**Python Webdriver**

**1. Getting Started with Simple Program in Python**

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.clear()

elem.send\_keys("pycon")

elem.send\_keys(Keys.RETURN)

assert "No results found." not in driver.page\_source

driver.close()

The above script can be saved into a file (eg:- python\_org\_search.py), then it can be run like this:

python python\_org\_search.py

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

2.2. Example Explained

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

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. A detailed explanation of finding elements is available in the[Locating Elements](https://selenium-python.readthedocs.io/locating-elements.html#locating-elements) chapter:

elem = driver.find\_element\_by\_name("q")

Next, we are sending keys, this is similar to entering keys using your keyboard. Special keys can be sent using *Keys* class imported from *selenium.webdriver.common.keys*. To be safe, we’ll first clear any pre-populated text in the input field (e.g. “Search”) so it doesn’t affect our search results:

elem.clear()

elem.send\_keys("pycon")

elem.send\_keys(Keys.RETURN)

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

assert "No results found." not in driver.page\_source

Finally, the browser window is closed. You can also call quit method instead of close. The quitwill exit entire browser whereas close` will close one tab, but if just one tab was open, 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 options for a tool/framework are py.test and nose.

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

import unittest

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

class PythonOrgSearch(unittest.TestCase):

def setUp(self):

self.driver = webdriver.Firefox()

def test\_search\_in\_python\_org(self):

driver = self.driver

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

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

elem = driver.find\_element\_by\_name("q")

elem.send\_keys("pycon")

elem.send\_keys(Keys.RETURN)

assert "No results found." not in driver.page\_source

def tearDown(self):

self.driver.close()

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

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

python test\_python\_org\_search.py

.

----------------------------------------------------------------------

Ran 1 test in 15.566s

OK

The above result shows that the test has been successfully completed.

2.4. Walk through of the example

Initially, all the basic modules required are imported. The [unittest](http://docs.python.org/library/unittest.html) module is a built-in Python based on Java’s JUnit. This module provides the framework for organizing the test cases. The *selenium.webdriver* module provides all the WebDriver implementations. Currently supported WebDriver implementations are Firefox, Chrome, Ie and Remote. The *Keys* class provide keys in the keyboard like RETURN, F1, ALT etc.

import unittest

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

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

class PythonOrgSearch(unittest.TestCase):

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

def setUp(self):

self.driver = webdriver.Firefox()

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

def test\_search\_in\_python\_org(self):

driver = self.driver

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

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

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

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

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

elem = driver.find\_element\_by\_name("q")

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

elem.send\_keys("pycon")

elem.send\_keys(Keys.RETURN)

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

assert "No results found." not in driver.page\_source

The *tearDown* method will get called after every test method. This is a place to do all cleanup actions. In the current method, the browser window is closed. You can also call *quit* method instead of *close*. The *quit* will exit the entire browser, whereas *close* will close a tab, but if it is the only tab opened, 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 looking like this:

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

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

from selenium import webdriver

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})

**2. 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](https://selenium-python.readthedocs.io/waits.html#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 the *clear* method:

element.clear()

3.2. Filling in forms

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

element = driver.find\_element\_by\_xpath("//select[@name='name']")

all\_options = element.find\_elements\_by\_tag\_name("option")

for option in all\_options:

print("Value is: %s" % option.get\_attribute("value"))

option.click()

This will find the first “SELECT” element on the page, and cycle through each of its OPTIONs in turn, printing out their values, and selecting each in turn.

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

from selenium.webdriver.support.ui import Select

select = Select(driver.find\_element\_by\_name('name'))

select.select\_by\_index(index)

select.select\_by\_visible\_text("text")

select.select\_by\_value(value)

WebDriver also provides features for deselecting all the selected options:

select = Select(driver.find\_element\_by\_id('id'))

select.deselect\_all()

This will deselect all OPTIONs from that particular SELECT on the page.

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

select = Select(driver.find\_element\_by\_xpath("//select[@name='name']"))

all\_selected\_options = select.all\_selected\_options

To get all available options:

options = select.options

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

# Assume the button has the ID "submit" :)

driver.find\_element\_by\_id("submit").click()

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

element.submit()

3.3. Drag and drop

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

element = driver.find\_element\_by\_name("source")

target = driver.find\_element\_by\_name("target")

from selenium.webdriver import ActionChains

action\_chains = ActionChains(driver)

action\_chains.drag\_and\_drop(element, target).perform()

3.4. Moving between windows and frames

It’s rare for a modern web application not to have any frames or to be constrained to a single window. WebDriver supports moving between named windows using the “switch\_to\_window” method:

driver.switch\_to\_window("windowName")

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

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

Alternatively, you can pass a “window handle” to the “switch\_to\_window()” method. Knowing this, it’s possible to iterate over every open window like so:

for handle in driver.window\_handles:

driver.switch\_to\_window(handle)

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

driver.switch\_to\_frame("frameName")

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

driver.switch\_to\_frame("frameName.0.child")

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

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

driver.switch\_to\_default\_content()

3.5. Popup dialogs

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

alert = driver.switch\_to\_alert()

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

3.6. Navigation: history and location

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

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

To move backward and forward in your browser’s history:

driver.forward()

driver.back()

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

3.7. Cookies

Before 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 = {‘name’ : ‘foo’, ‘value’ : ‘bar’}

driver.add\_cookie(cookie)

# And now output all the available cookies for the current URL

driver.get\_cookies()

**3. 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*

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

* find\_elements\_by\_name
* find\_elements\_by\_xpath
* find\_elements\_by\_link\_text
* find\_elements\_by\_partial\_link\_text
* find\_elements\_by\_tag\_name
* find\_elements\_by\_class\_name
* find\_elements\_by\_css\_selector

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

Example usage:

from selenium.webdriver.common.by import By

driver.find\_element(By.XPATH, '//button[text()="Some text"]')

driver.find\_elements(By.XPATH, '//button')

These are the attributes available for *By* class:

ID = "id"

XPATH = "xpath"

LINK\_TEXT = "link text"

PARTIAL\_LINK\_TEXT = "partial link text"

NAME = "name"

TAG\_NAME = "tag name"

CLASS\_NAME = "class name"

CSS\_SELECTOR = "css selector"

4.1. Locating by Id

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

For instance, consider this page source:

<html>

<body>

<form id="loginForm">

<input name="username" type="text" />

<input name="password" type="password" />

<input name="continue" type="submit" value="Login" />

</form>

</body>

<html>

The form element can be located like this:

login\_form = driver.find\_element\_by\_id('loginForm')

4.2. Locating by Name

Use this when you know *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')

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, 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*

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

* [XPath Checker](https://addons.mozilla.org/en-US/firefox/addon/xpath-checker/) - suggests XPath and can be used to test XPath results.
* [Firebug](https://addons.mozilla.org/en-US/firefox/addon/firebug/) - XPath suggestions are just one of the many powerful features of this very useful add-on.
* [XPath Helper](https://chrome.google.com/webstore/detail/hgimnogjllphhhkhlmebbmlgjoejdpjl) - for Google Chrome

4.4. Locating Hyperlinks by Link Text

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

For instance, consider this page source:

<html>

<body>

<p>Are you sure you want to do this?</p>

<a href="continue.html">Continue</a>

<a href="cancel.html">Cancel</a>

</body>

<html>

The continue.html link can be located like this:

continue\_link = driver.find\_element\_by\_link\_text('Continue')

continue\_link = driver.find\_element\_by\_partial\_link\_text('Conti')

4.5. Locating Elements by Tag Name

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

For instance, consider this page source:

<html>

<body>

<h1>Welcome</h1>

<p>Site content goes here.</p>

</body>

<html>

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

heading1 = driver.find\_element\_by\_tag\_name('h1')

4.6. Locating Elements by Class Name

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

For instance, consider this page source:

<html>

<body>

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

</body>

<html>

The “p” element can be located like this:

content = driver.find\_element\_by\_class\_name('content')

4.7. Locating Elements by CSS Selectors

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

For instance, consider this page source:

<html>

<body>

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

</body>

<html>

The “p” element can be located like this:

content = driver.find\_element\_by\_css\_selector('p.content')

1. **Waits**

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

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

5.1. Explicit Waits

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

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

driver = webdriver.Firefox()

driver.get("http://somedomain/url\_that\_delays\_loading")

try:

element = WebDriverWait(driver, 10).until(

EC.presence\_of\_element\_located((By.ID, "myDynamicElement"))

)

finally:

driver.quit()

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

**Expected Conditions**

There are some common conditions that are frequently of use when automating web browsers. Listed below are the names of each. Selenium Python binding provides some [convenience methods](http://selenium-python.readthedocs.io/api.html#module-selenium.webdriver.support.expected_conditions) so you don’t have to code an expected\_condition class yourself or create your own utility package for them.

1. title\_is
2. title\_contains
3. presence\_of\_element\_located
4. visibility\_of\_element\_located
5. visibility\_of
6. presence\_of\_all\_elements\_located
7. text\_to\_be\_present\_in\_element
8. text\_to\_be\_present\_in\_element\_value
9. frame\_to\_be\_available\_and\_switch\_to\_it
10. invisibility\_of\_element\_located
11. element\_to\_be\_clickable
12. staleness\_of
13. element\_to\_be\_selected
14. element\_located\_to\_be\_selected
15. element\_selection\_state\_to\_be
16. element\_located\_selection\_state\_to\_be
17. alert\_is\_present

from selenium.webdriver.support import expected\_conditions as EC

wait = WebDriverWait(driver, 10)

element = wait.until(EC.element\_to\_be\_clickable((By.ID, 'someid')))

The expected\_conditions module contains a set of predefined conditions to use with WebDriverWait.

**Custom Wait Conditions**

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

class element\_has\_css\_class(object):

"""An expectation for checking that an element has a particular css class.

locator - used to find the element

returns the WebElement once it has the particular css class

"""

def \_\_init\_\_(self, locator, css\_class):

self.locator = locator

self.css\_class = css\_class

def \_\_call\_\_(self, driver):

element = driver.find\_element(\*self.locator) # Finding the referenced element

if self.css\_class in element.get\_attribute("class"):

return element

else:

return False

# Wait until an element with id='myNewInput' has class 'myCSSClass'

wait = WebDriverWait(driver, 10)

element = wait.until(element\_has\_css\_class((By.ID, 'myNewInput'), "myCSSClass"))

5.2. Implicit Waits

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

from selenium import webdriver

driver = webdriver.Firefox()

driver.implicitly\_wait(10) # seconds

driver.get("http://somedomain/url\_that\_delays\_loading")

myDynamicElement = driver.find\_element\_by\_id("myDynamicElement")

1. **Page Objects**

This chapter is a tutorial introduction to page objects design pattern. A page object represents an area in the web application user interface that your test is interacting.

Benefits of using page object pattern:

* Creating reusable code that can be shared across multiple test cases
* Reducing the amount of duplicated code
* If the user interface changes, the fix needs changes in only one place

6.1. Test case

Here is a test case which searches for a word in python.org website and ensure some results are found.

import unittest

from selenium import webdriver

import page

class PythonOrgSearch(unittest.TestCase):

"""A sample test class to show how page object works"""

def setUp(self):

self.driver = webdriver.Firefox()

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

def test\_search\_in\_python\_org(self):

"""

Tests python.org search feature. Searches for the word "pycon" then verified that some results show up.

Note that it does not look for any particular text in search results page. This test verifies that

the results were not empty.

"""

#Load the main page. In this case the home page of Python.org.

main\_page = page.MainPage(self.driver)

#Checks if the word "Python" is in title

assert main\_page.is\_title\_matches(), "python.org title doesn't match."

#Sets the text of search textbox to "pycon"

main\_page.search\_text\_element = "pycon"

main\_page.click\_go\_button()

search\_results\_page = page.SearchResultsPage(self.driver)

#Verifies that the results page is not empty

assert search\_results\_page.is\_results\_found(), "No results found."

def tearDown(self):

self.driver.close()

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

6.2. Page object classes

The page object pattern intends creating an object for each web page. By following this technique a layer of separation between the test code and technical implementation is created.

The page.py will look like this:

from element import BasePageElement

from locators import MainPageLocators

class SearchTextElement(BasePageElement):

"""This class gets the search text from the specified locator"""

#The locator for search box where search string is entered

locator = 'q'

class BasePage(object):

"""Base class to initialize the base page that will be called from all pages"""

def \_\_init\_\_(self, driver):

self.driver = driver

class MainPage(BasePage):

"""Home page action methods come here. I.e. Python.org"""

#Declares a variable that will contain the retrieved text

search\_text\_element = SearchTextElement()

def is\_title\_matches(self):

"""Verifies that the hardcoded text "Python" appears in page title"""

return "Python" in self.driver.title

def click\_go\_button(self):

"""Triggers the search"""

element = self.driver.find\_element(\*MainPageLocators.GO\_BUTTON)

element.click()

class SearchResultsPage(BasePage):

"""Search results page action methods come here"""

def is\_results\_found(self):

# Probably should search for this text in the specific page

# element, but as for now it works fine

return "No results found." not in self.driver.page\_source

6.3. Page elements

The element.py will look like this:

from selenium.webdriver.support.ui import WebDriverWait

class BasePageElement(object):

"""Base page class that is initialized on every page object class."""

def \_\_set\_\_(self, obj, value):

"""Sets the text to the value supplied"""

driver = obj.driver

WebDriverWait(driver, 100).until(

lambda driver: driver.find\_element\_by\_name(self.locator))

driver.find\_element\_by\_name(self.locator).clear()

driver.find\_element\_by\_name(self.locator).send\_keys(value)

def \_\_get\_\_(self, obj, owner):

"""Gets the text of the specified object"""

driver = obj.driver

WebDriverWait(driver, 100).until(

lambda driver: driver.find\_element\_by\_name(self.locator))

element = driver.find\_element\_by\_name(self.locator)

return element.get\_attribute("value")

6.4. Locators

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

The locators.py will look like this:

from selenium.webdriver.common.by import By

class MainPageLocators(object):

"""A class for main page locators. All main page locators should come here"""

GO\_BUTTON = (By.ID, 'submit')

class SearchResultsPageLocators(object):

"""A class for search results locators. All search results locators should come here"""

pass

1. **WebDriver API**

This chapter covers all the interfaces of Selenium WebDriver.

**Recommended Import Style**

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

from selenium import webdriver

Then, you can access the classes like this:

webdriver.Firefox

webdriver.FirefoxProfile

webdriver.Chrome

webdriver.ChromeOptions

webdriver.Ie

webdriver.Opera

webdriver.PhantomJS

webdriver.Remote

webdriver.DesiredCapabilities

webdriver.ActionChains

webdriver.TouchActions

webdriver.Proxy

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

from selenium.webdriver.common.keys import Keys

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

from selenium.common.exceptions import [TheNameOfTheExceptionClass]

**Conventions used in the API**

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

Here is an example for property:

* current\_url

URL of the currently loaded page.

Usage:

driver.current\_url

Here is an example of a method:

* close()

Closes the current window.

Usage:

driver.close()

7.1. Exceptions

Exceptions that may happen in all the webdriver code.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.InvalidElementStateException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.InvalidElementStateException" \o "selenium.common.exceptions.InvalidElementStateException)**

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

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

Bases: **[selenium.common.exceptions.InvalidElementStateException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.InvalidElementStateException" \o "selenium.common.exceptions.InvalidElementStateException)**

Thrown when trying to select an unselectable element.

For example, selecting a ‘script’ element.

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

Bases: **[selenium.common.exceptions.InvalidElementStateException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.InvalidElementStateException" \o "selenium.common.exceptions.InvalidElementStateException)**

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

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when an error has occurred on the server side.

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

**\_\_init\_\_(***response*, *msg*)

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when activating an IME engine has failed.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: [selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

The coordinates provided to an interactions operation are invalid.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

Bases: **[selenium.common.exceptions.NoSuchElementException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.NoSuchElementException" \o "selenium.common.exceptions.NoSuchElementException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

An error occurred while executing JavaScript supplied by the user.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when switching to no presented alert.

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when the attribute of element could not be found.

You may want to check if the attribute exists in the particular browser you are testing against. Some browsers

may have different property names for the same property. (IE8’s .innerText vs. Firefox .textContent)

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when element could not be found.

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

* Check your selector used in your find\_by…
* Element may not yet be on the screen at the time of the find operation, (webpage is still loading) see selenium.webdriver.support.wait.WebDriverWait() for how to write a wait wrapper to wait for an element to appear.

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

Bases: **[selenium.common.exceptions.InvalidSwitchToTargetException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.InvalidSwitchToTargetException" \o "selenium.common.exceptions.InvalidSwitchToTargetException)**

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

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

Bases: **[selenium.common.exceptions.InvalidSwitchToTargetException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.InvalidSwitchToTargetException" \o "selenium.common.exceptions.InvalidSwitchToTargetException)**

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

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

print driver.window\_handles

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

A screen capture was made impossible.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

A new session could not be created.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when a reference to an element is now “stale”.

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

Possible causes of StaleElementReferenceException include, but not limited to:

* You are no longer on the same page, or the page may have refreshed since the element was located.
* The element may have been removed and re-added to the screen, since it was located. Such as an element being relocated. This can happen typically with a javascript framework when values are updated and the node is rebuilt.
* Element may have been inside an iframe or another context which was refreshed.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "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](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when a driver fails to set a cookie.

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

Thrown when an unexpected alert is appeared.

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

**\_\_init\_\_(***msg=None*, *screen=None*, *stacktrace=None*, *alert\_text=None*)

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

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

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

Bases: **[selenium.common.exceptions.WebDriverException](https://selenium-python.readthedocs.io/api.html" \l "selenium.common.exceptions.WebDriverException" \o "selenium.common.exceptions.WebDriverException)**

The requested command matched a known URL but did not match an method for that URL.

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

Bases: **exceptions.Exception**

Base webdriver exception.

**\_\_init\_\_***(msg=None, screen=None, stacktrace=None)*

7.2. Action Chains

The ActionChains implementation,

class **selenium.webdriver.common.action\_chains.ActionChains***(*driver*)*

Bases: object

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

Generate user actions.

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

ActionChains can be used in a chain pattern:

menu = driver.find\_element\_by\_css\_selector(".nav")

hidden\_submenu = driver.find\_element\_by\_css\_selector(".nav #submenu1")

ActionChains(driver).move\_to\_element(menu).click(hidden\_submenu).perform()

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

menu = driver.find\_element\_by\_css\_selector(".nav")

hidden\_submenu = driver.find\_element\_by\_css\_selector(".nav #submenu1")

actions = ActionChains(driver)

actions.move\_to\_element(menu)

actions.click(hidden\_submenu)

actions.perform()

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

\_\_init\_\_(driver)

Creates a new ActionChains.

|  |  |
| --- | --- |
| **Args:** | driver: The WebDriver instance which performs user 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 offset and releases the mouse button.

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

**key\_down**(*value*, *element=None*)

Sends a key press only, without releasing it.

Should only be used with modifier keys (Control, Alt and Shift).

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

Example, pressing ctrl+c:

ActionChains(driver).key\_down(Keys.CONTROL).send\_keys('c').key\_up(Keys.CONTROL).perform()

**key\_up**(*value*, *element=None*)

Releases a modifier key.

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

Example, pressing ctrl+c:

ActionChains(driver).key\_down(Keys.CONTROL).send\_keys('c').key\_up(Keys.CONTROL).perform()

**move\_by\_offset**(*xoffset*, *yoffset*)

Moving the mouse to an offset from current mouse position.

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

**move\_to\_element(***to\_element*)

Moving the mouse to the middle of an element.

|  |  |
| --- | --- |
| **Args:** | to\_element: The WebElement to move to. |

**move\_to\_element\_with\_offset**(*to\_element*, *xoffset*, *yoffset*)

Move the mouse by an offset of the specified element.

Offsets are relative to the top-left corner of the element.

|  |  |
| --- | --- |
| **Args:** | to\_element: The WebElement to move to.  xoffset: X offset to move to.  yoffset: Y offset to move to. |

**pause**(*seconds*)

Pause all inputs for the specified duration in seconds

**perform**()

Performs all stored actions.

**release**(*on\_element=None*)

Releasing a held mouse button on an element.

|  |  |
| --- | --- |
| **Args:** | on\_element: The element to mouse up. If None, releases on current mouse position. |

**reset\_actions**()

Clears actions that are already stored on the remote end.

**send\_keys***(*\**keys\_to\_send)*

Sends keys to current focused element.

|  |  |
| --- | --- |
| **Args:** | keys\_to\_send: The keys to send. Modifier keys constants can be found in the ‘Keys’ class. |

**send\_keys\_to\_element**(*element, \*keys\_to\_send*)

Sends keys to an element.

|  |  |
| --- | --- |
| **Args:** | element: The element to send keys.  keys\_to\_send: The keys to send. Modifier keys constants can be found in the ‘Keys’ class. |

7.3. Alerts: The Alert implementation.

*class***selenium.webdriver.common.alert.Alert**(*driver*)

Bases: **object**

Allows to work with alerts.

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

Accepting / Dismissing alert prompts:

Alert(driver).accept()

Alert(driver).dismiss()

Inputting a value into an alert prompt:

name\_prompt = Alert(driver) name\_prompt.send\_keys(“Willian Shakesphere”) name\_prompt.accept()

Reading a the text of a prompt for verification:

alert\_text = Alert(driver).text self.assertEqual(“Do you wish to quit?”, alert\_text)

**\_\_init\_\_(***driver*)

Creates a new Alert.

|  |  |
| --- | --- |
| Args: | driver: The WebDriver instance which performs user actions. |

**accept()**

Accepts the alert available.

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

**authenticate(username, password)**

Send the username / password to an Authenticated dialog (like with Basic HTTP Auth). Implicitly ‘clicks ok’

Usage:: driver.switch\_to.alert.authenticate(‘cheese’, ‘secretGouda’)

|  |  |
| --- | --- |
| Args: | -username: string to be set in the username section of the dialog -password: string to be set in the password section of the dialog |

**dismiss()**

Dismisses the alert available.

**send\_keys(keysToSend)**

Send Keys to the Alert.

|  |  |
| --- | --- |
| Args: | keysToSend: The text to be sent to Alert. |

**text**

Gets the text of the Alert

7.4. Special Keys

The Keys implementation.

class**selenium.webdriver.common.keys.Keys**

Bases**: object**

Set of special keys codes.

7.5. Locate elements By

These are the attributes which can be used to locate elements. See the [Locating Elements](https://selenium-python.readthedocs.io/locating-elements.html#locating-elements)chapter for example usages.

The By implementation.

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

Bases: **object**

Set of supported locator strategies.

**CLASS\_NAME =** 'class name'

**CSS\_SELECTOR =** 'css selector'

**ID =** 'id'

**LINK\_TEXT =** 'link text'

**NAME =** 'name'

**PARTIAL\_LINK\_TEXT =** 'partial link text'

**TAG\_NAME =** 'tag name'

**XPATH =** 'xpath'