Selenium in Python

&

Allure Report

By Rangika Herath

Table of Contents

Introduction

This project automates Facebook login and signup processes using Selenium WebDriver with Python, organized in a modular structure where the TA_1/ directory contains page object modules (login.py and signup.py) with reusable interaction methods, while the tests/ directory houses pytest test cases (test_login.py and test_signup.py) for executing automated scenarios. The primary objective is to test different credential combinations for login and various input combinations for signup, validating whether each attempt succeeds or fails, with automatic screenshot capture stored in the screenshots/ directory for evidence and debugging purposes. Test results are documented through Allure reporting with JSON results and screenshot attachments stored in allure-results/, enabling comprehensive test visualization and analysis. By combining Selenium WebDriver with pytest framework and Allure reporting, this project demonstrates functional testing of web application authentication and registration modules while reducing repetitive manual testing, improving accuracy, and providing detailed, traceable test documentation.

Objective

1. To automate the process of entering usernames and passwords into the Facebook login form using the login.py page object module in the TA_1/ directory.
2. To validate login attempts by checking whether the provided credentials are valid or invalid through test cases defined in tests/test_login.py.
3. To classify test cases as Pass when valid credentials allow successful login and as Fail when invalid credentials prevent login, with results documented in allure-results/ JSON files.
4. To capture screenshot evidence of test outcomes, storing login success and signup failure images in the screenshots/ directory for verification and debugging purposes.
5. To automate the process of filling out the Facebook signup form, including first name, last name, email, and password fields, using the signup.py page object module.
6. To validate each signup input individually through test cases in tests/test_signup.py, marking valid fields as Pass and invalid fields as Fail to ensure accurate input validation testing.
7. To generate comprehensive test reports using Allure reporting with JSON results and screenshot attachments for detailed test analysis and traceability.

Introduction to Selenium in Python

Why Selenium?

Selenium is an open-source framework used to automate web browsers for various purposes.

- **Open-Source & Cost-Effective**: Selenium is free to use, making it accessible to individuals and organizations of all sizes without licensing costs.
- **Cross-Browser & Cross-Platform Compatibility**: It supports all major browsers (Chrome, Firefox, Edge, Safari) and runs on different operating systems (Windows, macOS, Linux), ensuring consistent application behavior across platforms.
- **Real User Simulation**: Selenium WebDriver, the core component, interacts with web elements in a way that mimics real user actions (clicks, form submissions, navigation), which is ideal for functional and regression testing.

- **Scalability (Selenium Grid)**: Selenium Grid allows for running tests in parallel across multiple machines and environments, which significantly reduces execution time and boosts efficiency for large-scale projects.
- **Integration**: It integrates seamlessly with popular testing frameworks like pytest and unittest and CI/CD tools such as Jenkins, enabling automated testing within modern development pipelines.

Why Python with Selenium?

The Python programming language offers specific advantages that enhance the Selenium experience.

- **Simplicity and Readability**: Python's clean, less verbose syntax allows testers to write automation scripts with fewer lines of code compared to other languages like Java. This leads to faster script development and easier maintenance.
- **Gentle Learning Curve**: Python's straightforward nature makes it an ideal choice for those new to automation testing or programming in general, helping teams transition to automated testing more quickly.
- **Large Ecosystem and Community Support**: Python has a vast ecosystem of libraries (pandas, NumPy, pytest) that complement Selenium, enabling advanced data-driven testing scenarios and robust reporting. A large, active community provides abundant resources and support.
- **Rapid Prototyping**: As an interpreted (scripted) language, Python eliminates the need for compilation, which accelerates the development and execution of test cases.

In summary, the combination of Python's ease of use and Selenium's robust browser automation capabilities creates an efficient, flexible, and scalable solution for web application testing and automation tasks.

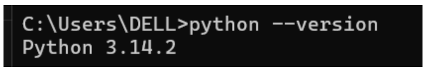
Python Selenium Setup Guide for Windows

1. Install Python

The first step is setting up the core language on your Windows machine.

- Check Windows Version: Determine if your system is 32-bit or 64-bit by right-clicking "This PC" (or "My Computer") and selecting Properties.
- Download & Install: Download the matching Python 3 installer from the official website.
- Crucial Step: During installation, you must check the box "Add Python to PATH".
- Customization: Choose "Customize installation," ensure "Install for all users" is checked, and note the installation path (e.g., C:\Program Files\Python3.14).
- Verification Command: Open the command prompt and type:

```
python –version
```


```
C:\Users\DELL>python --version
Python 3.14.2
```

2. Manual Path Configuration (If Needed)

If you forgot to add Python to the path during installation, you must do it manually to run commands from any directory.

- Environment Variables: Go to Advanced System Settings > Environment Variables.
- Better Approach: Create a new system variable named PYTHON_PATH. Include paths for the root folder, \Scripts, \Lib, \libs, and \DLLs, separated by semicolons.
- Update System Path: Add %PYTHON_PATH% to the existing system Path variable.

3. Install PyCharm IDE

PyCharm serves as the editor where you will write your automation scripts.

- Download: Get the Community Edition, which is free and open-source.
- Installation: Check the option to "Update PATH variable" during the setup.
- Project Creation: When creating a new project, select "Previously configured interpreter" and point it to your System Interpreter (the python.exe you installed in Step 1) rather than a virtual environment.

4. Install Selenium Library

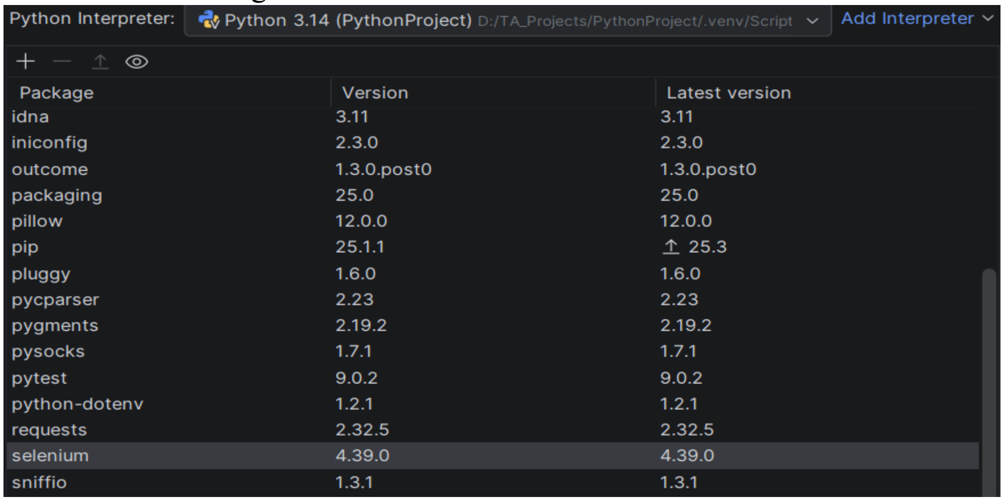
Selenium is installed using pip, the Python package installer that comes bundled with Python.

- Installation Command:
  pip install selenium
- Verification Command:
  pip list

5. Configure Selenium in PyCharm

You must ensure your specific PyCharm project recognizes the Selenium library.

- Check Settings: Go to File > Settings > Project > Python Interpreter.
- Add Library: If Selenium isn't listed, click the "+" icon, search for "selenium," and click "Install Package".


Python Interpreter:
Python 3.14 (PythonProject) D:/TA_Projects/PythonProject/.venv/Script
Add Interpreter

Package	Version	Latest version
idna	3.11	3.11
iniconfig	2.3.0	2.3.0
outcome	1.3.0.post0	1.3.0.post0
packaging	25.0	25.0
pillow	12.0.0	12.0.0
pip	25.1.1	↑ 25.3
pluggy	1.6.0	1.6.0
pycparser	2.23	2.23
pygments	2.19.2	2.19.2
pysocks	1.7.1	1.7.1
pytest	9.0.2	9.0.2
python-dotenv	1.2.1	1.2.1
requests	2.32.5	2.32.5
selenium	4.39.0	4.39.0
sniffio	1.3.1	1.3.1

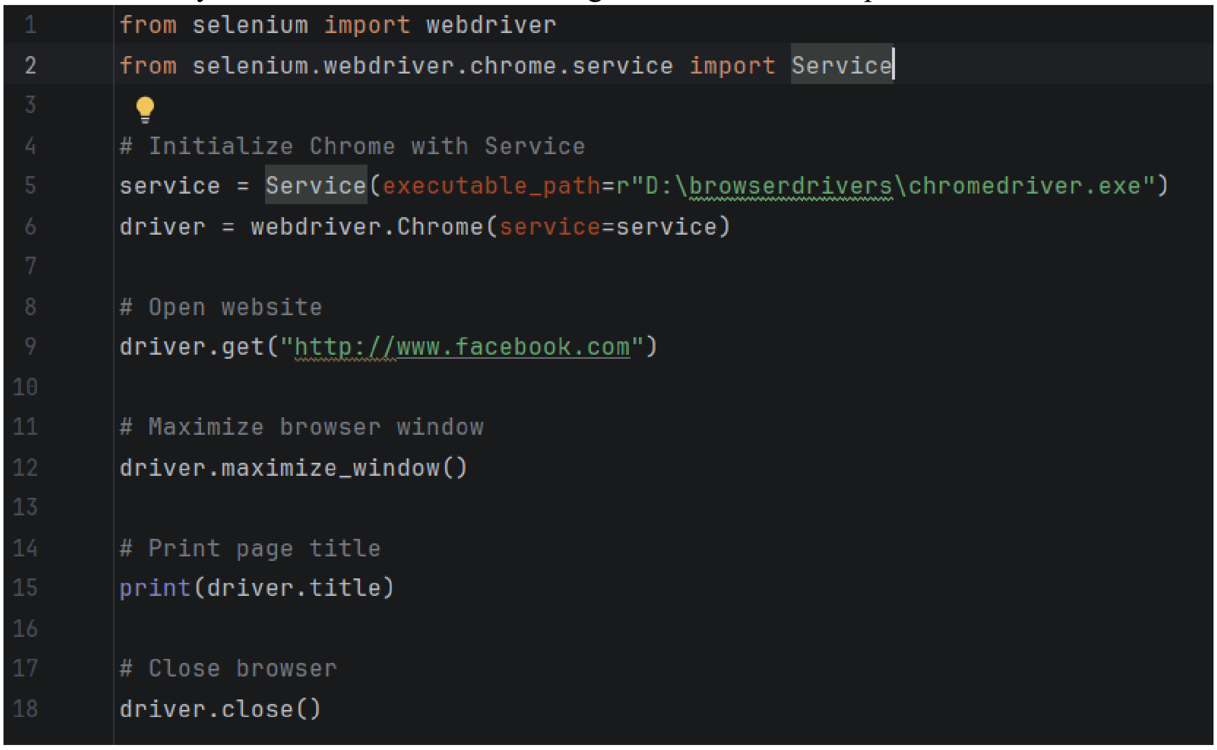
6. Set Up Browser Drivers

Selenium requires a specific driver to communicate with your browser.

- Check Browser Version: In Chrome, go to Help > About Google Chrome to find your version (e.g., Version 88).
- Download Driver: Download the matching chromedriver.exe for your version.
- Organization: Create a simple folder like C:\browser_drivers and extract the .exe there.

7. Write and Run the First Script

Create a new Python file and use the following code to test the setup:

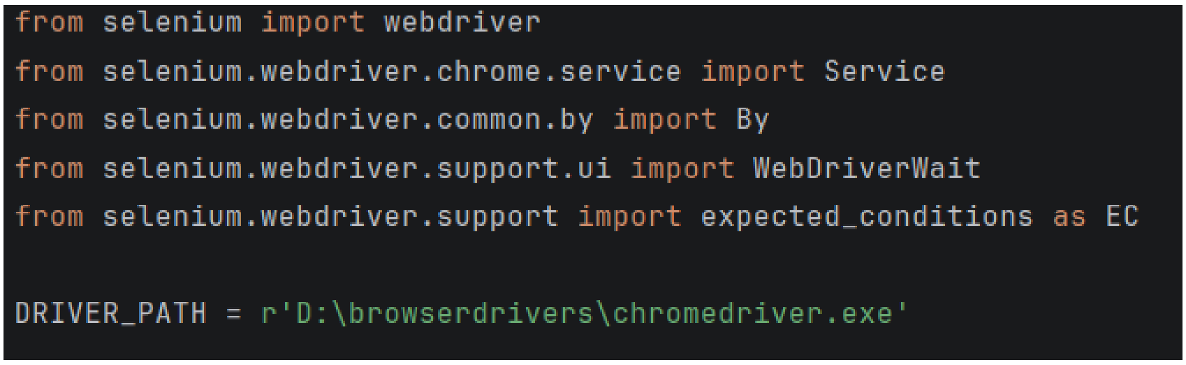


```python
from selenium import webdriver
from selenium.webdriver.chrome.service import Service

# Initialize Chrome with Service
service = Service(executable_path=r"D:\browserdrivers\chromedriver.exe")
driver = webdriver.Chrome(service=service)

# Open website
driver.get("http://www.facebook.com")

# Maximize browser window
driver.maximize_window()

# Print page title
print(driver.title)

# Close browser
driver.close()
```

Explanation of login.py and signup.py

1. Imports and Constants

Both files use the modern Selenium 4 approach with Service for driver management.



```python
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

DRIVER_PATH = r'D:\browserdrivers\chromedriver.exe'
```

Import	Purpose
os	File/directory operations for screenshots
datetime	Timestamp generation for filenames
Service	Modern Selenium 4 driver management
By	Locator strategies (NAME, ID, XPATH, CSS_SELECTOR)
WebDriverWait	Explicit wait mechanism
expected_conditions (EC)	Predefined wait conditions
TimeoutException	Handle element not found within timeout

2. Screenshot Helper Function

Captures screenshots with timestamp for debugging

```
def _save_screenshot(driver, prefix, success):
    screenshots_dir = os.path.join(os.getcwd(), "screenshots")
    os.makedirs(screenshots_dir, exist_ok=True)
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    status = "success" if success else "failure"
    filename = f"{prefix}_{status}_{ts}.png"
    driver.save_screenshot(os.path.join(screenshots_dir, filename))
```

3. Driver Initialization Pattern

Allows reusing an existing driver or creating a new one.

This allows two usage modes:
Standalone: Creates its own driver, quits it when done
Shared: Receives an existing driver, leaves it open for other tests

```
def run_function(driver=None, driver_path=DRIVER_PATH):
    local_driver = None
    if driver is None:
        s = Service(driver_path)
        local_driver = webdriver.Chrome(service=s)
        driver = local_driver
```

4. WebDriverWait for Element Handling

Both use explicit waits instead of time.sleep() for reliability.

```
wait = WebDriverWait(driver, timeout)
element = wait.until(EC.presence_of_element_located((By.NAME, "email")))
```

5. **Cleanup Pattern**

Ensures screenshots are taken and driver is closed only if locally created.

```
finally:
    _save_screenshot(driver, "prefix", success)
    if local_driver:
        local_driver.quit()
```

6. **Login-Specific Logic (login.py)**

Success Detection: URL changes OR login form disappears.

```
email = wait.until(EC.presence_of_element_located((By.NAME, "email")))
password = wait.until(EC.presence_of_element_located((By.NAME, "pass")))
login_btn = wait.until(EC.element_to_be_clickable((By.NAME, "login")))

email.clear()
email.send_keys(email_value)
password.clear()
password.send_keys(password_value)

before_url = driver.current_url
login_btn.click()

# Verify success by checking URL change or element disappearance
WebDriverWait(driver, timeout).until(
    EC.any_of(
        EC.url_changes(before_url),
        EC.invisibility_of_element_located((By.NAME, "email"))
```

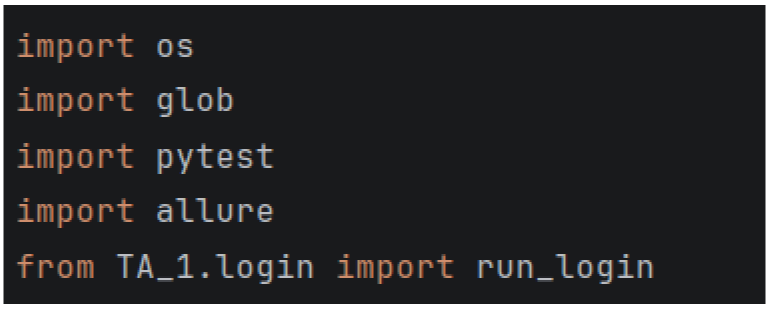
7. **Signup-Specific Logic (signup.py)**

Opening Registration Form (Multiple Locator Attempts):

```
for locator in [
    (By.CSS_SELECTOR, "a[data-testid='open-registration-form-button']"),
    (By.XPATH, "//*[text()='Create New Account']"),
    (By.XPATH, "//a[contains(., 'Create New Account')]")
]:
    try:
        btn = wait.until(EC.element_to_be_clickable(locator))
        btn.click()
        break
    except TimeoutException:
        continue
```

Explanation of login.py and signup.py

1. **Imports Section**

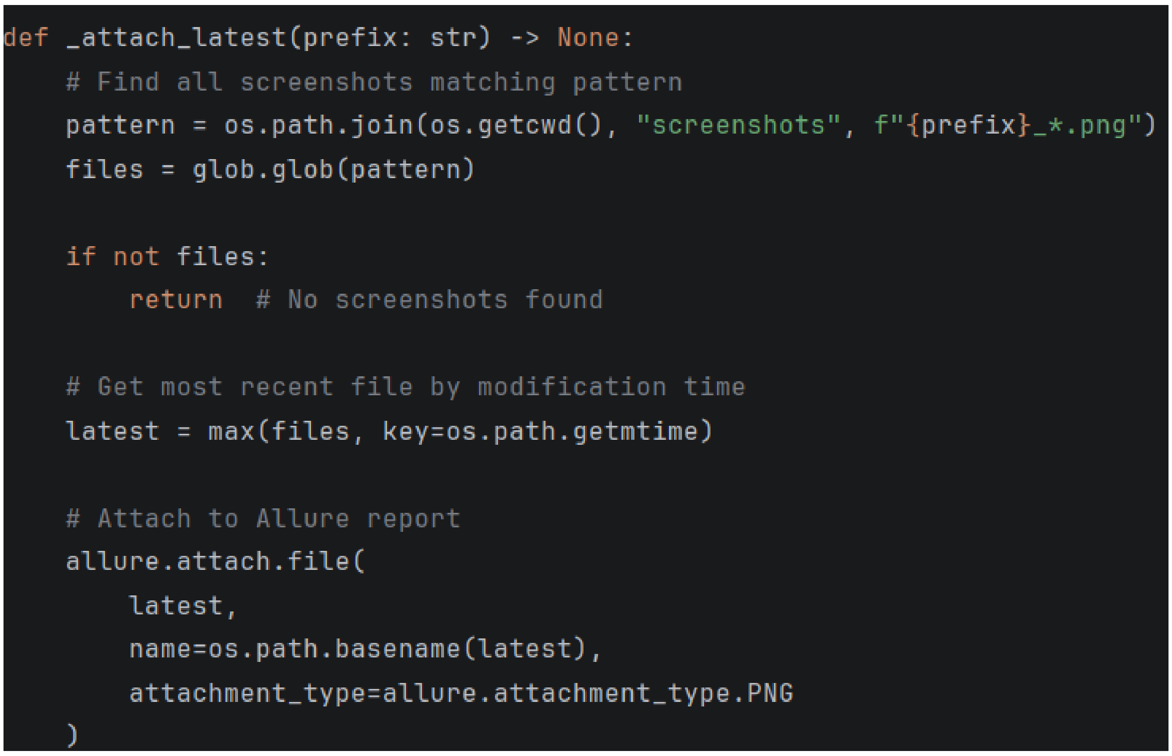


```
import os
import glob
import pytest
import allure
from TA_1.login import run_login
```

Import	Purpose
os	File path operations
glob	Pattern matching for screenshot files
pytest	Test framework (implicit usage)
allure	Test reporting with attachments
run_login/run_signup	Functions being tested

2. **Screenshot Attachment Helper**

Searches for files like screenshots/login_success_20250115_143052.png
Finds the newest one
Embeds it in the Allure HTML report



```
def _attach_latest(prefix: str) -> None:
    # Find all screenshots matching pattern
    pattern = os.path.join(os.getcwd(), "screenshots", f"{prefix}_*.png")
    files = glob.glob(pattern)

    if not files:
        return  # No screenshots found

    # Get most recent file by modification time
    latest = max(files, key=os.path.getmtime)

    # Attach to Allure report
    allure.attach.file(
        latest,
        name=os.path.basename(latest),
        attachment_type=allure.attachment_type.PNG
    )
```

3. Test Structure Pattern

Positive Test (Success Expected):	Negative Test (Failure Expected):
`def test_login_valid():` `ok = run_login("bob.alice069@gmail.com", "Secret@123", timeout=15)` `_attach_latest("login")` `assert ok  # Expects True`	`def test_login_invalid():` `ok = run_login("invalid.user@example.com", "wrongpassword", timeout=15)` `_attach_latest("login")` `assert not ok  # Expects False`

```
def test_login_valid():
    ok = run_login("bob.alice069@gmail.com", "Secret@123", timeout=15)
    _attach_latest("login")
    assert ok  # Expects True
```

```
def test_login_invalid():
    ok = run_login("invalid.user@example.com", "wrongpassword", timeout=15)
    _attach_latest("login")
    assert not ok  # Expects False
```

Challenges faced

During the execution of this project, several challenges were encountered that impacted test reliability. One significant issue was that Facebook frequently serves different versions of its login and signup pages, which caused some tests to fail unexpectedly.

Probable Reasons for Test Failures:
Dynamic element locators - Facebook changes element IDs, class names, or XPath structures across different page versions

Limitation:
Due to time constraints, no solution was implemented to reduce the complexity or handle these dynamic page variations. Potential solutions such as implementing robust waiting strategies, using multiple fallback locators, or detecting the page version before running specific tests were not applied in this project.

Allure Report Setup Guide for Windows

Prerequisites

The project requires three main components:
- Python packages: **selenium**, **pytest**, **allure-pytest**
- ChromeDriver binary matching your Chrome browser version
- OpenJDK

Installation Steps

1.Install Python Dependencies
Open PowerShell and run:

```
pip install selenium pytest allure-pytest
```

2. Configure ChromeDriver
Set the DRIVER_PATH variable in test_data.py to point to your ChromeDriver executable. Use a Windows path format:

```
DRIVER_PATH = "D:\\browserdrivers\\chromedriver.exe"
```

3. Install Allure CLI

Download the latest Allure release from the official repository at https://github.com/allure-framework/allure2/releases. Extract the archive to a permanent location such as C:\Tools\allure.

Add the bin directory to your system PATH:

```
# Add to current session
$env:PATH += ";C:\Tools\allure\bin"

# Add to persistent user PATH
setx PATH "$($env:PATH);C:\Tools\allure\bin"
```

Verify the installation:

```
allure --version
```

Note that Java must be installed on your system. Check with `java -version` and install OpenJDK if needed.

Project Structure

Create the package structure:

```
mkdir -p TA_1
type nul > TA_1\__init__.py
```

Place test files in a `tests` directory at the project root, parallel to the `TA_1` package folder.

Running Tests

Standalone Script Execution

Individual scripts can be run directly:

```
cd TA_1
python login.py
python signup.py
```

Screenshots are saved to the screenshots folder. When running standalone, Allure attachment functions remain dormant.

Running with Pytest

.venv and venv Folders

These folders serve as **Python virtual environments**. A virtual environment is an isolated Python workspace that keeps project dependencies separate from your system-wide Python installation.

```
python -m venv venv
```

Inside you will find:
- `Scripts` (Windows) or `bin` (Unix): Executable files including the Python interpreter and activation scripts
- `Lib\site-packages`: All installed packages specific to this project
- `pyvenv.cfg`: Configuration file linking back to the base Python installation

Activate your virtual environment if needed:

```
.\.venv\Scripts\Activate.ps1
```

Why Virtual Environments Exist

When you work on multiple Python projects, each may require different versions of the same package. Installing everything globally creates conflicts. Virtual environments solve this by giving each project its own independent set of packages.

Install or upgrade pytest:

```
python -m pip install -U pip setuptools wheel
python -m pip install pytest allure-pytest
```

Execute tests and generate Allure results:

```
python -m pytest -q --alluredir=allure-results
```

Test files should import functions like run_login and run_signup, assert the expected boolean results, then locate the most recent screenshot in the screenshots folder and attach it using allure.attach.file.

.allure Folder
This folder appears less commonly and typically relates to Allure configuration or cache data.

Generating Reports

After test execution, generate and view the HTML report:

```
allure generate allure-results
allure open allure-report
```

Troubleshooting

If the allure command fails, verify which executable is being used:

```
Get-Command allure | Format-List -Property *
where.exe allure
```

Check for conflicting packages in your virtual environment:

```
python -m pip show allure
python -m pip list | Select-String allure
```

Remove any incorrect allure packages from your virtual environment Scripts folder. The proper Allure CLI should be the manually installed distribution, not a pip package.

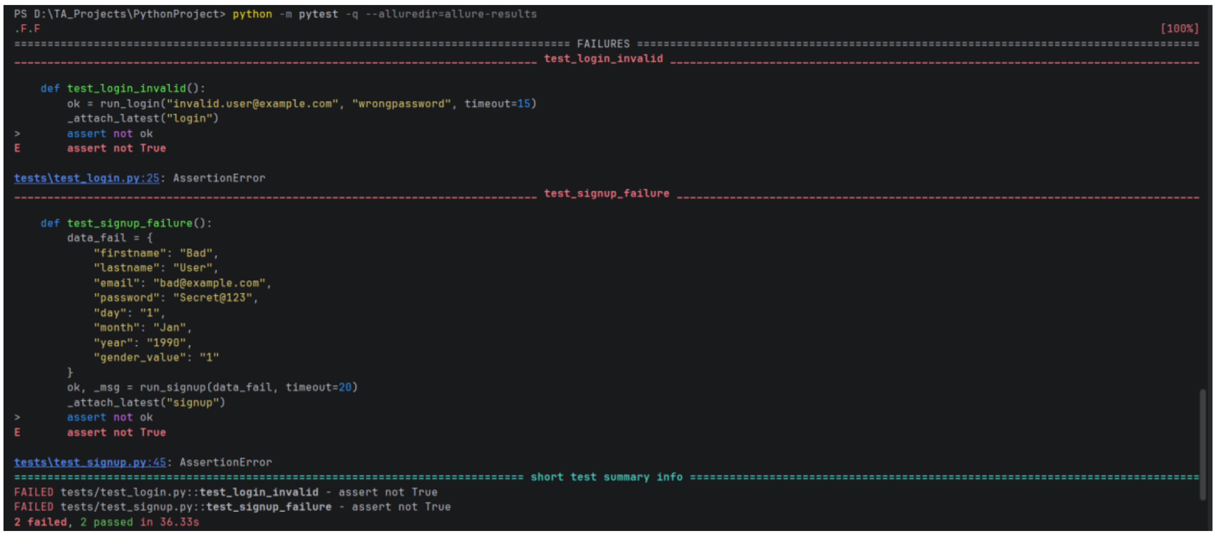
Optional: XAMPP Configuration

If your project includes a web component hosted through XAMPP, copy your site folder into the document root:

```
Copy-Item -Path .\my_local_site -Destination C:\xampp\htdocs\my_local_site -Recurse
Start-Process 'C:\xampp\xampp-control.exe'
Start-Process 'http://localhost/my_local_site/'
```

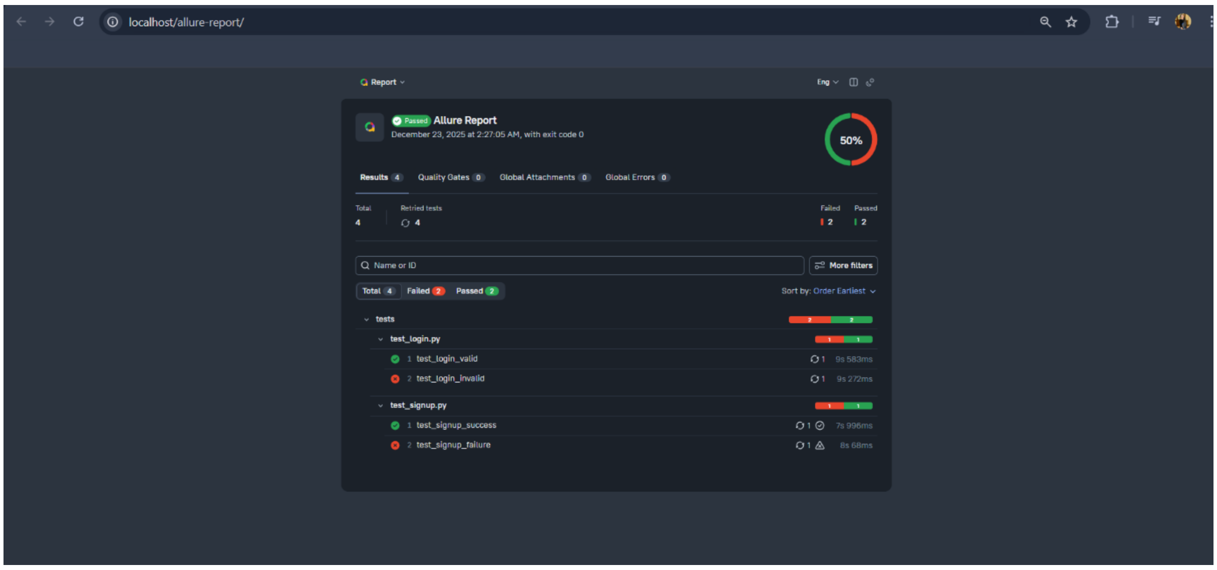
Start Apache and MySQL through the XAMPP Control Panel as needed.

PowerShell terminal output



```
PS D:\TA_Projects\PythonProject> python -m pytest -q --alluredir=allure-results
.F.F                                                                                                                   [100%]
=================================================================================== FAILURES ===================================================================================
_______________________________________________________________________________ test_login_invalid _______________________________________________________________________________

    def test_login_invalid():
        ok = run_login("invalid.user@example.com", "wrongpassword", timeout=15)
        _attach_latest("login")
>       assert not ok
E       assert not True

tests\test_login.py:25: AssertionError
______________________________________________________________________________ test_signup_failure _______________________________________________________________________________

    def test_signup_failure():
        data_fail = {
            "firstname": "Bad",
            "lastname": "User",
            "email": "bad@example.com",
            "password": "Secret@123",
            "day": "1",
            "month": "Jan",
            "year": "1990",
            "gender_value": "1"
        }
        ok, _msg = run_signup(data_fail, timeout=20)
        _attach_latest("signup")
>       assert not ok
E       assert not True

tests\test_signup.py:45: AssertionError
============================================================================ short test summary info =============================================================================
FAILED tests/test_login.py::test_login_invalid - assert not True
FAILED tests/test_signup.py::test_signup_failure - assert not True
2 failed, 2 passed in 36.33s
```

Allure report dashboard



localhost/allure-report/

Report

Passed Allure Report
December 23, 2025 at 2:27:05 AM, with exit code 0

50%

Results 4 | Quality Gates 0 | Global Attachments 0 | Global Errors 0

Total 4 | Retried tests 4 | Failed 2 | Passed 2

Name or ID | More filters

Total 4 | Failed 2 | Passed 2 | Sort by: Order Earliest

```
tests                                   2 | 2
  test_login.py                         1 | 1
    ● 1 test_login_valid          1   9s 583ms
    ● 2 test_login_invalid        1   9s 272ms
  test_signup.py                        1 | 1
    ● 1 test_signup_success       1   7s 996ms
    ● 2 test_signup_failure       1   8s 68ms
```

Conclusion

This project successfully demonstrates the use of Selenium WebDriver for automating and testing Facebook's login and signup functionalities. As shown in the Allure Report, the automation framework executed a total of 4 test cases with a 50% pass rate, where 2 tests passed and 2 tests failed. The login tests included test_login_valid which passed in 9s 583ms and test_login_invalid which failed in 9s 272ms, while the signup tests included test_signup_success which passed in 7s 996ms and test_signup_failure which failed in 8s 68ms. Through this automation approach, the system simulates user interactions, inputs test data, and validates outputs without manual intervention, reducing repetitive manual testing and improving accuracy. The integration with Allure reporting provides clear visibility into test results, enabling quick identification of potential issues in authentication and data validation. This project highlights the importance of automation in web application testing and provides a foundation for extending test coverage to other user interface components in the future.