**Selenium WebDriver in VS Code for Python automation testing:**

**Step 1: Install Visual Studio Code (VS Code)**

1. **Download and install VS Code from the** [**official website**](https://code.visualstudio.com/)**.**
2. **Install Python (if not already installed) from the** [**official Python website**](https://www.python.org/downloads/)**.**

### **Step 3: Create a New Project in VS Code**

1. Open **VS Code**.
2. Open the **Terminal** inside VS Code:
3. From the menu: **Terminal → New Terminal**.
4. Create a new project folder
5. Create a virtual environment for project isolation: python -m venv venv
6. Activate the virtual environment: .\venv\Scripts\activate
7. Install **Selenium** by running the following command in the VS Code terminal:  
   pip install selenium

### **Step 4: Download ChromeDriver**

1. Download the **ChromeDriver** that matches your installed **Google Chrome** version from ChromeDriver.
2. Platform Supported by Selenium(Chrome version 115+ -> Stable version)
3. Extract the downloaded ZIP file to a folder (e.g., C:\chromedriver)
4. Hold Shift + Right Click(Copy As Path)

### **Step 5: Write a Selenium Test**

1. Create a new Python file (e.g., SeleniumTest.py) in your VSCode project.
2. Add the following sample code to test Google Search using Selenium.

from selenium import webdriver

import time

from selenium.webdriver.chrome.service import Service

from selenium.webdriver.common.keys import Keys

from selenium.webdriver.common.by import By

service = Service(r"C:\Users\user*\D*ownloads*\W*ebDrivers\chromedriver*.*exe")

driver = webdriver.Chrome(*service*=service)

driver.maximize\_window()

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

search\_Box = driver.find\_element(By.NAME,"q")

search\_Box.send\_keys("Hero Isl")

time.sleep(2)

search\_Box.send\_keys(Keys.RETURN)

time.sleep(2)

driver.quit()

### **Step 7: Run Your Selenium Test**

1. Right-click on the Python file in Vs Code and select **Run Code**.
2. Vs Code will compile and execute the program.
3. You should see Chrome open, navigate to Google, perform a search, and close automatically.

# **🚀 How to Prepare Before Writing Selenium Scripts**

### **📌 Flowchart for Selenium Python Testing Framework**

* **plaintext**
* **CopyEdit**
* **┌────────────────────────────┐**
* **│ 1. Import Libraries │**
* **│ (selenium, unittest, etc.)│**
* **└───────────┬────────────────┘**
* **│**
* **▼**
* **┌────────────────────────────┐**
* **│ 2. Create Test Class │**
* **│ (unittest.TestCase) │**
* **└───────────┬────────────────┘**
* **│**
* **▼**
* **┌────────────────────────────┐**
* **│ 3. Setup WebDriver │**
* **│ (def setUpClass(cls)) │**
* **│ - Initialize WebDriver │**
* **│ - Set implicit wait │**
* **│ - Maximize window │**
* **└───────────┬────────────────┘**
* **│**
* **▼**
* **┌────────────────────────────┐**
* **│ 4. Define Interaction Methods │**
* **│ - Click an element │**
* **│ - Send text to input field │**
* **│ - Select dropdown options │**
* **│ - Switch frames etc. │**
* **└───────────┬────────────────┘**
* **│**
* **▼**
* **┌────────────────────────────┐**
* **│ 5. Define Test Case (test\_master) │**
* **│ - Open URL │**
* **│ - Perform login action │**
* **│ - Navigate through menus │**
* **│ - Interact with elements │**
* **└───────────┬────────────────┘**
* **│**
* **▼**
* **┌────────────────────────────┐**
* **│ 6. Tear Down Class │**
* **│ (def tearDownClass(cls)) │**
* **│ - Quit the driver │**
* **└────────────────────────────┘**

**Selenium Browser Commands Cheat Sheet**

## **1. Opening and Closing the Browser**

* WebDriver driver = new ChromeDriver(); → Launches a new Chrome browser.
* driver.quit(); → Closes all browser windows and ends the session.
* driver.close(); → Closes the current browser window only.

## **2. Navigating to a Web Page**

* driver.get("https://example.com"); → Opens a specific URL.
* driver.navigate().to("https://example.com"); → Navigates to a URL (alternative to get())

## **3. Navigating Between Pages**

* driver.navigate().back(); → Navigates to the previous page.
* driver.navigate().forward(); → Navigate to the next page.
* driver.navigate().refresh(); → Refreshes the current page.

## **4. Managing Browser Windows**

* driver.manage().window().maximize(); → Maximizes the browser window.
* driver.manage().window().minimize(); → Minimizes the browser window.
* driver.manage().window().fullscreen(); → Switches to full-screen mode.
* driver.manage().window().setSize(new Dimension(1200, 800)); → Resizes the browser.

## **5. Handling Browser Tabs and Windows**

* String mainWindow = driver.getWindowHandle(); → Gets the current window handle.
* Set<String> allWindows = driver.getWindowHandles(); → Gets all open windows.
* driver.switchTo().window(windowHandle); → Switches to a specific window.

## **6. Managing Browser Cookies**

* driver.manage().getCookies(); → Gets all cookies.
* driver.manage().getCookieNamed("cookie\_name"); → Gets a specific cookie.
* driver.manage().addCookie(new Cookie("name", "value")); → Adds a cookie.
* driver.manage().deleteCookieNamed("cookie\_name"); → Deletes a specific cookie.
* driver.manage().deleteAllCookies(); → Deletes all cookies.

## **7. Getting Page Information**

* String title = driver.getTitle(); → Gets the title of the page.
* String url = driver.getCurrentUrl(); → Gets the current URL.
* String pageSource = driver.getPageSource(); → Gets the page source (HTML).

## **8. Handling Alerts**

* Alert alert = driver.switchTo().alert(); → Switches to an alert.
* alert.accept(); → Clicks "OK" on the alert.
* alert.dismiss(); → Clicks "Cancel" on the alert.
* alert.getText(); → Retrieves the alert message.
* alert.sendKeys("text"); → Enters text into the alert.

## **9. Handling Frames**

* driver.switchTo().frame("frameNameOrIndex"); → Switches to a frame.
* driver.switchTo().parentFrame(); → Switches back to the parent frame.
* driver.switchTo().defaultContent(); → Switches back to the main content.

## **1. What are Locators?**

Locators in Selenium are used to identify elements on a webpage for automation testing.

## **2. Types of Locators in Selenium**

### **1. ID Locator**

* driver.findElement(By.id("elementID"));
* Finds an element using its unique ID attribute.

### **2. Name Locator**

* driver.findElement(By.name("elementName"));
* Locates an element by the name attribute.

### **3. Class Name Locator**

* driver.findElement(By.className("className"));
* Finds elements using the class name.

### **4. Tag Name Locator**

* driver.findElement(By.tagName("tagName"));
* Locates elements by their HTML tag.

### **5. Link\_Text Locator**

* driver.findElement(By.linkText("Exact Link Text"));
* Finds a hyperlink (anchor tag <a>) by its exact text.

### **6. Partial Link Text Locator**

* driver.findElement(By.partialLinkText("Partial Text"));
* Finds a hyperlink using part of the link text.

### **7. CSS Selector Locator**

* driver.findElement(By.cssSelector("cssExpression"));
* Select elements using CSS selectors.
* Example:
  + driver.findElement(By.cssSelector("input[type='text']"));

### **8. XPath Locator**

* driver.findElement(By.xpath("xpathExpression"));
* Select elements using XML path expressions.
* Example:
  + Absolute XPath: //html/body/div[1]/input
  + Relative XPath: //input[@id='username']

## **3. Finding Multiple Elements**

* List<WebElement> elements = driver.findElements(By.tagName("input"));
* Returns a list of all matching elements.

## **1. What is SendKeys?**

sendKeys() is a method in Selenium WebDriver used to simulate typing text into input fields, text areas, or any other editable elements.

## **2. Basic Usage of SendKeys**

WebElement element = driver.findElement(By.id("textbox"));

element.sendKeys("Hello, Selenium!");

* Finds an input element and types "Hello, Selenium!" into it.

## **3. Clearing Input Field Before Typing**

WebElement element = driver.findElement(By.id("textbox"));

element.clear();

element.sendKeys("New Text");

* Clears the existing text before sending new input.

## **4. Using Keyboard Keys with SendKeys (Keys Class)**

Selenium provides the Keys class to simulate keyboard actions like pressing Enter, Tab, Shift, etc.

### **Common Key Actions**

import org.openqa.selenium.Keys;

**Pressing Enter:** element.sendKeys(Keys.ENTER);

**Pressing Tab:** element.sendKeys(Keys.TAB);

**Pressing Backspace:** element.sendKeys(Keys.BACK\_SPACE)

**Pressing Delete:** element.sendKeys(Keys.DELETE);

**Pressing Spacebar:** element.sendKeys(Keys.SPACE);

**Pressing Arrow Keys:** element.sendKeys(Keys.ARROW\_DOWN);

element.sendKeys(Keys.ARROW\_UP);

element.sendKeys(Keys.ARROW\_LEFT);

element.sendKeys(Keys.ARROW\_RIGHT);

**Pressing Shift Key Combination:** element.sendKeys(Keys.SHIFT, "text");

**Pressing Control Key Combination:** element.sendKeys(Keys.CONTROL, "a"); // Select All

element.sendKeys(Keys.CONTROL, "c"); // Copy

element.sendKeys(Keys.CONTROL, "v"); // Paste

## **5. Using SendKeys for Multiple Inputs**

WebElement username = driver.findElement(By.id("username"));

WebElement password = driver.findElement(By.id("password"));

username.sendKeys("myUsername");

password.sendKeys("myPassword");

password.sendKeys(Keys.ENTER);

* Fills in a username and password field and submits using Enter.

## **What is a WebElement?**

A WebElement in Selenium represents an element on a web page that can be interacted with using WebDriver methods.

## **Selenium Interaction Methods**

### **1. Click an Element**

### **Method:**

### def click\_element(self, by, value, timeout=2): WebDriverWait(self.driver, timeout).until( EC.element\_to\_be\_clickable((by, value)) ).click()

**Usage**:

self.click\_element(By.LINK\_TEXT, "Transportation")

### **2. Send Text to an Input Field**

**Method:**

def send\_keys(self, by, value, text, timeout=2):

WebDriverWait(self.driver, timeout).until(

EC.visibility\_of\_element\_located((by, value))

).send\_keys(text)

### **3. Select Dropdown Option**

### **Method:**

### def select\_dropdown(self, by, value, option\_text, timeout=2): WebDriverWait(self.driver, timeout).until( EC.visibility\_of\_element\_located((by, value))) dropdown = Select(self.driver.find\_element(by, value)) dropdown.select\_by\_visible\_text(option\_text)

### **4. Hover Over an Element**

**Method:**

def hover\_over\_element(self, by, value, timeout=2):

element = WebDriverWait(self.driver, timeout).until

(EC.visibility\_of\_element\_located((by, value)))

ActionChains(self.driver).move\_to\_element(element).perform()

### **5. Right Click (Context Click)**

### **Method:**

### def right\_click(self, by, value, timeout=2):

### element = WebDriverWait(self.driver, timeout).until(

### EC.visibility\_of\_element\_located((by, value)))

### ActionChains(self.driver).context\_click(element).perform()

### **6. Drag and Drop**

**Method:**

def drag\_and\_drop(self, source\_by, source\_value, target\_by, target\_value, timeout=2):

source = WebDriverWait(self.driver, timeout).until(

EC.visibility\_of\_element\_located((source\_by, source\_value))

)

target = WebDriverWait(self.driver, timeout).until(

EC.visibility\_of\_element\_located((target\_by, target\_value))

)

ActionChains(self.driver).drag\_and\_drop(source, target).perform()

### **7. Handle Alerts**

**Method:**

def handle\_alert(self, action="accept", timeout=2):

alert = WebDriverWait(self.driver, timeout).until(EC.alert\_is\_present())

if action.lower() == "accept":

alert.accept()

elif action.lower() == "dismiss":

alert.dismiss()

# **Waits in Python Selenium**

## **Introduction**

In **Python Selenium**, **waits** are used to handle synchronization between script execution and web element loading. This prevents errors like **NoSuchElementException** caused by elements not being available immediately.

## **Types of Waits in Selenium**

### **1. Implicit Wait**

* Sets a **default waiting time** (in seconds) for all elements.
* If an element is not found immediately, Selenium waits for the specified time before throwing an exception.

#### **Example:**

from selenium import webdriver

driver = webdriver.Chrome()

driver.implicitly\_wait(10) # Wait up to 10 seconds for elements to appear

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

element = driver.find\_element("id", "username") # Will wait if needed

* **Use Case:** Useful when elements take time to load globally across the website.

### **2. Explicit Wait**

* Waits for a specific condition to be **true** before proceeding.
* Uses WebDriverWait and expected\_conditions.

#### **Example:**

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

from selenium import webdriver

driver = webdriver.Chrome()

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

wait = WebDriverWait(driver, 10) # Wait up to 10 seconds

element = wait.until(EC.presence\_of\_element\_located((By.ID, "username")))

element.send\_keys("testuser")

#### **Common Expected Conditions:**

* presence\_of\_element\_located((By.ID, "id"))
* visibility\_of\_element\_located((By.CLASS\_NAME, "classname"))
* element\_to\_be\_clickable((By.XPATH, "//button[text()='Submit']"))
* title\_contains("Dashboard")
* alert\_is\_present()
* **Use Case:** When you need to wait for a specific element’s condition (visibility, clickability, etc.).

### **3. Fluent Wait**

* An advanced version of explicit wait.
* Allows polling at regular intervals while waiting.

#### **Example:**

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

from selenium.webdriver.common.by import By

from selenium import webdriver

driver = webdriver.Chrome()

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

wait = WebDriverWait(driver, timeout=10, poll\_frequency=0.5) # Poll every 0.5 sec

element = wait.until(EC.visibility\_of\_element\_located((By.ID, "username")))

element.send\_keys("testuser")

* **Use Case:** When dealing with elements that load at unpredictable intervals.

## **Choosing the Right Wait Type**

✅ **Implicit Wait** → When all elements have a similar delay.  
 ✅ **Explicit Wait** → When you need to wait for specific elements.  
 ✅ **Fluent Wait** → When elements take an unpredictable time to load.

**Note:** Avoid mixing implicit and explicit waits in the same script to prevent unexpected behavior.

## **Conclusion**

### **Which One Should You Use?**

* Use **implicit wait** only when all elements take roughly the same time to load.
* Use **explicit wait** when dealing with dynamic elements that load at different times.

**Using JavaScript in Selenium WebDriver**

## **Introduction**

Selenium WebDriver provides various methods to interact with web elements, but sometimes the default methods (such as click() and send\_keys()) do not work as expected due to certain webpage structures, overlays, or dynamic content. In such cases, JavaScript execution through Selenium can help overcome these limitations.

## **Why Use JavaScript in Selenium?**

* **Handling Hidden or Invisible Elements**
  + Some elements may be present in the DOM but not visible on the webpage due to CSS properties like display: none or opacity: 0.
  + JavaScript can be used to make them visible or interact with them directly.

driver.execute\_script("arguments[0].style.display='block';", element)

* **Scrolling Elements into View**
  + If an element is outside the viewport, Selenium may not interact with it properly.
  + JavaScript ensures that the element is scrolled into view.

driver.execute\_script("arguments[0].scrollIntoView();", element)

* **Clicking Elements That Are Overlaid or Unresponsive**
  + Some elements may be covered by another element or might not respond to the default click() method.
  + JavaScript can trigger the click event directly.

driver.execute\_script("arguments[0].click();", element)

* **Handling Drop-downs and Select Elements**
  + Some dropdown menus do not work with Selenium's Select class.
  + JavaScript can interact with such elements effectively.

driver.execute\_script("arguments[0].click();", dropdown\_element)

* **Modifying Read-Only or Disabled Fields**
  + Some input fields may be read-only, preventing direct interaction.
  + JavaScript can remove restrictions and allow modifications.

driver.execute\_script("arguments[0].removeAttribute('readonly');", element)

element.send\_keys("New Value")

* **Retrieving Browser Information**
  + JavaScript can fetch details like the page title, URL, or cookies.

title = driver.execute\_script("return document.title;")

url = driver.execute\_script("return document.URL;")

* **Interacting with Shadow DOM Elements**
  + Some web applications use Shadow DOM elements that Selenium cannot access directly.
  + JavaScript helps in selecting such elements.

shadow\_root = driver.execute\_script("return arguments[0].shadowRoot;", shadow\_host\_element)

* **Triggering Hover and Other User Actions**
  + Selenium does not have built-in support for certain user interactions like hover effects.
  + JavaScript can simulate these actions.

driver.execute\_script("arguments[0].dispatchEvent(new Event('mouseenter'))", element)

## **When to Use JavaScript in Selenium?**

* When standard Selenium commands fail to interact with elements.
* When dealing with dynamically loaded content (AJAX-based elements).
* When handling elements hidden by overlays or modals.
* When interacting with disabled fields or elements inside Shadow DOM.
* When scrolling to elements that are out of view.
* When performing actions like hovering that Selenium does not directly support.

**Handling Forms with iframes in Selenium**

### **1. Inspect the Page for iframes (Manually)**

Before automating a form, check if it contains iframes:

* Open the webpage in **Google Chrome**.
* Right-click on the form field (e.g., input box) → **Inspect**.
* Look at the **HTML structure** to see if the field is inside an <iframe>.

### **2. Use Selenium to Detect iframes (Dynamically)**

def switch\_to\_iframe(self, element\_id):

*""" Switches to the correct iframe containing the specified element."""*

driver = self.driver

driver.switch\_to.default\_content()

for iframe in driver.find\_elements(By.TAG\_NAME, "iframe"):

driver.switch\_to.frame(iframe)

if driver.find\_elements(By.ID, element\_id):

return True

print(f"Switched to iframe containing {element\_id}")

driver.switch\_to.default\_content()

print(f"Unable to locate {element\_id} in any iframe!")

return False

### The selected code defines a method called `switch\_to\_iframe` within the `ItemMasterTest` class. This method is designed to switch the WebDriver's focus to the correct iframe that contains a specified element. Here's a breakdown of how it works:

### 1. The method takes an `element\_id` as a parameter, which is the ID of the element we're looking for within an iframe.

### 2. It starts by switching to the default content of the page using `driver.switch\_to.default\_content()`. This ensures we're starting from the top-level document.

### 3. The method then iterates through all iframe elements on the page using `driver.find\_elements(By.TAG\_NAME, "iframe")`.

### 4. For each iframe:

### - It switches the driver's focus to that iframe using `driver.switch\_to.frame(iframe)`.

### - It checks if the element with the specified `element\_id` exists within this iframe using `driver.find\_elements(By.ID, element\_id)`.

### - If the element is found, the method returns `True`, indicating success.

### - If the element is not found, it switches back to the default content using `driver.switch\_to.default\_content()` and continues to the next iframe.

### 5. If the element is not found in any iframe, the method returns `False`.

### **📌 driver.switch\_to.default\_content() in Selenium**

#### **What It Does:**

* **Switches the WebDriver's focus back to the main page** after interacting with an iframe.
* Ensures that Selenium can locate elements **outside the iframe** after performing actions inside one.

#### **Why It’s Needed:**

* If you switch to an iframe using driver.switch\_to.frame("iframe\_id"), Selenium **remains inside** that iframe.
* Any attempt to find an element **outside the iframe** without switching back will **result in an error**.
* driver.switch\_to.default\_content() brings Selenium **back to the main HTML document**.

**Commonly Used expected\_conditions (EC) in Selenium WebDriver**

expected\_conditions (often abbreviated as EC) is a module in **Selenium WebDriver** that provides a set of predefined conditions that can be used for **explicit waits**.

### **Usage:**

EC is used with **WebDriverWait** to pause execution until a certain condition is met before proceeding. This helps handle dynamic elements that take time to load

**presence\_of\_element\_located** – Waits until the element is present in the DOM.  
WebDriverWait(driver,10).until(EC.presence\_of\_element\_located((By.ID, "element\_id")))

**visibility\_of\_element\_located** – Waits until the element is visible on the page.  
WebDriverWait(driver,10).until(EC.visibility\_of\_element\_located((By.XPATH, "//button[@id='submit']")))

**element\_to\_be\_clickable** – Waits until the element is clickable.  
WebDriverWait(driver,10).until(EC.element\_to\_be\_clickable((By.CSS\_SELECTOR, "button.submit")))

**text\_to\_be\_present\_in\_element** – Waits until the given text appears in an element.  
WebDriverWait(driver,10).until(EC.text\_to\_be\_present\_in\_element((By.CLASS\_NAME, "status"), "Success"))

**alert\_is\_present** – Waits until an alert is present.  
WebDriverWait(driver, 10).until(EC.alert\_is\_present())

**staleness\_of** – Waits until the element is removed from the DOM.  
element = driver.find\_element(By.ID, "old\_element")

WebDriverWait(driver, 10).until(EC.staleness\_of(element))

### **Example Import:**

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

### **Which One to Use?**

* **For clicking** → element\_to\_be\_clickable
* **For ensuring element exists** → presence\_of\_element\_located
* **For ensuring visibility** → visibility\_of\_element\_located
* **For waiting for disappearance** → invisibility\_of\_element\_located
* **For handling iframes** → frame\_to\_be\_available\_and\_switch\_to\_it
* **For handling alerts** → alert\_is\_present
* **For ensuring text is updated** → text\_to\_be\_present\_in\_element
* **For verifying selection changes** → element\_located\_to\_be\_selected / element\_to\_be\_selected

**📌 Explanation of add\_commodities\_from\_excel Method**

This function reads an **Excel file (test\_data.xlsx)** and adds multiple commodities dynamically to a web form using Selenium WebDriver.

## **🛠️ Step-by-Step Breakdown**

### **1️⃣ Load Test Data from Excel**

data = pd.read\_excel("test\_data.xlsx") # Load test data

* Reads the Excel file.
* data.iterrows() will loop through each row, extracting **Commodity Name** and **Commodity Code**.

### **2️⃣ Click the "New Record" Button**

if self.switch\_to\_iframe("btn\_NewRecord"):

WebDriverWait(driver, 5).until(

EC.element\_to\_be\_clickable((By.ID, "btn\_NewRecord"))).click()

print("✅ New Record button clicked successfully")

* **Switches to the iframe** containing the "New Record" button.
* **Waits** until the button is **clickable**, then clicks it.
* This opens the form where commodities can be added.

### **3️⃣ Loop Through Each Row in the Excel File**

for index, row in data.iterrows():

commodity\_name = str(row["Commodity Name"])

commodity\_code = str(row["Commodity Code"])

* Iterates through **each row** of the Excel file.
* Extracts **commodity name** and **commodity code**.

### **4️⃣ Switch to the Form Iframe**

if self.switch\_to\_iframe("MasterName"):

* Ensures the script is inside the correct iframe where the form fields are present.

### **5️⃣ Fill the Commodity Form Fields**

master\_name\_field = WebDriverWait(driver, 5).until(

EC.presence\_of\_element\_located((By.ID, "MasterName")))

master\_name\_field.clear()

master\_name\_field.send\_keys(commodity\_name)

* Finds the **"MasterName" input field** and **waits** until it's present.
* Clears existing text (if any).
* **Enters the commodity name.**

code\_field = driver.find\_element(By.ID, "Code")

code\_field.clear()

code\_field.send\_keys(commodity\_code)

* Finds the **"Code" input field**.
* Clears any pre-filled text.
* **Enters the commodity code.**

### **6️⃣ Click the "Save & New" Button**

driver.find\_element(By.ID, "mysubmitNew").click()

* Clicks the **"Save & New"** button.
* **Saves the current commodity** and refreshes the form for the next entry.

### **7️⃣ Wait for Form to Reset Before Next Entry**

WebDriverWait(driver, 10).until(

EC.text\_to\_be\_present\_in\_element\_value((By.ID, "MasterName"), ""))

* Ensures the **"MasterName" field is empty** before entering the next commodity.
* This prevents **overlapping entries**.

### **8️⃣ Print Confirmation for Each Entry**

print(f"✅ Commodity '{commodity\_name}' (Code: {commodity\_code}) saved successfully!")

* Prints a success message after each commodity is saved.

### **9️⃣ Print Final Success Message**

print("🎉 All commodities added successfully!")

* Confirms **all commodities** from the Excel file have been added.

### 

### **🔹 Understanding the iFrame Switching Verification Update**

#### **📌 Current Code Explanation**

if self.switch\_to\_iframe("btn\_NewRecord"):

print("New Record button clicked successfully")

else:

print("Failed to switch to iframe for btn\_NewRecord")

🔹 This code **checks** if the function switch\_to\_iframe("btn\_NewRecord") returns True.  
🔹 If it does, it **prints** a success message.  
🔹 If it returns False, it **prints** an error message.

**⚠️ Problem:**

* Printing messages **does not stop the test execution** if switching fails.
* Even if switching fails, the script might continue executing the next steps, causing unexpected failures later.

#### **✅ Using Assertion Explanation**

assert self.switch\_to\_iframe("btn\_NewRecord"), "Failed to switch to iframe for btn\_NewRecord"

driver.find\_element(By.ID, "btn\_NewRecord").click()

🔹 **What’s Changed?**

* Instead of printing messages, it **directly asserts** that switch\_to\_iframe("btn\_NewRecord") returns True.
* If the function **fails to switch to the iframe**, the test **immediately fails** with the error "Failed to switch to iframe for btn\_NewRecord".

🔹 **Why This is Better?** ✅ If the assertion fails, the test stops execution **immediately**, preventing further errors.  
✅ Ensures the iframe switch is **successful before interacting** with the "btn\_NewRecord" button.  
✅ Provides a **clear failure reason** in the test report.

### 

### 

**🔹 Navigating Dashboard with a For Loop**

* link\_text is a **variable**, not a predefined function.
* It iterates over the menu\_items list, which contains the names of links.
* The for loop clicks each link using By.LINK\_TEXT and prints a success message.

**Example Code:**

menu\_items = ["Transportation", "Transportation Master »", "Common Masters »", "Item Master"]

for link\_text in menu\_items:

self.click\_element(By.LINK\_TEXT, link\_text)

print(f"{link\_text} link clicked successfully") # Printing link\_text directly