SOFTWARE TESTING – UNIT-3

What Is Selenium

Evaluate the Key components of Selenium with advantages of using Selenium

Selenium is a versatile open-source framework primarily used for automating web browsers. It allows developers and testers to write scripts that simulate user interactions, such as clicking buttons, entering text, and navigating web pages, to ensure web applications function correctly across different browsers and platforms.

Key Components of Selenium:

1. **Selenium WebDriver**: The core component that provides APIs for controlling web browsers programmatically. It supports multiple programming languages, including Java, Python, C#, and JavaScript.
2. **Selenium IDE** (Integrated Development Environment): A browser extension that allows users to record and playback user interactions with web applications, facilitating the creation of test scripts without extensive coding knowledge.
3. **Selenium Grid**: A tool that enables the execution of Selenium tests on multiple machines and browsers simultaneously, supporting parallel test execution and cross-browser testing.

Advantages of Using Selenium:

1. **Cross-Browser Compatibility**: Selenium supports various browsers, including Chrome, Firefox, Safari, and Edge, allowing tests to be executed across different browser environments.
2. **Multi-Language Support**: Test scripts can be written in several programming languages, such as Java, Python, C#, and JavaScript, providing flexibility based on the team's expertise.
3. **Integration with Testing Frameworks**: Selenium can be integrated with tools like TestNG, JUnit, and PyTest for efficient test management and reporting.
4. **Open-Source and Community Support**: Being open-source, Selenium has a large community that contributes to its continuous improvement, offering a wealth of resources, tutorials, and plugins

What is Selenium IDE

Analyse the features and uses of Selenium IDE with examples

Selenium IDE (Integrated Development Environment) is a tool used for automated testing of web applications. It's the most beginner-friendly component of the Selenium Suite, which also includes Selenium WebDriver and Selenium Grid.

Selenium IDE is a browser extension (available for Chrome and Firefox) that allows testers to:

- Record their interactions with a web application (like clicking buttons, filling forms)
- Playback those actions as automated tests
- Edit the test scripts using a simple UI
- Export tests to code in languages like JavaScript

Key Features:

- Record & Playback: Great for creating quick tests without needing to write code.
- Command Editor: Modify or enhance recorded commands.
- Assertions: Check if specific conditions (like element visibility) are met.
- Control Flow: Support for loops and conditional logic.
- Plugins: Extend functionality (like exporting to Selenium WebDriver scripts).
- Cross-browser support (via Selenium Grid if needed).

Who Uses It?

- Testers or QA engineers who need fast feedback.
- Developers who want to quickly prototype tests.
- Beginners learning Selenium before diving into coding.

Components of Selenium IDE

Evaluate the major components Selenium IDE with example use cases

Selenium IDE (Integrated Development Environment) is a tool for automating web browsers. It is primarily used for recording and playing back user interactions with a web application for testing purposes. The components of Selenium IDE are designed to facilitate the creation, execution, and debugging of automated tests for web applications.

Here are the main components of **Selenium IDE**:

1. Record/Playback Functionality

- **Recording**: Selenium IDE can record user interactions with a web application, such as clicking buttons, typing in text fields, and navigating between pages. This allows users to create test scripts without writing any code.
- **Playback**: After recording, you can play back the interactions to ensure that the web application behaves as expected. This is useful for regression testing.

2. Test Case Panel

This is the main panel in Selenium IDE where all the recorded or written test commands are displayed in a tabular format. The test case panel is divided into three main columns:

- **Command**: The specific action or operation that Selenium will perform (e.g., click, type, assert).
- **Target**: The target element that the command operates on (e.g., an HTML element like a button, input field, etc.).
- **Value**: The data associated with the command (e.g., the text to type in an input field or the URL to visit).

3. Test Suite Panel

Selenium IDE allows you to organize multiple test cases into a test suite. A **Test Suite** is a collection of test cases that can be executed together. The Test Suite panel helps you manage and execute these test cases in an organized manner.

4. Command Toolbar

This toolbar contains essential buttons for interacting with Selenium IDE:

- **Play**: Starts the execution of the test case(s).
- **Pause**: Pauses the execution of the test case.
- **Step**: Executes the test case step by step.
- **Stop**: Stops the execution immediately.
- **Record**: Starts or stops recording user interactions.
- **Clear**: Clears the current test or test suite.

5. Log Panel

The log panel displays the output of the test execution, including any errors or successes. It helps with debugging and provides a detailed report of the actions performed during the test.

6. Breakpoints

Breakpoints are markers that can be set at a specific step in a test case. When the test execution reaches a breakpoint, it pauses, allowing the user to inspect the state of the application and test variables. This is useful for debugging.

7. Selenium IDE Commands

Selenium IDE comes with a set of built-in commands that can be used to perform actions on web elements, navigate through pages, and assert conditions (like verifying if an element exists or contains specific text). Some common commands include:

- **open**: Navigate to a specific URL.
- **click**: Simulate a click on an element.
- **type**: Type text into a form field.
- **assert**: Check if a certain condition is true.
- **waitForElementPresent**: Wait for an element to appear on the page.

8. Selenium IDE Extensions

Selenium IDE supports the use of extensions to add additional functionality, such as connecting with other tools, integrating with CI/CD pipelines, or adding custom commands.

9. Plugins and Custom Commands

Users can write custom commands or install plugins to extend the functionality of Selenium IDE. These plugins may allow you to interact with web applications in ways that the default set of commands does not support.

10. Selenium IDE Options

You can configure several settings within Selenium IDE, such as:

- **Playback speed**: Adjust how quickly the test runs.
- **Timeouts**: Define the time Selenium IDE waits for elements before throwing an error.
- **Base URL**: Set a base URL that all relative URLs are appended to.

11. Selenium IDE Export Feature

After recording or creating tests, you can export your test scripts to other programming languages (like Java, Python, JavaScript, etc.) to run with Selenium WebDriver. This makes Selenium IDE a good starting point for writing more complex automated tests.

12. Command Editor

The command editor lets you manually add, edit, or modify commands in the test cases. This is useful if you need to fine-tune a recorded script or add commands that weren't captured during recording.

Installing Selenium IDE (Browser Extension)

Selenium IDE is available as an extension for Chrome and Firefox.

Chrome:

- Go to the Chrome Web Store.
- Click Add to Chrome.
- Confirm by clicking Add Extension.

Firefox:

- Visit the Firefox Add-ons page for Selenium IDE.
- Click Add to Firefox.
- Confirm by clicking Add.

Getting Started

After installation, click the Selenium IDE icon in your browser toolbar.

- Choose Create a new project.
- Start recording your actions by navigating through the site.

- Stop recording and save the test case

General Rules for Automation

Analyse the application of General Rules for Automation with reference to testing web applications

Automate Repetitive Tasks-Only automate tasks that are repetitive and follow a predictable pattern. If it's a one-time thing or needs constant changes, manual may be better.

1. **Start Small-**Don't try to automate everything at once. Start with simple processes and expand from there.
2. **Know the Process Well First**-Automating a broken or inefficient process will only speed up the mess. Optimize it first, then automate.
3. **Define Clear Goals**-Know what you want to achieve with automation — saving time, reducing errors, increasing scalability, etc.
4. **Ensure Error Handling-**Automation should include steps for what happens when something fails — logs, alerts, retries, etc.
5. **Maintain Transparency**-Keep track of what's automated, how it works, and what it's affecting. Documentation and audit logs are key.
6. **Security First**-Automation often has elevated privileges or access — make sure it's secure and follows best practices (e.g., don't hard-code credentials).
7. **Measure and Optimize**-Use metrics to evaluate if automation is working as intended. Refine

Recording Your First Test With Selenium IDE

Develop a procedure for recording a test with Selenium IDE

Recording your first test with Selenium IDE is a great way to get started with automated browser testing. Selenium IDE is a browser extension for Chrome and Firefox that allows you to record, edit, and play back tests without needing to write code.Here's a step-by-step guide to recording your first test:

1. Install Selenium IDE

- Chrome: Selenium IDE for Chrome
- Firefox: Selenium IDE for Firefox

2. Start a New Project

- Open Selenium IDE from your browser extensions.
- Click "Create a new project".
- Give your project a name.

3. Record a Test

- After naming your project, click "Record a new test in a new project."
- Give your test a name.
- Enter the base URL of the site you want to test (e.g., https://example.com).
- The browser will open a new tab — this is where the recording begins.

4. Perform Your Actions

- Click around the site as a normal user would:
- Fill out a form
- Click buttons
- Navigate pages
- Selenium IDE will capture these steps automatically.

5. Stop Recording

- When you're done, go back to the Selenium IDE tab and click "Stop recording."
- You'll now see a list of all the recorded steps.

6. Play Back the Test

- Click the Play button (▶️) to run the test and watch it replicate your steps.
- You can also step through the test or debug it using the controls.

7. Save Your Project

Save the project to your local machine (.side file format) for future use

Updating A Test To Assert Items Are On The Page USING SELENEUM

Create a procedure for updating a test to assert items on the page

If you're using **Selenium** (in Python, I assume), and you want to assert that certain items are on the page, you can do it by locating elements and checking their presence or contents.

Here's a simple example to **assert that specific items are visible on the page**:

Example: Asserting Text Items Are Present (PYTHON CODE)

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.common.exceptions import NoSuchElementException
driver = webdriver.Chrome()
driver.get("https://example.com")
# Let's say you expect these items to be on the page
expected_items = ["Item One", "Item Two", "Item Three"]
for item in expected_items:
  try:
    # Find element containing the text
    element = driver.find_element(By.XPATH, f"//*[contains(text(), '{item}')]")
    assert element.is_displayed(), f"'{item}' is not visible on the page"
  except NoSuchElementException:
    assert False, f"'{item}' not found on the page"
driver.quit()
```

Updating A Test To Verify Items On The Page Adding Selenium IDE Comments USING PYTHON CODE

Develop a procedure for updating a test to verify items such as logo, logi-button, sign-up link using Selenium IDE commands

You're using Python with Selenium WebDriver, but you want to add Selenium IDE-style comments (i.e., descriptive comments that explain each action, like you'd see in Selenium IDE scripts). Let's walk through an example.

Example Goal:Verify these items are present on the page:

- A logo
- A "Login" button
- A "Sign Up" link

Updated Python Selenium Test with Selenium IDE-style comments

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
# Set up the Chrome driver
options = Options()
```

```
options.add_argument("--start-maximized")
service = Service('/path/to/chromedriver')  # Update with your actual chromedriver path
driver = webdriver.Chrome(service=service, options=options)

try:
    # Open the target webpage
    driver.get("https://example.com")  # Replace with your actual URL
    # Verify the logo is displayed
    # Comment: assert that the logo element is present on the page
    logo = driver.find_element(By.ID, "site-logo")
    assert logo.is_displayed(), "Logo is not displayed"
    # Verify the Login button is visible
    # Comment: assert that the Login button is present and visible
    login_button = driver.find_element(By.ID, "login-button")
    assert login_button.is_displayed(), "Login button is not visible"
    # Verify the Sign Up link exists
    # Comment: assert that the Sign Up link is present on the page
    sign_up_link = driver.find_element(By.LINK_TEXT, "Sign Up")
    assert sign_up_link.is_displayed(), "Sign Up link is not displayed"
    print("All items verified successfully!")
finally:
    # Close the browser
    driver.quit()
```

How To Add Comments To Tests In Selenium

Design a code to illustrate how to add comments in Java languages

In Selenium, adding comments to your tests can be helpful for documentation, explaining test logic, or making it easier for other developers to understand your code. There are a few different ways to add comments depending on the programming language you're using.

Here's how to add comments in common languages used with Selenium:

1. Java:

In Java, you can use two types of comments:

- **Single-line comment**:
  ```
  // This is a single-line comment
  ```
- **Multi-line comment**:
  ```
  /* This is a multi-line comment
    that spans multiple lines
  */
  ```
- **Javadoc comment** (used for documenting classes, methods, etc.):
  ```
  /**
   * This is a Javadoc comment
   * used to explain the test case logic
   */
  ```

Example of a test with comments in Java:

```
public class LoginTest {
    public void testLoginFunctionality() {
        // Navigate to the login page
        driver.get("http://example.com/login");
        // Find the username and password fields and enter credentials
        driver.findElement(By.id("username")).sendKeys("testuser");  // Enter username
```

```
        driver.findElement(By.id("password")).sendKeys("password123");  // Enter password

        // Click the login button
        driver.findElement(By.id("loginButton")).click();  // Clicking login button

        // Assert successful login (e.g., checking if we are redirected)
        Assert.assertTrue(driver.getCurrentUrl().contains("dashboard"));
    } }}
```

Multiplying Windows USING SELENIUM

In Selenium, handling multiple windows or tabs is a common task when interacting with a web application that opens new windows (like popups, modals, etc.) or tabs. You can switch between them, perform actions, and close them accordingly. Below is a guide on how to handle multiple windows using Selenium WebDriver in Python.

Develop a procedure for handling multiple windows in Selenium with an example code segment

Steps For Handling Multiple Windows In Selenium:

Open a new window or tab (This could be triggered by clicking a link/button that opens a new window).
Get the window handles (Each window/tab has a unique handle that allows you to switch between them).
Switch between windows using the window handles.
Perform actions on the new window/tab (e.g., clicking, reading data).
Close the window and switch back to the original window.

Example Code:

```
from selenium import webdriver
import time
# Set up the webdriver (this example uses Chrome)
driver = webdriver.Chrome()
# Open the website
driver.get('https://www.example.com')
# Get the window handle for the current window
main_window_handle = driver.current_window_handle
# Assume a link or button opens a new window
driver.find_element_by_link_text('Click here to open a new window').click()
# Wait for the new window to open (you may need to adjust this wait depending on the page load time)
time.sleep(2)
# Get all window handles
window_handles = driver.window_handles
# Switch to the new window (not the main window)
for handle in window_handles:
    if handle != main_window_handle:
        driver.switch_to.window(handle)
        break

# Perform actions in the new window
# Example: Get the title of the new window
print(driver.title)
# Close the new window
driver.close()
# Switch back to the main window
driver.switch_to.window(main_window_handle)
```

```
# Perform actions on the main window (if necessary)
# Example: Get the title of the main window
print(driver.title)
# Close the browser
driver.quit()
```

Key points to remember:

window_handles: Returns a list of all open windows or tabs. The first one in the list is the main window, and others are additional windows or tabs.

switch_to.window(handle): Switches to the window or tab specified by the handle.

current_window_handle: Gets the handle of the currently focused window

Working With Multiple Windows. In Selenium

Working with multiple windows in Selenium is common when automating web applications that open new browser windows or tabs. To interact with multiple windows in Selenium, you need to handle the window handles and switch between them.

Create a a step-by-step guide procedure to manage multiple windows in Selenium WebDriver

Here's a step-by-step guide on how to manage multiple windows in Selenium WebDriver:

1. Get the Window Handles

Every browser window or tab has a unique identifier called a window handle. You can use driver.getWindowHandles() to get all window handles and then switch between them.

```
# Get all window handles
window_handles = driver.window_handles
```

2. Switch Between Windows

You can switch to a particular window using its window handle. For example:

```
# Switch to a specific window
driver.switch_to.window(window_handles[1])  # switch to second window (index 1)
```

If you have a specific window that you want to switch to, iterate over the window handles.

```
for handle in window_handles:
   driver.switch_to.window(handle)
   # Perform actions like clicking, verifying, etc.
```

3. Open a New Window or Tab

If you want to open a new tab or window (for example, by clicking on a link), you can simulate this action by using the Keys.CONTROL + "t" (for new tab) or Keys.CONTROL + "n" (for new window) depending on the browser.

```
from selenium.webdriver.common.keys import Keys
# Simulate CTRL + T for a new tab
driver.find_element_by_tag_name("body").send_keys(Keys.CONTROL + 't')
```

4. Close a Window or Tab

After you are done with a window or tab, you can close it using driver.close() and then switch back to the original window.

```
# Close the current window
driver.close()
# Switch back to the main window
driver.switch_to.window(window_handles[0])  # switch to the first window
```

5. Example: Switch Between Multiple Windows

Design a Selenium code to open a new window, switch between them, and perform actions

Let's see an example where we open a new window, switch between them, and perform actions:

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

```
# Initialize the driver
driver = webdriver.Chrome()
# Open a website
driver.get('https://www.google.com')
# Open a new window/tab (CTRL + T in this case)
driver.find_element_by_tag_name('body').send_keys(Keys.CONTROL + 't')
# Switch to the new window
driver.switch_to.window(driver.window_handles[1])
# Go to a different URL in the new window
driver.get('https://www.bing.com')
# Perform some actions on the new window (optional)
time.sleep(2)
# Switch back to the original window
driver.switch_to.window(driver.window_handles[0])
# Perform actions on the original window (optional)
time.sleep(2)
# Close the current window
driver.close()
# Close all windows and quit the driver
driver.quit()
```

Key Points:

driver.window_handles gives a list of window handles.
driver.switch_to.window(window_handle) is used to switch to a specific window.
driver.close() closes the current window.
driver.quit() will close all open windows and terminate the WebDriver session

SOFTWARE TESTING – UNIT-4

Locating Elements By ID Using Selenium

In Selenium, locating elements by ID is one of the most common and efficient methods. You can use the find_element_by_id method to locate an element using its ID attribute.

Develop a procedure for locating elements by ID in Selenium with an example code

Example in Python using Selenium:

```
from selenium import webdriver
# Initialize the webdriver (Chrome in this case)
driver = webdriver.Chrome()
# Navigate to a URL
driver.get("https://example.com")
# Locate an element by its ID
element = driver.find_element_by_id("elementID")
# Perform actions on the element (e.g., click)
element.click()
# Close the browser
driver.quit()
```

Steps:

1. Import the necessary modules: You'll need webdriver from selenium to interact with the browser.
2. Initialize WebDriver: In the example, Chrome is used, but you can replace webdriver.Chrome() with any browser you want to use (like webdriver.Firefox() or webdriver.Edge()).
3. Use find_element_by_id: The method find_element_by_id("elementID") finds the element on the page with the specified ID.
4. Perform actions on the element: After locating the element, you can perform actions such as clicking, sending keys (typing), or extracting text.
5. Close the browser: After your interaction, driver.quit() will close the browser window.
6. Important Notes: Unique ID: The ID should be unique for each element on the page. If the ID is not unique, the find_element_by_id method will return the first element that matches the ID.
7. Selenium 4 Update: In Selenium 4, the method to find elements has changed. Instead of using find_element_by_id(), you now use find_element() with a By selector:

Finding IDs of elements on the page with Fire bug using Selenium

Create a procedure to find the IDs of elements on a page using Firebug and Selenium

To find the IDs of elements on a page using Firebug and Selenium, you can follow these steps:

Step 1: Inspect Element using Firebug

- Open Firebug in your browser (typically in Firefox).
- You can press F12 to open the Developer Tools in Firefox.
- In older versions of Firefox, Firebug can be installed as an add-on, but the Developer Tools are now built into modern versions of Firefox.
- Navigate to the Elements Tab: Once Firebug is open, go to the "Inspector" or "Elements" tab. This will allow you to inspect the HTML structure of the webpage.
- Find the Element: Hover over the elements in the page or search for the element in the HTML structure to see its ID. Right-click the element in the browser window and click "Inspect Element" to highlight it in the Firebug/Developer Tools panel.

- Check the ID: Once the element is highlighted in the Inspector tab, look for the id attribute in the HTML of the selected element.

Step 2: Use Selenium to Interact with the Element

Now that you know the ID of the element, you can use Selenium to interact with it. Here's a sample Python code using Selenium WebDriver to find elements by their ID:

```
from selenium import webdriver
# Set up the WebDriver (make sure to replace with the path to your browser driver)
driver = webdriver.Chrome(executable_path="path_to_your_chromedriver")
# Open the website
driver.get('http://yourwebsite.com')
# Find the element by ID
element = driver.find_element_by_id('element_id')
# Interact with the element (example: clicking)
element.click()
# Close the browser
driver.quit()
```

Step 3: Using XPath or CSS Selectors (If No ID)

If the element doesn't have an ID, you can use XPath or CSS selectors to find it.

For example:

```
# Find an element using XPath
element = driver.find_element_by_xpath("//button[text()='Submit']")
# Or using CSS selector
element = driver.find_element_by_css_selector("button.submit-class")
```

Step 4: Automate Element Identification

If you need to identify and print the IDs of all elements on the page dynamically, you can use the following code:

```
from selenium import webdriver
# Set up the WebDriver
driver = webdriver.Chrome(executable_path="path_to_your_chromedriver")
# Open the page
driver.get('http://yourwebsite.com')
# Find all elements on the page
elements = driver.find_elements_by_xpath('//*')
# Loop through each element and print its ID (if it has one)
for element in elements:
    id_attribute = element.get_attribute('id')
    if id_attribute:
        print(f"Element ID: {id_attribute}")
# Close the browser
driver.quit()
```

Finding Elements By ID Using Selenium

In Selenium, you can find elements by their ID using the find_element_by_id() method. This is a straightforward way to locate elements on a webpage based on the id attribute.

Develop a python code to locate elements on a webpage based on the id attribute. Analyse the use of important commands used.

Python Example with Selenium:

First, ensure you have the Selenium package installed:

```
pip install selenium
```

Then, you can use this code to find an element by ID:

```
from selenium import webdriver
# Initialize the WebDriver (make sure you have a compatible driver for your browser)
driver = webdriver.Chrome()
# Open the desired webpage
driver.get('https://www.example.com')
# Find an element by its ID
element = driver.find_element_by_id('element_id')  # Replace 'element_id' with the actual ID

# Perform actions on the element (for example, printing its text)
print(element.text)
# Close the browser window
driver.quit()
```

Explanation:

- webdriver.Chrome() initializes a Chrome browser session (you can replace this with any browser driver you need, e.g., webdriver.Firefox() for Firefox).
- driver.get('https://www.example.com') navigates to the URL you specify.
- driver.find_element_by_id('element_id') finds the element with the specified ID. Replace 'element_id' with the actual ID attribute of the element you want to find.

Finding an element on the page by their ClassName in Selenium

Develop a python code to locate elements on a webpage using its class name. Analyse the use of important commands used.

To find an element on a page using its class name in Selenium, you can use the find_element_by_class_name() method in Python. Here's a simple example:

Example in Python with Selenium:

```
from selenium import webdriver
# Initialize the WebDriver (assuming you have ChromeDriver installed)
driver = webdriver.Chrome()
# Open a website
driver.get("https://example.com")
# Find an element by its class name
element = driver.find_element_by_class_name("your-class-name")
# Perform actions on the element, for example, printing its text
print(element.text)
# Close the driver
driver.quit()
```

Important Notes:

1. **WebDriver Setup**: Make sure you have installed the appropriate WebDriver for the browser you're using. For example, chromedriver for Google Chrome, geckodriver for Firefox, etc.
2. **Handling Multiple Elements**: If there are multiple elements with the same class name, you can use find_elements_by_class_name() to get a list of matching elements.

```
elements = driver.find_elements_by_class_name("your-class-name")
for elem in elements:
   print(elem.text)
```

Moving Elements On The Page Using Selenium

To move elements on a page using Selenium, you can use the Actions class in Selenium. The Actions class provides various methods to perform actions such as click, drag and drop, hover, and move by offset.

Steps to move elements:

- Import required modules: You need to import the necessary libraries to interact with the web elements.

- Initialize WebDriver: Start the WebDriver for the browser of your choice (e.g., Chrome, Firefox).
- Locate the elements: Use the appropriate locator to find the elements you want to interact with.
- Use ActionChains for movement: The ActionChains class is used to chain multiple actions, including moving elements by offset or performing drag-and-drop operations.

Example Code: Moving an element by an offset:

Design a python code to move an element by a specific offset on a webpage. Analyse the use of important commands used

In this example, we will move an element by a specific offset (e.g., move it 100px to the right and 50px down).

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.by import By
import time
# Initialize the WebDriver (using Chrome in this case)
driver = webdriver.Chrome(executable_path='path/to/chromedriver')  # Update with your path
driver.get('https://your-website.com')  # Update with your URL
# Find the element you want to move
element = driver.find_element(By.ID, 'element_id')  # Update with your element's locator
# Create an ActionChains object
actions = ActionChains(driver)
# Move the element by offset (100px right and 50px down)
actions.drag_and_drop_by_offset(element, 100, 50).perform()
# Pause to see the result (optional)
time.sleep(2)
# Close the browser
driver.quit()
```

Performing a drag-and-drop operation:

Design a python code to a drag-and-drop operation from one element to another on a webpage. Analyse the use of important commands used

In this example, we will perform a drag-and-drop operation from one element to another.

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.by import By
import time
# Initialize the WebDriver (using Chrome in this case)
driver = webdriver.Chrome(executable_path='path/to/chromedriver')  # Update with your path
driver.get('https://your-website.com')  # Update with your URL
# Find the source and target elements
source_element = driver.find_element(By.ID, 'source_id')  # Update with the source element
target_element = driver.find_element(By.ID, 'target_id')  # Update with the target element
# Create an ActionChains object
actions = ActionChains(driver)
# Perform the drag and drop operation
actions.drag_and_drop(source_element, target_element).perform()
# Pause to see the result (optional)
time.sleep(2)
# Close the browser
driver.quit()
```

Notes:

drag_and_drop_by_offset: This method moves the element by a specific pixel offset, defined by x and y.
drag_and_drop: This method moves the element from a source element to a target element.
You can adjust the time.sleep() value as needed or use explicit waits to handle timing better.

Finding Elements By Name Using Selenium

Design a python code to find elements by name on a webpage. Analyse the use of important commands used

To find elements by name using Selenium, you can use the find_element_by_name or find_elements_by_name methods. These methods allow you to locate an element or a list of elements based on their name attribute.

```
from selenium import webdriver
# Start the WebDriver (make sure you have the appropriate driver for your browser)
driver = webdriver.Chrome()
# Navigate to the desired webpage
driver.get('http://example.com')
# Find the element by its 'name' attribute
element = driver.find_element_by_name('element_name')
# Do something with the element, for example, print its text
print(element.text)
# Close the browser
driver.quit()
```

Finding Multiple Elements by Name

Design a python code to find multiple elements by name on a webpage. Analyse the use of important commands used

If there are multiple elements with the same name attribute, you can use find_elements_by_name, which will return a list of matching elements.

```
from selenium import webdriver
# Start the WebDriver
driver = webdriver.Chrome()
# Navigate to the webpage
driver.get('http://example.com')
# Find all elements by the 'name' attribute
elements = driver.find_elements_by_name('element_name')
# Iterate through the elements and perform actions
for element in elements:
   print(element.text)
# Close the browser
driver.quit()
```

Finding Elements By Link Text Using Selenium

In Selenium, you can find elements by their link text using the find_element_by_link_text() method. This method is used to locate anchor (<a>) tags that contain the specified link text.

Design a python code element by link text on a webpage. Analyse the use of important commands used

Example: Finding an element by link text

```
from selenium import webdriver
# Set up the WebDriver (for example, using Chrome)
driver = webdriver.Chrome()
# Open a website
driver.get("https://example.com")
```

```
# Find the link by its visible text
link = driver.find_element_by_link_text("Click here")  # Replace "Click here" with the link text you're searching for
# Click the link
link.click()
# Close the browser
driver.quit()
```

Explanation:

webdriver.Chrome(): Initializes the Chrome WebDriver. You can replace it with any other browser, like webdriver.Firefox(), depending on what browser you're using.

driver.get(): Navigates to the webpage you want to interact with.

find_element_by_link_text(): Finds the anchor element (<a>) that has the specified visible link text.

click(): Clicks on the link once it's located.

Finding Elements By Xpath In Selenium

Design a python code to find element by Xpath on a webpage. Analyse the use of important commands used

In Selenium, XPath (XML Path Language) is a powerful tool for navigating and selecting elements within an HTML document. It's particularly useful when other locators like ID or class attributes are not available or reliable. XPath allows you to locate elements based on various attributes, text content, and their hierarchical relationships within the DOM.

Finding a Single Element by XPath

To find a single element using XPath in Selenium, you can use the find_element method along with By.XPATH. Here's an example in Python:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
# Initialize the WebDriver (e.g., ChromeDriver)
driver = webdriver.Chrome()
# Open the desired webpage
driver.get("https://www.example.com")
# Find an element by XPath and perform an action
element = driver.find_element(By.XPATH, "//tagname[@attribute='value']")
element.click()  # Example action: clicking the element
# Close the browser
driver.quit()
```

In this example, find_element(By.XPATH, "//tagname[@attribute='value']") locates an element with the specified tag name and attribute value. Replace "//tagname[@attribute='value']" with the appropriate XPath expression for your target element.

Finding Multiple Elements by XPath

Design a python code to find multiple elements using on a webpage Xpath. Analyse the use of important commands used

To find multiple elements that match a given XPath expression, use the find_elements method:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
# Initialize the WebDriver
driver = webdriver.Chrome()
# Open the webpage
driver.get("https://www.example.com")
```

```
# Find all elements matching the XPath
elements = driver.find_elements(By.XPATH, "//tagname[@attribute='value']")
# Iterate through the elements and perform actions
for element in elements:
   print(element.text)  # Example action: printing the text content
# Close the browser
driver.quit()
```

This script retrieves all elements that match the specified XPath and iterates through them to perform desired actions.

Types of XPath Expressions

- **Absolute XPath**: Specifies the complete path from the root element to the target element. It's generally not recommended due to its fragility; any change in the page structure can break the XPath.

```
element = driver.find_element(By.XPATH, "/html/body/div[1]/div[2]/div/h1")
```

- **Relative XPath**: Starts the path from a specific element, making it more flexible and less prone to breaking with minor page structure changes.

```
element = driver.find_element(By.XPATH, "//div[@class='example-class']/h1")
```

Using XPath Functions

XPath provides functions like contains(), starts-with(), and text() to create more robust and flexible expressions:

- **contains()**: Matches elements whose attribute value contains the specified substring.
  `element = driver.find_element(By.XPATH, "//a[contains(@class, 'btn')]")`
- **starts-with()**: Matches elements whose attribute value starts with the specified substring.
  `element = driver.find_element(By.XPATH, "//input[starts-with(@id, 'user')]")`
- **text()**: Matches elements based on their text content.
  `element = driver.find_element(By.XPATH, "//button[text()='Submit']")`

Selenium WebDriver

Selenium WebDriver is a powerful tool for automating web application testing. It allows you to simulate user interactions with a web browser, making it possible to perform automated tests on web applications. Selenium WebDriver supports multiple programming languages such as Java, Python, C#, JavaScript, and Ruby, and it is compatible with a wide range of browsers (Chrome, Firefox, Safari, Internet Explorer, etc.).

Key Features of Selenium WebDriver:

Evaluate the Key Features of Selenium WebDriver with its major features and advantages

Cross-Browser Compatibility: Selenium WebDriver supports major browsers like Chrome, Firefox, Safari, and Internet Explorer. Tests can be run across different browsers to ensure compatibility and consistent behaviour of web applications.

Language Support: Selenium WebDriver provides bindings for several programming languages, including:

- Java
- Python
- C#
- JavaScript (Node.js)
- Ruby

This allows developers to write test scripts in the programming language they are most comfortable with.

1. Direct Interaction with the Browser: Unlike Selenium's older version, Selenium RC (Remote Control), WebDriver communicates directly with the browser, making it faster and more reliable. It does not require a server to execute commands, which reduces complexity and increases performance.

2. Support for Multiple Platforms: Selenium WebDriver can be used to automate tests on different operating systems, including Windows, macOS, and Linux.
3. Dynamic Content Handling: WebDriver can interact with elements on the page that may change dynamically (AJAX-based applications, JavaScript-heavy pages, etc.).

It can wait for elements to appear or for certain conditions to be met (e.g., element visibility or page load), making it more suited for modern web applications.
Element Locators: WebDriver allows elements to be identified using different types of locators:

- ID
- Name
- Class Name
- Tag Name
- CSS Selectors
- XPath

This flexibility makes finding and interacting with elements easier.
Support for Parallel Test Execution: WebDriver can be used with tools like TestNG, JUnit, and Cucumber to run tests in parallel, improving test efficiency and reducing execution time.
WebDriver API: WebDriver provides an easy-to-use API for interacting with browser elements. Some common actions include:

- Clicking buttons
- Filling out forms
- Navigating between pages
- Taking screenshots

Retrieving page titles, URLs, and other information

1. Headless Testing: WebDriver supports headless browser testing (browsers that do not have a graphical interface). This is useful for running tests on a server or in continuous integration (CI) environments.
2. Integration with Testing Frameworks: Selenium WebDriver can be integrated with testing frameworks like JUnit, TestNG, and Cucumber for structuring tests, generating reports, and organizing test cases.
3. Common Use Cases: Automated Functional Testing: Ensure the core functionality of the web application works as expected.
4. Regression Testing: Test the application after changes are made to check for unintended side effects.
5. Performance Testing: Evaluate the performance of web applications under heavy user load.
6. UI Testing: Verify the user interface elements for proper behavior and appearance.

Advantages of Selenium WebDriver:

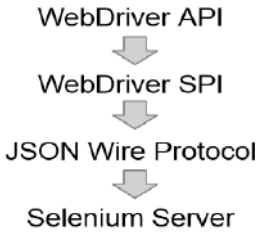
1. Faster than Selenium RC: Since WebDriver interacts directly with the browser, it is faster and more efficient.
2. Supports Multiple Browsers: It supports many popular browsers without requiring additional configurations.
3. Flexibility: Works well with different programming languages and testing frameworks.
4. Open-Source: Selenium WebDriver is free to use, and the community is large and active, offering support and plugins.
5. Disadvantages of Selenium WebDriver: Not Suitable for Non-Web Applications: Selenium is specifically designed for web applications, so it can't be used for automating desktop or mobile applications (unless combined with other tools like Appium for mobile).
6. Limited Reporting Capabilities: Selenium itself doesn't provide built-in reporting, so you may need to use third-party tools or integrate it with other frameworks like TestNG or JUnit to generate reports.

7. Requires Knowledge of Programming: Unlike record-and-playback tools, WebDriver requires some programming knowledge, making it less accessible to non-developers.

Architecture of Selenium Webdriver

Analyse the architecture of Webdriver with interactions between major components

Selenium WebDriver is composed of several key components that work together to perform automated web testing. Here's an overview of the architecture



The system is made up of four different sections.

1. WebDriver API

The WebDriver API is the part of the system that you interact with all the time. Things have changed from the 140 line long API that the Selenium RC API had. This is now more manageable and can actually fit on a normal screen. You will see this when you start using WebDriver in the next chapter. This is made up of the WebDriver and the WebElement objects.

```
driver.findElement(By.name("q"))
and
element.sendKeys("I love cheese");
```

These commands are then translated to the SPI, which is stateless. This can be seen in the next section.

2. WebDriver SPI

When code enters the Stateless Programming Interface or SPI, it is then called to a mechanism that breaks down what the element is, by using a unique ID, and then calling a command that is relevant. All of the API calls above then call down.

Using the example in the previous section would be like the following code, once it was in the SPI:

```
findElement(using="name", value="q")
sendKeys(element="webdriverID", value="I love cheese")
```

From there we call the JSON Wire protocol. We still use HTTP as the main transport mechanism. We communicate to the browsers and have a simple client server transport architecture the WebDriver developers created the JSON Wire Protocol.

3. JSON Wire protocol

The WebDriver developers created a transport mechanism called the JSON Wire Protocol. This protocol is able to transport all the necessary elements to the code that controls it. It uses a REST like API as the way to communicate.

4. Selenium server

The Selenium server, or browser, depending on what is processing, uses the JSON Wire commands to break down the JSON object and then does what it needs to. This part of the code is dependent on which browser it is running on.

Web Driver API using Selenium

Analyse the major functionalities of Webdriver with example code

Selenium is a powerful tool for controlling a web browser through the WebDriver API. It allows you to automate browser actions, interact with elements, and scrape data. The WebDriver API is part of Selenium's core functionality and provides the interface for controlling browsers like Chrome, Firefox, Safari, and Edge. Here's a basic overview of how you can use Selenium's WebDriver API in Python to automate browser actions:

Prerequisites:

Install the selenium package:

```
pip install selenium
```

Download the appropriate WebDriver for your browser:

- ChromeDriver
- GeckoDriver (Firefox)
- EdgeDriver
- SafariDriver

Make sure to place the driver in your system's PATH or specify its location in the script.

Example Code:

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

# Set up the WebDriver for Chrome
driver = webdriver.Chrome(executable_path='/path/to/chromedriver')  # Specify the correct path
# Open a webpage
driver.get('https://www.example.com')
# Wait for the page to load
time.sleep(2)  # Use WebDriverWait for more advanced waiting strategies
# Find an element on the page (e.g., a search input box)
search_box = driver.find_element(By.NAME, 'q')  # Find element by its name attribute
# Perform actions on the element (e.g., typing into the search box)
search_box.send_keys('Selenium WebDriver')
# Submit the form (if applicable)
search_box.send_keys(Keys.RETURN)
# Wait for some time to observe the results
time.sleep(3)
# Close the browser
driver.quit()
```

Key Components:

WebDriver: The main interface for controlling the browser. In the example above, webdriver.Chrome() starts the Chrome browser.

By Locators: Selenium uses various methods to locate elements on the page. For example:

- By.NAME
- By.ID
- By.CLASS_NAME
- By.XPATH
- By.CSS_SELECTOR

Actions: You can interact with elements such as clicking buttons, sending keystrokes, or scrolling.

Waiting: Use time.sleep() or better yet, WebDriverWait to wait for elements to be ready before interacting with them.

Advanced Usage:

Explicit Waits:

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

# Wait until the element is visible
element = WebDriverWait(driver, 10).until(
   EC.visibility_of_element_located((By.ID, "some_element_id"))
```

```
)
```

Handling Alerts:

```
alert = driver.switch_to.alert
alert.accept()  # Click the "OK" button on the alert
```

Handling Multiple Windows:

```
# Get current window handle
main_window = driver.current_window_handle
# Click a link that opens a new window
driver.find_element(By.LINK_TEXT, 'Open New Window').click()
# Switch to the new window
WebDriverWait(driver, 10).until(EC.new_window_is_opened(driver.window_handles))
new_window = driver.window_handles[-1]
driver.switch_to.window(new_window)
# Do something in the new window, then close it
driver.close()
# Switch back to the main window
driver.switch_to.window(main_window)
```

Taking Screenshots:

```
driver.save_screenshot('screenshot.png')
```

What is WebDriver SPI?

SPI stands for Service Provider Interface. In Selenium, WebDriver SPI is a way to allow different browser vendors to plug their own implementations of WebDriver into Selenium in a standardized way.
In short:

- Selenium provides an interface, and
- Browser vendors (like Chrome, Firefox, etc.) provide the implementation.

Why is WebDriver SPI useful?

With SPI, Selenium can dynamically load the appropriate driver for a browser without hardcoding it, improving:

- Modularity
- Pluggability
- Cross-browser support

Using Selenium with WebDriver (Typical Example)

If you're just trying to use Selenium WebDriver (not develop SPI-level integrations), here's a simple example in Python using Chrome:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
from webdriver_manager.chrome import ChromeDriverManager
# Set up the Chrome WebDriver
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()))
# Open a website
driver.get("https://example.com")
# Interact with the page
print(driver.title)
# Close the browser
driver.quit()
```

Finding Elements Using Selenium

Evaluate the various methods and strategies provided by Selenium to locate web elements

In Selenium, locating web elements is fundamental to automating browser interactions. Selenium provides two primary methods for this purpose

- **find_element**: Locates the first matching element on the page. If no element is found, it raises a NoSuchElementException. This method is suitable when you expect only one matching element.
- **find_elements**: Locates all matching elements on the page and returns them as a list. If no elements are found, it returns an empty list. This method is useful when multiple elements match your criteria.

To identify elements, Selenium offers various locator strategies:

1. **By ID**: Locates elements using the id attribute, which should be unique within a page.
   element = driver.find_element(By.ID, "elementId")
2. **By Name**: Locates elements using the name attribute.python
   element = driver.find_element(By.NAME, "elementName")
3. **By Class Name**: Locates elements using the class attribute.python
   element = driver.find_element(By.CLASS_NAME, "className")
4. **By Tag Name**: Locates elements using the HTML tag name.python
   element = driver.find_element(By.TAG_NAME, "tagName")
5. **By Link Text**: Locates hyperlink elements using the exact text within the link.
   element = driver.find_element(By.LINK_TEXT, "Link Text")
6. **By Partial Link Text**: Locates hyperlink elements using a partial match of the link text.
   element = driver.find_element(By.PARTIAL_LINK_TEXT, "Partial Link")
7. **By CSS Selector**: Locates elements using CSS selectors, allowing for complex queries.
   element = driver.find_element(By.CSS_SELECTOR, "div.classname > p")
8. **By XPath**: Locates elements using XPath expressions, providing a flexible way to navigate the DOM.
   element = driver.find_element(By.XPATH, "//div[@id='elementId']")