## Downloading Python bindings for Selenium

pip install selenium

## 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()

## 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()

## 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:

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()

## 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.

## 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.

## 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()

# 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

# 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.

## 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 so you don’t have to code an expected\_condition class yourself or create your own utility package for them.

* title\_is
* title\_contains
* presence\_of\_element\_located
* visibility\_of\_element\_located
* visibility\_of
* presence\_of\_all\_elements\_located
* text\_to\_be\_present\_in\_element
* text\_to\_be\_present\_in\_element\_value
* frame\_to\_be\_available\_and\_switch\_to\_it
* invisibility\_of\_element\_located
* element\_to\_be\_clickable
* staleness\_of
* element\_to\_be\_selected
* element\_located\_to\_be\_selected
* element\_selection\_state\_to\_be
* element\_located\_selection\_state\_to\_be
* alert\_is\_present

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

wait = WebDriverWait(driver, 10)

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

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

**Custom Wait Conditions**

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

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

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

*locator - used to find the element*

*returns the WebElement once it has the particular css class*

*"""*

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

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"))

## 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")

Prepare Your First Selenium Webdriver Automation Script Using Python

Let’s start with Selenium WebDriver and create a Python script that uses Selenium classes and functions to automate browser interaction.

Here we will show you a sample script that opens “www.google.com” enters a search text in the Google search text box. Test script then verifies the Google search page on which has the searched text displayed.

Selenium Webdriver Python Script For Firefox

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

# create a new Firefox session

driver = webdriver.Firefox()

driver.implicitly\_wait(30)

driver.maximize\_window()

# Navigate to the application home page

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

# get the search textbox

search\_field = driver.find\_element\_by\_id("lst-ib")

search\_field.clear()

# enter search keyword and submit

search\_field.send\_keys("Selenium WebDriver Interview questions")

search\_field.submit()

# get the list of elements which are displayed after the search

# currently on result page using find\_elements\_by\_class\_name method

lists= driver.find\_elements\_by\_class\_name("\_Rm")

# get the number of elements found

print ("Found " + str(len(lists)) + " searches:")

# iterate through each element and print the text that is

# name of the search

i=0

for listitem in lists:

print (listitem.get\_attribute("innerHTML"))

i=i+1

if(i>10):

break

# close the browser window

driver.quit()

Decoding The Above Script

Let’s discuss the script line by line to get a better understanding of the Selenium WebDriver statements in brief. There will be lot more about different Selenium WebDriver functions in upcoming posts.

Step-1.

The selenium webdriver module implements the classes that support different browsers including Firefox, Chrome, Internet Explorer, Safari, others and RemoteWebDriver too to test on browsers available on remote machines.

We need to import webdriver from the Selenium package to use the Selenium WebDriver methods as:

***from selenium import webdriver***

Step-2.

Next, we need the object of the browser which we’ll use to load the web page. The browser object provides a programmable interface to communicate with the browser using the Selenium commands. In the test script, we are using Firefox. We can create an instance of the Firefox as shown in the following code:

***driver = webdriver.Firefox()***

On executing this statement, a new Firefox window will launch. We had made the following settings for the driver instance:

***driver.implicitly\_wait(30)***  
***driver.maximize\_window()***

We configured a timeout for Selenium to launch the browser in 30 seconds. Next statement maximizes the browser window.

Step-3.

Next, we will navigate to the application, in our case ‘***http://www.google.com***,’ passing the given URL to the driver.get() method. After making a call to the get() method, Webdriver waits until the page gets rendered in the browser window and sends the control back to the script.

After the page gets loaded, Selenium will interact with various elements on the page. Next, in the test script, we will be looking at different Selenium WebDriver functions that search an HTML object, send a text to the web component, simulate keypress event, click buttons and select from drop downs, etc. Let’s see all these functions getting used in the next step.

Step-4.

***\****First of all, we’ll locate the Google Search textbox to supply the text input for the Search. The Search text box has an id attribute as <lst-ib>, and you can identify it from the code given below:

***search\_field = driver.find\_element\_by\_id(<lst-ib>)***

**\*** After locating the Search text box, we are trying to interact with the textbox element by clearing the previous value using the clear() method and then using the send\_keys() method to provide a new value. Subsequently calling the submit() method will forward the search request for processing. You can see a quick preview of these steps in the next few lines.

***search\_field.clear()***  
***search\_field.send\_keys(“Selenium WebDriver Interview questions”)***  
***search\_field.submit()***

**\*** After submitting the search request, Firefox driver will display the result page returned by Google. The result page shows a list of entries that match the searched text. Each of the entry in the list is captured in anchor <a> element and can be accessed using “find\_elements\_by\_class\_name” method. If used, it will return a list of elements as:

***lists= driver.find\_elements\_by\_class\_name(“\_Rm”)***

**\*** The list of items expands to many pages, so we are restricting our code to print first ten entries captured in the anchor tag. We are outputting the names of the entries using the **“innerHTML”**property of the anchor **<a>** elements:

i=0

for listitem in lists:

print (listitem.get\_attribute("innerHTML"))

i=i+1

if(i>10):

break

This example gave us a real insight into using Selenium WebDriver and Python together to create a simple test automation script. It is a very basic example script. We will use other interesting and complicated features of Selenium Library with Python in our upcoming posts.

Extract Link

**from** selenium **import** webdriver

options = webdriver.ChromeOptions()

options.add\_argument('--ignore-certificate-errors')

options.add\_argument("--test-type")

options.binary\_location = "/usr/bin/chromium"

driver = webdriver.Chrome(chrome\_options=options)

driver.get('https://www.w3.org/')

**for** a **in** driver.find\_elements\_by\_xpath('.//a'):

**print**(a.get\_attribute('href'))

Button Click

**from** selenium **import** webdriver

**import** time

options = webdriver.ChromeOptions()

options.add\_argument('--ignore-certificate-errors')

options.add\_argument("--test-type")

options.binary\_location = "/usr/bin/chromium"

driver = webdriver.Chrome(chrome\_options=options)

driver.get('http://codepad.org')

*# click radio button*

python\_button = driver.find\_elements\_by\_xpath("//input[@name='lang' and @value='Python']")[0]

python\_button.click()

*# type text*

text\_area = driver.find\_element\_by\_id('textarea')

text\_area.send\_keys("print('Hello World')")

*# click submit button*

submit\_button = driver.find\_elements\_by\_xpath('//\*[@id="editor"]/table/tbody/tr[3]/td/table/tbody/tr/td/div/table/tbody/tr/td[3]/input')[0]

submit\_button.click()

Take Screenshot

**from** selenium **import** webdriver

options = webdriver.ChromeOptions()

options.add\_argument('--ignore-certificate-errors')

options.add\_argument("--test-type")

options.binary\_location = "/usr/bin/chromium"

driver = webdriver.Chrome(chrome\_options=options)

driver.get('https://python.org')

driver.save\_screenshot("screenshot.png")

driver.close()

# 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

## 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()

## 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

## 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")

## 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**

**Framework :**

## ****What are Frameworks?****

A test automation framework is an integrated set of technologies, tools, processes and patterns brings organization of thought and clarity thereby simplifying and enabling a team perform Automation not only effectively but also efficiently. We would be focusing on the technology aspect of Framework on this website. I blogged about **attributes of frameworks** a while ago, however this section and tutorial is getting all the pieces together and connecting the dots. We have also talked about how do we **choose an Automation solution** at the beginning of Basic Tutorial and setting the stage and context. We touched upon frameworks there but more from the perspective of programming language, design patterns, process models et al. Here we would be focusing on the “design patterns”, otherwise sometimes referred to as technical frameworks in many contexts when we get tactical and implementation specific.

## ****Why do we need one?****

For the same reasons in the blog, I will try to keep it simple and move quickly to the coding aspects, however let’s spend some time on this because we need to know “why” we write the code we write in a framework.

1. **Maintainability**:  How do we ensure maintenance of scripts is optimal?  By that we mean, as we evolve by churning out feature files, step definitions, data and page objects, we need to have a strategy to be able to **diagnose**, **version control** and **audit** the artifacts produced. There should also be a **tight coupling**between the released version of source code and the corresponding set of acceptance tests which tested that released version.
2. **Re-usability**: What are the **specific artifacts** that can be re-used across feature teams? This is particularly helpful during integration testing when components/functions have to talk to each other. For example, data and page objects are re-usable artifacts
3. **Scalability**: How do we **scale the acceptance tests** and ultimately accelerate the feedback loop to verify/validate the software to release and market per deadlines?We can use Selenium GRID for parallel execution of tests. The necessary infrastructure required for a distributed parallel execution of tests needs to be discussed too. The technology needs for the same in the Technology section and any exceptions and dependencies on infrastructure (hardware/software) should be laid out.
4. **Configurability**: How do we configure the test automation framework and how much of it is**convention vs. configuration?**Environment configuration (switch environments),Logging (Detailed log to help debug)
5. **Auditability**: How do we keep track of work done and roll up metrics to serve QS dashboards or any other management required metrics. Reports (Historical vs. current), Auditability (System of record)

## ****Which one to select?****

Below are popular framework patterns in Test Automation space and I have tried my best to provide guidance to help select the framework based on situations.

1) **Keyword Driven** : Keywords are essentially blocks or functions that are named so that they can be consumed for a given functionality. At a simplest level, keyword takes in parameters and throws back some output. Keywords are written to abstract the complexity of repeatedly writing code. Think of Keywords as API’s that serve you (or your application) when called. In one of the frameworks I designed for a large company, Keywords.java was a Class file with list of Java methods with defined inputs and outputs. There is a very thin line between keywords and modules. We hear the term “modularization” or “modular framework” and so on and there is a intersection of spaces here.

So Keywords or modules would pretty much the same on this website. In the Ruby world, we can implement this kind of framework by defining modules and mixin them into strategic places. When we get into frameworks discussion, we will see that modules are crucial pieces to bake into the framework. Not exactly similar, but a module in Ruby is very close to an abstract class in Java, however there are differences too.

However Keywords/modules can get complicated too based  on data structures and layers in your test automation framework

2) **Data Driven**: If your applications is NOT too deep with pages , however each page can have scenarios that need to be tested with large datasets, you would want to write automation scripts with a focus on test data aka. data-driven. Tools like QTP already have excel sheet parsing etc that loops through rows and the same test case is executed for each data-set. It helps me to think in an easier way when I talk about excel for data driven, however please DO NOT think excel is the most optimum way to handle test data. In fact, excel is heavier than xml, yml or other data exchange formats. If you are at the beginning of designing a data-driven framework, go ahead and use a spreadsheet to manage your data, however as you mature and advance, try doing away with spreadsheets.

3) **Page Object**: This is a very popular pattern used these days for Test Automation Frameworks. If you think of a web based application as a network of pages that interact with each other AND there are large number of pages in your application that can change nature frequently, this pattern is the most favorable one to use.

Page-objects (or classes) are defined for each page, where a page contains the html locator information and also the methods that can operate on those page elements. Though we say page-objects (aka. classes) are written for each page, there are some nuances in terms of how we model a Web page into a page-object. A network of page-objects is supposed to model the state of the web application ultimately, however with asynchronous web applications, where parts of page get refreshed (as opposed to full web page. aka. DOM), we have to handle ajax calls based on the javascript library. Anyways, will mention the caveats, pros and cons, situations in which **web application (page html source) to page-object modeling becomes the most important piece** in your framework etc, will be dealt in a separate post at the end of frameworks tutorial.The advantage of this pattern from the decision making point is:

* If a html locator information changes for an element, changing it in the page-object reflects across all the scripts – nice right
* Gives us ability to define keywords too as we can define methods inside page-objects
* We can define navigation logic between the pages too as a way to tie the links between the pages

4) ***Hybrid***: A Hybrid model is permutations and combinations of the above three frameworks described