Selenium Python Bindings

Release 2

Baiju Muthukadan

August 29, 2013

CONTENTS

Author Baiju Muthukadan

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

CHAPTER

ONE

INSTALLATION

1.1 Introduction

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

Selenium Python bindings provide a convenient API to access Selenium WebDrivers like Firefox, Ie, Chrome, Remote etc.. The current supported Python versions are 2.6, 2.7, 3.2 and 3.3.

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

1.2 Downloading Python bindings for Selenium

You can download Python bindings for Selenium from the PyPI page for selenium package. You can also use easy_install or pip to install the bindings:

```
easy_install selenium
```

or:

```
pip install selenium
```

You may consider using virtualenv to create isolated Python environments.

1.3 Detailed instructions for Windows users

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

1. Install Python 2.7 using the MSI available in python.org download page.
2. Create a folder named `C:\seltests` and download virtualenv.py script into that folder.

   If you have downloaded and saved the program properly, please make sure `virtualenv.py` file exists at this location in your system: `C:\seltests\virtualenv.py`
3. Start a command prompt (using the `cmd.exe` program), then change to the `C:\seltests` folder and run the `virtualenv.py` script as given below.

```
C:
cd C:\seltests
C:\Python27\python.exe virtualenv.py selenv
```

This step will create a folder named `C:\seltests\selenv` which contains a virtual Python.

4. Use the `pip` command as given below to install *selenium*

```
C:\seltests\selenv\Scripts\pip.exe install selenium
```

Now installation has been completed! You can proceed to test your Selenium scripts.

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

```
C:\seltests\selenv\Scripts\python.exe C:\seltests\my_selenium_script.py
```

1.4 Downloading Selenium server

Note: The Selenium server is only required, if you want to use the remote WebDriver. See the *Using Selenium with remote WebDriver* section for more details.

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

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

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

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

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

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

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

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

CHAPTER

TWO

GETTING STARTED

2.1 Simple Usage

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

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

driver = webdriver.Firefox()
driver.get("http://www.python.org")
assert "Python" in driver.title
elem = driver.find_element_by_name("q")
elem.send_keys("selenium")
elem.send_keys(Keys.RETURN)
assert "Google" in driver.title
driver.close()
```

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

```
python python_org_search.py
```

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

2.2 Walk through of the example

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

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

Next, the instance of Firefox WebDriver is created.

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

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

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

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

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

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

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

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

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

After submission of the page, you should be reached in the Google site:

```
assert "Google" in driver.title
```

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

```
driver.close()
```

2.3 Using Selenium to write tests

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

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

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

class PythonOrgSearch(unittest.TestCase):

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

    def test_search_in_python_org(self):
        driver = self.driver
        driver.get("http://www.python.org")
        self.assertIn("Python", driver.title)
        elem = driver.find_element_by_name("q")
        elem.send_keys("selenium")
        elem.send_keys(Keys.RETURN)
        self.assertIn("Google", driver.title)

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

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

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

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

OK
```

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

2.4 Walk through of the example

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

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

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

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

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

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

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

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

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

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

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

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

Note: The *assertIn* API is only available in Python 2.7 and above.

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

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

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

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

After submission of the page, you should be reached in the Google site. You can confirm it by asserting "Google" in the title:

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

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

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

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

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

2.5 Using Selenium with remote WebDriver

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

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

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

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

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

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

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

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

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

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

```
driver = webdriver.Remote(
   command_executor='http://127.0.0.1:4444/wd/hub',
   desired_capabilities={'browserName': 'htmlunit',
                         'version': '2',
                        'javascriptEnabled': True})
```

CHAPTER

THREE

NAVIGATING

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

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

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

3.1 Interacting with the page

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

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

you could find it using any of:

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

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

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

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

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

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

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

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

```
element.clear()
```

3.2 Filling in forms

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

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

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

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

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

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

```
element.submit()
```

3.3 Drag and drop

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

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

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

3.4 Moving between windows and frames

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

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

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

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

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

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

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

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

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

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

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

3.5 Popup dialogs

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

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

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

3.6 Navigation: history and location

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

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

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

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

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

3.7 Cookies

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

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

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

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

3.8 Next, next steps!

This has been a high level walkthrough of WebDriver and some of its key capabilities. You may want to look at the *Test Design Considerations* chapter to get some ideas about how you can reduce the pain of maintaining your tests and how to make your code more modular.

CHAPTER

FOUR

LOCATING ELEMENTS

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

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

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

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

4.1 Locating by Id

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

For instance, conside this page source:

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
```

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

The form element can be located like this:

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

4.2 Locating by Name

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

For instance, conside this page source:

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

The username & password elements can be located like this:

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

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

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

4.3 Locating by XPath

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

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

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

For instance, conside this page source:

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

The form elements can be located like this:

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

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

The username element can be located like this:

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

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

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

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

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

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

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

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

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

4.4 Locating Hyperlinks by Link Text

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

For instance, conside 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')
```

CHAPTER

FIVE

WEBDRIVER API

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

This chapter cover all the interfaces of Selenium WebDriver.

Recommended Import Style

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

```
from selenium import webdriver
```

Then, you can access the classes like this:

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

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

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

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

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

Conventions used in the API

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

Here is an example for property:

- current_url

  URL of the current loaded page.

Usage:

```
driver.current_url
```

Here is an example for a method:

- close()

  Closes the current window.

  Usage:

  ```
  driver.close()
  ```

5.1 Exceptions

Exceptions that may happen in all the webdriver code.

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

Bases: selenium.common.exceptions.InvalidElementStateException

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

Bases: selenium.common.exceptions.InvalidElementStateException

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

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

Bases: selenium.common.exceptions.WebDriverException

An error has occurred on the server side.

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

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

Bases: selenium.common.exceptions.WebDriverException

Indicates that activating an IME engine has failed.

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

Bases: selenium.common.exceptions.WebDriverException

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

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

Bases: selenium.common.exceptions.WebDriverException

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

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

Bases: selenium.common.exceptions.WebDriverException

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

Bases: selenium.common.exceptions.NoSuchElementException

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

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

Bases: selenium.common.exceptions.WebDriverException

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

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

Bases: selenium.common.exceptions.WebDriverException

Indicates that the target provided to the actions move() method is invalid

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

Bases: selenium.common.exceptions.WebDriverException

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

Bases: selenium.common.exceptions.WebDriverException

find_element_by_* can't find the element.

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

Bases: selenium.common.exceptions.WebDriverException

find_element_by_* can't find the element.

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

Bases: selenium.common.exceptions.InvalidSwitchToTargetException

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

Bases: selenium.common.exceptions.InvalidSwitchToTargetException

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

Bases: selenium.common.exceptions.WebDriverException

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

Bases: selenium.common.exceptions.WebDriverException

Indicates that a reference to an element is now "stale" — the element no longer appears on the DOM of the page.

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

Bases: selenium.common.exceptions.WebDriverException

Thrown when a command does not complete in enough time.

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

Bases: selenium.common.exceptions.WebDriverException

Thrown when a driver fails to set a cookie.

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

Bases: selenium.common.exceptions.WebDriverException

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

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

Bases: exceptions.Exception

5.2 Action Chains

The ActionChains implementation

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

Bases: object

Generate user actions. All actions are stored in the ActionChains object. Call perform() to fire stored actions.

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 element 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

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

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

Args

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

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

Args

- xoffset: X offset to move to.
- yoffset: Y offset to move to.

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

Args

- to_element: The element to move to.

move_to_element_with_offset(*to_element*, *xoffset*, *yoffset*)
Move the mouse by an offset of the specificed element. Offsets are relative to the top-left corner of the element.

Args

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

perform()
Performs all stored actions.

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

Args

- on_element: The element to mouse up.

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

Args

- keys_to_send: The keys to send.

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.

5.3 Alerts

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

accept()
Accepts the alert available

dismiss()
Dismisses the alert available

send_keys(*keysToSend*)
Send Keys to the Alert

text
Gets the text of the Alert

5.4 Special Keys

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

ADD = u'\ue025'

ALT = u'\ue00a'

ARROW_DOWN = u'\ue015'

ARROW_LEFT = u'\ue012'

ARROW_RIGHT = u'\ue014'

ARROW_UP = u'\ue013'

BACK_SPACE = u'\ue003'

CANCEL = u'\ue001'

CLEAR = u'\ue005'

COMMAND = u'\ue03d'

CONTROL = u'\ue009'

DECIMAL = u'\ue028'

DELETE = u'\ue017'

DIVIDE = u'\ue029'

DOWN = u'\ue015'

END = u'\ue010'

ENTER = u'\ue007'

EQUALS = u'\ue019'

ESCAPE = u'\ue00c'

F1 = u'\ue031'

F10 = u'\ue03a'

F11 = u'\ue03b'

F12 = u'\ue03c'

F2 = u'\ue032'

F3 = u'\ue033'

F4 = u'\ue034'

F5 = u'\ue035'

F6 = u'\ue036'

F7 = u'\ue037'

F8 = u'\ue038'

F9 = u'\ue039'

HELP = u'\ue002'

HOME = u'\ue011'

INSERT = u'\ue016'

LEFT = u'\ue012'

LEFT_ALT = u'\ue00a'

LEFT_CONTROL = u'\ue009'

LEFT_SHIFT = u'\ue008'

META = u'\ue03d'

MULTIPLY = u'\ue024'

NULL = u'\ue000'

NUMPAD0 = u'\ue01a'

NUMPAD1 = u'\ue01b'

NUMPAD2 = u'\ue01c'

NUMPAD3 = u'\ue01d'

NUMPAD4 = u'\ue01e'

NUMPAD5 = u'\ue01f'

NUMPAD6 = u'\ue020'

NUMPAD7 = u'\ue021'

NUMPAD8 = u'\ue022'

NUMPAD9 = u'\ue023'

PAGE_DOWN = u'\ue00f'

PAGE_UP = u'\ue00e'

PAUSE = u'\ue00b'

RETURN = u'\ue006'

RIGHT = u'\ue014'

SEMICOLON = u'\ue018'

SEPARATOR = u'\ue026'

SHIFT = u'\ue008'

SPACE = u'\ue00d'

SUBTRACT = u'\ue027'

TAB = u'\ue004'

UP = u'\ue013'

5.5 Firefox WebDriver

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

Bases: selenium.webdriver.remote.webdriver.WebDriver

create_web_element(*element_id*)

Override from RemoteWebDriver to use firefox.WebElement.

quit()

Quits the driver and close every associated window.

NATIVE_EVENTS_ALLOWED = True

firefox_profile

5.6 Chrome WebDriver

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

Bases: selenium.webdriver.remote.webdriver.WebDriver

Controls the ChromeDriver and allows you to drive the browser.

You will need to download the ChromeDriver executable from http://code.google.com/p/chromedriver/downloads/list

quit()

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

5.7 Remote WebDriver

The WebDriver implementation.

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

Bases: object

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

Attributes

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

add_cookie(*cookie_dict*)

Adds a cookie to your current session.

Args

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

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

back()
Goes one step backward in the browser history.

Usage driver.back()

close()
Closes the current window.

Usage driver.close()

create_web_element(*element_id*)
Creates a web element with the specified element_id.

delete_all_cookies()
Delete all cookies in the scope of the session.

Usage driver.delete_all_cookies()

delete_cookie(*name*)
Deletes a single cookie with the given name.

Usage driver.delete_cookie('my_cookie')

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

Args

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

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

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

Args

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

Usage driver.execute_async_script('document.title')

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

Args

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

Usage driver.execute_script('document.title')

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

Usage Use the corresponding find_element_by_* instead of this.

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

Args

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

Usage driver.find_element_by_class_name('foo')

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

Args

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

Usage driver.find_element_by_css_selector('#foo')

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

Args

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

Usage driver.find_element_by_id('foo')

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

Args

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

Usage driver.find_element_by_link_text('Sign In')

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

Args

- name: The name of the element to find.

Usage driver.find_element_by_name('foo')

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

Args

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

Usage driver.find_element_by_partial_link_text('Sign')

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

Args

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

Usage driver.find_element_by_tag_name('foo')

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

Args

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

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

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

Usage Use the corresponding find_elements_by_* instead of this.

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

Args

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

Usage driver.find_elements_by_class_name('foo')

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

Args

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

Usage driver.find_element_by_css_selector('#foo')

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

Args

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

Usage driver.find_element_by_id('foo')

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

Args

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

Usage driver.find_elements_by_link_text('Sign In')

find_elements_by_name(*name*)
Finds elements by name.

Args

- name: The name of the elements to find.

Usage driver.find_elements_by_name('foo')

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

Args

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

Usage driver.find_element_by_partial_link_text('Sign')

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

Args

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

Usage driver.find_elements_by_tag_name('foo')

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

Args

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

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

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

Usage driver.forward()

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

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

Usage driver.get_cookie('my_cookie')

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

Usage driver.get_cookies()

get_screenshot_as_base64()

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

Usage driver.get_screenshot_as_base64()

get_screenshot_as_file(*filename*)

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

Args

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

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

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

Usage driver.get_window_position()

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

Usage driver.get_window_size()

implicitly_wait(*time_to_wait*)

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

Args

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

Usage driver.implicitly_wait(30)

is_online()
Returns a boolean if the browser is online or offline

maximize_window()
Maximizes the current window that webdriver is using

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

Usage driver.quit()

refresh()
Refreshes the current page.

Usage driver.refresh()

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

set_page_load_timeout(*time_to_wait*)

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

Args

- time_to_wait: The amount of time to wait

Usage driver.set_page_load_timeout(30)

set_script_timeout(*time_to_wait*)

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

Args

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

Usage driver.set_script_timeout(30)

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

Args

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

Usage driver.set_window_position(0,0)

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

Args

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

Usage driver.set_window_size(800,600)

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

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

Args

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

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

switch_to_active_element()
Returns the element with focus, or BODY if nothing has focus.

Usage driver.switch_to_active_element()

switch_to_alert()
Switches focus to an alert on the page.

Usage driver.switch_to_alert()

switch_to_default_content()
Switch focus to the default frame.

Usage driver.switch_to_default_content()

switch_to_frame(*frame_reference*)
Switches focus to the specified frame, by index, name, or webelement.

Args

- **frame_reference: The name of the window to switch to, an integer representing the index,** or a webelement that is an (i)frame to switch to.

Usage driver.switch_to_frame('frame_name') driver.switch_to_frame(1) driver.switch_to_frame(driver.find_elements_by_tag_name("iframe")[0])

switch_to_window(*window_name*)
Switches focus to the specified window.

Args

- window_name: The name or window handle of the window to switch to.

Usage driver.switch_to_window('main')

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

current_url
Gets the URL of the current page.

Usage driver.current_url

current_window_handle
Returns the handle of the current window.

Usage driver.current_window_handle

desired_capabilities
returns the drivers current desired capabilities being used

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

Usage

- driver.name

orientation
Gets the current orientation of the device

Usage orientation = driver.orientation

page_source
Gets the source of the current page.

Usage driver.page_source

title
Returns the title of the current page.

Usage driver.title

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

Usage driver.window_handles

5.8 WebElement

WebElement implementation.

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

classmethod is_local_file(**keys*)

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

Represents an HTML element.

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

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

click()
Clicks the element.

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

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

find_element_by_css_selector(*css_selector*)
Find and return an element by CSS selector.

find_element_by_id(*id_*)
Finds element by id.

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

find_element_by_name(*name*)
Find element by name.

find_element_by_partial_link_text(*link_text*)

find_element_by_tag_name(*name*)

find_element_by_xpath(*xpath*)
Finds element by xpath.

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

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

find_elements_by_css_selector(*css_selector*)
Find and return list of multiple elements by CSS selector.

find_elements_by_id(*id_*)

find_elements_by_link_text(*link_text*)

find_elements_by_name(*name*)

find_elements_by_partial_link_text(*link_text*)

find_elements_by_tag_name(*name*)

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

get_attribute(*name*)
Gets the attribute value.

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

is_enabled()
Whether the element is enabled.

is_selected()
Whether the element is selected.

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

submit()
Submits a form.

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

id

location
Returns the location of the element in the renderable canvas

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

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

parent

size
Returns the size of the element

tag_name
Gets this element's tagName property.

text
Gets the text of the element.

CHAPTER

SIX

APPENDIX: FREQUENTLY ASKED QUESTIONS

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

6.1 How to use ChromeDriver ?

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

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

You should see a `chromedriver` executable. Now you can 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 other documentation.

6.2 Does Selenium 2 supports XPath 2.0 ?

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

Selenium delegate 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 support XPath 1.0 only.

6.3 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.

6.4 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
print requests.head('http://www.python.org').headers['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.

6.5 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)
```

CHAPTER

SEVEN

INDICES AND TABLES

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

PYTHON MODULE INDEX

S