Phase 3.1: Waits in Selenium Python - Implicit vs Explicit

Overview

Waits in Selenium are crucial for handling dynamic web content that loads asynchronously. Modern web applications use JavaScript, AJAX, and other technologies that cause elements to appear/disappear dynamically. Without proper waits, your tests will fail intermittently due to timing issues.

Why Do We Need Waits?

Common Timing Issues

```
# ❌ BAD: This will often fail
driver.get("https://example.com")
driver.find_element(By.ID, "dynamic-button").click()  # Element might not be loaded yet

# ✅ GOOD: Using appropriate waits
driver.get("https://example.com")
wait = WebDriverWait(driver, 10)
button = wait.until(EC.element_to_be_clickable((By.ID, "dynamic-button")))
button.click()
```

Problems Without Waits

- **NoSuchElementException**: Element not found because it hasn't loaded yet
- **ElementNotInteractableException**: Element exists but not ready for interaction
- **StaleElementReferenceException**: Element reference becomes invalid
- **Flaky Tests**: Tests pass sometimes but fail randomly

1. Implicit Wait

What is Implicit Wait?

Implicit wait tells WebDriver to wait for a certain amount of time when trying to find elements. It's applied globally to the driver instance.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

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

# Set implicit wait - applies to ALL element searches
driver.implicitly_wait(10)  # Wait up to 10 seconds

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

```
    # This will wait up to 10 seconds for the element to appear
    element = driver.find_element(By.ID, "slow-loading-element")
    element.click()

    # All subsequent find_element calls will also wait up to 10 seconds
    another_element = driver.find_element(By.CLASS_NAME, "dynamic-content")

finally:
    driver.quit()
```

Implicit Wait Characteristics

- **Global**: Applies to all `find_element()` calls
- **Simple**: Easy to implement with one line of code
- **Automatic**: No need to specify wait conditions for each element
- **Limited**: Only waits for element presence, not specific states

Implicit Wait Best Practices

```
class WebDriverSetup:
    def __init__(self, implicit_wait_time=10):
        self.driver =
webdriver.Chrome(service=Service(ChromeDriverManager().install()))
        self.driver.implicitly_wait(implicit_wait_time)

    def get_driver(self):
        return self.driver

    def close(self):
        self.driver.quit()

# Usage
setup = WebDriverSetup(implicit_wait_time=15)
driver = setup.get_driver()

# All element searches will now wait up to 15 seconds
driver.get("https://example.com")
element = driver.find_element(By.ID, "any-element")  # Waits automatically
```

2. Explicit Wait

What is Explicit Wait?

Explicit wait allows you to wait for specific conditions to be met before proceeding. It's more flexible and precise than implicit wait.

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

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

try:
    driver.get("https://example.com")

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

```
    # Wait for specific condition
    element = wait.until(
        EC.presence_of_element_located((By.ID, "dynamic-element"))
    )

    print("Element found and is present in DOM")

except TimeoutException:
    print("Element not found within 10 seconds")

finally:
    driver.quit()
```

Common Expected Conditions

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

# Wait for element to be present in DOM
wait.until(EC.presence_of_element_located((By.ID, "element-id")))

# Wait for element to be visible
wait.until(EC.visibility_of_element_located((By.ID, "element-id")))

# Wait for element to be clickable
wait.until(EC.element_to_be_clickable((By.ID, "button-id")))

# Wait for text to be present in element
wait.until(EC.text_to_be_present_in_element((By.ID, "status"), "Complete"))

# Wait for element to be invisible
wait.until(EC.invisibility_of_element_located((By.ID, "loading-spinner")))

# Wait for title to contain specific text
wait.until(EC.title_contains("Expected Title"))

# Wait for URL to contain specific text
wait.until(EC.url_contains("success"))

# Wait for frame to be available and switch to it
wait.until(EC.frame_to_be_available_and_switch_to_it((By.ID, "frame-id")))

# Wait for alert to be present
wait.until(EC.alert_is_present())

# Wait for element to be selected
wait.until(EC.element_to_be_selected(checkbox_element))
```

Comprehensive Expected Conditions Examples

```
class ExplicitWaitExamples:
    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    def wait_for_login_success(self):
        """Wait for successful login indicators"""
        try:
            # Wait for welcome message to appear
            welcome_msg = self.wait.until(
                EC.visibility_of_element_located((By.CLASS_NAME, "welcome-
message"))
            )
```

```
            # Wait for URL to change to dashboard
            self.wait.until(EC.url_contains("/dashboard"))

            print("Login successful")
            return True

        except TimeoutException:
            print("Login failed or took too long")
            return False

    def wait_for_form_submission(self):
        """Wait for form submission to complete"""
        # Click submit button
        submit_btn = self.wait.until(
            EC.element_to_be_clickable((By.ID, "submit-btn"))
        )
        submit_btn.click()

        # Wait for loading spinner to appear
        self.wait.until(
            EC.visibility_of_element_located((By.CLASS_NAME, "loading-spinner"))
        )

        # Wait for loading spinner to disappear
        self.wait.until(
            EC.invisibility_of_element_located((By.CLASS_NAME, "loading-
spinner"))
        )

        # Wait for success message
        success_msg = self.wait.until(
            EC.visibility_of_element_located((By.CLASS_NAME, "success-message"))
        )

        return success_msg.text

    def wait_for_dropdown_options(self, dropdown_id):
        """Wait for dropdown options to load"""
        # Click dropdown to open
        dropdown = self.wait.until(
            EC.element_to_be_clickable((By.ID, dropdown_id))
        )
        dropdown.click()

        # Wait for options to appear
        options = self.wait.until(
            EC.presence_of_all_elements_located((By.CSS_SELECTOR,
f"#{dropdown_id} option"))
        )

        return options

    def wait_for_file_download(self, download_button_id, expected_filename):
        """Wait for file download to start"""
        import os
        import glob

        # Get download directory
        download_dir = os.path.expanduser("~/Downloads")

        # Count files before download
        files_before = len(os.listdir(download_dir))
```

```
        # Click download button
        download_btn = self.wait.until(
            EC.element_to_be_clickable((By.ID, download_button_id))
        )
        download_btn.click()

        # Wait for file to appear in downloads
        def file_downloaded(driver):
            files_after = len(os.listdir(download_dir))
            return files_after > files_before

        self.wait.until(file_downloaded)

        # Wait for specific file
        def specific_file_exists(driver):
            pattern = os.path.join(download_dir, f"*{expected_filename}*")
            return len(glob.glob(pattern)) > 0

        self.wait.until(specific_file_exists)
        print(f"File {expected_filename} downloaded successfully")
```

3. Custom Wait Conditions

Creating Custom Expected Conditions

```
class custom_expected_conditions:
    """Custom expected conditions for specific scenarios"""

    @staticmethod
    def element_attribute_to_include(locator, attribute, value):
        """Wait for element attribute to include specific value"""
        def _predicate(driver):
            try:
                element = driver.find_element(*locator)
                return value in element.get_attribute(attribute)
            except:
                return False
        return _predicate

    @staticmethod
    def element_count_to_be(locator, count):
        """Wait for specific number of elements"""
        def _predicate(driver):
            try:
                elements = driver.find_elements(*locator)
                return len(elements) == count
            except:
                return False
        return _predicate

    @staticmethod
    def page_to_load_completely():
        """Wait for page to load completely"""
        def _predicate(driver):
            return driver.execute_script("return document.readyState") ==
"complete"
        return _predicate

    @staticmethod
    def ajax_complete():
        """Wait for all AJAX requests to complete (jQuery)"""
        def _predicate(driver):
```

```
            try:
                return driver.execute_script("return jQuery.active == 0")
            except:
                return True  # If jQuery is not present, assume no AJAX
        return _predicate

# Usage of custom conditions
wait = WebDriverWait(driver, 10)

# Wait for element class to include 'active'
wait.until(
    custom_expected_conditions.element_attribute_to_include(
        (By.ID, "menu-item"), "class", "active"
    )
)

# Wait for exactly 5 product items to load
wait.until(
    custom_expected_conditions.element_count_to_be(
        (By.CLASS_NAME, "product-item"), 5
    )
)

# Wait for page to load completely
wait.until(custom_expected_conditions.page_to_load_completely())

# Wait for AJAX to complete
wait.until(custom_expected_conditions.ajax_complete())
```

4. Fluent Wait (Advanced Explicit Wait)

What is Fluent Wait?

Fluent Wait is more flexible than WebDriverWait, allowing you to configure polling frequency and ignore specific exceptions.

```
from selenium.webdriver.support.wait import WebDriverWait
from selenium.common.exceptions import NoSuchElementException,
ElementNotVisibleException

class FluentWaitExample:
    def __init__(self, driver):
        self.driver = driver

    def fluent_wait_example(self):
        """Advanced fluent wait with custom configuration"""

        # Create fluent wait
        wait = WebDriverWait(
            driver=self.driver,
            timeout=30,  # Maximum wait time
            poll_frequency=2,  # Check every 2 seconds
            ignored_exceptions=[
                NoSuchElementException,
                ElementNotVisibleException
            ]
        )

        # Use fluent wait
        element = wait.until(
            EC.visibility_of_element_located((By.ID, "slow-element"))
```

```
        )

        return element

    def wait_with_custom_message(self):
        """Fluent wait with custom error message"""

        wait = WebDriverWait(self.driver, 10, poll_frequency=1)

        try:
            element = wait.until(
                EC.element_to_be_clickable((By.ID, "submit-button")),
                message="Submit button was not clickable within 10 seconds"
            )
            return element

        except TimeoutException as e:
            print(f"Custom error: {e}")
            raise
```

5. Combining Implicit and Explicit Waits

Important Warning

```
# ⚠ WARNING: Mixing implicit and explicit waits can cause unexpected behavior

driver.implicitly_wait(10)  # Implicit wait: 10 seconds
wait = WebDriverWait(driver, 5)  # Explicit wait: 5 seconds

# This might wait up to 15 seconds (10 + 5) in some cases!
element = wait.until(EC.presence_of_element_located((By.ID, "element")))
```

Best Practice: Choose One Approach

```
# ✅ OPTION 1: Use only implicit waits (simpler, less flexible)
class ImplicitWaitStrategy:
    def __init__(self):
        self.driver =
webdriver.Chrome(service=Service(ChromeDriverManager().install()))
        self.driver.implicitly_wait(10)

    def find_element_safely(self, by, value):
        return self.driver.find_element(by, value)  # Auto-waits up to 10
seconds

# ✅ OPTION 2: Use only explicit waits (more flexible, recommended)
class ExplicitWaitStrategy:
    def __init__(self):
        self.driver =
webdriver.Chrome(service=Service(ChromeDriverManager().install()))
        # No implicit wait set
        self.wait = WebDriverWait(self.driver, 10)

    def find_element_safely(self, by, value):
        return self.wait.until(EC.presence_of_element_located((by, value)))
```

6. Practical Wait Scenarios

Scenario 1: E-commerce Product Search

```
def test_product_search():
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    wait = WebDriverWait(driver, 15)

    try:
        # Navigate to e-commerce site
        driver.get("https://example-shop.com")

        # Wait for search box and enter search term
        search_box = wait.until(
            EC.element_to_be_clickable((By.ID, "search-input"))
        )
        search_box.send_keys("laptop")

        # Click search button
        search_btn = wait.until(
            EC.element_to_be_clickable((By.ID, "search-button"))
        )
        search_btn.click()

        # Wait for loading spinner to appear
        wait.until(
            EC.visibility_of_element_located((By.CLASS_NAME, "loading-spinner"))
        )

        # Wait for loading spinner to disappear
        wait.until(
            EC.invisibility_of_element_located((By.CLASS_NAME, "loading-
spinner"))
        )

        # Wait for search results to load
        results = wait.until(
            EC.presence_of_all_elements_located((By.CLASS_NAME, "product-item"))
        )

        print(f"Found {len(results)} products")

        # Wait for "Load More" button to be clickable (if exists)
        try:
            load_more = wait.until(
                EC.element_to_be_clickable((By.ID, "load-more-button"))
            )
            load_more.click()

            # Wait for additional products to load
            wait.until(
                lambda driver: len(driver.find_elements(By.CLASS_NAME, "product-
item")) > len(results)
            )

        except TimeoutException:
            print("No 'Load More' button found or not clickable")

    finally:
        driver.quit()
```

Scenario 2: Form Validation Handling

```
def test_form_with_validation():
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-form.com")

        # Fill form fields
        email_field = wait.until(
            EC.presence_of_element_located((By.ID, "email"))
        )
        email_field.send_keys("invalid-email")  # Intentionally invalid

        password_field = wait.until(
            EC.presence_of_element_located((By.ID, "password"))
        )
        password_field.send_keys("123")  # Too short

        # Submit form
        submit_btn = wait.until(
            EC.element_to_be_clickable((By.ID, "submit"))
        )
        submit_btn.click()

        # Wait for validation errors to appear
        email_error = wait.until(
            EC.visibility_of_element_located((By.ID, "email-error"))
        )

        password_error = wait.until(
            EC.visibility_of_element_located((By.ID, "password-error"))
        )

        print(f"Email error: {email_error.text}")
        print(f"Password error: {password_error.text}")

        # Fix the errors
        email_field.clear()
        email_field.send_keys("valid@email.com")

        password_field.clear()
        password_field.send_keys("validpassword123")

        # Wait for errors to disappear
        wait.until(
            EC.invisibility_of_element_located((By.ID, "email-error"))
        )

        wait.until(
            EC.invisibility_of_element_located((By.ID, "password-error"))
        )

        # Submit again
        submit_btn.click()

        # Wait for success message
        success_msg = wait.until(
            EC.visibility_of_element_located((By.CLASS_NAME, "success-message"))
        )

        print(f"Success: {success_msg.text}")
```

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

7. Wait Utilities and Helper Functions

Comprehensive Wait Utility Class

```
class WaitUtils:
    def __init__(self, driver, default_timeout=10):
        self.driver = driver
        self.default_timeout = default_timeout

    def wait_for_element_clickable(self, locator, timeout=None):
        """Wait for element to be clickable"""
        wait_time = timeout or self.default_timeout
        wait = WebDriverWait(self.driver, wait_time)
        return wait.until(EC.element_to_be_clickable(locator))

    def wait_for_element_visible(self, locator, timeout=None):
        """Wait for element to be visible"""
        wait_time = timeout or self.default_timeout
        wait = WebDriverWait(self.driver, wait_time)
        return wait.until(EC.visibility_of_element_located(locator))

    def wait_for_text_in_element(self, locator, text, timeout=None):
        """Wait for specific text to appear in element"""
        wait_time = timeout or self.default_timeout
        wait = WebDriverWait(self.driver, wait_time)
        wait.until(EC.text_to_be_present_in_element(locator, text))
        return self.driver.find_element(*locator)

    def wait_for_page_title(self, expected_title, timeout=None):
        """Wait for page title to match expected title"""
        wait_time = timeout or self.default_timeout
        wait = WebDriverWait(self.driver, wait_time)
        return wait.until(EC.title_is(expected_title))

    def wait_for_url_change(self, current_url, timeout=None):
        """Wait for URL to change from current URL"""
        wait_time = timeout or self.default_timeout
        wait = WebDriverWait(self.driver, wait_time)
        return wait.until(lambda driver: driver.current_url != current_url)

    def wait_for_element_count(self, locator, expected_count, timeout=None):
        """Wait for specific number of elements"""
        wait_time = timeout or self.default_timeout
        wait = WebDriverWait(self.driver, wait_time)

        def count_matches(driver):
            elements = driver.find_elements(*locator)
            return len(elements) == expected_count

        wait.until(count_matches)
        return self.driver.find_elements(*locator)

    def wait_and_click(self, locator, timeout=None):
        """Wait for element to be clickable and click it"""
        element = self.wait_for_element_clickable(locator, timeout)
        element.click()
        return element

    def wait_and_send_keys(self, locator, text, timeout=None):
```

```
        """Wait for element to be visible and send keys"""
        element = self.wait_for_element_visible(locator, timeout)
        element.clear()
        element.send_keys(text)
        return element

# Usage example
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
wait_utils = WaitUtils(driver, default_timeout=15)

try:
    driver.get("https://example.com")

    # Use utility methods
    wait_utils.wait_and_click((By.ID, "login-button"))
    wait_utils.wait_and_send_keys((By.ID, "username"), "testuser")
    wait_utils.wait_and_send_keys((By.ID, "password"), "testpass")
    wait_utils.wait_and_click((By.ID, "submit"))

    # Wait for login success
    wait_utils.wait_for_text_in_element(
        (By.CLASS_NAME, "welcome"), "Welcome, testuser"
    )

finally:
    driver.quit()
```

8. Performance Considerations and Best Practices

Optimization Tips

```
# ✅ Use appropriate timeout values
short_wait = WebDriverWait(driver, 3)   # For fast operations
medium_wait = WebDriverWait(driver, 10) # For normal operations
long_wait = WebDriverWait(driver, 30)   # For slow operations

# ✅ Use specific expected conditions
wait.until(EC.element_to_be_clickable((By.ID, "button")))  # Specific condition
# Instead of just:
wait.until(EC.presence_of_element_located((By.ID, "button")))  # Less specific

# ✅ Avoid sleeping
import time
time.sleep(5)  # ❌ Bad - fixed wait regardless of actual loading time

# Instead use:
wait.until(EC.visibility_of_element_located((By.ID, "element")))  # ✅ Good
```

Error Handling Best Practices

```
def robust_wait_example():
    wait = WebDriverWait(driver, 10)

    try:
        # Try primary locator
        element = wait.until(
            EC.element_to_be_clickable((By.ID, "primary-button"))
        )
    except TimeoutException:
        try:
            # Fallback to alternative locator
```

```
            element = wait.until(
                EC.element_to_be_clickable((By.CLASS_NAME, "button-primary"))
            )
        except TimeoutException:
            # Final fallback
            print("Button not found with any locator")
            return None

    return element
```

9. Comparison: Implicit vs Explicit Wait

Aspect	Implicit Wait	Explicit Wait
Scope	Global (all find_element calls)	Specific conditions
Flexibility	Limited	High
Conditions	Only element presence	Many conditions available
Performance	Can be slower	More efficient
Complexity	Simple	More code required
Best For	Simple scripts	Production automation
Recommendation	Beginners	Advanced users

Summary

Waits in Selenium are essential for:

- **Handling Dynamic Content**: Elements that load asynchronously
- **Preventing Flaky Tests**: Tests that fail randomly due to timing
- **Improving Reliability**: Making tests more stable and predictable

Key Takeaways:

- **Choose One Strategy**: Use either implicit OR explicit waits, not both
- **Explicit Waits are Preferred**: More flexible and precise
- **Use Appropriate Timeouts**: Not too short (flaky) or too long (slow)
- **Handle Exceptions**: Always prepare for timeout scenarios
- **Custom Conditions**: Create custom waits for specific needs

Why waits are essential for handling dynamic web content

- **Implicit vs Explicit waits** with detailed comparisons
- **All common Expected Conditions** with practical examples
- **Custom wait conditions** for specific scenarios
- **Fluent wait** for advanced configurations
- **Real-world scenarios** like e-commerce and form handling
- **Utility classes** and helper functions
- **Best practices** and performance considerations

The guide includes extensive code examples that demonstrate both basic concepts and advanced patterns you'll encounter in real automation projects.

Key points to remember:

- **Use explicit waits** for production automation (more flexible)

- **Don't mix implicit and explicit waits**
- **Choose appropriate timeout values**
- **Handle TimeoutException properly**
- **Use specific expected conditions** rather than generic ones

Are you ready to move on to the next topic? The next one would be **Phase 3.2: Actions Class** covering advanced user interactions like mouse hover, right-click, drag and drop, etc. Let me know when you'd like me to cover that!

Selenium Actions Class - Complete Guide

What is Actions Class?

The Actions class in Selenium WebDriver is used to perform complex user interactions that go beyond simple click() and send_keys() methods. It allows you to simulate real user behaviors like mouse hover, right-click, drag and drop, keyboard shortcuts, and more.

Setting Up Actions Class

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

# Initialize driver
driver = webdriver.Chrome()
actions = ActionChains(driver)
```

1. Mouse Hover (Move to Element)

Mouse hover is essential for testing dropdown menus and tooltip interactions.

```
# Example: Hover over an element
def test_mouse_hover():
    driver.get("https://example.com")

    # Find the element to hover over
    hover_element = driver.find_element(By.ID, "menu-item")

    # Perform mouse hover
    actions.move_to_element(hover_element).perform()

    # Wait to see the hover effect
    time.sleep(2)

    # Now you can interact with elements that appear on hover
    dropdown_item = driver.find_element(By.ID, "dropdown-option")
    dropdown_item.click()
```

Real-world Example - Amazon Menu Hover

```
def test_amazon_menu_hover():
    driver.get("https://amazon.com")
```

```
    # Hover over "Accounts & Lists"
    account_menu = driver.find_element(By.ID, "nav-link-accountList")
    actions.move_to_element(account_menu).perform()

    time.sleep(2)

    # Click on "Your Orders" from the dropdown
    your_orders = driver.find_element(By.LINK_TEXT, "Your Orders")
    your_orders.click()
```

2. Right Click (Context Click)

Right-click opens context menus in web applications.

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

    # Find element to right-click
    element = driver.find_element(By.ID, "right-click-area")

    # Perform right-click
    actions.context_click(element).perform()

    # Handle the context menu that appears
    context_option = driver.find_element(By.ID, "context-menu-option")
    context_option.click()
```

Handling Browser Context Menu

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

    # Right-click on page
    actions.context_click().perform()  # Right-click at current mouse position

    # Note: Browser context menus are usually handled differently
    # You might need to use Keys.ESCAPE to close it
    from selenium.webdriver.common.keys import Keys
    actions.send_keys(Keys.ESCAPE).perform()
```

3. Double Click

Double-click is used for actions like selecting text or opening files.

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

    # Find element to double-click
    element = driver.find_element(By.ID, "double-click-element")

    # Perform double-click
    actions.double_click(element).perform()

    # Verify the double-click action worked
    assert "double-clicked" in element.get_attribute("class")
```

4. Drag and Drop

Drag and drop is common in modern web applications for reordering items or file uploads.

```
def test_drag_and_drop():
    driver.get("https://jqueryui.com/droppable/")

    # Switch to iframe if needed
    iframe = driver.find_element(By.CLASS_NAME, "demo-frame")
    driver.switch_to.frame(iframe)

    # Find source and target elements
    source = driver.find_element(By.ID, "draggable")
    target = driver.find_element(By.ID, "droppable")

    # Method 1: Using drag_and_drop()
    actions.drag_and_drop(source, target).perform()

    # Verify the drag and drop worked
    assert "ui-state-highlight" in target.get_attribute("class")
```

Alternative Drag and Drop Method

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

    source = driver.find_element(By.ID, "draggable")

    # Method 2: Using click_and_hold + move_by_offset + release
    actions.click_and_hold(source).move_by_offset(200, 100).release().perform()
```

Advanced Drag and Drop with Multiple Steps

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

    source = driver.find_element(By.ID, "item1")
    target = driver.find_element(By.ID, "target-area")

    # Perform drag and drop with pauses
    actions.click_and_hold(source).perform()
    time.sleep(1)  # Hold for a moment

    actions.move_to_element(target).perform()
    time.sleep(1)  # Pause before dropping

    actions.release().perform()
```

5. Keyboard Shortcuts and Key Combinations

Actions class can simulate keyboard shortcuts like Ctrl+C, Ctrl+V, etc.

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

def test_keyboard_shortcuts():
    driver.get("https://example.com")

    text_area = driver.find_element(By.ID, "text-input")
    text_area.click()
```

```
    # Type some text
    text_area.send_keys("Hello World")

    # Select all text (Ctrl+A)
    actions.key_down(Keys.CONTROL).send_keys("a").key_up(Keys.CONTROL).perform()

    # Copy text (Ctrl+C)
    actions.key_down(Keys.CONTROL).send_keys("c").key_up(Keys.CONTROL).perform()

    # Move to another field and paste (Ctrl+V)
    another_field = driver.find_element(By.ID, "another-input")
    another_field.click()
    actions.key_down(Keys.CONTROL).send_keys("v").key_up(Keys.CONTROL).perform()
```

6. Chaining Multiple Actions

You can chain multiple actions together before performing them.

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

    element1 = driver.find_element(By.ID, "element1")
    element2 = driver.find_element(By.ID, "element2")

    # Chain multiple actions

actions.move_to_element(element1).click().move_to_element(element2).context_clic
k().perform()
```

7. Complete Real-World Example

Here's a comprehensive example combining multiple Actions class features:

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
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
import time

class ActionsClassTest(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Chrome()
        self.actions = ActionChains(self.driver)
        self.wait = WebDriverWait(self.driver, 10)

    def test_complete_actions_scenario(self):
        # Navigate to a test site
        self.driver.get("https://jqueryui.com/droppable/")

        # Switch to iframe
        iframe = self.wait.until(EC.presence_of_element_located((By.CLASS_NAME,
"demo-frame")))
        self.driver.switch_to.frame(iframe)

        # Test drag and drop
        source = self.wait.until(EC.element_to_be_clickable((By.ID,
"draggable")))
        target = self.driver.find_element(By.ID, "droppable")
```

```
        # Hover over source first
        self.actions.move_to_element(source).perform()
        time.sleep(1)

        # Perform drag and drop
        self.actions.drag_and_drop(source, target).perform()

        # Verify the action
        self.assertIn("ui-state-highlight", target.get_attribute("class"))

        print("Drag and drop test completed successfully!")

    def tearDown(self):
        self.driver.quit()

if __name__ == "__main__":
    unittest.main()
```

8. Best Practices for Actions Class

Always Call perform()

```
# Wrong - action won't execute
actions.move_to_element(element)

# Correct - action will execute
actions.move_to_element(element).perform()
```

Use Explicit Waits with Actions

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

    # Wait for element to be present before acting on it
    element = WebDriverWait(driver, 10).until(
        EC.element_to_be_clickable((By.ID, "hover-menu"))
    )

    actions.move_to_element(element).perform()
```

Handle StaleElementReferenceException

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

    try:
        element = driver.find_element(By.ID, "dynamic-element")
        actions.move_to_element(element).perform()
    except StaleElementReferenceException:
        # Re-find the element
        element = driver.find_element(By.ID, "dynamic-element")
        actions.move_to_element(element).perform()
```

Practice Exercises

1. **Exercise 1**: Create a test that hovers over a main menu item and clicks on a submenu item.

2. **Exercise 2**: Implement a drag and drop test for reordering items in a list.
3. **Exercise 3**: Create a test that uses keyboard shortcuts to copy text from one field and paste it into another.
4. **Exercise 4**: Write a test that performs a right-click and selects an option from the context menu.

Common Issues and Solutions

Issue 1: Actions not working in headless mode

```
# Solution: Add window size for headless mode
chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument("--headless")
chrome_options.add_argument("--window-size=1920,1080")
driver = webdriver.Chrome(options=chrome_options)
```

Issue 2: Element not visible for interaction

```
# Solution: Scroll to element first
element = driver.find_element(By.ID, "target-element")
driver.execute_script("arguments[0].scrollIntoView();", element)
actions.move_to_element(element).perform()
```

Key topics covered:

- Mouse Hover (move_to_element)
- Right Click (context_click)
- Double Click
- Drag and Drop (multiple methods)
- Keyboard shortcuts and key combinations
- Chaining multiple actions
- Real-world examples and best practices

What you should practice:

1. Try the mouse hover examples with different websites
2. Implement drag and drop on sites like jQuery UI demos
3. Practice keyboard shortcuts (Ctrl+C, Ctrl+V, etc.)
4. Work through the provided exercises

The Actions Class is crucial for testing modern web applications that have rich user interactions. Take your time to practice these concepts thoroughly.

Handling Modern Web Elements with Actions Class - Real-Time Examples

🚀 Modern Web Elements Overview

Modern web applications use complex UI patterns that standard click() and send_keys() can't handle effectively. Here's how to tackle them with Actions Class.

1. 🎯 Dynamic Hover Menus (Like Amazon, eBay)

Amazon Navigation Menu

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
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

def test_amazon_mega_menu():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://www.amazon.com")

        # Handle cookie banner if present
        try:
            cookie_accept = wait.until(EC.element_to_be_clickable((By.ID, "sp-
cc-accept")))
            cookie_accept.click()
        except:
            pass

        # Hover over "Departments" or "All" menu
        all_menu = wait.until(EC.element_to_be_clickable((By.ID, "nav-hamburger-
menu")))
        actions.move_to_element(all_menu).perform()
        time.sleep(1)

        # Click to open the menu
        all_menu.click()

        # Wait for the side menu to appear
        side_menu = wait.until(EC.presence_of_element_located((By.ID, "hmenu-
content")))

        # Hover over a category (e.g., "Electronics")
        electronics = wait.until(EC.element_to_be_clickable((By.LINK_TEXT,
"Electronics")))
        actions.move_to_element(electronics).perform()
        time.sleep(1)

        # Click on subcategory
        electronics.click()

        print("Amazon mega menu navigation successful!")
```

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

Modern E-commerce Hover Menu (Generic Pattern)

```
def test_modern_hover_menu():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-ecommerce-site.com")

        # Modern hover menu with multiple levels
        main_category = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
"[data-menu='clothing']")))

        # Hover to reveal submenu
        actions.move_to_element(main_category).perform()
        time.sleep(0.5)  # Allow menu animation

        # Wait for submenu to be visible
        submenu = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
".submenu-container")))

        # Hover over submenu item
        submenu_item = submenu.find_element(By.CSS_SELECTOR, "[data-
category='shirts']")
        actions.move_to_element(submenu_item).perform()
        time.sleep(0.5)

        # Click on final item
        final_item = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
"[data-item='casual-shirts']")))
        actions.move_to_element(final_item).click().perform()

    finally:
        driver.quit()
```

2. 🖱 Complex Drag and Drop (Kanban Boards, Sortable Lists)

Trello-like Kanban Board

```
def test_kanban_drag_drop():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        # Example with a Trello-like board
        driver.get("https://trello.com/demo-board")

        # Wait for board to load
        time.sleep(3)

        # Find a card to drag
        card = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR, ".list-
card")))
```

```
        # Find target column
        target_column = driver.find_element(By.CSS_SELECTOR, ".list:last-
child .list-cards")

        # Perform drag and drop with smooth movement
        actions.click_and_hold(card).perform()
        time.sleep(0.5)  # Hold for visual feedback

        # Move gradually to target (more realistic)
        actions.move_to_element_with_offset(target_column, 0, 50).perform()
        time.sleep(0.5)

        # Release
        actions.release().perform()

        print("Kanban card moved successfully!")

    finally:
        driver.quit()
```

Sortable List with Visual Feedback

```
def test_sortable_list():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://jqueryui.com/sortable/")

        # Switch to iframe
        iframe = wait.until(EC.presence_of_element_located((By.CLASS_NAME,
"demo-frame")))
        driver.switch_to.frame(iframe)

        # Get list items
        items = wait.until(EC.presence_of_all_elements_located((By.CSS_SELECTOR,
"#sortable li")))

        # Drag first item to third position
        source = items[0]
        target = items[2]

        # Get positions for precise movement
        source_rect = source.rect
        target_rect = target.rect

        # Perform drag with offset calculation
        actions.click_and_hold(source).perform()
        time.sleep(0.3)

        # Move to target position
        offset_y = target_rect['y'] - source_rect['y'] + 20
        actions.move_by_offset(0, offset_y).perform()
        time.sleep(0.3)

        actions.release().perform()

        print("List item reordered successfully!")

    finally:
        driver.quit()
```

3. 🎨 Modern Sliders and Range Controls

Price Range Slider

```
def test_price_range_slider():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        # Example with a shopping site with price filters
        driver.get("https://example-shopping-site.com")

        # Find price slider handles
        min_slider = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".price-slider .min-handle")))
        max_slider = driver.find_element(By.CSS_SELECTOR, ".price-slider .max-
handle")

        # Move minimum price slider
        actions.click_and_hold(min_slider).move_by_offset(50,
0).release().perform()
        time.sleep(1)

        # Move maximum price slider
        actions.click_and_hold(max_slider).move_by_offset(-30,
0).release().perform()
        time.sleep(1)

        # Verify price range changed
        price_display = driver.find_element(By.CSS_SELECTOR, ".price-range-
display")
        print(f"Price range: {price_display.text}")

    finally:
        driver.quit()
```

Image Zoom/Pan Control

```
def test_image_zoom_pan():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        # Product page with zoomable image
        driver.get("https://example-product-page.com")

        # Find zoomable image
        zoom_image = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".zoom-image")))

        # Hover over different parts of the image
        image_rect = zoom_image.rect

        # Move to top-left corner
        actions.move_to_element_with_offset(zoom_image, -image_rect['width']//4,
-image_rect['height']//4).perform()
        time.sleep(1)

        # Move to center
        actions.move_to_element(zoom_image).perform()
```

```
        time.sleep(1)

        # Move to bottom-right
        actions.move_to_element_with_offset(zoom_image, image_rect['width']//4,
image_rect['height']//4).perform()
        time.sleep(1)

        print("Image zoom navigation completed!")

    finally:
        driver.quit()
```

4. 📱 Mobile-First Responsive Elements

Mobile Menu Toggle

```
def test_mobile_menu():
    driver = webdriver.Chrome()

    # Set mobile viewport
    driver.set_window_size(375, 667)  # iPhone size

    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://responsive-website.com")

        # Find hamburger menu button
        hamburger = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".hamburger-menu")))

        # Click to open mobile menu
        actions.click(hamburger).perform()
        time.sleep(0.5)

        # Wait for menu animation to complete
        mobile_menu =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".mobile-nav")))

        # Click on a menu item
        menu_item = mobile_menu.find_element(By.CSS_SELECTOR,
"a[href*='products']")
        actions.click(menu_item).perform()

        print("Mobile menu navigation successful!")

    finally:
        driver.quit()
```

Touch-like Interactions

```
def test_touch_interactions():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://touch-demo-site.com")

        # Simulate swipe gesture (for carousels, etc.)
```

```
        carousel = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".carousel-container")))

        # Swipe left (drag from right to left)
        actions.click_and_hold(carousel).move_by_offset(-200,
0).release().perform()
        time.sleep(1)

        # Swipe right
        actions.click_and_hold(carousel).move_by_offset(200,
0).release().perform()
        time.sleep(1)

        print("Touch-like swipe gestures completed!")

    finally:
        driver.quit()
```

5. 🎮 Interactive Gaming Elements

HTML5 Canvas Interactions

```
def test_canvas_interactions():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://html5-game-demo.com")

        # Find canvas element
        canvas = wait.until(EC.element_to_be_clickable((By.TAG_NAME, "canvas")))

        # Click at specific coordinates on canvas
        canvas_rect = canvas.rect

        # Click at center of canvas
        actions.move_to_element(canvas).click().perform()
        time.sleep(0.5)

        # Click at top-left quadrant
        actions.move_to_element_with_offset(canvas, -canvas_rect['width']//4, -
canvas_rect['height']//4).click().perform()
        time.sleep(0.5)

        # Drag across canvas (for drawing apps)
        actions.move_to_element_with_offset(canvas, -50, -
50).click_and_hold().move_by_offset(100, 100).release().perform()

        print("Canvas interactions completed!")

    finally:
        driver.quit()
```

6. 🔍 Search with Autocomplete

Modern Search with Dropdown

```
def test_modern_search_autocomplete():
    driver = webdriver.Chrome()
```

```
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://google.com")

        # Find search box
        search_box = wait.until(EC.element_to_be_clickable((By.NAME, "q")))

        # Type search query gradually (to trigger autocomplete)
        search_query = "selenium webdriver"
        for char in search_query:
            search_box.send_keys(char)
            time.sleep(0.1)  # Simulate human typing

        # Wait for autocomplete suggestions
        time.sleep(1)

        # Use arrow keys to navigate suggestions
        from selenium.webdriver.common.keys import Keys
        actions.send_keys(Keys.ARROW_DOWN).perform()
        time.sleep(0.5)
        actions.send_keys(Keys.ARROW_DOWN).perform()
        time.sleep(0.5)

        # Select suggestion
        actions.send_keys(Keys.ENTER).perform()

        print("Autocomplete search completed!")

    finally:
        driver.quit()
```

7. 🎯 Advanced Real-Time Example: E-commerce Product Interaction

```
def test_complete_product_interaction():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        # Navigate to product page
        driver.get("https://example-ecommerce.com/product/123")

        # 1. Hover over product images for zoom
        main_image = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".product-main-image")))
        actions.move_to_element(main_image).perform()
        time.sleep(1)

        # 2. Click on thumbnail images
        thumbnails = driver.find_elements(By.CSS_SELECTOR, ".product-thumbnails
img")
        for thumb in thumbnails[:3]:  # Click first 3 thumbnails
            actions.click(thumb).perform()
            time.sleep(0.5)

        # 3. Interact with size selector (if it's a custom dropdown)
        size_dropdown = driver.find_element(By.CSS_SELECTOR, ".size-selector")
        actions.click(size_dropdown).perform()
```

```
        time.sleep(0.5)

        # Select size option
        size_option = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
"[data-size='L']")))
        actions.click(size_option).perform()

        # 4. Adjust quantity with +/- buttons
        plus_button = driver.find_element(By.CSS_SELECTOR, ".quantity-plus")
        actions.click(plus_button).click(plus_button).perform()  # Click twice

        # 5. Hover over "Add to Cart" for any tooltip
        add_to_cart = driver.find_element(By.CSS_SELECTOR, ".add-to-cart-btn")
        actions.move_to_element(add_to_cart).perform()
        time.sleep(0.5)

        # 6. Click Add to Cart
        actions.click(add_to_cart).perform()

        # 7. Handle cart popup/modal
        cart_modal =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".cart-modal")))

        # Continue shopping or proceed to checkout
        continue_shopping = cart_modal.find_element(By.CSS_SELECTOR, ".continue-
shopping")
        actions.click(continue_shopping).perform()

        print("Complete product interaction successful!")

    finally:
        driver.quit()
```

8. 🛠 Best Practices for Modern Elements

Handle Dynamic Loading

```
def handle_dynamic_content():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://dynamic-content-site.com")

        # Wait for initial content
        initial_content =
wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, ".content-
container")))

        # Scroll to trigger lazy loading
        driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")

        # Wait for new content to load
        wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, ".lazy-
loaded-content")))

        # Now interact with the loaded content
        lazy_element = driver.find_element(By.CSS_SELECTOR, ".lazy-loaded-item")
        actions.move_to_element(lazy_element).click().perform()

    finally:
```

```
        driver.quit()
```

Handle Animations and Transitions

```
def handle_animations():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://animated-site.com")

        # Trigger animation
        trigger_element =
wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR, ".animate-trigger")))
        actions.click(trigger_element).perform()

        # Wait for animation to complete
        time.sleep(2)  # Or use WebDriverWait with custom condition

        # Interact with animated element
        animated_element =
wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR, ".animated-element")))
        actions.click(animated_element).perform()

    finally:
        driver.quit()
```

9. 🕵 Common Issues and Solutions

Issue: Element Not Interactable

```
def handle_non_interactable_element():
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example.com")

        element = wait.until(EC.presence_of_element_located((By.ID, "target-
element")))

        # Scroll element into view
        driver.execute_script("arguments[0].scrollIntoView({behavior: 'smooth',
block: 'center'});", element)
        time.sleep(1)

        # Wait for element to be clickable
        wait.until(EC.element_to_be_clickable((By.ID, "target-element")))

        # Now perform action
        actions.click(element).perform()

    finally:
        driver.quit()
```

Issue: Timing Problems with Modern SPAs

```
def handle_spa_timing():
```

```
    driver = webdriver.Chrome()
    actions = ActionChains(driver)
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://react-app.com")

        # Wait for React/Vue/Angular app to load
        wait.until(lambda d: d.execute_script("return document.readyState") ==
"complete")

        # Wait for specific app state
        wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, "[data-
testid='app-loaded']")))

        # Now safe to interact
        button = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".interactive-button")))
        actions.click(button).perform()

    finally:
        driver.quit()
```

🎯 Practice Challenges

1. **Challenge 1**: Automate a complete shopping flow including product hover, size selection, and cart interaction
2. **Challenge 2**: Create a test for a drag-and-drop file upload interface
3. **Challenge 3**: Automate a multi-level navigation menu on a real e-commerce site
4. **Challenge 4**: Handle a complex form with custom dropdowns, sliders, and multi-select elements

🚀 Next Steps

These examples cover modern web interaction patterns you'll encounter in real-world testing. Practice with actual websites and adapt the patterns to your specific testing needs.

Ready to move to the next topic: **Handling Alerts and Popups**?

Selenium Alerts and Popups - Complete Guide

🚨 Types of Alerts and Popups

Modern web applications use different types of alerts and popups:

1. **JavaScript Alerts** - Browser native alerts
2. **Modal Dialogs** - HTML/CSS based popups
3. **Notification Popups** - Toast notifications
4. **Cookie Banners** - GDPR compliance popups
5. **Authentication Popups** - Login modals
6. **Confirmation Dialogs** - Action confirmations

1. 🔔 JavaScript Alerts (Browser Native)

JavaScript alerts are handled using Selenium's Alert class, not regular WebElement methods.

Basic Alert Handling

```
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
from selenium.common.exceptions import TimeoutException, NoAlertPresentException
import time

def test_basic_alert():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        # Navigate to a page with alerts
        driver.get("https://www.w3schools.com/js/tryit.asp?
filename=tryjs_alert")

        # Switch to the iframe containing the demo
        iframe = wait.until(EC.frame_to_be_available_and_switch_to_it((By.ID,
"iframeResult")))

        # Click button to trigger alert
        alert_button = wait.until(EC.element_to_be_clickable((By.XPATH,
"//button[text()='Try it']")))
        alert_button.click()

        # Wait for alert to be present
        wait.until(EC.alert_is_present())

        # Switch to alert
        alert = driver.switch_to.alert

        # Get alert text
        alert_text = alert.text
        print(f"Alert text: {alert_text}")

        # Accept the alert (click OK)
        alert.accept()

        print("Basic alert handled successfully!")

    finally:
        driver.quit()
```

Confirmation Dialog (OK/Cancel)

```
def test_confirmation_dialog():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        # Navigate to confirmation dialog demo
        driver.get("https://www.w3schools.com/js/tryit.asp?
filename=tryjs_confirm")

        # Switch to iframe
```

```
        iframe = wait.until(EC.frame_to_be_available_and_switch_to_it((By.ID,
"iframeResult")))

        # Trigger confirmation dialog
        confirm_button = wait.until(EC.element_to_be_clickable((By.XPATH,
"//button[text()='Try it']")))
        confirm_button.click()

        # Wait for alert
        wait.until(EC.alert_is_present())
        alert = driver.switch_to.alert

        print(f"Confirmation text: {alert.text}")

        # Accept (OK button)
        alert.accept()

        # Verify result
        result = driver.find_element(By.ID, "demo")
        print(f"Result after OK: {result.text}")

        # Test Cancel button
        confirm_button.click()
        wait.until(EC.alert_is_present())
        alert = driver.switch_to.alert

        # Dismiss (Cancel button)
        alert.dismiss()

        # Verify result
        result = driver.find_element(By.ID, "demo")
        print(f"Result after Cancel: {result.text}")

    finally:
        driver.quit()
```

Prompt Dialog (Input Text)

```
def test_prompt_dialog():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        # Navigate to prompt dialog demo
        driver.get("https://www.w3schools.com/js/tryit.asp?
filename=tryjs_prompt")

        # Switch to iframe
        iframe = wait.until(EC.frame_to_be_available_and_switch_to_it((By.ID,
"iframeResult")))

        # Trigger prompt dialog
        prompt_button = wait.until(EC.element_to_be_clickable((By.XPATH,
"//button[text()='Try it']")))
        prompt_button.click()

        # Wait for alert
        wait.until(EC.alert_is_present())
        alert = driver.switch_to.alert

        print(f"Prompt text: {alert.text}")

        # Send text to prompt
```

```
        alert.send_keys("Selenium Automation")

        # Accept the prompt
        alert.accept()

        # Verify result
        result = driver.find_element(By.ID, "demo")
        print(f"Result: {result.text}")

    finally:
        driver.quit()
```

2. 🎭 Modal Dialogs (HTML/CSS Based)

Modal dialogs are HTML elements that appear as overlays. They require different handling than JavaScript alerts.

Basic Modal Handling

```
def test_basic_modal():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        # Example with Bootstrap modal
        driver.get("https://getbootstrap.com/docs/5.0/components/modal/")

        # Click button to open modal
        modal_trigger = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
"[data-bs-toggle='modal']")))
        modal_trigger.click()

        # Wait for modal to be visible
        modal = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
".modal.show")))

        # Interact with modal content
        modal_title = modal.find_element(By.CSS_SELECTOR, ".modal-title")
        print(f"Modal title: {modal_title.text}")

        # Close modal using close button
        close_button = modal.find_element(By.CSS_SELECTOR, ".btn-close")
        close_button.click()

        # Wait for modal to disappear
        wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".modal.show")))

        print("Modal handled successfully!")

    finally:
        driver.quit()
```

Complex Modal with Form

```
def test_modal_with_form():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
```

```
        # Example with a login modal
        driver.get("https://example-site-with-login-modal.com")

        # Open login modal
        login_button = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".login-btn")))
        login_button.click()

        # Wait for modal to appear
        modal = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
".login-modal")))

        # Fill login form inside modal
        email_field = modal.find_element(By.CSS_SELECTOR, "input[type='email']")
        password_field = modal.find_element(By.CSS_SELECTOR,
"input[type='password']")

        email_field.send_keys("test@example.com")
        password_field.send_keys("password123")

        # Submit form
        submit_button = modal.find_element(By.CSS_SELECTOR, ".submit-btn")
        submit_button.click()

        # Handle success/error message
        try:
            success_message =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".success-
message")))
            print(f"Success: {success_message.text}")
        except TimeoutException:
            error_message = driver.find_element(By.CSS_SELECTOR, ".error-
message")
            print(f"Error: {error_message.text}")

    finally:
        driver.quit()
```

Modal with Backdrop Click to Close

```
def test_modal_backdrop_close():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-modal-site.com")

        # Open modal
        modal_trigger = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".open-modal")))
        modal_trigger.click()

        # Wait for modal
        modal = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
".modal")))

        # Click on backdrop to close (outside the modal content)
        backdrop = driver.find_element(By.CSS_SELECTOR, ".modal-backdrop")
        backdrop.click()

        # Wait for modal to close
        wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".modal")))
```

```
        print("Modal closed by backdrop click!")

    finally:
        driver.quit()
```

3. 🍪 Cookie Banners and GDPR Popups

Cookie banners are common in modern websites due to GDPR compliance.

Handle Cookie Banner

```
def test_cookie_banner():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-site-with-cookies.com")

        # Wait for cookie banner to appear
        try:
            cookie_banner =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".cookie-
banner")))

            # Accept all cookies
            accept_button = cookie_banner.find_element(By.CSS_SELECTOR,
".accept-all")
            accept_button.click()

            # Wait for banner to disappear
            wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".cookie-banner")))

            print("Cookie banner accepted!")

        except TimeoutException:
            print("No cookie banner found")

        # Continue with main test
        # ... rest of your test logic

    finally:
        driver.quit()
```

Advanced Cookie Preferences

```
def test_cookie_preferences():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-site.com")

        # Handle cookie banner
        try:
            cookie_banner = wait.until(EC.visibility_of_element_located((By.ID,
"cookie-banner")))

            # Click "Manage Preferences" instead of "Accept All"
```

```
            manage_btn = cookie_banner.find_element(By.CSS_SELECTOR, ".manage-
cookies")
            manage_btn.click()

            # Wait for preferences modal
            preferences_modal =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".cookie-
preferences")))

            # Toggle specific cookie categories
            analytics_toggle = preferences_modal.find_element(By.CSS_SELECTOR,
"input[data-category='analytics']")
            if not analytics_toggle.is_selected():
                analytics_toggle.click()

            marketing_toggle = preferences_modal.find_element(By.CSS_SELECTOR,
"input[data-category='marketing']")
            if marketing_toggle.is_selected():
                marketing_toggle.click()  # Disable marketing cookies

            # Save preferences
            save_button = preferences_modal.find_element(By.CSS_SELECTOR,
".save-preferences")
            save_button.click()

            # Wait for modal to close
            wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".cookie-preferences")))

            print("Cookie preferences saved!")

        except TimeoutException:
            print("No cookie banner found")

    finally:
        driver.quit()
```

4. 🔔 Toast Notifications

Toast notifications are temporary messages that appear and disappear automatically.

Handle Toast Notifications

```
def test_toast_notifications():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-app.com")

        # Perform action that triggers toast
        submit_button = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".submit-form")))
        submit_button.click()

        # Wait for toast to appear
        toast = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
".toast-notification")))

        # Get toast message
        toast_message = toast.text
        print(f"Toast message: {toast_message}")
```

```
        # Verify toast type (success, error, warning)
        if "success" in toast.get_attribute("class"):
            print("Success toast detected")
        elif "error" in toast.get_attribute("class"):
            print("Error toast detected")

        # Option 1: Wait for toast to auto-disappear
        wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR, ".toast-
notification")))

        # Option 2: Manually close toast (if close button exists)
        # close_button = toast.find_element(By.CSS_SELECTOR, ".close-toast")
        # close_button.click()

        print("Toast notification handled!")

    finally:
        driver.quit()
```

5. 🖥 Authentication Popups (HTTP Basic Auth)

Some websites use HTTP Basic Authentication popups.

Handle HTTP Basic Auth

```
def test_http_basic_auth():
    # Method 1: Include credentials in URL
    driver = webdriver.Chrome()

    try:
        # Include username:password in URL
        driver.get("https://username:password@example.com/secure-page")

        # Continue with test
        secure_content = driver.find_element(By.CSS_SELECTOR, ".secure-content")
        print(f"Secure content: {secure_content.text}")

    finally:
        driver.quit()

# Method 2: Using Chrome options for basic auth
def test_basic_auth_with_options():
    from selenium.webdriver.chrome.options import Options

    chrome_options = Options()
    # This approach works for some scenarios
    chrome_options.add_argument("--disable-web-security")
    chrome_options.add_argument("--allow-running-insecure-content")

    driver = webdriver.Chrome(options=chrome_options)

    try:
        driver.get("https://httpbin.org/basic-auth/user/pass")
        # Handle the auth popup using JavaScript
        driver.execute_script("""
            // This approach has limitations and may not work in all cases
            return true;
        """)

    finally:
        driver.quit()
```

6. 🎯 Real-World Complete Example

E-commerce Site with Multiple Popups

```
def test_ecommerce_popups():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example-ecommerce.com")

        # 1. Handle cookie banner
        try:
            cookie_accept = wait.until(EC.element_to_be_clickable((By.ID,
"accept-cookies")))
            cookie_accept.click()
        except TimeoutException:
            pass

        # 2. Handle newsletter popup
        try:
            newsletter_popup =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".newsletter-
popup")))
            close_newsletter = newsletter_popup.find_element(By.CSS_SELECTOR,
".close-popup")
            close_newsletter.click()
            wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".newsletter-popup")))
        except TimeoutException:
            pass

        # 3. Navigate and trigger age verification popup
        alcohol_section = driver.find_element(By.LINK_TEXT, "Wine & Spirits")
        alcohol_section.click()

        try:
            age_popup =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".age-
verification")))
            confirm_age = age_popup.find_element(By.CSS_SELECTOR, ".confirm-
age")
            confirm_age.click()
            wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".age-verification")))
        except TimeoutException:
            pass

        # 4. Add product to cart and handle cart popup
        product = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".product-item")))
        product.click()

        add_to_cart = wait.until(EC.element_to_be_clickable((By.CSS_SELECTOR,
".add-to-cart")))
        add_to_cart.click()

        # Handle cart confirmation popup
        cart_popup =
wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR, ".cart-popup")))
        continue_shopping = cart_popup.find_element(By.CSS_SELECTOR, ".continue-
shopping")
        continue_shopping.click()
```

```
        print("All popups handled successfully!")

    finally:
        driver.quit()
```

7. 🛠 Best Practices and Utilities

Reusable Alert Handler

```
class AlertHandler:
    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    def handle_alert(self, action="accept", text_to_send=None):
        """
        Handle JavaScript alerts
        action: 'accept', 'dismiss'
        text_to_send: text for prompt dialogs
        """
        try:
            self.wait.until(EC.alert_is_present())
            alert = self.driver.switch_to.alert

            alert_text = alert.text
            print(f"Alert detected: {alert_text}")

            if text_to_send:
                alert.send_keys(text_to_send)

            if action == "accept":
                alert.accept()
            elif action == "dismiss":
                alert.dismiss()

            return alert_text
        except TimeoutException:
            print("No alert present")
            return None

    def handle_modal(self, modal_selector, close_method="button"):
        """
        Handle HTML modal dialogs
        close_method: 'button', 'backdrop', 'escape'
        """
        try:
            modal =
self.wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
modal_selector)))

            if close_method == "button":
                close_btn = modal.find_element(By.CSS_SELECTOR, ".close, .btn-
close, [data-dismiss='modal']")
                close_btn.click()
            elif close_method == "backdrop":
                backdrop = self.driver.find_element(By.CSS_SELECTOR, ".modal-
backdrop")
                backdrop.click()
            elif close_method == "escape":
                from selenium.webdriver.common.keys import Keys
                modal.send_keys(Keys.ESCAPE)
```

```
            self.wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
modal_selector)))
            return True
        except TimeoutException:
            print(f"Modal {modal_selector} not found or couldn't be closed")
            return False

    def dismiss_common_popups(self):
        """Dismiss common popups like cookies, newsletters, etc."""
        common_popups = [
            (".cookie-banner .accept", "Cookie banner"),
            (".newsletter-popup .close", "Newsletter popup"),
            (".age-verification .confirm", "Age verification"),
            (".promo-popup .dismiss", "Promotional popup")
        ]

        for selector, name in common_popups:
            try:
                popup = WebDriverWait(self.driver,
2).until(EC.element_to_be_clickable((By.CSS_SELECTOR, selector)))
                popup.click()
                print(f"Dismissed {name}")
            except TimeoutException:
                continue

# Usage example
def test_with_alert_handler():
    driver = webdriver.Chrome()
    alert_handler = AlertHandler(driver)

    try:
        driver.get("https://example.com")

        # Dismiss common popups
        alert_handler.dismiss_common_popups()

        # Handle specific alert
        alert_handler.handle_alert("accept")

        # Handle specific modal
        alert_handler.handle_modal(".login-modal")

    finally:
        driver.quit()
```

8. 🕳 Common Issues and Solutions

Issue: Alert not found

```
def safe_alert_handling():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example.com")

        # Trigger action that might show alert
        button = driver.find_element(By.ID, "trigger-button")
        button.click()

        # Safe alert handling
```

```
        try:
            alert = wait.until(EC.alert_is_present())
            print(f"Alert text: {alert.text}")
            alert.accept()
        except TimeoutException:
            print("No alert appeared")
        except NoAlertPresentException:
            print("Alert was not present when trying to handle it")

    finally:
        driver.quit()
```

Issue: Modal not closing

```
def force_close_modal():
    driver = webdriver.Chrome()
    wait = WebDriverWait(driver, 10)

    try:
        driver.get("https://example.com")

        # Open modal
        trigger = driver.find_element(By.CSS_SELECTOR, ".open-modal")
        trigger.click()

        # Wait for modal
        modal = wait.until(EC.visibility_of_element_located((By.CSS_SELECTOR,
".modal")))

        # Try multiple ways to close
        try:
            # Method 1: Close button
            close_btn = modal.find_element(By.CSS_SELECTOR, ".close")
            close_btn.click()
        except:
            try:
                # Method 2: Escape key
                from selenium.webdriver.common.keys import Keys
                modal.send_keys(Keys.ESCAPE)
            except:
                # Method 3: JavaScript

driver.execute_script("document.querySelector('.modal').style.display = 'none'")

        # Verify modal is closed
        wait.until(EC.invisibility_of_element_located((By.CSS_SELECTOR,
".modal")))

    finally:
        driver.quit()
```

9. 🎯 Practice Exercises

1. **Exercise 1**: Create a test that handles all three types of JavaScript alerts (alert, confirm, prompt)
2. **Exercise 2**: Build a reusable function to handle cookie banners on different websites
3. **Exercise 3**: Test a shopping cart flow that includes multiple modal dialogs
4. **Exercise 4**: Handle toast notifications and verify their messages

Key Topics Covered:

1. JavaScript Alerts (Browser Native)

- Basic alerts (OK button)
- Confirmation dialogs (OK/Cancel)
- Prompt dialogs (text input)

2. HTML Modal Dialogs

- Bootstrap-style modals
- Login/form modals
- Backdrop click handling

3. Modern Web Popups

- Cookie banners (GDPR compliance)
- Newsletter popups
- Toast notifications
- Age verification popups

4. Advanced Scenarios

- HTTP Basic Authentication
- Multiple popup handling
- Reusable utility classes

🔥 Real-World Applications:

- **E-commerce sites**: Cart popups, age verification, cookie banners
- **Social media**: Login modals, notification toasts
- **News sites**: Newsletter subscriptions, cookie consent
- **Banking sites**: Security alerts, confirmation dialogs

💡 Key Takeaways:

1. **JavaScript alerts** use `driver.switch_to.alert`
2. **HTML modals** use regular WebElement methods
3. **Always use WebDriverWait** for popup timing
4. **Handle common popups** at the start of tests
5. **Create reusable utilities** for repeated popup patterns

🛠️ Best Practices from the Guide:

- Use try/except blocks for optional popups
- Create reusable AlertHandler class
- Handle multiple popup types in sequence
- Verify popup disappearance after handling

🚀 Practice Suggestions:

1. Try the JavaScript alert examples on W3Schools
2. Test cookie banner handling on real websites
3. Practice modal dialog interactions
4. Build your own AlertHandler utility class

Taking Screenshots with Selenium Python

Overview

Screenshots are essential for debugging test failures, creating test reports, and documenting application states during test execution.

Types of Screenshots

1. Full Page Screenshots

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

# Setup driver
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
driver.get("https://example.com")

# Take full page screenshot
driver.save_screenshot("full_page.png")

# Alternative method
screenshot = driver.get_screenshot_as_png()
with open("full_page_alt.png", "wb") as file:
    file.write(screenshot)

driver.quit()
```

2. Element-Specific Screenshots

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import base64

driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
driver.get("https://example.com")

# Find specific element
element = driver.find_element(By.ID, "header")

# Take screenshot of specific element
element.screenshot("element_screenshot.png")

# Alternative method using base64
element_png = element.screenshot_as_png
with open("element_base64.png", "wb") as file:
    file.write(element_png)
```

```
driver.quit()
```

Screenshot Methods Available

WebDriver Methods

- `save_screenshot(filename)` - Saves screenshot to file
- `get_screenshot_as_png()` - Returns screenshot as binary data
- `get_screenshot_as_base64()` - Returns screenshot as base64 string

WebElement Methods

- `screenshot(filename)` - Element screenshot to file
- `screenshot_as_png` - Element screenshot as binary data
- `screenshot_as_base64` - Element screenshot as base64 string

Practical Examples

Example 1: Screenshot on Test Failure

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

class TestWithScreenshots:
    def setup_method(self):
        self.driver =
webdriver.Chrome(service=Service(ChromeDriverManager().install()))

    def teardown_method(self):
        self.driver.quit()

    def take_screenshot_on_failure(self, test_name):
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"failure_{test_name}_{timestamp}.png"
        self.driver.save_screenshot(filename)
        print(f"Screenshot saved: {filename}")

    def test_login_failure(self):
        try:
            self.driver.get("https://example.com/login")
            # Test logic that might fail
            assert False, "Intentional failure for demo"
        except AssertionError:
            self.take_screenshot_on_failure("login_test")
            raise
```

Example 2: Before and After Screenshots

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
```

```
def test_form_submission():
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

    try:
        driver.get("https://example.com/form")

        # Screenshot before form submission
        driver.save_screenshot("before_form_submission.png")

        # Fill and submit form
        name_field = driver.find_element(By.ID, "name")
        name_field.send_keys("John Doe")

        submit_button = driver.find_element(By.ID, "submit")
        submit_button.click()

        # Screenshot after form submission
        driver.save_screenshot("after_form_submission.png")

    finally:
        driver.quit()
```

Example 3: Screenshot Utility Class

```
import os
from datetime import datetime
from selenium.webdriver.remote.webdriver import WebDriver
from selenium.webdriver.remote.webelement import WebElement

class ScreenshotUtils:
    def __init__(self, driver: WebDriver, screenshot_dir: str = "screenshots"):
        self.driver = driver
        self.screenshot_dir = screenshot_dir

        # Create screenshots directory if it doesn't exist
        if not os.path.exists(screenshot_dir):
            os.makedirs(screenshot_dir)

    def take_screenshot(self, name: str = None) -> str:
        """Take full page screenshot"""
        if not name:
            name = f"screenshot_{datetime.now().strftime('%Y%m%d_%H%M%S')}"

        filename = f"{name}.png"
        filepath = os.path.join(self.screenshot_dir, filename)

        self.driver.save_screenshot(filepath)
        return filepath

    def take_element_screenshot(self, element: WebElement, name: str = None) ->
str:
        """Take screenshot of specific element"""
        if not name:
            name = f"element_{datetime.now().strftime('%Y%m%d_%H%M%S')}"

        filename = f"{name}.png"
        filepath = os.path.join(self.screenshot_dir, filename)

        element.screenshot(filepath)
        return filepath

    def compare_screenshots(self, before_path: str, after_path: str) -> bool:
```

```
        """Basic screenshot comparison (you'd use PIL or similar for real
comparison)"""
        # This is a placeholder - you'd implement actual image comparison
        return os.path.exists(before_path) and os.path.exists(after_path)

# Usage example
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
screenshot_utils = ScreenshotUtils(driver)

driver.get("https://example.com")
screenshot_path = screenshot_utils.take_screenshot("homepage")
print(f"Screenshot saved to: {screenshot_path}")

driver.quit()
```

Best Practices

1. Organized Screenshot Management

```
import os
from datetime import datetime

class ScreenshotManager:
    def __init__(self, base_dir="test_screenshots"):
        self.base_dir = base_dir
        self.current_test_dir = None

    def setup_test_directory(self, test_name):
        """Create directory for current test"""
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        self.current_test_dir = os.path.join(self.base_dir,
f"{test_name}_{timestamp}")
        os.makedirs(self.current_test_dir, exist_ok=True)

    def capture_step(self, driver, step_name):
        """Capture screenshot for a test step"""
        if not self.current_test_dir:
            raise ValueError("Test directory not set up")

        filename = f"{step_name}.png"
        filepath = os.path.join(self.current_test_dir, filename)
        driver.save_screenshot(filepath)
        return filepath
```

2. Integration with Test Frameworks

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

@pytest.fixture
def driver_with_screenshots(request):
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

    def take_screenshot_on_failure():
        if request.node.rep_call.failed:
            test_name = request.node.name
            driver.save_screenshot(f"failure_{test_name}.png")

    request.addfinalizer(take_screenshot_on_failure)
```

```
    yield driver
    driver.quit()

@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    outcome = yield
    rep = outcome.get_result()
    setattr(item, "rep_" + rep.when, rep)
```

Advanced Screenshot Techniques

1. Full Page Screenshot (for long pages)

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
import time

def take_full_page_screenshot(driver, filename):
    # Get page dimensions
    page_width = driver.execute_script("return document.body.scrollWidth")
    page_height = driver.execute_script("return document.body.scrollHeight")

    # Set window size to capture full page
    driver.set_window_size(page_width, page_height)
    time.sleep(2)  # Wait for rendering

    # Take screenshot
    driver.save_screenshot(filename)
```

2. Screenshot with Timestamp Overlay

```
from PIL import Image, ImageDraw, ImageFont
from datetime import datetime

def add_timestamp_to_screenshot(screenshot_path):
    # Open the screenshot
    image = Image.open(screenshot_path)

    # Create drawing context
    draw = ImageDraw.Draw(image)

    # Add timestamp
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    # Try to use a better font, fall back to default if not available
    try:
        font = ImageFont.truetype("arial.ttf", 20)
    except:
        font = ImageFont.load_default()

    # Add text to image
    draw.text((10, 10), timestamp, fill="red", font=font)

    # Save modified image
    timestamped_path = screenshot_path.replace(".png", "_timestamped.png")
    image.save(timestamped_path)
    return timestamped_path
```

Common Issues and Solutions

Issue 1: Screenshots are blank or black

Solution: Add wait before taking screenshot

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

# Wait for page to load completely
WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.TAG_NAME, "body"))
)
driver.save_screenshot("screenshot.png")
```

Issue 2: Element not visible in screenshot

Solution: Scroll to element before screenshot

```
from selenium.webdriver.common.action_chains import ActionChains

element = driver.find_element(By.ID, "target-element")

# Scroll element into view
driver.execute_script("arguments[0].scrollIntoView(true);", element)
time.sleep(1)  # Brief pause for scrolling

# Take element screenshot
element.screenshot("element_screenshot.png")
```

Issue 3: Inconsistent screenshot sizes

Solution: Set fixed window size

```
# Set consistent window size
driver.set_window_size(1920, 1080)
driver.get(url)
driver.save_screenshot("consistent_size.png")
```

Practice Exercises

1. **Basic Screenshot Practice**
    - Navigate to different websites and take screenshots
    - Practice both full page and element screenshots
2. **Failure Screenshot Implementation**
    - Create a test that fails and automatically captures a screenshot
    - Implement screenshot capture in test teardown
3. **Screenshot Comparison**
    - Take before/after screenshots of a form submission
    - Implement basic image comparison logic
4. **Advanced Screenshot Utility**
    - Build a comprehensive screenshot utility class
    - Include features like timestamping, directory organization, and element highlighting

Next Topic Preview

The next topic will be **Scrolling with JavaScript**, where we'll learn how to programmatically scroll web pages to access elements and content that's not immediately visible.

Key Takeaways

- Screenshots are crucial for test debugging and reporting
- Multiple methods available for different screenshot needs
- Proper organization and naming conventions are important
- Integration with test frameworks enhances debugging capabilities
- Handle common issues like timing and element visibility

Key points covered:

- Full page and element-specific screenshots
- Different screenshot methods and formats
- Practical examples with test integration
- Best practices for screenshot organization
- Advanced techniques like timestamp overlays
- Common issues and their solutions
- Practice exercises

Scrolling with JavaScript in Selenium Python

Overview

JavaScript scrolling is essential for interacting with elements that are not immediately visible on the page, handling long pages, infinite scroll content, and ensuring elements are in the viewport before interaction.

Why JavaScript Scrolling?

Common Scenarios:

- Elements below the fold (not visible without scrolling)
- Infinite scroll pages (social media feeds, product listings)
- Modal dialogs that require scrolling
- Sticky headers/footers that may obstruct elements
- Pages with lazy-loaded content

Basic Scrolling Methods

1. Scroll to Bottom of Page

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

```
import time

driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
driver.get("https://example.com")

# Scroll to bottom of page
driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
time.sleep(2)  # Wait for content to load

driver.quit()
```

2. Scroll to Top of Page

```
# Scroll to top of page
driver.execute_script("window.scrollTo(0, 0);")
```

3. Scroll by Pixels

```
# Scroll down by 500 pixels
driver.execute_script("window.scrollBy(0, 500);")

# Scroll up by 300 pixels
driver.execute_script("window.scrollBy(0, -300);")

# Scroll horizontally (right by 200 pixels)
driver.execute_script("window.scrollBy(200, 0);")
```

4. Scroll to Specific Coordinates

```
# Scroll to specific x, y coordinates
driver.execute_script("window.scrollTo(500, 1000);")
```

Element-Based Scrolling

1. Scroll Element into View

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

# Find element
element = driver.find_element(By.ID, "target-element")

# Scroll element into view (top)
driver.execute_script("arguments[0].scrollIntoView(true);", element)

# Scroll element into view (bottom)
driver.execute_script("arguments[0].scrollIntoView(false);", element)
```

2. Scroll to Element with Offset

```
# Scroll to element with custom offset
element = driver.find_element(By.CLASS_NAME, "content")

# Get element position
element_y = element.location['y']

# Scroll to element with 100px offset from top
driver.execute_script(f"window.scrollTo(0, {element_y - 100});")
```

3. Smooth Scrolling to Element

```
# Smooth scroll to element
element = driver.find_element(By.ID, "section-2")

driver.execute_script("""
    arguments[0].scrollIntoView({
        behavior: 'smooth',
        block: 'center',
        inline: 'nearest'
    });
""", element)

time.sleep(2)  # Wait for smooth scroll to complete
```

Advanced Scrolling Techniques

1. Infinite Scroll Handler

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

class InfiniteScrollHandler:
    def __init__(self, driver):
        self.driver = driver

    def scroll_to_load_all_content(self, max_scrolls=10, scroll_pause=2):
        """Scroll down until no more content loads or max_scrolls reached"""
        last_height = self.driver.execute_script("return
document.body.scrollHeight")
        scrolls = 0

        while scrolls < max_scrolls:
            # Scroll to bottom
            self.driver.execute_script("window.scrollTo(0,
document.body.scrollHeight);")

            # Wait for new content to load
            time.sleep(scroll_pause)

            # Check if page height increased (new content loaded)
            new_height = self.driver.execute_script("return
document.body.scrollHeight")

            if new_height == last_height:
                print("No more content to load")
                break

            last_height = new_height
            scrolls += 1
            print(f"Scroll {scrolls}: New content loaded")

        return scrolls

# Usage
driver.get("https://example.com/infinite-scroll")
scroll_handler = InfiniteScrollHandler(driver)
total_scrolls = scroll_handler.scroll_to_load_all_content()
print(f"Total scrolls performed: {total_scrolls}")
```

2. Scroll Within Specific Container

```
# Scroll within a specific div or container
container = driver.find_element(By.CLASS_NAME, "scrollable-container")

# Scroll within container to bottom
driver.execute_script("arguments[0].scrollTop = arguments[0].scrollHeight",
container)

# Scroll within container by specific amount
driver.execute_script("arguments[0].scrollTop += 300", container)
```

3. Horizontal Scrolling

```
# Horizontal scroll in a container
horizontal_container = driver.find_element(By.CLASS_NAME, "horizontal-scroll")

# Scroll horizontally to the right
driver.execute_script("arguments[0].scrollLeft += 500", horizontal_container)

# Scroll horizontally to the left
driver.execute_script("arguments[0].scrollLeft -= 200", horizontal_container)

# Scroll to far right
driver.execute_script("arguments[0].scrollLeft = arguments[0].scrollWidth",
horizontal_container)
```

Practical Examples

Example 1: Complete Form with Scrolling

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
import time

def fill_long_form_with_scrolling():
    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))

    try:
        driver.get("https://example.com/long-form")

        # Form fields to fill
        form_fields = [
            {"id": "first-name", "value": "John"},
            {"id": "last-name", "value": "Doe"},
            {"id": "email", "value": "john.doe@example.com"},
            {"id": "phone", "value": "1234567890"},
            {"id": "address", "value": "123 Main St"},
            {"id": "city", "value": "Anytown"},
            {"id": "comments", "value": "This is a test comment"}
        ]

        for field in form_fields:
            # Find element
            element = driver.find_element(By.ID, field["id"])

            # Scroll element into view
            driver.execute_script("arguments[0].scrollIntoView({block:
'center'});", element)
```

```
            time.sleep(1)

            # Clear and fill field
            element.clear()
            element.send_keys(field["value"])

            print(f"Filled {field['id']} with {field['value']}")

        # Scroll to submit button and click
        submit_button = driver.find_element(By.ID, "submit")
        driver.execute_script("arguments[0].scrollIntoView({block: 'center'});",
submit_button)
        time.sleep(1)
        submit_button.click()

    finally:
        driver.quit()
```

Example 2: Social Media Feed Scraping

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

def scrape_social_feed(driver, target_posts=50):
    """Scrape posts from a social media feed with infinite scroll"""
    posts = []
    last_post_count = 0
    attempts = 0
    max_attempts = 10

    while len(posts) < target_posts and attempts < max_attempts:
        # Get current posts
        current_posts = driver.find_elements(By.CLASS_NAME, "post")

        if len(current_posts) > last_post_count:
            # New posts loaded, extract data
            for post in current_posts[last_post_count:]:
                try:
                    post_text = post.find_element(By.CLASS_NAME, "post-
content").text
                    post_author = post.find_element(By.CLASS_NAME, "post-
author").text

                    posts.append({
                        "author": post_author,
                        "content": post_text
                    })

                    if len(posts) >= target_posts:
                        break

                except Exception as e:
                    print(f"Error extracting post data: {e}")
                    continue

            last_post_count = len(current_posts)
            attempts = 0  # Reset attempts on successful load
        else:
            attempts += 1

        # Scroll to load more content
        driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
        time.sleep(2)
```

```
    return posts[:target_posts]

# Usage
driver.get("https://example-social-media.com")
posts = scrape_social_feed(driver, target_posts=20)
print(f"Scraped {len(posts)} posts")
```

Example 3: Table Navigation with Scrolling

```
def navigate_large_table(driver):
    """Navigate and interact with a large data table"""

    # Find table
    table = driver.find_element(By.ID, "data-table")

    # Get all rows
    rows = table.find_elements(By.TAG_NAME, "tr")

    print(f"Found {len(rows)} rows in table")

    # Process each row (scroll into view first)
    for i, row in enumerate(rows[1:], 1):  # Skip header row
        # Scroll row into view
        driver.execute_script("arguments[0].scrollIntoView({block: 'center'});",
row)
        time.sleep(0.5)

        # Get row data
        cells = row.find_elements(By.TAG_NAME, "td")
        row_data = [cell.text for cell in cells]

        print(f"Row {i}: {row_data}")

        # Example: Click edit button if it exists
        try:
            edit_button = row.find_element(By.CLASS_NAME, "edit-btn")
            edit_button.click()
            print(f"Clicked edit for row {i}")

            # Handle edit modal (example)
            time.sleep(1)
            # ... edit operations ...

        except Exception as e:
            print(f"No edit button found for row {i}")

        # Break after processing a few rows (for demo)
        if i >= 5:
            break

# Usage
driver.get("https://example.com/large-table")
navigate_large_table(driver)
```

Advanced Scrolling Utilities

Comprehensive Scrolling Utility Class

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

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

class ScrollUtils:
    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    def scroll_to_element(self, element, alignment="center", smooth=True):
        """Scroll element into view with options"""
        if smooth:
            script = f"""
                arguments[0].scrollIntoView({{
                    behavior: 'smooth',
                    block: '{alignment}',
                    inline: 'nearest'
                }});
            """
            self.driver.execute_script(script, element)
            time.sleep(1)  # Wait for smooth scroll
        else:
            self.driver.execute_script(f"arguments[0].scrollIntoView({{block:
'{alignment}'}});", element)

    def scroll_by_pixels(self, x=0, y=0, smooth=False):
        """Scroll by specific pixel amounts"""
        if smooth:
            script = f"""
                window.scrollBy({{
                    left: {x},
                    top: {y},
                    behavior: 'smooth'
                }});
            """
        else:
            script = f"window.scrollBy({x}, {y});"

        self.driver.execute_script(script)

        if smooth:
            time.sleep(1)

    def scroll_to_coordinates(self, x=0, y=0, smooth=False):
        """Scroll to specific coordinates"""
        if smooth:
            script = f"""
                window.scrollTo({{
                    left: {x},
                    top: {y},
                    behavior: 'smooth'
                }});
            """
        else:
            script = f"window.scrollTo({x}, {y});"

        self.driver.execute_script(script)

        if smooth:
            time.sleep(1)

    def get_scroll_position(self):
        """Get current scroll position"""
        x = self.driver.execute_script("return window.pageXOffset;")
        y = self.driver.execute_script("return window.pageYOffset;")
```

```
        return {"x": x, "y": y}

    def get_page_dimensions(self):
        """Get page dimensions"""
        width = self.driver.execute_script("return document.body.scrollWidth;")
        height = self.driver.execute_script("return
document.body.scrollHeight;")
        return {"width": width, "height": height}

    def is_element_in_viewport(self, element):
        """Check if element is currently visible in viewport"""
        script = """
            var rect = arguments[0].getBoundingClientRect();
            return (
                rect.top >= 0 &&
                rect.left >= 0 &&
                rect.bottom <= (window.innerHeight ||
document.documentElement.clientHeight) &&
                rect.right <= (window.innerWidth ||
document.documentElement.clientWidth)
            );
        """
        return self.driver.execute_script(script, element)

    def scroll_until_element_visible(self, locator, max_scrolls=10,
scroll_amount=300):
        """Scroll down until element becomes visible"""
        for i in range(max_scrolls):
            try:
                element = self.driver.find_element(*locator)
                if self.is_element_in_viewport(element):
                    return element
            except:
                pass

            self.scroll_by_pixels(y=scroll_amount)
            time.sleep(0.5)

        raise Exception(f"Element {locator} not found after {max_scrolls}
scrolls")

    def infinite_scroll(self, max_scrolls=20, scroll_pause=2, callback=None):
        """Handle infinite scroll pages"""
        last_height = self.driver.execute_script("return
document.body.scrollHeight")
        scrolls = 0

        while scrolls < max_scrolls:
            # Scroll to bottom
            self.scroll_to_coordinates(y=last_height, smooth=True)

            # Wait for content to load
            time.sleep(scroll_pause)

            # Execute callback if provided (for data extraction)
            if callback:
                callback(scrolls)

            # Check for new content
            new_height = self.driver.execute_script("return
document.body.scrollHeight")

            if new_height == last_height:
                break
```

```
            last_height = new_height
            scrolls += 1

        return scrolls

# Usage Example
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
scroll_utils = ScrollUtils(driver)

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

# Scroll to specific element smoothly
element = driver.find_element(By.ID, "footer")
scroll_utils.scroll_to_element(element, alignment="center", smooth=True)

# Check if element is in viewport
print(f"Element visible: {scroll_utils.is_element_in_viewport(element)}")

# Get page dimensions
dimensions = scroll_utils.get_page_dimensions()
print(f"Page size: {dimensions}")

driver.quit()
```

Common Scrolling Patterns

1. Lazy Loading Content

```
def handle_lazy_loading(driver, content_selector):
    """Handle lazy-loaded images/content"""
    last_loaded_count = 0

    while True:
        # Scroll to bottom
        driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
        time.sleep(2)

        # Count loaded content
        loaded_elements = driver.find_elements(By.CSS_SELECTOR,
content_selector)
        current_count = len(loaded_elements)

        if current_count == last_loaded_count:
            break  # No new content loaded

        last_loaded_count = current_count
        print(f"Loaded {current_count} items")

    return loaded_elements
```

2. Pagination with Scroll

```
def handle_scroll_pagination(driver, next_button_selector, max_pages=5):
    """Handle pagination that requires scrolling to next button"""
    page = 1

    while page <= max_pages:
        print(f"Processing page {page}")

        # Process current page content
```

```
        # ... your content processing logic ...

        try:
            # Find and scroll to next button
            next_button = driver.find_element(By.CSS_SELECTOR,
next_button_selector)
            driver.execute_script("arguments[0].scrollIntoView({block:
'center'});", next_button)
            time.sleep(1)

            # Click next button
            next_button.click()
            time.sleep(2)  # Wait for page load

            page += 1

        except Exception as e:
            print(f"No more pages or error: {e}")
            break
```

Best Practices

1. Always Wait After Scrolling

```
# Bad
driver.execute_script("window.scrollTo(0, 1000);")
element.click()  # Might fail if scroll hasn't completed

# Good
driver.execute_script("window.scrollTo(0, 1000);")
time.sleep(1)  # Wait for scroll to complete
element.click()
```

2. Check Element Visibility

```
# Ensure element is visible before interaction
element = driver.find_element(By.ID, "target")

if not scroll_utils.is_element_in_viewport(element):
    scroll_utils.scroll_to_element(element)

element.click()
```

3. Handle Sticky Headers/Footers

```
def scroll_with_sticky_header_offset(driver, element, header_height=60):
    """Scroll considering sticky header"""
    element_y = element.location['y']
    scroll_to_y = element_y - header_height

    driver.execute_script(f"window.scrollTo(0, {scroll_to_y});")
```

Practice Exercises

1. **Basic Scrolling**
    - Navigate to a long webpage and practice different scrolling methods
    - Scroll to specific elements and take screenshots

2. **Infinite Scroll Implementation**
    - Find a website with infinite scroll (social media, news site)
    - Implement infinite scroll handling to load all content
3. **Form Navigation**
    - Create a long form and practice scrolling to each field before filling
4. **Table Processing**
    - Find a large data table and implement row-by-row processing with scrolling

Next Topic Preview

The next topic will be **Dealing with Captchas & File Uploads**, where we'll explore strategies for handling CAPTCHAs in automated tests and different methods for uploading files through web forms.

Key Takeaways

- JavaScript scrolling is essential for accessing off-screen elements
- Different scrolling methods serve different purposes (smooth vs instant, element vs coordinate-based)
- Infinite scroll requires special handling with height comparison
- Always wait after scrolling operations
- Utility classes can make scrolling operations more maintainable
- Consider sticky elements and viewport constraints when scrolling

Key concepts covered:

- Basic scrolling methods (to bottom, top, by pixels, coordinates)
- Element-based scrolling with smooth animations
- Advanced techniques for infinite scroll and lazy loading
- Practical examples (forms, tables, social media feeds)
- Comprehensive utility class for reusable scrolling operations
- Best practices and common patterns
- Practice exercises

Ready for the next topic?

The next topic in Phase 3 is **Dealing with Captchas & File Uploads**. This will cover:

- Different approaches to handle CAPTCHAs in automation (bypass strategies, third-party services)
- Various file upload methods using Selenium
- Handling different types of file upload widgets
- Integration with external tools like AutoIT for complex uploads

Dealing with Captchas & File Uploads in Selenium Python

Overview

CAPTCHAs and file uploads are common challenges in web automation. This guide covers strategies for handling both, including bypass techniques, third-party services, and various file upload methods.

Part 1: Handling CAPTCHAs

Understanding CAPTCHA Types

1. Text-based CAPTCHAs

- Distorted text that users need to read
- Mathematical problems
- Simple word recognition

2. Image-based CAPTCHAs

- reCAPTCHA (I'm not a robot checkbox)
- Image selection (traffic lights, cars, etc.)
- Puzzle solving

3. Behavioral CAPTCHAs

- Mouse movement patterns
- Typing rhythm analysis
- Browser fingerprinting

CAPTCHA Handling Strategies

Strategy 1: Bypass CAPTCHAs in Test Environment

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

def bypass_captcha_in_test_env():
    """Best approach: Disable CAPTCHA in test environment"""

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

    try:
        # Use test environment URL where CAPTCHA is disabled
        driver.get("https://test.example.com/login")  # Test environment

        # Fill login form normally
        username = driver.find_element(By.ID, "username")
        password = driver.find_element(By.ID, "password")
```

```
        username.send_keys("testuser")
        password.send_keys("testpass")

        # No CAPTCHA to handle in test environment
        login_button = driver.find_element(By.ID, "login-btn")
        login_button.click()

        print("Login successful without CAPTCHA")

    finally:
        driver.quit()
```

Strategy 2: Using Test-specific Parameters

```
def login_with_test_mode():
    """Use special test parameters to bypass CAPTCHA"""

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

    try:
        # Add test parameter to URL
        driver.get("https://example.com/login?test_mode=true")

        # Or use special test credentials that bypass CAPTCHA
        username = driver.find_element(By.ID, "username")
        password = driver.find_element(By.ID, "password")

        # Special test account that bypasses CAPTCHA
        username.send_keys("test_no_captcha_user")
        password.send_keys("special_test_password")

        login_button = driver.find_element(By.ID, "login-btn")
        login_button.click()

    finally:
        driver.quit()
```

Strategy 3: Manual Intervention Approach

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

def handle_captcha_with_manual_intervention():
    """Pause automation for manual CAPTCHA solving"""

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

    try:
        driver.get("https://example.com/login")

        # Fill form up to CAPTCHA
        username = driver.find_element(By.ID, "username")
        password = driver.find_element(By.ID, "password")

        username.send_keys("user@example.com")
        password.send_keys("password123")

        # Check if CAPTCHA is present
        captcha_elements = driver.find_elements(By.CLASS_NAME, "captcha")

        if captcha_elements:
```

```
            print("CAPTCHA detected. Please solve it manually.")
            print("You have 60 seconds to solve the CAPTCHA...")

            # Wait for user to solve CAPTCHA manually
            input("Press Enter after solving the CAPTCHA...")

            # Or wait for a specific element that appears after CAPTCHA is
solved
            # WebDriverWait(driver, 60).until(
            #     EC.element_to_be_clickable((By.ID, "login-btn"))
            # )

        # Continue with form submission
        login_button = driver.find_element(By.ID, "login-btn")
        login_button.click()

    finally:
        driver.quit()
```

Strategy 4: Third-party CAPTCHA Solving Services

```
import requests
import time

class CaptchaSolver:
    """Example integration with CAPTCHA solving service"""

    def __init__(self, api_key):
        self.api_key = api_key
        self.base_url = "https://api.captcha-service.com"  # Example URL

    def solve_text_captcha(self, captcha_image_path):
        """Solve text-based CAPTCHA using third-party service"""

        # Upload image to service
        with open(captcha_image_path, 'rb') as image_file:
            files = {'file': image_file}
            data = {'key': self.api_key, 'method': 'post'}

            response = requests.post(f"{self.base_url}/in.php", files=files,
data=data)

            if response.text.startswith('OK|'):
                captcha_id = response.text.split('|')[1]

                # Poll for result
                for _ in range(30):  # Wait up to 30 attempts
                    time.sleep(2)
                    result_response = requests.get(
                        f"{self.base_url}/res.php?
key={self.api_key}&action=get&id={captcha_id}"
                    )

                    if result_response.text.startswith('OK|'):
                        return result_response.text.split('|')[1]
                    elif result_response.text == 'CAPCHA_NOT_READY':
                        continue
                    else:
                        break

        return None

    def solve_recaptcha(self, site_key, page_url):
```

```
        """Solve reCAPTCHA using third-party service"""

        # Submit reCAPTCHA task
        data = {
            'key': self.api_key,
            'method': 'userrecaptcha',
            'googlekey': site_key,
            'pageurl': page_url
        }

        response = requests.post(f"{self.base_url}/in.php", data=data)

        if response.text.startswith('OK|'):
            captcha_id = response.text.split('|')[1]

            # Poll for result (reCAPTCHA takes longer)
            for _ in range(60):  # Wait up to 60 attempts
                time.sleep(3)
                result_response = requests.get(
                    f"{self.base_url}/res.php?
key={self.api_key}&action=get&id={captcha_id}"
                )

                if result_response.text.startswith('OK|'):
                    return result_response.text.split('|')[1]
                elif result_response.text == 'CAPCHA_NOT_READY':
                    continue
                else:
                    break

        return None

def solve_captcha_with_service():
    """Example of using CAPTCHA solving service"""

    driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
    captcha_solver = CaptchaSolver("your_api_key_here")

    try:
        driver.get("https://example.com/login")

        # Fill form
        username = driver.find_element(By.ID, "username")
        password = driver.find_element(By.ID, "password")

        username.send_keys("user@example.com")
        password.send_keys("password123")

        # Handle text CAPTCHA
        captcha_image = driver.find_element(By.ID, "captcha-image")
        captcha_input = driver.find_element(By.ID, "captcha-input")

        # Take screenshot of CAPTCHA
        captcha_image.screenshot("captcha.png")

        # Solve CAPTCHA
        captcha_solution = captcha_solver.solve_text_captcha("captcha.png")

        if captcha_solution:
            captcha_input.send_keys(captcha_solution)
            print(f"CAPTCHA solved: {captcha_solution}")
        else:
            print("Failed to solve CAPTCHA")
            return
```

```
        # Submit form
        login_button = driver.find_element(By.ID, "login-btn")
        login_button.click()

    finally:
        driver.quit()
```

Strategy 5: reCAPTCHA Handling

```
def handle_recaptcha():
    """Handle Google reCAPTCHA"""

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

    try:
        driver.get("https://example.com/contact")

        # Fill form
        name = driver.find_element(By.ID, "name")
        email = driver.find_element(By.ID, "email")
        message = driver.find_element(By.ID, "message")

        name.send_keys("John Doe")
        email.send_keys("john@example.com")
        message.send_keys("Test message")

        # Handle reCAPTCHA
        recaptcha_frame = driver.find_element(By.CSS_SELECTOR,
"iframe[src*='recaptcha']")
        driver.switch_to.frame(recaptcha_frame)

        # Click "I'm not a robot" checkbox
        checkbox = driver.find_element(By.ID, "recaptcha-anchor")
        checkbox.click()

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

        # Wait for user to solve image challenge if it appears
        print("If image challenge appears, please solve it manually...")
        time.sleep(10)  # Give time for manual solving

        # Submit form
        submit_button = driver.find_element(By.ID, "submit")
        submit_button.click()

    finally:
        driver.quit()
```

Part 2: File Uploads

Understanding File Upload Types

1. Simple Input File Upload

- Standard HTML `<input type="file">` element
- Most common and easiest to handle

2. Drag and Drop Upload

- Modern UI with drag-and-drop zones
- Requires JavaScript or ActionChains

3. Custom Upload Widgets

- Flash-based uploaders (legacy)
- JavaScript-based custom widgets

File Upload Methods

Method 1: Basic File Upload with send_keys()

```
import os
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

def basic_file_upload():
    """Basic file upload using send_keys"""

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

    try:
        driver.get("https://example.com/upload")

        # Find file input element
        file_input = driver.find_element(By.ID, "file-upload")

        # Get absolute path to file
        file_path = os.path.abspath("test_files/sample.pdf")

        # Upload file
        file_input.send_keys(file_path)

        # Submit form
        upload_button = driver.find_element(By.ID, "upload-btn")
        upload_button.click()

        print(f"File uploaded: {file_path}")

    finally:
        driver.quit()
```

Method 2: Multiple File Upload

```
def multiple_file_upload():
    """Upload multiple files at once"""

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

    try:
        driver.get("https://example.com/multi-upload")

        # Find file input that accepts multiple files
        file_input = driver.find_element(By.ID, "multiple-files")

        # Prepare file paths
        file_paths = [
```

```
            os.path.abspath("test_files/document1.pdf"),
            os.path.abspath("test_files/image1.jpg"),
            os.path.abspath("test_files/spreadsheet1.xlsx")
        ]

        # Upload multiple files (separate paths with newlines)
        file_input.send_keys('\n'.join(file_paths))

        # Or upload files one by one
        # for file_path in file_paths:
        #     file_input.send_keys(file_path)

        upload_button = driver.find_element(By.ID, "upload-btn")
        upload_button.click()

        print(f"Uploaded {len(file_paths)} files")

    finally:
        driver.quit()
```

Method 3: Drag and Drop File Upload

```
from selenium.webdriver.common.action_chains import ActionChains
import tempfile
import os

def drag_drop_file_upload():
    """Handle drag and drop file upload"""

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

    try:
        driver.get("https://example.com/drag-drop-upload")

        # Find drop zone
        drop_zone = driver.find_element(By.ID, "drop-zone")

        # Create temporary file for testing
        with tempfile.NamedTemporaryFile(mode='w', suffix='.txt', delete=False)
as temp_file:
            temp_file.write("This is a test file for drag and drop upload")
            temp_file_path = temp_file.name

        try:
            # Method 1: Use JavaScript to simulate file drop
            file_input_script = """
                var target = arguments[0];
                var files = arguments[1];

                var input = document.createElement('input');
                input.type = 'file';
                input.style.display = 'none';
                input.multiple = true;

                target.appendChild(input);

                var event = new Event('change', {bubbles: true});
                Object.defineProperty(event, 'target', {value: input,
enumerable: true});

                input.files = files;
                input.dispatchEvent(event);
            """
```

```
            # Execute JavaScript to handle file drop
            with open(temp_file_path, 'rb') as file:
                driver.execute_script(file_input_script, drop_zone,
temp_file_path)

            print("File dropped successfully")

        finally:
            # Clean up temporary file
            os.unlink(temp_file_path)

    finally:
        driver.quit()
```

Method 4: Hidden File Input Upload

```
def hidden_file_input_upload():
    """Handle hidden file input elements"""

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

    try:
        driver.get("https://example.com/custom-upload")

        # Make hidden file input visible
        hidden_input = driver.find_element(By.CSS_SELECTOR, "input[type='file']
[style*='display: none']")

        # Make element visible using JavaScript
        driver.execute_script("arguments[0].style.display = 'block';",
hidden_input)

        # Now upload file normally
        file_path = os.path.abspath("test_files/document.pdf")
        hidden_input.send_keys(file_path)

        # Trigger upload process
        upload_trigger = driver.find_element(By.ID, "upload-trigger")
        upload_trigger.click()

        print("File uploaded through hidden input")

    finally:
        driver.quit()
```

Advanced File Upload Scenarios

File Upload with Progress Tracking

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

def upload_with_progress_tracking():
    """Upload file and track upload progress"""

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

    try:
        driver.get("https://example.com/upload-with-progress")
```

```
        # Upload file
        file_input = driver.find_element(By.ID, "file-input")
        file_path = os.path.abspath("test_files/large_file.zip")
        file_input.send_keys(file_path)

        # Start upload
        upload_button = driver.find_element(By.ID, "start-upload")
        upload_button.click()

        # Track progress
        progress_bar = driver.find_element(By.ID, "progress-bar")

        while True:
            progress = progress_bar.get_attribute("value")
            max_value = progress_bar.get_attribute("max")

            if progress and max_value:
                percentage = (int(progress) / int(max_value)) * 100
                print(f"Upload progress: {percentage:.1f}%")

                if percentage >= 100:
                    break

            time.sleep(1)

        # Wait for upload completion message
        WebDriverWait(driver, 30).until(
            EC.presence_of_element_located((By.ID, "upload-complete"))
        )

        print("File upload completed successfully")

    finally:
        driver.quit()
```

File Upload with Validation

```
def upload_with_validation():
    """Handle file upload with client-side validation"""

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

    try:
        driver.get("https://example.com/validated-upload")

        # Test invalid file type first
        file_input = driver.find_element(By.ID, "file-input")
        invalid_file = os.path.abspath("test_files/invalid.exe")

        try:
            file_input.send_keys(invalid_file)
            upload_button = driver.find_element(By.ID, "upload-btn")
            upload_button.click()

            # Check for error message
            error_message = WebDriverWait(driver, 5).until(
                EC.presence_of_element_located((By.CLASS_NAME, "error-message"))
            )
            print(f"Validation error: {error_message.text}")

        except Exception as e:
            print("No validation error found")
```

```
        # Clear and upload valid file
        file_input.clear()
        valid_file = os.path.abspath("test_files/document.pdf")
        file_input.send_keys(valid_file)

        upload_button.click()

        # Wait for success message
        success_message = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "success-message"))
        )
        print(f"Upload success: {success_message.text}")

    finally:
        driver.quit()
```

Using External Tools for Complex Uploads

Method 1: AutoIT Integration (Windows Only)

```
import subprocess
import time

def upload_with_autoit():
    """Use AutoIT for complex file upload dialogs"""

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

    try:
        driver.get("https://example.com/flash-upload")

        # Click upload button that opens file dialog
        upload_button = driver.find_element(By.ID, "flash-upload-btn")
        upload_button.click()

        # Use AutoIT script to handle file dialog
        # AutoIT script should be compiled to .exe
        autoit_script = "file_upload.exe"  # Compiled AutoIT script
        file_path = r"C:\path\to\your\file.pdf"

        # Execute AutoIT script with file path as parameter
        subprocess.run([autoit_script, file_path])

        print("File uploaded using AutoIT")

    finally:
        driver.quit()

# Example AutoIT script content (save as .au3 and compile):
"""
; AutoIT script for file upload
WinWaitActive("Open") ; Wait for file dialog
Send($CmdLine[1]) ; Send file path from command line
Send("{ENTER}") ; Press Enter to confirm
"""
```

Method 2: PyAutoGUI for File Dialogs

```
import pyautogui
import time

def upload_with_pyautogui():
```

```
    """Use PyAutoGUI to handle file dialogs"""

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

    try:
        driver.get("https://example.com/file-dialog-upload")

        # Click button that opens file dialog
        browse_button = driver.find_element(By.ID, "browse-btn")
        browse_button.click()

        # Wait for file dialog to open
        time.sleep(2)

        # Type file path in dialog
        file_path = r"C:\path\to\your\file.pdf"
        pyautogui.write(file_path)

        # Press Enter to confirm
        pyautogui.press('enter')

        # Wait and click upload button
        time.sleep(1)
        upload_button = driver.find_element(By.ID, "upload-btn")
        upload_button.click()

        print("File uploaded using PyAutoGUI")

    finally:
        driver.quit()
```

File Upload Utility Class

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

class FileUploadUtils:
    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 30)

    def upload_file(self, file_input_locator, file_path,
upload_button_locator=None):
        """Basic file upload with optional upload button"""

        # Ensure file exists
        if not os.path.exists(file_path):
            raise FileNotFoundError(f"File not found: {file_path}")

        # Find file input
        file_input = self.driver.find_element(*file_input_locator)

        # Upload file
        absolute_path = os.path.abspath(file_path)
        file_input.send_keys(absolute_path)

        # Click upload button if provided
        if upload_button_locator:
            upload_button = self.wait.until(
                EC.element_to_be_clickable(upload_button_locator)
```

```
            )
            upload_button.click()

        return True

    def upload_multiple_files(self, file_input_locator, file_paths,
upload_button_locator=None):
        """Upload multiple files"""

        # Verify all files exist
        for file_path in file_paths:
            if not os.path.exists(file_path):
                raise FileNotFoundError(f"File not found: {file_path}")

        # Find file input
        file_input = self.driver.find_element(*file_input_locator)

        # Upload files
        absolute_paths = [os.path.abspath(path) for path in file_paths]
        file_input.send_keys('\n'.join(absolute_paths))

        # Click upload button if provided
        if upload_button_locator:
            upload_button = self.wait.until(
                EC.element_to_be_clickable(upload_button_locator)
            )
            upload_button.click()

        return len(file_paths)

    def wait_for_upload_completion(self, success_locator, timeout=60):
        """Wait for upload to complete"""

        try:
            success_element = WebDriverWait(self.driver, timeout).until(
                EC.presence_of_element_located(success_locator)
            )
            return success_element.text
        except Exception as e:
            print(f"Upload may have failed or timed out: {e}")
            return None

    def handle_upload_with_progress(self, file_input_locator, file_path,
                                    upload_button_locator, progress_locator):
        """Handle file upload with progress tracking"""

        # Upload file
        self.upload_file(file_input_locator, file_path, upload_button_locator)

        # Track progress
        progress_element = self.driver.find_element(*progress_locator)

        while True:
            progress_text = progress_element.text
            print(f"Upload progress: {progress_text}")

            # Check if complete (adjust condition based on your UI)
            if "100%" in progress_text or "Complete" in progress_text:
                break

            time.sleep(1)

        return True
```

```
# Usage example
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
upload_utils = FileUploadUtils(driver)

try:
    driver.get("https://example.com/upload")

    # Simple upload
    upload_utils.upload_file(
        (By.ID, "file-input"),
        "test_files/document.pdf",
        (By.ID, "upload-btn")
    )

    # Wait for completion
    result = upload_utils.wait_for_upload_completion((By.CLASS_NAME, "success-
message"))
    print(f"Upload result: {result}")

finally:
    driver.quit()
```

Best Practices

1. File Management

```
import tempfile
import os

def create_test_files():
    """Create test files for upload testing"""

    test_dir = "test_files"
    os.makedirs(test_dir, exist_ok=True)

    # Create different file types
    files = {
        "text_file.txt": "This is a test text file",
        "csv_file.csv": "Name,Age,City\nJohn,25,NYC\nJane,30,LA",
        "json_file.json": '{"name": "test", "type": "file"}'
    }

    created_files = []

    for filename, content in files.items():
        filepath = os.path.join(test_dir, filename)
        with open(filepath, 'w') as f:
            f.write(content)
        created_files.append(filepath)

    return created_files
```

2. Error Handling

```
def robust_file_upload(driver, file_path, max_retries=3):
    """Robust file upload with retry logic"""

    for attempt in range(max_retries):
        try:
            file_input = driver.find_element(By.ID, "file-input")
            file_input.send_keys(os.path.abspath(file_path))
```

```
            upload_button = driver.find_element(By.ID, "upload-btn")
            upload_button.click()

            # Wait for success
            WebDriverWait(driver, 30).until(
                EC.presence_of_element_located((By.CLASS_NAME, "upload-
success"))
            )

            print(f"Upload successful on attempt {attempt + 1}")
            return True

        except Exception as e:
            print(f"Upload attempt {attempt + 1} failed: {e}")
            if attempt < max_retries - 1:
                time.sleep(2)  # Wait before retry
                continue
            else:
                raise

    return False
```

Practice Exercises

1. **Basic Upload Practice**
    - Create test files of different formats
    - Practice uploading single and multiple files
2. **CAPTCHA Handling**
    - Find websites with different CAPTCHA types
    - Implement manual intervention approach
3. **Advanced Upload Scenarios**
    - Handle progress tracking
    - Implement validation error handling
4. **Integration Testing**
    - Combine file upload with form submission
    - Test upload limits and file type restrictions

Next Topic Preview

The next topic will be **Headless Browser Testing**, where we'll learn how to run Selenium tests without a visible browser window, configure headless options, and optimize performance for CI/CD environments.

Key Takeaways

- Best approach for CAPTCHAs is to disable them in test environments
- Use `send_keys()` for most file upload scenarios
- Handle hidden file inputs by making them visible with JavaScript
- Consider third-party services for CAPTCHA solving when necessary

- Implement robust error handling and retry logic
- Create reusable utility classes for common upload patterns
- Always use absolute file paths for uploads
- Test with various file types and sizes

Phase 3: Cross-Browser Testing & Parallel Execution

Overview

Cross-browser testing ensures your web application works consistently across different browsers and browser versions. Parallel execution allows you to run multiple tests simultaneously, significantly reducing test execution time.

4.1 Cross-Browser Testing Fundamentals

Understanding Browser Compatibility

- **Why Cross-Browser Testing Matters**: Different browsers render web pages differently and support various web standards
- **Common Browser Differences**: CSS rendering, JavaScript execution, HTML5 support, performance variations
- **Market Share Considerations**: Focus testing efforts on browsers with higher market share

Browser Support Matrix

Create a testing matrix covering:

- **Desktop Browsers**: Chrome, Firefox, Safari, Edge, Internet Explorer (legacy)
- **Mobile Browsers**: Chrome Mobile, Safari Mobile, Samsung Internet
- **Browser Versions**: Current, previous major version, and legacy versions if needed

4.2 Setting Up Multiple WebDrivers

WebDriver Configuration for Different Browsers

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options as ChromeOptions
from selenium.webdriver.firefox.options import Options as FirefoxOptions
from selenium.webdriver.edge.options import Options as EdgeOptions

def get_driver(browser_name):
    if browser_name.lower() == 'chrome':
        chrome_options = ChromeOptions()
        chrome_options.add_argument('--headless')  # Optional
        return webdriver.Chrome(options=chrome_options)

    elif browser_name.lower() == 'firefox':
        firefox_options = FirefoxOptions()
        firefox_options.add_argument('--headless')  # Optional
```

```
        return webdriver.Firefox(options=firefox_options)

    elif browser_name.lower() == 'edge':
        edge_options = EdgeOptions()
        edge_options.add_argument('--headless')  # Optional
        return webdriver.Edge(options=edge_options)

    else:
        raise ValueError(f"Browser {browser_name} not supported")
```

Browser-Specific Configurations

```
# Chrome-specific options
chrome_options = ChromeOptions()
chrome_options.add_argument('--disable-gpu')
chrome_options.add_argument('--no-sandbox')
chrome_options.add_argument('--window-size=1920,1080')

# Firefox-specific options
firefox_options = FirefoxOptions()
firefox_options.set_preference('dom.webnotifications.enabled', False)
firefox_options.set_preference('media.volume_scale', '0.0')
```

4.3 Parameterized Testing for Multiple Browsers

Using pytest for Cross-Browser Testing

```
import pytest
from selenium import webdriver

@pytest.fixture(params=['chrome', 'firefox', 'edge'])
def driver(request):
    browser = request.param
    driver = get_driver(browser)
    yield driver
    driver.quit()

def test_login_across_browsers(driver):
    driver.get("https://example.com/login")
    # Your test logic here
    assert "Login" in driver.title
```

Creating Browser Factory Pattern

```
class BrowserFactory:
    @staticmethod
    def create_driver(browser_name, headless=False):
        browser_name = browser_name.lower()

        if browser_name == 'chrome':
            options = ChromeOptions()
            if headless:
                options.add_argument('--headless')
            return webdriver.Chrome(options=options)

        elif browser_name == 'firefox':
            options = FirefoxOptions()
            if headless:
                options.add_argument('--headless')
            return webdriver.Firefox(options=options)
```

```
        else:
            raise ValueError(f"Unsupported browser: {browser_name}")
```

4.4 Parallel Test Execution

Introduction to Parallel Testing

- **Benefits**: Reduced execution time, better resource utilization, faster feedback
- **Challenges**: Test isolation, resource conflicts, result aggregation
- **When to Use**: Large test suites, regression testing, CI/CD pipelines

Using pytest-xdist for Parallel Execution

```
# Install pytest-xdist
pip install pytest-xdist

# Run tests in parallel
pytest -n 4  # Run with 4 parallel processes
pytest -n auto  # Automatically detect number of CPUs
```

Parallel Execution with TestNG (Java)

```
<!-- testng.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<suite name="ParallelSuite" parallel="methods" thread-count="3">
    <test name="CrossBrowserTest">
        <classes>
            <class name="com.example.LoginTest"/>
        </classes>
    </test>
</suite>
```

4.5 Managing WebDriver Instances in Parallel

Thread-Safe WebDriver Management

```
import threading
from selenium import webdriver

class ThreadLocalWebDriver:
    def __init__(self):
        self._driver = threading.local()

    def get_driver(self, browser_name):
        if not hasattr(self._driver, 'instance'):
            self._driver.instance = get_driver(browser_name)
        return self._driver.instance

    def quit_driver(self):
        if hasattr(self._driver, 'instance'):
            self._driver.instance.quit()
            delattr(self._driver, 'instance')

# Usage
driver_manager = ThreadLocalWebDriver()
```

```
def test_parallel_execution():
    driver = driver_manager.get_driver('chrome')
    # Test logic here
    driver_manager.quit_driver()
```

Pool-Based Driver Management

```
from concurrent.futures import ThreadPoolExecutor
import queue

class DriverPool:
    def __init__(self, browser_name, pool_size=3):
        self.browser_name = browser_name
        self.pool = queue.Queue()

        # Initialize driver pool
        for _ in range(pool_size):
            driver = get_driver(browser_name)
            self.pool.put(driver)

    def get_driver(self):
        return self.pool.get()

    def return_driver(self, driver):
        self.pool.put(driver)

    def cleanup(self):
        while not self.pool.empty():
            driver = self.pool.get()
            driver.quit()
```

4.6 Best Practices for Cross-Browser Testing

Test Design Considerations

1. **Browser-Agnostic Locators**: Use locators that work across browsers
2. **Timing Considerations**: Different browsers have different loading speeds
3. **Feature Detection**: Check for browser-specific features before using them
4. **Fallback Strategies**: Implement alternative approaches for unsupported features

Common Cross-Browser Issues

- **Element Visibility**: Different rendering engines may hide/show elements differently
- **JavaScript Execution**: Timing differences in JavaScript execution
- **CSS Rendering**: Layout differences between browsers
- **File Upload Behavior**: Different file dialog implementations

Handling Browser-Specific Code

```
def get_browser_name(driver):
    return driver.capabilities['browserName'].lower()

def handle_browser_specific_behavior(driver):
    browser_name = get_browser_name(driver)

    if browser_name == 'firefox':
        # Firefox-specific handling
```

```
        driver.execute_script("arguments[0].click();", element)
    elif browser_name == 'safari':
        # Safari-specific handling
        ActionChains(driver).click(element).perform()
    else:
        # Default behavior
        element.click()
```

4.7 Reporting and Analysis

Cross-Browser Test Reports

- **Aggregated Results**: Combine results from all browsers
- **Browser-Specific Failures**: Identify browser-specific issues
- **Performance Comparison**: Compare execution times across browsers

Sample Report Structure

```
class CrossBrowserReport:
    def __init__(self):
        self.results = {}

    def add_result(self, browser, test_name, status, duration):
        if browser not in self.results:
            self.results[browser] = []

        self.results[browser].append({
            'test': test_name,
            'status': status,
            'duration': duration
        })

    def generate_summary(self):
        summary = {}
        for browser, tests in self.results.items():
            passed = len([t for t in tests if t['status'] == 'PASSED'])
            total = len(tests)
            summary[browser] = f"{passed}/{total} passed"
        return summary
```

Key Takeaways

1. **Cross-browser testing is essential** for ensuring consistent user experience
2. **Parallel execution significantly reduces** test execution time
3. **Proper driver management** is crucial for stable parallel execution
4. **Browser-specific handling** may be needed for certain scenarios
5. **Comprehensive reporting** helps identify browser-specific issues

---

Practice Exercises:

1. Set up cross-browser testing for a simple login scenario across Chrome, Firefox, and Edge

2. Implement parallel execution using pytest-xdist
3. Create a browser factory pattern for managing different WebDriver instances
4. Design a test that handles browser-specific behaviors
5. Generate a cross-browser test report

Parallel Execution. This phase covers:

- Setting up multiple WebDriver instances for different browsers
- Parameterized testing across browsers using pytest
- Implementing parallel test execution to reduce runtime
- Thread-safe WebDriver management
- Best practices for handling cross-browser compatibility issues
- Reporting and analysis of cross-browser test results

The content includes practical code examples and real-world scenarios you'll encounter when testing across different browsers and running tests in parallel.