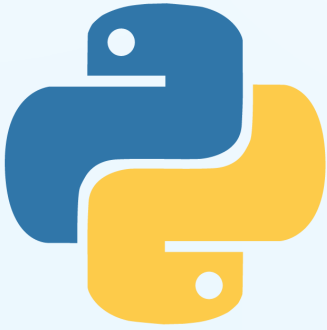
Selenium WebDriver Recipes in Python

The Problem Solving Guide to Selenium WebDriver



Zhimin Zhan

Selenium WebDriver Recipes in Python

The problem solving guide to Selenium WebDriver in Python

Zhimin Zhan

This book is for sale at http://leanpub.com/selenium-recipes-in-python

This version was published on 2016-08-27

This is a Leanpub book. Leanpub 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.

Also By Zhimin Zhan

Practical Web Test Automation

Watir Recipes

Selenium WebDriver Recipes in Ruby

Selenium WebDriver Recipes in Java

Learn Ruby Programming by Examples

Learn Swift Programming by Examples

API Testing Recipes in Ruby

Contents

Preface

After observing many failed test automation attempts by using expensive commercial test automation tools, I am delighted to see that the value of open-source testing frameworks has finally been recognized. I still remember the day (a rainy day at a Gold Coast hotel in 2011) when I found out that the Selenium WebDriver was the most wanted testing skill in terms of the number of job ads on the Australia's top job-seeking site.

Now Selenium WebDriver is big in the testing world. We all know software giants such as Facebook and LinkedIn use it, immensely-comprehensive automated UI testing enables them pushing out releases several times a day[1]. However, from my observation, many software projects, while using Selenium, are not getting much value from test automation, and certainly nowhere near its potential. A clear sign of this is that the regression testing is not conducted on a daily basis (if test automation is done well, it will happen naturally).

Among the factors contributing to test automation failures, a key one is that automation testers lack sufficient knowledge in the test framework. It is quite common to see some testers or developers get excited when they first create a few simple test cases and see them run in a browser. However, it doesn't take long for them to encounter some obstacles: such as being unable to automate certain operations. If one step cannot be automated, the whole test case does not work, which is the nature of test automation. Searching solutions online is not always successful, and posting questions on forums and waiting can be frustrating (usually, very few people seek professional help from test automation coaches). Not surprisingly, many projects eventually gave up test automation or just used it for testing a handful of scenarios.

The motivation of this book is to help motivated testers work better with Selenium. The book contains over 150 recipes for web application tests with Selenium. If you have read one of my other books: *Practical Web Test Automation*[2], you probably know my style: practical. I will let the test scripts do most of the talking. These recipe test scripts are 'live', as I have created the target test site and included offline test web pages. With both, you can:

1. **Identify** your issue
2. **Find** the recipe
3. **Run** the test case
4. **See** test execution in your browser

[1] http://www.wired.com/business/2013/04/linkedin-software-revolution/

[2] https://leanpub.com/practical-web-test-automation

Who should read this book

This book is for testers or programmers who are writing (or want to learn) automated tests with Selenium WebDriver. In order to get the most of this book, basic Ruby coding skill is required.

How to read this book

Usually, a 'recipe' book is a reference book. Readers can go directly to the part that interests them. For example, if you are testing a multiple select list and don't know how, you can look up in the Table of Contents, then go to the chapter. This book supports this style of reading. Since the recipes are arranged according to their levels of complexity, readers will also be able to work through the book from the front to back if they are looking to learn test automation with Selenium.

Recipe test scripts

To help readers to learn more effectively, this book has a dedicated site[3] which contains the sample test scripts and related resources.

As an old saying goes, "There's more than one way to skin a cat." You can achieve the same testing outcome with test scripts implemented in different ways. The recipe test scripts in this book are written for simplicity, there is always room for improvement. But for many, to understand the solution quickly and get the job done are probably more important.

If you have a better and simpler way, please let me know.

All recipe test scripts are Selenium 2 (aka Selenium WebDriver WebDriver) compliant, and can be run on Firefox, Chrome and Internet Explorer on multiple platforms. I plan to keep the test scripts updated with the latest stable Selenium version.

Send me feedback

I would appreciate your comments, suggestions, reports on errors in the book and the recipe test scripts. You may submit your feedback on the book's site.

Zhimin Zhan

March 2015

[3] http://zhimin.com/books/selenium-recipes

1. Introduction

Selenium is a free and open source library for automated testing web applications. I assume that you have had some knowledge of Selenium, based on the fact that you picked up this book (or opened it in your eBook reader).

1.1 Selenium

Selenium was originally created in 2004 by Jason Huggins, who was later joined by his other ThoughtWorks colleagues. Selenium supports all major browsers and tests can be written in many programming languages and run on Windows, Linux and Macintosh platforms.

Selenium 2 is merged with another test framework WebDriver (that's why you see 'selenium-webdriver') led by Simon Stewart at Google (update: Simon now works at FaceBook), Selenium 2.0 was released in July 2011.

1.2 Selenium language bindings

Selenium tests can be written in multiple programming languages such as Java, C#, Python, JavaScript and Ruby (the core ones). Quite commonly, I heard the saying such as *"This is a Java project, so we shall write tests in Java as well"*. I disagree. Software testing is to verify whether programmer's work meets customer's needs. In a sense, testers are representing customers. Testers should have more weight on deciding the test syntax than programmers. Plus, why would you mandate that your testers should have the same programming language skills as the programmers. In my subjective view, scripting languages such as Ruby and Python are more suitable for test scripts than compiled languages such as C# and Java (Confession: I have been programming in Java for over 10 years). By the way, we call them test scripts, for a reason.

All examples in this book are written in Selenium with Ruby binding. This does not mean this book is limited to testers/developers who know Ruby. As you will see the examples below, the use of Selenium in different bindings are very similar. Once you master one, you can apply it to others quite easily. Take a look at a simple Selenium test script in five different language bindings: Java, C#, JavaScript, Ruby and Python.

Java:

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class GoogleSearch  {
  public static void main(String[] args) {
    // Create a new instance of the html unit driver
    // Notice that the remainder of the code relies on the interface,
    // not the implementation.
    WebDriver driver = new FirefoxDriver();

    // And now use this to visit Google
    driver.get("http://www.google.com");

    // Find the text input element by its name
    WebElement element = driver.findElement(By.name("q"));

    // Enter something to search for
    element.sendKeys("Hello Selenium WebDriver!");

    // Submit the form based on an element in the form
    element.submit();

    // Check the title of the page
    System.out.println("Page title is: " + driver.getTitle());
  }
}
```

C#:

```
using System;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.Support.UI;

class GoogleSearch
{
  static void Main()
  {
    IWebDriver driver = new FirefoxDriver();
    driver.Navigate().GoToUrl("http://www.google.com");
    IWebElement query = driver.FindElement(By.Name("q"));
    query.SendKeys("Hello Selenium WebDriver!");
    query.Submit();
    Console.WriteLine(driver.Title);
  }
}
```

JavaScript:

```
var webdriver = require('selenium-webdriver');
var driver = new webdriver.Builder()
    .forBrowser('chrome')
    .build();

driver.get('http://www.google.com/ncr');
driver.findElement(webdriver.By.name('q')).sendKeys('webdriver');
driver.findElement(webdriver.By.name('btnG')).click();
driver.wait(webdriver.until.titleIs('webdriver - Google Search'), 1000);
console.log(driver.title);
```

Ruby:

```ruby
require "selenium-webdriver"

driver = Selenium::WebDriver.for :firefox
driver.navigate.to "http://www.google.com"

element = driver.find_element(:name, 'q')
element.send_keys "Hello Selenium WebDriver!"
element.submit

puts driver.title
```

Python:

```python
from selenium import webdriver

driver = webdriver.Firefox()
driver.get("http://www.google.com")

elem = driver.find_element_by_name("q")
elem.send_keys("Hello WebDriver!")
elem.submit()

print(driver.title)
```

1.3 Install Selenium Python

1. Download and install Python.

   Python 2 is pre-installed on Mac and most Linux distributions. For new Python projects, I would recommend Python 3, which was first released in 2008. Here are the instructions to install Python on Windows.

   Download latest installer[1] and run the installer.

[1] https://www.python.org/downloads

Accept all default options except "Add python.exe to Path" for convenience.

2. Install Selenium-WebDriver for Python.

   PIP is the package manager for Python. PIP comes with Python installer.

```
C:\Users\Administrator>pip install selenium
Collecting selenium
  Downloading selenium-2.50.1.tar.gz (809kB)
    100% |################################| 811kB 525kB/s
Installing collected packages: selenium
  Running setup.py install for selenium
Successfully installed selenium-2.50.1
```

3. Your target browses are installed, such as Chrome and Firefox.

Now you are ready to run Selenium script. Type in the above python script (*google search*) in a text editor such as NotePad and save as "google_search.py". Run the command below in a command window.

```
› py google_search.py
```

You will see Firefox browser starting up and performing a Google search.

1.4 Cross browser testing

The biggest advantage of Selenium over other web test frameworks, in my opinion, is that it supports all major web browsers: Firefox, Chrome and Internet Explorer. The browser market nowadays is more diversified (based on the StatsCounter[2], the usage share in December 2015 for Chrome, IE and Firefox are 52.76%, 10.27% and 9.3% respectively). It is logical that all external facing web sites require serious cross-browser testing. Selenium is a natural choice for this purpose, as it far exceeds other commercial tools and free test frameworks.

Firefox

Firefox (up to v46[3]) comes with WebDriver support. geckodriver[4] is required for Firefox 47+.

The test script below (in a file named: *ch01_open_firefox.py*) will open a web site in a new Firefox window.

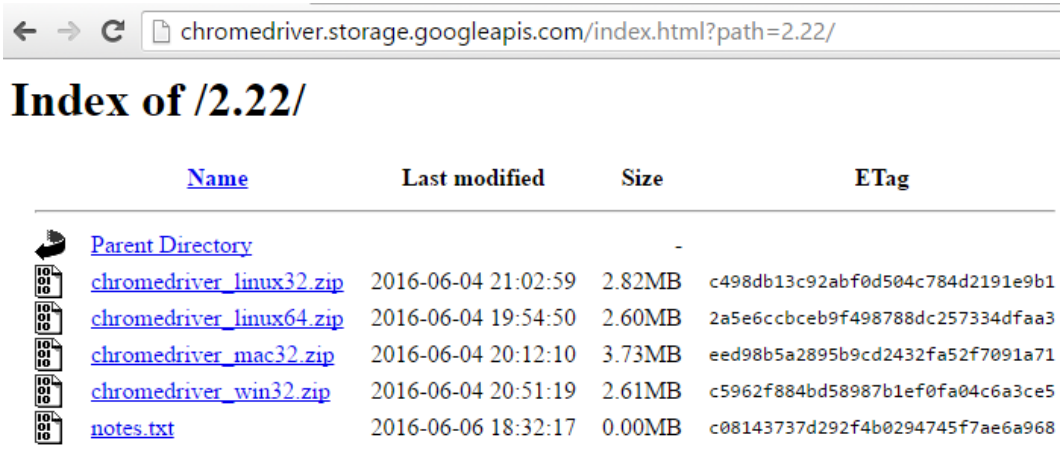
```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("http://testwisely.com/demo")
```

Chrome

To run Selenium tests in Google Chrome, besides the Chrome browser itself, *ChromeDriver* needs to be installed.

Installing ChromeDriver is easy: go to ChromeDriver site[5].

[2] http://en.wikipedia.org/wiki/Usage_share_of_web_browsers
[3] https://download-installer.cdn.mozilla.net/pub/firefox/releases/46.0.1/
[4] https://github.com/mozilla/geckodriver/releases/
[5] https://sites.google.com/a/chromium.org/chromedriver/downloads



Download the one for your target platform, unzip it and put **chromedriver** executable in your PATH. To verify the installation, open a command window (terminal for Unix/Mac), execute command *chromedriver*, You shall see:

```
C:\>chromedriver
Starting ChromeDriver 2.22.397933 (1cab651507b88dec79b2b2a22d1943c01833cc1b) on port 9515
Only local connections are allowed.
```

The test script below opens a site in a new Chrome browser window and closes it one second later.

```
from selenium import webdriver
import time

driver = webdriver.Chrome()
driver.get("http://testwisely.com/demo")
time.sleep(1)
driver.quit()
```

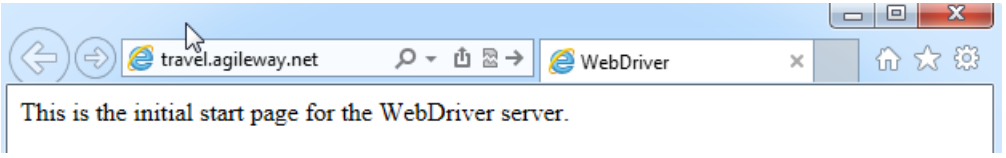
Internet Explorer

Selenium requires IEDriverServer to drive IE browser. Its installation process is very similar to *ChromeDriver*. IEDriverServer is available at http://www.seleniumhq.org/download/[6]. Choose the right one based on your windows version (32 or 64 bit).

[6] http://www.seleniumhq.org/download/

Download version 2.44.0 for (recommended) 32 bit Windows IE or 64 bit Windows IE
CHANGELOG

When a tests starts to execute in IE, before navigating the target test site, you will see this first:



Depending on the version of IE, configurations may be required. Please see IE and IEDriverServer Runtime Configuration[7] for details.

```
from selenium import webdriver
driver = webdriver.Ie()
driver.get("http://testwisely.com/demo")
```

Edge

Edge is Microsoft's new and default web browser on Windows 10. To drive Edge with WebDriver, you need download Microsoft WebDriver[8]. After installation, you will find the executable (*MicrosoftWebDriver.exe*) under *Program Files* folder, add it to your PATH.

However, I couldn't get it working after installing a new version of Microsoft WebDriver. One workaround is to specify the driver path in test scripts specifically:

```
from selenium import webdriver
import time
import os

# copy MicrosoftWebDriver.exe to the test script directory
dir = os.path.dirname(__file__)
edge_path = dir + "\MicrosoftWebDriver.exe"
driver = webdriver.Edge(edge_path)
driver.get("http://testwisely.com/demo")
```

[7] https://code.google.com/p/selenium/wiki/InternetExplorerDriver#Required_Configuration
[8] https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/

1.5 unittest - Python Unit Testing Framework

Selenium drives browsers. However, to make the effective use of Selenium scripts for testing, we need to put them in a test framework that defines test structures and provides assertions (performing checks in test scripts). In this book, I use *unittest*, also known as "PyUnit", the unit testing framework for Python. Here is an example.

```
import unittest
from selenium import webdriver

class FooBarTestCase(unittest.TestCase):

  @classmethod
  def setUpClass(cls):
    cls.driver = webdriver.Chrome()

  @classmethod
  def tearDownClass(cls):
    cls.driver.quit()

  def setUp(self):
    self.driver.get("http://travel.agileway.net")

  def tearDown(self):
    self.driver.find_element_by_link_text("Sign off").click()

  def test_first_case(self):
    self.assertEqual("Agile Travel", self.driver.title)
    self.driver.find_element_by_name("username").send_keys("agileway")
    # ...

  def test_second_case(self):
    self.driver.find_element_by_id("register_link").click()
    # ...
    self.assertIn("Register", self.driver.find_element_by_tag_name("body").t\
ext)
```

The keywords `class`, 'setUpClass, `setUp` and `def test_xxx`' define the structure of a test

script file.

- **`class FooBarTestCase(unittest.TestCase):`**

  Test suite name for grouping related test cases.
- **`setUpClass()`** and **`tearDownClass()`**.

  Optional test statements run before and after all test cases, typically starting a new browser window in `setupClass` and close it in `tearDownClass`.
- **`setUp()`** and **tearDown()**.

  Optional test statements run before and after each test case.
- **`def test_xxx(self):`**

  Individual test cases.
- **Assertions**

  `assertEqual()` and `assertIn` are PyUnit's two assertion methods which are used to perform checks. More assert methods[9]

You will find more about unittest from its home page[10]. However, I honestly don't think it is necessary. The part used for test scripts is not much and quite intuitive. After studying and trying out some examples, you will be quite comfortable with it.

1.6 Run recipe scripts

Test scripts for all recipes can be downloaded from the book site. They are all in ready-to-run state. I include the target web pages/sites as well as Selenium test scripts. There are two kinds of target web pages: local HTML files and web pages on a live site. Running tests written for a live site requires Internet connection.

Run tests in PyCharm IDE

The most convenient way to run one test case or a test suite is to do it in an IDE, such as PyCharm.

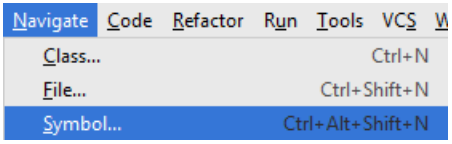
When you have a large number of test cases, then the most effective way to run all tests is done by a Continuous Integration process.

[9] https://docs.python.org/3/library/unittest.html#assert-methods
[10] https://docs.python.org/3/library/unittest.html

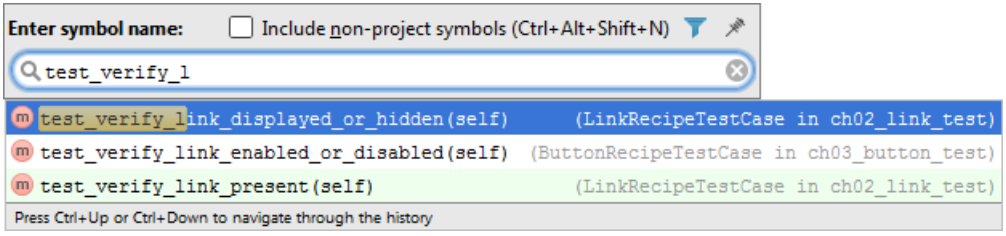
Find the test case

You can locate the recipe either by following the chapter or searching by name. There are over 150 test cases in the recipes test project. Here is the quickest way to find the one you want in PyCharm.

Select menu 'Navigation' → 'Go to Symbol ...'.

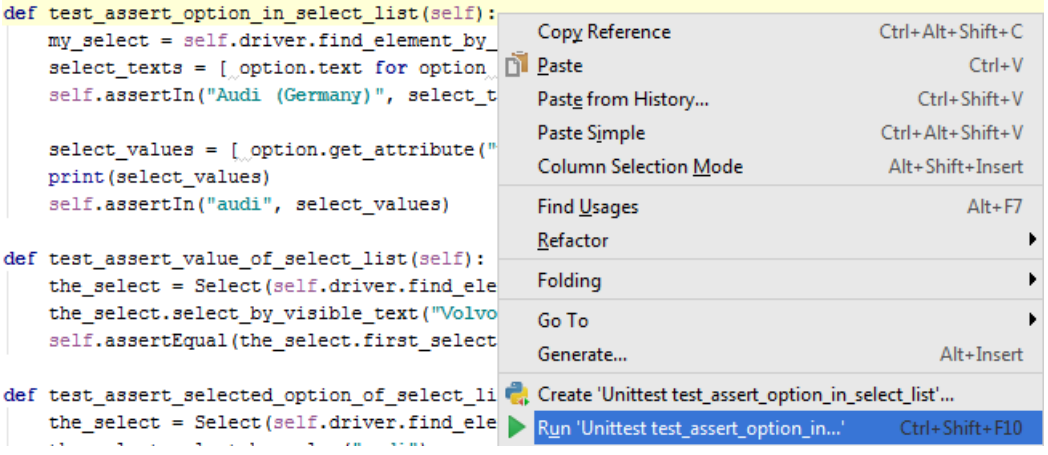


A pop up window lists all test cases in the project for your selection. The finding starts as soon as you type.

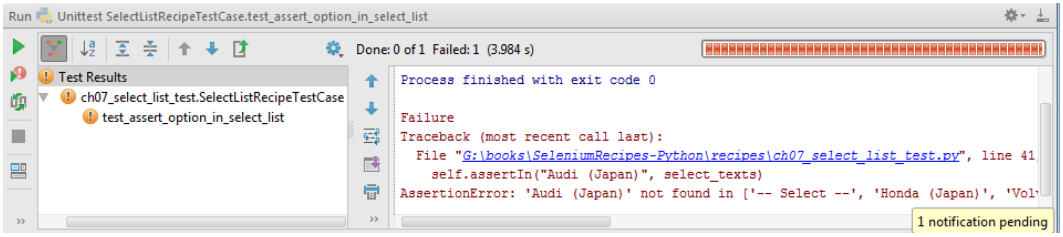


Run individual test case

Move mouse to a line within a test case (starting `def test_xxx(self):`). Right click and select "Run 'Unittest test_xxx'" to run this case.

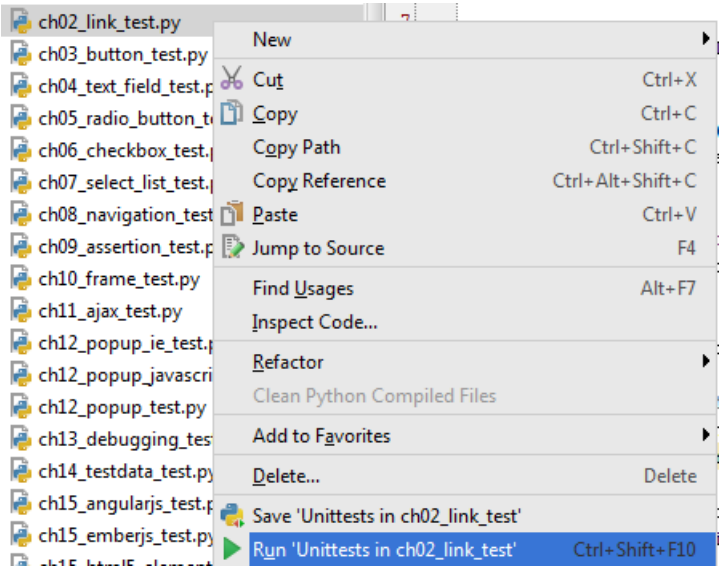


The below is a screenshot of execution panel when one test case failed,

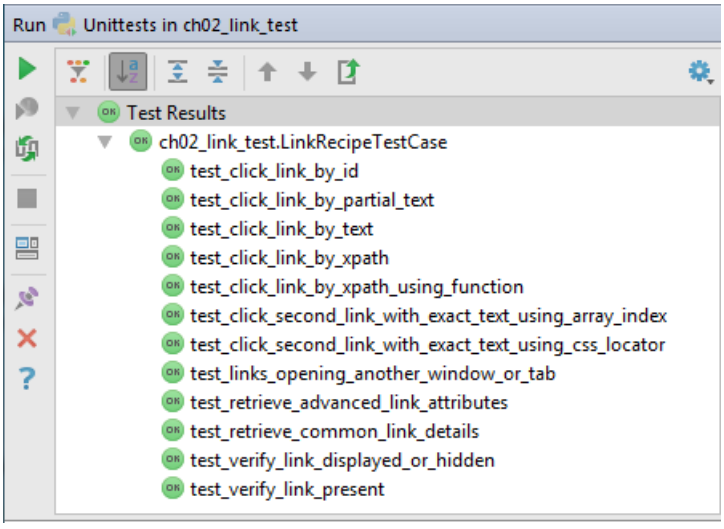


Run all test cases in a test script file

You can also run all test cases in a test script file by right clicking the file name in the project pane and select “Run ‘Unittests in test_file.py’”.



The below is a screenshot of the execution panel when all test cases in a test script file passed,



Run tests from command line

One advantage of open-source test frameworks, such as Selenium, is FREEDOM. You can edit the test scripts in any text editors and run them from a command line.

To run test cases in a test script file (named google_test.py), enter command

```
› python -m unittest google_test.py
```

Run multiple test script files in one go:

```
› python -m unittest first_test.py  second_test.py
```

The command syntax is identical for Mac OS X and Linux platforms.

2. Locating web elements

As you might have already figured out, to drive an element in a page, we need to find it first. Selenium uses what is called locators to find and match the elements on web page. There are 8 locators in Selenium:

Locator	Example
ID	`find_element_by_id("user")`
Name	`find_element_by_name("username")`
Link Text	`find_element_by_link_text("Login")`
Partial Link Text	`find_element_by_partial_link_text("Next")`
XPath	`find_element_by_xpath("//div[@id="login"]/input")`
Tag Name	`find_element_by_tag_name("body")`
Class Name	`find_element_by_class_name("table")`
CSS	`find_element_by_css_selector("#login > input[type="text"]")`

You may use any one of them to narrow down the element you are looking for.

2.1 Start browser

Testing websites starts with a browser. The test script below launches a Firefox browser window and navigate to a site.

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get("http://testwisely.com/demo")
```

Use `webdriver.Chrome` and `webdriver.Ie()` for testing in Chrome and IE respectively.

Test Pages

I prepared the test pages for the recipes, you can download them (in a zip file) at the book's site[a]. Unzip to a local directory and refer to test pages like this:

```
import urllib
# ...
site_path = os.path.dirname(os.path.realpath(__file__)) + "/../site"
site_url  = urllib.request.pathname2url(site_path)

driver.get(site_url + "/locators.html")
```

__file__ is the directory where the test script is.

[3] http://zhimin.com/books/selenium-recipes-python

I recommend, for beginners, to close the browser window at the end of a test case.

```
driver.quit
```

2.2 Find element by ID

Using IDs is the easiest and the safest way to locate an element in HTML. If the page is W3C HTML conformed[1], the IDs should be unique and identified in web controls. In comparison to texts, test scripts that use IDs are less prone to application changes (e.g. developers may decide to change the label, but are less likely to change the ID).

```
driver.find_element_by_id("submit_btn").click    # Button
driver.find_element_by_id("cancel_link").click   # Link
driver.find_element_by_id("username").send_keys("agileway")  # Textfield
driver.find_element_by_id("alert_div").text      # HTML Div element
```

2.3 Find element by Name

The name attributes are used in form controls such as text fields and radio buttons. The values of the name attributes are passed to the server when a form is submitted. In terms of least likelihood of a change, the name attribute is probably only second to ID.

[1] http://www.w3.org/TR/WCAG20-TECHS/H93.html

```
driver.find_element_by_name("comment").send_keys("Selenium Cool")
```

2.4 Find element by Link Text

For Hyperlinks only. Using a link's text is probably the most direct way to click a link, as it is what we see on the page.

```
driver.find_element_by_link_text("Cancel").click
```

2.5 Find element by Partial Link Text

Selenium allows you to identify a hyperlink control with a partial text. This can be quite useful when the text is dynamically generated. In other words, the text on one web page might be different on your next visit. We might be able to use the common text shared by these dynamically generated link texts to identify them.

```
# will click the "Cancel" link
driver.find_element_by_partial_link_text("ance").click
```

2.6 Find element by XPath

XPath, the XML Path Language, is a query language for selecting nodes from an XML document. When a browser renders a web page, it parses it into a DOM tree or similar. XPath can be used to refer a certain node in the DOM tree. If this sounds a little too much technical for you, don't worry, just remember XPath is the most powerful way to find a specific web control.

```
# clicking the checkbox under 'div2' container
driver.find_element_by_xpath("//*[@id='div2']/input[@type='checkbox']").clic\
k()
```

Some testers feel intimidated by the complexity of XPath. However, in practice, there is only limited scope of XPath to master for testers.

Avoid using copied XPath from Browser's Developer Tool

Browser's Developer Tool (right click to select 'Inspect element' to show) is very useful for identifying a web element in web page. You may get the XPath of a web element there, as shown below (in Chrome):

The copied XPath for the second "Click here" link in the example:

```
//*[@id="container"]/div[3]/div[2]/a
```

It works. However, I do not recommend this approach as the test script is fragile. If developer adds another `div` under `<div id='container'>`, the copied XPath is no longer correct for the element while `//div[contains(text(), "Second")]/a[text()="Click here"]` still works.

In summary, XPath is a very powerful way to locating web elements when `by_id`, `by_name` or `by_link_text` are not applicable. Try to use a XPath expression that is less vulnerable to structure changes around the web element.

2.7 Find element by Tag Name

There are a limited set of tag names in HTML. In other words, many elements share the same tag names on a web page. We normally don't use the `tag_name` locator by itself to locate an element. We often use it with others in a chained locators (see the section below). However, there is an exception.

```
driver.find_element_by_tag_name("body").text
```

The above test statement returns the text view of a web page. This is a very useful one as Selenium WebDriver does not have built-in method to return the text of a web page.

2.8 Find element by Class Name

The `class` attribute of a HTML element is used for styling. It can also be used for identifying elements. Commonly, a HTML element's class attribute has multiple values, like below.

```
<a href="back.html" class="btn btn-default">Cancel</a>
<input type="submit" class="btn btn-deault btn-primary">Submit</input>
```

You may use any one of them.

```
driver.find_element_by_class_name("btn-primary").click()  # Submit button
driver.find_element_by_class_name("btn").click()          # Cancel link

# the below will return error "Compound class names not permitted"
# driver.find_element_by_class_name("btn btn-deault btn-primary").click()
```

The `class_name` locator is convenient for testing JavaScript/CSS libraries (such as TinyMCE) which typically use a set of defined class names.

```
# inline editing
driver.find_element_by_id("client_notes").click()
time.sleep(0.5)
driver.find_element_by_class_name("editable-textarea").send_keys("inline not\
es")
time.sleep(0.5)
driver.find_element_by_class_name("editable-submit").click()
```

2.9 Find element by CSS Selector

You may also use CSS Path to locate a web element.

```
driver.find_element_by_css_selector("#div2 > input[type='checkbox']").click()
```

However, the use of CSS selector is generally more prone to structure changes of a web page.

2.10 Chain find_element to find child elements

For a page containing more than one elements with the same attributes, like the one below, we could use `XPath` locator.

```
<div id="div1">
  <input type="checkbox" name="same" value="on"> Same checkbox in Div 1
</div>
<div id="div2">
  <input type="checkbox" name="same" value="on"> Same checkbox in Div 2
</div>
```

There is another way: chain `find_element` to find a child element.

```
driver.find_element_by_id("div2").find_element_by_name("same").click()
```

2.11 Find multiple elements

As its name suggests, `find_elements` return a list of matched elements. Its syntax is exactly the same as `find_element`, i.e. can use any of 8 locators.

The test statements will find two checkboxes under `div#container` and click the second one.

```
checkbox_elems = driver.find_elements_by_xpath("//div[@id='container']//inpu\
t[@type='checkbox']")
print(len(checkbox_elems))  # => 2
checkbox_elems[1].click()
```

Sometimes `find_element` fails due to multiple matching elements on a page, which you were not aware of. `find_elements` will come in handy to find them out.

3. Hyperlink

Hyperlinks (or links) are fundamental elements of web pages. As a matter of fact, it is hyperlinks that makes the World Wide Web possible. A sample link is provided below, along with the HTML source.

Recommend Selenium

HTML Source

```
<a href="index.html" id="recommend_selenium_link" class="nav" data-id="123" \
style="font-size: 14px;">Recommend Selenium</a>
```

3.1 Start browser

Testing web sites starts with a browser.

```
from selenium import webdriver
driver  = webdriver.Firefox()
driver.get("http://testwisely.com/demo")
```

Use `webdriver.Chrome()` and `webdriver.Ie()` for testing in Chrome and IE respectively.

I recommend, for beginners, closing the browser window at the end of a test case.

```
driver.quit()
```

3.2 Click a link by text

Using text is probably the most direct way to click a link in Selenium, as it is what we see on the page.

```
driver.find_element_by_link_text("Recommend Selenium").click()
```

A Note for testers coming from Ruby

In Ruby, round brackets `()` is optional when calling a function.

```
driver.find_element(:link_text, "Register").click   # OK
```

However, the below test script won't work in Python.

```
driver.find_element_by_link_text("Register").click    # wrong
driver.find_element_by_link_text("Register").click()  # correct
```

This syntax issue is not a problem if it throws an error, but it does not. This test statement will execute, but not clicking the link!

3.3 Click a link by ID

```
driver.find_element_by_id("recommend_selenium_link").click()
```

Furthermore, if you are testing a web site with multiple languages, using IDs is probably the only feasible option. You do not want to write test scripts like below:

```
if is_italian?
  driver.find_element_by_link_text("Accedi").click
elsif is_chinese?  # a helper function determines the locale
  driver.find_element_by_link_text("□□").click
else
  driver.find_element_by_link_text("Sign in").click
end
```

3.4 Click a link by partial text

```
driver.find_element_by_partial_link_text("partial").click()
```

3.5 Click a link by XPath

The example below is finding a link with text 'Recommend Selenium' under a `<p>` tag.

```
driver.find_element_by_xpath("//p/a[text()='Recommend Selenium']").click()
```

Your might say the example before (find by :link_text) is simpler and more intuitive, that's correct. but let's examine another example:

First div Click here
Second div Click here

On this page, there are two 'Click here' links.

HTML Source

```
<div>
  First div
  <a href="link-url.html">Click here</a>
</div>
<div>
  Second div
  <a href="link-partial.html">Click here</a>
</div>
```

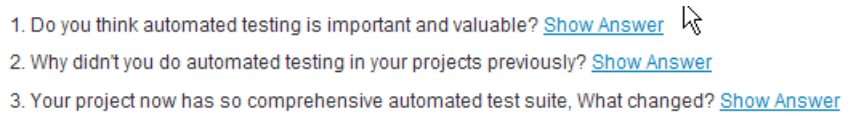
If a test case requires you to click the second 'Click here' link, the simple `find_element_by_link_text('Click here')` won't work (as it clicks the first one). Here is a way to accomplish using XPath:

```
driver.find_element_by_xpath('//div[contains(text(), "Second")]/a[text()="Cl\
ick here"]').click()
```

3.6 Click Nth link with exact same label

It is not uncommon that there are more than one link with exactly the same text. By default, Selenium will choose the first one. What if you want to click the second or Nth one?

The web page below contains three 'Show Answer" links,



To click the second one,

```
driver.find_elements_by_link_text("Show Answer")[1].click() # second link
```

`find_elements_xxx` return a list (also called array) of web controls matching the criteria in appearing order. Selenium (in fact Python) uses 0-based indexing, i.e., the first one is 0.

3.7 Click Nth link by CSS

You may also use CSS Path to locate a web element.

```
# the 3rd link
driver.find_element_by_css_selector("p  > a:nth-child(3)").click()
```

However, generally speaking, the use of stylesheet is more prone to changes.

3.8 Verify a link present or not?

```
assertTrue(driver.find_element_by_link_text("Recommend Selenium").is_display\
ed())
driver.find_element_by_link_text("Hide").click()
time.sleep(1) # delay 1 second
driver.find_element_by_link_text("Hide").click()
try:
  # different from Watir, selenium returns element not found if hidden
  # the below will throw NoSuchElementException
  self.assertFalse(driver.find_element_by_link_text("Recommend Selenium").is\
_displayed())
except:
  print("[Selenium] The hidden link cannot be found")

driver.find_element_by_link_text("Show").click()
time.sleep(1)
self.assertTrue(driver.find_element_by_link_text("Recommend Selenium").is_di\
splayed())
```

3.9 Getting link data attributes

Once a web control is identified, we can get its other attributes of the element. This is generally applicable to most of the controls.

```
assertIn("/site/index.html", driver.find_element_by_link_text("Recommend Sel\
enium").get_attribute("href"))
assertEqual(driver.find_element_by_link_text("Recommend Selenium").get_attri\
bute("id"), "recommend_selenium_link")
assertEqual(driver.find_element_by_id("recommend_selenium_link").text, "Reco\
mmend Selenium")
assertEqual(driver.find_element_by_id("recommend_selenium_link").tag_name, "\
a")
```

Also you can get the value of custom attributes of this element and its inline CSS style.

```
assertEqual(driver.find_element_by_id("recommend_selenium_link").get_attribu\
te("style"), "font-size: 14px;")
assertEqual(driver.find_element_by_id("recommend_selenium_link").get_attribu\
te("data-id"), "123")
```

3.10 Test links open a new browser window

Clicking the link below will open the linked URL in a new browser window or tab.

```
<a href="http://testwisely.com/demo" target="_blank">Open new window</a>
```

While we could use `switch_to` method (see chapter 10) to find the new browser window, it will be easier to perform all testing within one browser window. Here is how:

```
current_url = driver.current_url
new_window_url = driver.find_element_by_link_text("Open new window").get_att\
ribute("href")
driver.get(new_window_url)
# ... testing on new site
driver.find_element_by_name("name").send_keys("sometext")
driver.get(current_url) # back
```

In this test script, we use a local variable (a programming term) ‘current_url’ to store the current URL.

Resources

Books

- **Practical Web Test Automation**[1] by Zhimin Zhan

  Solving individual selenium challenges (what this book is for) is far from achieving test automation success. *Practical Web Test Automation* is the book to guide you to the test automation success, topics include:
  - Developing easy to read and maintain Watir/Selenium tests using next-generation functional testing tool
  - Page object model
  - Functional Testing Refactorings
  - Cross-browser testing against IE, Firefox and Chrome
  - Setting up continuous testing server to manage execution of a large number of automated UI tests
  - Requirement traceability matrix
  - Strategies on team collaboration and test automation adoption in projects and organizations
- **Selenium WebDriver Recipes in Ruby**[2] by Zhimin Zhan

  Selenium WebDriver recipe tests in Ruby, another popular script language that is quite similar to Python.
- **Selenium WebDriver Recipes in Java**[3] by Zhimin Zhan

  Sometimes you might be required to write Selenium WebDriver tests in Java. Master Selenium WebDriver in Java quickly by leveraging this book.
- **Selenium WebDriver Recipes in C#, 2nd Edition**[4] by Zhimin Zhan

  Selenium WebDriver recipe tests in C#, another popular language that is quite similar to Java.

[1] https://leanpub.com/practical-web-test-automation
[2] https://leanpub.com/selenium-recipes-in-ruby
[3] https://leanpub.com/selenium-recipes-in-java
[4] http://www.apress.com/9781484217412

- **Selenium WebDriver Recipes in Node.js**[5] by Zhimin Zhan

  Selenium WebDriver recipe tests in Node.js, a very fast implementation of WebDriver in JavaScript.
- **Watir Recipes**[6] by Zhimin Zhan

  While Selenium WebDriver is far more popular than Watir, in my opinion, Watir started real open-source testing for web applications. Learning Watir will help you design better test scripts. *(Jari Bakken, the author for selenium-webdriver gem, started with Watir, is also the author of watir-webdriver).*

Web Sites

- **Selenium Python API**[7]

  The API has searchable interface, The *Locators* and *Element* class are particularly important:
  - Locate elements By[8]
  - Element[9]
- **Python unitest**[10]

  Python's unit testing framework.
- **Selenium Home** (http://seleniumhq.org[11])

Tools

- **PyCharm**[12]

  Python IDE from JetBrains, the community edition is free.
- **BuildWise** (http://testwisely.com/buildwise[13])

  AgileWay's free and open-source continuous build server, purposely designed for running automated UI tests with quick feedback.

[5] https://leanpub.com/selenium-webdriver-recipes-in-nodejs
[6] https://leanpub.com/watir-recipes
[7] http://selenium-python.readthedocs.io/api.html
[8] http://selenium-python.readthedocs.io/api.html?highlight=find_element#locate-elements-by
[9] http://selenium-python.readthedocs.io/api.html#module-selenium.webdriver.remote.webelement
[10] https://docs.python.org/3/library/unittest.html
[11] http://seleniumhq.org
[12] https://www.jetbrains.com/pycharm
[13] http://testwisely.com/buildwise