Test Automation Using Python

Why Python for Automation testing?

- It is easy to learn and understand, simpler to code
- World is moving towards Artificial Intelligence with Machine learning, and Python plays a crucial role in implementing them.
- More jobs, Less competition

Selenium WebDriver

- Open source Tool to automates web applications
- Selenium can be coded in multiple languages- Java, Python, Javascript, C#, Ruby etc.
- FAQ: | already aware of Selenium -Java. Should I consider learning Selenium Python?

Course Prerequisites:

- This course assumes you do not have any prior knowledge in Python or Selenium. Every topic
- including Python Programming is taught from scratch with real-time examples

How this Course is Organized?

- Python Basics
- Selenium Fundamentals
- Pytest Fundamentals
- Logging and HTML Reporting
- Test Automation Framework
- Jenkins Integration with Automation Framework
- Miscellaneous

Glance on Selenium Features

Selenium Features

- Selenium is open Source Automation Testing tool
- It is exclusively for Web Based applications.
- Selenium supports multiple browsers - Chrome, Firefox, Internet Explorer, Safari
- Selenium works with Multiple Platforms Windows, Apple OS X, Linux
- Selenium can be coded in multiple languages - Java, C#, Python, Javascript, Python, php,Ruby

Instructions for Setting up Python and Selenium

- Install python.exe from their official website: https://www.python.org/downloads/
- Make sure you note Python Installation path on your machine.
    - It should be similar to: C:\Users(Your logged in User)\AppData\Local\Programs\Python\PythonXX
- Set Python home in System Variables
- Check if Python is Successfully Installed: `python --version`

Python in mac:

persent at : /Library/Frameworks/Python.framework/Versions/3.12/bin

What is PIP?

pip is the standard package manager for Python. It allows you to install and manage additional packages that are not part of the Python standard library.

Selenium installing instructions official link

https://pypi.org/project/selenium/

`pip install selenium`

This command will set up the Selenium WebDriver client library on your machine with all modules and classes that we will need to create automated scripts using Python

`pip install –U selenium`

The optional –U flag will upgrade the existing version of the installed package

Reasons Why ChromeDriver Checks Browser Version:

How It Works:

- Version Check:
    - When your Selenium script starts, it checks the version of the installed Chrome browser.
- Download Matching ChromeDriver:
    - If the correct version of ChromeDriver isn't installed, Selenium can download the appropriate version from the official ChromeDriver site.
- Installation:

    - The downloaded ChromeDriver is then installed locally and used for the Selenium session.

This process ensures that your browser automation runs smoothly and without issues caused by version mismatches.

In many organisation download of chrome driver is not allowed hence test fails so we can use chromedriver

service_obj = Service('/Users/vaibhavarde/Desktop/TestAutomation/Selenium_Python_Automation/01_mac-arm64/chromedriver')

driver = webdriver.Chrome(service = service_obj)

driver.get("https://google.com")

In Selenium 4, the relationship between **WebDriver**, **Chrome browser**, and **ChromeDriver** is crucial to understanding how browser automation works. Here's a breakdown of their roles and how they work together:

1. WebDriver:

- **WebDriver** is a **Selenium API** that allows you to interact with web browsers programmatically.
- It provides a way to control the browser (e.g., open it, navigate to web pages, fill forms, click buttons) by simulating user actions.
- **WebDriver** abstracts the communication between your code and the browser by using browser-specific drivers (like ChromeDriver for Chrome).

2. Chrome Browser:

- The **Chrome browser** is the actual web browser where your automation scripts will be executed.
- It is installed on your machine, and the **WebDriver** needs to communicate with it to control its actions.
- Each version of the Chrome browser is identified by a version number (e.g., 128.0.6613.138). The driver needs to match or be compatible with this version.

3. ChromeDriver:

- **ChromeDriver** is a separate executable that acts as a **bridge** between the Selenium **WebDriver** and the **Chrome browser**.
- It translates the WebDriver commands (like `driver.get()`, `driver.click()`, etc.) into low-level actions that the **Chrome browser** can understand.
- ChromeDriver must be installed on your system and must match or be compatible with the version of the Chrome browser you are using. If they are

incompatible, you get errors like "session not created" or version mismatches.

Relationship in Selenium 4:

1. **WebDriver** interacts with **ChromeDriver**, sending high-level commands like opening a page, clicking an element, or filling out a form.

2. **ChromeDriver** translates these WebDriver commands into the **Chrome DevTools Protocol**, which the **Chrome browser** can understand.

3. The **Chrome browser** performs the actions as per the commands received from **ChromeDriver**.

4. Results from the **Chrome browser** are sent back to the **ChromeDriver**, which in turn passes them back to the **WebDriver** API in your automation code.

Example Flow:

```
from selenium import webdriver

# WebDriver uses ChromeDriver to start the Chrome browser
driver = webdriver.Chrome()

# WebDriver sends a command to ChromeDriver to open a URL
driver.get("https://www.example.com")

# WebDriver sends a command to ChromeDriver to click an element
driver.find_element_by_id("example").click()

# WebDriver closes the browser through ChromeDriver
driver.quit()
```

Key Points in Selenium 4:

- In Selenium 4, **ChromeDriver** uses the **Chrome DevTools Protocol** to control the Chrome browser, which provides more stability and enhanced features like network mocking and improved performance.
- **ChromeDriver** and **Chrome browser** must be **compatible**. Mismatched versions lead to errors, so managing versions (using tools like `webdriver-manager`) is important.
- **WebDriver** provides the API layer for writing tests, but the actual communication with the browser is handled by **ChromeDriver**.

Summary:

- **WebDriver** is the high-level API that interacts with the **ChromeDriver**.
- **ChromeDriver** is a bridge that translates WebDriver commands into actions for the **Chrome browser** using the **Chrome DevTools Protocol**.
- **Chrome browser** is the target application where your tests are executed.

Each component plays a key role in browser automation, and they need to work together for smooth execution of Selenium tests.

In [ ]:
```
import time
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
# Chrome driver service - selenium will check current version of browser

# In many organisation download of chrome driver is not allowed hence tes

# service_obj = Service('/Users/vaibhavarde/Desktop/TestAutomation/Seleni
# driver = webdriver.Chrome(service = service_obj)

driver = webdriver.Chrome()

driver.get("https://google.com")

print(driver.title)
print(driver.current_url)

time.sleep(2)
```

In [ ]:
```
import time
from selenium import webdriver

driver = webdriver.Chrome()
# driver = webdriver.Firefox()
# driver = webdriver.Edge()

driver.get("https://google.com")

print(driver.title)
print(driver.current_url)

time.sleep(2)
```

Different Types of **Locators**

In Selenium, locators are used to identify elements on a web page. Here are different types of locators in Selenium using Python, along with sample code snippets:

1. **ID**

- **Description**: Locates an element using its `id` attribute.
- **Example**:
  `element = driver.find_element(By.ID, "element_id")`

2. **Name**

- **Description**: Locates an element using its `name` attribute.
- **Example**:
  `element = driver.find_element(By.NAME, "element_name")`

3. Class Name

- **Description**: Locates an element using its `class` attribute.
- **Example**:
  ```
  element = driver.find_element(By.CLASS_NAME, "class_name")
  ```

4. Tag Name

- **Description**: Locates elements using their tag name.
- **Example**:
  ```
  element = driver.find_element(By.TAG_NAME, "tag_name")
  ```

5. Link Text

- **Description**: Locates a hyperlink using its exact text.
- **Example**:
  ```
  element = driver.find_element(By.LINK_TEXT, "Exact Link Text")
  ```

6. Partial Link Text

- **Description**: Locates a hyperlink using a partial match of its text.
- **Example**:
  ```
  element = driver.find_element(By.PARTIAL_LINK_TEXT, "Partial
  Text")
  ```

7. XPath

- **Description**: Locates an element using an XPath expression.
- **Example**:
  ```
  element = driver.find_element(By.XPATH,
  "//tag[@attribute='value']")
  ```

8. CSS Selector

- **Description**: Locates an element using a CSS selector.
- **Example**:
  ```
  element = driver.find_element(By.CSS_SELECTOR, "css_selector")
  ```

9. DOM Locator

- **Description**: Locates elements based on DOM properties.
- **Example**:
  ```
  element = driver.execute_script("return
  document.querySelector('css_selector')")
  ```

10. Accessibility ID

- **Description**: Used for locating elements using the accessibility ID (for mobile web testing).
- **Example**:

```
element =
driver.find_element_by_accessibility_id("accessibility_id")
```

These locators are essential for interacting with web elements during Selenium automation.

Select

The `Select` class in Selenium is used to interact with <select> elements on web pages. These elements are typically used to create dropdown lists or combo boxes.

Key Methods:

1. **`select_by_index(index)`**: Selects an option by its index, starting from 0.

   ```
   from selenium.webdriver.support.ui import Select

   select = Select(driver.find_element_by_id("country"))
   select.select_by_index(2)  # Selects the third option
   ```
2. **`select_by_value(value)`**: Selects an option by its value attribute.

   ```
   select.select_by_value("US")  # Selects the option with the
   value "US"
   ```
3. **`select_by_visible_text(text)`**: Selects an option by its visible text.

   ```
   select.select_by_visible_text("United States")  # Selects the
   option with the text "United States"
   ```
4. **`deselect_all()`**: Deselects all options in a multiple-select element.

   ```
   select.deselect_all()
   ```
5. **`deselect_by_index(index)`**: Deselects an option by its index.

   ```
   select.deselect_by_index(0)  # Deselects the first option
   ```
6. **`deselect_by_value(value)`**: Deselects an option by its value attribute.

   ```
   select.deselect_by_value("CA")  # Deselects the option with
   the value "CA"
   ```
7. **`deselect_by_visible_text(text)`**: Deselects an option by its visible text.

   ```
   select.deselect_by_visible_text("Canada")  # Deselects the
   option with the text "Canada"
   ```

Example:

```
from selenium import webdriver
from selenium.webdriver.support.ui import Select

# Create a WebDriver instance
driver = webdriver.Chrome()

# Navigate to a webpage with a dropdown
driver.get("https://example.com")

# Find the select element
```

```
select_element = driver.find_element_by_id("country")

# Create a Select object
select = Select(select_element)

# Select an option
select.select_by_value("US")

# Deselect the option
select.deselect_by_value("US")
```

To retrieve the text after updating a value through a script in Selenium, you can follow these steps:

1. Update the Value Using JavaScript

You can use JavaScript to update a web element's value (e.g., for an input field) and then retrieve the updated value using the appropriate Selenium method.

2. Retrieve the Updated Value

After updating the value through the script, you can use `get_attribute("value")` for input fields or `text` for other elements like `div`, `span`, etc.

Example: Updating and Retrieving Value from an Input Field

```python
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://example.com")

# Locate the input field using any appropriate locator strategy
input_field = driver.find_element("id", "input_field_id")

# Update the value using JavaScript
driver.execute_script("arguments[0].value = 'New Value';",
input_field)

# Retrieve the updated value
updated_value = input_field.get_attribute("value")
print("Updated Value:", updated_value)
```

Example: Updating and Retrieving Text from Non-Input Elements

```python
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://example.com")
```

```
# Locate the element, for example, a span or div
element = driver.find_element("id", "element_id")

# Update the innerText using JavaScript
driver.execute_script("arguments[0].innerText = 'Updated Text';",
element)

# Retrieve the updated text
updated_text = element.text
print("Updated Text:", updated_text)
```

Explanation:

1. `execute_script`: This is used to run JavaScript code within the browser context. You can update the DOM directly with this.
2. `get_attribute("value")`: For input fields, this retrieves the current value after the change.
3. `element.text`: For non-input elements like `div`, `span`, etc., this retrieves the visible text.

This is how you can update a value using a script and retrieve the updated text or value with Selenium.

Implicit Wait vs. Explicit Wait in Selenium (Python)

In Selenium, **waiting** is crucial because it allows the script to pause for a certain period to ensure that web elements are fully loaded before performing any actions on them. There are two types of waits in Selenium: **Implicit Wait** and **Explicit Wait**. Both are used to handle scenarios where web elements take time to appear or become interactable, but they work differently.

---

1. Implicit Wait

- **Definition**: Implicit wait tells the WebDriver to poll the DOM for a certain amount of time when trying to find any element(s). Once you set the implicit wait, it applies globally to all elements in the script.
- **How It Works**: If the element is not immediately available, the WebDriver will wait for a specified duration before throwing a `NoSuchElementException`. If the element is found within the wait time, the script continues executing without waiting for the entire duration.

Code Example:

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.implicitly_wait(10)  # Wait for a maximum of 10 seconds
for elements to appear
```

```
driver.get("https://example.com")
element = driver.find_element_by_id("someElementId")  # Will wait
up to 10 seconds for the element to appear
```

- **When to Use**: Implicit wait is useful for most scenarios where you need to wait for elements across the whole script. However, it doesn't provide flexibility for different waits on specific elements or conditions.

---

2. **Explicit Wait**

- **Definition**: Explicit wait is used to wait for a **specific condition** to occur before proceeding with the execution. Unlike implicit wait, explicit wait is applied to individual elements and conditions.
- **How It Works**: With explicit wait, you define a wait condition (e.g., element to be visible, clickable, presence in the DOM, etc.). The WebDriver will wait for that specific condition to be true within a maximum time frame. If the condition is not met within the time limit, a `TimeoutException` is thrown.

Code Example:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Chrome()
driver.get("https://example.com")

# Wait until the element with the specific ID is visible, with a
maximum wait time of 10 seconds
element = WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.ID, "someElementId"))
)
```

- **When to Use**: Explicit waits are useful when specific elements require different waiting times or conditions. For instance, when waiting for a button to become clickable or waiting for a specific element to load after an AJAX call.

Differences Between Implicit and Explicit Waits

Implicit Wait	Explicit Wait
Applies globally across the entire WebDriver	Applies only to specific elements or conditions
Simple to use, set once	More flexible, supports multiple wait conditions
Waits for a fixed amount of time for elements	Waits for specific conditions to be met

Implicit Wait	Explicit Wait
No control over different conditions	Control over various conditions (e.g., clickable, visible)
Less efficient if waiting for different elements	More efficient as you can customize the wait for specific cases

---

Why `driver.find_elements` is an Exception

The behavior of the `find_elements` method differs from `find_element`, which explains why it behaves differently in terms of throwing exceptions.

- **`driver.find_element`**: This method **returns a single web element** and throws a `NoSuchElementException` if the element is not found within the wait time (either implicit or explicit).
- **`driver.find_elements`**: This method **returns a list of web elements**. If no elements are found, it **returns an empty list** rather than throwing an exception. Therefore, you won't see an exception like `NoSuchElementException` even if the elements are not present on the page. This makes `find_elements` useful when you expect multiple elements and want to handle scenarios where none of them are present without raising an exception.

Example of `find_elements`:

```python
elements = driver.find_elements(By.ID, "someElementId")

if not elements:
    print("No elements found!")
else:
    for element in elements:
        print(element.text)
```

In this case, `find_elements` will return an empty list if no elements are found, which is an expected behavior and doesn't cause an exception, allowing you to handle the absence of elements smoothly in your script.

Summary

- **Implicit Wait**: Waits globally for a specified amount of time for elements to appear in the DOM before throwing an exception.
- **Explicit Wait**: Waits for specific conditions on certain elements, offering more control and flexibility.
- **`find_elements` Exception**: `find_elements` returns an empty list when no elements are found instead of raising an exception, which is why it behaves differently from `find_element`.

Chaining of Web Elements in Selenium Python

Element chaining in Selenium refers to the process of locating an element and then searching for its child or nested elements by applying multiple locators one after another. This is useful when you need to find elements that are inside a specific container or parent element.

In Selenium, you can first find a parent element, then search for child elements within the context of that parent. This is more efficient than searching through the entire DOM each time and can help make element selection more specific and structured.

Example: Chaining Elements in Selenium Python

Consider the following HTML structure:

```
<div id="mainContainer">
    <div class="innerContainer">
        <p class="message">Welcome to Selenium Chaining!</p>
    </div>
</div>
```

Scenario:

We want to first find the `mainContainer` and then locate the `message` paragraph inside it.

Code Example: Chaining Web Elements

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

# Initialize the WebDriver
driver = webdriver.Chrome()

# Navigate to the website
driver.get("https://example.com")

# Locate the parent element (mainContainer)
main_container = driver.find_element(By.ID, "mainContainer")

# Within the parent element, locate the child element (message
inside innerContainer)
message = main_container.find_element(By.CLASS_NAME, "message")

# Fetch and print the text from the nested element
print(message.text)

# Close the WebDriver
driver.quit()
```

Explanation:

1. **Parent Element**: `main_container = driver.find_element(By.ID, "mainContainer")` locates the `mainContainer` element.
2. **Chained Search**: `message = main_container.find_element(By.CLASS_NAME, "message")` searches for the `message` paragraph inside `mainContainer` using its class name.
3. **Fetching Text**: The `.text` attribute is used to extract and print the text `"Welcome to Selenium Chaining!"` from the `message` element.

This approach limits the scope of the search to the `mainContainer` instead of the entire webpage, making it more efficient and less prone to errors when there are multiple elements with similar characteristics.

Benefits of Chaining Web Elements

- **Efficiency**: Reduces the scope of searching to a smaller part of the page.
- **Precision**: Ensures that you target child elements specifically inside a parent element.
- **Readability**: Your code becomes more structured and easier to follow, especially when dealing with complex DOMs.

Another Example: Chaining Multiple Levels

If the HTML has more levels of nesting, you can chain elements multiple times:

```html
<div id="mainContainer">
    <div class="innerContainer">
        <div class="textWrapper">
            <p class="message">Chaining multiple levels in 
Selenium</p>
        </div>
    </div>
</div>
```

Here's how to handle multiple levels:

```python
# Locate the parent element (mainContainer)
main_container = driver.find_element(By.ID, "mainContainer")

# Locate innerContainer inside mainContainer
inner_container = main_container.find_element(By.CLASS_NAME, 
"innerContainer")

# Locate textWrapper inside innerContainer
text_wrapper = inner_container.find_element(By.CLASS_NAME, 
"textWrapper")

# Locate the final message inside textWrapper
message = text_wrapper.find_element(By.CLASS_NAME, "message")

# Fetch and print the text from the message element
print(message.text)
```

In this example, the search is done in steps:

1. `mainContainer` -> `innerContainer` -> `textWrapper` -> `message`

Key Points:

- **Parent-Child Search**: Chaining web elements involves finding a parent element and then searching within it for child elements.
- **Efficiency**: It's more efficient as it narrows down the DOM search scope.
- **Readability**: Makes it easier to read and understand what part of the page you're interacting with.

By using element chaining, you can create reliable and efficient Selenium test scripts that handle complex page structures.

S14 : Handling Advance User Interaction and Child windows, Frames

ActionChains

Actions Chains in Selenium Python

In Selenium, **Action Chains** are used to perform complex user interactions like hovering over an element, right-clicking (context clicking), double-clicking, dragging and dropping, and clicking and holding, which can't be achieved using the standard Selenium methods.

The `ActionChains` class in Selenium provides a way to chain multiple actions together and perform them in sequence. These actions can simulate user interactions on the browser and web elements.

Importing ActionChains

You can import `ActionChains` from `selenium.webdriver.common.action_chains`.

```python
from selenium.webdriver.common.action_chains import ActionChains
```

Basic Syntax

```python
actions = ActionChains(driver)
```

Once you instantiate the `ActionChains` object, you can chain multiple actions and execute them using the `.perform()` method.

Commonly Used Actions in ActionChains

- **move_to_element**: Move the mouse to an element.
- **click**: Click on an element.
- **double_click**: Double-click on an element.
- **context_click**: Right-click (context click) on an element.
- **click_and_hold**: Click and hold the left mouse button on an element.
- **release**: Release the held mouse button.
- **drag_and_drop**: Drag an element and drop it at another location.
- **send_keys**: Send keys to the currently focused element.

Example: Hover Over an Element

Let's say you want to hover over an element and then click on a link that appears after hovering.

HTML Example:

```
<div id="menu">
    <button id="hoverButton">Hover over me</button>
    <a id="clickableLink" style="display:none">Click me!</a>
</div>
```

We will hover over the button, and the link will appear. Then we will click on the link.

Code Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.action_chains import ActionChains

# Initialize WebDriver
driver = webdriver.Chrome()

# Navigate to the web page
driver.get("https://example.com")

# Find the elements
hover_button = driver.find_element(By.ID, "hoverButton")
clickable_link = driver.find_element(By.ID, "clickableLink")

# Create the ActionChains object
actions = ActionChains(driver)

# Hover over the button
actions.move_to_element(hover_button).perform()

# Once the link appears, click on it
clickable_link.click()

# Close the browser
driver.quit()
```

Explanation:

1. **move_to_element(hover_button)**: This moves the mouse pointer over the `hoverButton`. It simulates hovering.
2. **perform()**: This executes the action that has been built up in the action chain (moving the mouse to the element).
3. **click()**: After the hover, the link appears, and Selenium clicks on it.

Example: Drag and Drop

In the following example, we will simulate dragging one element and dropping it into another location.

HTML Example:

```
<div id="dragContainer">
    <div id="source">Drag me</div>
    <div id="target">Drop here</div>
</div>
```

Code Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.action_chains import ActionChains

# Initialize WebDriver
driver = webdriver.Chrome()

# Navigate to the web page
driver.get("https://example.com")

# Locate the source and target elements
source_element = driver.find_element(By.ID, "source")
target_element = driver.find_element(By.ID, "target")

# Create ActionChains object
actions = ActionChains(driver)

# Perform drag and drop
actions.drag_and_drop(source_element, target_element).perform()

# Close the browser
driver.quit()
```

Explanation:

1. **drag_and_drop(source_element, target_element)**: This method handles dragging the `source_element` and dropping it onto the `target_element`.
2. **perform()**: Executes the drag-and-drop action.

Example: Double Click

If you need to double-click on an element, you can chain the `double_click` method.

HTML Example:

```html
<button id="dblClickButton">Double-click me</button>
```

Code Example

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

# Initialize WebDriver
driver = webdriver.Chrome()

# Navigate to the web page
driver.get("https://example.com")

# Find the double-click button
double_click_button = driver.find_element(By.ID,
"dblClickButton")

# Create ActionChains object
actions = ActionChains(driver)

# Double click on the button
actions.double_click(double_click_button).perform()

# Close the browser
driver.quit()
```

Explanation:

1. **double_click(double_click_button)**: This method simulates a double-click on the specified button.
2. **perform()**: Executes the double-click action.

Chaining Multiple Actions

You can chain multiple actions together before executing them with `perform()`. For instance, you could move to an element, click it, and then send keys.

```python
actions = ActionChains(driver)

# Move to element, click it, and send some keys
actions.move_to_element(element).click().send_keys("Selenium is
awesome!").perform()
```

Explanation:

1. **move_to_element(element)**: Moves the mouse pointer over the specified element.
2. **click()**: Clicks on the element.
3. **send_keys()**: Types the text `"Selenium is awesome!"` into the selected field.

Benefits of ActionChains

1. **Complex User Interactions**: Handle advanced user actions like drag-and-drop, hovering, and context-clicking.
2. **Chained Actions**: You can combine multiple actions into a single sequence.
3. **Realistic Simulations**: Simulate real user interactions in web applications for robust testing.

Conclusion

Action Chains in Selenium Python are essential when you need to perform complex user interactions. They provide functionality that goes beyond simple click or text-input actions and allow you to simulate realistic user behavior like hovering, dragging and dropping, and right-clicking. By chaining actions together, you can create more interactive and effective test scripts.

Switch Between Tabs or Child Windows

In Selenium Python, switching between tabs or windows is done using the `window_handles` and `switch_to.window()` methods. These allow you to move between different browser windows or tabs during the automation process.

Steps to Switch Between Tabs or Child Windows:

1. **Open a new tab or window** (triggered by clicking a link or using JavaScript).
2. **Store the handle of the current (parent) window**.
3. **Get all the window handles**.
4. **Switch to the child window or tab**.
5. **Perform actions on the child window/tab**.
6. **Switch back to the parent window**.

Key Methods:

- `driver.current_window_handle`: Returns the handle (ID) of the current window.
- `driver.window_handles`: Returns a list of handles of all the open browser windows.

- `driver.switch_to.window(handle)` : Switches to a specific window using its handle.

Example: Switching Between Tabs/Windows

Let's look at an example where you click a link that opens a new tab or window, and then switch between the parent and child windows.

HTML Example

```html
<a href="https://www.example.com" target="_blank">Open New
Tab</a>
```

Selenium Code

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

# Initialize the WebDriver
driver = webdriver.Chrome()

# Open a website
driver.get("https://yourwebsite.com")

# Store the current window handle (parent window)
parent_window = driver.current_window_handle

# Click a link that opens a new tab/window
driver.find_element(By.LINK_TEXT, "Open New Tab").click()

# Wait for the new window/tab to open (optional)
time.sleep(3)

# Get a list of all open window handles
all_windows = driver.window_handles

# Switch to the new window/tab (child window)
for window_handle in all_windows:
    if window_handle != parent_window:
        driver.switch_to.window(window_handle)
        break

# Now perform actions on the new tab/window
print("Child window title:", driver.title)

# Close the child window
driver.close()

# Switch back to the parent window
driver.switch_to.window(parent_window)

# Perform actions in the parent window
```

```python
print("Parent window title:", driver.title)

# Close the browser
driver.quit()
```

Explanation:

1. **`driver.get()`** : Opens the main webpage.
2. **`parent_window = driver.current_window_handle`** : Stores the handle (ID) of the parent window before the new tab/window opens.
3. **`driver.find_element(By.LINK_TEXT, "Open New Tab").click()`** : Clicks a link that opens a new tab or window.
4. **`driver.window_handles`** : Retrieves the handles of all open windows/tabs.
5. **`driver.switch_to.window(handle)`** : Switches to the child window.
6. **`driver.close()`** : Closes the child window after performing actions.
7. **`driver.switch_to.window(parent_window)`** : Switches back to the parent window.
8. **`driver.quit()`** : Closes the browser entirely.

Example: Switching Back and Forth Between Windows

You might have cases where you need to interact with both windows back and forth. Here's an example:

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

# Initialize WebDriver
driver = webdriver.Chrome()

# Open a website
driver.get("https://yourwebsite.com")

# Store the parent window handle
parent_window = driver.current_window_handle

# Click a link that opens a new tab/window
driver.find_element(By.LINK_TEXT, "Open New Tab").click()

# Get all window handles
all_windows = driver.window_handles

# Switch to the new tab/window
driver.switch_to.window(all_windows[1])

# Perform actions on the new tab/window
print("Switched to child window:", driver.title)

# Switch back to the parent window
driver.switch_to.window(parent_window)
```

```python
# Perform actions on the parent window
print("Switched back to parent window:", driver.title)

# Close the browser
driver.quit()
```

Explanation:

- `window_handles[1]` : Refers to the child window (since `window_handles[0]` is the parent window).
- **Switching back to parent**: After interacting with the child window, we switch back to the parent using the same process.

Handling Multiple Windows or Tabs

If you open multiple tabs or windows and need to switch between them, the process is the same, except you iterate through all the window handles to switch between them. Here's an example for handling more than two windows:

```python
all_windows = driver.window_handles
for handle in all_windows:
    driver.switch_to.window(handle)
    print("Current window title:", driver.title)
```

This loop will allow you to switch through all open windows/tabs and print their titles or perform other actions.

Best Practices for Handling Windows/Tabs:

- **Always store the parent window handle** before opening a new window/tab so you can switch back easily.
- **Close child windows** when they are no longer needed to avoid unnecessary resource usage.
- **Use waits or sleep** if the new window takes time to load, but prefer explicit waits to handle dynamic page loading.

Switching between frames and back to the original content

In Selenium Python, switching between frames and back to the original content is essential when interacting with web pages that have embedded **iframes**.

Key Methods for Switching Frames:

- `driver.switch_to.frame(frame_reference)` : Switches the context to the specified iframe.

- `driver.switch_to.default_content()` : Switches back to the main page (original content).

Ways to Identify a Frame:

1. **By index**: If multiple frames exist, you can switch by using the index number (starting from 0).
2. **By name or ID**: If the frame has an `id` or `name` attribute, you can use it to switch.
3. **By WebElement**: You can locate the iframe as a WebElement and switch to it.

Example: Switching to an Iframe and Back to the Main Content

Here's how you can switch to a frame, perform actions inside it, and then return to the original content.

HTML Example

```html
<iframe id="iframe1" src="https://www.example.com"></iframe>
<button id="buttonOutsideFrame">Click Me</button>
```

Selenium Python Code

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

# Initialize WebDriver
driver = webdriver.Chrome()

# Open a website with an iframe
driver.get("https://yourwebsite.com")

# Switch to the iframe by ID
driver.switch_to.frame("iframe1")

# Perform actions within the iframe
element_in_iframe = driver.find_element(By.TAG_NAME, "h1")
print("Text inside iframe:", element_in_iframe.text)

# Switch back to the main page (default content)
driver.switch_to.default_content()

# Perform actions in the main page
button = driver.find_element(By.ID, "buttonOutsideFrame")
button.click()

# Close the browser
driver.quit()
```

Explanation:

1. **`driver.switch_to.frame("iframe1")`** : Switches the context to the iframe with the ID `iframe1`.
2. **Perform actions inside the iframe**: You can now interact with elements inside the iframe.
3. **`driver.switch_to.default_content()`** : Switches back to the main document (original page).
4. **Interact with elements on the main page**: After switching back, you can interact with the rest of the page.

Switching Between Multiple Frames

If the page has multiple iframes, you can switch between them by index or using different reference methods. Here's how you can switch between multiple frames:

```
# Switch to the first iframe by index (0 is the first frame)
driver.switch_to.frame(0)

# Perform actions in the first frame
print("First frame title:", driver.title)

# Switch to the second iframe by index
driver.switch_to.default_content()  # First, return to the main
page
driver.switch_to.frame(1)           # Then, switch to the second
frame

# Perform actions in the second frame
print("Second frame title:", driver.title)

# Switch back to the original page
driver.switch_to.default_content()
```

Switching to Frame by WebElement

You can also locate the iframe as a WebElement and switch to it:

```
# Locate the iframe as a WebElement
iframe_element = driver.find_element(By.XPATH,
"//iframe[@id='iframe1']")

# Switch to the iframe using the WebElement
driver.switch_to.frame(iframe_element)

# Perform actions within the iframe
element_in_iframe = driver.find_element(By.TAG_NAME, "h1")
print("Text inside iframe:", element_in_iframe.text)

# Switch back to the main content
driver.switch_to.default_content()
```

Points to Remember:

- Always switch back to the default content after interacting with the frame, using `driver.switch_to.default_content()`.
- If there are nested frames (frames within frames), you may need to switch between them using multiple `switch_to.frame()` calls.
- If you try to interact with elements outside the frame without switching back, you'll get an `NoSuchElementException`.

S15 : Selenium Python Miscellaneous

How To Scroll using Selenium

Selenium doesn't have a built-in method to scroll the page. However, you can use JavaScript to scroll down a webpage with Selenium. This is because the browser's native scrolling behavior can be controlled via JavaScript, which Selenium allows you to execute.

Explanation:

To scroll the page in Selenium, you can use JavaScript's `window.scrollTo()` function to scroll the page to a certain position or scroll down to the bottom. This can be helpful when interacting with elements that are not immediately visible.

Selenium's `execute_script` method allows you to run custom JavaScript code within the browser.

Example:

Here's an example using Python with Selenium:

```
from selenium import webdriver
from time import sleep

# Setup WebDriver (Assuming ChromeDriver)
driver = webdriver.Chrome()

# Open a webpage
driver.get("https://example.com")

# Scroll down by a specific pixel value
driver.execute_script("window.scrollTo(0, 1000);")  # Scrolls
down 1000 pixels vertically

# Pause to observe the scrolling
sleep(2)

# Scroll to the bottom of the page
driver.execute_script("window.scrollTo(0,
document.body.scrollHeight);")
```

```python
# Pause to observe the scrolling
sleep(2)

# Close the browser
driver.quit()
```

Breakdown:

1. **Scroll by Pixel Value**: `window.scrollTo(0, 1000);`
   - The first argument (`0`) is the horizontal scroll (x-axis), and the second argument (`1000`) is the vertical scroll (y-axis). This scrolls down 1000 pixels.
2. **Scroll to the Bottom**: `window.scrollTo(0, document.body.scrollHeight);`
   - `document.body.scrollHeight` gets the total height of the page, ensuring the page is scrolled all the way to the bottom.

Why use JavaScript?

- **Dynamic Loading**: Pages with infinite scrolling or lazy-loaded elements require scrolling to reveal additional content. JavaScript scroll commands allow you to trigger the scroll event.
- **Element Visibility**: If a web element is located far down a page, scrolling ensures it becomes visible for Selenium to interact with it.

Scroll by the height of the current screen(ViewPort)

One can scroll by the height of the current screen (also known as the "viewport") using JavaScript with Selenium. The `window.innerHeight` property in JavaScript allows you to get the height of the current viewport, and you can use that to scroll by the height of the visible area instead of a fixed pixel value.

Here's how you can do it:

Example:

```python
from selenium import webdriver
from time import sleep

# Setup WebDriver (Assuming ChromeDriver)
driver = webdriver.Chrome()

# Open a webpage
driver.get("https://example.com")

# Get the height of the current window (viewport)
```

```
viewport_height = driver.execute_script("return
window.innerHeight;")

# Scroll down by the viewport height
driver.execute_script(f"window.scrollBy(0, {viewport_height});")

# Pause to observe the scrolling
sleep(2)

# Scroll again by the viewport height
driver.execute_script(f"window.scrollBy(0, {viewport_height});")

# Pause to observe the scrolling
sleep(2)

# Close the browser
driver.quit()
```

Explanation:

- **Get the Viewport Height**:
    - `window.innerHeight` gets the height of the visible area of the browser window (the viewport). This ensures the scroll happens exactly by the height of the screen being viewed.
- **Scroll by Viewport Height**:
    - `window.scrollBy(0, {viewport_height})` scrolls vertically by the height of the current screen, so each scroll moves the page by one screen height.

You can repeat this scroll as needed to simulate page-down behavior, which is useful for interacting with elements that are not in the visible portion of the page.

ScreenShot

In Selenium with Python, you can easily take a screenshot of the current browser window using the `save_screenshot()` or `get_screenshot_as_file()` methods. These methods save the screenshot in the specified file path.

Example 1: Using `save_screenshot()`

```
from selenium import webdriver

# Setup WebDriver (Assuming ChromeDriver)
driver = webdriver.Chrome()

# Open a webpage
driver.get("https://example.com")

# Take a screenshot and save it as a file
```

```
driver.save_screenshot("screenshot.png")

# Close the browser
driver.quit()
```

Example 2: Using `get_screenshot_as_file()`

```
from selenium import webdriver

# Setup WebDriver (Assuming ChromeDriver)
driver = webdriver.Chrome()

# Open a webpage
driver.get("https://example.com")

# Take a screenshot and save it as a file
driver.get_screenshot_as_file("screenshot.png")

# Close the browser
driver.quit()
```

Example 3: Screenshot of a specific element

If you want to capture a screenshot of a specific element on the page, you can do that by first finding the element and using `screenshot()` method.

```
from selenium import webdriver

# Setup WebDriver (Assuming ChromeDriver)
driver = webdriver.Chrome()

# Open a webpage
driver.get("https://example.com")

# Find the specific element (replace with actual element
selector)
element = driver.find_element("tag name", "h1")  # Use
appropriate selector

# Take a screenshot of the element
element.screenshot("element_screenshot.png")

# Close the browser
driver.quit()
```

Key Points:

- **Full page screenshot**: Use `driver.save_screenshot()` or `driver.get_screenshot_as_file()` to capture the entire page.
- **Element screenshot**: Use `element.screenshot()` to capture a specific element on the page.

These methods will store the screenshots in the specified location on your local file system. Let me know if you need more clarification!

Headless

Running Selenium tests in **headless mode** means running them without opening a visible browser window. This is especially useful when you want to run tests on servers or environments where you don't have a display or GUI.

To enable **headless mode** in Selenium with Python, you can pass the `--headless` option to the Chrome or Firefox browser.

Example with Chrome:

```python
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# Setup Chrome options
chrome_options = Options()
chrome_options.add_argument("--headless")  # Enable headless mode
chrome_options.add_argument("--window-size=1920x1080")  #
Optional: Set window size to prevent issues

# Setup WebDriver with headless Chrome
driver = webdriver.Chrome(options=chrome_options)

# Open a webpage in headless mode
driver.get("https://example.com")

# Get the title of the page
print(driver.title)

# Close the browser
driver.quit()
```

Example with Firefox:

```python
from selenium import webdriver
from selenium.webdriver.firefox.options import Options

# Setup Firefox options
firefox_options = Options()
firefox_options.headless = True  # Enable headless mode

# Setup WebDriver with headless Firefox
driver = webdriver.Firefox(options=firefox_options)

# Open a webpage in headless mode
driver.get("https://example.com")

# Get the title of the page
print(driver.title)
```

```
# Close the browser
driver.quit()
```

Other way for headless:

```
from selenium import webdriver

chrome_options =  webdriver.ChromeOptions()
chrome_options.add_argument("headless")

driver = webdriver.Chrome(options=chrome_options)
driver.get("https://rahulshettyacademy.com/AutomationPractice/")

print("Total height of page : ", driver.execute_script("return 
document.body.scrollHeight"))
driver.execute_script("window.scrollTo(0, 
document.body.scrollHeight)")
driver.execute_script("window.scrollTo(0, 0)")

# Get the height of the current window (viewport)
viewport_height = driver.execute_script("return 
window.innerHeight;")
driver.execute_script(f"window.scrollBy(0, {viewport_height});")
driver.execute_script("window.scrollTo(0, 0)")
print("Headless mode done!!")
```

Key Points:

- **Chrome headless mode**: You can use the `--headless` argument in Chrome's options.
- **Firefox headless mode**: Set `firefox_options.headless = True` in Firefox's options.
- **Window size**: It's a good practice to specify the `--window-size` argument in headless mode to avoid issues with element visibility or layout.

Benefits of Headless Mode:

1. **Faster execution**: No need to render a GUI, making the tests run faster.
2. **Automation-friendly**: Ideal for CI/CD pipelines or cloud environments without a display.
3. **Resource-efficient**: Consumes less memory and CPU.

Handling Certificate Errors

In Selenium, when you encounter websites with invalid or expired SSL certificates, your browser may block access and display a certificate error. This can cause issues when running Selenium automation tests. You can handle these certificate errors by configuring the browser to ignore them.

In Selenium with Python, you can set specific browser options to ignore certificate errors. Below are examples for Chrome and Firefox.

Handling Certificate Errors in Chrome:

To ignore certificate errors in **Chrome**, you can use the `--ignore-certificate-errors` argument in Chrome options.

Example:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# Setup Chrome options
chrome_options = Options()
chrome_options.add_argument("--ignore-certificate-errors")  #
Ignore certificate errors

# Initialize WebDriver with the Chrome options
driver = webdriver.Chrome(options=chrome_options)

# Open a webpage with an invalid certificate
driver.get("https://expired.badssl.com/")

# Print page title (to verify the page loads)
print(driver.title)

# Close the browser
driver.quit()
```

Handling Certificate Errors in Firefox:

For **Firefox**, you can configure Firefox profiles to accept untrusted SSL certificates automatically.

Example:

```
from selenium import webdriver
from selenium.webdriver.firefox.options import Options

# Setup Firefox profile
firefox_profile = webdriver.FirefoxProfile()
firefox_profile.accept_untrusted_certs = True  # Automatically
accept untrusted certificates

# Setup WebDriver with the profile
driver = webdriver.Firefox(firefox_profile=firefox_profile)

# Open a webpage with an invalid certificate
driver.get("https://expired.badssl.com/")

# Print page title (to verify the page loads)
print(driver.title)
```

```python
# Close the browser
driver.quit()
```

Key Points:

1. **Chrome**:
   - Use the `--ignore-certificate-errors` flag to make Chrome ignore SSL certificate errors.
   - This helps the browser load pages with invalid or self-signed certificates without blocking access.
2. **Firefox**:
   - Set `accept_untrusted_certs = True` in the Firefox profile.
   - This automatically accepts certificates without asking for user confirmation.
3. **Why Use This?**:
   - This is useful when automating tests on staging or development environments where certificates may not be fully valid (e.g., self-signed certificates).
4. **Security Consideration**:
   - Ignoring SSL certificate errors can be risky for live environments, so this should be limited to test cases and non-production use.

By using these options, your Selenium scripts will proceed even if the website's SSL certificate is invalid or expired, making it easier to test pages without running into certificate error blocks.

Chrome Options

In Selenium, **ChromeOptions** allows you to set various options that customize the behavior of the Chrome browser. These options are crucial when running tests in headless mode, dealing with SSL errors, managing browser performance, and customizing the browser environment for automation purposes.

Here are some **important Chrome options** along with examples of how and why they are used:

1. Headless Mode (`--headless`)

Why: This option allows running the browser in headless mode, which means the browser will run without a GUI (Graphical User Interface). This is useful for running tests in environments without a display (e.g., CI/CD pipelines).

Usage:

```python
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

chrome_options = Options()
chrome_options.add_argument("--headless")  # Run Chrome in headless mode

driver = webdriver.Chrome(options=chrome_options)
driver.get("https://www.example.com")
print(driver.title)
driver.quit()
```

2. Disable GPU (`--disable-gpu`)

Why: This is primarily used along with `--headless` in older versions of Chrome where headless mode could cause issues on systems using GPUs. It ensures smooth headless execution.

Usage:

```python
chrome_options.add_argument("--disable-gpu")  # Disable GPU for headless execution
```

3. Ignore Certificate Errors (`--ignore-certificate-errors`)

Why: This option ignores SSL certificate errors. It's particularly useful when testing on development or staging environments where self-signed or expired certificates might be used.

Usage:

```python
chrome_options.add_argument("--ignore-certificate-errors")  # Ignore SSL certificate warnings
```

4. Start Maximized (`--start-maximized`)

Why: Automatically starts Chrome in a maximized window, ensuring all elements are visible and interactable. This can prevent errors due to hidden elements caused by small windows.

Usage:

```python
chrome_options.add_argument("--start-maximized")  # Start browser maximized
```

5. Incognito Mode (`--incognito`)

Why: Opens Chrome in incognito (private browsing) mode. This is useful when you want to test scenarios without using the browser cache, cookies, or session data.

Usage:

```
chrome_options.add_argument("--incognito")  # Open browser in
incognito mode
```

6. Disable Extensions (`--disable-extensions`)

Why: Disables all installed extensions in the browser. This is useful for preventing extensions from interfering with test cases.

Usage:

```
chrome_options.add_argument("--disable-extensions")  # Disable
extensions
```

7. Disable Pop-up Blocking (`--disable-popup-blocking`)

Why: Disables Chrome's built-in pop-up blocker, ensuring that all pop-ups appear and can be tested.

Usage:

```
chrome_options.add_argument("--disable-popup-blocking")  #
Disable popup blocking
```

8. Window Size (`--window-size=width,height`)

Why: Specifies the browser's window size. Useful when testing responsiveness or running tests in headless mode where a specific window size may be required.

Usage:

```
chrome_options.add_argument("--window-size=1920,1080")  # Set
window size to 1920x1080
```

9. User-Agent (`--user-agent`)

Why: Allows you to specify a custom user-agent string, useful when testing how your website behaves with different browsers, devices, or versions.

Usage:

```
chrome_options.add_argument("user-agent=Mozilla/5.0 (iPhone; CPU
iPhone OS 13_2_3 like Mac OS X)")
```

10. No Sandbox (`--no-sandbox`)

Why: This option is mainly used when running tests in environments where Chrome is running without root privileges. It bypasses some security restrictions imposed by the sandbox.

Usage:

```
chrome_options.add_argument("--no-sandbox")  # Bypass OS security
model
```

11. Disable Notifications (`--disable-notifications`)

Why: Prevents Chrome from displaying notification pop-ups. This is important for avoiding interruptions during test runs.

Usage:

```
chrome_options.add_argument("--disable-notifications")  # Disable
notification popups
```

12. Disable Infobars (`--disable-infobars`)

Why: Disables the Chrome "Chrome is being controlled by automated test software" infobar that appears during automated test runs. It can help prevent this message from affecting UI interactions.

Usage:

```
chrome_options.add_argument("--disable-infobars")  # Disable
'Chrome is being controlled' message
```

13. Disable Browser Automation Flag (`--disable-blink-features=AutomationControlled`)

Why: Prevents websites from detecting that the browser is being controlled by automation tools. Some sites block requests if they detect automated browsing.

Usage:

```
chrome_options.add_argument("--disable-blink-
features=AutomationControlled")
```

Putting It All Together (Comprehensive Example):

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# Setup Chrome options
chrome_options = Options()
chrome_options.add_argument("--headless")                    #
Headless mode for CI environments
chrome_options.add_argument("--ignore-certificate-errors")    #
Ignore SSL certificate errors
chrome_options.add_argument("--window-size=1920,1080")        #
Set window size
chrome_options.add_argument("--disable-extensions")           #
Disable browser extensions
```

```
chrome_options.add_argument("--disable-infobars")           #
Disable 'Chrome is being controlled' infobar
chrome_options.add_argument("--disable-popup-blocking")     #
Disable popup blocking
chrome_options.add_argument("--disable-notifications")      #
Disable notification popups
chrome_options.add_argument("--incognito")                  #
Run in incognito mode

# Initialize WebDriver with the Chrome options
driver = webdriver.Chrome(options=chrome_options)

# Open a webpage
driver.get("https://example.com")

# Interact with the webpage (for example, printing the title)
print(driver.title)

# Close the browser
driver.quit()
```

Summary:

- **Headless mode** is useful for running tests without a GUI.
- **Ignore SSL certificate errors** is helpful in testing non-production environments.
- **Maximized window and window size** help avoid visibility issues.
- **Incognito mode** is handy for tests that need a fresh session.
- **Disabling extensions and pop-ups** ensures a clean testing environment without distractions or interference.

Each of these options is designed to give you more control over the Chrome browser environment and to make your automated tests more stable and flexible.

User-Agent strings can be modified to simulate browsers and devices from different countries. However, note that **User-Agent** strings do not inherently carry geographic location information. Websites often detect your location based on your **IP address** rather than the User-Agent string. But you can simulate different device/browser configurations for various locales or regions using **localized browsers** or **language preferences** in the User-Agent string.

Here are examples of User-Agent strings to simulate browsing from different countries, typically by adjusting the **language settings** within the User-Agent string.

1. **United States (English)**:

```
plaintext
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
Safari/537.36
```

2. **France (French) - Desktop**:

```
plaintext
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
Safari/537.36 Accept-Language: fr-FR,fr;q=0.9,en-
US;q=0.8,en;q=0.7
```

- **Explanation**: The `Accept-Language` parameter `fr-FR,fr` ensures the browser simulates French preferences.

3. **Germany (German) - Desktop**:

```
plaintext
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
Safari/537.36 Accept-Language: de-DE,de;q=0.9,en-
US;q=0.8,en;q=0.7
```

4. **India (English - IN) - Mobile (Chrome on Android)**:

```
plaintext
Mozilla/5.0 (Linux; Android 10; SM-A505F)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
Mobile Safari/537.36 Accept-Language: en-IN,en;q=0.9
```

- **Explanation**: This simulates a Samsung Galaxy mobile device with Indian English (`en-IN`) as the preferred language.

5. **Japan (Japanese) - Desktop**:

```
plaintext
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
Safari/537.36 Accept-Language: ja-JP,ja;q=0.9,en-
US;q=0.8,en;q=0.7
```

6. **China (Simplified Chinese) - Desktop**:

```
plaintext
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
Safari/537.36 Accept-Language: zh-CN,zh;q=0.9
```

- **Explanation**: The language preferences in the User-Agent string are set to Simplified Chinese (`zh-CN`).

7. **Brazil (Portuguese - BR) - Desktop**:

```
plaintext
Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0
```

```
Safari/537.36 Accept-Language: pt-BR,pt;q=0.9,en-
US;q=0.8,en;q=0.7
```

Example in Selenium Code:

You can use these **User-Agent** strings in Selenium to simulate a browser from different locales as shown below:

```python
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# Setup Chrome options
chrome_options = Options()
chrome_options.add_argument("--headless")               #
Headless mode
chrome_options.add_argument("user-agent=Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/117.0.0.0 Safari/537.36 Accept-Language: fr-
FR,fr;q=0.9,en-US;q=0.8,en;q=0.7")  # Simulate French browser

# Initialize WebDriver with the Chrome options
driver = webdriver.Chrome(options=chrome_options)

# Open a webpage
driver.get("https://www.example.com")

# Interact with the webpage
print(driver.title)

# Close the browser
driver.quit()
```

Important Notes:

1. **Location Detection**: The User-Agent primarily controls browser and device type detection, but geographic location is often determined using **IP address**.
2. **Accept-Language Header**: By modifying the `Accept-Language` parameter in the User-Agent string, you can simulate different regions' language preferences.

This approach helps in testing how websites behave when visited by users in different countries or using various locales, especially when combined with IP geolocation services or VPNs.
