

---

# **Selenium Python Bindings Documentation**

***Release 2.3***

**Baiju Muthukadan**

**Oct 26, 2017**



---

## Contents

---

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Installation</b>                                | <b>3</b>  |
| 1.1      | Introduction . . . . .                             | 3         |
| 1.2      | Downloading Selenium server . . . . .              | 3         |
| 1.3      | Downloading Python bindings for Selenium . . . . . | 3         |
| 1.4      | Running Selenium server . . . . .                  | 4         |
| <b>2</b> | <b>Getting Started</b>                             | <b>5</b>  |
| 2.1      | Simple Usage . . . . .                             | 5         |
| 2.2      | Walk through of the example . . . . .              | 5         |
| 2.3      | Using Selenium to write tests . . . . .            | 6         |
| 2.4      | Walk through of the example . . . . .              | 7         |
| 2.5      | Using Selenium with remote WebDriver . . . . .     | 8         |
| <b>3</b> | <b>Navigating</b>                                  | <b>11</b> |
| 3.1      | Interacting with the page . . . . .                | 11        |
| 3.2      | Filling in forms . . . . .                         | 12        |
| 3.3      | Drag and drop . . . . .                            | 13        |
| 3.4      | Moving between windows and frames . . . . .        | 13        |
| 3.5      | Popup dialogs . . . . .                            | 13        |
| 3.6      | Navigation: history and location . . . . .         | 14        |
| 3.7      | Cookies . . . . .                                  | 14        |
| 3.8      | Next, next steps! . . . . .                        | 14        |
| <b>4</b> | <b>Locating Elements</b>                           | <b>15</b> |
| 4.1      | Locating by Id . . . . .                           | 15        |
| 4.2      | Locating by Name . . . . .                         | 16        |
| 4.3      | Locating by XPath . . . . .                        | 16        |
| 4.4      | Locating Hyperlinks by Link Text . . . . .         | 18        |
| <b>5</b> | <b>Test Design Considerations</b>                  | <b>19</b> |
| <b>6</b> | <b>API</b>   | <b>21</b> |
| 6.1      | Exceptions . . . . .                               | 21        |
| 6.2      | Action Chains . . . . .                            | 23        |
| 6.3      | Alerts . . . . .                                   | 24        |
| 6.4      | Special Keys . . . . .                             | 25        |
| 6.5      | Firefox WebDriver . . . . .                        | 26        |

|          |  |           |
|----------|--|-----------|
| 6.6      | Chrome WebDriver . . . . .                             | 27        |
| 6.7      | Remote WebDriver . . . . .                             | 27        |
| <b>7</b> | <b>Appendix: Frequently asked questions</b>            | <b>35</b> |
| 7.1      | How to use ChromeDriver ? . . . . .                    | 35        |
| 7.2      | Does Selenium 2 supports XPath 2.0 ? . . . . .         | 35        |
| 7.3      | How to scroll down to the bottom of a page ? . . . . . | 35        |
| <b>8</b> | <b>References</b>                                      | <b>37</b> |
| <b>9</b> | <b>Indices and tables</b>                              | <b>39</b> |

**Author** [Baiju Muthukadan](#)

**Email** [baiju.m.mail AT gmail.com](mailto:baiju.m.mail@gmail.com)

**Version** 2.3



### 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 and Chrome. The current supported Python versions are Python 2.6 and Python 2.7. Python 3 is not yet supported. Selenium server is a Java program. Java Runtime Environment (JRE) 1.6 is recommended to run Selenium server. This article explain using Selenium 2 with WebDriver API. Selenium 1 API is not covered here.

### 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.

---

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 have root access in your system, you can also use your operating system instructions to install JRE.

### Downloading Python bindings for Selenium

You can download Python bindings for Selenium from the [PyPI page for selenium package](#). It has a dependency on `rdflib`, version 3.1.x.

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.

## Running Selenium server

You should have Java Runtime Environment (JRE) in the system. If *java* command is available in the PATH (environment variable), you can start the Selenium server using the command command given below. 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/absolute path to the *java* command, for eg:- `./jre1.6.0_26/bin/java`:

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



## Simple Usage

If you have installed Selenium server and Python bindings and able to run the server, 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.

## 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()
```

## Using Selenium to write tests

Selenium will be used mostly for writing test cases. You can write test cases using Python’s unittest module. 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

```

## 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 unittest module.

WebDriver offers a number of ways to find elements using one of the *find\_element\_by\_\** methods. For example, the input text element can be located by its *name* attribute using *find\_element\_by\_name* method. Detailed explanation of finding elements is available in the [Locating Elements](#) chapter:

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

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

```
elem.send_keys("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()
```

## Using Selenium with remote WebDriver

To use the remote WebDriver, you should have Selenium server running. 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 specify the values explicitly:

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



**Warning:** This section is a copy-paste from Java docs, so it requires some modification.

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

```
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".

## 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();
```

## 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.getValue() #<- FIXME: API
    option.setSelected() #<- FIXME: API
```

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 "Select", which provides useful methods for interacting with these.

```
select = driver.find_element_by_xpath("//select").select() #<- FIXME: API
select.deselectAll() #<- FIXME: API
select.selectByVisibleText("Edam") #<- FIXME: API
```

This will deselect all *OPTIONS* from the first *SELECT* on the page, and then select the *OPTION* with the displayed text of "Edam".

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();
```



## 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);
```

## 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*`.**

## Popup dialogs

Selenium WebDriver has built-in support for handling popup dialog boxes. After you've triggered an 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.

## 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.

## 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
```

## 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.

---

## 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 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*

### 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, 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')
```

## 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, 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')
```

## 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, 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 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

## Locating Hyperlinks by Link Text

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

For instance, consider this page source:

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

The `continue.html` link can be located like this:

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

## CHAPTER 5

---

### Test Design Considerations

---





This chapter cover all the interfaces of Selenium WebDriver.

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()
```

## Exceptions

**module:** `selenium.common.exceptions`

- class `WebDriverException(msg=None, screen=None, stacktrace=None)`  
base: `Exception`
- class `ErrorResponseException(response, msg)`  
base: `WebDriverException`

An error has occurred on the server side.

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

- `class InvalidSwitchToTargetException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

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

- `class NoSuchFrameException(msg=None, screen=None, stacktrace=None)`

base: `InvalidSwitchToTargetException`

The frame target to be switched doesn't exist.

- `class NoSuchWindowException(msg=None, screen=None, stacktrace=None)`

base: `InvalidSwitchToTargetException`

The window target to be switched doesn't exist.

- `class NoSuchElementException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

The `find_element_by_*` methods can't find the element.

- `class NoSuchAttributeException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

- `class StaleElementReferenceException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

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

- `class InvalidElementStateException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

- `class ElementNotVisibleException(msg=None, screen=None, stacktrace=None)`

base: `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.

- `class ElementNotSelectableException(msg=None, screen=None, stacktrace=None)`

base: `InvalidElementStateException`

- `class InvalidCookieDomainException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

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

- `class UnableToSetCookieException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

Thrown when a driver fails to set a cookie.

- `class RemoteDriverServerException(msg=None, screen=None, stacktrace=None)`

base: `WebDriverException`

- class TimeoutException(msg=None, screen=None, stacktrace=None)

Thrown when a command does not complete in enough time.

## Action Chains

**module:** selenium.webdriver.common.action\_chains

- class ActionChains(driver)

*driver:* The WebDriver instance which performs user actions.

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

- perform()

Performs all stored actions.

- click(on\_element=None)

Clicks an element.

*on\_element:* The element to click. If None, clicks on current mouse position.

- click\_and\_hold(on\_element)

Holds down the left mouse button on an element.

*on\_element:* The element to mouse down. If None, clicks on current mouse position.

- context\_click(on\_element)

Performs a context-click (right click) on an element.

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

- double\_click(on\_element)

Double-clicks an element.

*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.

*source:* The element to mouse down.

*target:* The element to mouse up.

- key\_down(key, element=None)

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

*key:* 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.

- key\_up(key, element=None)

Releases a modifier key.

*key:* 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.

- `move_by_offset(xoffset, yoffset)`  
Moving the mouse to an offset from current mouse position.  
*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.  
*to\_element*: The element to move to.
- `move_to_element_with_offset(to_element, xoffset, yoffset)`  
Move the mouse by an offset of the specified element. Offsets are relative to the top-left corner of the element.  
*to\_element*: The element to move to. *xoffset*: X offset to move to. *yoffset*: Y offset to move to.
- `release(on_element)`  
Releasing a held mouse button.  
*on\_element*: The element to mouse up.
- `end_keys(*keys_to_send)`  
Sends keys to current focused element.  
*keys\_to\_send*: The keys to send.
- `end_keys_to_element(self, element, *keys_to_send)`  
Sends keys to an element.  
*element*: The element to send keys. *keys\_to\_send*: The keys to send.

## Alerts

**module:** selenium.webdriver.common.alert

- class `Alert(driver)`
  - `text()`  
Gets the text of the Alert
  - `dismiss()`  
Dismisses the alert available
  - `accept()`  
Accepts the alert available
  - `send_keys(keysToSend)`  
Send Keys to the Alert  
*keysToSend*: Any character.

## Special Keys

**module:** selenium.webdriver.common.keys

- class Keys()
  - NULL = u'ue000'
  - CANCEL = u'ue001' # ^break
  - HELP = u'ue002'
  - BACK\_SPACE = u'ue003'
  - TAB = u'ue004'
  - CLEAR = u'ue005'
  - RETURN = u'ue006'
  - ENTER = u'ue007'
  - SHIFT = u'ue008'
  - LEFT\_SHIFT = u'ue008' # alias
  - CONTROL = u'ue009'
  - LEFT\_CONTROL = u'ue009' # alias
  - ALT = u'ue00a'
  - LEFT\_ALT = u'ue00a' # alias
  - PAUSE = u'ue00b'
  - ESCAPE = u'ue00c'
  - SPACE = u'ue00d'
  - PAGE\_UP = u'ue00e'
  - PAGE\_DOWN = u'ue00f'
  - END = u'ue010'
  - HOME = u'ue011'
  - LEFT = u'ue012'
  - ARROW\_LEFT = u'ue012' # alias
  - UP = u'ue013'
  - ARROW\_UP = u'ue013' # alias
  - RIGHT = u'ue014'
  - ARROW\_RIGHT = u'ue014' # alias
  - DOWN = u'ue015'
  - ARROW\_DOWN = u'ue015' # alias
  - INSERT = u'ue016'
  - DELETE = u'ue017'
  - SEMICOLON = u'ue018'

- EQUALS = u'ue019'
- NUMPAD0 = u'ue01a' # numbe pad keys
- 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'
- MULTIPLY = u'ue024'
- ADD = u'ue025'
- SEPARATOR = u'ue026'
- SUBTRACT = u'ue027'
- DECIMAL = u'ue028'
- DIVIDE = u'ue029'
- F1 = u'ue031' # function keys
- F2 = u'ue032'
- F3 = u'ue033'
- F4 = u'ue034'
- F5 = u'ue035'
- F6 = u'ue036'
- F7 = u'ue037'
- F8 = u'ue038'
- F9 = u'ue039'
- F10 = u'ue03a'
- F11 = u'ue03b'
- F12 = u'ue03c'
- META = u'ue03d'
- COMMAND = u'ue03d'

## Firefox WebDriver

**module:** selenium.webdriver.firefox.webdriver

- `class WebDriver(firefox_profile=None, firefox_binary=None, timeout=30)`

base: `selenium.webdriver.remote.webdriver.WebDriver`

- `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.

## Chrome WebDriver

**module:** `selenium.webdriver.chrome.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/selenium/downloads/list>

- `class WebDriver(executable_path="chromedriver", port=0)`

base: `selenium.webdriver.remote.webdriver.WebDriver`

*executable\_path*: path to the executable. If the default is used it assumes the executable is in the \$PATH

*port*: port you would like the service to run, if left as 0, a free port will be found

- `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.

## Remote WebDriver

**module:** `selenium.webdriver.remote.webdriver`

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>

- `class WebDriver(command_executor='http://127.0.0.1:4444/wd/hub', desired_capabilities=None, browser_profile=None)`

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

*command\_executor*: Either a `CommandExecutor` object or a string that specifies the URL of a remote server to send commands to.

*desired\_capabilities*: Dictionary holding predefined values for starting a browser

*browser\_profile*: A `selenium.webdriver.firefox.firefox_profile.FirefoxProfile` object. Only used if Firefox is requested.

Other Attributes:

*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 (This is set by passing *desired\_capabilities* argument)

- `name`

Returns the name of the underlying browser for this instance.

- `start_client()`  
Called before starting a new session. This method may be overridden to define custom startup behavior.
- `stop_client()`  
Called after executing a quit command. This method may be overridden to define custom shutdown behavior.
- `start_session(desired_capabilities, browser_profile=None)`  
Creates a new session with the desired capabilities.  
*desired\_capabilities*: A dictionary with following keys:
  - \* *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.
- `create_web_element(element_id)`  
Creates a web element with the specified `element_id`.
- `execute(driver_command, params=None)`  
Sends a command to be executed by a `command.CommandExecutor`.  
*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.
- `get(url)`  
Loads a web page in the current browser session.
- `title`  
Returns the title of the current page.
- `find_element_by_id(id_)`  
Finds an element by id.  
*id\_*: The id of the element to be found.  
Usage:

```
driver.find_element_by_id('foo')
```
- `find_elements_by_id(id_)`  
Finds multiple elements by id.  
*id\_*: The id of the elements to be found.  
Usage:

```
driver.find_element_by_id('foo')
```



- `find_element_by_xpath(xpath)`

Finds an element by xpath.

*xpath*: The xpath locator of the element to find.

Usage:

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

- `find_elements_by_xpath(xpath)`

Finds multiple elements by xpath.

*xpath*: The xpath locator of the elements to be found.

Usage:

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

- `find_element_by_link_text(link_text)`

Finds an element by link text.

*link\_text*: The text of the element to be found.

Usage:

```
driver.find_element_by_link_text('Sign In')
```

- `find_elements_by_link_text(text)`

Finds elements by link text.

*link\_text*: The text of the elements to be found.

Usage:

```
driver.find_elements_by_link_text('Sign In')
```

- `find_element_by_partial_link_text(link_text)`

Finds an element by a partial match of its link text.

*link\_text*: The text of the element to partially match on.

Usage:

```
driver.find_element_by_partial_link_text('Sign')
```

- `find_elements_by_partial_link_text(link_text)`

Finds elements by a partial match of their link text.

*link\_text*: The text of the element to partial match on.

Usage:

```
driver.find_element_by_partial_link_text('Sign')
```

- `find_element_by_name(name)`

Finds an element by name.

*name*: The name of the element to find.

Usage:

```
driver.find_element_by_name('foo')
```

- `find_elements_by_name(name)`

Finds elements by name.

*name*: The name of the elements to find.

Usage:

```
driver.find_elements_by_name('foo')
```

- `find_element_by_tag_name(name)`

Finds an element by tag name.

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

Usage:

```
driver.find_element_by_tag_name('foo')
```

- `find_elements_by_tag_name(name)`

Finds elements by tag name.

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

Usage:

```
driver.find_elements_by_tag_name('foo')
```

- `find_element_by_class_name(name)`

Finds an element by class name.

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

Usage:

```
driver.find_element_by_class_name('foo')
```

- `find_elements_by_class_name(name)`

Finds elements by class name.

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

Usage:

```
driver.find_elements_by_class_name('foo')
```

- `find_element_by_css_selector(css_selector)`

Finds an element by css selector.

*css\_selector*: The css selector to use when finding elements.

Usage:

```
driver.find_element_by_css_selector('#foo')
```

- `find_elements_by_css_selector(css_selector)`

Finds elements by css selector.

*css\_selector*: The css selector to use when finding elements.

Usage:

```
driver.find_element_by_css_selector('#foo')
```

- `execute_script(script, *args)`

Synchronously Executes JavaScript in the current window/frame.

*script*: The JavaScript to execute.

*\*args*: Any applicable arguments for your JavaScript.

Usage:

```
driver.execute_script('document.title')
```

- `execute_async_script(script, *args)`

Asynchronously Executes JavaScript in the current window/frame.

*script*: The JavaScript to execute.

*\*args*: Any applicable arguments for your JavaScript.

Usage:

```
driver.execute_async_script('document.title')
```

- `current_url`

URL of the current loaded page.

Usage:

```
driver.current_url
```

- `page_source`

Source code (HTML,CSS,JS etc.) of the current loaded page.

Usage:

```
driver.page_source
```

- `close()`

Closes the current window.

Usage:

```
driver.close()
```

- `quit()`

Quits the driver and closes every associated window.

Usage:

```
driver.quit()
```

- `current_window_handle`

Usage:

```
driver.current_window_handle
```

- `window_handles`

Returns the handles of all windows within the current session.

Usage:

```
driver.window_handles
```

- `switch_to_active_element()`

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

Usage:

```
driver.switch_to_active_element()
```

- `switch_to_window(window_name)`

Switches focus to the specified window.

*window\_name*: The name of the window to switch to.

Usage:

```
driver.switch_to_window('main')
```

- `switch_to_frame(index_or_name)`

Switches focus to the specified frame, by index or name.

*index\_or\_name*: The name of the window to switch to, or an integer representing the index to switch to.

Usage:

```
driver.switch_to_frame('frame_name')
driver.switch_to_frame(1)
```

- `switch_to_default_content()`

Switch focus to the default frame.

Usage:

```
driver.switch_to_default_content()
```

- `switch_to_alert()`

Switches focus to an alert on the page.

Usage:

```
driver.switch_to_alert()
```

- `back()`

Goes one step backward in the browser history.

Usage:

```
driver.back()
```

- `forward()`

Goes one step forward in the browser history.

Usage:

```
driver.forward()
```

- `refresh()`

Refreshes the current page.

Usage:

```
driver.refresh()
```

- `get_cookies()`

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

Usage:

```
driver.get_cookies()
```

- `get_cookie(name)`

Get a single cookie by name. Returns the cookie if found, None if not.

*name*: name of the cookie

Usage:

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

- `delete_cookie(name)`

Delete a particular cookie.

*name*: name of the cookie

Usage:

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

- `delete_all_cookies()`

Delete all cookies in the scope of the session.

Usage:

```
driver.delete_all_cookies()
```

- `add_cookie(cookie_dict)`

Adds a cookie to your current session.

*cookie\_dict*: A dictionary object, with the desired cookie name as the key, and the value being the desired contents.

Usage:

```
driver.add_cookie({'foo': 'bar', })
```

– `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.

*time\_to\_wait*: Amount of time to wait

Usage:

```
driver.implicitly_wait(30)
```

– `set_script_timeout(time_to_wait)`

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

*time\_to\_wait*: The amount of time to wait

Usage:

```
driver.set_script_timeout(30)
```

– `desired_capabilities`

returns the drivers current desired capabilities being used

– `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.

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

Usage:

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

– `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()
```

---

## Appendix: Frequently asked questions

---

### 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.

### Does Selenium 2 supports XPath 2.0 ?

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

### 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.



## CHAPTER 8

---

### References

---

- Blog post explaining how to use headless X for running Selenium tests: <http://coreygoldberg.blogspot.com/2011/06/python-headless-selenium-webdriver.html>
- Jenkins plugin for headless Selenium tests: <https://wiki.jenkins-ci.org/display/JENKINS/Xvnc+Plugin>



## CHAPTER 9

---

### Indices and tables

---

- [genindex](#)
- [modindex](#)
- [search](#)
- [glossary](#)