Selenium Python Bindings

Release 2

Baiju Muthukadan

September 20, 2014

Contents

1 Installation **3**
- 1.1 Introduction . . . 3
- 1.2 Downloading Python bindings for Selenium . . . 3
- 1.3 Detailed instructions for Windows users . . . 3
- 1.4 Downloading Selenium server . . . 4

2 Getting Started **5**
- 2.1 Simple Usage . . . 5
- 2.2 Walk through of the example . . . 5
- 2.3 Using Selenium to write tests . . . 6
- 2.4 Walk through of the example . . . 7
- 2.5 Using Selenium with remote WebDriver . . . 8

3 Navigating **11**
- 3.1 Interacting with the page . . . 11
- 3.2 Filling in forms . . . 12
- 3.3 Drag and drop . . . 13
- 3.4 Moving between windows and frames . . . 13
- 3.5 Popup dialogs . . . 13
- 3.6 Navigation: history and location . . . 14
- 3.7 Cookies . . . 14

4 Locating Elements **15**
- 4.1 Locating by Id . . . 16
- 4.2 Locating by Name . . . 16
- 4.3 Locating by XPath . . . 17
- 4.4 Locating Hyperlinks by Link Text . . . 18
- 4.5 Locating Elements by Tag Name . . . 18
- 4.6 Locating Elements by Class Name . . . 19
- 4.7 Locating Elements by CSS Selectors . . . 19

5 Waits **21**
- 5.1 Explicit Waits . . . 21
- 5.2 Implicit Waits . . . 22

6 Page Objects **23**

7 WebDriver API **25**
- 7.1 Exceptions . . . 26

7.2 Action Chains . . . 29
7.3 Alerts . . . 32
7.4 Special Keys . . . 32
7.5 Locate elements By . . . 34
7.6 Desired Capabilities . . . 35
7.7 Utilities . . . 36
7.8 Firefox WebDriver . . . 36
7.9 Chrome WebDriver . . . 36
7.10 Remote WebDriver . . . 37
7.11 WebElement . . . 44
7.12 UI Support . . . 47
7.13 Color Support . . . 49
7.14 Expected conditions Support . . . 49

8 Appendix: Frequently Asked Questions **53**
8.1 How to use ChromeDriver ? . . . 53
8.2 Does Selenium 2 support XPath 2.0 ? . . . 53
8.3 How to scroll down to the bottom of a page ? . . . 53
8.4 How to auto save files using custom Firefox profile ? . . . 54
8.5 How to upload files into file inputs ? . . . 54
8.6 How to use firebug with Firefox ? . . . 54
8.7 How to take screenshot of the current window ? . . . 55

9 Indices and tables **57**

Python Module Index **59**

Author Baiju Muthukadan

Note: This is not an official documentation. Official API documentation is available here.

CHAPTER 1

Installation

1.1 Introduction

Selenium Python bindings provides a simple API to write functional/acceptance tests using Selenium WebDriver. Through Selenium Python API you can access all functionalities of Selenium WebDriver in an intuitive way.

Selenium Python bindings provide a convenient API to access Selenium WebDrivers like Firefox, Ie, Chrome, Remote etc. The current supported Python versions are 2.7, 3.2, 3.3 and 3.4.

This documentation explains Selenium 2 WebDriver API. Selenium 1 / Selenium RC API is not covered here.

1.2 Downloading Python bindings for Selenium

You can download Python bindings for Selenium from the PyPI page for selenium package. However, a better approach would be to use pip to install the selenium package. Python 3.4 has pip available in the standard library. Using *pip*, you can install selenium like this:

```
pip install selenium
```

You may consider using virtualenv to create isolated Python environments. Python 3.4 has pyvenv which is almost same as virtualenv.

1.3 Detailed instructions for Windows users

Note: You should have internet connection to perform this installation.

1. Install Python 3.4 using the MSI available in python.org download page.
2. Start a command prompt using the `cmd.exe` program and run the `pip` command as given below to install *selenium*.

   ```
   C:\Python34\Scripts\pip.exe install selenium
   ```

Now you can run your test scripts using Python. For example, if you have created a Selenium based script and saved it inside `C:\my_selenium_script.py`, you can run it like this:

```
C:\Python34\python.exe C:\my_selenium_script.py
```

1.4 Downloading Selenium server

Note: The Selenium server is only required, if you want to use the remote WebDriver. See the *Using Selenium with remote WebDriver* section for more details. If you are a beginner learning Selenium, you can skip this section and proceed with next chapter.

Selenium server is a Java program. Java Runtime Environment (JRE) 1.6 or newer version is recommended to run Selenium server.

You can download Selenium server 2.x from the download page of selenium website. The file name should be something like this: `selenium-server-standalone-2.x.x.jar`. You can always download the latest 2.x version of Selenium server.

If Java Runtime Environment (JRE) is not installed in your system, you can download the JRE from the Oracle website. If you are using a GNU/Linux system and have root access in your system, you can also use your operating system instructions to install JRE.

If *java* command is available in the PATH (environment variable), you can start the Selenium server using this command:

```
java -jar selenium-server-standalone-2.x.x.jar
```

Replace *2.x.x* with actual version of Selenium server you downloaded from the site.

If JRE is installed as a non-root user and/or if it is not available in the PATH (environment variable), you can type the relative or absolute path to the *java* command. Similarly, you can provide relative or absolute path to Selenium server jar file. Then, the command will look something like this:

```
/path/to/java -jar /path/to/selenium-server-standalone-2.x.x.jar
```

CHAPTER 2

Getting Started

2.1 Simple Usage

If you have installed Selenium Python bindings, you can start using it from Python like this.

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

driver = webdriver.Firefox()
driver.get("http://www.python.org")
assert "Python" in driver.title
elem = driver.find_element_by_name("q")
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
assert "No results found." not in driver.page_source
driver.close()
```

The above script can be saved into a file (eg:- *python_org_search.py*), then it can be run like this:

```
python python_org_search.py
```

The *python* which you are running should have the *selenium* module installed.

2.2 Walk through of the example

The *selenium.webdriver* module provides all the WebDriver implementations. Currently supported WebDriver implementations are Firefox, Chrome, Ie and Remote. The *Keys* class provide keys in the keyboard like RETURN, F1, ALT etc.

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

Next, the instance of Firefox WebDriver is created.

```
driver = webdriver.Firefox()
```

The *driver.get* method will navigate to a page given by the URL. WebDriver will wait until the page has fully loaded (that is, the "onload" event has fired) before returning control to your test or script. It's worth noting that if your page uses a lot of AJAX on load then WebDriver may not know when it has completely loaded.:

```
driver.get("http://www.python.org")
```

The next line is an assertion to confirm that title has "Python" word in it:

```
assert "Python" in driver.title
```

WebDriver offers a number of ways to find elements using one of the *find_element_by_** methods. For example, the input text element can be located by its *name* attribute using *find_element_by_name* method. Detailed explanation of finding elements is available in the *Locating Elements* chapter:

```
elem = driver.find_element_by_name("q")
```

Next we are sending keys, this is similar to entering keys using your keyboard. Special keys can be send using *Keys* class imported from *selenium.webdriver.common.keys*:

```
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
```

After submission of the page, you should get the result if there is any. To ensure that some results are found, make an assertion:

```
assert "No results found." not in driver.page_source
```

Finally, the browser window is closed. You can also call *quit* method instead of *close*. The *quit* will exit entire browser where as *close* will close one tab, but if it just one tab, by default most browser will exit entirely.:

```
driver.close()
```

2.3 Using Selenium to write tests

Selenium is mostly used for writing test cases. The *selenium* package itself doesn't provide a testing tool/framework. You can write test cases using Python's unittest module. The other choices as a tool/framework are py.test and nose.

In this chapter, we use *unittest* as the framework of choice. Here is the modified example which uses unittest module. This is a test for *python.org* search functionality:

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class PythonOrgSearch(unittest.TestCase):

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

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("http://www.python.org")
        self.assertIn("Python", driver.title)
        elem = driver.find_element_by_name("q")
        elem.send_keys("pycon")
        assert "No results found." not in driver.page_source
        elem.send_keys(Keys.RETURN)

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

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

You can run the above test case from a shell like this:

```
python test_python_org_search.py
.
----------------------------------------------------------------------
Ran 1 test in 15.566s

OK
```

The above results shows that, the test has been successfully completed.

2.4 Walk through of the example

Initially, all the basic modules required are imported. The unittest module is a built-in Python based on Java's JUnit. This module provides the framework for organizing the test cases. The *selenium.webdriver* module provides all the WebDriver implementations. Currently supported WebDriver implementations are Firefox, Chrome, Ie and Remote. The *Keys* class provide keys in the keyboard like RETURN, F1, ALT etc.

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
```

The test case class is inherited from *unittest.TestCase*. Inheriting from *TestCase* class is the way to tell *unittest* module that, this is a test case:

```
class PythonOrgSearch(unittest.TestCase):
```

The *setUp* is part of initialization, this method will get called before every test function which you are going to write in this test case class. Here you are creating the instance of Firefox WebDriver.

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

This is the test case method. The test case method should always start with characters *test*. The first line inside this method create a local reference to the driver object created in *setUp* method.

```
def test_search_in_python_org(self):
    driver = self.driver
```

The *driver.get* method will navigate to a page given by the URL. WebDriver will wait until the page has fully loaded (that is, the "onload" event has fired) before returning control to your test or script. It's worth noting that if your page uses a lot of AJAX on load then WebDriver may not know when it has completely loaded.:

```
driver.get("http://www.python.org")
```

The next line is an assertion to confirm that title has "Python" word in it:

```
self.assertIn("Python", driver.title)
```

WebDriver offers a number of ways to find elements using one of the *find_element_by_** methods. For example, the input text element can be located by its *name* attribute using *find_element_by_name* method. Detailed explanation of finding elements is available in the *Locating Elements* chapter:

```
elem = driver.find_element_by_name("q")
```

Next we are sending keys, this is similar to entering keys using your keyboard. Special keys can be send using *Keys* class imported from *selenium.webdriver.common.keys*:

```
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
```

After submission of the page, you should get result as per search if there is any. To ensure that some results are found, make an assertion:

```
assert "No results found." not in driver.page_source
```

The *tearDown* method will get called after every test method. This is a place to do all cleanup actions. In the current method, the browser window is closed. You can also call *quit* method instead of *close*. The *quit* will exit entire browser, where as *close* will close a tab, but if it is just one tab, by default most browser will exit entirely.:

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

Final lines are some boiler plate code to run the test suite:

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

2.5 Using Selenium with remote WebDriver

To use the remote WebDriver, you should have Selenium server running. To run the server, use this command:

```
java -jar selenium-server-standalone-2.x.x.jar
```

While running the Selenium server, you could see a message looks like this:

```
15:43:07.541 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
```

The above line says that, you can use this URL for connecting to remote WebDriver. Here are some examples:

```
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   desired_capabilities=DesiredCapabilities.CHROME)

driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   desired_capabilities=DesiredCapabilities.OPERA)

driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   desired_capabilities=DesiredCapabilities.HTMLUNITWITHJS)
```

The desired capabilities is a dictionary, so instead of using the default dictionaries, you can specifies the values explicitly:

```
driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   desired_capabilities={'browserName': 'htmlunit',
```

```
 'version': '2',
'javascriptEnabled': True})
```

CHAPTER 3

Navigating

The first thing you'll want to do with WebDriver is navigate to a link. The normal way to do this is by calling `get` method:

```
driver.get("http://www.google.com")
```

WebDriver will wait until the page has fully loaded (that is, the `onload` event has fired) before returning control to your test or script. It's worth noting that if your page uses a lot of AJAX on load then WebDriver may not know when it has completely loaded. If you need to ensure such pages are fully loaded then you can use *waits*.

3.1 Interacting with the page

Just being able to go to places isn't terribly useful. What we'd really like to do is to interact with the pages, or, more specifically, the HTML elements within a page. First of all, we need to find one. WebDriver offers a number of ways to find elements. For example, given an element defined as:

```
<input type="text" name="passwd" id="passwd-id" />
```

you could find it using any of:

```
element = driver.find_element_by_id("passwd-id")
element = driver.find_element_by_name("passwd")
element = driver.find_element_by_xpath("//input[@id='passwd-id']")
```

You can also look for a link by its text, but be careful! The text must be an exact match! You should also be careful when using *XPATH in WebDriver*. If there's more than one element that matches the query, then only the first will be returned. If nothing can be found, a `NoSuchElementException` will be raised.

WebDriver has an "Object-based" API; we represent all types of elements using the same interface. This means that although you may see a lot of possible methods you could invoke when you hit your IDE's auto-complete key combination, not all of them will make sense or be valid. Don't worry! WebDriver will attempt to do the Right Thing, and if you call a method that makes no sense ("setSelected()" on a "meta" tag, for example) an exception will be raised.

So, you've got an element. What can you do with it? First of all, you may want to enter some text into a text field:

```
element.send_keys("some text")
```

You can simulate pressing the arrow keys by using the "Keys" class:

```
element.send_keys(" and some", Keys.ARROW_DOWN)
```

It is possible to call *send_keys* on any element, which makes it possible to test keyboard shortcuts such as those used on GMail. A side-effect of this is that typing something into a text field won't automatically clear it. Instead, what you type will be appended to what's already there. You can easily clear the contents of a text field or textarea with *clear* method:

```
element.clear()
```

3.2 Filling in forms

We've already seen how to enter text into a textarea or text field, but what about the other elements? You can "toggle" the state of drop down, and you can use "setSelected" to set something like an *OPTION* tag selected. Dealing with *SELECT* tags isn't too bad:

```
element = driver.find_element_by_xpath("//select[@name='name']")
all_options = element.find_elements_by_tag_name("option")
for option in all_options:
    print("Value is: %s" % option.get_attribute("value"))
    option.click()
```

This will find the first "SELECT" element on the page, and cycle through each of it's OPTIONs in turn, printing out their values, and selecting each in turn.

As you can see, this isn't the most efficient way of dealing with SELECT elements . WebDriver's support classes include one called "Select", which provides useful methods for interacting with these:

```
from selenium.webdriver.support.ui import Select
select = Select(driver.find_element_by_name('name'))
select.select_by_index(index)
select.select_by_visible_text("text")
select.select_by_value(value)
```

WebDriver also provides features for deselecting all the selected options:

```
select = Select(driver.find_element_by_id('id'))
select.deselect_all()
```

This will deselect all OPTIONs from the first SELECT on the page.

Suppose in a test, we need the list of all default selected options, Select class provides a property method that returns a list:

```
select = Select(driver.find_element_by_xpath("xpath"))
all_selected_options = select.all_selected_options
```

To get all available options:

```
options = select.options
```

Once you've finished filling out the form, you probably want to submit it. One way to do this would be to find the "submit" button and click it:

```
# Assume the button has the ID "submit" :)
driver.find_element_by_id("submit").click()
```

Alternatively, WebDriver has the convenience method "submit" on every element. If you call this on an element within a form, WebDriver will walk up the DOM until it finds the enclosing form and then calls submit on that. If the element isn't in a form, then the `NoSuchElementException` will be raised:

```
element.submit()
```

3.3 Drag and drop

You can use drag and drop, either moving an element by a certain amount, or on to another element:

```
element = driver.find_element_by_name("source")
target = driver.find_element_by_name("target")

from selenium.webdriver import ActionChains
action_chains = ActionChains(driver)
action_chains.drag_and_drop(element, target)
```

3.4 Moving between windows and frames

It's rare for a modern web application not to have any frames or to be constrained to a single window. WebDriver supports moving between named windows using the "switch_to_window" method:

```
driver.switch_to_window("windowName")
```

All calls to `driver` will now be interpreted as being directed to the particular window. But how do you know the window's name? Take a look at the javascript or link that opened it:

```
<a href="somewhere.html" target="windowName">Click here to open a new window</a>
```

Alternatively, you can pass a "window handle" to the "switch_to_window()" method. Knowing this, it's possible to iterate over every open window like so:

```
for handle in driver.window_handles:
    driver.switch_to_window(handle)
```

You can also swing from frame to frame (or into iframes):

```
driver.switch_to_frame("frameName")
```

It's possible to access subframes by separating the path with a dot, and you can specify the frame by its index too. That is:

```
driver.switch_to_frame("frameName.0.child")
```

would go to the frame named "child" of the first subframe of the frame called "frameName". **All frames are evaluated as if from *top*.**

Once we are done with working on frames, we will have to come back to the parent frame which can be done using:

```
driver.switch_to_default_content()
```

3.5 Popup dialogs

Selenium WebDriver has built-in support for handling popup dialog boxes. After you've triggerd and action that would open a popup, you can access the alert with the following:

```
alert = driver.switch_to_alert()
```

This will return the currently open alert object. With this object you can now accept, dismiss, read its contents or even type into a prompt. This interface works equally well on alerts, confirms, prompts. Refer to the API documentation for more information.

3.6 Navigation: history and location

Earlier, we covered navigating to a page using the "get" command (`driver.get("http://www.example.com")`) As you've seen, WebDriver has a number of smaller, task-focused interfaces, and navigation is a useful task. To navigate to a page, you can use *get* method:

```
driver.get("http://www.example.com")
```

To move backwards and forwards in your browser's history:

```
driver.forward()
driver.back()
```

Please be aware that this functionality depends entirely on the underlying driver. It's just possible that something unexpected may happen when you call these methods if you're used to the behaviour of one browser over another.

3.7 Cookies

Before we leave these next steps, you may be interested in understanding how to use cookies. First of all, you need to be on the domain that the cookie will be valid for:

```
# Go to the correct domain
driver.get("http://www.example.com")

# Now set the cookie. This one's valid for the entire domain
cookie = {"key": "value"})
driver.add_cookie(cookie)

# And now output all the available cookies for the current URL
all_cookies = driver.get_cookies()
for cookie_name, cookie_value in all_cookies.items():
    print("%s -> %s", cookie_name, cookie_value)
```

CHAPTER 4

Locating Elements

There are vaious strategies to locate elements in a page. You can use the most appropriate one for your case. Selenium provides the following methods to locate elements in a page:

- *find_element_by_id*
- *find_element_by_name*
- *find_element_by_xpath*
- *find_element_by_link_text*
- *find_element_by_partial_link_text*
- *find_element_by_tag_name*
- *find_element_by_class_name*
- *find_element_by_css_selector*

To find multiple elements (these methods will return a list):

- *find_elements_by_name*
- *find_elements_by_xpath*
- *find_elements_by_link_text*
- *find_elements_by_partial_link_text*
- *find_elements_by_tag_name*
- *find_elements_by_class_name*
- *find_elements_by_css_selector*

Apart from the public methods given above, there are two private methods which might be useful with locators in page objects. These are the two private methods: *find_element* and *find_elements*.

Example usage:

```
from selenium.webdriver.common.by import By

driver.find_element(By.XPATH, '//button[text()="Some text"]')
driver.find_elements(By.XPATH, '//button')
```

These are the attributes available for *By* class:

```
ID = "id"
XPATH = "xpath"
LINK_TEXT = "link text"
PARTIAL_LINK_TEXT = "partial link text"
NAME = "name"
TAG_NAME = "tag name"
CLASS_NAME = "class name"
CSS_SELECTOR = "css selector"
```

4.1 Locating by Id

Use this when you know *id* attribute of an element. With this strategy, the first element with the *id* attribute value matching the location will be returned. If no element has a matching *id* attribute, a `NoSuchElementException` will be raised.

For instance, conside this page source:

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
   <input name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
  </form>
 </body>
<html>
```

The form element can be located like this:

```
login_form = driver.find_element_by_id('loginForm')
```

4.2 Locating by Name

Use this when you know *name* attribute of an element. With this strategy, the first element with the *name* attribute value matching the location will be returned. If no element has a matching *name* attribute, a `NoSuchElementException` will be raised.

For instance, conside this page source:

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
   <input name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
</body>
<html>
```

The username & password elements can be located like this:

```
username = driver.find_element_by_name('username')
password = driver.find_element_by_name('password')
```

This will give the "Login" button as it occur before the "Clear" button:

```
continue = driver.find_element_by_name('continue')
```

4.3 Locating by XPath

XPath is the language used for locating nodes in an XML document. As HTML can be an implementation of XML (XHTML), Selenium users can leverage this powerful language to target elements in their web applications. XPath extends beyond (as well as supporting) the simple methods of locating by id or name attributes, and opens up all sorts of new possibilities such as locating the third checkbox on the page.

One of the main reasons for using XPath is when you don't have a suitable id or name attribute for the element you wish to locate. You can use XPath to either locate the element in absolute terms (not advised), or relative to an element that does have an id or name attribute. XPath locators can also be used to specify elements via attributes other than id and name.

Absolute XPaths contain the location of all elements from the root (html) and as a result are likely to fail with only the slightest adjustment to the application. By finding a nearby element with an id or name attribute (ideally a parent element) you can locate your target element based on the relationship. This is much less likely to change and can make your tests more robust.

For instance, conside this page source:

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
   <input name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
</body>
<html>
```

The form elements can be located like this:

```
login_form = driver.find_element_by_xpath("/html/body/form[1]")
login_form = driver.find_element_by_xpath("//form[1]")
login_form = driver.find_element_by_xpath("//form[@id='loginForm']")
```

1. Absolute path (would break if the HTML was changed only slightly)
2. First form element in the HTML
3. The form element with attribute named *id* and the value *loginForm*

The username element can be located like this:

```
username = driver.find_element_by_xpath("//form[input/@name='username']")
username = driver.find_element_by_xpath("//form[@id='loginForm']/input[1]")
username = driver.find_element_by_xpath("//input[@name='username']")
```

1. First form element with an input child element with attribute named *name* and the value *username*
2. First input child element of the form element with attribute named *id* and the value *loginForm*
3. First input element with attribute named 'name' and the value *username*

The "Clear" button element can be located like this:

```
clear_button = driver.find_element_by_xpath("//input[@name='continue'][@type='button']")
clear_button = driver.find_element_by_xpath("//form[@id='loginForm']/input[4]")
```

1. Input with attribute named *name* and the value *continue* and attribute named *type* and the value *button*
2. Fourth input child element of the form element with attribute named *id* and value *loginForm*

These examples cover some basics, but in order to learn more, the following references are recommended:

- W3Schools XPath Tutorial
- W3C XPath Recommendation
- XPath Tutorial - with interactive examples.

There are also a couple of very useful Add-ons that can assist in discovering the XPath of an element:

- XPath Checker - suggests XPath and can be used to test XPath results.
- Firebug - XPath suggestions are just one of the many powerful features of this very useful add-on.
- XPath Helper - for Google Chrome

4.4 Locating Hyperlinks by Link Text

Use this when you know link text used within an anchor tag. With this strategy, the first element with the link text value matching the location will be returned. If no element has a matching link text attribute, a `NoSuchElementException` will be raised.

For instance, consider this page source:

```
<html>
 <body>
  <p>Are you sure you want to do this?</p>
  <a href="continue.html">Continue</a>
  <a href="cancel.html">Cancel</a>
</body>
<html>
```

The continue.html link can be located like this:

```
continue_link = driver.find_element_by_link_text('Continue')
continue_link = driver.find_element_by_partial_link_text('Conti')
```

4.5 Locating Elements by Tag Name

Use this when you want to locate an element by tag name. With this strategy, the first element with the give tag name will be returned. If no element has a matching tag name, a `NoSuchElementException` will be raised.

For instance, conside this page source:

```
<html>
 <body>
  <h1>Welcome</h1>
  <p>Site content goes here.</p>
</body>
<html>
```

The heading (h1) element can be located like this:

```
heading1 = driver.find_element_by_tag_name('h1')
```

4.6 Locating Elements by Class Name

Use this when you want to locate an element by class attribute name. With this strategy, the first element with the matching class attribute name will be returned. If no element has a matching class attribute name, a `NoSuchElementException` will be raised.

For instance, consider this page source:

```
<html>
 <body>
  <p class="content">Site content goes here.</p>
</body>
<html>
```

The "p" element can be located like this:

```
content = driver.find_element_by_class_name('content')
```

4.7 Locating Elements by CSS Selectors

Use this when you want to locate an element by CSS selector syntaxt. With this strategy, the first element with the matching CSS selector will be returned. If no element has a matching CSS selector, a `NoSuchElementException` will be raised.

For instance, consider this page source:

```
<html>
 <body>
  <p class="content">Site content goes here.</p>
</body>
<html>
```

The "p" element can be located like this:

```
content = driver.find_element_by_css_selector('p.content')
```

Sauce Labs has good documentation on CSS selectors.

CHAPTER 5

Waits

These days most of the web apps are using AJAX techniques. When a page is loaded to browser, the elements within that page may load at different time intervals. This makes locating elements difficult, if the element is not present in the DOM, it will raise *ElementNotVisibleException* exception. Using waits, we can solve this issue. Waiting provides some time interval between actions performed - mostly locating element or any other operation with the element.

Selenium Webdriver provides two types of waits - implicit & explicit. An explicit wait makes WebDriver to wait for a certain condition to occur before proceeding further with executions. An implicit wait makes WebDriver to poll the DOM for a certain amount of time when trying to locate an element.

5.1 Explicit Waits

An explicit waits is code you define to wait for a certain condition to occur before proceeding further in the code. The worst case of this is time.sleep(), which sets the condition to an exact time period to wait. There are some convenience methods provided that help you write code that will wait only as long as required. WebDriverWait in combination with ExpectedCondition is one way this can be accomplished.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Firefox()
driver.get("http://somedomain/url_that_delays_loading")
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "myDynamicElement"))
    )
finally:
    driver.quit()
```

This waits up to 10 seconds before throwing a TimeoutException or if it finds the element will return it in 0 - 10 seconds. WebDriverWait by default calls the ExpectedCondition every 500 milliseconds until it returns successfully. A successful return is for ExpectedCondition type is Boolean return true or not null return value for all other ExpectedCondition types.

Expected Conditions

There are some common conditions that are frequently come across when automating web browsers. Listed below are Implementations of each. Selenium Python binding provides some convienence methods so you don't have to code an expected_condition class yourself or create your own utility package for them.

- title_is
- title_contains
- presence_of_element_located
- visibility_of_element_located
- visibility_of
- presence_of_all_elements_located
- text_to_be_present_in_element
- text_to_be_present_in_element_value
- frame_to_be_available_and_switch_to_it
- invisibility_of_element_located
- element_to_be_clickable - it is Displayed and Enabled.
- staleness_of
- element_to_be_selected
- element_located_to_be_selected
- element_selection_state_to_be
- element_located_selection_state_to_be
- alert_is_present

```
from selenium.webdriver.support import expected_conditions as EC

wait = WebDriverWait(driver, 10)
element = wait.until(EC.element_to_be_clickable((By.ID,'someid')))
```

The expected_conditions module contains a set of predefined conditions to use with WebDriverWait.

5.2 Implicit Waits

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0. Once set, the implicit wait is set for the life of the WebDriver object instance.

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.implicitly_wait(10) # seconds
driver.get("http://somedomain/url_that_delays_loading")
myDynamicElement = driver.find_element_by_id("myDynamicElement")
```

CHAPTER 6

Page Objects

Note: This chaper is a work in progress, right now I don't have time to finish it. If anyone interested, please send pull request in Github. Here is an example implementation of the page objects pattern: https://github.com/baijum/pitracker/tree/master/test/acceptance

This chapter is a tutorial introduction to use page objects design pattern. Here is a test case which searches for a word in python.org website and ensure some results are found.

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import page

class PythonOrgSearch(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        driver.get("http://www.python.org")

    def test_search_in_python_org(self):
        main_page = page.MainPage(self.driver)
        assert main_page.is_title_matches(), "python.org title doesn't match."
        main_page.search_text_element = "pycon"
        main_page.click_go_button()
        search_results_page = page.SearchResultsPage(self.driver)
        assert search_results_page.is_results_found(), "No results found."

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

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

The `page.py` will look like this:

```
from element import BasePageElement
from locators import MainPageLocators

class SearchTextElement(BasePageElement):

    locator = 'q'
```

```
class BasePage(object):

    def __init__(self, driver):
        self.driver = driver


class MainPage(BasePage):

    search_text_element = SearchTextElement()

    def is_title_matches(self):
        pass

    def click_go_button(self):
        element = self.driver.find_element(*MainPageLocators.GO_BUTTON)
        element.click()


class SearchResultsPage(BasePage):

    def is_results_found(self):
        pass
```

The `element.py` will look like this:

```
from selenium.webdriver.support.ui import WebDriverWait


class BasePageElement(object):

    def __set__(self, obj, value):
        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_id(self.locator))
        driver.find_element_by_id(self.locator).send_keys(value)

    def __get__(self, obj, owner):
        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_id(self.locator))
        element = driver.find_element_by_id(self.locator)
        return element.get_attribute("value")
```

The `locators.py` will look like this:

```
from selenium.webdriver.common.by import By

class MainPageLocators(object):
    GO_BUTTON = (By.ID, 'submit')

class SearchResultsPageLocators(object):
    pass
```

CHAPTER 7

WebDriver API

Note: This is not an official documentation. Official API documentation is available here.

This chapter cover all the interfaces of Selenium WebDriver.

Recommended Import Style

The API definitions in this chapter shows the absolute location of classes. However the recommended import style is as given below:

```
from selenium import webdriver
```

Then, you can access the classes like this:

```
webdriver.Firefox
webdriver.FirefoxProfile
webdriver.Chrome
webdriver.ChromeOptions
webdriver.Ie
webdriver.Opera
webdriver.PhantomJS
webdriver.Remote
webdriver.DesiredCapabilities
webdriver.ActionChains
webdriver.TouchActions
webdriver.Proxy
```

The special keys class (`Keys`) can be imported like this:

```
from selenium.webdriver.common.keys import Keys
```

The exception classes can be imported like this (Replace the `TheNameOfTheExceptionClass` with actual class name given below):

```
from selenium.common.exceptions import [TheNameOfTheExceptionClass]
```

Conventions used in the API

Some attributes are callable (or methods) and others are non-callable (properties). All the callable attributes are ending with round brackets.

Here is an example for property:

- current_url

  URL of the current loaded page.

Usage:

```
driver.current_url
```

Here is an example for a method:

- close()

  Closes the current window.

  Usage:

  ```
  driver.close()
  ```

7.1 Exceptions

Exceptions that may happen in all the webdriver code.

exception selenium.common.exceptions.**ElementNotSelectableException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.InvalidElementStateException

Thrown when trying to select an unselectable element.

For example, selecting a 'script' element.

exception selenium.common.exceptions.**ElementNotVisibleException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.InvalidElementStateException

Thrown when an element is present on the DOM, but it is not visible, and so is not able to be interacted with.

Most commonly encountered when trying to click or read text of an element that is hidden from view.

exception selenium.common.exceptions.**ErrorInResponseException**(*response*, *msg*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when an error has occurred on the server side.

This may happen when communicating with the firefox extension or the remote driver server.

exception selenium.common.exceptions.**ImeActivationFailedException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when activating an IME engine has failed.

exception selenium.common.exceptions.**ImeNotAvailableException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when IME support is not available. This exception is thrown for every IME-related method call if IME support is not available on the machine.

exception selenium.common.exceptions.**InvalidCookieDomainException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when attempting to add a cookie under a different domain than the current URL.

exception selenium.common.exceptions.**InvalidElementStateException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

exception selenium.common.exceptions.**InvalidSelectorException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.NoSuchElementException

Thrown when the selector which is used to find an element does not return a WebElement. Currently this only happens when the selector is an xpath expression and it is either syntactically invalid (i.e. it is not a xpath expression) or the expression does not select WebElements (e.g. "count(//input)").

exception selenium.common.exceptions.**InvalidSwitchToTargetException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when frame or window target to be switched doesn't exist.

exception selenium.common.exceptions.**MoveTargetOutOfBoundsException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when the target provided to the *ActionsChains* move() method is invalid, i.e. out of document.

exception selenium.common.exceptions.**NoAlertPresentException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when switching to no presented alert.

This can be caused by calling an operation on the Alert() class when an alert is not yet on the screen.

exception selenium.common.exceptions.**NoSuchAttributeException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when the attribute of element could not be found.

You may want to check if the attribute exists in the particular browser you are testing against. Some browsers may have different property names for the same property. (IE8's .innerText vs. Firefox .textContent)

exception selenium.common.exceptions.**NoSuchElementException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when element could not be found.

If you encounter this exception, you may want to check the following:

- Check your selector used in your find_by...
- Element may not yet be on the screen at the time of the find operation,

(webpage is still loading) see selenium.webdriver.support.wait.WebDriverWait() for how to write a wait wrapper to wait for an element to appear.

exception selenium.common.exceptions.**NoSuchFrameException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.InvalidSwitchToTargetException

Thrown when frame target to be switched doesn't exist.

exception selenium.common.exceptions.**NoSuchWindowException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.InvalidSwitchToTargetException

Thrown when window target to be switched doesn't exist.

To find the current set of active window handles, you can get a list of the active window handles in the following way:

```
print driver.window_handles
```

exception selenium.common.exceptions.**RemoteDriverServerException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

exception selenium.common.exceptions.**StaleElementReferenceException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when a reference to an element is now "stale".

Stale means the element no longer appears on the DOM of the page.

Possible causes of StaleElementReferenceException include, but not limited to:

- You are no longer on the same page, or the page may have refreshed since the element

was located. * The element may have been removed and re-added to the screen, since it was located. Such as an element being relocated. This can happen typically with a javascript framework when values are updated and the node is rebuilt. * Element may have been inside an iframe or another context which was refreshed.

exception selenium.common.exceptions.**TimeoutException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when a command does not complete in enough time.

exception selenium.common.exceptions.**UnableToSetCookieException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when a driver fails to set a cookie.

exception selenium.common.exceptions.**UnexpectedAlertPresentException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when an unexpected alert is appeared.

Usually raised when when an expected modal is blocking webdriver form executing any more commands.

exception selenium.common.exceptions.**UnexpectedTagNameException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: selenium.common.exceptions.WebDriverException

Thrown when a support class did not get an expected web element.

exception selenium.common.exceptions.**WebDriverException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: exceptions.Exception

Base webdriver exception.

7.2 Action Chains

The ActionChains implementation,

class selenium.webdriver.common.action_chains.**ActionChains**(*driver*)

Bases: object

ActionChains are a way to automate low level interactions such as mouse movements, mouse button actions, key press, and context menu interactions. This is useful for doing more complex actions like hover over and drag and drop.

Generate user actions. When you call methods for actions on the ActionChains object, the actions are stored in a queue in the ActionChains object. When you call perform(), the events are fired in the order they are queued up.

ActionChains can be used in a chain pattern:

```
menu = driver.find_element_by_css_selector(".nav")
hidden_submenu = driver.find_element_by_css_selector(".nav #submenu1")

ActionChains(driver).move_to_element(menu).click(hidden_submenu).perform()
```

Or actions can be queued up one by one, then performed.:

```
menu = driver.find_element_by_css_selector(".nav")
hidden_submenu = driver.find_element_by_css_selector(".nav #submenu1")

actions = ActionChains(driver)
actions.move_to_element(menu)
actions.click(hidden_submenu)
actions.perform()
```

Either way, the actions are performed in the order they are called, one after another.

click(*on_element=None*)

Clicks an element.

Args

- on_element: The element to click. If None, clicks on current mouse position.

click_and_hold(*on_element=None*)
Holds down the left mouse button on an element.

Args

- on_element: The element to mouse down. If None, clicks on current mouse position.

context_click(*on_element=None*)
Performs a context-click (right click) on an element.

Args

- on_element: The element to context-click. If None, clicks on current mouse position.

double_click(*on_element=None*)
Double-clicks an element.

Args

- on_element: The element to double-click. If None, clicks on current mouse position.

drag_and_drop(*source*, *target*)

Holds down the left mouse button on the source element, then moves to the target element and releases the mouse button.

Args

- source: The element to mouse down.
- target: The element to mouse up.

drag_and_drop_by_offset(*source*, *xoffset*, *yoffset*)

Holds down the left mouse button on the source element, then moves to the target offset and releases the mouse button.

Args

- source: The element to mouse down.
- xoffset: X offset to move to.
- yoffset: Y offset to move to.

key_down(*value*, *element=None*)

Sends a key press only, without releasing it. Should only be used with modifier keys (Control, Alt and Shift).

Args

- value: The modifier key to send. Values are defined in *Keys* class.
- element: The element to send keys. If None, sends a key to current focused element.

Example, pressing ctrl+c:

```
ActionsChains(driver).key_down(Keys.CONTROL).send_keys('c').key_up(Keys.CONTROL).perform()
```

key_up(*value*, *element=None*)
Releases a modifier key.

Args

- value: The modifier key to send. Values are defined in Keys class.
- element: The element to send keys. If None, sends a key to current focused element.

Example, pressing ctrl+c:

```
ActionsChains(driver).key_down(Keys.CONTROL).send_keys('c').key_up(Keys.CONTROL).perform()
```

move_by_offset(*xoffset*, *yoffset*)
Moving the mouse to an offset from current mouse position.

Args

- xoffset: X offset to move to, as a positive or negative integer.
- yoffset: Y offset to move to, as a positive or negative integer.

move_to_element(*to_element*)
Moving the mouse to the middle of an element.

Args

- to_element: The WebElement to move to.

move_to_element_with_offset(*to_element*, *xoffset*, *yoffset*)

Move the mouse by an offset of the specified element. Offsets are relative to the top-left corner of the element.

Args

- to_element: The WebElement to move to.
- xoffset: X offset to move to.
- yoffset: Y offset to move to.

perform()
Performs all stored actions.

release(*on_element=None*)
Releasing a held mouse button on an element.

Args

- on_element: The element to mouse up. If None, releases on current mouse position.

send_keys(**keys_to_send*)
Sends keys to current focused element.

Args

- keys_to_send: The keys to send. Modifier keys constants can be found in the

'Keys' class.

send_keys_to_element(*element*, **keys_to_send*)
Sends keys to an element.

Args

- element: The element to send keys.
- keys_to_send: The keys to send. Modifier keys constants can be found in the

'Keys' class.

7.3 Alerts

The Alert implementation.

class selenium.webdriver.common.alert.**Alert**(*driver*)
Bases: object

Allows to work with alerts.

Use this class to interact with alert prompts. It contains methods for dismissing, accepting, inputting, and getting text from alert prompts.

Accepting / Dismissing alert prompts:

```
Alert(driver).accept()
Alert(driver).dismiss()
```

Inputting a value into an alert prompt:

name_prompt = Alert(driver) name_prompt.send_keys("Willian Shakesphere") name_prompt.accept()

Reading a the text of a prompt for verification:

alert_text = Alert(driver).text self.assertEqual("Do you wish to quit?", alert_text)

accept()
Accepts the alert available.

Usage:: Alert(driver).accept() # Confirm a alert dialog.

dismiss()
Dismisses the alert available.

send_keys(*keysToSend*)
Send Keys to the Alert.

Args

- keysToSend: The text to be sent to Alert.

text
Gets the text of the Alert.

7.4 Special Keys

The Keys implementation.

class selenium.webdriver.common.keys.**Keys**
Bases: object

Set of special keys codes.

ADD = u'\ue025'

ALT = u'\ue00a'

ARROW_DOWN = u'\ue015'

ARROW_LEFT = u'\ue012'

ARROW_RIGHT = u'\ue014'

ARROW_UP = u'\ue013'

BACKSPACE = u'\ue003'

BACK_SPACE = u'\ue003'

CANCEL = u'\ue001'

CLEAR = u'\ue005'

COMMAND = u'\ue03d'

CONTROL = u'\ue009'

DECIMAL = u'\ue028'

DELETE = u'\ue017'

DIVIDE = u'\ue029'

DOWN = u'\ue015'

END = u'\ue010'

ENTER = u'\ue007'

EQUALS = u'\ue019'

ESCAPE = u'\ue00c'

F1 = u'\ue031'

F10 = u'\ue03a'

F11 = u'\ue03b'

F12 = u'\ue03c'

F2 = u'\ue032'

F3 = u'\ue033'

F4 = u'\ue034'

F5 = u'\ue035'

F6 = u'\ue036'

F7 = u'\ue037'

F8 = u'\ue038'

F9 = u'\ue039'

HELP = u'\ue002'

HOME = u'\ue011'

INSERT = u'\ue016'

LEFT = u'\ue012'

LEFT_ALT = u'\ue00a'

LEFT_CONTROL = u'\ue009'

LEFT_SHIFT = u'\ue008'

META = u'\ue03d'

MULTIPLY = u'\ue024'

NULL = u'\ue000'

NUMPAD0 = u'\ue01a'

NUMPAD1 = u'\ue01b'

NUMPAD2 = u'\ue01c'

NUMPAD3 = u'\ue01d'

NUMPAD4 = u'\ue01e'

NUMPAD5 = u'\ue01f'

NUMPAD6 = u'\ue020'

NUMPAD7 = u'\ue021'

NUMPAD8 = u'\ue022'

NUMPAD9 = u'\ue023'

PAGE_DOWN = u'\ue00f'

PAGE_UP = u'\ue00e'

PAUSE = u'\ue00b'

RETURN = u'\ue006'

RIGHT = u'\ue014'

SEMICOLON = u'\ue018'

SEPARATOR = u'\ue026'

SHIFT = u'\ue008'

SPACE = u'\ue00d'

SUBTRACT = u'\ue027'

TAB = u'\ue004'

UP = u'\ue013'

7.5 Locate elements By

These are the attributes which can be used to locate elements. See the *Locating Elements* chapter for example usages.
The By implementation.

class `selenium.webdriver.common.by.`**By**

Bases: `object`

Set of supported locator strategies.

classmethod is_valid(*by*)

CLASS_NAME = 'class name'

CSS_SELECTOR = 'css selector'

ID = 'id'

LINK_TEXT = 'link text'

NAME = 'name'

PARTIAL_LINK_TEXT = 'partial link text'

TAG_NAME = 'tag name'

XPATH = 'xpath'

7.6 Desired Capabilities

See the *Using Selenium with remote WebDriver* section for example usages of desired capabilities. The Desired Capabilities implementation.

class selenium.webdriver.common.desired_capabilities.**DesiredCapabilities**
Bases: object

Set of default supported desired capabilities.

Use this as a starting point for creating a desired capabilities object for requesting remote webdrivers for connecting to selenium server or selenium grid.

Usage Example:

from selenium import webdriver

selenium_grid_url = "http://198.0.0.1:4444/wd/hub"

Create a desired capabilities object as a starting point. capabilities = DesiredCapabilities.FIREFOX.copy() capabilities['platform'] = "WINDOWS" capabilities['version'] = "10"

Instantiate an instance of Remote WebDriver with the desired capabilities. driver = webdriver.Remote(desired_capabilities=capabilities,

command_executor=selenium_grid_url)

Note: Always use '.copy()' on the DesiredCapabilities object to avoid the side effects of altering the Global class instance.

ANDROID = {'platform': 'ANDROID', 'browserName': 'android', 'version': '', 'javascriptEnabled': True}

CHROME = {'platform': 'ANY', 'browserName': 'chrome', 'version': '', 'javascriptEnabled': True}

FIREFOX = {'platform': 'ANY', 'browserName': 'firefox', 'version': '', 'javascriptEnabled': True}

HTMLUNIT = {'platform': 'ANY', 'browserName': 'htmlunit', 'version': ''}

HTMLUNITWITHJS = {'platform': 'ANY', 'browserName': 'htmlunit', 'version': 'firefox', 'javascriptEnabled': True}

INTERNETEXPLORER = {'platform': 'WINDOWS', 'browserName': 'internet explorer', 'version': '', 'javascriptEnabled

IPAD = {'platform': 'MAC', 'browserName': 'iPad', 'version': '', 'javascriptEnabled': True}

IPHONE = {'platform': 'MAC', 'browserName': 'iPhone', 'version': '', 'javascriptEnabled': True}

OPERA = {'platform': 'ANY', 'browserName': 'opera', 'version': '', 'javascriptEnabled': True}

PHANTOMJS = {'platform': 'ANY', 'browserName': 'phantomjs', 'version': '', 'javascriptEnabled': True}

SAFARI = {'platform': 'ANY', 'browserName': 'safari', 'version': '', 'javascriptEnabled': True}

7.7 Utilities

The Utils methods.

selenium.webdriver.common.utils.**free_port**()
Determines a free port using sockets.

selenium.webdriver.common.utils.**is_connectable**(*port*)
Tries to connect to the server at port to see if it is running.

Args

- port: The port to connect.

selenium.webdriver.common.utils.**is_url_connectable**(*port*)
Tries to connect to the HTTP server at /status path and specified port to see if it responds successfully.

Args

- port: The port to connect.

7.8 Firefox WebDriver

class selenium.webdriver.firefox.webdriver.**WebDriver**(*firefox_profile=None*, *firefox_binary=None*, *timeout=30*, *capabilities=None*, *proxy=None*)
Bases: selenium.webdriver.remote.webdriver.WebDriver

quit()
Quits the driver and close every associated window.

NATIVE_EVENTS_ALLOWED = True

firefox_profile

7.9 Chrome WebDriver

class selenium.webdriver.chrome.webdriver.**WebDriver**(*executable_path='chromedriver'*, *port=0*, *chrome_options=None*, *service_args=None*, *desired_capabilities=None*, *service_log_path=None*)
Bases: selenium.webdriver.remote.webdriver.WebDriver

Controls the ChromeDriver and allows you to drive the browser.

You will need to download the ChromeDriver executable from http://chromedriver.storage.googleapis.com/index.html

quit()
Closes the browser and shuts down the ChromeDriver executable that is started when starting the ChromeDriver

7.10 Remote WebDriver

The WebDriver implementation.

class selenium.webdriver.remote.webdriver.**WebDriver**(*command_executor='http://127.0.0.1:4444/wd/hub'*, *desired_capabilities=None*, *browser_profile=None*, *proxy=None*, *keep_alive=False*)

Bases: object

Controls a browser by sending commands to a remote server. This server is expected to be running the WebDriver wire protocol as defined here: http://code.google.com/p/selenium/wiki/JsonWireProtocol

Attributes

- command_executor - The command.CommandExecutor object used to execute commands.
- error_handler - errorhandler.ErrorHandler object used to verify that the server did not return an error.
- session_id - The session ID to send with every command.
- capabilities - A dictionary of capabilities of the underlying browser for this instance's session.
- proxy - A selenium.webdriver.common.proxy.Proxy object, to specify a proxy for the browser to use.

add_cookie(*cookie_dict*)

Adds a cookie to your current session.

Args

- **cookie_dict: A dictionary object, with required keys - "name" and "value";** optional keys - "path", "domain", "secure", "expiry"

Usage: driver.add_cookie({'name' : 'foo', 'value' : 'bar'}) driver.add_cookie({'name' : 'foo', 'value' : 'bar', 'path' : '/'}) driver.add_cookie({'name' : 'foo', 'value' : 'bar', 'path' : '/', 'secure':True})

back()

Goes one step backward in the browser history.

Usage driver.back()

close()

Closes the current window.

Usage driver.close()

create_web_element(*element_id*)

Creates a web element with the specified element_id.

delete_all_cookies()

Delete all cookies in the scope of the session.

Usage driver.delete_all_cookies()

delete_cookie(*name*)

Deletes a single cookie with the given name.

Usage driver.delete_cookie('my_cookie')

execute(*driver_command*, *params=None*)
Sends a command to be executed by a command.CommandExecutor.

Args

- driver_command: The name of the command to execute as a string.
- params: A dictionary of named parameters to send with the command.

Returns The command's JSON response loaded into a dictionary object.

execute_async_script(*script*, **args*)
Asynchronously Executes JavaScript in the current window/frame.

Args

- script: The JavaScript to execute.
- *args: Any applicable arguments for your JavaScript.

Usage driver.execute_async_script('document.title')

execute_script(*script*, **args*)
Synchronously Executes JavaScript in the current window/frame.

Args

- script: The JavaScript to execute.
- *args: Any applicable arguments for your JavaScript.

Usage driver.execute_script('document.title')

find_element(*by='id'*, *value=None*)
'Private' method used by the find_element_by_* methods.

Usage Use the corresponding find_element_by_* instead of this.

Return type WebElement

find_element_by_class_name(*name*)
Finds an element by class name.

Args

- name: The class name of the element to find.

Usage driver.find_element_by_class_name('foo')

find_element_by_css_selector(*css_selector*)
Finds an element by css selector.

Args

- css_selector: The css selector to use when finding elements.

Usage driver.find_element_by_css_selector('#foo')

find_element_by_id(*id_*)
Finds an element by id.

Args

- id_ - The id of the element to be found.

Usage driver.find_element_by_id('foo')

find_element_by_link_text(*link_text*)
Finds an element by link text.

Args

- link_text: The text of the element to be found.

Usage driver.find_element_by_link_text('Sign In')

find_element_by_name(*name*)
Finds an element by name.

Args

- name: The name of the element to find.

Usage driver.find_element_by_name('foo')

find_element_by_partial_link_text(*link_text*)
Finds an element by a partial match of its link text.

Args

- link_text: The text of the element to partially match on.

Usage driver.find_element_by_partial_link_text('Sign')

find_element_by_tag_name(*name*)
Finds an element by tag name.

Args

- name: The tag name of the element to find.

Usage driver.find_element_by_tag_name('foo')

find_element_by_xpath(*xpath*)
Finds an element by xpath.

Args

- xpath - The xpath locator of the element to find.

Usage driver.find_element_by_xpath('//div/td[1]')

find_elements(*by='id'*, *value=None*)
'Private' method used by the find_elements_by_* methods.

Usage Use the corresponding find_elements_by_* instead of this.

Return type list of WebElement

find_elements_by_class_name(*name*)
Finds elements by class name.

Args

- name: The class name of the elements to find.

Usage driver.find_elements_by_class_name('foo')

find_elements_by_css_selector(*css_selector*)
Finds elements by css selector.

Args

- css_selector: The css selector to use when finding elements.

Usage driver.find_elements_by_css_selector('.foo')

find_elements_by_id(*id_*)
Finds multiple elements by id.

Args

- id_ - The id of the elements to be found.

Usage driver.find_element_by_id('foo')

find_elements_by_link_text(*text*)
Finds elements by link text.

Args

- link_text: The text of the elements to be found.

Usage driver.find_elements_by_link_text('Sign In')

find_elements_by_name(*name*)
Finds elements by name.

Args

- name: The name of the elements to find.

Usage driver.find_elements_by_name('foo')

find_elements_by_partial_link_text(*link_text*)
Finds elements by a partial match of their link text.

Args

- link_text: The text of the element to partial match on.

Usage driver.find_element_by_partial_link_text('Sign')

find_elements_by_tag_name(*name*)
Finds elements by tag name.

Args

- name: The tag name the use when finding elements.

Usage driver.find_elements_by_tag_name('foo')

find_elements_by_xpath(*xpath*)
Finds multiple elements by xpath.

Args

- xpath - The xpath locator of the elements to be found.

Usage driver.find_elements_by_xpath("//div[contains(@class, 'foo')]")

forward()
Goes one step forward in the browser history.

Usage driver.forward()

get(*url*)
Loads a web page in the current browser session.

get_cookie(*name*)
Get a single cookie by name. Returns the cookie if found, None if not.

Usage driver.get_cookie('my_cookie')

get_cookies()
Returns a set of dictionaries, corresponding to cookies visible in the current session.

Usage driver.get_cookies()

get_log(*log_type*)
Gets the log for a given log type

Args

- log_type: type of log that which will be returned

Usage driver.get_log('browser') driver.get_log('driver') driver.get_log('client') driver.get_log('server')

get_screenshot_as_base64()

Gets the screenshot of the current window as a base64 encoded string which is useful in embedded images in HTML.

Usage driver.get_screenshot_as_base64()

get_screenshot_as_file(*filename*)

Gets the screenshot of the current window. Returns False if there is any IOError, else returns True. Use full paths in your filename.

Args

- filename: The full path you wish to save your screenshot to.

Usage driver.get_screenshot_as_file('/Screenshots/foo.png')

get_screenshot_as_png()
Gets the screenshot of the current window as a binary data.

Usage driver.get_screenshot_as_png()

get_window_position(*windowHandle='current'*)
Gets the x,y position of the current window.

Usage driver.get_window_position()

get_window_size(*windowHandle='current'*)
Gets the width and height of the current window.

Usage driver.get_window_size()

implicitly_wait(*time_to_wait*)

Sets a sticky timeout to implicitly wait for an element to be found, or a command to complete. This method only needs to be called one time per session. To set the timeout for calls to execute_async_script, see set_script_timeout.

Args

- time_to_wait: Amount of time to wait (in seconds)

Usage driver.implicitly_wait(30)

maximize_window()
Maximizes the current window that webdriver is using

quit()
Quits the driver and closes every associated window.

Usage driver.quit()

refresh()
Refreshes the current page.

Usage driver.refresh()

save_screenshot(*filename*)

Gets the screenshot of the current window. Returns False if there is any IOError, else returns True. Use full paths in your filename.

Args

- filename: The full path you wish to save your screenshot to.

Usage driver.get_screenshot_as_file('/Screenshots/foo.png')

set_page_load_timeout(*time_to_wait*)

Set the amount of time to wait for a page load to complete before throwing an error.

Args

- time_to_wait: The amount of time to wait

Usage driver.set_page_load_timeout(30)

set_script_timeout(*time_to_wait*)

Set the amount of time that the script should wait during an execute_async_script call before throwing an error.

Args

- time_to_wait: The amount of time to wait (in seconds)

Usage driver.set_script_timeout(30)

set_window_position(*x*, *y*, *windowHandle='current'*)
Sets the x,y position of the current window. (window.moveTo)

Args

- x: the x-coordinate in pixels to set the window position
- y: the y-coordinate in pixels to set the window position

Usage driver.set_window_position(0,0)

set_window_size(*width*, *height*, *windowHandle='current'*)
Sets the width and height of the current window. (window.resizeTo)

Args

- width: the width in pixels to set the window to
- height: the height in pixels to set the window to

Usage driver.set_window_size(800,600)

start_client()
Called before starting a new session. This method may be overridden to define custom startup behavior.

start_session(*desired_capabilities*, *browser_profile=None*)
Creates a new session with the desired capabilities.

Args

- browser_name - The name of the browser to request.
- version - Which browser version to request.
- platform - Which platform to request the browser on.
- javascript_enabled - Whether the new session should support JavaScript.
- browser_profile - A selenium.webdriver.firefox.firefox_profile.FirefoxProfile object. Only used if Firefox is requested.

stop_client()
Called after executing a quit command. This method may be overridden to define custom shutdown behavior.

switch_to_active_element()
Deprecated use driver.switch_to.active_element

switch_to_alert()
Deprecated use driver.switch_to.alert

switch_to_default_content()
Deprecated use driver.switch_to.default_content

switch_to_frame(*frame_reference*)
Deprecated use driver.switch_to.frame

switch_to_window(*window_name*)
Deprecated use driver.switch_to.window

application_cache
Returns a ApplicationCache Object to interact with the browser app cache

current_url
Gets the URL of the current page.

Usage driver.current_url

current_window_handle
Returns the handle of the current window.

Usage driver.current_window_handle

desired_capabilities
returns the drivers current desired capabilities being used

log_types
Gets a list of the available log types

Usage driver.log_types

mobile

name
Returns the name of the underlying browser for this instance.

Usage

- driver.name

orientation
Gets the current orientation of the device

Usage orientation = driver.orientation

page_source
Gets the source of the current page.

Usage driver.page_source

switch_to

title
Returns the title of the current page.

Usage driver.title

window_handles
Returns the handles of all windows within the current session.

Usage driver.window_handles

7.11 WebElement

WebElement implementation.

class selenium.webdriver.remote.webelement.**LocalFileDetector**
Bases: object

classmethod is_local_file(**keys*)

class selenium.webdriver.remote.webelement.**WebElement**(*parent*, *id_*)
Bases: object

Represents an HTML element.

Generally, all interesting operations to do with interacting with a page will be performed through this interface.

clear()
Clears the text if it's a text entry element.

click()
Clicks the element.

find_element(*by='id'*, *value=None*)

find_element_by_class_name(*name*)
Finds an element within this element's children by their class name.

Args

- name - class name to search on.

find_element_by_css_selector(*css_selector*)
Find and return an element that's a child of this element by CSS selector.

Args

- css_selector - CSS selctor string, ex: 'a.nav#home'

find_element_by_id(*id_*)
Finds element within the child elements of this element.

Args

- **id_** - ID of child element to locate.

find_element_by_link_text(*link_text*)
Finds element with in this element's children by visible link text.

Args

- link_text - Link text string to search for.

find_element_by_name(*name*)
Find element with in this element's children by name. :Args:

•name - name property of the element to find.

find_element_by_partial_link_text(*link_text*)
Finds element with in this element's children by parial visible link text.

Args

- link_text - Link text string to search for.

find_element_by_tag_name(*name*)
Finds element with in this element's children by tag name.

Args

- name - name of html tag (eg: h1, a, span)

find_element_by_xpath(*xpath*)
Finds element by xpath.

Args xpath - xpath of element to locate. "//input[@class='myelement']"

Note: The base path will be relative to this element's location.

This will select the first link under this element.:

```
myelement.find_elements_by_xpath(".//a")
```

However, this will select the first link on the page.

myelement.find_elements_by_xpath("//a")

find_elements(*by='id'*, *value=None*)

find_elements_by_class_name(*name*)
Finds a list of elements within children of this element by their class name.

Args

- name - class name to search on.

find_elements_by_css_selector(*css_selector*)
Find and return list of multiple elements within the children of this element by CSS selector.

Args

- css_selector - CSS selctor string, ex: 'a.nav#home'

find_elements_by_id(*id_*)
Finds a list of elements within the children of this element with the matching ID.

Args

- **id_** - Id of child element to find.

find_elements_by_link_text(*link_text*)
Finds a list of elements with in this element's children by visible link text.

Args

- link_text - Link text string to search for.

find_elements_by_name(*name*)
Finds a list of elements with in this element's children by name.

Args

- name - name property to search for.

find_elements_by_partial_link_text(*link_text*)
Finds a list of elements with in this element's children by link text.

Args

- link_text - Link text string to search for.

find_elements_by_tag_name(*name*)
Finds a list of elements with in this element's children by tag name.

Args

- name - name of html tag (eg: h1, a, span)

find_elements_by_xpath(*xpath*)
Finds elements within the elements by xpath.

Args

- xpath - xpath locator string.

Note: The base path will be relative to this element's location.

This will select all links under this element.:

```
myelement.find_elements_by_xpath(".//a")
```

However, this will select all links in the page itself.

myelement.find_elements_by_xpath("//a")

get_attribute(*name*)
Gets the attribute value.

Args

- name - name of the attribute property to retieve.

Example:

```
# Check if the 'active' css class is applied to an element.
is_active = "active" in target_element.get_attribute("class")
```

is_displayed()
Whether the element would be visible to a user

is_enabled()
Whether the element is enabled.

is_selected()
Whether the element is selected.

Can be used to check if a checkbox or radio button is selected.

send_keys(**value*)
Simulates typing into the element.

Args

- value - A string for typing, or setting form fields. For setting

file inputs, this could be a local file path.

Use this to send simple key events or to fill out form fields:

```
form_textfield = driver.find_element_by_name('username')
form_textfield.send_keys("admin")
```

This can also be used to set file inputs.:

```
file_input = driver.find_element_by_name('profilePic')
file_input.send_keys("path/to/profilepic.gif")
# Generally it's better to wrap the file path in one of the methods
# in os.path to return the actual path to support cross OS testing.
# file_input.send_keys(os.path.abspath("path/to/profilepic.gif"))
```

submit()
Submits a form.

value_of_css_property(*property_name*)
Returns the value of a CSS property

id
Returns internal id used by selenium.

This is mainly for internal use. Simple use cases such as checking if 2 webelements refer to the same element, can be done using '==':

```
if element1 == element2:
    print("These 2 are equal")
```

location
Returns the location of the element in the renderable canvas

location_once_scrolled_into_view
CONSIDERED LIABLE TO CHANGE WITHOUT WARNING. Use this to discover where on the screen an element is so that we can click it. This method should cause the element to be scrolled into view.

Returns the top lefthand corner location on the screen, or None if the element is not visible

parent
Returns parent element is available.

size
Returns the size of the element

tag_name
Gets this element's tagName property.

text
Gets the text of the element.

7.12 UI Support

class selenium.webdriver.support.select.**Select**(*webelement*)

deselect_all()
Clear all selected entries. This is only valid when the SELECT supports multiple selections. throws NotImplementedError If the SELECT does not support multiple selections

deselect_by_index(*index*)
Deselect the option at the given index. This is done by examing the "index" attribute of an element, and not merely by counting.

Args

- index - The option at this index will be deselected

deselect_by_value(*value*)
Deselect all options that have a value matching the argument. That is, when given "foo" this would deselect an option like:

<option value="foo">Bar</option>

Args

- value - The value to match against

deselect_by_visible_text(*text*)
Deselect all options that display text matching the argument. That is, when given "Bar" this would deselect an option like:

<option value="foo">Bar</option>

Args

- text - The visible text to match against

select_by_index(*index*)
Select the option at the given index. This is done by examing the "index" attribute of an element, and not merely by counting.

Args

- index - The option at this index will be selected

select_by_value(*value*)
Select all options that have a value matching the argument. That is, when given "foo" this would select an option like:

<option value="foo">Bar</option>

Args

- value - The value to match against

select_by_visible_text(*text*)
Select all options that display text matching the argument. That is, when given "Bar" this would select an option like:

<option value="foo">Bar</option>

Args

- text - The visible text to match against

all_selected_options
Returns a list of all selected options belonging to this select tag

first_selected_option
The first selected option in this select tag (or the currently selected option in a normal select)

options
Returns a list of all options belonging to this select tag

class selenium.webdriver.support.wait.**WebDriverWait**(*driver*, *timeout*, *poll_frequency=0.5*, *ignored_exceptions=None*)

Bases: object

until(*method*, *message=''*)
Calls the method provided with the driver as an argument until the return value is not False.

until_not(*method*, *message=''*)
Calls the method provided with the driver as an argument until the return value is False.

7.13 Color Support

class selenium.webdriver.support.color.**Color**(*red*, *green*, *blue*, *alpha=1*)
Bases: object

Color conversion support class

Example:

```
from selenium.webdriver.support.color import Color

print(Color.from_string('#00ff33').rgba)
print(Color.from_string('rgb(1, 255, 3)').hex)
print(Color.from_string('blue').rgba)
```

static from_string(*str_*)

hex

rgb

rgba

7.14 Expected conditions Support

class selenium.webdriver.support.expected_conditions.**alert_is_present**
Bases: object

Expect an alert to be present.

class selenium.webdriver.support.expected_conditions.**element_located_selection_state_to_be**(*loc*
*is_*

Bases: object

An expectation to locate an element and check if the selection state specified is in that state. locator is a tuple of (by, path) is_selected is a boolean

class selenium.webdriver.support.expected_conditions.**element_located_to_be_selected**(*locator*)
Bases: object

An expectation for the element to be located is selected. locator is a tuple of (by, path)

class selenium.webdriver.support.expected_conditions.**element_selection_state_to_be**(*element*, *is_selected*)

Bases: object

An expectation for checking if the given element is selected. element is WebElement object is_selected is a Boolean."

class selenium.webdriver.support.expected_conditions.**element_to_be_clickable**(*locator*)
Bases: object

An Expectation for checking an element is visible and enabled such that you can click it.

class selenium.webdriver.support.expected_conditions.**element_to_be_selected**(*element*)
Bases: object

An expectation for checking the selection is selected. element is WebElement object

class selenium.webdriver.support.expected_conditions.**frame_to_be_available_and_switch_to_it**(*l*
Bases: object

An expectation for checking whether the given frame is available to switch to. If the frame is available it switches the given driver to the specified frame.

class selenium.webdriver.support.expected_conditions.**invisibility_of_element_located**(*locator*)
Bases: object

An Expectation for checking that an element is either invisible or not present on the DOM.

locator used to find the element

class selenium.webdriver.support.expected_conditions.**presence_of_all_elements_located**(*locator*)
Bases: object

An expectation for checking that there is at least one element present on a web page. locator is used to find the element returns the list of WebElements once they are located

class selenium.webdriver.support.expected_conditions.**presence_of_element_located**(*locator*)
Bases: object

An expectation for checking that an element is present on the DOM of a page. This does not necessarily mean that the element is visible. locator - used to find the element returns the WebElement once it is located

class selenium.webdriver.support.expected_conditions.**staleness_of**(*element*)
Bases: object

Wait until an element is no longer attached to the DOM. element is the element to wait for. returns False if the element is still attached to the DOM, true otherwise.

class selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element**(*locator*, *text_*)
Bases: object

An expectation for checking if the given text is present in the specified element. locator, text

class selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element_value**(*locat*
*text_*)
Bases: object

An expectation for checking if the given text is present in the element's locator, text

class selenium.webdriver.support.expected_conditions.**title_contains**(*title*)
Bases: object

An expectation for checking that the title contains a case-sensitive substring. title is the fragment of title expected returns True when the title matches, False otherwise

class selenium.webdriver.support.expected_conditions.**title_is**(*title*)
Bases: object

An expectation for checking the title of a page. title is the expected title, which must be an exact match returns True if the title matches, false otherwise.

class selenium.webdriver.support.expected_conditions.**visibility_of**(*element*)
Bases: object

An expectation for checking that an element, known to be present on the DOM of a page, is visible. Visibility means that the element is not only displayed but also has a height and width that is greater than 0. element is the WebElement returns the (same) WebElement once it is visible

class selenium.webdriver.support.expected_conditions.**visibility_of_element_located**(*locator*)
Bases: object

An expectation for checking that an element is present on the DOM of a page and visible. Visibility means that the element is not only displayed but also has a height and width that is greater than 0. locator - used to find the element returns the WebElement once it is located and visible

CHAPTER 8

Appendix: Frequently Asked Questions

Another FAQ: https://code.google.com/p/selenium/wiki/FrequentlyAskedQuestions

8.1 How to use ChromeDriver ?

Download the latest chromedriver from download page. Unzip the file:

```
unzip chromedriver_linux32_x.x.x.x.zip
```

You should see a `chromedriver` executable. Now you can create an instance of Chrome WebDriver like this:

```
driver = webdriver.Chrome(executable_path="/path/to/chromedriver")
```

The rest of the example should work as given in other documentation.

8.2 Does Selenium 2 support XPath 2.0 ?

Ref: http://seleniumhq.org/docs/03_webdriver.html#how-xpath-works-in-webdriver

Selenium delegates XPath queries down to the browser's own XPath engine, so Selenium support XPath supports whatever the browser supports. In browsers which don't have native XPath engines (IE 6,7,8), Selenium supports XPath 1.0 only.

8.3 How to scroll down to the bottom of a page ?

Ref: http://blog.varunin.com/2011/08/scrolling-on-pages-using-selenium.html

You can use the *execute_script* method to execute javascript on the loaded page. So, you can call the JavaScript API to scroll to the bottom or any other position of a page.

Here is an example to scroll to the bottom of a page:

```
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

The window object in DOM has a scrollTo method to scroll to any position of an opened window. The scrollHeight is a common property for all elements. The *document.body.scrollHeight* will give the height of the entire body of the page.

8.4 How to auto save files using custom Firefox profile ?

Ref: http://stackoverflow.com/questions/1176348/access-to-file-download-dialog-in-firefox

Ref: http://blog.codecentric.de/en/2010/07/file-downloads-with-selenium-mission-impossible/

The first step is to identify the type of file you want to auto save.

To identify the content type you want to download automatically, you can use curl:

```
curl -I URL | grep "Content-Type"
```

Another way to find content type is using the requests module, you can use it like this:

```
import requests
content_type = requests.head('http://www.python.org').headers['content-type']
print(content_type)
```

Once the content type is identified, you can use it to set the firefox profile preference: `browser.helperApps.neverAsk.saveToDisk`

Here is an example:

```
import os

from selenium import webdriver

fp = webdriver.FirefoxProfile()

fp.set_preference("browser.download.folderList",2)
fp.set_preference("browser.download.manager.showWhenStarting",False)
fp.set_preference("browser.download.dir", os.getcwd())
fp.set_preference("browser.helperApps.neverAsk.saveToDisk", "application/octet-stream")

browser = webdriver.Firefox(firefox_profile=fp)
browser.get("http://pypi.python.org/pypi/selenium")
browser.find_element_by_partial_link_text("selenium-2").click()
```

In the above example, `application/octet-stream` is used as the content type.

The `browser.download.dir` option specify the directory where you want to download the files.

8.5 How to upload files into file inputs ?

Select the `<input type="file">` element and call the `send_keys()` method passing the file path, either the path relative to the test script, or an absolute path. Keep in mind the differences in path names between Windows and Unix systems.

8.6 How to use firebug with Firefox ?

First download the Firebug XPI file, later you call the `add_extension` method available for the firefox profile:

```
from selenium import webdriver

fp = webdriver.FirefoxProfile()
```

```
fp.add_extension(extension='firebug-1.8.4.xpi')
fp.set_preference("extensions.firebug.currentVersion", "1.8.4") #Avoid startup screen
browser = webdriver.Firefox(firefox_profile=fp)
```

8.7 How to take screenshot of the current window ?

Use the *save_screenshot* method provided by the webdriver:

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.get('http://www.python.org/')
driver.save_screenshot('screenshot.png')
driver.quit()
```

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

S

selenium.common.exceptions, 26
selenium.webdriver.chrome.webdriver, 36
selenium.webdriver.common.action_chains, 29
selenium.webdriver.common.alert, 32
selenium.webdriver.common.by, 34
selenium.webdriver.common.desired_capabilities, 35
selenium.webdriver.common.keys, 32
selenium.webdriver.common.utils, 36
selenium.webdriver.firefox.webdriver, 36
selenium.webdriver.remote.webdriver, 37
selenium.webdriver.remote.webelement, 44
selenium.webdriver.support.color, 49
selenium.webdriver.support.expected_conditions, 49
selenium.webdriver.support.select, 47
selenium.webdriver.support.wait, 48