Preface

This book is not a full and comprehensive treatise that outlines every possible permutation of Selenium (the open-source software test automation tool for web applications). There are other books that already do this. My goal, instead, is to teach you the necessary pieces to use Selenium successfully for your circumstance.

What you have before you is a distilled and actionable guide culled from my consulting practice and full time positions held doing Quality Assurance over the past ten years.

My goal in writing this is to provide you with the materials I wish existed when I was starting out with automated acceptance testing. I hope it serves you well.

What This Book Will Cover

This book focuses on the latest stable version of Selenium 4 (a.k.a. Selenium WebDriver) and its use to test desktop browsers.

Record and Playback tools like Selenium IDE are a great option nowadays (no, really). But they will not be covered in this book. Instead, an approach of writing well factored tests, in code, is the focus of this book.

Who This Book Is For

This book is for anyone who wants to take automated acceptance testing seriously and isn't afraid to get their hands a little dirty.

That is to say, this book is for anyone who wants to use computers for what they're good at, and free you up (and potentially the people on your team) to do what they are inherently good at (which does not include repetitive, mundane testing tasks). And don't worry if you're new to programming. I'll cover the essentials so you'll have a good place to start from.

About The Examples In This Book

The examples in this book are written in JavaScript, but the strategies and patterns used are applicable regardless of your technical stack.

The tests in this book are written to exercise functionality from an open-source project I created and maintain called the-internet -- available here on GitHub and viewable here on Heroku.

The test examples are written to run against pytest with pip3 managing the third-party dependencies.

All of the code examples from the book are available in an accompanying zip file. It contains folders for each chapter where code was written or altered. Chapters with multiple parts will have multiple sub-folders (e.g., code examples referenced in Part 2 of Chapter 9 can be found in `09/02/` in the zip file).

How To Read This Book

Chapters 1 through 5 focus on the things you need to consider when it comes to test strategy, programming language selection, and good test design. Chapter 6 is where we first start to code. From there, the examples build upon each other through chapter 16.

Chapter 17 paints a picture of the Selenium landscape so you're better able to find information on your own.

Feedback

If you find an error in the book (e.g., grammar issue, code issue, etc.) or have questions/feedback -- please feel free to e-mail me at dhaeffner@gmail.com.

If you submit something and I end up using it in a future version of the book I'll give you a shout-out in the Acknowledgements.

Acknowledgements

A huge thanks to Unmesh Gundecha and Peter Bittner!

Unmesh's work was a big help and an inspiration when it finally came time for me to write a Python edition of my book. His book titled "Learning Selenium Testing Tools with Python" is a great resource and I highly recommend it if you are looking to use `unittest` in your Selenium practice.

Peter provided a tremendous amount of feedback to me about the code examples in the first edition of this book. He recommended ways I could strive to make many of the examples more idiomatic, readable, and resilient. Thank you Peter! The book is still better off because of your input.

Cheers,
Dave H

Table of Contents

Chapter 1
Selenium In A Nutshell

Selenium is a software robot sent from the future to help us test web applications. But keep in mind that it's not one of those fancy shape-shifting robots than can run really fast. It's more like one of those really strong robots that's not very fast and is best suited for accomplishing a certain objective.

That is to say -- Selenium is really good at a specific set of things. If you know what those are and stick to them then you can easily write reliable, scalable, and maintainable tests that you and your team can trust.

What Selenium Is and Is Not Good At

Selenium is built to automate browsers, specifically human interaction with them. Things like navigating to pages, clicking on elements, typing text into input fields, etc.

It's less ideal for checking lower-level functionality like HTTP status codes or HTTP headers. While you can use Selenium this way, it requires additional setup of a third-party tool (e.g., a proxy server like BrowserMob Proxy), and it is a slippery slope since there are numerous edge cases to consider at this level.

Selenium Highlights

Selenium works on every major browser, in every major programming language, and on every major operating system. Each language binding and browser are actively being developed to stay current. Yes, even Internet Explorer (thanks to Jim Evans!). And WebDriver (the thing which drivers Selenium) has become a W3C specification.

Selenium can be run on your local computer, on a remote server, on a set of servers (with Selenium Grid), or on a third-party cloud provider (like Sauce Labs). As your test suite grows, your test runs will take longer to complete. To speed them up you will want to run them in parallel, which is where the benefit of having your own servers or using a cloud provider comes in -- that, and the ability to have numerous browser and operating system combinations to run your tests on.

One of the guiding philosophies of Selenium is to be able to write your tests once and run them across multiple browsers. While this is a rosy proposition, it's not entirely accurate. There are some gotchas to watch out for when you get into it. But don't worry, We'll step through these in detail throughout the book.

Chapter 2
Defining A Test Strategy

A great way to increase your chances of success with automated web testing is to first map out a testing strategy. The best way to do it is to answer these four questions:

1. How does your business make money?
2. What features in your application are being used?
3. What browsers are your users using?
4. What things have broken in the application before?

NOTE: For organizations that don't deal directly in dollars and cents (e.g., non-profits, federal agencies, etc.) you should focus on how the application generates value for the end user and the organization.

After answering these questions you will have an understanding of the functionality and browsers that matter for the application you're testing. This will help you focus your efforts on the things that matter most.

This strategy works best for applications with existing functionality and does not speak directly to testing new functionality that's being actively developed. That's not to say that the two couldn't co-exist. It largely depends on your available resources and pace of development. But in order to reach high quality at speed, you first have to go slow.

What To Do With The Answers

After answering these questions you should end up with a prioritized punch list (a.k.a., backlog) of test automation tasks that you can work off of and track progress against.

Question 1: Money/Value

Every company's application makes money (or generates value) through core functionality that is defined by a series of increasingly-valuable user interactions -- a.k.a. a "funnel". Your answers to this question will help you determine what your funnel is.

These items will be your highest priority for automation. Start an automation backlog to keep track of them.

Question 2: Usage Data

Odds are your application offers a robust set of functionality well beyond your funnel. Your answers to this question will help highlight what it is. And if you're basing these answers on usage data (e.g., Google Analytics, etc.), then it will be broken down from highly used to lightly used.

Tack these items onto your automation backlog (below the items from question #1) based on their frequency of use.

Question 3: Browsers

Now that you know what functionality is business critical and widely adopted by your users, you need to determine what browsers to focus your automated web testing efforts on. Your usage data will tell you this as well. It will help you determine which browsers you can reasonably avoid testing in (e.g., based on non-existent or low usage numbers).

Note the top 2 (or 3 depending on your numbers), but focus on the top 1 for now. This is the browser you will start using for automation.

Question 4: Risky Bits

To round out the strategy it is also best to think about what things have broken in the application before. To answer this question it's best to check your defect/bug tracker (if you have one) and to ask your team. What you end up with may read like a laundry list of browser specific issues or functionality that has been flaky or forgotten about in the past. But it's all useful information.

Be sure to check this list against your automation backlog. If something's not there, add it to the bottom of the backlog. If it is there, make a note in the backlog item that it has been an issue in the past.

If the issue has happened numerous times and has the potential to occur again, move the item up in the backlog. And if issues keep cropping up that are related to a specific browser, compare this browser to your short list of browsers from question #3. If it's a browser that's not in your list but it's still important (e.g., a small pocket of influential users), track it on the backlog, but put it at the bottom.

Now You Are Ready

Having answered these questions, you should now have a prioritized backlog of critical business functionality, a short list of browsers to focus on, and an understanding of the risky parts of your application to watch out for. With it, you're on the right track -- focusing on things that matter for your business and its users.

Chapter 3
Picking A Language

In order to work well with Selenium you need to choose a programming language to write your automated acceptance tests in. Conventional wisdom will tell you to choose the same language that the application is written in. That way if you get stuck you can ask the developers on your team for help. But if you're not proficient in this language (or new to programming), then your progress will be slow and you'll likely end up asking for more of a developer's help than they have time for -- hindering your automation efforts and setting you up for failure.

A great way to determine which language to go with is to answer one simple question: Who will own the automated tests?

The answer to this, and the discussion that unfolds from it, will help you more effectively choose a programming language.

What To Do With The Answer

If you're a tester (or a team of testers) and you will be building and maintaining the test automation, then it's best to consider what languages you (and your team) already have experience with, or are interested in working with. Since your Selenium tests don't need to be written in the same language as the application you are testing, have a discussion with your team to see what interests them and go from there.

If you're a developer who is working on the application and just looking to add automated acceptance testing to the mix, then it makes sense to continue with the same language.

Some Additional Food For Thought

As your suite of tests grows you will find the need to add functionality to make things easier to use, run faster, offer better reporting, etc. When this happens you will effectively be doing two jobs; no longer just writing automated tests but also building and maintaining a framework (a.k.a. a test harness).

As you're considering which language to go with consider what open source frameworks already exist for the languages you are considering. Going with one can save you a lot of time and give you a host of functionality out of the box that you would otherwise have to create yourself -- and they're FREE.

You can find a list of open source Selenium WebDriver frameworks and their respective languages here.

Outro

Choosing a programming language for automated testing is not a decision that should be taken lightly. If you're just starting out then considering and discussing these things will help position you for long term success.

With all that being said, the rest of this book will show you how to build your own test harness (in addition to writing well factored tests) in Python 3 with pytest.

Chapter 4
A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to Selenium) so you have some working knowledge and a vocabulary. This will help you more effectively comprehend what you will see throughout this book (and in your work afterwards).

Don't get too hung up on the details though. If something doesn't make sense it should once we dig into the full examples in the following chapters. Consider this more of a glossary than a code school.

Installation

There's a great guide on "Properly Installing Python" which you can find here. It covers Mac OSX, Windows, and Ubuntu.

NOTE: For doing proper software development in Python you'd want to consider something like Virtual Environments to effectively manage third-party dependencies. But for the needs of the examples in this book, it's not necessary.

Installing Third-Party Libraries

There are over 184,000 third-party libraries (a.k.a. "packages") available for Python through PyPI (the Python Package Index). To install packages from it you use a program called `pip3`.

To install them you use `pip3 install package-name` from the command-line.

Here is a list of the packages that will be used throughout the book.

- `pytest`
- `pytest-randomly`
- `pytest-xdist`
- `selenium`

Interactive Prompt

One of the immediate advantages to using a scripting language like Python is that you get access to an interactive prompt. Just type `python` from the command-line. It will load a prompt that looks like this:

```
> python3
Python 3.7.2 (default, Jan 13 2019, 12:50:01)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In this prompt you can type out Python code. It will evaluate it and return the result. As you step through this chapter it will serve as a great place to practice the commands and concepts you will see.

When you're done, type `quit()` and hit enter. Or press `CTRL + d`.

Choosing A Text Editor

In order to write Python code, you will need to use a text editor. Some popular ones are Vim, Emacs, Sublime Text.

There's also the option of going for an IDE (Integrated Development Environment) like PyCharm or Visual Studio Code.

It's important to pick an editor that works for you and makes you productive. So if you're new to programming and text editors then it's probably best to go with something more intuitive like Sublime Text or PyCharm.

Programming Concepts In A Nutshell

Programming can be a deep and intimidating rabbit hole if you're new to it. But don't worry. When it comes to automated browser testing there is only a small subset of programming that we really need to know in order to get started. Granted, the more you know, the better off you'll be. But you don't need to know a whole lot in order to be effective right now.

Of all the programming concepts out there, here are the ones you'll need to pay attention right now:

- Object Structures (Variables, Methods, and Classes)
- Scope
- Types of Objects (Strings, Integers, Data Structures, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Decorators
- Inheritance

Let's step through each and how they pertain to testing with Selenium.

Object Structures

Variables

Variables are places to store and retrieve values of various types (e.g., Strings, Integers, etc. -- more on these later). Variables are created and then referenced by their name.

A variable name:

- can be one or more words in length
- use an underbar (`_`) to separate the words (e.g., `example_variable`)
- start with a lowercase letter
- are often entirely lowercase
- multiple word variables can be combined into a single word for better readability

You can store things in them by using an equals sign (`=`) after their name. In Python, a variable takes on the type of the value you store in it (more on object types later).

```
>>> example_variable = "42"
>>> print(type(example_variable))
# outputs: <class 'str'>
# 'str' is short for "string"

>>> example_variable = 42
>>> print(type(example_variable))
# outputs: <class 'int'>
# 'int' is short for "integer"
```

In the above example `print` is used to output a message. This is a common command that is useful for generating output to the terminal.

In Selenium, a common example of a variable is when we need to store an element (or the value from an element) in the beginning of a test to be referenced later. For example, getting a page's title.

```
page_title = driver.title
```

`driver` is the variable we will use to interact with Selenium.

Methods

One way to group common actions (a.k.a. behavior) for easy reuse is to place them into methods. We define a method with the opening keyword `def` , a name (in the same fashion as a variable), and end the method with white-space. Referencing a method is done the same way as a variable -- by its name. And the lines of code within the method need to be indented with spaces.

```
def example_method():
  # your code
  # goes here

example_method()
```

Additionally, we can specify arguments we want to pass into the method when calling it.

```
>>> def say(message):
...     print(message)
...
>>> say("Hello World!")
# outputs: Hello World!
```

When setting an argument, we can also set a default value to use if no argument is provided.

```
>>> def say(message="Hello World!")
...     print(message)
...
>>> say()
# outputs: # Hello World!
>>> say("something else")
# outputs: something else
```

We'll see something like this used in Selenium when we are telling Selenium how to wait with explicit waits (more on that in Chapter 10).

Classes

Classes are a useful way to represent concepts that will be reused numerous times in multiple places. They can contain variables and methods and are defined with the word `class` followed by the name you wish to give it.

Class names:

- start with a capital letter
- should be PascalCase for multiple words (e.g., `class ExampleClass`)
- should be descriptive (e.g., a noun, whereas methods should be a verb)
- end with whitespace (just like methods)

You first have to define a class, and then create an instance of it (a.k.a. instantiation) in order to use it. Once you have an instance you can access the methods within it to trigger an action. Methods in classes need to use the keyword `self` as an argument in order to be properly referenced.

```
>>> class Message():
...     def say(self, message = "Hello World!"):
...             print(message)
...
>>> message_instance = Message()
>>> message_instance.say("This is an instance of a class")
# outputs: This is an instance of a class
```

An example of this in Selenium is the representation of a web page -- also known as a 'Page Object'. In it you will store the page's elements and behavior we want to interact with.

```
class LoginPage():
    _login_form = {"by": By.ID, "value": "login"}
    _username_input = {"by": By.ID, "value": "username"}
    _password_input = {"by": By.ID, "value": "password"}

    def with_(self, username, password):
# ...
```

The variables that start with underscores (e.g., `_`) are considered internal (or "local") variables, the values in curly brackets (`{}`) are called dictionaries. More on all of that soon.

Scope

Now that we have our different object structures it's worth briefly covering how they can and cannot access one another. The easiest way to do this is by discussing the different types of variables you are likely to run into.

Local Variables

Local variables enable you to store and retrieve values from within a confined area (this is no different than what was described in the variables section earlier). A variable that was declared within a method is a great example of this. It is useful within the method it was declared, but inaccessible outside of it.

Instance Variables

Instance variables enable you to store and retrieve values more broadly (e.g., both inside and outside of methods). They are named the same way as regular variables, except that they start with `self.`. This is only applicable for variables within a class.

A common example you will see throughout this book is the usage of `self.driver`. This is an instance of Selenium stored in an instance variable. This object is what enables us to control the browser and by storing it as an instance variable we'll be able to use it where necessary.

NOTE: There are also class variables, which are similar to instance variables in terms of their scope. They do not require `.self` as part of their declaration. We'll see these when we get into Page Objects in Chapter 8.

Private Objects

Python is a dynamic language with few constraints (which is very much the opposite of compiled languages like Java). The idea of limited object access (either implied by an object's scope or explicitly by a `private` keyword like in other programming languages) isn't an enforcible concept in Python. But there is a saying in Python -- "We are all responsible users".

It means that rather than relying on hard constraints we should rely on a set of conventions. These conventions are meant to indicate which elements should and should not be accessed directly.

The convention for this is to start variable and method names which are meant to be private (e.g., only meant to be used internally where the object was declared) with an underscore. (e.g., `_example_variable` or `_example_method():`).

Environment Variables

Environment variables are a way to pass information into our program from outside of it. They are also a way to make a value globally accessible (e.g., across an entire program, or set of programs). They can be set and retrieved from within your code by:

- importing the built-in `os` Python library (e.g., `import os`)
- using the `os.environ` lookup syntax
- specify the environment variable name with it

Environment variables are often used to store configuration values that could change. A great example of this is the access key for a third-party service provider that we'll use in our tests.

```
import os
_credentials = '%s:%s' % (os.environ["SAUCE_USERNAME"],
                          os.environ["SAUCE_ACCESS_KEY"])
```

Types of Objects

Strings

Strings are alpha-numeric characters packed together (e.g., letters, numbers, and most special characters) surrounded by either single (`'`) or double (`"`) quotes.

You'll run into Strings when working with copy on a page (e.g., pulling a page's URL, title, or h1 tag to determine if your test is in the right place before proceeding).

Numbers

The two common types of numbers you will run into with testing are Integers (whole numbers) and Float (decimals). Or `'int'` and `'float'` with regards to how the Python language refers to them.

If you have a test that needs to pull some values from a page and add/subtract/multiply/divide them, then this will come in handy. Although you may need to convert the values from a String to an Integer first. But don't sweat it, this is a trivial thing to do in Python.

```
int("42")
```

The conversion from a `'str'` to an `'int'` is done with `int()` method. If you're working with decimals, you can use the `decimal` library built into Python to convert it.

```
>>> import decimal
>>> decimal.Decimal("42.00")
```

Data Structures

Data Structures enable you to gather a set of data for later use. In Python there are numerous data structures. The one we'll want to pay attention to is Dictionaries.

Dictionaries are an unordered set of data stored in key/value pairs. The keys are unique and are used to look up the data in the dictionary.

```
>>> a_dictionary = {"this": "that", "the": "other"}
>>> print(a_dictionary["this"])
# outputs: that
>>> print(a_dictionary["the"])
# outputs: other
```

You'll end up working with Dictionaries in your Page Objects to store and retrieve your page's locators.

```
class LoginPage():
    _login_form = {"by": By.ID, "value": "login"}
    _username_input = {"by": By.ID, "value": "username"}
    _password_input = {"by": By.ID, "value": "password"}
    # ...
```

Booleans

Booleans are binary values that are returned when asking a question of your code. They are what enable us to complete assertions.

There are numerous ways to ask questions. Some involve various comparison operators (e.g., `==`, `!=`, `<`, `>`, `<=`, `>=`). The response is either `True` or `False`.

```
>>> 2 + 2 == 4
True
```

Selenium also has commands that return a boolean result when we ask questions of the page we're testing.

```
element.is_displayed()
# outputs: returns `True` if the element is on the page and visible
```

Actions

A benefit of booleans is that we can use them to perform an assertion.

Assertions

Assertions are made against booleans and result in either a passing or failing test. In order to leverage assertions we will need to use a testing framework (e.g., pytest, unittest, or nose). For the examples in this book we will be using pytest (version 2.9.2).

pytest has a built-in assertion method which accepts an argument for something that returns a boolean.

```
driver.get("http://the-internet.herokuapp.com")
assert(driver.title == "The Internet")

# or

driver.get("http://the-internet.herokuapp.com")
title_present = driver.title == "The Internet"
assert(title_present)
```

Both approaches will work, resulting in a passing assertion. If this is the only assertion in your test then this will result in a passing test. More on this and other good test writing practices in Chapter 5.

Conditionals

Conditionals work with booleans as well. They enable you execute different code paths based on their values.

The most common conditionals in Python are `if`, `elif` (short for else/if), and `else` statements.

```
number = 10
if number > 10:
  print("The number is greater than 10")
elif number < 10:
  print("The number is less than 10")
elif number == 10:
  print("The number is 10")
else:
  print("I don't know what the number is.")
end

# outputs: The number is 10
```

You'll end up using conditionals in your test setup code to determine which browser to load based on a configuration value. Or whether or not to run your tests locally or somewhere else.

```
if config.host == "localhost":
        if config.browser == "firefox":
            driver_ = webdriver.Firefox()
        elif config.browser == "chrome":
            driver_ = webdriver.Chrome()
```

More on that in chapters 12 and 13.

Decorators

Decorators are a form of metadata. They are used by various libraries to enable additional functionality in your tests.

The most common way we're going to use this is for our test setup and teardown. In pytest we do this with a fixture (which is a function that gets called before our test with the option to execute it after the test as well).

```
@pytest.fixture
    def driver(self, request):
        driver_ = webdriver.Firefox()

        def quit():
            driver_.quit()

        request.addfinalizer(quit)
        return driver_
```

The decorator with the `@` symbol, has a name to go along with it (e.g., `@pytest.fixture`), and is placed above the object that it effects (e.g., the `driver` method). We'll see decorators used for a few different things throughout the book.

Inheritance

Classes have the ability to connect to one-another through parent/child inheritance. By having a single parent class we can store common actions in methods that can be readily available to all child classes.

Inheritance is done when declaring a child class by:

- providing the class name
- importing the parent class (when in another file)
- specifying the parent class name as a declaration argument

```
>>> class Parent():
...     hair_color = "brown"
...
>>> class Child(Parent):
...     pass
...
>>> c = Child()
>>> c.hair_color

# outputs: brown
```

You'll see this in your tests when writing all of the common Selenium actions you intend to use into methods in a parent class. Inheriting this class will allow you to call these methods in your child classes (more on this in Chapter 9).

Additional Resources

Here are some additional resources that can help you continue your Python learning journey.

- A list of getting started resources from the Python project
- Interactive online resources from learnpython.org, codecademy, and Code School
- Learn Python the Hard Way

Chapter 5

Anatomy Of A Good Acceptance Test

In order to write automated web tests that are easy to maintain, perform well, and are ultimately resilient there are some simple guidelines to follow:

- Write atomic and autonomous tests
- Group like tests together in small batches
- Be descriptive
- Use a Test Runner
- Store tests in a Version Control System

Atomic & Autonomous Tests

Each test needs to be concise (e.g., testing a single feature rather than multiple features) and be capable of being run independently (e.g., sets up its own data rather than relying on a previous test to do it). Doing this may require a mental shift, discipline, and more up front effort. But it will make a dramatic impact on the quality, effectiveness, and maintainability of your tests. Especially when you get into parallel test execution.

Grouping Tests

As your test suite grows you will have numerous test files. Each one containing a grouping of tests that exercise similar functionality. These test files should be grouped together in a simple directory structure as the groupings become obvious. If you want to create a test run of disparate tests, this is something that is easy to handle when using a Test Runner (covered briefly below, and in-depth in Chapter 16).

Being Descriptive

A test file should have a high level name that denotes what the tests within it are exercising. Each test should have an informative name (even if it is a bit verbose). Also, each test (or grouping of tests) should include some helpful metadata (e.g., tags, or categories) which can provide additional information about the test as well as enable flexible test execution (more on that in Chapter 16). This way all or part of your test suite can be run, and the results will be informative thanks to helpful naming.

This enables developers to run a subset of tests to exercise functionality they just modified while also enabling you to intelligently wire your test suite up to a Continuous Integration (CI) server for fast and dynamic feedback (more on CI servers in Chapter 17).

Test Runners

At the heart of every test harness is some kind of a test runner that does a lot of the heavy lifting (e.g., test execution, centralized configuration, test output, etc.). Rather than reinvent the wheel you can use one of the many test runners that exist today. With it you can bolt on third party libraries to extend its functionality if there's something missing.

Version Control

In order to effectively collaborate with other testers and developers on your team, your test code must live in a version control system. Look to see what your development team uses and add your code to it. Otherwise, set up one of the following:

- Git
- Mercurial
- Subversion

Keep in mind that your test code can live in a separate repository from the code of the application you're testing. Combining them may be advantageous but if all you're doing is writing and running tests against web endpoints (which is a majority of what your Selenium tests will be doing) then leaving your test code in a separate repository is a fine way to go.

Chapter 6
Writing Your First Test

Fundamentally, Selenium works with two pieces of information -- the element on a page you want to use and what you want to do with it. This one-two punch will be repeated over and over until you achieve the outcome you want in your application. At which point you will perform an assertion to confirm that the result is what you intended.

Let's take logging in to a website as an example. With Selenium you would:

1. Visit the login page of a site
2. Find the login form's username field and input the username
3. Find the login form's password field and input the password
4. Find the submit button and click it

Selenium is able to find and interact with elements on a page by way of various locator strategies. The list includes (sorted alphabetically):

- Class
- CSS Selector
- ID
- Link Text
- Name
- Partial Link Text
- Tag Name
- XPath

While each serves a purpose, you only need to know a few to start writing effective tests.

How To Find Locators

The simplest way to find locators is to inspect the elements on a page. The best way to do this is from within your web browser. Fortunately, popular browsers come pre-loaded with development tools that make this simple to accomplish.

When viewing the page, right-click on the element you want to interact with and click Inspect Element. This will bring up a small window with all of the markup for the page but zoomed into your highlighted selection. From here you can see if there are unique or descriptive attributes you can work with.

How To Find Quality Elements

You want to find an element that is unique, descriptive, and unlikely to change.

Ripe candidates for this are `id` and `class` attributes. Whereas text (e.g., the text of a link) is less ideal since it is more apt to change. If the elements you are attempting to work with don't have unique `id` or `class` attributes directly on them, look at the element that houses them (a.k.a. the parent element). Oftentimes the parent element has a unique element that you can use to start with and walk down to the child element you want to use.

When you can't find any unique elements have a conversation with your development team letting them know what you are trying to accomplish. It's typically a trivial thing for them to add helpful semantic markup to a page to make it more testable. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy and painful process which might yield working test code but it will be brittle and hard to maintain.

Once you've identified the target elements and attributes you'd like to use for your test, you need to craft locators using one Selenium's strategies.

An Example

Part 1: Find The Elements And Write The Test

Here's the markup for a standard login form (pulled from [the login example on the-internet]).

```
<form name="login" id="login" action="/authenticate" method="post">
   <div class="row">
    <div class="large-6 small-12 columns">
      <label for="username">Username</label>
      <input type="text" name="username" id="username">
    </div>
  </div>
  <div class="row">
    <div class="large-6 small-12 columns">
      <label for="password">Password</label>
      <input type="password" name="password" id="password">
    </div>
  </div>
    <button class="radius" type="submit"><i class="icon-2x icon-signin"> Login
</i></button>
</form>
```

Notice the element attributes on the form. The username input field has a unique `id`, as does the password input field. The submit button doesn't, but it's the only button on the page so we can

easily find it and click it.

Let's put these elements to use in our first test. First we'll need to create a new folder called `tests` in the root of our project directory. In it we'll create a new test file called `login_test.py` and a requisite Python file for the directory called `__init__.py`.

We'll also need to create a `vendor` directory for third-party files and download `geckodriver` into it. Grab the latest release for your operating system from here and unpack its contents into the `vendor` directory. This is a required file (known as a browser driver) in order to make Selenium work with Firefox. We'll cover browser drivers more in-depth in Chapter 12.

When we're done our directory structure should look like this.

```
tests
    __init__.py
    login_test.py
vendor
    geckodriver
```

Here is the code we will add to the test file for our Selenium commands, locators, etc.

```
# filename: tests/login_test.py
import pytest
import os
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.firefox.service import Service as FirefoxService


class TestLogin():

    @pytest.fixture
    def driver(self, request):
        _geckodriver = os.path.join(os.getcwd(), 'vendor', 'geckodriver')
        if os.path.isfile(_geckodriver):
            _service = FirefoxService(executable_path=_geckodriver)
            driver_ = webdriver.Firefox(service=_service)
        else:
            driver_ = webdriver.Firefox()

        def quit():
            driver_.quit()

        request.addfinalizer(quit)
        return driver_


    def test_valid_credentials(self, driver):
        driver.get("http://the-internet.herokuapp.com/login")
        driver.find_element(By.ID, "username").send_keys("tomsmith")
        driver.find_element(By.ID, "password").send_keys("SuperSecretPassword!")
        driver.find_element(By.CSS_SELECTOR, "button").click()
```

After importing the requisite classes for pytest and Selenium we declare a test class (e.g., `class TestLogin()`). We then declare a method within the class called `driver`. At the top of the method we add a decorator to denote that this is a fixture (e.g., `pytest.fixture`). By default fixture methods in pytest are called around each test method. So we'll use this to both setup and teardown our instance of Selenium.

To create an instance of Selenium we call `webdriver.Firefox()` and store the respone in a variable. Since the name of the method is already `driver`, we refer to this variable as `driver_` (to avoid a naming conflict). This variable gets returned at the end of the method, which means it will get used in our test (more on that soon). In order for this to work we provide the path to the `geckodriver` file, which we do by finding it and storing it in a local variable called `_geckodriver`

at the top of the method and passing it in as an argument to Selenium when creating the instance, through the use of a Service object (e.g., `_service = FirefoxService(executable_path=_geckodriver)` and `webdriver.Firefox(service=_service)`).

If there's not a `geckodriver` file, then the instance is created without specifying it, which will cause Selenium to look for `geckodriver` on the system path.

NOTE: The filename **`geckodriver`** was used in the example above, which works on Linux and Mac. If you're using Windows you will need to change this value to **`geckodriver.exe`**.

The `driver` method has two parameters, `self` and `request`. `self` is a required parameter for class methods, `request` is a parameter made available to fixtures. It enables access to loads of things during a test run. For now, the relevant piece is the ability to call `request.addfinalizer`. Actions passed to `addfinalizer` get executed after a test method completes. So we're calling `driver_.quit()` and passing it into the `addfinalizer` method.

Our test method starts with the word `test_` (that's how pytest knows it's a test) and it has two parameters (`self` and `driver`). `driver` is our access to the fixture method created at the top of the class. Since it returns a browser instance we can reference this variable directory to use Selenium commands. In this test we're visiting the login page by its URL (with `driver.get()`), finding the input fields by their ID (with `driver.find_element(By.ID, "username")`), inputting text into them (with `.send_keys()`), and submitting the form by clicking the submit button (e.g., `By.CSS_SELECTOR("button")).click()`).

If we save this and run it (by running `pytest` from the command-line), it will run and pass. But there's one thing missing -- an assertion. In order to find an element to write an assertion against we need to see what the markup of the page is after submitting the login form.

Part 2: Figure Out What To Assert

Here is the markup that renders on the page after logging in.

```
<div class="row">
  <div id="flash-messages" class="large-12 columns">
    <div data-alert="" id="flash" class="flash success">
      You logged into a secure area!
      <a href="#" class="close">x</a>
    </div>
  </div>
</div>

<div id="content" class="large-12 columns">
  <div class="example">
    <h2><i class="icon-lock"></i> Secure Area</h2>
    <h4 class="subheader">Welcome to the Secure Area. When you are done click logout
below.</h4>
    <a class="button secondary radius" href="/logout"><i class="icon-2x icon-signout">
Logout</i></a>
  </div>
</div>
```

There are a couple of elements we can use for our assertion in this markup. There's the flash message class (most appealing), the logout button (appealing), or the copy from either the `h2` or the flash message (least appealing).

Since the flash message class name is descriptive, denotes a successful login, and is less likely to change than the copy, let's go with that.

```
class="flash success"
```

When we try to access an element like this (e.g., with a multi-worded class) we will need to use a CSS selector or an XPath.

NOTE: Both CSS selectors and XPath can work well, but the examples throughout this book will focus on how to use CSS selectors.

A Quick Primer on CSS Selectors

In web design CSS (Cascading Style Sheets) are used to apply styles to the markup (HTML) on a page. CSS is able to do this by declaring which bits of the markup it wants to alter through the use of selectors. Selenium operates in a similar manner but instead of changing the style of elements, it interacts with them by clicking, getting values, typing, sending keys, etc.

CSS selectors are a pretty straightforward and handy way to write locators, especially for hard to reach elements.

For right now, here's what you need to know. In CSS, class names start with a dot (`.`). For classes with multiple words, put a dot in front of each word, and remove the space between them (e.g., `.flash.success` for `class='flash success'`).

For a good resource on CSS Selectors I encourage you to check out Sauce Labs' write up on them.

Part 3: Write The Assertion And Verify It

Now that we have our locator, let's add an assertion that uses it.

```
# filename: tests/login_test.py
#  ...
    def test_valid_credentials(self, driver):
        driver.get("http://the-internet.herokuapp.com/login")
        driver.find_element(By.ID, "username").send_keys("tomsmith")
        driver.find_element(By.ID, "password").send_keys("SuperSecretPassword!")
        driver.find_element(By.CSS_SELECTOR, "button").click()
        assert driver.find_element(By.CSS_SELECTOR, ".flash.success").is_displayed()
```

With `assert` we are checking for a `True` Boolean response. If one is not received the test will fail. With Selenium we are seeing if the success message element is displayed on the page (with `.is_displayed`). This Selenium command returns a boolean. So if the element is rendered on the page and is visible (e.g., not hidden or covered up by an overlay), `True` will be returned, and our test will pass.

When we save this and run it (e.g., `pytest` from the command-line) it will run and pass just like before, but now there is an assertion which will catch a failure if something is amiss.

Just To Make Sure

Just to make certain that this test is doing what we think it should, let's change the locator in the assertion to attempt to force a failure and run it again. A simple fudging of the locator will suffice.

```
assert driver.find_element(By.CSS_SELECTOR, ".flash.successasdf").is_displayed()
```

If it fails then we can feel reasonably confident that the test is doing what we expect and we can change the assertion back to normal before committing our code.

This trick will save you more trouble that you know. Practice it often.

Want the rest of the book?

Buy your copy HERE