Selenium with Python Essentials

A rapid reference guide for developers and QA automation engineers using Selenium with Python to automate web browser interactions. Covers setup, element location, interactions, waits, navigation, screenshots, and common error handling.

Core Actions & Setup

SETUP & IMPORTS

1. Install Selenium:

```
pip install selenium
pip install webdriver_manager
```

2. Basic Imports:

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

3. Launching Browsers (Chrome Example):

```
from selenium.webdriver.chrome.service import
Service
from webdriver_manager.chrome import
ChromeDriverManager

# For Chrome
service =
Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service)

# For Firefox
# from selenium.webdriver.firefox.service
import Service as FirefoxService
# from webdriver_manager.firefox import
GeckoDriverManager
# service =
FirefoxService(GeckoDriverManager().install())
# driver = webdriver.Firefox(service=service)
```

4. Headless Mode (Chrome):

```
from selenium.webdriver.chrome.options import
Options
chrome_options = Options()
chrome_options.add_argument("--headless")
driver = webdriver.Chrome(service=service,
options=chrome_options)
```

Pro Tip: Use `webdriver_manager` to automatically handle WebDriver binaries, saving you from manual downloads and path configurations.

LOCATING ELEMENTS

`find_element()` vs `find_elements()`

- `find_element()`: Returns the first matching element or raises `NoSuchElementException`.
- `find_elements()`: Returns a list of all matching elements (empty list if none found).

By.ID

```
driver.find_element(By.ID, "username")
```

By.NAME

```
driver.find_element(By.NAME, "password")
```

By.CLASS_NAME

```
driver.find_element(By.CLASS_NAME, "submit-
button")
```

By.TAG_NAME

```
driver.find_element(By.TAG_NAME, "a") # First
link
driver.find_elements(By.TAG_NAME, "div") # All
div elements
```

By.LINK_TEXT / By.PARTIAL_LINK_TEXT

```
driver.find_element(By.LINK_TEXT, "Click Me")
driver.find_element(By.PARTIAL_LINK_TEXT,
"Click")
```

By.XPATH

```
driver.find_element(By.XPATH,
"//input[@type='text']")
driver.find_element(By.XPATH,
"//div[contains(text(), 'Hello')]")
```

By.CSS_SELECTOR

```
driver.find_element(By.CSS_SELECTOR,
"#loginForm input[type='submit']")
driver.find_element(By.CSS_SELECTOR, ".menu-
item:nth-child(2)")
```

Pro Tip: Prioritize `By.ID` as it's the most robust and fastest. If not available, `By.CSS_SELECTOR` is generally preferred over `By.XPATH` for performance and readability, though XPATH is more powerful for complex queries.

INTERACTING WITH ELEMENTS

Clicking an element

```
button = driver.find_element(By.ID,
"submitBtn")
button.click()
```

Typing into a text field

```
username_field = driver.find_element(By.NAME,
"username")
username_field.send_keys("testuser")
```

Clearing text

```
text_area = driver.find_element(By.TAG_NAME,
"textarea")
text_area.clear()
```

Sending special keys

```
input_field.send_keys(Keys.RETURN) # Press
Enter
input_field.send_keys(Keys.CONTROL + 'a') #
Select all
```

Dropdowns (Select class)

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

dropdown = Select(driver.find_element(By.ID,
"myDropdown"))
dropdown.select_by_visible_text("Option Two")
dropdown.select_by_value("value3")
dropdown.select_by_index(0) # First option

# Get all options
options = dropdown.options
```

Checkboxes & Radio Buttons

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

radio_button = driver.find_element(By.ID,
"optionRadio")
radio_button.click()
```

Getting element text / attribute

```
element = driver.find_element(By.ID,
"statusMessage")
print(element.text) # Get visible text
print(element.get_attribute("value")) # Get
input value
print(element.get_attribute("href")) # Get link
URL
```

Common Pitfall: `ElementNotInteractableException` often means the element is not yet visible or clickable. Use explicit waits before attempting interaction.

Advanced Techniques & Control

WAITING STRATEGIES

`time.sleep()`

```
import time
time.sleep(5) # Waits for 5 seconds (NOT RECOMMENDED for robustness)
```

Note: This is a fixed delay and should be avoided in most cases as it makes tests slow and brittle.

Implicit Wait

```
driver.implicitly_wait(10) # Waits up to 10 seconds for elements to appear.
```

Explanation: Applied globally for the WebDriver instance. Selenium will poll the DOM for a certain amount of time when trying to find an element.

Explicit Wait (`WebDriverWait`)

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

wait = WebDriverWait(driver, 10) # Wait up to 10 seconds

# Wait for element to be clickable
login_button = wait.until(EC.element_to_be_clickable((By.ID, "loginBtn")))
login_button.click()
```

Explanation: Waits for a specific condition to be met before proceeding. More flexible and robust than implicit waits.

Common `expected_conditions` (EC)

- `EC.presence_of_element_located(locator)`
- `EC.visibility_of_element_located(locator)`
- `EC.element_to_be_clickable(locator)`
- `EC.text_to_be_present_in_element(locator, text)`
- `EC.title_contains(title)`
- `EC.alert_is_present()`

Custom Wait Conditions

```
# Wait until an element's attribute has a specific value
wait.until(lambda d: d.find_element(By.ID, "status").get_attribute("data-state") == "ready")
```

Pro Tip: Use explicit waits for specific conditions, especially before interacting with dynamic elements. Use implicit waits as a fallback for general element presence, but be cautious as they can sometimes conflict with explicit waits causing unexpected delays.

NAVIGATION & BROWSER CONTROL

Navigate to a URL

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

Browser history

```
driver.back()     # Go back in browser history
driver.forward()  # Go forward in browser history
driver.refresh()  # Refresh the current page
```

Window size & position

```
driver.maximize_window()
driver.minimize_window()
driver.set_window_size(1024, 768) # Width, Height
```

Close vs Quit

```
driver.close() # Closes the current window/tab
driver.quit()  # Closes all windows/tabs and quits the browser session
```

Switching Windows/Tabs

```
# Open new tab (example)
driver.execute_script("window.open('about:blank','_blank');")

# Get all window handles (unique IDs)
window_handles = driver.window_handles

# Switch to the new tab (usually the last one)
driver.switch_to.window(window_handles[-1])
driver.get("https://www.new-tab.com")

# Switch back to the original tab
driver.switch_to.window(window_handles[0])
```

Switching Frames

```
# By ID or Name
driver.switch_to.frame("myiframe")

# By WebElement
iframe_element = driver.find_element(By.TAG_NAME, "iframe")
driver.switch_to.frame(iframe_element)

# Switch back to the main content
driver.switch_to.default_content()
```

Common Pitfall: Forgetting to call `driver.quit()` at the end of your test suite. This leaves browser processes running, consuming resources and potentially causing issues in subsequent runs.

SCREENSHOTS & LOGGING

Capture full page screenshot

```
driver.save_screenshot("full_page.png")
```

Capture element screenshot

```
element = driver.find_element(By.ID, "myElement")
element.screenshot("element_screenshot.png")
```

Basic Python Logging

```
import logging

logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s',
                    filename='test_log.log',
                    filemode='w')

logging.info("Navigating to homepage")
logging.warning("Element not found within 5 seconds")
logging.error("Test failed: Login button not clickable")
```

Pro Tip: Include timestamps in your screenshot filenames (e.g., `screenshot_{time.strftime('%Y%m%d-%H%M%S')}.png`) to easily differentiate them when debugging test failures over time.

COMMON ERRORS & DEBUGGING

`NoSuchElementException`

Cause: Element not found on the page or not present in the DOM when searched.
Solution: Verify locator, ensure element is present/visible (use explicit waits), or switch to correct frame/window.

`TimeoutException`

Cause: An explicit wait condition was not met within the specified timeout duration.
Solution: Increase wait time, check if the condition is realistic, or investigate why the element isn't reaching the expected state.

`ElementNotInteractableException`

Cause: Element is present but not in a state to be interacted with (e.g., hidden, disabled, overlaid).
Solution: Wait for the element to become clickable/visible, use `execute_script` to interact if truly necessary, or check for pop-ups/modals obstructing it.

`StaleElementReferenceException`

Cause: The element reference you hold is no longer attached to the DOM (e.g., page refresh, element reloaded).
Solution: Re-locate the element after the DOM change. Often, wrapping the interaction in a `try-except` block and re-trying the find operation helps.

Using `try/except`

```
from selenium.common.exceptions import NoSuchElementException

try:
    element = driver.find_element(By.ID, "nonExistentElement")
    element.click()
except NoSuchElementException:
    print("Element not found, handling gracefully.")
    # Log the error, take screenshot, or continue test differently
```

Debugging Tips

- **Print Statements:** Simple but effective for tracing execution.
- **`pdb` (Python Debugger):** Insert `import pdb; pdb.set_trace()` to pause execution and inspect variables.
- **Browser Developer Tools:** Crucial for inspecting elements, network requests, and console errors.
- **Screenshots:** Capture screenshots on failure to visualize the state of the page.

Pro Tip: Most common Selenium errors stem from timing issues. Master explicit waits and understand the state of your application's DOM to write robust and reliable automation scripts.