Lecture 32
Selenium

ECE 422: Reliable and Secure Systems Design

Instructor: An Ran Chen
Term: 2024 Winter

Schedule for today

- Introduction to Selenium
  - History of Selenium
  - Why is automated testing important?
  - How can Selenium be useful?
- Selenium basics
  - Locating elements
  - XPath syntax
- Demo: Cookie Clicker
  - Navigating pages
  - Waits for elements

What is Selenium?

Selenium is an open-source, automated testing tool used to test web applications across different browsers.

- Goal: Automate testing process in software development life cycle (SDLC)

Selenium test suite comprises of four tools:

- Selenium IDE: plugin for recording user interaction and playing back
- Selenium Remote Control (RC) (deprecated): server that allows users to write application tests
- Selenium WebDriver: remote control interface for writing application tests
- Selenium Grid (mostly for Continuous Integration): server for parallel execution of tests on different browsers and different operating systems

History of Selenium

- 2005: Selenium IDE
  - Chrome and Firefox plugin that records user interactions on the browser and plays them back as automated tests
- 2007: Selenium Remote Control (RC) (deprecated)
  - Selenium Server + Client libraries
  - Selenium Server: launches and kills browsers
  - Client libraries: manages user interactions with the browser
- 2008: Selenium WebDriver 2.0
- 2016: Selenium WebDriver 3
- 2021: Selenium WebDriver 4
  - Programming interface that instructs the behavior of web browsers

How can Selenium be useful?

- Essential tool for QA analyst
  - Regression Testing
  - Integration with Continuous Integration (CI) Pipelines
- Project proposal for web developer
  - Functional bugs in UI
  - Cross-Browser Testing
- Assistance for UI tester
  - Complex Use Case / Application Flows / Acceptance Tests
  - UI/UX Testing
- (Legal) Web scraping tool for freelancer
  - Data mining and extraction

Advantages of Selenium

Selenium offers a competitive edge over others tool:

- Open source: available on GitHub
  - Transparency + Flexibility + Security
- Multiple browsers, languages and platforms supports
  - Languages: Python, Java, C#, JavaScript, Ruby, Rust
  - Browsers: Chrome, Firefox, Safari, Internet Explorer, Microsoft Edge
  - Platforms: Windows, MacOS, Ubuntu
- Framework supports
  - TestNG and JUnit
  - Behat + Mink
- Parallel test execution
  - Optimize Continuous Integration and Delivery (DevOps)

Schedule for today

- Introduction to Selenium
  - History of Selenium
  - Why is automated testing important?
  - How can Selenium be useful?
- Selenium basics
  - Locating elements
  - XPath syntax
- Demo: Cookie Clicker
  - Navigating pages
  - Waits for elements

Why is automated testing important?

Before automated testing was introduced, manual testing was the norm. However, it had several drawbacks:

- Expensive: Require a full time Quality Assurance (QA) team
- Error-prone: Human-driven manual process for review and validation
- Slow: New features on-hold until the QA team finished testing
- Redundant and tedious: Manually execution of use cases every time a new update was pushed to production

Most modern Agile and DevOps development requires continuous testing.

Test automation is becoming an industrial relevant topic in software education community: Software Testing Education workshop (TestEd 2024 as part of ICST)

Schedule for today

- Introduction to Selenium
  - History of Selenium
  - Why is automated testing important?
  - How can Selenium be useful?
- Selenium basics
  - Locating elements
  - XPath syntax
- Demo: Cookie Clicker
  - Navigating pages
  - Waits for elements

Selenium basics

Writing an automated test in Selenium typically involves three steps:

- Locating elements
  - Obtaining element references to work with
  - Selenium uses locator strategies to uniquely identify each HTML element
- Navigating pages
  - Interacting with web elements
  - Only 5 basic commands on each element: click, send keys, clear, submit, select
- Waits for elements
  - Avoid race condition in browser (delay exists in loading elements)
  - Example: Clicking on a button before the button element is present in the DOM

Locating elements

Selenium provides built-in methods to locate web elements in a page:

- find_element() for locate a single element
- find_elements() for locating multiple elements

These methods takes two parameters as inputs → find_element(Locator, Value):

- Locator: identifies the locator used for finding the element
- Value: gives the value of the locator

Example: Find element with locator By.CLASS_NAME

```
<a aria-label="University of Alberta"
    class="navbar-brand en-logo"
    href="https://www.ualberta.ca/index.html">
</a>
```

```
find_element(By.CLASS_NAME, "en-logo")
```

or

```
find_element(By.CLASS_NAME, "navbar-brand")
```

Available attributes for Locator

Locator (used in find_element) has eight available attributes:

Locator attribute	Sample element	Example of code
By.ID	<h1 id="header">Header</h1>	find_element(By.ID, "header")
By.NAME	<h1 name="header">Header</h1>	find_element(By.NAME, "header")
By.CLASS_NAME	<h1 class="header">Header</h1>	find_element(By.CLASS, "header")
By.LINK_TEXT	Home	find_element(By.LINK_TEXT, "Home")
By.PARTIAL_LINK_TEXT	Home	find_element(By.LINK_TEXT, "ome")
By.TAG_NAME	Header	find_element(By.TAG_NAME, "h1")
By.XPATH	Header	find_element(By.XPATH, "//h1")
By.CSS_SELECTOR	<h1 class="header">Header</h1>	find_element(By.CSS_SELECTOR, "h1.header")

Locating by XPath

XPath (XML Path Language) is a path expression language designed to select nodes or node-sets in a XML (or HTML) document.

- Nodes are selected by following a path

One of the main reasons for using XPath is for its reliability:

- Id and name attributes may not exist on all elements
  - By.ID, By.NAME
- Link attributes are associated with links only
  - By.LINK_TEXT, By.PARTIAL_LINK_TEXT
- Tag and class attributes may not always be unique
  - By.TAG_NAME, By.CSS_SELECTOR, By.CLASS_NAME

XPath syntax

Overall idea: Tagname[some identifiers] + Path to the tag

XPath = // tagname [@ Attribute = 'Value']

XPath syntax:

- //: Select particular nodes in the HTML tree (Set search space)
  - Single forward slash (/) selects only the immediate child elements
  - Double forward slash (//) selects all descendants of the current node, regardless of their level
- Tagname: Set tagname of the particular node
- @: Select attribute
- Attribute: Set attribute name of the node
- Value: Set value of the attribute

Absolute and relative XPath

There are two types of XPath:

- Absolute: Find element through its absolute path, starting from the root element
  - By using the single forward slash (/)
- Relative: Find elements anywhere on the page
  - By using the double forward slash (//)

Example of XPath: Look for the UAlberta logo (Try it yourself!)

- Absolute: /html/body/header/div/div/div/div/a[@aria-label='University of Alberta']
  - Starting from the root element <html>, go through this path and find element
- Relative: //header//a[@aria-label='University of Alberta']
  - Anywhere on the page: Find the first <header>
  - Anywhere inside this header: Find the first <a> with aria-label attribute equals to University of Alberta

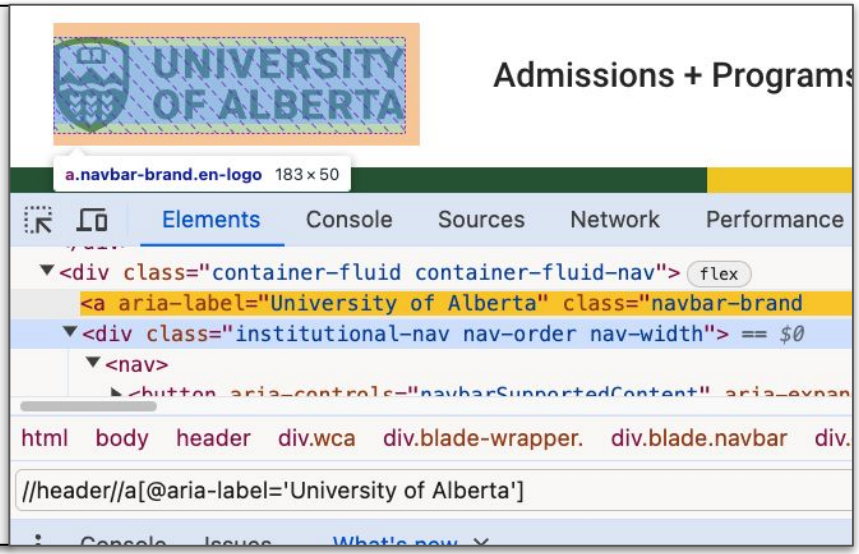
Try it yourself!

Use the following XPath to find the first UoA logo in the <header> of UAlberta.ca:

- /html/body/header/div/div/div/div/a[@aria-label='University of Alberta']
- //header//a[@aria-label='University of Alberta']

Instructions: On UAlberta website

- Inspect the main webpage
- In the Elements view, press Ctrl + F (or command + F)
- Search //a[@aria-label='University of Alberta'] in the source code
- All UoA should now be highlighted



Schedule for today

- Introduction to Selenium
  - History of Selenium
  - Why is automated testing important?
  - How can Selenium be useful?
- Selenium basics
  - Locating elements
  - XPath syntax
- Demo: Cookie Clicker
  - Navigating pages
  - Waits for elements

Demo: Cookie Clicker

Cookie Clicker is a 2013 web game designed to have users click on a big cookie on the screen.

- Steam version released in 2021, reaching top 15 of Steam games at the time

Build an automation script for clicking the cookie 100 times

- Input: Cookie Clicker URL
- Learning Objectives: Click + Wait + Scrap + Assert
- Output: Assert that the cookie has been clicked 100 times
- Source code available on GitHub: automation script

Demo guide (Python)

Step 1: Installation

- Python language bindings for selenium package

Step 2: Python script setup

- Getting started with WebDriver

Step 3: Navigate and click

- Interacting with the page
- Explicit Waits

Step 4: Assertion

- Test case with assertions

Step 1: Installation

Step 1: To run Selenium, we first need to install a WebDriver.

- WebDriver is responsible for managing the content interactions, without requiring browser-specific code

Every browser provides WebDriver supports:

- Chrome: ChromeDriver
- Firefox: GeckoDriver
- Edge: Microsoft Edge WebDriver
- Safari: SafariDriver

If you are using Chrome version 115 or newer, check the Chrome for Testing availability dashboard

Note that this demo uses: chromedriver for mac-x64 (click here to download)

Step 2: Python script setup

main.py available

Step 2: create a Python script (e.g., main.py) in the same folder as the WebDriver

- main.py is the automation script we use for clicking the cookie

Set up the script by importing the WebDriver, then try to launch the browser:

- Import the WebDriver and By class (for locating elements)

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

- Create an instance of Chrome WebDriver

```
driver = webdriver.Chrome()
```

- Use driver.get method to navigate to a given URL

```
driver.get("https://orteil.dashnet.org/cookieclicker/")
```

Step 2: Python script setup

- Launch browser in incognito mode (reproductivity, without cookies)

```
chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument("--incognito")
```

- Add a "pause" time to make the browser visible

```
import time
time.sleep(10)
```

- Close the browser once done

```
driver.quit()
```

Step 3: Navigate and click

Step 3: Select a language and click on the cookie

- Step 3.1: Select the English element from language popup
  - Appear on first visit (incognito for test case rerun)
  - Take time to load (wait for the element to appear)
- Step 3.2: Click on the cookie
  - Take time to load (wait for the element to appear)
  - Click 100 times (add loop)

Navigating pages

Navigating pages = interacting with HTML elements (e.g., links) within a page

- Locating + simulating a click

Example: Click on a web element

- Locating elements with XPath

```
logo_xpath = "//*[@aria-label='University of Alberta']"
logo_element = driver.find_element(By.XPATH, logo_xpath)
```

- Simulate a click on elements

```
logo_element.click()
```

Waits for elements

When a page is loaded by the web browser, the elements within that page are often loaded at different time intervals.

- This makes locating element difficult
- Elements that we are locating may not be present in the DOM yet

Solution: Assign explicit waits for elements

- Wait for a condition to occur before proceeding further in the code

Example: Wait until the element appears for 10 seconds

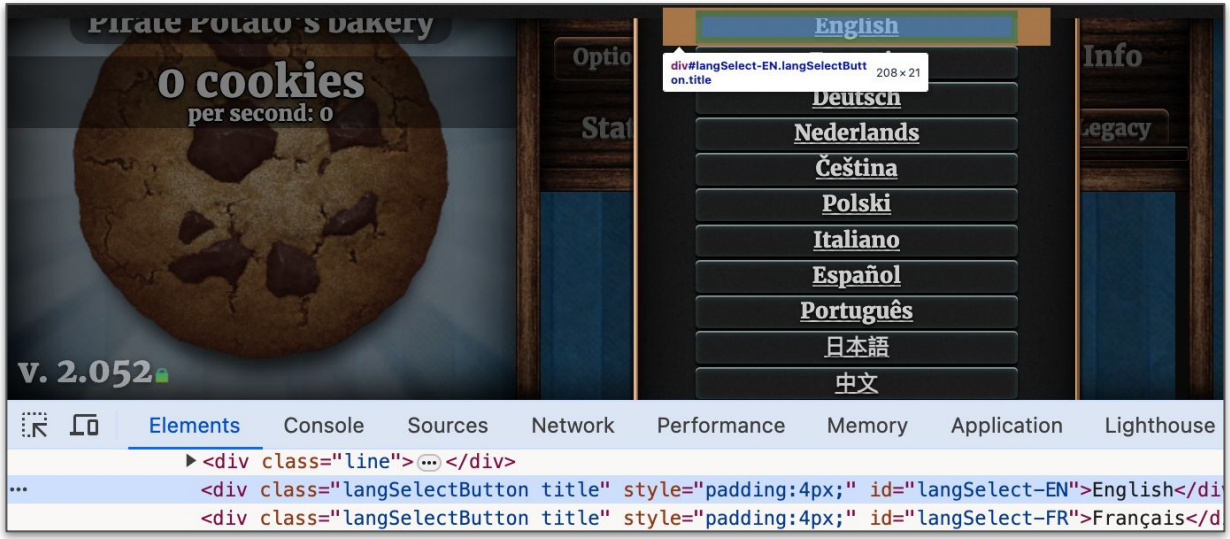
```
wait = WebDriverWait(driver, 10)
wait.until(EC.presence_of_element_located((By.XPATH, element_xpath)))
```

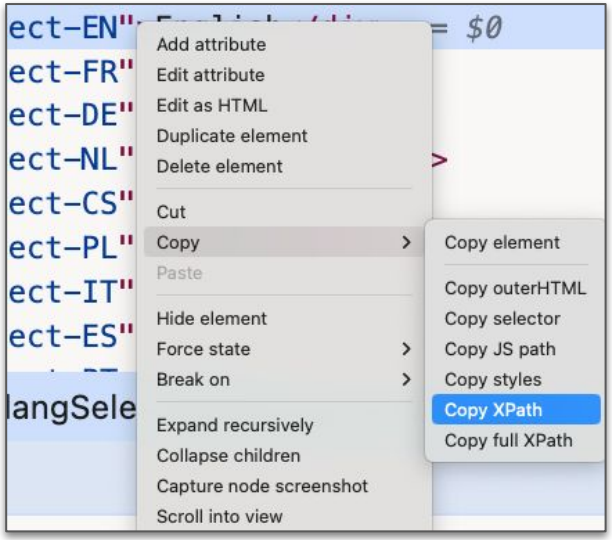
Step 3: Navigate and click

main.py available

Step 3.1: Select the English element from language popup

- Inspect the page and write (or copy) the XPath for "English" web element





Step 3: Navigate and click

Step 3.1: Select the English element from language popup

- Wait for the English element to appear

```
language_xpath = "//*[@id='langSelect-EN']"
wait = WebDriverWait(driver, 10)
language_element = wait.until(
    EC.presence_of_element_located((By.XPATH, language_xpath))
)
```

- Click on English

```
language_element.click()
```

Step 3: Navigate and click

main.py available

Step 3.2: Click on the cookie

- Wait for the cookie element to appear

```
cookie_xpath = "//*[@id='bigCookie']"
wait = WebDriverWait(driver, 10)
cookie_element = wait.until(
    EC.presence_of_element_located((By.XPATH, cookie_xpath))
)
```

- Click on cookie

```
cookie_element.click()
```

Step 3: Navigate and click

Step 3.2: Click on the cookie

- Find cookie counter element and get integer

```
count = 0
count_xpath = "//*[@id='cookies']"
count = driver.find_element(By.XPATH, count_xpath).text.split(' ')[0]
```

- Click 100 times (string to int)
    - Find count and cookie elements after each click (avoid stale elements)

```
count = 0
count_xpath = "//*[@id='cookies']"
while int(count) < 100:
    cookie_element = driver.find_element(By.XPATH, cookie_xpath)
    cookie_element.click()
    count = driver.find_element(By.XPATH, count_xpath).text.split(' ')[0]
```

Step 4: Assertion

main.py available

Step 4: Assert that the cookie is less than 100, after an upgrade

- Upgrade for cookie auto-generation

```
grandma_xpath = "//*[@id='product1']"
grandma_element = driver.find_element(By.XPATH, grandma_xpath)
grandma_element.click()
```

- Capture count element again and assert that it is less than 100
  - If the assertion fails, then the script throws an AssertionError

```
count = driver.find_element(By.XPATH, count_xpath).text.split(' ')[0]
assert int(count) < 100
```