Intro

Web scraping involves two broad categories of web pages—pages that need rendering and pages that do not need rendering. Web scraping the pages that need rendering is often called web scraping dynamic web pages. This is where Selenium shines.

Selenium is one of the oldest and perhaps the most widely known tool. Selenium development began as early as 2004. This began as a tool for functional testing and the potential of web scraping was soon realized.

The biggest reason for Selenium's is that it supports writing scripts in multiple programming languages, including Python. It means that you can write Python code to mimic human behavior. The Python script will open the browser, visits web pages, enter text, click buttons, and copy text. This can be combined with other features in Python to save data in simple CSV or complex databases.

In this guide on how to web scrape with Selenium, we will be using Python 3. The code should work with any version of Python above 3.6.

Selenium Package

Firstly, to download the Selenium package, execute the pip command in your terminal:

`pip install selenium`

In Selenium, a **WebDriver** (or driver) acts as the essential bridge between your script and the web browser. Because browsers like Chrome, Firefox, and Edge are built differently, Selenium cannot talk to them directly; it requires a driver to translate your Python commands into actions the browser understands.

Why Drivers Are Necessary

- **Translation**: The driver receives requests from your code and converts them into instructions for the browser's native engine.
- **Browser Isolation**: The driver launches a fresh, "clean" instance of the browser controlled by automated software.
- **Version Compatibility**: You must use a driver version that specifically matches the version of the browser installed on your machine.

How to Use Drivers in Selenium 4.0

In the previous version (Selenium 3), you passed the driver's file path directly into the browser constructor. In **Selenium 4.0**, you must use the **Service object** to manage the driver executable.

1. The Automatic Way (Recommended)

The modern standard is to use the webdriver-manager library. This removes the need to manually download .exe files or check your Chrome version.

Python

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

# Automatically downloads and links the correct driver version
service = Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service)
```

2. The Manual Way

If you prefer to keep the driver file in a specific folder (as mentioned in your original guide), you must now point to it using the Service class.

Python

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service

# You must wrap the path in a Service object for Selenium 4.0
service = Service(executable_path='C:/WebDrivers/chromedriver.exe')
driver = webdriver.Chrome(service=service)
```

Driver Setup Comparison

Task	Selenium 3 (Old)	Selenium 4 (New)
Path Setting	Passed as executable_path string	Passed via a Service object
Download	Manual download from vendor site	Automated via webdriver-manager
Browser Version	Must be checked and matched manually	Handled automatically by the manager

Data Extraction with Selenium - Locating Elements

As discussed, Selenium 4 has replaced the individual find_element_by_* methods with a unified find_element() method that uses the By class. Below are the explanations for each locator strategy, updated to the modern syntax.

1. By ID

- **Explanation**: Finds an element by its unique id attribute. This is usually the fastest and most reliable way to locate an element because IDs are meant to be unique on a page.
- **Example**:
- Python
  ```
  # Target: <h1 id="greatID">
  element = driver.find_element(By.ID, "greatID")
  ```

2. By NAME

- **Explanation**: Finds an element by its name attribute. This is frequently used for form elements like <input> or <textarea>.
- **Example**:
- Python
  ```
  # Target: <input name="search_query">
  element = driver.find_element(By.NAME, "search_query")
  ```

3. By XPATH (Recommended)

- **Explanation**: Finds elements using XPath syntax to navigate the DOM (Document Object Model). It allows for complex queries, such as moving from a child element back to a parent or finding elements based on partial text.
- **Key Symbols**:
  - /: Selects a direct child.
  - //: Selects any descendant regardless of depth.
  - []: Used for specific attributes or functions, like contains().
- **Example**:
- Python
  ```
  # Finds a link that contains the text "Humor"
  element = driver.find_element(By.XPATH, '//a[contains(text(),"Humor")]')
  ```
- **/ : Select child element. /html/body/div/p[1]** will find the first p which is in a div tag, which in turn is a child of body element. This means that if a **<div><p>something</p></div>** will not be selected.

- **//: Select all descendant element** from the current element. //p will find all p elements, whether they are in a div or not.
- **[@attributename='value']:** It looks for a specific attribute with a specific value. This can also be used as **[@attributename]** to search for the presence of this attribute, irrespective of the value.
- XPath functions such as **contains()** can be used for a partial match

4. By CSS SELECTOR (Recommended)

- **Explanation**: Finds elements using CSS selector syntax. It is often faster than XPath and is the standard way developers style elements.
- **Example**:
- Python
  ```python
  # Target: All books inside a product pod
  elements = driver.find_elements(By.CSS_SELECTOR, ".product_pod h3 a")
  ```

5. By LINK TEXT

- **Explanation**: Specifically targets <a> (anchor) elements by matching their visible text exactly.
- **Example**:
- Python
  ```python
  # Target: <a href="...">Log In</a>
  element = driver.find_element(By.LINK_TEXT, "Log In")
  ```

6. By PARTIAL LINK TEXT

- **Explanation**: Similar to Link Text, but it finds <a> elements by matching only a portion of their visible text.
- **Example**:
- Python
  ```python
  # Target: <a href="...">Click here to see more details</a>
  element = driver.find_element(By.PARTIAL_LINK_TEXT, "see more details")
  ```

7. By TAG NAME

- **Explanation**: Finds elements based on their HTML tag (e.g., h1, div, p, button).
- **Example**:
- Python
  ```python
  # Target: The first <h1> tag on the page
  element = driver.find_element(By.TAG_NAME, "h1")
  ```

8. By CLASS NAME

- **Explanation**: Finds elements by their class attribute. Note that if an element has multiple classes (e.g., class="btn btn-primary"), you should only use one of them here or switch to a CSS Selector.
- **Example**:
- Python
  ```
  # Target: <h1 class="someclass">
  element = driver.find_element(By.CLASS_NAME, "someclass")
  ```

Comparison Table for Quick Reference

Strategy	When to Use	Reliability
ID	When the element has a unique ID.	Very High
CSS Selector	For general styling classes or nested elements.	High
XPath	For complex navigation (e.g., finding parents).	High
Name	For form inputs and text fields.	Medium
Link Text	For navigation menus and specific links.	Medium
Tag Name	To grab all elements of a certain type (like all <a>).	Low

WebElement

In Selenium, a **WebElement** is an object that represents a specific HTML element on a webpage (like a button, a text box, or a heading). Once you locate an element using find_element(), you use WebElement methods to interact with it.

1. element.text
- **What it does:** Extracts the visible text contained within an element (and its sub-elements).

- **Use Case:** Reading headlines, prices, or descriptions.
- **Example:**
- Python

```python
heading = driver.find_element(By.TAG_NAME, "h1")
print(heading.text)  # Output: "All products"
```

2. element.click()

- **What it does:** Simulates a mouse click on the element.
- **Use Case:** Pressing buttons, following links, or selecting checkboxes.
- **Example:**
- Python

```python
login_button = driver.find_element(By.ID, "login-btn")
login_button.click()
```

3. element.get_attribute('name')

- **What it does:** Retrieves the value of a specific HTML attribute (like href, src, class, or id).
- **Use Case:** Scraping image URLs from src or links from href.
- **Example:**
- Python

```python
book_link = driver.find_element(By.CSS_SELECTOR, ".product_pod h3 a")
url = book_link.get_attribute("href")
print(url)  # Output: "catalogue/a-light-in-the-attic_1000/index.html"
```

4. element.send_keys('text')

- **What it does:** Simulates typing into an input field or a textarea.
- **Use Case:** Filling out search bars, login forms, or passwords.
- **Example:**
- Python

```python
search_bar = driver.find_element(By.NAME, "search")
search_bar.send_keys("Science Fiction")
```

Summary Table

Method	Type	Purpose	Source
.text	Attribute	Get visible text	

.click()	Method	Interact with clickable items	
.get_attribute()	Method	Get hidden data like URLs or classes	
.send_keys()	Method	Input text or passwords into forms	

Waiting for Elements to Appear

In modern web scraping, timing is everything. Dynamic websites often use JavaScript to load content after the initial page load, meaning your script might try to grab an element before it even exists.

While time.sleep() is a tempting "quick fix," it is inefficient because it forces the script to wait for a fixed amount of time regardless of whether the element is ready or not. Instead, Selenium 4.0 offers two smarter ways to handle this.

1. Implicit Waits
An **Implicit Wait** is a global "safety net" you set once at the start of your script. It tells the WebDriver to poll the DOM for a specific amount of time before throwing an error if an element isn't found.

- **How it works**: If you set it to 10 seconds and the element appears in 2 seconds, Selenium moves on immediately.
- **Code Example**:

```Python
# Set it once after driver initialization
driver.implicitly_wait(10) # 10 seconds
```

- **Best Use Case**: Use this when you want a general buffer for slow-loading pages.

---

2. Explicit Waits

Explicit Waits provide much finer control. Instead of a global setting, you define a specific condition for a specific element. This is the "gold standard" for professional scraping.

- **How it works**: You define a WebDriverWait object and tell it to wait until a specific expected_condition (EC) is met.
- **Code Example (Selenium 4.0 Style)**:

```Python
```

```python
from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions as EC


try:
    # Wait up to 10 seconds for the product containers to appear
    books = WebDriverWait(driver, 10).until(
        EC.presence_of_all_elements_located((By.CSS_SELECTOR, '.product_pod'))
    )
    print(f"Found {len(books)} books!")
except Exception as e:
    print("Timed out waiting for elements.")
finally:
    driver.quit()
```

Common Expected Conditions (EC)

Selenium 4.0 supports a variety of events you can wait for:

Condition	Purpose
presence_of_element_located	The element exists in the DOM.
visibility_of_element_located	The element is visible on the screen.
element_to_be_clickable	The element can be clicked (not grayed out).

url_contains	The URL has changed to a specific string.
text_to_be_present_in_element	A specific word has appeared inside an element.

Key Differences at a Glance

Feature	Implicit Wait	Explicit Wait
Scope	Global (applies to all elements).	Local (applies to one specific call).
Complexity	Very easy to implement.	Requires more code and imports.
Control	Low; only waits for "presence."	High; can wait for visibility, clickability, etc.
Performance	Can slow down tests if many elements are missing.	Optimized for speed and specific logic.

Pro Tip: Avoid mixing Implicit and Explicit waits in the same script. Doing so can cause unpredictable wait times where the driver waits much longer than intended.

Scenario	time.sleep(10)	WebDriverWait(driver, 10)
Element appears in 2s	Script sits idle for the full 10 seconds.	Script continues immediately after 2 seconds.

Element appears in 12s	Script tries to run at 10s and fails (crash).	Script waits until 10s and then throws a TimeoutException.
Efficiency	Very Low (wastes time).	Very High (dynamic and fast).