Selenium In-Built and External Methods for GUI Testing

Here’s a comprehensive list of Selenium's in-built and external functions/methods that are commonly used for GUI script creation:

1. In-Built Methods in Selenium WebDriver

Browser Control Methods

- get(url) – Open a specific URL.
- close() – Close the current window.
- quit() – Close all browser windows and end the WebDriver session.
- maximize_window() – Maximize the browser window.
- minimize_window() – Minimize the browser window.
- back() – Navigate to the previous page.
- forward() – Navigate to the next page.
- refresh() – Refresh the current page.

Finding Elements Methods

- find_element_by_id(id) – Locate an element by its ID.
- find_element_by_name(name) – Locate an element by its name.
- find_element_by_class_name(class_name) – Locate an element by its class name.
- find_element_by_tag_name(tag_name) – Locate an element by its tag name.
- find_element_by_css_selector(css_selector) – Locate an element using a CSS selector.
- find_element_by_xpath(xpath) – Locate an element using XPath.
- find_elements_by_* – Variants to locate multiple elements.

Element Interaction Methods

- click() – Click on an element.
- send_keys(keys) – Type text into an input field.
- clear() – Clear the text from an input field.
- submit() – Submit a form.
- get_attribute(attribute_name) – Retrieve an attribute's value of an element.
- get_text() – Get the text of an element.

Browser Information Methods

- current_url – Get the current page URL.
- title – Get the page title.
- page_source – Get the HTML source of the page.
- window_handles – Get a list of all open window handles.
- current_window_handle – Get the current window's handle.

Handling Cookies Methods

- get_cookies() – Get all cookies in the session.
- get_cookie(name) – Get a specific cookie by name.
- add_cookie(cookie_dict) – Add a cookie to the session.
- delete_cookie(name) – Delete a specific cookie by name.
- delete_all_cookies() – Delete all cookies.

Alerts and Pop-ups Methods

- switch_to.alert – Switch focus to an alert.
- alert.accept() – Accept the alert.
- alert.dismiss() – Dismiss the alert.
- alert.send_keys(text) – Send text to the alert prompt.

Frames and Windows Handling

- switch_to.frame(frame_reference) – Switch to a specific frame.
- switch_to.default_content() – Switch back to the main content from a frame.

- switch_to.window(window_handle) – Switch to another browser window.

Waits (Implicit and Explicit)

- implicitly_wait(seconds) – Set an implicit wait for element location.
- WebDriverWait(driver, timeout) – Explicit wait for an element or condition.
- until() – Wait until a certain condition is met.
- until_not() – Wait until a certain condition is no longer true.

Actions for Keyboard and Mouse Events

- ActionChains(driver) – Create a chain of keyboard or mouse actions.
- move_to_element(element) – Move the mouse to a specific element.
- click_and_hold(element) – Click and hold on an element.
- double_click(element) – Double-click an element.
- context_click(element) – Perform a right-click.
- send_keys_to_element(element, keys) – Send keys to an element.

Screenshot Methods

- get_screenshot_as_file(filename) – Save a screenshot as a file.
- get_screenshot_as_base64() – Return the screenshot in base64 format.

JavaScript Execution

- execute_script(script, *args) – Execute JavaScript on the page.

---

2. External Libraries and Functions for Selenium GUI Scripting

1. Selenium Grid

Selenium Grid allows parallel testing across multiple browsers and machines.

- Remote(webdriver_url, capabilities) – Connects to a remote WebDriver for distributed testing.

- desired_capabilities – Defines browser capabilities for remote testing.

2. Page Object Model (POM) Support

This is a design pattern used in Selenium for creating object-oriented page classes.

- Custom Classes/Methods for Page Actions – Define actions for each web page in separate classes.
- page_factory.PageFactory.init_elements(driver, this) – Initialize page elements in the Page Object Model.

3. WebDriverWait & Expected Conditions (Explicit Waits)

- WebDriverWait(driver, timeout).until(condition) – Wait until a condition is met.
- ExpectedConditions.element_to_be_clickable(locator) – Wait until an element is clickable.
- ExpectedConditions.presence_of_element_located(locator) – Wait until an element is present in the DOM.
- ExpectedConditions.visibility_of_element_located(locator) – Wait until an element is visible.

4. Test Frameworks Integration

Selenium can be integrated with various test frameworks like:

- JUnit (for Java)
  - @Test – Annotation to mark a test method.
  - Assert.assertTrue(condition) – Validate a test condition.
- **TestNG** * (for Java)
  - @Test – Annotation for test methods.
  - @BeforeClass, @AfterClass – Setup and teardown methods.
- PyTest (for Python)
  - assert condition – Validate test conditions in Python.
  - pytest.fixture() – Setup and teardown for test cases.

5. Browser-Specific Drivers

- Chrome WebDriver:
  - ChromeOptions() – Customize Chrome browser options.
  - add_argument('--headless') – Run Chrome in headless mode.
- Firefox WebDriver:
  - FirefoxOptions() – Customize Firefox browser options.
  - add_argument('--headless') – Run Firefox in headless mode.

6. Selenium External Tools for GUI Testing

- Sikuli – Visual-based testing tool, especially for image-based GUI automation.
  - find(image) – Locate an image on the screen.
  - click(image) – Click an image on the screen.
  - Integration with Selenium can be done to handle non-web-based elements.
- AutoIt – Automates Windows GUI interaction.
  - AutoItX.send(text) – Sends keystrokes to the application.
  - AutoItX.mouse_click(button, x, y) – Simulate mouse clicks.
  - Useful for handling file upload/download dialogs, windows, etc., alongside Selenium.

7. Data-Driven Testing Libraries

Selenium can be used with data-driven testing frameworks:

- Apache POI (Java) – For reading and writing Excel files.
  - Workbook workbook = WorkbookFactory.create(file) – Read Excel file.
  - Sheet sheet = workbook.getSheet(sheetName) – Access a specific Excel sheet.
- Pandas (Python) – For working with data in Python.
  - pandas.read_excel(file) – Read data from an Excel file for data-driven testing.

8. Logging and Reporting Libraries

- Log4j (Java) – For generating logs in Java-based Selenium scripts.
    - Logger.getLogger(name) – Get the logger instance.
    - logger.info(message) – Log informational messages.
- Allure – Generates test reports with detailed visualizations and steps.

9. Headless Browser Libraries

- HTMLUnitDriver – Lightweight browser simulator.
    - driver = new HtmlUnitDriver() – Start a headless HTMLUnit browser.
- PhantomJS (Deprecated) – Formerly used for headless browser automation.
    - driver = new PhantomJSDriver() – Start a headless PhantomJS browser.

These in-built and external libraries, along with their methods and functions, are widely used in Selenium for creating robust GUI test scripts.

Essential TestNG Annotations for Selenium Automation

This guide covers key **internal and external TestNG annotations** used in Selenium test automation. It explains how these annotations help manage test execution flow, data-driven tests, retries, and reporting for efficient Selenium test management.

1. TestNG Annotations

1. @Test

This annotation marks a method as a test method.

Example:

java

Copy code

```
@Test

public void testGoogleTitle() {

    WebDriver driver = new ChromeDriver();

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

    Assert.assertEquals(driver.getTitle(), "Google");

    driver.quit();

}
```

2. @BeforeSuite

Executed before any of the tests in the suite are run. Typically used for setup tasks.

Example:

java

Copy code

```
@BeforeSuite

public void setUpSuite() {

    System.out.println("Setting up the suite");

}
```

3. @AfterSuite

Executed after all the tests in the suite have run. Used for cleanup tasks.

Example:

java

Copy code

```
@AfterSuite

public void tearDownSuite() {

    System.out.println("Tearing down the suite");

}
```

4. @BeforeTest

Executed before any test cases that belong to the <test> tag in the XML file.

Example:

java

Copy code

```
@BeforeTest

public void beforeTest() {

    System.out.println("Before Test");

}
```

5. @AfterTest

Executed after all test cases that belong to the <test> tag have been executed.

Example:

java

Copy code

```
@AfterTest

public void afterTest() {

    System.out.println("After Test");

}
```

6. @BeforeClass

Runs before the first test method in the current class is executed. Often used to initialize WebDriver.

Example:

java

Copy code

```
@BeforeClass

public void setUp() {

    System.out.println("Setting up before class");

}
```

7. @AfterClass

Executed after all test methods in the current class have been run. Useful for cleanup after a class.

Example:

java

Copy code

```
@AfterClass

public void tearDown() {

    System.out.println("Cleaning up after class");

}
```

8. @BeforeMethod

Runs before each test method. Commonly used to initialize browser or web page state.

Example:

java

Copy code

```
@BeforeMethod

public void setUp() {

    WebDriver driver = new ChromeDriver();

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

}
```

9. @AfterMethod

Executed after each test method. Typically used to close the browser.

Example:

java

Copy code

```
@AfterMethod

public void tearDown() {

    driver.quit();

}
```

10. @BeforeGroups

Runs before any test method that belongs to the specified groups.

Example:

java

Copy code

```
@BeforeGroups("regression")

public void setUpGroup() {
```

```
    System.out.println("Setting up for regression group");
}
```

11. @AfterGroups

Runs after all test methods that belong to the specified groups.

Example:
java
Copy code

```
@AfterGroups("regression")
public void tearDownGroup() {
    System.out.println("Tearing down after regression group");
}
```

12. @DataProvider

Used to provide data for test methods. It allows the same test to run multiple times with different data sets.

Example:
java
Copy code

```
@DataProvider(name = "loginData")
public Object[][] getData() {
    return new Object[][] { {"user1", "pass1"}, {"user2", "pass2"} };
}
@Test(dataProvider = "loginData")
public void loginTest(String username, String password) {
```

```
    WebDriver driver = new ChromeDriver();

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

    driver.findElement(By.id("username")).sendKeys(username);

    driver.findElement(By.id("password")).sendKeys(password);

    driver.quit();

}
```

13. @Factory

Used to execute a test class with multiple instances. It is used to run the same test with different objects.

Example:

java

Copy code

```
@Factory

public Object[] factoryMethod() {

    return new Object[] {new TestClass1(), new TestClass2()};

}
```

14. @Parameters

Used to pass parameters from the testng.xml file to the test method.

Example:

java

Copy code

```
@Parameters({"url", "username"})

@Test
```

```
public void loginTest(String url, String username) {
    WebDriver driver = new ChromeDriver();
    driver.get(url);
    driver.findElement(By.id("username")).sendKeys(username);
    driver.quit();
}
```

External Annotations for TestNG with Selenium

1. @Listeners (External via implementing Listeners Interface)

Used to track the status of test execution such as test start, success, failure, or skip. You must implement ITestListener to track test events.

Example:
java
Copy code

```
@Listeners(TestListener.class)
public class SampleTest {
    @Test
    public void sampleTest() {
        Assert.assertTrue(true);
    }
}
public class TestListener implements ITestListener {
```

```
    @Override
    public void onTestSuccess(ITestResult result) {
        System.out.println("Test Passed: " + result.getName());
    }
    @Override
    public void onTestFailure(ITestResult result) {
        System.out.println("Test Failed: " + result.getName());
    }
}
```

2. @RetryAnalyzer (External via Retry Analyzer Interface)

Used to retry failed tests automatically. You need to implement IRetryAnalyzer to specify retry logic.

Example:

java

Copy code

```
@Test(retryAnalyzer = RetryAnalyzer.class)
public void testWithRetry() {
    Assert.assertTrue(false);  // Test will retry if it fails
}


public class RetryAnalyzer implements IRetryAnalyzer {
    private int count = 0;
```

```
    private static final int maxTry = 3;

    @Override

    public boolean retry(ITestResult result) {

        if (count < maxTry) {

            count++;

            return true;

        }

        return false;

    }

}
```

3. @Listeners (External - ExtentReports)

Used to generate external reports for test execution. You need to configure ExtentReports and use listeners for logging.

Example:
java
Copy code

```
@Listeners(ExtentReportListener.class)

public class ExtentReportTest {

    @Test

    public void testExtentReport() {

        Assert.assertTrue(true);

    }
```

```
}
```

4. @Ignore (External - Used for disabling tests with JUnit integration)

Can be used when integrating JUnit with TestNG to skip specific tests.

Example:
java
Copy code

```
@Ignore

public void testToBeIgnored() {

   System.out.println("This test will be ignored.");

}
```

These annotations are essential when working with **TestNG** and **Selenium**, offering flexibility in test setup, execution, and reporting.

Utilizing Headless Browsers in Selenium Automation Testing

Headless browsers enable Selenium to execute tests without a graphical user interface, making them ideal for CI/CD pipelines and server environments.

Libraries like HtmlUnitDriver, Headless Chrome, and Headless Firefox allow for faster test execution while still supporting full browser capabilities. By leveraging these headless options, testers can run automated scripts efficiently, ensuring quick feedback and resource optimization during testing processes.

Here’s a list of **headless browser libraries** commonly used in Selenium automation testing:

1. HtmlUnitDriver

- A pure Java headless browser that simulates a browser environment without a graphical user interface.

Usage:
java
Copy code
```
WebDriver driver = new HtmlUnitDriver();

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

2. PhantomJS (Deprecated)

- A headless WebKit scriptable with a JavaScript API. Previously popular for headless testing.

Usage:
java
Copy code
```
WebDriver driver = new PhantomJSDriver();

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

3. Headless Chrome

- Chrome can run in headless mode, providing full browser capabilities without a GUI.

Usage:
java
Copy code
```
ChromeOptions options = new ChromeOptions();

options.addArguments("--headless");
```

```
WebDriver driver = new ChromeDriver(options);

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

4. Headless Firefox

- Firefox can also be run in headless mode, similar to Chrome.

Usage:
java
Copy code
```
FirefoxOptions options = new FirefoxOptions();

options.setHeadless(true);

WebDriver driver = new FirefoxDriver(options);

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

5. Cypress

- Although not a traditional Selenium driver, Cypress is a modern testing framework that runs headlessly in the background and can be used for end-to-end testing.

Usage:
bash
Copy code
```
npx cypress run --headless
```

6. Selenium Wire

- An extended version of Selenium that can work with headless browsers and capture requests/responses.

Usage:
python

Copy code

```
from seleniumwire import webdriver

driver = webdriver.Firefox(options=options)

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

7. Puppeteer

- While not directly a Selenium library, Puppeteer is a Node.js library that provides a high-level API over the Chrome DevTools Protocol, allowing headless Chrome testing.

Usage:

javascript

Copy code

```
const puppeteer = require('puppeteer');

(async () => {

  const browser = await puppeteer.launch({ headless: true });

  const page = await browser.newPage();

  await page.goto('http://example.com');

  await browser.close();

})();
```

8. Playwright

- Similar to Puppeteer, Playwright is a newer framework that supports multiple browsers (Chromium, Firefox, WebKit) in headless mode.

Usage:

javascript

Copy code

```
const { chromium } = require('playwright');

(async () => {

  const browser = await chromium.launch({ headless: true });

  const page = await browser.newPage();

  await page.goto('http://example.com');

  await browser.close();

})();
```

These libraries enable efficient automated testing in headless environments, making them suitable for continuous integration and deployment scenarios.
