Selenium WebDriver

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

About the Tutorial

Selenium Webdriver is a robust tool for testing the front end of an application and to perform tasks on the browser. Selenium tests can be created in multiple programming languages like Python, Java, and so on. This tutorial shall provide you with a detailed understanding on Selenium in Python language and its salient features.

Audience

This tutorial is designed for professionals working in software testing who want to improve their knowledge on front end testing. The tutorial contains a good amount of hands-example on all important topics in Selenium with Python.

Prerequisites

Before going through this tutorial, you should have knowledge on Python programming. Also, understanding software testing is needed to start with this tutorial.

Copyright & Disclaimer

Table of Contents

1. Selenium Webdriver – Introduction

Selenium Webdriver is a robust tool for testing the front end of an application and to perform tasks on the browser. Selenium tests can be created in multiple programming languages like Python, Java, C#, JavaScript, and so on.

Selenium with Python combination is comparatively easy to understand and it is short in verbose. The APIs available in Python enable us to create a connection with the browser using Selenium.

Selenium provides various Python commands which can be used for creating tests for different browsers like Chrome, Firefox, IE, and so on. It can be used in various platforms like Windows, Mac, Linux, and so on.

Reasons to learn Selenium with Python

- Python is easier to learn and compact in terms of programming.
- While creating tests in Selenium with Java, we have to take care of the beginning and ending braces. In Python, simply code indentation needs to be taken care of.
- Tests developed in Selenium with Python run faster than those written in Java.

Reasons to learn Selenium Webdriver

The reasons to learn Selenium Webdriver are mentioned below:

- It is open source and comes without any licensing cost.
- It can perform mouse and keyboard actions like drag and drop, keypress, click and hold, and so on.
- It has a very friendly API.
- It can be integrated with frameworks like TestNG and JUnit, build tools like Maven, continuous integration tools like Jenkins.
- It has a huge community support.
- It can execute test cases in headless mode.

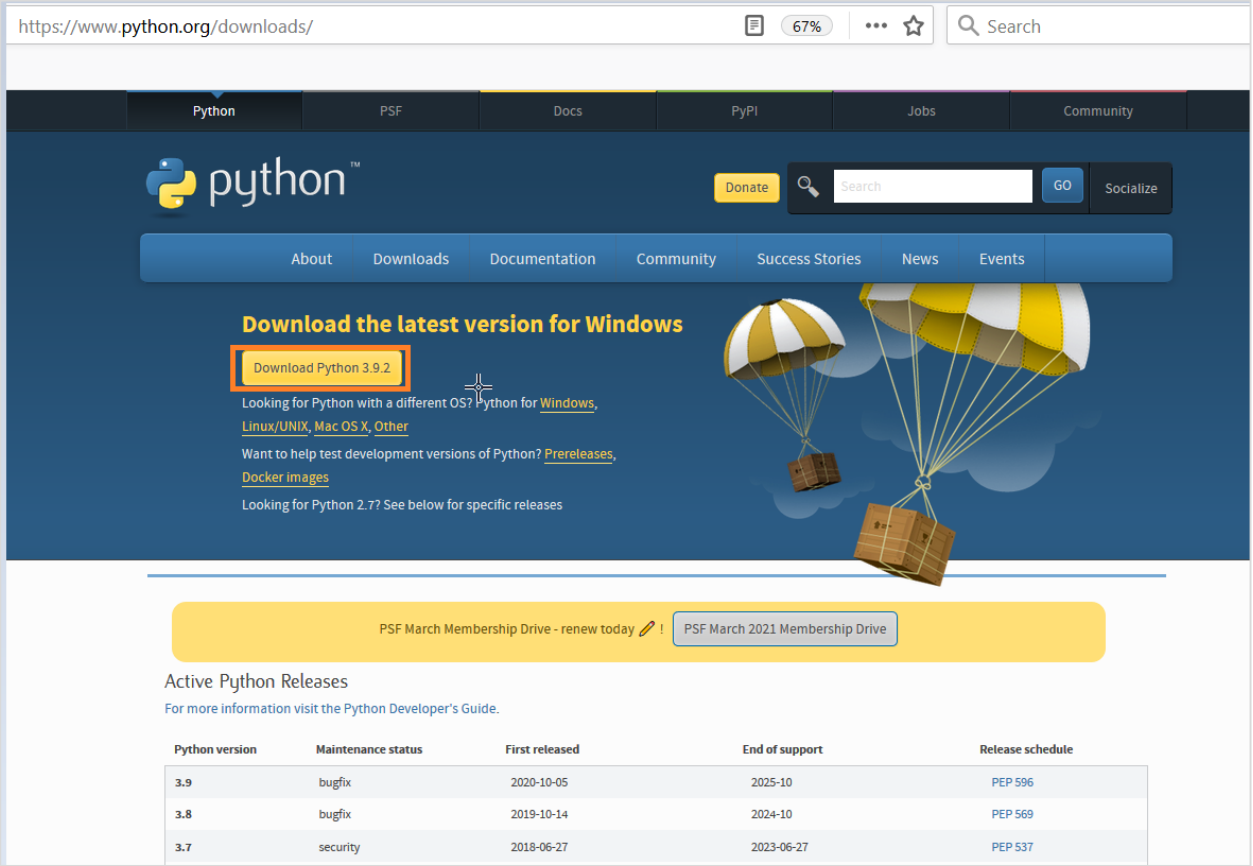
2. Selenium Webdriver — Installation

The installation and setup of Selenium webdriver in Python can be done with the steps listed below:

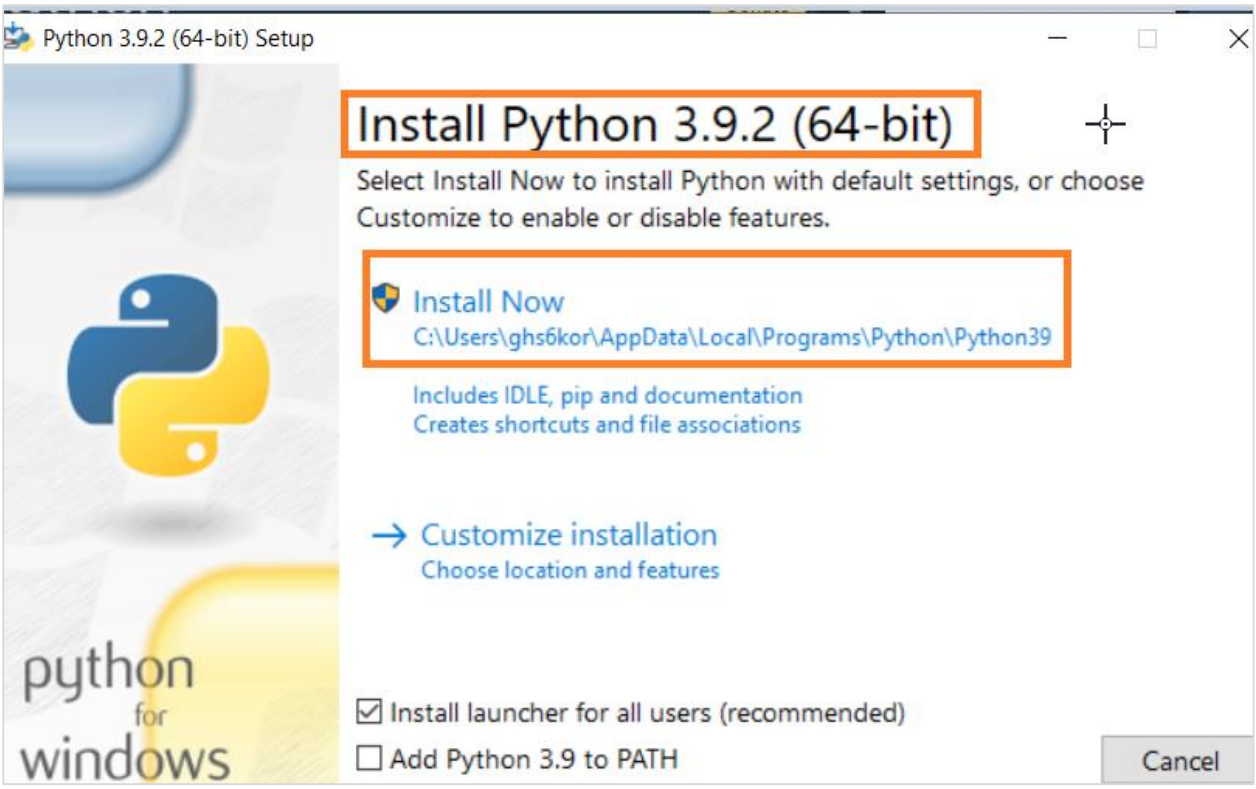
Step 1: Navigate to the site having the below link:

https://www.python.org/downloads/

Step 2: Click on the Download Python <version number> button.



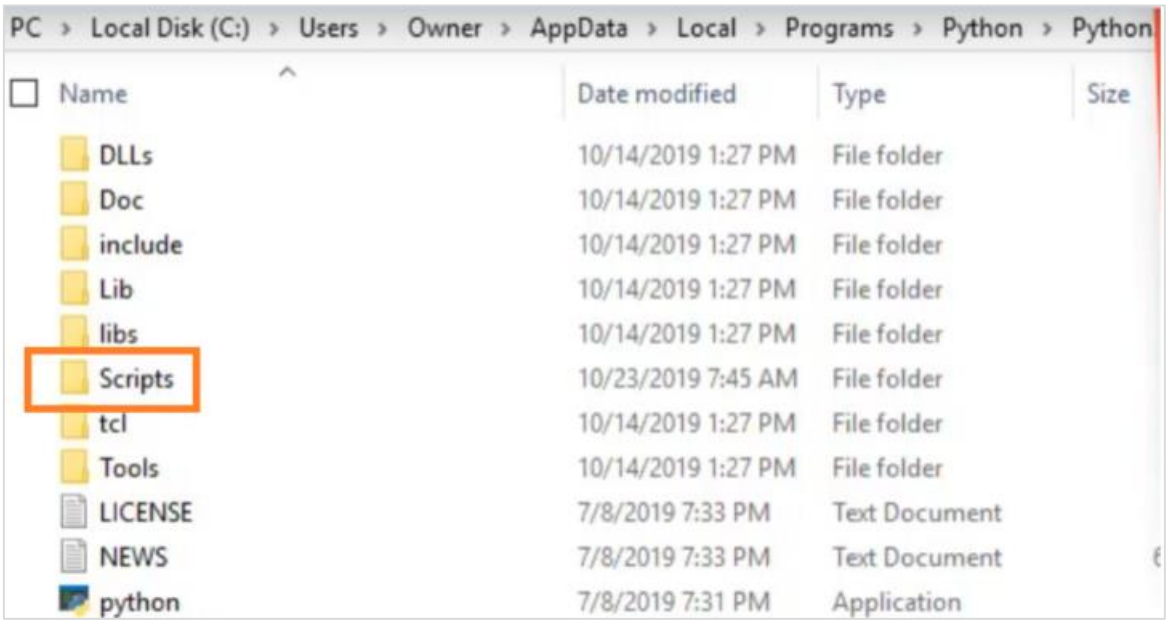
Step 3: The executable file for Python should get downloaded in our system. On clicking it, the Python installation page should get launched.



Step 4: Python should be downloaded in the following path:

C:\Users\<User>\AppData\Local\Programs\Python\Python<version>

Step 5: For the Windows users, we have to configure the path of the Python and the Scripts folder (created inside the Python folder) in the Environment variables.



Step 6: To check if Python has successfully installed, execute the command: python --version. The Python version should get displayed.

Step 7: For the Selenium bindings installation, run the command mentioned below:

```
pip install selenium.
```

Step 8: A new folder called the Selenium should now be generated within the Python folder. To upgrade to the latest Selenium version, run the command given below:

```
pip install --U selenium.
```

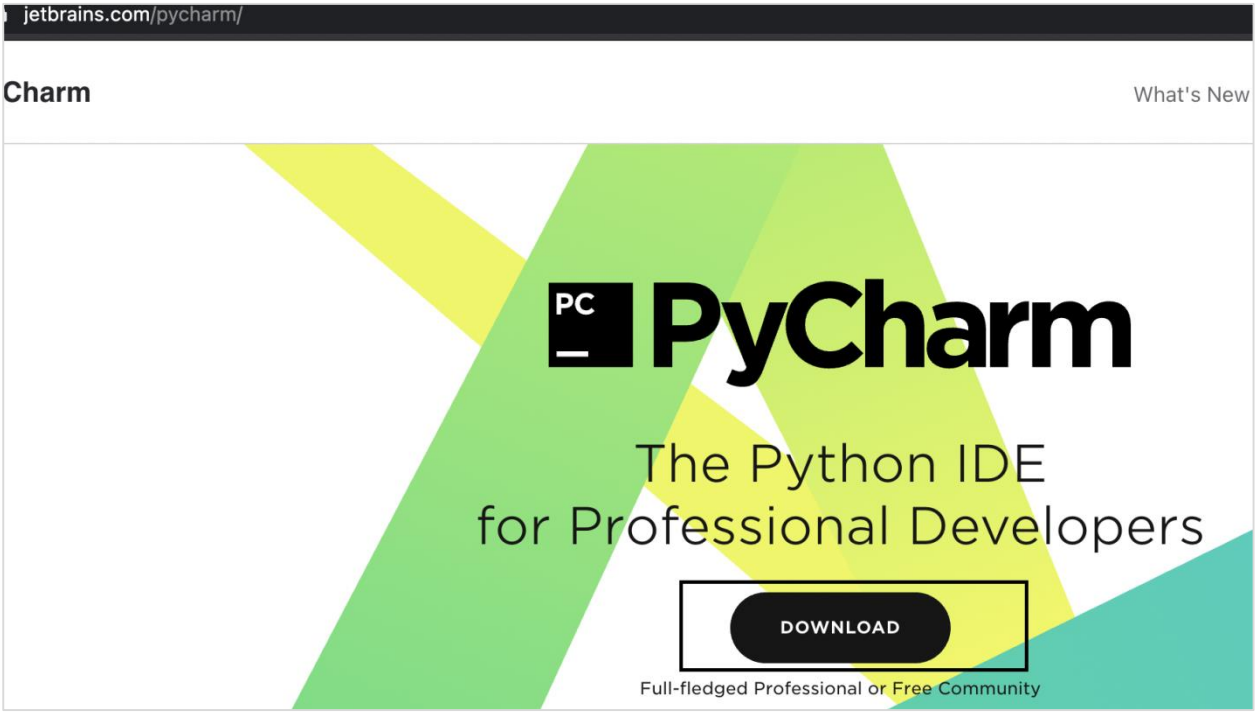
Step 9: To verify if Selenium has been installed properly, execute the command mentioned below:

```
pip show Selenium.
```

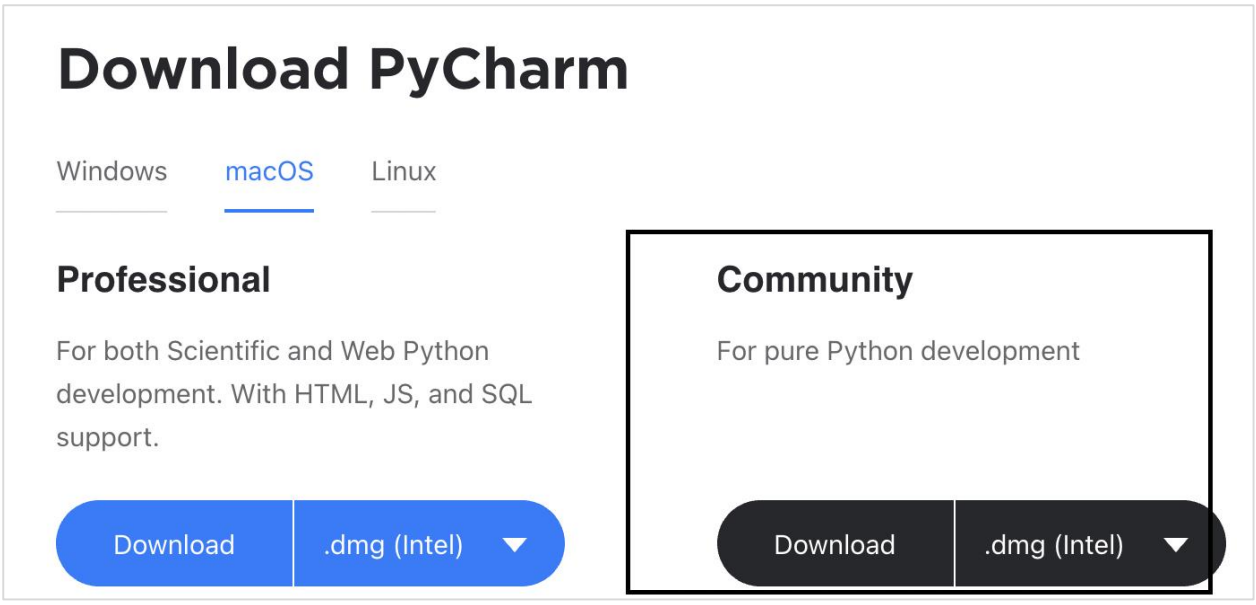
Step 10: Next, we have to download the Python editor called PyCharm from the below link:

https://www.jetbrains.com/pycharm/

Step 11: Click on Download.

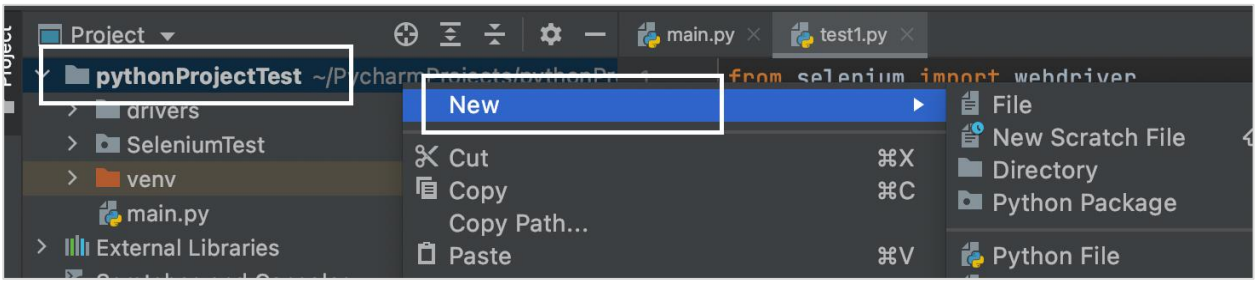


Step 12: For Selenium webdriver in Python, click on the Download button which is below the Community version (free for use).

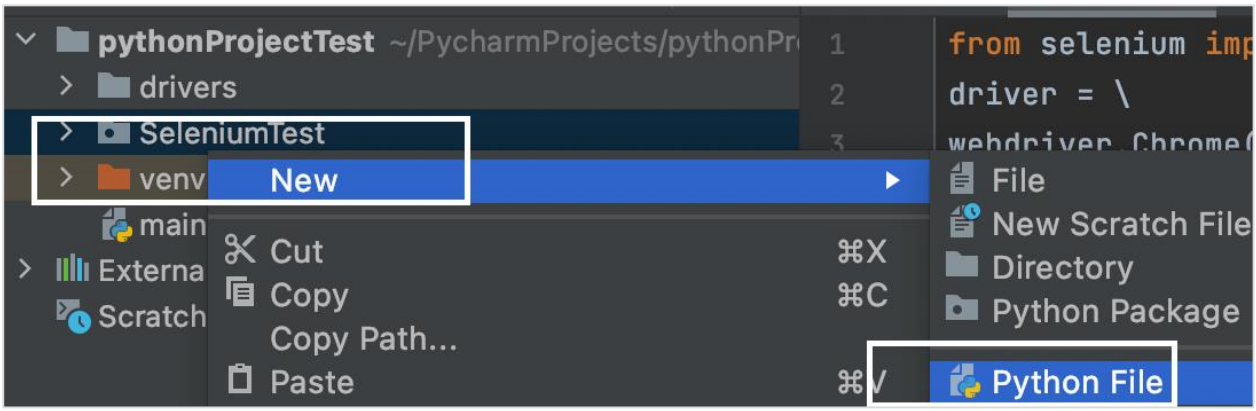


Step 13: After installation of PyCharm, we have to create a new project from File -> New Project -> Give a project name, say pythonProjectTest. Then, click on **Create**.

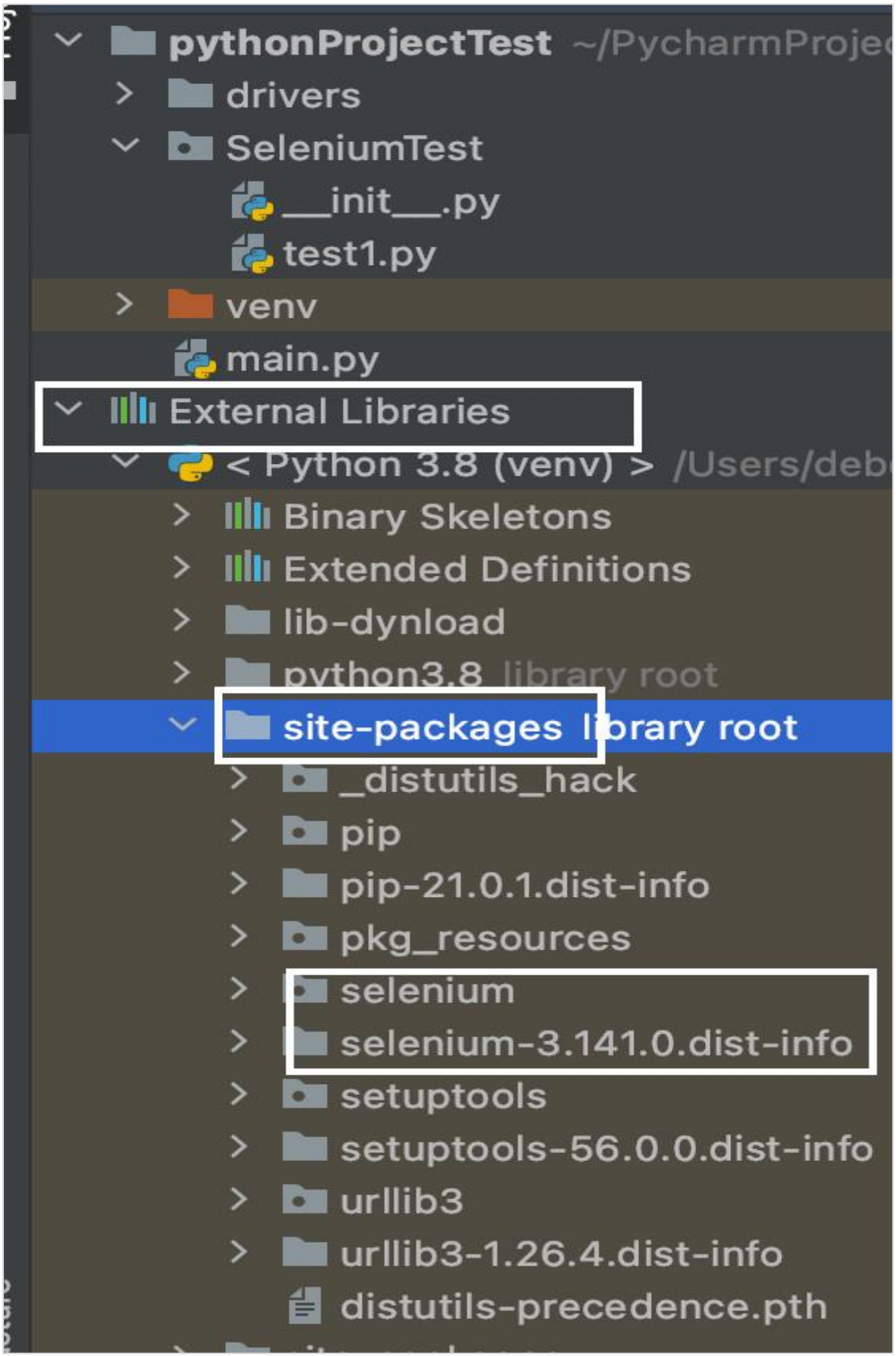
Step 14: We have to create a Python package by right-clicking on the new project we created in Step13, click on New then select Python Package. Give a package name, say SeleniumTest and proceed.



Step 15: We have to create a Python file by right-clicking on the new package we created in Step 14, click on New then select Python File. Give a package name, say test1.py and proceed.



Step 16: To view the Selenium packages in our project, click on External Libraries and then expand the site-packages folder.



3. Selenium Webdriver — Browser Navigation

We can open a browser and navigate to an application with the help of Selenium webdriver in Python. This is done with the help of the get method. While automating a test, the very first step that we create is launching an application with a URL.

The **syntax** of Selenium Webdriver is as follows:

```
driver.get("<url>")
driver.get("https://www.tutorialspoint.com/index.htm")
```

For a get method, the webdriver waits till the page is completely loaded before moving to the next step. If we try to launch a web page having numerous AJAX calls, then the webdriver is unaware when the page is completely loaded.

To fix this issue, we have to apply waits in our code.

Code Implementation

The code implementation for selenium webdriver is as follows:

```
from selenium import webdriver
#set chromedriver.exe path
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/questions/index.php")
#get page title
print('Page title: ' + driver.title)
#quit browser
driver.quit()
```

Output

The output is given below:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Page title: The Best Technical Questions And Answers

Process finished with exit code 0
```

The output shows the message - **Process with exit code 0.** This means that the above Python code executed successfully. Also, the page title of the application (obtained from the driver.title method) - The Best Technical Questions and Answers get printed in the console.

4. Selenium Webdriver – Identify Single Element

Once we navigate to a webpage, we have to interact with the web elements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

By Id

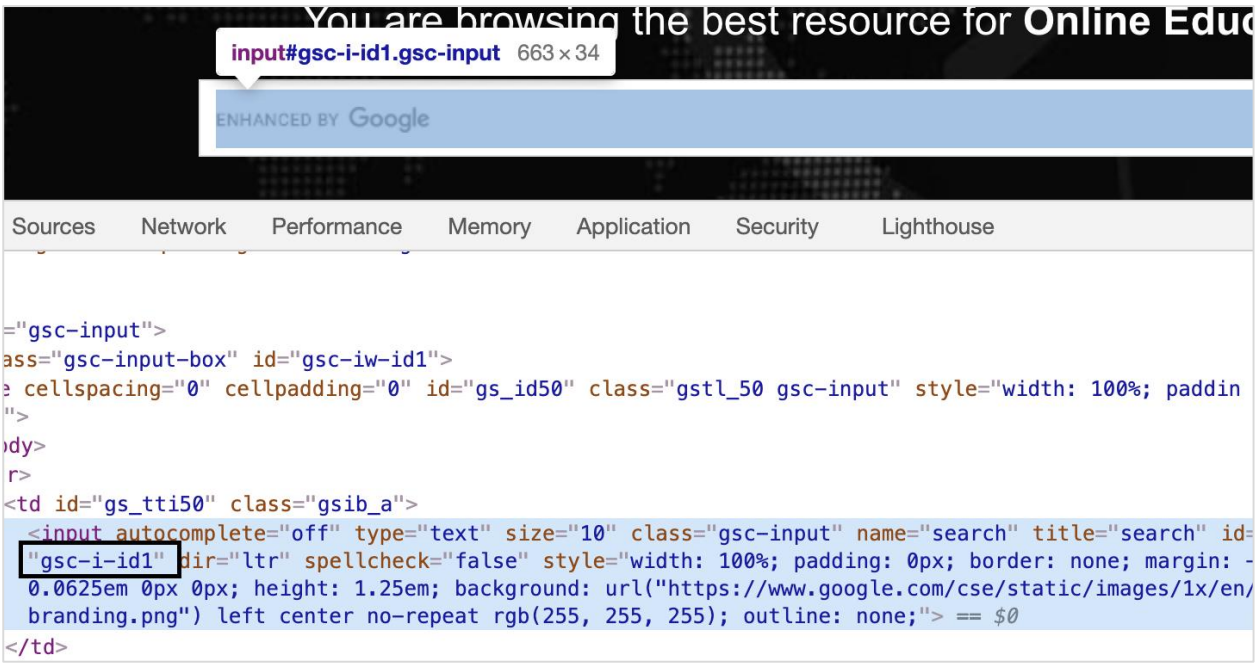
For this, our first job is to identify the element. We can use the id attribute for an element for its identification and utilize the method find_element_by_id. With this, the first element with the matching value of the attribute id is returned.

In case there is no element with the matching value of the id attribute, NoSuchElementException shall be thrown.

The **syntax** for identifying an element is as follows:

```
driver.find_element_by_id("value of id attribute")
```

Let us see the html code of a web element:



The edit box highlighted in the above image has an id attribute with value gsc-i-id1. Let us try to input some text into this edit box after identifying it.

Code Implementation

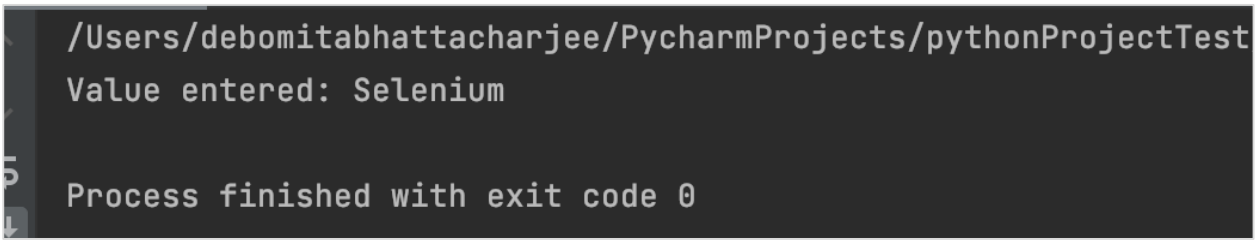
The code implementation of identifying a web element is as follows:

```
from selenium import webdriver

#set chromedriver.exe path
```

```
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify edit box with id
l = driver.find_element_by_id('gsc-i-id1')
#input text
l.send_keys('Selenium')
#obtain value entered
v = l.get_attribute('value')
print('Value entered: ' + v)
#driver quit
driver.quit()
```

Output

The output is given below:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium gets printed in the console.

By Name

Once we navigate to a webpage, we have to interact with the web elements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

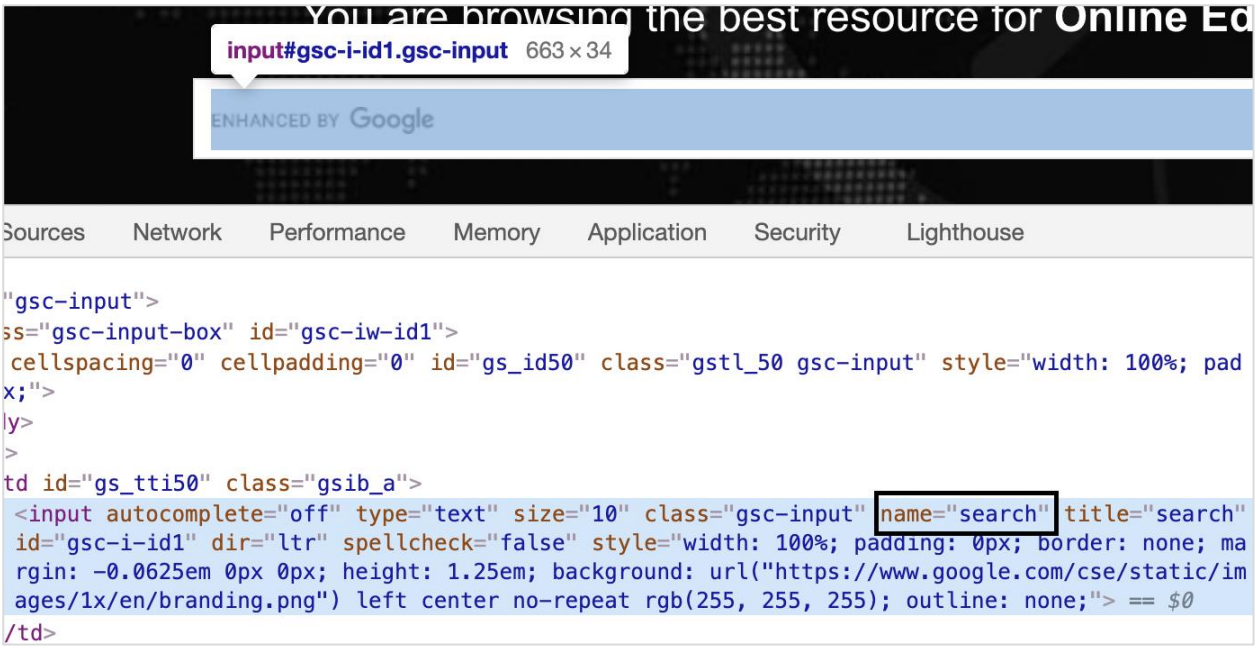
For this, our first job is to identify the element. We can use the name attribute for an element for its identification and utilize the method find_element_by_name. With this, the first element with the matching value of the attribute name is returned.

In case there is no element with the matching value of the name attribute, NoSuchElementException shall be thrown.

The **syntax** for identifying single element by name is as follows:

```
driver.find_element_by_name("value of name attribute")
```

Let us see the html code of a web element as given below:



The edit box highlighted in the above image has a name attribute with value search. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation of identifying single element by name is as follows:

```
from selenium import webdriver
#set chromedriver.exe path
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify edit box with name
l = driver.find_element_by_name('search')
#input text
l.send_keys('Selenium Java')
#obtain value entered
v = l.get_attribute('value')
print('Value entered: ' + v)
#driver close
driver.close()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Value entered: Selenium Java

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium Java gets printed in the console.

By ClassName

Once we navigate to a webpage, we have to interact with the web elements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

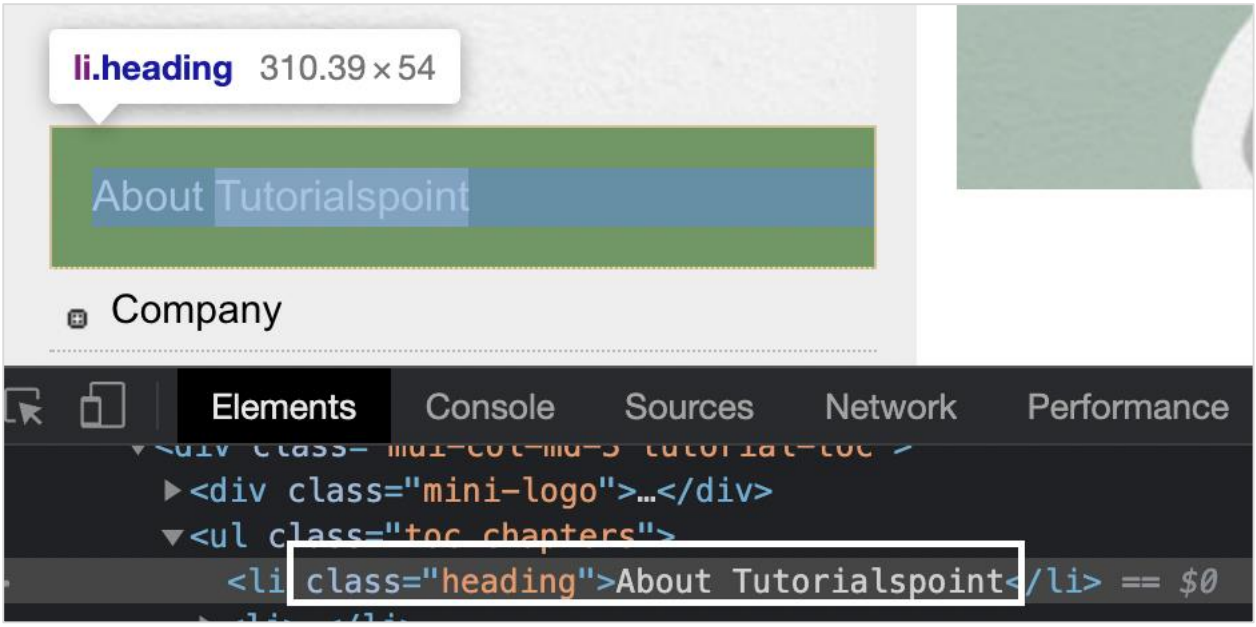
For this, our first job is to identify the element. We can use the class attribute for an element for its identification and utilise the method find_element_by_class_name. With this, the first element with the matching value of the attribute class is returned.

In case there is no element with the matching value of the class attribute, NoSuchElementException shall be thrown.

The **syntax** for identifying single element by Classname is as follows :

```
driver.find_element_by_class_name("value of class attribute")
```

Let us see the html code of a web element as given below:



The web element highlighted in the above image has a class attribute with value heading. Let us try to obtain the text of that element after identifying it.

Code Implementation

The code implementation of identifying single element by Classname is as follows:

```
from selenium import webdriver
#set chromedriver.exe path
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify edit box with class
l = driver.find_element_by_class_name('heading')
#identify text
v = l.text
#text obtained
print('Text is: ' + v)
#driver close
driver.close()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Text is: About Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the text of the webelement (obtained from the text method) - About Tutorialspoint gets printed in the console.

By TagName

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

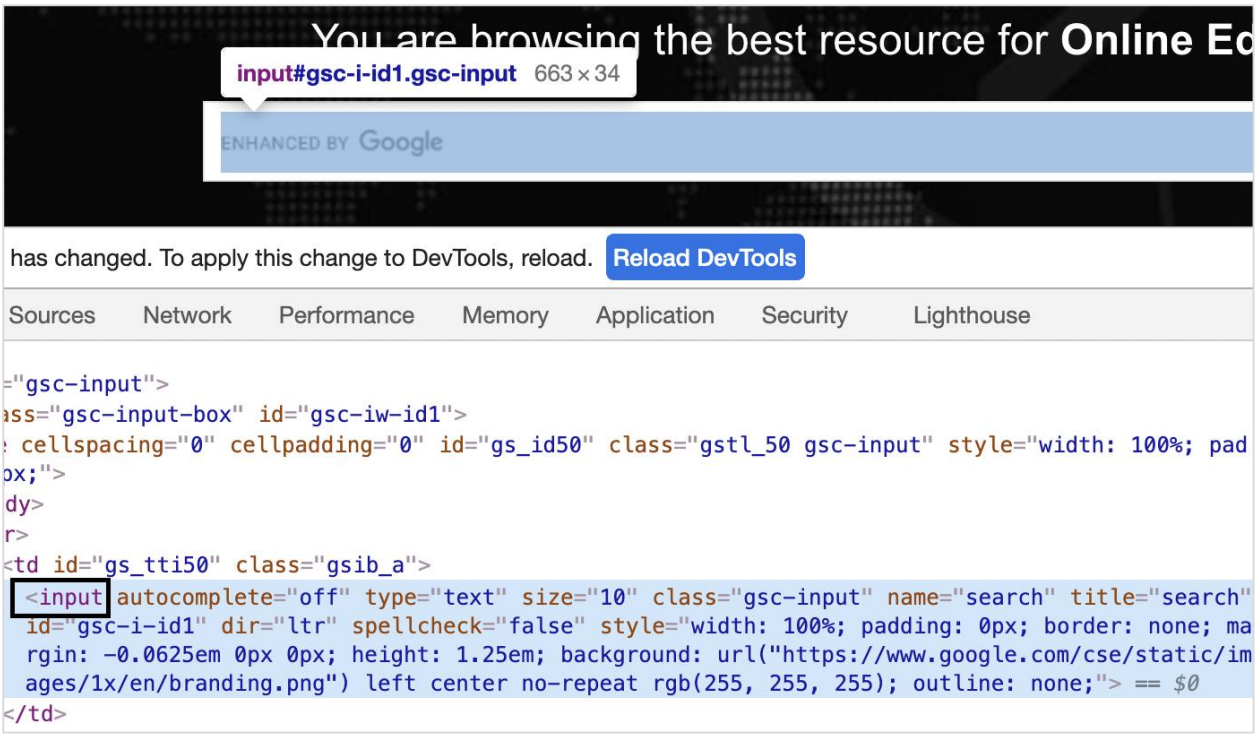
For this, our first job is to identify the element. We can use the tagname for an element for its identification and utilise the method find_element_by_tag_name. With this, the first element with the matching tagname is returned.

In case there is no element with the matching tagname, NoSuchElementException shall be thrown.

The **syntax** for identifying single element by Tagname is as follows:

```
driver.find_element_by_tag_name("tagname of element")
```

Let us see the html code of a web element as given below:



The edit box highlighted in the above image has a tagname - input. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation of identifying single element by Tagname is as follows:

```
from selenium import webdriver
#set chromedriver.exe path
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify edit box with tagname
l = driver.find_element_by_tag_name('input')
#input text
l.send_keys('Selenium Python')
#obtain value entered
v = l.get_attribute('value')
print('Value entered: ' + v)
#driver close
driver.close()
```

Output

The output is as follows

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Value entered: Selenium Python

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium Python gets printed in the console.

By Link Text

Once we navigate to a webpage, we may interact with a webelement by clicking a link to complete our automation test case. The link text is used for an element having the anchor tag.

For this, our first job is to identify the element. We can use the link text attribute for an element for its identification and utilize the method find_element_by_link_text. With this, the first element with the matching value of the given link text is returned.

In case there is no element with the matching value of the link text, NoSuchElementException shall be thrown.

The **syntax** for identifying single element by Link Text is as follows:

```
driver.find_element_by_link_text("value of link text")
```

Let us see the html code of a web element as given below:

The link highlighted in the above image has a tagname - a and the link text - Privacy Policy. Let us try to click on this link after identifying it.

Code Implementation

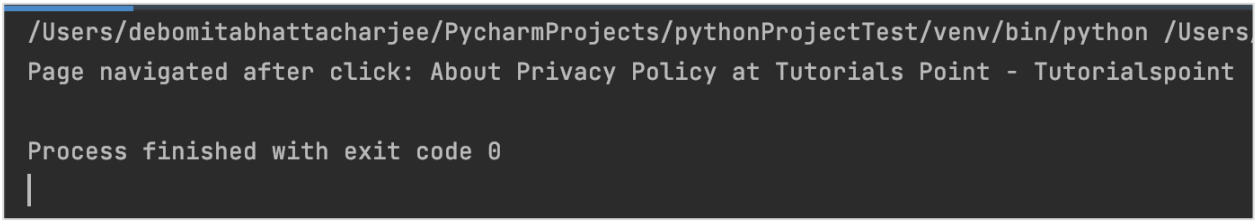
The code implementation of identifying single element by Link Text is as follows:

```
from selenium import webdriver

driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
```

```
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify link with link text
l = driver.find_element_by_link_text('Privacy Policy')
#perform click
l.click()
print('Page navigated after click: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application (obtained from the driver.title method) - About Privacy Policy at Tutorials Point - Tutorialspoint gets printed in the console.

By Partial Link Text

Once we navigate to a webpage, we may interact with a web element by clicking a link to complete our automation test case. The partial link text is used for an element having the anchor tag.

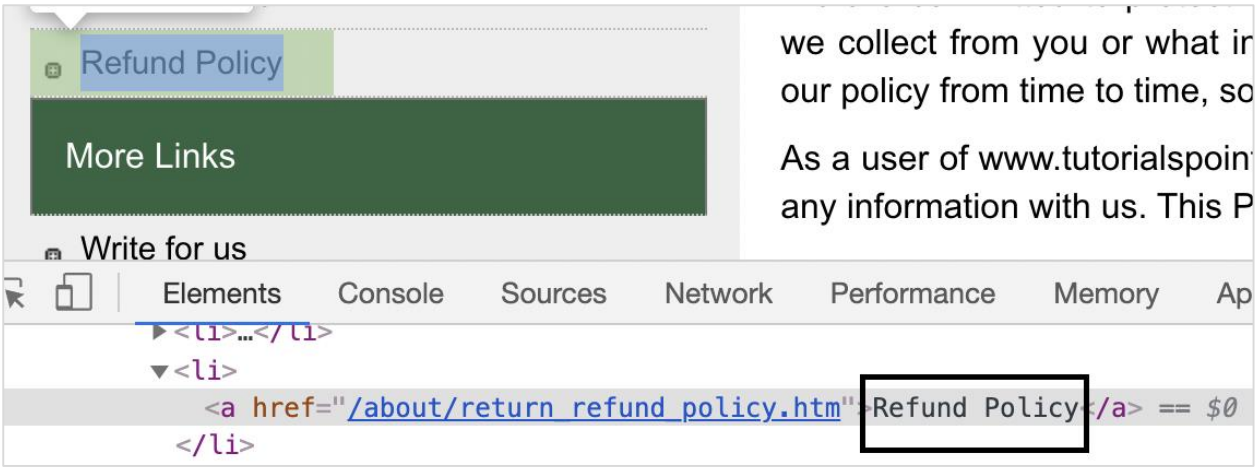
For this, our first job is to identify the element. We can use the partial link text attribute for an element for its identification and utilize the method find_element_by_partial_link_text. With this, the first element with the matching value of the given partial link text is returned.

In case there is no element with the matching value of the partial link text, NoSuchElementException shall be thrown.

The **syntax** for identifying single element by Partial Link Text is as follows:

```
driver.find_element_by_partial_link_text("value of partial ink text")
```

Let us see the html code of a web element as given below:



The link highlighted in the above image has a tagname - a and the partial link text - Refund. Let us try to click on this link after identifying it.

Code Implementation

The code implementation for identifying single element by Partial Link Text is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify link with partial link text
l = driver.find_element_by_partial_link_text('Refund')
#perform click
l.click()
print('Page navigated after click: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python /Use
Page navigated after click: Return, Refund, & Cancellation Policy - Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application (obtained from the driver.title method) - Return, Refund & Cancellation Policy - Tutorialspoint gets printed in the console.

By CSS Selector

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

For this, our first job is to identify the element. We can create a css selector for an element for its identification and use the method find_element_by_css_selector. With this, the first element with the matching value of the given css is returned.

In case there is no element with the matching value of the css, NoSuchElementException shall be thrown.

The syntax for identifying single element by CSS Selector is as follows:

```
driver.find_element_by_css_selector("value of css")
```

Rules to create CSS Expression

The rules to create a css expression are discussed below:

- To identify the element with css, the expression should be tagname[attribute='value']. We can also specifically use the id attribute to create a css expression.
- With id, the format of a css expression should be tagname#id. For example, input#txt [here input is the tagname and the txt is the value of the id attribute].
- With class, the format of css expression should be tagname.class. For example, input.cls-txt [here input is the tagname and the cls-txt is the value of the class attribute].
- If there are n children of a parent element, and we want to identify the nth child, the css expression should have nth-of –type(n).

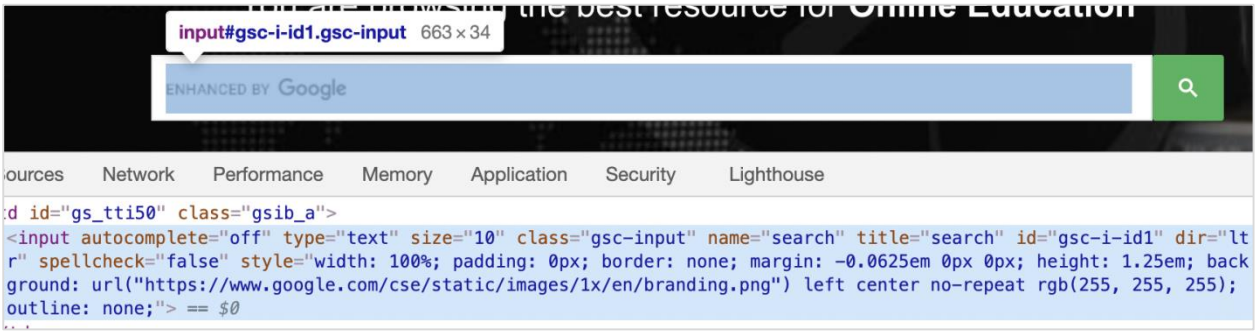
In the above code, if we want to identify the fourth li childof ul[Questions and Answers], the css expression should be ul.reading li:nth-of-type(4). Similarly, to identify the last child, the css expression should be ul.reading li:last-child.

For attributes whose values are dynamically changing, we can use ^= to locate an element whose attribute value starts with a particular text. For example, input[name^='qa'] Here, input is the tagname and the value of the name attribute starts with qa.

For attributes whose values are dynamically changing, we can use $= to locate an element whose attribute value ends with a particular text. For example, input[class$='txt'] Here, input is the tagname and the value of the class attribute ends with txt.

For attributes whose values are dynamically changing, we can use *= to locate an element whose attribute value contains a specific sub-text. For example, input[name*='nam'] Here, input is the tagname and the value of the name attribute contains the sub-text nam.

Let us see the html code of a web element as given below:



The edit box highlighted in the above image has a name attribute with value search, the css expression should be input[name='search']. Let us try to input some text into this edit box after identifying it.

Code Implementation

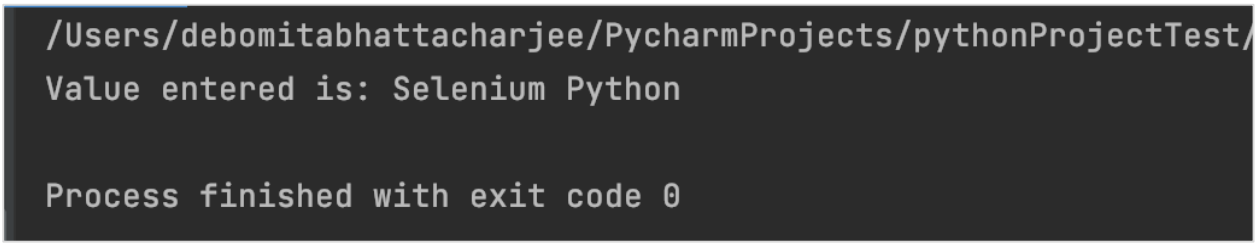
The code implementation of identifying single element by CSS Selector is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify element with css
l = driver.find_element_by_css_selector("input[name='search']")
l.send_keys('Selenium Python')
v = l.get_attribute('value')
print('Value entered is: ' + v)
```

```
#driver quit
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium Python gets printed in the console.

ByXpath

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

For this, our first job is to identify the element. We can create an xpath for an element for its identification and use the method find_element_by_xpath. With this, the first element with the matching value of the given xpath is returned.

In case there is no element with the matching value of the xpath, NoSuchElementException shall be thrown.

The syntax for identifying single element by Xpath is as follows:

```
driver.find_element_by_xpath("value of xpath")
```

Rules to create Xpath Expression

The rules to create a xpath expression are discussed below:

- To identify the element with xpath, the expression should be //tagname[@attribute='value']. There can be two types of xpath – relative and absolute. The absolute xpath begins with / symbol and starts from the root node upto the element that we want to identify.

For example,

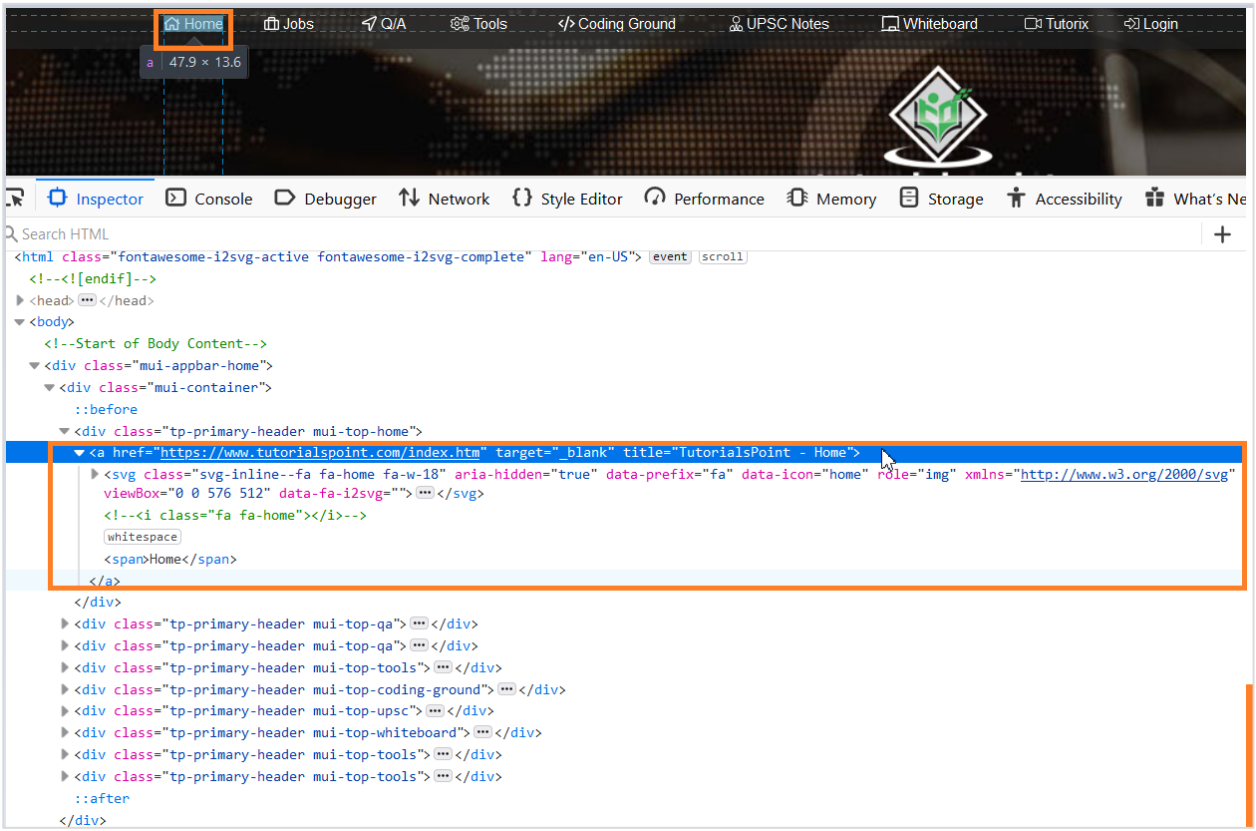
```
        /html/body/div[1]/div/div[1]/a
```

- The relative xpath begins with // symbol and does not start from the root node.

For example,

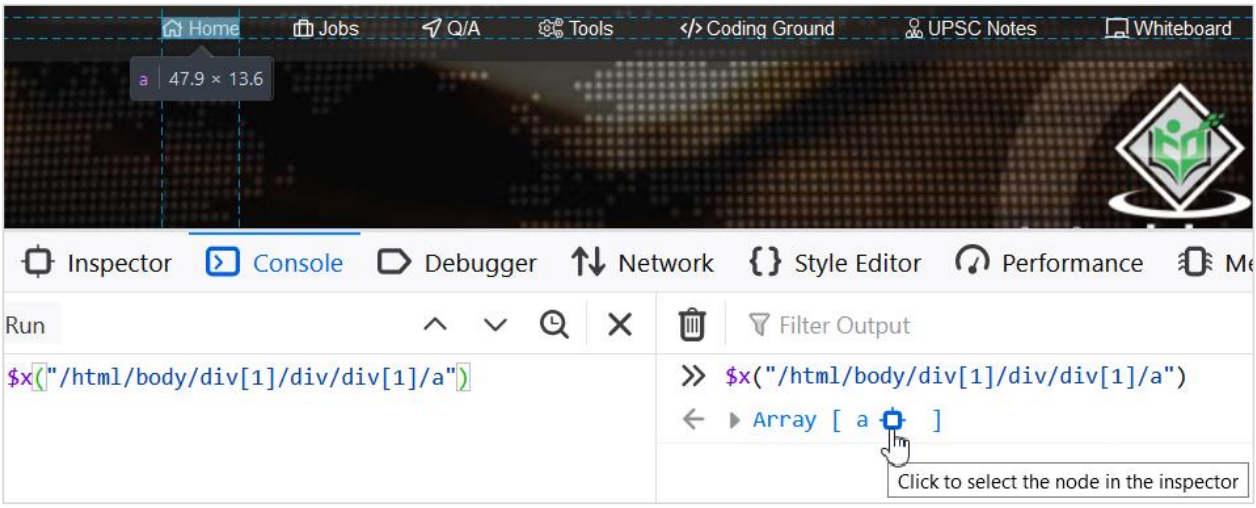
```
//img[@alt='tutorialspoint']
```

Let us see the html code of the highlighted link - Home starting from the root.



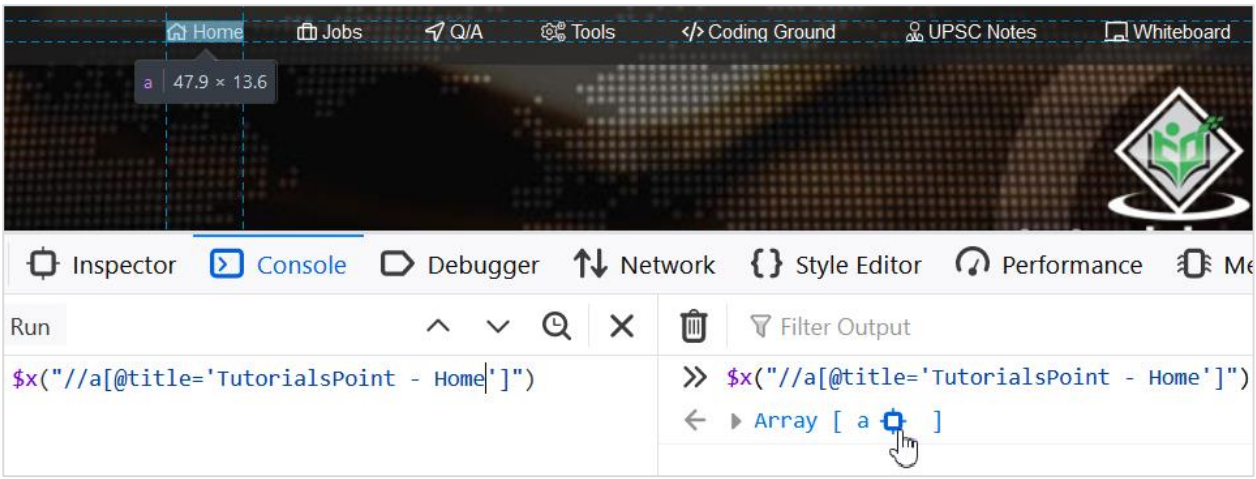
The absolute xpath for this element can be as follows:

```
/html/body/div[1]/div/div[1]/a.
```



The relative xpath for element Home can be as follows:

```
//a[@title='TutorialsPoint - Home'].
```



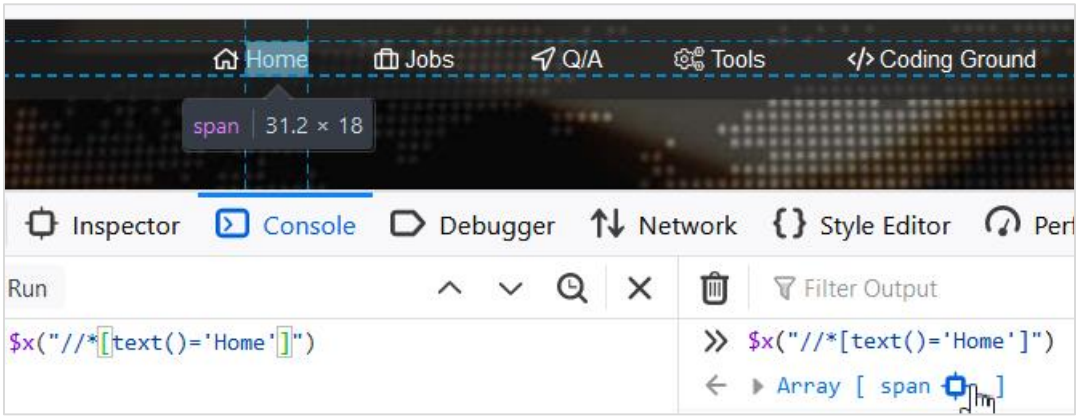
Functions

There are also functions available which help to frame relative xpath expressions.

text()

It is used to identify an element with its visible text on the page. The xpath expression is as follows:

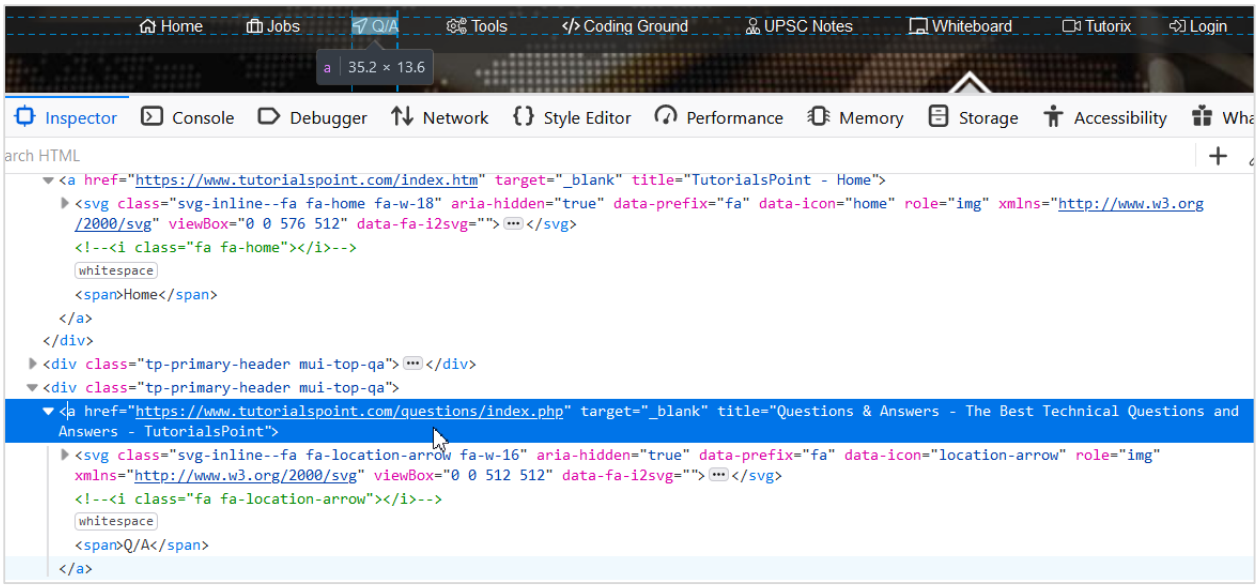
```
//*[text()='Home'].
```



starts-with

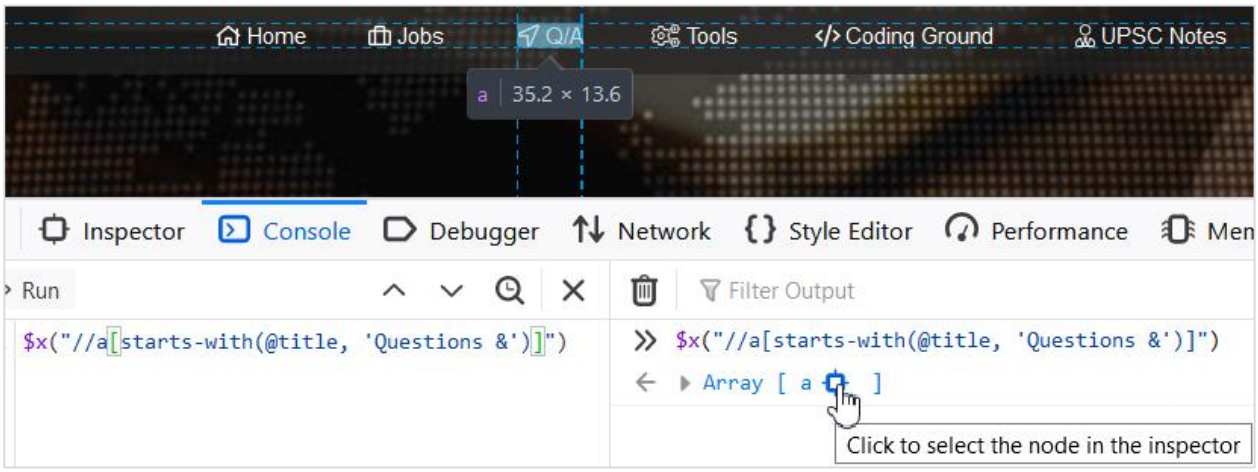
It is used to identify an element whose attribute value begins with a specific text. This function is normally used for attributes whose value changes on each page load.

Let us see the html of the link Q/A:



The xpath expression should be as follows:

```
//a[starts-with(@title, 'Questions &')].
```

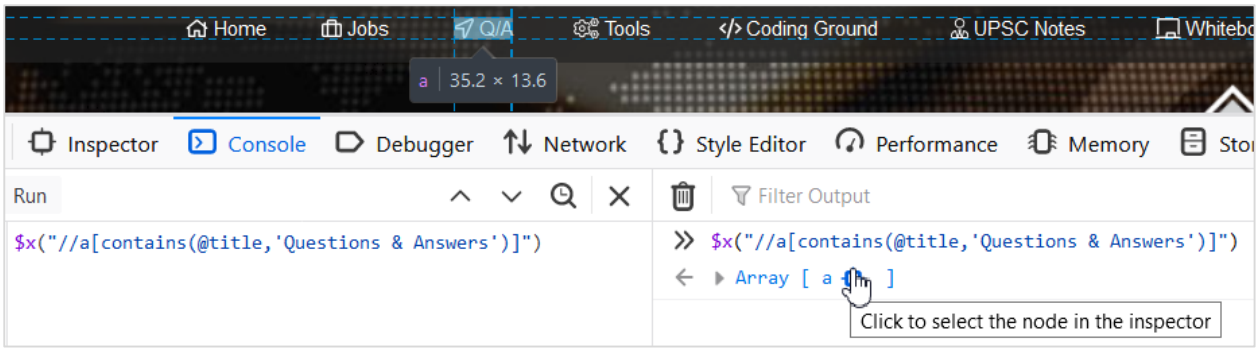


contains()

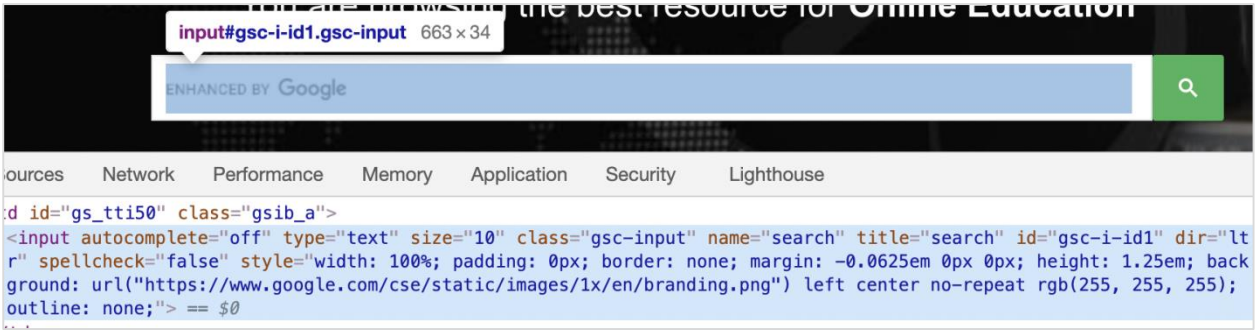
It identifies an element whose attribute value contains a sub-text. This function is normally used for attributes whose value changes on each page load.

The xpath expression is as follows:

```
//a[contains(@title, 'Questions & Answers')].
```



Let us see the html code of a webelement as shown below:



The edit box highlighted in the above image has a name attribute with value search, the xpath expression should be //input[@name='search']. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation of identifying single element by XPath is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify element with xpath
l = driver.find_element_by_xpath("//input[@name='search']")
l.send_keys('Selenium Python')
v = l.get_attribute('value')
print('Value entered is: ' + v)
#driver quit
driver.quit()
```

Output

The output is as follows

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/
Value entered is: Selenium Python

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium Python gets printed in the console.

5. Selenium Webdriver — Identify Multiple Elements

In this chapter, we will learn how to identify multiple elements by various options. Let us begin by understanding identifying multiple elements by Id.

By id

It is not recommended to identify multiple elements by the locator id, since the value of an id attribute is unique to an element and is applicable to a single element on the page.

By Class name

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

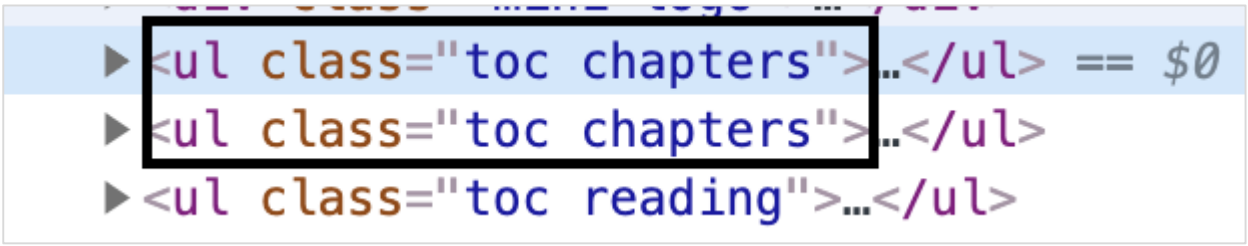
For this, our first job is to identify the elements. We can use the class attribute for elements for their identification and utilise the method find_elements_by_class_name. With this, all the elements with the matching value of the attribute class are returned in the form of list.

In case there are no elements with the matching value of the class attribute, an empty list shall be returned.

The **syntax** for identifying multiple elements by Classname is as follows:

```
driver.find_elements_by_class_name("value of class attribute")
```

Let us see the html code of webelements having class attribute as given below:



The value of the class attribute highlighted in the above image is toc chapters. Let us try to count the number of such webelements.

Code Implementation

The code implementation for identifying multiple elements by Classname is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
```

```
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify elements with class attribute
l = driver.find_elements_by_class_name("chapters")
#count elements
s = len(l)
print('Count is:')
print(s)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/
Count is:
2

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the total count of webelements having the class attributes value chapters (obtained from the len method) - 2 gets printed in the console.

By Tagname

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

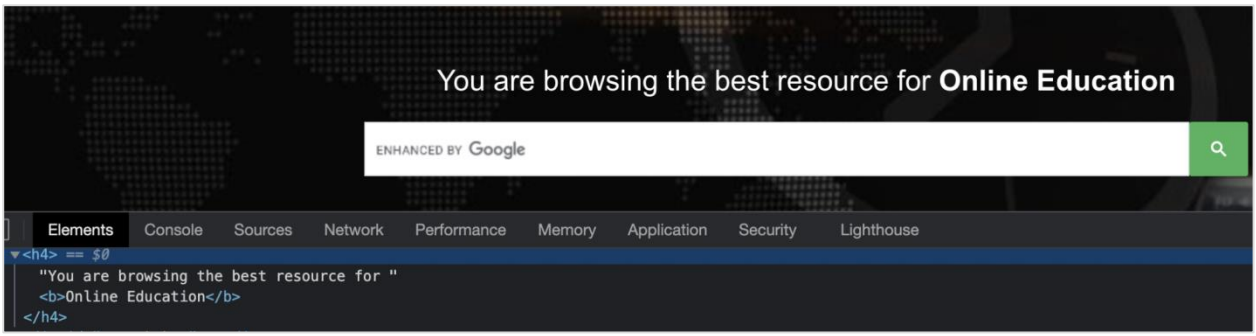
For this, our first job is to identify the elements. We can use the tagname for elements for their identification and utilise the method find_elements_by_tag_name. With this, all the elements with the matching value of the tagname are returned in the form of list.

In case there are no elements with the matching value of the tagname, an empty list shall be returned.

The **syntax** for identifying multiple elements by Tagname is as follows:

```
driver.find_elements_by_tag_name("value of tagname")
```

Let us see the html code of a webelement, which is as follows:



The value of the tagname highlighted in the above image is h4. Let us try to count the number of webelements having tagname as h4.

Code Implementation

The code implementation for identifying multiple elements by Tagname is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify elements with tagname
l = driver.find_elements_by_tag_name("h4")
#count elements
s = len(l)
print('Count is:')
print(s)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/
Count is:
1

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the total count of webelement having the tagname as h4 (obtained from the len method) - 1 gets printed in the console.

By Partial Link Text

Once we navigate to a webpage, we may have to interact with the webelements by clicking a link to complete our automation test case. The partial link text is used for elements having the anchor tag.

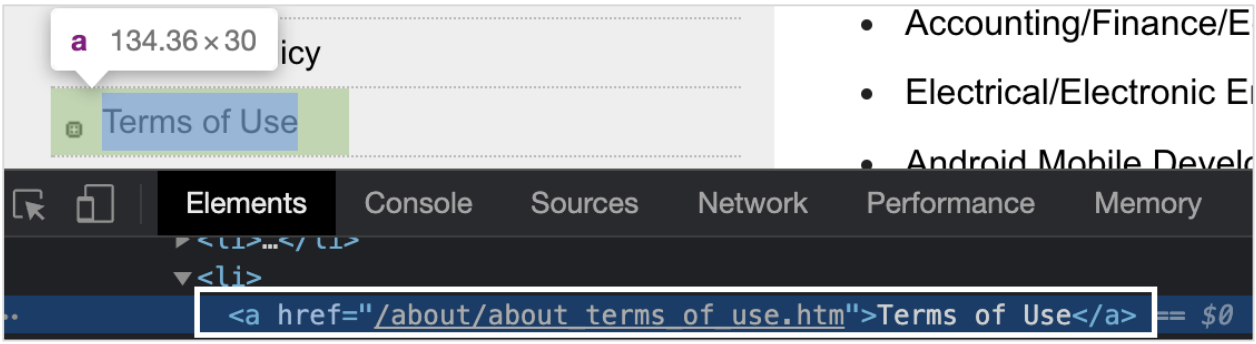
For this, our first job is to identify the elements. We can use the partial link text attribute for elements for their identification and utlize the method find_elements_by_partial_link_text. With this, all the elements with the matching value of the given partial link text are returned in the form of a list.

In case there are no elements with the matching value of the partial link text, an empty list shall be returned.

The **syntax** for identifying multiple elements by Partial Link Text is as follows:

```
driver.find_elements_by_partial_link_text("value of partial link text")
```

Let us see the html code of link, which is as follows:



The link highlighted - Terms of Use in the above image has a tagname - a and the partial link text - Terms. Let us try to identify the text after identifying it.

Code Implementation

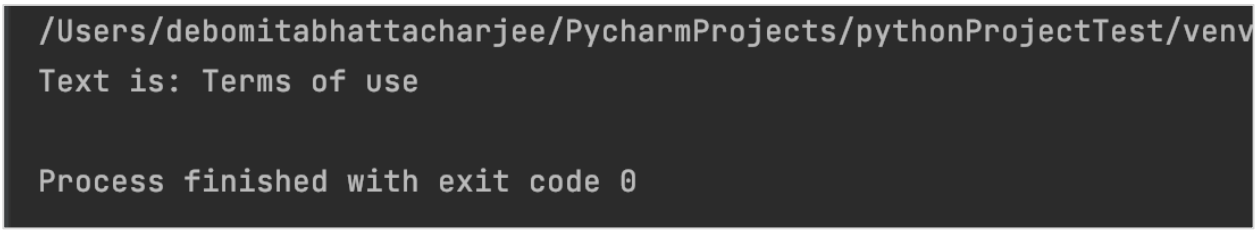
The code implementation for identifying multiple elements by Partial Link Text is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify elements with partial link text
l = driver.find_elements_by_partial_link_text('Terms')
#count elements
s = len(l)
#iterate through list
for i in l:
```

```
#obtain text
   t = i.text
print('Text is: ' + t)
#driver quit
driver.quit()
```

Output

The output is given below:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the text of the link identified with the partial link text locator (obtained from the text method) - Terms of use gets printed in the console.

By Link Text

Once we navigate to a webpage, we may have to interact with the webelements by clicking a link to complete our automation test case. The link text is used for elements having the anchor tag.

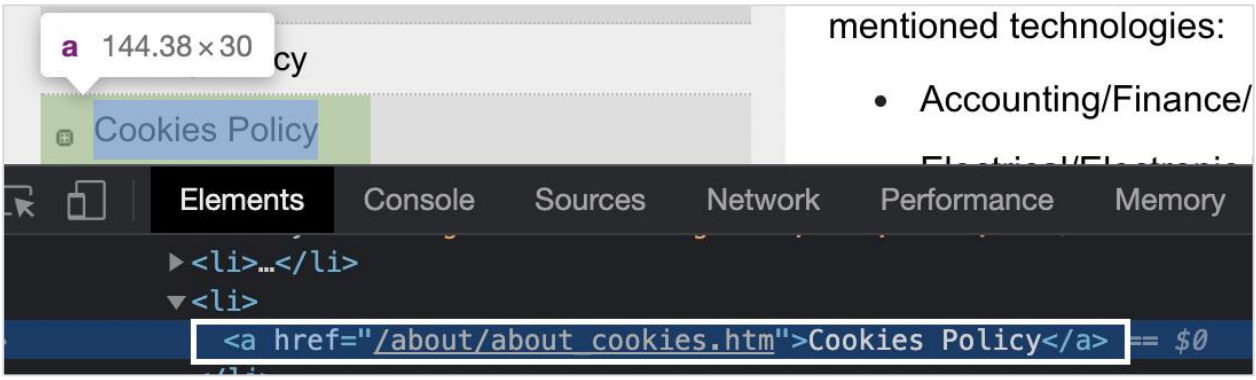
For this, our first job is to identify the elements. We can use the link text attribute for elements for their identification and utilize the method find_elements_by_link_text. With this, all the elements with the matching value of the given link text are returned in the form of a list.

In case there are no elements with the matching value of the link text, an empty list shall be returned.

The **syntax** for identifying multiple elements by Link Text is as follows:

```
driver.find_elements_by_link_text("value of link text")
```

Let us see the html code of link, which is as follows:



The link highlighted - Cookies Policy in the above image has a tagname - a and the link text - Cookies Policy. Let us try to identify the text after identifying it.

Code Implementation

The code implementation for identifying multiple elements by Link Text is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify elements with link text
l = driver.find_elements_by_link_text('Cookies Policy')
#count elements
s = len(l)
#iterate through list
for i in l:
#obtain text
   t = i.text
print('Text is: ' + t)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/
Text is: Cookies Policy

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the text of the link identified with the link text locator (obtained from the text method) - Cookies Policy gets printed in the console.

By Name

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

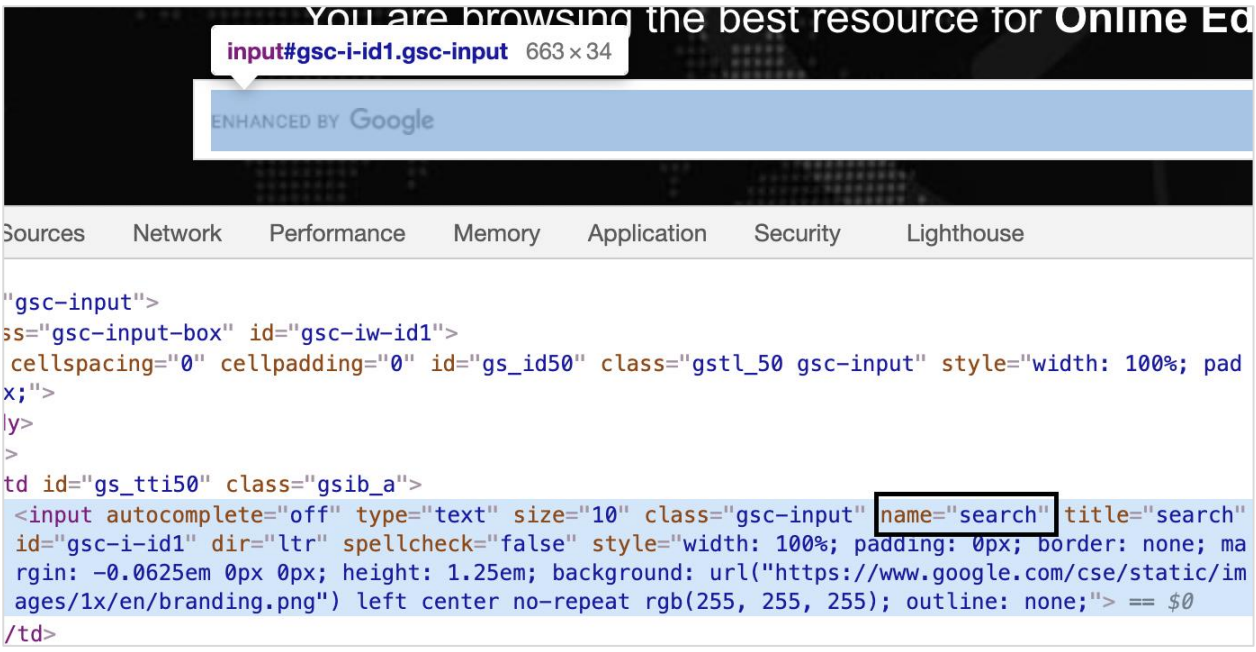
For this, our first job is to identify the elements. We can use the name attribute of elements for their identification and utilize the method find_elements_by_name. With this, the elements with the matching value of the attribute name are returned in the form of a list.

In case there is no element with the matching value of the name attribute, an empty list shall be returned.

The **syntax** for identifying multiple elements by Name is as follows:

```
driver.find_elements_by_name("value of name attribute")
```

Let us see the html code of an webelement, which is as follows:



The edit box highlighted in the above image has a name attribute with value search. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation for identifying multiple elements by Name is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify elements with name attribute
l = driver.find_elements_by_name('search')
#count elements
s = len(l)
#iterate through list
for i in l:
#obtain text
   t = i.send_keys('Selenium Python')
   v = i.get_attribute('value')
print('Value entered is: ' + v)
#driver quit
driver.quit()
```

Output

The output is as follows

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/
Value entered is: Selenium Python

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium Python gets printed in the console.

By CSS Selector

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

For this, our first job is to identify the elements. We can create a css selector for their identification and utilize the method find_elements_by_css_selector. With this, the elements with the matching value of the given css are returned in the form of list.

In case there is no element with the matching value of the css, an empty list shall be returned.

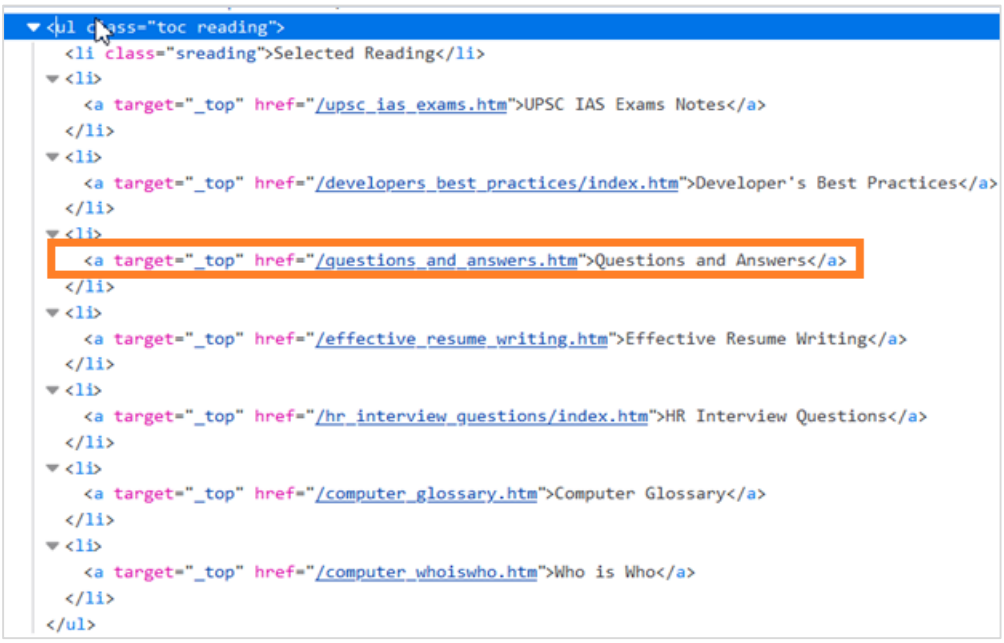
The **syntax** for identifying multiple elements by CSS Selector is as follows:

```
driver.find_elements_by_css_selector("value of css")
```

Rules for CSS Expression

The rules to create a css expression are discussed below:

- To identify the element with css, the expression should be tagname[attribute='value']. We can also specifically use the id attribute to create a css expression.
- With id, the format of a css expression should be tagname#id. For example, input#txt [here input is the tagname and the txt is the value of the id attribute].
- With class, the format of css expression should be tagname.class . For example, input.cls-txt [here input is the tagname and the cls-txt is the value of the class attribute].
- If there are n children of a parent element, and we want to identify the nth child, the css expression should have nth-of –type(n).



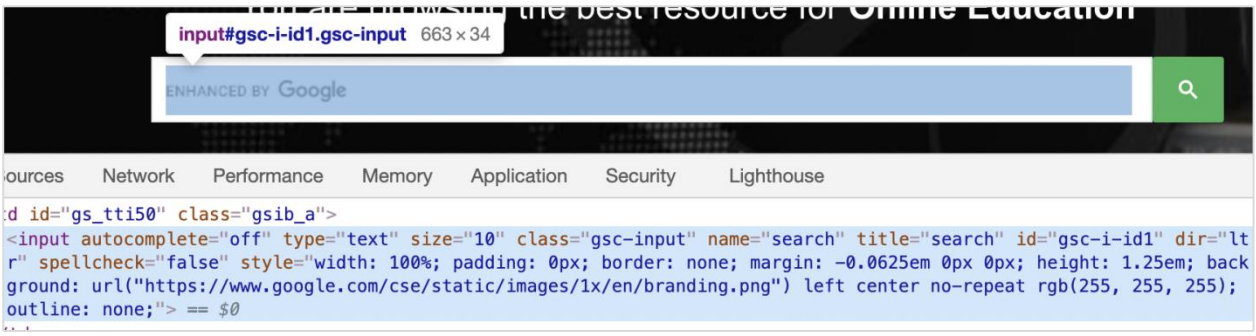
In the above code, if we want to identify the fourth li child of ul[Questions and Answers], the css expression should be ul.reading li:nth-of-type(4). Similarly, to identify the last child, the css expression should be ul.reading li:last-child.

For attributes whose values are dynamically changing, we can use ^= to locate an element whose attribute value starts with a particular text. For example, input[name^='qa'] [here input is the tagname and the value of the name attribute starts with qa].

For attributes whose values are dynamically changing, we can use $= to locate an element whose attribute value ends with a particular text. For example, input[class$='txt'] Here, input is the tagname and the value of the class attribute ends with txt.

For attributes whose values are dynamically changing, we can use *= to locate an element whose attribute value contains a specific sub-text. For example, input[name*='nam'] Here, input is the tagname and the value of the name attribute contains the sub-text nam.

Let us see the html code of a webelement:



The edit box highlighted in the above image has a name attribute with value search, the css expression should be input[name='search']. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation for identifying multiple elements by CSS Selector is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify elements with css
l = driver.find_elements_by_css_selector("input[name='search']")
#count elements
s = len(l)
#iterate through list
for i in l:
#obtain text
  t = i.send_keys('Tutorialspoint')
```

```
  v = i.get_attribute('value')
print('Value entered is: ' + v)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Value entered is: Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Tutorialspoint gets printed in the console.

By Xpath

Once we navigate to a webpage, we have to interact with the webelements available on the page like clicking a link/button, entering text within an edit box, and so on to complete our automation test case.

For this, our first job is to identify the elements. We can create an xpath for their identification and utilize the method find_elements_by_xpath. With this, the elements with the matching value of the given xpath are returned in the form of a list.

In case there is no element with the matching value of the xpath, an empty list shall be returned.

The **syntax** for identifying multiple elements by Xpath is as follows:

```
driver.find_elements_by_xpath("value of xpath")
```

Rules for Xpath Expression

The rules to create a xpath expression are discussed below:

- To identify the element with xpath, the expression should be //tagname[@attribute='value']. There can be two types of xpath – relative and absolute. The absolute xpath begins with / symbol and starts from the root node upto the element that we want to identify.

For example,

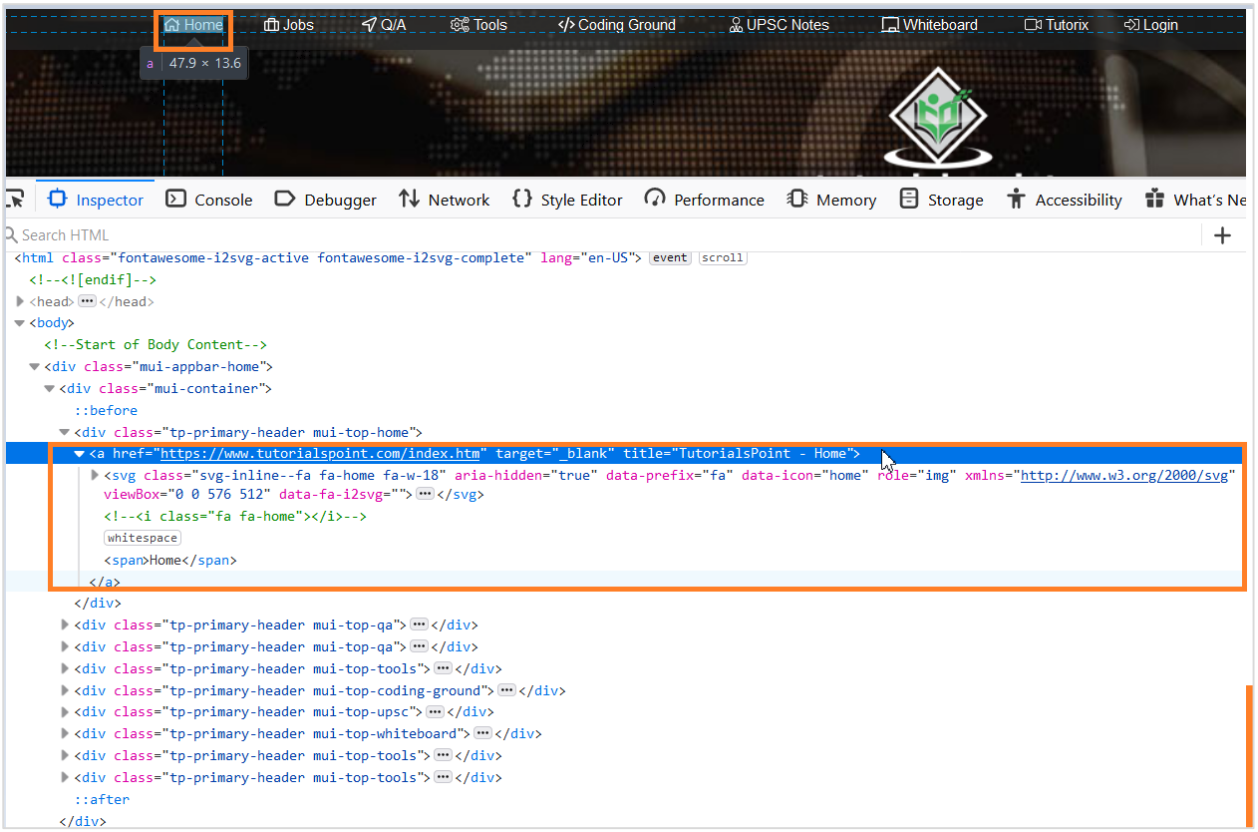
```
        /html/body/div[1]/div/div[1]/a
```

- The relative xpath begins with // symbol and does not start from the root node.

For example,

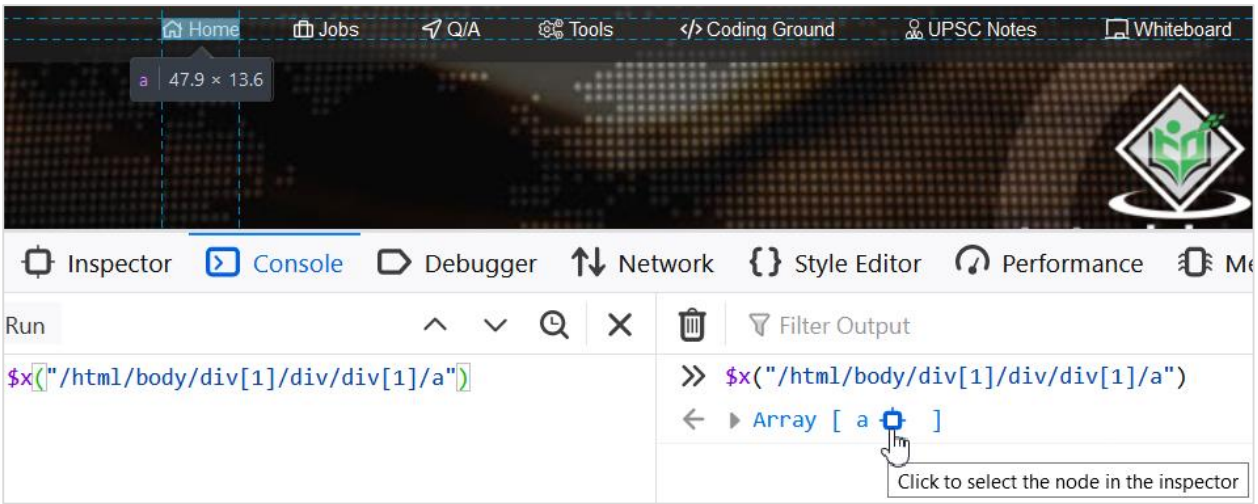
```
//img[@alt='tutorialspoint']
```

Let us see the html code of the highlighted link - Home starting from the root.



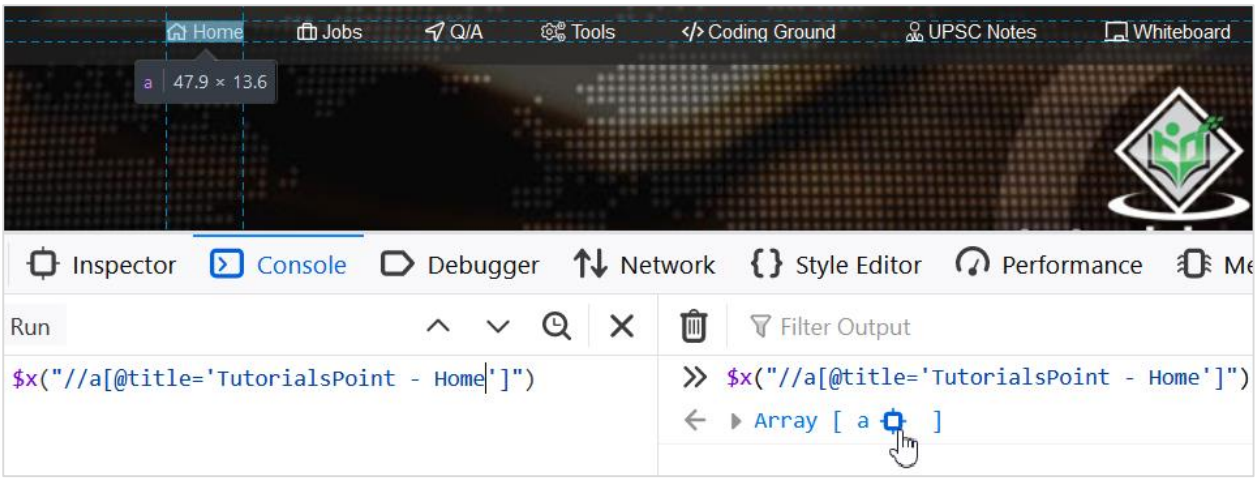
The absolute xpath for the element Home can be as follows:

```
/html/body/div[1]/div/div[1]/a.
```



The relative xpath for element Home can be as follows:

```
//a[@title='TutorialsPoint - Home'].
```



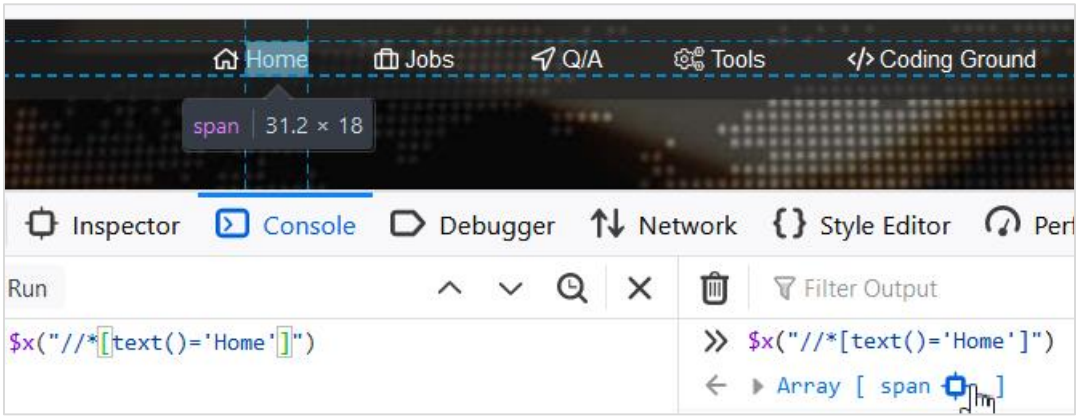
Functions

There are also functions available which help to frame relative xpath expressions:-

text()

It is used to identify an element with the help of the visible text on the page. The xpath expression is as follows:

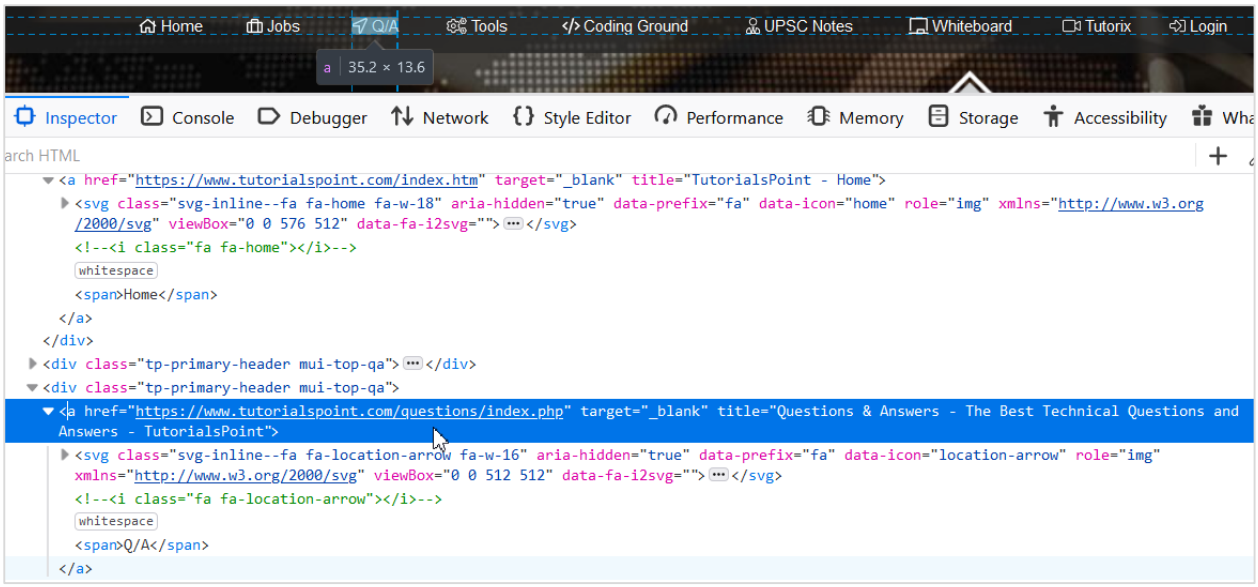
```
//*[text()='Home'].
```



starts-with

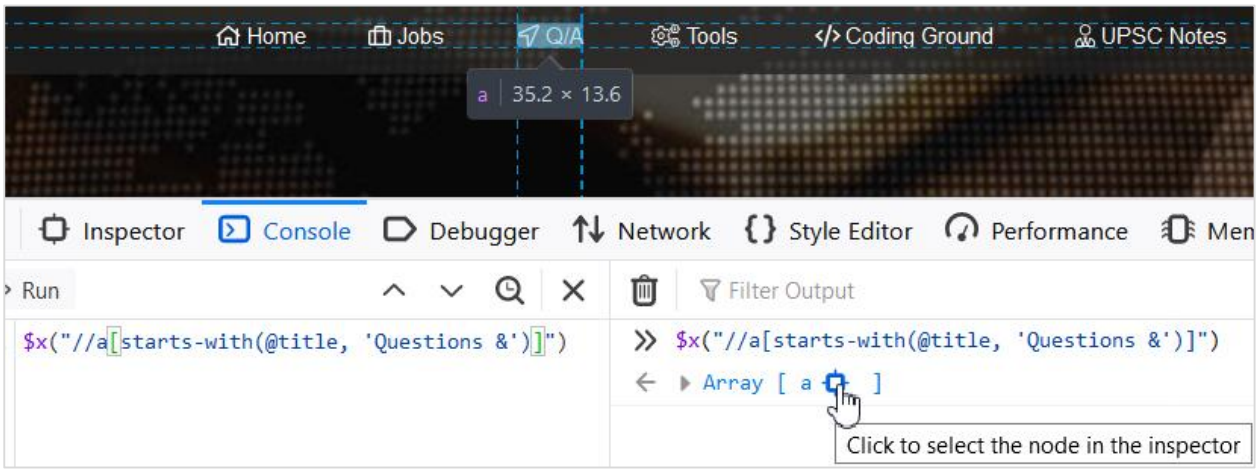
It is used to identify an element whose attribute value begins with a specific text. This function is normally used for attributes whose value changes on each page load.

Let us see the html of the element Q/A:



The xpath expression should be as follows:

```
//a[starts-with(@title, 'Questions &')].
```

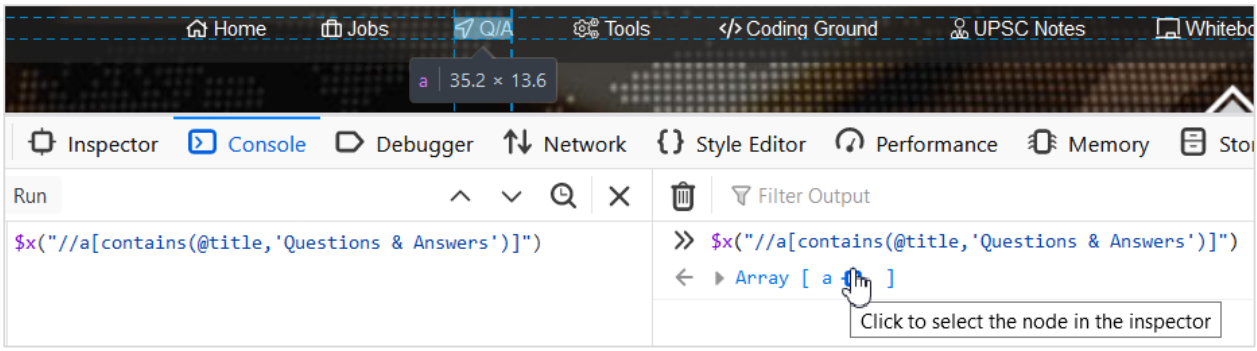


contains()

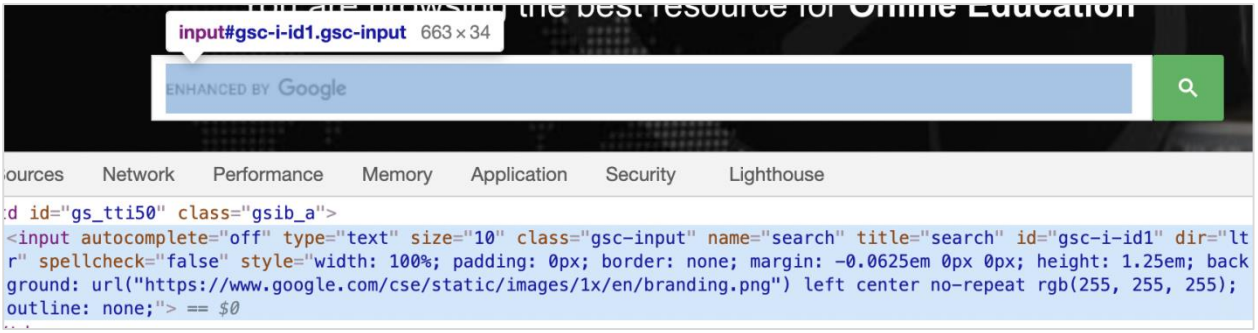
It identifies an element whose attribute value contains a sub-text. This function is normally used for attributes whose value changes on each page load.

The xpath expression is as follows:

```
//a[contains(@title, 'Questions & Answers')].
```



Let us see the html code of a webelement:



The edit box highlighted in the above image has a name attribute with value search, the xpath expression should be //input[@name='search']. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation for identifying multiple elements by Xpath is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify elements with xpath
l = driver.find_elements_by_xpath("//input[@name='search']")
#count elements
s = len(l)
#iterate through list
for i in l:
#obtain text
  t = i.send_keys('Tutorialspoint - Selenium')
  v = i.get_attribute('value')
print('Value entered is: ' + v)
#driver quit
```

```
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/
Value entered is: Tutorialspoint - Selenium

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Tutorialspoint - Selenium gets printed in the console.

6. Selenium Webdriver — Explicit and Implicit Wait

Let us understand what an explicit wait in the Selenium Webdriver is.

Explicit Wait

An explicit wait is applied to instruct the webdriver to wait for a specific condition before moving to the other steps in the automation script.

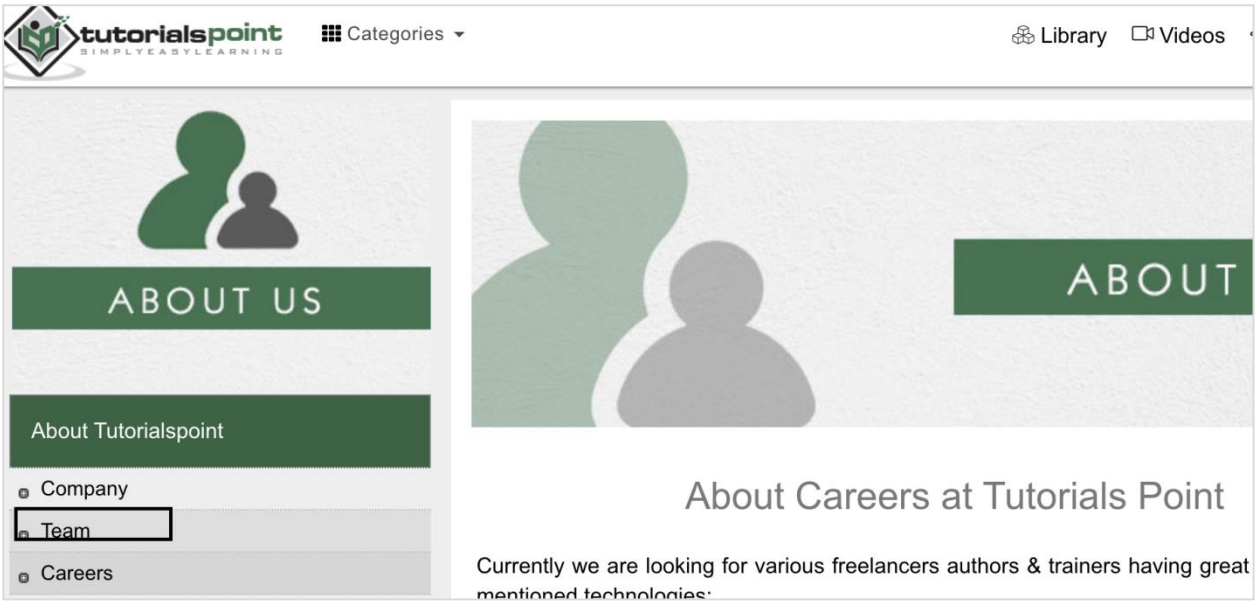
Explicit wait is implemented using the WebDriverWait class along with expected_conditions. The expected_conditions class has a group of pre-built conditions to be used along with the WebDriverWait class.

Pre-built Conditions

The pre-built conditions which are to be used along with the WebDriverWait class are given below:

- alert_is_present
- element_selection_state_to_be
- presence_of_all_elements_located
- element_located_to_be_selected
- text_to_be_present_in_element
- text_to_be_present_in_element_value
- frame_to_be_available_and_switch_to_it
- element_located_to_be_selected
- visibility_of_element_located
- presence_of_element_located
- title_is
- title_contains
- visibility_of
- staleness_of
- element_to_be_clickable
- invisibility_of_element_located
- element_to_be_selected

Let us wait for the text - Team @ Tutorials Point which becomes available on clicking the link - Team on the page.



On clicking the Team link, the text Team @ Tutorials Point appears.

Code Implementation

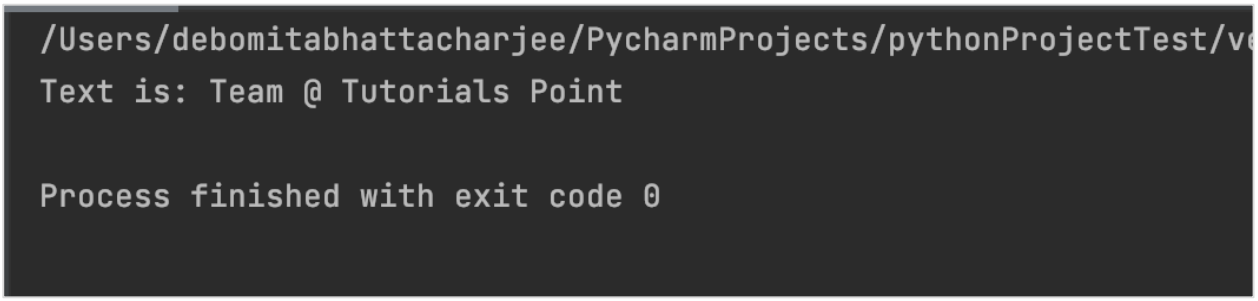
The code implementation for the explicit wait is as follows:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.support.wait import WebDriverWait
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify element
l = driver.find_element_by_link_text('Team')
l.click()
#expected condition for explicit wait
w = WebDriverWait(driver, 5)
w.until(EC.presence_of_element_located((By.TAG_NAME, 'h1')))
s = driver.find_element_by_tag_name('h1')
#obtain text
t = s.text
```

```
print('Text is: ' + t)
#driver quit
driver.quit()
```

Output

The output is mentioned below:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the text (obtained from the text method) - Team @ Tutorials Point gets printed in the console.

Implicit Wait

An implicit wait is applied to instruct the webdriver for polling the DOM (Document Object Model) for a specific amount of time while making an attempt to identify an element which is currently unavailable.

The default value of the implicit wait time is 0. Once a wait time is set, it remains applicable through the entire life of the webdriver object. If an implicit wait is not set and an element is still not present in DOM, an exception is thrown.

The **syntax** for the implicit wait is as follows:

```
driver.implicitly_wait(5)
```

Here, a wait time of five seconds is applied to the webdriver object.

Code Implementation

The code implementation for the implicit wait is as follows:

```
from selenium import webdriver
#set path of chromedriver.exe
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait of 0.5s
driver.implicitly_wait(0.5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify link with link text
```

```
l = driver.find_element_by_link_text('FAQ')
#perform click
l.click()
print('Page navigated after click: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is mentioned below:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/pyt
Page navigated after click: Frequently Asked Questions - Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. On clicking on the FAQ link, the webdriver waits for 0.5 seconds and then moves to the next step. Also, the title of the next page(obtained from the driver.title method) - Frequently Asked Questions - Tutorialspoint gets printed in the console.

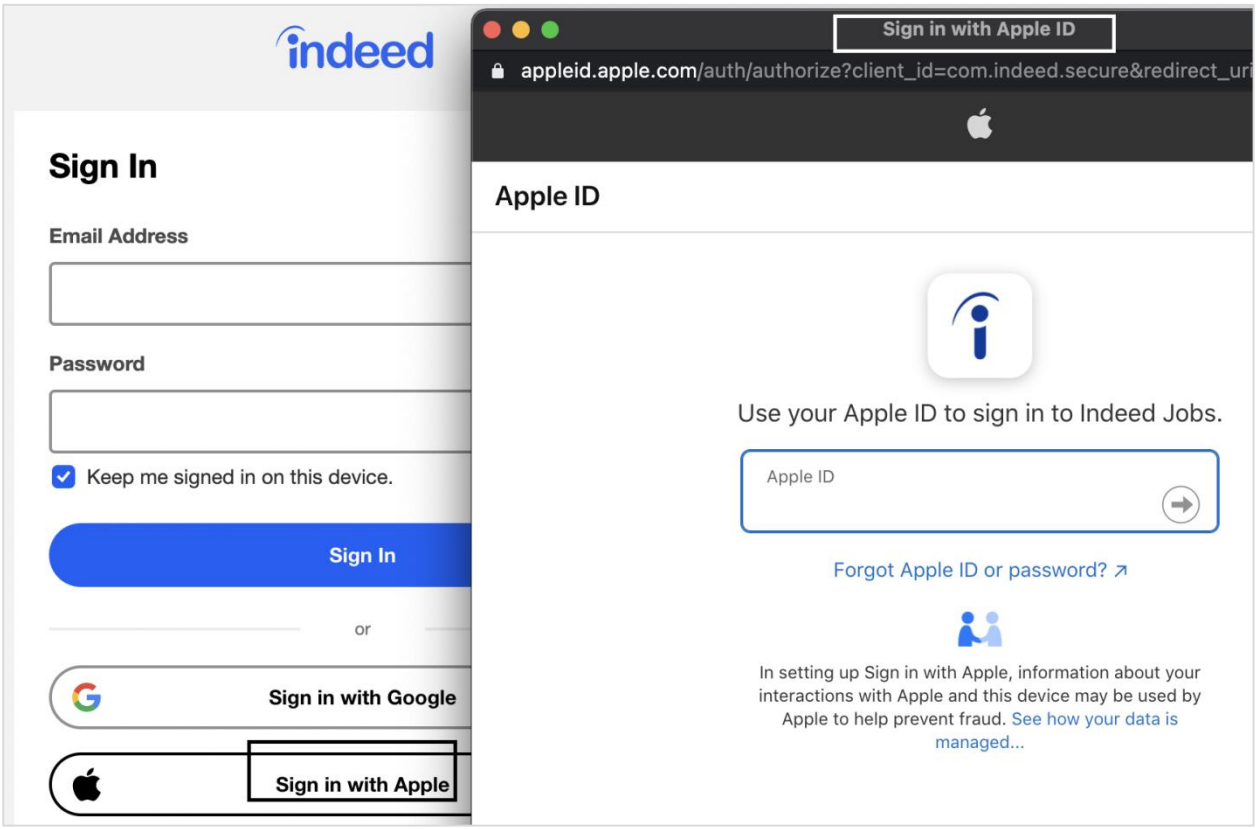
7. Selenium Webdriver — Pop-ups

A new pop-up window can open on clicking a link or a button. The webdriver by default has control over the main page, in order to access the elements on the new pop-up, the webdriver control has to be switched from the main page to the new pop-up window.

Methods

The methods to handle new pop-ups are listed below:

- **driver.current_window_handle:** To obtain the handle id of the window in focus.
- **driver.window_handles:**To obtain the list of all the opened window handle ids.
- **driver.swtich_to.window(<window handle id>):**To switch the webdriver control to an opened window whose handle id is passed as a parameter to the method.



On clicking the Sign in with Apple button, a new pop-up opens having the browser title as Sign in with Apple ID Let us try to switch to the new pop-up and access elements there.

Code Implementation

The code implementation for the pop-ups is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://the-internet.herokuapp.com/windows")
#identify element
s = driver.find_element_by_link_text("Click Here")
s.click()
#current main window handle
m= driver.current_window_handle
#iterate over all window handles
for h in driver.window_handles:
#check for main window handle
   if h != m:
       n = h
#switch to new tab
driver.switch_to.window(n)
print('Page title of new tab: ' + driver.title)
#switch to main window
driver.switch_to.window(m)
print('Page title of main window: ' + driver.title)
#quit browser
driver.quit()
```

Output

The output is as follows

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/
Page title of new pop-up: Sign in with Apple ID
Page title of main window: Sign In | Indeed Accounts

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. First the page title of the new pop-up(obtained from the method title) - Sign in with Apple ID gets printed in the console. Next, after switching the webdriver control to the main window, its page title - Sign In | Indeed Accounts get printed in the console.

8. Selenium Webdriver — Backward and Forward Navigation

We can move backward and forward in browser history with the help of the Selenium webdriver with Python. To navigate a step forward in history the method forward is used. To navigate a step backward in history the method back is used.

The **syntax** for backward and forward navigation is as follows:

```
driver.forward()
driver.back()
```

Code Implementation

The code implementation for backward and forward navigation is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(0.8)
#url 1 launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#url 2 launch
driver.get("https://www.tutorialspoint.com/online_dev_tools.htm")
#back in history
driver.back()
print('Page navigated after back: ' + driver.title)
#forward in history
driver.forward()
print('Page navigated after forward: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Page navigated after back: About Careers at Tutorials Point - Tutorialspoint
Page navigated after forward: Online Development and Testing Tools

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. After launching the two URLs, the webdriver navigates back in the browser history and the title of the previous page(obtained from the driver.title method) - About Careers at Tutorialspoint - Tutorialspoint gets printed in the console.

Again, the webdriver navigates forward in the browser history and the title of the following page(obtained from the driver.title method) - Online Development and Testing Tools gets printed in the console.

9. Selenium Webdriver — Cookies

Selenium webdriver can handle cookies. We can add a cookie, obtain a cookie with a particular name, and delete a cookie with the help of various methods in Selenium.

Methods

The methods to handle cookies are listed below:

- **add_cookie**: Used to add a cookie to the present session.
- **get_cookie**: Used to get a cookie with a particular name. It yields none, if there is no cookie available with the given name.
- **get_cookies**: Used to get all the cookies for the present URL.
- **delete_cookie**: Used to delete a cookie with a particular name.
- **delete_all_cookies**: Used to delete all the cookies for the present URL.

Code Implementation

The code implementation for handling cookies is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#add a cookie
c = {'name': 'c1', 'value': 'val1'}
driver.add_cookie(c)
#get a cookie
l = driver.get_cookie('c1')
print('Cookie is: ')
print(l)
#get all cookies
m = driver.get_cookies()
print('Cookies are: ')
print(m)
#delete a cookie
driver.delete_cookie('c1')
#check cookie after deletion
```

```
l = driver.get_cookie('c1')
print('Cookie is: ')
print(l)
#close driver
driver.close()
```

Output

The output is as follows:

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. First, the details of the cookie which is added to the current session get printed in the console.

Next, the details of all the cookies which are present to the current session get printed in the console. After the deletion of the cookie c1, we have tried to obtain the details of the cookie c1. Since it is deleted, None is returned by the get_cookie method.

10. Selenium Webdriver — Exceptions

If an error occurs, any of the methods fail or an unexpected error happens, an exception is thrown. In Python, all the exceptions are obtained from the BaseException class.

Selenium Exceptions

Some of the common Selenium Exceptions are listed below:

- **ElementNotInteractableException**: It is thrown if a webelement is attached to the DOM, but on trying to access the same webelement a different webelement gets accessed.
- **ElementClickInterceptedException**: It is thrown if a click operation on a webelement could not happen because another webelement covering that webelement receives the click.
- **ElementNotVisibleException**: It is thrown if a webelement is attached to the DOM, but invisible on the page and inaccessible.
- **ElementNotSelectableException**: It is thrown if we make an attempt to select a webelement which is not selectable.
- **ImeActivationFailedException**: It is thrown if we fail to activate an IME engine.
- **ErrorInResponseException**: It is thrown if there is an issue on the server side.
- **InsecureCertificateException**: It is thrown if a user gets a certificate warning while navigating an application. It is due to a TLS certificate which is no longer active and valid.
- **ImeNotAvailableException**: It is thrown if there is no support for the IME engine.
- **InvalidCookieDomainException**: It is thrown if we try to add a cookie under a varied domain than the present URL.
- **InvalidArgumentException**: It is thrown if the argument passed to a command is no longer valid.
- **InvalidElementStateException**: It is thrown if we try to access a webelement which is not in a valid state.
- **InvalidCoordinatesException**: It is thrown if the coordinates for interactions are not valid.
- **InvalidSessionIdException**: It is thrown if the session id is not available in the group of live sessions. Thus the given session is either non-existent or inactive.

- **InvalidSelectorException**: It is thrown if the locator used to identify an element does not yield a webelement.
- **MoveTargetOutOfBoundsException**: It is thrown if the target given in the ActionChains method is out of the scope of the document.
- **InvalidSwitchToTargetException**: It is thrown if the frame id/name or the window handle id to be switched to is incorrect.
- **NoSuchAttributeException**: It is thrown if an element attribute is not detected.
- **NoAlertPresentException**: It is thrown if we try to switch to an alert which is non-existent.
- **NoSuchFrameException**: It is thrown if we try to switch to a frame which is non-existent.
- **StaleElementReferenceException**: It is thrown if an element reference is currently stale.
- **NoSuchWindowException**: It is thrown if we try to switch to a window which is non-existent.
- **UnexpectedAlertPresentException**: It is thrown if an alert appears unexpectedly in an automation flow.
- **UnableToSetCookieException**: It is thrown if the webdriver is unsuccessful in setting a cookie.
- **UnexpectedTagNameException**: It is thrown if a support class has not received an anticipated webelement.
- **NoSuchElementException**: It is thrown if the selector used is unable to locate a webelement.

Let us see an example of a code which throws an exception.

Code Implementation

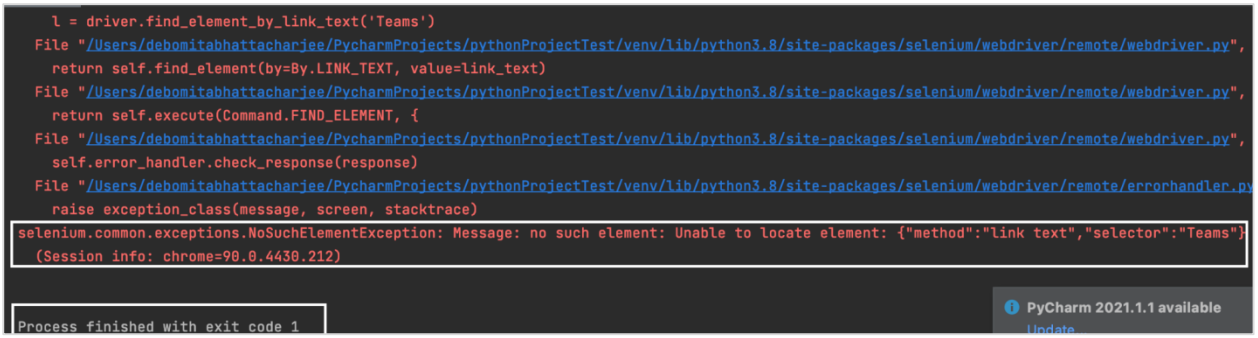
The code implementation for the Selenium Exceptions is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify element with an incorrect link text value
l = driver.find_element_by_link_text('Teams')
l.click()
#driver quit
```

```
driver.quit()
```

Output

The output is given below:



The output shows the message - Process with exit code 1 meaning that the above Python code has encountered an error. Also, NoSuchElementException is thrown since the locator link text is not able to detect the link Teams on the page.

11. Selenium Webdriver — Action Class

Selenium can perform mouse movements, key press, hovering on an element, drag and drop actions, and so on with the help of the ActionsChains class. We have to create an instance of the ActionChains class which shall hold all actions in a queue.

Then the method - perform is invoked which actually performs the tasks in the order in which they are queued. We have to add the statement from selenium.webdriver import ActionChains to work with the ActionChains class.

The syntax for ActionChains class is as follows:

#Method 1 - chained pattern

```
e =driver.find_element_by_css_selector(".txt")
a = ActionChains(driver)
a.move_to_element(e).click().perform()
```

#Method 2 - queued actions one after another

```
e =driver.find_element_by_css_selector(".txt")
a = ActionChains(driver)
a.move_to_element(e)
a.click()
a.perform()
```

In both the above methods, the actions are performed in sequence in which they are called, one by one.

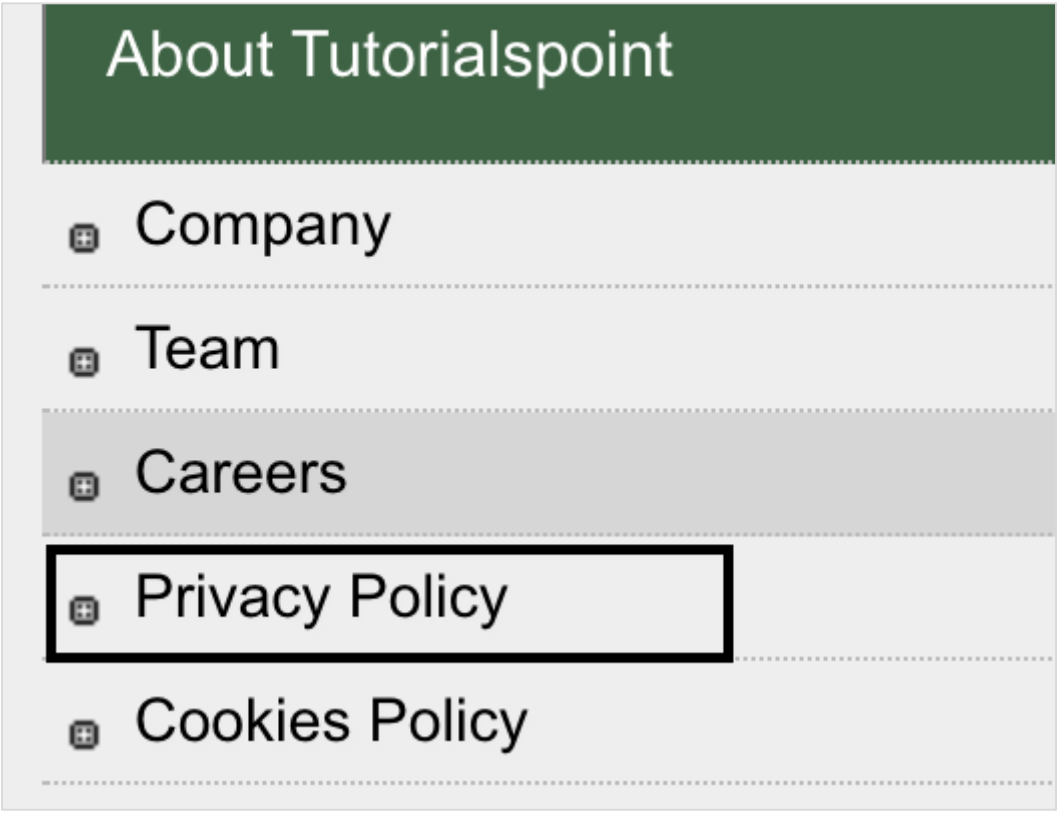
Methods

The methods of ActionChains class are listed below:

- **click**: It is used to click a webelement.
- **click_and_hold**: It is used to hold down the left mouse button on a webelement.
- **double_click**: It is used to double click a webelement.
- **context_click**: It is used to right click a webelement.
- **drag_and_drop_by_offset**: It is used to first perform pressing the left mouse on the source element, navigating to the target offset and finally releasing the mouse.
- **drag_and_drop**: It is used to first perform pressing the left mouse on the source element, navigating to the target element and finally releasing the mouse.
- **key_up**: It is used to release a modifier key.
- **key_down**: It is used for a keypress without releasing it.

- **move_to_element**: It is used to move the mouse to the middle of a webelement.
- **move_by_offset**: It is used to move the mouse to an offset from the present mouse position.
- **Perform**: It is used to execute the queued actions.
- **move_to_element_by_offset**: It is used to move the mouse by an offset of a particular webelement. The offsets are measured from the left-upper corner of the webelement.
- **Release**: It is used to release a held mouse button on a webelement.
- **Pause**: It is used to stop every input for a particular duration in seconds.
- **send_keys**: It is used to send keys to the present active element.
- **reset_actions**: It is used to delete all actions that are held locally and in remote.

Let us click on the link - Privacy Policy using the ActionChains methods:



Code Implementation

The code implementation for ActionChains class is as follows:

```
from selenium import webdriver
from selenium.webdriver import ActionChains
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
```

```
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify element
s = driver.find_element_by_link_text("Privacy Policy")
#instance of ActionChains
a= ActionChains(driver)
#move to element
a.move_to_element(s)
#click
a.click().perform()
#get page title
print('Page title: ' + driver.title)
#driver quit
driver.close()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bi
Page title: About Privacy Policy at Tutorials Point - Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application(obtained from the driver.title method) - About Privacy Policy at Tutorials Point - Tutorialspoint gets printed in the console.

12. Selenium Webdriver — Create a Basic Test

To create a basic test in Selenium with Python, the below steps need to be executed:

Step 1: Identify the browser in which the test has to be executed. As we type webdriver in the editor, all the available browsers like Chrome, Firefox get displayed. Also, we have to pass the path of the chromedriver executable file path.

The syntax to identify the browser is as follows:

```
driver = webdriver.Chrome(executable_path='<path of chromedriver>')
```

Step 2: Launch the application URL with the get method.

The **syntax** for launching the application URL is as follows:

```
    driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
```

Step 3: Identify webelement with the help of any of the locators like id, class, name, tagname, link text, partial link text, css or xpath on the page.

The **syntax** to identify the webelement is as follows:

```
l = driver.find_element_by_partial_link_text('Refund')
```

Step 4: After the element has been located, perform an action on it like inputting a text, clicking, and so on.

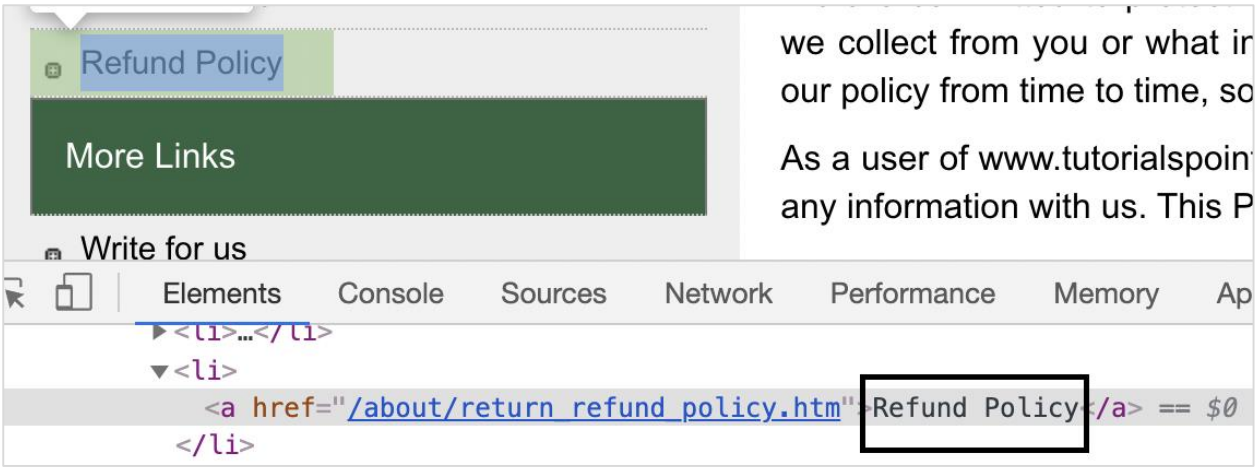
The **syntax** for performing an action is as follows:

```
driver.find_element_by_partial_link_text('Refund').click()
```

Step 5: Finish the test by quitting the webdriver session. For example,

```
   driver.quit();
```

Let us see the html code of a webelement:



The link highlighted in the above image has a tagname - a and the partial link text - Refund. Let us try to click on this link after identifying it.

Code Implementation

The code implementation to create a basic test is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify link with partial link text
l = driver.find_element_by_partial_link_text('Refund')
#perform click
l.click()
print('Page navigated after click: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python /Use
Page navigated after click: Return, Refund, & Cancellation Policy - Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application (obtained from the driver.title method) - Return, Refund & Cancellation Policy - Tutorialspoint gets printed in the cons

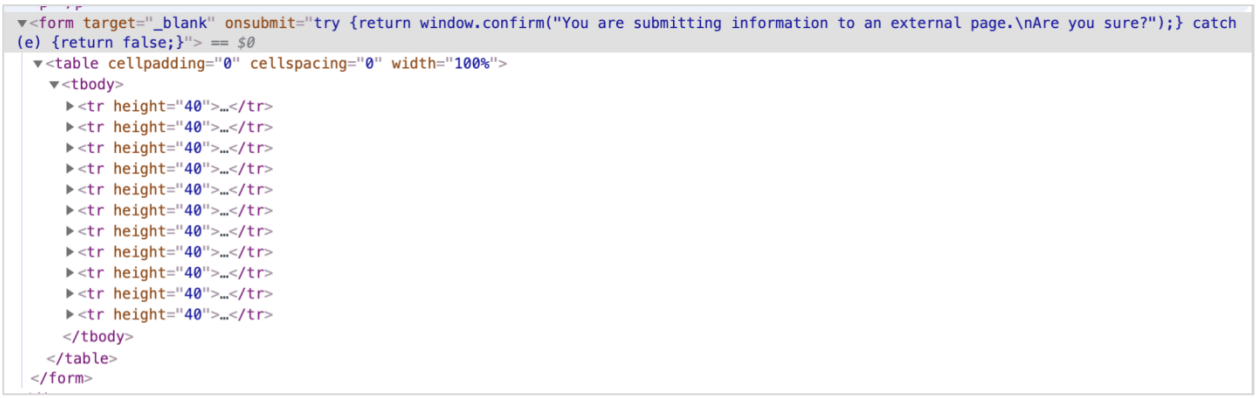
13. Selenium Webdriver — Forms

Selenium webdriver can be used to submit a form. A form in a page is represented by the <form> tag. It contains sub-elements like the edit box, dropdown, link, and so on. Also, the form can be submitted with the help of the submit method.

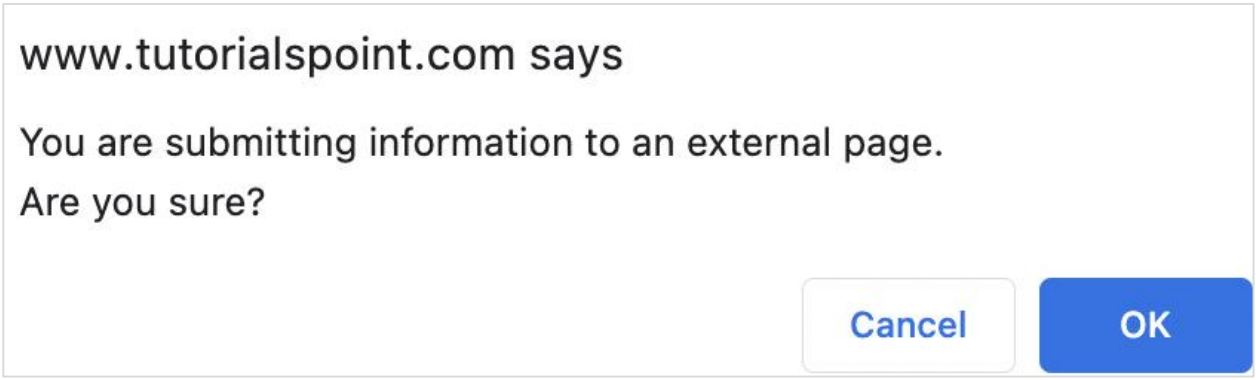
The **syntax** for forms is as follows:

```
src = driver.find_element_by_css_selector("#draggable")
src.submit()
```

Let us see the html code of elements within the form tag.



On submitting a form with the above html code, the below alert message is displayed.



Code Implementation

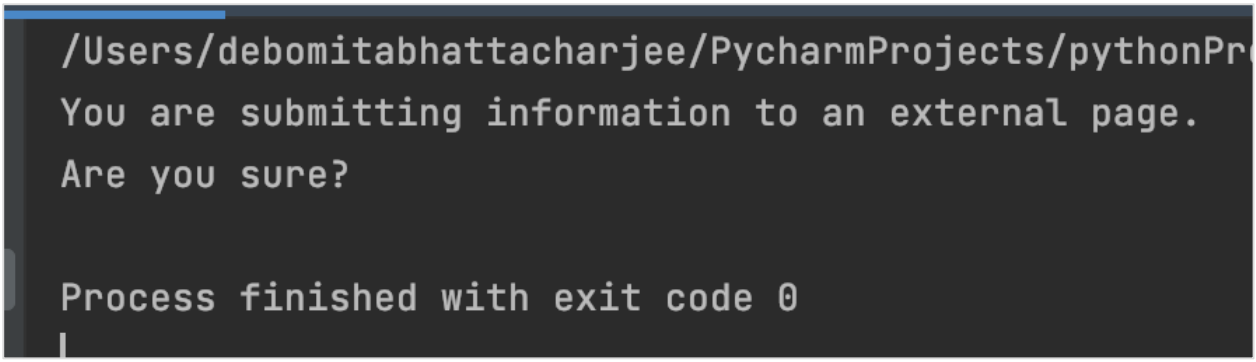
The code implementation for submitting a form is as follows:

```
from selenium import webdriver
from selenium.webdriver.common.alert import Alert
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
```

```
#url launch
driver.get("https://www.tutorialspoint.com/selenium/selenium_automation_practice.htm")
#identify element within form
b = driver.find_element_by_name("firstname")
b.send_keys('Tutorialspoint')
e = driver.find_element_by_name("lastname")
e.send_keys('Online Studies')
#submit form
e.submit()
# instance of Alert class
a = Alert(driver)
# get alert text
print(a.text)
#accept alert
a.accept()
#driver quit
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the Alert text - You are submitting information to an external page.

Are you sure?

The above message gets printed in the console.

14. Selenium Webdriver — Drag and Drop

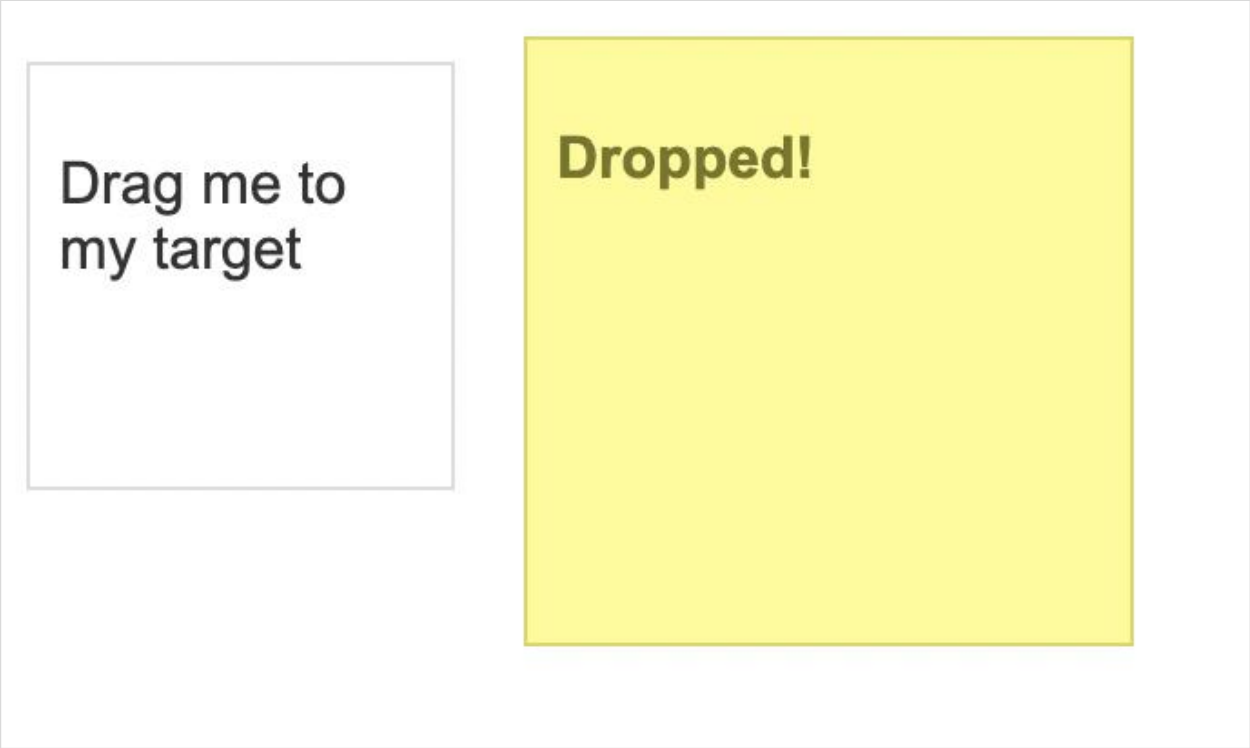
Selenium can perform mouse movements, key press, hovering on an element, drag and drop actions, and so on with the help of the ActionsChains class. The method drag_and_drop first performs pressing the left mouse on the source element, navigating to the target element and finally releasing the mouse.

The **syntax** for drag and drop is as follows:

```
drag_and_drop(s, t)
```

Here, s is the source element on which the left mouse button is pressed and t is the target element. We have to add the statement from selenium.webdriver import ActionChains to work with the ActionChains class.

Let us perform the drag and drop functionality for the below elements:



In the above image, the element with the name - Drag me to my target has to be dragged and dropped to the element - Dropped!.

Code Implementation

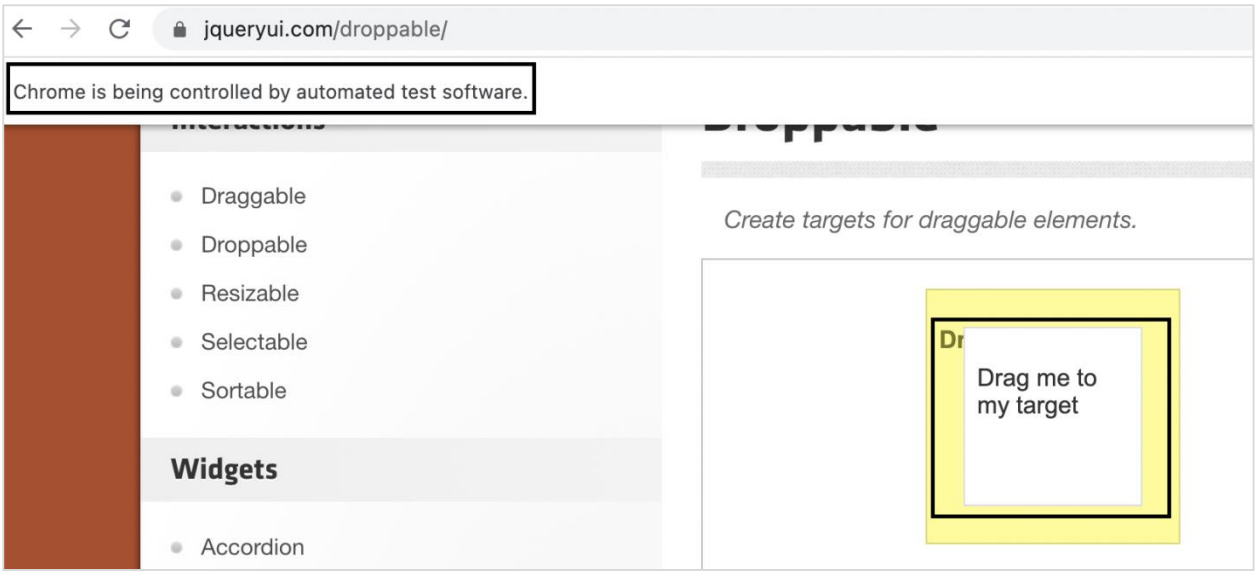
The code implementation for drag and drop is as follows:

```
from selenium import webdriver


from selenium.webdriver import ActionChains
```

```
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')

#implicit wait time

driver.implicitly_wait(5)

driver.maximize_window()

#url launch

driver.get("https://jqueryui.com/droppable/")

#switch to frame

driver.switch_to.frame(0)

#identify source element

src = driver.find_element_by_css_selector("#draggable")

#identify target element

trgt = driver.find_element_by_css_selector("#droppable")

#instance of ActionChains

a= ActionChains(driver)

#drag and drop then perform

a.drag_and_drop(src, trgt)

a.perform()
```

Output

The output is as follows:



After execution, the element with the name - Drag me to my target has been dragged and dropped to the element - Dropped!.

The frames in an html code are represented by the frames/iframe tag. Selenium can handle frames by switching the webdriver access from the main page to the frame.

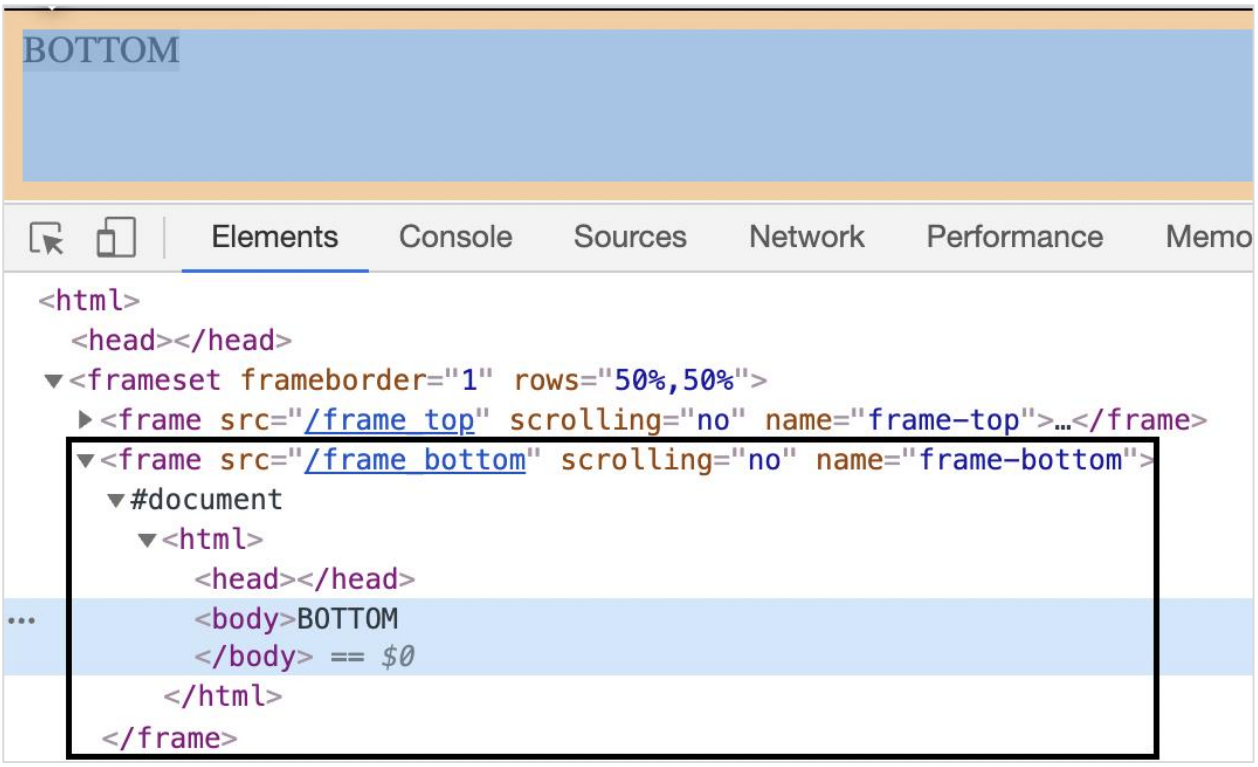
Selenium Webdriver Frames

Methods

The methods to handle frames are listed below:

- **driver.switch_to_frame("framename"):** framename is the name of the frame.
- **driver.switch_to_frame("framename.0.frame1"):** Used to access the sub-frame in a frame by separating the path with dot. Here, it would point to the frame with name frame1 which is the first sub-frame of the frame named framename.
- **driver.switch_to_default_content():**Used to switch the webdriver access from a frame to the main page.

Let us see the html code of an element inside a frame.



The tagname highlighted in the above image is frame and the value of the name attribute is frame_bottom.

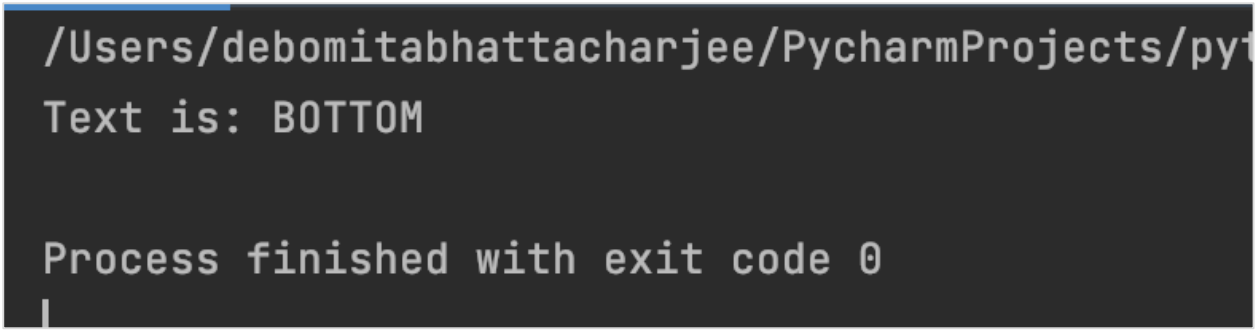
Code Implementation

The code implementation to handle frames is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://the-internet.herokuapp.com/nested_frames")
#switch to frame
driver.switch_to.frame('frame-bottom')
#identify source element
s = driver.find_element_by_tag_name("body")
#obtain text
t = s.text
print('Text is: ' + t)
#quit browser
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the text within the frame (obtained from the text method) - BOTTOM gets printed in the console.

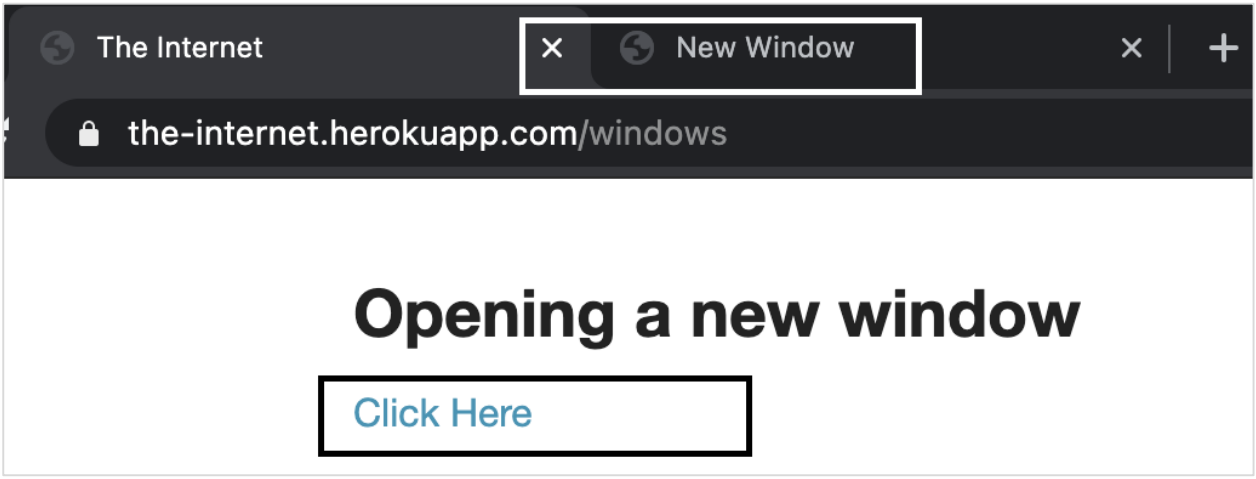
15. Selenium Webdriver — Windows

A new pop-up window or a tab can open on clicking a link or a button. The webdriver by default has control over the main page, in order to access the elements on the new window, the webdriver control has to be switched from the main page to the new pop-up window or tab.

Methods

The methods to handle new windows are listed below:

- **driver.current_window_handle**: To obtain the handle id of the window in focus.
- **driver.window_handles**: To obtain the list of all the opened window handle ids.
- **driver.swtich_to.window(<window handle id>)**: To switch the webdriver control to an opened window whose handle id is passed as a parameter to the method.



On clicking the Click Here link, a new tab opens having the browser title as New Window. Let us try to switch to the new tab and access elements in there.

Code Implementation

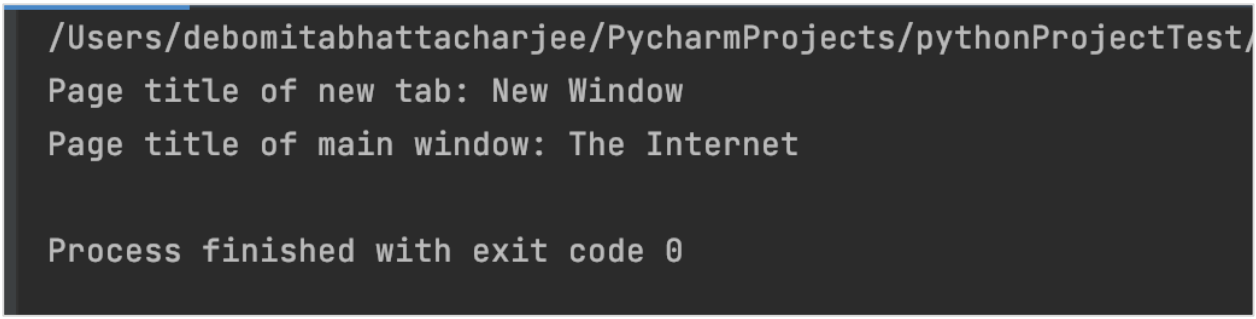
The code implementation for opening a new window is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://the-internet.herokuapp.com/windows")
```

```
#identify element
s = driver.find_element_by_link_text("Click Here")
s.click()
#current main window handle
m= driver.current_window_handle
#iterate over all window handles
for h in driver.window_handles:
#check for main window handle
   if h != m:
       n = h
#switch to new tab
driver.switch_to.window(n)
print('Page title of new tab: ' + driver.title)
#switch to main window
driver.switch_to.window(m)
print('Page title of main window: ' + driver.title)
#quit browser
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. First the page title of the new tab(obtained from the method title) - New Window gets printed in the console. Next, after switching the webdriver control to the main window, its page title - The Internet gets printed in the console.

16. Selenium Webdriver — Alerts

Selenium webdriver is capable of handling Alerts. The class selenium.webdriver.common.alert.Alert(driver) is used to work with Alerts. It has methods to accept, dismiss, enter and obtain the Alert text.

Methods

The methods under the Alert class are listed below:

- **accept()**: For accepting an Alert.
- **dismiss()**: For dismissing an Alert.
- **text()**:For obtaining Alert text.
- **send_keys(keysToSend)**:For entering text in Alert.

Code Implementation

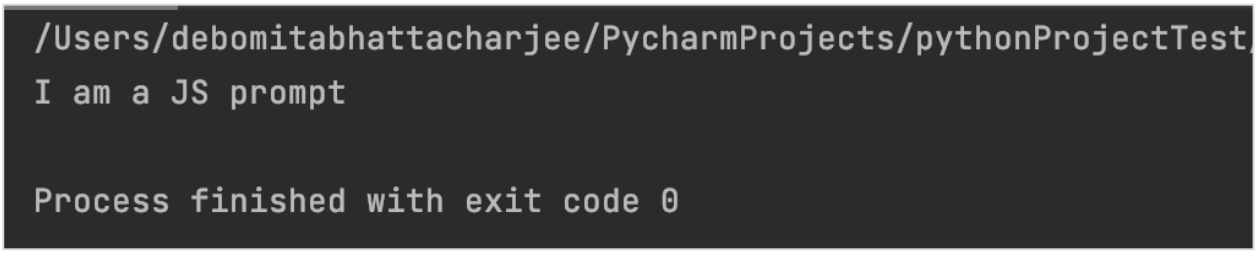
The code implementation for alerts is as follows:

```
from selenium import webdriver
#import Alert class
from selenium.webdriver.common.alert import Alert
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(0.8)
#url launch
driver.get("https://the-internet.herokuapp.com/javascript_alerts")
#identify element
l = driver.find_element_by_xpath("//*[text()='Click for JS Prompt']")
l.click()
# instance of Alert class
a = Alert(driver)
# get alert text
print(a.text)
#input text in Alert
a.send_keys('Tutorialspoint')
#dismiss alert
a.dismiss()
l.click()
#accept alert
```

```
a.accept()
#driver quit
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the Alert text - I am a JS prompt gets printed in the console.

17. Selenium Webdriver — Handling Links

Selenium can be used to handle links on a page. A link is represented by the anchor tag. A link can be identified with the help of the locators like - link text and partial link text.

We can use the link text attribute for an element for its identification and utilize the method find_element_by_link_text. With this, the first element with the matching value of the given link text is returned.

The **syntax** for handling links is as follows:

```
driver.find_element_by_link_text("value of link text")
```

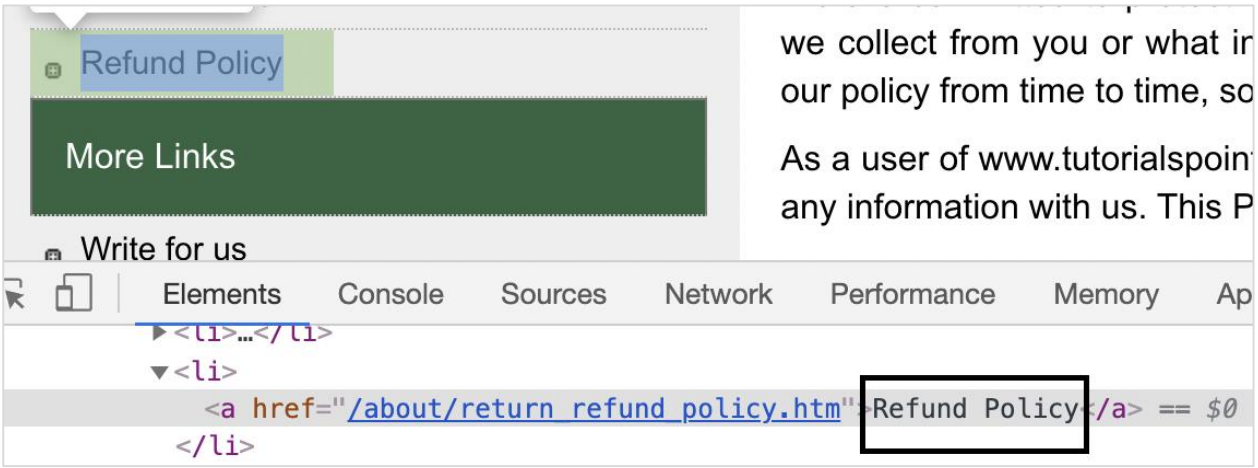
We can also use the partial link text attribute for an element for its identification and utilize the method find_element_by_partial_link_text. With this, the first element with the matching value of the given partial link text is returned.

For both the locators, if there is no element with the matching value of the partial link text/link text, NoSuchElementException shall be thrown.

The **syntax** for using the partial link text is as follows:

```
driver.find_element_by_partial_link_text("value of partial ink text")
```

Let us see the html code of a webelement, which is as follows:



The link highlighted in the above image has a tagname - a and the partial link text - Refund. Let us try to click on this link after identifying it.

Code Implementation

The code implementation for handling links is as follows:

```
from selenium import webdriver

driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
```

```
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify link with partial link text
l = driver.find_element_by_partial_link_text('Refund')
#perform click
l.click()
print('Page navigated after click: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python /Use
Page navigated after click: Return, Refund, & Cancellation Policy - Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application(obtained from the driver.title method) - Return, Refund & Cancellation Policy - Tutorialspoint gets printed in the console.

Let us now see the html code of another webelement:

The link highlighted in the above image has a tagname - a and the link text - Privacy Policy. Let us try to click on this link after identifying it.

Code Implementation

The code implementation for handling link is as follows:

```
from selenium import webdriver
```

```
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify link with link text
l = driver.find_element_by_link_text('Privacy Policy')
#perform click
l.click()
print('Page navigated after click: ' + driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows:

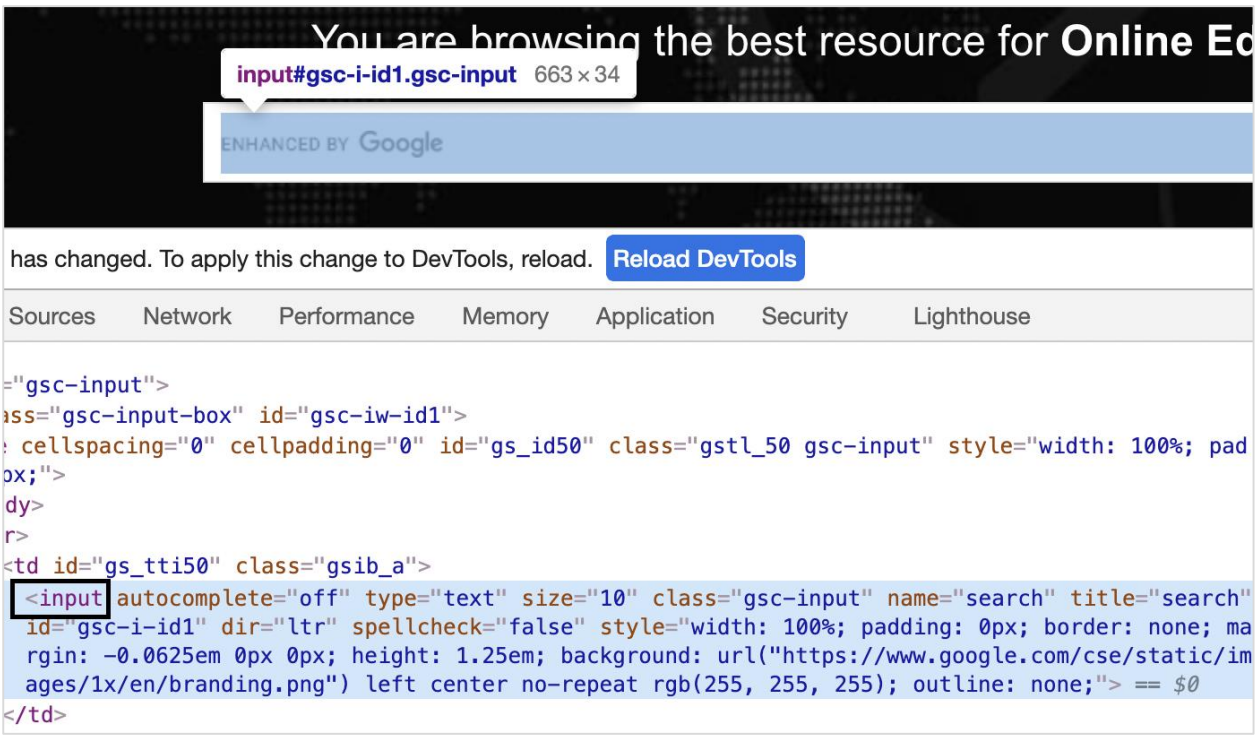
The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application(obtained from the driver.title method) - About Privacy Policy at Tutorials Point - Tutorialspoint gets printed in the console.

18. Selenium Webdriver — Handling Edit Boxes

Selenium can be used to input text to an edit box. An edit box is represented by the input tag and its type attribute should have the value as text. It can be identified with any of the locators like - id, class, name, css, xpath and tagname.

To input a value into an edit box, we have to use the method send_keys.

Let us see the html code of a webelement:



The edit box highlighted in the above image has a tagname - input. Let us try to input some text into this edit box after identifying it.

Code Implementation

The code implementation for handling edit box is as follows:

```
from selenium import webdriver
#set chromedriver.exe path
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#identify edit box with tagname
l = driver.find_element_by_tag_name('input')
#input text
l.send_keys('Selenium Python')
```

```
#obtain value entered
v = l.get_attribute('value')
print('Value entered: ' + v)
#driver close
driver.close()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python
Value entered: Selenium Python

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value entered within the edit box (obtained from the get_attribute method) - Selenium Python gets printed in the console.

19. Selenium Webdriver — Color Support

Selenium has the color conversion support class. We have to add the statement from selenium.webdriver.support.color import Color to convert colors to rgba/hex format.

Code Implementation

The code implementation for color conversion support is as follows:

```
from selenium import webdriver
from selenium.webdriver.support.color import Color
#color conversion to rgba format
print(Color.from_string('#00fe37').rgba)
#color conversion to hex format
print(Color.from_string('rgb(1, 200, 5)').hex)
#color conversion to rgba format
print(Color.from_string('green').rgba)
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pyth
rgba(0, 254, 55, 1)
#01c805
rgba(0, 128, 0, 1)

Process finished with exit code 0
```

20. Selenium Webdriver — Generating HTML Test Reports in Python

We can generate HTML reports with our Selenium test using the Pytest Testing Framework. To configure Pytest, we have to run the following command:

```
pip install pytest.
```

Once the installation is done, we can run the command to check the Pytest version installed:

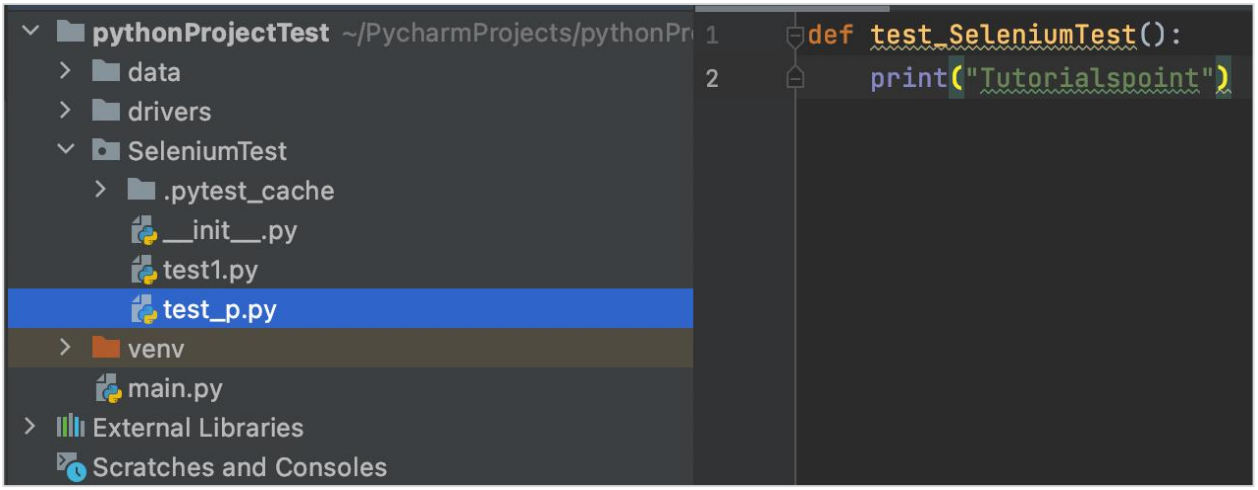
```
pytest –version
```

As a Pytest standard, the Python file containing the Pytest should start with test_ or end with _test. Also, all the test steps should be within a method whose name should start with test_.

To run a Pytest file, we can open the terminal and move from the current directory to the directory of the Pytest file that we want to execute. Then, run the command mentioned below:

```
py.test -v -s.
```

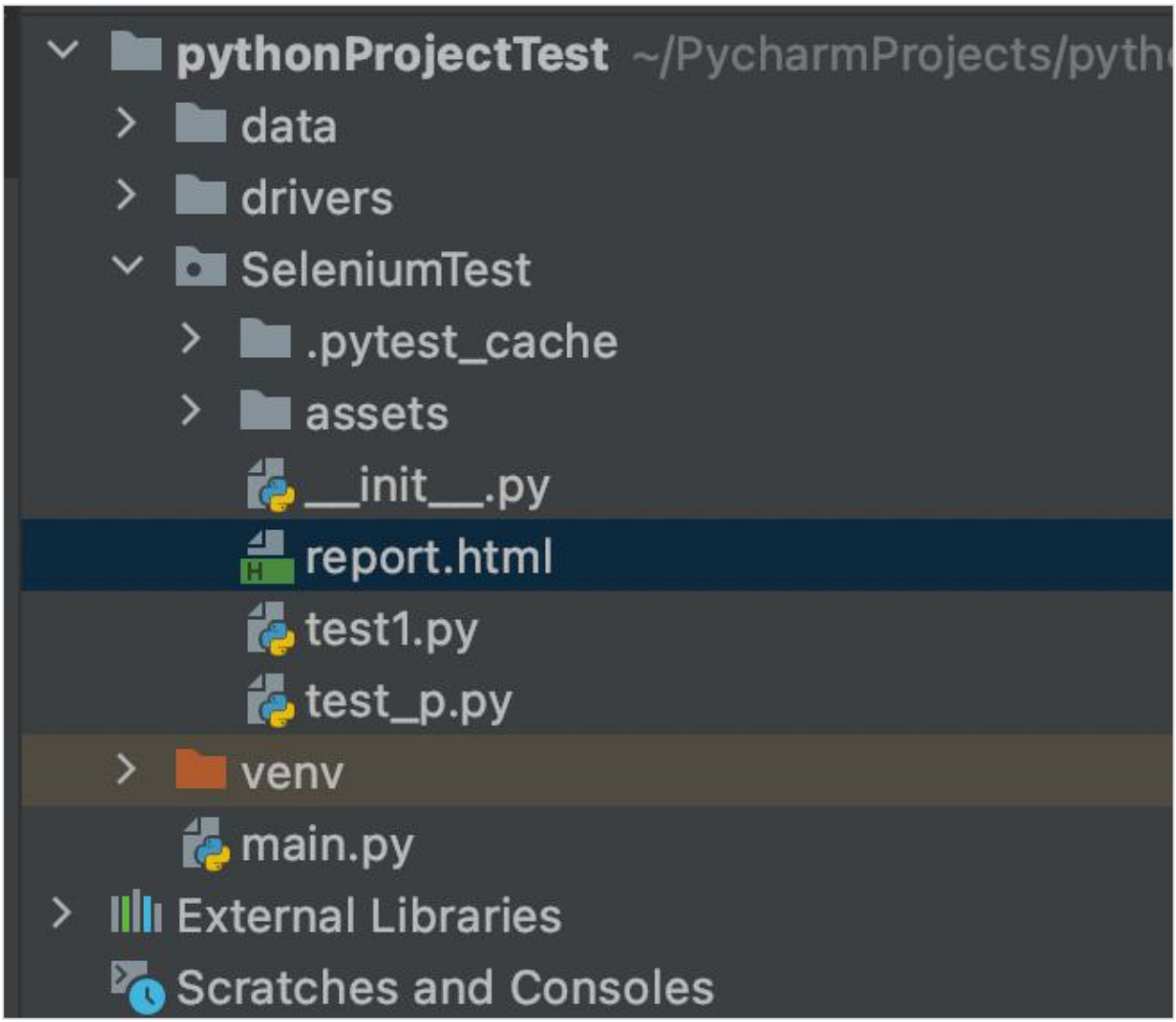
Let us look at a project structure following the Pytest Test Framework.



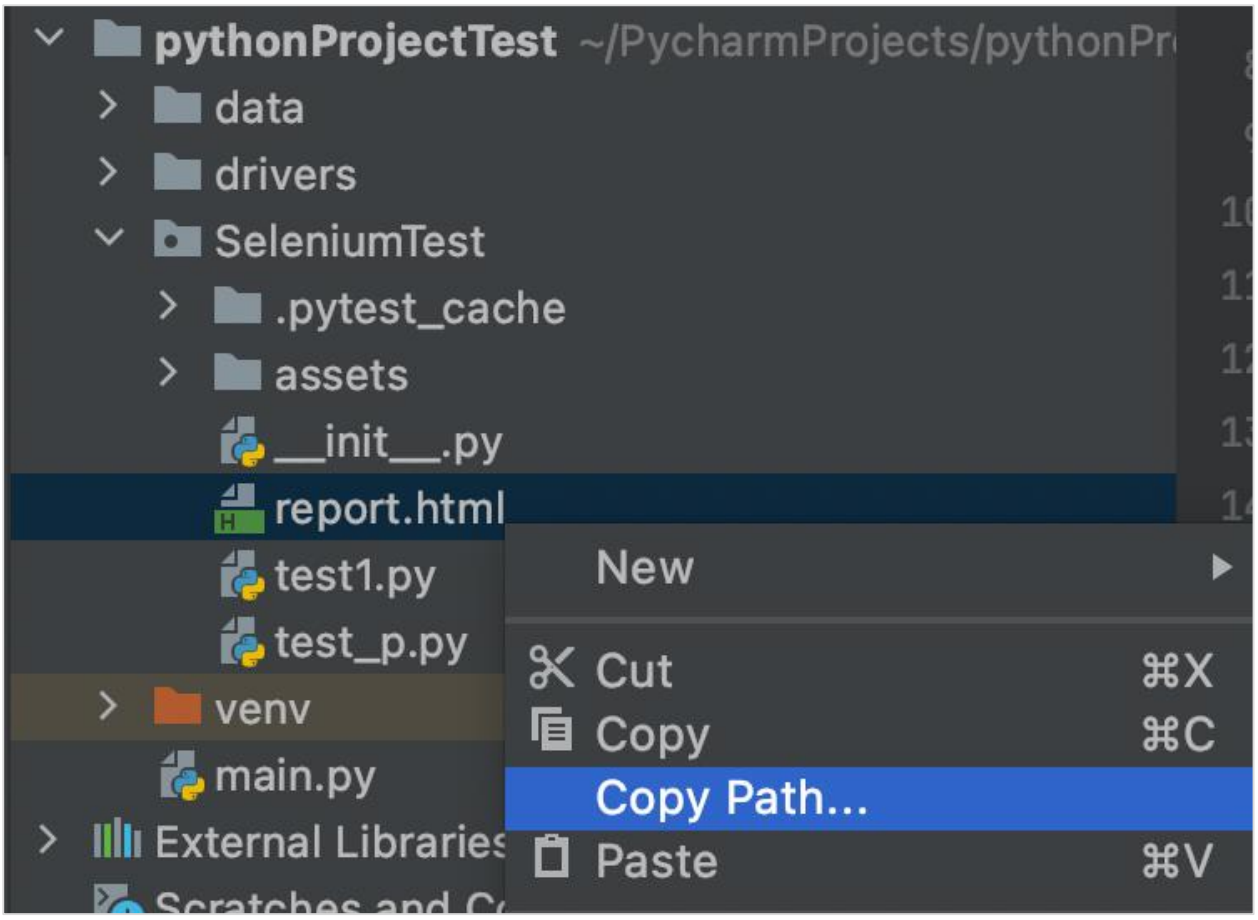
In the above image, it shows that the Pytest file has the name test_p.py and it contains a test method with the name test_SeleniumTest.

To generate a HTML report for a Selenium test, we have to install a plugin with the command: pip install pytest-html. To generate the report, we have to move from the current directory to the directory of the Pytest file that we want to execute. Then run the command: pytest --html=report.html.

After this command is successfully executed, a new file called the report.html gets generated within the project.



Right-click on the report.html and select the option Copy Path.



Open the path of the file copied in a browser, to get the HTML report.

report.html

Report generated on 31-May-2021 at 23:08:55 by pytest-html v3.1.1

Environment

Packages	{"pluggy": "0.13.1", "py": "1.9.0", "pytest": "6.2.4"}
Platform	macOS-10.16-x86_64-i386-64bit
Plugins	{"html": "3.1.1", "metadata": "1.11.0"}
Python	3.8.5

Summary

1 tests ran in 0.01 seconds.

(Un)check the boxes to filter the results.

1 passed, 0 skipped, 0 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration	Links
Passed (show details)	test_p.py::test_SeleniumTest	0.00	

The HTML report gives information of the Environment on which the test is executed. It also contains the information on test Summary and Results.

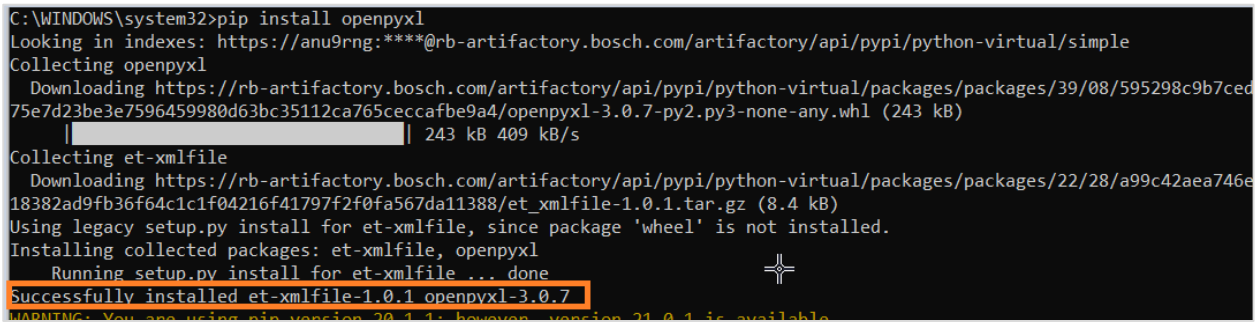
21. Selenium Webdriver — Read/Write data from Excel

We can read and write data from an excel sheet in Selenium webdriver in Python. An excel workbook consists of multiple sheets and each sheet consists of cells and columns.

To work with Excel in Python (with extensions .xlsx, .xlsm, and so on) we have to utilise the OpenPyXL library. To install this package, we have to run the following command:

```
pip install openpyxl.
```

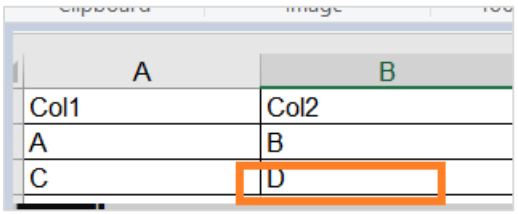
Also, we have to add the statement import openpyxl in our code.



To open an excel workbook, the method is load_workbook and pass the path of the excel file as a parameter to this method. To identify the active sheet, we have to use the active method on the workbook object.

To read a cell, the method cell is applied on the active sheet and the row and column numbers are passed as parameters to this method. Then, the value method is applied on a particular cell to read values within it.

Let us read the value at the third row and second column having the value D as shown below in an excel workbook of name Data.xlsx:



Code Implementation

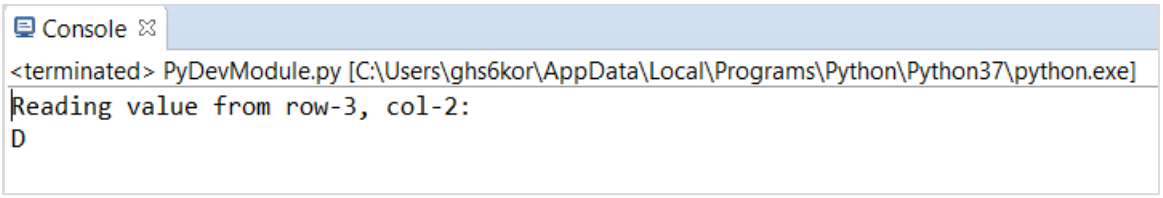
The code implementation read/write data from Excel to Selenium Webdriver in Python is as follows:

```
import openpyxl

#configure workbook path

b = openpyxl.load_workbook("C:\\Data.xlsx")

#get active sheet

sht = b.active
```

```
#get cell address within active sheet
cl = sht.cell (row = 3, column = 2)
#read value with cell
print("Reading value from row-3, col-2: ")
print (cl.value)
```

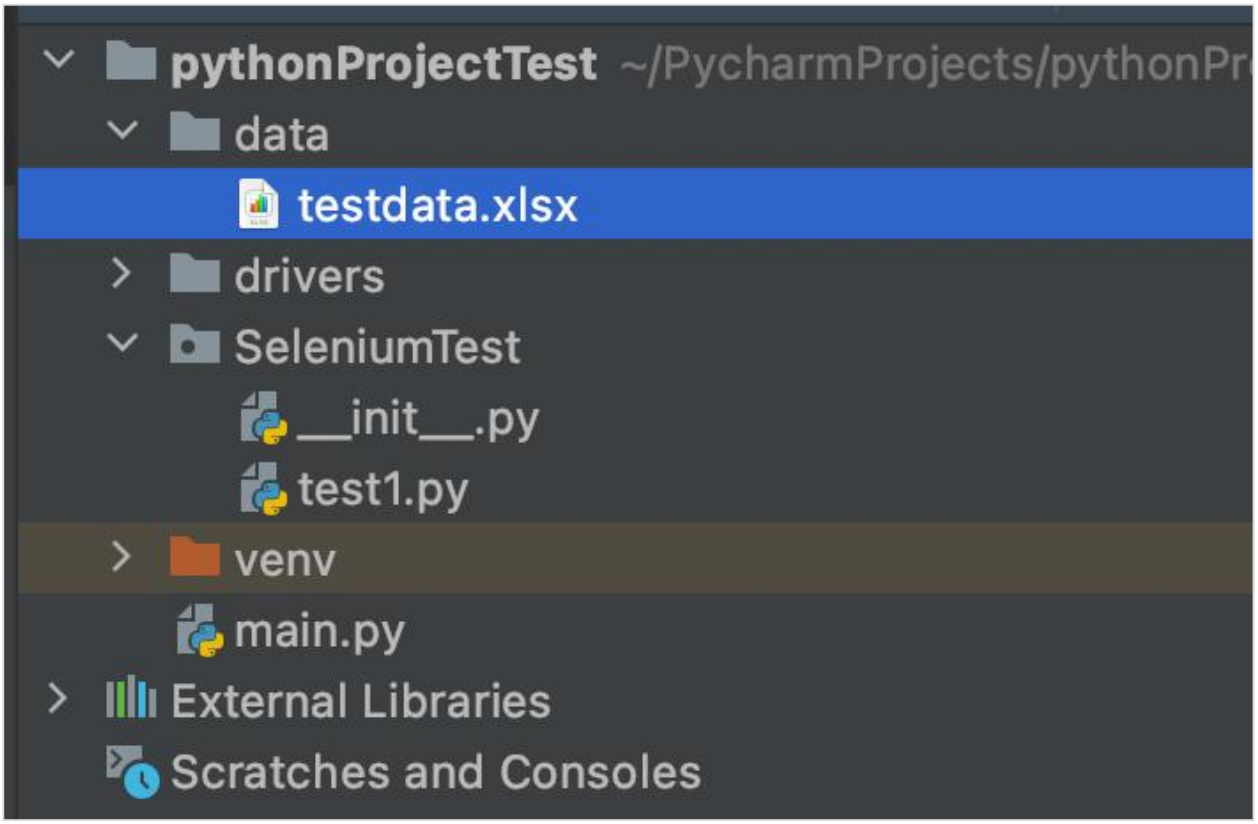
Output

The output is as follows:



To write a cell, the method cell is applied on the active sheet and the row and column numbers are passed as parameters to this method. Then, the value method is applied on a particular cell to write on it. Finally, the workbook is to be saved with the method save, the path of the file to be saved is passed as a parameter to this method.

We shall take an Excel name testdata.xlsx and save it within the data folder within our project. We shall write the value - Selenium Python in the third row and seventh column.



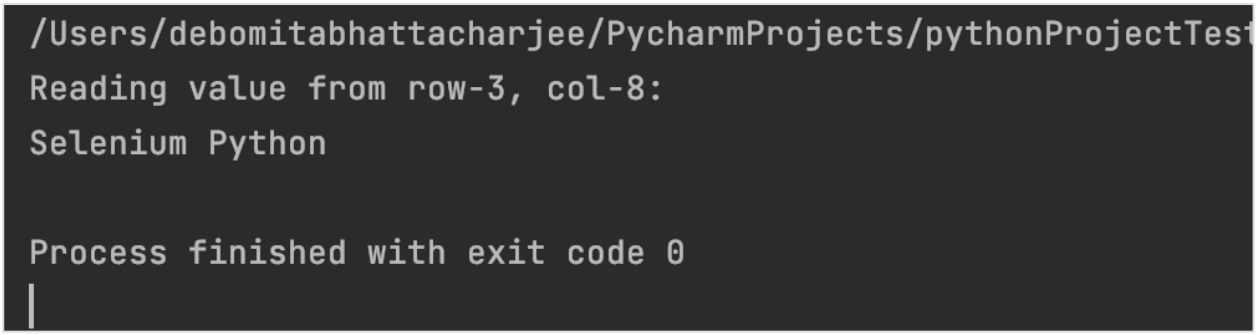
Code Implementation

The code implementation for working on workbook in Selenium Webdriver is as follows:

```
from selenium import webdriver
import openpyxl
#load workbook
b= openpyxl.load_workbook('../data/testdata.xlsx')
#get active worksheet
sh = b.active
# write value in third row, 8th column
sh.cell(row=3, column=8).value = "Selenium Python"
#save workbook
b.save("../data/testdata.xlsx")
#identify cell
cl = sh.cell(row=3, column=8)
#read cell value
print("Reading value from row-3, col-8: ")
print(cl.value)
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the value - Selenium Python is successfully written on the cell with address - row-3 and column - 8.

22. Selenium Webdriver — Handling Checkboxes

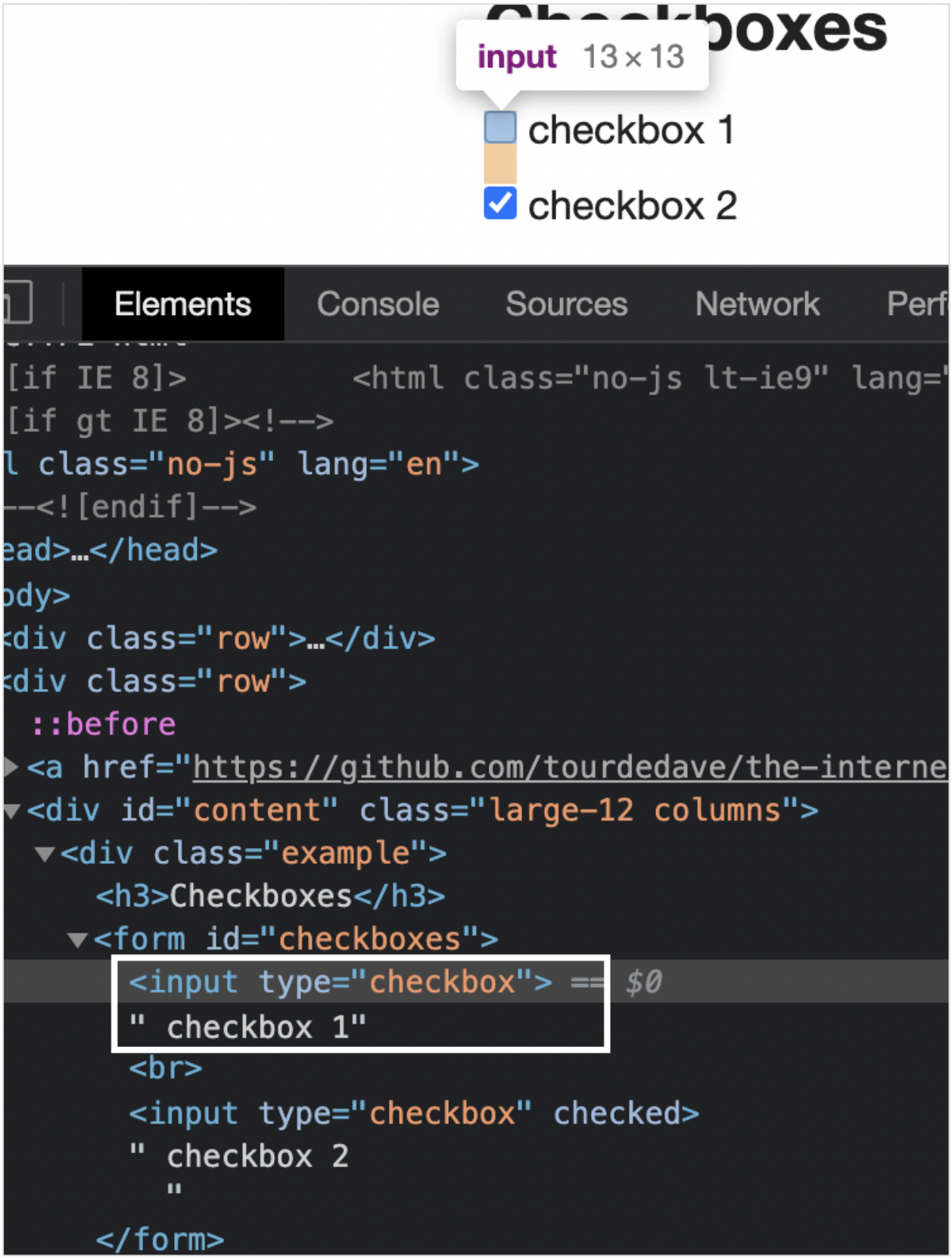
We can handle checkboxes with Selenium webdriver. A checkbox is represented by input tagname in the html code and its type attribute should have the value as checkbox.

Methods

The methods to handle the checkboxes are listed below:

- Click: Used to check a checkbox.
- is_selected:Used to check if a checkbox is checked or not. It returns a boolean value, true is returned in case a checkbox is checked.

Let us see the html code of a checkbox, which is as follows:



Code Implementation

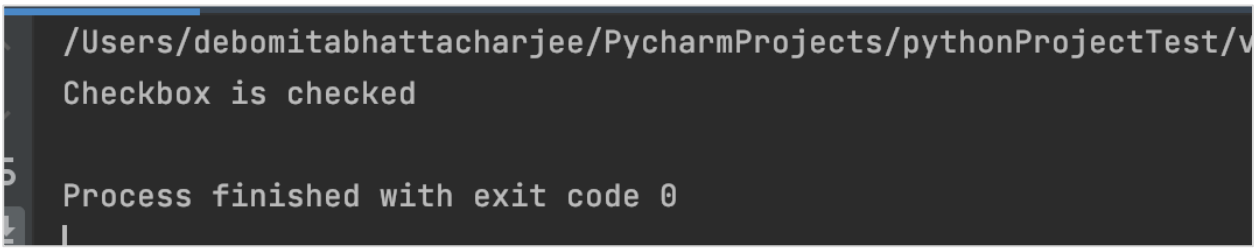
The code implementation for handling checkboxes is as follows:

```
from selenium import webdriver
```

```
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://the-internet.herokuapp.com/checkboxes")
#identify element
l = driver.find_element_by_xpath("//input[@type='checkbox']")
l.click()
if l.is_selected():
   print('Checkbox is checked')
else:
   print('Checkbox is not checked')
#close driver
driver.close()
```

Output

The output is as follows:



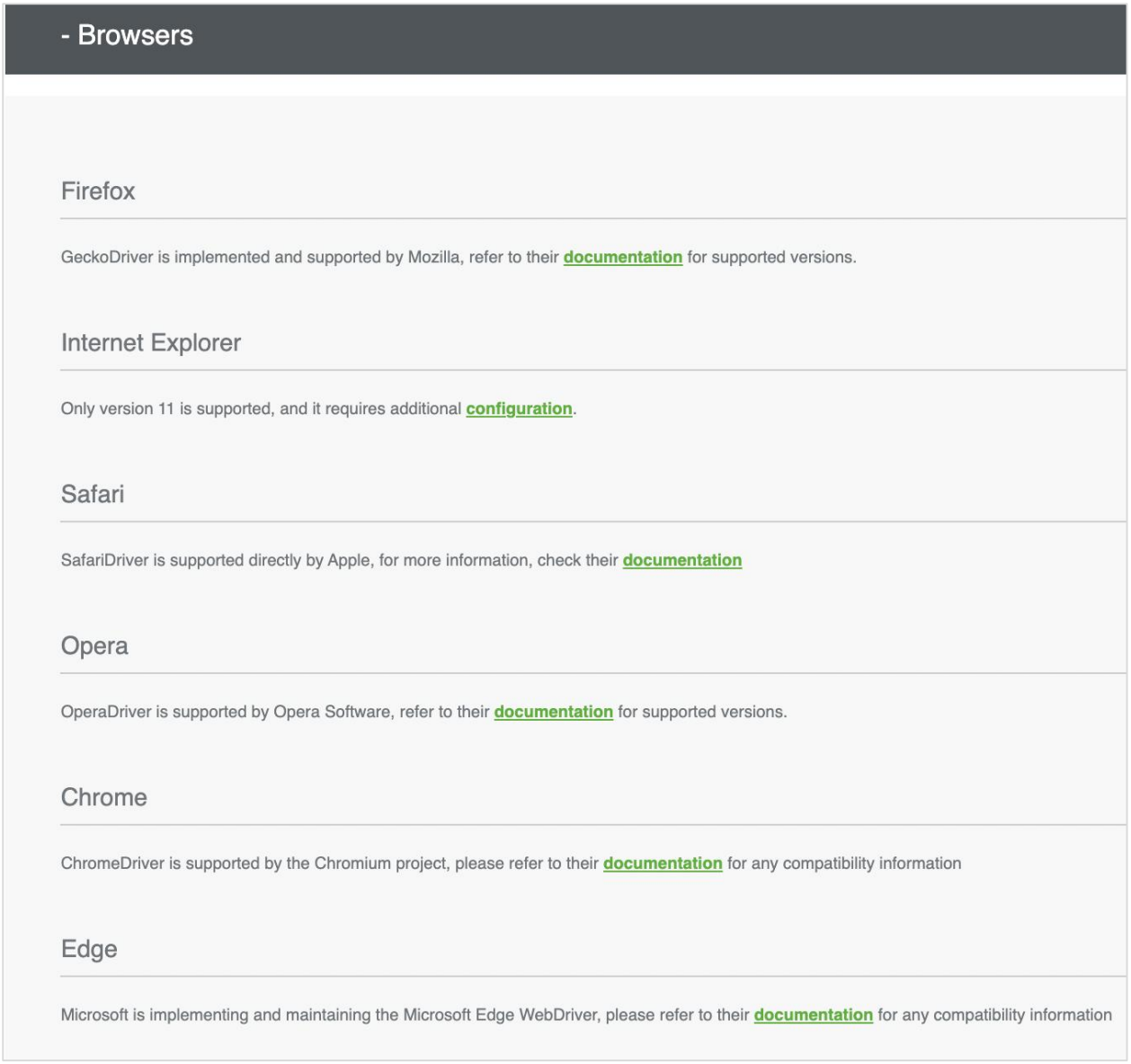
The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the message - Checkbox is checked is printed since the is_selected method applied on the checkbox returned true value.

23. Selenium Webdriver — Executing Tests in Multiple Browsers

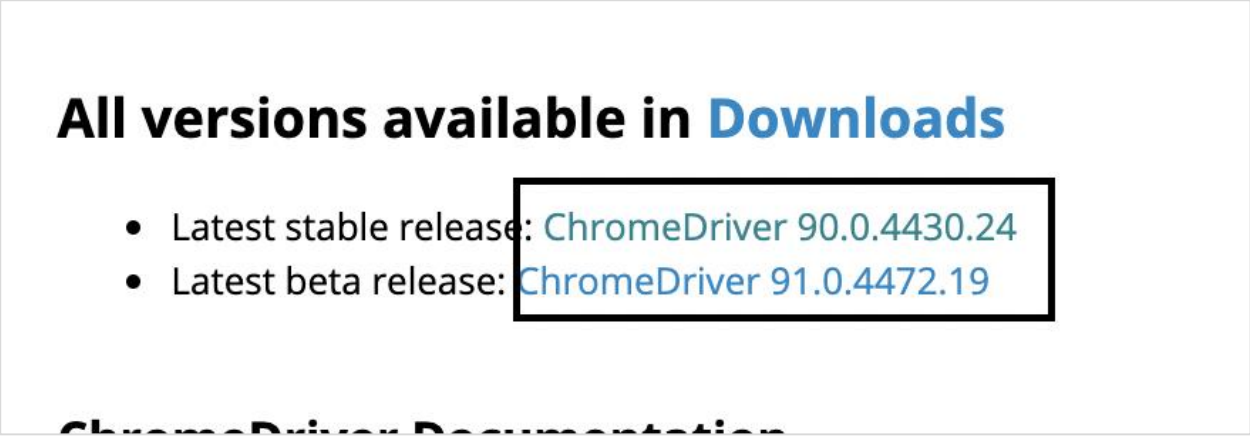
Selenium supports multiple browsers like Chrome, Firefox, Safari, IE, and so on. For running the tests in a particular browser we should have to download the executable file for that browser from the below link:

https://www.selenium.dev/downloads/

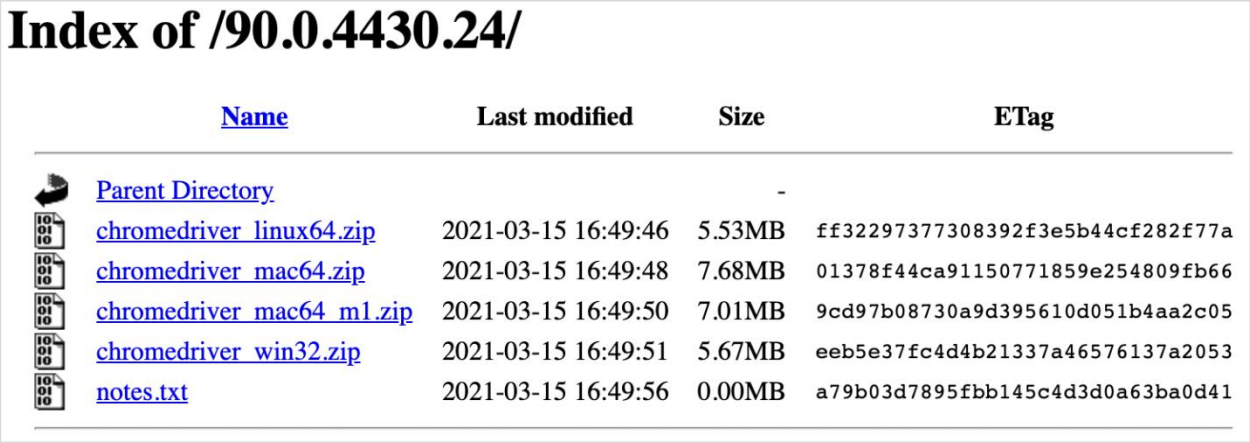
Once the link is launched, scroll down to the Browsers section. Under this, all the available browsers which support execution are listed. Click on the documentation link to download the corresponding executable file.



For example, to trigger the tests on Chrome, click on the documentation link. In the next page, the list of all the versions of chromedriver shall be available.



Click on a link to download the chromedriver.exe file which matches with our local Chrome browser version. On the following page, we shall be directed to the zip files available for download for the platforms Windows, Linux, and Mac.

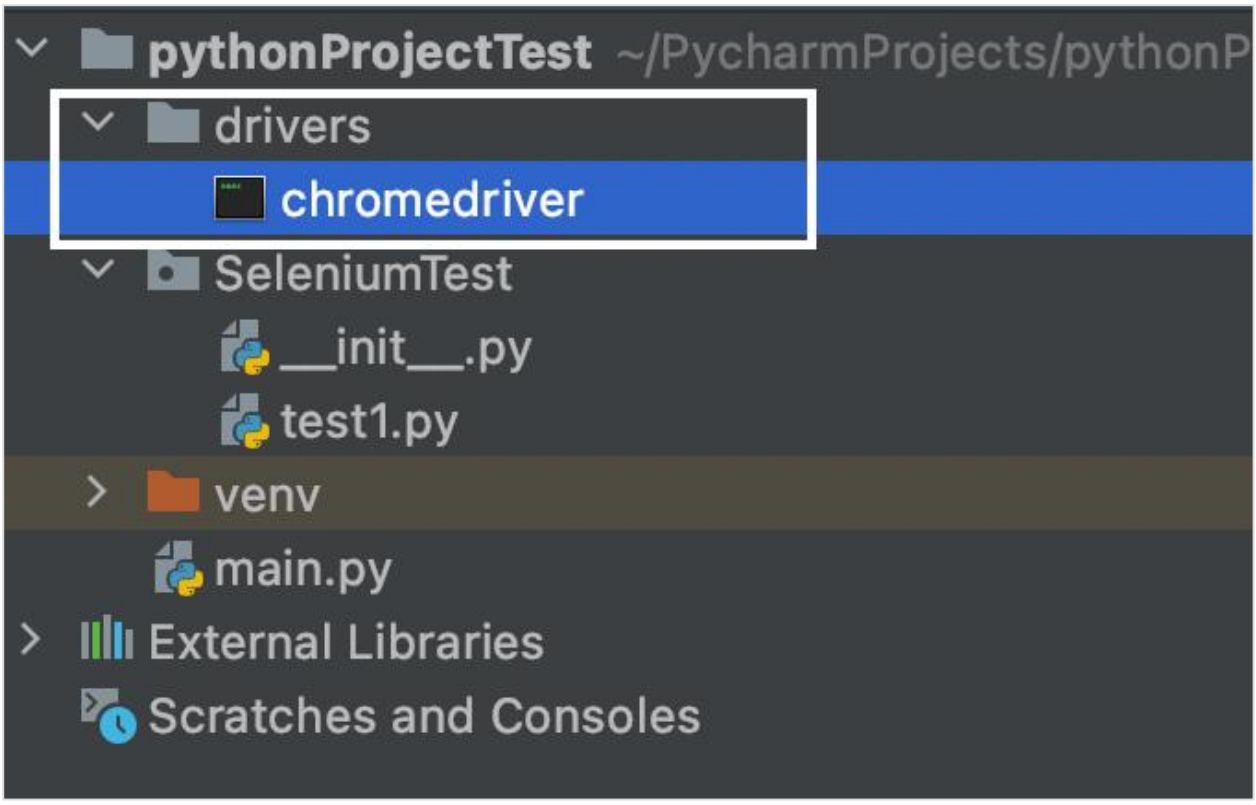


Index of /90.0.4430.24/

Name	Last modified	Size	ETag
Parent Directory		-	
chromedriver_linux64.zip	2021-03-15 16:49:46	5.53MB	ff32297377308392f3e5b44cf282f77a
chromedriver_mac64.zip	2021-03-15 16:49:48	7.68MB	01378f44ca91150771859e254809fb66
chromedriver_mac64_m1.zip	2021-03-15 16:49:50	7.01MB	9cd97b08730a9d395610d051b4aa2c05
chromedriver_win32.zip	2021-03-15 16:49:51	5.67MB	eeb5e37fc4d4b21337a46576137a2053
notes.txt	2021-03-15 16:49:56	0.00MB	a79b03d7895fbb145c4d3d0a63ba0d41

Click on a link to download the chromedriver.exe file which matches with our local operating system. Once the download is done, unzip the file and save it within the project directory.

For example in our project, we have saved the chromedriver.exe file within the drivers folder. Then we have to specify the path of this file within the webdriver.Chrome(executable_path='<path of chromedriver>').



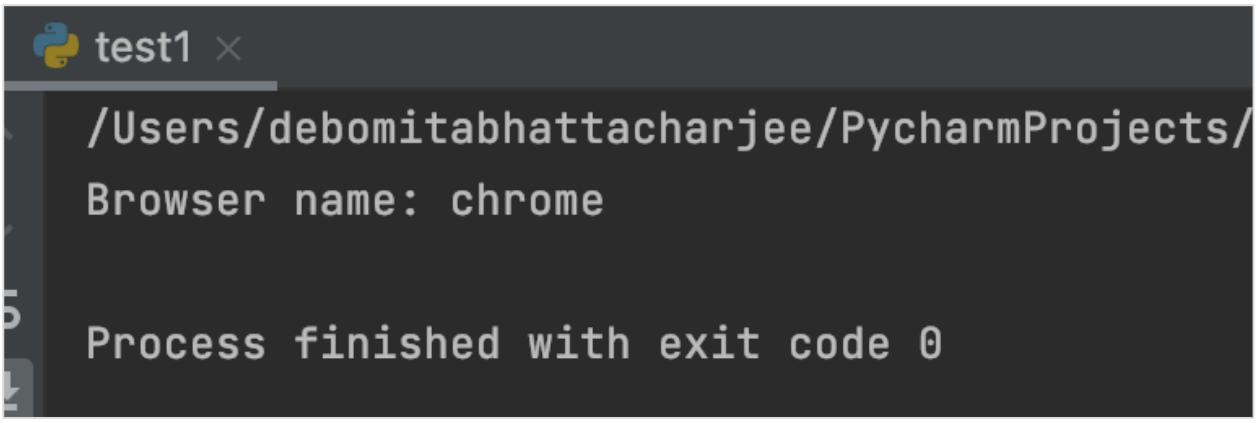
Code Implementation

The code implementation for supporting multiple browsers is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#get browse name
l = driver.capabilities['browserName']
print('Browser name: ' + l)
#driver quit
driver.quit()
```

Output

The output is as follows:



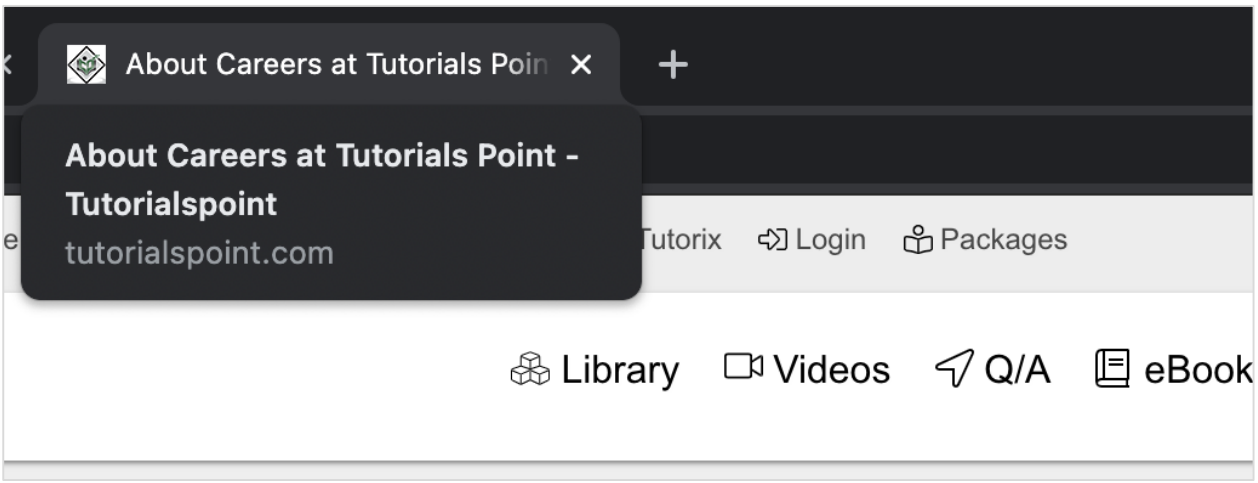
The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the browser in which the test has executed - chrome gets printed in the console.

Similarly, if we want to execute the test in the Firefox browser (versions greater than 47), we have to use the geckodriver.exe file.

24. Selenium Webdriver — Headless Execution

Selenium supports headless execution. In the Chrome browser, the headless execution can be implemented with the help of the ChromeOptions class. We have to create an object of this class and apply the add_arguments method on it. Finally, pass the parameter --headless to this method.

Let us obtain the title - About Careers at Tutorials Point - Tutorialspoint of the page launched in a headless mode:



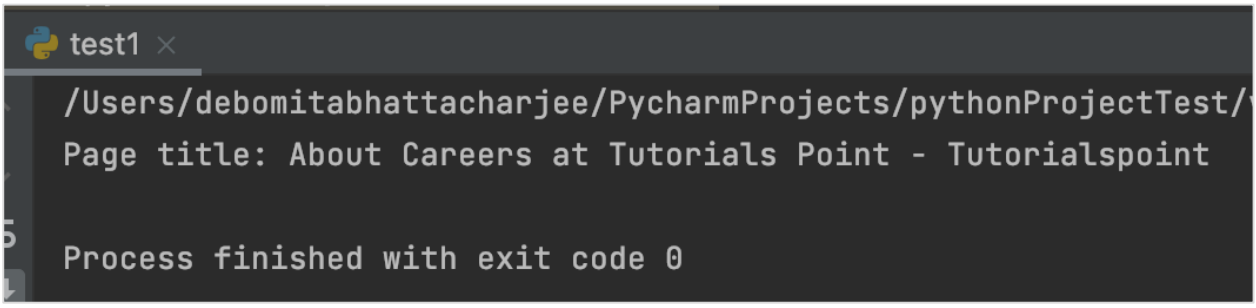
Code Implementation

The code implementation for the headless execution is as follows:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
#object of Options class
c = Options()
#passing headless parameter
c.add_argument("--headless")
#adding headless parameter to webdriver object
driver = webdriver.Chrome(executable_path='../drivers/chromedriver', options=c)
# implicit wait time
driver.implicitly_wait(5)
# url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
print('Page title: ' + driver.title)
# driver quit
driver.quit()
```

Output

The output is as follows



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application(obtained from the driver.title method) - About Careers at Tutorials Point - Tutorialspoint gets printed in the console.

25. Selenium Webdriver — Wait Support

Selenium provides wait support for implementations of explicit and fluent waits for synchronization. For this, we have to use the class selenium.webdriver.support.wait.WebDriverWait.

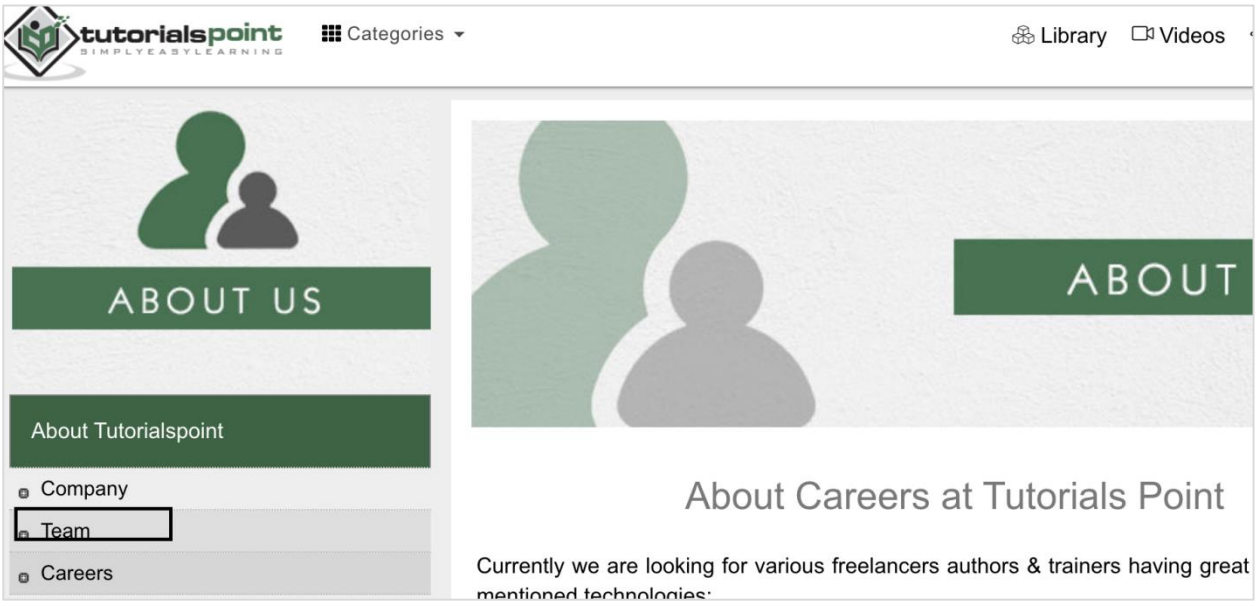
The **syntax** for the wait support is as follows:

```
w = WebDriverWait(driver, 5)
w.until(EC.presence_of_element_located((By.TAG_NAME, 'h1')))
```

Once we create an object of the WebDriverWait class, we can apply the below methods on them:

- **until**: It is used to invoke the method given with the driver as a parameter until the return value is true.
- **until_not**: It is used to invoke the method given with the driver as a parameter until the return value is not true.

Let us wait for the text Team @ Tutorials Point which becomes available on clicking the link - Team on the page with the help of WebDriverWait methods.



On clicking the Team link, the text Team @ Tutorials Point appears.

Code Implementation

The code implementation for wait support is as follows:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.support.wait import WebDriverWait
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify element
l = driver.find_element_by_link_text('Team')
l.click()
#expected condition for explicit wait
w = WebDriverWait(driver, 5)
w.until(EC.presence_of_element_located((By.TAG_NAME, 'h1')))
s = driver.find_element_by_tag_name('h1')
#obtain text
t = s.text
print('Text is: ' + t)
#driver quit
driver.quit()
```

Output

The output is as follows:

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/ve
Text is: Team @ Tutorials Point

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the text (obtained from the text method) - Team @ Tutorials Point gets printed in the console.

26. Selenium Webdriver — Select Support

Selenium can handle static dropdowns with the help of the Select class. A dropdown is identified with select tagname and its options are represented with the tagname option. The statement - from selenium.webdriver.support.select import Select should be added to work with Select class.

```
<select id="dropdown">
  <option value disabled="disabled" selected="selected">Please select an option</option>
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
</select>
```

Methods

The methods under the Select class are listed below:

select_by_visible_text (arg)

It shall select all the options which displayed text matches with the argument.

The syntax for selecting options displaying text matches is as follows:

```
sel = Select (driver.find_element_by_id ("name"))
sel.select_by_visible_text ('Visible Text')
```

select_by_value (arg)

It shall select all the options having a value that matches with the argument. The syntax for selecting all options having matching value as per the argument is as follows:

```
sel = Select (driver.find_element_by_id ("name"))
sel.select_by_value ('Value')
```

select_by_index (arg)

It shall select an option that matches with the argument. The index begins from zero.

The syntax for selecting the option having matching value as per the argument is as follows:

```
sel = Select (driver.find_element_by_id ("name"))
sel.select_by_index (1)
```

deselect_by_visible_text (arg)

It shall deselect all the options which displayed text matches with the argument.

The syntax for deselecting all options having matching value as per the argument is as follows:

```
sel = Select (driver.find_element_by_id ("name"))
sel.deselect_by_visible_text ('Visible Text')
```

deselect_by_value (arg)

It shall deselect all the options having a value that matches with the argument.

The syntax for deselecting all options having matching value as per the argument is as follows:

```
sel = Select (driver.find_element_by_id ("name"))
sel.deselect_by_value ('Value')
```

deselect_by_index(arg)
It shall deselect the option that matches with the argument. The index begins from zero.

The syntax for deselecting an option having matching value as per the argument is as follows:

```
sel = Select(driver.find_element_by_id ("name"))
sel.deselect_by_index(1)
```

all_selected_options

It shall yield all the options which are selected for a dropdown.

first_selected_option

It shall yield the first selected option for a multi-select dropdown or the currently selected option in a normal dropdown.

options

It shall yield all the options available under the select tagname.

deselect_all

It shall clear all the selected options in a multi-select dropdown.

Code Implementation

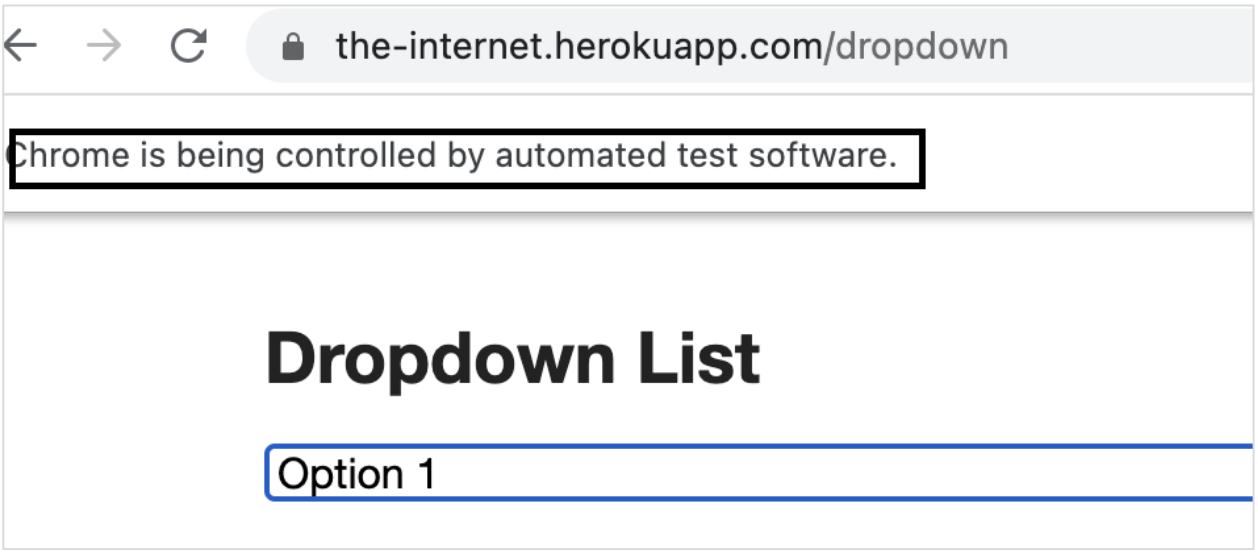
The code implementation for handling static dropdowns with Select class is as follows:

```
from selenium import webdriver
from selenium.webdriver.support.select import Select
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
```

```
driver.implicitly_wait(5)
#url launch
driver.get("https://the-internet.herokuapp.com/dropdown")
#object of Select
s= Select(driver.find_element_by_id("dropdown"))
#select option by value
s.select_by_value("1")
```

Output

The output is as follows:



The output shows that the option "**Option 1**" gets selected in the dropdown.

27. Selenium Webdriver — JavaScript Executor

Selenium can execute JavaScript commands with the help of the execute_script method. The command to be executed is passed as a parameter to this method. We can perform browser operations like clicking a link with the help of the JavaScript Executor.

The **syntax** for executing the Javascript commands is as follows:

```
b = driver.find_element_by_id("txt")
 driver.execute_script ("arguments[0].click();",b)
```

Code Implementation

The code implementation for executing the Javascript commands is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#click with JavaScript Executor
b = driver.find_element_by_link_text("Cookies Policy")
driver.execute_script ("arguments[0].click();",b)
print('Page title after click: '+ driver.title)
#driver quit
driver.quit()
```

Output

The output is as follows

```
/Users/debomitabhattacharjee/PycharmProjects/pythonProjectTest/venv/bin/python /U
Page title after click: About Cookies Policy at Tutorials Point - Tutorialspoint

Process finished with exit code 0
```

The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application after the click (obtained from the driver.title method) - About Cookies Policy at Tutorials Point - Tutorialspoint gets printed in the console.

execute_script

Selenium cannot directly handle scrolling functionality directly. Selenium can execute JavaScript commands with the help of the method - execute_script. The JavaScript command to be executed is passed as a parameter to this method.

The **syntax** for executing the Javascript commands with the help of execute_script method is as follows:

```
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

The method scrollTo is used to scroll to a location in the browser window. The scrollHeight is a property of an element. The document.body.scrollHeight yields the height of the webpage.

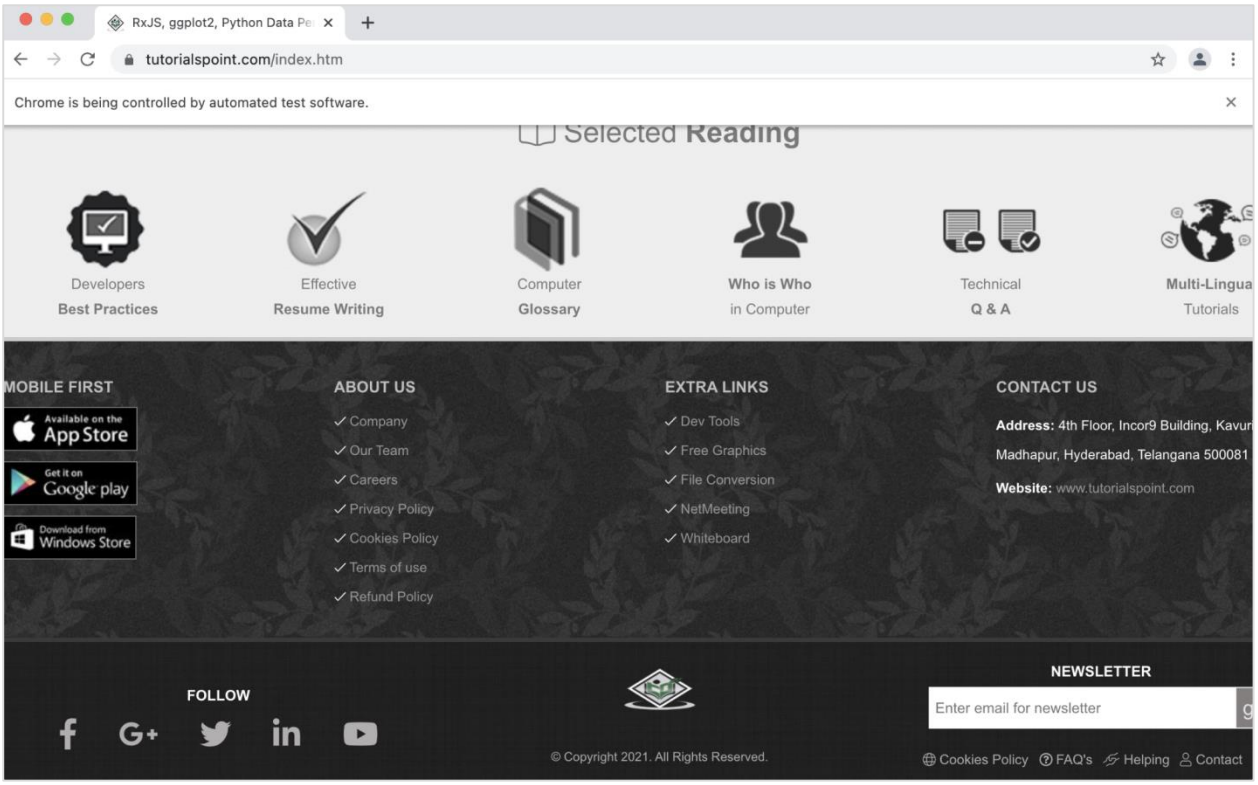
Code Implementation

The code implementation for executing the Javascript commands with the help of execute_script method is as follows is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#scroll to page bottom
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

Output

The output is as follows:



The output shows that the web page is scrolled to the bottom of the page.

28. Selenium Webdriver — Chrome WebDriver Options

Selenium Chrome webdriver Options are handled with the class - **selenium.webdriver.chrome.options.Options**.

Methods

Some of the methods of the above mentioned class are listed below:

- **add_argument(args)**: It is used to append arguments to a list.
- **add_encoded_extension(ext):**It is used to append base 64 encoded string and the extension data to a list that will be utilised to get it to the ChromeDriver.
- **add_experimental_option(n, val):** It is used to append an experimental option which is passed to the Chrome browser.
- **add_extension(ext):** It is used to append the extension path to a list that will be utilised to get it to the ChromeDriver.
- **set_capability(n, val):** It is used to define a capability.
- **to_capabilities(n, val):** It is used to generate capabilities along with options and yields a dictionary with all the data.
- **arguments:**It is used to yield arguments list required for the browser.
- **binary_location:** It is used to obtain the binary location. If there is no path, an empty string is returned.
- **debugger_address:** It is used to yield the remote devtools object.

experimental_options: It is used to yield a dictionary of the Chrome experimental options.

- **extensions:** It is used to yield an extensions list which shall be loaded to the Chrome browser.
- **headless:**It is used to check if the headless argument is set or not.

Code Implementation

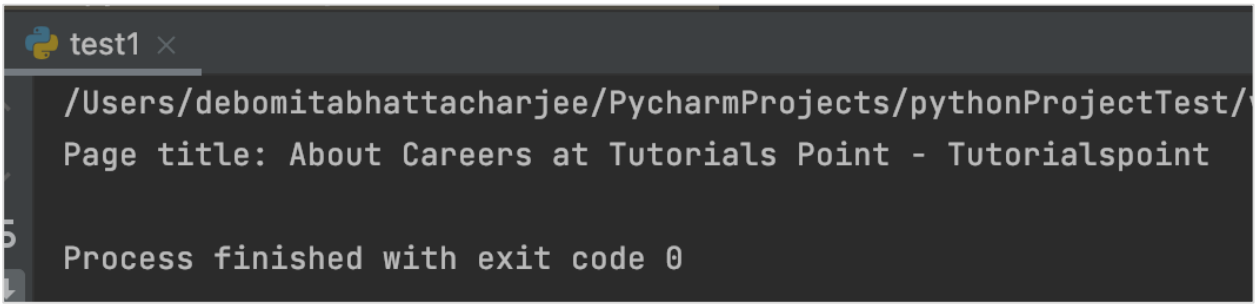
The code implementation for the Selenium Chrome Webdriver options is as follows:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
#object of Options class
c = Options()
#passing headless parameter
```

```
c.add_argument("--headless")
#adding headless parameter to webdriver object
driver = webdriver.Chrome(executable_path='../drivers/chromedriver', options=c)
# implicit wait time
driver.implicitly_wait(5)
# url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
print('Page title: ' + driver.title)
# driver quit
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the page title of the application(obtained from the driver.title method) - About Careers at Tutorials Point - Tutorialspoint gets printed in the console.

29. Selenium Webdriver — Scroll Operations

Selenium cannot directly handle scrolling functionality directly. Selenium can execute JavaScript commands with the help of the method - execute_script. The JavaScript command to be executed is passed as a parameter to this method.

The **syntax** for executing the Javascript commands is as follows:

```
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

The method scrollTo is used to scroll to a location in the browser window. The scrollHeight is a property of an element. The document.body.scrollHeight yields the height of the webpage.

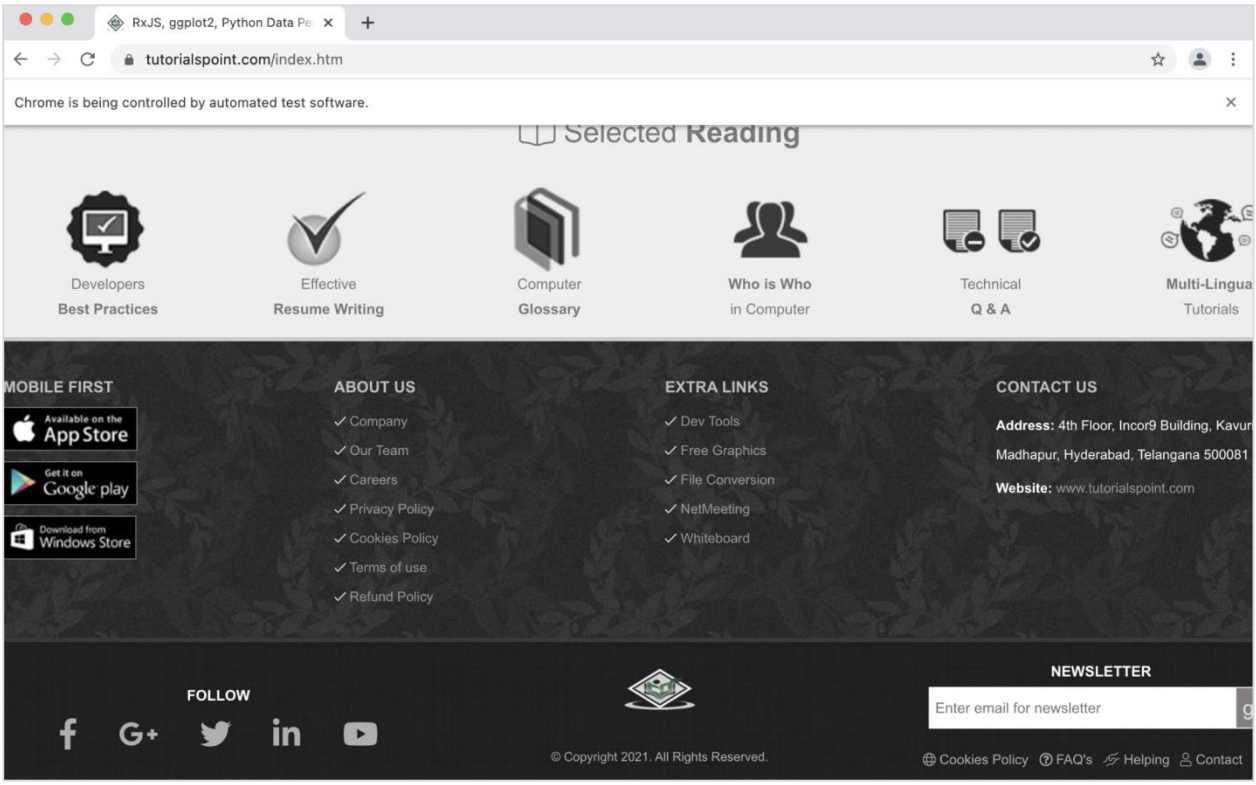
Code Implementation

The code implementation for executing the Javascript commands is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#scroll to page bottom
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
```

Output

The output is as follows:



The output shows that the web page is scrolled to the bottom of the page.

30. Selenium Webdriver — Capture Screenshots

We can capture screenshots with the Selenium webdriver using the save_screenshot method. The path of the screenshot captured is passed as a parameter to this method.

The **syntax** for capturing the screenshot is as follows:

```
driver.save_screenshot('logo.png')
```

Here, an image with the name logo.png should get saved within the project.

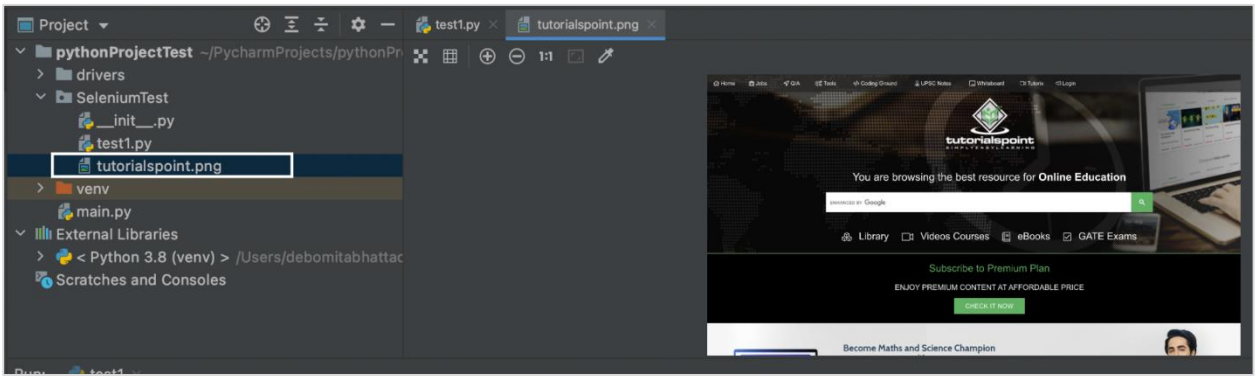
Code Implementation

The code implementation for capturing the screenshot is as follows:

```
from selenium import webdriver
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/index.htm")
#capture screenshot - tutorialspoint.png within project
driver.save_screenshot('tutorialspoint.png')
#close driver
driver.close()
```

Output

The output is as follows:



The output shows that an image tutorialspoint.png gets created within the project. It contains the captured screenshot.

31. Selenium Webdriver — Right Click

Selenium can perform mouse movements, key press, hovering on an element, right-click, drag and drop actions, and so on with the help of the ActionsChains class. The method context_click performs right-click or context click on an element.

The **syntax** for using the right click or context click is as follows:

```
context_click(e=None)
```

Here, e is the element to be right-clicked. If '**None**' is mentioned, the click is performed on the present mouse position. We have to add the statement from selenium.webdriver import ActionChains to work with the ActionChains class.

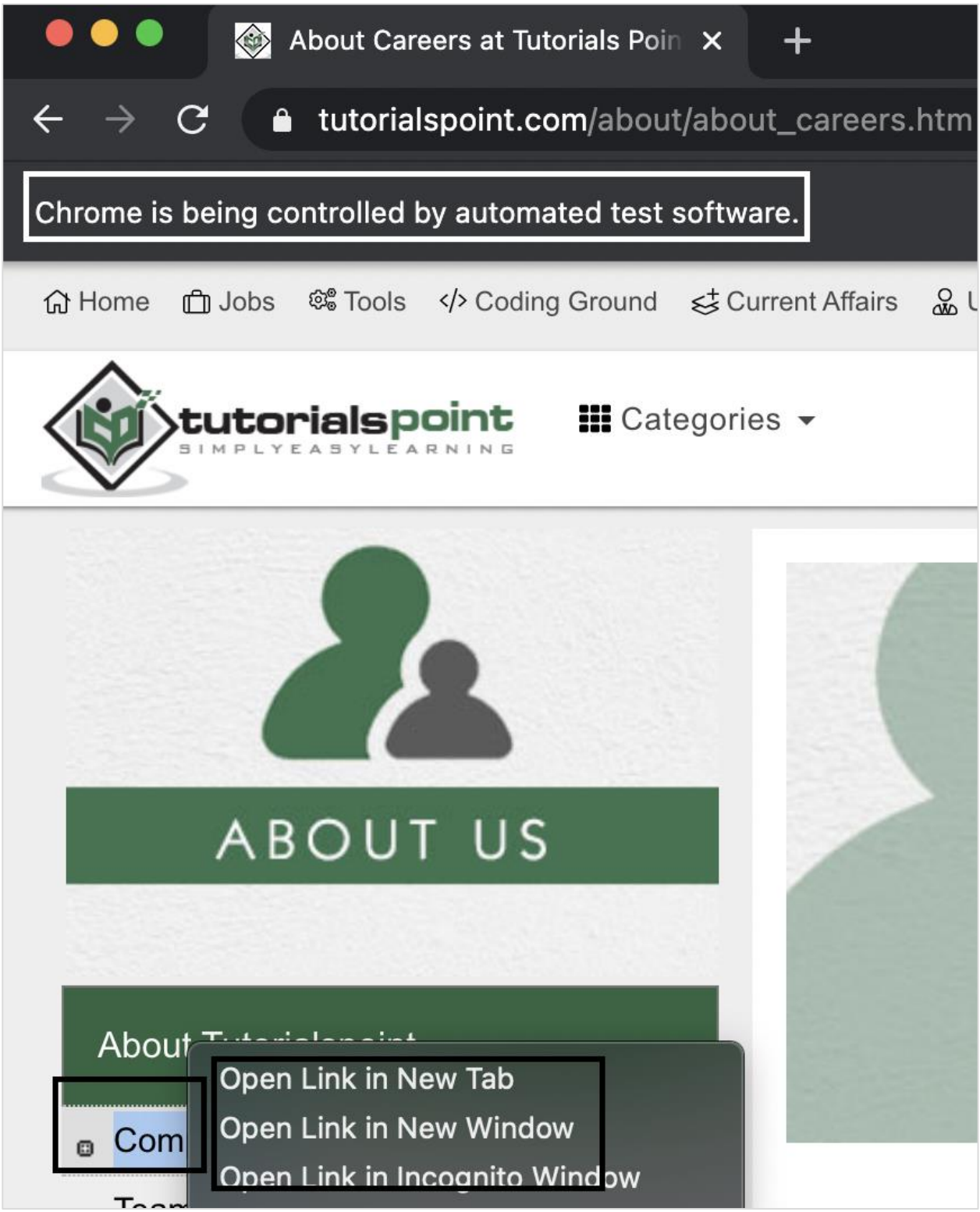
Code Implementation

The code implementation for using the right click or context click is as follows:

```
from selenium import webdriver
from selenium.webdriver import ActionChains
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("https://www.tutorialspoint.com/about/about_careers.htm")
#identify element
s = driver.find_element_by_xpath("//*[text()='Company']")
#object of ActionChains
a = ActionChains(driver)
#right click then perform
a.context_click(s).perform()
```

Output

The output is as follows:



After execution, the link with the name - Company has been right-clicked and all the new options get displayed as a result of the right-click.

32. Selenium Webdriver — Double Click

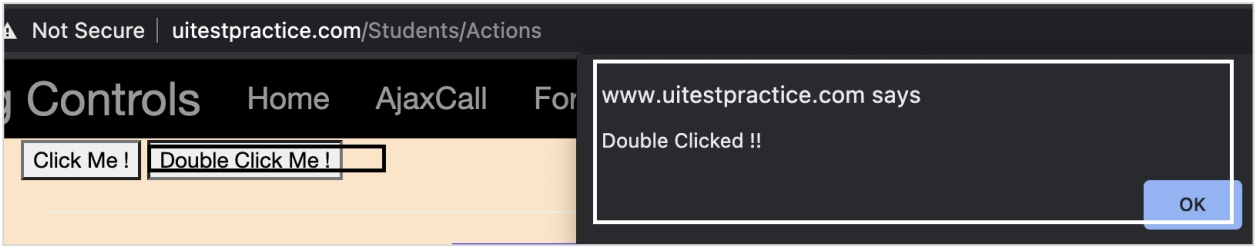
Selenium can perform mouse movements, key press, hovering on an element, double click, drag and drop actions, and so on with the help of the ActionsChains class. The method double_click performs double-click on an element.

The **syntax** for using the double click is as follows:

```
double_click(e=None)
```

Here, e is the element to be double-clicked. If None is mentioned, the click is performed on the present mouse position. We have to add the statement from selenium.webdriver import ActionChains to work with the ActionChains class.

Let us perform the double click on the below element:



In the above image, it is seen that on double clicking the Double Click me! button, an alert box gets generated.

Code Implementation

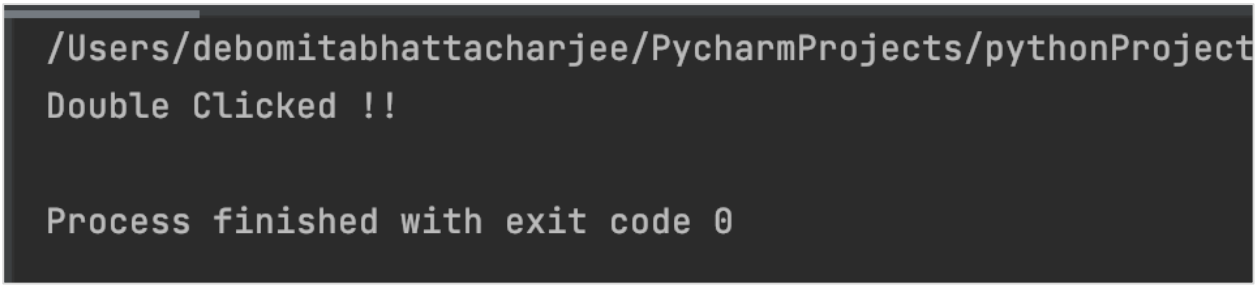
The code implementation for using the double click is as follows:

```
from selenium import webdriver
from selenium.webdriver import ActionChains
from selenium.webdriver.common.alert import Alert
driver = webdriver.Chrome(executable_path='../drivers/chromedriver')
#implicit wait time
driver.implicitly_wait(5)
#url launch
driver.get("http://www.uitestpractice.com/Students/Actions")
#identify element
s = driver.find_element_by_name("dblClick")
#object of ActionChains
a = ActionChains(driver)
#right click then perform
a.double_click(s).perform()
```

```
#switch to alert
alrt = Alert(driver)
# get alert text
print(alrt.text)
#accept alert
alrt.accept()
#driver quit
driver.quit()
```

Output

The output is as follows:



The output shows the message - Process with exit code 0 meaning that the above Python code executed successfully. Also, the Alert text - Double Clicked! gets printed in the console. The Alert got generated by double clicking the Double Click me! button.