Selenium Using Python

Introduction

Selenium is a robust and open-source automated testing framework designed for web applications. It excels in providing seamless compatibility across different web browsers and operating systems. Unlike HP's Quick Test Pro (QTP), now known as UFT, Selenium is tailored exclusively for automating web-based applications. Proficient utilization of Selenium yields results commonly recognized as Selenium testing.

Selenium is primarily designed for testing web applications, and it is not typically used for testing desktop or mobile applications. Web applications are software programs hosted on remote servers, accessible through web browsers over the internet, making Selenium a valuable tool for automating and conducting tests within this specific context.

What is Selenium WebDriver?

Selenium WebDriver is an internet framework that permits you to carry out useful net-based application testing to ensure that it works as anticipated. These APIs are an open source series that supports the automation of all of the main browsers available on the market, the usage of WebDriver. WebDriver is an API and protocol that defines a language-impartial interface for handling internet browser behavior. Each browser is supported by a selected WebDriver implementation, known as a motive force. The motive force is the element that guarantees that it delegates responsibilities to the browser and handles the communique to and from Selenium and the browser.

Selenium WebDriver is used to perform functional web-based application testing to ensure that it performs as expected. It mainly supports browsers like Firefox, Chrome, Safari and Internet Explorer. It also allows you to run cross-browser tests.

It was founded by Simon Stewart, a ThoughtWorks' Consultant in Australia, in 2006. Selenium WebDriver was the first cross-platform testing framework that could control the browser at the OS level. Selenium WebDriver is a successor to Selenium RC. Selenium WebDriver accepts commands (sent in Selenium or via a Client API) and sends them to the browser.

Installing Selenium in Python:

- Use pip to install the selenium package. Python 3 has pip available in the standard library. Here is the code
- Proceed with installing the Selenium library using the command prompt or terminal.

    - `pip install selenium or pip3 install selenium`

- To check version of selenium in cmd

    - `pip show selenium`

Installing Drivers in Python:

- Additionally, you need to download the appropriate WebDriver for the browser you want to automate, such as ChromeDriver for Chrome.
- First you have to see which version that your chrome has installed in your system. Then click on this link https://chromedriver.chromium.org/downloads.
- Note: Make sure that you have the same version of chrome driver installed.
- After downloading the WebDriver, we need to configure it in our development environment while also specifying the WebDriver executable path so that Selenium can locate and utilize it.
- You can achieve this by either setting the system path or providing the WebDriver path directly in our code.

What are Browser Elements?

- Browser elements are different components present on web pages. The most common elements you will notice while browsing is:
- Once the browser is open, you can navigate through different web pages using Selenium.

    - Text boxes
    - CTA buttons
    - Images
    - Hyperlinks
    - Radio buttons/checkboxes
    - Text area/error messages
    - Dropdown box/list box/combo box
    - Web table/HTML table
    - Frame, etc.

HTML Basic:

- HTML is the markup language used to structure web pages.
- Selenium provides methods to find elements based on their attributes like id, class name, tag name, etc.
- When working with Selenium for web automation, you often interact with HTML elements on web pages.Common HTML tags you'll frequently use with Selenium:

    - **<a> (Anchor)**: Useful for extracting hyperlinks, which can lead to data sources.

    - **<p> (Paragraph)**: Commonly used for text content, including textual data that you may want to scrape.

- **<h1>, <h2>, <h3>,...<h6> (Headings)**: Often used for titles and section headings that provide structure to web pages and data.
- **<ul>, <ol> (Unordered and Ordered Lists)**: Useful for extracting structured lists of data.
- ** (List Item):** Elements within lists, which can contain valuable data.
- **<table>:** Used for presenting tabular data. Essential for scraping structured data tables.
- **<tr> (Table Row)**: Rows within tables, containing data points.
- **<td> (Table Data)**: Cells within tables, where data is presented.
- ** :** Frequently used as a container for various content, including data.
- **<form>:** Used for web forms, which may be a source of data.
- **<input>:** Found within forms and can hold data that can be extracted.
- **<textarea>:** Often used for multi-line text input, where textual data might be present.
- **<label>:** Linked to form elements, providing descriptions or labels for input fields.
- **<img>:** Useful for extracting image data or checking image attributes.
- **<iframe>:** May contain data or content embedded from other sources.
- **<script>**: Contains JavaScript code, which sometimes loads or manipulates data.
- **<input>**: Used for various input fields like text boxes, radio buttons, and checkboxes.
- **<select>**: Used for dropdown lists, which may contain selectable data.

- Once an element is located, you can perform actions like clicking, entering text, or extracting data.You can interact with elements using methods like click, send_keys, and text extraction methods.

CSS Basic:

- In Selenium, you can use CSS selectors to locate and interact with elements on a web page. When working with web scraping or data extraction for data science purposes, some common CSS selectors can be useful for targeting specific elements.

- Common CSS selectors you might find useful in Selenium for data science:

- **Tag Selector:** Select elements based on their HTML tag type.

  Example: a to select all anchor (link) elements.

- **Class Selector:** Select elements based on their class attribute.

  Example: .classname to select elements with a specific class name.

- **ID Selector:** Select elements based on their unique ID attribute.

  Example: #elementId to select an element with a specific ID.

- **Attribute Selector:** Select elements based on attributes other than class and ID.

  Example: [attribute=value] to select elements with a specific attribute and value.

- **Descendant Selector:** Select elements that are descendants of another element.

  Example: parentElement childElement to select child elements within a parent.

- **Child Selector:** Select direct child elements of a parent element.

  Example: parentElement > childElement to select immediate child elements.

- **Adjacent Sibling Selector:** Select an element that is immediately preceded by a specified element.

  Example: element + sibling to select an adjacent sibling.

- **General Sibling Selector:** Select elements that share the same parent and have the specified element as a sibling.

  Example: element ~ sibling to select general siblings.

- **Nth-child Selector:** Select the nth child of a parent element.

  Example: parentElement:nth-child(n) to select the nth child.

- **Pseudo-class Selectors:** Select elements based on their state or position.

  Example: :hover to select elements when hovered over.

- **Substring Selector:** To select elements based on a substring of their attribute value.

  Example : To select all elements with the attribute "data-id" that contain the substring "123", you can use the following CSS selector: [data-id*="123"]

Locating and Fetching Elements

- Locating and fetching elements in Selenium is a vital ability for internet scraping and statistics extraction in information science tasks. To effectively acquire information, you may want to become aware of and interact with specific HTML elements on an internet web page.

  Import Selenium:

- First, you need to import the Selenium library in your Python script.

  - `import selenium`
  - `from selenium import webdriver`

  Launch a Web Browser:

- Initialize a web driver for your chosen browser (e.gChrome, Firefox, Edge).

  - `driver = webdriver.Chrome() # Or other browser options.`

  Navigate to a Web Page:

- Use the driver to navigate to the web page you want to scrape.

  - `driver.get("https://example.com")`

Locating Elements:

- You can locate elements using various methods provided by Selenium.
- The find_element method in Selenium is a fundamental and powerful feature used to locate and interact with specific web elements on a web page. It is commonly used for web automation, web testing, and web scraping

  - find_element
  - find_elements

- The attributes available for the By class are used to locate elements on a page. These are the attributes available for By class:

  - **ID**: Finding elements by id property
  - **NAME**: Finding elements by id property.
  - **XPATH**: Finding elements by xml property
  - **LINK_TEXT**: Only fetches links using exact text match
  - **PARTIAL_LINK_TEXT**: Only fetches link using substring match
  - **TAG_NAME**: Fetches all the elements with the specific html tag

- **CLASS_NAME**: Fetches all the elements by matching class attributes
- **CSS_SELECTOR:** Fetches using the css properties

These are the attributes available for By class:

```
ID = “id”
NAME = “name”
XPATH = “xpath”
LINK_TEXT=”link text”
PARTIAL_LINK_TEXT = “partial link text”
TAG_NAME = “tag name”
CLASS_NAME = “class name”
CSS_SELECTOR = “css selector”
```

- The ‘By’ class is used to specify which characteristic is used to discover factors on a page. These are the diverse approaches the attributes are used to discover elements on a page.

```
find_element(By.ID, "id")
find_element(By.NAME, "name")
find_element(By.XPATH, "xpath")
find_element(By.LINK_TEXT, "link text")
find_element(By.PARTIAL_LINK_TEXT, "partial link text")
find_element(By.TAG_NAME, "tag name")
find_element(By.CLASS_NAME, "class name")
find_element(By.CSS_SELECTOR, "css selector")
```

- **Locating by Id:**
    - This method is simplest whilst you are already aware of the "identity" attribute associated with a particular detail. When using this approach, the first detail possessing an identical "id" attribute might be retrieved. In the event that no detail suits the required "id" attribute, a NoSuchElementException could be raised.
- **Locating by Name:**
    - This approach is first-class acceptable whilst you possess information of the "name" characteristic related to a specific element. By using this approach, the preliminary detail featuring a matching "name" attribute will be obtained. In instances where no detail shares a corresponding "name" attribute, the operation will result in a NoSuchElementException being raised.

- **Locating by XPath:**
    - XPath, a powerful language utilized in Selenium, allows targeting web application elements with precision. It proves particularly advantageous when standard identification attributes like id or name fall short. By using XPath, you can locate elements either absolutely (although it's not recommended) or more commonly, relative to nearby elements that possess id or name attributes. This relative approach enhances reliability and safeguards your tests against structural changes within the web application. Moreover, XPath offers the flexibility to specify elements using alternative attributes apart from id and name, thereby making it an invaluable tool for web automation and testing.

- **Locating Hyperlinks by Link Text:**
- If you know the specific text of the link within an anchor tag, this method is appropriate. It will retrieve the first element with a matching link text. However, if there is no element with the same link text attribute, using this strategy will result in raising a NoSuchElementException.

- **Locating Elements by Tag Name:**
    - To locate a specific element based on its HTML tag name, you can use this method. It allows you to access the initial element that matches the specified tag name. If there is no element with the given tag name, an exception called NoSuchElementException will be raised.

- **Locating Elements by Class Name:**
    - When you need to locate an element based on its class name, this method is the most suitable option. It will retrieve the first element that has a matching class name attribute. In case no element matches the specified class name, this method will raise a NoSuchElementException.

- **Locating Elements by CSS Selectors:**

    - This method is the way to go when you aim to search for an element using CSS selector syntax. Using this option, you get a basic element that matches the given CSS selector. Where no element matches the specified CSS selector, the use of this method will result in a NoSuchElementException.

What is an Exception?

- An exception is an event or condition that deviates from the normal flow of a program and can disrupt its execution. Exceptions in Selenium are typically related to issues encountered during web automation, testing, or web scraping tasks.

What is an Exception Handling?

- Exception handling in Selenium is a critical aspect of writing robust and reliable automation scripts. Selenium automation scripts interact with web browsers and web applications, and various unexpected situations can occur during script execution. Exception handling is the practice of anticipating and managing these exceptions or errors to ensure that your script continues to run smoothly even in the face of unexpected issues.

  Common Exception:

- **NoSuchElementException**: Occurs when an element cannot be found on the web page.

- **TimeoutException:** Occurs when a specific operation exceeds the given time limit.

Exception handling using try and catch:

- Use try-except blocks to catch specific exceptions.

- Catch the NoSuchElementException and perform alternative actions or log an error message.

- Use the finally block for code that needs to run regardless of exceptions.

Typing into fields:

- Typing into fields is a common task in web automation.
- Typing into fields in Selenium, such as text fields or input boxes, is a common action in web automation. You can use the send_keys method to simulate keyboard input.
- Locate the numeric field using locator methods provided by Selenium (e.g., find_element_by_id, find_element_by_name, find_element_by_xpath).
- Use the send_keys() method to type into the numeric field.

Logging in with Existing ID:

- Logging in with an existing ID is a common scenario in web automation.Selenium provides capabilities to automate the login process.
- Create an instance of the WebDriver and launch the web browser (e.g., Chrome).
- Navigate to the login page of the sample website using the get() method.

- Locate the username/email and password fields using locator methods provided by Selenium.
- Use the send_keys() method to enter the login credentials.

Implicit Wait & Explicit Wait:

- The Implicit Wait in Selenium is used to tell the web driver to wait for a certain amount of time before it throws a “No Such Element Exception”. The default setting is 0. Once we set the time, the web driver will wait for the element for that time before throwing an exception.

  Implicit Waits:

- Global wait applied to WebDriver instance
- Maximum time Selenium waits for an element to appear
- Set using implicitly_wait() method
- Useful for setting a default wait time for all elements

  Explicit Wait:

- The Explicit Wait in Selenium is used to tell the Web Driver to wait for certain conditions (Expected Conditions) or maximum time exceeded before throwing an “ElementNotVisibleException” exception. It is an intelligent kind of wait, but it can be applied only for specified elements. It gives better options than implicit wait as it waits for dynamically loaded Ajax elements.
- Once we declare an explicit wait we have to use “ExpectedConditions” or we can configure how frequently we want to check the condition using Fluent Wait. These days while implementing we are using Thread.Sleep() generally it is not recommended to use
- Wait for a specific condition to be met before proceeding
- Create WebDriverWait instance with a timeout
- Apply conditions using the until() method
- More granular control over waiting process
  Example:
  Element to be **clickable**
  Element to be **visible**
  Element to have a **specific attribute value**

Implicit Wait VS Explicit Wait:

- **Implicit Wait:**
  Global wait for all elements.
  Suitable for consistent delay in element loading.

- **Explicit Wait:**
  Specific to a condition or element.
  Wait for specific conditions to be met.

Radio Buttons, Checkboxes, and Dropdowns:

- In web applications, forms play a crucial role in user interactions. They allow users to provide input, make selections, and submit data

- We'll focus on three commonly encountered form elements:

    - Radio buttons
    - Checkboxes
    - Dropdown menus

Radio Buttons:

- Radio buttons are a fundamental form element in web applications, primarily used for enabling users to select a single option from a set of mutually exclusive choices. They provide a straightforward way to gather user preferences, such as gender, payment methods, or yes/no responses. To interact with radio buttons in a web automation context using a tool like Selenium, you can locate the radio button element based on its attributes like ID, name, CSS selector, or XPath. Once the radio button is identified, the click() method is commonly used to select the desired option. This action simulates a user's click on the radio button, activating the chosen option and ensuring that only one option from the group can be selected at a time. By automating this process, testers and developers can efficiently handle radio button selections as part of their testing and web interaction scenarios.

Checkboxes:

- Checkboxes are important features of web documents, allowing users to select multiple options from the available options. Whether it's displaying preferences, applying filters, or creating multiple selections, checkboxes offer a lot of versatility. The interaction with checkboxes in Selenium follows a similar structure to radio buttons. You can search for checkbox elements by their attributes such as ID, name, CSS selector, or XPath. Once checked, the `click()` method is used to change the selection state of the checkbox. This method responds to a user click, enabling or disabling the checkbox as needed.

- In addition to selecting checkboxes, it is often necessary to systematically check the current state. The `is_selected()` method in Selenium plays an important role in this regard. It lets you check if a checkbox is currently selected, facilitates self-authentication and ensures user selection is accurate in web testing The

combination of these techniques allows testers and developers to better manage checkbox interactions at in a web application.

Dropdown Menus:

- A drop-down menu is a common feature used to select an option from a list of options. They are overridden in web forms, allowing users to select specific options or choose from available options. Interaction with dropdown menus is a standard requirement when developing web automation testing in Selenium. If you want to make changes to dropdowns, you generally have to find the dropdown element in the web page. Once found, Selenium provides a dedicated tool for this task, called the Select class. This tutorial provides advanced techniques for working with dropdowns, making it easier to perform actions such as selection by option value, index, or visible information

- Selecting the class allows testers and developers to interact with dropdown menus in a systematic manner, mimicking user behavior. Whether it's selecting a specific option, managing the current selection, or iterating through available options, the Select class is a valuable resource for handling the low-level interaction exactly as a web-application test part of the system