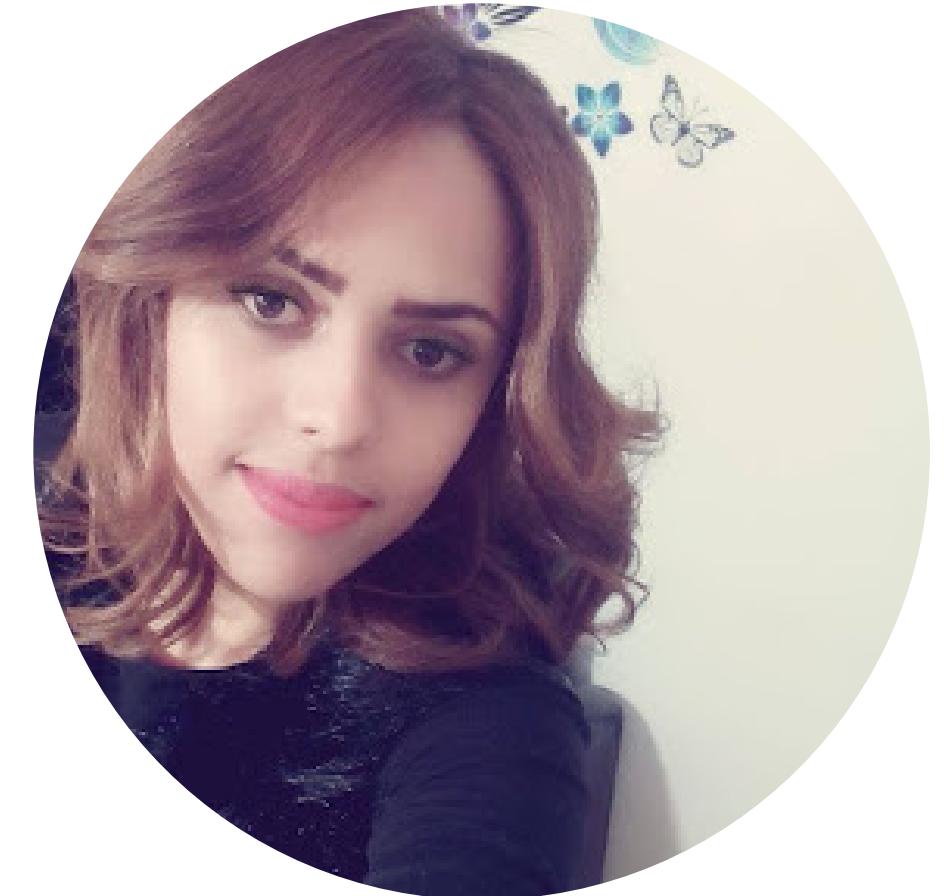


QA

with Marwa

A4Q Selenium Tester Foundation

02 Juillet 2025



Présenté par
Marwa Bettaieb

Agenda

QA

With Marwa

01

Base de l'automatisation des tests

02

Technologies Internet pour l'automatisation
des tests d'applications Web

03

Utiliser Selenium WebDriver Termes

04

Préparer des scripts de test maintenables

Mécanismes de logs et de reporting en automatisation des tests

Dans l'automatisation des tests, les scripts automatisés jouent un rôle essentiel : ils simulent les actions humaines sur le système testé (SUT) en utilisant des commandes clavier et souris. Plutôt que de concevoir ces scripts isolément, l'utilisation de bibliothèques spécialisées comme pytest apporte une solution plus efficace.

Pourquoi utiliser pytest ?

- pytest est un framework d'automatisation de tests écrit en Python, qui facilite l'écriture et l'organisation des tests.
- Il permet aussi d'automatiser des tests complexes, notamment en intégration avec Selenium WebDriver pour automatiser les interactions web.
- pytest exécute automatiquement tous les tests présents dans le répertoire courant ainsi que ses sous-répertoires.
- Il détecte les fichiers de test suivant les patterns "test_*.py" ou "*_test.py".

Mécanismes de logs et de reporting en automatisation des tests

Fonctionnement de pytest

- Lancer la commande pytest sans options exécute tous les tests détectés dans le répertoire courant et ses sous-dossiers.
- Plusieurs options permettent d'adapter l'exécution :
 - pytest -v : mode verbeux, affiche le nom complet des tests exécutés au lieu d'un simple point.
 - pytest -q : mode silencieux, limite les informations affichées pendant l'exécution.
 - pytest --html=report.html : génère un rapport HTML détaillé de l'exécution des tests.

Annotation des tests

pytest permet d'annoter les tests pour influencer leur comportement :

- @pytest.mark.skip : marque un test pour qu'il soit ignoré lors de l'exécution.
- @pytest.mark.xfail : indique que le test est censé échouer ; il sera exécuté mais un échec sera attendu.

Mécanismes de logs et de reporting en automatisation des tests

Importance des logs détaillés :Lorsqu'un testeur manuel

rencontre un échec lors de l'exécution d'un cas de test, il comprend généralement bien les circonstances ayant conduit à cet échec. En revanche, dans l'automatisation, les messages d'erreur générés sont souvent vagues, manquant de contexte, ce qui complique le diagnostic précis.

Par exemple, une défaillance peut survenir à l'étape N, alors que le rapport indique une erreur à l'étape N+1, rendant difficile la compréhension de la cause réelle.

Améliorer le diagnostic grâce aux logs

Pour pallier ce problème, il est crucial d'intégrer des logs détaillés tout au long de l'exécution des scripts automatisés. Ces logs doivent contenir des informations sur :

- Les données utilisées à chaque étape,
- Les comportements observés,
- Les états du système testé (SUT) avant et après chaque action.

Un enregistrement précis permet de distinguer si l'échec provient du système testé ou d'un problème dans le script d'automatisation.

Mécanismes de logs et de reporting en automatisation des tests

Rôle des logs dans les projets critiques et petits projets

- Dans les projets critiques, les logs détaillés sont indispensables pour les audits et peuvent être activés uniquement en cas d'échec pour limiter la surcharge.
- Dans les petits projets d'automatisation, un succès entraîne souvent une demande accrue d'automatisation, car plus un projet réussit, plus les attentes managériales augmentent.

Logging en Python avec Selenium WebDriver

Python propose une bibliothèque de logging puissante et flexible, très utile pour suivre l'exécution des tests automatisés avec Selenium WebDriver.

Les testeurs peuvent insérer des messages à tout moment dans leurs scripts, par exemple :

- Messages d'information sur le déroulement du test,
- Exemple : « Je suis sur le point de cliquer sur le bouton XYZ »
- Avertissements sur des événements inhabituels,
- Exemple : « L'ouverture du fichier <ABC> a pris plus de temps que prévu »
- Messages d'erreur déclenchant des exceptions et arrêtant le test.

Mécanismes de logs et de reporting en automatisation des tests

Niveaux de messages dans la bibliothèque logging de Python

La bibliothèque propose cinq niveaux de messages, du moins grave au plus grave :

- Niveau Description
- DEBUG Informations détaillées pour diagnostiquer les problèmes.
- INFO Informations générales sur le déroulement du test.
- WARNING Avertissements sur des événements inattendus mais non bloquants.
- ERROR Erreurs majeures affectant le test.
- CRITICAL Problèmes critiques, souvent bloquants.

03-Utiliser Selenium WebDriver Termes

QA

with Marwa

Exemple:

```
import logging
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.common.exceptions import NoSuchElementException

# Configuration des logs
logging.basicConfig(
    filename='logs/google_test.log',
    level=logging.DEBUG, # Niveau minimum de log à capturer
    format='%(asctime)s - %(levelname)s - %(message)s'
)

try:
    logging.info("Initialisation du navigateur")
    driver = webdriver.Chrome()

    logging.info("Accès à la page Google")
    driver.get("https://www.google.com")

    logging.debug("Tentative de localisation de la barre de recherche")
    try:
        search_box = driver.find_element(By.NAME, "q")
        logging.info("✅ Barre de recherche trouvée")
    except NoSuchElementException:
        logging.error("❌ Barre de recherche non trouvée !")
        raise

    logging.info("Saisie du mot 'ChatGPT' dans la barre de recherche")
    search_box.send_keys("ChatGPT")
    search_box.submit()

    logging.debug("Recherche soumise, attente du chargement des résultats")

    # Exemple d'avertissement simulé
    logging.warning("⚠️ Aucun contrôle explicite de la page de résultats (à implémenter)")

    # Exemple d'erreur simulée
    if "Erreur" in driver.title:
        logging.error("❌ Le titre contient le mot 'Erreur'")
        raise Exception("Page de résultats incorrecte")

    logging.info("✅ Test terminé avec succès")

except Exception as e:
    logging.critical(f"🔥 Erreur critique lors de l'exécution du test : {e}")
    raise

finally:
    logging.info("Fermeture du navigateur")
    driver.quit()
```

Les assertions:

Lors de l'exécution d'un cas de test, il est essentiel de vérifier le comportement effectif d'un système en définissant des résultats attendus pour chaque action sur le SUT (System Under Test).

En Python, les assertions permettent de vérifier que des conditions sont vraies à un moment donné dans le code

```
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By

@pytest.fixture
def driver():
    driver = webdriver.Chrome()
    driver.maximize_window()
    yield driver
    driver.quit()

def test_login_success(driver):
    # Aller à la page de login
    driver.get("https://practicetestautomation.com/practice-test-login/")

    # Remplir les champs de login
    driver.find_element(By.ID, "username").send_keys("student")
    driver.find_element(By.ID, "password").send_keys("Password123")
    driver.find_element(By.ID, "submit").click()

    # Vérification du contenu du message affiché
    welcome_message = driver.find_element(By.TAG_NAME, "h1").text

    # Assertion sur le texte exact
    assert welcome_message == "Logged In Successfully", "Message de succès incorrect"

    # Assertion sur un mot-clé partiel
    assert "Logged" in welcome_message, "Le mot 'Logged' est absent du message"
```



Le reporting dans les tests automatisés

Le reporting est un processus distinct de l'enregistrement des logs, bien qu'il s'en inspire fortement. Il vise à communiquer efficacement les résultats des tests aux différentes parties prenantes du projet (équipes QA, développeurs, chefs de projet, clients...).

🔍 Logs vs Reporting : quelle différence ?

Logs	Reporting
Détails techniques de l'exécution	Résumé global des résultats
Utiles pour les testeurs et les automaticiens	Destinés à un public plus large (non technique)
Enregistrés en temps réel	Générés après exécution
Très détaillés (étape par étape)	Synthétiques, visuels (tableaux, graphiques...)

Naviguer dans différentes URLs avec Selenium WebDriver

Pour exécuter des tests automatisés sur différents navigateurs à l'aide de Selenium WebDriver, il est nécessaire d'utiliser des pilotes (drivers) spécifiques à chaque navigateur. Ces pilotes servent d'interface entre Selenium et le navigateur.

Pilotes (drivers) selon les navigateurs

Navigateur	Pilote requis	Nom du fichier / package
Google Chrome	ChromeDriver	chromedriver.exe
Mozilla Firefox	GeckoDriver	geckodriver.exe
Edge	Edge WebDriver	msedgedriver.exe ou MicrosoftWebDriver.msi
Internet Explorer	IE Driver	IEDriverServer.exe
Safari	Safari Driver (intégré sur macOS)	safaridriver
HtmlUnit	HtmlUnit Driver (Java headless)	htmlunit-driver (non utilisé en Python)

Exemple minimal avec Selenium en Python

```
from selenium import webdriver

# Si chromedriver est bien dans le PATH, pas besoin de spécifier son chemin
driver = webdriver.Chrome()

driver.get("https://www.google.com")
print(driver.title)

driver.quit()
```

Naviguer dans différentes URLs avec Selenium WebDriver

Méthodes de navigation principales

Méthode Selenium	Description
driver.get(url)	Charge une nouvelle page Web via l'URL spécifiée.
driver.back()	Revient à la page précédente dans l'historique du navigateur.
driver.forward()	Avance d'une page dans l'historique (après un back()).
driver.refresh()	Recharge la page courante (F5).
driver.current_url	Retourne l'URL actuelle de la page.
driver.title	Retourne le titre de la page affichée (utile pour vérifications).

Naviguer dans différentes URLs avec Selenium WebDriver

Méthodes de navigation principales

Explications des méthodes de navigation :

- driver.get(url): Utilisée pour naviguer vers une nouvelle URL.
- driver.back(): Permet de revenir à la dernière page visitée, semblable au bouton "Retour" d'un navigateur.
- driver.forward(): Avance d'une page si vous êtes revenu en arrière, comme le bouton "Avancer".
- driver.refresh(): Recharge la page actuelle, ce qui peut être utile pour mettre à jour les informations affichées.
- driver.get_current_url() : Retourne l'URL de la page active, utile pour vérifier si vous êtes sur la bonne page.
- driver.get_title() : Retourne le titre de la page, ce qui peut aider à valider que la navigation a réussi

Naviguer dans différentes URLs avec Selenium WebDriver

Vérification de la page actuelle après navigation:

Lors de l'automatisation de tests avec Selenium WebDriver, il est fortement recommandé de valider que la page ouverte est bien celle attendue après une action de navigation.

Pourquoi vérifier la page courante ?

Naviguer vers une page ne garantit pas toujours que la page correcte s'est chargée (problème réseau, redirection, erreur 404, etc.).

C'est pourquoi Selenium fournit deux attributs essentiels dans l'objet WebDriver :

- ◆ **driver.current_url**

Renvoie l'URL exacte de la page chargée.Utile pour confirmer qu'on a bien été redirigé vers la bonne page.

- ◆ **driver.title**

Renvoie le titre de la page (ce qui s'affiche dans l'onglet du navigateur).Permet de confirmer que le contenu de la page est celui attendu.



Exemple pratique : vérification après navigation

```
from selenium import webdriver
import time

# Initialisation du navigateur
driver = webdriver.Chrome()

# Étape 1 : Aller sur le site de Python
driver.get("https://www.python.org")

# Vérification de l'URL et du titre
assert "python.org" in driver.current_url, "✗ Mauvaise URL"
assert "Welcome to Python.org" == driver.title, "✗ Titre inattendu"

print("✓ Navigué vers Python.org avec succès")

time.sleep(2)
```

```
# Étape 2 : Aller sur Wikipedia
driver.get("https://www.wikipedia.org")

# Vérifications
assert driver.current_url ==
"https://www.wikipedia.org/", "✗ URL
incorrecte"
assert "Wikipedia" in driver.title, "✗ Le titre ne
contient pas 'Wikipedia'

print("✓ Navigué vers Wikipedia avec succès")

driver.quit()
```

Naviguer dans différentes URLs avec Selenium WebDriver

🧪 Ouvrir plusieurs navigateurs avec Selenium

◆ Principe :

- Chaque instance de navigateur nécessite un objet WebDriver séparé.
- Il est possible d'ouvrir plusieurs fenêtres en parallèle, que ce soit :
 - deux fois le même navigateur (ex. deux Chrome),
 - ou deux navigateurs différents (ex. Chrome et Firefox).
- Il faut fermer chaque instance à la fin du test pour libérer les ressources

```
from selenium import webdriver
import time

# Initialiser Chrome
chrome_driver = webdriver.Chrome()
chrome_driver.get("https://www.google.com")
print("✅ Chrome lancé - Titre :", chrome_driver.title)

# Initialiser Firefox
firefox_driver = webdriver.Firefox()
firefox_driver.get("https://www.wikipedia.org")
print("✅ Firefox lancé - Titre :", firefox_driver.title)

# Attente pour observer les deux navigateurs
time.sleep(5)

# Fermer les navigateurs proprement
chrome_driver.quit()
firefox_driver.quit()
```

Changer de contexte de fenêtre (onglets)

◆ Contexte

Dans Selenium, chaque onglet ou fenêtre a un identifiant unique appelé handle.

Pour manipuler plusieurs onglets dans un même navigateur, on utilise :

Attribut / Méthode	Description
driver.window_handles	Liste de tous les identifiants des onglets ouverts.
driver.switch_to.window()	Permet de changer de contexte vers un autre onglet.

Changer de contexte de fenêtre (onglets)

Exemple : ouvrir deux onglets et basculer entre eux:

```
from selenium import webdriver
import time

# Lancer le navigateur (Chrome dans cet exemple)
driver = webdriver.Chrome()

# Ouvrir le 1er site
driver.get("https://www.google.com")
print("● Onglet 1 - Google")
time.sleep(2)

# Ouvrir un nouvel onglet (onglet 2)
driver.execute_script("window.open('https://www.wikipedia.org',
'_blank');")
time.sleep(2)
# Obtenir la liste des onglets
onglets = driver.window_handles
```

```
# Bascule vers le 2ème onglet (Wikipedia)
driver.switch_to.window(onglets[1])
print("● Onglet 2 - Wikipedia")
print("Titre :", driver.title)
time.sleep(2)

# Revenir au 1er onglet (Google)
driver.switch_to.window(onglets[0])
print("🔄 Retour à l'onglet 1")
print("Titre :", driver.title)
time.sleep(2)

# Fermer tous les onglets
driver.quit()
```

Changer de cadre (frame) ou de fenêtre avec Selenium

✿ Pourquoi manipuler les frames ?

- Certains sites utilisent des cadres (iframes) pour afficher du contenu (ex. : publicités, éditeurs intégrés, cartes...).
- Dans ces cas, Selenium ne peut pas interagir avec les éléments à l'intérieur de la frame tant que tu n'as pas changé de contexte vers cette frame.

◆ Méthodes utiles pour les frames:

Méthode	Description
driver.switch_to.frame(name_or_id)	Bascule vers une frame par son id ou name.
driver.switch_to.frame(web_element)	Bascule vers une frame identifiée comme WebElement.
driver.switch_to.parent_frame()	Revient à la frame parente .
driver.switch_to.default_content()	Revient à la page principale (en sortant de toutes les frames).

Changer de cadre (frame) ou de fenêtre avec Selenium

◆ Méthodes pour la fenêtre du navigateur

Méthode	Effet
driver.minimize_window()	Réduit la fenêtre.
driver.maximize_window()	Agrandit la fenêtre.
driver.fullscreen_window()	Passe en plein écran.

Changer de cadre (frame) ou de fenêtre avec Selenium



Exemple : changer de frame et manipuler la fenêtre

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time

driver = webdriver.Chrome()
driver.maximize_window()

# Page contenant une iframe (exemple de test public)
driver.get("https://the-internet.herokuapp.com/iframe")

time.sleep(2)

# Bascule vers la frame avec son ID
driver.switch_to.frame("mce_0_ifr")

# Interagir avec un élément dans la frame
text_area = driver.find_element(By.ID, "tinymce")
text_area.clear()
text_area.send_keys("✓ Texte saisi dans la frame")

time.sleep(2)
```

```
# Revenir au contenu principal (hors de la frame)
driver.switch_to.default_content()
# Réduction de la fenêtre
driver.minimize_window()
time.sleep(1)
# Maximisation
driver.maximize_window()
time.sleep(1)
# Plein écran
driver.fullscreen_window()
time.sleep(2)
# Fermer
driver.quit()
```

Changer de cadre (frame) ou de fenêtre avec Selenium

Résumé:

Objectif	Méthode Selenium
Aller dans une frame par ID ou name	switch_to.frame("frame_id")
Aller dans une frame via un élément	switch_to.frame(frame_element)
Revenir à la frame parente	switch_to.parent_frame()
Revenir à la page principale	switch_to.default_content()
Redimensionner la fenêtre	minimize_window(), maximize_window(), fullscreen_window()

Capturer des captures d'écran de pages Web

◆ Pourquoi capturer des captures d'écran ?

Contrairement aux testeurs manuels, les scripts d'automatisation ne voient pas l'écran. Les captures d'écran deviennent donc essentielles dans les cas suivants :

- Lorsqu'un échec est détecté (pour analyse visuelle).
- Pour les tests visuels ou d'interface.
- Pour les audits (ex. : logiciel critique ou réglementé).
- Pour vérifier le comportement sur différentes configurations systèmes.

Capturer des captures d'écran de pages Web

💡 Quand prendre des captures d'écran ?

- À la fin d'un test (dans le bloc tearDown).
- Juste après une action importante (ex. : soumission de formulaire).
- Lorsqu'une erreur est capturée dans un bloc try/except.

💼 Méthodes disponibles

Fonction Selenium	Description
driver.save_screenshot("nom.png")	Capture toute la page visible du navigateur.
element.screenshot("nom.png")	Capture uniquement un élément spécifique .

Exemple complet : capturer page entière + un élément

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time
```

Initialiser le navigateur

```
driver = webdriver.Chrome()
driver.get("https://www.python.org")
driver.maximize_window()
```

Attente pour bien charger la page

```
time.sleep(2)
```

1. Capture de toute la page

```
driver.save_screenshot("screenshot_page.png")
print("✅ Capture de la page entière enregistrée.")
```

2. Capture d'un élément spécifique (le champ de recherche)

```
search_input = driver.find_element(By.ID, "id-search-field")
search_input.screenshot("screenshot_element.png")
print("✅ Capture de l'élément enregistrée.")
```

Fermer le navigateur

```
driver.quit()
```

Capturer des captures d'écran de pages Web

⚠ Problèmes lors de la capture d'écran en automatisation

1. 🐢 Temps de traitement (latence de capture)



Prendre une capture d'écran, surtout d'une page complète ou d'un élément complexe, peut prendre du temps.

Cela dépend des ressources de la machine (CPU/RAM), du navigateur et des animations actives sur la page.



Solutions :

Utiliser `time.sleep()` ou `WebDriverWait` avant la capture pour s'assurer que la page est complètement rendue.

Réduire les animations CSS si possible pendant les tests.

Capturer des captures d'écran de pages Web

Exemple:

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

# Exemple de capture après attente d'un élément
WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, "id-search-field")))
driver.save_screenshot("final_state.png")
```

Capturer des captures d'écran de pages Web

2. Changements dynamiques de l'interface (AJAX, JavaScript, SPA)

Problème :

- Si des composants de la page sont chargés asynchronement (AJAX, chargement différé, React/Angular...), la capture peut se faire trop tôt, avant que les éléments ne soient visibles ou finalisés.

Solutions :

- Utiliser des attentes explicites (WebDriverWait) sur l'apparition ou la visibilité de l'élément.
- Capturer l'élément lui-même, plutôt que la page entière.

Capturer des captures d'écran de pages Web

Exemple:

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC

element = WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.ID, "id-search-field"))
)
element.screenshot("search_field_ready.png")
```

Capturer des captures d'écran de pages Web

Résumé:

Problème	Solution recommandée
Capture trop rapide	WebDriverWait ou sleep() pour attendre le bon moment
Animation ou contenu dynamique	Attendre la fin du chargement AJAX / JS avant de capturer
État incomplet visible	Capturer l'élément spécifique plutôt que toute la page
Nom de fichier écrasé	Ajouter un horodatage ou un nom de test unique

Capturer des captures d'écran de pages Web

◆ Pourquoi capturer des captures d'écran ?

Contrairement aux testeurs manuels, les scripts d'automatisation ne voient pas l'écran. Les captures d'écran deviennent donc essentielles dans les cas suivants :

- Lorsqu'un échec est détecté (pour analyse visuelle).
- Pour les tests visuels ou d'interface.
- Pour les audits (ex. : logiciel critique ou réglementé).
- Pour vérifier le comportement sur différentes configurations systèmes.

Capturer des captures d'écran de pages Web

Solutions alternatives pour traiter les captures d'écran

◆ 1. Capture en Base64 (chaîne encodée)

Avantages :

- Pas besoin de créer un fichier image sur le disque.
- Idéal pour :
 - Envoyer directement par API (ex. : Slack, Email, API REST...)
 - Stocker dans une base de données
 - Afficher dans un rapport HTML via une balise

Capturer des captures d'écran de pages Web

Solutions alternatives pour traiter les captures d'écran

📌 **Utilisation en Selenium Python :**

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.get("https://www.python.org")
# Capture en Base64
image_base64 = driver.get_screenshot_as_base64()
print(image_base64[:100]) # Affiche les 100 premiers caractères
driver.quit()
```

Capturer des captures d'écran de pages Web

Solutions alternatives pour traiter les captures d'écran

◆ 2. 📦 Capture en binaire (PNG brut)

✓ Avantages :

- Permet de manipuler l'image directement en mémoire, sans l'écrire dans un fichier.
- Utile pour :
 - Traitement avec des bibliothèques comme Pillow
 - Envoi via des flux réseau
 - Compression, transformation ou comparaison d'images

Capturer des captures d'écran de pages Web

Solutions alternatives pour traiter les captures d'écran

📌 **Utilisation en Selenium Python :**

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://www.python.org")

# Capture de l'image en binaire
image_data = driver.get_screenshot_as_png()

# Exemple : enregistrer dans un fichier si besoin
with open("capture_directe.png", "wb") as f:
    f.write(image_data)

driver.quit()
```

Capturer des captures d'écran de pages Web

Comparatif : Base64 vs Binaire

Méthode	Format de sortie	Cas d'usage idéal
get_screenshot_as_base64()	Chaîne texte encodée Base64	Intégration dans API, base de données, rapport HTML
get_screenshot_as_png()	Données binaires (PNG)	Traitement d'image (Pillow), comparaison, transformation

Localiser les éléments de l'interface graphique

◆ Objectif

Pour automatiser les interactions avec une page web (clic, saisie de texte, vérification...), il est essentiel d'identifier précisément les éléments HTML. Selenium fournit pour cela des méthodes puissantes :

Localiser les éléments de l'interface graphique

Méthode	Utilisation	Exemple
find_element(By.ID, ...)	Par ID unique	driver.find_element(By.ID, "username")
find_element(By.CLASS_NAME, ...)	Par nom de classe CSS	driver.find_element(By.CLASS_NAME, "btn-primary")
find_element(By.TAG_NAME, ...)	Par nom de balise HTML	driver.find_element(By.TAG_NAME, "input")
find_element(By.XPATH, ...)	Par chemin XPath	driver.find_element(By.XPATH, "//input[@type='email']")
find_element(By.CSS_SELECTOR, ...)	Par sélecteur CSS	driver.find_element(By.CSS_SELECTOR, "div#content p")

Localiser les éléments de l'interface graphique

un exemple pratique pour illustrer clairement la différence entre :

- `find_element()` → retourne le premier élément correspondant
- `find_elements()` → retourne une liste de tous les éléments correspondants

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://www.wikipedia.org")

# 1. Par ID
search_box = driver.find_element(By.ID, "searchInput")

# 2. Par nom de classe
lang_links = driver.find_elements(By.CLASS_NAME, "link-box")

# 3. Par balise
input_tags = driver.find_elements(By.TAG_NAME, "input")

# 4. Par XPath
logo = driver.find_element(By.XPATH, '//*[@class="central-featured-logo"]')
```

5. Par CSS Selector

```
search_button = driver.find_element(By.CSS_SELECTOR, "button.pure-button")
```

Actions (par exemple taper un mot)

```
search_box.send_keys("Python")
```

```
driver.quit()
```

Localiser les éléments de l'interface graphique

Exemple : récupérer les liens <a> sur une page web avec Selenium

```
from selenium import webdriver
from selenium.webdriver.common.by import By

# Lancer le navigateur
driver = webdriver.Chrome()
driver.get("https://www.wikipedia.org")

# 🔘 1. Récupérer le **premier lien** trouvé (find_element)
first_link = driver.find_element(By.TAG_NAME, "a")
print("👉 Premier lien trouvé :", first_link.get_attribute("href"))
```

```
# 🔘 2. Récupérer **tous les liens** (find_elements)
all_links = driver.find_elements(By.TAG_NAME, "a")
print("🔗 Nombre total de liens trouvés :", len(all_links))

# Afficher les 5 premiers liens
for i, link in enumerate(all_links[:5]):
    print(f" 🔘 Lien {i+1} :", link.get_attribute("href"))

driver.quit()
```

Résultat attendu

- La console affiche l'URL du premier lien.
- Puis elle affiche le nombre total de liens <a> sur la page.
- Ensuite, elle affiche les 5 premiers liens parmi la liste récupérée.

Localiser les éléments de l'interface graphique

Résumé : Différence entre les deux

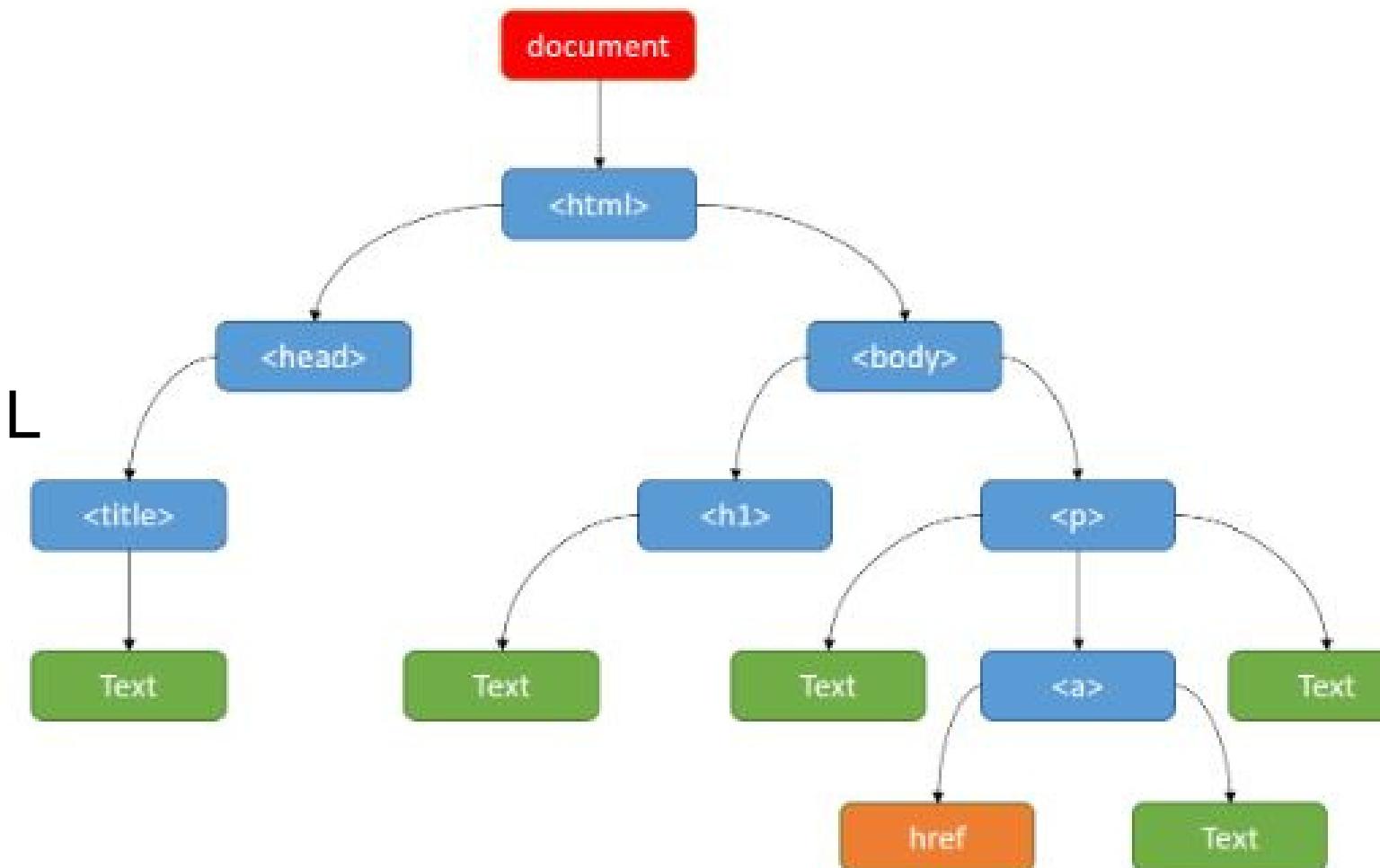
Méthode	Retourne	Utilisation typique
find_element(...)	Le premier élément trouvé	Interaction unique : champ, bouton, etc.
find_elements(...)	Une liste d'éléments	Parcours de tableaux, listes, balises répétées

Localiser les éléments de l'interface graphique

Le DOM (Document Object Model) est un concept clé dans le développement et les tests web. Lorsqu'une page web est chargée, le navigateur génère un modèle DOM, une représentation hiérarchique de la page en tant qu'arbre d'objets. Ce modèle standardisé permet d'accéder dynamiquement au contenu, à la structure et au style de la page, comme défini par le W3C

Le DOM définit :

- Tous les éléments HTML en tant qu'objets.
- Les propriétés de chaque élément HTML.
- Les méthodes d'accès aux éléments HTML.
- Les événements qui peuvent affecter les éléments HTML



HTML Methods – Localisation des éléments

Les méthodes suivantes dépendent de la recherche d'éléments de l'écran en recherchant des artefacts HTML dans le document..

Localisation par ID

```
elment = driver.find_element(By.ID, "identifiant_unique")
```

Avantages :

- Performance élevée : Accès rapide aux éléments.
- Unicité garantie : Un ID est unique dans le document HTML.
- Modifications possibles : Un testeur peut ajouter des IDs pour faciliter la localisation.

Inconvénients :

- Dynamique : Les IDs générés automatiquement peuvent changer dynamiquement.
- Portabilité limitée : Les ID ne sont pas appropriés pour le code qui est utilisé à plusieurs endroits, par exemple, un emplate utilisé pour générer des en-têtes et des pieds de page pour différents modes de dialogue..
- Accès restreint : Il se peut qu'un testeur ne soit pas autorisé à modifier le code du SUT.

HTML Methods – Localisation des éléments

Localisation par nom de classe

```
element = driver.find_element(By.CLASS_NAME, "nom-de-classe")
```

Avantages :

- *Utilisation flexible* : Les noms de classes peuvent être utilisés à plusieurs endroits dans le DOM, mais vous pouvez limiter la localisation à la page chargée (par exemple dans une fenêtre popup modale).
- *Modifications possibles* : Un testeur peut facilement ajouter des classes (note : ces changements doivent être revus et vérifiés au moins sous la forme d'unerevue informelle).

Inconvénients :

- *Précision nécessaire* : Comme les noms de classe peuvent être utilisés à plusieurs endroits, il faut faire plus attention de ne pas localiser le mauvais élément.
- *Accès restreint* : Le testeur pourrait ne pas être autorisé à ajouter des classes dans le code source.

HTML Methods – Localisation des éléments

Localisation par nom de balise

```
element = driver.find_element(By.TAG_NAME, "tagname")
```

Avantages :

- Utilité si unique : Si une balise est unique à une page, vous pouvez restreindre l'endroit où chercher.

Inconvénients :

- Ambiguïté possible : Si une balise n'est pas unique à une page, vous pouvez trouver le mauvais élément.

HTML Methods – Localisation des éléments

Localisation par texte de lien :deux méthodes Selenium correspondantes

texte complet du lien:

```
element = driver.find_element(By.LINK_TEXT, "Texte complet du lien")
```

Texte partiel du lien :

```
element = driver.find_element(By.PARTIAL_LINK_TEXT, "Texte partiel")
```

Avantages :

- Si le texte du lien est unique à une page, vous pouvez trouver l'élément.
- Le texte du lien est visible par l'utilisateur (dans la plupart des cas), il est donc facile de savoir ce que le code de test recherche.
- Un texte de lien partiel est un peu moins susceptible d'être modifié que le texte intégral du lien.

Inconvénients :

- Le texte du lien est plus susceptible de changer qu'un ID ou un nom de classe.
- L'utilisation d'un texte partiel peut rendre plus difficile l'identification unique d'un lien unique..

Méthodes XPath

Qu'est-ce que XPath ?

- XPath (XML Path Language) est un langage utilisé pour localiser des nœuds dans un document XML/HTML.
- Très puissant pour cibler précisément un élément dans un DOM complexe.
- En tests WebDriver, XPath est largement utilisé pour sélectionner des éléments difficiles à cibler autrement.

Type	Description	Exemple simplifié	Avantages / Inconvénients
Chemin absolu	Démarre à la racine (/html/body/...) et décrit tout le chemin jusqu'à l'élément ciblé.	/html/body/div[2]/a[1]	✗ Très fragile : toute modification casse le chemin
Chemin relatif	Démarre n'importe où (//div/a), cherchant des éléments quel que soit l'endroit dans le DOM.	//div[@class='menu']/a[1]	✓ Robuste et flexible face aux changements du DOM

Méthodes XPath

- Utilisation des expressions XPath pour divers attributs .

XPath permet de rechercher des éléments en utilisant divers attributs (id, class, name, type, etc.), ce qui offre une grande flexibilité.

Voici quelques exemples de requêtes XPath courantes :

- Par ID : //input[@id='champ2']
- Par classe : //div[@class='classeExemple']
- Par texte : //button[text()='Envoyer']
- Par position : //div/input[position()=2]

Méthodes XPath

XPath est effectivement très puissant pour localiser des éléments dans une page web, et l'utilisation de fonctions de recherche génériques en passant le type d'attribut permet de rendre le code plus flexible et réutilisable.

```
from selenium.webdriver.common.by import By

def find_element(driver, attribute_type, attribute_value):
    # Construction de la chaîne XPath en fonction de l'attribut
    if attribute_type == "id":
        path_string = f"//[@id='{attribute_value}'"
    elif attribute_type == "class":
        path_string = f"//[@class='{attribute_value}'"
    elif attribute_type == "name":
        path_string = f"//[@name='{attribute_value}'"
    else:
        raise ValueError("Type d'attribut non pris en charge")

    # Recherche de l'élément en utilisant XPath directement
    try:
        element = driver.find_element(By.XPATH, path_string)
        return element
    except:
        # Retourne None si l'élément n'est pas trouvé
        return None
```

Méthodes XPath

Le **f** avant les guillemets indique que c'est une **f-string**.

Le **{attribute_value}** à l'intérieur des accolades dans la chaîne sera remplacé par la valeur de la variable `attribute_value`. C'est une façon pratique et lisible de formater des chaînes de caractères sans avoir à utiliser `.format()` ou des opérateurs de concaténation (+)

```
# Recherche d'un élément par identifiant
element_by_id = find_element(driver, "id", "monIdentifiant")

# Recherche d'un élément par classe
element_by_class = find_element(driver, "class", "maClasse")

# Recherche d'un élément par nom
element_by_name = find_element(driver, "name", "monNom")
```

Méthodes de sélection CSS(By.CSS_SELECTOR)

Les sélecteurs CSS sont un autre moyen très courant de localiser les éléments HTML dans une page, inspiré du style CSS.

Exemple HTML

```
html
<input type="text" id="username" class="form-input" name="user">
```

Exemple de sélection CSS

```
python

driver.find_element(By.CSS_SELECTOR, "#username")      # par ID
driver.find_element(By.CSS_SELECTOR, ".form-input")    # par classe
driver.find_element(By.CSS_SELECTOR, "input[name='user']") # par attribut
```

CSS avec règles de la section 2.3 (Structure de nœud)

```
python

from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://example.com") # Remplace par ton URL

# Sélectionne le premier <p class="paragraphe">
paragraph = driver.find_element(By.CSS_SELECTOR, "p.paragraphe")
print(paragraph.text)

driver.quit()
```

Méthodes de sélection CSS(By.CSS_SELECTOR)

Avantage	Explication	Inconvénient	Détail
 Ciblage précis	Permet de combiner plusieurs critères (tag, class, id, attributs).	 Pas de recherche par texte	Contrairement à XPath, CSS ne permet pas de cibler un élément par texte visible (text()).
 Restriction à un contexte	Possibilité de localiser un élément dans un div, section, etc.	 Non unique	Si plusieurs éléments ont la même classe, vous risquez de cibler le mauvais .
 Plus rapide que XPath	Les navigateurs optimisent généralement mieux les sélecteurs CSS.	 Limité aux éléments statiques	Moins flexible que XPath pour les cas très complexes (ex. : relations entre nœuds).
 Lisible et familier	Syntaxe connue des développeurs front-end (HTML/CSS).		

Méthodes de sélection CSS(By.CSS_SELECTOR)

Exemple complet avec Selenium

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://example.com")

# Sélecteur par ID
username = driver.find_element(By.CSS_SELECTOR,
"#username")

# Sélecteur par classe
input_field = driver.find_element(By.CSS_SELECTOR,
".form-input")
```

```
# Sélecteur par attribut
button =
driver.find_element(By.CSS_SELECTOR,
"button[type='submit']")

# Combinaison de sélecteurs
link =
driver.find_element(By.CSS_SELECTOR,
"div.nav a.active")
driver.quit()
```

Localisation via des conditions prédéfinies

Selenium avec Python intègre un module appelé **expected_conditions** qui peut être importé à partir de **selenium.webdriver.support** avec des conditions prédéfinies. Vous pouvez créer des classes de conditions personnalisées, mais les classes prédéfinies devraient satisfaire la plupart de vos besoins. Ces classes offrent une plus grande spécificité que les localisateurs mentionnés ci-dessus. En d'autres termes, ils ne se contentent pas de déterminer si un élément existe, ils permettent aussi de vérifier les états spécifiques dans lesquels cet élément se trouve. Par exemple, la fonction **element_to_be_selected()** détermine non seulement que l'élément existe, mais elle vérifie également s'il est dans un état sélectionné.

Cette liste n'est pas exhaustive, mais en donne quelques exemples :

- 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)

Localisation via des conditions prédéfinies

1. alert_is_present

◆ Définition

Cette condition vérifie si une alerte JavaScript est présente à l'écran. Elle est utile lorsqu'un test déclenche une alerte et qu'il faut attendre qu'elle apparaisse avant de continuer.

◆ Syntaxe

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

WebDriverWait(driver, 10).until(EC.alert_is_present())
```

◆ Exemple complet

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

driver = webdriver.Chrome()
driver.get("https://example.com/alerte")

# Attendre jusqu'à ce que l'alerte apparaisse
WebDriverWait(driver, 10).until(EC.alert_is_present())

# Accepter l'alerte
alert = driver.switch_to.alert
alert.accept()
```

Localisation via des conditions prédéfinies

2. element_selection_state_to_be(element, is_selected)

- ◆ Définition

Cette condition vérifie si l'état de sélection (ex. : case à cocher, bouton radio) d'un élément est égal à une valeur donnée (True ou False).

- ◆ Syntaxe

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

WebDriverWait(driver, 10).until(
    EC.element_selection_state_to_be(element, True)
)
```

3. element_to_be_clickable(locator)

- ◆ Définition :

Cette condition vérifie que l'élément est présent dans le DOM, visible ET activé, donc cliquable.

- ◆ Syntaxe :

```
WebDriverWait(driver, 10).until(
    EC.element_to_be_clickable((By.ID, "mon-bouton"))
)
```

Localisation via des conditions prédéfinies

4. element_to_be_selected(element)

- ◆ Définition :

Cette condition vérifie si un élément est sélectionné (utile pour les checkbox, radio buttons, ou <option> dans des listes déroulantes).

- ◆ Syntaxe

```
WebDriverWait(driver, 10).until(  
    EC.element_to_be_selected(mon_element)  
)
```

5. frame_to_be_available_and_switch_to_it(locator)

- ◆ Définition :

Cette condition :

1. Vérifie que la frame (cadre) est disponible dans le DOM,
2. Et bascule automatiquement dessus avec driver.switch_to.frame()

- ◆ Syntaxe

```
WebDriverWait(driver, 10).until(  
    EC.frame_to_be_available_and_switch_to_it((By.ID, "my-frame"))  
)
```

Localisation via des conditions prédéfinies

6. **invisibility_of_element_located(locator)**

- ◆ Définition : Cette condition vérifie que
 - L'élément n'est pas visible sur la page ou
 - Qu'il n'existe plus dans le DOM.

Très utile pour attendre la disparition d'un loader, d'un message temporaire, etc.

- ◆ Syntaxe

```
WebDriverWait(driver, 10).until(  
    EC.invisibility_of_element_located((By.ID, "loading-spinner"))  
)
```

7. **presence_of_element_located(locator)**

- ◆ Définition : Vérifie que l'élément est présent dans le DOM, même s'il n'est pas visible à l'écran.
- ◆ Utilise lorsque tu veux t'assurer qu'un élément a bien été chargé dans la page, même si tu ne peux pas encore interagir avec lui.

- ◆ Syntaxe

```
WebDriverWait(driver, 10).until(  
    EC.presence_of_element_located((By.ID, "champ-email"))  
)
```

Localisation via des conditions prédéfinies

8. `text_to_be_present_in_element(locator, text_)`

◆ Définition :

Vérifie qu'un texte précis est présent dans un élément donné (souvent un `<div>`, ``, `<p>`, etc.).

📌 Utile pour attendre qu'un message s'affiche, comme "Connexion réussie" ou "Chargement terminé".

◆ Syntaxe

```
WebDriverWait(driver, 10).until  
    EC.text_to_be_present_in_element((By.ID, "message"), "Succès")  
)
```

9. `title_is(title)`

◆ Définition: Vérifie que le titre de la page Web correspond exactement au texte fourni.

◆ Utilité :

- S'assurer qu'une page a bien été chargée.
- Contrôler la navigation correcte entre pages.

◆ Syntaxe

```
WebDriverWait(driver, 10).until  
    EC.title_is("Titre attendu")  
)
```

Localisation via des conditions prédéfinies

✓ 10. `visibility_of_element_located(locator)`

- ◆ Définition :

Vérifie que l'élément est présent dans le DOM ET visible (c'est-à-dire affiché à l'écran, avec hauteur > 0 et largeur > 0).

- ◆ Utilité :

- Confirmer que l'élément est prêt à être utilisé/interagi (contrairement à `presence_of_element_located` qui ne garantit pas la visibilité).

```
WebDriverWait(driver, 10).until(  
    EC.visibility_of_element_located((By.ID, "champ-login"))  
)
```

Localisation via des conditions prédéfinies

Un exemple complet en Python avec Selenium WebDriver montrant l'utilisation de plusieurs méthodes de localisation combinées avec des conditions prédéfinies (Expected Conditions) — un scénario réaliste et bien structuré :

🎯 Objectif du test automatisé :

- Aller sur une page de connexion.
- Attendre que les champs soient visibles.
- Vérifier que les boutons sont cliquables.
- Vérifier que certains textes sont présents.
- Utiliser différents types de localiseurs : ID, classe, CSS, XPath.
- Prendre une capture d'écran en cas de succès.

03-Utiliser Selenium WebDriver Termes

QA

with Marwa

Code complet d'exemple :

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

# Initialiser le navigateur
driver = webdriver.Chrome()
driver.get("https://example.com/login")

wait = WebDriverWait(driver, 10)

try:
    # Attendre que le champ login soit VISIBLE (via ID)
    login_input = wait.until(
        EC.visibility_of_element_located((By.ID, "username")))
    login_input.send_keys("MonNom")

    # Attendre que le champ mot de passe soit PRÉSENT (via CLASS_NAME)
    password_input = wait.until(
        EC.presence_of_element_located((By.CLASS_NAME, "password-input")))
    password_input.send_keys("MonMotDePasse")
# Vérifier que le bouton "Connexion" est CLIQUABLE (via CSS_SELECTOR)
    login_button = wait.until(
        EC.element_to_be_clickable((By.CSS_SELECTOR, "button.login-btn")))

```

```
# Vérifier que le texte "Connexion" est bien présent dans le bouton (via
# XPATH + text_to_be_present)
    wait.until(
        EC.text_to_be_present_in_element(
            (By.XPATH, "//button[@class='login-btn']"),
            "Connexion"
        )
    )

# Cliquer sur le bouton de connexion
    login_button.click()

# Vérifier que le titre de la page suivante est celui attendu
    wait.until(EC.title_is("Tableau de bord"))

# Prendre une capture d'écran de la page d'accueil après connexion
    driver.save_screenshot("screenshot_connexion_succes.png")

    print("✅ Test réussi - utilisateur connecté")

except Exception as e:
    print(f"❌ Test échoué : {e}")
    driver.save_screenshot("screenshot_erreur.png")

finally:
    time.sleep(3)
    driver.quit()
```

Obtenir l'état des éléments de l'interface graphique

🎯 Pourquoi ?

Dans les tests automatisés, il ne suffit pas de localiser un élément. Il faut aussi vérifier son état : est-il visible, activé, sélectionné, etc. Cela permet de garantir que les interactions sont possibles et que le SUT (System Under Test) se comporte comme attendu.

📌 Principales propriétés et méthodes pour vérifier l'état d'un élément Web :

03-Utiliser Selenium WebDriver Termes

QA

with Marwa

Méthode / Propriété	Description
element.is_displayed()	Retourne True si l'élément est visible à l'écran.
element.is_enabled()	Retourne True si l'élément est activé (cliquable ou éditable).
element.is_selected()	Retourne True si l'élément est sélectionné (checkbox, bouton radio).
element.get_attribute("attribut")	Permet de lire la valeur d'un attribut HTML (value, class, type, etc.).
element.text	Retourne le texte affiché dans l'élément.
element.tag_name	Retourne le nom de la balise HTML (ex: input, div, a).
element.size	Renvoie un dictionnaire avec la taille : {'height': x, 'width': y}.
element.location	Renvoie la position (x, y) de l'élément sur la page.

Obtenir l'état des éléments de l'interface graphique

 Exemple complet en Python :

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://example.com")

# Localisation de l'élément
element = driver.find_element(By.ID, "checkbox")

# Vérifications d'état
print("Visible :", element.is_displayed())
print("Activé :", element.is_enabled())
print("Sélectionné :", element.is_selected())
print("Texte :", element.text)
print("Tag :", element.tag_name)
print("Attribut class :", element.get_attribute("class"))
print("Taille :", element.size)
print("Position :", element.location)

driver.quit()
```

Obtenir l'état des éléments de l'interface graphique

🎯 À quoi ça sert dans un vrai test ?

Scénario	Vérification utile
Vérifier qu'un bouton est cliquable	<code>element.is_enabled()</code>
Vérifier qu'un champ ou message est bien affiché	<code>element.is_displayed()</code>
Vérifier qu'une case est déjà cochée	<code>element.is_selected()</code>
Comparer le texte affiché à un résultat attendu	<code>element.text</code>
Inspecter une valeur dynamique dans un champ	<code>element.get_attribute("value")</code>
S'assurer que la mise en page n'est pas cassée (largeur/hauteur)	<code>element.size</code>

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

🎯 Objectif :

Simuler les actions de l'utilisateur (clavier, souris) sur les éléments de l'interface graphique, comme un utilisateur réel le ferait.

📌 Avant d'interagir avec un élément Web :

Le script doit vérifier trois conditions :

Vérification	Objectif
Présence dans le DOM	L'élément existe bien sur la page
Visibilité à l'écran	L'élément est visible (affiché)
Activation (état interactif)	L'élément est activé (non désactivé ou grisé)

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Commandes WebDriver courantes pour interagir avec les éléments

Action utilisateur	Commande Selenium WebDriver	Exemple
Saisir du texte	element.send_keys("Texte")	champ_nom.send_keys("Marwa")
Vider un champ	element.clear()	champ_nom.clear()
Cliquer sur un bouton, checkbox, lien, etc.	element.click()	bouton_envoyer.click()
Lire un texte affiché	element.text	assert "Bienvenue" in message.text
Lire une valeur (ex : dans un champ input)	element.get_attribute("value")	champ.get_attribute("value")
Soumettre un formulaire	element.submit()	formulaire.submit()
Sélectionner une option dans une liste déroulante	Select(element).select_by_visible_text("Option")	voir diapo suivante



Exemple complet

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import Select
```

```
driver = webdriver.Chrome()
driver.get("https://example.com/formulaire")
```

Saisir du texte dans un champ

```
nom = driver.find_element(By.ID, "input-nom")
if nom.is_displayed() and nom.is_enabled():
    nom.clear()
    nom.send_keys("Marwa")
```

Cliquer sur une case à cocher

```
checkbox = driver.find_element(By.ID, "accept")
if checkbox.is_displayed() and not checkbox.is_selected():
    checkbox.click()
```

```
# Sélectionner une option dans une liste déroulante
select = Select(driver.find_element(By.ID, "pays"))
select.select_by_visible_text("Tunisie")
```

Cliquer sur un bouton

```
submit_btn = driver.find_element(By.CSS_SELECTOR,
"button[type='submit']")
if submit_btn.is_enabled():
    submit_btn.click()

driver.quit()
```



Astuce : Manipuler les menus déroulants

```
from selenium.webdriver.support.ui import Select

select_element = driver.find_element(By.ID, "mon-select")
select = Select(select_element)
select.select_by_index(1)
select.select_by_value("valeur")
select.select_by_visible_text("Texte visible")
```

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

🎯 Objectif :

Simuler les actions de l'utilisateur (clavier, souris) sur les éléments de l'interface graphique, comme un utilisateur réel le ferait.

📌 Avant d'interagir avec un élément Web :

Le script doit vérifier trois conditions :

Vérification	Objectif
Présence dans le DOM	L'élément existe bien sur la page
Visibilité à l'écran	L'élément est visible (affiché)
Activation (état interactif)	L'élément est activé (non désactivé ou grisé)

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Manipulation des champs de texte

Avant de saisir du texte dans un champ de texte modifiable, assurez-vous que l'élément est accessible, affiché et activé. Effacez ensuite le texte existant de l'élément et entrez la nouvelle chaîne de caractères souhaitée.

 **Effacer le texte existant :**

```
element.clear();
```

 **Saisir le texte souhaité dans le champ :**

```
element.sendKeys("le texte souhaité");
```

Cliquez sur des éléments web

Cliquer sur un élément web , tel qu'un bouton, un lien, une image ou un bouton radio , simule un clic de souris dans un test automatisé. Avant d'effectuer cette action, il est essentiel de vérifier que l'élément est bien cliquable. Pour cela, Selenium propose la méthode **elementToBeClickable()** de la classe **ExpectedConditions**, qui permet d'attendre que l'élément soit visible et activé.

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Une fois cette condition remplie, on peut appeler la méthode **click()** pour exécuter le clic. Par exemple :

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement bouton = wait.until(ExpectedConditions.elementToBeClickable(By.id("btnEnvoyer")));
bouton.click();
```

Si l'élément est un lien ou un bouton, le clic peut entraîner un changement de contexte (comme le chargement d'une nouvelle page), ce qui est généralement observable à l'écran. Dans le cas d'un bouton radio, il est recommandé de vérifier s'il a bien été sélectionné après le clic en utilisant la méthode **isSelected()** :

```
if (boutonRadio.isSelected()) {
    System.out.println("Le bouton radio est bien sélectionné.");
}
```

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Manipulation des cases à cocher

L'objectif principal est de vérifier l'état actuel de la case à cocher, puis de la manipuler selon l'état souhaité (sélectionnée ou non). Cette manipulation est cruciale pour garantir que le test automatisé reflète correctement l'intention fonctionnelle. Supposons que la case à cocher est stockée dans une variable **checkbox**, et que nous avons une variable booléenne **wantChecked** qui indique si elle doit être cochée (true) ou décochée (false).

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Exemple complet en Python :

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://exemple.com/formulaire")

checkbox = driver.find_element(By.ID, "termsCheckbox") # Localiser la case
want_checked = True # État souhaité : True = coché, False = décoché

# Étape 1 : Cocher ou décocher si nécessaire
if checkbox.is_selected() != want_checked:
    checkbox.click()

# Étape 2 : Vérifier l'état final
if checkbox.is_selected() == want_checked:
    print("✅ La case est maintenant dans l'état souhaité.")
else:
    print("❌ L'état de la case ne correspond pas à l'état attendu.")
```

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Manipulation des menus déroulants

Pour interagir avec les éléments de sélection (menus déroulants <select>), Selenium met à disposition la classe **Select** du module **selenium.webdriver.support.ui**. Cette classe permet de sélectionner une option de plusieurs façons : par valeur, par texte visible ou par index.

Importations et initialisation

```
from selenium import webdriver  
from selenium.webdriver.common.by import By  
from selenium.webdriver.support.ui import Select
```

Démarrer le navigateur

```
driver = webdriver.Chrome()  
driver.get("https://example.com")
```

Localiser le menu déroulant

```
dropdown = driver.find_element(By.ID, "dropdown") #l'ID  
select = Select(dropdown)
```

Désélectionner toutes les options

```
select.deselect_all()
```

Sélectionner une option par valeur

```
select.select_by_value("option_value")
```

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Sélectionner une option par texte visible

```
select.select_by_visible_text("Nom de l'option")
```

Récupérer la première option sélectionnée

```
first_selected = select.first_selected_option  
print(f"Première option sélectionnée : {first_selected.text}")
```

Désélectionner une option par valeur

```
select.deselect_by_visible_text("Nom de l'option")
```

Lister toutes les options disponibles

```
all_options = select.options  
for option in all_options:  
    print(f" Option disponible : {option.text}")
```

Récupérer la liste des options sélectionnées

```
selected_options = select.all_selected_options  
for option in selected_options:  
    print(f" Option sélectionnée : {option.text}")
```

Fermer le navigateur

```
driver.quit()
```

Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

Travailler avec les boîtes de dialogue modal

Une boîte de dialogue modale est une fenêtre qui s'ouvre par-dessus la fenêtre principale du navigateur et empêche toute interaction avec la page sous-jacente jusqu'à ce que la boîte soit fermée. Ce type de boîte de dialogue est souvent utilisé pour obliger l'utilisateur à effectuer une action, comme fournir des informations ou confirmer une décision, avant de continuer la navigation. Les boîtes de dialogue modales sont courantes dans les applications de commerce électronique et autres applications web, par exemple pour demander la confirmation de l'ajout d'un article dans un panier ou pour obtenir des informations d'identification utilisateur.

03-Utiliser Selenium WebDriver Termes

QA

with Marwa

Objectif global du script :

Automatiser l'ouverture, l'interaction et la fermeture d'une boîte de dialogue modale, puis vérifier que le reste de la page devient accessible après.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Démarrer le navigateur et accéder à la page cible
driver = webdriver.Chrome()
driver.get("https://example.com")

# Attendre que la boîte de dialogue modale soit visible
WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.ID, "modal_id")) # Remplace "modal_id"
)

# Interagir avec les éléments dans la boîte modale
confirm_button = driver.find_element(By.ID, "confirm_button") # Remplace "confirm_button"
confirm_button.click()

# Attendre que le modal soit fermé (invisible)
WebDriverWait(driver, 10).until(
    EC.invisibility_of_element_located((By.ID, "modal_id"))
)

# Poursuivre le test après la fermeture du modal
print(" La boîte de dialogue a été traitée avec succès.")

# Fermer le navigateur
driver.quit()
```

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Les invites utilisateur (ou "User prompts") sont des fenêtres modales qui demandent une action de l'utilisateur avant de pouvoir interagir à nouveau avec la page principale du navigateur. En automatisation de tests, ignorer ces invites entraîne souvent des erreurs, car elles bloquent l'exécution du script tant qu'elles ne sont pas traitées.

Les invites utilisateur se présentent sous trois formes principales, définies par le W3C :

1. **Alert** : une boîte de dialogue simple qui affiche un message d'information et un bouton "OK".
2. **Confirm** : une boîte de dialogue qui demande une confirmation avec les options "OK" et "Annuler".
3. **Prompt** : une boîte de dialogue qui permet à l'utilisateur de saisir une entrée textuelle, en plus des options "OK" et "Annuler"

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Manipuler une alerte

Une boîte de dialogue bloquante, générée par le navigateur, qui affiche un message d'information et propose uniquement un bouton "OK". Elle ne demande aucune saisie ni choix utilisateur.

Ce type d'invite est utilisé pour notifier une information importante (ex. : action réussie, message d'erreur). Selon l'ISTQB, les interruptions non gérées comme les alertes bloquent l'exécution automatique, donc doivent être anticipées par les scripts de test.

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

**Exemple Selenium —
Gestion complète de l'alerte avec attente
avant déclenchement**

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Initialisation du navigateur
driver = webdriver.Chrome()
driver.get("https://demoqa.com/alerts") # Site de test avec alerte

# 1. Cliquer sur le bouton qui déclenche une alerte après un délai
driver.find_element(By.ID, "timerAlertButton").click()

# 2. Attendre que l'alerte apparaisse (elle s'affiche après 5 secondes)
WebDriverWait(driver, 10).until(EC.alert_is_present())

# 3. Passer à l'alerte et l'accepter
alert = driver.switch_to.alert
print("Texte de l'alerte :", alert.text)
alert.accept()

print(" Alerte traitée avec succès.")

# Fermer le navigateur
driver.quit()
```

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Manipuler une boîte de confirmation (Confirm)

Une boîte de dialogue JavaScript bloquante qui propose à l'utilisateur de valider ou annuler une action. Elle contient deux boutons : "OK" (valider) et "Annuler" (refuser).

Elle est utilisée lorsqu'une décision explicite est requise, comme la suppression d'un enregistrement ou la confirmation d'un envoi.

Objectif en test automatisé

Une boîte de type confirm() représente une condition d'interruption synchronisée, bloquant la suite du test tant qu'une réponse n'est pas fournie. Ignorer cette étape entraîne l'échec du scénario, car le navigateur empêche toute autre action tant que la boîte n'a pas été validée ou annulée.

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Exemple Selenium – Traitement d'une boîte Confirm

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
```

```
# Initialiser le navigateur
driver = webdriver.Chrome()
driver.get("https://demoqa.com/alerts")
```

```
# 1. Cliquer sur le bouton qui déclenche un Confirm
driver.find_element(By.ID, "confirmButton").click()
```

```
# 2. Attendre l'apparition de la boîte de confirmation
WebDriverWait(driver, 5).until(EC.alert_is_present())
```

```
# 3. Basculer vers l'alerte
confirm = driver.switch_to.alert
print("Texte de confirmation :", confirm.text)
```

```
# 4. Cliquer sur "Annuler" pour refuser
confirm.dismiss() # ou confirm.accept() pour valider
```

```
# 5. Vérifier le message affiché après la réponse
message = driver.find_element(By.ID, "confirmResult").text
print("Résultat :", message)
```

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Manipuler une invite de saisie (Prompt)

Pour automatiser une invite de saisie JavaScript (Prompt) avec Selenium, il faut d'abord basculer le contexte sur l'alerte, puis envoyer la chaîne de caractères souhaitée avant d'accepter ou d'annuler la boîte.

Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Exemple Selenium –

Manipuler une invite de saisie (Prompt)

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# 1. Cliquer sur le bouton qui déclenche l'invite modale (prompt)
driver.find_element(By.ID, "promptButton").click()

# 2. Attendre que le champ de saisie du prompt soit visible
wait = WebDriverWait(driver, 10)
input_prompt = wait.until(EC.visibility_of_element_located((By.ID, "promptInput")))

# 3. Envoyer le texte dans ce champ
input_prompt.send_keys("Mon texte")

# 4. Cliquer sur le bouton OK du prompt (valider)
btn_ok = driver.find_element(By.ID, "promptOkButton")
btn_ok.click()
```

QA

with marwa



Fin du Chapitre 3
Merci pour votre attention !

bettaiebmarwa@gmail.com

👉 On passe au Chapitre 4...