**TDD : Test Driven Development (Cucumeber)**

**Installation**

The recommended installation method is using pip:

pip install --upgrade robotframework-seleniumlibrary

pip install robotframework-seleniumlibrary

Running this command installs also the latest Selenium and Robot Framework versions, but you still need to install browser drivers separately. The --upgrade option can be omitted when installing the library for the first time.

Those migrating from Selenium2Library can install SeleniumLibrary so that it is exposed also as Selenium2Library:

pip install --upgrade robotframework-selenium2library

The above command installs the normal SeleniumLibrary as well as a new Selenium2Library version that is just a thin wrapper to SeleniumLibrary. That allows importing Selenium2Library in tests while migrating to SeleniumLibrary.

To install the last legacy Selenium2Library version, use this command instead:

pip install robotframework-selenium2library==1.8.0

With resent versions of pip it is possible to install directly from the GitHub repository. To install latest source from the master branch, use this command:

pip install git+https://github.com/robotframework/SeleniumLibrary.git

Please note that installation will take some time, because pip will clone the SeleniumLibrary project to a temporary directory and then perform the installation.

See Robot Framework installation instructions for detailed information about installing Python and Robot Framework itself. For more details about using pip see its own documentation.

**Browser drivers**

After installing the library, you still need to install browser and operating system specific browser drivers for all those browsers you want to use in tests. These are the exact same drivers you need to use with Selenium also when not using SeleniumLibrary. More information about drivers can be found from Selenium documentation.

The general approach to install a browser driver is downloading a right driver, such as chromedriver for Chrome, and placing it into a directory that is in PATH. Drivers for different browsers can be found via Selenium documentation or by using your favorite search engine with a search term like selenium chrome browser driver. New browser driver versions are released to support features in new browsers, fix bug, or otherwise, and you need to keep an eye on them to know when to update drivers you use.

Alternatively, you can use a tool called WebdriverManager which can find the latest version or when required, any version of appropriate webdrivers for you and then download and link/copy it into right location. Tool can run on all major operating systems and supports downloading of Chrome, Firefox, Opera & Edge webdrivers.

Here's an example:

pip install webdrivermanager

webdrivermanager firefox chrome --linkpath /usr/local/bin

**Usage**

To use SeleniumLibrary in Robot Framework tests, the library needs to first be imported using the Library setting as any other library. The library accepts some import time arguments, which are documented in the keyword documentation along with all the keywords provided by the library.

When using Robot Framework, it is generally recommended to write as easy-to-understand tests as possible. The keywords provided by SeleniumLibrary are pretty low level, though, and often require implementation specific arguments like element locators to be passed as arguments. It is thus typically a good idea to write tests using Robot Framework's higher level keywords that utilize SeleniumLibrary keywords internally. This is illustrated by the following example where SeleniumLibrary keywords like Input Text are primarily used by higher level keywords like Input Username.

Users.robot (here robot is extension of file)

${LOGIN URL} variable

\*\*\* Settings \*\*\*

**Documentation** Simple example using SeleniumLibrary.

**Library** SeleniumLibrary

\*\*\* Variables \*\*\*

${LOGIN URL} http://localhost:7272

${BROWSER} Chrome

\*\*\* Test Cases \*\*\*

Valid Login

Open Browser To Login Page

Input Username demo

Input Password mode

Submit Credentials

Welcome Page Should Be Open

[**Teardown**] Close Browser

\*\*\* Keywords \*\*\*

Open Browser To Login Page

Open Browser ${LOGIN URL} ${BROWSER}

Title Should Be Login Page

Input Username

[**Arguments**] ${username}

Input Text username\_field ${username}

Input Password

[**Arguments**] ${password}

Input Text password\_field ${password}

Submit Credentials

Click Button login\_button

Welcome Page Should Be Open

Title Should Be Welcome Page

To book a flight we need to perform the following steps:

1. Open a http://blazedemo.com/ web page.
2. Choose any departure city from a drop-down list.
3. Choose any destination city from a drop-down list.
4. Click on “Find Flights” button.
5. Verify if there are flights found on Flights search result page.

To implement the test, the steps  above need to be converted to keywords. If we tried to implement a test case without user defined keywords, our test would look like the following (consider that we are using SeleniumLibrary from the Robot Framework).

search\_flights.robot:

\*\*\* Settings \*\*\*

Library SeleniumLibrary

\*\*\* Test Cases \*\*\*

The user can search for flights

[Tags] search\_flights

Open browser http://blazedemo.com/ Chrome

Select From List By Value xpath://select[@name='fromPort'] Paris

Select From List by Value xpath://select[@name='toPort'] London

Click Button css:input[type='submit']

@{flights}= Get WebElements css:table[class='table']>tbody tr

Should Not Be Empty ${flights}

Close All Browsers

To be able to use keywords from external libraries (like SeleniumLibrary) we need to import it. This should be done in the “Settings” section of the code in Robot with the setting “Library”. The “Settings” section is used not only for importing external libraries and resources, but also for defining metadata for test suites and test cases. We will come back to “Settings” section later in the blog post.

Underneath the “Settings” section there is “Test Cases” section where you should add all the test cases within a test cases file. As you can see, we have used keywords from SeleniumLibrary to open the browser and to select appropriate values from departure and destination drop-down lists.

In the Robot Framework, any keyword can accept any number of arguments. In our case, for example, the keyword “Open browser” accepts two arguments: the URL to open and the browser this URL should be opened in. The keyword “Select From List by Value” accepts the selector of the web element as the first argument, and accepts the value to select as the second argument.

The Tags section is used to assign a logical group to a test case. Furthermore, it can be used to execute tests only with the specified tag. I will explain how to achieve this later in the blog post.

There is one more thing that needs to be explained in more detail here. The result of the “Get WebElements” keyword is assigned to the variable “@flights”. You can use the variable to describe keyword arguments, to store a value in a test suite or to save the results of keyword execution. The names of variables are case-insensitive, and as well as names of keywords, and spaces and underscores are ignored. The test case can be re-written with usage of variables in the following way:

\*\*\* Settings \*\*\*

Library SeleniumLibrary

\*\*\* Variables \*\*\*

${URL} http://blazedemo.com/

${BROWSER} Chrome

\*\*\* Test Cases \*\*\*

The user can search for flights

Open browser ${URL} ${BROWSER}

Such styling adds more readability to the test case. Variables can be scalar (with the $ prefix), lists (with the @ prefix, dictionaries (with the & prefix) and environment (with the % prefix).

After saving the available flights as web elements in the list “flights”, we use the built in keyword “Should Not Be Empty” to verify there is at least one flight found. As a last step we close all opened browsers.

Now, it’s high time to make some improvements using Robot Framework’s capabilities. If you want to add more test cases to the test suite, you will notice that anytime you want to interact with the page, you need to open it. And, after test passed, you need to close the browser. With the help of the “Settings” section that was mentioned previously, you can manage these tear up and tear down operations.

Updated test case:

\*\*\* Settings \*\*\*

Library SeleniumLibrary

Suite Setup Open browser ${URL} ${BROWSER}

Suite Teardown Close All Browsers

\*\*\* Variables \*\*\*

${URL} http://blazedemo.com/

${BROWSER} Chrome

\*\*\* Test Cases \*\*\*

The user can search for flights

[Tags] search\_flights

Select From List By Value xpath://select[@name='fromPort'] Paris

Select From List by Value xpath://select[@name='toPort'] London

Click Button css:input[type='submit']

@{flights}= Get WebElements css:table[class='table']>tbody tr

Should Not Be Empty ${flights}

As you can see, we have moved open browser and closing all browsers to suite setup and suite teardown respectively.

# Creating Keywords

As another improvement, we can develop custom keywords to select departure and destination cities and to find flights.

Search\_flights\_keywords.robot:

\*\*\* Settings \*\*\*

Library SeleniumLibrary

\*\*\* Variables \*\*\*

${URL} http://blazedemo.com/

${BROWSER} Chrome

\*\*\* Keywords \*\*\*

Open Home Page

Open browser ${URL} ${BROWSER}

Close Browsers

Close All Browsers

Select Departure City

[Arguments] ${departure\_city}

Select From List By Value xpath://select[@name='fromPort'] ${departure\_city}

Select Destination City

[Arguments] ${destination\_city}

Select From List by Value xpath://select[@name='toPort'] ${destination\_city}

Search For Flights

Click Button css:input[type='submit']

There are available Flights

@{flights}= Get WebElements css:table[class='table']>tbody tr

Should Not Be Empty ${flights}

All keywords we are intending to use should be created in the “Keywords” section.  You can notice that we don’t have any test cases in the file. Such files are considered as resource files and they have slightly different properties than test cases files: in the “Settings” section you cannot use metadata settings such as Suite Setup, Suite TearDown or Tags settings. You can only use the import settings (Library, Resource, Variables) and Documentation.

Going back to the created keywords: In “Select Departure City” and “Select Destination City” keywords we have an [Arguments] property, which means these keywords require arguments. A keyword can have any number of arguments but it’s advised not to overload it with too many. If a keyword has many arguments it’s harder to understand what it does and it becomes prone to errors.

Our test cases file search\_flights.robot was changed to reflect the modifications:

\*\*\* Settings \*\*\*

Resource search\_flights\_keywords.robot

Suite Setup Open Home Page

Suite Teardown Close Browsers

\*\*\* Test Cases \*\*\*

The user can search for flights

[Tags] search\_flights

Select Departure City Paris

Select Destination City London

Search For Flights

There are available Flights

Resources files are loaded via the setting “Resource”. Now our test looks more elegant and it is easy to understand what the test steps are.

# Initialization Files

The next thing we can do is to add a setup suite and a teardown suite to the test suite initialization file. As mentioned previously in the blog post, a test cases file forms a test suite. But test suite files can be put to a higher-level test suite by creating directories. On its own a directory cannot have the setup and teardown information, but the Robot Framework has initialization files for that. The name of the initialization file should be \_\_init\_\_.ext, where ext should be one of the supported formats.

Let’s create a directory for our test suite and name it search\_flights.

Our initialization file will be search\_flights/\_\_init\_\_.robot (don’t confuse it with python \_\_init\_\_.py)

\*\*\* Settings \*\*\*

Suite Setup Open Home Page

Suite Teardown Close Browsers

Resource search\_flights\_keywords.robot

Like resource files, initialization files cannot have test cases and not all settings are supported. Variables and keywords that were created or imported will not be available in the lower level suites. Use initialization files to provide common tags, documentation and setup/teardown operations.

With the initialization file, our test case looks even simpler:

\*\*\* Settings \*\*\*

Resource search\_flights\_keywords.robot

\*\*\* Test Cases \*\*\*

The user can search for flights

Select Departure City Paris

Select Destination City London

Search For Flights

There are available Flights

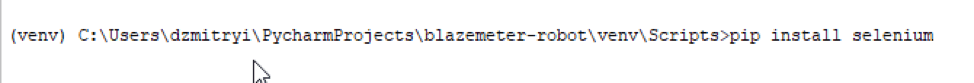
It’s a good practice to keep test cases as simple as possible. It will make them more stable and easy to modify when needed.

# Creating Keywords from Functions or Methods

Ok, we saw how to create a test case using existing keywords that are grouped into user-defined keywords. But what if we need to save data to a database when selecting the departure city? Or to log information to a file? Those are cases when we need to form a keyword from a function or method of the class. The Robot Framework allows us to do that.

Let’s modify our test to use keywords formed from python class methods.

1. First, we need to install the Selenium library with pip:



2. Download the Chrome WebDriver from the Selenium download page.

3. Let’s create a Python class that will manage the WebDriver session:

class WebDriverManager(object):

\_\_driver = None

@classmethod

def get\_web\_driver(cls, browser):

if cls.\_\_driver is None:

if (browser.lower()) == "chrome":

cls.\_\_driver = webdriver.Chrome("C:/drivers/chromedriver.exe")

return cls.\_\_driver

The WebDriverManager class stores the created driver session in the \_\_driver variable and returns it when the get\_web\_driver method is called.

4. Since web pages elements are not loaded simultaneously, we need to wait until the element becomes visible on the page. In Selenium, this can be achieved through the WebDriverWait class. We will create the class Web that will encapsulate the usage of the WebDriverWait class instance by exposing simple methods to search for web elements on the page by xpath.

from selenium.webdriver.common.by import By

from selenium.webdriver.support.wait import WebDriverWait

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

from web\_driver.web\_driver\_manager import WebDriverManager

class Web(object):

\_driver = None

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

self.\_driver = WebDriverManager.get\_web\_driver(browser)

self.\_wait = WebDriverWait(self.\_driver, 10)

def get\_web\_element\_by\_xpath(self, xpath):

return self.\_wait.until(EC.presence\_of\_element\_located((By.XPATH, xpath)))

def get\_web\_elements\_by\_xpath(self, xpath):

return self.\_wait.until(EC.presence\_of\_all\_elements\_located((By.XPATH, xpath)))

def open(self, path):

self.\_driver.get(path)

def close\_all(self):

self.\_driver.quit()

5. Now, we need a class that will expose methods to interact with elements on the Flights search page.

from web\_driver.web import Web

class SearchFlightPage(object):

\_\_url = "http://blazedemo.com/"

def open(self):

self.\_web.open(self.\_\_url)

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

self.\_web = Web(browser)

def select\_departure\_city(self, city):

self.\_web.get\_web\_element\_by\_xpath("//select[@name='fromPort']/option[@value='{}']".format(city)).click()

def select\_destination\_city(self, city):

self.\_web.get\_web\_element\_by\_xpath("//select[@name='toPort']/option[@value='{}']".format(city)).click()

def search\_for\_flights(self):

self.\_web.get\_web\_element\_by\_xpath("//input[@type='submit']").click()

def get\_found\_flights(self):

return self.\_web.get\_web\_elements\_by\_xpath("//table[@class='table']/tbody/tr")

def close(self):

self.\_web.close\_all()

This class uses a Web instance to access web elements on the web page and has a method to select values from departure and destination drop-down lists, clicking on “Find Flights” button, and getting found flights.

This class also has the methods open() and close(), which are self-explanatory.

Let’s have a look at our modified test case:

\*\*\* Settings \*\*\*

Library pages//SearchFlightPage.py Chrome

Suite Setup Open

Suite Teardown Close

\*\*\* Test Cases \*\*\*

The user can search for flights

Select Departure City Paris

Select Destination City London

@{flights}= Get Found Flights

Should Not Be Empty ${flights}

We are using SearchFlightsPage.py library to import class methods as keywords.

6. There is still one thing we can improve here. As you see we are doing a little programming by saving a list of flights into a variable and then passing it further to ‘Should Not Be Empty” keyword.

We can bypass this by saving a test variable using the keyword “Set Test Variable” in the resource file.

Have a look at search\_flights\_keywords.robot:

\*\*\* Settings \*\*\*

Library pages//SearchFlightPage.py Chrome

\*\*\* Keywords \*\*\*

Open search page

Open

Close pages

Close

Select departure

[Arguments] ${city}

select departure city ${city}

select destination

[Arguments] ${city}

select destination city ${city}

Search Flights

search for flights

@{flights}= Get Found Flights

set test variable ${flights}

Flights are found

Should Not Be Empty ${flights}

In the “Search Flights” keyword we are saving search results into the test variable “${flights}”. Later, in “Flights are found” keyword we read this variable.

Our modified test:

\*\*\* Settings \*\*\*

Resource search\_flights\_keywords.robot

Suite Setup Open search page

Suite Teardown Close pages

\*\*\* Test Cases \*\*\*

The user can search for flights

Select Departure Paris

Select Destination London

Search Flights

Flights are found

The choice to use an already developed keyword or to create your own is always up to a test developer. Try to keep things as simple as possible: if you don’t need additional behavior that can be implemented only in python modules or classes, use already developed keywords – it will save your time.

# Executing Your Tests in Robot

To execute robot tests in your prompt, type: **robot path/to/tests.**

‘path/to/tests’ should be a name of a suite file or a suite directory.

You can select which tests to be executed from a suite by filtering by tags. Any robot test or suite can have a tag assigned. To execute tests with an assigned tag, type the following in your prompt:

Robot –include [tag] path/to/tests

# Executing Your Robot Test in Taurus

You can also create your test scripts with Taurus or upload existing ones, in a much easier way. In Taurus, you can run your test in loops to get transaction time statistics. By using the command -cloud you can delegate the test to the cloud through BlazeMeter, getting an online report and without having to have your own infrastructure.

Create and execute a configuration for your Robot test in Taurus and you will get a detailed result of how your application works.

Consider the following configuration to check if our “Find Flights” scenario will work for 1 min:

execution:

- executor: selenium

runner: robot

hold-for: 1m

scenario:

script: search\_flights/

reporting:

- final-stats

- blazemeter