number, description, and cost.
4. Stores the data in a CSV file.

The code uses:

* **Selenium WebDriver** for browser automation.
* **webdriver\_manager** for automatic ChromeDriver management.
* **pandas** for saving structured data as a CSV.
* **re (regular expressions)** for parsing model numbers from URLs, IDs, or text.

**TableScraper Class**

The scraper is encapsulated inside a class to keep the code organized and reusable.

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

* **Purpose:** Initializes the Chrome WebDriver with optional headless mode.
* **Steps:**
  + Creates Options() to set browser preferences.
  + Adds "--start-maximized" to open Chrome in a full window.
  + Adds "--headless=new" if headless=True, so it runs without showing a GUI.
  + Uses **ChromeDriverManager** to automatically install the correct ChromeDriver version.
  + Initializes a **WebDriverWait** object with the given timeout (15s default) for waiting on page elements.
* **Key benefit:** Automates browser setup without manually downloading drivers.

**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.

**load\_all\_products(self)**

* **Purpose:** Continuously scrolls down until all products are loaded (handles infinite scrolling).
* **Steps:**
  1. Keeps track of prev\_count (previous number of product elements).
  2. Scrolls to the bottom of the page using JavaScript:
  3. Wait 3 seconds to let new products load via AJAX.
  4. If no new products appear (new\_count == prev\_count), break the loop.
  5. Otherwise, print progress: "Loaded X products...".

**fetch\_data(self)**

* **Purpose:** Extracts product details from all loaded product cards.
* **Steps:**
  1. Initialize an empty list data.
  2. Get all product <li> elements using the same XPath as load\_all\_products().
  3. For **each product card:**
     + **Product Name & URL:**
  4. If missing, defaults to empty strings.

**Model Number (part number):**  
Extracted using **regular expressions** in three fallback stages:

1. From URL: /p/(\d+)
2. From <li> element id: (\d+)$
3. From text in product name: (\d{6,}) (long number pattern)

Return data list.

**Key Observations**

* **OOP Approach:** Encapsulates scraper logic into methods → makes it easy to reuse or extend.
* **Error Handling:** Uses try-except to avoid crashes when elements are missing.
* **Regex Logic:** Clever fallback to extract product model numbers from multiple sources.
* **Potential Weakness:** The hardcoded XPath for cost may fail if product rows have different IDs.
* **Infinite Scroll:** Automatically handled by counting loaded items.

**Conclusion**

The implemented scraper successfully automates the extraction of detailed product information, overcoming dynamic page loading and missing element issues. Its object-oriented structure makes the codebase organized, while the use of fallback regex logic ensures robust data parsing. Although the hardcoded XPath for cost elements could limit adaptability if the website structure changes, the overall design demonstrates how Selenium, combined with Python libraries, can produce scalable and efficient web scrapers. This project serves as a practical template for building similar data collection tools for e-commerce platforms and other dynamic websites.

\*\*\*