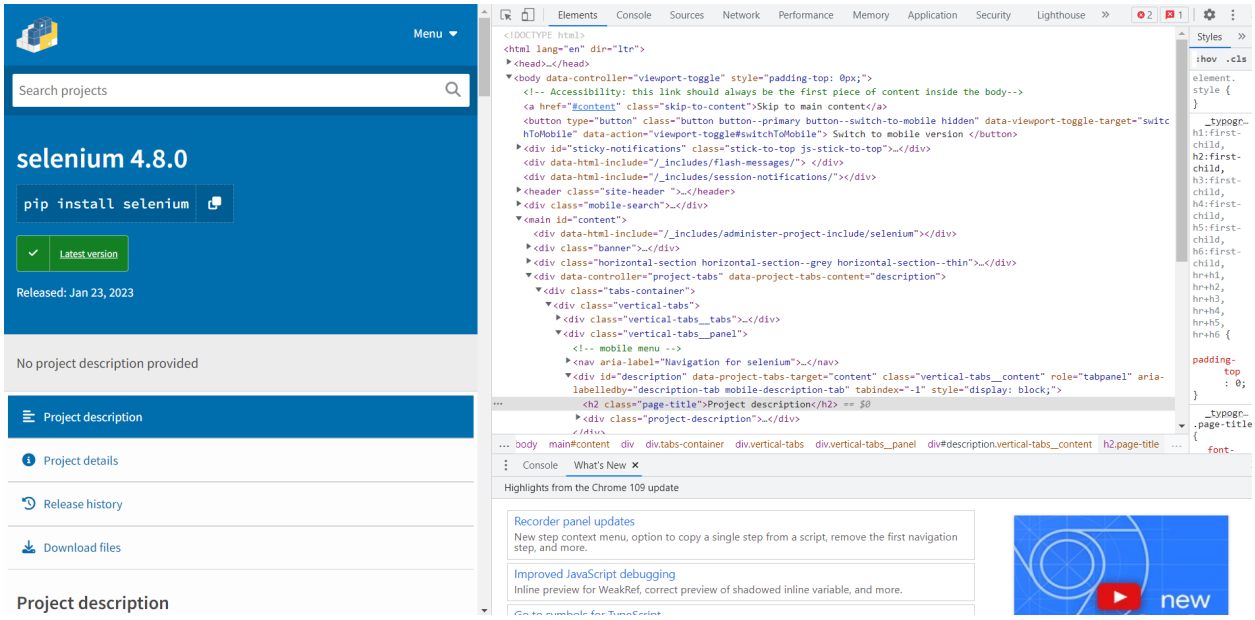
Brief Selenium and HTML Overview

Selenium is a Python module that is able to automate actions in browsers such as Google Chrome, Firefox, Microsoft Edge, or Safari.

Selenium uses a web driver object and is able to pick out and manipulate html elements from a web browser, so this segment will briefly explain the parts of HTML needed to use Selenium. You won't need to know how to code in HTML, but you will need to be able to identify certain elements in order to call them in Python.

The easiest way to find HTML elements in a web page is using the inspect element. To open it, right click on your web browser and click "Inspect." This should pull up a frame with HTML code on the right side of your screen. By hovering over different segments of code, you can see which HTML elements it includes. A screenshot of the inspect element is attached below.



The attributes Selenium uses to identify HTML files are their ID, name, class name, XPath, CSS selector, link text, partial link text, and tag name.

All html code begins with a tag, then adds attributes. The example on the next page is a part of the code from one of the elements of the main Selenium webpage, and will be explained in further detail.

```
<input id= “mobile-search” class= “search-form_search” type=
    “text” name= “q”
```

The tag, <input>, is highlighted in light blue. Every HTML element has a tag. Tags mark the beginning and ending of an HTML element, and can be used to find elements in Selenium. However, finding elements by tag name is not recommended in most cases as the same tags are used for multiple elements, making it difficult to find the specific element you want. Tags that will be used in the later demos are , or division, <a>, which means hyperlink, and <input>, which means the element takes an external input. When searching for a tag name in Selenium, use it as a string. With the above example, you would use “input” to search.

The other elements—ID, class, type, and name—are attributes. ID, name, and class are commonly used to find elements in Selenium. However, using class to search for elements was inconsistent when the 113 TA’s attempted it. You would use the string “mobile-search” to find the example element in Selenium.

XPath is the most consistent method to find elements with Selenium, but it is very inflexible. The XPath is the exact path from the top of the HTML code to a specific element, and will not work correctly if the path is incorrect. To find the XPath of an element, you must go into the inspect tool, and find the specific element you are looking for. As you hover your mouse over an element, it should be highlighted on the left side of the screen that displays the webpage. Once you find the element, right click it. In the drop-down menu, hover over “Copy” and press “Copy full XPath” to copy the XPath of that element. Looking for an element in the inspect tool can be tedious, especially for longer and more complex web pages, which is why ID, name, and class are used more often. “/html/body/main/div[4]/div/div/div[2]/nav/ul/li[2]/a” is an example XPath that could be used in Selenium.

Link text and partial link text search are exactly what they sound like. You put the link name or a part of the link name in to search for it. For example, on the Selenium webpage, searching for an element with link text “Project details” would select the hyperlink element with that display name. Partial text works the same way, except it only requires a part of the link text.

CSS selector requires tag name and one of the attributes of the element being searched for. CSS selector will not be utilized in the demos, so we won’t go as in-depth into it. A table for CSS searcher

syntax is below, taken from https://www.guru99.com/locators-in-selenium-ide.html.

Method	Target Syntax	Example
Tag and ID	css=*tag#id*	css=input#email
Tag and Class	css=*tag.class*	css=input.inputtext
Tag and Attribute	css=*tag*[*attribute*=*value*]	css=input[name=lastName]
Tag, Class, and Attribute	css=*tag.class*[*attribute*=*value*]	css=input.inputtext[tabindex=1]

Actual syntax for element searching will be discussed later, but this is to give you the base knowledge for HTML needed to use Selenium at this level.

Imports

Here's a few Selenium imports that will be used in the demos:

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

By allows us to search for elements in Selenium using different attributes.

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

Keys will allow us to send non-string keys to text input boxes.

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

ActionChains allow us to create action chains (duh).

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

WebDriverWait and **EC** allow us to perform explicit waits, where the webdriver checks if an element is in the driver and waits until it appears.

Basic Selenium Syntax

Here's a list of the basic Selenium commands used in the demos.

If you're interested in more complicated automation, we've included a list of suggestions at the end of this section.

```
driver.get("<link>")
```

This command takes a link in the form of a string as an argument, and opens the link in the current tab that Selenium is controlling.

driver.quit()
This command takes no argument and will close the automated browser window.

driver.back()
This command brings you to the previous webpage opened—it functions as the left arrow in the top left corner of the browser, and will not raise an error.

driver.forward()
This command functions as the right arrow in the top left corner of the browser. It will not raise an error.

driver.maximize_window()
This command maximizes the window that Selenium pulls up.

driver.window_handles
This is a list of all the current window handles in the Selenium-controlled browser. Taking specific indices will return the string name of that tab, so driver.window_handles[0] will return the string name of the leftmost tab.

driver.switch_to.window("<window handle name>")
This command will switch to the window in Selenium with the same name as the string <window handle name>.

driver.execute_script("<script>")
This command will execute the JavaScript command in <script>. In the demos, this was only used to open new tabs. The corresponding JavaScript is "window.open('');".

driver.find_element(By.<method>, "<string argument>")
This command will return **the first** HTML element that has the attributes specified on the current page Selenium has pulled up. <method> can be filled by any of the options discussed in the HTML overview: ID (By.ID), name (By.NAME), class name (By.CLASS_NAME), XPath (By.XPATH), CSS selector (By.CSS_SELECTOR), link text (By.LINK_TEXT), partial link text (By.PARTIAL_LINK_TEXT), and tag name (By.TAG_NAME). The way to find the string argument is defined in the HTML overview in the beginning of this writeup, and is also described in the demos. **Selenium will return an error if it does not find any element that matches the specifications.**

driver.find_elements(By.<method>, <string argument>)
This command is the same as driver.find_element except it returns a list of **all elements** on the current page Selenium has pulled up that contain the attributes specified. **Selenium will return an empty list if it does not find any element that matches the specifications.**

<element name>.click()
This command will attempt to click the current element stored in <element name>. **If it is not a clickable/interactable element, Selenium will raise an error message.**

<element name>.send_keys(<keys>)
This command will attempt to send the keys in the argument to the current element stored in <element name>. **If the element cannot have keys sent to it, Selenium will raise an error message.** <keys> can be a string with the text that the user wants to put into the element, or keys using the object Keys which was imported. The keys that are not string values must be accessed using Keys, including the enter key (Keys.RETURN), backspace key (Keys.BACKSPACE), arrow keys, and others. This is shown more clearly in the demos.

<element name>.get_attribute("<attribute name>")
This command returns a string of the attribute in <attribute name> from the element stored in <element name>. For example, using element.get_attribute("name") will return the name of the element as a string.

WebDriverWait(driver, <time in seconds>).until(EC.presence_of_element_located((By.<method>, "<string argument>")))
This is an ugly piece of code. However, this is an incredibly useful command. This command will wait up to the integer <time in seconds> until the driver has located an element with (By.<method>, "<string argument>"). Once it has been located, it will return the element. However, **Selenium will raise an error if <time in seconds> has passed and there is no element that matches the specifications.** This is useful when trying to automate web pages that load slowly.

<action name> = ActionChains(driver)
This creates an action chain! From there, you can add commands to the action chain, such as <action name>.move_to_element(<element>) or <action name>.click().

<action name>.perform()

This performs the actions stored in the action chain. This is similar to a function in that you can call it multiple times and will not change.

Overall, this is very surface-level stuff that you can do in Selenium.

More Selenium features can be found at https://selenium-python.readthedocs.io/api.html.

Additionally, the YouTube tutorials from Tech With Tim can be useful for extra demos. Here's the link to the first video in his series: https://www.youtube.com/watch?v=Xjv1sY630Uc&list=PLzMcBGfZo4-n40rB1XaJ0ak1bemvlqumQ&index=1.