Local Library - Automated Testing Tutorial

Objectives: Provide students with instructions on how to use the Selenium WebDriver software to automate user interface functional testing using Python and the Python unit test library.

Background: Selenium web-drivers are widely used in software testing with a number of frameworks. We will be using the native Selenium commands and Pycharm to help us build our automated test scripts. Pycharm has built in support for Selenium and Python unit testing. The goal is to create software automation scripts which can be used as a part of continuous integration / continuous delivery of software.

Directions:

1. We will be using the Local Library application created in a previous assignment. Open this project in Pycharm. It is important that you make this application a Pycharm project if you did not already do so in order to use Selenium to automate your testing.

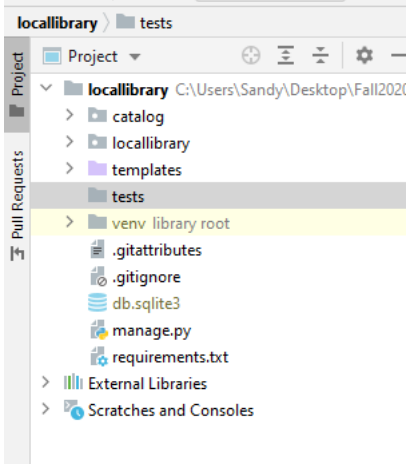
2. Open your Pycharm Project for the 'local library' assignment and open the Terminal window for the project if it is not already open.

3. Activate your virtual environment if it is not already activated. **If it is activated, you will see the (venv) to the far left of the prompt in the Terminal window of your application.**

   PC: The command to activate the virtual environment in the Terminal window is (assuming you named your virtual environment venv):
   venv\Scripts\activate
   Mac:
   source venv/Scripts/activate

4. Download and install the Selenium Python package. In the Terminal window, enter the following command:
   pip install -U Selenium
   (Be sure your Windows or Mac user account has admin rights to do this)
   For your reference the site is: https://pypi.Python.org/pypi/Selenium.

5. Download the Selenium WebDriver for your choice of browser:
   a. **Add a new folder/directory to the locallibrary folder (highest level folder/directory) for your application named 'tests'. This folder should be at the same level as the templates folder.**

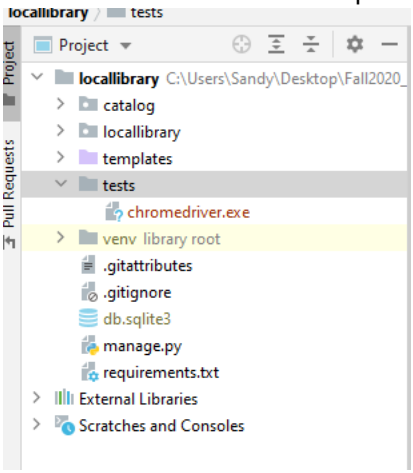


   b. **Mac - Safari**:
      i. No need to install the web driver for Safari: "Safari now provides native support for the WebDriver API. Starting with Safari 10 on OS X El Capitan and macOS Sierra, Safari comes bundled with a new driver implementation that's maintained by the Web Developer Experience team at Apple"
      ii. Before Safari will work with automated testing, you need to configure Safari to allow automation.

iii. As a feature intended for developers, Safari's WebDriver support is turned off by default.
iv. To turn on WebDriver support, do the following:
    1. Ensure that the Develop menu is available in Safari. It can be turned on by opening Safari preferences (*Safari > Preferences* in the menu bar), going to the *Advanced* tab, and ensuring that the *Show Develop menu in menu bar* checkbox is checked.
    2. Enable Remote Automation in the Develop menu. This is toggled via *Develop > Allow Remote Automation* in the menu bar.
    https://webkit.org/blog/6900/webdriver-support-in-safari-10/

c. **Windows and Mac with web driver other than Safari**:
    i. Go to the Python language bindings for Selenium WebDriver page: https://pypi.org/project/selenium/
    ii. Scroll down to the 'Drivers' table on this page and select the driver for the browser you prefer. Chrome is used in the demo below. For the Mac, use Safari or Chrome.
        1. Download the driver that matches your version of your web browser. Selenium provides versions for Chrome 89, 88, and 87.
    iii. Select the download for your system, saving the file to the folder you created in step 3a above.
    iv. Extract the files
    v. Move the driver application (executable file) to the 'tests' folder.

d. If you downloaded the chromedriver, your folders will now look like this in Pycharm (it will look similar for the other drivers as well with the exception of Safari on the Mac):



Note the Chromedriver.exe is in the 'tests' folder not in another folder.

6. Now we will begin to create our automated test script. Create the test Python files in the 'locallibrary/tests' folder.
    - We will use the built-in unittest library in Python which is similar to Junit in Java. This will allow us run one or more test scripts and provide feedback on the success or failure of the tests. 'Time' will be used to pause but not stop the script so you can view what is occurring as the tests execute. This can be removed or changed to 0 when using for real testing. Do not remove these before running your tests. The Time may need to be increased in order for the process to communicate with the browser.
    - **The files in this demonstration use the Chrome driver,** but you may use the driver of your choice. Change the appropriate lines in the test cases to reflect your choice of driver.

7. In the locallibrary/tests folder, create a new Python file -> select 'Python unit test' as the file type, and name the file **unitTest_Login.py**

8. Replace the text in the file with the code shown below – IF NOT USING THE CHROME DRIVER, CHANGE THE DRIVER TO REFLECT YOUR DRIVER OF CHOICE.

```
9. import unittest
   import time
   from selenium import webdriver
   from selenium.webdriver.common.keys import Keys
   from selenium.common.exceptions import NoSuchElementException


   class ll_ATS(unittest.TestCase):

       def setUp(self):
           self.driver = webdriver.Chrome()

       def test_ll(self):
           user = "testuser"
           pwd = "test123"

           driver = self.driver
           driver.maximize_window()
           driver.get("http://127.0.0.1:8000/admin")
           elem = driver.find_element_by_id("id_username")
           elem.send_keys(user)
           elem = driver.find_element_by_id("id_password")
           elem.send_keys(pwd)
           time.sleep(3)
           elem.send_keys(Keys.RETURN)
           driver.get("http://127.0.0.1:8000")
           time.sleep(3)
           #assert "Logged in"
           try:
               # attempt to find the 'Logout' button - if found, logged in
              elem =
   driver.find_element_by_xpath("/html/body/div/div/div[1]/ul[2]/li[3]/a")

              assert True

           except NoSuchElementException:
               self.fail("Login Failed - user may not exist")
               assert False

           time.sleep(3)


   def tearDown(self):
       self.driver.close()


   if __name__ == "__main__":
       unittest.main()
```

REPLACE THE VALUES 'testuser' AND 'test123' IN THE ABOVE CODE WITH A VALID USERNAME AND PASSWORD FOR YOUR APPICATION OR CREATE A USER WITH THIS USERNAME AND PASSWORD FOR YOUR APPLICATION. You will need to use a username and password you created for your application in the above file.

Note: All the commands are defined in the Selenium WebDriver documentation which provide the same commands in several popular languages. Click here for detail Selenium Commands. This site also lists pros and cons of each of the browser drivers.

Let's walk through the code shown above. The test_ll method is defined and provides a place for a user ID and password we need to log onto our application.

Next the Chrome browser is opened and maximized.

Now we get to an important part of any screen reading tool and that is **LOCATING THE SCREEN ELEMENT.** In order for you to use application components in a browser screen, you need to be able to identify to the program the element you want to manipulate.

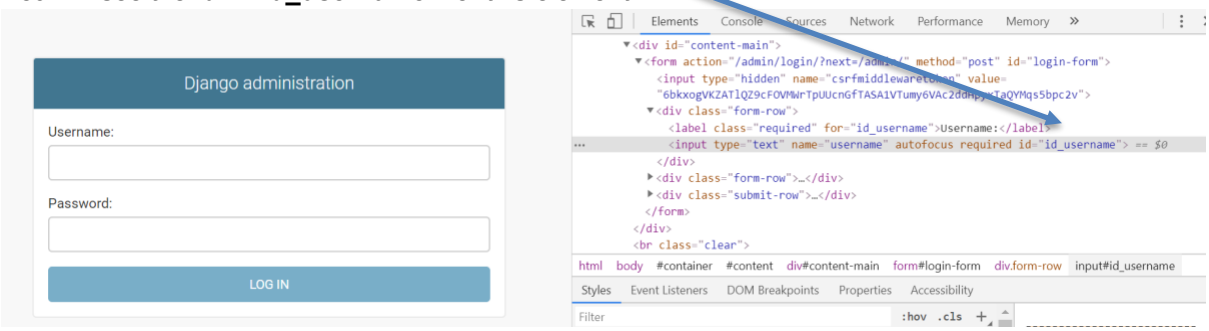
The first element we want to find is the username field on the admin screen:

```
elem = driver.find_element_by_id("id_username")
```

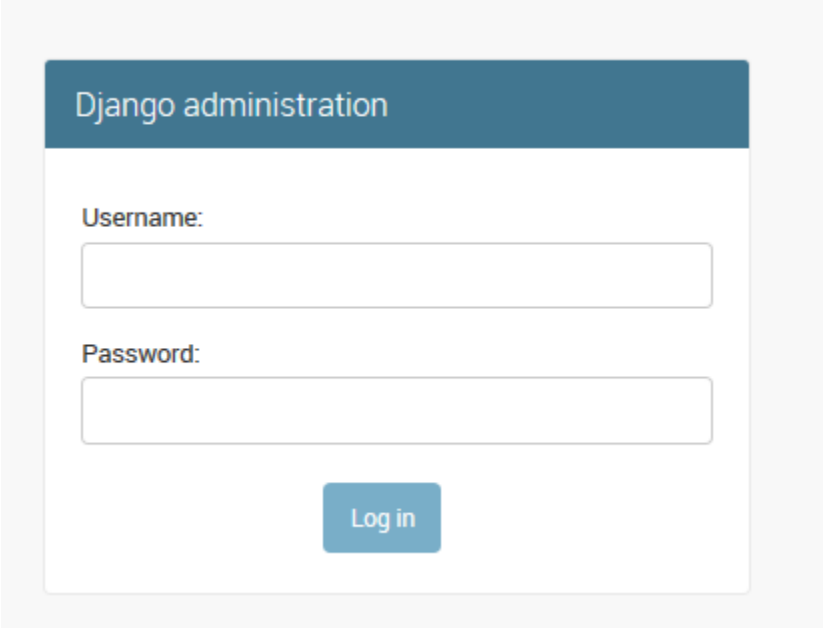
To find an element's ID, show the source html in a browser and find the ID of the element.

In Chrome, for example, right-click on element you want to use. In this case, right-click on the username text box and select 'Inspect'. The html for this element will be highlighted and will show the information about this element on the screen, including the id.

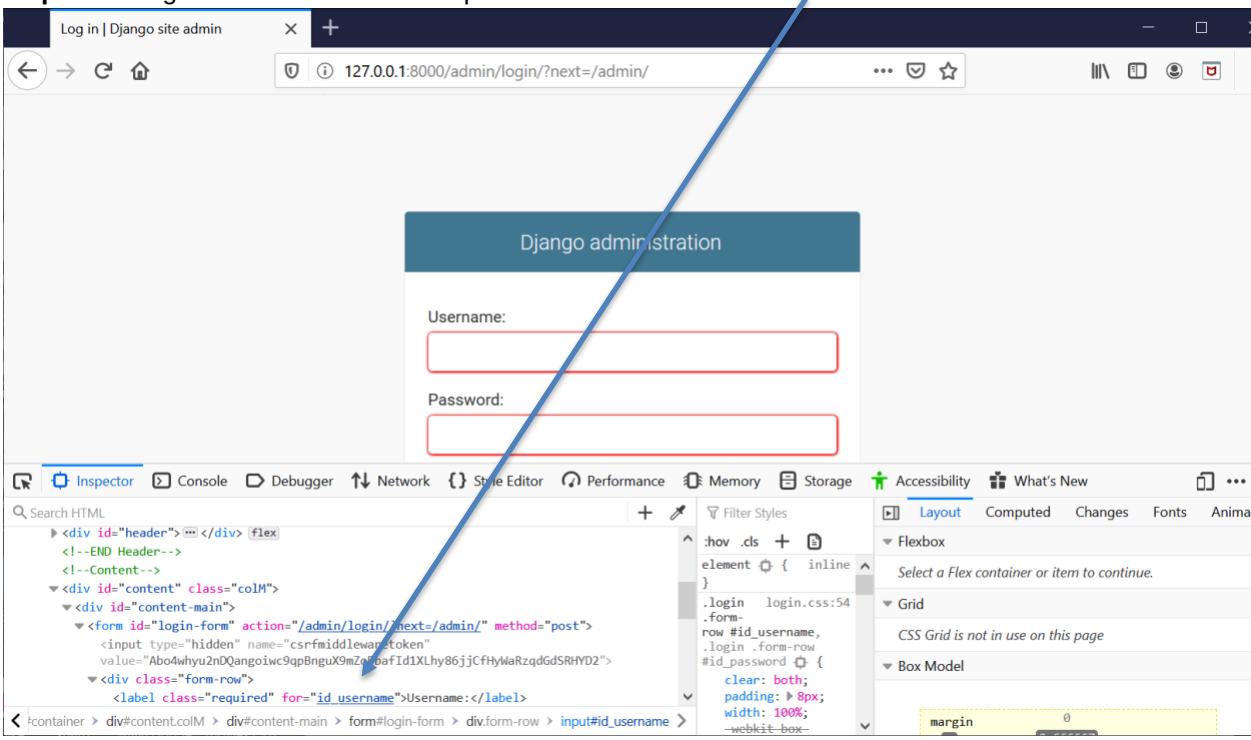
You will see the **id = "id_username"** for this element.



The Firefox browser also provides an easy way to view the element ids:



We need to navigate to the Username box. Below shows the Firefox developer tool called **Inspector**. Right-click the Username input box to view the name.



Both the User ID and Password boxes are clearly labeled with ID labels.

Next: we send the username box the 'keys' or text for the user name:

```
elem.send_keys(user)
```

Then we repeat these steps for the password box.

Next, we send the RETURN command which is like pressing enter on many keyboards, and if the username and password are correct, we are logged in.

```
elem.send_keys(Keys.RETURN)
```

Next, we return to the application home page to prepare for the next action with:

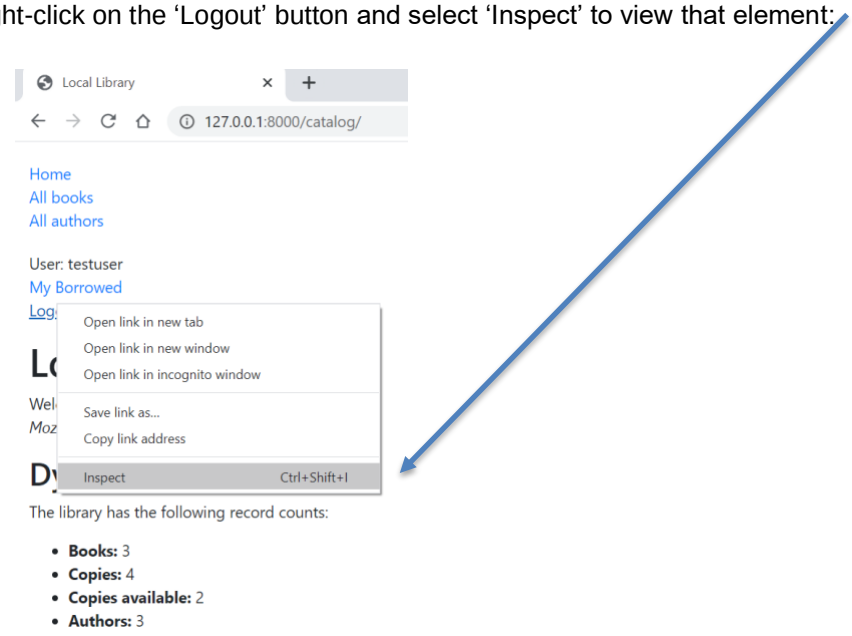
```
driver.get("http://127.0.0.1:8000")
```

Next, we try to find the 'Logout' symbol to determine if the login was successful. If the 'Logout' is found, the logon was successful, and we set the assert status to true. If it was not found, logon failed, and we set the self.fail message and set the assert status to false indicating the test failed.

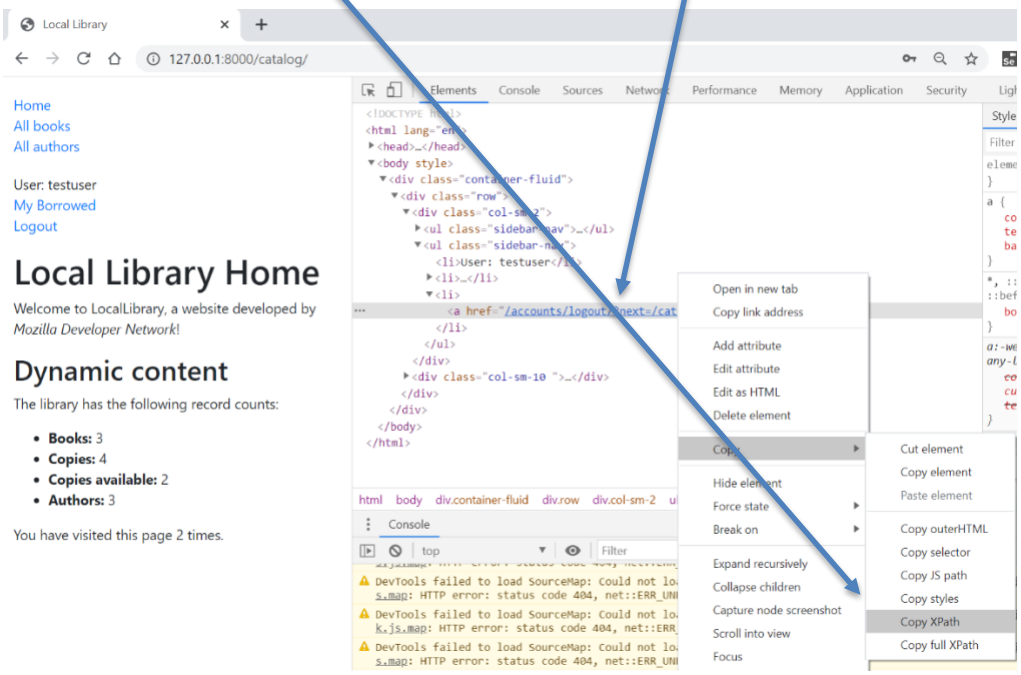
```
try:
# attempt to find the 'Logout' button - if found, logged in
    elem =
   driver.find_element_by_xpath("/html/body/div/div/div[1]/ul[2]/li[3]/a")

   assert True

except NoSuchElementException:
    self.fail("Login Failed - user may not exist")
    assert False
```

Note that for this element, we will locate the item by 'XPath'.
To find the 'XPath' of the element, log on using the **http://127.0.0.1:8000/admin** page, go back to the main page, **http://127.0.0.1:8000**,

In Chrome: Right-click on the ‘Logout’ button and select ‘Inspect’ to view that element:



then right-click on the high-lighted text in the Inspect pane, then select ‘Copy –>Copy XPath’.



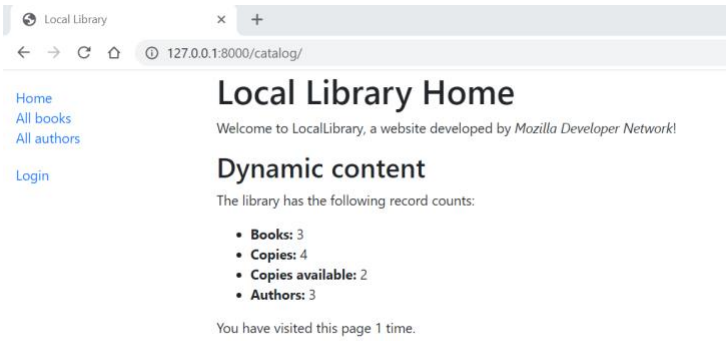
<u>In Firefox:</u> Right-click on the + and select ‘Inspect Element’. Then right-click on the highlighted text and select ‘Copy - > XPath’

This XPath is then pasted into the Python file for the driver.find_element_by_xpath method.

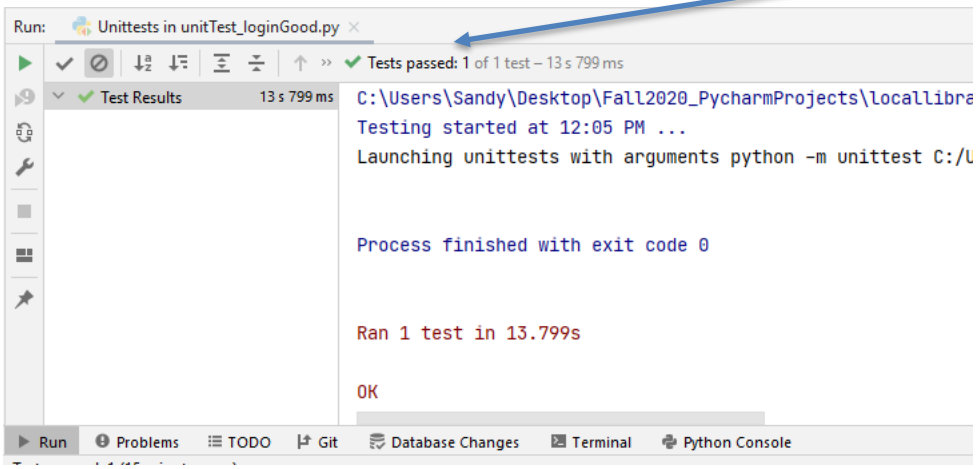
In order for this test to pass successfully, you will need to create an account with the username of ‘testuser’ and a password of ‘test123’ (no quotes). Alternatively, you can change the values in the test file to use an account you previously created.

IMPORTANT: The application must be running for the tests to execute successfully. Use the standard command to run the server from the Terminal window:
python manage.py runserver

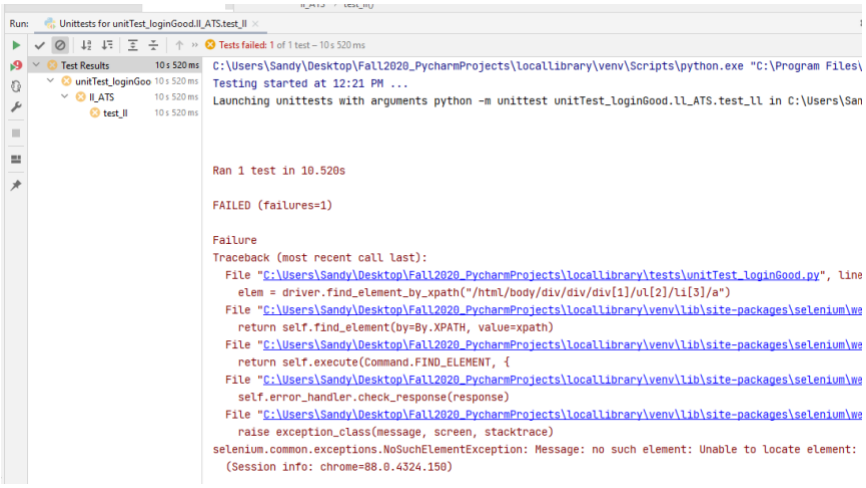
10. **Do not sign in as a user at this point**. We will be testing the login process. Log out of the application if you were logged in but leave it running locally. Note the "**Logout**" text is not visible in the screen shot shown below:



11. Run the unitTest_loginGood.py in Python by right-clicking the file 'unitTest_loginGood.py' file or selecting Run from the menu bar with the 'unitTest_loginGood.py' file open in PyCharm. You should see the application page open and the login process execute automatically as defined in the unit test file. The test is successful when you see something similar the following in the Pycharm 'Run' pane – specifically notice 'Tests passed:



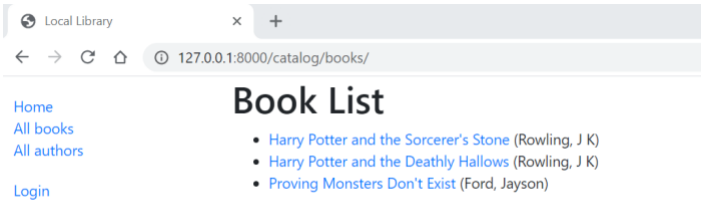
12. If the test fails, you will see something like the following in the Run pane (you will need to scroll through the Run pane to view this output - specifically notice 'Tests failed:



13. If you scroll down in the test window, you will see a number of error messages. Towards the bottom, you will see something similar to the following text displayed below. Note that this is expected if the user does not exist when we are trying to test the login capability:

```
AssertionError: Login Failed - user may not exist



Assertion failed

Process finished with exit code 1


Assertion failed

Assertion failed
```

14. In the next test, we will verify the Book List page is displayed when the user clicks the ‘All books’ button in the navigation pane. the add of a new Client. To develop this automated test, we need to determine what actions happen to bring up the add ‘Book List’ screen below. Note that you may not have any books in your database, but the Book List should still be displayed when the ‘All books’ button is clicked.



The user clicks on the “All books” button in the navigation panel We will mimic this process in the unit test file. Note that for this test, the user need not be logged in to the appication.

- We will treat the ‘All books’ like a button.
- We will need to find the XPath address of the object (All books).
- You can find the XPath as demonstrated in the previous test above.

15. In the ‘locallibrary/tests’ folder, create a new Python file -> select ‘Python unit test’ as the file type, and name the file **unitTest_viewBookList.py**

Add the code shown on the next page to this file – note where the XPaths are referenced.

Reminder: IF NOT USING THE CHROME DRIVER, CHANGE THE DRIVER REFERENCED IN THE PYTON CODE TO REFLECT YOUR DRIVER OF CHOICE.

.

```
# Unit test file to determine if the Book List page is displayed when the user
# clicks the 'All books' button in the navigation pane of the local library
# application

import unittest
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.common.exceptions import NoSuchElementException


class ll_ATS(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Chrome()

    def test_ll(self):

        driver = self.driver
        driver.maximize_window()

        driver.get("http://127.0.0.1:8000")
        time.sleep(3)
        # assert "Logged in"
        # find 'All books' and click it - note this is all one Python statement
        elem =
driver.find_element_by_xpath("/html/body/div/div/div[1]/ul[1]/li[2]/a").click()

        time.sleep(5)
        try:
        # verify Book List exists on new screen after clicking "All books" button
        # attempt to find the 'Logout' button - if found, logged in
            elem = driver.find_element_by_xpath("/html/body/div/div/div[2]/h1")

            assert True

        except NoSuchElementException:
           self.fail("Book List does not appear when All books clicked")
           assert False

    time.sleep(2)


def tearDown(self):
    self.driver.close()


if __name__ == "__main__":
    unittest.main()
```

Ensure your application is running and run this Python unit test program. You will see the application open, and the login, the entries for the New Client and the Save. This now completes the test and we can assert that we have created a new client. When you run this in PyCharm, it should look like this in the Run pane:

If the test fails, you will see a message that the test failed, similar to the error message displayed for the previous test.

CREATE YOUR OWN AUTOMATED USER INTERFACE TESTS:

Now it is your turn to create your own Python test scripts as described in the assignment. The Selenium commands that you have learned in this tutorial will help you to accomplish this. Most CRUD applications involve text boxes, tabs, and buttons. You need to identify these and take the appropriate action as you would when you are running a manual functional test.

You should write at least 3 automated test scripts to validate your interface. Note that these tests are not designed to test if the database returns the appropriate data when requested. Database tests are not conducted with Selenium.

POSSIBLE ISSUES WHEN RUNNING YOUR TESTS:

ISSUE 1 – 'CLICK' OPERATION NOT PROCESSED:

The above test to verify Book List screen appears when the 'All books' is clicked.
This test may fail, especially on the Mac. For some reason, the 'click' operation is not processed.

To fix this problem, replace this line:

```
elem = driver.find_element_by_xpath("/html/body/nav/ul/li/a").click()
```

With these 2 lines:

```
elem =
driver.find_element_by_xpath("/html/body/div/div/div[1]/ul[1]/li[2]/a")
elem.send_keys(Keys.RETURN)
```

ISSUE 2 – TESTS REPORT FAILURE WHEN THEY SHOULD BE SUCCESSFUL

If your computer is running slow, you may also need to increase the 'sleep' times to allow the system to display the screen correctly.
Experiment with longer sleep times to see if the problem is resolved.
For example, replace

```
time.sleep(3)
```

with

```
time.sleep(5)
```