

# Selenium

---

## 1.1. Introduction

Les **bindings Python de Selenium** fournissent une API simple pour écrire des tests fonctionnels ou d'acceptation à l'aide de **Selenium WebDriver**. Grâce à l'**API Selenium Python**, vous pouvez accéder à toutes les fonctionnalités de **Selenium WebDriver** de manière intuitive.

Les **bindings Python de Selenium** offrent une API pratique pour accéder aux **Selenium WebDrivers** tels que **Firefox**, **le Chrome**, **Remote**, etc. Les versions de **Python** actuellement prises en charge sont **3.5 et supérieures**.

---

## 1.2. Installer les bindings Python pour Selenium

Utilisez `pip` pour installer le paquet **selenium**. **Python 3** inclut `pip` dans sa bibliothèque standard. Avec `pip`, vous pouvez installer **selenium** de la manière suivante :

```
pip install selenium
```

Vous pouvez envisager d'utiliser **virtualenv** pour créer des environnements **Python** isolés. **Python 3** propose également **venv**, qui est très similaire à **virtualenv**.

Vous pouvez aussi télécharger les **bindings Python de Selenium** depuis la page **PyPI** du paquet **selenium** et les installer manuellement.

---

## 2. Premiers Pas

### 2.1. Utilisation Simple

Si vous avez installé les **bindings Python de Selenium**, vous pouvez commencer à les utiliser dans un script Python comme ceci :

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

driver = webdriver.Firefox()
driver.get("http://www.python.org")
assert "Python" in driver.title
elem = driver.find_element(By.NAME, "q")
elem.clear()
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
```

```
assert "No results found." not in driver.page_source
driver.close()
```

Le script ci-dessus peut être enregistré dans un fichier (par exemple : `python_org_search.py`), puis exécuté comme ceci :

```
python python_org_search.py
```

Le **Python** que vous utilisez doit avoir le module **selenium** installé.

## 2.2. Explication de l'exemple

Le module `selenium.webdriver` fournit toutes les implémentations de **WebDriver**. Actuellement, les implémentations prises en charge sont : **Firefox**, **Chrome**, **IE** et **Remote**. La classe `Keys` permet d'utiliser des touches du clavier comme `RETURN`, `F1`, `ALT`, etc. La classe `By` est utilisée pour localiser les éléments dans un document HTML.

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

Ensuite, une instance de **Firefox WebDriver** est créée :

```
driver = webdriver.Firefox()
```

La méthode `driver.get` permet de naviguer vers une page en utilisant son URL. **WebDriver** attendra que la page soit entièrement chargée (c'est-à-dire que l'événement `onload` soit déclenché) avant de rendre la main au script. Attention : si votre page utilise beaucoup d'**AJAX**, **WebDriver** peut ne pas détecter correctement la fin du chargement :

```
driver.get("http://www.python.org")
```

La ligne suivante est une assertion qui vérifie que le titre de la page contient bien le mot "Python" :

```
assert "Python" in driver.title
```

**WebDriver** propose plusieurs manières de trouver des éléments grâce à la méthode `find_element`. Par exemple, un champ texte peut être localisé par son attribut `name` :

```
elem = driver.find_element(By.NAME, "q")
```

Ensuite, on envoie du texte à ce champ, comme si on tapait sur un clavier. Les touches spéciales (comme `RETURN`) peuvent être envoyées via la classe `Keys`. Par précaution, on commence par effacer tout texte pré-rempli dans le champ (comme "Search") :

```
elem.clear()
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
```

Une fois la page soumise, vous devriez obtenir des résultats si des correspondances sont trouvées. Pour s'en assurer, on ajoute une assertion :

```
assert "No results found." not in driver.page_source
```

Enfin, on ferme la fenêtre du navigateur. Vous pouvez aussi utiliser `quit()` à la place de `close()`. La méthode `quit()` quitte entièrement le navigateur, tandis que `close()` ferme uniquement l'onglet actif (et quitte aussi le navigateur s'il n'y avait qu'un seul onglet ouvert) :

```
driver.close()
```

## 3. Navigation

### 3.1. Naviguer vers une page

La première chose que vous voudrez faire avec **WebDriver**, c'est naviguer vers un lien. Pour cela, on utilise généralement la méthode `get` :

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

**WebDriver** attendra que la page soit complètement chargée (c'est-à-dire que l'événement `onload` soit déclenché) avant de rendre la main à votre script ou test. Attention : si votre page utilise beaucoup d'**AJAX** au chargement, **WebDriver** pourrait ne pas détecter que la page est entièrement chargée. Dans ce cas, vous pouvez utiliser des **waits**.

### 3.2. Interagir avec la page

Naviguer ne suffit pas, on souhaite aussi **interagir avec les éléments HTML** d'une page. Pour cela, il faut d'abord localiser un élément. Voici un exemple :

```
<input type="text" name="passwd" id="passwd-id" />
```

On peut le trouver de différentes manières :

```
element = driver.find_element(By.ID, "passwd-id")
element = driver.find_element(By.NAME, "passwd")
element = driver.find_element(By.XPATH, "//*[@id='passwd-id']")
element = driver.find_element(By.CSS_SELECTOR, "input#passwd-id")
```

On peut aussi chercher un lien par son texte, mais attention : le texte doit correspondre **exactement**. De plus, l'usage d'**XPATH** peut être délicat : si plusieurs éléments correspondent,

seul le premier est retourné. Si aucun élément n'est trouvé, une exception **NoSuchElementException** sera levée.

**WebDriver** utilise une API orientée objet : tous les types d'éléments sont représentés avec la même interface. Cela signifie que certains appels ne seront pas pertinents selon le type d'élément, et lèveront des exceptions si utilisés de manière incorrecte.

Vous pouvez ensuite interagir avec l'élément, par exemple, pour saisir du texte :

```
element.send_keys("some text")
element.send_keys(" and some", Keys.ARROW_DOWN)
```

⚠ Taper dans un champ ne l'efface pas automatiquement. Ce que vous tapez est **ajouté**. Pour vider un champ avant :

```
element.clear()
```

### 3.3. Remplir des formulaires

On a vu comment saisir du texte. Pour les éléments comme les listes déroulantes (**SELECT**), voici comment procéder :

```
element = driver.find_element(By.XPATH, "///select[@name='name']")
all_options = element.find_elements(By.TAG_NAME, "option")
for option in all_options:
    print("Value is: %s" % option.get_attribute("value"))
    option.click()
```

Mais ce n'est pas très pratique. **WebDriver** fournit une classe `Select` pour cela :

```
from selenium.webdriver.support.ui import Select
select = Select(driver.find_element(By.NAME, 'name'))

select.select_by_index(index)
select.select_by_visible_text("text")
select.select_by_value(value)
```

Pour désélectionner toutes les options sélectionnées :

```
select = Select(driver.find_element(By.ID, 'id'))
select.deselect_all()
```

Récupérer les options sélectionnées par défaut :

```
all_selected_options = select.all_selected_options
```

Obtenir toutes les options disponibles :

```
options = select.options
```

Soumettre le formulaire en cliquant sur un bouton `submit` :

```
driver.find_element(By.ID, "submit").click()
```

Ou via la méthode `submit()` :

```
element.submit()
```

### 3.4. Drag and Drop

Pour déplacer un élément :

```
element = driver.find_element(By.NAME, "source")
target = driver.find_element(By.NAME, "target")

from selenium.webdriver import ActionChains
action_chains = ActionChains(driver)
action_chains.drag_and_drop(element, target).perform()
```

### 3.5. Changer de fenêtres et de frames

Pour basculer entre différentes fenêtres :

```
driver.switch_to.window("windowName")
```

Vous pouvez aussi utiliser les **window handles** :

```
for handle in driver.window_handles:
    driver.switch_to.window(handle)
```

Pour changer de **frame** ou **iframe** :

```
driver.switch_to.frame("frameName")
driver.switch_to.frame("frameName.0.child") # frame enfant
```

Revenir au contenu principal :

```
driver.switch_to.default_content()
```

### 3.6. Boîtes de dialogue (popup)

**Selenium WebDriver** prend en charge les popups de type `alert`, `confirm`, `prompt`. Une fois le popup déclenché :

```
alert = driver.switch_to.alert
```

Vous pouvez ensuite **accepter**, **rejeter**, **lire** le texte ou **saisir** une valeur.

### 3.7. Navigation : historique et position

Pour naviguer :

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

Aller en avant / en arrière dans l'historique :

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

⚠ Ce comportement dépend du navigateur et du **driver** utilisé.

### 3.8. Cookies

Pour travailler avec les **cookies**, il faut d'abord être sur le bon domaine :

```
driver.get("http://www.example.com")  
  
cookie = {'name' : 'foo', 'value' : 'bar'}  
driver.add_cookie(cookie)  
  
driver.get_cookies()
```

## 4. Localisation des éléments

Il existe différentes stratégies pour localiser des éléments sur une page. Vous pouvez utiliser celle qui convient le mieux à votre cas. Selenium fournit la méthode suivante pour localiser des éléments sur une page :

```
find_element
```

Pour trouver plusieurs éléments (ces méthodes renverront une liste) :

```
find_elements
```

Exemple d'utilisation :

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

driver.find_element(By.XPATH, '//button[text()="Some text"]')
driver.find_elements(By.XPATH, '//button')
```

Les attributs disponibles pour la classe `By` permettent de localiser des éléments sur une page. Voici les attributs disponibles :

- ID = "id"
- NAME = "name"
- XPATH = "xpath"
- LINK\_TEXT = "link text"
- PARTIAL\_LINK\_TEXT = "partial link text"
- TAG\_NAME = "tag name"
- CLASS\_NAME = "class name"
- CSS\_SELECTOR = "css selector"

Voici comment ces attributs sont utilisés pour localiser des éléments :

```
find_element(By.ID, "id")
find_element(By.NAME, "name")
find_element(By.XPATH, "xpath")
find_element(By.LINK_TEXT, "link text")
find_element(By.PARTIAL_LINK_TEXT, "partial link text")
find_element(By.TAG_NAME, "tag name")
find_element(By.CLASS_NAME, "class name")
find_element(By.CSS_SELECTOR, "css selector")
```

Pour localiser plusieurs éléments ayant le même attribut, utilisez `find_elements` à la place de `find_element`.

## 4.1. Localisation par ID

Utilisez cette stratégie lorsque vous connaissez l'attribut `id` d'un élément. Le premier élément avec un `id` correspondant sera retourné. Sinon, une exception `NoSuchElementException` sera levée.

Exemple :

```
<form id="loginForm">
  <input name="username" type="text" />
  <input name="password" type="password" />
  <input name="continue" type="submit" value="Login" />
</form>
```

Pour localiser le formulaire :

```
login_form = driver.find_element(By.ID, 'loginForm')
```

## 4.2. Localisation par nom ( **name** )

Même principe, mais basé sur l'attribut **name** .

```
username = driver.find_element(By.NAME, 'username')
password = driver.find_element(By.NAME, 'password')
continue_button = driver.find_element(By.NAME, 'continue') # Le bouton "Login"
```

## 4.3. Localisation par XPath

XPath est un langage puissant pour localiser des nœuds dans un document XML/HTML.

Exemples :

```
login_form = driver.find_element(By.XPATH, "/html/body/form[1]")
login_form = driver.find_element(By.XPATH, "//form[@id='loginForm']")
username = driver.find_element(By.XPATH, "//input[@name='username']")
clear_button = driver.find_element(By.XPATH, "//input[@name='continue'][@type='button']")
```

## 4.4. Localisation par texte de lien

Utilisez cette méthode si vous connaissez le texte exact d'un lien :

```
<a href="continue.html">Continue</a>
<a href="cancel.html">Cancel</a>
```

```
continue_link = driver.find_element(By.LINK_TEXT, 'Continue')
continue_link = driver.find_element(By.PARTIAL_LINK_TEXT, 'Conti')
```

## 4.5. Localisation par nom de balise ( **tag name** )

```
<h1>Welcome</h1>
```

```
heading1 = driver.find_element(By.TAG_NAME, 'h1')
```

## 4.6. Localisation par nom de classe ( **class name** )



```
<p class="content">Site content goes here.</p>
```

```
content = driver.find_element(By.CLASS_NAME, 'content')
```

## 4.7. Localisation par sélecteur CSS

```
<p class="content">Site content goes here.</p>
```

```
content = driver.find_element(By.CSS_SELECTOR, 'p.content')
```

## 5. Waits

De nos jours, la plupart des applications web utilisent des techniques AJAX. Lorsque la page est chargée par le navigateur, les éléments de cette page peuvent apparaître à des intervalles différents. Cela rend la localisation des éléments difficile : si un élément n'est pas encore présent dans le DOM, une fonction de localisation lèvera une exception `ElementNotVisibleException`.

Pour résoudre ce problème, on utilise des **attentes (waits)**. Elles permettent d'introduire un délai entre les actions effectuées – principalement lors de la localisation ou de l'interaction avec un élément.

Selenium WebDriver fournit deux types d'attentes :

- **Attente implicite (implicit wait)**
- **Attente explicite (explicit wait)**

### 5.1. Attentes explicites

Une attente explicite vous permet de définir une condition à attendre avant de poursuivre l'exécution du code. Le cas extrême est `time.sleep()`, qui impose un délai fixe. Mais Selenium fournit des méthodes plus souples pour n'attendre que le temps nécessaire.

Un exemple d'utilisation avec `WebDriverWait` et `ExpectedConditions` :

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

driver = webdriver.Firefox()
driver.get("http://somedomain/url_that_delays_loading")

try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "myDynamicElement"))
```

```
)  
finally:  
    driver.quit()
```

Dans cet exemple, Selenium attendra **au maximum 10 secondes** pour que l'élément correspondant soit trouvé. Si aucun élément n'est trouvé, une `TimeoutException` sera levée.

Par défaut, `WebDriverWait` vérifie la condition toutes les 500 millisecondes.

## Conditions prédéfinies (Expected Conditions)

Voici une liste de conditions souvent utilisées dans les tests automatisés :

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

Exemple :

```
from selenium.webdriver.support import expected_conditions as EC  
  
wait = WebDriverWait(driver, 10)  
element = wait.until(EC.element_to_be_clickable((By.ID, 'someid')))
```

Le module `expected_conditions` contient toutes ces méthodes prêtes à l'emploi.

## Alert & Windows

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>alert_is_present()</code>	None	When a JavaScript alert is expected	Will throw if alert isn't present after timeout
<code>new_window_is_opened(current_handles)</code>	<code>list</code> of handles	Waiting for a popup or tab	Requires pre-captured list of window handles
<code>number_of_windows_to_be(num)</code>	<code>int</code>	Waiting for tab count to change	Use carefully in async popup flows

## URL & Title

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>title_is(title)</code>	<code>str</code>	Verifying exact page title	Case-sensitive match
<code>title_contains(substring)</code>	<code>str</code>	Checking part of the title	Case-sensitive
<code>url_to_be(url)</code>	<code>str</code>	Waiting for final redirect	Must match exactly
<code>url_contains(substring)</code>	<code>str</code>	Checking for a keyword in URL	Case-sensitive
<code>url_matches(pattern)</code>	<code>regex str</code>	Advanced URL validation	Match failures if regex is too specific
<code>url_changes(old_url)</code>	<code>str</code>	Waiting for redirects/navigation	Need to provide the current URL first

## Element Presence & Visibility

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>presence_of_element_located(locator)</code>	<code>tuple</code>	Ensure element exists in DOM	Doesn't ensure visibility
<code>presence_of_all_elements_located(locator)</code>	<code>tuple</code>	Wait for any number of matching elements	Returns list, may be empty initially
<code>visibility_of_element_located(locator)</code>	<code>tuple</code>	Ensure element is visible	Element must exist and be non-hidden
<code>visibility_of_all_elements_located(locator)</code>	<code>tuple</code>	Wait for all elements to be visible	If one is hidden, condition fails
<code>visibility_of_any_elements_located(locator)</code>	<code>tuple</code>	Wait until at least one element is visible	Requires some elements already loaded
<code>visibility_of(element)</code>	<code>WebElement</code>	Verify visibility of an already-located element	Will raise <code>StaleElementReferenceException</code> if element is removed

## Invisibility & Staleness

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>invisibility_of_element_located(locator)</code>	<code>tuple</code>	Wait for loading spinner or modal to disappear	Safe even if element never appears
<code>invisibility_of_element(element)</code>	<code>WebElement</code>	Wait for a specific element to hide or be removed	Can throw <code>StaleElementReferenceException</code>
<code>staleness_of(element)</code>	<code>WebElement</code>	Wait for element to detach from DOM (e.g., on reload)	Use after DOM changes, not before

## Clickability & Selection

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>element_to_be_clickable(locator_or_element)</code>	<code>tuple</code> or <code>WebElement</code>	Button or link is ready to be clicked	Must be visible <b>and</b> enabled
<code>element_selection_state_to_be(element, state)</code>	<code>WebElement</code> , <code>bool</code>	Checkbox, radio inputs	Element must exist
<code>element_to_be_selected(element)</code>	<code>WebElement</code>	Ensure a field is selected	Use when element is known
<code>element_located_to_be_selected(locator)</code>	<code>tuple</code>	Wait for a radio/checkbox to be selected	Element must exist in DOM
<code>element_located_selection_state_to_be(locator, state)</code>	<code>tuple</code> , <code>bool</code>	Wait for specific selection state	Use in toggle/checkbox/radio UIs

## Frames & Switches

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>frame_to_be_available_and_switch_to_it(locator)</code>	<code>tuple</code>	Switch to iframe when available	Automatically switches frame for you

## Text & Attributes

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>text_to_be_present_in_element(locator, text)</code>	<code>tuple</code> , <code>str</code>	Wait for a specific message or content	Fails if element is slow to update
<code>text_to_be_present_in_element_attribute(locator, attr, text)</code>	<code>tuple</code> , <code>str</code> , <code>str</code>	Attribute value updates (e.g.,	Attribute must exist

		class or href)	
<code>text_to_be_present_in_element_value(locator, text)</code>	<code>tuple, str</code>	For input fields (value attribute)	For text <b>inside</b> form elements
<code>element_attribute_to_include(locator, attribute)</code>	<code>tuple, str</code>	Wait for attribute to be included	Attribute might already exist, use with state checks if needed

## Logical Combinations

Condition	Input Type	Best Use Case	Caveats/Warnings
<code>all_of(*conditions)</code>	One or more expected conditions	Wait for all to pass together	Use to build composite conditions
<code>any_of(*conditions)</code>	One or more expected conditions	Proceed if <b>any one</b> condition is met	Useful for waiting on optional elements
<code>none_of(*conditions)</code>	One or more expected conditions	Ensure none of the conditions hold true	Can be used to detect absence of failure

Would you like this formatted as a **PDF**, **Markdown file**, or code snippet?

## Conditions personnalisées


Si aucune des conditions ci-dessus ne vous convient, vous pouvez créer une condition personnalisée:

```
class element_has_css_class(object):
    """Vérifie si un élément possède une certaine classe CSS."""

    def __init__(self, locator, css_class):
        self.locator = locator
        self.css_class = css_class

    def __call__(self, driver):
        element = driver.find_element(*self.locator)
        if self.css_class in element.get_attribute("class"):
            return element
        return False

# Exemple d'utilisation :
wait = WebDriverWait(driver, 10)
element = wait.until(element_has_css_class((By.ID, 'myNewInput'), "myCSSClass"))
```

 **Remarque :** Vous pouvez aussi envisager d'utiliser la bibliothèque `polling2`, à installer séparément, pour des conditions plus avancées.

## 5.2. Attentes implicites

Une **attente implicite** demande à WebDriver d'attendre un certain temps lors de la recherche d'un ou plusieurs éléments, s'ils ne sont pas immédiatement disponibles dans le DOM.

Elle est définie une seule fois et reste active pendant toute la durée de vie du WebDriver.

Exemple :

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.implicitly_wait(10) # secondes
driver.get("http://somedomain/url_that_delays_loading")
myDynamicElement = driver.find_element(By.ID, "myDynamicElement")
```