

# INTRODUCTION AU WEB SCRAPING

# DÉFINITION DU WEB SCRAPING

Le Web Scraping est une technique utilisée pour extraire des données de sites Web. Cela se fait par programmation où un script simule la navigation d'un utilisateur sur le Web. Les données extraites peuvent être sauvegardées dans une base de données ou un fichier.

# APPLICATIONS DU WEB SCRAPING

- Collecte de données pour l'analyse de marché
- Surveillance des prix de la concurrence
- Extraction de données pour la génération de leads
- Agrégation de contenu pour les portails d'informations
- Surveillance des changements sur des sites web spécifiques

# DIFFÉRENCES ENTRE WEB SCRAPING ET WEB CRAWLING

## **Web Scraping**

Cible des données spécifiques sur un site

Orienté vers l'extraction de données

Souvent utilisé pour des besoins précis

## **Web Crawling**

Parcourt et indexe le contenu du web

Orienté vers l'analyse de liens et de contenu

Utilisé par les moteurs de recherche

# OUTILS COURANTS POUR LE WEB SCRAPING

- **Selenium**: Automatise les navigateurs web
- **BeautifulSoup**: Parse le HTML et extrait les données
- **Scrapy**: Framework de scraping et de crawling
- **Requests**: Envoie des requêtes HTTP en Python

# LANGAGES DE PROGRAMMATION UTILISÉS EN WEB SCRAPING

- Python: Populaire, avec de nombreuses bibliothèques
- JavaScript: Utile pour les sites dynamiques
- Ruby: Simplicité d'écriture de scripts de scraping
- PHP: Intégration facile avec les sites web

# LIMITES ET CONSIDÉRATIONS LÉGALES DU WEB SCRAPING

- Respecter les fichiers `robots.txt` des sites
- Éviter de surcharger les serveurs avec des requêtes excessives
- Prendre en compte les lois sur la protection des données (ex: RGPD)
- Vérifier les conditions d'utilisation des sites web cibles

# PRÉSENTATION DE SELENIUM

# QU'EST-CE QUE SELENIUM ?

Selenium est un ensemble d'outils open source pour automatiser les navigateurs web. Il permet d'effectuer des actions dans le navigateur comme si un utilisateur humain les effectuait. Selenium prend en charge de nombreux navigateurs et langages de programmation.

# COMPOSANTS DE SELENIUM

- **Selenium IDE** : Extension de navigateur pour l'enregistrement et la lecture de scripts.
- **Selenium WebDriver** : API pour contrôler le navigateur de manière programmatique.
- **Selenium Grid** : Permet d'exécuter des tests sur différentes machines et navigateurs simultanément.

# SELENIUM WEBDRIVER ET SES AVANTAGES

- **Directement contrôle le navigateur** : Interagit avec les pages web de manière réaliste.
- **Support multi-navigateurs** : Compatible avec Chrome, Firefox, Safari, etc.
- **Langages multiples** : Python, Java, C#, Ruby, etc.
- **Communauté active** : Large base d'utilisateurs et de contributeurs.

# UTILISATIONS DE SELENIUM POUR LE WEB SCRAPING

- Automatiser la navigation sur des pages web.
- Extraire des données dynamiques générées par JavaScript.
- Gérer les cookies et les sessions.
- Remplir et soumettre des formulaires.

# COMPARAISON DE SELENIUM AVEC D'AUTRES OUTILS DE WEB SCRAPING

- **Selenium** : Interagit avec le navigateur, idéal pour les sites dynamiques.
- **BeautifulSoup** : Analyse le HTML, efficace pour les sites statiques.
- **Scrapy** : Cadre de scraping puissant, mais moins adapté aux interactions complexes.

# LIMITATIONS DE SELENIUM

- **Performance** : Plus lent que les analyseurs HTML statiques.
- **Complexité** : Courbe d'apprentissage plus raide que les outils dédiés au scraping.
- **Dépendance au navigateur** : Nécessite un navigateur et un pilote correspondant.
- **Ressources** : Consomme plus de ressources système.

# INSTALLATION DE SELENIUM WEBDRIVER

# TÉLÉCHARGEMENT DE SELENIUM WEBDRIVER

Pour utiliser Selenium, il faut d'abord télécharger le WebDriver correspondant à votre navigateur :

- Chrome : ChromeDriver
- Firefox : GeckoDriver
- Edge : EdgeDriver
- Safari : SafariDriver

Visitez les sites officiels pour télécharger la version compatible avec votre navigateur.

# INSTALLATION DE SELENIUM AVEC PIP

Utilisez pip pour installer Selenium :

```
pip install selenium
```

Cela installera la bibliothèque Selenium nécessaire pour interagir avec les navigateurs web.

# VÉRIFICATION DE LA VERSION DE SELENIUM INSTALLÉE

Pour vérifier la version de Selenium installée, exéutez :

```
python -c "import selenium; print(selenium.__version__)"
```

Assurez-vous d'avoir la dernière version stable pour une compatibilité optimale.

# TÉLÉCHARGEMENT DU NAVIGATEUR SPÉCIFIQUE WEBDRIVER

1. Chrome : [ChromeDriver](#)
2. Firefox : [GeckoDriver](#)
3. Edge : [EdgeDriver](#)
4. Safari : Préinstallé avec le système d'exploitation

Choisissez le WebDriver correspondant à la version de votre navigateur.

# INSTALLATION DU WEBDRIVER DANS LE PATH DU SYSTÈME D'EXPLOITATION

- Windows :
  - Copiez le WebDriver dans un dossier de votre choix.
  - Ajoutez le chemin du dossier à la variable d'environnement PATH.
- macOS/Linux :
  - Déplacez le WebDriver dans `/usr/local/bin`.
  - Le système reconnaîtra automatiquement le WebDriver.

# CONFIGURATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

# INSTALLATION DE L'INTERPRÉTEUR PYTHON

- Téléchargez et installez Python depuis le site officiel : [python.org](https://python.org)
- Assurez-vous d'inclure Python dans le PATH lors de l'installation
- Ouvrez un terminal ou une invite de commande et tapez :

```
python --version
```

- La version de Python devrait s'afficher, confirmant l'installation

# CONFIGURATION DE L'ÉDITEUR DE CODE OU DE L'IDE

- Choisissez un éditeur de code ou un IDE supportant Python (ex: VSCode, PyCharm)
- Installez l'éditeur de votre choix et lancez-le
- Configurez les plugins ou extensions Python si nécessaire
- Assurez-vous que l'éditeur reconnaît l'interpréteur Python

# INSTALLATION DES PACKAGES NÉCESSAIRES AVEC PIP

- Ouvrez un terminal ou une invite de commande
- Utilisez pip pour installer Selenium et webdriver-manager :

```
pip install selenium
pip install webdriver-manager
```

- Vérifiez l'installation avec :

```
pip list
```

# CONFIGURATION DU PATH POUR LES WEBDRIVER

- Les WebDriver permettent à Selenium d'interagir avec les navigateurs
- `webdriver-manager` gère automatiquement les WebDriver
- Pour les configurer manuellement, téléchargez le WebDriver correspondant à votre navigateur
- Ajoutez le chemin du WebDriver au PATH de votre système

# VÉRIFICATION DE LA VERSION DU NAVIGATEUR

- Ouvrez le navigateur que vous souhaitez automatiser
- Accédez aux paramètres ou à l'aide pour trouver la version du navigateur
- Assurez-vous que la version du WebDriver correspond à celle du navigateur
- Mettez à jour le navigateur ou le WebDriver si nécessaire

# INSTALLATION DES EXTENSIONS DE NAVIGATEUR POUR LE DÉBOGAGE

- Certaines extensions peuvent aider au développement et au débogage de scripts Selenium
- Exemple : Selenium IDE est disponible pour Firefox et Chrome
- Installez Selenium IDE depuis le magasin d'extensions de votre navigateur
- Utilisez l'extension pour enregistrer et jouer des actions dans le navigateur

# DÉMARRAGE D'UN NAVIGATEUR AVEC SELENIUM

# INSTALLATION DU WEBDRIVER

Pour utiliser Selenium, vous devez installer un WebDriver spécifique à votre navigateur.

- Chrome: chromedriver
- Firefox: geckodriver
- Edge: msedgedriver

Téléchargez le WebDriver correspondant à la version de votre navigateur.

# IMPORTATION DES BIBLIOTHÈQUES SELENIUM

Avant de démarrer le navigateur, importez les bibliothèques nécessaires :

```
from selenium import webdriver
```

Assurez-vous que Selenium est installé via pip :

```
pip install selenium
```

# CRÉATION D'UNE INSTANCE DE NAVIGATEUR

Pour démarrer un navigateur, créez une instance de WebDriver :

```
driver = webdriver.Chrome() # Pour Chrome
driver = webdriver.Firefox() # Pour Firefox
driver = webdriver.Edge() # Pour Edge
```

Cette instance vous permettra de contrôler le navigateur.

# OUVERTURE D'UNE PAGE WEB AVEC GET()

Utilisez la méthode `get()` pour ouvrir une page web :

```
driver.get('http://www.exemple.com')
```

La méthode attend l'URL de la page que vous souhaitez ouvrir.

# INTERACTION AVEC LES ÉLÉMENTS DE LA PAGE WEB

# CLIC SUR LES ÉLÉMENTS

Pour interagir avec les éléments cliquables d'une page web :

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

# Localiser l'élément
element = driver.find_element(By.ID, "mon_id")

# Effectuer un clic
element.click()
```

Utilisation courante :

- Boutons
- Liens
- Cases à cocher

# REmplissage de formulaires

Pour remplir des champs de formulaire :

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

# Localiser le champ
champ = driver.find_element(By.NAME, "nom")

# Saisir du texte
champ.send_keys("Texte exemple")
```

Types de champs :

- Texte
- Email
- Mot de passe

# SÉLECTION DANS DES LISTES DÉROULANTES

Utilisation de `Select` pour interagir avec les listes déroulantes :

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

# Localiser la liste
liste = Select(driver.find_element(By.ID, "ma_liste"))

# Sélectionner par valeur visible
liste.select_by_visible_text("Option 1")
```

Méthodes alternatives :

- `select_by_index(index)`
- `select_by_value(value)`

# GESTION DES ALERTES ET POP-UPS

Pour manipuler les alertes et pop-ups :

```
# Attendre et obtenir l'alerte
alerte = driver.switch_to.alert

# Accepter l'alerte
alerte.accept()

# Ou la rejeter
alerte.dismiss()
```

# NAVIGATION ENTRE LES FENÊTRES ET ONGLETS

Pour changer de fenêtre ou d'onglet :

```
# Obtenir l'identifiant de la fenêtre actuelle
main_window = driver.current_window_handle

# Changer pour une nouvelle fenêtre
driver.switch_to.window("autre_fenetre_id")
```

Retourner à la fenêtre principale :

```
driver.switch_to.window(main_window)
```

# EXÉCUTION DE SCRIPTS JAVASCRIPT

Pour exécuter du JavaScript :

```
# Exécuter un script simple
driver.execute_script("alert('Hello World');")

# Avec des éléments
element = driver.find_element(By.ID, "mon_id")
driver.execute_script("arguments[0].click();", element)
```

Utilisation :

- Manipulation avancée du DOM
- Tests de performance et de charge

# LOCALISATION DES ÉLÉMENTS DU DOM

# INTRODUCTION AU DOCUMENT OBJECT MODEL (DOM)

Le DOM est une interface de programmation pour les documents HTML et XML. Il représente la page de sorte que les programmes peuvent changer la structure, le style et le contenu. Selenium utilise le DOM pour interagir avec les éléments de la page web.

# UTILISATION DE FIND\_ELEMENT\_BY\_ID

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

driver = webdriver.Chrome()
driver.get("http://exemple.com")
element = driver.find_element(By.ID, "identifiant")
```

- Trouve le premier élément avec l'ID spécifié.
- L'ID doit être unique dans le DOM.

# UTILISATION DE FIND\_ELEMENT\_BY\_NAME

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.NAME, "nom")
```

- Sélectionne le premier élément avec l'attribut **name** correspondant.
- Utile pour les formulaires.

# UTILISATION DE FIND\_ELEMENT\_BY\_XPATH

```
element = driver.find_element_by_xpath("//tag[@attribut='valeur']")
```

- Utilise les expressions XPath pour naviguer dans le DOM.
- Très puissant et flexible.

# UTILISATION DE FIND\_ELEMENT\_BY\_LINK\_TEXT

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.LINK_TEXT, "Texte complet du lien")
```

- Trouve un lien (<a> tag) par le texte exact qu'il contient.
- Pratique pour les menus de navigation.

# UTILISATION DE FIND\_ELEMENT\_BY\_PARTIAL\_LINK\_TEXT

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.PARTIAL_LINK_TEXT, "partie du texte")
```

- Similaire à `find_element_by_link_text` mais trouve le lien avec une partie du texte.

# UTILISATION DE FIND\_ELEMENT\_BY\_TAG\_NAME

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.TAG_NAME, "h1")
```

- Trouve le premier élément avec le nom de balise spécifié.
- Utile pour des balises comme <h1>, <p>, etc.

# UTILISATION DE FIND\_ELEMENT\_BY\_CLASS\_NAME

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.CLASS_NAME, "classe")
```

- Sélectionne le premier élément avec la classe CSS spécifiée.
- Les classes peuvent être utilisées sur plusieurs éléments.

# STRATÉGIES DE LOCALISATION D'ÉLÉMENTS

- ID: Rapide, doit être unique.
- Name: Bien pour les formulaires.
- XPath: Très flexible, peut être complexe.
- Link Text: Idéal pour les liens.
- Tag Name: Bon pour les éléments communs.
- Class Name: Utile pour les éléments avec des styles spécifiques.

# DIFFÉRENCES ENTRE FIND\_ELEMENT ET FIND\_ELEMENTS

Méthode	Description
find_element	Retourne le premier élément correspondant trouvé.
find_elements	Retourne une liste de tous les éléments correspondants.

- `find_elements` est utile pour travailler avec des groupes d'éléments.

# UTILISATION DES SÉLECTEURS CSS

# SÉLECTEURS CSS DE BASE

- `*` : sélectionne tous les éléments.
- `element` : sélectionne tous les éléments de type spécifié.
- `.class` : sélectionne tous les éléments avec la classe spécifiée.
- `#id` : sélectionne l'élément avec l'ID spécifié.
- `element, element` : sélectionne tous les éléments qui correspondent à l'un des sélecteurs.

# SÉLECTION PAR ID

- Utilisez # suivi de l'ID de l'élément.
- Exemple :

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.CSS_SELECTOR, '#uniqueId')
```

- Sélectionne l'élément unique avec l'ID `uniqueId`.

# SÉLECTION PAR CLASSE

- Utilisez . suivi du nom de la classe.
- Exemple :

```
from selenium.webdriver.common.by import By  
  
elements = driver.find_elements(By.CSS_SELECTOR, '.className')
```

- Sélectionne tous les éléments avec la classe `className`.

# SÉLECTION PAR ATTRIBUT

- Utilisez `[attribut]` pour un attribut quelconque.
- Utilisez `[attribut='valeur']` pour un attribut avec une valeur spécifique.
- Exemple :

```
from selenium.webdriver.common.by import By
elements = driver.find_elements(By.CSS_SELECTOR, '[type="text"]')
```

- Sélectionne tous les éléments avec l'attribut `type` ayant la valeur `text`.

# SÉLECTION PAR DESCENDANT

- Utilisez `ancestor descendant` pour sélectionner un descendant d'un ancêtre.
- Exemple :

```
from selenium.webdriver.common.by import By  
  
elements = driver.find_elements(By.CSS_SELECTOR, 'div span')
```

- Sélectionne tous les `<span>` qui sont descendants de `<div>`.

# SÉLECTION PAR ENFANT DIRECT

- Utilisez parent > child pour sélectionner un enfant direct.
- Exemple :

```
from selenium.webdriver.common.by import By
elements = driver.find_elements(By.CSS_SELECTOR, 'ul > li')
```

- Sélectionne tous les <li> qui sont enfants directs de <ul>.

# COMBINAISON DE SÉLECTEURS

- Combinez les sélecteurs pour cibler des éléments spécifiques.
- Exemple :

```
from selenium.webdriver.common.by import By  
  
element = driver.find_element(By.CSS_SELECTOR, 'div.note#summary > p:first-of-type')
```

- Sélectionne le premier <p> enfant direct de <div> avec la classe **note** et l'ID **summary**.

# PRIORITÉ ET SPÉCIFICITÉ DES SÉLECTEURS CSS

- ID > Classe > Élément.
- Combinaisons et pseudo-classes augmentent la spécificité.
- L'ordre des règles CSS compte : la dernière règle définie l'emporte.
- `!important` outrepasse les autres déclarations (à utiliser avec prudence).

# UTILISATION DES SÉLECTEURS XPATH

# COMPRENDRE LA SYNTAXE XPATH

XPath est un langage de requête pour sélectionner des nœuds dans un document XML. Il est utilisé dans Selenium pour naviguer dans l'arbre DOM d'une page web. La syntaxe de base est

```
//tagname[@attribut='valeur'].
```

# SÉLECTION D'ÉLÉMENTS PAR TAG

Pour sélectionner tous les éléments d'un certain type, utilisez `//tagname`. Exemple pour sélectionner tous les paragraphes :

```
from selenium.webdriver.common.by import By
paragraphs = driver.find_elements(By.XPATH, "//p")
```

# UTILISATION DES PRÉDICATS

Les prédictats sont utilisés pour filtrer les nœuds par des conditions spécifiques. Ils sont placés entre crochets `[ condition ]`. Exemple pour sélectionner le premier paragraphe :

```
from selenium.webdriver.common.by import By  
  
first_paragraph = driver.find_element(By.XPATH, "//p[1]")
```

# SÉLECTION D'ÉLÉMENTS PAR ATTRIBUT

Pour sélectionner des éléments par leur attribut, utilisez `[@attribut='valeur']`. Exemple pour sélectionner un élément avec l'id "main":

```
from selenium.webdriver.common.by import By
main_element = driver.find_element(By.XPATH, "//*[@id='main']")
```

# NAVIGATION DANS L'ARBRE DOM

- Utilisez / pour sélectionner un élément enfant direct.
- Utilisez // pour sélectionner un élément quel que soit sa profondeur.
- Utilisez .. pour remonter au parent direct.

Exemple pour sélectionner un enfant div de class "container":

```
from selenium.webdriver.common.by import By  
  
container_div = driver.find_element(By.XPATH, "//div[@class='container']")
```

# SÉLECTION D'ÉLÉMENTS PAR POSITION

Utilisez la syntaxe `[n]` pour sélectionner un élément à une position spécifique. Exemple pour sélectionner le troisième élément li :

```
from selenium.webdriver.common.by import By  
  
third_li = driver.find_element(By.XPATH, "(//li)[3]")
```

# UTILISATION DES FONCTIONS XPATH

XPath comprend des fonctions pour des opérations plus complexes. Exemples :

- `text()` pour sélectionner le texte d'un élément.
- `contains()` pour sélectionner des éléments contenant une certaine valeur.

```
from selenium.webdriver.common.by import By
elements_with_text = driver.find_elements(By.XPATH, "//*[contains(text(),'texte')]")
```

# COMBINAISON DE SÉLECTEURS XPATH

Les sélecteurs XPath peuvent être combinés pour des requêtes complexes. Exemple pour sélectionner un lien dans le deuxième paragraphe :

```
from selenium.webdriver.common.by import By
link_in_second_paragraph = driver.find_element(By.XPATH, "//p[2]//a")
```

# RÉCUPÉRATION DE DONNÉES À PARTIR DU WEB

# INSTALLATION DE SELENIUM ET DES DRIVERS NÉCESSAIRES

- Installer Selenium via pip :

```
pip install selenium
```

- Télécharger le driver correspondant à votre navigateur :
  - Chrome : [chromedriver](#)
  - Firefox : [geckodriver](#)
- Mettre le driver dans le PATH ou spécifier son emplacement directement dans le code.

# IMPORTATION DES BIBLIOTHÈQUES SELENIUM EN PYTHON

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

- `webdriver` pour contrôler le navigateur.
- `Keys` pour simuler les frappes au clavier.
- `By` pour spécifier la méthode de sélection des éléments.

# INITIALISATION DU NAVIGATEUR AVEC SELENIUM WEBDRIVER

```
driver = webdriver.Chrome() # Pour Chrome  
# driver = webdriver.Firefox() # Pour Firefox
```

- Crée une instance du navigateur.
- Remplacez **Chrome** par le navigateur de votre choix.

# OUVERTURE D'UNE PAGE WEB AVEC SELENIUM

```
driver.get("http://www.exemple.com")
```

- Ouvre l'URL spécifiée dans le navigateur.
- Attend que la page soit complètement chargée.

# UTILISATION DE LA MÉTHODE FIND\_ELEMENT POUR RÉCUPÉRER DES ÉLÉMENTS

```
element = driver.find_element(By.ID, "identifiant")
# Autres méthodes : By.NAME, By.XPATH, By.LINK_TEXT, etc.
```

- Récupère le premier élément correspondant au sélecteur.

# UTILISATION DE LA MÉTHODE FIND\_ELEMENTS POUR RÉCUPÉRER PLUSIEURS ÉLÉMENTS

```
elements = driver.find_elements(By.CLASS_NAME, "classe")
```

- Récupère une liste d'éléments correspondant au sélecteur.

# EXTRACTION DE TEXTE À PARTIR D'ÉLÉMENTS WEB

```
texte = element.text
```

- Récupère le texte d'un élément web spécifique.

# EXTRACTION D'ATTRIBUTS À PARTIR D'ÉLÉMENTS WEB (COMME HREF POUR LES LIENS)

```
lien = element.get_attribute('href')
```

- Récupère la valeur de l'attribut 'href' d'un élément web.

# UTILISATION DES MÉTHODES GET\_ATTRIBUTE ET TEXT

- `get_attribute('attribut')` pour obtenir la valeur d'un attribut.
- `.text` pour obtenir le texte visible de l'élément.

# ENREGISTREMENT DES DONNÉES RÉCUPÉRÉES

```
with open('donnees.txt', 'w') as fichier:  
    fichier.write(texte + "\n")
```

- Sauvegarde le texte récupéré dans un fichier texte.
- Utilisez le mode d'écriture adapté ('w', 'a', etc.).

# GESTION DE LA NAVIGATION ENTRE LES PAGES

# OUVERTURE D'UNE NOUVELLE PAGE WEB

Pour ouvrir une nouvelle page web avec Selenium :

1. Importez le webdriver depuis `selenium`.
2. Instanciez le navigateur de votre choix.
3. Utilisez la méthode `get` avec l'URL souhaitée.

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('http://www.example.com')
```

# UTILISATION DE LA MÉTHODE GET

La méthode `get` permet de :

- Charger une nouvelle page web dans la fenêtre du navigateur.
- Attendre que la page soit entièrement chargée avant de continuer l'exécution du script.

```
driver.get('http://www.example.com')
```

# NAVIGATION VERS L'ARRIÈRE AVEC BACK

Utilisez la méthode `back` pour :

- Naviguer vers la page précédente de l'historique du navigateur.
- Simuler l'action de cliquer sur le bouton "Précédent".

```
driver.back()
```

# NAVIGATION VERS L'AVANT AVEC FORWARD

Utilisez la méthode **forward** pour :

- Naviguer vers la page suivante de l'historique du navigateur.
- Simuler l'action de cliquer sur le bouton "Suivant".

```
driver.forward()
```

# ACTUALISATION DE LA PAGE AVEC REFRESH

La méthode `refresh` permet de :

- Recharger la page web actuelle.
- Simuler l'action de cliquer sur le bouton "Actualiser" ou "Rafraîchir".

```
driver.refresh()
```

# GESTION DES ONGLETS ET FENÊTRES

Pour gérer les onglets et fenêtres :

- Utilisez `driver.window_handles` pour obtenir la liste des fenêtres/onglets.
- Changez l'onglet/fenêtre actif avec `driver.switch_to.window`.

```
driver.switch_to.window(driver.window_handles[1])
```

# FERMETURE D'UN ONGLET OU D'UNE FENÊTRE AVEC CLOSE ET QUIT

- `close` ferme l'onglet/fenêtre actif.
- `quit` ferme toutes les fenêtres et termine le processus du WebDriver.

```
driver.close() # Ferme l'onglet/fenêtre actif
driver.quit() # Ferme tous les onglets/fenêtres et termine le WebDriver
```

# ATTENTES EXPLICITES ET IMPLICITES

# ATTENTES IMPLICITES (IMPLICIT WAITS)

Les attentes implicites définissent un délai d'attente pour que WebDriver recherche des éléments s'ils ne sont pas immédiatement disponibles.

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.implicitly_wait(10) # Secondes
```

- L'attente est définie une fois et vaut pour toute la session.
- Le driver attendra jusqu'à 10 secondes avant de lever une `NoSuchElementException`.

# ATTENTES EXPLICITES (EXPLICIT WAITS)

Les attentes explicites permettent de définir des conditions spécifiques pour un élément particulier, avec un délai d'attente maximal.

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

wait = WebDriverWait(driver, 10)
element = wait.until(EC.presence_of_element_located((By.ID, 'myElement')))
```

- Utilisées pour des éléments nécessitant plus de temps pour être disponibles.
- Plus flexibles que les attentes implicites.

# WEBDRIVERWAIT

**WebDriverWait** est utilisé pour attendre une certaine condition ou le maximum de temps défini avant de continuer l'exécution du code.

```
from selenium.webdriver.support.ui import WebDriverWait  
  
# Exemple d'utilisation de WebDriverWait  
wait = WebDriverWait(driver, 10) # Attente maximale de 10 secondes
```

- Peut être utilisé avec diverses conditions d'attente.
- Plus précis que les attentes implicites.

# EXPECTED CONDITIONS

Les conditions attendues (**Expected Conditions**) sont des stratégies pour attendre qu'un certain état du DOM soit atteint.

```
from selenium.webdriver.support import expected_conditions as EC  
  
# Exemple d'expected conditions  
visibility = EC.visibility_of_element_located((By.ID, 'myElement'))
```

- Fournit des méthodes pour des conditions courantes (visibilité, clicabilité, etc.).
- Utilisées avec `WebDriverWait`.

# POLLING FREQUENCY

La fréquence de sondage détermine à quelle fréquence WebDriver vérifie la condition donnée dans `WebDriverWait`.

```
from selenium.webdriver.support.ui import WebDriverWait
# Utilisation de la fréquence de sondage
wait = WebDriverWait(driver, 10, poll_frequency=0.5) # Vérifie la condition toutes les 0.5 secondes
```

- La valeur par défaut est de 0.5 secondes.
- Peut être ajustée pour optimiser les performances.

# TIMEOUTS

Les timeouts sont les durées maximales d'attente pour les opérations de chargement des pages, les scripts et les attentes explicites.

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.set_page_load_timeout(10) # Timeout pour le chargement d'une page
driver.set_script_timeout(10) # Timeout pour l'exécution d'un script
```

- Important pour ne pas laisser un test en attente indéfiniment.
- Doit être géré avec soin pour éviter les faux négatifs.

# GESTION DES EXCEPTIONS LIÉES AUX ATTENTES

Lorsque les attentes ne sont pas remplies, Selenium lève des exceptions qu'il faut gérer.

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

try:
    element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, 'myElement')))
except TimeoutException:
    print("Élément non trouvé dans le temps imparti.")
```

- Les exceptions courantes incluent `TimeoutException` et `NoElementException`.
- La gestion des exceptions évite les interruptions brutales des scripts.

# GESTION DES FENÊTRES ET DES ONGLETS

# OUVERTURE D'UNE NOUVELLE FENÊTRE OU ONGLET

Pour ouvrir une nouvelle fenêtre ou un nouvel onglet dans Selenium :

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.execute_script("window.open('about:blank', '_blank');")
```

- `window.open()` ouvre une nouvelle fenêtre ou onglet.
- '`about:blank`' charge une page vide.
- '`_blank`' spécifie d'ouvrir le lien dans un nouvel onglet ou fenêtre.

# CHANGEMENT DE CONTEXTE ENTRE FENÊTRES/ONGLETS

Pour changer de contexte entre les fenêtres ou onglets :

```
# Obtenez la liste des identifiants de fenêtre
window_handles = driver.window_handles

# Passez à la nouvelle fenêtre
driver.switch_to.window(window_handles[1])
```

- `window_handles` est une liste des identifiants.
- `switch_to.window()` permet de changer de fenêtre/onglet.

# FERMETURE D'UNE FENÊTRE OU ONGLET

Pour fermer une fenêtre ou un onglet :

```
# Ferme l'onglet ou la fenêtre actuelle
driver.close()

# Revenir à la fenêtre principale (si nécessaire)
driver.switch_to.window(driver.window_handles[0])
```

- **close()** ferme l'onglet/fenêtre actif.
- Utilisez **switch\_to.window()** pour revenir à une autre fenêtre.

# RÉCUPÉRATION DES IDENTIFIANTS DE FENÊTRES/ONGLETS

Pour récupérer les identifiants des fenêtres/onglets :

```
# Liste des identifiants de toutes les fenêtres/onglets ouverts
window_handles = driver.window_handles

# Exemple d'affichage
for handle in window_handles:
    print(handle)
```

- `window_handles` retourne une liste des identifiants.
- Chaque fenêtre/onglet ouvert a un identifiant unique.

# CAPTURE D'ÉCRAN DES PAGES WEB

# CONFIGURATION DE L'ENVIRONNEMENT POUR LES CAPTURES D'ÉCRAN

- Installer le package Selenium :

```
pip install selenium
```

- Télécharger le pilote (driver) correspondant à votre navigateur.
- Ajouter le chemin du pilote à la variable d'environnement PATH ou spécifier le chemin directement dans le script.

# UTILISATION DE LA MÉTHODE GET\_SCREENSHOT\_AS\_FILE

- Prendre une capture d'écran de la page entière :

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://www.exemple.com")
driver.get_screenshot_as_file('page_entiere.png')
```

# DÉFINITION DU CHEMIN DE SAUVEGARDE DES CAPTURES D'ÉCRAN

- Spécifier le chemin absolu ou relatif :

```
chemin = '/chemin/vers/dossier/capture.png'  
driver.get_screenshot_as_file(chemin)
```

- Utiliser des noms de fichier dynamiques pour éviter les écrasements.

# GESTION DES EXCEPTIONS LORS DE LA CAPTURE D'ÉCRAN

- Utiliser try-except pour gérer les erreurs :

```
try:  
    driver.get_screenshot_as_file('page_entiere.pri  
except Exception as e:  
    print("Erreur lors de la capture d'écran : ", e)
```

# CAPTURE D'ÉCRAN D'ÉLÉMENTS SPÉCIFIQUES AVEC ELEMENT.SCREENSHOT

- Cibler un élément spécifique pour la capture :

```
element = driver.find_element_by_id('element_id')
element.screenshot('element.png')
```

- Compatible avec les éléments individuels, pas seulement avec la page entière.

# AUTOMATISATION DES SAISIES DE FORMULAIRE

# IDENTIFICATION DES ÉLÉMENTS DE FORMULAIRE

Pour interagir avec les formulaires, identifiez les éléments :

- Utilisez les attributs `id`, `name`, `class`, `xpath`, `css_selector`.
- Exemple avec `find_element_by_name` :

```
from selenium import webdriver
driver = webdriver.Chrome()
element = driver.find_element_by_name('username')
```

# REmplissage des champs de texte

Pour remplir un champ de texte :

- Utilisez la méthode `send_keys` sur l'élément.
- Exemple de remplissage :

```
element.send_keys("VotreTexteIci")
```

# SÉLECTION DANS LES LISTES DÉROULANTES

Pour sélectionner une option :

- Utilisez la classe `Select` de `selenium.webdriver.support.ui`.
- Exemple de sélection :

```
from selenium.webdriver.support.ui import Select
select_element = Select(driver.find_element_by_name('dropdown'))
select_element.select_by_visible_text('OptionTexte')
```

# INTERACTION AVEC LES BOUTONS RADIO

Pour interagir avec les boutons radio :

- Localisez l'élément et utilisez `click` pour le sélectionner.
- Exemple :

```
radio_button = driver.find_element_by_id('radioButtonId')
radio_button.click()
```

# COCHER DES CASES À COCHER

Pour cocher une case :

- Trouvez l'élément et utilisez `click` s'il n'est pas déjà coché.
- Exemple :

```
checkbox = driver.find_element_by_id('checkboxId')
if not checkbox.is_selected():
    checkbox.click()
```

# SOUMISSION DU FORMULAIRE

Pour soumettre un formulaire :

- Utilisez `submit` sur un élément du formulaire.
- Exemple :

```
element.submit()
```

# GESTION DES ALERTES ET DES POP-UPS

Pour gérer les alertes :

- Utilisez `switch_to.alert` puis `accept` ou `dismiss`.
- Exemple :

```
alert = driver.switch_to.alert
alert.accept() # ou alert.dismiss()
```

# ATTENTE EXPLICITE OU IMPLICITE DES ÉLÉMENTS DE FORMULAIRE

Pour attendre un élément :

- Attente implicite : `driver.implicitly_wait(10)` pour attendre 10 secondes.
- Attente explicite : Utilisez `WebDriverWait` avec `expected_conditions`.
- Exemple d'attente explicite :

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

wait = WebDriverWait(driver, 10)
element = wait.until(EC.presence_of_element_located((By.ID, 'elementId')))
```

# GESTION DES COOKIES

# COMPRENDRE LES COOKIES WEB

- Les cookies sont de petits fichiers stockés par le navigateur.
- Ils contiennent des données spécifiques à un utilisateur et un site web.
- Utilisés pour conserver l'état de la session, les préférences, et suivre l'activité.
- Structure clé-valeur; chaque cookie a un nom et une valeur associée.

# ACCÉDER AUX COOKIES AVEC SELENIUM

- Selenium permet d'interagir avec les cookies via la classe WebDriver.
- Utilisation de `driver.get_cookies()` pour récupérer tous les cookies.
- Accès à un cookie spécifique avec `driver.get_cookie('nom')`.

# AJOUTER DES COOKIES

- Ajout de cookies avec `driver.add_cookie(cookie_dict)`.
- `cookie_dict` est un dictionnaire avec des clés comme `name`, `value`, `path`, etc.
- Exemple :

```
driver.add_cookie({'name': 'foo', 'value': 'bar'})
```

# SUPPRIMER DES COOKIES

- Suppression d'un cookie spécifique avec `driver.delete_cookie('nom')`.
- Pour supprimer tous les cookies, utiliser `driver.delete_all_cookies()`.

# MODIFIER DES COOKIES

- Selenium ne fournit pas de méthode directe pour modifier les cookies.
- Supprimer le cookie existant, puis ajouter un nouveau cookie avec les modifications.
- Exemple :

```
driver.delete_cookie('foo')
driver.add_cookie({'name': 'foo', 'value': 'new_value'})
```

# EXTRAIRE DES COOKIES

- Pour sauvegarder les cookies, extraire avec `get_cookies()` et les stocker.
- Exemple de sauvegarde dans une variable :

```
saved_cookies = driver.get_cookies()
```

# GESTION DE LA SESSION AVEC LES COOKIES

- Les cookies de session sont essentiels pour maintenir une session utilisateur active.
- En sauvegardant et en chargeant ces cookies, on peut persister l'état de la session.
- Exemple de chargement des cookies sauvegardés :

```
for cookie in saved_cookies:  
    driver.add_cookie(cookie)
```

# GESTION DES EXCEPTIONS ET DES ERREURS

# INTRODUCTION AUX EXCEPTIONS

Les exceptions sont des événements qui interrompent le flux normal d'un programme. Elles sont générées lorsqu'une erreur se produit. En Python, les erreurs et les exceptions sont gérées pour éviter l'arrêt du programme. Selenium utilise les exceptions pour gérer les erreurs lors du web scraping.

# TRY ET EXCEPT EN PYTHON

La structure `try` et `except` permet de gérer les exceptions :

```
try:  
    # Code à essayer  
except Exception as e:  
    # Code à exécuter en cas d'erreur
```

Elle permet d'exécuter du code en toute sécurité.

# GESTION DES EXCEPTIONS SPÉCIFIQUES

En Python, on peut cibler des exceptions spécifiques :

```
try:  
    # Code  
except ValueError:  
    # Gestion d'une ValueError  
except TypeError:  
    # Gestion d'une TypeError
```

Cela permet une meilleure précision dans le traitement des erreurs.

# UTILISATION DE FINALLY

Le bloc `finally` s'exécute toujours, erreur ou non :

```
try:  
    # Code  
except Exception as e:  
    # Gestion de l'exception  
finally:  
    # Code exécuté après le bloc try/except
```

Idéal pour des opérations de nettoyage.

# LEVÉE D'EXCEPTIONS PERSONNALISÉES

Pour générer une exception personnalisée :

```
if condition_non_souhaitee:  
    raise Exception("Message d'erreur personnalisé")
```

Permet d'alerter sur des conditions spécifiques.

# GESTION DU TIMEOUT

```
from selenium.webdriver.support.ui import WebDriverWait
try:
    element = WebDriverWait(driver, 10).until(condition)
except TimeoutException:
    # Gestion du timeout
```

Permet d'attendre un élément un certain temps.

# GESTION DES ERREURS DE CONNEXION

```
from selenium.common.exceptions import WebDriverException

try:
    driver.get(url)
except WebDriverException:
    # Gestion des erreurs de connexion
```

Important pour gérer les problèmes de réseau.

# ERREURS DE NAVIGATION (PAGE NON TROUVÉE, ERREUR 404)

```
try:  
    driver.get(url)  
    # Vérifier le statut de la page  
except Error404Exception:  
    # Gestion de l'erreur 404
```

Permet de gérer les liens brisés ou les pages supprimées.

# ERREURS DE SÉLECTION D'ÉLÉMENTS (ÉLÉMENTS NON TROUVÉS)

```
from selenium.common.exceptions import NoSuchElementException

try:
    element = driver.find_element_by_id("id_inexistant")
except NoSuchElementException:
    # Gestion lorsque l'élément n'est pas trouvé
```

Crucial pour éviter les arrêts lors de sélections d'éléments.

# LOGGING DES ERREURS

Enregistrer les erreurs dans un fichier de log :

```
import logging

logging.basicConfig(filename='errors.log', level=logging.ERROR)

try:
    # Code pouvant générer une erreur
except Exception as e:
    logging.error("Erreur rencontrée : %s", e)
```

Permet de suivre les erreurs survenues lors du scraping.

# BONNES PRATIQUES DE WEB SCRAPING

# IDENTIFICATION DES DONNÉES À EXTRAIRE

- Déterminer l'information nécessaire
- Analyser la structure de la page web
- Repérer les sélecteurs CSS ou XPATH
- Utiliser l'inspecteur du navigateur pour identifier les données
- Planifier la structure de sortie (CSV, JSON, base de données)

# LIMITATION DE LA FRÉQUENCE DES REQUÊTES

- Respecter la charge du serveur web
- Utiliser `time.sleep()` pour espacer les requêtes
- Se conformer aux directives du fichier `robots.txt`
- Envisager l'utilisation de proxies pour réduire la fréquence
- Éviter le scraping intensif qui peut mener à un blocage IP

# UTILISATION DES EN-TÊTES HTTP APPROPRIÉS

- Définir un **User-Agent** réaliste dans les en-têtes
- Inclure d'autres en-têtes standards (Accept-Language, Accept)
- Faire en sorte que les requêtes semblent provenir d'un navigateur réel
- Éviter de se faire bloquer par des mesures anti-scraping

# GESTION DES SESSIONS ET DES COOKIES

- Utiliser **Session de requests** ou **Options** de Selenium pour gérer les cookies
- Maintenir la session pour naviguer comme un utilisateur authentique
- Gérer les cookies pour accéder à des pages nécessitant une connexion
- Sauvegarder et restaurer les sessions si nécessaire

# STOCKAGE ET TRAITEMENT ÉTHIQUE DES DONNÉES

- Stocker les données de manière sécurisée
- Ne pas partager les données sensibles sans autorisation
- Anonymiser les données personnelles
- Respecter les lois sur la protection des données (GDPR, CCPA, etc.)
- Utiliser les données uniquement pour les usages prévus et éthiques

# DOCUMENTATION ET MAINTIEN DU CODE DE SCRAPING

- Commenter le code pour expliquer les sélecteurs et la logique
- Documenter les étapes du scraping et les structures de données
- Mettre à jour le code en fonction des changements de la page web
- Utiliser un système de contrôle de version (Git) pour suivre les modifications
- Prévoir des tests pour vérifier la validité des données extraites

# RESPECT DES RÈGLES DU ROBOTS.TXT

# COMPRÉHENSION DU FICHIER ROBOTS.TXT

Le fichier **robots .txt** est un fichier texte placé à la racine d'un site web. Il indique aux robots d'indexation (web crawlers) les sections autorisées ou interdites du site. Le format du fichier est standardisé et peut être interprété par les robots respectueux des règles.

# UTILISATION DE LA BIBLIOTHÈQUE ROBOTPARSER DE PYTHON

```
import urllib.robotparser  
  
rp = urllib.robotparser.RobotFileParser()  
rp.set_url("http://www.exemple.com/robots.txt")  
rp.read()
```

Utilisation de `robotparser` pour lire et analyser `robots.txt`.

# ANALYSE DES DIRECTIVES "USER-AGENT"

```
user_agent = 'MonBot'  
rp.can_fetch(user_agent, "http://www.exemple.com/chemin")
```

User-agent spécifie à quel robot s'appliquent les règles. can\_fetch() vérifie si l'accès est autorisé pour le User-agent.

# IDENTIFICATION DES SECTIONS "DISALLOW" ET "ALLOW"

- Disallow: Chemins d'accès interdits aux robots.
- Allow: Chemins d'accès autorisés, même si une règle Disallow générale est définie.

Analyser ces sections pour déterminer les restrictions d'accès.

# RESPECT DES DÉLAIS DE CRAWL SPÉCIFIÉS DANS "CRAWL-DELAY"

```
crawl_delay = rp.crawl_delay(user_agent)
```

**Crawl-delay** définit le délai en secondes entre deux requêtes successives au serveur. Respecter ce délai pour éviter de surcharger le serveur.

# VÉRIFICATION DE LA PERMISSION AVANT DE SCRAPER

Avant de commencer le scraping, vérifiez toujours :

1. Les directives `Disallow` et `Allow` pour votre User-agent.
2. Le délai spécifié par `Crawl-delay`.
3. Les éventuelles conditions supplémentaires spécifiées par le site.

Respecter `robots.txt` est crucial pour un scraping éthique et légal.

# LIMITATIONS ET CONSIDÉRATIONS LÉGALES DU WEB SCRAPING

# DISTINCTION ENTRE WEB SCRAPING ET WEB CRAWLING

- **Web Scraping** : Extraction de données spécifiques d'une page web.
- **Web Crawling** : Exploration automatisée de pages web pour indexer le contenu.
- **Objectif différent** : Scraping pour collecte de données, Crawling pour cartographie du web.

# LIMITES TECHNIQUES DE SELENIUM

- **Navigateur requis** : Selenium nécessite un navigateur pour fonctionner.
- **Moins efficace** : Plus lent que les outils HTTP spécifiques au scraping.
- **Gestion des ressources** : Consommation plus élevée de ressources système.

# DÉPENDANCE À LA STRUCTURE DU SITE WEB

- **Sélecteurs spécifiques** : Selenium dépend des sélecteurs CSS/XPath du site.
- **Fragilité** : Les changements dans le site peuvent casser le script de scraping.
- **Maintenance** : Nécessite une mise à jour régulière des sélecteurs.

# GESTION DES SITES DYNAMIQUES AVEC JAVASCRIPT

- **Interactivité** : Selenium peut interagir avec des éléments dynamiques.
- **Chargement asynchrone** : Capable d'attendre que le contenu soit chargé via JavaScript.
- **Complexité** : Requiert une compréhension approfondie du comportement du site.

# PROBLÈMES DE PERFORMANCE ET D'EFFICACITÉ

- **Vitesse** : Plus lent que les requêtes directes sans navigateur.
- **Charge serveur** : Plus de ressources serveur utilisées pour charger des pages complètes.
- **Optimisation** : Nécessite une optimisation pour réduire l'impact sur la performance.

# RISQUE DE BLOCAGE PAR LES SITES WEB

- **Détection** : Les comportements automatisés peuvent être détectés et bloqués.
- **Limitation de requêtes** : Les sites peuvent limiter le nombre de requêtes par IP.
- **Contournement** : Utilisation de proxies et rotation d'IP pour réduire le risque.

# CONSIDÉRATIONS ÉTHIQUES DU WEB SCRAPING

- **Consentement** : Respecter les données et le consentement des utilisateurs.
- **Impact** : Évaluer l'impact sur les performances du site cible.
- **Transparence** : Être transparent sur l'utilisation des données collectées.

# IMPLICATIONS DU DROIT D'AUTEUR

- **Propriété** : Les données extraites peuvent être soumises au droit d'auteur.
- **Utilisation** : Éviter l'utilisation commerciale sans autorisation.
- **Attribution** : Citer la source lorsque les données sont utilisées publiquement.

# RESPECT DE LA VIE PRIVÉE ET DES DONNÉES PERSONNELLES

- **Données personnelles** : Éviter de collecter sans consentement.
- **Anonymisation** : Anonymiser les données sensibles lorsque possible.
- **Sécurité** : Assurer la sécurité des données collectées.

# CONFORMITÉ AU RÈGLEMENT GÉNÉRAL SUR LA PROTECTION DES DONNÉES (RGPD)

- **Consentement** : Obtenir le consentement pour le traitement des données personnelles.
- **Droits** : Respecter les droits des utilisateurs (accès, rectification, suppression).
- **Transparence** : Informer clairement sur l'usage des données collectées.

# LÉGISLATION SPÉCIFIQUE PAR PAYS OU RÉGION

- **Variabilité** : Les lois varient considérablement d'un pays à l'autre.
- **Recherche** : Se renseigner sur la législation locale avant de scraper.
- **Conformité** : S'assurer de respecter les lois spécifiques à chaque région.

# CONSÉQUENCES JURIDIQUES DU NON-RESPECT DES CONDITIONS D'UTILISATION

- **Violations** : Non-respect peut entraîner des poursuites légales.
- **Conditions d'utilisation** : Lire et respecter les termes du site cible.
- **Prudence** : Agir avec prudence pour éviter les conséquences juridiques.