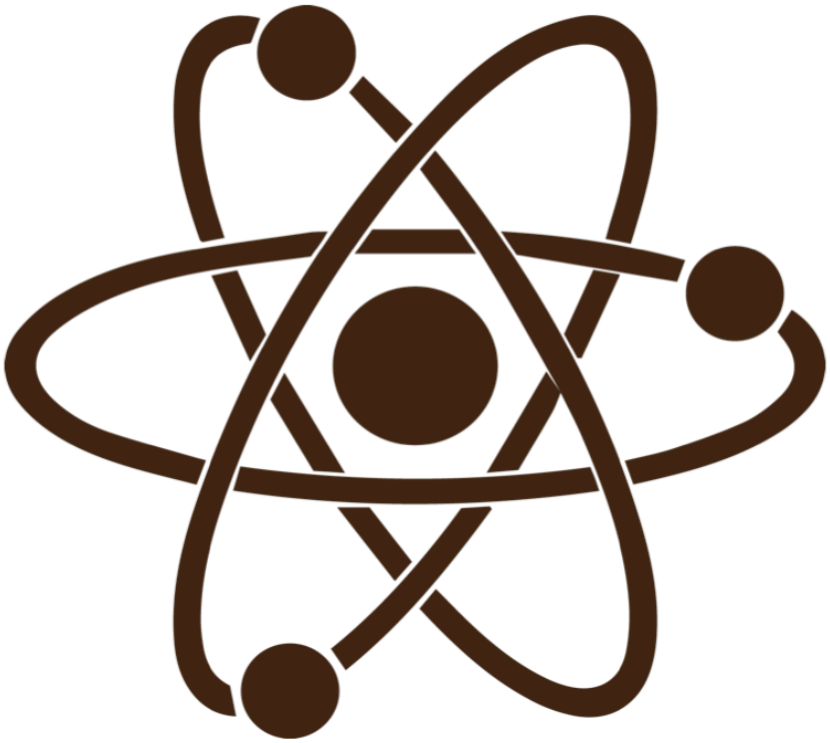
Elemental Selenium



Python Tips

By Dave Haeffner

Version 1.0.0

Preface

This book is a compendium of tips from my weekly Selenium tip newsletter (Elemental Selenium). Its aim is to give you a glimpse into the things you'll see in the wild, and a reference for how to deal with them.

These tips were originally only available in Ruby and I've been steadily converting them over to the officially supported programming languages offered by Selenium. I'm excited to finally add Python to the collection. The initial push was thanks to the contributions I've received since open-sourcing my example code (which you can see here).

A big thanks to Mike Millgate, Isaul Vargas, and Peter Bittner for their contributions! Mike provided code for five of the tips in this compendium, with help from Isaul & Peter who provided code reviews. Thank you!

The tips differ from The Selenium Guidebook in that they do not build upon previous examples. Instead, they serve as standalone works that can be consumed individually. So feel free to read them in order or jump around.

Enjoy!

Table of Contents

1. How To Upload a File
2. How To Download a File
3. How To Download a File Without a Browser
4. How To Take A Screenshot on Failure
5. How To Use Selenium Grid
6. How To Opt-out of A/B Tests
7. How To Access Basic Auth
8. How To Test Checkboxes
9. How To Test For Disabled Elements
10. How To Select From a Dropdown List
11. How To Work with Frames
12. How To Add Growl Notifications To Your Tests
13. How To Visually Verify Your Locators
14. How To Work With Hovers
15. How To Work With HTML Data Tables
16. How To Work With JavaScript Alerts
17. How To Press Keyboard Keys
18. How To Work with Multiple Windows
19. How To Right-click

Chapter 1

How To Upload a File

The Problem

Uploading a file is a common piece of functionality found on the web. But when trying to automate it you get prompted with a with a dialog box that is just out of reach for Selenium.

In these cases people often look to a third-party tool to manipulate this window (e.g., AutoIt). While this can help solve your short-term need, it sets you up for failure later by chaining you to a specific platform (e.g., AutoIt only works on Windows), effectively limiting your ability to test this functionality on different browser & operating system combinations.

A Solution

A work-around for this problem is to side-step the system dialog box entirely. We can do this by using Selenium to insert the full path of the file we want to upload (as text) into the form and then submit the form.

Let's step through an example.

An Example

NOTE: We are using a file upload example found on the-internet.

First let's pull in our requisite libraries for interacting with the operating system (e.g., `import os`), our testing framework (e.g., `import unittest`), and driving the browser with Selenium (e.g., `from selenium import webdriver`).

```
# filename: upload.py

import os
import unittest
from selenium import webdriver

# ...
```

Now to create a new class file and add in the test's setup and teardown.

```
# filename: upload.py
# ...
class Upload(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
```

After specifying the class and establishing inheritance so it's a test case for `unittest` (e.g., `class Upload(unitest.TestCase)`) we create methods. The first method, `setUp(self):`, will execute before each test in this class. In it we are launching a new instance of Firefox with Selenium and storing it in a class variable that we'll use throughout this class (e.g., `self.driver = webdriver.Firefox()`). After our test executes the second method, `tearDown(self):`, will execute. This calls `self.driver.quit()` which ends the session by closing the browser instance and the destroying the Selenium object in-memory.

Now to wire up our test.

```
# filename: upload.py
# ...
    def test_example_1(self):
        driver = self.driver
        filename = 'some-file.txt'
        file = os.path.join(os.getcwd(), filename)
        driver.get('http://the-internet.herokuapp.com/upload')
        driver.find_element_by_id('file-upload').send_keys(file)
        driver.find_element_by_id('file-submit').click()

        uploaded_file = driver.find_element_by_id('uploaded-files').text
        assert uploaded_file == filename, "uploaded file should be %s" % filename

if __name__ == "__main__":
    unittest.main()
```

After declaring the test method (e.g., `def test_example_1(self):`) we store `self.driver` in a local variable (e.g., `driver = self.driver`). This way we don't have to litter our test method with `self.driver` and can call `driver` instead. Next we specify the file we'd like to upload. There's already a text file called `some-file.txt` in the current working directory, so we grab both the file name and its full path, storing both of these values in variables (e.g., `filename` and `file` respectively).

Next we visit the page with the upload form, input the string value of `file` (e.g., the full path to the file plus the filename with its extension), and submit the form. After the file is uploaded to the

page it will display the filename it just processed. We use this text on the page to perform our assertion (making sure the uploaded file is what we expect).

The two lines at the end of the file are so the file can be executed directly from the command-line.

Expected Behavior

When we save this file and run it (e.g., `python upload.py` from the command-line) this is what will happen:

- Open the browser
- Visit the upload form page
- Inject the file path into the form and submit it
- Page displays the uploaded filename
- Grab the text from the page and assert it's what we expect
- Close the browser

Outro

This approach will work across all browsers. If you want to use it with a remote instance (e.g., on Selenium Grid or Sauce Labs) then you'll want to have a look at `file_detector`.

Happy Testing!

Chapter 2

How To Download a File

The Problem

Just like with uploading files we hit the same issue with downloading them. A dialog box just out of Selenium's reach.

A Solution

With some additional configuration when setting up Selenium we can easily side-step the dialog box. This is done by instructing the browser to download files to a specific location without triggering the dialog box.

After the file is downloaded we can perform some simple checks to make sure the file is what we expect.

Let's dig in with an example.

An Example

Let's start by pulling in our requisite libraries for interacting with the operating system (e.g., `import os`), creating a temporary directory and cleaning it up, using our testing framework (e.g., `import unittest`), and driving the browser with Selenium (e.g., `from selenium import webdriver`).

```
# filename: download.py
import os
import time
import shutil
import tempfile
import unittest
from selenium import webdriver

# ...
```

Now to create a test class and add our test's setup.

```
# filename: download.py
class Download(unittest.TestCase):

    def setUp(self):
        self.download_dir = tempfile.mkdtemp()

        profile = webdriver.FirefoxProfile()
        profile.set_preference("browser.download.dir", self.download_dir)
        profile.set_preference("browser.download.folderList", 2)
        profile.set_preference(
            "browser.helperApps.neverAsk.saveToDisk",
            "images/jpeg, application/pdf, application/octet-stream")
        profile.set_preference("pdfjs.disabled", True)
        self.driver = webdriver.Firefox(firefox_profile=profile)

# ...
```

Our `setUp(self):` method is where the magic is happening in this example. In it we're creating a uniquely named temp directory (e.g., `self.download_dir = tempfile.mkdtemp()`), configuring a browser profile object (for Firefox in this case), and plying it with the necessary configuration parameters to make it automatically download the file where we want (e.g., in the newly created temp directory).

Here's a breakdown of each of the browser preferences being set:

- `browser.download.dir` accepts a string. This is how we set the custom download path. It needs to be an absolute path.
- `browser.download.folderList` takes a number. It tells Firefox which download directory to use. `2` tells it to use a custom download path, wheras `1` would use the browser's default path, and `0` would place them on the Desktop.
- `browser.helperApps.neverAsk.saveToDisk` tells Firefox when not to prompt for a file download. It accepts a string of the file's MIME type. If you want to specify more than one, you do it with a comma-separated string (which we've done).
- `pdfjs.disabled` is for when downloading PDFs. This overrides the sensible default in Firefox that previews PDFs in the browser. It accepts a boolean.

This profile object is then passed into our instance of Selenium (e.g., `self.driver = webdriver.Firefox(firefox_profile=profile)`).

Now let's take care of our test's teardown.

```
# filename: download.py
# ...
  def tearDown(self):
      self.driver.quit()
      shutil.rmtree(self.download_dir)
# ...
```

In `tearDown(self):` we close the browser instance and then clean up the temp directory by deleting it, which will recursively delete the files in the folder before deleting it.

Now to wire up our test.

```
# filename: download.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/download')
        download_link = driver.find_element_by_css_selector('.example a')
        download_link.click()

        time.sleep(1.0)  # necessary for slow download speeds

        files = os.listdir(self.download_dir)
        files = [os.path.join(self.download_dir, f)
                 for f in files]  # add directory to each filename
        assert len(files) > 0, "no files were downloaded"
        assert os.path.getsize(files[0]) > 0, "downloaded file was empty"

if __name__ == "__main__":
    unittest.main()
```

After visiting the page we find the first download link and click it. The click triggers an automatic download to the temp directory created in `setUp()`. We need to wait for the download to finish, so we add a brief sleep (e.g., `time.sleep(1.0)`). After the file downloads, we perform some rudimentary checks to make sure the unique temp directory isn't empty and then check the file to see that it isn't empty either.

The last two lines of the file are so the file can be executed directly from the command-line.

Expected Behavior

When we save this file and run it (e.g., `python download.py` from the command-line) this is what will happen:

- Create a uniquely named temp directory in the present working directory
- Open the browser
- Visit the page
- Find and click the first download link on the page
- Automatically download the file to the temp directory without prompting
- Check that the temp directory is not empty
- Check that the downloaded file is not empty
- Close the browser
- Delete the temp directory

Outro

A similar approach can be applied to some other browsers with varying configurations. But downloading files this way is not sustainable or recommended. Mark Collin articulates this point well in his prominent write-up about it here. In a future tip I'll cover a more reliable, faster, and scalable browser agnostic approach to downloading files. Stay tuned.

Happy Testing!

Chapter 3

How To Download a File Without a Browser

The Problem

In a previous chapter we stepped through how to download files with Selenium by configuring the browser to download them locally and verifying their file size when done.

While this works it requires a custom configuration that is inconsistent from browser to browser.

A Solution

Ultimately we shouldn't care if a file was downloaded or not. Instead, we should care that a file can be downloaded. And we can do that by using an HTTP client alongside Selenium in our test.

With an HTTP library we can perform a header (or `HEAD`) request for the file. Instead of downloading the file we'll receive header information for the file which contains information like the content type and content length (amongst other things). With this information we can easily confirm the file is what we expect without onerous configuration, local disk usage, or lengthy download times (depending on the file size).

Let's dig with an example.

An Example

To start things off let's pull in our requisite libraries (`import unittest` for our test framework, `from selenium import webdriver` to drive the browser, and `import httplib` for our HTTP library), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: file_download_revisited.py
import unittest
from selenium import webdriver
import httplib # Use http.client if using Python 3.x.x


class FileDownloadRevisited(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
# ...
```

Now we're ready to wire up our test.

It's just a simple matter of visiting the page with download links, grabbing a URL from one of them, and performing a `HEAD` request with it.

```
# filename: file_download_revisited.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/download')
        download_link = driver.find_element_by_css_selector('.example a').get_attribute
('href')

        connection = httplib.HTTPConnection('the-internet.herokuapp.com')
        connection.request('HEAD', download_link)
        response = connection.getresponse()
        content_type = response.getheader('Content-type')
        content_length = response.getheader('Content-length')

        assert content_type == 'application/octet-stream'
        assert content_length > 0

if __name__ == "__main__":
    unittest.main()
```

Once we receive the response we can check its header for the `Content-type` and `Content-length` to make sure the file is the correct type and not empty.

Expected Behavior

When you save this and run it (e.g., `python file_download_revisited.py` from the command-line) here is what will will happen:

- Open the browser
- Load the page
- Grab the URL of the first download link
- Perform a `HEAD` request against it with an HTTP library
- Store the response
- Check the response headers to see that the file type is correct
- Check the response headers to see that the file is not empty

Outro

Compared to the browser specific configuration with Selenium this is hands down a leaner, faster, and more maintainable approach. But unfortunately it only works with files served up from a flat URL. So if you're trying to test file downloads that are generated in-memory as part of the browser session (a.k.a. not accessible from a URL) then you'll need to reach for the browser specific Selenium configuration.

Happy Testing!

Chapter 4

How To Take A Screenshot on Failure

The Problem

With browser tests it can often be challenging to track down the issue that caused a failure. By itself a failure message along with a stack trace is hardly enough to go on. Especially when you run the test again and it passes.

A Solution

A simple way to gain insight into your test failures is to capture screenshots at the moment of failure. And it's a quick and easy thing to add to your tests.

Let's dig in with an example.

An Example

Let's start by importing our requisite libraries (`import unittest` for our test framework, `from selenium import webdriver` to drive the browser, and `import sys` to determine when there's a test failure), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: screenshot.py
import sys
import unittest
from selenium import webdriver


class ScreenShotOnFailure(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        if sys.exc_info()[0]:
            self.driver.save_screenshot("failshot_%s.png" % self._testMethodName)
        self.driver.quit()
# ...
```

In `tearDown` we check to see if `sys.exc_info()[0]` exists. If it does, then there's been a test failure and we capture a screenshot through the help of Selenium's `.save_screenshot` method.

`.save_screenshot` accepts a filename as a string (e.g., `'failshot.png'`). To make the filename unique we use the test method name (e.g., `self._testMethodName`). When this command executes it will save an image file to the local system in the current working directory.

Now to wire up a test which will fail.

```
# filename: screenshot.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com')
        assert driver.title == 'blah blah blah'

if __name__ == "__main__":
    unittest.main()
```

Expected Behavior

When we save this file and run it (`python screenshot.py` from the command-line) here is what will happen:

- Open the browser
- Load the homepage of the-internet
- Check the text of the page header and fail
- Output a failure message in the terminal
- Capture a screenshot in the current working directory
- Close the browser

Outro

Happy Testing!

Chapter 5

How To Use Selenium Grid

The Problem

If you're looking to run your tests on different browser and operating system combinations but you're unable to justify using a third-party solution like Sauce Labs then what do you do?

A Solution

With Selenium Grid you can stand up a simple infrastructure of various browsers on different operating systems to not only distribute test load, but also give you a diversity of browsers to work with.

A brief Selenium Grid primer

Selenium Grid is part of the Selenium project. It lets you distribute test execution across several machines. You can connect to it with Selenium Remote by specifying the browser, browser version, and operating system you want. You specify these values through Selenium Remote's `Capabilities`.

There are two main elements to Selenium Grid -- a hub, and nodes. First you need to stand up a hub. Then you can connect (or "register") nodes to that hub. Nodes are where your tests will run, and the hub is responsible for making sure your tests end up on the right one (e.g., the machine with the operating system and browser you specified in your test).

Let's step through an example.

An Example

Part 1: Grid Setup

Selenium Grid comes built into the Selenium Standalone Server. So to get started we'll need to download the latest version of it from here.

Then we need to start the hub.

```
> java -jar selenium-server-standalone.jar -role hub
19:05:12.718 INFO - Launching Selenium Grid hub
...
```

After that we can register nodes to it.

```
> java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register
19:05:57.880 INFO - Launching a Selenium Grid node
...
```

NOTE: This example only demonstrates a single node on the same machine as the hub. To span nodes across multiple machines you will need to place the standalone server on each machine and launch it with the same registration command (replacing **`http://localhost`** with the location of your hub, and specifying additional parameters as needed).

Now that the grid is running we can view the available browsers by visiting our Grid's console at `http://localhost:4444/grid/console`.

To refine the list of available browsers, we can specify an additional `-browser` parameter when registering the node. For instance, if we wanted to only offer Safari on a node, we could specify it with `-browser browserName=safari`, which would look like this:

```
java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register -browser browserName=safari
```

We could also repeat this parameter again if we wanted to explicitly specify more than one browser.

```
java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register -browser browserName=safari -browser browserName=
chrome -browser browserName=firefox
```

There are numerous parameters that we can use at run time. You can see a full list here.

Part 2: Test Setup

Now let's wire up a simple test script to use our new Grid.

First we'll need to pull in our requisite libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: grid.py
import unittest
from selenium import webdriver


class Grid(unittest.TestCase):

    def setUp(self):
        url = 'http://localhost:4444/wd/hub'
        desired_caps = {}
        desired_caps['browserName'] = 'firefox'
        self.driver = webdriver.Remote(url, desired_caps)

    def tearDown(self):
        self.driver.quit()

    def test_page_loaded(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com')
        assert driver.title == 'The Internet'

if __name__ == "__main__":
    unittest.main()
```

Notice in `setUp` we're using Selenium Remote (e.g., `webdriver.Remote`) to connect to the grid. And we are telling the grid which browser we want to use with a `desired_caps` dictionary (e.g., `desired_caps['browserName'] = 'firefox'`).

You can see a full list of the available Selenium `Capabilities` options here.

Expected Behavior

When we save this file and run it (e.g., `python grid.py` from the command-line) here is what will happen:

- test connects to the grid hub
- hub determines which node has the necessary browser/platform combination
- hub opens an instance of the browser on the found node
- test runs on the new browser instance
- test completes and the browser closes on the node

Outro

If you're looking to set up Selenium Grid to work with Internet Explorer or Chrome, be sure to read up on how to set them up since there is additional configuration required for each. And if you run into issues, be sure to check out the browser driver documentation for the browser you're working with:

- ChromeDriver
- EdgeDriver
- FirefoxDriver
- InternetExplorerDriver
- SafariDriver

Also, it's worth noting that while Selenium Grid is a great option for scaling your test infrastructure, it by itself will NOT give you parallelization. That is to say, it can handle as many connections as you throw at it (within reason), but you will still need to find a way to execute your tests in parallel.

Happy Testing!

Chapter 6

How To Opt-out of A/B Tests

The Problem

Occasionally when running tests you may see unexpected behavior due to A/B testing (a.k.a. split testing) of the application you're working with.

In order to keep your tests running without issue we need a clean way to opt-out of these split tests.

A quick primer on A/B testing

Split testing is a simple way to experiment with an application's features to see which changes lead to higher user engagement.

A simple example would be testing variations of an e-mail landing page to see if more people sign up. In such a split test there would be the control (how the application looks and behaves now) and variants (e.g., 2 or 3 changes that could include changing text or images on the page, element positioning, color of the submit button, etc).

Once the variants are configured, they are put into rotation, and the experiment starts. During this experiment each user will see a different version of the feature and their engagement with it will be tracked. Split tests live for the length of the experiment or until a winner is found (e.g., tracking indicates that a variant converted higher than the control). If no winner is found, new variants may be created and another experiment tried. If a winner is found, then the winning variant becomes the new control and the feature gets updated accordingly.

A Solution

Thankfully there are some standard opt-out mechanisms built into A/B testing platforms. They tend to come in the form of an appended URL or forging a cookie. Let's dig in with an example of each approach with a popular A/B testing platform, Optimizely.

An Example

Our example page is from the-internet and can be seen here. There are three different versions of the page that are available. On each page the heading text will vary:

- Control: `A/B Test Control`
- Variation 1: `A/B Test Variation 1`
- Opt-out: `No A/B Test`

Let's kick things off by loading our requisite libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: ab_test_opt_out.py
import unittest
from selenium import webdriver


class ABTestOptOut(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
# ...
```

Now let's wire up our first test to step through loading the split testing page and verifying that the text changes after we forge an opt-out cookie.

```
# filename: ab_test_opt_out.py
# ...
    def test_forge_cookie_on_target_page(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/abtest')
        heading_text = driver.find_element_by_tag_name('h3').text
        assert heading_text in ['A/B Test Variation 1', 'A/B Test Control']
        driver.add_cookie({'name' : 'optimizelyOptOut', 'value' : 'true'})
        driver.refresh()
        heading_text = driver.find_element_by_tag_name('h3').text
        assert heading_text == 'No A/B Test'

# ...
```

After navigating to the page we confirm that we are in one of the A/B test groups by grabbing the heading text and checking to see if it matches what we expect. After that we add the opt-out cookie, refresh the page, and then confirm that we are no longer in the A/B test group by checking the heading text again.

We could also load the opt-out cookie before navigating to this page.

```
# filename: ab_test_opt_out.py
# ...
    def test_forge_cookie_on_homepage_then_navigate_to_target_page(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com')
        driver.add_cookie({'name' : 'optimizelyOptOut', 'value' : 'true'})
        driver.get('http://the-internet.herokuapp.com/abtest')
        heading_text = driver.find_element_by_tag_name('h3').text
        assert heading_text == 'No A/B Test'

# ...
```

Here we are navigating to the main page of the site first and then adding the opt-out cookie. After that we navigate to the split test page and then perform our check. Alternatively, we could opt out without forging a cookie. Instead we just need to append an opt out parameter to the URL.

```
# filename: ab_test_opt_out.py
    def test_url_parameter(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/abtest?optimizely_opt_out=true')
        driver.switch_to.alert.dismiss()
        heading_text = driver.find_element_by_tag_name('h3').text
        assert heading_text == 'No A/B Test'

if __name__ == "__main__":
    unittest.main()
```

By appending `?optimizely_opt_out=true` we achieve the same outcome as before. Keep in mind that this approach triggers a JavaScript alert, so we have to switch to and dismiss it (e.g., `driver.switch_to.alert.dismiss()`) before performing our check.

Expected Behavior

When we save this file and run it (e.g., `python ab_test_opt_out.py` from the command-line) here is what will happen with either of the tests:

- Open the browser
- Opt-out of the split tests (either by cookie or appended URL)
- Visit the split testing page
- Grab the header text
- Confirm that the session is opted out of the split test
- Close the browser

Outro

Happy Testing!

Chapter 7

How To Access Basic Auth

The Problem

Sometimes you'll work with applications that are secured behind Basic HTTP Authentication (a.k.a. Basic Auth). In order to access them you'll need to pass credentials to the site when requesting a page. Otherwise you'll get a system level pop-up prompting you for a username and password -- rendering Selenium helpless.

Before Selenium 2 we were able to accomplish this by injecting credentials into a custom header, but now the cool kid way to do it it was something like BrowserMob Proxy. And some people are solving this with browser specific configurations too.

But all of this feels heavy. Instead, let's look at a simple approach that is browser agnostic and quick to setup.

A Solution

By specifying the username and password in the URL when visiting a page with Selenium, we can side-step the system level dialog box and avoid setting up a proxy server. This approach will work for both HTTP or HTTPS pages.

Let's take a look at an example.

An Example

Let's start by requiring our requisite libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: basic_auth_1.py

import unittest
from selenium import webdriver


class BasicAuth1(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
```

Now to add our test.

```
# filename: basic_auth_1.py
# ...
    def test_visit_basic_auth_secured_page(self):
        driver = self.driver
        driver.get('http://admin:admin@the-internet.herokuapp.com/basic_auth')
        page_message = driver.find_element_by_css_selector('.example p').text
        assert page_message == 'Congratulations! You must have the proper credentials.'

if __name__ == "__main__":
    unittest.main()
```

In the test we're loading the page by passing in the username and password in the front of the URL (e.g., `http://admin:admin@`). Once it loads we grab text from the page to make sure we ended up in the right place.

Alternatively, we could have accessed this page as part of the test setup (after creating an instance of Selenium). This would have cached the Basic Auth session in the browser, enabling us to visit the page again without having to specify credentials. This is particularly useful if you have numerous pages behind Basic Auth.

Here's what that would look like.

```
# filename: basic_auth_2.py
import unittest
from selenium import webdriver


class BasicAuth1(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get('http://admin:admin@the-internet.herokuapp.com/basic_auth')

    def tearDown(self):
        self.driver.quit()

    def test_visit_basic_auth_secured_page(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/basic_auth')
        page_message = driver.find_element_by_css_selector('.example p').text
        assert page_message == 'Congratulations! You must have the proper credentials.'

if __name__ == "__main__":
    unittest.main()
```

NOTE: If your application serves both HTTP and HTTPS pages from behind Basic Auth then you will need to load one of each type before executing your test steps. Otherwise you will get authorization errors when switching between HTTP and HTTPS because the browser can't use Basic Auth credentials interchangeably (e.g. HTTP for HTTPS and vice versa).

Expected Behavior

When you save the first example and run it (e.g., `python basic_auth_1.py`), here is what will happen:

- Open the browser
- Visit the page using Basic Auth
- Get the page text
- Assert that the text is what we expect
- Close the browser

And when you save the second example and run it (e.g., `python basic_auth_2.py`), here is what will happen:

- Open the browser
- Visit the page using Basic Auth in the setup
- Navigate to the Basic Auth page (without providing credentials)

- Get the page text
- Assert that the text is what we expect

Outro

Hopefully this tip will help save you from getting tripped by Basic Auth when you come across it.

Happy Testing!

Chapter 8

How To Test Checkboxes

The Problem

Checkboxes are an often used element in web applications. But how do you work with them in your Selenium tests? Intuitively you may reach for a method that has the word 'checked' in it -- like `.checked` or `.is_checked`. But this doesn't exist in Selenium. So how do you do it?

A Solution

There are two ways to approach this -- by seeing if an element has a `checked` attribute (a.k.a. performing an attribute lookup), or by asking an element if it has been selected.

Let's step through each approach to see their pros and cons.

An Example

For reference, here is the markup from the page we will be testing against (an example from the-internet).

```
<form>
  <input type="checkbox"> unchecked<br>
  <input type="checkbox" checked=""> checked
</form>
```

Let's kick things off by requiring our dependent libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: checkboxes.py
import unittest
from selenium import webdriver


class Checkboxes(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Before we dig into writing a test to verify the state of the page, let's walk through both checkbox approaches to see what Selenium gives us.

To do that we'll want to grab all of the checkboxes on the page and iterate through them. Once using an attribute lookup and again asking if the element is selected. For each we'll output the return values we get from Selenium.

```
# filename: checkboxes.py
# ...
    def test_list_values_for_different_approaches(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/checkboxes')
        checkboxes = driver.find_elements_by_css_selector('input[type="checkbox"]')

        print("With .get_attribute('checked')")
        for checkbox in checkboxes:
            print(checkbox.get_attribute('checked'))

        print("\nWith .is_selected")
        for checkbox in checkboxes:
            print(checkbox.is_selected())

if __name__ == "__main__":
    unittest.main()
```

When we save our file and run it (e.g., `python checkboxes.py` from the command-line), here is the output we'll see.

```
With .attribute('checked')
None
true

With .is_selected
False
True
```

With the attribute lookup, depending on the state of the checkbox, we receive either a `None` string value or a `true` boolean value. Whereas with `.is_selected` we get a boolean value either way.

Let's see what these approaches look like when put to use in our test.

```
# filename: checkboxes.py
# ...
    def test_list_values_for_different_approaches(self):
        # ...
        assert checkboxes[-1].get_attribute('checked')
        # or
        assert checkboxes[-1].is_selected()

if __name__ == "__main__":
    unittest.main()
```

With either approach we can simply assert on the return value (even if it's a string value of `'None'`) and have things work as expected. We can confirm this by asserting on the checkbox which is not selected.

```
# filename: checkboxes.py
# ...
    def test_list_values_for_different_approaches(self):
        # ...
        assert checkboxes[0].get_attribute('checked')
        # or
        assert checkboxes[0].is_selected()
```

An `AssertionError` will be raised for either assertion.

```
======================================================================
FAIL: test_list_values_for_different_approaches (__main__.Checkboxes)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "45-checkboxes/python/checkboxes.py", line 31, in
test_list_values_for_different_approaches
    assert checkboxes[0].get_attribute('checked')
AssertionError

----------------------------------------------------------------------
Ran 1 test in 3.356s

FAILED (failures=1)

shell returned 1
```

Expected Behavior

When we save and run the file (e.g., `python checkboxes.py` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find all of the checkboxes on the page
- Assert that the last checkbox (the one that is supposed to be checked on initial page load) is checked
- Close the browser

Outro

Attribute lookups are generally meant for pulling information out of the page for review. However in this case they lend themselves to seeing if a checkbox is checked. But there is a method which was built for this use case that is more readable and has a predictable set of return values. So `is_selected` should be the thing you reach for, knowing that an attribute lookup is there as a solid backup if you find you need it.

Happy Testing!

Chapter 9

How To Test For Disabled Elements

The Problem

On occasion you may have the need to check if an element on a page is disabled or enabled. Sounds simple enough, but how do you do it? It's not a well known function of Selenium. So doing a trivial action like this can quickly become a pain.

A Solution

If we look at the API documentation for Selenium's Element class we can see there is an available method called `is_enabled` that can help us accomplish what we want.

Let's take a look at how to use it.

An Example

For this example we will use a dropdown list from the-internet. In this list there a few options to select, one which should be disabled. Let's find this element and assert that it is disabled. First let's require our dependent libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: disabled_elements.py
import unittest
from selenium import webdriver


class DisabledElements(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Now let's wire up our test.

```
# filename: disabled_elements.py
# ...
    def test_dropdown(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/dropdown')
        dropdown_list = driver.find_elements_by_tag_name('option')
        assert dropdown_list[0].is_enabled() is False

if __name__ == "__main__":
    unittest.main()
```

After loading the page, we find all of the elements that have an option tag (which are all of the items in the dropdown list). This returns a list of elements, so we use the first one (which is the one with the text of `'Please select an option'`).

Once we have the element we want we see if it's enabled (with `.is_enabled`) and assert based on the response.

And since we grabbed all of the dropdown list options, we can easily test the opposite case by checking the second or third option in the list.

```
assert dropdown_list[1].is_enabled() is True
```

Expected Behavior

When we save this file and run it (e.g., `python disabled_elements.py` from the command-line) here is what will happen:

- Open a browser
- Visit the page
- Grab all dropdown list elements
- Assert that the first element in the list is not enabled
- Assert that the second element in the list is enabled
- Close the browser

Outro

Hopefully this tip has helped make the simple task of seeing if an element is enabled or disabled more approachable.

Happy Testing!

Chapter 10

How To Select From a Dropdown List

The Problem

Selecting from a dropdown list seems like one of those simple things. Just grab the list by it's element and select an item within it based on the text you want.

While it sounds pretty straightforward, there is a bit more finesse to it.

Let's take a look at a couple of different approaches.

An Example

First let's pull in our requisite libraries, declare the test class, and wire up some simple `setUp` and `tearDown` methods.

```
# filename: dropdown.py
import unittest
from selenium import webdriver


class DropDown(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Now lets' wire up our test.

```
# filename: dropdown.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/dropdown')
        dropdown_list = driver.find_element_by_id('dropdown')
        options = dropdown_list.find_elements_by_tag_name('option')
        for opt in options:
            if opt.text == 'Option 1':
                opt.click()
                break
        for opt in options:
            if opt.is_selected():
                selected_option = opt.text
                break
        assert selected_option == 'Option 1', "Selected option should be Option 1"
```

After visiting the example application we find the dropdown list by its ID and store it in a variable. We then find each clickable element in the dropdown list (e.g., each `option`) with `find_elements_by_tag_name`.

Grabbing all of the options returns a collection that we iterate over and when the text matches what we want it will click on it.

We finish the test by performing a check to see that our selection was made correctly. This is done by reiterating over the dropdown options collection one more time. This time we're getting the text of the item that was selected, storing it in a variable, and making an assertion against it.

While this works, there is a simpler, built-in way to do this with Selenium. Let's give that a go.

Another Example

```
# filename: dropdown.py
import unittest
from selenium import webdriver
from selenium.webdriver.support.select import Select as WebDriverSelect
# ...

    def test_example_2(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/dropdown')
        dropdown = driver.find_element_by_id('dropdown')
        select_list = WebDriverSelect(dropdown)
        select_list.select_by_visible_text('Option 1')
        selected_option = select_list.first_selected_option.text
        assert selected_option == 'Option 1', "Selected option should be Option 1"

if __name__ == "__main__":
    unittest.main()
```

Similar to the first example, we are finding the dropdown list by its ID. But instead of iterating over its option elements and clicking based on a conditional check, we are leveraging a built-in helper function of Selenium, `Select`, and its `select_by_visibile_text` method to choose the item we want.

We then ask the `select_list` what option was selected by using the `first_selected_option` method. This returns a Selenium Element that we grab the text from, storing it in a variable (e.g., `selected_option`).

Then we perform our assertion against this variable (just like in the previous example).

NOTE: In addition to selecting by text, you can also select by value.

```
select_list.select_by_value('1')
```

Expected Behavior

If you save this file with either of these examples and run it (e.g., `python dropdown.py` from the command-line) here is what will happen:

- Open the browser
- Visit the example application
- Find the dropdown list

- Select the requested item from the dropdown list
- Assert that the selected option is the one you expect
- Close the browser

Outro

Hopefully this tip will help you breeze through selecting items from a dropdown list.

Happy Testing!

Chapter 11

How To Work with Frames

The Problem

On occasion you'll run into a relic of the front-end world -- frames. And when writing a test against them, you can easily get tripped up if you're not paying attention.

A Solution

Rather than gnash your teeth when authoring your tests, you can easily work with the elements in a frame by telling Selenium to switch to that frame first. Then the rest of your test should be business as usual.

Let's dig in with some examples.

An Example

We'll first need to pull in our requisite libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: frames.py

import unittest
from selenium import webdriver


class Frames(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Now onto our test. In it we'll step through an example of nested frames which can be found on the-internet.

```
# filename: frames.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/nested_frames')
        driver.switch_to.frame('frame-top')
        driver.switch_to.frame('frame-middle')
        assert driver.find_element_by_id('content').text == "MIDDLE", "content should
be MIDDLE"

# ...
```

With Selenium's `.switch_to.frame` method we can easily switch to the frame we want. It accepts either an ID or name attribute. But in order to get the text of the middle frame (e.g., a frame nested within another frame), we need to switch to the parent frame (e.g., the top frame) and then switch to the child frame (e.g., the middle frame).

Once we've done that we're able to find the element we need, grab its text, and assert that it's what we expect.

While this example helps illustrate the point of frame switching, it's not very practical.

A More Practical Example

Here is a more likely example you'll run into -- working with a WYSIWYG Editor like TinyMCE. You can see the page we're testing here.

```
# filename: frames.py
# ...
    def test_example_2(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/tinymce')
        driver.switch_to.frame('mce_0_ifr')
        editor = driver.find_element_by_id('tinymce')
        before_text = editor.text
        editor.clear()
        editor.send_keys('Hello World!')
        after_text = editor.text
        assert after_text != before_text, "%s equals %s" % (before_text, after_text)


if __name__ == "__main__":
    unittest.main()
```

Once the page loads we switch into the frame that contains TinyMCE and...

- grab the original text and store it
- clear and input new text
- grab the new text value
- assert that the original and new texts are not the same

Keep in mind that if we need to access a part of the page outside of the frame we are currently in we'll need to switch to it. Thankfully Selenium has method that enables us to quickly jump back to the top level of the page -- `switch_to.default_content`.

Here is what that looks like in practice.

```
driver.switch_to.default_content()
  assert driver.find_element_by_css_selector('h3').text != "", "element should not be empty"
```

Expected Behavior

If we save the file and run it (e.g., `python frames.py` from the command-line) here is what will happen:

Example 1

- Open the browser
- Visit the page
- Switch to the nested frame
- Grab the text from the frame and assert that Selenium is in the correct place
- Close the browser

Example 2

- Open the browser
- Visit the page
- Switch to the frame that contains the TinyMCE editor
- Grab and clear the text in the editor
- Input and grab new text in the edtitor
- Assert that the original and new text entries don't match
- Switch to the top level of the page
- Grab the text from the top of the page and assert that it's not empty
- Close the browser

Outro

Now you're ready to handily defeat frames when they cross your path.

Happy Testing!

Chapter 12

How To Add Growl Notifications To Your Tests

The Problem

Good test reports are a fundamental component of successful test automation. But running down a test failure by looking at a test report can be a real pain sometimes.

Leaving you with no choice but to roll up your sleeves and get your hands dirty with debug statements, or step through things piece by piece -- all for the sake of trying to track down a transient issue.

A Solution

By leveraging something like jQuery Growl you can output non-interactive debugging statements directly to the page you're testing. This way you can see helpful information and more-likely correlate it to the test actions that are being taken. This can a boon for your test runs when coupled with screenshots and/or video recordings of your test runs

Let's step through an example of how to set this up.

An Example

First we'll need to pull in our requisite libraries (`import unittest` for our test framework, `from selenium import webdriver` to drive the browser, and `import time` to add a delay in our script so we're able to see the notification messages), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: growl.py
import unittest
from selenium import webdriver
import time


class Growl(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Now for our test. We'll need to visit the page we want to display notifications on and do some work with JavaScript to load jQuery, jQuery Growl, and styles for jQuery Growl. After that we can issue commands to jQuery Growl to make notification messages display on the page.

```
# filename: growl.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com')

        # Check for jQuery on the page, add it if needbe
        driver.execute_script(
            "if (!window.jQuery) {"
            "var jquery = document.createElement('script');"
            "jquery.type = 'text/javascript';"
            "jquery.src =
'https://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js';"
            "document.getElementsByTagName('head')[0].appendChild(jquery);}")

        # Use jQuery to add jquery-growl to the page
        driver.execute_script(
"$.getScript('http://the-internet.herokuapp.com/js/vendor/jquery.growl.js')")

        # Use jQuery to add jquery-growl styles to the page
        driver.execute_script(
            "$('head').append('"
            "<link rel=stylesheet "
            "href=http://the-internet.herokuapp.com/css/jquery.growl.css "
            "type=text/css />');")

        # jquery-growl w/ no frills
        driver.execute_script("$.growl({ title: 'GET', message: '/' });")

        time.sleep(5)

if __name__ == "__main__":
    unittest.main()
```

And if we wanted to see color-coded notifications, then we could use one of the following:

```
driver.execute_script("$.growl.error({ title: 'ERROR', message: 'your error message
goes here' });")
        driver.execute_script("$.growl.notice({ title: 'Notice', message: 'your notice
message goes here' });")
        driver.execute_script("$.growl.warning({ title: 'Warning!', message: 'your
warning message goes here' });")
```

Expected Behavior

When we save this file and run it (e.g., `python growl.py`) here is what will happen:

- Browser opens
- Load the page
- Add jQuery, jQuery Growl, and jQuery Growl notifications to the page
- Display a set of notification messages in the top-right corner of the page
- Notification messages disappear
- Browser closes

Outro

In order to use this approach, you will need to load jQuery Growl on every page you want to display output to -- which can be a bit of overhead. But if you want rich messaging like this then that's the price you have to pay (unless you can get your team to add it to the application under test).

In a future tip I'll step through how to access Selenium logging output so we can wire it up to these notifications.

I'd like to give a big thanks to Jon Austen (Twitter, GitHub) for giving me the idea to use jQuery Growl with Selenium.

Happy Testing!

Chapter 13

How To Visually Verify Your Locators

This is a pseudo guest post from Brian Goad. I've adapted a blog post of his with permission. You can see the original here. Brian is a Test Engineer at Digitalsmiths. You can follow him on Twitter at @bbbco and check out his testing blog here.

The Problem

It's likely that you'll run into odd test behavior that makes you question the locators you're using in a test. But how do you interrogate your locators to make sure they are doing what you expect?

A Solution

By leveraging some simple JavaScript and CSS styling, we can highlight the element on the page that we're targeting. This way we can visually inspect it to make sure it is the one that we want.

Let's take a look at an example.

An Example

For our initial setup let's pull in our requisite libraries (`import unittest` for our test framework, `from selenium import webdriver` to drive the browser, and `import time` to add a delay in our script so we're able to see the notification messages), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: highlight_elements.py
import unittest
from selenium import webdriver
import time


class HighlightElements(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Now let's create a `highlight` helper method that will accept a Selenium WebDriver `element` and a time to wait (e.g., `duration`) as arguments.

By setting a duration, we can control how long to highlight an element on the page before reverting the styling back. And we can make this an optional argument by setting a default value for it (e.g., 3 seconds).

```
# filename: highlight_elements.py
# ...
    def highlight(self, element, duration=3):
        driver = self.driver

        # Store original style so it can be reset later
        original_style = element.get_attribute("style")

        # Style element with dashed red border
        driver.execute_script(
            "arguments[0].setAttribute(arguments[1], arguments[2])",
            element,
            "style",
            "border: 2px solid red; border-style: dashed;")

        # Keep element highlighted for a spell and then revert
        if (duration > 0):
            time.sleep(duration)
            driver.execute_script(
                "arguments[0].setAttribute(arguments[1], arguments[2])",
                element,
                "style",
                original_style)

# ...
```

There are three things going on here.

1. We store the style of the element so we can revert it back when we're done
2. We change the style of the element so it visually stands out (e.g., a red dashed border)
3. We revert the style of the element back after 3 seconds

We're accomplishing the style change through JavaScript's `setAttribute` function. And we're executing it with Selenium's `execute_script` command.

Now to use this in our test is simple, we just prepend a `find_element` command with a call to the `highlight` method.

```
# filename: highlight_element.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/large')
        self.highlight(driver.find_element_by_id('sibling-2.3'))

if __name__ == "__main__":
    unittest.main()
```

Expected Behavior

When we save this file and run it (e.g., `python highlight_elements.py` from the command-line) here is what will happen.

- Browser opens
- Load the page
- Find the element
- Change the styling of the element so it has a red dashed-line border
- Wait 3 seconds
- Revert the styling to remove the border
- Browser closes

Outro

If you wanted to take this a step further, you could leverage this approach along with an interactive debugger. You can read more about how to do that here in Brian's other guest post.

Alternatively, you could verify your locators by using a browser plugin like FireFinder. You can read more about how to do that in tip 35.

Happy Testing!

Chapter 14

How To Work With Hovers

The Problem

If you need to work with mouse hovers in your tests it may not be obvious how to do this with Selenium. And a quick search through the documentation can easily leave you befuddled forcing you to go spelunking through StackOverflow for the solution.

A Solution

By leveraging Selenium's Action Builder (a.k.a. ActionChains in the Python Selenium bindings) we can handle more complex user interactions like hovers. This is done by telling Selenium which element we want to move the mouse to, and then performing what we need to after.

Let's dig in with an example.

An Example

Our example application is available here on the-internet. It has a few avatars displayed in a grid layout. When you hover over each of them, they display additional user information and a link to view a full profile.

Let's write a test that will hover over the first avatar and make sure that this additional information appears.

First we'll include our requisite libraries, declare the test class, and wire up some simple `setUp` and `tearDown` methods.

```
# filename: hovers.py
import unittest
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains


class Hovers(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Now let's write our test.

```
# filename: hovers.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/hovers')
        avatar = driver.find_element_by_class_name('figure')
        ActionChains(driver).move_to_element(avatar).perform()
        avatar_caption = driver.find_element_by_class_name('figcaption')
        assert avatar_caption.is_displayed()

if __name__ == "__main__":
    unittest.main()
```

After loading the page we find the first avatar and store it in a variable (`avatar`). We then use the `.move_to_element` method and feed it the avatar variable (which triggers the hover).

We then check to see if the additional user information is displayed with `.is_displayed` in our assertion.

Expected Behavior

When we save this file and run it (e.g., `python hover.py` from the command-line) here is what will happen:

- Open the browser
- Visit the page

- Hover over the first avatar
- Assert that the caption appeared on the page
- Close the browser

Outro

Happy Testing!

Chapter 15

How To Work With HTML Data Tables

The Problem

Odds are at some point you've come across the use of tables in a web application to display data or information to a user, giving them the option to sort and manipulate it. Depending on your application it can be quite common and something you will want to write an automated test for.

But when the table has no helpful, semantic markup (e.g. easy to use `id` or `class` attributes) it quickly becomes more difficult to work with and write tests against it. And if you're able to pull something together, it will likely not work against older browsers.

A Solution

You can easily traverse a table through the use of CSS Pseudo-classes.

But keep in mind that if you care about older browsers (e.g., Internet Explorer 8, et al), then this approach won't work on them. In those cases your best bet is to find a workable solution for the short term and get a front-end developer to update the table with helpful attributes.

A quick primer on Tables and CSS Pseudo-classes

Understanding the broad strokes of an HTML table's structure goes a long way in writing effective automation against it. So here's a quick primer.

A table has...

- a header (e.g. `<thead>`)
- a body (e.g. `<tbody>`).
- rows (e.g. `<tr>`) -- horizontal slats of data
- columns -- vertical slats of data

Columns are made up of cells which are...

- a header (e.g., `<th>`)
- one or more standard cells (e.g., `<td>` -- which is short for table data)

CSS Pseudo-classes work by walking through the structure of an object and targeting a specific part of it based on a relative number (e.g. the third `<td>` cell from a row in the table body). This

works well with tables since we can grab all instances of a target (e.g. the third `<td>` cell from each `<tr>` in the table body) and use it in our test -- which would give us all of the data for the third column.

Let's step through some examples for a common set of table functionality like sorting columns in ascending and descending order.

An Example

NOTE: You can see the application under test here. It's an example from the-internet. In the example there are 2 tables. We will start with the first table and then work with the second.

We kick things off by pulling in our requisite libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: tables.py
import unittest
from selenium import webdriver


class Tables(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()

# ...
```

Here is the markup from the first table example we're working with. Note that it does not have any `id` or `class` attributes.

```
<table id="table1" class="tablesorter">
    <thead>
      <tr>
        <th><span>Last Name</span></th>
        <th><span>First Name</span></th>
        <th><span>Email</span></th>
        <th><span>Due</span></th>
        <th><span>Web Site</span></th>
        <th><span>Action</span></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Smith</td>
        <td>John</td>
        <td>jsmith@gmail.com</td>
        <td>$50.00</td>
        <td>http://www.jsmith.com</td>
        <td>
          <a href='#edit'>edit</a>
          <a href='#delete'>delete</a>
        </td>
      </tr>
```

There are 6 columns (`Last Name` , `First Name` , `Email` , `Due` , `Web Site` , and `Action`). Each one is sortable by clicking on the column header. The first click should sort them in ascending order, the second click in descending order.

There is a small sampling of data in the table to work with (4 rows worth). So we should be able to sort the data, grab it, and confirm that it sorted correctly. So lets do that in our first test with the `Due` column using a CSS Pseudo Class.

```
# filename: tables.py
# ...
    def test_sort_number_column_in_ascending_order_with_limited_locators(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/tables')
        driver.find_element_by_css_selector('#table1 thead tr th:nth-of-type(4)').click
()
        due_column = driver.find_elements_by_css_selector('#table1 tbody tr
td:nth-of-type(4)')
        dues = [float(due.text.replace('$','')) for due in due_column]
        assert dues == sorted(dues)

# ...
```

After loading the page we find and click the column heading that we want with a CSS Pseudo-class (e.g. `#table1 thead tr th:nth-of-type(4)`). This locator targets the 4th `<th>` element in the table heading section (e.g., `<thead>`) (which is the `Due` column heading).

We then use another pseudo-class to find all `<td>` elements within the `Due` column by looking for the 4th `<td>` of each row in the table body. Once we have them we grab each of their text values, clean them up (`.replace('$','')`), convert them to a number (`flat()`), and store them all in a list called `dues`. We then compare this collection to a sorted version of itself to see if they match. If they do, then the `Due` column was sorted in ascending order and the test will pass.

If we wanted to test for descending order, we would need to click the `Due` heading twice after loading the page. Other than that the code is identical except for the assertion which is checking the same thing but reversing the sort order.

```
# filename: tables.py
# ...
    def test_sort_number_column_in_descending_order_with_limited_locators(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/tables')
        driver.find_element_by_css_selector('#table1 thead tr th:nth-of-type(4)').click
()
        driver.find_element_by_css_selector('#table1 thead tr th:nth-of-type(4)').click
()
        due_column = driver.find_elements_by_css_selector('#table1 tbody tr
td:nth-of-type(4)')
        dues = [float(due.text.replace('$','')) for due in due_column]
        assert dues == sorted(dues, reverse=True)

# ...
```

We can easily use this approach to test a different column (e.g., one that doesn't deal with numbers) and see that it gets sorted correctly too. Here's a test that exercises the `Email` column.

```
# filename: tables.py
# ...
    def test_sort_text_column_in_ascending_order_with_limited_locators(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/tables')
        driver.find_element_by_css_selector('#table1 thead tr th:nth-of-type(3)').click
()
        email_column = driver.find_elements_by_css_selector('#table1 tbody tr
td:nth-of-type(3)')
        emails = [email.text for email in email_column]
        assert emails == sorted(emails)

# ...
```

The mechanism for this is the same as before, except that we don't need to clean the text up or convert it before performing our assertion.

But What About Older Browsers?

Now we have some working tests that will load the page and check sorting for a couple of columns in both ascending and descending order. Great! But if we run these again an older browser (e.g., Internet Explorer 8, etc.) it will throw an exception stating `Unable to find element`. This is because older browsers don't support CSS Pseudo-classes.

You've come a long way, so it's best to get value out of what you've just written. To do that you can run these tests on current browsers and submit a request to your front-end developers to update the table markup with some semantic `class` attributes. Later, when these new locators have been implemented on the page, you can revisit these tests and update them accordingly.

Here is markup of what our original table would look like with some helpful attributes added in. It's also the markup from the second table on our application under test.

```
<table id="table2" class="tablesorter">
    <thead>
      <tr>
        <th><span class='last-name'>Last Name</span></th>
        <th><span class='first-name'>First Name</span></th>
        <th><span class='email'>Email</span></th>
        <th><span class='dues'>Due</span></th>
        <th><span class='web-site'>Web Site</span></th>
        <th><span class='action'>Action</span></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td class='last-name'>Smith</td>
        <td class='first-name'>John</td>
        <td class='email'>jsmith@gmail.com</td>
        <td class='dues'>$50.00</td>
        <td class='web-site'>http://www.jsmith.com</td>
        <td class='action'>
          <a href='#edit'>edit</a>
          <a href='#delete'>delete</a>
        </td>
      </tr>
```

With these new attributes the locators in our sorting tests become a lot simpler and more expressive. Let's write a new `Due` ascending order test to demonstrate.

```
# filename: tables.py
# ...
    def test_sort_number_column_in_ascending_order_with_helpful_locators(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/tables')
        driver.find_element_by_css_selector('#table2 thead .dues').click()
        due_column = driver.find_elements_by_css_selector('#table2 tbody .dues')
        dues = [float(due.text.replace('$','')) for due in due_column]
        assert dues == sorted(dues)

if __name__ == "__main__":
    unittest.main()
```

Not only will these selectors work in current and older browsers, but they are also more resilient to changes in the table layout since they are not using hard-coded numbers that rely on the column order.

Expected Behavior

When we save this file and run it (e.g., `python tables.py` from the command-line) here is what will happen:

- Browser opens
- Load the page
- Click the column heading
- Grab the values for the target column
- Assert that the column is sorted in the correct order (ascending or descending depending on the test)
- Close the browser

Outro

CSS Pseudo-classes are a great resource and unlock a lot of potential for your tests. They enable a bit of CSS gymnastics but that's only assuming you've come up with a test strategy that rules out older browsers. If you don't have a test strategy or are curious to see how yours compares, check out tip 18.

For more info on CSS Pseudo-classes see this write-up by Sauce Labs, and maybe the W3C spec for CSS3 if you're feeling adventurous. And for a more in-depth walk-through on HTML Table design check out Treehouse's write-up here.

Happy Testing!

Chapter 16

How To Work With JavaScript Alerts

The Problem

If your application triggers any JavaScript pop-ups (a.k.a. alerts, dialogs, etc.) then you need to know how to handle them in your Selenium tests.

A Solution

Built into Selenium is the ability to switch to an alert window and either accept or dismiss it. This way your tests can continue unencumbered by dialog boxes that may feel just out of reach.

Let's dig in with an example.

An Example

Our example application is available here on the-internet. It has various JavaScript Alerts available (e.g., an alert, a confirmation, and a prompt). Let's demonstrate testing a confirmation dialog (e.g., a prompt which asks the user to click `Ok` or `Cancel`).

First, we'll include our requisite libraries, declare the test class, and wire up some simple `setUp` and `tearDown` methods.

```
# filename: javascript_alerts.py
import unittest
from selenium import webdriver


class JavaScriptAlerts(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
# ...
```

Now let's write our test.

```
# filename: javascript_alerts.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/javascript_alerts')
        driver.find_elements_by_tag_name('button')[1].click()
        popup = driver.switch_to.alert
        popup.accept()
        result = driver.find_element_by_id('result').text
        assert result == 'You clicked: Ok'

if __name__ == "__main__":
    unittest.main()
```

A quick glance at the page's markup shows that there are no unique IDs on the buttons. So to click on the second button (to trigger the JavaScript confirmation dialog) we find all of the buttons on the page using `find_elements` and click on the second one. Since `find_elements` returns a list of all found elements, we can assume that the first item can be selected using `[0]` (since lists in Python start counting at `0`). So the second item would be `[1]`.

After click the button to trigger the JavaScript Alert we use Selenium's `switch_to.alert` method to focus on the JavaScript pop-up and use `.accept()` to click `Ok`. If we wanted to click `Cancel` we would have used `.dismiss()`.

After accepting the alert, our main browser window will automatically regain focus and the page will display the result that we chose. This text is what we use for our assertion, making sure that the words `You clicked: Ok` are displayed on the page.

Expected Behavior

When we save this file and run it (e.g., `python javascript_alerts.py` from the command-line) here is what will happen:

- Open the browser
- Load the page
- Click the second button on the page
- JavaScript Confirmation Alert appears
- Accept the JavaScript Confirmation Alert
- Assert that the result on the page is what we expect
- Close the browser

Outro

Happy Testing!

Chapter 17

How To Press Keyboard Keys

The Problem

On occasion you'll come across functionality that requires the use of keyboard key presses in your tests.

Perhaps you'll need to tab to traverse from one portion of the page to another, back out of some kind of menu or overlay with the escape key, or even submit a form with Enter.

But how do you do it and where do you start?

A Solution

You can easily issue a key press by using the `send_keys` command.

This can be done to a specific element, or generically with Selenium's Action Builder (which has been documented on the Selenium project's Wiki page for Advanced User Interactions). Either approach will send a key press. The latter will send it to the element that's currently in focus in the browser (so you don't have to specify a locator along with it), whereas the prior approach will send the key press directly to the element found.

Let's step through a couple of examples.

An Example

First we'll include our requisite libraries, declare the test class, and wire up some simple `setUp` and `tearDown` methods.

```
# filename: keyboard_keys.py
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.action_chains import ActionChains

class KeyboardKeys(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
# ...
```

Now we can wire up our test.

Let's use an example from the-internet that will display what key has been pressed (link). We'll use the result text that gets displayed to perform our assertion.

```
# filename: keyboard_keys.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/key_presses')
        driver.find_element_by_class_name('example').send_keys(Keys.SPACE)
        assert driver.find_element_by_id('result').text == 'You entered: SPACE'
        ActionChains(driver).send_keys(Keys.TAB).perform()
        assert driver.find_element_by_id('result').text == 'You entered: TAB'

if __name__ == "__main__":
    unittest.main()
```

After visiting the page we find an element that's visible (e.g., the div that contains the example on the page) and send the space key to it (e.g., `.send_keys(Keys.SPACE`). Then we grab the resulting text (e.g., `driver.find_element_by_id('result').text`) and assert that it says what we expect (e.g., `'You entered: SPACE'`).

Alternatively, we can also issue a key press without finding the element first by using the Action Builder (e.g., `ActionChains`).

```
# filename: keyboard_keys.py
# ...
    def test_example_1(self):
        # ...
        ActionChains(driver).send_keys(Keys.TAB).perform()
        assert driver.find_element_by_id('result').text == 'You entered: TAB'

if __name__ == "__main__":
    unittest.main()
```

Expected Behavior

When we save this file and run it (e.g. `python keyboard_keys.py` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find the element and send the space key to it
- Find the result text on the page and check that it's what we expect
- Send the tab key to the element that's currently in focus
- Find the result text on the page and check that it's what we expect
- Close the browser

Outro

If you have a specific element that you want to issue key presses to, then finding the element first is the way to go. But if you don't have a receiving element, or you need to string together multiple key presses, then the Action Builder is what you should reach for.

Happy Testing!

Chapter 18

How To Work with Multiple Windows

The Problem

Occasionally you'll run into a link or action in the application you're testing that will open a new window. In order to work with both the new and originating windows you'll need to switch between them.

On the face of it, this is a pretty straightforward concept. But lurking within it is a small gotcha to watch out for that will bite you in some browsers and not others.

Let's step through a couple of examples to demonstrate.

An Example

First we'll need to pull in our requisite libraries (`import unittest` for our test framework and `from selenium import webdriver` to drive the browser), declare our test class, and wire up some test `setUp` and `tearDown` methods.

```
# filename: new_window.py
import unittest
from selenium import webdriver


class Windows(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
# ...
```

Now let's write a test that exercises new window functionality from an application. In this case, we'll be using the new window example found on the-internet.

```
# filename: new_window.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/windows')
        driver.find_element_by_css_selector('.example a').click()
        driver.switch_to_window(driver.window_handles[0])
        assert driver.title != "New Window", "title should not be New Window"
        driver.switch_to_window(driver.window_handles[-1])
        assert driver.title == "New Window", "title should be New Window"

# ...
```

After loading the page we click the link which spawns a new window. We then grab the window handles (a.k.a. unique identifier strings which represent each open browser window) and switch between them based on their order (assuming that the first one is the originating window, and that the last one is the new window). We round this test out by performing a simple check against the title of the page to make sure Selenium is focused on the correct window.

While this may seem like a good approach, it can present problems later. That's because the order of the window handles is not consistent across all browsers. Some return in the order opened, others alphabetically.

Here's a more resilient approach. One that will work across all browsers.

A Better Example

```
# filename: new_window.py
# ...
    def test_example_2(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/windows')

        first_window = driver.window_handles[0]
        driver.find_element_by_css_selector('.example a').click()
        all_windows = driver.window_handles
        for window in all_windows:
            if window != first_window:
                new_window = window
        driver.switch_to_window(first_window)
        assert driver.title != "New Window", "title should not be New Window"
        driver.switch_to_window(new_window)
        assert driver.title == "New Window", "title should be New Window"

if __name__ == "__main__":
    unittest.main()
```

After loading the page we store the window handle in a variable (e.g., `first_window`) and then proceed with clicking the new window link.

Now that we have two windows open we grab all of the window handles and search through them to find the new window handle (e.g., the handle that doesn't match the first one we've already stored). We store the result in another variable (e.g., `new_window`) and then switch between the windows. Each time checking the page title to make sure the correct window is in focus.

Expected Behavior

- Open the browser
- Visit the page
- Find the window handle for the current window
- Click a link that opens a new window
- Find the window handle out of all available window handles
- Switch between the windows
- Assert that the correct window is in focus
- Close the browser

Outro

Hat tip to Jim Evans for providing the info for this tip.

Happy Testing!

Chapter 19

How To Right-click

The Problem

Sometimes you'll run into an app that has functionality hidden behind a right-click menu (a.k.a. a context menu). These menus tend to be system level menus that are untouchable by Selenium. So how do you test this functionality?

A Solution

By leveraging Selenium's Action Builder (a.k.a. ActionChains in the Selenium Python bindings) we can issue a right-click command (a.k.a. a `context_click`).

We can then select an option from the menu by traversing it with keyboard arrow keys (which we can issue with the Action Builder's `send_keys` command).

NOTE: For a full write-up on working with keyboard keys in Selenium, see Chapter 17.

Let's dig in with an example.

An Example

Let's start by pulling in our requisite libraries, declare the test class, and wire up some simple `setUp` and `tearDown` methods.

```
# filename: right_click.py
import unittest
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys


class RightClick(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()

    def tearDown(self):
        self.driver.quit()
# ...
```

Now we're ready to write our test.

We'll use an example from the-internet that will render a custom context menu when we right-click on a specific area of the page (link). Clicking the context menu will trigger a JavaScript alert which will say `You selected a context menu`. We'll grab this text and use it to assert that the menu was actually triggered.

```
# filename: right_click.py
# ...
    def test_example_1(self):
        driver = self.driver
        driver.get('http://the-internet.herokuapp.com/context_menu')
        menu_area = driver.find_element_by_id('hot-spot')
        ActionChains(driver).context_click(
            menu_area).send_keys(
            Keys.ARROW_DOWN).send_keys(
            Keys.ARROW_DOWN).send_keys(
            Keys.ARROW_DOWN).send_keys(
            Keys.ENTER).perform()
        alert = driver.switch_to.alert
        assert alert.text == 'You selected a context menu'

if __name__ == "__main__":
    unittest.main()
```

Expected Behavior

When we save this file and run it (e.g., `python right_click.py`) from the command-line) here is what will happen:

- Open the browser and visit the page
- Find and right-click the area which will render a custom context menu
- Select the context menu option with keyboard keys
- JavaScript alert appears
- Grab the text of the JavaScript alert
- Assert that the text from the alert is what we expect
- Close the browser

Outro

To learn more about context menus, you can read this write-up from the Tree House blog. And for more thorough examples on working with keyboard keys and JavaScript alerts in your Selenium tests, check out Chapters 16 and 17.

Happy Testing!