



Practical Web Test Automation with Playwright and Pytest

Courtney Zhan

Practical Web Test Automation with Playwright and Pytest

Courtney Zhan

This book is available at [Leapub](#).

This version was published on 2026-09-06



This is a [Leapub](#) book. Leapub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Courtney Zhan

Contents

Forward	i
Preface	iii
What is unique about this book?	iv
Who should read this book?	v
How to read this book?	v
Send me feedback	v
1: What is Web Test Automation?	1
1.1: Test automation benefits	2
1.2: Reality check	3
1.3: Reasons for test automation failures	4
1.4: Successful web test automation	7
1.5: Why Playwright framework?	8
1.6: Why pytest and Python language?	9
1.7: Learning approach	10
1.8: Next action	12
2: Test Execution Environment Setup	13
2.1: Install Python with pyenv (recommended)	13
2.2: Install Playwright and pytest	15
2.3: Alternative Python setup: system wide	19
2.4: Alternative Python setup: with venv	19
2.5: Review	22
3: First Automated Test	23

CONTENTS

3.1: Test Design	23
3.2: Installing TestWise (under 1 minute)	24
3.3: Create Automated Test	26
3.4: Create a test case	33
4: How Automated Testing works	38
4.1: Web test automation rhythm	38
4.2: Test syntax frameworks	38
4.3: pytest execution hooks	39
4.4: Run pytest tests from the command line	39
4.5: Review	40
5: Test Tool	41
5.1: TestWise: Next-Gen Functional Testing IDE	41
5.2: Visual Studio Code Programming Editor	57
5.3: PyCharm Development IDE	61
5.4: Review	68
6: Playwright Syntax in Nutshell	70
6.1: Browser, Context, and Page	70
6.2: Playwright locators	70
6.3: get_by_role()	70
6.4: get_by_label()	71
6.5: get_by_placeholder()	71
6.6: get_by_text()	71
6.7: get_by_test_id()	71
6.8: locator() with CSS/XPath	72
6.9: Page-level convenience methods	72
6.10: Locator chaining	73
6.11: Multiple matching elements	74
6.12: Playwright vs Selenium WebDriver	74
6.13: Review	74
7: Gain Productivity	75

CONTENTS

7.1: Playwright CodeGen	75
7.2: Playwright Trace Viewer	75
7.3: Some Generic IDE Features	75
7.4: Keyboard shortcuts	76
7.5: More TestWise productivity features	76
7.6: Review	77
8: Case Study	78
8.1: Test site	78
8.2: Preparation	78
8.3: Create a test project	78
8.4: Test Suite: Sign in	78
8.5: Test Suite: Select Flights	79
8.6: Enter passenger details	80
8.7: Book confirmation after payment	80
8.8: Run all tests	81
8.9: Review	81
9: Maintainable Functional Test Design	82
9.1: Record/Playback leads to unmaintainable test scripts	82
9.2: Success criteria	82
9.3: Maintainable automated test design	82
9.4: Maintain with ease	83
9.5: Case Study: refine test scripts	83
10: Test Automation Characteristics	85
10.1: Specific	85
10.2: Clean	85
10.3: Independent	85
10.4: Frequent	85
10.5: Focused	86
10.6: Programmable	86
10.7: Creative	86
10.8: Sustainable	86

CONTENTS

10.9: Review	86
11: Functional Test Refactoring	88
11.1: Code refactoring	88
11.2: Functional test refactoring	88
11.3: Tool support	88
11.4: Case study	88
11.5: Summary	91
11.6: Review	91
12: Set up Continuous Execution	92
12.1: Conventional Playwright test execution in CI/CD is suboptimal	92
12.2: BuildWise CT server setup	92
12.3: Sequential Execution	93
12.4: Parallel execution	93
12.5: Review	95
13: Preparing for Real-World Test Automation	96
13.1: Syntax errors	96
13.2: Beyond syntax errors	96
13.3: Set up source control	96
13.4: Test Data	97
13.5: Choose Robust Locators	97
13.6: Stabilize Tests	98
13.7: Test script maintenance is the primary effort	98
13.8: Custom libraries	99
13.9: Debugging	99
13.10: What is the best learning method?	99
13.11: Review	99
14: Team Collaboration	101
14.1: Pre-requisite	101
14.2: Developer-Tester Collaboration Scenario 1: “It worked on my machine”	101

CONTENTS

14.3: Developer-Tester Collaboration Scenario 2: Programmers help QA engineers with technical challenges	102
14.4: Developer-Tester Collaboration Scenario 3: Acceptance Test-Driven Development	102
14.5: Business Analysts find E2E test automation useful too	103
14.6: Review	103
15: Clarifications	104
15.1: Real E2E Test Automation is far more challenging than coding	104
15.2: Neither possible or necessary to achieve 100% coverage	104
15.3: Gherkin BDD Framework	104
15.4: Headless testing	104
15.5: Codeless/Scriptless Test Automation	105
15.6: AI/Agentic Test Automation	105
15.7: Cross-browser functional testing	105
15.8: Data-Driven Test	105
15.9: Review	105
16: Adopting End-to-End Test Automation	107
16.1: Seek executive sponsorship	107
16.2: Choose a test framework	107
16.3: Select a test tool	107
16.4: Run as part of the CI/CD pipeline	108
16.5: Find a mentor	108
16.6: Manage expectations	108
16.7: Solo test automation	108
16.8: Common mistakes	108
16.9: Review	110
17: AI Won't Replace Real Human E2E Test Automation	111
Afterword	112
Appendix 1 Functional Test Refactoring Catalog	113

Extract Function	113
Move Test Scripts	113
Extract to Page Class	113
Introduce Page Object	113
Rename	114
Appendix 2 Case Study: Test Automation in WhenWise App Development . .	115
Build Stats	115
Questions and Answers	116
Resources	117
Books	117
Tools	117
Blog	117
References	118

Forward

Practical Web Test Automation was my first book, and it holds a special place in my heart.

The journey began in 2006 after a colleague encouraged me to share the success of our software project following my introduction of web test automation. Our team was achieving production releases every sprint, essentially practicing what would later become known as DevOps, even though the term did not yet exist. Team morale was high, so was team's morale. Much of that success came from the tremendous benefits of well-designed end-to-end (E2E) test automation.

The book evolved from the training material I used to teach new team members. After years of writing, reviewing, and refining, the first edition was published in 2009.

Since then, it has helped hundreds of readers. Many have shared encouraging feedback, and some even told me it helped them secure their first test automation engineering job. Its strength has always been its practical approach: carefully designed, hands-on examples that help readers become productive quickly.

“Practical Web Test Automation” has gone through several editions, the most significant change came in the second edition, where the Watir framework was replaced with Selenium WebDriver. The title was appended “with Selenium WebDriver” to reflect this change. Despite that, the core principles and techniques remain the same. In fact, I still refer to parts of the book myself from time to time.

If someone new to test automation asked me for one book to learn the fundamentals, I would still recommend “Practical Web Test Automation with Selenium WebDriver”. So why a Playwright (in Python) edition?

When I entered the field, Ruby was the language of choice for test automation. Watir, the first widely adopted web automation framework, even included “Ruby”

in its original name (Web Application Testing in Ruby), reflecting the creators' appreciation of the language. The authors of the classic Agile Testing book also recommended Ruby for automated acceptance testing.

As test automation became mainstream, many new frameworks emerged, and the industry increasingly followed the preferences of software developers rather than experienced test automation engineers. One popular misconception was that E2E tests should always be written in the same programming language as the application. That is simply not true. E2E testing is black-box testing. It validates application behavior from the user's perspective and is independent of the application's implementation language. Throughout my career, I have successfully built Selenium-Ruby test suites for applications written in Java, C#, JavaScript, PHP, Python and Ruby.

With JavaScript becoming the dominant language for web development, numerous JavaScript-based automation frameworks have come and gone over the years, including PhantomJS, Protractor, TestCafe, Puppeteer, Cypress, and Playwright.

In recent years, some readers have asked for a Playwright edition. Given Playwright's rapid growth and its position as one of today's leading web automation frameworks, those requests are understandable.

Instead of rewriting the book myself, I encouraged my daughter to take on the project. Having grown up surrounded by test automation and now an experienced software engineer herself, she brings both a fresh perspective and hands-on experience with modern tools.

Courtney has done an excellent job. I hope this book helps you learn Playwright as effectively as "Practical Web Test Automation with Selenium WebDriver" has helped readers over the years. More importantly, teaches the principles of building reliable, maintainable, and effective end-to-end test automation quickly.

Zhimin Zhan

Brisbane, Australia

August 2026

Preface

From the day I (*still in middle school*) first learned end-to-end test automation from my father, I was taught not to fixate on any single framework or language. The core principles of E2E automation matter far more than the tools used to implement them, contrary to what many commercial vendors claim. Once those principles are understood, switching frameworks and languages becomes straightforward.

My published books reflect this philosophy:

- [Selenium WebDriver Recipes in C#](#)¹ (Apress, 2023) – **Selenium WebDriver** with **C#**
- [Web Test Automation in Action: Volume 1: Hands-On Exercises with Selenium WebDriver and Playwright](#)² (Leanpub, 2024) – Selenium WebDriver with **Ruby** and **Playwright** with **TypeScript**
- [Effective Web Test Automation with Selenium WebDriver and JUnit](#)³ (Leanpub, 2025) – Selenium WebDriver with **Java**
- [Practical Mobile Test Automation with Appium for iOS](#)⁴ (forthcoming) – **Appium** with Ruby

It is common for software teams to spend weeks, or even months, debating which automation framework, programming language, or tool they should adopt. Frankly, that rarely makes sense. Web test automation is independent of the application's coding language, and the underlying testing principles remain the same regardless

¹<https://www.amazon.com.au/Selenium-WebDriver-Recipes-Practical-Solutions/dp/B0CJ2DBVNG>

²<https://leanpub.com/web-test-automation-in-action-volume-1>

³<https://leanpub.com/effective-web-test-automation-selenium-junit/>

⁴<https://leanpub.com/practical-mobile-test-automation-with-appium-ios>

of the framework. Lengthy debates are often a sign that few people involved have successfully built and maintained a production-quality E2E test suite.

Like every software engineer, I have my own preferences. But those preferences have never prevented me from building a small suite of reliable, readable, and maintainable automated E2E tests in Playwright or Selenium WebDriver and running them on a Continuous Testing server from day one. The framework and language are implementation details; the engineering principles matter far more.

My hope is that this edition of Practical Web Test Automation, written using Playwright, pytest, and Python, reinforces that message. E2E test automation is a practical engineering discipline, not a framework-specific skill. Once the fundamentals are understood, moving between frameworks and programming languages becomes straightforward.

What is unique about this book?

This is not a book that simply lists Playwright syntax. You could probably ask AI to generate a comprehensive syntax reference from the official Playwright documentation. The result would be large, but that is not the right way to learn web test automation.

This book focuses on the essence of web test automation. It helps beginners get started quickly through hands-on exercises, write their first Playwright test the right way, learn to debug errors, become productive, and then move on to the core skills of **maintainable test design** and **functional test refactoring**.

The book contains carefully designed exercises that help readers build practical skills and confidence step by step. The goal is not simply to write a few test scripts, but to learn how to build and maintain a sizeable automated test suite.

The book then introduces a Continuous Testing solution for running the test suite effectively and reliably.

Finally, it covers various aspects and challenges you may encounter when implementing real-world E2E test automation, whether you are doing it in your own

workplace or for clients.

Who should read this book?

This book is for anyone involved in developing or testing web applications, including testers, software engineers, architects, and managers. No prior experience with web test automation is required. A basic understanding of programming and automated functional testing with frameworks such as Selenium WebDriver is helpful, but not necessary.

How to read this book?

I recommend reading this book from cover to cover. If you're eager to start writing tests, feel free to jump to Chapter 2 and return to the introductory chapter later.

Many chapters include hands-on exercises with step-by-step instructions that take around 10–30 minutes to complete. You can work through them as you read or revisit them afterward. The most important thing is to practice—E2E test automation is learned by doing.

Send me feedback

I'd love to hear from you. Comments, suggestions, bug reports, and improvements to the examples are all welcome. You can send your feedback via [the book website](http://courtneyzhan.com/books/practical-web-test-automation-playwright-pytest)⁵.

Courtney Zhan
Tokyo, Japan

⁵<http://courtneyzhan.com/books/practical-web-test-automation-playwright-pytest>

1: What is Web Test Automation?

Web Test Automation, also known as automated functional testing for web applications, is the practice of using software tools to automatically execute test cases in a browser. Instead of manually repeating tedious steps, test scripts interact with the application exactly as an end user would—clicking buttons, entering text, selecting options, and navigating pages—to verify that it behaves according to requirements.

When an automated test runs, you can watch a web browser perform these interactions automatically. The test then validates that the application’s actual behavior matches the expected results and reports any discrepancies as test failures.

Web Test Automation is a form of black-box functional testing because it verifies an application’s externally observable behavior without relying on knowledge of its internal implementation. Since most business requirements are ultimately exposed through the user interface, Web Test Automation plays a vital role in validating end-to-end business workflows. It has become one of the most widely adopted software testing practices, helping teams improve software quality, increase test coverage, and provide rapid feedback throughout the software development life-cycle.

Functional Testing vs Unit Testing vs Non-Functional Testing

Functional testing is to verify function requirements: **what** the system does – that is, whether it meets the specified business and user requirements. For example, “*User can request a password reset by providing a valid email address*”. Functional testing is the focus of this book.

Unit testing is a white-box testing technique performed by developers at the source code level. Unit tests are more accurately described as “programmer tests”, as they are written by the developers who implement the code rather than by software testers. A software application can pass all its unit tests yet still fail many functional tests, because unit tests reflect the programmer’s perspective while functional tests reflect the end user’s perspective.

Non-functional testing is the testing of **how** the system works. It evaluates characteristics such as performance, scalability, reliability, security, and usability. For example, “*The home page must respond within five seconds under normal load.*” Many types of non-functional testing, particularly performance and load testing, also rely heavily on automated testing tools.

1.1: Test automation benefits

Test automation provides many benefits throughout the software development lifecycle. The following are four commonly recognized advantages:

- **Reliable.** Automated tests execute the same steps precisely every time they run, reducing the risk of human errors and improving consistency.
- **Fast.** Automated tests can execute significantly faster than manual testing, allowing teams to receive feedback more quickly.
- **Repeatable.** Once automated tests are created, they can be executed repeatedly with minimal additional effort. They can run at any time, including during lunch breaks, overnight, or as part of an automated build process.
- **Regression Testing.** Regression testing ensures that software changes, such as bug fixes or new features, do not introduce unexpected defects into existing functionality.



“The intent of regression testing is to ensure that a change, such as a bug fix, did not introduce new faults” [Myers, Glenford 04]. “The only practical way to manage regression testing is to automate it.” [McConnell]

Comprehensive manual regression testing is almost impossible to conduct for two reasons: the time required and human errors.

What do I like about test automation? (by Zhimin Zhan)

If you want me to use only one adjective to describe web test automation, it is **fun**. As a software engineer, I enjoy creating something that can do the work for me. I have a habit of triggering the execution of a test suite before going out for lunch. I like the feeling of “still working” while enjoying a meal. When I come back, a test report is there.

As a product owner, I think it is essential to release software frequently without fear. To achieve that, comprehensive test automation via UI is a must.

1.2: Reality check

With more software projects adopting agile methodologies and more application development moving towards the Web, you would assume that web test automation is everywhere now. The answer is, sadly, no. In fact, functional testing in many projects is still executed in pretty much the same way: manually. For the past two decades, I have seen automated UI testing was done poorly or not at all in numerous “agile” projects, however, it has been talked a lot. Michael Feathers, a renowned agile mentor and the author of [Working Effectively with Legacy Code](http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052)¹, summarized better than what I can in this blog article.

¹<http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052>



UI Test Automation Tools are Snake Oil - Michael Feathers

It happens over and over again. I visit a team and I ask about their testing situation. We talk about unit tests, exploratory testing, the works. Then, I ask about automated end-to-end testing and they point at a machine in the corner. That poor machine has an installation of some highly-priced per seat testing tool (or an open-source one, it doesn't matter), and the chair in front of it is empty. We walk over, sweep the dust away from the keyboard, and load up the tool. Then, we glance through their set of test scripts and try to run them. The system falls over a couple of times and then they give me that sheepish grin and say "we tried." I say, "don't worry, everyone does." [Feathers 10]

Michael Feathers' summary of the status of automated end-to-end testing was spot on: software teams were either 'pretending automated UI testing' or 'not doing it at all'.

1.3: Reasons for test automation failures

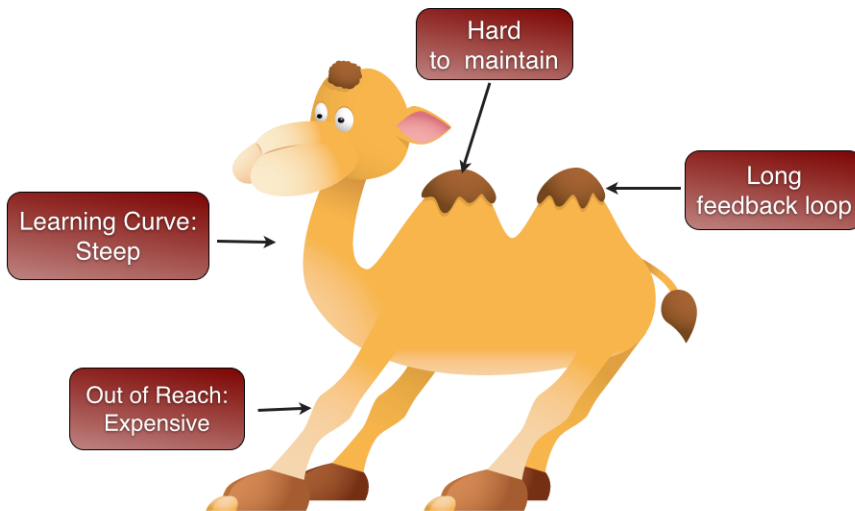
The software testing survey conducted by Innovative Defense Technologies in 2007 [IDT07] shows "73% of survey respondents believe Automated Testing is beneficial but few automate". The survey identified the following as the top reasons why organizations did not automate their software testing, even though they recognized its benefits:

- Lack of time
- Lack of budget
- Lack of expertise

These reasons may sound reasonable at first. However, they raise an interesting question: if test automation is expected to save time and reduce costs, why are lack of time and budget among the main barriers to adopting it?

To make these challenges easier to understand, we can use an analogy. Imagine a person attempting to climb over a standing two-hump camel from the front. A software project's journey toward successful test automation can be compared to this difficult climb.

Let's examine each of the following challenges that he encounters:



1. Out of reach: Expensive

Commercial test automation tools are often expensive. I will not list specific prices here; in fact, some so-called leading testing tool vendors, such as HP QuickTest Professional (QTP), did not even publish pricing information on their websites, which is telling in itself. *(I remember my father once showing me a quote for IBM Rational Functional Tester that was priced at \$11,000 per license.)*

Before the 2000s, end-to-end test automation, regardless of its outcome, was largely affordable only for large companies and government projects. Even in those organizations, adoption was often limited, typically with only a single QA engineer using a single tool license.

Fortunately, free and open-source test automation frameworks emerged, such as Watir and Selenium, both of which were featured in the classic book *Agile Testing* (2009) by Lisa Crispin and Janet Gregory. Unlike many commercial tools that relied heavily on record-and-playback utilities, these frameworks encouraged writing automated tests as real code for the flexibility needed using programming languages such as Ruby.

However, it took quite a long time the idea of free and open-source testing frameworks is still not appealing to many test managers. Today, open-source end-to-end test automation frameworks have become mainstream. In web test automation, Selenium WebDriver and Playwright are two of the leading frameworks, well ahead of many others.

The cost barrier has largely disappeared: the tools themselves are freely available. The real challenges have shifted from purchasing tools to acquiring the necessary skills.

2. Steep Learning Curves: Difficult to learn

Traditional commercial test automation tools often relied on a Record and Playback approach with test scripts in a vendor-proprietary syntax. Although they appeared simple in sales demonstrations, real-world maintenance was difficult. A small application change could break hours of recorded scripts, and testers eventually had to edit the generated scripts manually.

Open-source frameworks such as Selenium WebDriver and Playwright provide greater flexibility and maintainability, but they require programming skills. This creates a learning barrier for many testers who do not have a programming background or feel uncomfortable learning to code. Furthermore, general-purpose programming IDEs are designed for developers rather than testers and can be overwhelming for newcomers.

3. Hump 1: Hard to maintain

Software under development changes frequently, and automated UI test scripts are sensitive to application changes. Even a small UI modification, such as extra command in a JavaScript or CSS, can break many test scripts. This maintenance

challenge is the most fundamental reason that many test automation efforts fail.

4. Hump 2: Long feedback loop

Compared with programmer tests, which should execute in less than 0.1 seconds when well designed, UI-based functional tests are significantly slower. There is limited ability to improve their raw execution speed, and as the test suite grows, so does the execution time.

This creates a long feedback gap between a code commit and the availability of test results. If developers continue making changes during this gap, the team can easily fall into a “tail-chasing” cycle, reducing productivity and impacting morale.

New Challenge for testing Web applications

With the widespread adoption of AJAX (Asynchronous JavaScript and XML) and the increasing use of JavaScript, modern websites have become more dynamic and interactive. These changes make web applications more difficult to automate reliably compared with traditional static web pages.

New Challenge in AI Coding era

AI coding, including approaches such as vibe coding and agentic coding, has become one of the hottest trends in the software industry since 2025. This book does not attempt to evaluate the merits of these approaches. Instead, it focuses on one practical observation: as AI generates code more frequently, the demand for effective end-to-end (E2E) testing will continue to grow.

AI-generated code can introduce defects, break existing features, and create regression issues. Manual E2E testing cannot keep pace with the increasing speed of AI-assisted development. Software teams therefore need to upskill in effective E2E test automation.

1.4: Successful web test automation

Having identified the common causes of test automation failures, we can define the key requirements for successful web test automation:

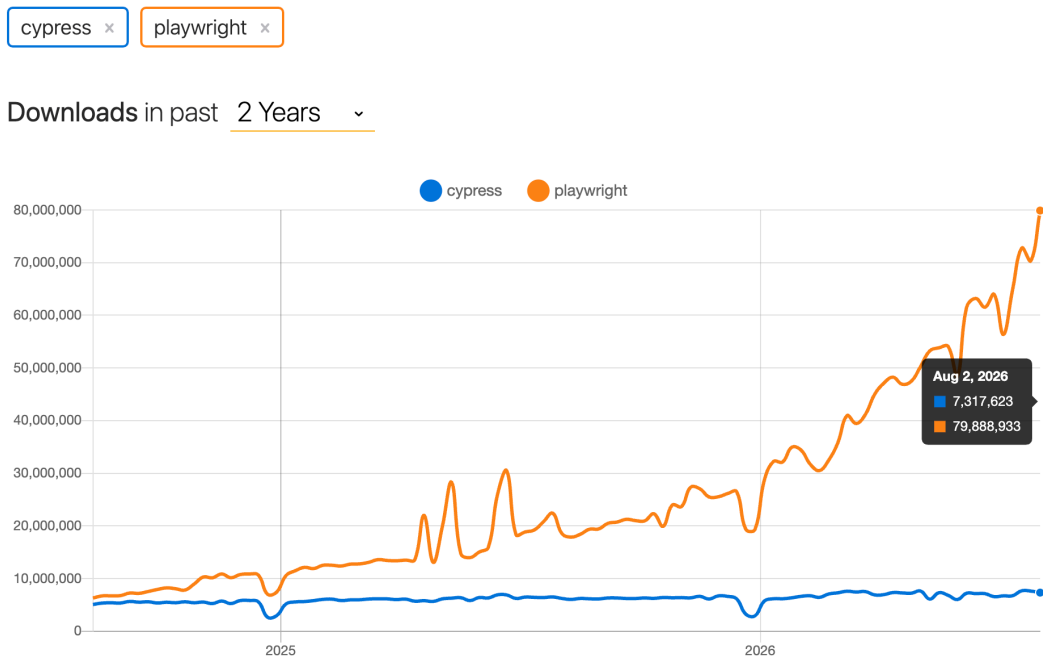
1. Test scripts must be easy to read, understand, and maintain.
2. The test framework must be reliable and flexible, preferably free and open-source.
3. The testing tools must be easy to learn, affordable, and support collaboration across the team.
4. Test execution must be fast enough to provide timely feedback.

Is achieving all of this possible? The answer is **yes**. The purpose of this book is to demonstrate practical approaches to building effective web test automation that meets these goals.

1.5: Why Playwright framework?

Playwright is a free and open-source browser automation framework developed by Microsoft. Since its initial release in 2020, its adoption has grown rapidly. For example, according to [npmtrends](https://npmtrends.com)², which tracks JavaScript package downloads, as of 2026-08-02, Playwright records roughly 11 times as many downloads as Cypress, another well-known JavaScript web test automation framework.

²<https://npmtrends.com/cypress-vs-playwright>



This represents remarkable growth, increasing from near parity to a 11:1 download ratio in approximately two years.

In many software companies, Selenium WebDriver and Playwright are now the two most commonly approved web test automation frameworks.

1.6: Why pytest and Python language?

pytest is the de facto test framework for Python.

Playwright supports multiple programming languages, including Python, JavaScript/TypeScript, Java, and .NET. Among them, Python is my first choice.

There are several reasons:

- **Python is a proper scripting language**

Among Playwright's language bindings, Java and C# are compiled languages, while TypeScript is not traditionally considered a scripting language. Python, however, is a true scripting language. Automated test scripts are better written in a scripting language, naturally.

*(Depending on factors such as reader feedback, I may also write future editions of **Practical Web Test Automation with Playwright** series using the other three languages, all of which I am also familiar with)*

- **Python syntax is similar to Ruby.**

Ruby has long been (and still is) my preferred scripting language for end-to-end test automation. However, Playwright does not currently support Ruby, making Python a natural alternative.

- **Python is one of the most popular programming languages.**

Python has ranked No. 1 in IEEE Spectrum's annual Top Programming Languages ranking for many consecutive years. This suggests strong market demand and excellent career opportunities for Python developers.

1.7: Learning approach

This book is not another collection of testing theories—there are already plenty of those available. Instead, it takes a practical, hands-on approach. We will learn web test automation by working through real examples using the Playwright test framework and the TestWise functional testing IDE/Visual Studio Code. The most effective way to learn automation is by writing and running tests.

My grandfather was a highly respected high school mathematics teacher in our town. His teaching approach was simple: **teach through examples**. He selected carefully designed exercises, provided concise explanations, and guided students when they encountered difficulties. This is also how I learned test automation, and through my experience, I believe this approach is one of the most effective ways for testers to quickly master automation skills.

Most web applications behave consistently across desktop platforms, including Windows, Linux, and macOS. Therefore, any of these desktop operating systems can be used to follow the examples in this book. (*Mobile testing is a separate discipline and is covered in my other book, Practical Mobile Test Automation*)

For consistency, all examples in this book are demonstrated on macOS, mainly for practical reasons:

- This book is written in my spare time, and my primary personal computer is a MacBook.
- I am also writing “Practical Mobile Test Automation”, which requires the iOS Simulator that is only available on macOS.
- Mac computers are a popular choice in the Python community.
- Test execution is generally faster on macOS than on Windows.

The frameworks and tools used throughout this book are:

Web Browser	Chromium
Automation Framework	Playwright
Test Framework	pytest
Scripting Language	Python
Testing Tool	TestWise IDE / Visual Studio Code
Continuous Testing Server	BuildWise

All the frameworks, tools, techniques and test scripts are directly applicable for Windows and Linux as well.

On testing tools, I use mainly TestWise, a functional testing IDE that supports Selenium WebDriver, Watir and Appium (*TestWise maybe used free of charge*), in this book. However, readers who prefer their own editors or IDEs can still follow

along, as all test scripts in this book are plain text files. Instructions for running tests from the command line are also provided.

Example test scripts for each chapter can be downloaded from the book website. Readers can open these scripts in TestWise and run them immediately. Screen-casts are also provided to demonstrate the techniques in practice.

This book focuses on testing standard HTML-based websites and excludes vendor-specific or deprecated technologies such as Flash and Silverlight, which are now rarely used in modern web applications.

1.8: Next action

Enough theory for now. Let's roll up sleeves and write some automated tests.

2: Test Execution Environment Setup

The goal of setting up the test execution environment is straightforward: to be able to run test scripts from a terminal using a command such as:

```
% pytest tests/login_test.py
```

Here, `pytest` is the test runner (*and % is the shell prompt*). It should be available in the system's `PATH` so that the command can be executed from any directory.

2.1: Install Python with `pyenv` (recommended)

Python installation can get complex (see below), this certainly unnecessary for the purpose of test automation.

Fortunately, `pyenv` is a simple and easy-to-use Python version management tool. It makes installing and switching between Python versions straightforward. Inspired by **`rbenv`** (for Ruby language), `pyenv` follows the same simple command-line approach, making it easy to learn and use.

2.1.1: Prerequisite

Installing Python on your Mac is quick and straightforward. First, install Xcode's command line tools and HomeBrew (a package manager).

- **XCode Command Line Tools**

```
xcode-select --install
```

- **HomeBrew**

[HomeBrew](https://brew.sh)¹ is the open source package manager for mac. Copy and run the command at the top of the page. It should look something like:

```
/bin/bash -c \"  
\"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)\"
```

2.1.2: Install Python 3

Run the commands below in a terminal.

1. **Install Pyenv first.**

```
% brew install pyenv
```

2. **Install a stable Python 3 distribution.**

```
% pyenv install 3.14.6
```

Output:

```
Installed Python-3.14.6 to /Users/me/.pyenv/versions/3.14.6
```

3. **Set to always use the newly installed Python.**

```
% pyenv global 3.14.6
```

4. **Verify Python installation.**

¹<https://brew.sh>

```
% pyenv versions
system
* 3.14.6 (set by /Users/courtney/.pyenv/version)
```

2.1.3: Add pyenv to your Shell profile

You must tell your terminal where to look for pyenv's version of Python. Add the following to the bottom of `~/.zshrc` or `~/.zprofile`.

```
export PYENV_ROOT="$HOME/.pyenv"
[[ -d $PYENV_ROOT/bin ]] && export PATH="$PYENV_ROOT/bin:$PATH"
eval "$(pyenv init -)"
```

Start a new terminal, run the command below to verify:

```
% which python
/Users/me/.pyenv/shims/python

% python --version
Python 3.14.6
```

2.2: Install Playwright and pytest

A test automation solution actually consists of two independent frameworks:

- **Automation framework** – Such as Selenium WebDriver or Playwright.

It drives the application (a web browser for web applications) and interacts with the application under test.

- **Test Syntax framework**– Such as pytest or RSpec.

It provides the test structure, execution engine, fixtures, and assertion capabilities. Some refer them as test runners.

Please note that these two frameworks are independent of each other. For example, Playwright can be used with pytest for Python, Mocha for JavaScript, JUnit for Java, or NUnit for C#.

1. Install Playwright

```
% pip install playwright
```

Output like:

```
Collecting playwright
  Downloading playwright-1.61.0-py3-none-macosx_11_0_arm64.whl.metadata
...
Installing collected packages: typing-extensions, greenlet, pyee, playwright
Successfully installed greenlet-3.5.3 playwright-1.61.0 pyee-13.0.1 typing-extens\
ions-4.16.0
```

Then, download the required browser engines (Chromium, Firefox, WebKit).

```
% playwright install
```

You shall see output like below.

```
Downloading Chrome for Testing 149.0.7827.55 ...
...
Chrome for Testing 149.0.7827.55 (playwright chromium v1228) downloaded ...
Chrome Headless Shell 149.0.7827.55 (playwright chromium-headless-shell v1228) ...
...
Firefox 151.0 (playwright firefox v1532) downloaded to ...
...
WebKit 26.5 (playwright webkit v2311) downloaded ...
```

2. Install pytest test syntax framework

```
% pip install pytest
```

Verify pytest.

```
% pytest --version
pytest 9.1.1

% which pytest
/Users/me/.pyenv/shims/pytest
```

3. Install **pytest-playwright**

pytest-playwright is the official, Microsoft-developed plugin that integrates the Playwright automation library with the pytest testing framework. It is optional, but recommended.

```
% pip install pytest-playwright
```

For testing, you will need to add some utility libraries, such as Faker. You can install these easily with the `pip install` command. For now, we'll just do the basic installation and install those libraries later when we need them.

2.2.1: Verify the Playwright + pytest Installation

Create a Python script named `hello_playwright.py` using your preferred text editor, and enter (or paste) the following content:

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    page = browser.new_page()
    page.goto("https://whenwise.agileway.net")
    page.locator("input[name='search']").fill("driving")
    page.locator("input[name='search']").press("Enter")
    browser.close()
```

Run the script file:

```
% python hello_playwright.py
```

A Chromium browser window will launch, navigate to the WhenWise sandbox site, perform a search, and then close automatically. It may open and close too quickly for you to see. If needed, add a short delay before closing the browser with `page.wait_for_timeout(3000)`, which pauses execution for 3 seconds.

Next, we verify playwright with pytest.

Create another python script file named `hello_pytest.py` with the following content:

```
from playwright.sync_api import sync_playwright, expect

def test_pytest_site():
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=False)
        page = browser.new_page()
        page.goto("https://pytest.org")
        page.get_by_role("link", name="Auto-discovery").click()
        expect(page).to_have_title("Wrong shall failed")
        browser.close()
```

Run the test script file:

```
% pytest hello_pytest.py
```

The automated test (not just a script) will open the pytest website, click the “Auto-discovery” link, and then close the browser. The test will fail because the expected page title is intentionally incorrect:

```
===== short test summary info =====
FAILED hello_pytest.py::test_pytest_site - AssertionError: Page title expected to\
be 'Wrong shall failed'
===== 1 failed in 6.67s =====
```

This demonstrates that Playwright assertions work correctly with pytest: when the actual page title does not match the expected value, pytest reports the failure. Change it to the actual title (“Good Integration Practices - pytest documentation”) and rerun it and the test should pass.

```
===== test session starts =====
platform darwin -- Python 3.14.6, pytest-9.1.1, pluggy-1.6.0
rootdir: /Users/me
collected 1 item

hello_pytest.py . [100%]

===== 1 passed in 1.04s =====
```

2.3: Alternative Python setup: system wide

Before macOS 12, Python 2 was pre-installed with the operating system. Installing a separate Python 3 alongside the system Python can lead to confusion and package management issues, which is one reason pyenv has become the preferred approach.

It is still possible to install the standard Python 3 distribution on macOS:

```
% brew install python
```

However, installing Python packages may produce a warning about modifying an externally managed Python environment, may require the `--break-system-packages` option.

```
% pip3 install --break-system-packages pytest
```

It can become more complicated over time. Using **pyenv** avoids these issues by keeping Python installations isolated and easier to manage.

2.4: Alternative Python setup: with venv

This approach stores the execution dependencies inside the project folder, similar to JavaScript's `node_modules`. Many tutorials recommend it as the default setup. However, I disagree for test automation. After explaining how it works, I'll also explain why I don't recommend it.

Activate your venv:

```
% python3 -m venv .venv
% source .venv/bin/activate
```

Once activated, you will see `(.venv)` appended to the front of your terminal prompt.

Install in this virtual environment.

```
(.venv) % pip3 install playwright
(.venv) % pip3 install pytest
(.venv) % playwright install
(.venv) % pip3 install pytest-playwright
```

Now check: which `pytest`. It should show something like:

```
/your/project/.venv/bin/pytest
```

Then `pytest` will use the same Python that has Playwright.

Your `pytest` project then has:

```
project/  
├── .venv/  
├── tests/  
└── ...
```

2.4.0.1: Verify the test execution

```
(.venv) % pytest tests/new_test.py --headed
```

You shall see a Chrome browser launching and run a simple test.

2.4.0.2: Why I don't recommend venv setup?

The main advantage of this approach is that it removes external dependencies by keeping everything inside the project. However, it also comes with several drawbacks, that are especially unsuitable for E2E test automation.

1. Large .venv folder

As you can see, for a minimal test script setup, the .venv folder is already over 152MB.

```
(.venv) % du -h -d 1  
56K  ./ch02-first-project  
152M ./.  
152M .
```



Tip: If you adopt this approach, be sure to add .venv to your .gitignore file so the virtual environment is not committed to your Git repository.

2. E2E test automation has few library dependencies

End-to-end test automation projects typically rely on only a small number of libraries, and those libraries rarely need to be updated. Dependency conflicts are uncommon.

3. Most E2E test projects use the same libraries

In practice, most E2E test automation projects use the same set of testing libraries. Installing a single global test execution environment with the required libraries is usually sufficient. I rarely encounter a situation where one project requires a completely different version of Playwright or other automation libraries than another.

4. Complicates test execution

Before running tests, you must first activate the virtual environment. This adds an extra step every time you work with the project. Given the limited benefits discussed above, I believe this unnecessarily complicates test execution, especially when running tests on CI/CD build agents.

2.5: Review

If you use **pyenv**, installing Playwright and pytest can be as straightforward as installing Selenium and RSpec. Without it, however, the setup can become troublesome if you're not careful.

3: First Automated Test

A journey of a thousand miles must begins with a single step.

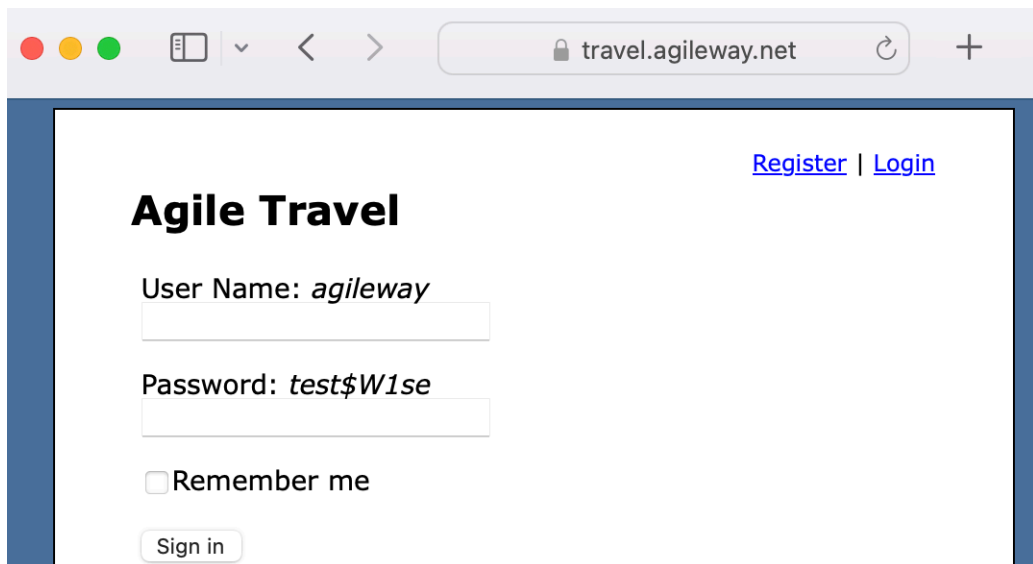
– Lao Tzu

Let's write an automated web test. If you are new to automated testing, don't feel intimidated. You are going to see your first automated test running in a Chrome browser in about 10 minutes, and that includes installing the test tool!

3.1: Test Design

A test starts with a requirement (called User Story in agile projects). Quite commonly, the first simple requirement to test is User Authentication. We will use this requirement for our first test in this exercise.

By analyzing the requirement and the application (see the screenshot below),



The screenshot shows a web browser window with the address bar displaying "travel.agileway.net". The page content includes a blue header bar with the text "Agile Travel" in bold. To the right of the header, there are links for "Register" and "Login". Below the header, there is a login form with two input fields: "User Name: agileway" and "Password: test\$W1se". Below the password field, there is a checkbox labeled "Remember me". At the bottom of the form, there is a "Sign in" button.

We can start to collect the test data:

Site URL: `https://travel.agileway.net`

User Login/Password: **agileway/test\$W1se**

and design the test steps:

1. Click the 'Login' link.
2. Enter username "agileway".
3. Enter password "test\$W1se".
4. Click the "Sign in" button.
5. Verify: "Welcome agileway" appears.

You might by now be saying *"there is no difference from manual testing"*. Your observation is correct. If you currently work as a manual tester, you probably feel relief at knowing your test design skills can apply to automated testing. As a matter of fact, we usually perform the test steps manually as verification of test design before writing automated test scripts.

Now we are going to automate it. The main purpose of this exercise is to help you write an automated Playwright test case and watch it running in a browser, in a matter of minutes. Don't pay attention to details yet, as it will become clear as we move on. If you get stuck, follow the screencast for this exercise at [the book site](#)¹.

3.2: Installing TestWise (under 1 minute)

In this exercise, we will use TestWise, a functional testing IDE (created by Zhimin Zhan) designed for open-source test automation frameworks such as Selenium, Appium, and Playwright. TestWise is free for light use, with no license required.

Prerequisite

¹<http://courtneyzhan.com/books/practical-web-test-automation-playwright-pytest>

- A PC with macOS or Windows 10+ or Linux
- Playwright pytest execution environment installed (*Chapter 2*)

Download

TestWise IDE can available for all three platforms: macOS, Windows and Linux, can be downloaded at <https://agileway.com.au/testwise/downloads>.

Install

- **TestWise IDE**

- macOS

Standard installation procedure: open the DMG file, drag the app to the 'Applications' folder.

- Windows platform

Double click `TestWise-x.x-setup.exe` to install, accept all default options. The default installation folder is `C:\agileway\TestWise7`. Launch TestWise after the installation completes.

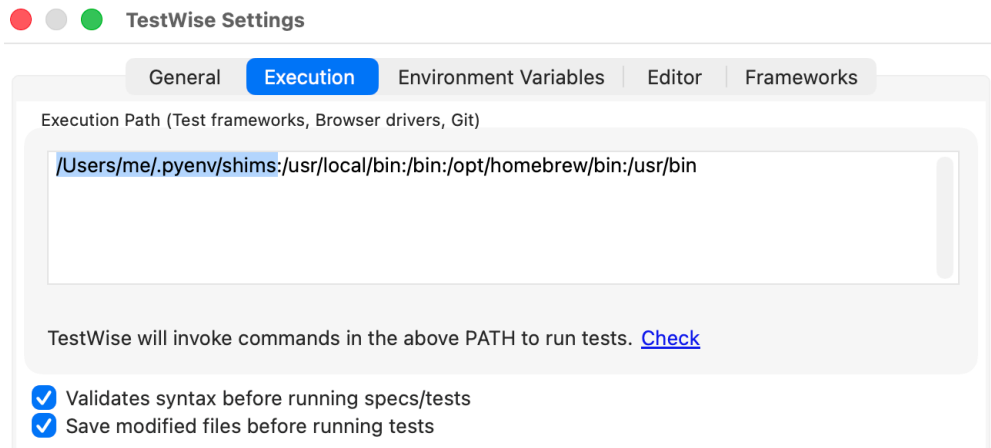
If you using TestWise Ruby edition on Windows, can skip the below in the 'Install' and 'Setup' sections below.

- Linux

`tar -zxvf testwise-x.x.zip` then run `install.sh` there.

- **Set up Execution Path in TestWise**

Make sure add your Python (and pytest) path, e.g. `/Users/me/.pyenv/shims`, to the TestWise's **Execution** settings (as below).



The PATH setting in my TestWise:

```
/Users/me/.pyenv/shims:/usr/local/bin:/bin:/opt/homebrew/bin:/usr/bin
```

3.3: Create Automated Test

Now we are ready to create our automated test: “User can log in the site”. Hope you still remember the test design steps and test data.

3.3.1: Create a new test project

TestWise organizes test scripts into projects. Since we are starting from scratch, the first step is to create a new project. If TestWise already has a sample project open, close it before creating the new project.

Select menu **File** -> **New Project**, which will bring up the window shown below.

New Project

Project name: AgileTravel

Location: /Users/me/testprojects/AgileTravelTests ...

Enter or select an empty folder for the test project.

Test Automation Frameworks

Automation Driver: ☐ Selenium-WebDriver ☐ Appium ☐ Watir ☒ Playwright ☐ API

Test Script Syntax: ☐ RSpec ☒ Pytest ☐ Mocha ☐ Cucumber ☐ Playwright Test

Project Settings

☒ Website URL ☐ App Executable ☐ Bundle ID

https://travel.agileway.net

☒ Create project skeleton

☐ Include the helper for load testing

OK Cancel

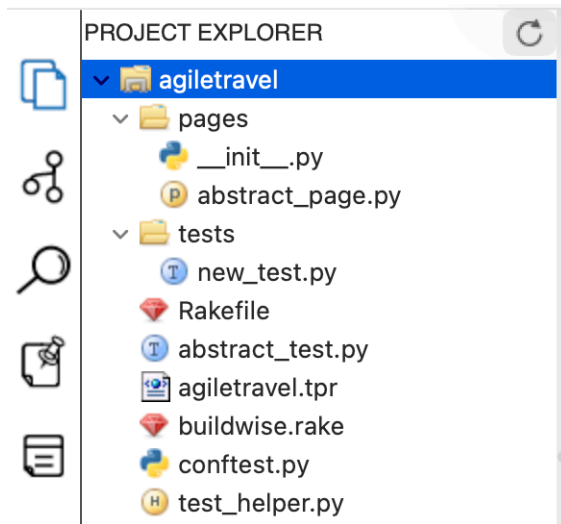
Enter a project name, project folder (*a newly created one*) and the URL of the web site to be tested.

In this case, enter “Agile Travel”, “/Users/me/testprojects/AgileTravelTests” and “http://travel.agileway.net” respectively.

Select ‘**Playwright**’ as the automation driver and ‘**Pytest**’ as the test syntax framework, then click ‘**OK**’ button. TestWise will create the project with skeleton files.

Please note, we just enter the host name of URL, not https://travel.agileway.net/login, rather https://travel.agileway.net.

3.3.2: Test project skeleton

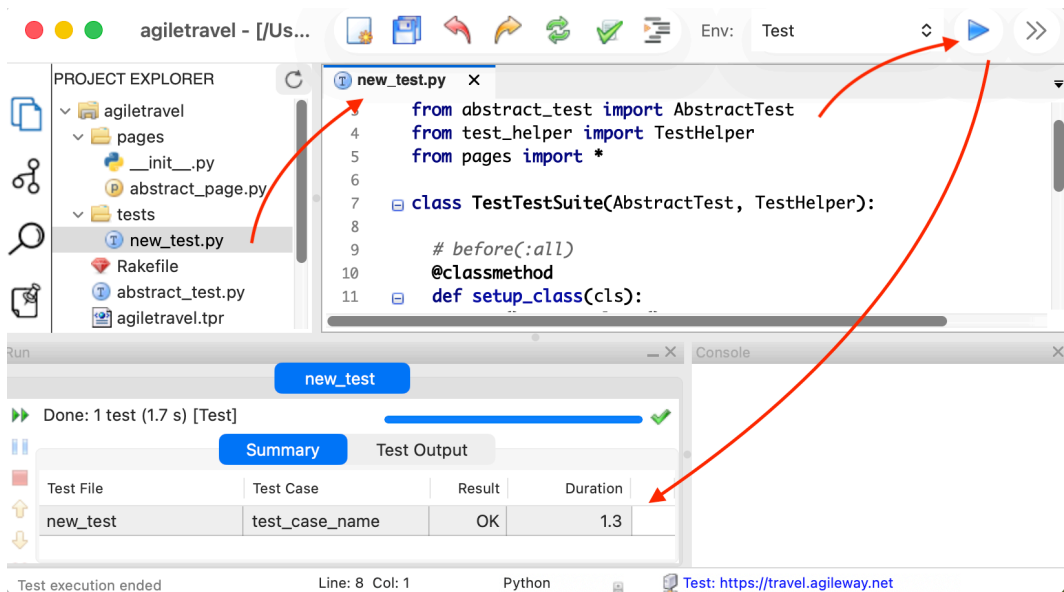


For now, you only need to know that test scripts are stored in the **tests** folder. We will explore the other files and folders as we progress through the book.

3.3.3: Run the sample test

You will find **new_test.py** under the **tests** folder. This is a sample test script provided by TestWise to help you get started.

To run the test, click the file name in the left project explorer to open it, then click the Run button (the blue triangle ►) on the toolbar to execute the test script.



You should see a Chromium browser launch, open the Agile Travel website, and then close automatically after the test finishes.

- Run the test script from the command line to verify that the test environment is configured correctly.

3.3.3.1: If no browser launched ...

If the browser does not launch or the test does not execute successfully, check the following:

1. Python and pytest setup.

Run the test script from the command line to verify that the test environment is configured correctly.

```
% pytest tests/new_test.py
```

2. The Playwright browser binaries are not installed.

In this case, the test will fail with an error indicating that the required browser is missing. Then run `playwright install`.

3. The pytest execution not in TestWise execution path.

Verify the `pytest` executable configured in the `PATH` setting in TestWise.

4. `pytest: error: unrecognized arguments: -headed`

You are missing the `pytest playwright` extension. Install it:

```
% pip install pytest-playwright
```

Alternatively, add an entry to `conftest.py`; the sample already includes examples.

5. `argparse.ArgumentError: argument --tracing: conflicting option string: --tracing`

You specified the `--tracing` argument option in `conftest.py`, but `pytest-playwright`, which you have already installed, already provides this option. Simply remove the `--tracing` option from `conftest.py`.

6. The test is running in headless mode.

The test executes normally, but the browser window remains hidden. To verify this, force Playwright to run in headed mode by using the following command:

```
% pytest tests/login_test.py -s --headed
```

If you are using TestWise, you can enable or disable headless mode using the Headless checkbox in the **Execution** Settings.

3.3.4: Recording Test Steps using Playwright Codegen

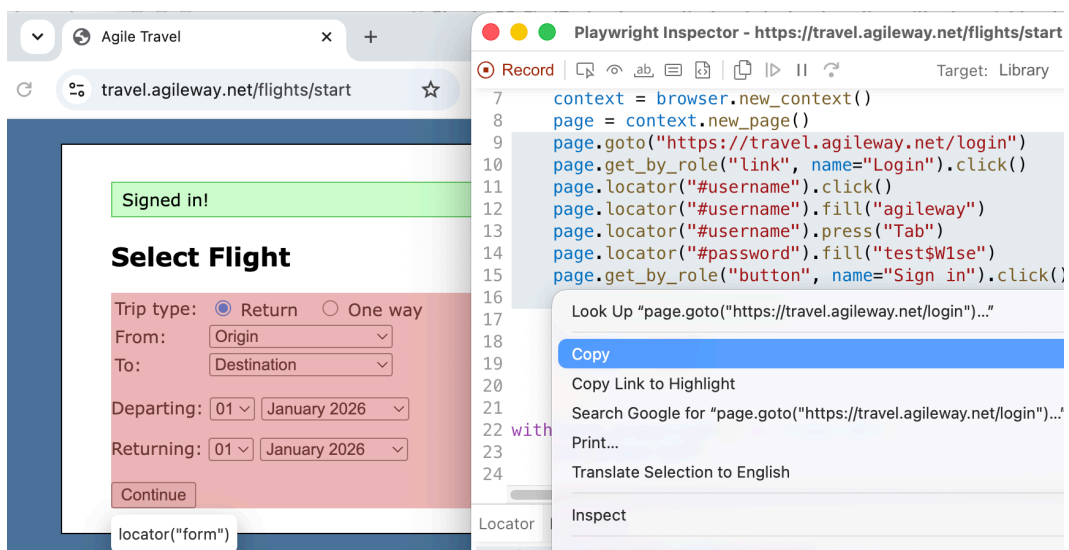
Playwright has a built-in codegen (Test Generator) feature, which tracks your interactions in a browser window and automatically writes the corresponding automation code. To put simply, it is like a record-n-playback tool.

I Don't recommend this way, more on this later

Start Playwright Codegen with the command:

```
% playwright codegen --target=python https://travel.agileway.net
```

This will launch a Chromium browser opening the agile travel site and also a Playwright Inspector window for codegen.



Perform the test steps below manually:

1. Click the 'Login' link

2. Enter username 'agileway'
3. Enter password 'test\$W1se'
4. Click the 'Sign in' button
5. Unable to perform.

The Playwright codegen does not generate test assertions automatically. (It mainly records interactions (click, fill, select, etc.) and locators.)

Test steps are recorded along the way. Once done, inside the **Playwright Inspector** window, copy recorded test steps to clipboard.

```
page.get_by_role("link", name="Login").click()
page.locator("#username").click()
page.locator("#username").fill("agileway")
page.locator("#username").press("Tab")
page.locator("#password").fill("test$W1se")
page.get_by_role("button", name="Sign in").click()
page.get_by_text("Welcome agileway | Sign off ()").click()
```

Why Record/Playback is not good for real test automation?

Recording makes it quick and easy to generate test steps, but the resulting scripts are typically low quality and difficult to maintain. This approach has been tried by commercial test automation tools for decades and has consistently failed to deliver maintainable test suites.



“Record/playback testing tools should be clearly labeled as ‘training wheels.’” – Jason Huggins, creator of Selenium at AAFTT Workshop 2009.

Although I strongly discourage using recorders (or Playwright Codegen) to create automated E2E test scripts, it is still worth understanding how it works. It can be

useful in certain situations, such as learning and exploration, but should be used with caution.

The alternative is to write tests by hand. Many newcomers worry that hand scripting is difficult or time-consuming, but with the right techniques and tools, it is neither. Throughout this book, you will learn a practical and efficient approach to writing maintainable automated tests by hand.

3.4: Create a test case

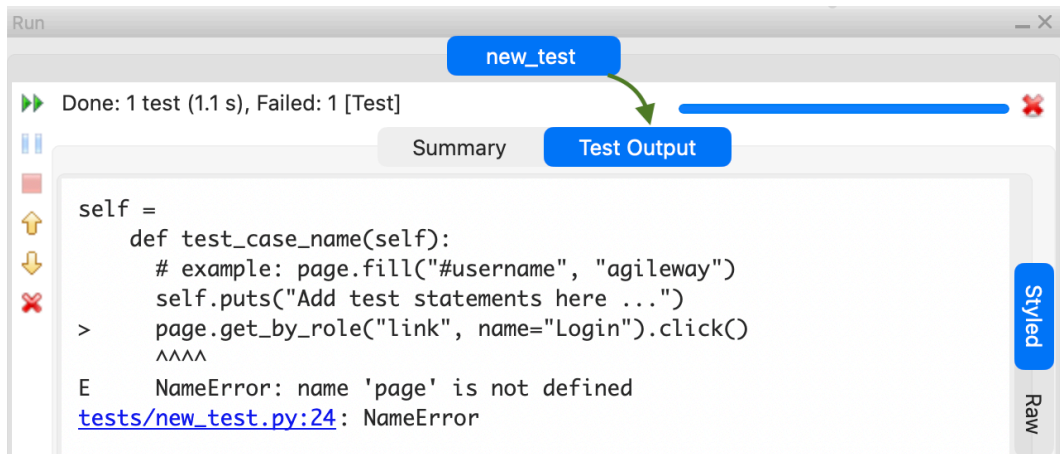
Now you should have some test steps created by a recorder or manual scripting.

Switch to the TestWise IDE, with `new_test.py` open in the editor. Paste the recorded test scripts into the test case.

```
21  def test_case_name(self):  
22      # example: page.fill("#username", "agileway")  
23      self.puts("Add test statements here ...")  
24      page.get_by_role("link", name="Login").click()  
25      page.locator("#username").click()  
26      page.locator("#username").fill("agileway")  
27      page.locator("#username").press("Tab")  
28      page.locator("#password").fill("test$W1se")  
29      page.get_by_role("button", name="Sign in").click()  
30      page.get_by_text("Welcome agileway | Sign off").click()
```



Run the test script. It launched a browser (good, means syntax OK) but failed quickly.



The error message, “**NameError: name ‘page’ is not defined**”, indicates that `page` is not defined within the test case. This is because the `page` is available through `self`.

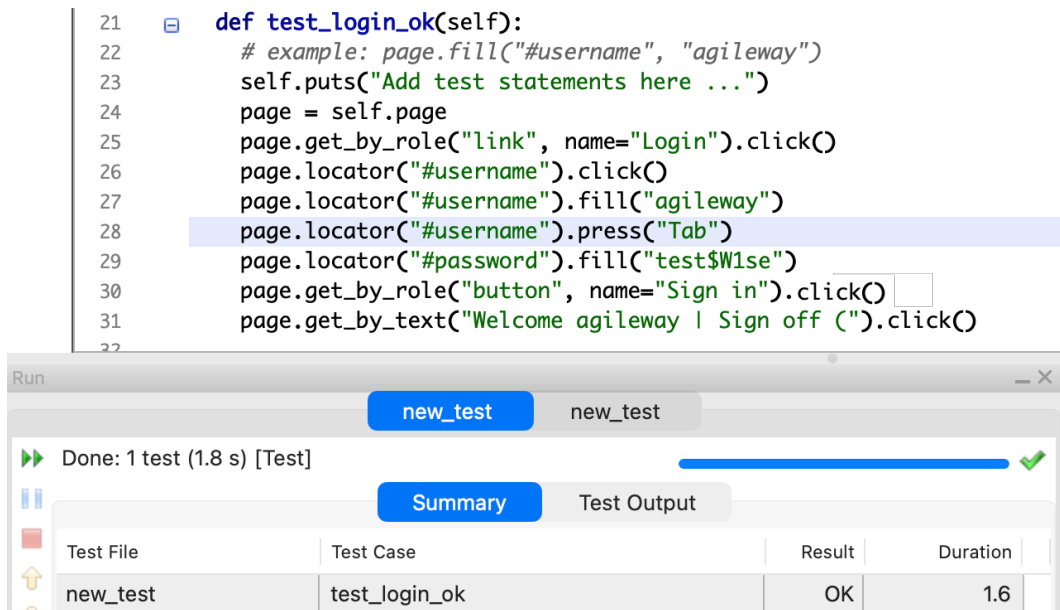
There are two ways to fix this:

- Replace every `page.` with `self.page`, or
- Add `page = self.page` before the test steps.

The second approach is preferred because it keeps the recorded Playwright scripts unchanged and makes the test steps easier to read.

While editing the file, rename the test case name by changing `def test_case_name` to `def test_login_ok`.

Run the test script again. This time, the test passes successfully.



Good! The test is now passing, but it still needs a few cleanups and one missing verification step. Let's finish it.

1. Remove the generated placeholder line

Delete the generated placeholder line: `self.puts("Add test statement here ...")`.

2. Remove unnecessary recorded steps.

A common issue with browser recorders is that they generate extra actions that are not required. For example, these two statements can be removed:

- `page.locator("#username").click()`
- `page.locator("#username").press("Tab")`

3. Add the missing assertion step.

The recorded last step `page.get_by_text("Welcome agileway | Sign off").click()` is invalid (not assertion).

Replace it with:

```
expect(page.locator("body")).to_contain_text("Welcome agileway").
```

The completed test case is shown below.

```
def test_login_ok(self):
    page = self.page
    page.get_by_role("link", name="Login").click()
    page.locator("#username").fill("agileway")
    page.locator("#password").fill("test$Wise")
    page.get_by_role("button", name="Sign in").click()
    expect(page.locator("body")).to_contain_text("Welcome agileway")
```

3.4.1: Full Test Script

The full test scripts in the TestWise shall be like the below:

```
from playwright.sync_api import Page, expect

from abstract_test import AbstractTest
from test_helper import TestHelper
from pages import *

class TestLogin(AbstractTest, TestHelper):

    # before(:all)
    @classmethod
    def setup_class(cls):
        super().setup_class()
        cls.page.goto(TestHelper.site_url())

    def test_login(self):
        self.page.get_by_role("link", name="Login").click()
        self.page.fill("#username", "agileway")
        self.page.locator('input[name="password"]').fill("test$Wise")
        self.page.locator("xpath=//input[@value='Sign in']").click()
        expect(self.page.locator("body")).to_contain_text("Welcome agileway")
```

Run the refined test script again and verify that it passes.

3.4.2: Review

Let's review what was accomplished in this chapter. In addition to creating the test itself, we:

- Set up the Playwright and pytest test execution environment.
- Installed the TestWise IDE.
- Created a new test project in TestWise.
- Recorded test steps using Playwright Codegen.
- Created a test script by pasting the recorded steps.
- Executed the test case in a browser and observed both passing and failing results.
- Refined recorded steps and added the missing assertion.

Hopefully, all of this was completed in about 10 minutes! If not, don't worry, a screencast of this exercise is available on the book's website.

4: How Automated Testing works

This content is not available in the sample book.

4.1: Web test automation rhythm

This content is not available in the sample book.

4.1.1: Identify web elements

This content is not available in the sample book.

4.1.2: Interact with web elements

This content is not available in the sample book.

4.1.3: Verify the Result

This content is not available in the sample book.

4.2: Test syntax frameworks

This content is not available in the sample book.

4.2.1: unittest

This content is not available in the sample book.

4.2.2: pytest

This content is not available in the sample book.

4.2.3: behave (not recommended)

This content is not available in the sample book.

4.3: pytest execution hooks

This content is not available in the sample book.

4.3.1: pytest's xUnit-style setup methods

This content is not available in the sample book.

4.3.2: pytest fixtures

This content is not available in the sample book.

4.4: Run pytest tests from the command line

This content is not available in the sample book.

4.4.1: Test execution types

This content is not available in the sample book.

4.4.2: pytest Command-Line Options

This content is not available in the sample book.

4.5: Review

This content is not available in the sample book.

5: Test Tool

For more than three decades, QA engineers have experimented with commercial record-and-playback test automation tools that promised to simplify test automation. While these tools made it easy to create the initial test, they consistently struggled with real-world applications, especially long-term maintenance. As a result, most commercial test automation vendors faded, while web test automation became dominated by free and open-source frameworks such as Selenium WebDriver and Playwright, which require QA engineers to write scripts using a programming language.

Yet, for newcomers and many teams, manually writing automated test scripts remains challenging. Syntax errors are easy to introduce, and debugging them in code editors can be time-consuming and frustrating. Some readers may have already experienced these challenges while completing the exercises in previous chapters.

This is exactly where modern purpose-built test automation tools come in to help.

“A workman is only as good as his tools.” – English proverb

5.1: TestWise: Next-Gen Functional Testing IDE

After Watir sparked a new revolution in web test automation through an open-source framework, software engineers typically wrote automated tests using programming editors or coding IDEs. While these tools were powerful, they were not designed specifically for test automation and often proved inefficient for developing and maintaining automated tests.

Recognizing this need, my father began developing his own test automation IDE in his spare time in 2006. He named it **iTest2** (later renamed to **TestWise**), reflecting his goal of making automated testing simple and easy enough for everyone to use, including non-technical team members.

iTest2 was successfully adopted by several development teams. One notable example was a large government project, where almost the entire project team, including business analysts, used iTest2 daily.

Apparently, other testing experts also recognized the need for this kind of tool.

Next-Gen Functional Testing Tool

Agile Alliance workshop [AAFTT07] on Next-Generation Functional Testing (Oct 2007), Envision “We are lacking IDE that facilitate things like:

- refactoring test elements
- command completion
- incremental syntax validation
- keyboard navigation
- debugging ...“

Most of the above highlighted next-generation functional testing features had already been implemented in iTest2.

- In the presentation “[Trends in Agile Testing](https://www.slideshare.net/slideshow/trends-in-agile-testing-by-lisa-crispin/2016980#24)”¹ (2009) by Lisa Crispin, co-author of the Agile Testing book, iTest2 was listed as the first testing tool in the “Testing Tool Spectrum” slide (#24).

¹<https://www.slideshare.net/slideshow/trends-in-agile-testing-by-lisa-crispin/2016980#24>

- iTest2 was demonstrated at Agile 2009 AA-FFT in Chicago, and listed in Ward Cunningham (co-author of Agile Manifesto)'s [Functional Testing Tools list](#)².

Mike Longin, an Agile 2009 speaker, after seeing the demonstration, wrote in a blog post: “iTest2 was also shown off and really goes a long way in my mind to show off the capabilities of an IDE for functional testing tools. As the project lead for SWAT, one of my biggest pushes has been to take the tools I get in Visual Studios and bring them to the functional testing world. At least for Watir, iTest2 is there.”

- A finalist for the 2010 Ruby Award in Japan, judged by the software legend Matz.

「iTest2 - Automated Web Testing made easy」

◆ Zhimin Zhan (オーストラリア)

webアプリケーションを簡単にテストできるシステム。

In E2E test automation, many senior automation engineers and architects are more inclined to abandon their existing test suites and chase the next new automation framework than to settle down, build a maintainable test suite, and deliver lasting value.

Over the past two decades, numerous test automation frameworks have come and gone. According to [two independent industry surveys of QA engineers](#)³ conducted in December 2023 and January 2024 on the choice of frameworks for automated web application testing, Selenium WebDriver (first released in 2011) ranked first in adoption, with Playwright ranking second.

²<https://github.com/WardCunningham/aافتt/blob/master/tool-features.xls>

³<https://agileway.substack.com/p/selenium-webdriver-is-still-the-best>

TestWise is a dedicated functional testing IDE built for hands-on QA engineers. It enables developers and testers to create, refine, refactor, and debug Selenium WebDriver and Playwright tests in multiple programming languages, helping teams build maintainable automation suites with maximum productivity.

5.1.1: TestWise Project

TestWise organizes test scripts into projects. A project is simply a folder that contains all test-related files, such as test scripts, test data, and other supporting resources.

5.1.2: Create a new test project

Select menu **File** -> **New Project**, which will bring up the window shown below.

New Project

Project name: AgileTravel

Location: /Users/me/testprojects/AgileTravelTests ...

Enter or select an empty folder for the test project.

Test Automation Frameworks

Automation Driver: ☐ Selenium-WebDriver ☐ Appium ☐ Watir ☒ Playwright ☐ API

Test Script Syntax: ☐ RSpec ☒ Pytest ☐ Mocha ☐ Cucumber ☐ Playwright Test

Project Settings

☒ Website URL ☐ App Executable ☐ Bundle ID

https://travel.agileway.net

☒ Create project skeleton

☐ Include the helper for load testing

OK Cancel

- **Project Name:**

A descriptive name for the project. Keep it short and avoid special characters.

- **Project folder:**

The absolute path to an empty folder where the test project will be created.

- **Automation Driver**

Select the automation framework to use. TestWise supports multiple frameworks, including Selenium WebDriver, Appium, and Playwright.

- **Test Script Syntax**

Select the test syntax framework, which also determines the scripting language. For example, choosing pytest uses Python, while RSpec uses Ruby.

- **Target website URL**

Enter the base URL of the application under test.

For example, use `https://agileway.net` rather than `https://agileway.net/login`.

Click the **OK** button. TestWise will create the test project with the required project skeleton files and then open the newly created project.

5.1.3: Test project skeleton

The following is a brief overview of the test project structure automatically generated by TestWise.

```
project/
├── tests/
│   └── new_test.py
├── pages/
│   ├── __init__.py
│   └── abstract_page.py
├── buildwise.rake
├── test_helper.py
├── abstract_test.py
├── conftest.py
└── {{project_name}}.tpr
```

- **tests: Test Scripts**

Stores the test script files. By convention, pytest test files are named either `test_xxx.py` OR `xxx_test.py`.

- **pages: Page classes**

Contains the Page Object Model (POM) classes, a test design pattern that will be introduced in Chapter 9. Page class files follow the naming convention `xxx_page.py`.

`pages/abstract_page.py` defines the base page class from which all page classes inherit.

The special `pages/__init__.py` exports the page classes for convenient importing (to use in test scripts).

- **test_helper.py**

Contains reusable helper functions shared across test scripts.

- **conftest.py**

Custom pytest configuration.

- **abstract_test.py**

Defines the base test class from which all test classes inherit.

- **buildwise.rake**

The build configuration file for integration with the BuildWise Continuous Testing (CT) Server.

- **PROJECT_NAME.tpr**

The TestWise project file. Open this file in TestWise to load the test project.

Important note: Although this project skeleton is created by TestWise, it does not introduce any dependencies on TestWise. The generated test scripts are standard Playwright and pytest code, and they can be executed using a normal Playwright-pytest environment.

5.1.4: Test Project Structure Setup is often mis-referred as “Automated Test Framework”

You may have seen “*designing or creating a test automation framework*” listed as a requirement in test automation job advertisements. Of course, employers are not expecting you to build an entirely new automation framework like Selenium WebDriver or Playwright from scratch!

In reality, this usually refers to creating a test automation project structure.

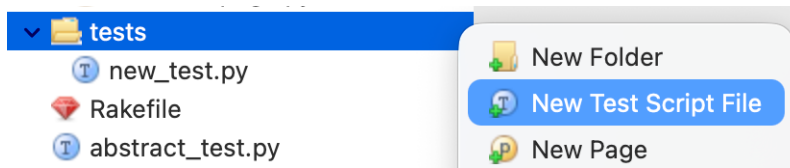
A good test automation project structure incorporates proven practices such as reusable test helpers, the Page Object Model (POM), tests organization, and integration with Testing IDEs and Continuous Testing tools. Some of these topics will be covered in later chapters.

The good news is that TestWise already provides one proven web test automation project structure out of the box, for free.

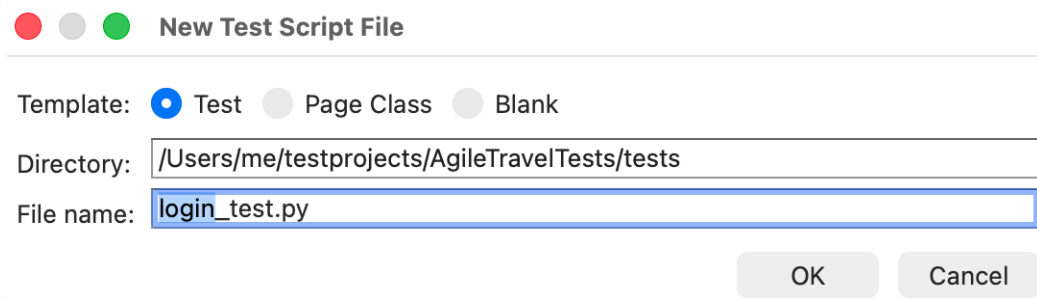
5.1.5: Create a test script file

In Chapter 3, we started with a pre-created sample test script. Now, let's create a brand-new test script from scratch.

In the Project Explorer, right-click the `tests` folder and select “**New Test Script File**”.



A dialog will appear. Enter “`login_test.py`” in the ‘File name:’ field and press Enter. TestWise will create a new test script file: `/{{Project}}/tests/login_test.py`.



Tip: Give your test script a name that reflects the feature or requirement being tested. This makes it much easier to locate later.

A new editor tab will open with the newly created test script.

```

4     from test_helper import TestHelper
5     from pages import *
6
7     class TestLogin(AbstractTest, TestHelper):
8
9         # before(:all)
10        @classmethod
11        def setup_class(cls):
12            super().setup_class()
13            cls.page.goto(TestHelper.site_url())
14            # TestHelper.sign_in(cls, "agileway", "test$Wise")
15
16        # before(:each)
17        def setup_method(self):
18            # adding steps before each test case, after added,
19            pass
20
21        def test_case_one(self):
22            # example: page.fill("#username", "agileway")

```

The new test script contains the same skeleton as the previous-seen sample test. TestWise automatically generates the basic test structure (*which can be customized in the project settings*), allowing you to start writing the test logic quickly.

Go ahead and run the test script you have just created. It will simply open the target website.

5.1.6: Run individual test case

So far, you might have run the test script file in TestWise (▶ on the toolbar) several times. A test script file can contain one or more test cases.

In **pytest**, a test case is defined as a top-level function or a method inside a test class. The function name must start with **test_**, as shown in the example below:

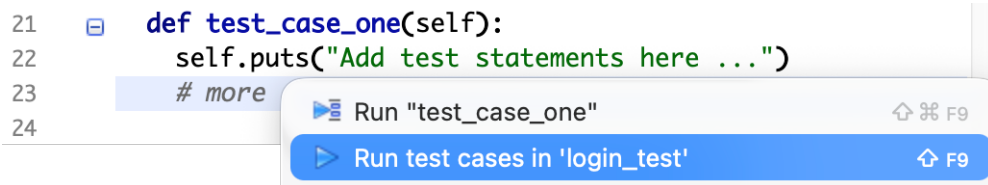
```
def test_{{case_name}}(self):  
    # test steps  
    # example: page.fill("#username", "agileway")  
    # ...
```

Here are examples of two cases in one test script file.

```
21  def test_case_one(self):  
22      self.puts("Add test statements here ...")  
23      # more  
24  
25  def test_case_two(self):  
26      page.fill("#username", "agileway")  
27      page.fill("#password", "test$Wise")
```

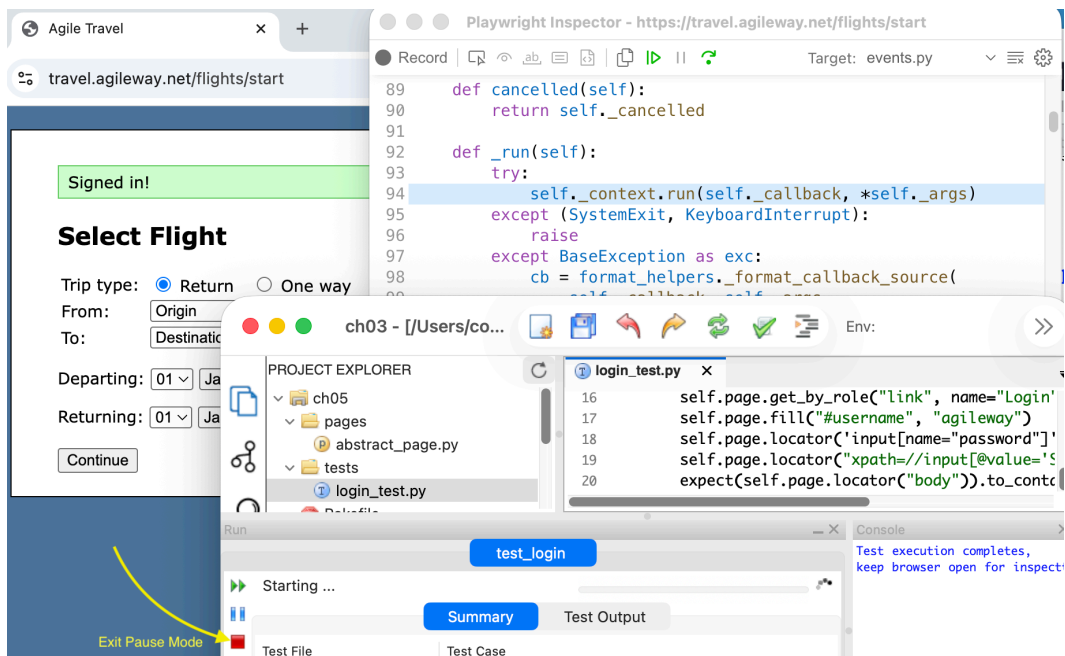
During development, refinement, and debugging, we usually work with individual test cases rather than running the entire test script file.

Right-click a line within the test case (line 21 to 23), then select 'Run "{{test_case_name}}"'.
name}}'.



5.1.6.1: Browser window kept open in individual test case execution mode

After executing a single test case in TestWise, you will notice one key difference: the browser window remains open after test execution completes, only in individual test case execution mode with TestWise.



The implementation is simple: TestWise instructs Playwright to enter a “**Pause state**”, keeping both the browser window and the Playwright Inspector window open.

While in this state, you can continue interacting with the application manually in the browser, if necessary. Once you have finished your inspection or debugging, click the Stop button (*the square icon*) in the Run panel. This will close both the browser window and the Playwright Inspector.

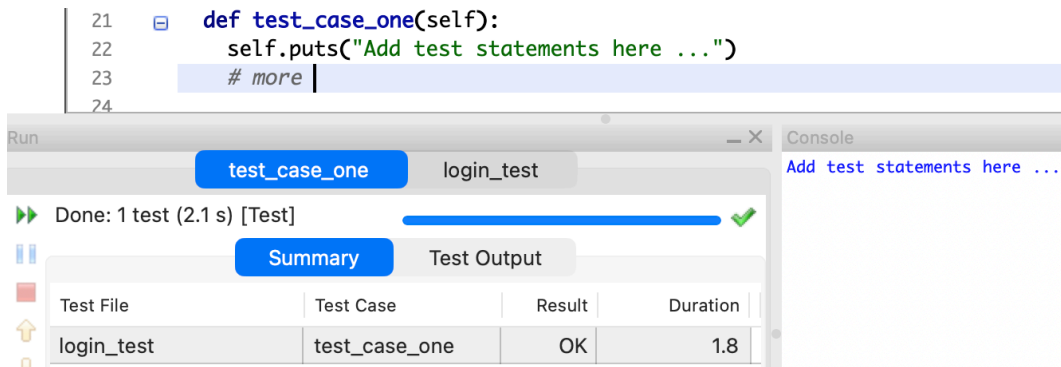
In normal test execution, the expected behaviour is for the browser to close automatically once the test finishes, allowing the next test run to start with a clean environment.

However, when working on test scripts inside a testing IDE, QA engineers are typically developing new tests, updating or refactoring existing tests, or debugging failures. In these situations, keeping the browser open provides valuable visibility into the execution result and allows QA engineers to manually take control of the browser to inspect the state, verify behaviour, and continue troubleshooting.

For software testers, this behaviour is intuitive and practical. After manually running a test case, or even executing a specific test step, they naturally want to inspect the browser state, verify the outcome, understand any issues, and quickly continue improving the test.

5.1.6.2: Test Result

Below is the execution output panel looks like in TestWise.



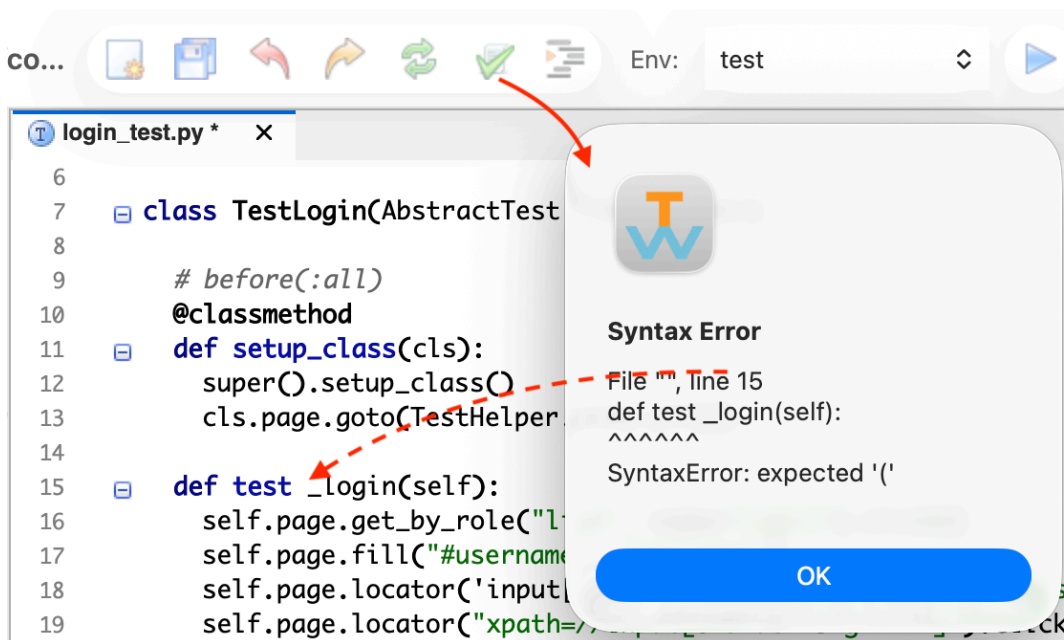
Astute readers may notice two new things:

1. The heading of the execution tab (in TestWise) is different when running an individual test case. It shows the test case name, while running the entire test script shows the test script file name.
2. The test output message, “*Add test statement here...*”, appears in the TestWise console panel. The standard Python `print()` function won't reach there. This is done using a wrapper function named `puts()`, which is defined in `TestHelper`.

5.1.7: Syntax Validation

For beginners, it is easy to make typos in the test script (e.g. spelling mistake or missing parentheses). All program editors have syntax highlighting (colour coding to help find issues easily), and TestWise has a button on the top bar to check the syntax as well.

If the test script in the current editor is invalid, a pop-up window will appear showing the validation results.



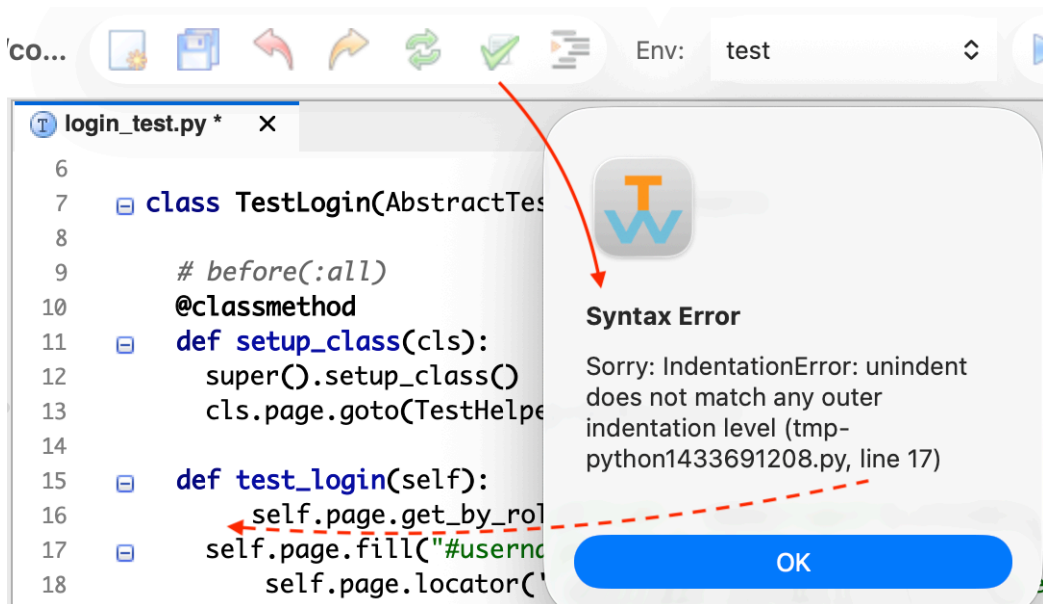
For beginners, if nothing happens after pressing the Run button, try validating the test script first.

5.1.8: Indenting

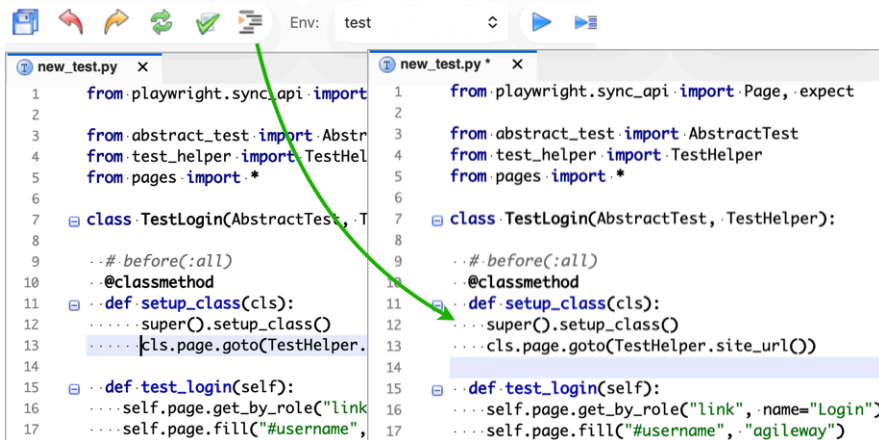
Unlike many programming languages, Python is strict about indentation. Because indentation is part of Python's syntax, incorrectly indented code is treated as a syn-

tax error. This can be frustrating for engineers coming from other programming backgrounds.

TestWise's syntax validation tool also detects indentation syntax errors.



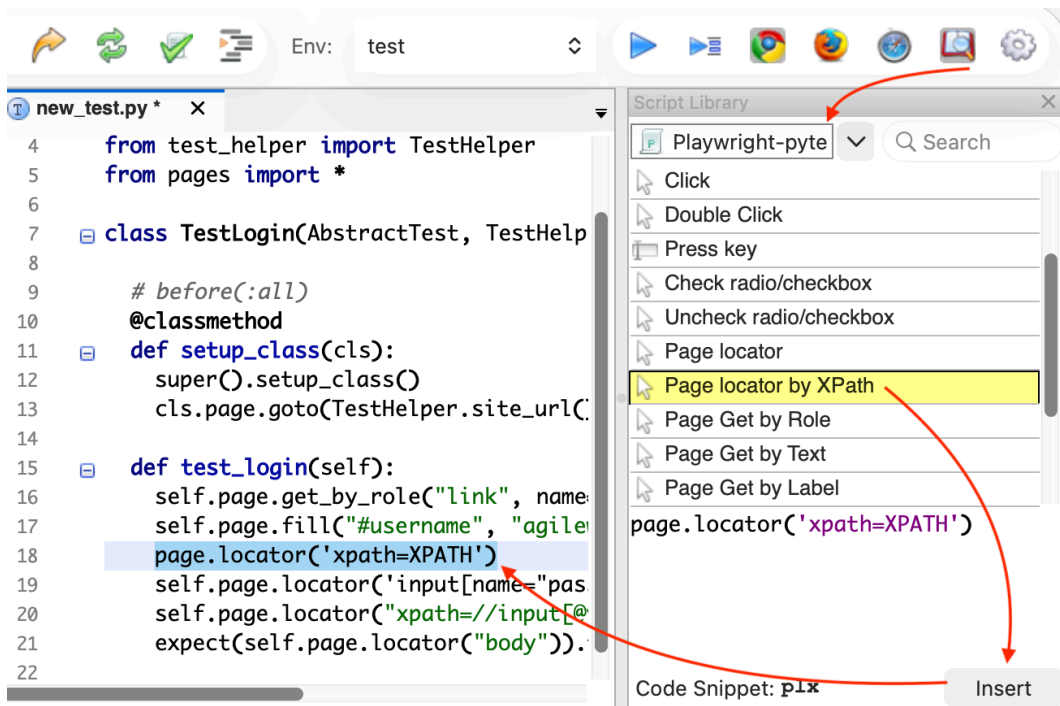
Even with valid Python scripts, inconsistent indentation can still occur. In such cases, the TestWise's reindent tool can help reformat the code and maintain consistent indentation.



5.1.9: Script Library

When beginners practise automated test exercises on their own, they often encounter problems such as forgetting the correct syntax.

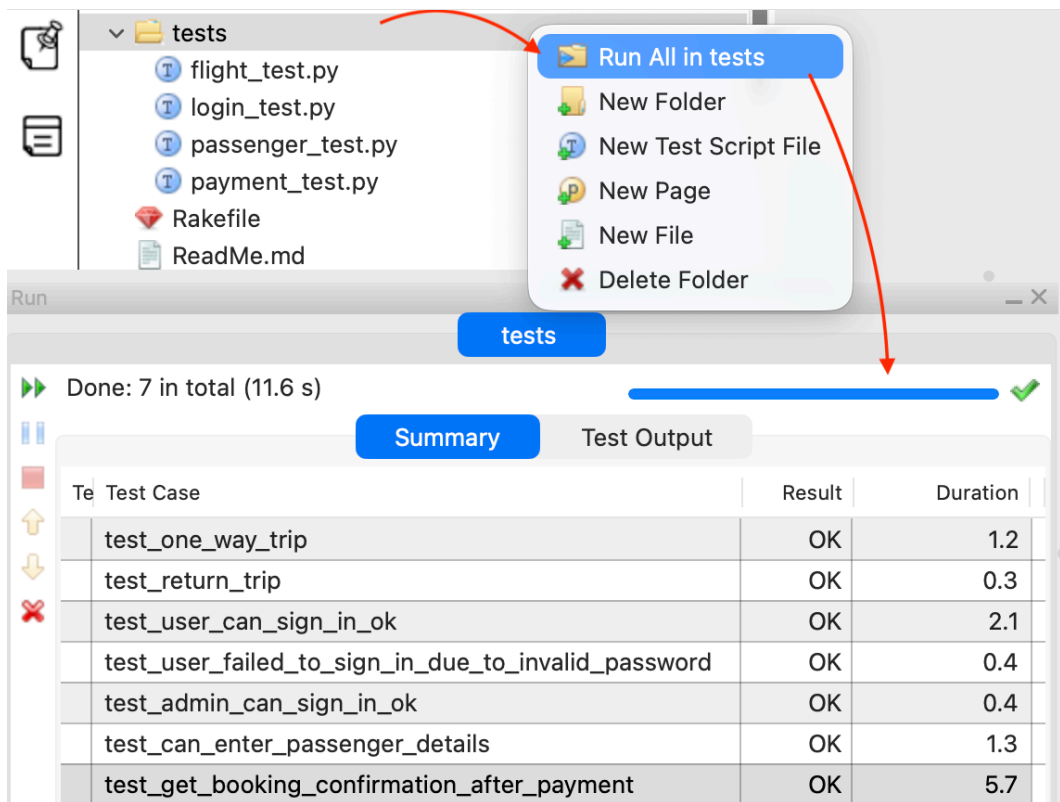
One way to overcome this is to use the **Script Library** in TestWise to quickly look up commonly used statements.



Click the “**Insert**” button to add the selected statement into the editor (test script).

5.1.10: Run all test in tests folder

As you implement more tests and gradually build your test suite, you may occasionally need to run the entire suite. In TestWise, you can right-click the tests folder in the left *Project Explorer* and select “**Run All in tests**”.



This execution mode works well for small test suites. However, for larger suites, it is not recommended to run everything manually from the IDE. Instead, you should execute the suite through a Continuous Testing server. Chapter 12 covers this approach in detail.

5.2: Visual Studio Code Programming Editor

Playwright + pytest scripts are written in Python. While a dedicated test automation IDE such as **TestWise** provides many productivity features for developing and debugging automated tests, you can also use a general-purpose programming editor. One of the most popular choices is **Visual Studio Code** (VS Code).

In this section, I will briefly show you how to set up Visual Studio Code to develop and run Playwright + Pytest scripts.

5.2.1: Installation

If you have already installed Python, Playwright and pytest as described in the previous sections, the only additional software you need is Visual Studio Code.

You can download Visual Studio Code for Windows, macOS, or Linux from: <https://code.visualstudio.com/download>.

5.2.1.1: VS Code Extensions

Open the **Extensions** view in Visual Studio Code and install the following extensions:

- [Python](#)⁴
- [Playwright](#)⁵

After installing the extensions, open the **Command Palette** (Ctrl+Shift+P on Windows/Linux or Cmd+Shift+P on macOS), search for **Test: Install Playwright**, and run the command.

For more information about Python testing in VS Code, see [here](#)⁶.

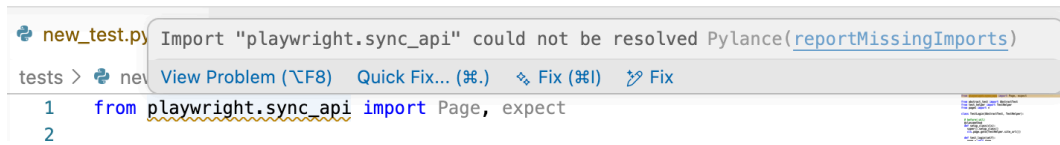
5.2.1.2: Select the Python interpreter

After the extensions are installed, you may notice warning messages in your Playwright test scripts.

⁴<https://marketplace.visualstudio.com/items?itemName=ms-python.python>

⁵<https://marketplace.visualstudio.com/items?itemName=ms-playwright.playwright>

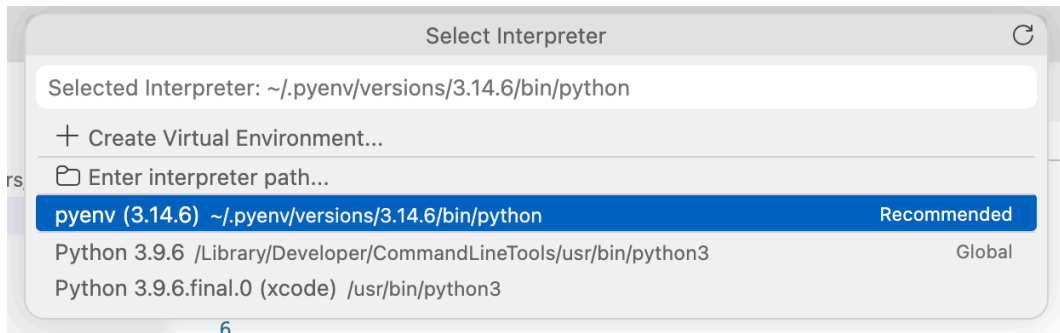
⁶<https://code.visualstudio.com/docs/python/testing>



This usually means Visual Studio Code has selected the wrong Python interpreter. By default, VS Code often chooses the system Python installation. I recommend selecting the Python interpreter that you installed earlier in this book.

To change the interpreter:

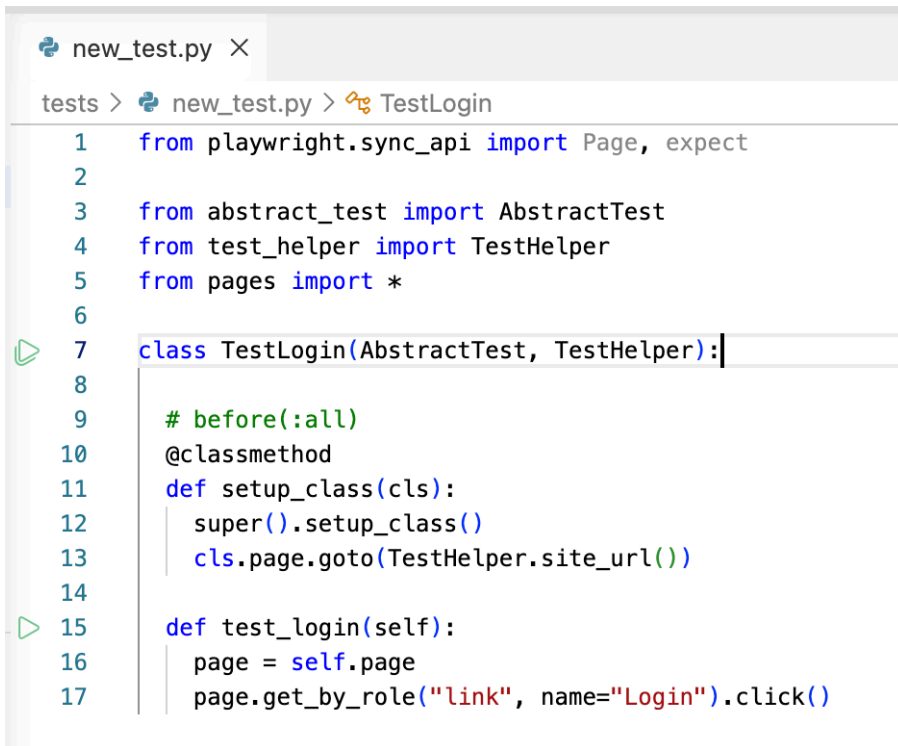
1. Open the Command Palette.
2. Search for “Python: Select Interpreter”.
3. Select the Python interpreter that contains your Playwright and Pytest installation.



Once the correct interpreter is selected, the warnings should disappear.

5.2.1.3: Run Tests

After the setup is complete, Visual Studio Code automatically detects your Pytest test cases. You will see green Run Test icons next to individual test methods and test classes.



```
new_test.py X
tests > new_test.py > TestLogin

1  from playwright.sync_api import Page, expect
2
3  from abstract_test import AbstractTest
4  from test_helper import TestHelper
5  from pages import *
6
7  class TestLogin(AbstractTest, TestHelper):
8
9      # before(:all)
10     @classmethod
11     def setup_class(cls):
12         super().setup_class()
13         cls.page.goto(TestHelper.site_url())
14
15     def test_login(self):
16         page = self.page
17         page.get_by_role("link", name="Login").click()
```

- Click the green Run Test icon beside a test method to execute that test only.
- Click the Run Test icon beside a test class to execute all tests in that class.

When a test starts, the **Test Results** pane opens automatically. The left panel displays the console output produced during the test run, while the right panel shows the final test results.

The screenshot shows the Visual Studio Code editor with a file named `new_test.py` open. The code defines a `TestLogin` class that inherits from `AbstractTest` and `TestHelper`. It includes a `before(:all)` classmethod `setup_class` and a `test_login` instance method. The `test_login` method uses `self.page` to click a link with the role 'link' and name 'Login'.

Below the editor, the `TEST RESULTS` panel is active, showing the output of running the tests. The output indicates that the tests passed successfully.

```

with-playwright-pytest/sources/ch03-test_project', '/Users/
courtney/work/books/practical-web-test-automation-with-play
wright-pytest/sources/ch03-test_project/tests/new_test.py::
TestLogin::test_login']
===== test session starts =====
platform darwin -- Python 3.14.6, pytest-9.1.1, pluggy-1.6.
0
rootdir: /Users/courtney/work/books/practical-web-test-autom
ation-with-playwright-pytest/sources/ch03-test_project
collected 1 item

tests/new_test.py .
[100%]

===== 1 passed in 4.22s =====
Finished running tests!

```

The status bar at the bottom shows the file is at line 7, column 43, with 2 spaces, UTF-8 encoding, LF line endings, and is a Python file using the `pyenv (3.14.6)` interpreter.

Although Visual Studio Code is a powerful programming editor with excellent editing and debugging capabilities, it is a general-purpose development tool. As your Playwright test suite grows, you may find that a dedicated test automation IDE can significantly improve your productivity by providing features specifically designed for developing, debugging and maintaining automated tests.

5.3: PyCharm Development IDE

PyCharm is a popular IDE for Python, created by JetBrains. It is free to download and use (there is a professional license too, but for automated testing, the free version is good enough). PyCharm includes excellent built-in support for Pytest,

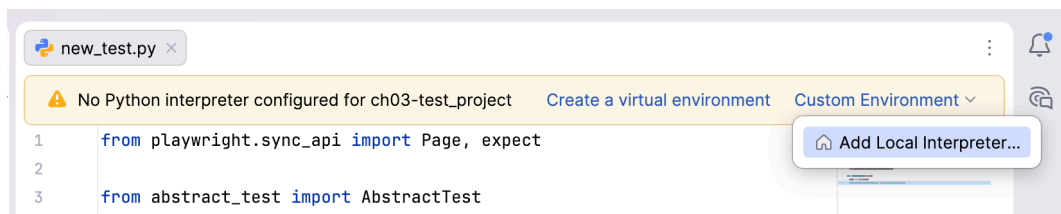
and Playwright integration is available through a plugin, making it a convenient environment for developing and running automated tests.

Download and install PyCharm: <https://www.jetbrains.com/pycharm/download>.

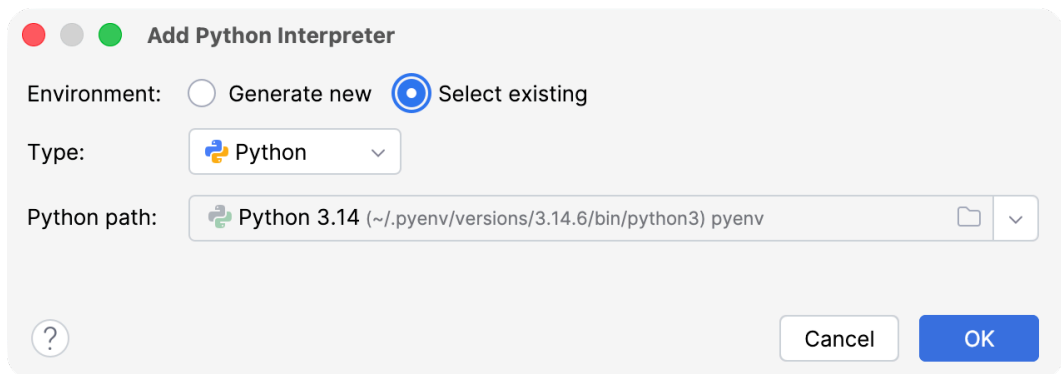
Follow the installation wizard.

5.3.0.1: Set up Python Interpreter

After you open an existing project, it will prompt you to select a Python interpreter.

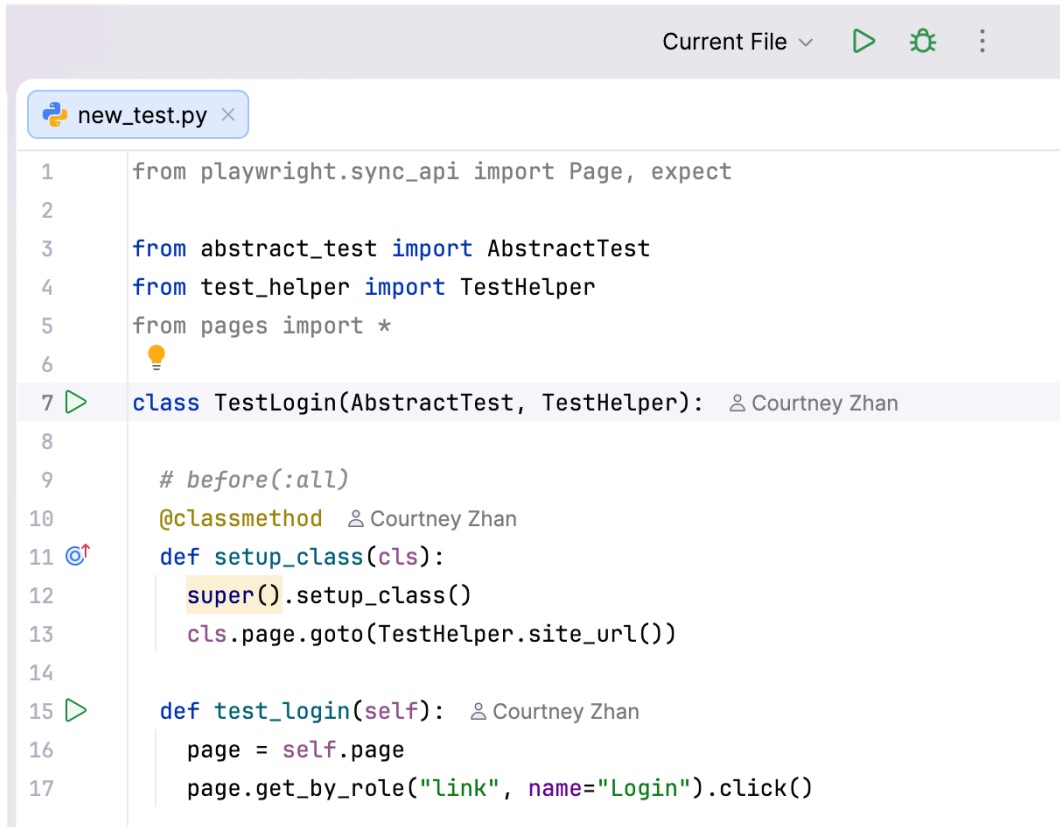


It will give you the option of having PyCharm generate (download and install) a version for you, or you can choose an existing one. I recommend choosing the existing interpreter we installed earlier from pyenv.



5.3.0.2: Run Tests

After the set up is completed, you will see the green “play” button next to the test suite and test cases:



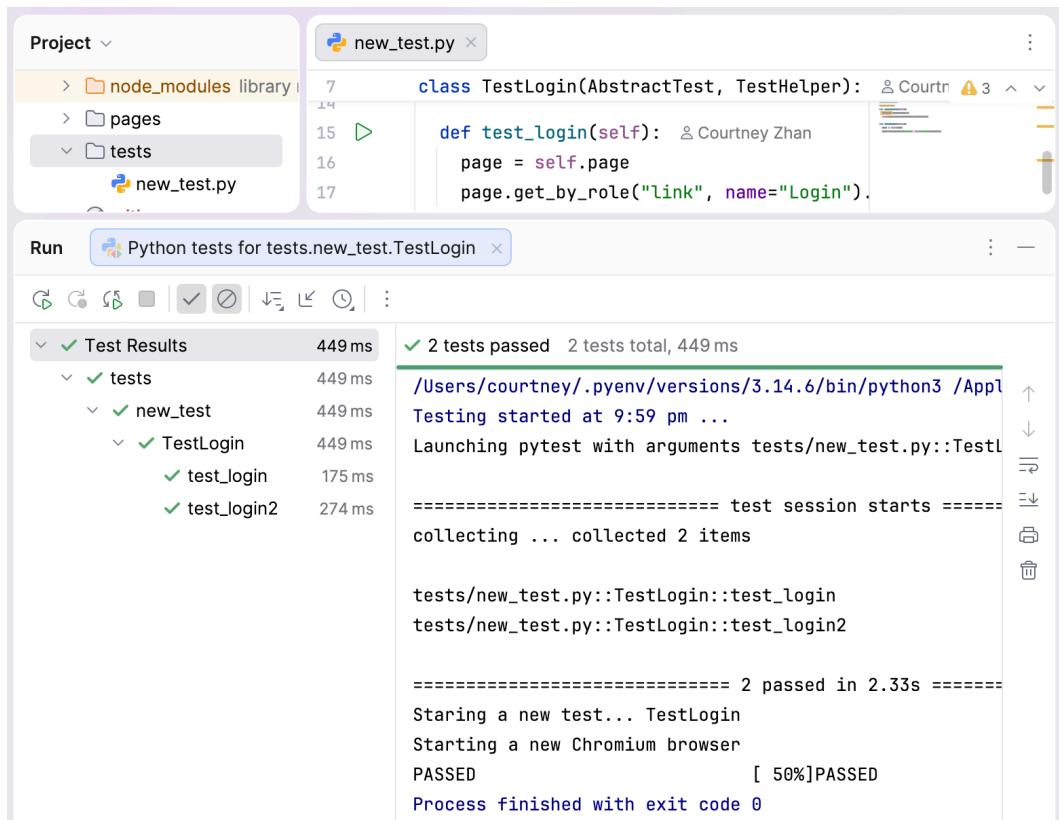
```
Current File ▾ ▶ ⚙ ⋮

new_test.py ×
1 from playwright.sync_api import Page, expect
2
3 from abstract_test import AbstractTest
4 from test_helper import TestHelper
5 from pages import *
6
7 ▶ class TestLogin(AbstractTest, TestHelper):  ⓘ Courtney Zhan
8
9     # before(:all)
10    @classmethod  ⓘ Courtney Zhan
11    def setup_class(cls):
12        super().setup_class()
13        cls.page.goto(TestHelper.site_url())
14
15    ▶ def test_login(self):  ⓘ Courtney Zhan
16        page = self.page
17        page.get_by_role("link", name="Login").click()
```

- **To run a specific test** Click the **green play** button next to the test name. Or right-click inside the test case itself and select the “**Run ‘Python tests for file_name.TestSuiteName.test_case_name’**” option.
- **To run all the tests in the class** Click the **green play** button next to the class. Alternatively, there is a play button at the top of the IDE to run all the tests in the file. Or, you can right-click anywhere inside the test file

(but not within a specific test case) and select the **“Run ‘Python tests for file_name.TestSuiteName’”** option.

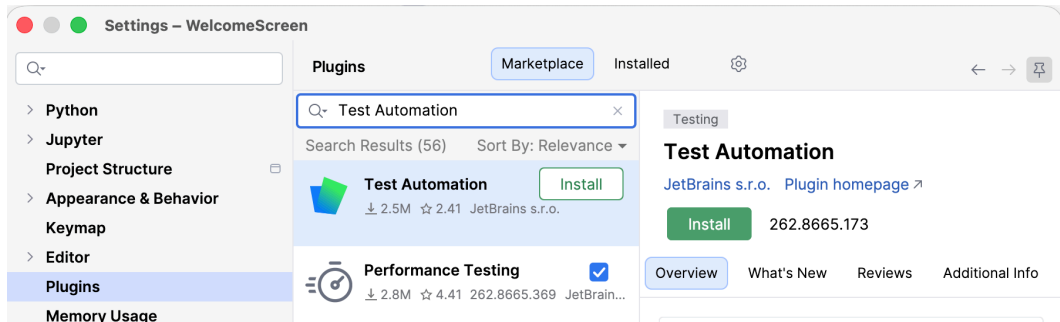
PyCharm will open the **Run tool** window, where you can view the test results, console output, and execution time. The test summary is displayed on the left, while the console output and timing information appear on the right.



5.3.0.3: Creating a Playwright Project

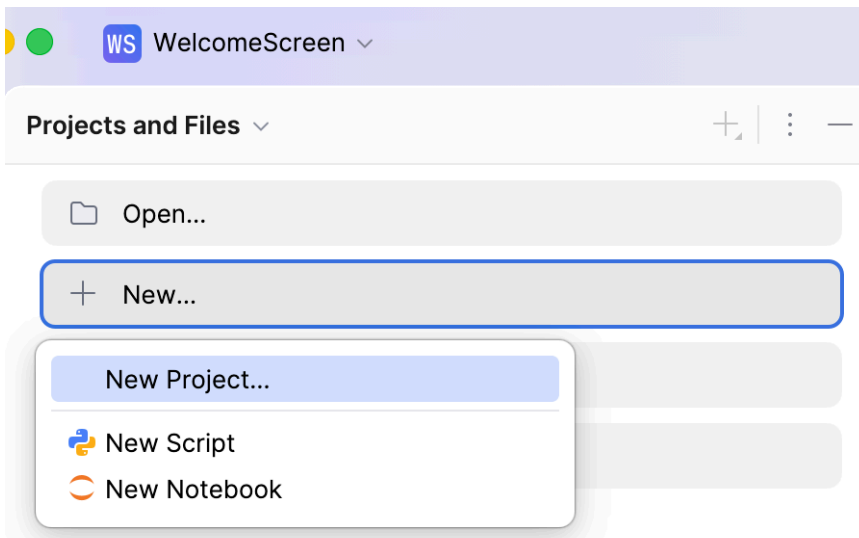
From PyCharm, you can also create Playwright projects from scratch.

First, install the Test Automation plugin. Open PyCharm, then open Settings (or Cmd + ,), then click on “Plugins” > “**Marketplace**”. Search for the “**Test Automation**” plugin offered by Jetbrains⁷ and install it.



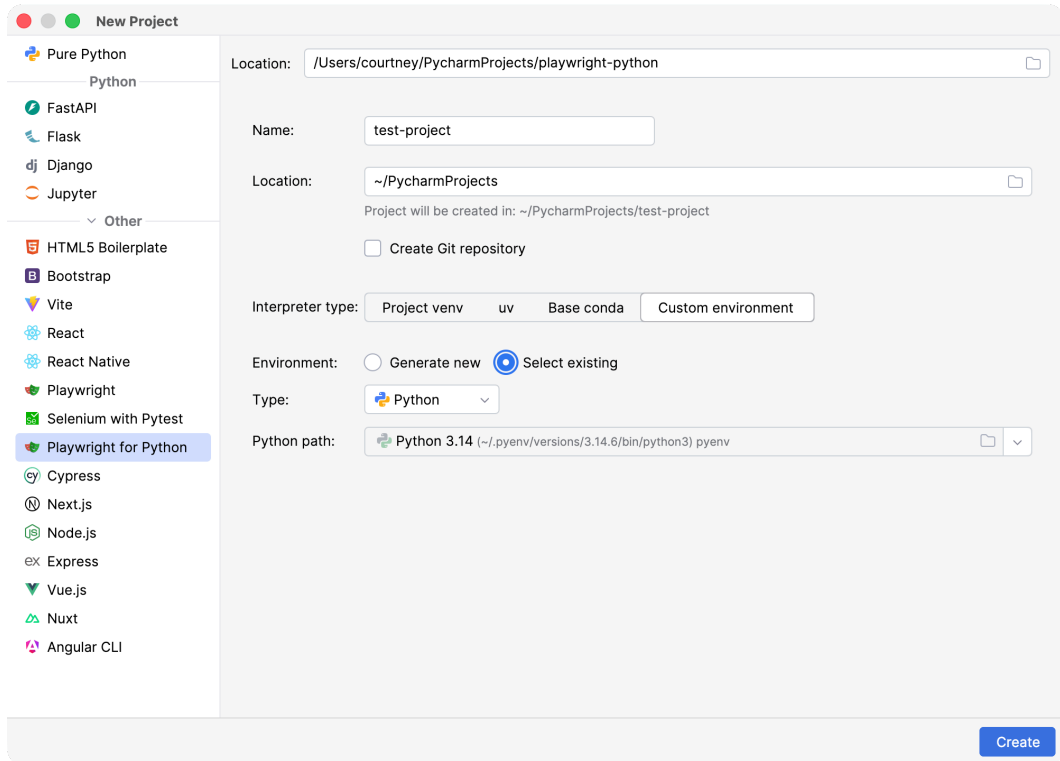
After it installs, follow the prompt to restart your IDE.

Then, on the PyCharm welcome page, click “+ **New**” to create a new project.



⁷(<https://plugins.jetbrains.com/plugin/20175-test-automation>)

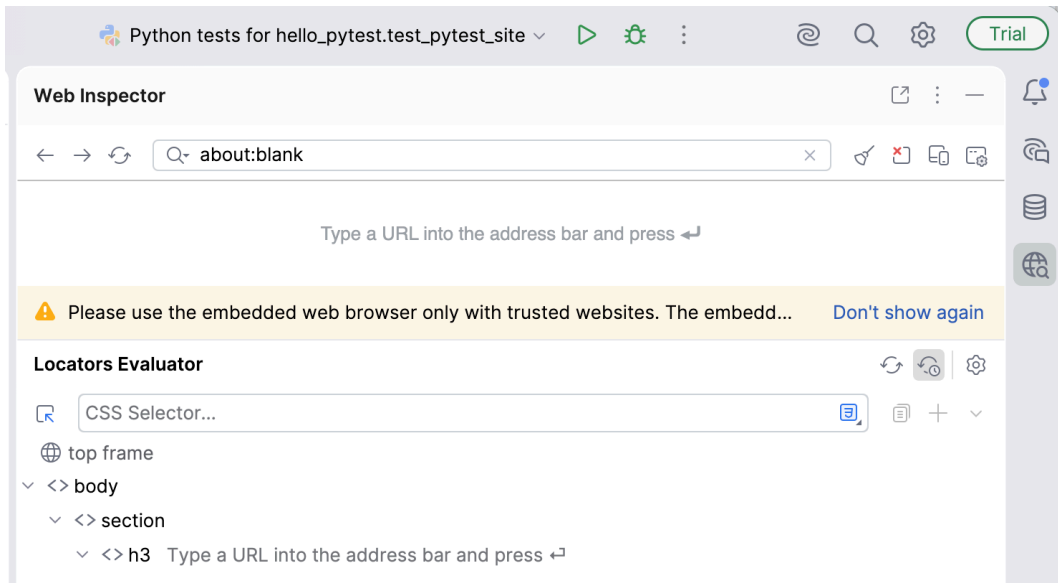
Open the “Other” project types on the left pane and select “**Playwright for Python**”. I recommend to use the Python version installed earlier via pyenv.



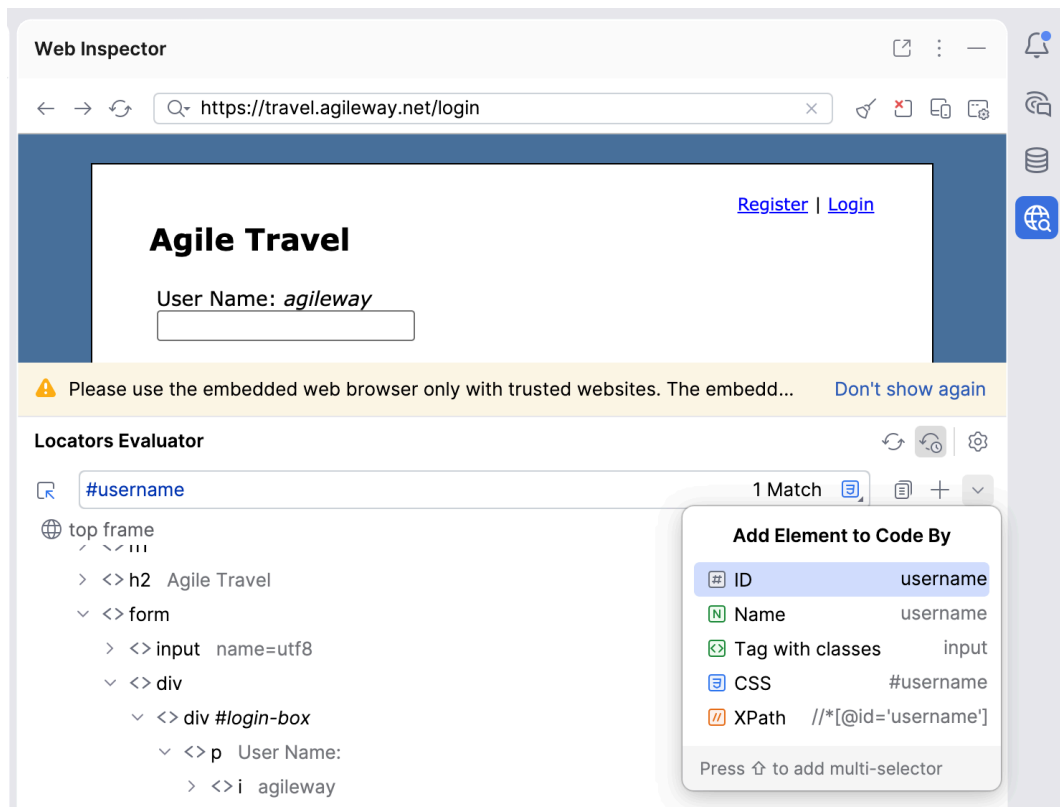
It will create a blank PyCharm project for you, so you still need to create the project structure yourself.

5.3.0.4: PyCharm Web Inspector

The PyCharm Playwright project comes with a Web Inspector that can help identify locators. Open it from the left hand pane (the globe icon with the magnifying glass), or open with the toolbar's View > Tool Windows > Web Inspector.



It's almost identical to a browser inspector, but has tools to help auto insert locator statements. Enter the URL in the browser pane, then from the CSS locator pane, click on an item, and use the plus button's drop down to find and insert an appropriate locator (or, use the plus button directly to have PyCharm pick the best locator and insert it for you).



5.4: Review

In this chapter, I introduced three types of tools that can help you develop Playwright pytest scripts more efficiently:

- Dedicated test automation IDE: AgileWay's **TestWise**
- Programming editor: Microsoft's **Visual Studio Code**
- Python IDE: JetBrains **PyCharm**

These are by no means the only choices. All three are free or inexpensive, so cost is rarely the deciding factor. What matters is choosing the tool that best fits the

way you work and helps you stay productive. For small projects, almost any editor will do. However, as your E2E test suite grows, the right development environment can make a significant difference in how quickly you develop, debug, and maintain your test scripts.

6: Playwright Syntax in Nutshell

This content is not available in the sample book.

6.1: Browser, Context, and Page

This content is not available in the sample book.

6.1.1: Relationship

This content is not available in the sample book.

6.2: Playwright locators

This content is not available in the sample book.

6.3: `get_by_role()`

This content is not available in the sample book.

6.3.1: Playwright role types

This content is not available in the sample book.

6.3.2: Examples

This content is not available in the sample book.

6.4: `get_by_label()`

This content is not available in the sample book.

6.5: `get_by_placeholder()`

This content is not available in the sample book.

6.6: `get_by_text()`

This content is not available in the sample book.

6.6.1: Using an Exact Text Match

This content is not available in the sample book.

6.6.2: Examples

This content is not available in the sample book.

6.7: `get_by_test_id()`

This content is not available in the sample book.

6.8: `locator()` with CSS/XPath

This content is not available in the sample book.

6.9: Page-level convenience methods

This content is not available in the sample book.

6.9.1: `fill()`

This content is not available in the sample book.

6.9.2: `click()` and `dblclick()`

This content is not available in the sample book.

6.9.3: `check()` and `uncheck`

This content is not available in the sample book.

6.9.4: `select_option()`

This content is not available in the sample book.

6.9.5: `hover()` and `focus()`

This content is not available in the sample book.

6.9.6: `press()`

This content is not available in the sample book.

6.9.7: `press_sequentially()`

This content is not available in the sample book.

6.9.8: `drag_and_drop()`

This content is not available in the sample book.

6.9.9: `set_input_files()`

This content is not available in the sample book.

6.10: Locator chaining

This content is not available in the sample book.

6.11: Multiple matching elements

This content is not available in the sample book.

6.12: Playwright vs Selenium WebDriver

This content is not available in the sample book.

6.13: Review

This content is not available in the sample book.

7: Gain Productivity

This content is not available in the sample book.

7.1: Playwright CodeGen

This content is not available in the sample book.

7.2: Playwright Trace Viewer

This content is not available in the sample book.

7.3: Some Generic IDE Features

This content is not available in the sample book.

7.3.1: Snippet: faster (and safer) Typing Playwright statements

This content is not available in the sample book.

7.3.2: Auto Complete

This content is not available in the sample book.

7.3.3: Refactoring

This content is not available in the sample book.

7.4: Keyboard shortcuts

This content is not available in the sample book.

7.5: More TestWise productivity features

This content is not available in the sample book.

7.5.1: Run tests against different browsers

This content is not available in the sample book.

7.5.1.1: How it works

This content is not available in the sample book.

7.5.2: Run tests against different server environments

This content is not available in the sample book.

7.5.2.1: How It Works?

This content is not available in the sample book.

7.6: Review

This content is not available in the sample book.

8: Case Study

This content is not available in the sample book.

8.1: Test site

This content is not available in the sample book.

8.2: Preparation

This content is not available in the sample book.

8.3: Create a test project

This content is not available in the sample book.

8.4: Test Suite: Sign in

This content is not available in the sample book.

8.4.1: Positive Case: User can sign in OK

This content is not available in the sample book.

8.4.1.1: Test Case Design

This content is not available in the sample book.

8.4.1.2: Typing the test steps in TestWise Editor

This content is not available in the sample book.

8.4.2: Negative Case: User failed to sign in due to invalid password

This content is not available in the sample book.

8.4.3: Run all test cases in the login_test.py

This content is not available in the sample book.

8.5: Test Suite: Select Flights

This content is not available in the sample book.

8.5.1: Case 1: One-way trip

This content is not available in the sample book.

8.5.2: Case 2: Return trip

This content is not available in the sample book.

8.5.3: Technique: use execution hooks

This content is not available in the sample book.

8.5.4: Technique: check dynamic UI

This content is not available in the sample book.

8.6: Enter passenger details

This content is not available in the sample book.

8.7: Book confirmation after payment

This content is not available in the sample book.

8.7.1: Technique: Testing AJAX

This content is not available in the sample book.

8.7.1.1: What does this experiment tell us?

This content is not available in the sample book.

8.7.2: Technique: Displaying value from specific HTML element in console

This content is not available in the sample book.

8.8: Run all tests

This content is not available in the sample book.

8.9: Review

This content is not available in the sample book.

9: Maintainable Functional Test Design

This content is not available in the sample book.

9.1: Record/Playback leads to unmaintainable test scripts

This content is not available in the sample book.

9.1.1: Beginners: Record → Refine → Run

This content is not available in the sample book.

9.2: Success criteria

This content is not available in the sample book.

9.2.1: Intuitive to read

This content is not available in the sample book.

9.2.2: Easy to update

This content is not available in the sample book.

9.3: Maintainable automated test design

This content is not available in the sample book.

9.3.1: Reusable function

This content is not available in the sample book.

9.3.2: Page Object

This content is not available in the sample book.

9.4: Maintain with ease

This content is not available in the sample book.

9.5: Case Study: refine test scripts

This content is not available in the sample book.

9.5.1: DRY with Reusable Functions

This content is not available in the sample book.

9.5.1.1: Parameterizing functions

This content is not available in the sample book.

9.5.2: DRY with Page Objects

This content is not available in the sample book.

9.5.3: Review

This content is not available in the sample book.

10: Test Automation Characteristics

This content is not available in the sample book.

10.1: Specific

This content is not available in the sample book.

10.2: Clean

This content is not available in the sample book.

10.3: Independent

This content is not available in the sample book.

10.4: Frequent

This content is not available in the sample book.

10.4.1: Frequency as a need

This content is not available in the sample book.

10.4.2: Frequency as an outcome

This content is not available in the sample book.

10.5: Focused

This content is not available in the sample book.

10.6: Programmable

This content is not available in the sample book.

10.7: Creative

This content is not available in the sample book.

10.8: Sustainable

This content is not available in the sample book.

10.9: Review

This content is not available in the sample book.

11: Functional Test Refactoring

This content is not available in the sample book.

11.1: Code refactoring

This content is not available in the sample book.

11.2: Functional test refactoring

This content is not available in the sample book.

11.3: Tool support

This content is not available in the sample book.

11.4: Case study

This content is not available in the sample book.

11.4.1: Clean up

This content is not available in the sample book.

11.4.1.1: Remove unnecessary `click()` steps

This content is not available in the sample book.

11.4.1.2: Remove unnecessary Tab key presses

This content is not available in the sample book.

11.4.1.3: Better locator

This content is not available in the sample book.

11.4.2: Reformat into PyTest format

This content is not available in the sample book.

11.4.2.1: Add an assertion

This content is not available in the sample book.

11.4.2.2: Test script

This content is not available in the sample book.

11.4.3: Extract “sign in” function

This content is not available in the sample book.

11.4.4: Extract “sign off” function

This content is not available in the sample book.

11.4.5: Extract FlightPage

This content is not available in the sample book.

11.4.6: Move function to test helper

This content is not available in the sample book.

11.4.7: Extract PassengerPage

This content is not available in the sample book.

11.4.8: Extract PaymentPage

This content is not available in the sample book.

11.4.9: Move to execution hooks

This content is not available in the sample book.

11.4.10: Full test scripts

This content is not available in the sample book.

11.5: Summary

This content is not available in the sample book.

11.6: Review

This content is not available in the sample book.

12: Set up Continuous Execution

This content is not available in the sample book.

12.1: Conventional Playwright test execution in CI/CD is suboptimal

This content is not available in the sample book.

12.1.1: Multiple workers

This content is not available in the sample book.

12.1.2: What about sharding?

This content is not available in the sample book.

12.2: BuildWise CT server setup

This content is not available in the sample book.

12.2.1: Pre-requisite

This content is not available in the sample book.

12.2.2: Install BuildWise CT Server

This content is not available in the sample book.

12.2.3: Start BuildWise CT Server

This content is not available in the sample book.

12.3: Sequential Execution

This content is not available in the sample book.

12.3.1: Create a build project

This content is not available in the sample book.

12.3.2: Execute the test suite

This content is not available in the sample book.

12.4: Parallel execution

This content is not available in the sample book.

12.4.1: Create a parallel build project

This content is not available in the sample book.

12.4.2: Trigger a parallel build

This content is not available in the sample book.

12.4.3: Install BuildWise Agent on build machines

This content is not available in the sample book.

12.4.4: BuildWise Agent machine setup

This content is not available in the sample book.

12.4.5: Configure the BuildWise Agent

This content is not available in the sample book.

12.4.5.1: 1. General Settings

This content is not available in the sample book.

12.4.5.2: 2. Enable for Applications.

This content is not available in the sample book.

12.4.5.3: 3. Check Python is Available

This content is not available in the sample book.

12.4.6: Parallel execution

This content is not available in the sample book.

12.4.7: Benefits of Parallel Execution in BuildWise

This content is not available in the sample book.

12.5: Review

This content is not available in the sample book.

13: Preparing for Real-World Test Automation

This content is not available in the sample book.

13.1: Syntax errors

This content is not available in the sample book.

13.1.1: How to minimize syntax errors?

This content is not available in the sample book.

13.2: Beyond syntax errors

This content is not available in the sample book.

13.3: Set up source control

This content is not available in the sample book.

13.3.1: Git Installation

This content is not available in the sample book.

13.3.2: Initialize a Local Repository

This content is not available in the sample book.

13.3.3: Set Up a Shared Git Repository

This content is not available in the sample book.

13.3.4: Cloning the Repository

This content is not available in the sample book.

13.3.5: Frequently used Git commands after set up

This content is not available in the sample book.

13.4: Test Data

This content is not available in the sample book.

13.5: Choose Robust Locators

This content is not available in the sample book.

13.5.1: Don't use dynamic IDs

This content is not available in the sample book.

13.5.2: Avoid copying CSS Selectors from Browser Inspector

This content is not available in the sample book.

13.5.3: Better using the XPath

This content is not available in the sample book.

13.6: Stabilize Tests

This content is not available in the sample book.

13.6.1: Effective solution to address unstable test executions

This content is not available in the sample book.

13.7: Test script maintenance is the primary effort

This content is not available in the sample book.

13.8: Custom libraries

This content is not available in the sample book.

13.9: Debugging

This content is not available in the sample book.

13.9.1: Playwright `page.pause()`

This content is not available in the sample book.

13.9.2: Inspect the pages visually and from the test script

This content is not available in the sample book.

13.10: What is the best learning method?

This content is not available in the sample book.

13.11: Review

This content is not available in the sample book.

14: Team Collaboration

This content is not available in the sample book.

14.1: Pre-requisite

This content is not available in the sample book.

14.1.1: Version Control Server

This content is not available in the sample book.

14.1.2: A Shared Testing Tool

This content is not available in the sample book.

14.2: Developer-Tester Collaboration Scenario 1: “It worked on my machine”

This content is not available in the sample book.

14.2.1: Benefits

This content is not available in the sample book.

14.3: Developer-Tester Collaboration Scenario 2: Programmers help QA engineers with technical challenges

This content is not available in the sample book.

14.3.1: Benefits

This content is not available in the sample book.

14.4: Developer-Tester Collaboration Scenario 3: Acceptance Test-Driven Development

This content is not available in the sample book.

14.4.1: QA Engineer writes the test before the feature exists

This content is not available in the sample book.

14.4.2: Development guided by the automated test

This content is not available in the sample book.

14.4.3: Benefits

This content is not available in the sample book.

14.5: Business Analysts find E2E test automation useful too

This content is not available in the sample book.

14.6: Review

This content is not available in the sample book.

15: Clarifications

This content is not available in the sample book.

15.1: Real E2E Test Automation is far more challenging than coding

This content is not available in the sample book.

15.2: Neither possible or necessary to achieve 100% coverage

This content is not available in the sample book.

15.3: Gherkin BDD Framework

This content is not available in the sample book.

15.4: Headless testing

This content is not available in the sample book.

15.4.1: Embarrassing headless testing history For Many Software Engineers

This content is not available in the sample book.

15.4.2: The so-called benefit is far outweighed its flaws.

This content is not available in the sample book.

15.5: Codeless/Scriptless Test Automation

This content is not available in the sample book.

15.6: AI/Agentic Test Automation

This content is not available in the sample book.

15.7: Cross-browser functional testing

This content is not available in the sample book.

15.8: Data-Driven Test

This content is not available in the sample book.

15.9: Review

This content is not available in the sample book.

16: Adopting End-to-End Test Automation

This content is not available in the sample book.

16.1: Seek executive sponsorship

This content is not available in the sample book.

16.1.1: Benefit realization

This content is not available in the sample book.

16.1.2: Software maintenance

This content is not available in the sample book.

16.2: Choose a test framework

This content is not available in the sample book.

16.3: Select a test tool

This content is not available in the sample book.

16.4: Run as part of the CI/CD pipeline

This content is not available in the sample book.

16.5: Find a mentor

This content is not available in the sample book.

16.6: Manage expectations

This content is not available in the sample book.

16.7: Solo test automation

This content is not available in the sample book.

16.8: Common mistakes

This content is not available in the sample book.

16.8.1: Aiming too long

This content is not available in the sample book.

16.8.2: Juggling with test frameworks

This content is not available in the sample book.

16.8.3: Overestimate test automation effort

This content is not available in the sample book.

16.8.4: Underestimate test automation effort

This content is not available in the sample book.

16.8.5: Managers/Developers think “maintaining automated tests slows down the development”

This content is not available in the sample book.

16.8.6: No Test Automation Mentors

This content is not available in the sample book.

16.8.7: Lack of a real CI/CD Process

This content is not available in the sample book.

16.9: Review

This content is not available in the sample book.

17: AI Won't Replace Real Human E2E Test Automation

This content is not available in the sample book.

Afterword

This content is not available in the sample book.

Practice with clear goals

This content is not available in the sample book.

Learn from common mistakes

This content is not available in the sample book.

Apply your newly learned test automation at work

This content is not available in the sample book.

Use automation beyond testing

This content is not available in the sample book.

Continue learning and growing

This content is not available in the sample book.

Appendix 1 Functional Test Refactoring Catalog

This content is not available in the sample book.

Extract Function

This content is not available in the sample book.

Move Test Scripts

This content is not available in the sample book.

Move Function to Helper

This content is not available in the sample book.

Extract to Page Class

This content is not available in the sample book.

Introduce Page Object

This content is not available in the sample book.

Rename

This content is not available in the sample book.

Appendix 2 Case Study: Test Automation in WhenWise App Development

This content is not available in the sample book.

Build Stats

This content is not available in the sample book.

Over 456,000 test executions over 8 years

This content is not available in the sample book.

622 comprehensive automated UI test cases

This content is not available in the sample book.

Build time

This content is not available in the sample book.

Test Automation enables real Agile

This content is not available in the sample book.

Questions and Answers

This content is not available in the sample book.

Resources

This content is not available in the sample book.

Books

This content is not available in the sample book.

Tools

This content is not available in the sample book.

Blog

This content is not available in the sample book.

References

[Crispin & Gregory 09] Lisa Crispin and Janet Gregory. 2009. *Agile Testing*. Addison-Wesley Professional.

[Hunt & Thomas 00] Andrew Hunt and David Thomas. 2000. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.

[Fowler et al. 99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional.

[Fowler 00] Martin Fowler, "[Continuous Integration \(original version\)](#)"¹ (Sep 10, 2010)

[NEWS 10a] "[Pay system not properly tested: report](#)"², Austrian Financial Review, Jun 29, 2010.

[NEWS 11c] "[What would Larry Page do? Leadership lessons from Google's doyen](#)"³. April 18, 2011.

[Feathers 10] Michael Feathers. "[UI Test Automation Tools are Snake Oil](#)"⁴, Object Mentor Blog (Jan 4 2010).

[AAFTT07] [Agile Alliance Functional Testing Tools Visioning Workshop](#)⁵. 2007.

[Myers, Glenford 04]. *The Art of Software Testing*. Wiley. ISBN 978-0471469124.

[FSF] [The Free Software Definition](#)⁶

¹<http://martinfowler.com/articles/originalContinuousIntegration.html>

²<https://www.afr.com/companies/healthcare-and-fitness/pay-system-not-properly-tested-20100629-iv9nd>

³<https://fortune.com/2011/04/18/what-would-larry-page-do-leadership-lessons-from-google-doyen/>

⁴<http://blog.objectmentor.com/articles/2010/01/04/ui-test-automation-tools-are-snake-oil>

⁵<http://www.infoq.com/news/2007/10/next-gen-functional-testing>

⁶<http://www.gnu.org/philosophy/free-sw.html>

About the Cover Photo

The cover features the Shinkansen bullet train, photographed by the author at Shinagawa Station, Tokyo.

Book Cover Designed by Courtney Zhan.