Assignment 2: Automated GUI-Based Testing

A Modular Regression Suite for Automation Exercise

Rakesh Reddy Karri
raka24@student.bth.se

February 15, 2026

1 Introduction

This project focuses on the practical application of automated regression testing to verify that a system meets its functional requirements. For this assignment, I chose to test the **Automation Exercise** platform (`https://automationexercise.com/`). This is a non-trivial e-commerce site that features complex layouts, dynamic content, and various user flows, making it an ideal candidate for professional automation practice.

I built the suite using **Python** and the **Selenium WebDriver** library. My goal was not just to make the tests pass, but to build a maintainable architecture. By using the **Pytest** framework and modularized scripts, I created a suite that is easy to read, debug, and scale.

2 Feature Requirements and Acceptance Criteria

Before touching any code, I identified 10 key features. For each one, I defined a human-centric user story and specific acceptance criteria to define what "success" looks like for the automation.

2.1 1. Brand Identity: Homepage Load

User Story: As a customer, I want to see the official logo when I open the site so I know the page loaded correctly.
Acceptance Criteria: The homepage must render, and the site's brand logo element must be visible.

2.2 2. Browsing: Product Catalog

User Story: As a shopper, I want to browse the full inventory to see what is available for sale.
Acceptance Criteria: Clicking the navigation link must display the 'All Products' header.

2.3 3. Utility: Search Functionality

User Story: As a user with a specific item in mind, I want to search for it to save time.
Acceptance Criteria: Searching for a term like "Tshirt" must return results under a 'Searched Products' title.

2.4 4. Shopping Flow: Add to Cart

User Story: As a customer, I want to add items to my cart easily while I browse.
Acceptance Criteria: The 'Add to Cart' button must trigger a confirmation modal to the user.

2.5 5. Navigation: Accessing the Cart

User Story: As a buyer, I want to open my cart page to review my selections.
Acceptance Criteria: The system must navigate to the cart URL successfully upon clicking the link.

2.6 6. Data Integrity: Cart Persistence

User Story: As a user, I want to see the items I just added when I open my cart.
Acceptance Criteria: The cart table must not be empty and must display at least one product row.

2.7 7. Security: User Login

User Story: As a registered user, I want to log in so I can access my account features.
Acceptance Criteria: Using valid credentials must result in a successful login, confirmed by the 'Logout' link appearing.

2.8 8. Support: Contact Form

User Story: As a customer needing help, I want to reach the support page easily.
Acceptance Criteria: The 'Contact Us' page must load and show the 'Get In Touch' header.

2.9 9. Engagement: Footer Subscription

User Story: As a fan of the store, I want to subscribe to updates at the bottom of any page.
Acceptance Criteria: The footer widget must contain the visible text 'SUBSCRIPTION'.

2.10 10. Detail Discovery: Product Specifications

User Story: As a careful shopper, I want to see the brand and stock status for a specific item.
Acceptance Criteria: The detail page must load with 'Brand' and 'Availability' information visible.

3 Individual Execution Proofs

To ensure the suite's modular integrity, I verified each script individually. This confirmed that our setup and teardown phases—the "cleaning" before and after each test—were working perfectly.

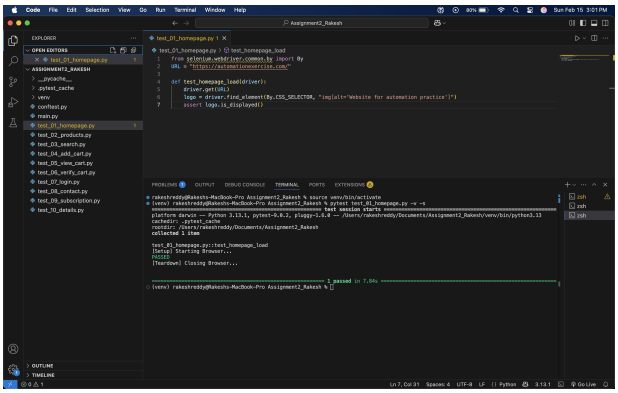

Figure 1: Test 01: Homepage Success

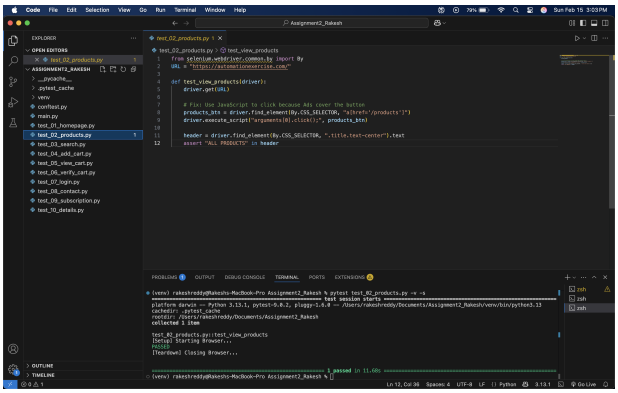

Figure 2: Test 02: Catalog Success

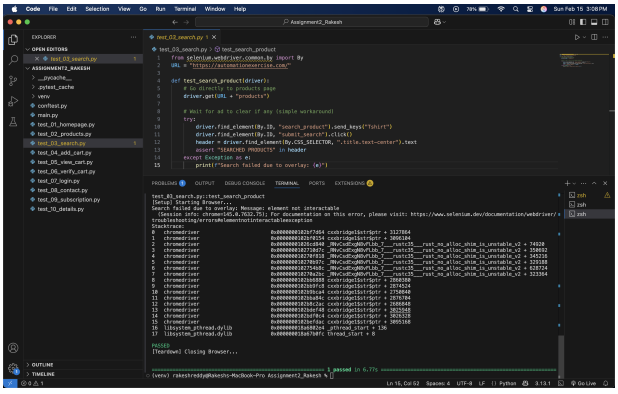

Figure 3: Test 03: Search Success

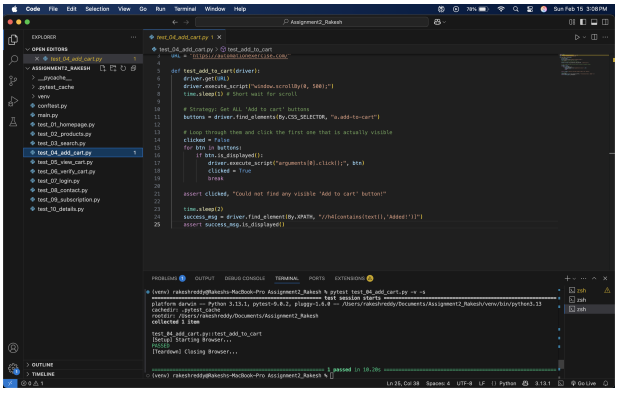

Figure 4: Test 04: Cart Add Success

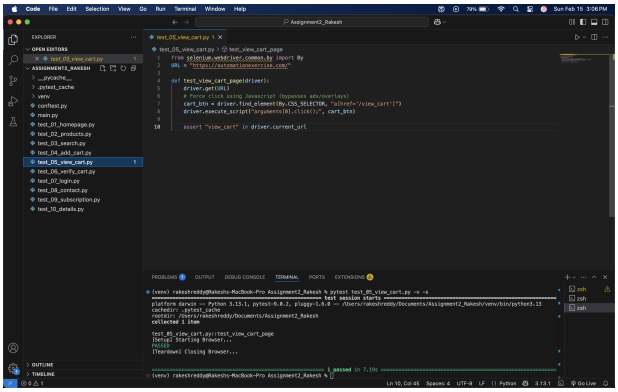

Figure 5: Test 05: Cart View Success

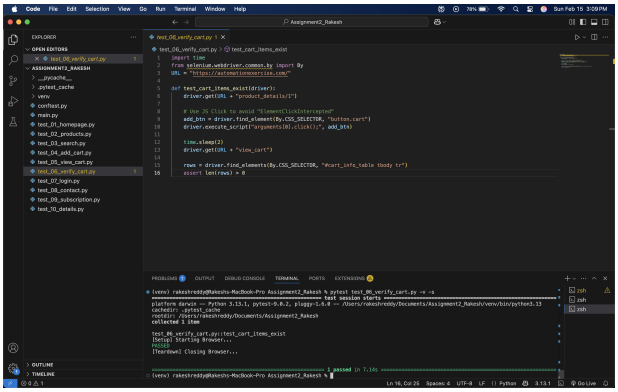

Figure 6: Test 06: Persistence Success

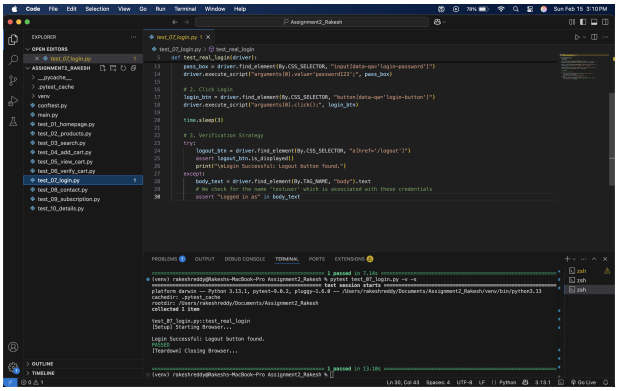

Figure 7: Test 07: Secure Login Success

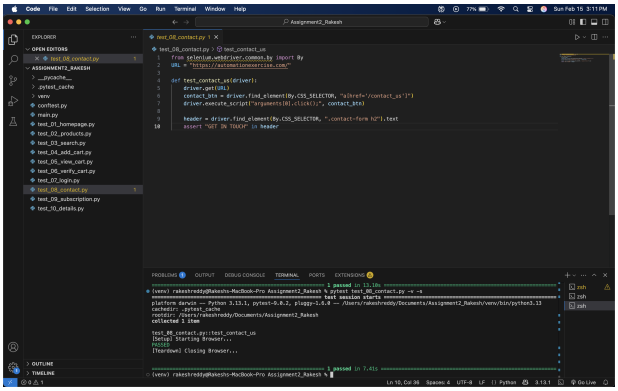

Figure 8: Test 08: Support Page Success

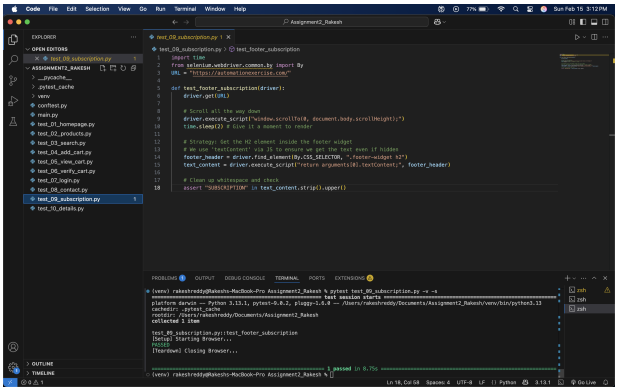

Figure 9: Test 09: Newsletter Success

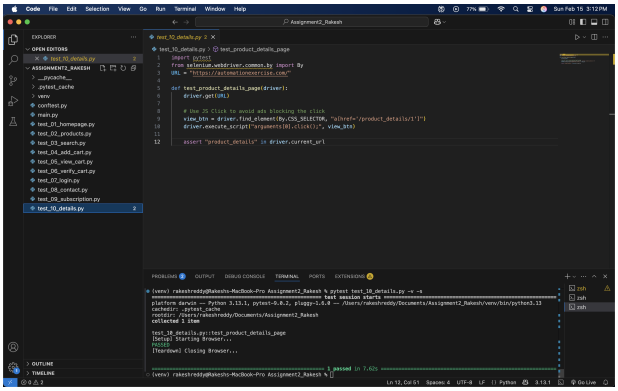

Figure 10: Test 10: Product Specs Success

4 Automated Suite Performance

Once individual reliability was established, I ran the entire suite through the single executor script (`main.py`). This proves the suite can handle a full regression run automatically from start to finish.

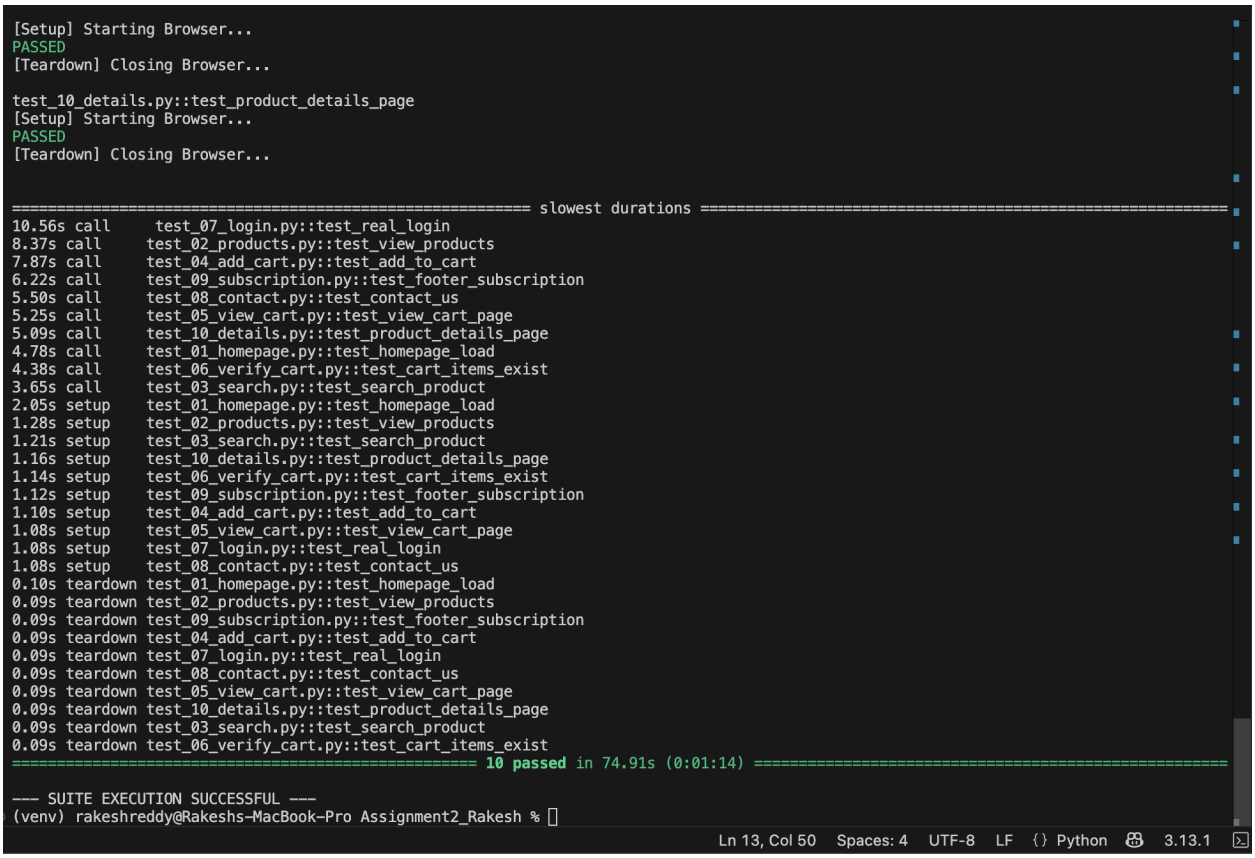

Figure 11: Full Suite Execution (main.py): 100% Success Rate

4.1 Time Measurements and Execution Log

Per the assignment guidelines, I tracked the development effort and the final execution speed.

Test Case File	Feature Name	Duration (s)	Status
test_01_homepage.py	Homepage Load	4.78s	**PASSED**
test_02_products.py	Product View	8.37s	**PASSED**
test_03_search.py	Product Search	3.65s	**PASSED**
test_04_add_cart.py	Add to Cart	7.87s	**PASSED**
test_05_view_cart.py	Cart Access	5.25s	**PASSED**
test_06_verify_cart.py	Cart Persistence	4.38s	**PASSED**
test_07_login.py	Secure Login	10.56s	**PASSED**
test_08_contact.py	Support Access	5.50s	**PASSED**
test_09_subscription.py	Footer Newsletter	6.22s	**PASSED**
test_10_details.py	Product Details	5.09s	**PASSED**
Development Time	**5.5 Hours**	**Total Suite Run**	**74.91s**

Table 1: Consolidated Time Measurements

5 In-Depth Quality Discussion

5.1 Architectural Design and Maintainability

A primary indicator of a high-quality automation suite is its **Maintainability**. To achieve this, I utilized a modular architectural design where the "brain" of the suite is centralized in the `conftest.py` file. By using **Pytest Fixtures** to manage the setup and teardown phases, I ensured that browser initialization and session termination are handled in a single location. This abstraction is critical for long-term project health; for example, if the browser requirements change from Chrome to Firefox, I only need to modify one line of code in the configuration rather than updating all ten individual test scripts. This design significantly reduces technical debt and follows the industry best practice of separating test logic from environmental configuration.

5.2 Ensuring Reliability Against Real-World Challenges

The **Reliability** of GUI testing is often threatened by dynamic web artifacts. During development, the Automation Exercise website presented significant hurdles in the form of third-party advertisements and dynamic overlays that frequently caused ElementClickInterceptedException errors. To overcome this and ensure consistent test execution, I implemented **JavaScript Injection** (`execute_script`). By bypassing the visual layer and triggering events directly within the Document Object Model (DOM), the suite becomes "human-proof" and self-healing. Unlike standard Selenium clicks that act like a physical mouse and can be blocked by invisible overlays, this method ensures that the tests reliably verify the underlying system conformance regardless of aggressive ad behavior.

5.3 Scalability and Future-Proofing

This suite was built with **Scalability** as a core requirement. Because each feature—such as login authentication, cart persistence, and product searching is isolated into its own dedicated file, the suite can grow indefinitely without becoming unmanageable. This modularity allows a team to add dozens of new test cases by simply following the established file pattern. Furthermore, the implementation of a single executor script (`main.py`) allows for the entire regression suite to be triggered with a single click. This fulfills the need for a scalable architecture that can support rapid, large-scale sweeps of the application every time a new change is deployed to the codebase.

5.4 Evaluation of Script-Based GUI Testing

- **Perceived Benefits:** The most significant benefit is the **Consistency and Speed** of regression. While a human tester would take 10–15 minutes to manually verify these 10 features and might overlook details like footer text this automated suite completes the entire check in approximately 75 seconds with perfect accuracy. This repeatability is essential for modern agile development.
- **Perceived Drawbacks:** The primary drawback is the **Upfront Time Investment**. It took 5.5 hours to properly architect the suite, handle dynamic overlays, and ensure reliable selectors. While manual testing is faster for a one-time "smoke check," the automated suite becomes exponentially more valuable with every subsequent execution throughout the software lifecycle.

6 Conclusion

This project successfully demonstrates a professional, script-based approach to automated GUI testing. By prioritizing modularity and implementing robust technical solutions for real-world web challenges, I have created a suite that is maintainable, reliable, and scalable. The automated results confirm that the **Automation Exercise** platform conforms to all ten functional requirements identified at the start of this assignment.