**About Selenium**

* Selenium is a python package that allows you to be able to interact with html elements of a browser, automating and simulating actions such as clicking or typing in keys with these elements. This allows you to be able to create automated tests that simulate these actions, enabling performing tests in a repeated manner without manual input.

**About Selenium Scripts**

* I have created 4 python scripts, 2 being classes, that are the main components when making a new automated script. These being:
  + CommonSelenMethods.py
  + CreateData.py
  + ElementsDict.py
  + ExportScriptResults.py
* All four together are used within each new automated script, with further explanation of each below.

**About CommonSelenMethods.py**

* This is the main bulk of the automated tests. It is a class that creates an object (I call it selen\_ob in the main scripts) that contains the driver, timeout, steps list, dictionary, and actions. This is the main object you will be calling methods on within the main automated script to perform actions within the web browser.
* A picture containing text, software, multimedia software, font

  Description automatically generated
* There are of course, getter and setter methods for each of the components of the object.
* There is a long list of various methods within the class that are called upon the object to perform those actions in the web browser. Most of them are pretty self-explanatory, like clicking on something or typing in something.
* When you are calling these methods, you will usually be passing in the web element you want it to be used on, which is a string name denoting the name of the element from the dictionary which is from another class and will be explained a bit more later.
* A picture containing text, screenshot, font

  Description automatically generated
* You will notice a self.wait in most of these methods which is basically a wait timer that is called with a constant variable passed into it, so they all have a definite wait time that you can change all at once by changing the constant variable’s value at the top.
* A picture containing text, font, screenshot, design

  Description automatically generated
* You will notice some methods calling a create\_wait method. This method is the main method for creating the web element to be interacted with by passing in which element you want from the dictionary and waiting for it to be either clickable or visible before interacting with it.
* A screenshot of a computer program

  Description automatically generated with medium confidence
* This is in a try, except block, so the script will timeout if it is unable to locate the element.
* Another important method is the switch\_to\_diff\_iframe method. Some web elements on certain websites will be in different “iframes” so you need to switch between them in order to interact with elements within that iframe.
* A picture containing text, screenshot, font, software

  Description automatically generated
* You will notice the Steps being appended with certain strings. The Steps is a list of basically all the actions that the main object does, and every time it performs one, a string is appended to the list, and eventually outputted when the script is finishing running to show all the steps that were performed. This will be shown a bit later.

**About CreateData.py**

* CreateData is a python file that basically creates the main selen\_ob from the CommonSelenMethods class and the various parts of the object to be passed into when creating the object such as the Steps, dictionary, and timeout. This is where you set the main timeout for the script in general with a simple int.
* A screen shot of a computer program

  Description automatically generated with low confidence

**About ElementsDict.py**

* This class creates an empty dictionary object and has one very long method that creates the web elements and then proceeds to add them to their respective separate dictionary and then all those separate dictionaries are combined into one big dictionary of web elements at the end and returns it.
* **A screen shot of a computer program

  Description automatically generated with low confidence**
* The line breaks are mainly there for better organization and readability. It firsts lists out a number of variables each containing an xpath. This is information that helps the browser identify the web element in a web page’s html structure. Think of it is as a dna code.
* The variable will always start with a By.XPATH and then a unique xpath that will correctly identify the web element and nothing else. This requires a bit of work for finding the correct combination of xpath components to include and what to set them equal to correctly pick the right web element and nothing else on the web page.
* **A screenshot of a computer

  Description automatically generated**
* To identify a web element in the chrome browser, press ctrl+shift+I to pull up the inspect tool. This lists out all the html structure for the particular web page you are looking at.
* **A screenshot of a computer program

  Description automatically generated with medium confidence**
* Press this arrow icon to go into selection mode where if you click anywhere on the web page, it will show the corresponding html code that creates it and will highlight it in blue.
* **A picture containing text, electronics, screenshot, software

  Description automatically generated**
* Take for example, the google logo. If we wanted to add this to our elements dictionary, we need to figure out what unique parameters of the xpath to always correctly identify this web element and nothing else.
* The xpath might look something like this.
  + //img[@class=”lnXdpd” and @alt=”Google” and contains(@src, “png”)]
  + The first part after the two dashes denotes what type of element it is.
  + Parameters are denoted by an @
  + contains(@param, “specific text”) is very helpful if you just want a certain part of a param
* You can test out if your xpath correctly identifies the web element by press crtl+f, and then pasting in your xpath like so.
* **A screenshot of a computer program

  Description automatically generated with medium confidence**
* It will state how many web elements can be identified with that xpath and highlight them in green. You always want to make sure it’s just one. Though there are many cases where it’s impossible to create xpath for a certain element because there are multiple that have matching parameters. In that case you have to do a number of workarounds.
* An example would be //input[@class="asbasinp"])[2]. The “[2]” at the end denotes which number element you want, so this is the second one on the web page.
* There might be a case where you have to identify an element by selecting one that is unique above it in the html. For example below.
* **A screenshot of a computer program

  Description automatically generated with medium confidence**
* //div[@class="k1zIA rSk4se"]//child::img takes the div element that is underlined and at the end with “//child::img” is able to grab that img element. Think of the div element as a “parent” and the style element and img element as its “children.” The child tag tells the xpath to look for the children html below it and the “::img” tells it to look for the child that is a img element. If we did “::style” instead it would have chosen the style element instead.” You can also still use the “[number]” method as well with this to choose a certain child even further if there are multiple with the same type of element.
* Once you finally get a working xpath set it to the variable and then at the end add it to a separate dictionary with related elements.
* A screen shot of a computer program

  Description automatically generated with low confidence
* The key is the name of the element and then the value is the variable with the same name that contains the xpath.

**About ExportScriptResults**

* In the ExportScriptResults file there are two methods.
* A screen shot of a computer code

  Description automatically generated with low confidence
* The export\_setup method simply grabs the date and time and the path to store the results to. This is hard coded to a c drive location. It will create a folder called script results and then another folder within it with the date. The results file will have the script name and then the date at the end. If none of these folders exist, it will simply create them.
* A screen shot of a computer

  Description automatically generated with medium confidence
* The create\_and\_export\_script\_results method takes the Steps list that was mentioned before and will then append to the Status list the word pass for every string in the steps list to indicate which steps successfully passed in the script. It will then create 2 sheets in the excel file. One for the steps and statuses and another for any data to export.
* This is what the script results folder will look like
* A screenshot of a computer

  Description automatically generated
* And inside a folder are the various excel files
* A screenshot of a computer

  Description automatically generated with medium confidence
* And the excel file might look something like this.
* A screenshot of a computer

  Description automatically generated with low confidence
* There are three separate sheets at the bottom.
* I did include the ability for the script to take a screenshot once it finishes running which I have commented out. This step isn’t really necessary and it also creates a problem I will explain later.
* In the main script, the main method I will usually have the main steps in a try block and the exporting of results in a finally block. This means that if the script fails at any point, it will still execute the steps in the finally block, which means it will always export results.
* A picture containing text, screenshot, font

  Description automatically generated
* If you close the automated browser the finally steps will still execute, but if you enable the screenshot code, it won’t be able to since it needs the browser to exist in order to take the screenshot.