ATTAFI YOUSRA

Certified Selenium Tester

Part 3
Using Selenium WebDriver

Levels of knowledge of learning objectives

1

K1: remember

2

K2: understand

3

K3: apply

4

K4: analyze

Logging and reporting mechanisms

Logs and reports

Logging of automated test execution, done correctly, can greatly help a test analyst toQuickly determine whether the failure was caused by the SUT or not.

Good log management can make thedifference between an automation project that fails and one that succeeds in delivering value.

Logging is a way to trackthe eventsthat occur during execution.

This recording can be done before and after a step and can include the data that was used in the step and the behavior that occurred as a result of the step.The more detailed this record is, the better it will help to understand the result of the test run.

Logs and reports

- Reporting is based on the recording of execution logs, but they are different.

  - **The recording of execution logs** provides information on the execution of automated scripts to the test analyst and to the automation engineers responsible for correcting the tests if necessary.
  - **Reporting** is to provide this execution information as well as other contextual information to the various stakeholders, external and external, who need or want it.

- It is important for test automation engineers to determine who wants or needs test execution reports, and what information is of interest to them.

Reporting: distribution

- There are several ways to manage the distribution of automated test reporting.
- For example, by putting them at the disposition of stakeholders so they can download them whenever they want.
- Or by sending them to stakeholders who wish to receive them as soon as they are ready.
- In both cases, it is necessary seek to automate the creation and distribution of reports to eliminate a manual task to be carried out.

Reporting: content

- Reports should contain:
  - an overview of the tested system
  - an overview of the environment(s) on which the tests were carried out
  - the results obtained during execution.

- Each stakeholder may have a different view of the results and these needs must be met by the automation team.

- Often stakeholders want to see changes in test results, not just their status at a specific point in time.

Test execution layer

The test automation solution must provide a mechanism that implements the **test execution layer**.

One way to implement such a mechanism is to fully code the test automation scripts. Such scripts can then be executed like any other script, for example by **Python**.

Rather than completely creating such scripts,instead we can use existing unit test librariesthat facilitate the execution of tests and the management of execution results.

pytest

- **Pytest** is a framework test automation that makes it easy to write tests for Python, but also for Selenium WebDriver with Python.
- Pytest makes it easy writing simple tests, but also makes it possible to more complex automated tests.
- When called, Pytest executes all tests in the current directory or its subdirectories. It specifically searches for all files that match the patterns:
- **"test__*.py" or :*__test.py"** and then run them.
- Pytest will execute all methods containing the characters **“test**” in their name.

pytest

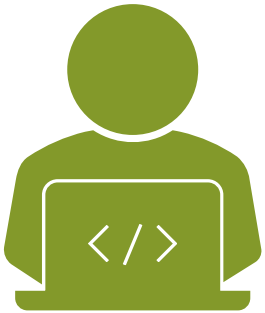
- When run without options, *pytest* runs all tests in the current directory and its subdirectories
- Some possible options:

  pytest -v : Verbose mode that displays full test names.

  pytest -q : Silent mode which displays less information on output.

  pytest --html=report.html : Adds a report in an HTML file.

- Tests can be annotated to be treated in a particular way.
- **@pytest.mark.skip** before a test definition, prevents pytest from running the test.
- **@pytest.mark.xfail** before a test definition, it informs pytest that the test should fail.



Python logs

- The Python logging library has five different levels of messages that can be logged. From lowest to highest level:
  - **DEBUG**: to diagnose problems
  - **INFO**: to inform about the course of the test
  - **WARNING**: something unexpected happened, a potential problem
  - **ERROR**: A major problem has occurred
  - **CRITICAL**: a critical problem has occurred

Assertion in Python

- When executing a test case, it is important to test an actual behavior of the system.

- Each test case must therefore have expected results linked to actions of the SUT that can be verified. Python has a built-in mechanism to check whether the correct data or behavior has occurred: assertion.

- An assertion is a statement that is expected to be true at some point in the script.

- Syntax:

- **assert sumVariable==7, "sumVariable should equal 7."**

Reporting and logs: Selenium WD test case steps

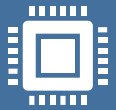
- A step of a Selenium WD test case usually includes the following actions.
  - Locate a displayed web element.
  - Take action on this web element.
  - Make sure the situation went well.

- For the first action (1), if the element is not found, or is found in an incorrect state, an exception is thrown.
- For action two (2), if the attempt to act on the web element fails, an exception is thrown.
- For action three (3), we can use an assertion to verify that the expected behavior has occurred, or that the expected value has been received.

First steps

Start a test automation session

To test a Chrome browser, you must first download the browser driver and install this file on the tester's workstation.



This is usually achieved by editing the Path environment variable so that the operating system finds it when it needs it.

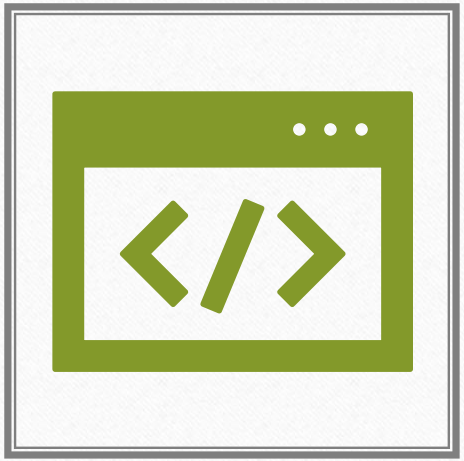
Automation engineers should familiarize themselves with the documentation available on the various support websites for the various browsers, and the Selenium WebDriver support site.

Start a test automation session

- To work with web pages, you must first open a web browser.
- This can be achieved by creating a WebDriver object. Instantiating the WebDriver object creates the programming interface between a script and the web browser. It will also run a WebDriver process if needed for a particular web browser.
- It will also launch the web browser itself.

- Example :

- **from selenium import webdriver**
- **driver = webdriver.Chrome()**



Navigate to a URL

- To navigate to the desired site page, we use the get() function
- Example :
- **driver.get('https://wwww.python.org')**
- After opening a page or navigating to a different page, it is advisable to check if the correct page has been opened.
- The WebDriver object contains two useful attributes for this: **current_url** And **title.**
- Example :
- **assert driver.current_url=='https://wwww.python.org'**
- **assert driver.title == 'Welcome to Python.org', errMsg**

Navigating and Refreshing Pages

- To simulate forward and backward navigation in the web browser, one can use the methods **back()** And **forward()** of the WebDriver object.
- Example :
- **driver. back()**
- **driver. forward()**
- It is also possible to refresh the current page with the web browser. This can be done by calling WebDriver's refresh() method.
- Example :
- **driver. refresh()**

Closing the browser

- At the end of the test, you should close the web browser process and all other driver processes that were running. If you don't close the browser, it will stay open even after the test is finished.

- To close the browser controlled by webdriver, call the method **quit()** of the webdriver object.

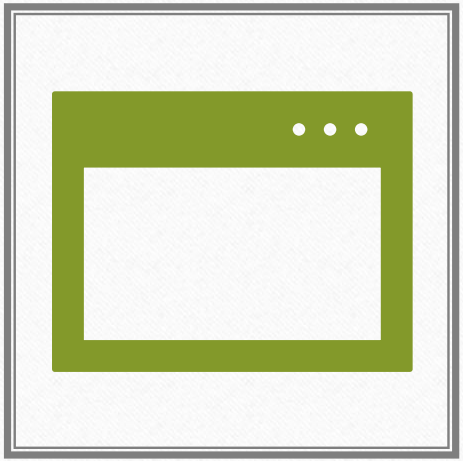
- Example :

- **driver. quit()**

- It is best to place the quit() function in the part of the test script that is executed regardless of the test result.

- Typically, test libraries have their own mechanisms for setting and executing test termination code – called "**tear down**".

Close a browser window/tab



- To close a tab or a window, without stopping the driver

- **driver.close()**

- This method takes no parameters and closes the active tab/window.
- Change the context to the desired tab to be able to close this tab.
- After closing, calling any other WebDriver command other than **driver.switch_to.window()** will then throw an exception **NoSuchWindowException** since the object reference still points to the window that no longer exists.

Maximize / minimize windows

- Selenium allows minimizing and maximizing web browser windows and putting it in full screen mode.

- The Python shortcuts for this function are:

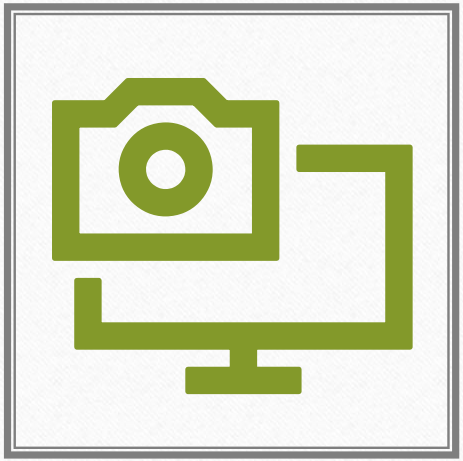
- **driver. maximize_window()**
- **driver.minimize_window()**
- **driver.fullscreen_window()**

Take screenshots of web pages

Take screenshots of web pages

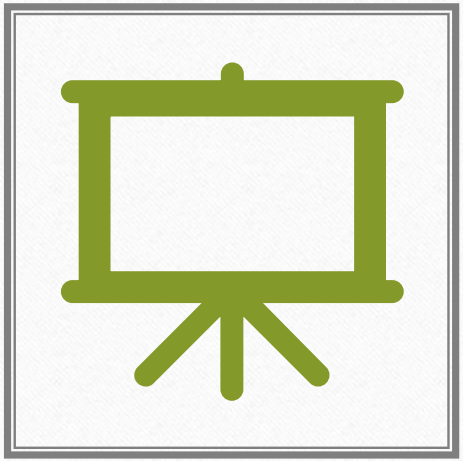
- Selenium cannot reliably verify the layout and appearance of web pages.
- Taking screenshots of a particular page or element on the screen allows them to be checked visually.
- It is better to save them in the logs or in a known location, and manage file naming (unique names and locations) so that the automation script does not overwrite screenshots taken earlier in the execution.

Why take screenshots?



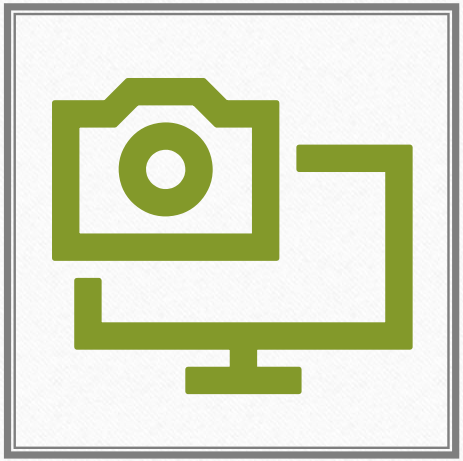
- Screenshots can be useful in the following cases:
  - When a failure has occurred.
  - When the test is very visual.
  - When it comes to critical soft ware for safety or the mission, which may require a test audit.

When to take screenshots?



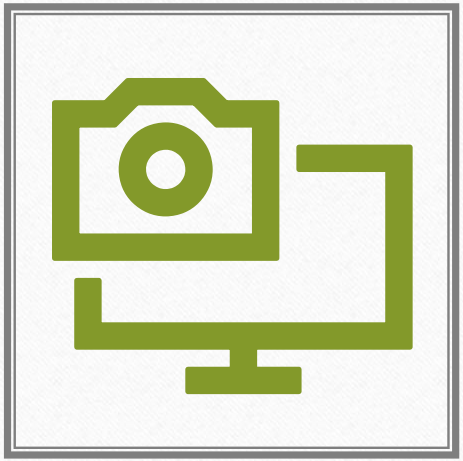
- Typically, the parts of test scripts that take screenshots are placed right after the test steps that control the user interface or in end functions test (step "*tear down*").
- But since these captures are a valuable tool for understanding the results of automated tests, they can be made at any place deemed useful in the script.

Scope of screenshots



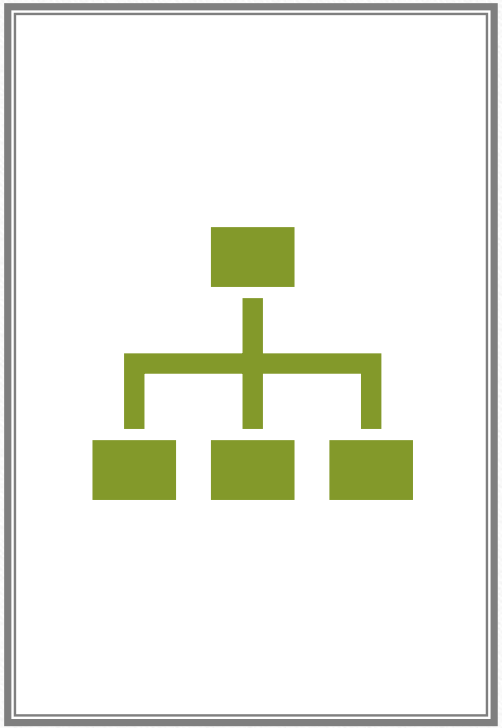
- Screenshots can be taken at two different scopes:
  - full browser page
  - a single element in the browser page.
- Both methods use the same function call but are called from different contexts.
- If the state of the GUI changes rapidly, the screenshot taken may not show the exact state of the page or the expected element.

Take a screenshot



- Screenshot of the whole screen
- **driver.get_screenshot_as_file("c:\\temp\\sc.png")**
- Screenshot of a specific webelement
- **ele = driver.find_element_by_id("btLogin")**
- **ele.screenshot("c:\\temp\\sc_ele.png")**
- Screenshot in base64 (for recording in base, for example)
- **img_b64 = driver.get_screenshot_as_base_64("c:\\temp\\sc.png")**
- **img_b64 = ele.screenshot_as_base_64("c:\\temp\\sc_ele.png")**
- Screenshot in png stored in binary in a variable
- **png_str=driver.get_screenshot_as_png**
- **png_str = ele.screenshot_as_png**

Locate GUI elements

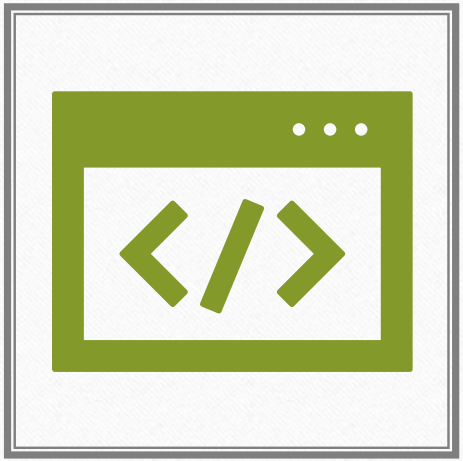


DOM

- According to the W3C, the DOM is defined as follows:

- "*The W3C Document Object Model (DOM) is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update document content, structure, and style. .*"

DOM



•When a webpage is loaded in the browser, the browser creates a DOM, modeling the webpage as a tree of objects. This DOM defines a standard for accessing the web page.

- The DOM defines:
  - All HTML elements as objects.
  - Properties of all HTML elements.
  - The methods which can be used to access all HTML elements.
  - The events Who affect all HTML elements.

Introduction

Most operations with WebDriver require locating user interface elements in the **DOM** of the currently active screen.

To do this, we use the methods **find_element_** Or **find_elements_** of WebDriver.

find_element_, find_elements_

- For searching, the following location methods can be used:
  - **by_id (id_)**
  - **by_class_name (name)**
  - **by_tag_name (name)**
  - **by_xpath (xpath)**
  - **by_css_selector (css_selector)**

- The argument taken by the function is a string representing the element or elements sought.

- The return values are different;
  - **find_element_**returns a single web element
  - **find_elements_**returns a list of web items

Localization by HTML methods: By ID

- The element in the DOM:
- **<element id="unique_id">**
- The code to locate it:
- **element_found=driver.find_element_by_id("unique_id")**

- Benefits:
  - Efficient way to carry out the operation.
  - By definition, each ID must be unique within the HTML document.
  - A tester can easily add IDs to the SUT
- Disadvantages:
  - IDs can be generated automatically, which means they can be changed dynamically.
  - IDs are not appropriate for code that is used in multiple places
  - A tester may not be allowed to modify the SUT code.

Localization by HTML methods: By class

- The element in the DOM:
- **<element class="class_name1">**
- The code to locate it:
- **element_found=driver.find_element_by_class_name("class_name1")**

- Benefits :
  - Class names can be used in multiple places in the DOM, but you can limit the location to the loaded page (eg in a modal popup window).
  - A tester can easily add class names to the SUT

- Disadvantages:
  - Class names can be used in more than one place, so be careful not to locate the wrong element.
  - A tester may not be allowed to modify the SUT code.

Localization by HTML methods: By tag name

- The element in the DOM:
- **<H2>**
- The code to locate it:
- **element_found=driver.find_element_by_tag_name("H3")**

- Advantage :
  - If a tag is unique to a page, you can restrict where to search.
- Inconvenience :
  - If a tag is not unique to a page, you may find the wrong element.

Localization by HTML methods: By link text

- The element in the DOM:
- **Next Page**
- The code to locate it:
- **element_found=driver.find_element_by_link_text("Next Page")**
- **Or**
- **element_found=driver.find_element_by_partial_link_text("Next Pa")**
- Benefits:
  - Easy if the link text is unique to a page,
  - If the link text is visible to the user, it's easy to tell what the test code is looking for.
  - Partial link text is less likely to change than full link text.
- Disadvantages:
  - The link text is subject to change often.
  - Using partial text can make it more difficult to uniquely identify a unique link.

Localization by XPath Methods

- XPath can be used to select specific nodes using different criteria in the DOM, which is an HTML document.
- WebDriver can use XPath to find a specific node, and from there locate an element.
- You can specify a pathabsolute or relative pathfrom a found node that matches a criterion.

- Example of absolute path:
- **element_found=driver.find_element_by_xpath("/html/body/form/input[2]")**
- Example relative path:
- **element_found=driver.find_element_by_xpath("//form2[@id='exact_form']/input[2]")**

- Both will return the second input field in the HTML snippet.

Localization by XPath Methods

- Benefits :
  - You can find elements that don't have unique attributes(id, class, name, etc.).
  - You can use XPath in generic locators, using the different "By" strategies (by id, by class, etc.) if needed.
- Disadvantages:
  - Absolute XPath code is "fragile" and may not work after a small change in the HTML structure.
  - Relative XPath code may find wrong node if the attribute or element you are looking for is not unique on the page.
  - As XPath may be implemented differently across browsers, additional effort may be required to run WebDriver tests in each environment.

Localization by CSS selector

- The code to locate by CSS selector :

  Element = driver.find_element_by_css_selector('p.paragraph')

- Advantage:
  - If an item is unique to a page, you can narrow where you search.
- Inconvenience :
  - If an element is not unique to a page, you may find the wrong element.

Location by predefined conditions

- Selenium/Python integrates predefined conditions through the module. **expected_conditions** which can be imported from **selenium.webdriver.support**
- You can create custom condition classes, but the predefined classes should satisfy most needs.
- These classes offer greater specificity than the locators mentioned above: they don't just determine if an item exists, they also allow check the specific states this element is in.
- For example, the **element_to_be_selected()** function not only determines that the element exists, but it also checks if it is in a selected state.

Location by predefined conditions

•Some examples :

- **alert_is_present**
- **element_selection_state_to_be(element, is_selected)**
- **element_to_be_clickable(locator)**
- **element_to_be_selected(element)**
- **frame_to_be_available_and_switch_to_it(locator)**
- **invisibility_of_element_located(locator)**
- **presence_of_element_located(locator)**
- **text_to_be_present_in_element(locator, text_)**
- **title_is(title)**
- **visibility_of_element_located(locator)**

Get status of GUI elements

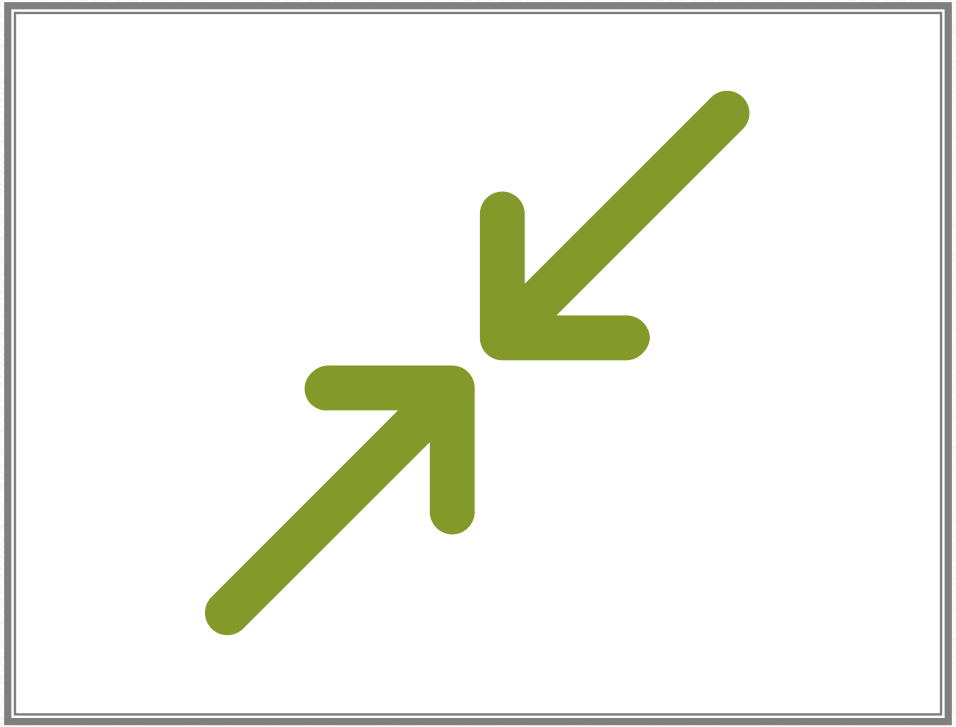
Get status of GUI elements

- There may be several reasons for which we need to access the information of an element.
  - To make sure that the state is as expected at any point in the test.
  - Make sure a control is in such a state that it can be manipulated as needed in the test case
  - Ensure that after the control has been manipulated, it is now in the expected state.
  - Make sure that the expected results are correct after running a test.
- Some properties of a web element can be accessed using a method of the WebElement class.
- Not all web elements have properties.

Common properties and access methods

Property/Method	arguments	Returns	Description
`get_attribute()`	property to recover	property, attribute or None	Gets the property. If no property, get the name attribute. If neither, returns None
`get_property()`	property to recover	property	Gets the property.
`is_displayed()`		boolean	Returns true if the element is visible
`is_enabled()`		boolean	Returns true if the element is enabled
`is_selected()`		boolean	Returns true if the checkbox or radio button is selected
`rental`		Location X,Y	Returns the X,Y location on the render canvas Height, Width
`size`		Height width	Returns the height and width of the element
`tag_name`		tag_name property	Returns the element's tag_name
`text`		item text	Returns the text associated with the element

Interact with UI elements



Interact with UI elements

- When we wish to manipulate a web element, several aspects of the element can have an impact:
  - The existence of the element
  - The visibility of the element
  - Element activation
- Depending on the website, how the HTML code was written, etc. there can be a disparity as to whether or not a web element needs to be displayed/activated before manipulating it.

Manipulating Text Fields

- When typing into an editable text field, you usually want to clear the text of the item first, then type the desired string into the field.

- Example :

- **element.clear()**
- **string_to_type = ‘ABC'**
- **element.send_keys(string_to_type)**

Clicking on web elements

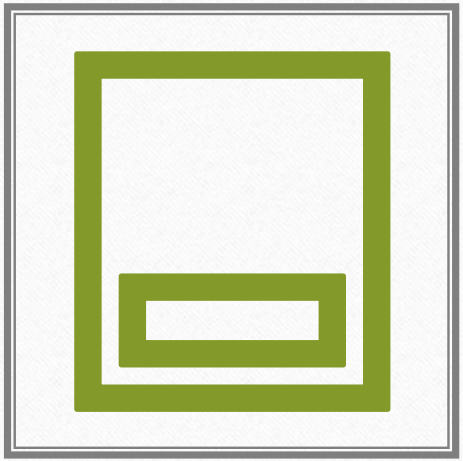
- Clicking on an element simulates a mouse click. This can be done on a radio button, link or image; basically any place you can manually click with your mouse.

- Example :

- **element.click()**

- It is important to verify that the web element is actually clickable. You can use the method **element_to_be_clickable** of style **expected_condition** to wait for it to become clickable.

Manipulation of checkboxes



- It's best to treat checkboxes differently than other clickable controls.
- Indeed each time you click on a checkbox, it toggles the state *selected* from true to false or from false to true.
- Before manipulating them, it is better to know the previous state and the desired state of the control.

```
def set_check_box(element, want_checked):
if want_checked and not element._is_selected():
element.click()
elif element.is_selected() and not want_checked:
element.click()
```

Handling drop-down menus

- Drop-down menus (selection controls) are used by many websites to allow users to select one of the options offered. Some of these drop-down menus allow multiple list items to be selected simultaneously.

- There are many ways to work with a select field's list. These include options for selecting and deselecting single or multiple items.

Handling drop-down menus

- The Selenium API provides developers with a class that allows you to manipulate drop-down menus. it's about the Select class
- from selenium.webdriver.support.select import Select
- select_category = Select(driver.find_element_by_id('searchDropdownBox'))
- select_category.select_by_visible_text('English and foreign books')

- The constructor of the Select class takes an instantiated WebElement as a parameter.
- The constructor will crash if this WebElement does not behave like a select

Manipulating drop-down menus: selection and deselection methods

Property/Method	arguments	Description
`click()`		Search through the HTML code to find a desired element and click on it.
`select_by_value()`	value	Select by value
`select_by_visible_text()`	text	Select all items that display matching text
`select_by_index()`	index	Select an element by its index
`deselect_all(()`		Deselect all items
`deselect_by_index()`	index	Unselect by index
`deselect_by_value()`	value	Unselect by value
`deselect_by_visible_text()`	text	Deselect based on visible text

Manipulating Drop-Down Menus: Selection Control Methods

Property/Method	Description
`all_selected_options`	Returns the list of all selected items
`first_selected_option`	Returns the first selected item
`options`	Returns the list of all options in the list

Working with Modal Dialogs

A modal dialog opens on top of a browser window and does not allow access to the underlying window until it has been processed. These dialog boxes are similar to user prompts, but different enough to discuss separately.

All the code for the modal dialog is in the HTML that made it appear.

Therefore, manipulating the modal dialog is like finding its code in the calling page and manipulating it.

Working with Modal Dialogs: Example

- Suppose we want to click the Proceed to Checkout button in this modal dialog. The first step would be to determine the location of the code for the modal dialog.

- In this example, the ID of the section representing the modal element is layer_cart.

- **modal = driver.find_element_by_id('layer_cart')**

- The next task is to identify the element representing the button in the modal dialog

- **proceed_button = modal.find_element_by_class_name(button_medium)**

- **proceed_button.click()**

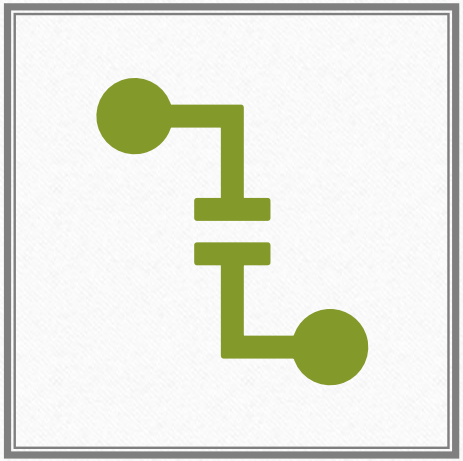
Interact with user prompts

- User prompts are modal windows where the user must interact with commands before they can continue to interact with the browser itself. The alert dialog is often used to make sure the user is aware of some important information.
- They are generally not processed automatically. Therefore, if your script tries to ignore the prompt and continue sending commands to the browser window, an unexpected alert open error is thrown on the next action.

Interact with user prompts

- The W3C defines three different alert-type dialog boxes:
  - Alert
  - confirm
  - Prompt
- Each user prompt is associated with a user prompt message.
- Since alert dialogs are not part of the web page, they require special treatment.
- WebDriver/Python has a set of methods that allow you to control the alert dialog from your automation script. These methods are common to all three user prompt dialog boxes.

Interact with user prompts



- First create a reference using a method of the WebDriver object called **switch_to**.
- **alert = driver. switch_to. alert**
- Get the text in the alert and compare to an expected
- **msg_alert=alert.text**
- **assert "XYZ" = in msg_alert, "the xepxcted text not found"**
- Alert dialogs can be closed in two ways.
- **# press OK**
- **alert.accept()**
- **# press CANCEL, or the closing X**
- **alert.dismiss()**

Timing mechanisms

Timing mechanisms

Selenium WebDriver with Python has several different wait mechanisms that an automation engineer can use when setting up synchronization for their automation.

To usean explicit wait mechanismis generally not the best approach for this, but this method is commonly used by many automation engineers.

A (bad) solution

- The following code should be recognized by almost anyone who has ever automated tests:

- **import time**
- **..**
- **time.sleep(5)**

- While it can be a good solution at times, this is not the case in general. This type of waiting mechanism considers the worst-case response time. As this does not occur frequently, a significant portion of this waiting time is wasted.

Implicit Expectations, Explicit Expectations

- Selenium with WebDriver has two main types of wait mechanisms : implicit expectations and explicit expectations.

Implicit expectation

- An Implicit Wait in WebDriver is defined when the object WebDriver is created for the first time.

- **driver = webdriver.chrome()**
- **driver.implicitly_wait(10)**

- Implicit wait will be in effect until WebDriver is disabled. This mechanism allows a WebDriver to query the DOM for a certain amount of time when it tries to find an element that is not found immediately (default 0 sec).
- The code above tells the WebDriver to watch for any element to appear for ten seconds, polling the DOM every millisecond.
- If the item appears during this time, the script continues to run.

Explicit wait

- Explicit wait times require the automation engineer to define exactly how long WebDriver should wait for a particular element to appear, or a particular state of that element.

- In Python, these wait times are coded using the WebDriver method **WebDriverWait()**, in conjunction with a **ExpectedCondition**. Expected conditions are defined for many common conditions that can occur and can be accessed by including the module **selenium.webdriver.support**as shown below :

- **from selenium.webdriver.support.ui import WebDriverWait**

- **from selenium.webdriver.support import expected_conditions as EC**

- This code provides the automation engineer with the possibility of using the predefined expected waiting times that are available.

Explicit wait: example

- Calling the explicit wait can be done using the following code:

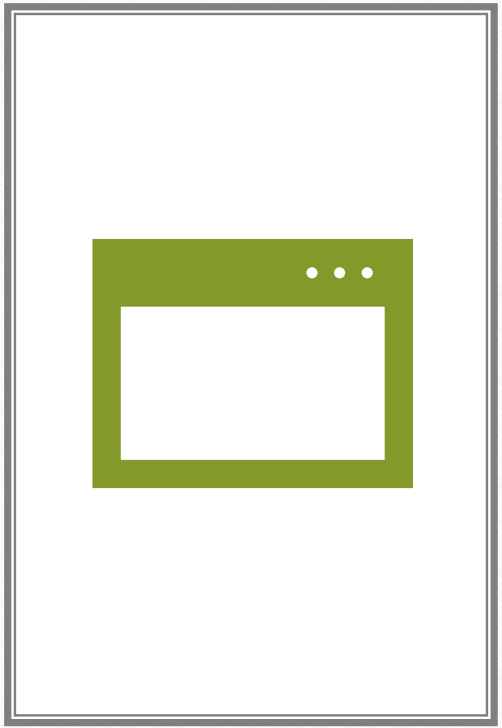
- **from selenium.webdriver.support.ui import WebDriverWait**
- **from selenium.webdriver.support import expected_conditions as EC**
- **from selenium.webdriver.common.by import By**
- **wait = WebDriverWait(driver, 10)**
- **element = wait.until(EC.element_to_be_clickable((By.ID, 'someID')))**

- This code will wait for up to 10 seconds, checking every 500 milliseconds, for an element to be identified by a defined ID '**someID**'. If the web element is not found after 10 seconds, the code will raise a **TimeoutException**. If the web element is found, a reference to it will be placed in the variable 'element', and the script will continue.

Property/Method
title_is
title_contains
presence_of_element_located / presence_of_all_elements_located
visibility_of
text_to_be_present_in_element
text_to_be_present_in_element_value
frame_to_be_available_and_switch_to_it
invisibility_of_element_located
element_to_be_clickable
staleness_of
element_to_be_selected / element_located_to_be_selected
element_selection_state_to_be / element_located_selection_state_to_be
alert_is_present

Explicit expectation: some possible variations of expected_condition

Change browser window context

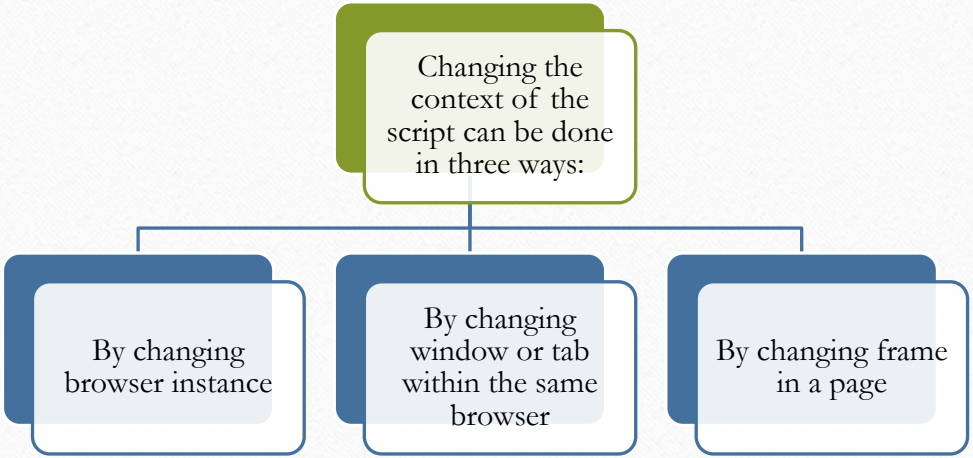


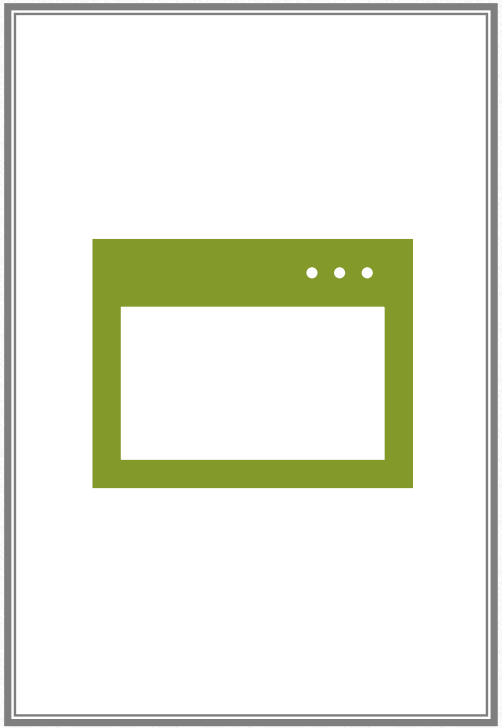
Change window context

•Sometimes you have to change the current context of the graphical interface you are testing. Several reasons can justify this:

- Check the test result in another system or run a test step in another application.
- The GUI of the application requires switching between frames or windows.
- The need for frame change. By changing the context of a particular frame, you will be able to find items within that frame.

Change window context





Open multiple webdrivers

- To open multiple browsers, multiple WebDriver objects can be created. Each WebDriver object controls a browser.
- WebDriver objects can control the same type of web browser (eg, Chrome, Firefox) or different types.
- Each browser is independently controlled by a WebDriver object.

Change the context with the same webdriver

- You can force the opening of a new window/tab with the command:

- **driver.execute_script("window.open()")**

- The new window/tab will be managed by the same object webdriver which opened it in this case.

- A web application can also open new tabs/windows on its own

Change the context with the same webdriver

- To switch between open tabs in a browser, one must first know the list of all open tabs.

- This list is in the attribute **window_handles** of the WebDriver object.

Change the context with the same webdriver

- Could you do scroll through all tabs/windows using the following code:

- **for handle in driver.window_handles:**
- **driver.switch_to.window(handle)**

- The safest way to determine which window is the currently open window is to use the attribute **driver.title**

Change the context with the same webdriver: frames

- To switch between frames on a page:
- You can identify a frame with its ID in the DOM
- The command to change the context to this frame is:
- **driver.switch_to.frame("foo")**

- You can also identify a frame in the same way as a classic web element.
- **frm_message = driver.find_element_by_name('message')**
- **driver.switch_to.frame(frm_message)**

- You can return to the parent page using:
- **driver.switch_to.default_content()**

Thank You

ATTAFI YOUSRA
EXPERT EN TEST LOGICIEL
@ : ATTAFI.YOSRA@HOTMAIL.FR
LINKEDIN : Yousra ATTAFI