vamsipolineni

Vamsi.bucchanngari@gmail.com

Mastering Selenium with Python

End-to-End Web Automation with Frameworks and Git Integration

Abstract

This document provides an in-depth exploration of **Selenium automation using Python**, offering a complete roadmap for aspiring and professional automation testers. Starting with the fundamentals, it explains what Selenium is, how it interacts with browsers, and why it's one of the most powerful tools in modern test automation.

Readers are introduced to the WebDriver API, **element locating** strategies using XPath, CSS Selectors, and various By methods. It includes hands-on guidance on interacting with web elements using **ActionChains,** handling **pop-ups and alerts,** switching between **windows** and **frames**, and using JavaScript with **execute\_script()** for enhanced control.

The guide then dives into **synchronization techniques**, including implicit, explicit, and fluent waits—ensuring stable and reliable test execution. It also covers **web tables**, **dropdown handling**, and dynamic DOM interactions.

For those interested in structured automation, the document outlines how to build test frameworks using Python's **unittest** and **pytest**, along with powerful concepts like **fixtures**, **parameterization,** and **HTML reporting**. Real-world test examples and reusable fixtures make this section highly practical.

Moreover, a dedicated section on **Git essentials** equips readers with the skills to manage source code, branches, and collaborative workflows effectively. From initialization to pull requests, versioning, and stashing, Git is demystified with commands, use cases, and terminal tips.

Whether you're working on small-scale projects or building enterprise-grade automation suites, this document bridges the gap between theory and practical implementation. By the end, readers will not only be confident in using Selenium with Python but also proficient in managing their code and collaborating using Git.

## What is Selenium ?

Selenium is a powerful tool for controlling web browsers through programs and performing browser automation. It is functional for all browsers, works on all major OS and its scripts are written in various languages i.e., [Python](https://www.geeksforgeeks.org/python-programming-language/), [Java](https://www.geeksforgeeks.org/java/), [C#](https://www.geeksforgeeks.org/csharp-programming-language/), etc, we will be working with Python. Selenium Tutorial covers all topics such as - WebDriver, WebElement, Unit Testing with selenium. This Python Selenium Tutorial covers Selenium from basics to advanced and professional uses.

* **Open Source and Portable**- Selenium is an open-source and portable Web testing Framework.
* **Combination of tools and DSL**– Selenium is a combination of tools and DSL (Domain Specific Language) to carry out various types of tests.
* **Easier to understand and implement**- Selenium commands are categorized in terms of different classes which make it easier to understand and implement.
* **Reduce test execution time**– Selenium supports parallel test execution that reduces the time taken to execute parallel tests.
* **Lesser resources required**– Selenium requires fewer resources when compared to its competitors like UFT, RFT, etc.
* **Supports Multiple Operating Systems**- Android, iOS, Windows, Linux, Mac, Solaris.
* **Supports Multiple Browsers**- Google Chrome, Mozilla Firefox, Internet Explorer, Edge, Opera, Safari, etc.
* **Parallel Test Execution**- It also supports parallel test execution which reduces time and increases the efficiency of tests.

## Locating Strategies of Elements

There are two type of strategies - **Locating Single Elements** and **Locating Multiple Elements.**

Locating Single Elements strategy has eight strategies. Locating Multiple Elements strategy has also same strategies except **By.ID.**

**Locating Single Elements -** The first element with the Locator attribute value matching the location will be returned.

**Locating Multiple Elements -** All the elements with the Locator attribute value matching the location will be returned.

**1. By.ID**

With this strategy, the element with the id attribute value matching the location will be returned. If no element has a matching id attribute, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.ID, "id\_of\_element")*

**2. By.NAME**

With this strategy, the element with the name attribute value matching the location will be returned. If no element has a matching name attribute, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.NAME, "name\_of\_element")*

**3. By.LINK\_TEXT**

With this strategy, the element with the link text value matching the location will be returned. If no element has a matching link text attribute, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.LINK\_TEXT, "Text of Link")*

**4. By.PARTIAL\_LINK\_TEXT**

With this strategy, the element with the partial link text value matching the location will be returned. If no element has a matching partial link text attribute, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.PARTIAL\_LINK\_TEXT, "Text of Link")*

**5. By.TAG\_NAME**

With this strategy, the element with the given tag name will be returned. If no element has a matching tag name, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.TAG\_NAME, "Tag name")*

**6. By.CLASS\_NAME**

With this strategy, the element with the matching class attribute name will be returned. If no element has a matching class attribute name, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.CLASS\_NAME, "class\_of\_element")*

**7. By.CSS\_SELECTOR**

With this strategy, the element with the matching CSS selector will be returned. If no element has a matching CSS selector, a NoSuchElementException will be raised.

***Syntax:*** *driver.find\_element(By.CSS\_SELECTOR, "CSS Selectors")*

**Some other Strategies:**

1. Syntax for locating a web element through CSS - Tag and ID Selector is written as:

*("Tag#Value of id attribute")*

1. Syntax for locating a web element through CSS - Tag and Class Selector is written as:

*("Tag.Value of Class attribute")*

1. Syntax for locating a web element through CSS - Tag and Attribute Selector is written as:

*("Tag[Attribute=value]")*

1. Syntax for locating a web element through CSS - Tag, Class and Attribute Selector is written as:

*("Tag.class[attribute=value]")*

1. Syntax for locating a web element through CSS - Sub-String Matches Technique using ^ operator is written as:

*("Tag[attribute^=prefix of the string]")*

**8. By.XPATH**

With this strategy, the element with pattern of xpath matching the location will be returned. If no element has a matching element attribute, a NoSuchElementException will be raised.

**Syntax:** *driver.find\_element(By.XPATH, "xpath")*

**Some other Strategies:**

1. The syntax for locating elements through XPath Absolute is written as:

*//html/body/tag1[index]/tag2[index]/.../tagN[index]*

1. The syntax for locating elements through XPath- Single Attribute can be written as:

*//<HTML tag>[@attribute\_name='attribute\_value']*

*or*

*//\*[@attribute\_name='attribute\_value']*

**3.** The syntax for locating elements through XPath- Multiple Attribute can be written as:

*//<HTML tag>[@attribute\_name1='attribute\_value1'][@attribute\_name2='attribute\_value2]*

*or*

*//\*[@attribute\_name1='attribute\_value1'][@attribute\_name2='attribute\_value2]*

1. The syntax for locating elements through XPath- Using And can be written as:

*//<HTML tag>[@attribute\_name1='attribute\_value1' and @attribute\_name2='attribute\_value2]*

*or*

*//\*[@attribute\_name1='attribute\_value1' and @attribute\_name2='attribute\_value2]*

1. The syntax for locating elements through XPath- Using Or can be written as:

*//<HTML tag>[@attribute\_name1='attribute\_value1' or @attribute\_name2='attribute\_value2]*

*or*

*//\*[@attribute\_name1='attribute\_value1' or @attribute\_name2='attribute\_value2']*

1. The syntax for locating elements through XPath- Using contains() method can be written as:

*//<HTML tag>[contains(@attribute\_name,'attribute\_value')]*

*or*

*//\*[contains(@attribute\_name,'attribute\_value')]*

1. The syntax for locating elements through XPath- Using starts-with() method can be written as:

*//<HTML tag>[starts-with(@attribute\_name,'attribute\_value')]*

*or*

*//\*[starts-with(@attribute\_name,'attribute\_value')]*

1. The syntax for locating elements through XPath- Using text() method can be written as:

*//\*[text()=' ’]*

1. Using XPath- last() method, we can write the code along with the dynamic XPath location as***:***

*//input[@type='text'])[last()]*

## ActionChains

**ActionChains** is a class in Selenium that allows you to perform advanced user interactions with the web browser, such as mouse movements, clicks, drag-and-drop, keyboard keypresses, and hovering.

**Practice site 1** : [*https://demo.automationtesting.in/Register.html*](https://demo.automationtesting.in/Register.html)

**Practice site 2** *:* [*https://www.saucedemo.com/*](https://www.saucedemo.com/)

**Practice site 3** : [*https://www.globalsqa.com/*](https://www.globalsqa.com/)

**Practice site 4** : [*https://the-internet.herokuapp.com/*](https://the-internet.herokuapp.com/)

| ***Method*** | ***Description*** |
| --- | --- |
| *.click(on\_element)* | Clicks on an element |
| *.double\_click(on\_element)* | Double-clicks on an element |
| *.context\_click(on\_element)* | Right-clicks on an element |
| *.move\_to\_element(to\_element)* | Moves the mouse to the middle of an element (used for hover) |
| *.click\_and\_hold(on\_element)* | Clicks and holds on an element |
| *.release(on\_element)* | Releases the held mouse button |
| *.drag\_and\_drop(source, target)* | Drags an element and drops it onto another |
| *.send\_keys(keys)* | Sends keys to the active element |
| *.perform()* | Executes the actions stored in the chain |

## What are Waits in Selenium?

**Waits** in Selenium are used to **pause the execution** of the script **until a condition is met**, like:

* A button appears
* A page finishes loading
* An element becomes clickable

This helps avoid errors like **"Element not found"** or **"Element not interactable"** when the page is slow.

**1. time.sleep() (Static Wait)**

**Definition**: Pauses the script for a fixed number of seconds, no matter what.

*import time*

*time.sleep(5)* # Waits exactly 5 seconds

* **Not smart** — waits even if the element loads faster.

**2. Implicit Wait**

**Definition**: Tells Selenium to **wait for a certain amount of time** while trying to find any element.

*driver.implicitly\_wait(10)* # Waits up to 10 seconds for all elements

* Doesn’t wait for conditions like visibility or clickability

**3. Explicit Wait (Most powerful)**

**Definition**: Waits **until a specific condition is true**, like element is visible, clickable, or present.

*from selenium.webdriver.support.ui import WebDriverWait*

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

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

*wait = WebDriverWait(driver, 10)*

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

* Waits **only as long as needed**

**4. Fluent Wait**

**Fluent Wait** is a type of **explicit wait** that:

* Waits for a condition (like visibility or clickability),
* Checks the condition at regular **polling intervals** (e.g., every 0.5 seconds),
* **Ignores specific exceptions** while waiting.

*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*

*from selenium.common.exceptions import NoSuchElementException, TimeoutException*

# Fluent Wait setup (10 seconds timeout, polling every 500ms)

*wait = WebDriverWait(driver, 10, poll\_frequency=0.5, ignored\_exceptions=[NoSuchElementException])*

## Selenium element methods in Python:

**1. .click()**

* **What**: Clicks the element like a user would.
* **When**: Use it for buttons, checkboxes, radio buttons, or links.

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

**2. .send\_keys("text")**

* **What**: Types the given text into an input field.
* **When**: Use it to fill forms, search boxes, or login inputs.

*driver.find\_element(By.NAME, "username").send\_keys("admin")*

**3. .clear()**

* **What**: Clears existing text in a text box.
* **When**: Before entering new text, especially if the field is pre-filled.

*search\_box = driver.find\_element(By.ID, "search")*

*search\_box.clear()*

*search\_box.send\_keys("Selenium")*

**4. .text**

* **What**: Returns the visible text inside the element.
* **When**: Use to validate messages, labels, or content on the screen.

*msg = driver.find\_element(By.ID, "message").text*

*print("Displayed message:", msg)*

**5. .get\_attribute("attribute\_name")**

* **What**: Gets the value of a specific HTML attribute (like href, value, placeholder).
* **When**: Use to extract link URLs, form values, etc.

*link = driver.find\_element(By.TAG\_NAME, "a")*

*print(link.get\_attribute("href"))*

**6. .is\_displayed()**

* **What**: Returns True if the element is visible on the page.
* **When**: Use to check if an element (like error or popup) is showing.

*if driver.find\_element(By.ID, "error").is\_displayed():*

*print("Error is shown")*

**7. .is\_enabled()**

* **What**: Returns True if the element is enabled (can be clicked or typed in).
* **When**: Use for buttons or form fields that may be disabled.

*btn = driver.find\_element(By.ID, "submit")*

*print("Button enabled?", btn.is\_enabled())*

**8. .is\_selected()**

* **What**: Returns True if a checkbox or radio button is selected.
* **When**: Use to verify if an option is selected.

*chk = driver.find\_element(By.ID, "agree")*

*if chk.is\_selected():*

*print("Checkbox is selected")*

**9. screenshot("file.png")**

* **What**: Captures a screenshot of **just the element**, not the whole page.
* **When**: Use in visual testing or when debugging UI issues.

*logo = driver.find\_element(By.ID, "site-logo")*

*logo.screenshot("logo.png")*

**10. tag\_name**

* **What**: Returns the tag of the element (like input, a, button).
* **When**: Use to verify or debug the type of element.

*tag = driver.find\_element(By.NAME, "email").tag\_name*

*print("Tag is:", tag)*

**11. value\_of\_css\_property("property")**

* **What**: Gets a CSS property value like color, font-size, etc.
* **When**: Use to validate styles, colors, or design elements.

*color = driver.find\_element(By.ID, "header").value\_of\_css\_property("color")*

*print("Header color is:", color)*

## Selenium Dropdown Methods (Using Select Class)

**1. select\_by\_visible\_text("text")**

**Definition**: Selects the option that has the **exact visible label/text** on the dropdown.

**Where to use**: When you know the **text shown to the user** (e.g., "Male", "India", "Yes").

*select.select\_by\_visible\_text("India")*

**2. select\_by\_value("value")**

**Definition**: Selects an option by the **value attribute** in the HTML code.

**Where to use**: When you're working with dropdowns whose <option> tags have values like this:

*<option value="IN">India</option>*

*<option value="US">USA</option>*

*select.select\_by\_value("IN")*

**3. select\_by\_index(index)**

**Definition**: Selects an option based on its **position** in the dropdown (starting from 0).

**Where to use**: When you don’t know the text/value, but want the **1st, 2nd, or nth option**.

*select.select\_by\_index(2) # Selects 3rd option*

**4. options**

**Definition**: Returns a **list of all <option> elements** inside the dropdown.

***Where to use****: To print or loop through all dropdown items.*

*for opt in select.options:*

*print(opt.text)*

**5. first\_selected\_option**

**Definition**: Returns the **currently selected option** as a WebElement.

**Where to use**: To check or validate which option is selected.

*print(select.first\_selected\_option.text)*

## What is an Alert?

**Definition**: An **alert** is a small **JavaScript popup box** that appears in the browser and requires user action (OK, Cancel, or input).

Selenium needs to **switch to the alert** before interacting with it.

**When to Use:**

Use alert handling when your web app triggers popups like:

* **Simple messages** (e.g., “Form submitted!”)
* **Delete confirmations**
* **Logout confirmations**
* **Prompt inputs** (e.g., “Enter your name”)

| **Method** | **Definition** | **Where to Use** |
| --- | --- | --- |
| driver.switch\_to.alert | Switches the control to the alert box | Before interacting with any alert |
| alert.accept() | Clicks **OK** on the alert | To confirm or proceed |
| alert.dismiss() | Clicks **Cancel** on the alert | To cancel an action |
| alert.text | Gets the text message inside the alert | To validate the alert message |
| alert.send\_keys("text") | Enters input into a prompt alert | When alert has an input box |

## What are Window Handles?

**Definition:** In Selenium, window handles are unique IDs that help you switch between multiple tabs or windows opened by the browser during testing.

Each window or tab opened by the browser gets a unique handle (string ID), which you can use to switch focus.

**When to Use:**

Use window handling when:

* Clicking a link opens a new tab or window
* You want to switch back and forth between parent and child windows
* Automating popups, social logins, or external pages

Important Window Handling Methods

| **Method** | **Description** |
| --- | --- |
| *driver.current\_window\_handle* | Returns the handle (ID) of the current window |
| *driver.window\_handles* | Returns a list of all open window handles |
| *driver.switch\_to.window(handle)* | Switches Selenium's control to the specified window |
| *driver.close()* | Closes the current window |
| *driver.quit()* | Closes all windows and ends session |

**Example:**

**Site to practice**: https://demoqa.com/browser-windows

*from selenium import webdriver*

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

*import time*

*driver = webdriver.Chrome()*

*driver.get("https://demoqa.com/browser-windows")*

*driver.maximize\_window()*

# Get current (parent) window

*parent\_window = driver.current\_window\_handle*

# Click button to open a new tab

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

*time.sleep(2)*

# Get all window handles

*all\_windows = driver.window\_handles*

# Switch to new (child) tab

*for window in all\_windows:*

*if window != parent\_window:*

*driver.switch\_to.window(window)*

*break*

# Perform action in new tab

*print("Child window text:", driver.find\_element(By.ID, "sampleHeading").text)*

*time.sleep(2)*

# Close child tab

*driver.close()*

# Switch back to parent window

*driver.switch\_to.window(parent\_window)*

*print("Switched back to parent window.")*

*driver.quit()*

## What is a Frame?

**Definition:** A frame or iframe is an HTML element that loads another HTML document within the same page. Selenium can’t interact with elements inside a frame unless it switches into that frame.

**When to Use Frames?**

You use frame handling in Selenium when:

* The element you want to interact with is inside a <frame> or <iframe>
* You see a “element not interactable” or “no such element” error even when the XPath looks correct
* You inspect the page and see nested documents (via <iframe> in DevTools)
* *driver.switch\_to.parent\_frame()* doesn’t return a reference to any frame. It Just switches control one level up.

**Selenium Methods for Handling Frames**

| **Method** | **Description** |
| --- | --- |
| *driver.switch\_to.frame(index)* | Switches to a frame by index |
| *driver.switch\_to.frame(name\_or\_id)* | Switches to a frame by name or ID |
| *driver.switch\_to.frame(webelement)* | Switches to a frame using a WebElement |
| *driver.switch\_to.default\_content()* | Switches back to the main page (out of all frames) |
| *driver.switch\_to.parent\_frame()* | Goes one level up from nested frame |

## What is Execute Script?

**Defination:** This method executes **JavaScript code** in the context of the currently loaded web page.It’s useful when Selenium’s standard methods don’t work or are limited.

execute\_script() bridges the gap between **Selenium** and **JavaScript-based web behavior**.

**Syntax:** *driver.execute\_script(script, \*args)*

**When to use:**

**1. Scroll down**

To scroll the page manually when elements are not in view

*driver.execute\_script("window.scrollBy(0, 500);")*

**2. Scroll to element**

Useful when the element is not visible without scrolling.

*element = driver.find\_element(By.ID, "target")*

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

**Why arguments[0] in execute\_script()?**

**When you use:**

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

You're passing the element from Python into the JavaScript code as the **first argument**. In JavaScript, these arguments are accessed using the arguments object.

**How It Works:**

*driver.execute\_script("arguments[0].click();", element1, element2, element3)*

**In the JavaScript string:**

* arguments[0] → refers to element1
* arguments[1] → refers to element2
* arguments[2] → refers to element3

**3. Click an element via JS**

Useful for elements blocked by overlays or not fully loaded.

*button = driver.find\_element(By.ID, "submitBtn")*

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

**4. Set input value**

Same as SendKeys driving input to text field.

*input\_field = driver.find\_element(By.ID, "name")*

*driver.execute\_script("arguments[0].value='Vamsi';", input\_field)*

**5. Get page title, Get Current URL**

To get the Title and URL of the current page.

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

*print(title)*

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

*print("URL:", url)*

**6. Get Element Dimensions**

offsetWidth and offsetHeight return the size of the element.

*element = driver.find\_element(By.ID, "myElement")*

*width = driver.execute\_script("return arguments[0].offsetWidth;", element)*

*height = driver.execute\_script("return arguments[0].offsetHeight;", element)*

**7. Get Inner Text of an Element**

* innerText gives you the **visible text** content.
* You can also use textContent (includes hidden text too).

*element = driver.find\_element(By.XPATH, "//h1")*

*text = driver.execute\_script("return arguments[0].innerText;", element)*

*print("Inner Text:", text)*

## What is a Web Table?

A **WebTable** is an HTML table (<table>) used to display data in **rows** and **columns** format on a webpage. Selenium can be used to read, validate, or interact with the data inside these tables.

It consists of:

* <table> — main container
* <thead> — header row (<th>)
* <tbody> — data rows (<tr> with <td>)
* <tfoot> — optional footer

**XPath Structure**

**Table ID:** table1

| **Element** | **XPath Example** |
| --- | --- |
| Entire table | //table[@id='table1'] |
| Header row | //table[@id='table1']/thead/tr/th |
| All rows | //table[@id='table1']/tbody/tr |
| All cells | //table[@id='table1']/tbody/tr/td |
| Specific cell | //table[@id='table1']/tbody/tr[2]/td[3] |

**Example:**

*name = driver.find\_element(By.XPATH, "//table[@id='table1']/tbody/tr[1]/td[1]").text*

*assert name == "John Doe"*

## Unit Testing in Selenium with unittest

**1. Test Class**

class GoogleSearchTest(unittest.TestCase):

Every test must be inside a **class** that inherits from unittest.TestCase.

This makes it a **test case class**, and all test methods inside it can use testing features like:

* + assert methods (assertEqual, assertIn, etc.)
  + Setup and teardown logic

**2. setUp(self)**

*def setUp(self):*

*self.driver = webdriver.Chrome()*

Called **before each test method** runs.

Used to:

* + Initialize the browser
  + Open the application URL
  + Set timeouts or prepare data

Automatically runs **once per test case** to keep tests isolated.

**3. test\_ methods**

*def test\_title(self):*

*self.assertIn("Google", self.driver.title)*

* All test methods must start with test\_ — otherwise, they won’t run.
* Each method is treated as **one individual test**.
* Inside each, you write actions and **assertions** to verify expected behavior.

**4. tearDown(self)**

*def tearDown(self):*

*self.driver.quit()*

Called **after each test method** finishes (pass or fail).

Used to:

* + Close the browser
  + Delete test data
  + Clean up resources
  + Ensures no leftover state affects the next test.

**5. Assertion Methods**

These are the **heart** of unit tests. They check whether conditions are met and decide whether a test **passes or fails**.

If the assertion fails, the test fails, and the remaining part of that test does **not run**.

**5.1. assertEqual(a, b)**

checks whether two values a and b are **equal** (i.e., a == b).  
If they are equal, the test **passes**. If not, the test **fails** and shows a message.

*self.assertEqual(first, second, msg=None)*

* **first** – Actual result from your code
* **second** – Expected value
* **msg** – Optional custom failure message

*welcome\_message = driver.find\_element(By.ID, "welcome").text*

*self.assertEqual(welcome\_message, "Login Successfull", "Unexpected login failure message.")*

***5.2.* assertNotEqual(a, b)**

Checks that **two values are *not* equal** (a != b).  
If they are **equal**, the test will **fail**.

*self.assertNotEqual(actual, expected, msg=None)*

* actual → the value your function or code returns
* expected → the value that it should *not* be
* msg → optional message shown if the test fails

*welcome\_message = driver.find\_element(By.ID, "welcome").text*

*self.assertNotEqual(welcome\_message, "Login Failed", "Unexpected login failure message.")*

* This passes if the message is **not** "Login Failed".
* Fails if the text **is** "Login Failed".

**5.3. assertFalse()**

Checks whether a given **expression or condition evaluates to False**.

If the condition is anything **truthy** (like True, non-empty strings/lists, or any non-zero value), the test will **fail**.

**Syntax:**

*self.assertFalse(expression, msg=None)*

* **expression** → any condition that should be False
* **msg** (optional) → custom error message on failure

**Example 3: In Selenium – Check Element is Hidden**

*error\_msg = driver.find\_element(By.ID, "error")*

*self.assertFalse(error\_msg.is\_displayed(), "Error message is unexpectedly visible!")*

* Passes if the error message is **not displayed**
* Fails if it's visible (i.e., condition is True)

**5.4. assertTrue()**

Checks whether a given **expression or condition evaluates to True**.

If the condition **is not True** (i.e., it’s False, None, 0, '', or any falsy value), the test **fails**.

**Syntax:**

*self.assertTrue(expression, msg=None)*

* **expression** → any condition that should be True
* **msg** (optional) → custom failure message

**Scenario: Checking if a button is visible on the page**

*login\_button = driver.find\_element(By.ID, "login")*

*self.assertTrue(login\_button.is\_displayed(), "Login button is not visible")*

* This passes if the button is **visible**
* Fails with a message if it's not

**5.5. assertIn(member, container)**

Checks whether a **value (member) is present inside** a **container** like:

* a string
* a list
* a tuple
* dictionary keys
* or any iterable.

If the value **is present**, the test **passes**.  
If it's **not present**, the test **fails**.

**Syntax:**

*self.assertIn(member, container, msg=None)*

* member: The value you're checking for
* container: The string, list, or object that should contain it
* msg (optional): Message shown if the test fails.

*welcome\_message = "Welcome, Vamsi!"*

*self.assertIn("Vamsi", welcome\_message) # Passes*

**5.6. assertNotIn(member, container)**

Checks whether a value (member) **is NOT present** in a container (like a list, string, or dictionary).

If the value **is present**, the test **fails**.  
If the value is **absent**, the test **passes**.

**Syntax:**

*self.assertNotIn(member, container, msg=None)*

* member: the item you **expect not to be there**
* container: the collection or string you're checking
* msg: optional message to show if the test fails

*text = "Login successful"*

*self.assertNotIn("Error", text) # Passes*

**5.7. assertIs(a, b)**

Checks if **a is b** — i.e., they are the **same object in memory**, not just equal in value.

Think of it like comparing **identity**, not just **equality**.

**Syntax:**

*self.assertIs(a, b, msg=None)*

* a, b → two variables you expect to point to the **same object**
* msg (optional) → custom failure message

*element1 = driver.find\_element(By.ID, "submit")*

*element2 = element1*

*self.assertIs(element1, element2) # Passes*

**5.*8.* assertIsNone()**

Asserts that the given **value is None** (literally the Python None object).

If the value **is not None**, the test will **fail**.

**Syntax:**

*self.assertIsNone(value, msg=None)*

* value → the variable or return value you're checking
* msg → optional custom message if the test fails

*try:*

*element = driver.find\_element(By.ID, "not-existing")*

*except:*

*element = None*

*self.assertIsNone(element, "Element unexpectedly found!")*

*This passes if the element wasn't found.  
Fails if element is not None.*

**5.9. assertIsNotNone()**

Asserts that a given value **is *not* None**.

If the value **is None**, the test **fails**.  
If the value **is anything other than None**, the test **passes**.

**Syntax:**

*self.assertIsNotNone(value, msg=None)*

* value → the value you're checking
* msg → optional message to display on failure

*element = driver.find\_element(By.ID, "search")*

*self.assertIsNotNone(element, "Search field not found!") # Passes if element exists*

**6. unittest.main()**

*if \_\_name\_\_ == "\_\_main\_\_":*

*unittest.main()*

This is the **entry point** to run all tests in the script.

It:

* + Automatically finds all methods starting with test\_
  + Runs *setUp() → test\_() → tearDown()* for each test
  + Prints a summary result of all tests

assertEqual(a, b) checks whether two values a and b are **equal** (i.e., a == b).  
If they are equal, the test **passes**. If not, the test **fails** and shows a message.

## What is pytest?

pytest is a **testing framework** for Python that allows you to write simple, scalable, readable, and powerful test cases.

**Basic File Naming Convention**

* Test files must start with test\_ or end with \_test.py
* Test functions must start with test\_

**Example:**

*def add(x, y): return x + y*

*def test\_add(): assert add(2, 3) == 5*

Core pytest Contents

**1. assert Statements**

pytest uses Python's built-in assert, but shows detailed failure output.

*def test\_string():*

*assert "hello" in "hello world"*

**2. Fixtures**

Used to set up and tear down test dependencies like browser drivers or database connections.

*import pytest*

*@pytest.fixture*

*def sample\_list():*

*return [1, 2, 3]*

*def test\_list\_length(sample\_list):*

*assert len(sample\_list) == 3*

**3. Parametrization**

Run the same test with different inputs.

*import pytest*

*@pytest.mark.parametrize("x, y, result", [ (2, 3, 5), (10, 5, 15) ])*

*def test\_add(x, y, result):*

*assert x + y == result*

**3. Command Line Usage**

*pytest* Run all tests in the current folder

*pytest -v* Verbose output

*pytest --html=report.html* Generate HTML report (with plugin)

*pytest -s* Show Print() Output

*pytest -k “testname”* Run matching tests

*pytest -m “markname”* Run marked tests

**4. Skipping & Expected Failures**

*import pytest*

*@pytest.mark.skip(reason="not ready")*

*def test\_temp():*

*assert False*

*@pytest.mark.xfail(reason="bug to be fixed")*

*def test\_fail():*

*assert False*

**5. Fixtures with scope**

*@pytest.fixture(scope="session") # once per session @pytest.fixture(scope="module") # once per module @pytest.fixture(scope="function") # default, runs for every test*

**Test Example:**

*import pytest*

*from selenium import webdriver*

*@pytest.fixture def driver():*

*driver = webdriver.Chrome()*

*yield driver*

*driver.quit()*

*def test\_google(driver):*

*driver.get("*[*https://www.google.com*](https://www.google.com/)*") assert "Google" in driver.title*

## What is pytest.ini?

pytest.ini is a configuration file that lets you:

* Define custom marks
* Set default command-line options
* Configure logging, test paths, and test naming rules

So you don’t have to type long CLI commands every time — it makes test execution cleaner and repeatable.

**Example:**

[pytest]

markers =

smoke: Smoke tests

regression: Regression tests

login: Login-related tests

addopts = -v -s --html=Reports/report.html --maxfail=1

testpaths = testscripts

python\_files = test\_\*.py

python\_classes = Test\*

python\_functions = test\_\*

| **Setting** | **Purpose** |
| --- | --- |
| markers | Declare custom markers you’ll use in tests (@pytest.mark.smoke) |
| addopts | Default command-line flags (e.g., -v, -s, generate HTML report, stop after 1 fail) |
| testpaths | Tells pytest **where to look for test files** |
| python\_files | Pattern for valid test file names (test\_\*.py) |
| python\_classes | Classes starting with Test will be collected |
| python\_functions | Functions starting with test\_ will be treated as tests |

## What is logging?

The logging module helps you:

* Record **debug info**, **test steps**, **errors**, and **exceptions**
* Track what's happening **inside your automation**
* Avoid using print() — more control & flexibility

**Levels:**

**DEBUG***:* Detailed debug info, used by developers for tracing bugs.

*Example:* Element X locator returned multiple matches.

**INFO***:* General information about the execution, such as steps or confirmations.

*Example:* Navigated to login page, entered username.

**WARNING:** Something unexpected happened, but the program continues.

*Example:* Page took longer than expected to load.

**ERROR**: A serious issue that caused a test or function to fail.

*Example:* Login failed, incorrect credentials.

**CRITICAL**: A fatal error, often related to system crash or failure to continue.

*Example:* Browser crashed, unable to continue tests

logging.basicConfig(

filename="Reports/test\_log.log",

level=logging.INFO,

format="%(asctime)s - %(levelname)s - %(message)s"

)

**1. DEBUG: Internal diagnostic info**

*logging.debug("Locator for search box returned multiple elements")*

Use when tracing internal Selenium behavior or debugging test data.

**2. INFO: General progress**

*logging.info("Navigated to Myntra login page")*

*logging.info("Entered phone number and clicked continue")*

Use this for step-by-step logging of expected actions.

**3. WARNING: Non-blocking issue**

*logging.warning("Search suggestions not loaded within 5 seconds")*

Use when a delay, deprecation, or minor issue occurs — but test continues.

**4. ERROR: Test failure due to issue**

*try:*

*driver.find\_element(By.XPATH, "//button[@id='checkout']").click()*

*except Exception as e:*

*logging.error(f"Checkout button not clickable: {str(e)}")*

Use in try-except blocks when a test can't proceed as expected.

**5. CRITICAL: System-wide failure**

*try:*

*driver = webdriver.Chrome()*

*except Exception as e:*

*logging.critical(f"Browser failed to launch: {str(e)}")*

*raise SystemExit("Test aborted due to fatal browser issue")*

Use when the whole test suite should stop — e.g., browser not available.

**Example:**

*from utilities.util import Utils*

*from bin.pages.locators import \**

*import pytest*

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

*from selenium.webdriver import ActionChains*

*import time*

*from selenium.webdriver.support.ui import WebDriverWait*

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

*config\_data = Utils().read\_json\_file\_data()*

*class TestAction:*

*@pytest.fixture*

*def my\_fixture(self):*

*self.driver = Utils().get\_driver\_object()*

*self.driver.get(config\_data["myntra\_url"])*

*self.driver.maximize\_window()*

*yield self.driver*

*self.driver.quit()*

*@pytest.mark.test1*

*def test\_myntra\_navbar\_item\_check(self, my\_fixture):*

*# my\_fixture.get(config\_data["myntra\_url"])*

*nav\_bar\_path = my\_fixture.find\_elements(By.XPATH, nav\_bar\_items)*

*action = ActionChains(my\_fixture)*

*for nav in nav\_bar\_path:*

*action.move\_to\_element(nav).perform()*

*time.sleep(2)*

*@pytest.mark.test2*

*def test\_myntra\_searchbox\_check(self, my\_fixture):*

*text\_field = my\_fixture.find\_element(By.XPATH, search\_box)*

*text\_field.send\_keys("shoe")*

*text\_icon = my\_fixture.find\_element(By.XPATH, search\_icon)*

*text\_icon.click()*

*time.sleep(2)*

*my\_fixture.back()*

*time.sleep(2)*

*@pytest.mark.test3*

*def test\_myntra\_profile\_wishlist\_bag\_check(self, my\_fixture):*

*# wait = WebDriverWait(my\_fixture, 10)*

*# profile\_ele = wait.until(EC.visibility\_of\_element\_located((By.XPATH, profile)))*

*profile\_ele = my\_fixture.find\_element(By.XPATH, profile)*

*ActionChains(my\_fixture).move\_to\_element(profile\_ele).perform()*

*time.sleep(2)*

*# wishlist\_ele = wait.until(EC.element\_to\_be\_clickable((By.XPATH, wishlist)))*

*wishlist\_ele = my\_fixture.find\_element(By.XPATH, wishlist)*

*wishlist\_ele.click()*

*time.sleep(2)*

*my\_fixture.back()*

*time.sleep(2)*

*# bag\_ele = wait.until(EC.element\_to\_be\_clickable((By.XPATH, bag)))*

*bag\_ele = my\_fixture.find\_element(By.XPATH, bag)*

*bag\_ele.click()*

*time.sleep(2)*

*my\_fixture.back()*

*time.sleep(2)*

*@pytest.mark.test4*

*def test\_myntra\_topwear\_items(self, my\_fixture):*

*men\_items = my\_fixture.find\_element(By.XPATH, men)*

*ActionChains(my\_fixture).move\_to\_element(men\_items).perform()*

*time.sleep(2)*

*men\_topwear = my\_fixture.find\_elements(By.XPATH, men\_topwear\_items)*

*print(len(men\_topwear))*

*for li in men\_topwear:*

*ActionChains(my\_fixture).move\_to\_element(li).perform()*

*time.sleep(1)*

*print(li.text)*

*time.sleep(2)*

## Git Essentials (With Explanations & Use Cases)

**1. Git Initialization & Configuration**

**1.1.** git init

**Explanation**: Initializes a new Git repository in your current folder.  
**Use Case**: You’re starting a new local project and want to begin tracking changes.

**1.2.** git config --global user.name "Vamsi"

**Explanation**: Sets your name for all commits on your system.  
**Use Case**: Identify yourself across all repositories.

**1.3.** git config --global user.email "you@example.com"

**Explanation**: Sets the email for Git commits.  
**Use Case**: Required for tracking your commit authorship on GitHub/GitLab.

**2. Basic Workflow Commands**

**2.1.** git clone <url>

**Explanation**: Clones an existing remote repository.  
**Use Case**: You want to work on an existing GitHub project.

**2.2.** git status

**Explanation**: Shows the state of the working directory and staging area.  
**Use Case**: See which files are changed, staged, or untracked.

**2.3.** git add <file> or git add .

**Explanation**: Stages changes to be committed.  
**Use Case**: You edited a file and want to prepare it for committing.

**2.4.** git commit -m "message"

**Explanation**: Saves staged changes with a message.  
**Use Case**: You’ve completed a feature or fix and want to create a snapshot.

**2.5.** git log

**Explanation**: Shows commit history.  
**Use Case**: You want to see what changes have been made over time.

**3. Branching & Merging**

**3.1.** git branch <branch\_name>

**Explanation**: Creates a new branch.  
**Use Case**: You're starting a new feature and don’t want to affect the main code.

**3.2**. git checkout <branch> or git switch <branch>

**Explanation**: Switches to a branch.  
**Use Case**: Move between features or bug fixes.

**3.3.** git merge <branch>

**Explanation**: Combines another branch into the current one.  
**Use Case**: You finished a feature and want to merge it into main.

**3.4.** git branch -d <branch>

**Explanation**: Deletes a branch.  
**Use Case**: After a feature branch is merged, you clean it up.

**4. Remote Repositories**

**4.1.** git remote add origin <url>

**Explanation**: Adds a remote repository.  
**Use Case**: Connect your local repo to GitHub.

**4.2.** git push origin <branch>

**Explanation**: Pushes commits to remote.  
**Use Case**: Make your changes visible to team members or online.

**4.3.** git pull

**Explanation**: Fetch and merge changes from the remote repository.  
**Use Case**: Update your local branch with latest remote changes.

**4.4.** git fetch

**Explanation**: Downloads changes from remote but doesn’t merge.  
**Use Case**: You want to review remote changes before merging.

**5. Undoing Things**

**5.1.** git reset <file>

**Explanation**: Removes file from staging area.  
**Use Case**: You staged a file by mistake.

**5.2.** git reset --soft HEAD~1

**Explanation**: Undo the last commit but keep changes staged.  
**Use Case**: You forgot to edit your last commit message.

**5.3.** git reset --hard HEAD~1

**Explanation**: Deletes last commit and discards changes.  
**Use Case**: Dangerous! Use when you want to erase last commit completely.

**5.4.** git checkout -- <file>

**Explanation**: Reverts file back to last committed state.  
**Use Case**: You want to discard local changes in a file.

**6. .gitignore**

**6.1.** .gitignore file

**Explanation**: Tells Git which files/folders to ignore.  
**Use Case**: Avoid committing log files, node\_modules/, or .env.

*\*.log*

*\_\_pycache\_\_/*

*.env*

**7. Stashing (Work in Progress)**

**7.1.** git stash

**Explanation**: Temporarily saves changes you don't want to commit yet.  
**Use Case**: You need to switch branches but don’t want to lose work.

**7.2.** git stash pop

**Explanation**: Reapplies stashed changes and removes them from stash.  
**Use Case**: Continue where you left off after switching branches.

**8. Tagging Versions**

**8.1.** git tag v1.0

**Explanation**: Marks a specific commit as a release.  
**Use Case**: Version your software for deployment or release.

**8.2.** git tag -a v1.0 -m "Release 1.0"

**Explanation**: Annotated tag with a message.  
**Use Case**: Recommended for real release versions.

**9. Rebasing (Advanced)**

**9.1.** git rebase <branch>

**Explanation**: Moves your branch’s base to a new branch.  
**Use Case**: Cleaner history when integrating feature branches.

**9.2.** it rebase -i HEAD~3

**Explanation**: Interactive rebase to edit/squash commits.  
**Use Case**: Clean up multiple small commits into one before pushing.

**10. Clean Up**

**10.1.** git clean -f

**Explanation**: Deletes untracked files.  
**Use Case**: You created temp files not added to Git and want to remove them.

**10.2.** git clean -fd

**Explanation**: Deletes untracked files and folders.  
**Use Case**: Reset repo to a cleaner state.

**11. Search and Inspect**

**11.1** git diff

**Explanation**: Shows changes not yet staged.  
**Use Case**: Review what you’ve edited before staging.

**11.2.** git diff --staged

**Explanation**: Shows staged changes.  
**Use Case**: Double-check before commit.

**11.3.** git show <commit>

**Explanation**: View details about a specific commit.  
**Use Case**: See changes and metadata for a commit.

# Steps to Raise a Pull Request (PR) — Terminal + GitHub

**1. Create a New Branch (Locally)**

Always create a branch for your feature or bug fix:

*git checkout -b feature/login-form*

Make your changes, test them, then:

*git add .*

*git commit -m "Add login form UI and validation"*

**2. Push Your Branch to GitHub**

*git push origin feature/login-form*

This pushes your new branch to the remote GitHub repository.

**3. Raise the PR on GitHub**

Now go to <https://github.com>:

1. Open your repository (e.g., username/repository name)
2. GitHub will show a banner:

"Compare & pull request"

1. Click **"Compare & pull request"**.
2. Review the changes:
   * **Base branch**: usually main or dev
   * **Compare branch**: your feature branch
3. Add a **title** and **description** of your changes.
4. Click **"Create pull request"**

**4. You Want to Link Your Local Folder to a Different Remote Repository**

Maybe your folder is **already linked to one GitHub repo**, but you want to push it to a **different one**.

You can:

* **Remove Old Remote and Add New One:**

*git remote remove origin*

*git remote add origin https://github.com/vamsipolineni/new-repo.git*

*git push -u origin main*

Or, if you're keeping both:

* **Add a Second Remote:**

*git remote add new-origin https://github.com/vamsipolineni/another-repo.git*

*git push -u new-origin main*

This way, you push to either **origin** or **new-origin** manually.