



# Python Desktop Application Testing with Appium

## Course Outline:

[Module 1 - Appium/WinAppDriver](#)

[Module 2 - General Framework Discussion and Overview](#)

[Module 3 - Python Concepts used in Appium Framework Development](#)

[Module 4 - Testing Local Applications with Appium and Unit Test Frameworks](#)

[Module 5 - Testing with the UI Recorder](#)

[Module 6 - Adding Assertions and Object Synchronization](#)

[Module 7 - Options for Execution and Debugging](#)

[Module 8 - Creating a WPF Application](#)

[Module 9 - Test Case Template Leveraging PyTest](#)

[Module 10 - Page Object Model \(POM\)](#)

[Module 11 - Adding Data Inputs for the POM](#)

[Module 12 - Test Driven Development \(TDD\) and the Page Object Model](#)

[Module 13 - Keyword Driven Framework \(KDF\)](#)

[Module 14 - Adding Data Inputs to the KDF](#)

[Module 15 - User Acceptance Testing \(UAT\) and the Keyword Driven Framework](#)

[Module 16 - Enhancing the Template with Reporting Capability](#)

[Module 17 - Save and Utilize the Template on a New Application](#)

[Module 18 - Installing JMeter](#)

[Module 19 - Creating and Using Assertions](#)

[Module 20 - Creating and Using Listeners](#)

[Module 21 - Testing Web Services \(API\)](#)

[Module 22 - Getting to Know JMeter Topologies](#)

# Module 1 - Appium/WinAppDriver

## Key Points:

- Appium is an Open Source test automation framework that can be used to test Native, Hybrid, and Mobile web applications (primarily used for mobile testing).
  - Appium is built as an extension to Selenium, so it can support languages such as Java, Python, Ruby, C#, JavaScript, and PHP or any language that supports Selenium.
- Appium is built to support cross platform testing, allowing users to reuse code with the same API for iOS, Android, and Windows Applications.
- Features:
  - Client/Server Framework: Appium receives connections from the client, reads provided commands, performs given tasks on the chosen application, and returns a HTTP response.
  - Desired Capabilities: information sent to the Appium server that distinguishes capabilities of the testing session such as the platform name (e.g. Windows, etc.) and application.
  - Session: Clients must begin a session prior to testing, allowing for Appium to receive requests, desired capabilities, and commands.
  - GUI: includes features to set capabilities, an inspector to manage the application's elements, and JSON representation of commands.
- WinAppDriver is a UI test automation tool developed by Microsoft for solely Windows applications that replicates Selenium Web Driver.
- WinAppDriver tests Universal Windows Platform (UWP), Windows Forms (WinForms), Windows Presentation Foundation (WPF), and Classic Windows (Win32) apps on Windows 10 PCs.

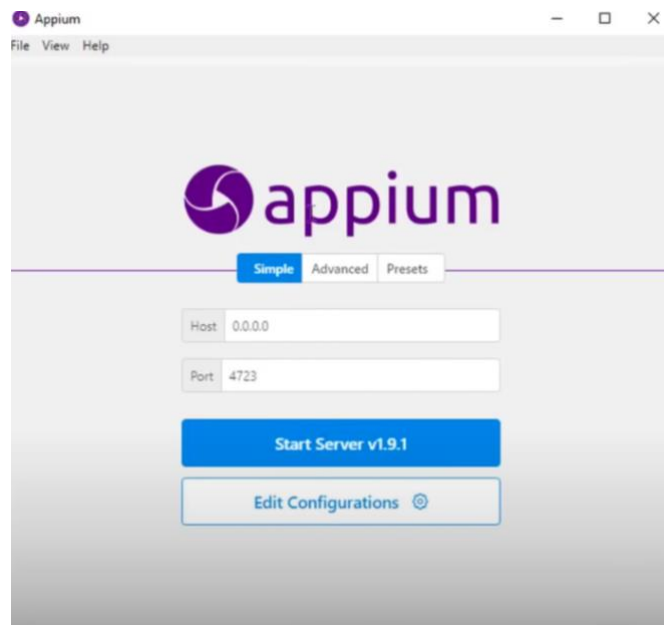
- Windows Inspector Tool is a Windows-based tool that allows users to view any UI element's accessibility information and automation properties.
- Inspector also provides a branch structure of the application's automation elements.

The following exercise outlines Appium installation:

### Exercise 1: Installation of Appium:

1. **Download the latest, stable release of the Appium GUI, found on GitHub.**
  - <https://github.com/appium/appium-desktop>
2. **Install Appium Desktop Client by completing the Appium setup executable.**

**Target:** Confirm that you have successfully launched Appium!



---

The following exercise outlines WinAppDriver installation:

### Exercise 2: Installation of WinAppDriver:

1. Download the latest, stable release of WinAppDriver, found on GitHub.
  - <https://github.com/microsoft/WinAppDriver>
2. Enable developer mode in Developer Settings.
3. Install Windows Application Driver by completing the setup executable.
4. Navigate to the location of WinAppDriver.exe from the installation directory and start the program.
  - C:\Program Files (x86)\Windows Application Driver
  - Optional: Add a shortcut to WinAppDriver to allow for easier access from the desktop.

**Target:** Confirm that you have successfully launched WinAppDriver!



---

The following exercise outlines Windows Inspector Tool installation:

### **Exercise 3: Installation of Inspect:**

1. Download the latest version of Windows 10 SDK, found on Microsoft's website.

- Inspect can only be found within Windows 10 SDK (Software Development Kit).
- <https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk/>

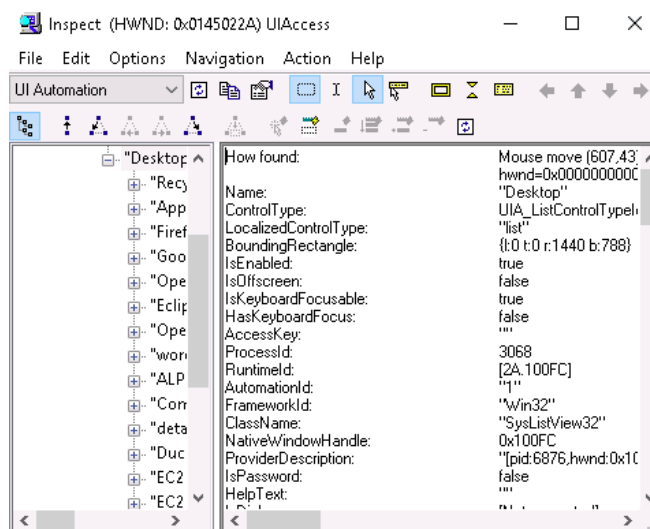
**2. Ensure PyCharm is closed prior to installation.**

**3. Install Inspect by completing the SDK setup executable.**

**4. Navigate to the location of Inspect.exe from the installation directory and start the program.**

- C:\Program Files (x86)\Windows Kits\10\bin\10.0.19041.0\x64
- Optional: Add a shortcut to Inspect to allow for access from the desktop.

**Target:** Confirm that you have successfully launched Windows Inspector Tool!



## Module 2 - General Framework Discussion and Overview

Key Points:

- Why is a Framework important?
  - Allows quick reactions to changes in the app.

- Allows users to handle object property changes.
- Ability to include more people in the Test Case development process.
- Ability to share code with other developers in the group (commenting) to explain the developer's reasoning behind code.
- Simplifies onboarding new people by allowing new developers to understand past tests.
- Abstractions from the Code, reusable code.
- Framework Generations:
  - Generation 1: (Record & Playback)
    - Pros: able to get it working very quickly, playback for a short period of time, generators code for us.
    - Cons: not maintainable over time, recordings are 1 for 1 with test cases, multiple, repetitive fixes when needing to adjust scripts.
  - Generation 2: (Custom Code Focused)
    - Pros: more control using the tools of each programming language.
    - Cons: heavy coding involved and leaves people outside of the test case development process.
  - Generation 3: (Frameworks)
    - Pros: lots of structure, using more OOP concepts for structure instead of pure code, centralizes test case data from our scripts (removes 1 for 1 recordings from Gen.1), and is able to involve subject matter experts in the TC development.
    - Cons: not appropriate for all situations (I.E. small routines).
- Framework Structures (Gen. 3):
  - Page Object Model (POM)
    - Pages:

- Page Name (i.e. LoginPage).
  - Constructors
  - Objects or Elements (i.e. UserName, Password, LoginButton).
  - Methods [i.e. logIntoApplication() - use the 3 above objects to login with correct values], [i.e. failLogin() - use the 3 above objects to login with incorrect values].
  - NOTE: with POM, methods and data are together.
- Test-Suite (Template Format):

Test-Suite (file)

StartTest()

LogIntoApplication()

failLogin()

AdditionalPagesMethod()

StartTest2()

failLogin()

AdditionalPagesMethod()

- Keyword Driven Approach:
  - Uses an external data source to drive the automation code (CSV, Excel, or Database).
  - Accessed during runtime and drives the automation code based on inputs.
  - Example (outside file):
    - TestCase, Page, Object, Value, Keyword
    - TC1, Login, UserName, pqj, EnterData (Editbox)
    - TC1, Login, Password, 123, EnterData (Editbox)
    - TC1, Login, LoginButton, n/a, Click (Button)



- Elements such as "EnterData", and "Click" are the methods, which can be used over and over.
- Benefits (for the organization or team):
  - Methods done in POM, which required more programming, can now be done by outside system analysts.

The following exercise tests concepts from Modules 1 and 2:

### **Exercise 1: Quiz**

1. Appium supports cross platform testing, allowing users to reuse code for iOS, Android, and Windows Applications. ( T / F )
2. Explain how Appium's Client/Server framework operates.
3. Windows Inspector Tool can be used to inspect lines of code within an application. ( T / F )
4. Explain why Desired Capabilities are necessary for an Appium session.
5. Frameworks prevent new developers from joining the development process, adding a layer of security to tests. ( T / F )
6. List and explain three reasons why frameworks are helpful and necessary.
7. Appium is an extension to Selenium, so it supports any Selenium-supported language. ( T / F )
8. Compare and contrast Generation 1 and 2 testing.
9. The Page Object Model and Keyword Driven frameworks are Generation 2 Testing. ( T / F )
10. Explain why Custom Code Focused frameworks are a bad choice for people with little coding experience.

11. The Page Object Model is a UI repository design pattern in Appium, in which each application page is represented by a Java class with a corresponding test page. ( T / F )
  12. Explain what objects are in the Page Object Model framework.
  13. The Appium GUI includes features to set capabilities, an inspector to manage the application's elements, and JSON representation of commands. ( T / F )
  14. Explain how Keyword Driven frameworks utilize external files.
  15. The four parts of a Page Object Model framework are Page Name, Objects or Elements, Methods, and Files. ( T / F )
  16. Explain some of the benefits of Generation 3 frameworks.
  17. WinAppDriver was originally built to test Windows applications, but now it can also test mobile applications such as IOS and Android. ( T / F )
  18. List and explain benefits of the Keyword Driven Approach.
  19. Frameworks allow users to handle object property changes. ( T / F )
  20. Compare and contrast the cons of Generation 2 and 3 frameworks.
- 

## Module 3 - Python Concepts used in Appium Framework Development

### Key Points:

- Reference Types:
  - All values in Python are references.
  - When you declare a variable of a reference type, the variable contains the value *None* until you explicitly create an instance of the class.

- Or, you can assign it an object of a compatible type that has already been created.
  - The created variable holds a reference to the location of the object.
- Declaring Classes:
  - To define a class, state *class* and a unique class name.
  - Within the class body, users can state constructors, properties, methods, and events (class members).
- Creating Objects:
  - An object is a concrete entity based on a class (an instance of a class).
  - To create an object, state the object name and the name of the class the object is based on
    - i.e. `object1 = Customer();`
  - The name (object1) is a reference to an object based on the Customer class.
- Class Inheritance:
  - Classes allow users to inherit from any other class and implement multiple inheritance.
  - Users must use a derivation to specify the base class to inherit data and behavior from (depicted with parenthesis and base class following the class definition).
    - i.e. `public class Manager (Employee)`
  - In this scenario, the new class *Manager* inherits data from the *Employee* class.
  - Classes can be declared as *abstract*, meaning that they can not be instantiated and instead can only be used through derived classes (classes that implement its methods).
- Reflection:
  - Reflection provides objects that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an

existing object, or get the type from an existing object and invoke its methods or access its fields and properties.

- Reflection can be used to access attributes, instantiating types, and building new objects at runtime
- To invoke a method at runtime, load the DLL via `Assembly.Load` (or `Assembly.LoadFrom`) and then call `Assembly.GetTypes`. For each type, call `Type.GetMethods`. When you have a `MethodInfo`, you can call `MethodInfo.Invoke` on it.
- For the Keyword Driven approach to work (because of external sources), we need to be able to access our POM classes, pages, and object properties at runtime.

The following exercise demonstrates the creation of classes and methods:

### **Exercise 1: Classes and Methods**

#### **1. Open PyCharm, create a new project, and name it “ClassesAndMethods.”**

- Name the file “ClassesAndMethods” and store the file in `C:\Users\Administrator\Documents\Appium Python Training\ClassesAndMethods`. If this folder has not been created, create a new folder in this location.
- Creating a folder for each individual project allows for easier storage.

#### **2. Declare a public class, titled Person.**

```
class Person:
```

#### **3. Within this class, create a new initializer with a string variable.**

```
# Initializer that takes an optional argument:
```

```
def __init__(self, Name=None):
```

```
self.name = Name
```

**4. Create a method to represent objects of the class as a string.**

```
# Method that represents the class as a string.
```

```
def __str__(self):
```

```
    return 'This is a person named {}'.format(self.name)
```

**5. Create another class beneath the first, titled TestPerson.**

```
class TestPerson:
```

**6. Create an object of the first class named person1.**

```
# Call the constructor of the first class
```

```
person1 = Person()
```

**7. Include code to display person1's name.**

```
# Get the string representation of the person2 instance
```

```
print(person1)
```

**8. Call the second created constructor with the parameter *Sarah Jones*.**

```
# Call the constructor and pass in a name
```

```
person2 = Person("Sarah Jones")
```

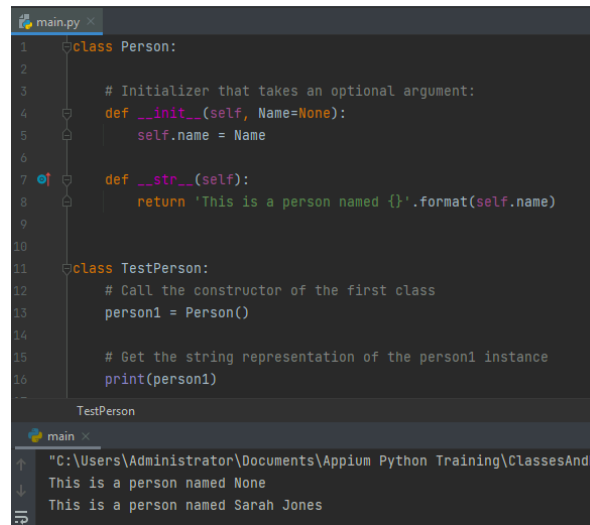
**9. Lastly, include code for the console to display person2's name.**

```
// Get the string representation of the person2 instance.
```

```
print(person2)
```

**10. Run the program! Ensure that the output matches the expected results.**

**Target:** Confirm successful creation of classes and methods!



```
1 class Person:
2
3     # Initializer that takes an optional argument:
4     def __init__(self, Name=None):
5         self.name = Name
6
7     def __str__(self):
8         return 'This is a person named {}'.format(self.name)
9
10
11 class TestPerson:
12     # Call the constructor of the first class
13     person1 = Person()
14
15     # Get the string representation of the person1 instance
16     print(person1)
```

TestPerson

main

"C:\Users\Administrator\Documents\AppData Python Training\ClassesAndMe  
This is a person named None  
This is a person named Sarah Jones

The following exercise demonstrates the use of reflection to reverse a sequence:

## Exercise 2: Reflection

### 1. Open PyCharm, create a new project, and name it “Reflection.”

- Name the file “Reflection” and store the file in

C:\Users\Administrator\Documents\AppData Python Training\Reflection. If

this folder has not been created, create a new folder in this location.

### 2. Create a method that will reverse a sequence.

```
def reverse(sequence):
```

### 3. In the method, add two variables to store the type of the parameter and an

empty object of the parameter type.

```
sequence_type = type(sequence)
```

```
empty_sequence = sequence_type()
```

### 4. Include conditional logic in case of empty parameters.

```
if sequence == empty_sequence:
```

```
    return empty_sequence
```

**5. Include the following code to reverse the sequence.**

```
rest = reverse(sequence[1:])  
  
first_sequence = sequence[0:1]
```

**6. Combine the two variables to create the final result.**

```
# Combine the result  
  
final_result = rest + first_sequence
```

**7. Return the resulting sequence.**

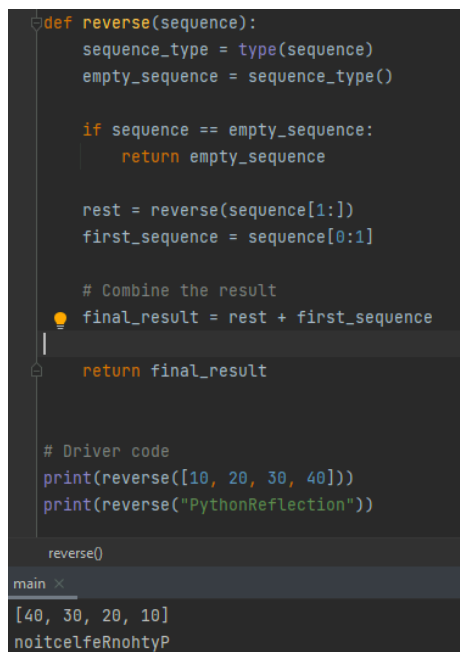
```
return final_result
```

**8. Now, include the following to display the values of the reversed sequences on the console.**

```
# Driver code  
  
print(reverse([10, 20, 30, 40]))  
  
print(reverse("PythonReflection"))
```

**9. Run the program! Ensure that the output matches the expected results.**

**Target:** Confirm successful use of Python Reflection!



```
def reverse(sequence):  
    sequence_type = type(sequence)  
    empty_sequence = sequence_type()  
  
    if sequence == empty_sequence:  
        return empty_sequence  
  
    rest = reverse(sequence[1:])  
    first_sequence = sequence[0:1]  
  
    # Combine the result  
    final_result = rest + first_sequence  
  
    return final_result  
  
# Driver code  
print(reverse([10, 20, 30, 40]))  
print(reverse("PythonReflection"))  
  
reverse()  
main ×  
[40, 30, 20, 10]  
noitcelfErnohtyP
```

## Module 4 - Testing Local Applications with Appium and Unit Test Frameworks

### Key Points:

- Both Appium and WinAppDriver can be used for testing automation; however, they differ in capabilities and design.
  - Appium is predominantly designed for mobile testing (iOS, Android devices) but when it is paired with WinAppDriver, it grants Appium the capability to test Windows applications.
  - WinAppDriver, an Appium-compatible WebDriver, solely automates testing for Windows Desktop Applications and can be used without Appium.
  - To begin a session with WinAppDriver, include the following Desired Capabilities:
    - platformName: Windows
    - deviceName: WindowsPC
    - app: <AppBeingTested>
- Supported Capabilities that can be used to create a WinAppDriver Session:

Capabilities:	Descriptions:	Example:
app	application identifier/executable path	C:\Windows\System32\Notepad.exe
appArguments	launch arguments	<a href="https://github.com/Microsoft/WinAppDriver">https://github.com/Microsoft/WinAppDriver</a>
appTopLevelWindow	existing application top level window to attach to	0xB822E2
appWorkingDir	application working directory	C:\Temp
platformName	target platform name	Windows
platformVersion	target platform version	1.0

- Both Windows Inspector Tool and Appium Desktop Client can be used to locate any UI



element's automation properties, which are used to test functionality of buttons, textboxes, etc. Additionally, WinAppDriver's UI Recorder can be used to locate UI elements, which we will cover later in this course.

- Windows Inspector Tool has more comprehensive data on Windows Desktop applications
- Appium Desktop Client has more user-friendly, modern GUI for inspecting automation properties, yet functionality varies on Windows applications
- Supported Locator Elements to find UI Elements in Inspect.exe:

Client API:	Locator Strategy:	Attribute in Inspect	Example:
FindElementByAccessibilityId	accessibility ID	AutomationId	AppNameTitle
FindElementByClassName	class name	ClassName	TextBlock
FindElementById	ID	RuntimeId (decimal)	42.333896.3.1
FindElementByName	name	Name	Calculator
FindElementByTagName	tag name	LocalizedControlType (upper camel case)	Text
FindElementByXPath	xpath	Any	//Button[0]

- There is two main ways to automate testing (UnitTest, aka PyUnit, and PyTest)
  - UnitTest is the standard automation framework for Python
  - Inspired by JUnit, all assertion methods/cleanup are provided through the base class *TestCase*
  - PyTest is an installed framework that allows for easy testing, both small and large scale. PyTest is preferred to UnitTest as it follows standard Python naming convention and includes additional features (parameterization and ordering).

- To run tests using Pytest, use the following code in the terminal:

Selected Tests	Code:
All tests in a project	pytest
All tests in a directory	pytest tests/my-directory
All tests in a file	pytest tests/my-directory/test_demo.py
One test method	pytest -v tests/my-directory/test_demo.py::test_specific_function
One class of tests	pytest -v tests/my-directory/test_demo.py::TestClassName
One method within a class	pytest -v tests/my-directory/test_demo.py::TestClassName::test_specific_method

The following exercise outlines the creation of a basic test to automate Notepad:

### Exercise 1: Notepad Automation Test:

#### 1. Open PyCharm and create a new project.

- Use a blank template to create the Python test.
- Name the file “AppiumTest1” and store the file in  
C:\Users\Administrator\Documents\AppData Python Training\AppDataTest1.  
If this folder has not been created, create a new folder in this location.

#### 2. In the menu bar, click on File, then Settings. Open the Project:

**AppiumTest1** tab and click on **Python Interpreter**. On the opened window, click the plus button and search for **Appium**.

- Within the opened window, click the plus button and search for Appium-Python-Client in the tab.
- Select Appium-Python-Client and install the latest, stable package to your project.
- Once installation is complete, a success message will appear. Close out of the tab and select OK. Ensure Appium has been installed successfully before continuing.

### **3. Open the terminal in the project, and include the following code.**

```
pip install selenium==3.141.0
```

- This installation is necessary before each project as current Selenium versions are unstable and are incompatible with Python. Downgrading to Selenium 3.141 is necessary for testing with Python.

### **4. Open main.py.**

### **5. At the top of the file, include the following imports.**

```
from appium import webdriver

from appium.webdriver.common.appiumby import AppiumBy
```

### **6. Add the desired capabilities for the Notepad Application.**

- Since we are using a legacy application (Notepad.exe) for testing, we must specify the full path of the executable file to WindowsDriver. Otherwise, for UWP applications, you must use the Application ID.
- Creating a dictionary to store the desired capabilities allows for easy modification and implementation.

```
desired_capabilities = {
    "platformName": "Windows",
```

```
"deviceName": "WindowsPC",  
}
```

## 7. Include the path of the application.

- This dictionary can be used to not only specify the path of an application, but also to specify all other capabilities (See chart on page 6).

```
"app": "r\"C:\\Windows\\System32\\Notepad.exe"
```

- The two parameters (“app” and the application’s path) give information to WinAppDriver about the project testing. Adding “r” before the path allows you to provide the raw string without implementing \n (newline) by accident in \Notepad.

**NOTE:** To determine the path of a file (as is necessary in this step):

- On Windows, navigate to the file’s location from your computer’s library, right-click the file once, then Copy as path. Alternatively, click inside the file path box to the left of the library’s search bar to see the file or folder’s path.
- On macOS/Linux, navigate to your file’s location through the Finder, click it, then copy it by pressing Command+C. With the file on your clipboard, pasting it in a textual context will give the file path.

## 8. Assign a WindowsDriver Object to the Notepad Session.

```
notepad_session = webdriver.Remote("http://127.0.0.1:4723", desired_capabilities)
```

- The first parameter will give the address of WinAppDriver’s server and the second parameter is the created object, desired\_capabilities.
- Including these parameters informs the server about the application WinAppDriver should launch.

**NOTE:** WinAppDriver operates on a specific IP address and port. By default, it is <http://127.0.0.1:4723>. However, you can check the IP address and port by opening WinAppDriver. The address will be provided when running the program.

## 9. Implement a try/except/finally block.

```
try:  
  
except WebDriverException as e:  
  
finally:
```

- Include the following import at the top of the file:

```
from selenium.common.exceptions import WebDriverException
```

## 10. Add handling for exceptions under the except block.

```
print("App not started.")  
  
print(e)
```

## 11. Add code to end the session under the finally block.

```
notepad_session.quit()
```

- This ensures that the session will be ended safely even if any exceptions are encountered.

## 12. Open WinAppDriver.

## 13. Include testing by utilizing the Windows Inspector Tool.

- Open the inspector tool from the desktop shortcut.
- View element information by using the watch cursor and hovering the mouse over an element within the Notepad application.

## 14. Incorporate more testing with Element Name and Automation ID in the try block.

```
notepad_session.find_element(AppiumBy.NAME, "Format").click()

notepad_session.find_element(AppiumBy.NAME, "Font...").click()

notepad_session.find_element(AppiumBy.ACCESSIBILITY_ID,
"1001").send_keys("DejaVu Sans")

notepad_session.find_element(AppiumBy.NAME, "Bold").click()

notepad_session.find_element(AppiumBy.NAME, "18").click()

notepad_session.find_element(AppiumBy.ACCESSIBILITY_ID, "1").click()
```

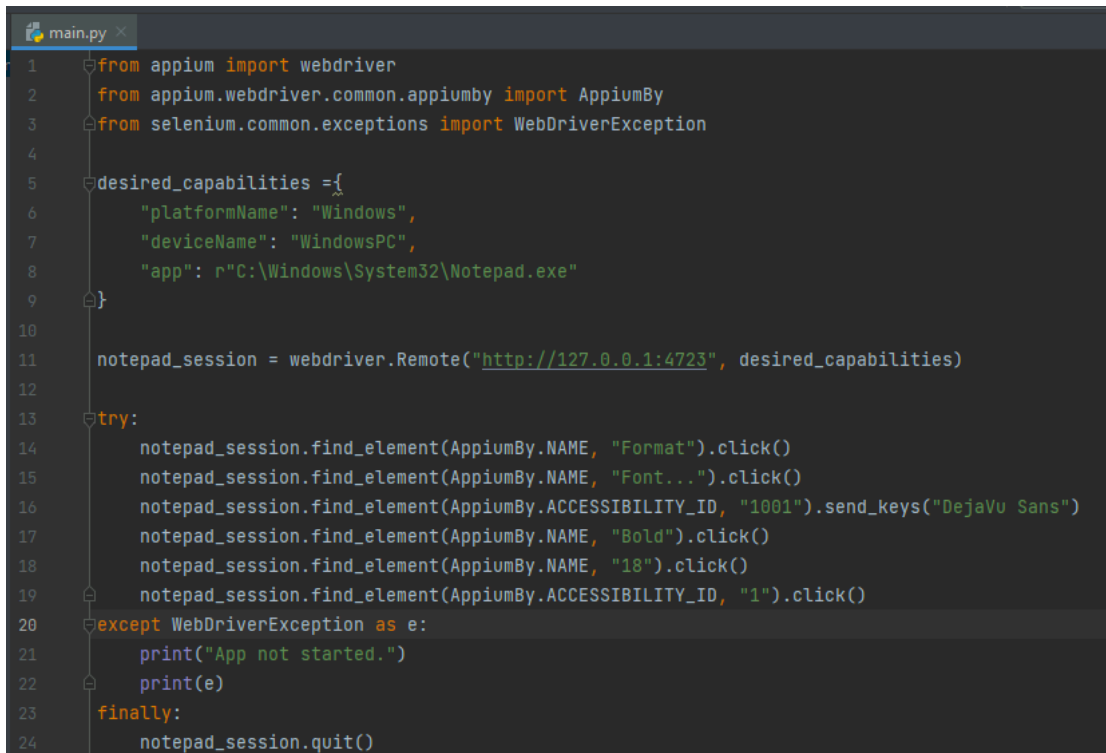
**NOTE:** Accessibility ID utilizes the Automation ID as a parameter to find the desired element. Some elements may not have assigned Automation IDs, so you must use the element's name.

**NOTE:** *.click()* will select the chosen element and *.send\_keys* allows for keyboard input into the chosen element.

## 15. Run the completed test.

- The notepad application should open, change font settings, and close.

**Target:** Confirm successful testing of the Notepad application!



```

1 from appium import webdriver
2 from appium.webdriver.common.appiumby import AppiumBy
3 from selenium.common.exceptions import WebDriverException
4
5 desired_capabilities = {
6     "platformName": "Windows",
7     "deviceName": "WindowsPC",
8     "app": r"C:\Windows\System32\notepad.exe"
9 }
10
11 notepad_session = webdriver.Remote("http://127.0.0.1:4723", desired_capabilities)
12
13 try:
14     notepad_session.find_element(AppiumBy.NAME, "Format").click()
15     notepad_session.find_element(AppiumBy.NAME, "Font...").click()
16     notepad_session.find_element(AppiumBy.ACCESSIBILITY_ID, "1001").send_keys("DejaVu Sans")
17     notepad_session.find_element(AppiumBy.NAME, "Bold").click()
18     notepad_session.find_element(AppiumBy.NAME, "18").click()
19     notepad_session.find_element(AppiumBy.ACCESSIBILITY_ID, "1").click()
20 except WebDriverException as e:
21     print("App not started.")
22     print(e)
23 finally:
24     notepad_session.quit()

```

The following exercise outlines the creation of an unittest test to automate Notepad:

## Exercise 2: Notepad unittest Automation Test:

### 1. Open PyCharm and create a new project.

- We will use the unittest framework to create the Python test.
  - unittest Test Projects offer more capabilities and speed when automated testing compared to standard projects. Uncheck the box to create a main.py welcome script.
- Name the file “AppiumTest2” and store the file in the Appium Python Training folder, creating a new folder for this project.

**2. In the menu bar, click on File, then Settings. Open the Project:**

**AppiumTest2 tab and click on Python Interpreter. On the opened window, click the plus button and search for Appium.**

- Within the opened window, click the plus button and search for Appium-Python-Client in the tab.
- Select Appium-Python-Client and install the latest, stable package to your project.
- Once installation is complete, a success message will appear. Close out of the tab and select OK. Ensure Appium has been installed successfully before continuing.

**3. Open the terminal in the project, and include the following code.**

```
pip install selenium==3.141.0
```

**4. In the project hierarchy, click on AppiumTest2 and select New, then Python file. Select Python Unit Test and name it test\_cases.**

- We will implement the unittest framework, which is different from standard projects. There will be setup and teardown methods. We will begin by implementing the framework.

**5. Include the following imports at the top of the file.**

```
import unittest  
  
from appium import webdriver  
  
from appium.webdriver.common.appiumby import AppiumBy
```

**6. Rename the class that inherits from the TestCase base class and remove the default test.**

```
class AllTestCases(unittest.TestCase):
```



## 7. Include variables to store the session information.

```
desired_cap = {  
    "platformName": "Windows",  
    "deviceName": "WindowsPC",  
    "app": r"C:\Windows\System32\notepad.exe"  
}  
  
link = "http://127.0.0.1:4723"
```

- We will now use the dictionary to specify capabilities on the application  
(See chart on page 6 for complete list of Windows capabilities)

## 8. Create the setUp() method, which will run before every test.

```
def setUp(self):
```

## 9. In the setUp method, create the session.

```
self.notepad_session = webdriver.Remote(self.link, self.desired_cap)
```

## 10. Open Notepad.

- Type a message and save the new file as AppiumText.txt. Store the file on desktop for future reference, and close the notepad application.

## 11. Add an additional capability to open your new text file. To find the text path, hold shift and right-click on AppiumText.txt, then click Copy Path.

```
"appArguments": r"C:\Users\Administrator\Desktop\AppiumText.txt"
```

## 12. Open WinAppDriver.

- WinAppDriver must be opened prior to testing as the server must be open to receive commands.

## 13. Within PyCharm, click View, then Tool Windows and select Run.

- This will open the test explorer, which you can move and pin on the left column.

**NOTE:** The Test Explorer is used to run tests in PyCharm. Each test will appear in the test column. Users can run individual tests by right-clicking on the test and selecting Run, or users can click the green run button at the top of the explorer.

#### 14. Create the first test.

```
def test_1(self):  
    pass
```

#### 15. Below the AllTestCases class, notice the following code will run the tests.

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

- This is used in every project using UnitTest. This will run all the available test cases.

#### 16. Run the test.

- Notepad should open AppiumText.txt, and the test should be successful in the test explorer.

#### 17. Include additional testing by utilizing the Windows Inspector Tool.

- Open the inspector tool from the desktop shortcut.
- View element information by using the watch cursor and hovering the mouse over an element within the Notepad application.

#### 18. Incorporate more testing with Class Name and Element Name.

```
self.notepad_session.find_element(AppiumBy.CLASS_NAME, "Edit").send_keys("This is  
some text.")  
  
self.notepad_session.find_element(AppiumBy.NAME, "Edit").click()
```

#### 19. Include a teardown method that will close the application and session.

```
def tearDown(self):
```

```
self.notepad_session.quit()
```

## 20. Run the test.

- Notepad should open your file, add text, and click the edit button, before ending successfully.

**Target:** Confirm successful UnitTest testing of the Notepad application!

```
class AllTestCases(unittest.TestCase):
    desired_cap = {
        "platformName": "Windows",
        "deviceName": "WindowsPC",
        "app": r"C:\Windows\System32\notepad.exe",
        "appArguments": r"C:\Users\Administrator\Desktop\AppiumText.txt"
    }
    link = "http://127.0.0.1:4723"

    def setUp(self): # runs before every test
        self.notepad_session = webdriver.Remote(self.link, self.desired_cap)

    def test_1(self):
        self.notepad_session.find_element(AppiumBy.CLASS_NAME, "Edit").send_keys("This is some text.")
        self.notepad_session.find_element(AppiumBy.NAME, "Edit").click()

    def tearDown(self): # runs after each test
        self.notepad_session.quit()

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

---

The following exercise outlines the creation of a PyTest test to automate Calculator:

### Exercise 3: Calculator PyTest Automation Test:

#### 1. Open PyCharm and create a new project.

- We will use the PyTest framework to create the Python test.
- Name the file “AppiumTest3” and store the file in the Appium Python Training folder, creating a new folder for this project.

#### 2. In the menu bar, click on File, then Settings. Open the Project:

**AppiumTest3 tab and click on Python Interpreter. On the opened window, click the plus button and search for Appium.**

- Within the opened window, click the plus button and search for Appium-Python-Client in the tab.
- Select Appium-Python-Client and install the latest, stable package to your project.
- Once installation is complete, a success message will appear. Close out of the tab and select OK. Ensure Appium has been installed successfully before continuing.

**3. Open the terminal in the project, and include the following code.**

```
pip install selenium==3.141.0
```

**4. Now, include the following code in the terminal to install PyTest.**

```
pip install -U pytest
```

**5. In the project hierarchy, right-click on AppiumTest3 and select New, then Python file. Select Python file and name it test\_cases.**

- Note that the file name and test methods must begin with “test\_” for PyTest to detect the tests.

**6. Include the following imports at the top of the file.**

```
from appium import webdriver

from appium.webdriver.common.appiumby import AppiumBy
```

**7. Create the class that will host our test methods.**

```
class TestCases:
```

**8. Add four regions to organize each segment of the class.**

```
# region Appium Session Info

# endregion

# region Setup

# endregion
```

```
# region Tests

# endregion

# region Teardown

# endregion
```

**9. In the first region, include variables to store the session information.**

```
desired_cap = {

    "platformName": "Windows",

    "deviceName": "WindowsPC",

    "app": r"C:\Windows\System32\win32calc.exe"

}

link = "http://127.0.0.1:4723"
```

**10. In the next region, create the setUp() method, which will run before every test.**

```
def setup_method(self):
```

- Note that this is the required syntax to create a setup method.

**11. In the setUp method, create the session.**

```
self.calculator_session = webdriver.Remote(self.link, self.desired_cap)
```

**12. In the test region, create the first test.**

```
def test_1(self):

    pass
```

**13. Open WinAppDriver.**

**14. To run the test, paste the following into the terminal.**

```
pytest
```

- The calculator application should open! Check the test explorer to ensure the test ran successfully.

**15. Within the test method, remove pass and include testing with Element**

## **Name and Automation Id.**

```
self.calculator_session.find_element(AppiumBy.NAME, "6").click()

self.calculator_session.find_element(AppiumBy.NAME, "3").click()

self.calculator_session.find_element(AppiumBy.ACCESSIBILITY_ID, "94").click()

self.calculator_session.find_element(AppiumBy.ACCESSIBILITY_ID, "135").click()

self.calculator_session.find_element(AppiumBy.ACCESSIBILITY_ID, "121").click()
```

- Open the Windows Inspector Tool from the desktop shortcut to find details on each element (name, accessibility ID, etc.)

## **16. Add a second test method.**

```
def test_2(self):
```

## **17. Add testing using Element Name.**

```
self.calculator_session.find_element(AppiumBy.NAME, "7").click()

self.calculator_session.find_element(AppiumBy.NAME, "Subtract").click()

self.calculator_session.find_element(AppiumBy.NAME, "3").click()

self.calculator_session.find_element(AppiumBy.NAME, "Equals").click()
```

## **18. Use the Appium text variable and an assertion to check the result of the calculation.**

```
result = self.calculator_session.find_element(AppiumBy.NAME, 'Result').text

assert int(result) == 4
```

- Using .text will return a string representation of the element data.

## **19. In the last region, include a teardown method to end the testing.**

```
def teardown_method(self):

    self.calculator_session.quit()
```

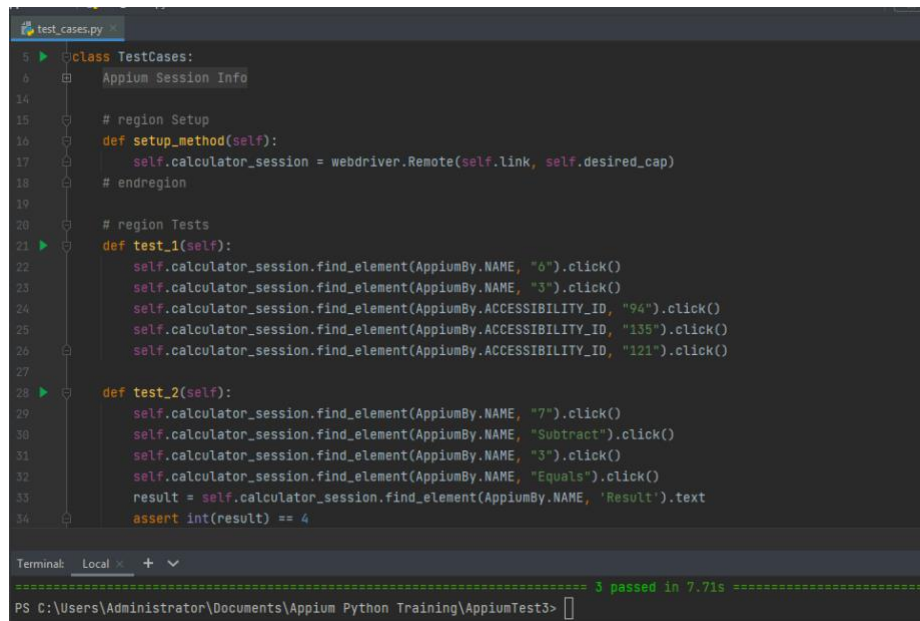
- Note that this is the required syntax to create a teardown method.

## **20. Run the tests.**

- Calculator should open, calculations should be solved, the application

should close, and both tests should pass.

**Target:** Confirm successful PyTest testing of the Calculator application!



```
test_cases.py
5 class TestCases:
6     Appium Session Info
14
15     # region Setup
16     def setup_method(self):
17         self.calculator_session = webdriver.Remote(self.Link, self.desired_cap)
18     # endregion
19
20     # region Tests
21     def test_1(self):
22         self.calculator_session.find_element(AppiumBy.NAME, "6").click()
23         self.calculator_session.find_element(AppiumBy.NAME, "3").click()
24         self.calculator_session.find_element(AppiumBy.ACCESSIBILITY_ID, "94").click()
25         self.calculator_session.find_element(AppiumBy.ACCESSIBILITY_ID, "135").click()
26         self.calculator_session.find_element(AppiumBy.ACCESSIBILITY_ID, "121").click()
27
28     def test_2(self):
29         self.calculator_session.find_element(AppiumBy.NAME, "7").click()
30         self.calculator_session.find_element(AppiumBy.NAME, "Subtract").click()
31         self.calculator_session.find_element(AppiumBy.NAME, "3").click()
32         self.calculator_session.find_element(AppiumBy.NAME, "Equals").click()
33         result = self.calculator_session.find_element(AppiumBy.NAME, 'Result').text
34         assert int(result) == 4

Terminal: Local + -
===== 3 passed in 7.71s =====
PS C:\Users\Administrator\Documents\Appium Python Training\AppiumTest3>
```

## Module 5 - Testing UI Recorder

### Key Points:

- UI Recorder is an extension tool for WinAppDriver that serves to record keyboard and mouse interactions with an application to locate UI element information.
- The recorder's interface displays two panels, each highlighting different UI element information:
  - The first panel displays the generated XPath Query of a selected UI element.
  - The second panel displays either the XML node attributes or generated C# code of a recorded action.
- UI recorder only provides the XPath of a selected element, opposed to Inspect.exe, which provides several IDs and tags.

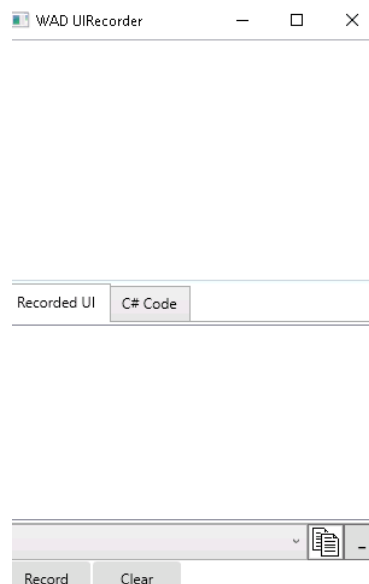
- However, XPath is useful both when navigating through the structure of an application as it displays each element's hierarchy and when Inspect.exe does not provide unique locator tags to an element.

The following exercise outlines the installation of UI Recorder:

### Exercise 1: Installation of UI Recorder:

1. **Download the latest release of WinAppDriver's UI Recorder, found on GitHub.**
  - <https://github.com/microsoft/WinAppDriver/releases/tag/UIR-v1.1>
2. **Extract the downloaded zip file.**
3. **Open WinAppDriverUiRecorder.exe.**
  - Optional: Add a shortcut to UI Recorder to allow for access from the desktop.

**Target:** Confirm successful installation of UI Recorder!





The following exercise outlines the use of UI Recorder:

## **Exercise 2: Use of UI Recorder to Locate Element Information:**

- 1. Open AppiumTest3 in the test projects folder.**
- 2. Open UI Recorder.**
- 3. Open Calculator.**
- 4. Click Record in the UI Recorder, then begin selecting buttons in the Calculator application.**
  - Notice that the two panels begin displaying the XPath and Recorded UI of your actions.
- 5. Create a new test method in AppiumTest3, where we will find an element by XPath.**

```
def test_3(self):  
    self.calculator_session.find_element(AppiumBy.NAME, "5").click()  
    self.calculator_session.find_element(AppiumBy.NAME, "1").click()
```

- 6. Include the following code to search by XPath.**

```
self.calculator_session.find_element(AppiumBy.XPATH, " ").click()
```

- *We will fill the empty spot with the XPath later on.*
- 7. In the UI Recorder, clear the session and start a new recording.**
  - 8. In Calculator, press the Clear button, labeled as C.**
  - 9. Pause the recording.**
    - The top panel should show the XPath for the Clear element in Calculator.

**NOTE:** When using UI Recorder, you must wait for the flashing rectangle to stop before pressing another element or pausing the recording. Otherwise, the UI Recorder will not properly track the elements.

#### 10. Right click on the XPath box and select Copy Full XPath.

```
/Pane[@ClassName=\"#32769\"][@Name=\"Desktop\"]/Window[@ClassName=\"CalcFrame\"][@Name=\"Calculator\"]/Pane[@ClassName=\"CalcFrame\"]/Button[@ClassName=\"Button\"][@Name=\"Clear\"]
```

#### 11. Within PyCharm, paste the XPath into the empty quotes.

**NOTE:** To use the XPath from UI Recorder, the complete XPath must be modified in order for WinAppDriver to read it. Escaped double quotes, \", must be replaced with single quotes. Additionally, you must change the starting point of the XPath.

#### 12. Remove the escaped double quotes and replace them with single quotes.

```
/Pane[@ClassName='#32769'][@Name='Desktop']/Window[@ClassName='CalcFrame'][@Name='Calculator']/Pane[@ClassName='CalcFrame']/Button[@ClassName='Button'][@Name='Clear']
```

#### 13. Change the starting point of the XPath to remove information about the device.

- This is done to not repeat information already specified in the original Desired Capabilities.

```
/Window[@ClassName='CalcFrame'][@Name='Calculator']/Pane[@ClassName='CalcFrame']/Button[@ClassName='Button'][@Name='Clear']
```

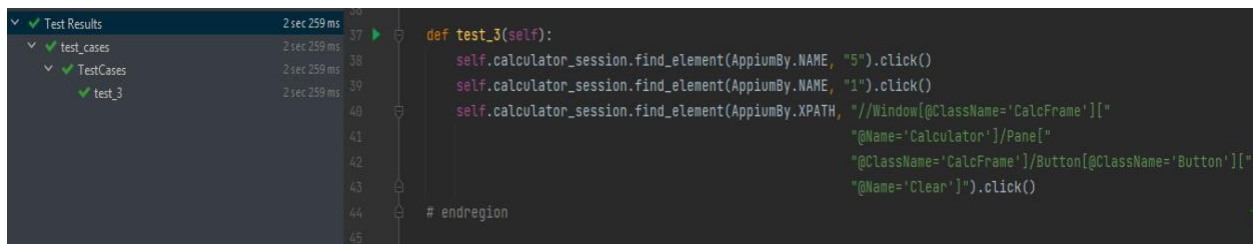
#### 14. Lastly, add an additional / before Window.

```
//Window[@ClassName='CalcFrame'][@Name='Calculator']/Pane[@ClassName='CalcFrame']/Button[@ClassName='Button'][@Name='Clear']
```

## 15. Run all tests.

- The Calculator app should open, perform a calculation, then clear the result before closing and ending the test.

**Target:** Confirm successful testing with UI Recorder!



The screenshot shows a test runner interface with a tree view on the left and a code editor on the right. The tree view shows a hierarchy of test results: 'Test Results' (2sec 259ms), 'test\_cases' (2sec 259ms), 'TestCases' (2sec 259ms), and 'test\_3' (2sec 259ms). The code editor shows a Python test method 'test\_3' with the following code:

```
def test_3(self):
    self.calculator_session.find_element(AppiumBy.NAME, "5").click()
    self.calculator_session.find_element(AppiumBy.NAME, "1").click()
    self.calculator_session.find_element(AppiumBy.XPATH, "//Window[@ClassName='CalcFrame']["
        "@Name='Calculator']/Pane["
        "@ClassName='CalcFrame']/Button[@ClassName='Button']["
        "@Name='Clear']").click()
# endregion
```

## Module 6 - Adding Assertions and Object Synchronization

### Key Points:

- Assertions:
  - Assertions are incorporated within test methods to be used as a verification point to mark that test script is passed or failed.
  - To verify a test state, assertions check various conditions such as object properties in the test.
  - When implemented into test methods, the user must state desired, expected conditions to compare to runtime conditions.
  - The Assert class is used to execute these verifications.
- Object Synchronization:
  - `WaitForControlExist()` // for specific object/properties
  - `Thread.Sleep(timeOut)` // generic wait time

- Playback.PlaybackSettings.WaitForReadyTimeout = #timeout // implicit wait for all steps
- The following code can be implemented into testing to check if a dialog/alert box exists:

```

alert_box = self.driver.switch_to.alert

if alert_box.is_displayed():

    alert.accept()

```

- This try/catch block switches to the alert window of the application and checks if the window is displayed. If the alert box is displayed, it will accept the alert before continuing. Otherwise, it will just continue.

The following exercise outlines the use of assertions:

### **Exercise 1: Assertions:**

- 1. Open a new test project in PyCharm and name it “Assertions.”**
- 2. In the project hierarchy, right-click on Assertions and select New, then Python file. Select Python Unit Test and name it test\_assertions.**
- 3. Create a new boolean variable set to True.**

```

first_option = True

```

- 4. Remove the default test and create the first test.**

```

def test_true(self):

```

- 5. Now assert that the variable is true.**

```

self.assertTrue(self.first_option, "The variable is set to {}".format(self.first_option))

```

- Note that it is optional to pass in a failure message. Including the second parameter will print out a message to the console if the test fails.

- 6. Run the test. The test should run successfully and pass.**

**7. Create a new boolean variable set to true.**

```
second_option = True
```

**8. Create a second test.**

```
def test_false(self):
```

**9. Now assert that the variable is false.**

```
self.assertFalse(self.second_option, "The variable is set to {}".format(self.second_option))
```

**10. Run the test. The test should fail as our variable, second\_option, is not false.**

**11. Fix this test by updating the boolean variable and run the test again.**

```
second_option = True
```

**12. Create a third test.**

```
def test_inequality(self):
```

**13. Create two strings, both set to different values.**

```
first_string = "value1"
```

```
second_string = "value2"
```

**14. Assert that these values are not equal to each other.**

```
self.assertNotEqual(first_string, second_string, "{} is equal to {}".format(first_string, second_string))
```

- Note that this method takes 3 possible parameters. The first two are compared, and the third is an optional failure message.

**15. Run the new test, and ensure that it passes.**

**16. Lastly, create a fourth test.**

```
def test_equality(self):
```

**17. Create two integer values, set to different numbers.**

```
a = 23
```

b = 7

### 18. Assert that the addition of these values equals 30.

```
self.assertEqual(30, a+b)
```

- Note that the first parameter is the expected result, while the second parameter is the actual result.

**Target:** Confirm successful use of assertions to verify tests!

```
class MyTestCase(unittest.TestCase):
    first_option = True

    def test_true(self):
        self.assertTrue(self.first_option, "The variable is set to {}".format(self.first_option))

    second_option = False

    def test_false(self):
        self.assertFalse(self.second_option, "The variable is set to {}".format(self.second_option))

    def test_inequality(self):
        first_string = "Joe"
        second_string = "Joseph"
        self.assertNotEqual(first_string, second_string, "{} is equal to {}".format(first_string, second_string))



    def test_equality(self):
        a = 23
        b = 7
        self.assertEqual(30, a+b)

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

---

## Module 7 - Options for Execution and Debugging


### Key Points:

- Execution:
  - The Test Explorer is used to run tests in PyCharm with the UnitTest framework. Each test will appear in the test column.
  - To run all tests, press the top left button in the test explorer: 
  - Users can display passed tests and ignored tests, or sort the tests by clicking the respective icon: 

- Users can also run individual tests by right-clicking the desired test and selecting Run in the code.

- It is also possible to view test history, import, and export tests using the following

buttons: 

- To create a Test Suite to filter tests, highlight the desired tests. Then, right-click on the tests and select *Add to Playlist* and *Create New Playlist*. The playlist will open in a new test explorer window, where you can run the playlist tests. You can save the playlist for later use by pressing the save button: 

- An additional way of running tests is clicking the green arrow next to the desired test, and selecting Run tests. To run all tests, select the green arrow next to the unittest.main() method call.

- To order PyTest tests, install the package and implement the order attribute tag above test definitions. The order value is used to determine the order to run the unit tests. Tests without an attribute will be run after the order has been executed.

```
pip install pytest-order
```

```
@pytest.mark.order(1)
```

- **NOTE:** Tests do not wait for prior tests to finish. If multiple threads are in use, a test may be started while some earlier tests are still being run.

- Debugging:

- Debugging is the process of detecting and removing existing and potential errors in a software code that can cause it to behave unexpectedly or crash.
- To debug within PyCharm, users must use the Debugging mode, which includes many features such as breakpoints, runtime information, and stepping into code.

- When running tests on applications with pop-up/alert windows, you include methods to dismiss the windows and continue with the test. These private methods should be included in the public method of every test that encounters a pop-up window.

The following exercise outlines the use of execution and debugging:

### **Exercise 1: Execution and Debugging:**

#### **1. Open AppiumTest2.**

#### **2. To debug, start your app with the debugger attached. To do so, press Run then Debug and select Python tests in test\_cases.py.**

#### **3. Add a breakpoint by entering debugging mode and clicking in the gray margin to the left of a line of code.**

- Before debugging you must add breakpoints, an intentional stopping place for the debugger. These allow users to examine code as it runs, stopping at chosen lines of code.

#### **4. Create a watch window to view object property information at runtime.**

- Users can view object property information at runtime using the debugging mode. After stopping at a breakpoint, view the Debug window. Then in the variable tab, select the eye symbol and uncheck Show Watches in the Variable tab. A new window will open where you can add variables or expressions to track their property information at each breakpoint.

#### **5. Step into the code.**

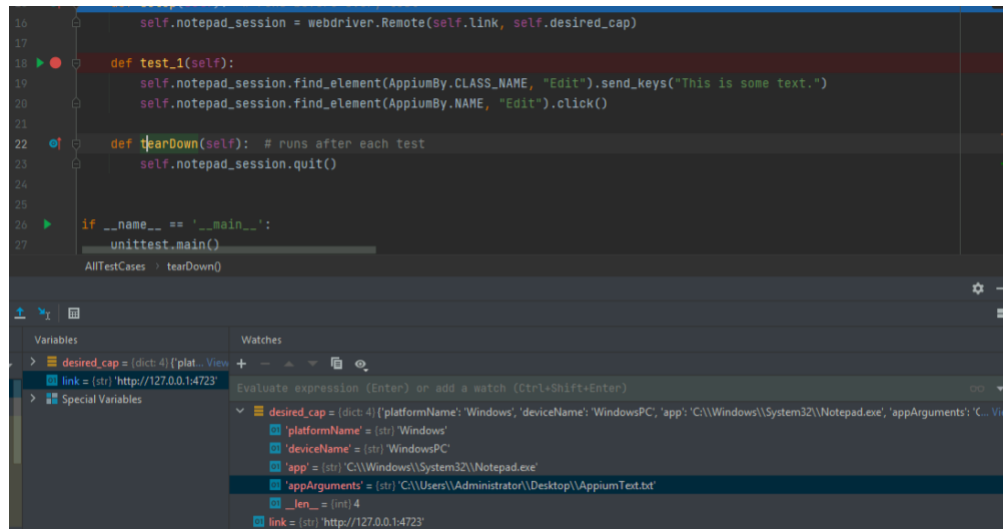
- When stopped at a breakpoint, Step Into in the debug window. You can now step into the code and step through each breakpoint to find bugs and



errors. In the toolbar, there are three buttons, displayed as blue arrows, to step into, over, and out of the code.

## 6. End debugging mode.

**Target:** Confirm successful execution and debugging of an Appium test!



```
16 self.notepad_session = webdriver.Remote(self.link, self.desired_cap)
17
18 def test_1(self):
19     self.notepad_session.find_element(AppiumBy.CLASS_NAME, "Edit").send_keys("This is some text.")
20     self.notepad_session.find_element(AppiumBy.NAME, "Edit").click()
21
22 def tearDown(self): # runs after each test
23     self.notepad_session.quit()
24
25
26 if __name__ == '__main__':
27     unittest.main()
AllTestCases tearDown()
```

**Variables**

- `desired_cap` = {dict: 4} [platformName, deviceName, app, appArguments]
- `link` = (str) 'http://127.0.0.1:4723'
- Special Variables**

**Watches**

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

- `desired_cap` = {dict: 4} [platformName, deviceName, app, appArguments]
- `platformName` = (str) 'Windows'
- `deviceName` = (str) 'WindowsPC'
- `app` = (str) 'C:\\Windows\\System32\\Notepad.exe'
- `appArguments` = (str) 'C:\\Users\\Administrator\\Desktop\\AppiumText.txt'
- `len` = (int) 4
- `link` = (str) 'http://127.0.0.1:4723'

## Module 8 - Creating a WPF Application

### Key Points:

- Windows Presentation Foundation (WPF) is a UI development framework that allows users to create Windows desktop applications
- WPF applications are built through two languages (XAML and C#)
  - XAML markup serves as the front-end language to create UI elements and alter the appearance of the application
  - C# serves as WPF's back-end language to provide functionality such as actions, events, and behavior

The following exercise outlines the creation of the WPF application's first layout:

## Exercise 1: First Window Layout:

### 1. Open Visual Studio Community and create a new project.

- Use the WPF (.NET Framework) template for this project.
- Name the project "LoginWPFApp" and store the file in  
C:\Users\Administrator\Documents\Appium Python Training\.

### 2. Navigate to MainWindow.xaml and edit the main window for the application.

- Change the height to 300 and the width to 400.
- Include the following code after the window title (line 8) to adjust the startup location, window style, and resize mode.

```
Height="300" Width="400" WindowStartupLocation="CenterScreen"
WindowStyle="None" ResizeMode="NoResize"
```

- These adjustments should be placed before the >, which closes the window element adjustments.

### 3. Add a background for the application.

- Within the <Grid> element, add a background to the user interface with the following code:

```
<Grid.Background>
    <LinearGradientBrush StartPoint="0.1, 0" EndPoint="0.9,1">
        <GradientStop Color="DodgerBlue" Offset="1" />
        <GradientStop Color="BlueViolet" Offset="0" />
    </LinearGradientBrush>
</Grid.Background>
<Border Height="260" VerticalAlignment="Top" CornerRadius="0 0 0 0"
Background="LightGray">
```

</Border>

**NOTE:** The XAML markup language functions through elements, which can be placed within each other. Each element begins and ends with a tag: i.e. <Button> </Button>. Ensure that each element is properly indented when placed within another element. All of the above code should be indented and placed within the <Grid> </Grid> element so that each tag surrounds the enclosed elements.

**4. Create three stack panels (A, B, C) to form the layout for the login application in the grid element.**

- Set the orientation property of the first general stack panel (A) to *Horizontal*, the width of the first enclosed stack panel (B) to 300, and the width of the second enclosed panel (C) to 100.

```
<StackPanel Orientation="Horizontal">  
    <StackPanel Width="300">  
    </StackPanel>  
    <StackPanel Width="100">  
    </StackPanel>  
</StackPanel>
```

- Stack panels are layout panels that arrange internal elements horizontally or vertically.

**5. Within stack panel B, create another stack panel that defines set margins.**

```
<StackPanel Margin="20 40">  
</StackPanel>
```

**6. Create a textblock within the stack panel created in Step 5 to title the application.**

```
<TextBlock Text="User Login" Margin="20" Foreground="White" FontSize="38"  
TextAlignment="Right">
```

```
</TextBlock>
```

- A title should appear at the top of the application within the design window above the XAML code.

**7. Underneath the created textblock, add another stack panel for the username input and label.**

```
<StackPanel Orientation="Horizontal" Margin="10">  
</StackPanel>
```

**8. Include a textblock in this stack panel to label the username input box.**

```
<TextBlock Text="Username:" Foreground="White" >  
</TextBlock>
```

**9. Next, create the textbox for users to input their username.**

```
<TextBox AutomationProperties.AutomationId="1" x:Name="userTxt"  
BorderBrush="White" CaretBrush="BlueViolet" SelectionBrush="DodgerBlue" Margin="10  
0" Width="250" Background="White" Foreground="Black">  
</TextBox>
```

- An input box and label for the username should be visible on the application in the design window.

**10. After the stack panel created in Step 7, create another stack panel for the password input and label.**

```
<StackPanel Orientation="Horizontal" Margin="10">  
</StackPanel>
```

**11. Implement a textblock in this panel to label the password input box.**

```
<TextBlock Text="Password:" Foreground="White" Width="55">  
</TextBlock>
```

**12. Underneath the textblock, add a protected password box to enter and handle user passwords.**

```

<PasswordBox AutomationProperties.AutomationId="2" PasswordChar="•"
x:Name="passwordTxt" BorderBrush="White" CaretBrush="BlueViolet"
SelectionBrush="DodgerBlue" Margin="10, 0" Width="250" Background="White"
Foreground="Black">
</PasswordBox>

```

- There should now be a second input box, directly below the username field.

**13. Underneath this stack panel, include another stack panel to include the application's first buttons.**

```

<StackPanel Margin="15,10,-100,10" Orientation="Horizontal"
HorizontalAlignment="Center" Width="335">
</StackPanel>

```

**14. In this panel, add the first button for users to login, which we will provide functionality to later in the MainWindow.cs file.**

```

<Button AutomationProperties.AutomationId="3" Width="90" Height="30" Margin="50 0 0
0" Background="DodgerBlue" Foreground="White" ToolTip="Login"
x:Name="loginButton" Click="loginButton_Click" Content="Login">
</Button>

```

**15. Add an additional button for users to clear all fields.**

```

<Button AutomationProperties.AutomationId="4" Content="Clear Fields"
Foreground="White" Background="DodgerBlue" Margin="20 0 0 0" Width="90"
Height="30" x:Name="clearbutton" Click="clearButton_Click" ToolTip="Clear Fields">
</Button>

```

**16. Within stack panel C, include a button for users to close out of the application.**

```

<Button AutomationProperties.AutomationId="5" Content="Close" Foreground="White"

```

```
Margin="10 20" Background="DodgerBlue" ToolTip="Close" x:Name="closeButton"

Click="closeButton_Click" Width="50" Height="20">

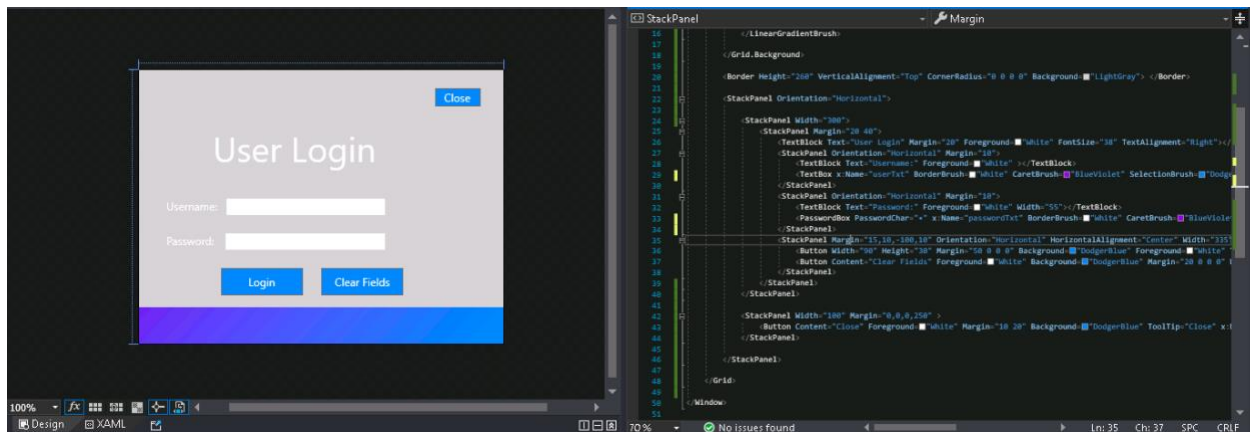
</Button>
```

- The design window should now show the completed layout of the application's first window.

## 17. Start the program.

- You should encounter build errors as we have not provided code to each button to enable their behavior.

**Target:** Confirm successful layout of the application's first window!



The following exercise outlines the creation of the WPF application's second layout:

## Exercise 2: Second Window Layout:

1. In the Solution Explorer window, right click on LoginWPFApp and click add. Then, click New Item and add a Window (WPF).
2. Name the new file LoginWindow.xaml.
3. Navigate to LoginWindow.xaml, then begin designing the layout by specifying properties of the new page.

- Change the height to 250 and the width to 400.
- Include the following code after the window title (line 8) to adjust the startup location, window style, and resize mode.

```
Height="250" Width="400" WindowStartupLocation="CenterScreen"
WindowStyle="None" ResizeMode="NoResize"
```

- These adjustments should be placed before the `>`, which closes the window element adjustments.

#### 4. Add a background to the page's layout within the `<Grid>` `</Grid>` element.

```
<Grid.Background>
    <LinearGradientBrush StartPoint="0.1, 0" EndPoint="0.9,1">
        <GradientStop Color="DodgerBlue" Offset="1" />
        <GradientStop Color="BlueViolet" Offset="0" />
    </LinearGradientBrush>
</Grid.Background>
<Border Height="210" VerticalAlignment="Top" CornerRadius="0 0 0 0"
Background="LightGray">
</Border>
```

#### 5. Create a stack panel in the `<Grid>` element, where we will put a textblock and two buttons.

- Set the vertical alignment to *Top* and the orientation to *Vertical*.

```
<StackPanel VerticalAlignment="Top" Orientation="Vertical">
</StackPanel>
```

#### 6. Implement a textblock in this panel to welcome users into the application.

```
<TextBlock Margin="10" TextAlignment="Center" FontSize="30" Text="Welcome to your
account." Foreground="White">
</TextBlock>
```

7. Add a button for users to return back to the login screen. We will use the code-behind feature of WPF to add functionality later.

```
<Button AutomationProperties.AutomationId="6" Foreground="White"
Background="DodgerBlue" Margin="30" Width="90" Height="30" Content="Back to
Login" x:Name="BackButton" Click="BackButton_Click" >

</Button>
```

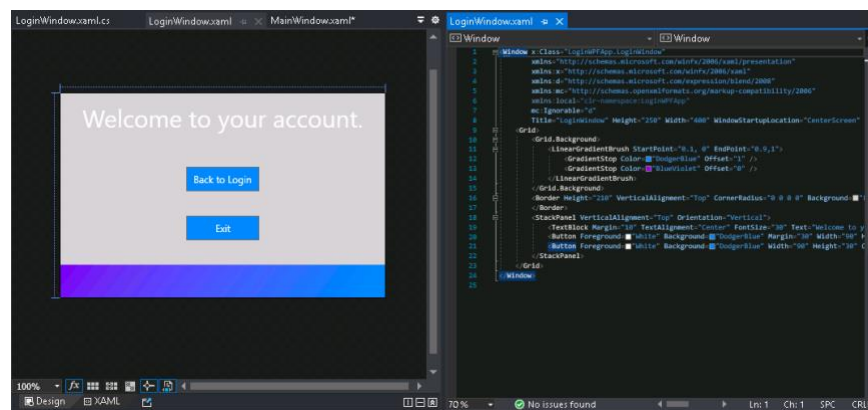
8. Include a second button for users to exit out of the application.

```
<Button AutomationProperties.AutomationId="7" Foreground="White"
Background="DodgerBlue" Width="90" Height="30" Content="Exit" x:Name="ExitButton"
Click="ExitButton_Click">

</Button>
```

- The window design should now be complete with a background, title, and two labeled buttons.

**Target:** Confirm successful layout of the application's second window!



The following exercise outlines the creation of the WPF application's behavior code:

### Exercise 3: Back-End Code of the WPF Application:

1. Navigate to MainWindow.xaml, and click on the Login button.



- Double-click on the Login button.
- MainWindow.xaml.cs should open and create a new event-handling method with the login button.

## **2. Add an if-statement to allow users in when they provide the correct information.**

- After adding the code, the method should look like the following:

```
private void loginButton_Click(object sender, RoutedEventArgs e)
{
    if (userTxt.Text == "value" && passwordTxt.Password == "value")
    {
        new LoginWindow().Show();
        this.Hide();
    }
}
```

- Replace “value” with your own credentials for each parameter.

## **3. Include an else-statement to create an error box in case the user provides the wrong information.**

- Add this code within the same method:

```
else
{
    MessageBox.Show("The username or password is incorrect.", "Login
Failed", MessageBoxButton.OK, MessageBoxImage.Error);
    userTxt.Clear();
    passwordTxt.Clear();
    userTxt.Focus();
}
}
```

**4. Navigate back to MainWindow.xaml and press the Clear Fields button.**

- Double-click on the Clear Fields button.
- MainWindow.xaml.cs should open and create a new event-handling method with the Clear Fields button.

**5. Within the new method, use Clear() and Focus() to remove text from the input boxes and direct the user back to the username input box.**

- The clearButton method should look like the following :

```
private void clearButton_Click(object sender, RoutedEventArgs e)
{
    userTxt.Clear();
    passwordTxt.Clear();
    userTxt.Focus();
}
```

**6. In the MainWindow.xaml file, locate the code for the Close button.**

- Double-click on the Close button to open a new event-handling method for this button.

**7. Add the following code in the new method to provide functionality to the close button and close the application and program.**

```
private void closeButton_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}
```

**Target:** Confirm successful functionality of the application's first window!

```
26 }
27
28 //reference
29 private void loginButton_Click(object sender, RoutedEventArgs e)
30 {
31     if (userTxt.Text == "Ryan" && passwordTxt.Password == "Tobin")
32     {
33         new LoginWindow().Show();
34         this.Hide();
35     }
36     else
37     {
38         MessageBox.Show("The username or password is incorrect.", "Login Failed", MessageBoxButton.OK, MessageBoxImage.Error);
39         userTxt.Clear();
40         userTxt.Focus();
41         passwordTxt.Clear();
42     }
43 }
44
45
46 //reference
47 private void closeButton_Click(object sender, RoutedEventArgs e)
48 {
49     Application.Current.Shutdown();
50 }
51
52 //reference
53 private void clearButton_Click(object sender, RoutedEventArgs e)
54 {
55     userTxt.Clear();
56     passwordTxt.Clear();
57     userTxt.Focus();
58 }
59
60 }
61
```

5 % | No issues found

The following exercise outlines the completion of the WPF application's behavior code:

#### Exercise 4: Back-End Code of the WPF Application:

1. **Navigate to LoginWindow.xaml, and click on the Back to Login button.**
  - o Double-click on the Back to login button.
  - o A new method should appear in LoginWindow.xaml.cs.
2. **Add the following code to return back to the starting page when the button is pressed.**

```
private void BackButton_Click(object sender, RoutedEventArgs e)
{
    this.Hide();
    new MainWindow().Show();
}
```

3. **Return back to the XAML file, and click on the Exit button.**

- o Double click on the Exit button.

- Then, navigate to the new event-handling method.

#### 4. Use Shutdown() to close the application and end the program when this button is pressed.

```
private void ExitButton_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}
```

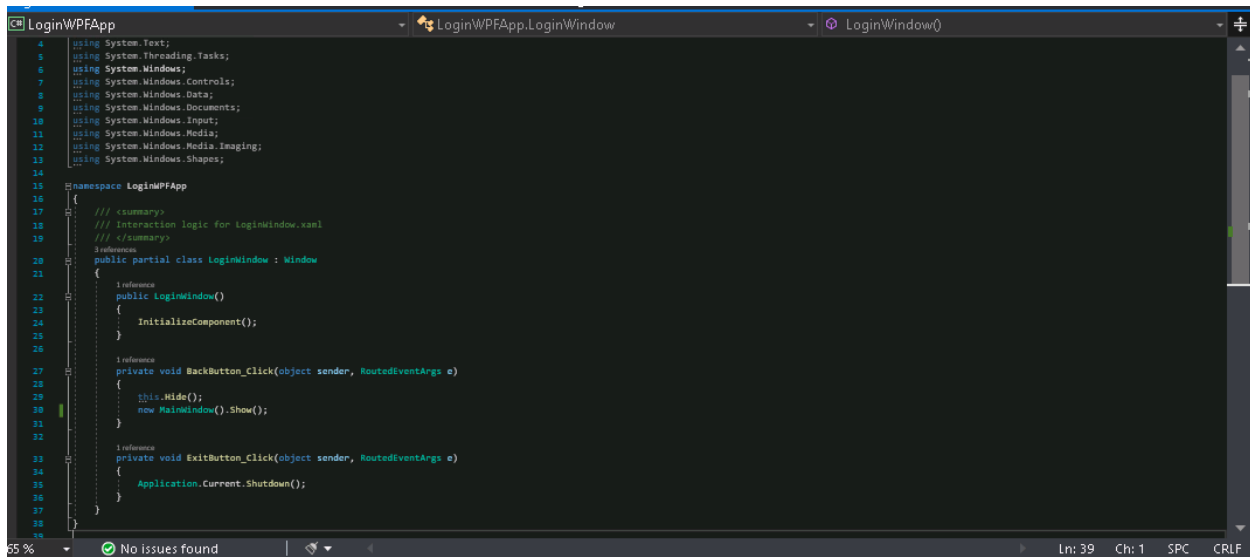
#### 5. Run the application.

- All functions should now be working properly. Test the functionality of the both page's buttons and textboxes.

#### 6. In the toolbar, click Build, then Build LoginWPFApp.

- Navigate to the project's location in the File Explorer.
- The .exe file will be located in LoginWPFApp/bin/debug for future reference.

**Target:** Confirm successful functionality of the WPF Login application!



## Module 9 - Test Case Template Leveraging PyTest

### Key Points:

- To create the main testing framework that will be used for the entirety of the project.
  - This will be the basis for the test framework deliverable.
- To create summary comment blocks and regions to organize testing projects.
- To define namespaces and class names.
- To save the project as a testing template within PyCharm for further use.

The following exercise outlines the creation of a PyCharm Test Case Template:

### Exercise 1: Test Case Template:

#### 1. Open PyCharm and create a new project.

- We will use the PyTest framework to create the Python test.
- Name the project *TestingTemplate* and save to the Appium Python Training folder, creating a new folder for this project.

#### 2. In the project hierarchy, create two new directories.

- Name the first directory “Tests” and create a Python file, named `test_cases.py`, into this directory.
- Name the second directory “Pages” and add a new Python file, named `first_page.py`, and move this to the Pages directory.

#### 3. In `FirstPage.py`, make a class and name it “`FirstPage`.”

```
class FirstPage:
```

#### 4. In this class, create two regions.

- Name the first region “Constructor,” the second “Objects,” and the third “Methods.”

```
# region Objects
```

```
# endregion
```

```
# region Methods
```

```
# endregion
```

## 5. Navigate to TestCases.py.

## 6. Add the following code to reference the previous class page.

```
from Pages.FirstPage import FirstPage
```

## 7. Create a class named TestCases.

```
class TestCases:
```

## 8. In the terminal, install pytest.

```
pip install pytest
```

## 9. Create a test region.

```
# region Tests
```

```
# endregion
```

## 10. Create a new default test.

```
def test_1(self):
```

```
    assert True
```

## 11. Create a Setup region after the Test method.

```
# region Setup
```

```
# endregion
```

## 12. Create a setup\_class method that will run once before the first test.

```
@classmethod
```

```
def setup_class(cls):
```

```
pass
```

**13. Create the setup\_method, which will run before every test.**

```
def setup_method(self):
```

```
pass
```

**14. Create a Teardown region after the Setup region.**

```
# region Teardown
```

```
# endregion
```

**15. Create a setup\_class method that will run once before the first test.**

```
@classmethod
```

```
def teardown_class(cls):
```

```
pass
```

**16. Create the setup\_method, which will run before every test.**

```
def teardown_method(self):
```

```
pass
```

**17. Save the project.**

**18. In the toolbar, click File then Save File as Template.**

- Name the file TestCases and click Ok.

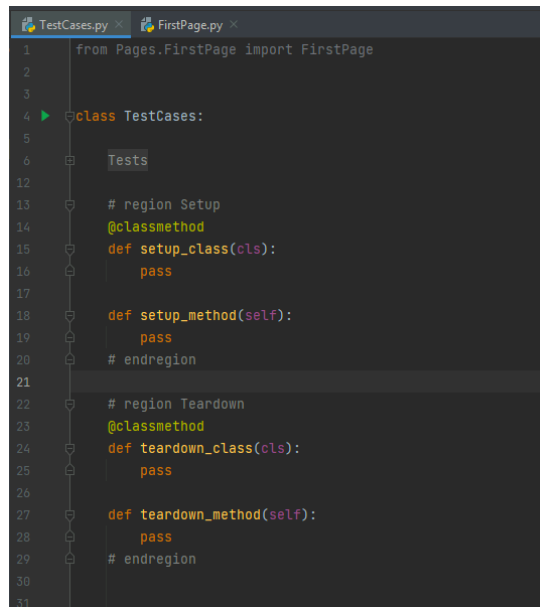
**19. Open FirstPage.py, and Click File then Save File as Template.**

- Name the file FirstPage and click Ok.

**20. Relaunch PyCharm, and create a new project.**

- To utilize the created templates, open the project and in the project hierarchy, right click to create a new file. Select either FirstPage or TestCases to add it to your project.

**Target:** Confirm successful creation of Appium Test Templates!



```
1 from Pages.FirstPage import FirstPage
2
3
4 class TestCases:
5
6     Tests
7
8
9
10
11
12
13     # region Setup
14     @classmethod
15     def setup_class(cls):
16         pass
17
18     def setup_method(self):
19         pass
20     # endregion
21
22     # region Teardown
23     @classmethod
24     def teardown_class(cls):
25         pass
26
27     def teardown_method(self):
28         pass
29     # endregion
30
31
```

---

## Module 10 - Page Object Model (POM)

### Key Points:

- Page Object Model (POM) tests build upon the standard functionality of normal Appium tests.
- POM allows for the separation and simplification of testing automation by putting object elements in a separate class file than the test commands.
  - The object repository is independent of each test, allowing for the reuse of each object without redefining element information.
  - This allows users to create multiple tests, each with different purposes, using the same object repository.
- POM templates allow for easier implementation of Appium tests to run several unique tests on the same application.

The following exercise outlines the creation of an Object/Method Repository:



## **Exercise 1: Object/Method Repository:**

### **1. Open PyCharm and create a new project and implement the created templates.**

- Name the project “PageObjectModel” and store the file in the Appium project folder, creating an individual folder for this project.
- Create two directories, “Tests” and “Pages.” Create two new files using both templates, named test\_cases and login\_page. Move test\_cases into the Tests directory and login\_page into the Pages directory.

### **2. In the menu bar, click on File, then Settings. Open the Project:**

**PageObjectModel tab and click on Python Interpreter. On the opened window, click the plus button and search for Appium-Python-Client.**

- Once installation is complete, a success message will appear. Close out of the tab and select OK. Ensure Appium has been installed successfully before continuing.

### **3. Open the terminal in the project, and include the following code.**

```
pip install selenium==3.141.0  
pip install pytest
```

### **4. In the project hierarchy, add a new Python file to Tests called \_\_init\_\_.py**

- Note that this is required for PyTest to locate individual tests when using the PageObjectModel. The file may be blank, but must be located in the same directory as the test\_cases file.

### **5. In the new login\_page class file, rename the class to “LoginWindow.”**

```
class LoginWindow:
```

### **6. Add the following import.**

```
from appium.webdriver.common.appiumby import AppiumBy
```

- 7. In the Objects region, create an instance variable and constructor to initialize the session driver.**

```
def __init__(self, driver):  
    self.driver = driver
```

- 8. Then, begin defining new Windows Elements for each element on the login window of the application.**

```
self.username_field = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "1")  
self.password_field = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "2")  
self.login_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "3")  
self.clear_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "4")  
self.close_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "5")
```

**NOTE:** The automation IDs provided in the prior code are the IDs that were assigned when creating the WPF application. Users can check each element's ID using the Inspect.exe tool. Additionally, users can change and assign IDs in the XAML markup. The following code defines the automation ID "newButton" for a button:

```
<Button AutomationProperties.AutomationId="newButton" </Button>
```

- 9. In the methods region, create a new method to login to the application.**

```
def login_to_application(self):
```

- 10. Add the following code to automate testing to login to the application.**

```
self.username_field.send_keys("Joe")  
self.password_field.send_keys("Johnson")  
self.login_button.click()
```

- Replace "value" with your own credentials for each parameter.

- 11. Add a second file named error\_page in the Pages directory, using the page template.**

**12. Create a new class, named ErrorWindow.**

```
class ErrorWindow:
```

**13. Add the following import.**

```
from appium.webdriver.common.appiumby import AppiumBy
```

**14. In the Objects region, create an instance variable and constructor to initialize the session driver.**

```
def __init__(self, driver):  
    self.driver = driver
```

**15. In the same constructor, define the OK element on the error box.**

```
self.okButton = self.driver.find_element(AppiumBy.NAME, "OK")
```

**16. In the methods region, create a method to close the error box.**

```
def clear_error(self):  
    self.okButton.click()
```

**Target:** Confirm successful creation of an Object/Method Repository!

```
from appium.webdriver.common.appiumby import AppiumBy  
  
class LoginWindow:  
    # region Objects  
    def __init__(self, driver):  
        self.driver = driver  
        self.userNameField = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "1")  
        self.passwordField = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "2")  
        self.loginButton = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "3")  
        self.clearButton = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "4")  
        self.closeButton = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "5")  
    # endregion  
    # region Methods  
    def login_to_application(self):  
        self.userNameField.send_keys("Joe")  
        self.passwordField.send_keys("Johnson")  
        self.loginButton.click()  
    # endregion
```

---

The following exercise outlines the completion of a Page Object Model:

## Exercise 2: Page Object Model:

### 1. Navigate to test\_cases.py.

### 2. Add the following imports:

```
from appium import webdriver

from Pages.login_page import LoginWindow

from Pages.error_page import ErrorWindow
```

### 3. Create two Appium sessions and set them to None.

```
app_settings_session = None

appium_session = None
```

- Creating two Appium sessions allows us to deal with error dialog boxes and pop-up windows.

### 4. Within the setup\_class() method, add the following code to open

#### WinAppDriver when the test is run.

```
os.startfile(r"C:\Program Files (x86)\Windows Application Driver\WinAppDriver.exe")
```

- Add the following import to the file:

```
import os
```

### 5. Create variables to store the application and session information.

```
app_path = r"C:\Users\Administrator\Documents\AppData Python
Training\LoginWPFApp\bin\debug\LoginWPFApp.exe"

session_link = "http://127.0.0.1:4723"

desired_cap = {
    'platformName': "Windows",
    'deviceName': "WindowsPC",
    'app': app_path
}

desired_cap_root = {
```

```

        'app': "Root"
    }

```

## 6. Add code to start the Appium sessions.

```

cls.appium_session = webdriver.Remote(session_link, desired_cap)
cls.app_settings_session = webdriver.Remote(session_link, desired_cap_root)
cls.appium_session.close_app()

```

- The Root reference enables Appium to automate testing and view the entire desktop for UI elements.

## 7. In `setup_method()`, add code to launch the application in between each test.

```

self.appium_session.launch_app()

```

## 8. In the `teardown class method`, add code to close the Appium session.

```

cls.app_settings_session.close_app()

```

## 9. Next, add the following to close all instances of the WPF application after the last test.

```

app_list = psutil.pids()
for i in range(0, len(app_list)):
    try:
        p = psutil.Process(app_list[i])
        if p.cmdline()[0].find(r"C:\Users\Administrator\Documents\AppData Python "
                                r"Training\LoginWPFApp\bin\debug\LoginWPFApp.exe") != -1:
            p.kill()
        elif p.cmdline()[0].find(r"C:\Program Files (x86)\Windows Application
                                Driver\WinAppDriver.exe") != -1:
            p.kill()
    except:
        pass

```

- Install and import the package psutil to utilize this code:

```
import psutil
```

- This code will iterate through opened applications and close the application if it is an instance of the WPF app or WinAppDriver.

**10. In `teardown_method()`, add the following code to close the application in between each test.**

```
self.app_settings_session.close_app()
```

**11. In the Tests section, rename the method to `test_login()`.**

```
def test_login(self):
```

**12. Remove `assert True`.**

**13. Add the following code to create an instance of the `LoginWindow` class and call the `LoginWindow` method.**

```
self.login_page = LoginWindow(self.app_settings_session)
```

```
self.login_page.login_to_application()
```

**14. In the terminal, type `pytest` to run the tests.**

```
pytest
```

- The application should open, and the test should run successfully!

**Target:** Confirm successful creation of a Page Object Model to test an application!

```
@classmethod
def setup_class(cls):
    os.startfile(r"C:\Program Files (x86)\Windows Application Driver\WinAppDriver.exe")
    app_path = r"C:\Users\Administrator\Documents\Apium Python Training\LoginWPFApp\bin\debug\LoginWPFApp.exe"
    session_link = "http://127.0.0.1:4723"
    desired_cap = {
        'platformName': "Windows",
        'deviceName': "WindowsPC",
        'app': app_path
    }
    desired_cap_root = {
        'app': "Root"
    }
    cls.appium_session = webdriver.Remote(session_link, desired_cap)
    cls.app_settings_session = webdriver.Remote(session_link, desired_cap_root)
    cls.appium_session.close_app()
```

---

The following exercise outlines the creation of a second Object/Method Repository:

### **Exercise 3: Object/Method Repository:**

- 1. In the Pages directory, make a third file, using the page template. Name it `welcome_page.py`.**
- 2. Create a new class, named `WelcomeWindow`.**
- 3. Add the following import.**

```
from appium.webdriver.common.appiumby import AppiumBy
```

- 4. In the Objects region, create an instance variable and constructor to initialize the session driver.**

```
def __init__(self, driver):  
    self.driver = driver
```

- 5. Then, begin defining new Windows Elements for each element on the welcome window of the application.**

```
self.return_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "6")  
self.close_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "7")
```

- 6. In the methods region, create a new method to return to login.**

```
def back_to_login(self):
```

- 7. Add the following code to automate testing to login to the application.**

```
self.return_button.click()
```

- 8. Create a second method to exit the application.**

```
def exit_application(self):
```

- 9. Add the following code to click the exit button:**

```
self.close_button.click()
```

**Target:** Confirm successful creation of a second Object/Method Repository!

```

class WelcomeWindow:

    # region Objects
    def __init__(self, driver):
        self.driver = driver
        self.return_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "backButton")
        self.close_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "exitButton")
    # endregion

    # region Methods
    def back_to_login(self):
        self.return_button.click()

    def exit_application(self):
        self.close_button.click()
    # endregion

```

---

The following exercise outlines the completion of a Page Object Model:

#### Exercise 4: Page Object Model:

1. Navigate to `test_cases.py`.

2. Add the following import.

```
from Pages.welcome_page import WelcomeWindow
```

3. In the Tests section, create a new test.

```
def test_return(self):
```

4. Add an attribute tag above the test to order both tests. Import `pytest` to utilize this tag.

```
import pytest
```

```
@pytest.mark.order(2)
```

5. Add the following code to reach the welcome page of the application.

```
self.login_page = LoginWindow(self.app_settings_session)
```

```
self.login_page.login_to_application()
```



**NOTE:** When adding tests that do not start on the first page, you must call methods to reach the desired page. Doing so allows each test to run independently of one another.

**6. Pass through the Appium session into an instance of the welcome page class in the method.**

```
self.welcome_page = WelcomeWindow(self.app_settings_session)
```

**7. Add the following code to call the BacktoLogin method.**

```
self.welcome_page.back_to_login()
```

**8. Create a new test to exit the application.**

```
def test_exit(self):
```

**9. Order the new test.**

```
@pytest.mark.order(3)
```

**10. Include code to reach the desired window of the application.**

```
self.login_page = LoginWindow(self.app_settings_session)
```

```
self.login_page.login_to_application()
```

**11. Add the following code to call the ExitApplication method.**

```
self.welcome_page = WelcomeWindow(self.app_settings_session)
```

```
self.welcome_page.exit_application()
```

**12. In the terminal, Run All Tests by typing pytest.**

```
pytest
```

- The application should open, and the tests should run successfully!

**Target:** Confirm successful creation of a Page Object Model to test an application!

```

# region Tests

@pytest.mark.order(1)
def test_login(self):
    self.login_page = LoginWindow(self.app_settings_session)
    self.login_page.login_to_application()

@pytest.mark.order(2)
def test_return(self):
    self.login_page = LoginWindow(self.app_settings_session)
    self.login_page.login_to_application()
    self.welcome_page = WelcomeWindow(self.app_settings_session)
    self.welcome_page.back_to_login()

@pytest.mark.order(3)
def test_exit(self):
    self.login_page = LoginWindow(self.app_settings_session)
    self.login_page.login_to_application()
    self.welcome_page = WelcomeWindow(self.app_settings_session)
    self.welcome_page.exit_application()

# endregion

```

---

## Module 11 - Adding Data Inputs for the POM

### Key Points:

- Data inputs can be added to pass through data from the test to the class structure, allowing for increased management of tests from the test structure and page.
- To add data inputs, we will add attribute tags before PyTest tests.

The following exercise outlines the addition of data inputs to a Page Object Model:

### Exercise 1: Data Inputs:

1. Open the PageObjectModel project.
2. Navigate to login\_page.py.
3. Open the methods region.
4. Add a method that will accept two strings as parameters.

```
def login_failure(self, username, password):
```

- 5. Include an action that will send the username parameter into the username box.**

```
self.username_field.send_keys(username)
```

- 6. Now, include the password parameter and action.**

```
self.username_field.send_keys(username)
```

```
self.password_field.send_keys(password)
```

- 7. Lastly, press the login button to complete the method.**

```
self.username_field.send_keys(username)
```

```
self.password_field.send_keys(password)
```

```
self.login_button.click()
```

- 8. Now, we will change the method to return the username.**

- 9. Create a variable of the username text to be returned (must be put before the login button click).**

```
username_text = self.username_field.text
```

- 10. Return the new variable in the last line of the method.**

```
return username_text
```

- 11. Navigate to test\_cases.py.**

- 12. Create a new unit test that will run last.**

```
@pytest.mark.order(4)
```

```
def test_failed_login(self):
```

- 13. Add an attribute tag to the test to pass through new values as the username and password.**

```
@pytest.mark.parametrize("username,password", [("username1", "password1")])
```

- This tag can be separate from the ordering tag, placed directly below it.

**14. Add multiple tuples in the tag to run the same test multiple times with different values.**

```
@pytest.mark.parametrize("username,password", [("username1", "password1"),
("username2", "password2")])
```

**15. Update the test to pass through the attribute strings as parameters.**

```
def test_failed_login(self, username, password):
```

- **NOTE:** The names of the string parameters passed through can be different from the parameters names in the class page. This will not affect the results, and it will still pass through correctly.

**16. Update the code to call the LoginWindow method, passing through the new variables.**

```
self.login_page = LoginWindow(self.app_settings_session)
self.login_page.login_failure(username, password)
```

**17. Add the following code to close the error message:**

```
self.error_page = ErrorWindow(self.app_settings_session)
self.error_page.clear_error()
```

**18. Run the test!**

- The test should run successfully, passing through the two created strings as the username and password.

**Target:** Confirm successful use of Data Inputs to test an application!

```

@pytest.mark.order(3)
def test_exit(self):
    self.login_page = LoginWindow(self.app_settings_session)
    self.login_page.login_to_application()
    self.welcome_page = WelcomeWindow(self.app_settings_session)
    self.welcome_page.exit_application()

@pytest.mark.order(4)
@pytest.mark.parametrize("username,password", [("username1", "password1"), ("username2", "password2")])
def test_failed_login(self, username, password):
    self.login_page = LoginWindow(self.app_settings_session)
    self.login_page.login_failure(username, password)
    self.error_page = ErrorWindow(self.app_settings_session)
    self.error_page.clear_error()

```

---

## Module 12 - Test Driven Development (TDD) and the Page Object Model

### Key Points:

- Test Driven Development is a software development process, in which test cases are created before software is fully developed to establish necessary requirements.
- Developers first start with an automated test. Tests will be developed for an application page, creating constructors, objects, and methods to test the future application.
- Then, the test will be run; however, it will fail as the application has not been built yet.
- Developers will then use the test to design a portion of the application, ensuring that the test will pass after the new page is completed.
- After publishing and building the application, the tests will be rerun and should now pass.
- This process, which works very well with the POM framework, is repeated until the application is fully developed and passes all tests.

The following exercise follows Test Driven Development to update the login application:

### Exercise 1: Test Driven Development:

1. Open the WPF application.

- 2. In a PyCharm window, open the PageObjectModel project.**
- 3. In the POM, create a new file and class, titled VerificationWindow for a new page of the application.**
- 4. In the new class, add the necessary constructor, objects, and methods.**
  - The new page should include three objects: a textbox for an authentication code, a button to return to the first screen, and a continue button.
  - Create two methods, one to successfully enter the code and continue and one to fail to authenticate the account.
- 5. Include data inputs for at least one of the methods.**
- 6. Create a new unit test to test the new application window.**
- 7. Run the test even though it will fail.**
- 8. Update the application with the new window, including the provided elements from the test.**
- 9. Update the back-end code of the previously created windows.**
  - The verification window should open after the login screen but before the welcome page.
- 10. Now, repeat the process for a portal page.**
  - Define five elements, four buttons to open four different browser links and one button to quit the application. Create five tests, four to open the links and one to exit.
  - Update the application to match the element information and ensure the tests pass.
  - **NOTE:** You may need to add a continue button on the welcome page to

open the portal.

## 11.Run the tests!

**Target:** Confirm successful use of TDD to update and test the application!

```
class VerificationWindow:

    # region Objects
    def __init__(self, driver):
        self.driver = driver
        self.code_field = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "9")
        self.continue_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "10")
        self.return_button = self.driver.find_element(AppiumBy.ACCESSIBILITY_ID, "11")
    # endregion

    # region Methods
    def verify_successfully(self):
        self.code_field.send_keys("1234")
        self.continue_button.click()

    def verify_failure(self, code):
        self.code_field.send_keys(code)
        self.continue_button.click()

    def return_to_main(self):
        self.return_button.click()
    # endregion
```

---

## Module 13 - Keyword Driven Framework (KDF)

### Key Points:

- Keyword Driven framework is a form of software testing used for automated testing. This framework separates the documentation of test cases – including both the data and functionality to use – from the prescription of the way the test cases are executed.
- Keyword Driven framework depends on an external data source to drive the automation code (CSV, Excel, or Database).
- Data from within the external file is accessed during runtime and drives the automation code based on user inputs.
- KDF testing allows non-programmers to run and test applications in an easier, less technical form, while still depending on POM framework functionality.

The following exercise demonstrates the Keyword Driven Framework in the POM:

### **Exercise 1: Keyword Driven Framework:**

- 1. Open the Page Object Model project.**
- 2. In the project hierarchy, create a new directory, titled “Keywords.”**
  - Add a new Python file, titled “keywords,” to the folder.
- 3. In Keywords.cs, create a class named LowLevelKeywords.**

```
class LowLevelKeywords:
```

- 4. Create a method, where we will define keywords.**

```
def process_keywords(self, keyword, prop_instance, value):
```

- 5. Add case blocks to identify different keywords and provide functionality.**

- These keywords allow the tests to perform actions such as entering text, clicking buttons, and verifying tests.

```
match keyword:
```

```
    case "EnterData":
```

```
        prop_instance.send_keys(value)
```

```
    case "Click":
```

```
        prop_instance.click()
```

```
    case _:
```

```
        raise Exception("Keyword {} not supported.".format(keyword))
```

- 6. In UnitTest1, add a reference to the keywords class.**

```
from Keywords.keywords import LowLevelKeywords
```

- 7. In the setup class method, add the following to instance the keyword class:**

```
keyword_class = LowLevelKeywords()
```



8. **Comment out the `setup_method` and `teardown_method` so the application does not reset in between each test. Additionally, comment out the last line of the `setup` class method.**

```
cls.appium_session.close_app()
```

9. **Comment out the tests already created and put them in a region within the Tests region.**

```
# region Standard Tests  
  
# endregion
```

10. **Create a new region in the tests region for Keyword tests.**

```
# region Keyword Tests  
  
# endregion
```

11. **Create a new test with five parameters.**

```
def test_kwf(self, page, element, keyword, value):
```

12. **Add the following code in the test to pass through the parameters to the keyword class:**

```
class_type = globals()[page]  
class_instance = class_type(self.app_settings_session)  
get_prop_info = getattr(class_instance, element)  
self.keyword_class.process_keywords(keyword, get_prop_info, value)
```

13. **Add attribute tags to specify fields and values for the test.**

```
@pytest.mark.parametrize("page,element,keyword,value", [("LoginWindow",  
"username_field", "EnterData", "Value"), ("LoginWindow", "password_field", "EnterData",  
"Value"), ("LoginWindow", "login_button", "Click", "Null")])
```

- Replace *Value* with your own credentials to login.
- **NOTE:** The keywords must match those defined in the keyword class.

## 14. Run the tests! The application should open and login to the application successfully.

pytest

**Target:** Confirm successful use of KDF to test the application!

```
class LowLevelKeywords:
    def process_keywords(self, keyword, prop_instance, value):
        match keyword:
            case "EnterData":
                prop_instance.send_keys(value)
            case "Click":
                prop_instance.click()
            case _:
                raise Exception("Keyword {} not supported.".format(keyword))
```

---

## Module 14 - Adding Data Inputs to the KDF

### Key Points:

- Data inputs allow users to separate test case information from test methods and classes, instead storing them in external files, such as a CSV file.
- Storing data inputs in external files allows companies and workers with little programming experience to run and automate testing without needing to build tests or technical skills.

The following exercise demonstrates the addition of data inputs to the KDF:

### Exercise 1: Data Inputs with the KDF:

1. Open the Page Object Model project.

2. In the project hierarchy, create a new directory, titled “Data.”
3. Add a new file to the directory and name it test\_case\_data.csv.
4. In test\_case\_data, add the following placeholder information:

```
Placeholder,Step,Page,Object,Keyword,Value
```

- These elements will hold the test case data for each test, allowing users to separate data inputs into the new CSV file.

5. In test\_cases, import a CSV reader. Right-click on the error and install the package.

```
import pandas as pd
```

6. Above the test region, create a new region for the CSV reader.

```
# region CSV Reader  
  
# endregion
```

7. In this region, add a method for the session to read the CSV file elements.

```
@staticmethod  
def get_test_data():
```

8. Using the CSV reader, open the file by using the CSV file path.

```
df = pd.read_csv(r"C:\Users\Administrator\Documents\AppData Python "  
                r"Training\PageObjectModel\Data\test_case_data.csv", index_col=0)
```

9. Add the following code to allow the session to read the file elements:

```
for i, row in df.iterrows():  
    step = str(row[0])  
    page = str(row[1])  
    object = str(row[2])  
    keyword = str(row[3])  
    value = str(row[4])  
    yield step, page, object, keyword, value
```

**10. Above test\_kwf, comment the parameterize tag.**

**11. Create a new test tag that will use parameters from the CSV file.**

```
@pytest.mark.parametrize("step,page,element,keyword,value", get_test_data())
```

**12. Now, add test data to the CSV file.**

```
Placeholder,***STEP1***,LoginWindow,username_field,EnterData,Value
```

```
Placeholder,***STEP2***,LoginWindow,password_field,EnterData,Value
```

```
Placeholder,***STEP3***,LoginWindow,login_button,Click,Null
```

- Replace *Value* with your own credentials.

**13. Run the test!**

```
pytest
```

**Target:** Confirm successful use of data inputs in the Keyword Driven Framework!

```
# region CSV Reader

@staticmethod
def get_test_data():
    df = pd.read_csv(r"C:\Users\Administrator\Documents\AppData Python "
                    r"Training\PageObjectModel\Data\test_case_data.csv", index_col=0)
    for i, row in df.iterrows():
        step = str(row[0])
        page = str(row[1])
        object = str(row[2])
        keyword = str(row[3])
        value = str(row[4])
        yield step, page, object, keyword, value

# endregion
```

---

## Module 15 - User Acceptance Testing (UAT) and the KDF

Key Points:

- User Acceptance Testing is the process of testing applications for the intended users.

- This testing will focus more on business verifications, rather than low level functionality that was done with the POM methods. SME will be involved to create the test case flows, rather than programmers.
- Keyword Driven Approach makes this overall process faster and easier for users.
- User Acceptance Testing builds on the POM as it leverages all the work done, such as the object repository.

The following exercise demonstrates User Acceptance Testing:

### **Exercise 1: User Acceptance Testing:**

- 1. Open the Page Object Model project.**
- 2. Create three new keywords and add code to provide functionality in the LowLevelKeywords class.**

- Use the following keywords to start:

case "VerifyText":

*ENTER CODE HERE*

case "ClearText":

*ENTER CODE HERE*

case "IsDisplayed":

*ENTER CODE HERE*

- 3. Create new test cases, each under the Data folder.**
- 4. Utilize each of the new keywords and ensure successful testing.**
- 5. Run the new tests!**

**Target:** Confirm successful User Acceptance Testing with the KDF!

```
1 Placeholder, Step, Page, Object, Keyword, Value
2 Placeholder, ***STEP1***, LoginWindow, username_field, EnterData, John
3 Placeholder, ***STEP2***, LoginWindow, username_field, ClearText, Null
4 Placeholder, ***STEP3***, LoginWindow, username_field, EnterData, John
5 Placeholder, ***STEP4***, LoginWindow, password_field, EnterData, Smith
6 Placeholder, ***STEP5***, LoginWindow, username_field, VerifyText, John
7 Placeholder, ***STEP6***, LoginWindow, clear_button, IsDisplayed, Null
8 Placeholder, ***STEP7***, LoginWindow, login_button, Click, Null
9 Placeholder, ***STEP8***, VerificationWindow, code_field, EnterData, 1234
10 Placeholder, ***STEP9***, VerificationWindow, continue_button, Click, Null
```

---

## Module 16 - Enhancing the Template with Reporting Capability

### Key Points:

- Enabling HTML reports allows tests to report test information such as test duration and error reports after each test
- This reporting capability gives users more control over each test and data on application state during and after tests.
- Application HTML reports can be added to the test detail summary to provide quick accessibility to test summaries.

The following exercise demonstrates Addition Reporting Capability:

### Exercise 1: Addition Reporting Capability:

1. Open the Page Object Model project.
2. Navigate to test\_cases.py, and create a new region below the CSV reader region.

```
# region Test Report
# endregion
```

3. Inside this region, create a new method that will provide test summaries.

```
@staticmethod
```

```
def report_summary(step, page, element, keyword, value):
```

**4. Add statements that will provide test reporting summaries after each test.**

```
print(step)
```

```
print("Page: {}".format(page))
```

```
print("Action: {} on Element: {}".format(keyword, element))
```

```
print("Value: {}".format(value))
```

**5. Inside the keyword test, call the new method, passing in each parameter from the CSV file.**

```
self.report_summary(step, page, element, keyword, value)
```

**6. In the terminal, install pytest's HTML reporter.**

```
pip install pytest-html
```

- We will use the Pytest HTML package to create HTML reports and summaries.

**7. To run tests and create an HTML report, add the following in the terminal.**

```
pytest --html=report.html
```

**8. In the project hierarchy, navigate to and open report.html. In the toolbar, click View, Open in Browser, and Chrome.**

- The HTML report should open in a new tab with provided information on each test. Users can sort tests by duration, name, or test result. The browser will update each time the tests are run.

**9. Run the tests!**

- The tests should pass successfully and the test detailed summary should provide information about each completed test!

**Target:** Confirm successful implementation of Addition Reporting Capability!

## Summary

9 tests ran in 8.00 seconds.

(Un)check the boxes to filter the results.

☒ 9 passed, ☐ 0 skipped, ☐ 0 failed, ☐ 0 errors, ☐ 0 expected failures, ☐ 0 unexpected passes

## Results

[Show all details](#) / [Hide all details](#)

▲ Result	▼ Test
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP1***-LoginWindow-username_field-EnterData-Ryan]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP2***-LoginWindow-username_field-ClearText-Null]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP3***-LoginWindow-username_field-EnterData-Ryan]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP4***-LoginWindow-password_field-EnterData-Tobin]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP5***-LoginWindow-username_field-VerifyText-Ryan]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP6***-LoginWindow-clear_button-IsDisplayed-Null]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP7***-LoginWindow-login_button-Click-Null]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP8***-VerificationWindow-code_field-EnterData-1234]
Passed <a href="#">(show details)</a>	Tests/test_cases.py::TestCases::test_kw[***STEP9***-VerificationWindow-continue_button-Click-Null]

---

## Module 17 - Save and Utilize the Template on a New Application

### Key Points:

- All of these skills can work together to build and test an application.
- Utilizing the necessary test design (simple Appium tests, POM, or KDF) allows for easier and improved testing that fits the requirements of any application.
- Test Driven Development is especially useful when building an application.

The following exercise utilizes all of the foregoing skills:

### Exercise 1: Applying the Framework to a New Application:

1. Open PyCharm and create a new project.
2. Referencing the WPF project, begin building out a PyQt5 GUI application with multiple elements and pages. The following installation and imports will be necessary.



```
pip install pyqt5

from PyQt5.QtWidgets import *

from PyQt5 import QtCore

from PyQt5.QtGui import *

import sys
```

**3. Use the following code to build the PyQt5 application once it is complete.**

```
pip install pyinstaller

pyinstaller.exe --onefile --windowed app.py
```

**4. Work with this new application to create an Appium test with multiple tests, including each element and page.**

**Target:** Confirm successful creation and testing of a PyQt5 GUI!

```
def window():

    def login_button_clicked():
        if username_field.text() == "Ryan" and password_field.text() == "Tobin":
            window2.show()
            username_field.clear()
            password_field.clear()
            window.hide()

    def back_button_clicked():
        window2.hide()
        window.show()

    app = QApplication(sys.argv)
    window = QWidget()
    window.setGeometry(50, 50, 300, 300)
    window.setWindowTitle("Login Application")

    window2 = QWidget()
    window2.setGeometry(50, 50, 300, 300)
    window2.setWindowTitle("Welcome!")

    username_label = QLabel("Username: ", window)
    username_label.move(55, 50)
    username_field = QLineEdit(window)
    username_field.setAccessibleName("username_field")
    username_field.move(120, 50)
```

---

## Module 18 - Installing JMeter

Key Points:

- Apache JMeter is a free and 100% pure Java desktop application and the leading open-source tool to test:
  - Load - the sheer volume your program can process. For example, its ability to download a large series of files from the internet.
  - Performance - how quickly and efficiently users can move through your program. For example, its performance against that of a similar system.
- JMeter is designed to load test client/server software test performance both on static and dynamic resources (Files, Java Servlets, CGI Scripts, Java Objects, Databases, FTP Servers).
- Features:
  - Full Test IDE that allows fast Test Plan recording (from browsers or native applications), building, and debugging.
  - Command-line mode to load-test from any Java-compatible operating system.
  - Dynamic HTML reporting.
  - Multi-threading framework to allow concurrent sampling by many threads and simultaneous sampling of different functions by separate thread groups.
- Sample testing capabilities:
  - Identify the maximum operating capacity of an application.
  - Identify any bottlenecks and determine which element is causing degradation.
  - Demonstrate that a system meets performance criteria.

The following exercise outlines JMeter installation:

### **Exercise 1: Installing JMeter:**

*Completion of this course requires user installation of the Java Development Kit (JDK). This kit comes with the Java Runtime Environment (JRE). If you do not have the ability to run the Java Setup Executioner, please consult your System Administrator.*

**1. Ensure installation of Java on your system by running `$ java -version` from the command line.**

- If the output does not indicate installation and version, follow the instructions below for Java setup on your system. Else, proceed to step 2.

**Java Installation and Setup:**

Begin by downloading the latest version of the Java SE Development Kit and running the inherent setup. Then, create environment variables:

Windows:	macOS/Linux:
<ul style="list-style-type: none"><li>• Navigate to Control Panel -&gt;System and Security-&gt;System-&gt;Advanced system settings-&gt;Environment Variables.</li><li>• Create a new user variable called 'JAVA_HOME'.<ul style="list-style-type: none"><li>○ If you have permission, you may create it under a system variable.</li></ul></li><li>• Paste the path to your JDK and click OK.</li><li>• If you have system permission, edit the existing 'Path' variable under system variables. Otherwise, create a new user variable called 'Path' and paste the path to your JDK's bin directory (java.exe).</li><li>• Click OK and finish.</li></ul> <p><b>Note:</b> the arguments on the Path variable are separated by semicolons. If you are editing an existing Path variable, make sure you add a ";" before your value and DO NOT overwrite the existing path in its entirety.</p>	<ul style="list-style-type: none"><li>• Open the terminal/command line.</li><li>• Run <code>\$ vi .bash_profile</code> to open your path settings in a text editor.</li><li>• Scroll to the end of the file and add <code>export JAVA_HOME=\$(/usr/libexec/java_home)</code> on the last line.</li><li>• Press esc and enter <code>:wq</code> to save the file and exit the editor</li><li>• Run <code>\$ source .bash_profile</code> in the terminal to refresh.</li><li>• Check that running <code>\$ echo \$JAVA_HOME</code> in the terminal yields <code>/Library/Java/JavaVirtualMachines/YourJDKVersion/Contents/Home</code>.</li></ul>

**2. Download the binary JMeter zip file from: <https://jmeter.apache.org/>**

**3. Unzip the file.**

- Your unzipped file will now appear in the Downloads location of your computer's library; you may store it at any location now. It will be useful to note the exact names and full paths of both this folder and its

encompassing folder.

**4. Start the application (you may wish to refer back to this methodology as you are getting to know JMeter in the next few exercises).**

- Windows:
  - Navigate to the location of your downloaded JMeter folder, open the encompassed bin folder, then the jmeter.bat file.
- macOS/Linux:
  - Open the terminal. Using the `$ cd FolderName` command, change your directory to the container of the JMeter folder, then the JMeter folder, then its bin folder. Run `$ sh jmeter.sh` to open the application.

**Target:** Confirm that you have successfully launched JMeter!

---

## Module 19 - Creating and Using Assertions

### Key Points:

- Assertion basics:
  - JMeter components which evaluate information in the response from the application under test (AUT), sometimes using Regular Expressions.
  - Can be added as a child of any Sampler, such as an HTTP Request, to assert something about the response of that particular component.
  - Compare actual runtime results to expected values.
  - Assure no errors occur in your operations.

- Assertions are important and very useful in any software automation tools to assert attributes of the response during test execution.
- An Assertion Listener should be added to the Thread Group to view Assertion results.
- Failed Assertions will appear in the Table and Tree View Listeners and as part of the error percentage in the Aggregate and Summary reports.

The following exercise outlines the creation and use of several Assertion types:

### **Exercise 1: Assertion Types:**

- 1. Create a new JMeter Test Plan.**
- 2. Create a Homepage Request (you can use Google as the UR)**
- 3. Right-clicking Homepage in the sidebar, create a Response Assertion (Add->Assertions->Response Assertion).**
- 4. In the Response Assertion form, specify the following:**
  - In Field to Test section, select the radio button as Response Code  
*Pattern Matching Rule: Equals*  
*Pattern to Test: 200*
    - This will check a successful connection to the server.
- 5. Specify Thread Properties from the Thread Group form:**
  - Number of Threads (Users): 5*
  - Ramp-Up Period (in seconds): 20*
  - Loop Count: 1*
- 6. Add Assertion Results Listeners from the Thread Group.**
- 7. Run the test and view each Listener.**
  - A green Status verifies that each thread has passed the Assertion - in this case, the response code was found equal to 200!

**8. Delete 200 as the Assertion's Pattern to Test and instead add 300.**

**9. Rerun the test and view each Listener.**

- A red Status shield indicates a failed Assertion on each thread.
- Under the Results Tree, expand a thread to display information about the assertion failure, including its type and the expected vs. actual value.

**10. Reset the Response Assertion's Pattern to Test to 200.**

**11. Create a Duration Assertion from Homepage.**

- View your Results Table. Identify a Sample Time (ms) within your result range. Specify this number as the Duration in milliseconds in your Duration Assertion form.

**12. Clear previous results, then rerun the test and view each Listener.**

- How many threads ran within the given time and passed the Duration Assertion?

**13. Create a Size Assertion from Homepage.**

- View your Results Table. Identify the byte size of each sample and specify this number as the Size in bytes to Assert for your Size Assertion.

**14. Clear previous results, then rerun the test.**

**15. View each Listener.**

- Did any tests fail the Size Assertion? If so, were they larger or smaller than the Size to Assert?
- Be sure to use the expander triangle in the Results Tree to identify the type of Assertion (if any) that failed since several Assertions are now part of the Test Plan.

**16. Create an HTML Assertion from Homepage.**

**17. Clear previous results, then rerun the test.**

- View Assertion Results. What kind of errors do you see?

**18. Note the two numbers beside Tidy Parser errors and Tidy Parser warnings in the Assertion result of the Results Tree. Specify the Error and Warning thresholds of your HTML Assertion to mirror these numbers.**

**19. Clear previous results, then rerun the test.**

- You should no longer see a failed HTML Assertion, as you have configured the Test Plan to allow the initially found number of errors and warnings in the HTML syntax of the response data.

**Target:** Confirm that your Test Plan has functional Response, Duration, Size, and HTML Assertions.

---

## Module 20 - Creating and Using Listeners

Key Points:

- Listener basics:
  - JMeter components which allow you to view, save, and read test results such as response time metrics or assertion success/failure, for example.
  - Present results in tabular/graphical form.
  - Include a panel where a file to write to or read from can be specified.
  - Can be configured to save different items in a .csv or .xml format to the result log files through the Config Popup.

- Certain Listeners may utilize significant memory because they keep a copy of every sample in their scope.

The following exercise outlines the creation and use of several Listener types:

### **Exercise 1: Listener Types:**

- 1. From Exercise4 at the sidebar, add a Thread Group.**
- 2. From the Thread Group, add an HTTP Request Sampler with the following:**

*Name: Homepage*

*Server Name or IP: any*

*Path : /*

- 3. Add a View Results in Table Listener from the Thread Group.**
  - Since it is a child of the Thread Group, this Listener would only apply to this Group if others were present within the Test Plan.
- 4. Run the test and view the Results Table.**

**NOTE:** Latency is the Request's time to the first byte, Connect Time (ms) is its time to connect to the server. Like the Results Table, each Listener has an option to save results to a .csv or .xml file for records. Once this file is specified for a Listener, running a test saves the results to this file.

- 5. Add a View Results Tree Listener from the Thread Group.**
- 6. Rerun the test and view the results in this Tree format.**
  - Navigate the Sampler result, Request, and Response data tabs to view new details of the results.
- 7. Add an Aggregate Report Listener from the Thread Group.**
- 8. Navigate to the Thread Group form and specify the following properties:**

*Number of Threads (users): 20*



*Ramp-Up Period (in seconds): 5*

*Loop Count: Infinite*

**9. Rerun the test and view the live and ongoing results in the Aggregate Report.**

- This Report accumulates metrics and gives us a single line aggregate report from a test run.

**10. Shut down the test run.**

**11. Add a Graph Results Listener from the Thread Group.**

**12. Rerun the test while viewing the live results in this visual format.**

- The Average, Median, and Deviation figures represent Response time of the server while Throughput is the number of Requests per time measure.

**13. Shut down the test run.**

**14. Navigate to the Thread Group form and specify the following properties:**

*Number of Threads (users): 5*

*Ramp-Up Period (in seconds): 1*

*Loop Count: 1*

**15. Add a Summary Report Listener from the Thread Group.**

**16. Rerun the test and view the results in this Summary Report format.**

- Which labels are common between Aggregate and Summary Reports?

**17. Add a Simple Data Writer Listener from the Thread Group**

**18. Rerun the test - Do you see your results file? What does it look like in terms of rows and columns?**

**Target:** Confirm that your Test Plan has functional View Results in Table, View Results Tree, Aggregate Report, Graph Results, and Summary Report Listeners. Confirm that

you have successfully written custom test results to a file using the Simple Data Writer Listener.

---

## Module 21 - Testing Web Services (API)

### Key Points:

- A Web Service is a medium to support client-server communications for World Wide Web applications.
- The Web Service performs certain functions for the client when invoked.
- Web Service methods:
  - Simple Object Access Protocol (SOAP) to provide an envelope for sending Web Services messages over the Internet.
  - Representational State Transfer (REST), a stateless architecture that uses HTTP protocol.
  - SOAP vs. REST.
- Web Services involve .xml and .json formats and implement a request/response design.
- Crucial components of a JMeter Web Services Test Plan include:
  - A Thread Group.
  - An HTTP Request Sampler.
  - A Listener.
  - Implementation of POST and GET data transfer methods.
    - GET - Transfers Request parameter in the URL string.
    - POST - Transfers Request parameter in the message body (more secure).

The following exercise explores Web Service methods and their incorporation into JMeter Test Plans:

## **Exercise 1: Testing SOAP GET, SOAP POST, REST GET, REST POST:**

### **Testing SOAP GET:**

- 1. Visit the webpage for OpenWeatherMap API and at the top menu, click Sign Up.**
- 2. Once you have created and signed into your account, navigate to the API keys menu and locate your Default Key.**
  - Replace {YourAPIKey} with this value below.
- 3. Open JMeter and from Exercise9, add a Thread Group and name it 'Thread Group - SOAP'.**
- 4. From the Thread Group, add an HTTP Request Sampler with the following properties:**

*Name: SOAP GET Request*

*Server Name or IP: api.openweathermap.org*

*Path: /data/2.5/weather*

- 5. In the Send Parameters With the Request form of the SOAP GET Request, add the following inputs:**

*Name: q          Value: Boston          Include Equals?: ✓*

*Name: appid      Value: {YourAPIKey}*

*Include Equals?: ✓*

- 6. Add a View Results Tree Listener.**

**7. Run the test and view the Response data tab within the View Results Tree form.**

- Did you receive Boston's weather from the server?

**Testing SOAP POST:**

**8. Create a second HTTP Request Sampler from the Thread Group and specify the following properties:**

*Name: SOAP POST Request*

*Protocol [http]: https*

*Server Name or IP: www.flickr.com*

*Path: /services/soap*

*Method: POST*

**9. In the Body Data tab of the SOAP POST Request, add the following input:**

```
<s:Envelope
  xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
>
  <s:Body>
    <x:FlickrRequest xmlns:x="urn:flickr">
      <api_key>04046430b2487c541a163cbef3d54750</api_key>
      <method>flickr.cameras.getBrandModels</method>
      <brand>apple</brand>
    </x:FlickrRequest>
  </s:Body>
</s:Envelope>
```

- This initiates a SOAP POST Request to the flickr.com Server. Information

about the API is available at this link. Above we select a calling convention and send a request to the API endpoint specifying a method and arguments for the action.

**10. Clear all results and rerun the test.**

**11. Navigate to the View Results Tree form.**

- Has your SOAP POST Request been successful?
- View the Request and Response data tabs of the POST Request to see the POST data.

## Testing Rest Request with JSON (instructor led example):

- a. Change the [www.flickr.com](http://www.flickr.com) SOAP example to be a REST test that returns a JSON format
- b. Add a JSR223 PostProcessor

```
import groovy.json.JsonBuilder
import groovy.json.JsonSlurper

log.info("b4 response#####")

def response = prev.getResponseDataAsString()

def cleanResponse = response.replaceAll("jsonFlickrApi\\(", "")
log.info("*****" + cleanResponse)
cleanResponse = cleanResponse.replaceAll("\\\\", "")

log.info("***** This is the clean response: " + cleanResponse)
def jsonSlurper = new JsonSlurper()
def json = jsonSlurper.parseText(cleanResponse)

log.info("value of a response: " + json)
def results = json.results

def builder = new JsonBuilder(results)
vars.put("results", builder.toPrettyString())

f = new FileOutputStream("C:/output.json", true);
p = new PrintStream(f);

p.println(cleanResponse);

p.close();
f.close();
```

- c. Add a HTTP Request for File

- d. Add JSON Assertion
- e. Add JSON Extractor
- f. Add a Debug Post Processor

**Target:** Confirm that you have made successful GET and POST Requests with both SOAP and REST methods from JMeter.

---

## Module 22 - Getting to Know JMeter Topologies

### Key Points:

- Local Setup:
  - Components:
    - JMeter GUI
    - Browser/Proxy
    - Application Under Test (AUT)
  - Record feature:
    - Allows you to modify pre-recorded tests to run against many different environments.
    - Local proxy server setup (HTTP or HTTPS) allows JMeter to record user activity when using a browser.
    - JMeter creates test sample objects and stores them .
    - You need to set up your browser to use a proxy server for all HTTP and HTTPS requests.
    - HTTPS connections use an authentication process for the connection between the server and the browser.

- An HTTPS certificate is presented to the browser and then the browser authenticates the certificate by checking that it is signed by a Certificate Authority (CA) that is linked to one of the server's in-built root CAs.
  - If the check fails the user is prompted to decide if the connection should be made.
  - JMeter mimics the certificate exchange and validation.
- Distributed Setup:
  - Involves using multiple systems to configure JMeter.
  - There are three components we must understand to do distributed testing in Jmeter:
  - Master - the system that is running the JMeter GUI. The Master acts as a hub that controls each Slave machine.
  - Slave - a JMeter server that receives data from the master and sends it to the Target under test. You can have an unlimited number of Slaves interacting with the test Target as this simulates multiple IP addresses accessing the Target.
  - Target - web server under test. The target gets requests from the slaves and conducts the operations. Results are returned from the Target to the Master.

The following exercise explores JMeter Topologies:

## **Exercise 1:**

### **Remote-Distributed Execution with HTML report**

- Edit rmi in the file user.prop all the way at the bottom
- Look at JMeter properties, we can turn off the whole KeyStore thing for starters
- Then edit jmeter.prop, search for "host" , add new IP
- Start the jmeter-server.bat to launch the node



- This should end with creating a bat file that runs remote execution. Command  
Line execution for a remote server: -R <IP Address>

[https://jmeter.apache.org/usermanual/jmeter\\_distributed\\_testing\\_step\\_by\\_step.html](https://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.html)

### **Test the option to create an HTML Report from an existing results file**

```
del "C:\apache-jmeter-5.0\bin\jmeter_reports\*.*"
del /q "C:\apache-jmeter-5.0\bin\jmeter_reports\*"
FOR /D %%p IN ("C:\apache-jmeter-5.0\bin\jmeter_reports\*.*") DO rmdir "%%p" /s /q
Jmeter -Jjmeter.save.saveservice.subresults=false -n -t "C:\apache-jmeter-
5.0\bin\load_test.jmx" -R <IP Address > -l "C:\apache-jmeter-
5.0\bin\jmeter_reports\ResultsTable.csv" -e -o "C:\apache-jmeter-
5.0\bin\jmeter_reports\html_folder"
```

---