**Python Automation and Testing**

Types of Test Automation according to Mike Cohn:

1. User Interface test

Tests UI.

1. Service/API layer test

Functionally test.

1. Unit test

Tests code.

![A close up of text on a white background

Description automatically generated]()

Python Libraries used for

1. Unit and API testing: unitest, pytest, nose
2. UI testing: selenium and python

Selenium Automation Framework

Advantages:

1. Automates browsers
2. Used for Web UI (actions performed by a user in a web browser window) automation

Installing Selenium in Windows

1. In a browser, open <https://selenium-python.readthedocs.io/installation.html>
2. To install Selenium, open a CMD prompt, and type the following:

C:\> pip.exe install selenium

1. Install the driver for the browser of your choice (Firefox), in your browser go to <https://github.com/mozilla/geckodriver/releases>, download and install https://github.com/mozilla/geckodriver/releases/download/v0.26.0/geckodriver-v0.26.0-win64.zip
2. Additionally, add the driver to the system path (in windows you can just copy it to C:\windows\system32\

Better if you copy the driver to the python.exe path

1. Add the path to python.exe and firefox.exe to the system path

Test Selenium-Python

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE

from selenium import webdriver;

browser= webdriver.Firefox();

//\* creates an instance of the browser driver

browser.get('http://www.seleniumhq.org');

It should open 'http://www.seleniumhq.org' with Firefox.

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

import time

driver = webdriver.Firefox()

//\* for chrome use, driver = webdriver.Chrome()

//\* for edge use, driver = webdriver.Edge()

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

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

//\* we look for the element “q”, which is the search box on that page

//\* from the source code of the page:

<input id="id-search-field" name="q" type="search" role="textbox" class="search-field" placeholder="Search" value="" tabindex="1">

elem.clear()

elem.send\_keys("pycon")

//\* we search for the word “pycon”

elem.send\_keys(Keys.RETURN)

time.sleep(8)

driver.close()

//\* the browser’s tab (not the whole browse) will be closed after 8 seconds

HTML DOM Structure

The DOM presents an HTML document as a tree structure with elements, attributes, and text.

With the DOM, programmers can create and build documents, navigate their structure, and add, modify, or delete elements and content.

![A close up of text on a white background

Description automatically generated]()

Each element under the document root, that is HTML, there's an element node,

These element nodes have text nodes containing the text that is within the element.

If there were an element with attributes, an attribute node would be created for that element and any text for the attribute would create a text node under that attribute node.

HTML Document Sample (file:///C:/Users/rick.yamamoto/Desktop/Training/Python/Python%20Automation%20and%20Testing/Ex\_Files\_Python\_Automation\_Testing/Ex\_Files\_Python\_Automation\_Testing/Exercise%20Files/CH02/html\_code\_02.html”)

<html>

<meta http-equiv="Content-Type" content="text/html;charset=ISO-8859-1">

 <body>

<h3 class="content"> This is a form example</h3>

  <form id="loginForm">

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

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

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

<input name="continue" type="reset" value="Clear" class="Button" />

  </form>

 </body>

<html

Locating elements by the HTML id Attribute

It specifies a unique id for an HTML element.

It is case sensitive.

We use this method to locate elements when the id attribute of an element is known.

When selenium search for an ID, the first element with a matching attribute value will be returned.

Test Selenium for HTML id Attribute search

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE

from selenium import webdriver

driver= webdriver.Firefox()

driver.get(“file:///C:/Users/rick.yamamoto/Desktop/Training/Python/Python%20Automation%20and%20Testing/Ex\_Files\_Python\_Automation\_Testing/Ex\_Files\_Python\_Automation\_Testing/Exercise%20Files/CH02/html\_code\_02.html”)

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

print("My login form element is:")

print(login\_form)

//\* <selenium.webdriver.firefox.webelement.FirefoxWebElement (session="0a2801a8-a7d8-4cd6-a26a-6cbad8f9744f", element="bf2baf1c-c365-4a4a-be72-d1deaaec571c")>

driver.close()

Locating elements by the HTML name Attribute

The name attribute does not need to be unique in the HTML document

Use this method to look at elements when the name attribute of an element is known.

In the case that there are multiple elements with the same name, the first element with the matching attribute value will be returned.

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE

from selenium import webdriver

driver= webdriver.Firefox()

driver.get(“file:///C:/Users/rick.yamamoto/Desktop/Training/Python/Python%20Automation%20and%20Testing/Ex\_Files\_Python\_Automation\_Testing/Ex\_Files\_Python\_Automation\_Testing/Exercise%20Files/CH02/html\_code\_02.html”)

username = driver.find\_element\_by\_name('username')

print("My input element is:")

print(username)

driver.close()

Locating elements by XPath

XPath stands for XML path language. It uses path expressions to identify and navigate nodes in an XML document.

As HTML is an implementation of XML, the power of XPath can be extended to HTML.

XPath is used to locate elements when a specific ID or name attribute is not available for a node. It can be used to locate nodes in absolute or relative terms.

In the case of an absolute path, the path of the name is specified right from the root node.

In the case of relative paths, the path of the node is specified relative to another node that has an identifying attribute.

Both XPath search can be compromised with a slight change on the HTML structure of the document. However, the relative XPath is better over the absolute one.

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to locate the form element with ID loginForm using XPath:

from selenium import webdriver

driver= webdriver.Firefox()

driver.get(“file:///C:/Users/rick.yamamoto/Desktop/Training/Python/Python%20Automation%20and%20Testing/Ex\_Files\_Python\_Automation\_Testing/Ex\_Files\_Python\_Automation\_Testing/Exercise%20Files/CH02/html\_code\_02.html”)

login\_form\_absolute = driver.find\_element\_by\_xpath("/html/body/form[1]")

//\* Uses the absolute path to search for the 1st form element

login\_form\_relative = driver.find\_element\_by\_xpath("//form[1]")

//\* searches for the 1st form element on the 3rd level

//\* find\_element\_by\_xpath("///form[1] will search on the 4th level.

login\_form\_id = driver.find\_element\_by\_xpath("//form[@id='loginForm']")

//\* finds a form element with id=”loginForm” on the 3rd level

print("My login form is:")

print(login\_form\_absolute)

print(login\_form\_relative)

print(login\_form\_id)

driver.close()

Locating elements by the HTML class Attribute:

A class is used to group a number of elements that have the same format or the same transition or have the same form, color, or anything in common

Use this method to locate elements when we know the class attribute of one or more elements.

From the HMTL code:

<h3 class="content"> This is a form example</h3>

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to locate the form element by class:

from selenium import webdriver

driver= webdriver.Firefox()

driver.get(“file:///C:/Users/rick.yamamoto/Desktop/Training/Python/Python%20Automation%20and%20Testing/Ex\_Files\_Python\_Automation\_Testing/Ex\_Files\_Python\_Automation\_Testing/Exercise%20Files/CH02/html\_code\_02.html”)

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

//\* returns the first element with a class=’content’

print("My class element form is:")

print(content)

print(login\_form\_relative)

print(login\_form\_id)

driver.close()

Navigating Through Pages

Page Interaction: (type in the search box and submit)

Interacting with a page as actions would be performed by a user while they are browsing the webpage

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to locate that search box, enter out text to the search box, and perform the search on python.org

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

import time

driver= webdriver.Firefox()

driver.get(“http://python.org”);

search = driver.find\_element\_by\_name(‘q’);

search.clear();

search.send\_keys(“pycon”);

search.send\_keys(Keys.RETURN);

time.sleep(4);

driver.close();

Filling Forms:

Selenium comes with a wide toolbox of classes and functions that can help you deal with other form elements. The select class helps you deal with select elements in the form and it also comes with a submit method that works on every element in the form.

html\_code\_03\_02.html

<html>

<body>

<form id="loginForm">

<select id="numReturnSelect" name="numReturnSelect">

<option value="200">200</option>

<option value="250">250</option>

<option value="500" >500</option>

<option value="800">800</option>

<option value="15000">15000</option>

<option value="85000">85000</option>

</select>

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

</form>

</body>

<html>

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to locate that search box, enter out text to the search box, and perform the search on one of the exercise file called html\_code\_03\_02.html:

from selenium import webdriver

import time

from selenium.webdriver.support.ui import Select

driver= webdriver.Firefox()

driver.get(“file:////C:\Users\rick.yamamoto\Desktop\Training\Python\Python Automation and Testing\Ex\_Files\_Python\_Automation\_Testing\Ex\_Files\_Python\_Automation\_Testing\Exercise Files\CH03\03\_02\ html\_code\_03\_02.html”);

select = Select(driver.find\_element\_by\_name(‘numReturnSelect’));

select.select\_by\_index(4)

//\* index 4 is “1500”

time.sleep(2);

select.select\_by\_visible\_text(“200”)

time.sleep(2);

select.select\_by\_value(“250”);

time.sleep(2);

options = select.options

print(options)

//\* prints all option of the Select form

submit\_button = driver.find\_element\_by\_name(“continue)

submit\_button.submit();

//\* submit the html form

time.sleep(2);

driver.close();

Drag and Drop

sites performing actions such as entering text and filling forms sometimes allow drag and drop.

In the Selenium WebDriver you can drag and drop a source element to a target element and drag and drop a source element by a specific x-offset and a specific y-offset. Let's look at a code example to see how this can be done.

ActionChains is a class provided by the Selenium WebDriver which enables us to automate actions such as drag and drop, hover, etc.

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to drag and drop

from selenium import webdriver

import time

from selenium.webdriver import ActionChains

driver= webdriver.Firefox()

driver.get(“http://jqueryuicom/droppable”);

driver.switch\_to\_frame(0)

//\* switch the frame to the first one so that we can access the drag and drop interface directly.

action\_chains = ActionChains(driver)

source = driver.find\_element\_by\_id(‘draggable’)

target = driver.find\_element\_by\_id(“droppable’)

action\_chains.drag\_and\_drop\_by\_offset(source, 100, 100).perform()

//\* I want to drag my source element by an x-offset of 100 and a y-offset of 100. And then I'm going to call the perform method to perform or execute these actions by the action chains.

//\* This actually move the source to 100px,100x down,right

time.sleep(2)

actions\_chains.drag\_and\_drop(source,target).perform()

//\* call drag and drop from the source element to the target element

time.sleep(2)

driver.close();

Waits

When a webpage is loaded by the browser, the elements may be loaded at different time intervals due to Ajax.

Some elements (image, or video) may take much longer to load as compared to others. This poses a problem while locating elements on a page.

If an element is not found by a script at that time, an exception is raised, and your script will run into an issue and stop.

Waiting adds a time interval between the actions performed by the WebDriver. That is, it adds a wait between locating elements and performing operations on them.

Explicit Wait

An explicit wait stops execution until a certain condition is satisfied.

if you're testing an image and you want to wait for the presence of that image to be loaded, then you will add and explicit wait.

So it is always better to add explicit waits when expecting certain elements to take much longer to load than the other elements in the webpage, and you want to wait for those elements because they're essential to your script being able to run.

A combination of the classes WebDriverWait and ExpectedConditions are used to add explicit waits.

These convenient methods could test conditions ranging from the title list to visibility, an element being located, an alert being present, text being present, and much more.

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to add explicit waits. I'm going to add a wait of 10 seconds. If the element is loaded within 10 seconds, then this script proceeds. In the case that the element is not loaded within 10 seconds a timeout exception is thrown:

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://python.org);

try:

element = WebDriverWait(driver, 10).until(EC.presence\_of\_element\_located((By.ID, “Start-shell”))

finally:

driver.quit()

//\* If the element “Start-shell” takes within 10 seconds, the script has not timed out and proceeded to close the driver. In the event that the element would not have been loaded within 10 seconds, the script would have thrown an element not found exception

Implicit Wait

An implicit wait polls the DOM for a given amount of time while trying to locate a specific element.

The typical use case for implicit waits is when you have a slow internet connection and you know that every element in the site will load much slowly than expected

So, you add an implicit wait before every call and If the element is loaded within the specified time, then the script moves on. If the element is not loaded within the specified amount of time, then an exception is thrown.

The main difference between implicit and explicit waits is that explicit wait, waits for a certain condition only. Implicit wait waits before every call that is made from your script.

Run the following script on your Python IDLE. We are using Visual Studio Code as IDLE. We are going to ensure that every call for every element waits for at least ten seconds before the script moves on. If it is not loaded within ten sections, then an exception is thrown.

from selenium import webdriver

driver= webdriver.Firefox()

driver .implicitly\_wait(10)

//\* This will ensure that every call for every element waits for at least ten seconds before the script moves on. If it is not loaded within ten sections, then an exception is thrown.

driver.get(“http://python.org);

myDynamicElement = driver.find\_element\_by\_id(‘start-shell’)

//\* I'm looking for an element with a specific ID, (keyboard clicking) and the ID of the element is start-shell. (keyboard clicking) So here again, first it will wait for the site to be loaded for 10 seconds, and then it will wait for the element to load for ten seconds.

driver.close()