BROWSER AUTOMATION USING PYTHON-SELENIUM

SANDEEP SURYAPRASAD

COURSE INDEX

USEFUL LINKS

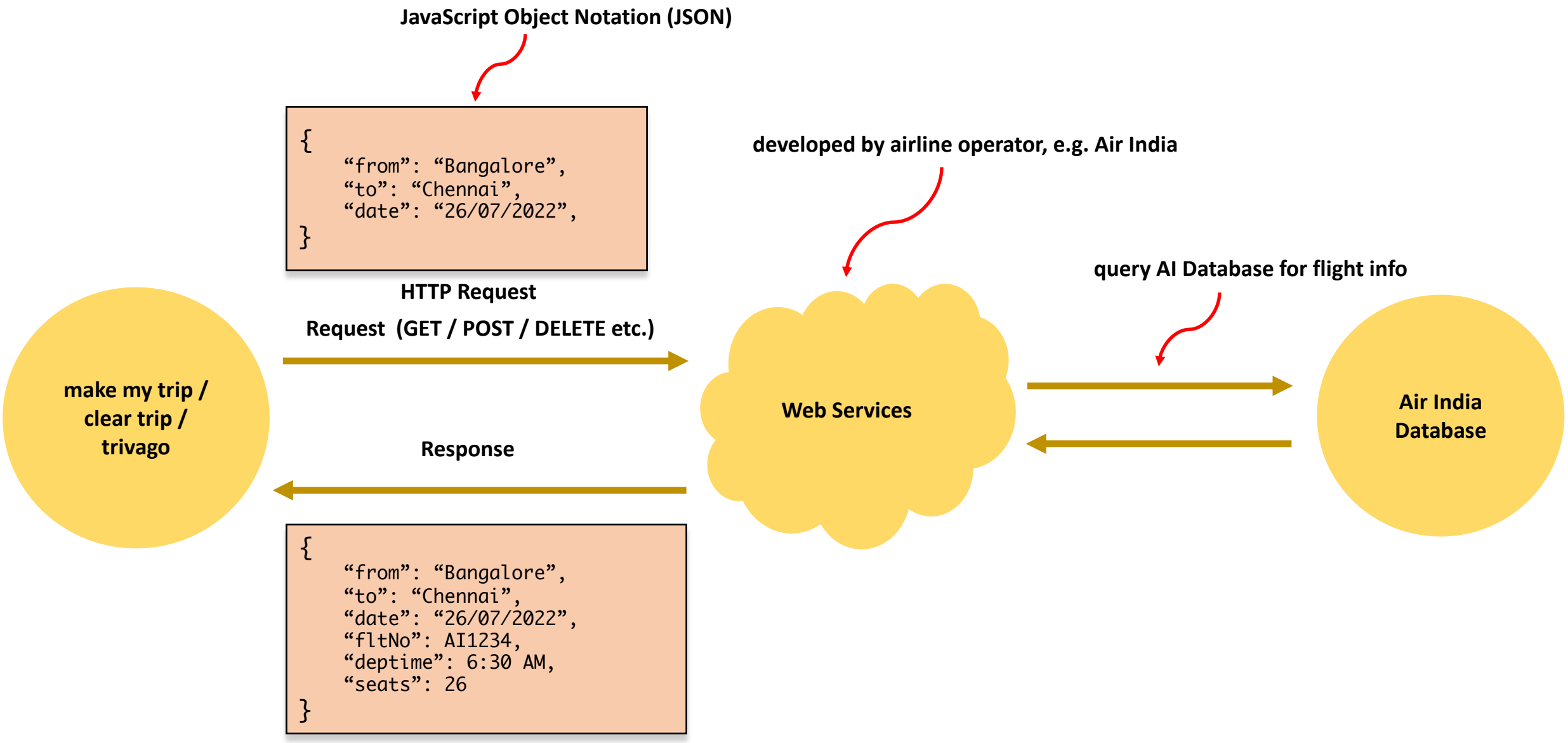
Demo Websites for Practice

- http://demowebshop.tricentis.com/
- https://services.smartbear.com/samples/TestComplete14/smartstore/
- https://demo.actitime.com/login.do
- https://opensource-demo.orangehrmlive.com/index.php/auth/login

Selenium Documentation

- https://www.selenium.dev/documentation/en/

ONLINE FLIGHT RESERVATION

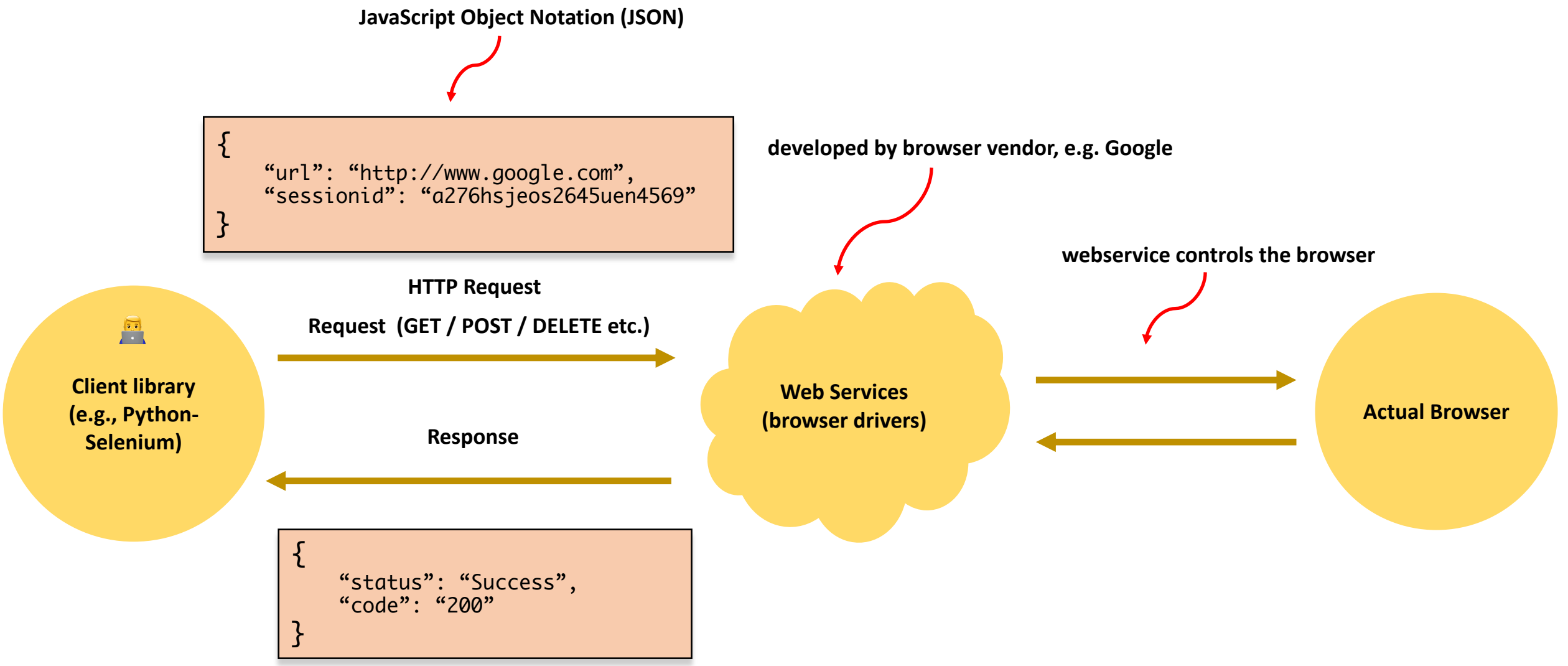


BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

SELENIUM ARCHITECTURE

SELENIUM ARCHITECTURE



CLIENT LIBRARIES / LANGUAGE BINDINGS

- 👉 Selenium supports multiple languages such as Python, Java, C#, JavaScript, Ruby.
- 👉 Client libraries provide various methods to perform different browser actions. e.g. get, title, find_element
- 👉 As automation developers we call these methods from our development environment. e.g. VSCode
- 👉 Once we execute the script, the client libraries convert our code that we have written into a JSON (JavaScript Object Notation) format and sent as a request to Driver over HTTP.

BROWSER DRIVERS

- Each browser has its own implementation of **"WebDriver" protocol** (mandatory services that need's to be implemented in order for selenium to interact with browser) called drivers.
- The browser drivers are responsible for controlling the actual browser's since the browser implementation details will be known only to the developer of driver.
- Each browser driver will be maintained by respective browser vendor. e.g. Chrome Driver is maintained by Google and Safari Driver is maintained by Apple.
- Each method in the client library is mapped to a specific web-service in the driver.
- The driver interprets the incoming request from the client and controls the actual browser.
- Once the browser operation is complete, the response is sent back to the client/client library by driver in JSON format.
- Client library interprets the JSON response and prints the response in readable format in the VSCode console.

BROWSER AUTOMATION
USING
PYTHON-SELENIUM

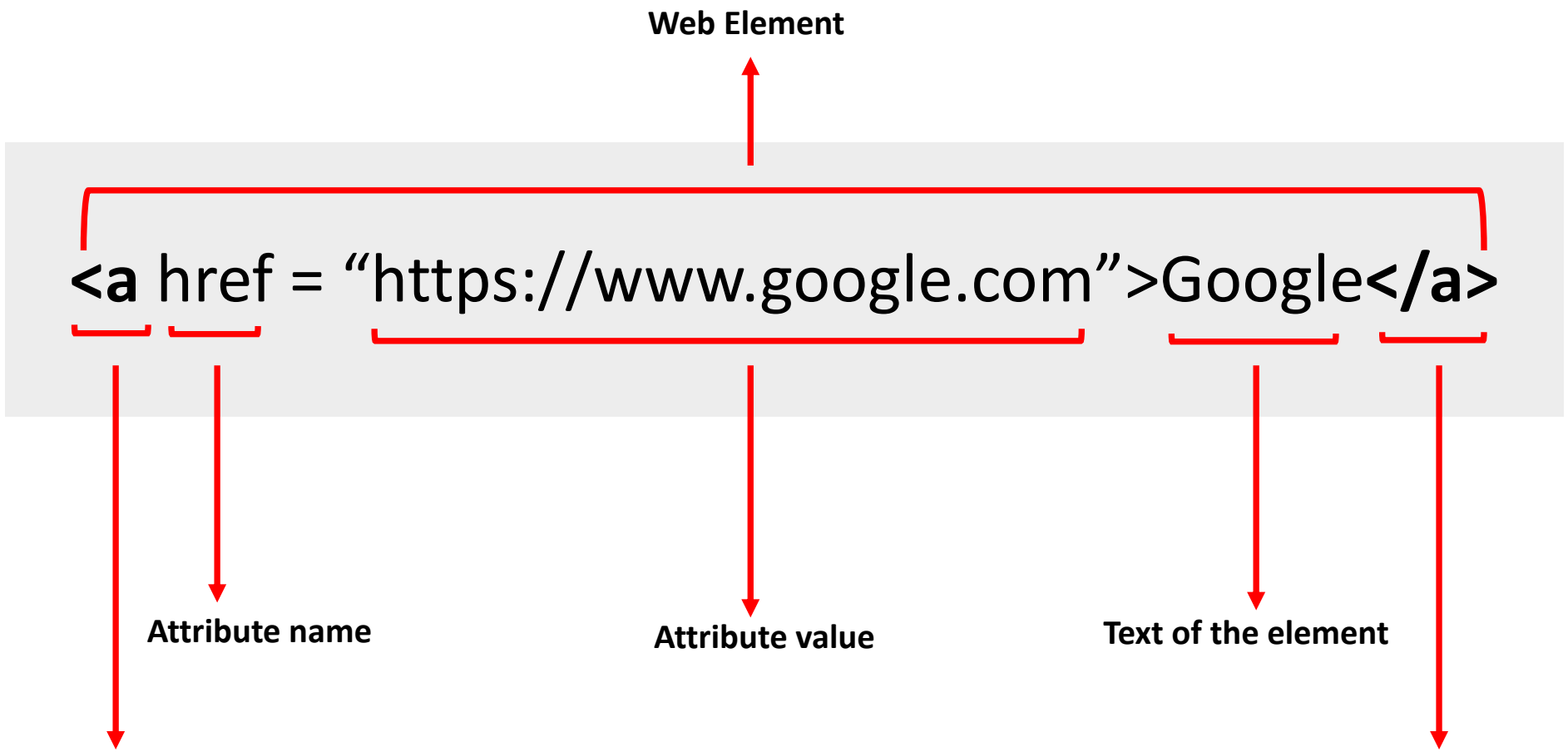
SANDEEP SURYAPRASAD

HTML FUNDAMENTALS

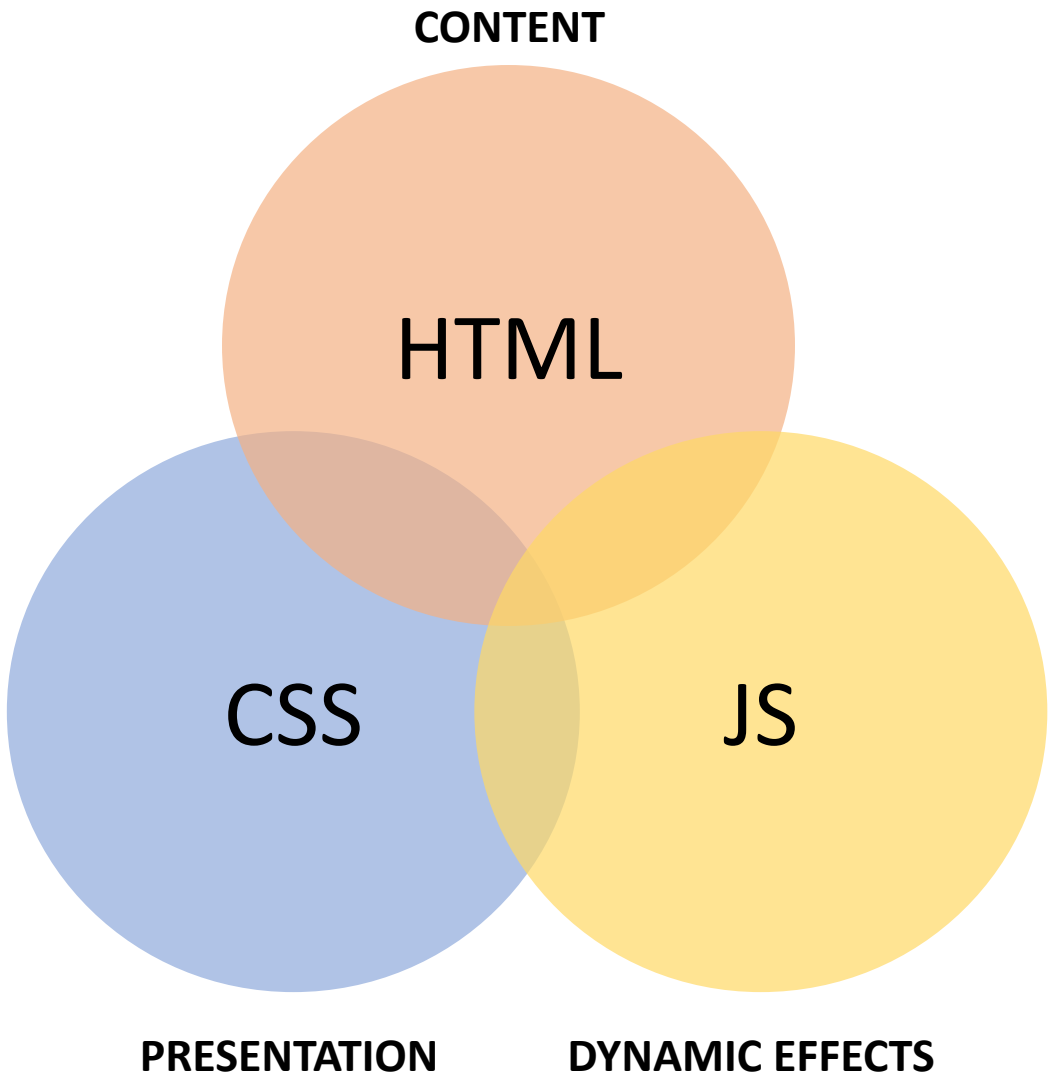
HTML

- **H**yper **T**ext **M**arkup **L**anguage
- HTML is a markup language that web developers use **to structure and design the content** of a webpage (HTML is not a programming language)
- HTML consists of **elements (web elements)** that describe different types of contents like links, headings, images, text boxes, radio buttons, check-boxes etc.
- Browser understands HTML and **renders HTML code as websites**

HTML ELEMENT

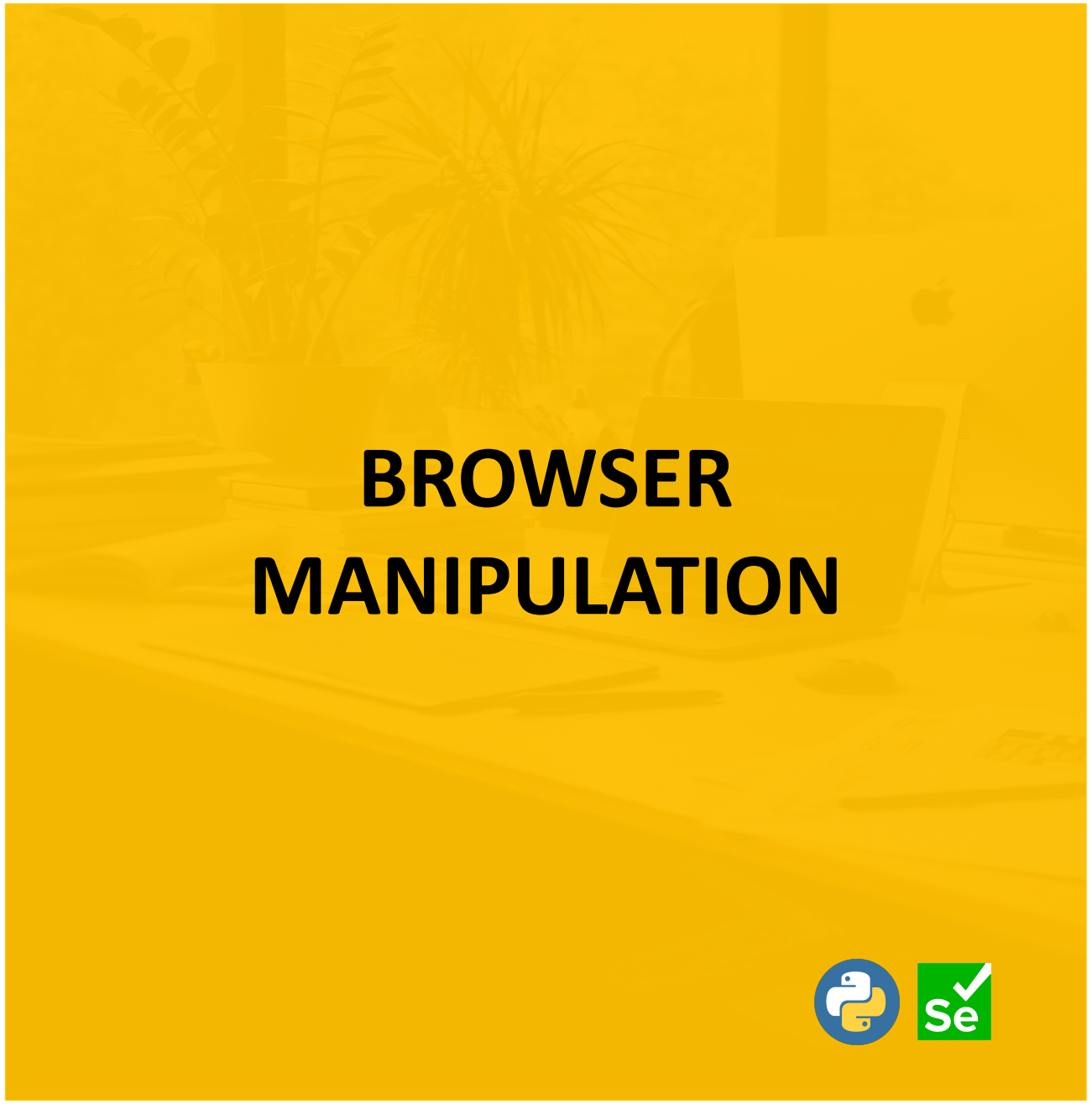


3 LANGUAGES OF FRONT-END



BROWSER AUTOMATION USING PYTHON-SELENIUM

SANDEEP SURYAPRASAD



LAUNCHING DIFFERENT BROWSERS

Path of driver executable

```
from selenium import webdriver

driver = webdriver.Chrome("./chromedriver")
```

Creating an instance of Chrome Browser

```
from selenium import webdriver

driver = webdriver.Firefox("./geckodriver")
```

Creating an instance of Firefox Browser

```
from selenium import webdriver

driver = webdriver.Safari()
```

Creating an instance of Safari Browser

Path of safari driver will be automatically taken from the path /usr/bin

COMMON BROWSER ACTIONS

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
```

Launch URL

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
driver.maximize_window()
```

Maximizes the browser

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
driver.minimize_window()
```

Minimizes the browser

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
driver.refresh()
```

Refresh the browser

COMMON BROWSER ACTIONS

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
current_title = driver.title
print(current_title)
```

Fetches the title of the webpage

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
url = driver.current_url
print(url)
```

Fetches the current URL of the webpage

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
driver.quit()
```

Closes the current browser session including any pop-up windows opened by selenium

```
from selenium import webdriver
driver = webdriver.Chrome("./chromedriver")
driver.get("http://www.google.com")
driver.close()
```

Closes the current browser session, but does not close any pop-up windows opened by selenium

BROWSER AUTOMATION USING PYTHON-SELENIUM

SANDEEP SURYAPRASAD

LOCATING ELEMENTS

- In order for selenium to identify elements on the webpage, selenium makes uses of 8 different locators to identify the objects.
- There are 8 different built-in element location strategies in Webdriver

Locator	Description
name	Locates elements whose NAME attribute matches the search value
id	Locates elements whose ID attribute matches the search value
link text	Locates anchor elements (link) whose visible text matches the search value
partial link text	Locates anchor elements (link) whose visible text contains the search value. If multiple elements are matching, only the first one will be selected.
css selector	Locates elements matching a CSS selector
class name	Locates elements whose class name contains the search value (compound class names are not permitted)
xpath	Locates elements matching an XPath expression
tag name	Locates elements whose tag name matches the search value

LOCATING ELEMENTS

find_element

- `find_element` method returns a web element if the element is found in the DOM/HTML.
- If no element is found, `find_element` method throws "`NoSuchElementException`"
- If there are multiple elements matching the same locator, `find_element` method returns the first matching element from the DOM.

LOCATING ELEMENTS

`find_elements`

- Returns list of web elements.
- Each item of the list is a web element.
- If there are no web elements which matches the given property and its value, `find_elements` method returns an empty list.

LOCATING ELEMENTS

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("chromedriver")
driver.get('file:///Users/sandeep/Desktop/demo-html/demo.html')
elements = driver.find_elements_by_name("download")

for element in elements:
    element.click()
```

Check all the checkbox's

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("chromedriver")
driver.get('file:///Users/sandeep/Desktop/demo-html/demo.html')
elements = driver.find_elements_by_name("download")

for element in elements[::-1]:
    element.click()
```

Check all the checkbox's in reversed order

LOCATING ELEMENTS

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("chromedriver")
driver.get('file:///Users/sandeep/Desktop/demo-html/demo.html')
elements = driver.find_elements_by_name("download")

for element in elements[::2]:
    element.click()
```

Check alternate checkbox's

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("chromedriver")
driver.get('file:///Users/sandeep/Desktop/demo-html/demo.html')
elements = driver.find_elements_by_name("download")

elements[0].click()
elements[-1].click()
```

Check alternate checkbox's

LOCATING ELEMENTS

xpath

- xpath stands for "**path of the element**" in **HTML Tree**

- There are two kinds of xpath,
  Absolute xpath
  Relative xpath

- **Absolute Xpath**: Specifying the complete or absolute path of the element is called absolute xpath.
  e.g. /html/body/div/input (single forward slash represents immediate)

- **Relative Xpath**: Specifying the relative path of the element is called relative xpath.
  e.g. //input,
  //input[@name="fname"]

LOCATING ELEMENTS

xpath

- General Syntax of Relative Xpath

  //HTMLTAG[@attributename="attribute_value"]

  e.g. //input[@id="Firstname"]

  //input[@name="Lastname"]

- If the text of the element is used in the xpath, below is the general syntax

  //HTMLTAG[text()="actual_text"]

  e.g. //a[text()="Register"]

- If the text or attribute value of the element has leading or/and trailing white spaces, we need to use "**contains**"

  //HTMLTAG[contains(@attribute, "attribute_value")]

  //HTMLTAG[contains(text(), "actual_text")]

XPATH

dependent-independent technique

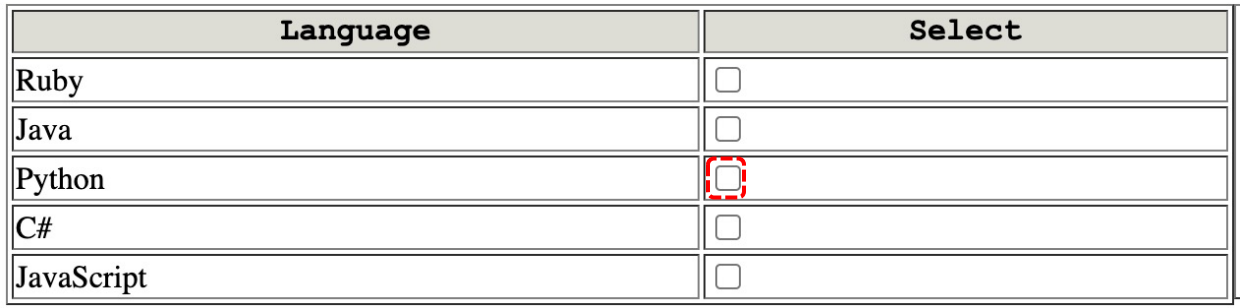
xpath of checkbox corresponding to row “**Python**”, “**Java**”, “**JavaScript**”, “**Ruby**”

```
//td[text()='Python']/..//input[@name='download']
```

```
//td[text()=Java']/..//input[@name='download']
```

```
//td[text()=JavaScript']/..//input[@name='download']
```

```
//td[text()=Ruby']/..//input[@name='download']
```



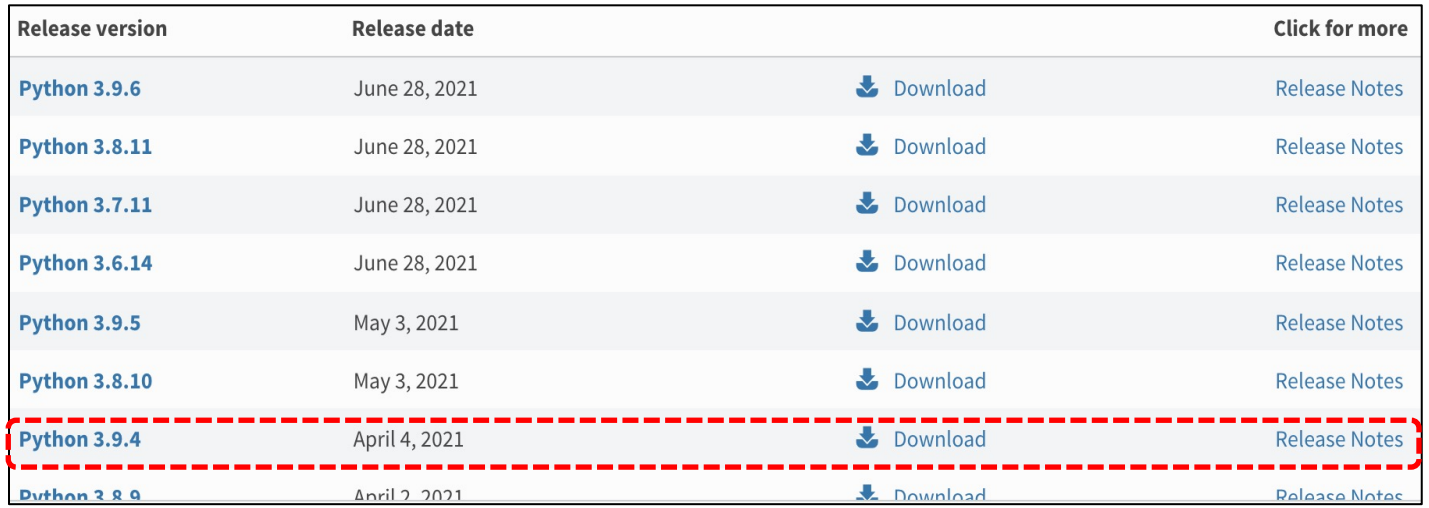
Language	Select
Ruby	☐
Java	☐
Python	☐
C#	☐
JavaScript	☐

XPATH

dependent-independent technique

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

xpath of “**Release Notes**” link of python release version **3.9.4**



Release version	Release date		Click for more
Python 3.9.6	June 28, 2021	Download	Release Notes
Python 3.8.11	June 28, 2021	Download	Release Notes
Python 3.7.11	June 28, 2021	Download	Release Notes
Python 3.6.14	June 28, 2021	Download	Release Notes
Python 3.9.5	May 3, 2021	Download	Release Notes
Python 3.8.10	May 3, 2021	Download	Release Notes
Python 3.9.4	April 4, 2021	Download	Release Notes
Python 3.8.9	April 2, 2021	Download	Release Notes

```
//a[text()='Python 3.9.4']/../..//a[text()='Release Notes']
```

XPATH

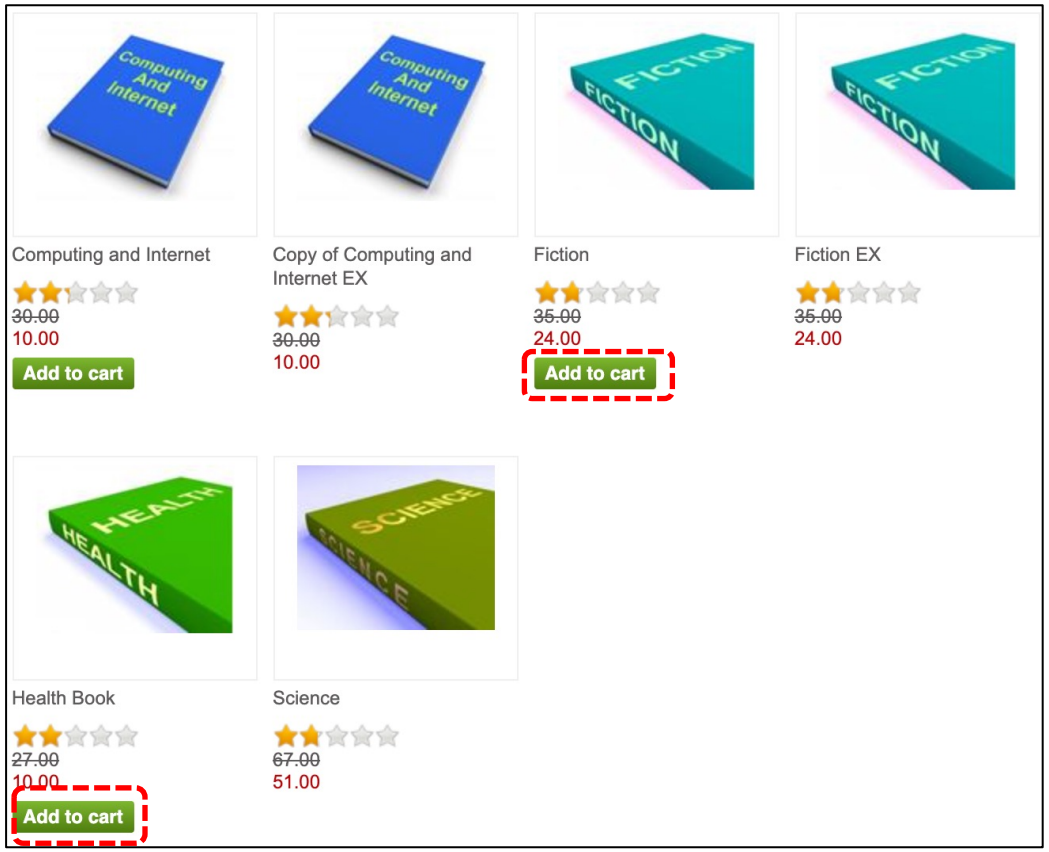
dependent–independent technique

http://demowebshop.tricentis.com/books

xpath of “**Add to cart**” button of “**Health**” and “**Fiction**” books

```
//a[text()=Health']/../..//input[@value='Add to cart']
```

```
//a[text()='Fiction']/../..//input[@value='Add to cart']
```



XPATH

dependent-independent technique

validate the prices of all the gadgets against the expected price

```
from selenium import webdriver
from time import sleep
driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)
# dictionary with Gadget name and its expected price
expected_prices = { "$25 Virtual Gift Card": 25.00,
                    "14.1-inch Laptop": 1590.00,
                    "Build your own cheap computer": 800.00,
                    "Build your own computer": 1200.00,
                    "Build your own expensive computer": 1800.00,
                    "Simple Computer": 800.00
                  }
# Iterate through the dictionary, get the gadget name and its corresponding expected prices.
for gadget, price in expected_prices.items():
    _xpath = f"//a[text()='{gadget}']/../..//span[@class='price actual-price']"
    actual_price = driver.find_element_by_xpath(_xpath).text
    # Compare actual and expected prices, print "PASS" or "FAIL"
    # Before comparing, typecast the actual_price which is read from the application to float()
    # Since actual price which is returned (by text property) will be a string.
    if float(actual_price) == price:
        print('PASS')
    else:
        print('FAIL')
    # You can as well assert the actual and expected prices
    # assert float(actual_price) == price
```

http://demowebshop.tricentis.com/

XPATH

dependent–independent technique

http://demowebshop.tricentis.com/

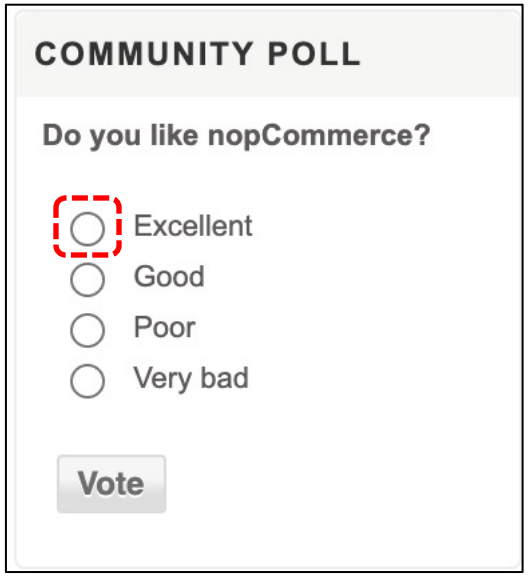
xpath of radio button corresponding to rating “**Good**”, “**Excellent**”, “**Poor**”, “**Very bad**” in demowebshop

```
//label[text()='Good']/..//input[@type='radio']
```

```
//label[text()=Excellent']/..//input[@type='radio']
```

```
//label[text()=Poor']/..//input[@type='radio']
```

```
//label[text()=Very bad']/..//input[@type='radio']
```



XPATH

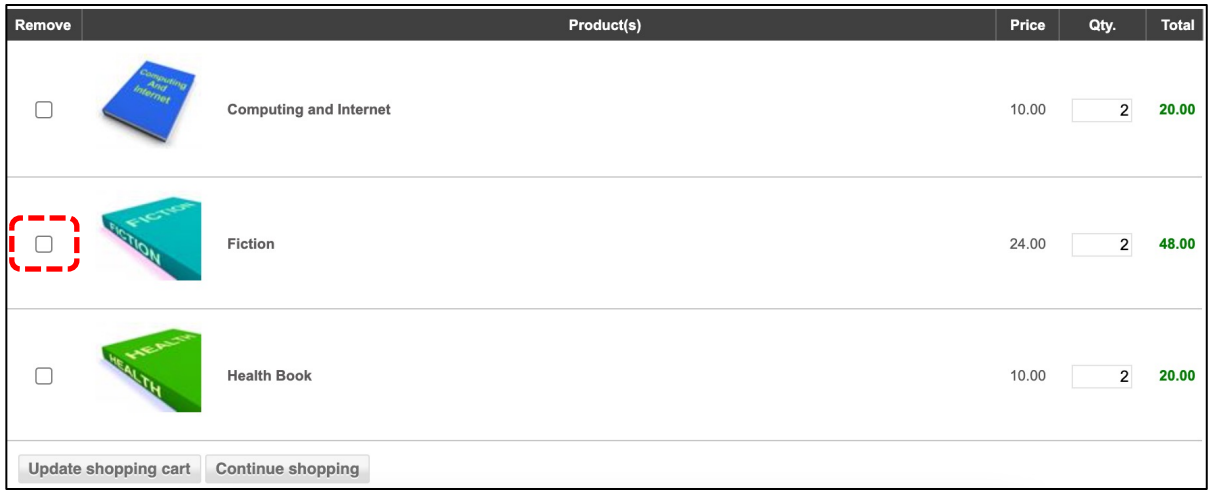
dependent–independent technique

xpath path of checkbox corresponding to book “**Fiction**” in demowebshop

```
//a[text()='Fiction']/../..//input[@type='checkbox']
```

```
//a[text()='Health Book']/../..//input[@type='checkbox']
```

http://demowebshop.tricentis.com/

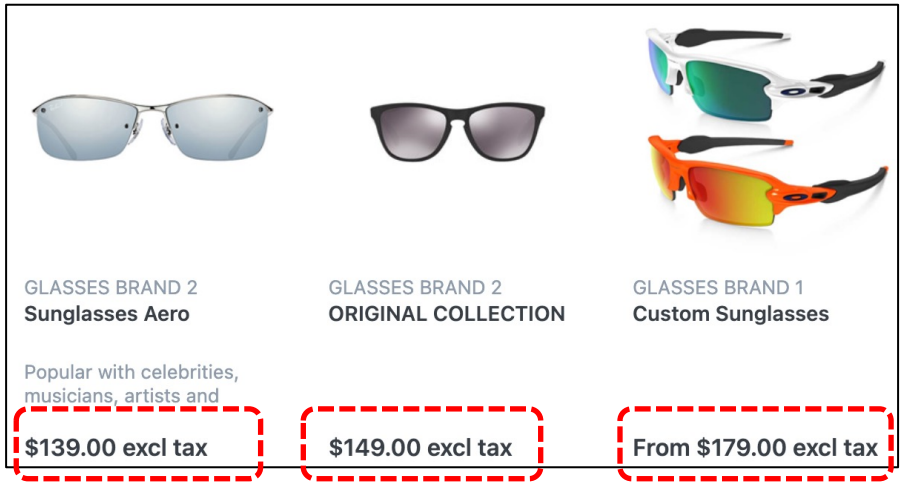


XPATH

dependent-independent technique

https://services.smartbear.com/samples/TestComplete14/smartstore/sunglasses

Get the price tag of all the sun-glasses




```python
from selenium import webdriver
from time import sleep
import re

driver = webdriver.Chrome("./chromedriver")
driver.get("https://services.smartbear.com/samples/TestComplete14/smartstore/sunglasses")

sleep(5)

# Dictionary with sun-glass name and its expected prices
sun_glasses = {'Sunglasses Aero': 139.00, 'ORIGINAL COLLECTION': 149.00, 'Custom Sunglasses': 179.00}

for glass, expected_price in sun_glasses.items():
    _xpath = f"//span[text()='{glass}']/../../..//span[@class='art-price']"
    price_tag = driver.find_element_by_xpath(_xpath).text
    price = re.findall(r'\d+.\d+', price_tag)
    if float(price[0]) == expected_price:
        print("PASS")
    else:
        print("FAIL")
```

XPATH

dependent-independent technique

Get the price tag of all the new products in smart bear webpage

https://services.smartbear.com/samples/TestComplete14/smartstore/newproducts

```
from selenium import webdriver
from time import sleep
import re

driver = webdriver.Chrome("./chromedriver")
driver.get("https://services.smartbear.com/samples/TestComplete14/smartstore/newproducts")

sleep(5)
prices = driver.find_elements_by_xpath("//span[contains(@class,'art-price')]")

for item in prices:
    price = item.text
    _price = re.findall(r'\d+,?\d*\.\d+', price)[0]
    print(_price)
    sleep(0.3)
```

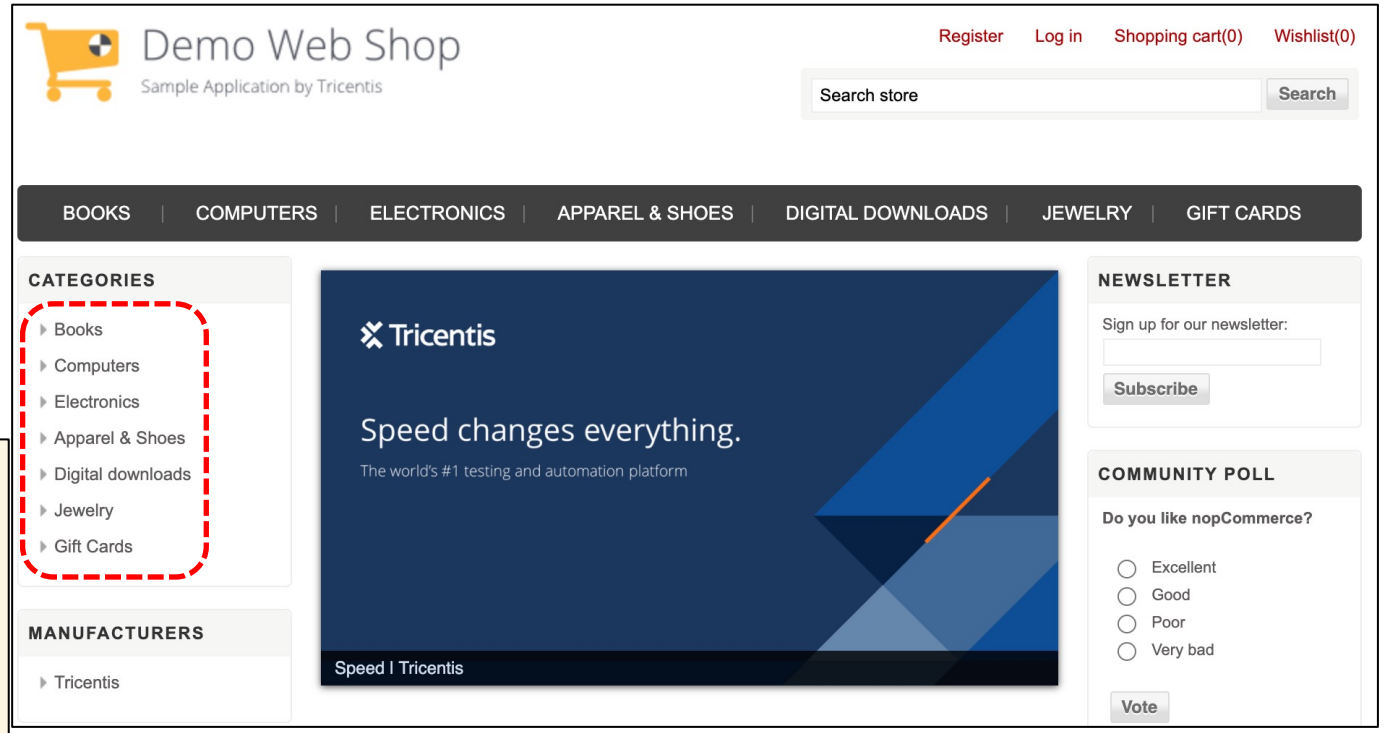
regex to match price of each product

XPATH

parent-child relationship

Get the link text of all links present in left navigation bar in demowebshop webpage

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

# Print link text of all the links present in the left
# navigation bar of demowebshop
_xpath = "//div[@class='block block-category-navigation']//a"
links = driver.find_elements_by_xpath(_xpath)
for link in links:
    print(link.text)
    sleep(1)
```

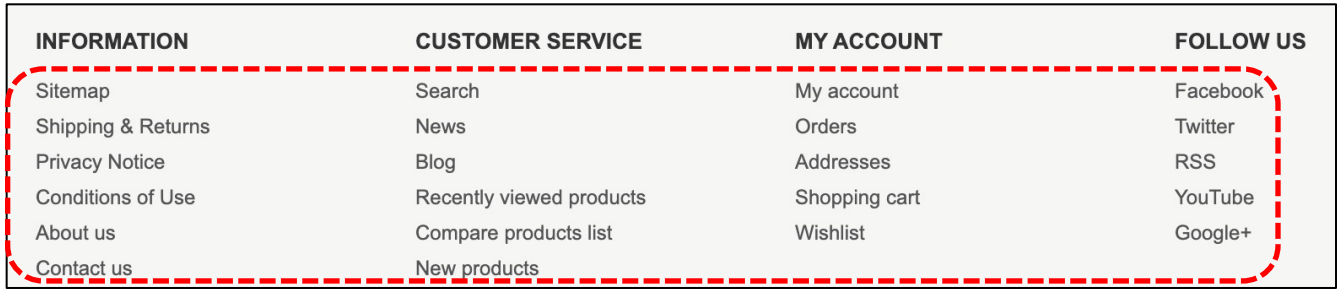


XPATH

parent-child relationship

Get link text of all footer links in demowebshop webpage

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

# Print link text of all the link present in the footer
# of demowebshop.
_xpath = "//div[@class='footer-menu-wrapper']//a"
footer_links = driver.find_elements_by_xpath(_xpath)
for link in footer_links:
    print(link.text)
    sleep(1)
```

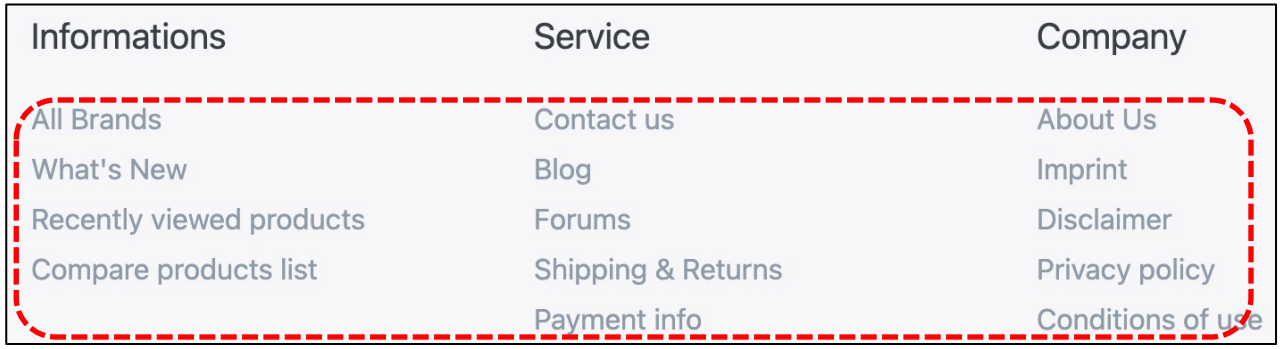



```
sandeep@Sandeeps-MacBook-Pro demo % python3 -i test_note.py
Sitemap
Shipping & Returns
Privacy Notice
Conditions of Use
About us
Contact us
Search
News
Blog
Recently viewed products
Compare products list
New products
My account
Orders
Addresses
Shopping cart
Wishlist
Facebook
Twitter
RSS
YouTube
Google+
>>>
```

XPATH

parent-child relationship

Get link text of all footer links in smart bear webpage




```python
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("https://services.smartbear.com/samples/TestComplete14/smartstore/")
sleep(5)

footer_links = driver.find_elements_by_xpath("//div[@class='container footer-main']//a")

for link in footer_links:
    print(link.text)
    sleep(0.5)
```

```
sandeep@Sandeeps-MacBook-Pro demo % python3 -i test_note.py
All Brands
What's New
Recently viewed products
Compare products list
Contact us
Blog
Forums
Shipping & Returns
Payment info
About Us
Imprint
Disclaimer
Privacy policy
Conditions of use
>>>
```

XPATH

Get link text of all the header links in yahoo.com

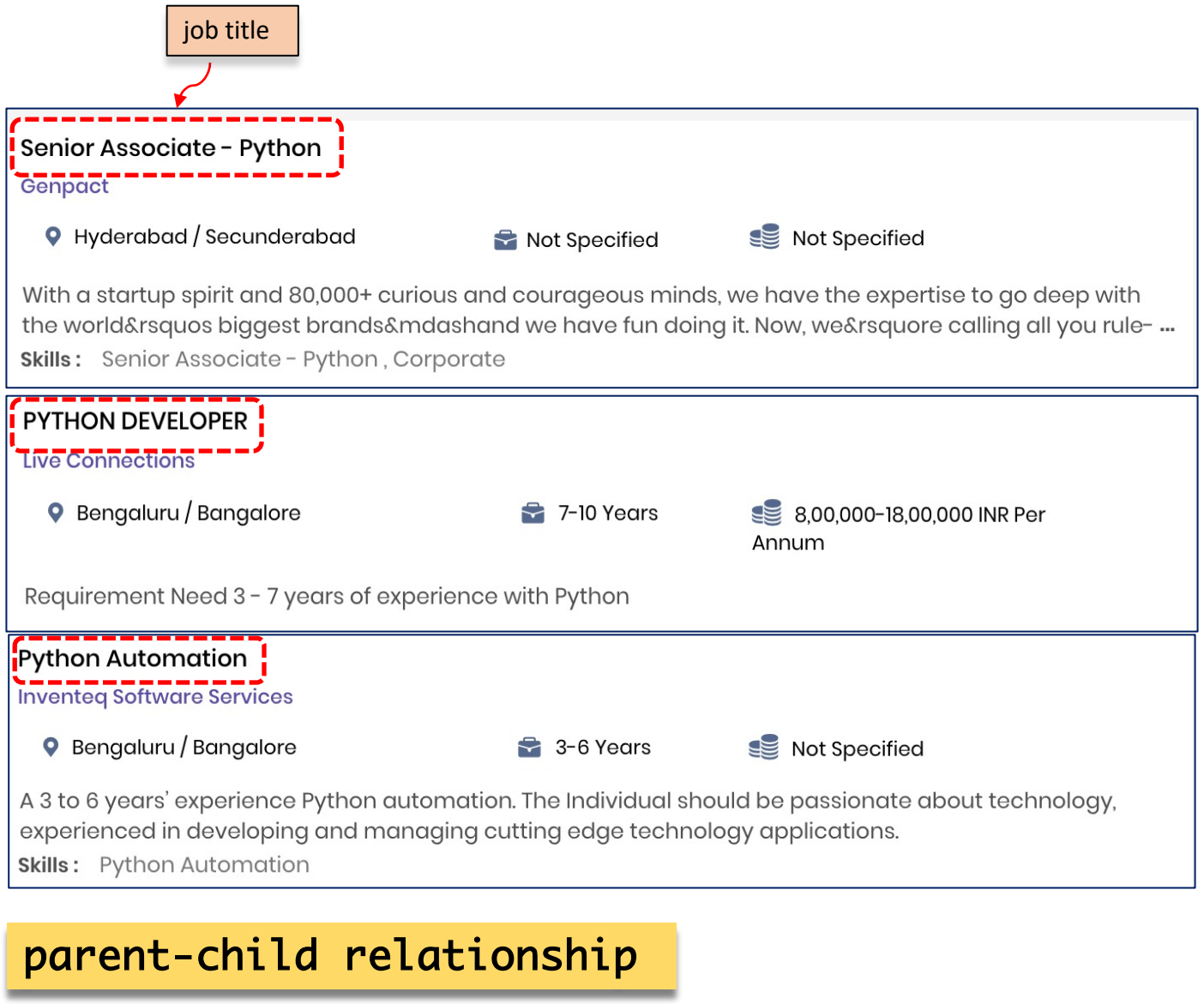
parent-child relationship

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

# Print the link text of all the links present on Yahoo.com headers
_xpath = "//div[@class='ybar-mod-navigation _yb_wkk3w    _yb_cynpj  ']/ul/li/a"
header_links = driver.find_elements_by_xpath(_xpath)

for item in header_links:
    print(item.text)
    sleep(1)
```

XPATH

Get link text of all the job titles in search results of monster.com

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

# Print the jobs titles of all the jobs in
# monster.com search results
_xpath = "//div[@class='job-tittle']/h3/a"
links = driver.find_elements_by_xpath(_xpath)

for link in links:
    print(link.text)
    sleep(1)
```



parent-child relationship

BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

SELECT ELEMENT

SELECT ELEMENT

- 👉 All list box related methods are implemented inside the class “`Select`”.
- 👉 In order to access methods of ”`Select`” class, we need to create an object instance to the ”`Select`” class and pass a “`WebElement`” (list box) as constructor argument.
- 👉 There are 3 different methods available in “`Select`” class to select an item from the list box.
    1. select_item_by_visible_text
    2. select_item_by_index
    3. select_by_value

SELECT ELEMENT

select_by_visible_text

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
select.select_by_visible_text("Audi")
sleep(1)
select.select_by_visible_text("Mercedes")
```

select_by_visible_text Selects an <option> based upon its text

```
<select id="standard_cars"> == $0
  <option value="sel">Select car:</option>
  <option value="aud">Audi</option>
  <option value="bmw">BMW</option>
  <option value="for">Ford</option>
  <option value="hda">Honda</option>
```

visible text

SELECT ELEMENT

select_by_index

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
select.select_by_index(4)
sleep(1)
select.select_by_index(8)
```

- `select_by_index` Selects an <option> based upon the **<select>** element's internal index
- Index starts from 1

SELECT ELEMENT

select_by_value

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
select.select_by_value("jgr")
sleep(1)
select.select_by_value("vol")
```

select_by_value selects an <option> based upon its value attribute

```
<option value="jgr">Jaguar</option>
<option value="lr">Land Rover</option>
<option value="merc">Mercedes</option>
<option value="min">Mini</option>
<option value="nin">Nissan</option>
<option value="toy">Toyota</option>
<option value="vol">Volvo</option> == $0
```

value attribute

SELECT ELEMENT

`first_selected_option`

- Returns the **option (WebElement)** which is currently selected in the list box.
- Since, `first_selected_option` returns a WebElement, we need to call the method `text`, to get the text of the option that is currently selected.

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
select.select_by_visible_text('Mercedes')
currently_selected_option = select.first_selected_option.text
```

`first_selected_option.text` return's the text of currently selected option, "Mercedes"

SELECT ELEMENT

all_selected_option

- Returns the **option (WebElement)** which is currently selected in the list box.
- Since, `all_selected_options` returns a list of Web Elements, we need to call the method `text`, to get the text of the option that is currently selected.

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("multiple_cars")
select = Select(lst_box)
select.select_by_visible_text('Mercedes')
select.select_by_visible_text('Audi')
select.select_by_visible_text('Toyota')

# Returns list of all the options that are currently
# selected in multi-select listbox
all_selected_options = select.all_selected_options

# Prints text of each option
for item in all_selected_options:
    print(item.text)
```

SELECT ELEMENT

options

- Returns the list of all the **options (each option is a WebElement)** present in the list box.
- Each item of the list is an **option** element (WebElement).
- To get the text of all the options, we need to iterate over the list that is returned by **options** method and call the attribute **"text"** on each item of the list.

```
driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
all_options = select.options

for item in all_options:
    print(item.text)
    sleep(1)
```

SELECT ELEMENT

options

Selecting each item in the list box one by one

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
all_options = select.options

for item in all_options:
    select.select_by_visible_text(item.text)
    sleep(1)
```

SELECT ELEMENT

options

Selecting each item in the list box one by one in reversed order

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
all_options = select.options

for item in all_options[::-1]:
    select.select_by_visible_text(item.text)
    sleep(1)
```

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
all_options = select.options

for item in reversed(all_options):
    select.select_by_visible_text(item.text)
    sleep(1)
```

SELECT ELEMENT

options

Print index at which the “Mercedes” is present

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome('./chromedriver')
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("standard_cars")
select = Select(lst_box)
all_options = select.options

# Get the text of each option and build a new list
items = [item.text for item in all_options]

# Check if "Mercedes" is present in the listbox
# If "Mercedes" is present, print the index of it.
for index, item in enumerate(items):
    if item == "Mercedes":
        print(f'{item} is present at index {index}')
```

SELECT ELEMENT

selecting multiple items in multi-select

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome('./chromedriver')
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("multiple_cars")
select = Select(lst_box)

select.select_by_visible_text("Audi")
sleep(1)
select.select_by_visible_text("Mercedes")
sleep(1)
select.select_by_visible_text("Toyota")
```

SELECT ELEMENT

deselect_by_visible_text

deselect_by_index

deselect_by_value

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome('./chromedriver')
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("multiple_cars")
select = Select(lst_box)

# de-select by visible text
select.deselect_by_visible_text("Audi")
sleep(1)
# de-select by index
select.deselect_by_visible_index(8)
sleep(1)

# de-select by value
select.deselect_by_visible_text("merc")
sleep(1)
```

SELECT ELEMENT

selecting all items in multi-select

- Unlike "`deselect_all`" method, there is no direct method in **Select** class to select all the items of multi-select.
- To select all the items of multi-select, get the text of each option and select each item one by one using for loop

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome('./chromedriver')
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("multiple_cars")
select = Select(lst_box)
all_options = select.options

# Get the text of each option and build a new list
items = [item.text for item in all_options]

for item in items:
    select.select_by_visible_text(item)
    sleep(1)
```

SELECT ELEMENT

is_multiple

- 👉 Returns "True" if the select element is multiple select.
- 👉 Returns "False" if the select element is single select.

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

lst_box = driver.find_element_by_id("multiple_cars")
select = Select(lst_box)
# Prints True
print(select.is_multiple)
```

BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

SYNCHRONIZATION

SYNCHRONIZATION

- One of the main challenges for successful browser automation is to match the pace of your script execution with the pace of the application.

- In browser automation, before performing any operation on any web element, the script has to wait for the element to be visible(or/and enabled) on the webpage.

- Selenium provides two different way's to Achieve Synchronization.
    1. Explicit Wait
    2. Implicit Wait

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

`visibility_of_element_located`

- `visibility_of_element_located` checks if the web element is present on the DOM and also visible on the webpage. (Both conditions will be checked)
- "`TimeoutException`" will be raised if either the element is not loaded onto the DOM or the element is not visible on the web page within timeout period.
- The above condition makes sure that the element is present on DOM and also visible on the web page.
- If the element is visible and is disabled due to some reasons, then "`ElementNotInteractableException`" is raised.

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

visibility_of_element_located

Wait for visibility of “FirstName” textbox in demo html page

```
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/loading.html")
# Create an object instance to WebDriverWait class
wait = WebDriverWait(driver, timeout=10)
# Wait until the firstname textbox is visible on the webpage
wait.until(expected_conditions.visibility_of_element_located(("name", "fname")))
# Enter "Hello world" in firstname textbox
driver.find_element_by_name("fname").send_keys("Hello world")
```

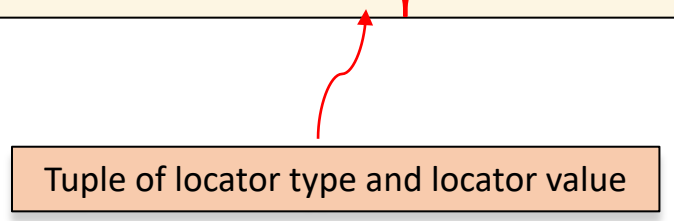
Tuple of locator type and locator value

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

visibility_of_element_located

wait for progress bar to complete 100% in demo html page

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/progressbar.html")
sleep(2)

# Click on 'Click Me' button on the demo-webpage
driver.find_element_by_xpath("//button[text()='Click Me']").click()
wait = WebDriverWait(driver, timeout=10)
# Wait until the progress bar is loaded completely (100%) and visible on the webpage
wait.until(expected_conditions.visibility_of_element_located(("xpath", "//div[text()='100%']")))
print('Done!')
```



SYNCHRONIZATION – WebDriverWait (Explicit Wait)

`until`

- `until` method takes a callable as an argument.
- A **callable** can be either a class which has implemented `__call__` method or a **decorator**.
- `Until` method calls the callable repeatedly with time interval of 0.5 seconds (POLL_FREQUENCY) between the function calls.
- If the callable returns Boolean `True` (or the something which evaluates to Boolean True) within timeout period, the script execution will continue as normal without any exceptions.
- If the callable does not return Boolean `True` (or the something which evaluates to Boolean True) within timeout period, until method raises **TimeoutException**

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

- 👉 one of the short comings of `visibility_of_element_located` is that it only checks if the web element is loaded in the DOM and visible on the webpage, but it does not check if the element is enabled or disabled 😞
- 👉 In-order to include this extra functionality, which is to check for enablement of the web element, extend the existing class `visibility_of_element_located` and customize it to add the extra functionality to check for enablement.
- 👉 Idea here is to re-define `__call__` method of parent class in child class using inheritance.

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

_visibility_of_element_located (customized class)

```
# Create an object instance to WebDriverWait class
wait = WebDriverWait(driver, timeout=10)
# Wait until the firstname textbox is visible on the webpage
wait.until(_visibility_of_element_located(("name", "fname")))
# Enter "Hello world" in firstname textbox
driver.find_element_by_name("fname").send_keys("Hello world")
```

passing object instance of customized class to **until** method

```
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support.expected_conditions import visibility_of_element_located
from selenium.webdriver.remote.webelement import WebElement

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/loading.html")

class _visibility_of_element_located(visibility_of_element_located):
    def __call__(self, driver):
        # Call the parent (visibility_of_element_located) class __call__ method
        # __call__ method of parent class returns a webelement if the element is
        # present in the DOM and also visible on the webpage.
        # If one of the above conditions is not satisfied, then __call__ method
        # returns boolean False
        displayed = super().__call__(driver)
        # Below code checks if the return value of __call__ is of type WebElement
        # If the return type is WebElement, then check if it is enabled.
        if isinstance(displayed, WebElement):
            # "is_enabled" method returns boolean True if the element is enabled.
            # otherwise, "is_enabled" method returns boolean False
            return displayed.is_enabled()
        else:
            return False
```

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

is_visible_enabled (using decorator, alternate solution) 😉

`is_visible_enabled` is a user defined decorator which returns a callable (reference to wrapper function)

```
def is_visible_enabled(locator):
    def wrapper(driver):
        displayed = driver.find_element(*locator).is_displayed()
        enabled = driver.find_element(*locator).is_enabled()
        return displayed and enabled
    return wrapper
```

```
# Create an object instance to WebDriverWait class
wait = WebDriverWait(driver, timeout=10)
# Wait until the firstname textbox is visible on the webpage
wait.until(is_visible_enabled(("name", "fname")))
```

calling decorator

Passing locator type and locator value in tuple

SYNCHRONIZATION – WebDriverWait (Explicit Wait)

invisibility_of_element_located

- `invisibility_of_element_located` takes by locator as an argument. It throws `TimeoutException` if the element is visible even after the timeout period that is specified.
- If the web element does not exist on the DOM, `NoSuchElementException` or `StaleElementReferenceException` is NOT RAISED.

element_to_be_clickable

- element_to_be_clickable takes by locator as an argument. It throws `ElementNotClickableException` if the element to be clicked is either not visible or not enabled or missing from the DOM.
- For more information about explicit wait and conditions, please refer below link

https://www.selenium.dev/documentation/webdriver/waits/

SYNCHRONIZATION – Implicit Wait

`implicitly_wait`

- 👉 WebDriver polls the DOM for a certain duration when trying to any element.
- 👉 An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available.
- 👉 The **default setting is 0,** meaning disabled. Once set, the implicit wait is set for the **life of the driver or browser session**.

BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

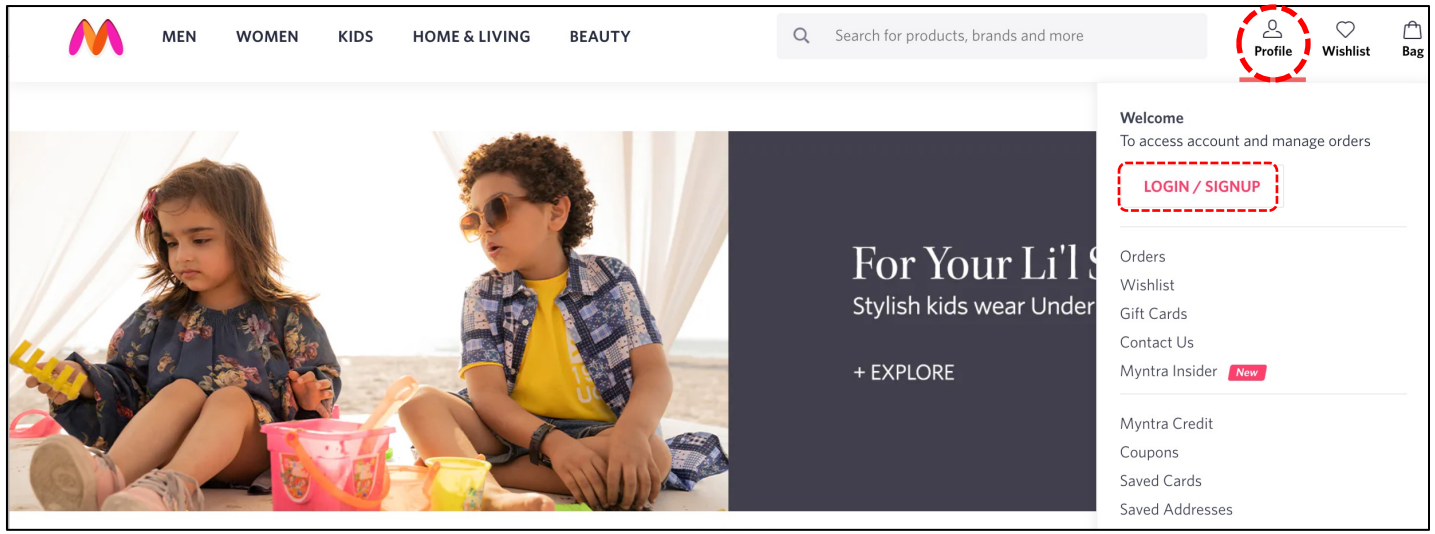
MOUSE ACTIONS

MOUSE ACTIONS

move_to_element

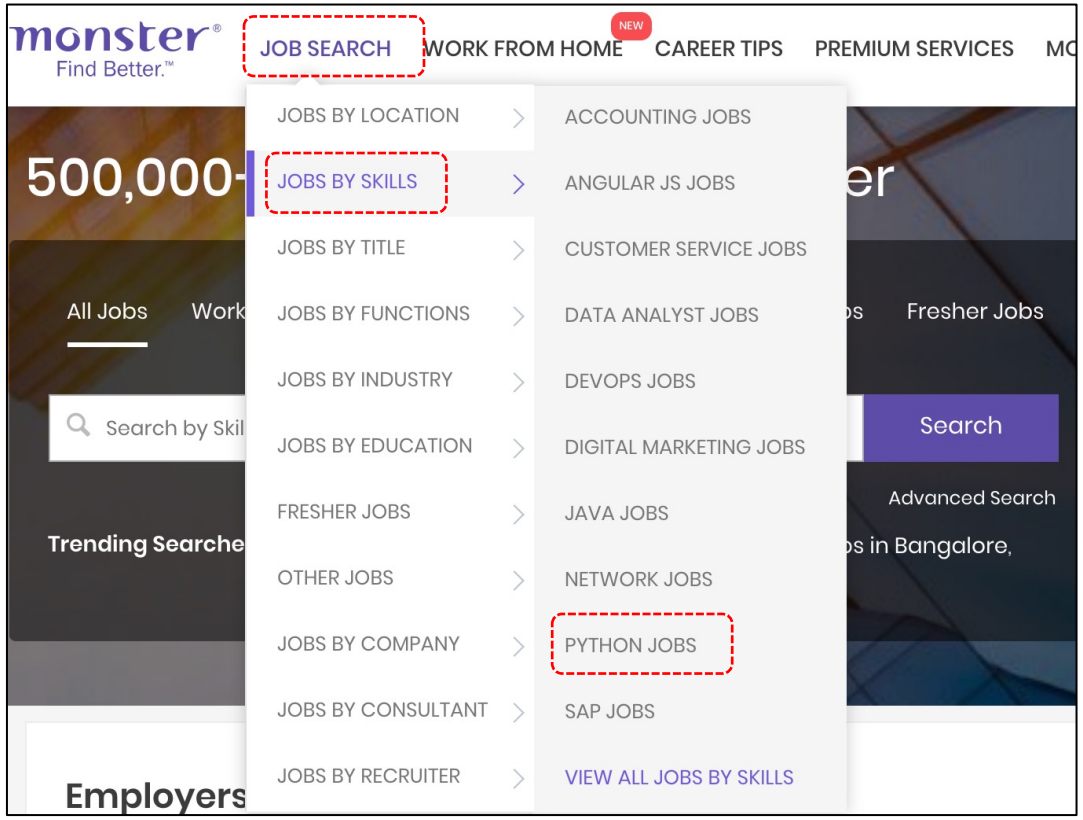
```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Chrome("./chromedriver")
driver.get("https://www.myntra.com/")
sleep(5)

actions = ActionChains(driver)

profile = driver.find_element_by_xpath("//span[text()='Profile']")
# Mouse Hover on "Profile"
actions.move_to_element(profile).perform()
sleep(1)

driver.find_element_by_xpath("//a[text()='login / Signup']").click()
```

passing web element as argument



MOUSE ACTIONS

move_to_element




```python
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Chrome("./chromedriver")
driver.get("https://www.monsterindia.com/")
sleep(5)

actions = ActionChains(driver)

jobs_search = driver.find_element_by_xpath("//a[text()='Job search']")

actions.move_to_element(jobs_search).perform()
sleep(2)

jobs_by_skills = driver.find_element_by_xpath("//a[text()='Jobs by Skills']")

actions.move_to_element(jobs_by_skills).perform()
sleep(2)

python_jobs = driver.find_element_by_xpath("//a[contains(text(), 'Python Jobs')]")
actions.move_to_element(python_jobs).perform()
sleep(2)

python_jobs.click()
```

MOUSE ACTIONS

double_click

- 👉 Find the WebElement on which double click operation to be performed
- 👉 Pass the WebElement to **double_click** function as argument.

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(2)

actions = ActionChains(driver)

click_me = driver.find_element_by_xpath("//button[text()='Double-click me']")

actions.double_click(click_me).perform()
```

MOUSE ACTIONS

send_keys

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.PAGE_DOWN).perform()
```

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.ARROW_DOWN).perform()
```

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.PAGE_UP).perform()
```

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.ARROW_UP).perform()
```

MOUSE ACTIONS

send_keys

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.TAB).perform()
```

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.ENTER).perform()
```

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

actions = ActionChains(driver)
actions.send_keys(Keys.ESCAPE).perform()
```

MOUSE ACTIONS

context_click

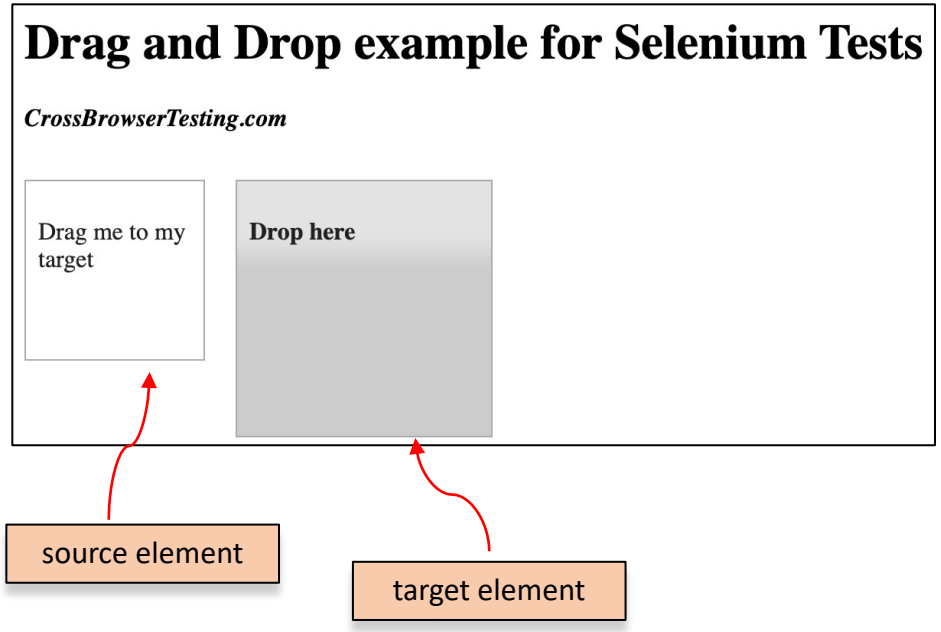
```python
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(5)

# Store 'register' link web element
register_link = driver.find_element_by_xpath("//a[text()='Register']")

# Perform context-click action on the element
actions = ActionChains(driver)
actions.context_click(register_link).perform()
```

context menu

MOUSE ACTIONS

drag_and_drop

```
from selenium import webdriver
from time import sleep
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

driver = webdriver.Chrome("./chromedriver")
# driver.get("http://demowebshop.tricentis.com/")
driver.get("https://crossbrowsertesting.github.io/drag-and-drop")
sleep(2)

# Store 'box A' as source element
source_element = driver.find_element_by_id("draggable")

# Store 'box B' as source element
target_element  = driver.find_element_by_id("droppable")

# Performs drag and drop action of source_element onto the target_element
actions = ActionChains(driver)
actions.drag_and_drop(source_element, target_element).perform()
```



BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

MULTIPLE WINDOWS

HANDLING MULTIPLE WINDOWS

`window_handles`

- If your site opens a new tab or window, Selenium will let you work with it using a window handle.
- Each window has a unique identifier which remains persistent in a single session. (alpha Numeric ID)
- You can get the window handles of all the windows using `window_handles`.
- window handle at **0th index** of the list will always be the window handle of **parent window**.
- When a new window/tab opens, the driver control will be present on parent window. To work with new window, you will need to switch to it.
- When you are finished with a window or tab opened in your browser, you should close it and switch back to the window you were using previously (parent window).
- Forgetting to switch back to another window handle after closing a window will leave WebDriver executing on the now closed page, and will trigger a `NoSuchWindow` Exception. You must switch back to a valid window handle in order to continue execution.

HANDLING MULTIPLE WINDOWS

- launch demowebshop.
- click “Twitter” link at the footer of the webpage.
- ”twitter” webpage opens in a new tab.
- get window handles and switch to twitter tab and enter some text in search field
- switch back to parent window and click on “Register” link

```python
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
# Click on "Twitter" link present on the footer of the webpage
driver.find_element_by_xpath("//a[text()='Twitter']").click()

# "Twitter" page opens in a new tab in the browser
# In-Order to switch the tab, get window handles
# window_handles retunrs a list of window ID's
handles = driver.window_handles

# switch to window handle of "Twitter"
driver.switch_to.window(handles[1])
sleep(2)

# Enter text in search text box on "Twitter" page
driver.find_element_by_xpath("//input[@placeholder='Search Twitter']").send_keys("hello")

# Switch back to Parent window
driver.switch_to.window(handles[0])
sleep(2)

# Click on "Register" link present on Demowebshop page
driver.find_element_by_xpath("//a[text()='Register']").click()
```

HANDLING MULTIPLE WINDOWS

- launch monster.com.
- search “python” jobs.
- click on first job title in search results
- click “APPLY” button

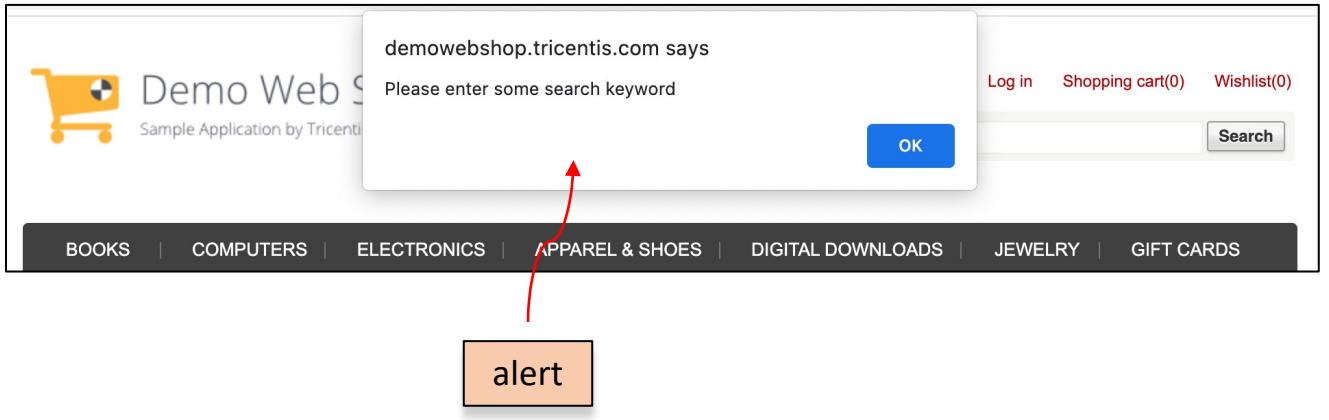
```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
# Launch "monster.com"
driver.get("https://www.monsterindia.com/")
sleep(3)

driver.find_element_by_id("SE_home_autocomplete").send_keys("python")
sleep(2)
driver.find_element_by_xpath("(//strong[text()='Python'])[1]").click()
sleep(1)
driver.find_element_by_xpath("//input[@value='Search']").click()
sleep(5)
driver.find_element_by_xpath("(//div[@class='job-tittle']/h3/a)[1]").click()
sleep(4)
# Get the window handles
handles = driver.window_handles

# Switch to Child Browser
driver.switch_to.window(handles[1])

# Click in APPLY Button
driver.find_element_by_xpath("(//button[text()='APPLY'])[1]").click()
```

HANDLING MULTIPLE WINDOWS

JavaScript Alert

- An Alert shows a custom message, and a single button which dismisses the alert, labelled in most browsers as OK.
- WebDriver can get the text from the popup and accept or dismiss these alerts.

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("http://demowebshop.tricentis.com/")
sleep(3)
# Click on Search Button without entering any Search Keyword
driver.find_element_by_xpath("//input[@value='Search']").click()
sleep(1)
# Get the text of the JavaScript Alert
print(driver.switch_to.alert.text)
# Click's on OK
driver.switch_to.alert.accept()
```

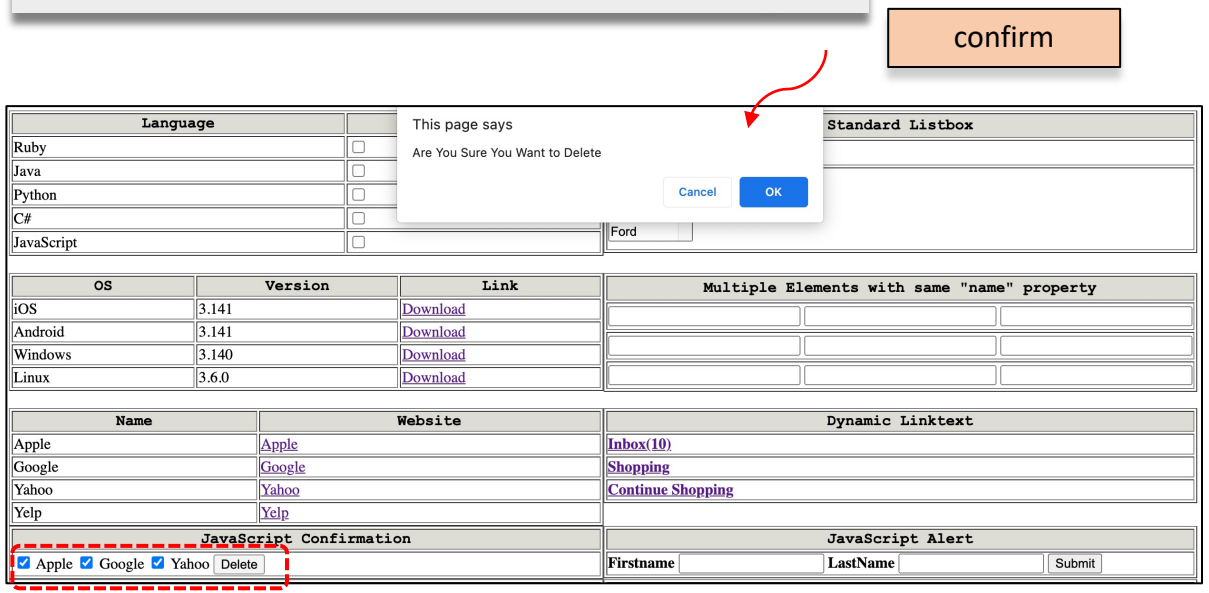


HANDLING MULTIPLE WINDOWS

JavaScript Confirm

👉 A confirm box is similar to an alert, except the user can also choose to cancel the message.

confirm



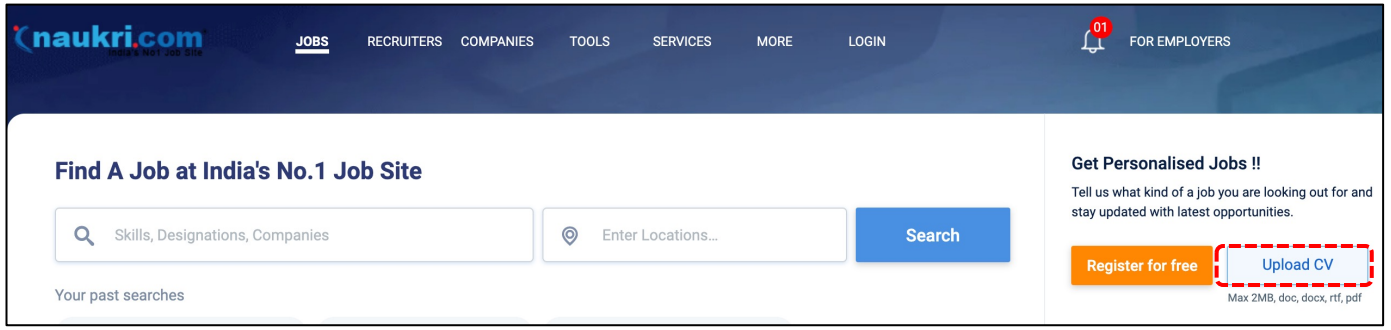

```python
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("file:///Users/sandeep/Desktop/demo-html/demo.html")
sleep(3)
# Get all the checkboxes
elements = driver.find_elements_by_xpath("//input[@class='alert']")
# Check all the checkboxes
for item in elements:
    item.click()
    sleep(1)
# Click on "delete" button
driver.find_element_by_id("delete").click()
sleep(1)
# Switch to the alert and Accept the alert
driver.switch_to.alert.accept()
sleep(1)
# Click on "delete" button
driver.find_element_by_id("delete").click()
sleep(1)
# Switch to the alert and Accept the alert
driver.switch_to.alert.dismiss()
```

HANDLING MULTIPLE WINDOWS

file upload

- As soon the "Upload CV" button is clicked, file explorer window/popup will be opened, asking us to browse the CV to upload.
- we can handle the file browse/explorer popup using third party library like **PyAutoIT**.
- Instead of using the third party library to handle this popup, we can directly use "**send_keys**" method on the control which has "**file**" as value of the attribute "**type**"

```
from selenium import webdriver
from time import sleep

driver = webdriver.Chrome("./chromedriver")
driver.get("https://www.naukri.com/")
sleep(5)

driver.find_element_by_xpath("//input[@type='file']").send_keys("/Users/sandeep/Desktop/Test.docx")
```




```
▼<div class="wdgt-upload-btn">
   <label id="wdgt-file-upload" for="file_upload" title="Upload your CV to Re
   gister" class="btn" data-ga-click="ParserFlow|UploadCVClick" data-uba-
   click="UploadCVClick"> Upload CV </label>
   <input type="file" id="file_upload"> == $0
 </div>
```

file upload control

HANDLING MULTIPLE WINDOWS

file download

👉 File download popup can be handled either by using 3rd party library called PyAutoIT or programmatically (through `ChromeOptions` and `FireFoxProfile` class).

```
from selenium import webdriver
from time import sleep

opts = webdriver.ChromeOptions()
opts.add_experimental_option("prefs",{"download.default_directory": r"/users/sandeep/documents",
                                      "safebrowsing.enabled": True})
driver = webdriver.Chrome('./chromedriver', options=opts)
driver.get("https://www.whatsapp.com/download/")
sleep(5)
driver.find_element_by_xpath("//a[text()='Download for Mac OS X']").click()
```

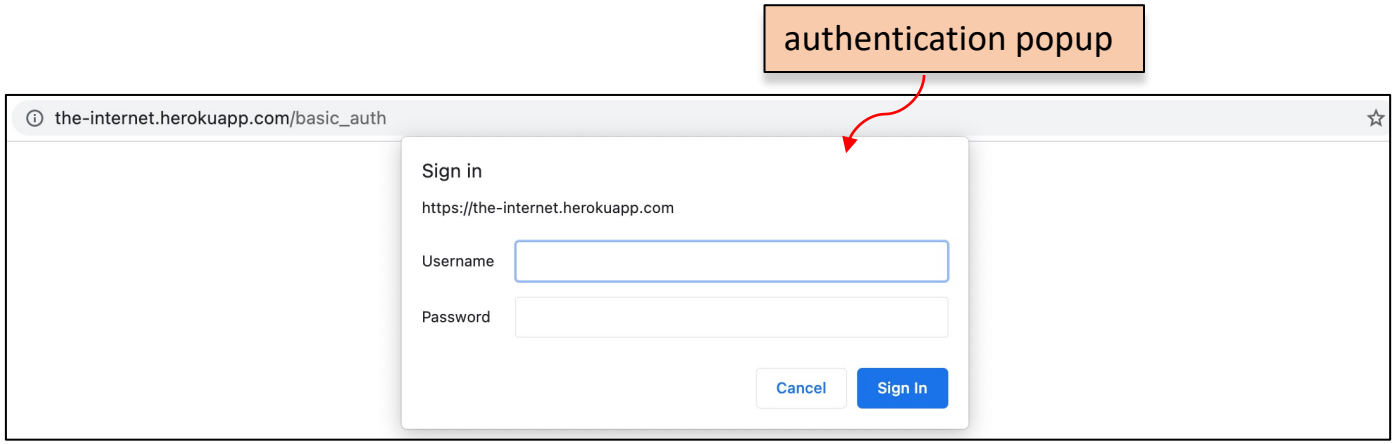
```
from selenium import webdriver
from time import sleep

profile = webdriver.FirefoxProfile()
profile.set_preference("browser.download.folderList", 2)
profile.set_preference("browser.download.dir", r'/users/sandeep/documents')
profile.set_preference("browser.helperApps.neverAsk.saveToDisk", "application/octet-stream")
driver = webdriver.Firefox(profile)
driver.get("https://www.whatsapp.com/download/")
sleep(5)
driver.find_element_by_xpath("//a[text()='Download for Mac OS X']").click()
```

HANDLING MULTIPLE WINDOWS

Authentication

- 👉 when we launch the url which asks for authentication, authentication popup will be displayed, asking user to authenticate.
- 👉 We can programmatically avoid this popup by directly passing the username and password in the url itself as shown in the code.

```
from selenium import webdriver

driver = webdriver.Chrome("./chromedriver")
driver.get("https://the-internet.herokuapp.com/basic_auth")
```




```
from selenium import webdriver

driver = webdriver.Chrome("./chromedriver")
driver.get("https://admin:admin@the-internet.herokuapp.com/basic_auth")
```

username password

BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

iFRAMES

iFRAMES

iframes

- An iFrame (Inline Frame) is an HTML document embedded inside the current HTML document or a website
- iFrame is defined by an **<iFrame></iFrame>** tag in HTML. With this tag you can identify an iFrame while inspecting the HTML tree
- Webdriver can't perform an action on web element automatically when object or web element are inside the frame
- In order to work with frame web elements we should pass driver control to the frame before performing an action

iFRAMES

HTML Code

```
<div id="modal">
  <iframe id="buttonframe" name="myframe"  src="https://seleniumhq.github.io">
    <button>Click here</button>
  </iframe>
</div>
```

switching frame using name/id

```
# Switch frame by id
driver.switch_to.frame('buttonframe')

# Now, Click on the button
driver.find_element_xpath("//button[text()='Click here']")
```

switching frame by index

```
# switching to second iframe based on index
iframe = driver.find_elements_by_tag_name('iframe')[1]

# switch to selected iframe
driver.switch_to.frame(iframe)
```

switching frame using webelement

```
# Store iframe web element
iframe = driver.find_element_xpath("//iframe[@id='buttonframe']")

# switch to selected iframe
driver.switch_to.frame(iframe)

# Now click on button
driver.find_element_xpath("//buttpn[text()='Click here']")
```

coming out of frame

```
# switch back to default content
driver.switch_to.default_content()
```

BROWSER AUTOMATION USING PYTHON-SELENIUM

SANDEEP SURYAPRASAD

PYTEST

What is Unit Testing 🤔

A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property.

What do unit tests look like 🤔

A unit can be almost anything you want it to be -- a line of code, a method, or a class. Generally though, smaller is better.

Smaller tests give you a much more granular view of how your code is performing.

There is also the practical aspect that when you test very small units, your tests can be run fast; like a thousand tests in a second fast.

Who Should Create The Unit Tests 🤔

The programmer who wrote the code will likely know how to access the parts that can be tested easily and how to mock objects that can't be accessed otherwise.

PYTEST

What is pytest 🤔

The pytest framework makes it easy to write small tests using Python.

Advantages of pytest

- 👉 Very easy to start with because of its simple and easy syntax.
- 👉 Can run tests in parallel.
- 👉 Can run a specific test or a subset of tests.
- 👉 Dependency test.
- 👉 Grouping of test methods.
- 👉 Generates report.

PYTEST

install pytest

```
pip install pytest
```

check the version of pytest

```
pytest --version
```

test discovery

- 👉 The module name should either start with **test_*** or ***_test**.
- 👉 All the classes inside the module should start from **Test*** (without an `init` method)
- 👉 All the test methods should start from **test_***.

PYTEST

pytest fixtures

- Pytest fixture is a callable (normally a function or a generator) decorated with inbuilt pytest decorator @fixture
- Fixtures are used for **dependency injection** or **to pass the data to the test functions**
- Fixtures are accessed by test functions by passing the name of the fixture to test functions as argument.
- Fixtures are used to run a piece of code repeatedly before and/or after every test method/class/module/session based on the defined scope.

PYTEST

Advantages of fixtures

- 👉 Each test can be run independently irrespective of previous test method is failed or passed
- 👉 Setup and teardown methods run irrespective of test's fail or pass
- 👉 Many tests can share the same setup/teardown
- 👉 **Setup/Teardown** in same function
- 👉 Pass data to test with **return** or **yield** statement

PYTEST

Passing fixture that returns a string "hello world"

```
from pytest import fixture
@fixture
def greet():
    return "hello world"

# Passing fixture to test method
def test_greet(greet):
    assert "hello world" == greet
```

passing fixture as an argument to test function

Passing fixture that returns driver instance to the test method

```
from pytest import fixture
from selenium import webdriver

@fixture
def _driver():
  driver = webdriver.Chrome("chromedriver")
  driver.get("http://google.com")
  return driver
```

passing fixture as an argument to test function

```
def test_login(_driver):
    _driver.find_element_by_xpath("//a[text()='Log in']").click()
    _driver.find_element_by_id("Email").send_keys("john.doe@company.com")
    _driver.find_element_by_id("Password").send_keys("Password123")
    _driver.quit()
```

PYTEST

setup and **teardown** method using fixtures

👉 statements **before yield** keyword run's **once before every test function** and statements **after yield** keyword run's **once after every test function**. Thus fixture acting as setup and tear down method.

```
from selenium import webdriver
from pytest import fixture

@fixture
def _driver():
    print('Launcing Browser')
    driver = webdriver.Chrome('chromedriver')
    driver.get("http://demowebshop.tricentis.com/")
    yield driver    # passing driver instance to test method.
    print('Closing Browser')
    driver.quit()

def test_login(_driver):
    _driver.find_element_by_xpath("//a[text()='Log in']").click()
    _driver.find_element_by_id("Email").send_keys("john.doe@company.com")
    _driver.find_element_by_id("Password").send_keys("Password123")
```

PYTEST

conftest.py (sharing fixtures)

- 👉 Fixtures can be shared or re-used in different test methods and across multiple files through a special python file "**conftest.py**"
- 👉 The advantage of having the fixture in "**conftest.py**" is that you don't have to import the fixture you want to use in each and every test. The fixture present in "**conftest.py**" automatically get discovered by pytest.
- 👉 Both **conftest.py** and the test module should be in the same package! If **conftest.py** is in other package than the test module, then **conftest.py** module will not be automatically discovered.
- 👉 Pytest checks if the **conftest.py** file is present in the **current package**. If it is not present, if checks at the **project level**. If there is a **conftest.py** file at project level, the fixture is automatically discovered.
- 👉 The discovery of fixture functions starts at **test classes**, then **test modules**, then **conftest.py** files.

PYTEST

scoping of fixtures

- "**function**" Called once per test function (default)
- "**module**" Called once per module
- "**class**" Called once per class
- "**session**" Called once per-run

```python
from pytest import fixture

@fixture(scope="session")
def fix_session():
    print('\n running setup SESSION scope')
    yield
    print('\n running teardown SESSION scope')

@fixture(scope="module")
def fix_mod():
    print('\n running setup MODULE scope')
    yield
    print('\n running teardown MODULE scope')

@fixture(scope="class")
def fix_class():
    print('\n running setup CLASS scope')
    yield
    print('\n running teardown CLASS scope')

@fixture()
def fix_func():
    print('\n running setup FUNCTION scope')
    yield
    print('\n running teardown FUNCTION scope')
```

PYTEST

test dependency

- In order make one test method to depend on the test result of another test method, we need to install a plugin **pytest-dependency**
- It allows to mark some tests as dependent from other tests. These tests will then be skipped if any of the dependencies did fail or has been skipped.

```python
from pytest import mark

@mark.dependency()
def test_login():
    print('logging in')
    assert False

@mark.dependency(depends=["test_login"])
def test_logout():
    print('logging out')
```

- Both the tests are decorated with mark.dependency()
- This will cause the test results to be registered internally and thus other tests may depend on them.

```
test_note.py::test_login logging in
FAILED
test_note.py::test_logout SKIPPED

================================================ FAILURES ================================================
_____________________________________________ test_login _____________________________________________

    @mark.dependency()
    def test_login():
        print('logging in')
>       assert False
E       assert False

test_note.py:8: AssertionError
========================================== short test summary info ==========================================
FAILED test_note.py::test_login - assert False
====================================== 1 failed, 1 skipped in 0.24s ======================================
```

since, test_login function is failed, pytest has skipped test_logout function

PYTEST

grouping tests

executes on those test functions marked as ”smoke”

```
sandeep@Sandeeps-MacBook-Pro demo % pytest -vs test_note.py -m smoke
============================================================ test session starts ============================================================
platform darwin -- Python 3.8.3, pytest-6.1.0, py-1.9.0, pluggy-0.13.1 -- /Library/Frameworks/Python.framework/Versions/3.8/bin/pyth
on3.8
cachedir: .pytest_cache
metadata: {'Python': '3.8.3', 'Platform': 'macOS-10.16-x86_64-i386-64bit', 'Packages': {'pytest': '6.1.0', 'py': '1.9.0', 'pluggy':
'0.13.1'}, 'Plugins': {'metadata': '1.10.0', 'html': '3.1.1', 'xdist': '2.1.0', 'ordering': '0.6', 'dependency': '0.5.1', 'forked':
'1.3.0'}}
rootdir: /Users/sandeep/Desktop/selenium_training/demo
plugins: metadata-1.10.0, html-3.1.1, xdist-2.1.0, ordering-0.6, dependency-0.5.1, forked-1.3.0
collected 4 items / 2 deselected / 2 selected

test_note.py::test_valiate_login executing login test
PASSED
test_note.py::test_registration executing registration test
PASSED
```

executes on those test functions marked as ”regression”

```
sandeep@Sandeeps-MacBook-Pro demo % pytest -vs test_note.py -m regression
============================================================ test session starts ============================================================
platform darwin -- Python 3.8.3, pytest-6.1.0, py-1.9.0, pluggy-0.13.1 -- /Library/Frameworks/Python.framework/Versions/3.8/bin/pyth
on3.8
cachedir: .pytest_cache
metadata: {'Python': '3.8.3', 'Platform': 'macOS-10.16-x86_64-i386-64bit', 'Packages': {'pytest': '6.1.0', 'py': '1.9.0', 'pluggy':
'0.13.1'}, 'Plugins': {'metadata': '1.10.0', 'html': '3.1.1', 'xdist': '2.1.0', 'ordering': '0.6', 'dependency': '0.5.1', 'forked':
'1.3.0'}}
rootdir: /Users/sandeep/Desktop/selenium_training/demo
plugins: metadata-1.10.0, html-3.1.1, xdist-2.1.0, ordering-0.6, dependency-0.5.1, forked-1.3.0
collected 4 items / 2 deselected / 2 selected

test_note.py::test_shopping executing shopping test
PASSED
test_note.py::test_payment executing payment test
PASSED
```

```python
from pytest import mark

@mark.smoke
def test_valiate_login():
    print('executing login test')

@mark.regression
def test_shopping():
    print('executing shopping test')

@mark.regression
def test_payment():
    print('executing payment test')

@mark.smoke
def test_registration():
    print('executing registration test')
```

- 👉 You can group the tests in using ”**mark**” decorator.
- 👉 You can provide any meaning full group name.
- 👉 In the above example, we have two groups, “**smoke**” and “**regression**”

PYTEST

ignoring scripts/tests

```
tests/
|-- example
|   |-- test_example_01.py
|   |-- test_example_02.py
|   '-- test_example_03.py
|-- foobar
|   |-- test_foobar_01.py
|   |-- test_foobar_02.py
|   '-- test_foobar_03.py
'-- hello
    '-- world
        |-- test_world_01.py
        |-- test_world_02.py
        '-- test_world_03.py
```

folder structure

```
pytest --ignore=tests/hello/ # Skips all the scripts inside folder "hello"
pytest --ignore=tests/foobar/ --ignore=tests/hello/   # Skips all the scripts inside folder "foobar" and "hello"
pytest --ignore=test_example_01.py  # skips all the tests in module test_exmaple_01.py
```

running only failed scripts

```
pytest --last-failed --last-failed-no-failures none
```

PYTEST

generating reports

- To Generate test results in html format, we need to install a plugin pytest-html
- While running the pytest from terminal, include the following in command line arguments

```
pip install pytest-html
```

command to generate html report

```
sandeep@Sandeeps-MacBook-Pro demo % pytest test_note.py --html="reports.html"
============================================================ test session starts ============================
============================
platform darwin -- Python 3.8.3, pytest-6.1.0, py-1.9.0, pluggy-0.13.1
rootdir: /Users/sandeep/Desktop/selenium_training/demo
plugins: metadata-1.10.0, html-3.1.1, xdist-2.1.0, ordering-0.6, dependency-0.5.1, forked-1.3.0
collected 4 items


test_note.py ..FF
```

Show all details / Hide all details

Result	Test	Duration	Links
Failed *(hide details)*	test_note.py::test_shopping	0.00	

```
def test_shopping():
        print('executing shopping test')
        # Making test to fail
>       assert True == False
E       assert True == False

test_note.py:17: AssertionError
 -----------------------------Captured stdout call-----------------------------
executing shopping test
```

Failed *(hide details)*	test_note.py::test_payment	0.00	

```
def test_payment():
        print('executing payment test')
        # Making test to fail
>       assert True == False
E       assert True == False

test_note.py:22: AssertionError
 -----------------------------Captured stdout call-----------------------------
executing payment test
```

Passed *(hide details)*	test_note.py::test_valiate_login	0.00	

```
 -----------------------------Captured stdout call-----------------------------
executing login test
```

Passed *(hide details)*	test_note.py::test_registration	0.00	

```
 -----------------------------Captured stdout call-----------------------------
executing registration test
```

PYTEST

Official Documentation

- 👉 https://docs.pytest.org/en/stable/contents.html
- 👉 https://pytest-dependency.readthedocs.io/en/stable/usage.html
- 👉 https://github.com/pytest-dev/pytest-html

BROWSER AUTOMATION
USING
PYTHON-SELENIUM

SANDEEP SURYAPRASAD

COMMON EXCEPTIONS

COMMON EXCEPTIONS IN SELENIUM

NoSuchElementException

- 👉 This exception is raised when the element is not found in DOM
- 👉 The exception is raised by find_element method
- 👉 You may need to check the selector that you are using in find_element method to rectify the issue

StaleElementReferenceException

- 👉 Thrown when a reference to an element is now "stale" or Lost
- 👉 The possible cause for this exception is that you are no longer on the same page, or the page may
- 👉 have refreshed since the element was located
- 👉 The element may have been removed and re-added to the web page, since it was located
- 👉 Element may have been inside an iframe or another context which was refreshed

COMMON EXCEPTIONS IN SELENIUM

NoSuchAttributeException

- 👉 Thrown when the attribute of element could not be found
- 👉 An attribute could be anything that you are trying to access after dot operator. It can be a method, property, variable etc.

NoAlertPresentException

- 👉 Thrown when switching to no presented alert
- 👉 This can be caused by calling an operation on the Alert() class when an alert is not yet on the screen

COMMON EXCEPTIONS IN SELENIUM

`ElementNotVisibleException`

- 👉 Thrown when an element is present on the DOM, but it is not visible
- 👉 Most commonly encountered when trying to click or edit or read text of an element that is hidden from view

`ElementNotInteractableException`

- 👉 Thrown when an element is present on the DOM but can not interaction with the element
- 👉 Possible cause may be the element is disabled

COMMON EXCEPTIONS IN SELENIUM

TimeoutException

- 👉 Usually thrown by until method of WebDriverWait class
- 👉 Possible cause would be when the command does not complete with in specified timeout period

NoSuchFrameException

- 👉 Thrown when frame target to be switched doesn't exist