Selenium Python Bindings

Release 2

Baiju Muthukadan

Jun 07, 2023

Contents

Author Baiju Muthukadan

Note: This is not an official documentation. If you would like to contribute to this documentation, you can fork this project in GitHub and send pull requests. You can also send your feedback to my email: baiju.m.mail AT gmail DOT com. So far 60+ community members have contributed to this project (See the closed pull requests). I encourage contributors to add more sections and make it an awesome documentation! If you know any translation of this document, please send a PR to update the below list.

Translations:

- Chinese
- Japanese

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 3.5 and above.

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

1.2 Installing Python bindings for Selenium

Use pip to install the selenium package. Python 3 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 has venv which is almost the same as virtualenv.

You can also download Python bindings for Selenium from the PyPI page for selenium package. and install manually.

1.3 Instructions for Windows users

1. Install Python 3 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:\Python39\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:\Python39\python.exe C:\my_selenium_script.py
```

1.4 Installing from Git sources

To build Selenium Python from the source code, clone the official repository. It contains the source code for all official Selenium flavors, like Python, Java, Ruby and others. The Python code resides in the `/py` directory. To build, you will also need the Bazel build system.

Note: Currently, as Selenium gets near to the 4.0.0 release, it requires Bazel 3.2.0 (Install instructions), even though 3.3.0 is already available.

To build a Wheel from the sources, run the following command from the repository root:

```
bazel //py:selenium-wheel
```

This command will prepare the source code with some preprocessed JS files needed by some webdriver modules and build the `.whl` package inside the `./bazel-bin/py/` directory. Afterwards, you can use `pip` to install it.

1.5 Drivers

Selenium requires a driver to interface with the chosen browser. Firefox, for example, requires geckodriver, which needs to be installed before the below examples can be run. Make sure it's in your *PATH*, e. g., place it in */usr/bin* or */usr/local/bin*.

Failure to observe this step will give you an error *selenium.common.exceptions.WebDriverException: Message: 'geckodriver' executable needs to be in PATH.*

Other supported browsers will have their own drivers available. Links to some of the more popular browser drivers follow.

Chrome:	https://sites.google.com/chromium.org/driver/
Edge:	https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
Firefox:	https://github.com/mozilla/geckodriver/releases
Safari:	https://webkit.org/blog/6900/webdriver-support-in-safari-10/

For more information about driver installation, please refer the official documentation.

1.6 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 the 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 a 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
from selenium.webdriver.common.by import By

driver = webdriver.Firefox()
driver.get("http://www.python.org")
assert "Python" in driver.title
elem = driver.find_element(By.NAME, "q")
elem.clear()
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 Example Explained

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. The *By* class is used to locate elements within a document.

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

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. *Be aware 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 the word "Python" in it:

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

WebDriver offers a number of ways to find elements using the *find_element* method. For example, the input text element can be located by its *name* attribute using the *find_element* method and using By.NAME as its first parameter. A 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 sent using the *Keys* class imported from *selenium.webdriver.common.keys*. To be safe, we'll first clear any pre-populated text in the input field (e.g. "Search") so it doesn't affect our search results:

```
elem.clear()
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 the *quit* method instead of *close*. The *quit* method will exit the browser whereas *close* will close one tab, but if just one tab was open, by default most browsers 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 options for a tool/framework are pytest and nose.

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

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

(continues on next page)

(continued from previous page)

```
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")
        elem.send_keys(Keys.RETURN)
        self.assertNotIn("No results found.", driver.page_source)


    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 result shows that the test has been successfully completed.

Note: To run the above test in IPython or Jupyter, you should pass a couple of arguments to the *main* function as shown below:

```
unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

2.4 Walkthrough of the example

Initially, all the basic modules required are imported. The unittest module is a built-in Python module 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 provides keys in the keyboard like RETURN, F1, ALT etc. The *By* class is used to locate elements within a document.

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

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

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

The *setUp* method 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 an instance of a 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 creates 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. *Be aware 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 the word "Python" in it:

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

WebDriver offers a number of ways to find elements using the *find_element* method. For example, the input text element can be located by its *name* attribute using the *find_element* 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 sent using the *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 as per search if there is any. To ensure that some results are found, make an assertion:

```
self.assertNotIn("No results found.", 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 the *quit* method instead of *close*. The *quit* method will exit the entire browser, whereas *close* will close a tab, but if it is the only tab opened, by default most browsers 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 the 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 looking 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 the remote WebDriver. Here are some examples:

```
from selenium import webdriver

driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   options=webdriver.ChromeOptions()
)

driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   options=webdriver.FirefoxOptions()
)
```

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. *Be aware 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']")
element = driver.find_element(By.CSS_SELECTOR, "input#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 the *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 the 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 its 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 a "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 that particular 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, "//select[@name='name']"))
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).perform()
```

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 triggered 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 backward and forward 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 behavior of one browser over another.

3.7 Cookies

Before moving to the next section of the tutorial, 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 = {'name' : 'foo', 'value' : 'bar'}
driver.add_cookie(cookie)

# And now output all the available cookies for the current URL
driver.get_cookies()
```

CHAPTER 4

Locating Elements

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

- *find_element*

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

- *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')
```

The attributes available for the *By* class are used to locate elements on a page. These are the attributes available for *By* class:

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

The 'By' class is used to specify which attribute is used to locate elements on a page. These are the various ways the attributes are used to locate elements on a page:

```
find_element(By.ID, "id")
find_element(By.NAME, "name")
find_element(By.XPATH, "xpath")
find_element(By.LINK_TEXT, "link text")
```

(continues on next page)

(continued from previous page)

```
find_element(By.PARTIAL_LINK_TEXT, "partial link text")
find_element(By.TAG_NAME, "tag name")
find_element(By.CLASS_NAME, "class name")
find_element(By.CSS_SELECTOR, "css selector")
```

If you want to locate several elements with the same attribute replace find_element with find_elements.

4.1 Locating by Id

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

For instance, consider 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 the *name* attribute of an element. With this strategy, the first element with a matching *name* attribute will be returned. If no element has a matching *name* attribute, a `NoSuchElementException` will be raised.

For instance, consider 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 occurs 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 supports the simple methods of locating by id or name attributes and extends them by opening 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, consider 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 *id* set to *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 *name* set to *username*
2. First input child element of the form element with attribute *id* set to *loginForm*
3. First input element with attribute *name* set to *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 *name* set to *continue* and attribute *type* set to *button*
2. Fourth input child element of the form element with attribute *id* set to *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.

Here is a couple of very useful Add-ons that can assist in discovering the XPath of an element:

- xPath Finder - Plugin to get the elements xPath.
- XPath Helper - for Google Chrome

4.4 Locating Hyperlinks by Link Text

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

For instance, consider 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 name. With this strategy, the first element with the matching class name attribute will be returned. If no element has a matching class name attribute, 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 using CSS selector syntax. With this strategy, the first element matching the given CSS selector will be returned. If no element matches the provided 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 by the browser, the elements within that page may load at different time intervals. This makes locating elements difficult: if an element is not yet present in the DOM, a locate function will raise an *ElementNotVisibleException* exception. Using waits, we can solve this issue. Waiting provides some slack between actions performed - mostly locating an element or any other operation with the element.

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

5.1 Explicit Waits

An explicit wait is a code you define to wait for a certain condition to occur before proceeding further in the code. The extreme 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.wait 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()
```

In the code above, Selenium will wait for a maximum of 10 seconds for an element matching the given criteria to be found. If no element is found in that time, a TimeoutException is thrown. By default, WebDriverWait calls the ExpectedCondition every 500 milliseconds until it returns success. ExpectedCondition will return *true* (Boolean) in case of success or *not null* if it fails to locate an element.

Expected Conditions

There are some common conditions that are frequently of use when automating web browsers. Listed below are the names of each. Selenium Python binding provides some convenience 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
- 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.

Custom Wait Conditions

You can also create custom wait conditions when none of the previous convenience methods fit your requirements. A custom wait condition can be created using a class with *__call__* method which returns *False* when the condition doesn't match.

```
class element_has_css_class(object):
  """An expectation for checking that an element has a particular css class.

  locator - used to find the element
  returns the WebElement once it has the particular css class
  """
  def __init__(self, locator, css_class):
```

(continues on next page)

(continued from previous page)

```
    self.locator = locator
    self.css_class = css_class

  def __call__(self, driver):
    element = driver.find_element(*self.locator)    # Finding the referenced element
    if self.css_class in element.get_attribute("class"):
        return element
    else:
        return False

# Wait until an element with id='myNewInput' has class 'myCSSClass'
wait = WebDriverWait(driver, 10)
element = wait.until(element_has_css_class((By.ID, 'myNewInput'), "myCSSClass"))
```

Note: polling2 Library

You may also consider using polling2 library which you need to install separately.

5.2 Implicit Waits

An implicit wait tells WebDriver to poll the DOM for a certain amount of time when trying to find any element (or elements) not immediately available. The default setting is 0 (zero). Once set, the implicit wait is set for the life of the WebDriver object.

```
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

This chapter is a tutorial introduction to the Page Objects design pattern. A page object represents an area where the test interacts within the web application user interface.

Benefits of using page object pattern:

- Easy to read test cases
- Creating reusable code that can share across multiple test cases
- Reducing the amount of duplicated code
- If the user interface changes, the fix needs changes in only one place

6.1 Test case

Here is a test case that searches for a word on the *python.org* website and ensures some results. The following section will introduce the *page* module where the page objects will be defined.

```
import unittest
from selenium import webdriver
import page

class PythonOrgSearch(unittest.TestCase):
    """A sample test class to show how page object works"""

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

    def test_search_in_python_org(self):
        """Tests python.org search feature. Searches for the word "pycon" then
        verified that some results show up.  Note that it does not look for
        any particular text in search results page. This test verifies that
        the results were not empty."""
```

(continues on next page)

(continued from previous page)

```
        #Load the main page. In this case the home page of Python.org.
        main_page = page.MainPage(self.driver)
        #Checks if the word "Python" is in title
        self.assertTrue(main_page.is_title_matches(), "python.org title doesn't match.
↪")
        #Sets the text of search textbox to "pycon"
        main_page.search_text_element = "pycon"
        main_page.click_go_button()
        search_results_page = page.SearchResultsPage(self.driver)
        #Verifies that the results page is not empty
        self.assertTrue(search_results_page.is_results_found(), "No results found.")

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

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

6.2 Page object classes

The page object pattern intends to create an object for each part of a web page. This technique helps build a separation between the test code and the actual code that interacts with the web page.

The `page.py` will look like this:

```
from element import BasePageElement
from locators import MainPageLocators

class SearchTextElement(BasePageElement):
    """This class gets the search text from the specified locator"""

    #The locator for search box where search string is entered
    locator = 'q'


class BasePage(object):
    """Base class to initialize the base page that will be called from all
    pages"""

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


class MainPage(BasePage):
    """Home page action methods come here. I.e. Python.org"""

    #Declares a variable that will contain the retrieved text
    search_text_element = SearchTextElement()

    def is_title_matches(self):
        """Verifies that the hardcoded text "Python" appears in page title"""

        return "Python" in self.driver.title
```

(continues on next page)

(continued from previous page)

```
    def click_go_button(self):
        """Triggers the search"""

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


class SearchResultsPage(BasePage):
    """Search results page action methods come here"""

    def is_results_found(self):
        # Probably should search for this text in the specific page
        # element, but as for now it works fine
        return "No results found." not in self.driver.page_source
```

6.3 Page elements

The `element.py` will look like this:

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


class BasePageElement(object):
    """Base page class that is initialized on every page object class."""

    def __set__(self, obj, value):
        """Sets the text to the value supplied"""

        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_name(self.locator))
        driver.find_element_by_name(self.locator).clear()
        driver.find_element_by_name(self.locator).send_keys(value)

    def __get__(self, obj, owner):
        """Gets the text of the specified object"""

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

6.4 Locators

One of the practices is to separate the locator strings from the place where they are getting used. In this example, locators of the same page belong to the same class.

The `locators.py` will look like this:

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

class MainPageLocators(object):
    """A class for main page locators. All main page locators should come here"""

    GO_BUTTON = (By.ID, 'submit')

class SearchResultsPageLocators(object):
    """A class for search results locators. All search results locators should
    come here"""

    pass
```

CHAPTER 7

WebDriver API

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

This chapter covers all the interfaces of Selenium WebDriver.

Recommended Import Style

The API definitions in this chapter show 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 the 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 currently loaded page.

  Usage:

```
driver.current_url
```

Here is an example of 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.**ElementClickInterceptedException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

The Element Click command could not be completed because the element receiving the events is obscuring the element that was requested to be clicked.

exception selenium.common.exceptions.**ElementNotInteractableException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.InvalidElementStateException*

Thrown when an element is present in the DOM but interactions with that element will hit another element due to paint order.

exception selenium.common.exceptions.**ElementNotSelectableException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = 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: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = 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.**ImeActivationFailedException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Thrown when activating an IME engine has failed.

exception selenium.common.exceptions.**ImeNotAvailableException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = 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.**InsecureCertificateException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Navigation caused the user agent to hit a certificate warning, which is usually the result of an expired or invalid TLS certificate.

exception selenium.common.exceptions.**InvalidArgumentException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

The arguments passed to a command are either invalid or malformed.

exception selenium.common.exceptions.**InvalidCookieDomainException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = 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.**InvalidCoordinatesException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

The coordinates provided to an interaction's operation are invalid.

exception selenium.common.exceptions.**InvalidElementStateException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Thrown when a command could not be completed because the element is in an invalid state.

This can be caused by attempting to clear an element that isn't both editable and resettable.

exception selenium.common.exceptions.**InvalidSelectorException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

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.**InvalidSessionIdException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Occurs if the given session id is not in the list of active sessions, meaning the session either does not exist or that it's not active.

exception selenium.common.exceptions.**InvalidSwitchToTargetException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

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

exception selenium.common.exceptions.**JavascriptException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

An error occurred while executing JavaScript supplied by the user.

exception selenium.common.exceptions.**MoveTargetOutOfBoundsException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = 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: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = 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: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = 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.**NoSuchCookieException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

No cookie matching the given path name was found amongst the associated cookies of the current browsing context's active document.

exception selenium.common.exceptions.**NoSuchElementException**(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = 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: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.InvalidSwitchToTargetException*

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

exception selenium.common.exceptions.**NoSuchShadowRootException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Thrown when trying to access the shadow root of an element when it does not have a shadow root attached.

exception selenium.common.exceptions.**NoSuchWindowException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = 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.**ScreenshotException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

A screen capture was made impossible.

exception selenium.common.exceptions.**SeleniumManagerException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Raised when an issue interacting with selenium manager occurs.

exception selenium.common.exceptions.**SessionNotCreatedException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

A new session could not be created.

exception selenium.common.exceptions.**StaleElementReferenceException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = 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: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Thrown when a command does not complete in enough time.

exception selenium.common.exceptions.**UnableToSetCookieException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Thrown when a driver fails to set a cookie.

exception selenium.common.exceptions.**UnexpectedAlertPresentException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*, *alert_text: Optional[str] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

Thrown when an unexpected alert has appeared.

Usually raised when an unexpected modal is blocking the webdriver from executing commands.

__init__(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*, *alert_text: Optional[str] = None*) → None

Initialize self. See help(type(self)) for accurate signature.

exception selenium.common.exceptions.**UnexpectedTagNameException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

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

exception selenium.common.exceptions.**UnknownMethodException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: *selenium.common.exceptions.WebDriverException*

The requested command matched a known URL but did not match any methods for that URL.

exception selenium.common.exceptions.**WebDriverException**(*msg: Optional[str] = None*, *screen: Optional[str] = None*, *stacktrace: Optional[Sequence[str]] = None*)

Bases: Exception

Base webdriver exception.

__init__(*msg: Optional[str] = None, screen: Optional[str] = None, stacktrace: Optional[Sequence[str]] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

7.2 Action Chains

The ActionChains implementation,

class selenium.webdriver.common.action_chains.**ActionChains**(*driver, duration=250*)

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.

__init__(*driver, duration=250*)
Creates a new ActionChains.

Args

- driver: The WebDriver instance which performs user actions.
- duration: override the default 250 msecs of DEFAULT_MOVE_DURATION in PointerInput

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:

```
ActionChains(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:

```
ActionChains(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 in-view center point of the element.

Args

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

pause(*seconds*)
Pause all inputs for the specified duration in seconds.

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.

reset_actions()
Clears actions that are already stored locally and on the remote end.

scroll(*x: int*, *y: int*, *delta_x: int*, *delta_y: int*, *duration: int = 0*, *origin: str = 'viewport'*)
Sends wheel scroll information to the browser to be processed.

Args

- x: starting X coordinate
- y: starting Y coordinate
- delta_x: the distance the mouse will scroll on the x axis
- delta_y: the distance the mouse will scroll on the y axis

scroll_by_amount(*delta_x: int*, *delta_y: int*)
Scrolls by provided amounts with the origin in the top left corner of the viewport.

Args

- delta_x: Distance along X axis to scroll using the wheel. A negative value scrolls left.
- delta_y: Distance along Y axis to scroll using the wheel. A negative value scrolls up.

scroll_from_origin(*scroll_origin: selenium.webdriver.common.actions.wheel_input.ScrollOrigin*, *delta_x: int*, *delta_y: int*)

Scrolls by provided amount based on a provided origin. The scroll origin is either the center of an element or the upper left of the viewport plus any offsets. If the origin is an element, and the element is not in the viewport, the bottom of the element will first be scrolled to the bottom of the viewport.

Args

- origin: Where scroll originates (viewport or element center) plus provided offsets.
- delta_x: Distance along X axis to scroll using the wheel. A negative value scrolls left.
- delta_y: Distance along Y axis to scroll using the wheel. A negative value scrolls up.

Raises If the origin with offset is outside the viewport. - MoveTargetOutOfBoundsException - If the origin with offset is outside the viewport.

scroll_to_element(*element: selenium.webdriver.remote.webelement.WebElement*)

If the element is outside the viewport, scrolls the bottom of the element to the bottom of the viewport.

Args

- element: Which element to scroll into the viewport.

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)

__init__ (*driver*)
Creates a new Alert.

Args

- driver: The WebDriver instance which performs user actions.

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 = '\ue025'

ALT = '\ue00a'

ARROW_DOWN = '\ue015'

ARROW_LEFT = '\ue012'

ARROW_RIGHT = '\ue014'

ARROW_UP = '\ue013'

BACKSPACE = '\ue003'

BACK_SPACE = '\ue003'

CANCEL = '\ue001'

CLEAR = '\ue005'

COMMAND = '\ue03d'

CONTROL = '\ue009'

DECIMAL = '\ue028'

```
DELETE = '\ue017'
DIVIDE = '\ue029'
DOWN = '\ue015'
END = '\ue010'
ENTER = '\ue007'
EQUALS = '\ue019'
ESCAPE = '\ue00c'
F1 = '\ue031'
F10 = '\ue03a'
F11 = '\ue03b'
F12 = '\ue03c'
F2 = '\ue032'
F3 = '\ue033'
F4 = '\ue034'
F5 = '\ue035'
F6 = '\ue036'
F7 = '\ue037'
F8 = '\ue038'
F9 = '\ue039'
HELP = '\ue002'
HOME = '\ue011'
INSERT = '\ue016'
LEFT = '\ue012'
LEFT_ALT = '\ue00a'
LEFT_CONTROL = '\ue009'
LEFT_SHIFT = '\ue008'
META = '\ue03d'
MULTIPLY = '\ue024'
NULL = '\ue000'
NUMPAD0 = '\ue01a'
NUMPAD1 = '\ue01b'
NUMPAD2 = '\ue01c'
NUMPAD3 = '\ue01d'
NUMPAD4 = '\ue01e'
NUMPAD5 = '\ue01f'
NUMPAD6 = '\ue020'
```

NUMPAD7 = '\ue021'

NUMPAD8 = '\ue022'

NUMPAD9 = '\ue023'

PAGE_DOWN = '\ue00f'

PAGE_UP = '\ue00e'

PAUSE = '\ue00b'

RETURN = '\ue006'

RIGHT = '\ue014'

SEMICOLON = '\ue018'

SEPARATOR = '\ue026'

SHIFT = '\ue008'

SPACE = '\ue00d'

SUBTRACT = '\ue027'

TAB = '\ue004'

UP = '\ue013'

ZENKAKU_HANKAKU = '\ue040'

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.

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.

CHROME = {'browserName': 'chrome'}

EDGE = {'browserName': 'MicrosoftEdge'}

FIREFOX = {'acceptInsecureCerts': True, 'browserName': 'firefox', 'moz:debuggerAddre

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

HTMLUNITWITHJS = {'browserName': 'htmlunit', 'javascriptEnabled': True, 'platform':

INTERNETEXPLORER = {'browserName': 'internet explorer', 'platformName': 'windows'}

IPAD = {'browserName': 'iPad', 'platform': 'mac', 'version': ''}

IPHONE = {'browserName': 'iPhone', 'platform': 'mac', 'version': ''}

SAFARI = {'browserName': 'safari', 'platformName': 'mac'}

WEBKITGTK = {'browserName': 'MiniBrowser', 'platform': 'ANY', 'version': ''}

WPEWEBKIT = {'browserName': 'MiniBrowser', 'platform': 'ANY', 'version': ''}

7.7 Touch Actions

7.8 Proxy

The Proxy implementation.

class selenium.webdriver.common.proxy.**Proxy**(*raw=None*)
Bases: object

Proxy contains information about proxy type and necessary proxy settings.

__init__(*raw=None*)
Creates a new Proxy.

Args

- raw: raw proxy data. If None, default class values are used.

add_to_capabilities(*capabilities*)
Adds proxy information as capability in specified capabilities.

Args

- capabilities: The capabilities to which proxy will be added.

auto_detect
Returns autodetect setting.

autodetect = False

ftpProxy = ''

ftp_proxy
Returns ftp proxy setting.

httpProxy = ''

http_proxy
Returns http proxy setting.

noProxy = ''

no_proxy
Returns noproxy setting.

proxyAutoconfigUrl = ''

proxyType = {'ff_value': 6, 'string': 'UNSPECIFIED'}

proxy_autoconfig_url
Returns proxy autoconfig url setting.

proxy_type
Returns proxy type as *ProxyType*.

socksPassword = ''

socksProxy = ''

socksUsername = ''

socksVersion = None

socks_password
Returns socks proxy password setting.

socks_proxy
Returns socks proxy setting.

socks_username
Returns socks proxy username setting.

socks_version
Returns socks proxy version setting.

sslProxy = ''

ssl_proxy
Returns https proxy setting.

class selenium.webdriver.common.proxy.**ProxyType**
Bases: object

Set of possible types of proxy.

Each proxy type has 2 properties: 'ff_value' is value of Firefox profile preference, 'string' is id of proxy type.

classmethod load(*value*)

AUTODETECT = {'ff_value': 4, 'string': 'AUTODETECT'}

DIRECT = {'ff_value': 0, 'string': 'DIRECT'}

MANUAL = {'ff_value': 1, 'string': 'MANUAL'}

PAC = {'ff_value': 2, 'string': 'PAC'}

RESERVED_1 = {'ff_value': 3, 'string': 'RESERVED1'}

SYSTEM = {'ff_value': 5, 'string': 'SYSTEM'}

UNSPECIFIED = {'ff_value': 6, 'string': 'UNSPECIFIED'}

class selenium.webdriver.common.proxy.**ProxyTypeFactory**
Bases: object

Factory for proxy types.

static make(*ff_value*, *string*)

7.9 Utilities

The Utils methods.

selenium.webdriver.common.utils.**find_connectable_ip**(*host: Union[str, bytes, bytearray, None], port: Optional[int] = None*) → Optional[str]

Resolve a hostname to an IP, preferring IPv4 addresses.

We prefer IPv4 so that we don't change behavior from previous IPv4-only implementations, and because some drivers (e.g., FirefoxDriver) do not support IPv6 connections.

If the optional port number is provided, only IPs that listen on the given port are considered.

Args

- host - A hostname.
- port - Optional port number.

Returns A single IP address, as a string. If any IPv4 address is found, one is returned. Otherwise, if any IPv6 address is found, one is returned. If neither, then None is returned.

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

selenium.webdriver.common.utils.**is_connectable**(*port: int, host: Optional[str] = 'localhost'*) → bool

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: Union[int, str]*) → bool
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.

selenium.webdriver.common.utils.**join_host_port**(*host: str*, *port: int*) → str
Joins a hostname and port together.

This is a minimal implementation intended to cope with IPv6 literals. For example, _join_host_port('::1', 80) == '[::1]:80'.

Args

- host - A hostname.
- port - An integer port.

selenium.webdriver.common.utils.**keys_to_typing**(*value: Iterable[Union[str, int, float]]*) → List[str]
Processes the values that will be typed in the element.

7.10 Service

class selenium.webdriver.common.service.**Service**(*executable: str*, *port: int = 0*, *log_file: Union[int, IO[Any]] = -3*, *env: Optional[Mapping[Any, Any]] = None*, *start_error_message: Optional[str] = None*, ***kwargs*)
Bases: abc.ABC

The abstract base class for all service objects. Services typically launch a child program in a new process as an interim process to communicate with a browser.

Parameters

- **executable** – install path of the executable.
- **port** – Port for the service to run on, defaults to 0 where the operating system will decide.
- **log_file** – (Optional) file descriptor (pos int) or file object with a valid file descriptor. subprocess.PIPE & subprocess.DEVNULL are also valid values.
- **env** – (Optional) Mapping of environment variables for the new process, defaults to *os.environ*.

__init__(*executable: str*, *port: int = 0*, *log_file: Union[int, IO[Any]] = -3*, *env: Optional[Mapping[Any, Any]] = None*, *start_error_message: Optional[str] = None*, ***kwargs*) → None
Initialize self. See help(type(self)) for accurate signature.

assert_process_still_running() → None
Check if the underlying process is still running.

command_line_args() → List[str]
A List of program arguments (excluding the executable).

is_connectable() → bool
Establishes a socket connection to determine if the service running on the port is accessible.

send_remote_shutdown_command() → None
Dispatch an HTTP request to the shutdown endpoint for the service in an attempt to stop it.

start() → None
Starts the Service.

Exceptions

- WebDriverException : Raised either when it can't start the service or when it can't connect to the service

stop() → None
Stops the service.

path

service_url
Gets the url of the Service.

7.11 Application Cache

The ApplicationCache implementation.

class selenium.webdriver.common.html5.application_cache.**ApplicationCache**(*driver*)
Bases: object

__init__(*driver*)
Creates a new Application Cache.

Args

- driver: The WebDriver instance which performs user actions.

CHECKING = 2

DOWNLOADING = 3

IDLE = 1

OBSOLETE = 5

UNCACHED = 0

UPDATE_READY = 4

status
Returns a current status of application cache.

7.12 Firefox WebDriver

class selenium.webdriver.firefox.webdriver.**WebDriver**(*firefox_profile=None*, *firefox_binary=None*, *capabilities=None*, *proxy=None*, *executable_path='geckodriver'*, *options=None*, *service_log_path='geckodriver.log'*, *service_args=None*, *service=None*, *desired_capabilities=None*, *log_path=None*, *keep_alive=True*)

Bases: *selenium.webdriver.remote.webdriver.WebDriver*

__init__(*firefox_profile=None*, *firefox_binary=None*, *capabilities=None*, *proxy=None*, *executable_path='geckodriver'*, *options=None*, *service_log_path='geckodriver.log'*, *service_args=None*, *service=None*, *desired_capabilities=None*, *log_path=None*, *keep_alive=True*) → None

Starts a new local session of Firefox.

Based on the combination and specificity of the various keyword arguments, a capabilities dictionary will be constructed that is passed to the remote end.

The keyword arguments given to this constructor are helpers to more easily allow Firefox WebDriver sessions to be customised with different options. They are mapped on to a capabilities dictionary that is passed on to the remote end.

As some of the options, such as *firefox_profile* and *options.profile* are mutually exclusive, precedence is given from how specific the setting is. *capabilities* is the least specific keyword argument, followed by *options*, followed by *firefox_binary* and *firefox_profile*.

In practice this means that if *firefox_profile* and *options.profile* are both set, the selected profile instance will always come from the most specific variable. In this case that would be *firefox_profile*. This will result in *options.profile* to be ignored because it is considered a less specific setting than the top-level *firefox_profile* keyword argument. Similarly, if you had specified a *capabilities["moz:firefoxOptions"]["profile"]* Base64 string, this would rank below *options.profile*.

Parameters

- **firefox_profile** – Deprecated: Instance of `FirefoxProfile` object or a string. If undefined, a fresh profile will be created in a temporary location on the system.
- **firefox_binary** – Deprecated: Instance of `FirefoxBinary` or full path to the Firefox binary. If undefined, the system default Firefox installation will be used.
- **capabilities** – Deprecated: Dictionary of desired capabilities.
- **proxy** – Deprecated: The proxy settings to use when communicating with Firefox via the extension connection.
- **executable_path** – Deprecated: Full path to override which geckodriver binary to use for Firefox 47.0.1 and greater, which defaults to picking up the binary from the system path.
- **options** – Instance of `options.Options`.
- **service** – (Optional) service instance for managing the starting and stopping of the driver.
- **service_log_path** – Deprecated: Where to log information from the driver.

- **service_args** – Deprecated: List of args to pass to the driver service
- **desired_capabilities** – Deprecated: alias of capabilities. In future versions of this library, this will replace 'capabilities'. This will make the signature consistent with RemoteWebDriver.
- **keep_alive** – Whether to configure remote_connection.RemoteConnection to use HTTP keep-alive.

context(*context*)

Sets the context that Selenium commands are running in using a *with* statement. The state of the context on the server is saved before entering the block, and restored upon exiting it.

Parameters **context** – Context, may be one of the class properties *CONTEXT_CHROME* or *CONTEXT_CONTENT*.

Usage example:

```
with selenium.context(selenium.CONTEXT_CHROME):
    # chrome scope
    ... do stuff ...
```

get_full_page_screenshot_as_base64() → str

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

Usage

```
driver.get_full_page_screenshot_as_base64()
```

get_full_page_screenshot_as_file(*filename*) → bool

Saves a full document screenshot of the current window to a PNG image file. 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. This should end with a *.png* extension.

Usage

```
driver.get_full_page_screenshot_as_file('/Screenshots/foo.png')
```

get_full_page_screenshot_as_png() → bytes

Gets the full document screenshot of the current window as a binary data.

Usage

```
driver.get_full_page_screenshot_as_png()
```

install_addon(*path*, *temporary=False*) → str

Installs Firefox addon.

Returns identifier of installed addon. This identifier can later be used to uninstall addon.

Parameters

- **temporary** – allows you to load browser extensions temporarily during a session
- **path** – Absolute path to the addon that will be installed.

Usage

```
driver.install_addon('/path/to/firebug.xpi')
```

quit() → None
Quits the driver and close every associated window.

save_full_page_screenshot(*filename*) → bool
Saves a full document screenshot of the current window to a PNG image file. 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. This should end with a *.png* extension.

Usage

```
driver.save_full_page_screenshot('/Screenshots/foo.png')
```

set_context(*context*) → None

uninstall_addon(*identifier*) → None
Uninstalls Firefox addon using its identifier.

Usage

```
driver.uninstall_addon('addon@foo.com')
```

CONTEXT_CHROME = 'chrome'

CONTEXT_CONTENT = 'content'

firefox_profile

7.13 Firefox WebDriver Options

class selenium.webdriver.firefox.options.**Log**
Bases: object

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

to_capabilities() → dict

class selenium.webdriver.firefox.options.**Options**
Bases: selenium.webdriver.common.options.ArgOptions

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

enable_mobile(*android_package: str = 'org.mozilla.firefox', android_activity=None, device_serial=None*)
Enables mobile browser use for browsers that support it.

Args android_activity: The name of the android package to start

set_preference(*name: str, value: Union[str, int, bool]*)
Sets a preference.

to_capabilities() → dict
Marshals the Firefox options to a *moz:firefoxOptions* object.

KEY = 'moz:firefoxOptions'

binary
Returns the FirefoxBinary instance.

binary_location

Returns The location of the binary.

default_capabilities
Return minimal capabilities necessary as a dictionary.

headless

Returns True if the headless argument is set, else False

preferences

Returns A dict of preferences.

profile

Returns The Firefox profile to use.

7.14 Firefox WebDriver Profile

exception selenium.webdriver.firefox.firefox_profile.**AddonFormatError**
Bases: Exception

Exception for not well-formed add-on manifest files.

class selenium.webdriver.firefox.firefox_profile.**FirefoxProfile**(*profile_directory=None*)
Bases: object

__init__(*profile_directory=None*)
Initialises a new instance of a Firefox Profile.

Args

- profile_directory: Directory of profile that you want to use. If a directory is passed in it will be cloned and the cloned directory will be used by the driver when instantiated. This defaults to None and will create a new directory when object is created.

add_extension(*extension='webdriver.xpi'*)

set_preference(*key*, *value*)
sets the preference that we want in the profile.

update_preferences()

ANONYMOUS_PROFILE_NAME = 'WEBDRIVER_ANONYMOUS_PROFILE'

DEFAULT_PREFERENCES = None

accept_untrusted_certs

assume_untrusted_cert_issuer

encoded
A zipped, base64 encoded string of profile directory for use with remote WebDriver JSON wire protocol.

path
Gets the profile directory that is currently being used.

port
Gets the port that WebDriver is working on.

7.15 Firefox WebDriver Binary

class selenium.webdriver.firefox.firefox_binary.**FirefoxBinary**(*firefox_path=None*, *log_file=None*)

Bases: object

__init__(*firefox_path=None*, *log_file=None*)
Creates a new instance of Firefox binary.

Args

- firefox_path - Path to the Firefox executable. By default, it will be detected from the standard locations.
- **log_file - A file object to redirect the firefox process output to. It can be sys.stdout.** Please note that with parallel run the output won't be synchronous. By default, it will be redirected to /dev/null.

add_command_line_options(**args*)

kill()
Kill the browser.

This is useful when the browser is stuck.

launch_browser(*profile*, *timeout=30*)
Launches the browser for the given profile name.

It is assumed the profile already exists.

which(*fname*)
Returns the fully qualified path by searching Path of the given name.

NO_FOCUS_LIBRARY_NAME = 'x_ignore_nofocus.so'

7.16 Firefox WebDriver Extension Connection

exception selenium.webdriver.firefox.extension_connection.**ExtensionConnectionError**

Bases: Exception

An internal error occurred int the extension.

Might be caused by bad input or bugs in webdriver

class selenium.webdriver.firefox.extension_connection.**ExtensionConnection**(*host*, *firefox_profile*, *firefox_binary=None*, *timeout=30*)

Bases: *selenium.webdriver.remote.remote_connection.RemoteConnection*

__init__(*host*, *firefox_profile*, *firefox_binary=None*, *timeout=30*)
Initialize self. See help(type(self)) for accurate signature.

connect()
Connects to the extension and retrieves the session id.

classmethod connect_and_quit()
Connects to an running browser and quit immediately.

classmethod is_connectable()
Tries to connect to the extension but do not retrieve context.

quit(*sessionId=None*)

7.17 Chrome WebDriver

class selenium.webdriver.chrome.webdriver.**WebDriver**(*executable_path='chromedriver'*, *port=0*, *options: selenium.webdriver.chrome.options.Options = None*, *service_args=None*, *desired_capabilities=None*, *service_log_path=None*, *chrome_options=None*, *service: selenium.webdriver.chrome.service.Service = None*, *keep_alive=None*)
Bases: selenium.webdriver.chromium.webdriver.ChromiumDriver

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

__init__(*executable_path='chromedriver'*, *port=0*, *options: selenium.webdriver.chrome.options.Options = None*, *service_args=None*, *desired_capabilities=None*, *service_log_path=None*, *chrome_options=None*, *service: selenium.webdriver.chrome.service.Service = None*, *keep_alive=None*) → None
Creates a new instance of the chrome driver. Starts the service and then creates new instance of chrome driver.

Args

- executable_path - Deprecated: path to the executable. If the default is used it assumes the executable is in the $PATH
- port - Deprecated: port you would like the service to run, if left as 0, a free port will be found.
- options - this takes an instance of ChromeOptions
- service - Service object for handling the browser driver if you need to pass extra details
- service_args - Deprecated: List of args to pass to the driver service
- desired_capabilities - Deprecated: Dictionary object with non-browser specific capabilities only, such as "proxy" or "loggingPref".
- service_log_path - Deprecated: Where to log information from the driver.
- keep_alive - Deprecated: Whether to configure ChromeRemoteConnection to use HTTP keep-alive.

create_options() → selenium.webdriver.chrome.options.Options

7.18 Chrome WebDriver Options

class selenium.webdriver.chrome.options.**Options**
Bases: selenium.webdriver.chromium.options.ChromiumOptions

enable_mobile(*android_package: str = 'com.android.chrome'*, *android_activity: Optional[str] = None*, *device_serial: Optional[str] = None*) → None
Enables mobile browser use for browsers that support it.

Args android_activity: The name of the android package to start

default_capabilities
Return minimal capabilities necessary as a dictionary.

7.19 Chrome WebDriver Service

class selenium.webdriver.chrome.service.**Service**(*executable_path: str = 'chromedriver'*, *port: int = 0*, *service_args: Optional[List[str]] = None*, *log_path: Optional[str] = None*, *env: Optional[Mapping[str, str]] = None*, ***kwargs*)
Bases: selenium.webdriver.chromium.service.ChromiumService

A Service class that is responsible for the starting and stopping of *chromedriver*.

Parameters

- **executable_path** – install path of the chromedriver executable, defaults to *chromedriver*.
- **port** – Port for the service to run on, defaults to 0 where the operating system will decide.
- **service_args** – (Optional) List of args to be passed to the subprocess when launching the executable.
- **log_path** – (Optional) String to be passed to the executable as *–log-path*.
- **env** – (Optional) Mapping of environment variables for the new process, defaults to *os.environ*.

__init__(*executable_path: str = 'chromedriver'*, *port: int = 0*, *service_args: Optional[List[str]] = None*, *log_path: Optional[str] = None*, *env: Optional[Mapping[str, str]] = None*, ***kwargs*) → None
Initialize self. See help(type(self)) for accurate signature.

7.20 Remote WebDriver

The WebDriver implementation.

class selenium.webdriver.remote.webdriver.**BaseWebDriver**
Bases: object

Abstract Base Class for all Webdriver subtypes.

ABC's allow custom implementations of Webdriver to be registered so that isinstance type checks will succeed.

class selenium.webdriver.remote.webdriver.**WebDriver**(*command_executor='http://127.0.0.1:4444'*, *desired_capabilities=None*, *browser_profile=None*, *proxy=None*, *keep_alive=True*, *file_detector=None*, *options: Union[selenium.webdriver.common.options.BaseOptions, List[selenium.webdriver.common.options.BaseOptions]] = None*)

Bases: *selenium.webdriver.remote.webdriver.BaseWebDriver*

Controls a browser by sending commands to a remote server. This server is expected to be running the WebDriver wire protocol as defined at https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol.

Attributes

- session_id - String ID of the browser session started and controlled by this WebDriver.
- **capabilities - Dictionary of effective capabilities of this browser session as returned** by the remote server. See https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities
- command_executor - remote_connection.RemoteConnection object used to execute commands.
- error_handler - errorhandler.ErrorHandler object used to handle errors.

__init__(*command_executor='http://127.0.0.1:4444'*, *desired_capabilities=None*, *browser_profile=None*, *proxy=None*, *keep_alive=True*, *file_detector=None*, *options: Union[selenium.webdriver.common.options.BaseOptions, List[selenium.webdriver.common.options.BaseOptions]] = None*) → None

Create a new driver that will issue commands using the wire protocol.

Args

- **command_executor - Either a string representing URL of the remote server or a custom** remote_connection.RemoteConnection object. Defaults to 'http://127.0.0.1:4444/wd/hub'.
- **desired_capabilities - A dictionary of capabilities to request when** starting the browser session. Required parameter.
- **browser_profile - A selenium.webdriver.firefox.firefox_profile.FirefoxProfile object.** Only used if Firefox is requested. Optional.
- **proxy - A selenium.webdriver.common.proxy.Proxy object. The browser session will** be started with given proxy settings, if possible. Optional.
- **keep_alive - Whether to configure remote_connection.RemoteConnection to use** HTTP keep-alive. Defaults to True.
- **file_detector - Pass custom file detector object during instantiation. If None,** then default LocalFileDetector() will be used.
- options - instance of a driver options.Options class

add_cookie(*cookie_dict*) → None

Adds a cookie to your current session.

Args

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

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}) driver.add_cookie({'name': 'foo', 'value': 'bar', 'sameSite': 'Strict'})

add_credential(*credential: selenium.webdriver.common.virtual_authenticator.Credential*) → None

Injects a credential into the authenticator.

add_virtual_authenticator(*options: selenium.webdriver.common.virtual_authenticator.VirtualAuthenticatorOptions*) → None

Adds a virtual authenticator with the given options.

back() → None

Goes one step backward in the browser history.

Usage

```
driver.back()
```

bidi_connection()

close() → None

Closes the current window.

Usage

```
driver.close()
```

create_web_element(*element_id: str*) → selenium.webdriver.remote.webelement.WebElement

Creates a web element with the specified *element_id*.

delete_all_cookies() → None

Delete all cookies in the scope of the session.

Usage

```
driver.delete_all_cookies()
```

delete_cookie(*name*) → None

Deletes a single cookie with the given name.

Usage

```
driver.delete_cookie('my_cookie')
```

execute(*driver_command: str*, *params: dict = None*) → dict

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: str*, **args*)

Asynchronously Executes JavaScript in the current window/frame.

Args

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

Usage

```
script = "var callback = arguments[arguments.length - 1]; " \
         "window.setTimeout(function(){ callback('timeout') },␣
↪3000);"
driver.execute_async_script(script)
```

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('return document.title;')
```

file_detector_context (*file_detector_class*, **args*, ***kwargs*)
Overrides the current file detector (if necessary) in limited context. Ensures the original file detector is set afterwards.

Example:

with webdriver.file_detector_context(UselessFileDetector): someinput.send_keys('/etc/hosts')

Args

- **file_detector_class - Class of the desired file detector. If the class is different** from the current file_detector, then the class is instantiated with args and kwargs and used as a file detector during the duration of the context manager.
- **args - Optional arguments that get passed to the file detector class during** instantiation.
- kwargs - Keyword arguments, passed the same way as args.

find_element (*by='id'*, *value: Optional[str] = None*) → selenium.webdriver.remote.webelement.WebElement
Find an element given a By strategy and locator.

Usage

```
element = driver.find_element(By.ID, 'foo')
```

Return type *WebElement*

find_elements (*by='id'*, *value: Optional[str] = None*) → List[selenium.webdriver.remote.webelement.WebElement]
Find elements given a By strategy and locator.

Usage

```
elements = driver.find_elements(By.CLASS_NAME, 'foo')
```

Return type list of WebElement

forward () → None
Goes one step forward in the browser history.

Usage

```
driver.forward()
```

fullscreen_window() → None
Invokes the window manager-specific 'full screen' operation.

get(*url: str*) → None
Loads a web page in the current browser session.

get_cookie(*name*) → Optional[Dict[KT, VT]]
Get a single cookie by name. Returns the cookie if found, None if not.

Usage

```
driver.get_cookie('my_cookie')
```

get_cookies() → List[dict]
Returns a set of dictionaries, corresponding to cookies visible in the current session.

Usage

```
driver.get_cookies()
```

get_credentials() → List[selenium.webdriver.common.virtual_authenticator.Credential]
Returns the list of credentials owned by the authenticator.

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_pinned_scripts() → List[str]

get_screenshot_as_base64() → str
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*) → bool
Saves a screenshot of the current window to a PNG image file. 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. This should end with a *.png* extension.

Usage

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

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

Usage

```
driver.get_screenshot_as_png()
```

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

Usage

```
driver.get_window_position()
```

get_window_rect() → dict
Gets the x, y coordinates of the window as well as height and width of the current window.

Usage

```
driver.get_window_rect()
```

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

Usage

```
driver.get_window_size()
```

implicitly_wait(*time_to_wait: float*) → None
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() → None
Maximizes the current window that webdriver is using.

minimize_window() → None
Invokes the window manager-specific 'minimize' operation.

pin_script(*script: str*, *script_key=None*) → selenium.webdriver.remote.script_key.ScriptKey
Store common javascript scripts to be executed later by a unique hashable ID.

print_page(*print_options: Optional[selenium.webdriver.common.print_page_options.PrintOptions] = None*) → str
Takes PDF of the current page.

The driver makes a best effort to return a PDF based on the provided parameters.

quit() → None
Quits the driver and closes every associated window.

Usage

```
driver.quit()
```

refresh() → None
Refreshes the current page.

Usage

```
driver.refresh()
```

remove_all_credentials() → None
Removes all credentials from the authenticator.

remove_credential(*credential_id: Union[str, bytearray]*) → None
Removes a credential from the authenticator.

remove_virtual_authenticator() → None
Removes a previously added virtual authenticator.

The authenticator is no longer valid after removal, so no methods may be called.

save_screenshot(*filename*) → bool
Saves a screenshot of the current window to a PNG image file. 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. This should end with a *.png* extension.

Usage

```
driver.save_screenshot('/Screenshots/foo.png')
```

set_page_load_timeout(*time_to_wait: float*) → None
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: float*) → None
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_user_verified(*verified: bool*) → None
Sets whether the authenticator will simulate success or fail on user verification.

verified: True if the authenticator will pass user verification, False otherwise.

set_window_position(*x*, *y*, *windowHandle: str = 'current'*) → dict
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_rect(*x=None*, *y=None*, *width=None*, *height=None*) → dict
Sets the x, y coordinates of the window as well as height and width of the current window. This method is only supported for W3C compatible browsers; other browsers should use *set_window_position* and *set_window_size*.

Usage

```
driver.set_window_rect(x=10, y=10)
driver.set_window_rect(width=100, height=200)
driver.set_window_rect(x=10, y=10, width=100, height=200)
```

set_window_size(*width*, *height*, *windowHandle: str = 'current'*) → None
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(*capabilities: dict*, *browser_profile=None*) → None
Creates a new session with the desired capabilities.

Args

- capabilities - a capabilities dict to start the session with.
- 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.

unpin(*script_key: selenium.webdriver.remote.script_key.ScriptKey*) → None
Remove a pinned script from storage.

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

capabilities
returns the drivers current capabilities being used.

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.

file_detector

log_types
Gets a list of the available log types. This only works with w3c compliant browsers.

Usage

```
driver.log_types
```

mobile

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

Usage

```
name = 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

Returns

- SwitchTo: an object containing all options to switch focus into

Usage

```
element = driver.switch_to.active_element
alert = driver.switch_to.alert
driver.switch_to.default_content()
driver.switch_to.frame('frame_name')
```

(continues on next page)

(continued from previous page)

```
driver.switch_to.frame(1)
driver.switch_to.frame(driver.find_elements(By.TAG_NAME, "iframe
↪")[0])
driver.switch_to.parent_frame()
driver.switch_to.window('main')
```

timeouts
Get all the timeouts that have been set on the current session.

Usage

:: driver.timeouts

Return type Timeout

title
Returns the title of the current page.

Usage

```
title = driver.title
```

virtual_authenticator_id
Returns the id of the virtual authenticator.

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

Usage

```
driver.window_handles
```

selenium.webdriver.remote.webdriver.**create_matches**(*options: List[selenium.webdriver.common.options.BaseOptions]*) → Dict[KT, VT]

selenium.webdriver.remote.webdriver.**get_remote_connection**(*capabilities*, *command_executor*, *keep_alive*, *ignore_local_proxy=False*)

selenium.webdriver.remote.webdriver.**import_cdp**()

7.21 Remote WebDriver WebElement

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

Abstract Base Class for WebElement.

ABC's will allow custom types to be registered as a WebElement to pass type checks.

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

Represents a DOM element.

Generally, all interesting operations that interact with a document will be performed through this interface.

All method calls will do a freshness check to ensure that the element reference is still valid. This essentially determines whether the element is still attached to the DOM. If this test fails, then an `StaleElementReferenceException` is thrown, and all future calls to this instance will fail.

__init__ (*parent*, *id_*) → None
Initialize self. See help(type(self)) for accurate signature.

clear () → None
Clears the text if it's a text entry element.

click () → None
Clicks the element.

find_element (*by='id'*, *value=None*) → selenium.webdriver.remote.webelement.WebElement
Find an element given a By strategy and locator.

Usage

```
element = element.find_element(By.ID, 'foo')
```

Return type *WebElement*

find_elements (*by='id'*, *value=None*) → List[selenium.webdriver.remote.webelement.WebElement]
Find elements given a By strategy and locator.

Usage

```
element = element.find_elements(By.CLASS_NAME, 'foo')
```

Return type list of WebElement

get_attribute (*name*) → str
Gets the given attribute or property of the element.

This method will first try to return the value of a property with the given name. If a property with that name doesn't exist, it returns the value of the attribute with the same name. If there's no attribute with that name, `None` is returned.

Values which are considered truthy, that is equals "true" or "false", are returned as booleans. All other non-`None` values are returned as strings. For attributes or properties which do not exist, `None` is returned.

To obtain the exact value of the attribute or property, use `get_dom_attribute()` or `get_property()` methods respectively.

Args

- name - Name of the attribute/property to retrieve.

Example:

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

get_dom_attribute (*name*) → str
Gets the given attribute of the element. Unlike `get_attribute()`, this method only returns attributes declared in the element's HTML markup.

Args

- name - Name of the attribute to retrieve.

Usage

```
text_length = target_element.get_dom_attribute("class")
```

get_property(*name*) → str | bool | WebElement | dict
Gets the given property of the element.

Args

- name - Name of the property to retrieve.

Usage

```
text_length = target_element.get_property("text_length")
```

is_displayed() → bool
Whether the element is visible to a user.

is_enabled() → bool
Returns whether the element is enabled.

is_selected() → bool
Returns whether the element is selected.

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

screenshot(*filename*) → bool
Saves a screenshot of the current element to a PNG image file. 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. This should end with a *.png* extension.

Usage

```
element.screenshot('/Screenshots/foo.png')
```

send_keys(**value*) → None
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*) → str
The value of a CSS property.

accessible_name
Returns the ARIA Level of the current webelement.

aria_role
Returns the ARIA role of the current web element.

id
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
The location of the element in the renderable canvas.

location_once_scrolled_into_view
THIS PROPERTY MAY 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
Internal reference to the WebDriver instance this element was found from.

rect
A dictionary with the size and location of the element.

screenshot_as_base64
Gets the screenshot of the current element as a base64 encoded string.

Usage

```
img_b64 = element.screenshot_as_base64
```

screenshot_as_png
Gets the screenshot of the current element as a binary data.

Usage

```
element_png = element.screenshot_as_png
```

shadow_root
Returns a shadow root of the element if there is one or an error. Only works from Chromium 96 and Firefox 96 onwards. Previous versions of browsers will throw an assertion exception.

Returns

- ShadowRoot object or
- NoSuchShadowRoot - if no shadow root was attached to element

size
The size of the element.

tag_name
This element's `tagName` property.

text
The text of the element.

7.22 Remote WebDriver Command

class selenium.webdriver.remote.command.**Command**
Bases: object

Defines constants for the standard WebDriver commands.

While these constants have no meaning in and of themselves, they are used to marshal commands through a service that implements WebDriver's remote wire protocol:

https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

ADD_COOKIE = 'addCookie'

ADD_CREDENTIAL = 'addCredential'

ADD_VIRTUAL_AUTHENTICATOR = 'addVirtualAuthenticator'

CLEAR_ELEMENT = 'clearElement'

CLICK_ELEMENT = 'clickElement'

CLOSE = 'close'

CONTEXT_HANDLES = 'getContextHandles'

CURRENT_CONTEXT_HANDLE = 'getCurrentContextHandle'

DELETE_ALL_COOKIES = 'deleteAllCookies'

DELETE_COOKIE = 'deleteCookie'

DELETE_SESSION = 'deleteSession'

ELEMENT_SCREENSHOT = 'elementScreenshot'

EXECUTE_ASYNC_SCRIPT = 'executeAsyncScript'

FIND_CHILD_ELEMENT = 'findChildElement'

FIND_CHILD_ELEMENTS = 'findChildElements'

FIND_ELEMENT = 'findElement'

FIND_ELEMENTS = 'findElements'

FIND_ELEMENTS_FROM_SHADOW_ROOT = 'findElementsFromShadowRoot'

FIND_ELEMENT_FROM_SHADOW_ROOT = 'findElementFromShadowRoot'

FULLSCREEN_WINDOW = 'fullscreenWindow'

GET = 'get'

GET_ALL_COOKIES = 'getCookies'

GET_AVAILABLE_LOG_TYPES = 'getAvailableLogTypes'

GET_COOKIE = 'getCookie'

GET_CREDENTIALS = 'getCredentials'

GET_CURRENT_URL = 'getCurrentUrl'

```
    GET_ELEMENT_ARIA_LABEL = 'getElementAriaLabel'
    GET_ELEMENT_ARIA_ROLE = 'getElementAriaRole'
    GET_ELEMENT_ATTRIBUTE = 'getElementAttribute'
    GET_ELEMENT_PROPERTY = 'getElementProperty'
    GET_ELEMENT_RECT = 'getElementRect'
    GET_ELEMENT_TAG_NAME = 'getElementTagName'
    GET_ELEMENT_TEXT = 'getElementText'
    GET_ELEMENT_VALUE_OF_CSS_PROPERTY = 'getElementValueOfCssProperty'
    GET_LOG = 'getLog'
    GET_NETWORK_CONNECTION = 'getNetworkConnection'
    GET_PAGE_SOURCE = 'getPageSource'
    GET_SCREEN_ORIENTATION = 'getScreenOrientation'
    GET_SHADOW_ROOT = 'getShadowRoot'
    GET_TIMEOUTS = 'getTimeouts'
    GET_TITLE = 'getTitle'
    GET_WINDOW_RECT = 'getWindowRect'
    GO_BACK = 'goBack'
    GO_FORWARD = 'goForward'
    IS_ELEMENT_ENABLED = 'isElementEnabled'
    IS_ELEMENT_SELECTED = 'isElementSelected'
    MINIMIZE_WINDOW = 'minimizeWindow'
    NEW_SESSION = 'newSession'
    NEW_WINDOW = 'newWindow'
    PRINT_PAGE = 'printPage'
    QUIT = 'quit'
    REFRESH = 'refresh'
    REMOVE_ALL_CREDENTIALS = 'removeAllCredentials'
    REMOVE_CREDENTIAL = 'removeCredential'
    REMOVE_VIRTUAL_AUTHENTICATOR = 'removeVirtualAuthenticator'
    SCREENSHOT = 'screenshot'
    SEND_KEYS_TO_ELEMENT = 'sendKeysToElement'
    SET_NETWORK_CONNECTION = 'setNetworkConnection'
    SET_SCREEN_ORIENTATION = 'setScreenOrientation'
    SET_TIMEOUTS = 'setTimeouts'
    SET_USER_VERIFIED = 'setUserVerified'
    SET_WINDOW_RECT = 'setWindowRect'
```

```
SWITCH_TO_CONTEXT = 'switchToContext'
SWITCH_TO_FRAME = 'switchToFrame'
SWITCH_TO_PARENT_FRAME = 'switchToParentFrame'
SWITCH_TO_WINDOW = 'switchToWindow'
UPLOAD_FILE = 'uploadFile'
W3C_ACCEPT_ALERT = 'w3cAcceptAlert'
W3C_ACTIONS = 'actions'
W3C_CLEAR_ACTIONS = 'clearActionState'
W3C_DISMISS_ALERT = 'w3cDismissAlert'
W3C_EXECUTE_SCRIPT = 'w3cExecuteScript'
W3C_EXECUTE_SCRIPT_ASYNC = 'w3cExecuteScriptAsync'
W3C_GET_ACTIVE_ELEMENT = 'w3cGetActiveElement'
W3C_GET_ALERT_TEXT = 'w3cGetAlertText'
W3C_GET_CURRENT_WINDOW_HANDLE = 'w3cGetCurrentWindowHandle'
W3C_GET_WINDOW_HANDLES = 'w3cGetWindowHandles'
W3C_MAXIMIZE_WINDOW = 'w3cMaximizeWindow'
W3C_SET_ALERT_VALUE = 'w3cSetAlertValue'
```

7.23 Remote WebDriver Error Handler

class selenium.webdriver.remote.errorhandler.**ErrorCode**

Bases: object

Error codes defined in the WebDriver wire protocol.

```
ELEMENT_CLICK_INTERCEPTED = [64, 'element click intercepted']
ELEMENT_IS_NOT_SELECTABLE = [15, 'element not selectable']
ELEMENT_NOT_INTERACTABLE = [60, 'element not interactable']
ELEMENT_NOT_VISIBLE = [11, 'element not visible']
IME_ENGINE_ACTIVATION_FAILED = [31, 'ime engine activation failed']
IME_NOT_AVAILABLE = [30, 'ime not available']
INSECURE_CERTIFICATE = ['insecure certificate']
INVALID_ARGUMENT = [61, 'invalid argument']
INVALID_COOKIE_DOMAIN = [24, 'invalid cookie domain']
INVALID_COORDINATES = ['invalid coordinates']
INVALID_ELEMENT_COORDINATES = [29, 'invalid element coordinates']
INVALID_ELEMENT_STATE = [12, 'invalid element state']
INVALID_SELECTOR = [32, 'invalid selector']
INVALID_SESSION_ID = ['invalid session id']
```

```
INVALID_XPATH_SELECTOR = [51, 'invalid selector']
INVALID_XPATH_SELECTOR_RETURN_TYPER = [52, 'invalid selector']
JAVASCRIPT_ERROR = [17, 'javascript error']
METHOD_NOT_ALLOWED = [405, 'unsupported operation']
MOVE_TARGET_OUT_OF_BOUNDS = [34, 'move target out of bounds']
NO_ALERT_OPEN = [27, 'no such alert']
NO_SUCH_COOKIE = [62, 'no such cookie']
NO_SUCH_ELEMENT = [7, 'no such element']
NO_SUCH_FRAME = [8, 'no such frame']
NO_SUCH_SHADOW_ROOT = ['no such shadow root']
NO_SUCH_WINDOW = [23, 'no such window']
SCRIPT_TIMEOUT = [28, 'script timeout']
SESSION_NOT_CREATED = [33, 'session not created']
STALE_ELEMENT_REFERENCE = [10, 'stale element reference']
SUCCESS = 0
TIMEOUT = [21, 'timeout']
UNABLE_TO_CAPTURE_SCREEN = [63, 'unable to capture screen']
UNABLE_TO_SET_COOKIE = [25, 'unable to set cookie']
UNEXPECTED_ALERT_OPEN = [26, 'unexpected alert open']
UNKNOWN_COMMAND = [9, 'unknown command']
UNKNOWN_ERROR = [13, 'unknown error']
UNKNOWN_METHOD = ['unknown method exception']
XPATH_LOOKUP_ERROR = [19, 'invalid selector']
```

class selenium.webdriver.remote.errorhandler.**ErrorHandler**

Bases: object

Handles errors returned by the WebDriver server.

check_response(*response: Dict[str, Any]*) → None

Checks that a JSON response from the WebDriver does not have an error.

Args

- response - The JSON response from the WebDriver server as a dictionary object.

Raises If the response contains an error message.

7.24 Remote WebDriver Mobile

class selenium.webdriver.remote.mobile.**Mobile**(*driver*)

Bases: object

class ConnectionType(*mask*)

Bases: object

__init__(*mask*)
Initialize self. See help(type(self)) for accurate signature.

airplane_mode

data

wifi

__init__(*driver*)
Initialize self. See help(type(self)) for accurate signature.

set_network_connection(*network*)
Set the network connection for the remote device.

Example of setting airplane mode:

```
driver.mobile.set_network_connection(driver.mobile.AIRPLANE_MODE)
```

AIRPLANE_MODE = <selenium.webdriver.remote.mobile.Mobile.ConnectionType object>

ALL_NETWORK = <selenium.webdriver.remote.mobile.Mobile.ConnectionType object>

DATA_NETWORK = <selenium.webdriver.remote.mobile.Mobile.ConnectionType object>

WIFI_NETWORK = <selenium.webdriver.remote.mobile.Mobile.ConnectionType object>

context
returns the current context (Native or WebView).

contexts
returns a list of available contexts.

network_connection

7.25 Remote WebDriver Remote Connection

class selenium.webdriver.remote.remote_connection.**RemoteConnection**(*remote_server_addr: str*, *keep_alive: bool = False*, *ignore_proxy: bool = False*)

Bases: object

A connection with the Remote WebDriver server.

Communicates with the server using the WebDriver wire protocol: https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

__init__(*remote_server_addr: str*, *keep_alive: bool = False*, *ignore_proxy: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

close()
Clean up resources when finished with the remote_connection.

execute(*command*, *params*)
Send a command to the remote server.

Any path substitutions required for the URL mapped to the command should be included in the command parameters.

Args

- command - A string specifying the command to execute.
- params - A dictionary of named parameters to send with the command as its JSON payload.

classmethod get_certificate_bundle_path()

Returns Paths of the .pem encoded certificate to verify connection to command executor

classmethod get_remote_connection_headers(*parsed_url*, *keep_alive=False*)
Get headers for remote request.

Args

- parsed_url - The parsed url
- keep_alive (Boolean) - Is this a keep-alive connection (default: False)

classmethod get_timeout()

Returns Timeout value in seconds for all http requests made to the Remote Connection

classmethod reset_timeout()
Reset the http request timeout to socket._GLOBAL_DEFAULT_TIMEOUT.

classmethod set_certificate_bundle_path(*path*)
Set the path to the certificate bundle to verify connection to command executor. Can also be set to None to disable certificate validation.

Args

- path - path of a .pem encoded certificate chain.

classmethod set_timeout(*timeout*)
Override the default timeout.

Args

- timeout - timeout value for http requests in seconds

browser_name = None

7.26 Remote WebDriver Utils

selenium.webdriver.remote.utils.**dump_json**(*json_struct: Any*) → str

selenium.webdriver.remote.utils.**load_json**(*s: Union[str, bytes]*) → Any

7.27 Internet Explorer WebDriver

class selenium.webdriver.ie.webdriver.**WebDriver**(*executable_path='IEDriverServer.exe'*, *capabilities=None*, *port=0*, *timeout=30*, *host=None*, *log_level=None*, *service_log_path=None*, *options: selenium.webdriver.ie.options.Options = None*, *service: selenium.webdriver.ie.service.Service = None*, *desired_capabilities=None*, *keep_alive=None*)

Bases: *selenium.webdriver.remote.webdriver.WebDriver*

Controls the IEServerDriver and allows you to drive Internet Explorer.

__init__(*executable_path='IEDriverServer.exe'*, *capabilities=None*, *port=0*, *timeout=30*, *host=None*, *log_level=None*, *service_log_path=None*, *options: selenium.webdriver.ie.options.Options = None*, *service: selenium.webdriver.ie.service.Service = None*, *desired_capabilities=None*, *keep_alive=None*) → None

Creates a new instance of the Ie driver.

Starts the service and then creates new instance of Ie driver.

Args

- executable_path - Deprecated: path to the executable. If the default is used it assumes the executable is in the $PATH
- capabilities - Deprecated: capabilities Dictionary object
- port - Deprecated: port you would like the service to run, if left as 0, a free port will be found.
- timeout - Deprecated: no longer used, kept for backward compatibility
- host - Deprecated: IP address for the service
- log_level - Deprecated: log level you would like the service to run.
- service_log_path - Deprecated: target of logging of service, may be "stdout", "stderr" or file path.
- options - IE Options instance, providing additional IE options
- desired_capabilities - Deprecated: alias of capabilities; this will make the signature consistent with RemoteWebDriver.
- keep_alive - Deprecated: Whether to configure RemoteConnection to use HTTP keep-alive.

create_options() → selenium.webdriver.ie.options.Options

quit() → None

Quits the driver and closes every associated window.

Usage

```
driver.quit()
```

7.28 Android WebDriver

7.29 Opera WebDriver

7.30 PhantomJS WebDriver

7.31 PhantomJS WebDriver Service

7.32 Safari WebDriver

class selenium.webdriver.safari.webdriver.**WebDriver**(*port=0*, *executable_path='/usr/bin/safaridriver'*, *reuse_service=False*, *desired_capabilities={'browserName': 'safari', 'platformName': 'mac'}*, *quiet=False*, *keep_alive=True*, *service_args=None*, *options: selenium.webdriver.safari.options.Options = None*, *service: selenium.webdriver.safari.service.Service = None*)

Bases: *selenium.webdriver.remote.webdriver.WebDriver*

Controls the SafariDriver and allows you to drive the browser.

__init__(*port=0*, *executable_path='/usr/bin/safaridriver'*, *reuse_service=False*, *desired_capabilities={'browserName': 'safari', 'platformName': 'mac'}*, *quiet=False*, *keep_alive=True*, *service_args=None*, *options: selenium.webdriver.safari.options.Options = None*, *service: selenium.webdriver.safari.service.Service = None*) → None

Creates a new Safari driver instance and launches or finds a running safaridriver service.

Args

- port - The port on which the safaridriver service should listen for new connections. If zero, a free port will be found.
- executable_path - Path to a custom safaridriver executable to be used. If absent, /usr/bin/safaridriver is used.
- reuse_service - If True, do not spawn a safaridriver instance; instead, connect to an already-running service that was launched externally.
- desired_capabilities: Dictionary object with desired capabilities (Can be used to provide various Safari switches).
- quiet - If True, the driver's stdout and stderr is suppressed.
- **keep_alive - Whether to configure SafariRemoteConnection to use** HTTP keep-alive. Defaults to True.
- service_args : List of args to pass to the safaridriver service
- service - Service object for handling the browser driver if you need to pass extra details

debug()

get_permission(*permission*)

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

set_permission(*permission*, *value*)

7.33 Safari WebDriver Service

class selenium.webdriver.safari.service.**Service**(*executable_path: str = '/usr/bin/safaridriver', port: int = 0, quiet: bool = False, service_args: Optional[List[str]] = None, env: Optional[Mapping[str, str]] = None, **kwargs*)
Bases: *selenium.webdriver.common.service.Service*

A Service class that is responsible for the starting and stopping of *safaridriver* This is only supported on MAC OSX.

Parameters

- **executable_path** – install path of the safaridriver executable, defaults to */usr/bin/safaridriver*.
- **port** – Port for the service to run on, defaults to 0 where the operating system will decide.
- **quiet** – Suppress driver stdout & stderr, redirects to os.devnull if enabled.
- **service_args** – (Optional) List of args to be passed to the subprocess when launching the executable.
- **env** – (Optional) Mapping of environment variables for the new process, defaults to *os.environ*.

__init__(*executable_path: str = '/usr/bin/safaridriver', port: int = 0, quiet: bool = False, service_args: Optional[List[str]] = None, env: Optional[Mapping[str, str]] = None, **kwargs*) → None
Initialize self. See help(type(self)) for accurate signature.

command_line_args() → List[str]
A List of program arguments (excluding the executable).

service_url
Gets the url of the SafariDriver Service.

7.34 Select Support

class selenium.webdriver.support.select.**Select**(*webelement*)
Bases: object

__init__(*webelement*) → None
Constructor. A check is made that the given element is, indeed, a SELECT tag. If it is not, then an UnexpectedTagNameException is thrown.

Args

- webelement - SELECT element to wrap

Example: from selenium.webdriver.support.ui import Select

Select(driver.find_element(By.TAG_NAME, "select")).select_by_index(2)

deselect_all() → None
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 examining the "index" attribute of an element, and not merely by counting.

Args

- index - The option at this index will be deselected

throws NoSuchElementException If there is no option with specified index in SELECT

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

throws NoSuchElementException If there is no option with specified value in SELECT

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 examining the "index" attribute of an element, and not merely by counting.

Args

- index - The option at this index will be selected

throws NoSuchElementException If there is no option with specified index in SELECT

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

throws NoSuchElementException If there is no option with specified value in SELECT

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

throws NoSuchElementException If there is no option with specified text in SELECT

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.

7.35 Wait Support

class selenium.webdriver.support.wait.**WebDriverWait**(*driver*, *timeout: float*, *poll_frequency: float = 0.5*, *ignored_exceptions: Optional[Iterable[Type[Exception]]] = None*)

Bases: object

__init__(*driver*, *timeout: float*, *poll_frequency: float = 0.5*, *ignored_exceptions: Optional[Iterable[Type[Exception]]] = None*)
Constructor, takes a WebDriver instance and timeout in seconds.

Args

- driver - Instance of WebDriver (Ie, Firefox, Chrome or Remote)
- timeout - Number of seconds before timing out
- poll_frequency - sleep interval between calls By default, it is 0.5 second.
- ignored_exceptions - iterable structure of exception classes ignored during calls. By default, it contains NoSuchElementException only.

Example:

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

element = WebDriverWait(driver, 10).until(lambda x: x.find_element(By.ID,
↪"someId"))

is_disappeared = WebDriverWait(driver, 30, 1, (ElementNotVisibleException)).\

            until_not(lambda x: x.find_element(By.ID, "someId").is_
↪displayed())
```

until(*method*, *message: str = ''*)
Calls the method provided with the driver as an argument until the return value does not evaluate to False.

Parameters

- **method** – callable(WebDriver)
- **message** – optional message for `TimeoutException`

Returns the result of the last call to *method*

Raises *selenium.common.exceptions.TimeoutException* if timeout occurs

until_not(*method*, *message: str = ''*)

Calls the method provided with the driver as an argument until the return value evaluates to `False`.

Parameters

- **method** – callable(WebDriver)
- **message** – optional message for `TimeoutException`

Returns the result of the last call to *method*, or `True` if *method* has raised one of the ignored exceptions

Raises *selenium.common.exceptions.TimeoutException* if timeout occurs

7.36 Color Support

class selenium.webdriver.support.color.**Color**(*red: Any*, *green: Any*, *blue: Any*, *alpha: Any = 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)
```

__init__(*red: Any*, *green: Any*, *blue: Any*, *alpha: Any = 1*) → None

Initialize self. See help(type(self)) for accurate signature.

classmethod from_string(*str_: str*) → selenium.webdriver.support.color.Color

hex

rgb

rgba

7.37 Event Firing WebDriver Support

class selenium.webdriver.support.event_firing_webdriver.**EventFiringWebDriver**(*driver: se-le-nium.webdriver.rem event_listener: se-le-nium.webdriver.supp*

Bases: object

A wrapper around an arbitrary WebDriver instance which supports firing events.

__init__(*driver: selenium.webdriver.remote.webdriver.WebDriver*, *event_listener: selenium.webdriver.support.abstract_event_listener.AbstractEventListener*) → None

Creates a new instance of the EventFiringWebDriver.

Args

- driver : A WebDriver instance
- event_listener : Instance of a class that subclasses AbstractEventListener and implements it fully or partially

Example:

```
from selenium.webdriver import Firefox
from selenium.webdriver.support.events import EventFiringWebDriver,␣
↪AbstractEventListener

class MyListener(AbstractEventListener):
    def before_navigate_to(self, url, driver):
        print("Before navigate to %s" % url)
    def after_navigate_to(self, url, driver):
        print("After navigate to %s" % url)

driver = Firefox()
ef_driver = EventFiringWebDriver(driver, MyListener())
ef_driver.get("http://www.google.co.in/")
```

back() → None

close() → None

execute_async_script(*script*, **args*)

execute_script(*script*, **args*)

find_element(*by='id'*, *value=None*) → selenium.webdriver.remote.webelement.WebElement

find_elements(*by='id'*, *value=None*) → List[selenium.webdriver.remote.webelement.WebElement]

forward() → None

get(*url: str*) → None

quit() → None

wrapped_driver

Returns the WebDriver instance wrapped by this EventsFiringWebDriver.

class selenium.webdriver.support.event_firing_webdriver.**EventFiringWebElement**(*webelement: selenium.webdriver.remote.webelement.WebElement*, *ef_driver: selenium.webdriver.support.event_firing_webdriver.EventFiringWebDriver*)

Bases: object

A wrapper around WebElement instance which supports firing events.

__init__(*webelement: selenium.webdriver.remote.webelement.WebElement*, *ef_driver: selenium.webdriver.support.event_firing_webdriver.EventFiringWebDriver*) → None
Creates a new instance of the EventFiringWebElement.

clear() → None

click() → None

find_element(*by='id'*, *value=None*) → selenium.webdriver.remote.webelement.WebElement

find_elements(*by='id'*, *value=None*) → List[selenium.webdriver.remote.webelement.WebElement]

send_keys(**value*) → None

wrapped_element
Returns the WebElement wrapped by this EventFiringWebElement instance.

7.38 Abstract Event Listener Support

class selenium.webdriver.support.abstract_event_listener.**AbstractEventListener**
Bases: object

Event listener must subclass and implement this fully or partially.

after_change_value_of(*element*, *driver*) → None

after_click(*element*, *driver*) → None

after_close(*driver*) → None

after_execute_script(*script*, *driver*) → None

after_find(*by*, *value*, *driver*) → None

after_navigate_back(*driver*) → None

after_navigate_forward(*driver*) → None

after_navigate_to(*url: str*, *driver*) → None

after_quit(*driver*) → None

before_change_value_of(*element*, *driver*) → None

before_click(*element*, *driver*) → None

before_close(*driver*) → None

before_execute_script(*script*, *driver*) → None

before_find(*by*, *value*, *driver*) → None

before_navigate_back(*driver*) → None

before_navigate_forward(*driver*) → None

before_navigate_to(*url: str*, *driver*) → None

before_quit(*driver*) → None

on_exception(*exception*, *driver*) → None

7.39 Expected conditions Support

selenium.webdriver.support.expected_conditions.**alert_is_present**()
An expectation for checking if an alert is currently present and switching to it.

selenium.webdriver.support.expected_conditions.**all_of**(**expected_conditions*)
An expectation that all of multiple expected conditions is true.

Equivalent to a logical 'AND'. Returns: When any ExpectedCondition is not met: False. When all ExpectedConditions are met: A List with each ExpectedCondition's return value.

selenium.webdriver.support.expected_conditions.**any_of**(**expected_conditions*)
An expectation that any of multiple expected conditions is true.

Equivalent to a logical 'OR'. Returns results of the first matching condition, or False if none do.

selenium.webdriver.support.expected_conditions.**element_attribute_to_include**(*locator*, *attribute_*)
An expectation for checking if the given attribute is included in the specified element.

locator, attribute

selenium.webdriver.support.expected_conditions.**element_located_selection_state_to_be**(*locator*, *is_selec*
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

selenium.webdriver.support.expected_conditions.**element_located_to_be_selected**(*locator*)
An expectation for the element to be located is selected.

locator is a tuple of (by, path)

selenium.webdriver.support.expected_conditions.**element_selection_state_to_be**(*element*, *is_selected*)
An expectation for checking if the given element is selected.

element is WebElement object is_selected is a Boolean.

selenium.webdriver.support.expected_conditions.**element_to_be_clickable**(*mark*)
An Expectation for checking an element is visible and enabled such that you can click it.

element is either a locator (text) or an WebElement

selenium.webdriver.support.expected_conditions.**element_to_be_selected**(*element*)
An expectation for checking the selection is selected.

element is WebElement object

selenium.webdriver.support.expected_conditions.**frame_to_be_available_and_switch_to_it**(*locato*
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.

selenium.webdriver.support.expected_conditions.**invisibility_of_element**(*element*)
An Expectation for checking that an element is either invisible or not present on the DOM.

element is either a locator (text) or an WebElement

selenium.webdriver.support.expected_conditions.**invisibility_of_element_located**(*locator*)
An Expectation for checking that an element is either invisible or not present on the DOM.

locator used to find the element

selenium.webdriver.support.expected_conditions.**new_window_is_opened**(*current_handles*)
An expectation that a new window will be opened and have the number of windows handles increase.

selenium.webdriver.support.expected_conditions.**none_of**(**expected_conditions*)
An expectation that none of 1 or multiple expected conditions is true.

Equivalent to a logical 'NOT-OR'. Returns a Boolean

selenium.webdriver.support.expected_conditions.**number_of_windows_to_be**(*num_windows*)
An expectation for the number of windows to be a certain value.

selenium.webdriver.support.expected_conditions.**presence_of_all_elements_located**(*locator*)
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

selenium.webdriver.support.expected_conditions.**presence_of_element_located**(*locator*)
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

selenium.webdriver.support.expected_conditions.**staleness_of**(*element*)
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.

selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element**(*locator*, *text_*)
An expectation for checking if the given text is present in the specified element.

locator, text

selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element_attribute**(*locat*
at-
tribu
*text_*
An expectation for checking if the given text is present in the element's attribute.

locator, attribute, text

selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element_value**(*locator*, *text_*)
An expectation for checking if the given text is present in the element's value.

locator, text

selenium.webdriver.support.expected_conditions.**title_contains**(*title: str*)
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

selenium.webdriver.support.expected_conditions.**title_is**(*title: str*)
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.

selenium.webdriver.support.expected_conditions.**url_changes**(*url: str*)
An expectation for checking the current url.

url is the expected url, which must not be an exact match returns True if the url is different, false otherwise.

selenium.webdriver.support.expected_conditions.**url_contains**(*url: str*)
An expectation for checking that the current url contains a case- sensitive substring.

url is the fragment of url expected, returns True when the url matches, False otherwise

selenium.webdriver.support.expected_conditions.**url_matches**(*pattern: str*)
An expectation for checking the current url.

pattern is the expected pattern. This finds the first occurrence of pattern in the current url and as such does not require an exact full match.

selenium.webdriver.support.expected_conditions.**url_to_be**(*url: str*)
An expectation for checking the current url.

url is the expected url, which must be an exact match returns True if the url matches, false otherwise.

selenium.webdriver.support.expected_conditions.**visibility_of**(*element*)
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

selenium.webdriver.support.expected_conditions.**visibility_of_all_elements_located**(*locator*)
An expectation for checking that all elements are present on the DOM of a page and visible. Visibility means that the elements are not only displayed but also has a height and width that is greater than 0.

locator - used to find the elements returns the list of WebElements once they are located and visible

selenium.webdriver.support.expected_conditions.**visibility_of_any_elements_located**(*locator*)
An expectation for checking that there is at least one element visible on a web page.

locator is used to find the element returns the list of WebElements once they are located

selenium.webdriver.support.expected_conditions.**visibility_of_element_located**(*locator*)
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://github.com/SeleniumHQ/selenium/wiki/Frequently-Asked-Questions

8.1 How to use ChromeDriver ?

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

```
unzip chromedriver_linux64.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

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Z