**Automation Sample Test Description**

# Contents

Contents 1

Summary 1

Setup Instructions 2

Script Execution 2

Abstraction in Automation Test Frameworks 3

Test Framework Directories 4

# Summary

Documentation is always important in creating context and utility for any kind of testing. The utility of this sample I have provided should be considered within a very narrow scope. It provides an example of my ability to create an automation script for the purpose of assessment which can then serve as a larger part of discussion about overall approach. This script itself is not an example of how a larger test automation framework would be laid out. With some modification a script like this sample can be placed into a larger framework that will provide a structure that will allow for simpler test creation and maintenance of a larger set of tests that would be expected for testing any software release. Besides simplifying things for the individual test engineer a framework like this should be considered essential for providing a systematic path that will allow for more approachable use in a team context. It is important keep this distinction in mind and this sample must be considered this way.

This automation sample does the following:

1. Demonstrates use of a data driven methodology using CSV files and Python panda dataframe arrays to structure and drive tests in an approach that is highly scalable and maintainable.
2. Uses two Python-based Selenium scripts to allow for convenience of assessment and illustrate differences between how Selenium interfaces with Firefox and Chrome browsers
3. Registers multiple users by filling out the automation practice website page and logs them in individually and in sequence to the automation practice website.
4. Performs essential object recognition validation tasks with the implementation of wait commands to manage Javascript page delays that affect recognition and interaction with fields in the form
5. It generates results screenshots allowing for visual confirmation by the test team. This is the extent of the results reporting.
6. It provides very simple inline debugging structure using print statements for feedback during test execution during test execution allowing for quicker identification of problems

# Setup Instructions

Since the scripts rely on an internet based website there must be access from the machine running these tests. The following needs to be installed into the Python path of a machine so that it is accessible by the scripts. For purposes of simplicity here I recommend using a Linux or MacOS based system. pip is the preferred mechanism for installing these files and packages.

1. Selenium Webdriver
2. Pandas data analysis library

The scripts and csv files that drive the tests are included in the zipped package provided so no direct setup is required when unzipping the package.

# Script Execution

Inside the AutoPractice folder the two Python executables FillByXpathChrome.py and FillByXpathFirefox.py drive the each set of tests individually. There is no test harness that ties them together. They are essentially duplicates of each other with the only difference they run the two different browsers. Run these scripts by issuing python <script name> at the command line.

There are two controls or ways to most directly adjust the execution of the script:

1. From the corresponding two CSV files chrome\_datasheet.csv and firefox\_datasheet.csv by directly changing the data. For the automationpractice.com website the email address acts as the primary key and additional user profiles can be created by replacing the leading character of “a” to “b” or similar character naming convention. There is a limit of 32 characters on the automationpractice.com site but otherwise there is freedom in the format though it most likely needs a valid email extention (@example.com)
2. From the for loop where the data set range can be adjusted to narrow or select a particular range and number of user profiles intended to be loaded. The data in the two csv files is currently sitting at 200 rows apiece.

These two control points allow for the fine tuning of the test data for customizing tests for different testing purposes. The automationpractice.com website is a very basic bare bones ecommerce type of demo site but it does perform some basic form validation on the fields such as data type and field length so these had to be accommodated while creating the test data and similar concerns would apply for any web interface under test.

Results reporting is limited in this script to satisfactory completion of the script and a visual and manual check of the screenshots generated. Realistically, because of the relative simplicity of the script and what it is testing a spot check of a couple items is sufficient. In the case of including this script into a larger test framework better reporting mechanisms such as unittest or pytest would offer some of the more compelling approaches to providing at-a-glance status of test results.

# Abstraction in Automation Test Frameworks

As mentioned at the beginning of this document this automation sample is not an example of what a maintainable Selenium based automation framework that would be included in a production workflow. However with modification the sample could be easily tailored to fit. The utility of a well-designed framework quickly becomes apparent when setting up automated tests to be used in a professional development environment. Generally the approach to frameworks is to provide an easy to navigate structure for creation, execution, and maintenance of scripts. This means using basics like a standard coding style-sheet convention such as Python PEP-8 (see <https://www.python.org/dev/peps/pep-0008/>) But more importantly it means providing a standard file structure that telegraphs the purpose of the files and code. This means since Python is selected that making full use of the object oriented features of the language will allow us to create control points that allow us to essentially create lists of execution tasks. The Sample script did and did not provide an illustration of how this should be done. By using the data-driven csv file technique it showed the power of driving tests from lists but the AutoPractice Sample fell short or may more accurately departed from the principle of breaking the tests into a more abstracted list driven approach. See below for a short illustration in code.

class LoginPage(WebDriver):

def \_\_init\_\_(self, driver):

super().\_\_init\_\_(driver)

self.driver = driver

# Locators

\_login\_link = "Login"

\_email\_field = "user\_email"

\_password\_field = "user\_password"

\_login\_button = "submit"

def clickLoginLink(self):

self.elementClick(self.\_login\_link, locatorType="link")

def enterEmail(self, email):

self.sendKeys(email, self.\_email\_field)

def enterPassword(self, password):

self.sendKeys(password, self.\_password\_field)

def clickLoginButton(self):

self.elementClick(self.\_login\_button, locatorType="name")

def login(self, email, password):

self.clickLoginLink()

self.enterEmail(email)

self.enterPassword(password)

self.clickLoginButton()

You will notice at the bottom of this short script the function definition of login which provides the very simple and easily visible list of all the actions be performed in the script. This kind of approach makes it very easy for test maintenance as well as new test script additions.

# Test Framework Directories

This abstraction approach demonstrates the importance of creating a good directory structure to organizing the data. Clearly, much of this is dictated by the features and component parts of any particular software project. However there are still some useful general categories that can be generally applied. What I would recommend to start with is something like the following directory structure for a Selenium automation effort using Python.

Directories inside the Python Project:

**base**  - this directory holds the scripts that perform common tasks that can be abstracted out across all the pages (see below for an example breakdown of what a script like this would be)

**pages**  - this directory holds the scripts that execute on each specific page using page object model.

**tests**  - this directory holds scripts for the main entry points and test harness listings (see below for small example of what this might look like)

**utilities**  – this directory holds scripts supporting tasks like data-analysis scripts

**config-files** – things like the selenium chromedriver file

**screenshots** - this actually should fit inside the pages directory since screenshots are directly tied to specific pages

**Base class example** The following shows how a base class might be written to abstract out a verify title task that would be common to all pages

class BasePage(SeleniumDriver):

def \_\_init\_\_(self, driver):

"""

Inits BasePage class

Returns:

None

"""

super(BasePage, self).\_\_init\_\_(driver)

self.driver = driver

self.util = Util()

def verifyPageTitle(self, titleToVerify):

"""

Verify the page Title

Parameters:

titleToVerify: Title on the page that needs to be verified

"""

try:

actualTitle = self.getTitle()

return self.util.verifyTextContains(actualTitle, titleToVerify)

except:

self.log.error("Failed to get page title")

print\_stack()

return False

**Test Class Example** (For Test Scripts in Test Directory using unittest)

import unittest

from tests.home.login\_tests import LoginTests

# Get all tests from the test classes

tc1 = unittest.TestLoader().loadLoginTests)

tc2 = unittest.TestLoader().loadDataTests)

# Create a test suite combining all test classes

smokeTest = unittest.TestSuite([tc1, tc2])

unittest.TextTestRunner(verbosity=2).run(smokeTest)