What is Robot Framework?

> Robot Framework is an open-source, generic test automation framework for acceptance testing, acceptance test-driven development (ATDD), and robotic process automation (RPA).

It provides a flexible, keyword-driven approach to writing test cases and automating tasks.

Developed in Python, Robot Framework allows users to create high-level, human-readable test cases that can be easily understood by both technical and non-technical stakeholders.

Key features of Robot Framework include:

1. **Keyword-Driven Testing:** Test cases are written in a tabular format using keywords and arguments. This makes the test cases easily readable, understandable, and maintainable.

   💡 Below is an example of a simple keyword-driven test case written in Robot Framework. In this example, I'll create a basic test scenario for a login functionality using keywords:

```
*** Settings ***
Library    SeleniumLibrary

*** Variables ***
${BROWSER}    Chrome
${URL}        <http://www.amazon.com>
${USERNAME}   myusername
${PASSWORD}   mypassword
```

```
*** Test Cases ***
Login Test
    [Tags]    Smoke
    Open Browser    ${URL}    ${BROWSER}
    Input Username    ${USERNAME}
    Input Password    ${PASSWORD}
    Click Login Button
    Verify Successful Login

*** Keywords ***
Open Browser
    [Arguments]    ${url}    ${browser}
    Open Browser    ${url}    ${browser}

Input Username
    [Arguments]    ${username}
    Input Text    id=username    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    id=password    ${password}

Click Login Button
    Click Button    id=loginButton

Verify Successful Login
    Page Should Contain    Welcome, ${USERNAME}
    Close Browser
```

In this example:

- **Settings:** The SeleniumLibrary is imported to provide keywords for web testing.
- **Variables:** Various variables are defined for the test, such as the browser to use, the URL of the application, and login credentials.

- **Test Case (`Login Test`):** The main test case is composed of high-level keywords like `Open Browser`, `Input Username`, `Input Password`, `Click Login Button`, and `Verify Successful Login`.
- **Keywords (`Open Browser`, `Input Username`, `Input Password`, `Click Login Button`, `Verify Successful Login`):** These keywords abstract the lower-level details of interactions with the application. They take arguments, perform actions, and make verifications.

This is a simplified example, and in a real-world scenario, these keywords might be implemented in external libraries or resource files. The idea is to have a modular and readable structure where the test case focuses on high-level actions without getting into the implementation details.

---

2. **Extensible:** Robot Framework is highly extensible, and users can create custom libraries in Python, Java, or other languages to extend its functionality. There is also a wide range of built-in and third-party libraries available for various purposes, such as web testing, database testing, API testing, and more.

> 💡 Below is an example of creating a simple custom library in Python for Robot Framework. This custom library provides a keyword called `Multiply` that multiplies two numbers.

2.1. Create a new Python file named `CustomLibrary.py`:

```
# CustomLibrary.py
class CustomLibrary:
    def multiply(self, a, b):
        """Multiply two numbers."""
        result = int(a) * int(b)
        return result
```

2.2. Create a Robot Framework test file named `CustomLibraryTest.robot`:

```
*** Settings ***
Library    CustomLibrary.py

*** Test Cases ***
Multiply Test
    ${result}=    Multiply    3    4
    Should Be Equal As Numbers    ${result}    12
```

2.3. Run the test with the Robot Framework:

Open a terminal or command prompt and navigate to the directory containing both `CustomLibrary.py` and `CustomLibraryTest.robot`.

Run the following command:

```
robot CustomLibraryTest.robot
```

This will execute the test case, and you should see the test pass since the custom library multiplies 3 and 4, resulting in 12.

In this example, `CustomLibrary.py` is a simple Python class with a `multiply` method. The `multiply` method takes two arguments, multiplies them, and returns the result. The Robot Framework test file (`CustomLibraryTest.robot`) imports and uses this custom library to perform a multiplication test.

This demonstrates the extensibility of Robot Framework, where you can create your own libraries in Python or other supported languages to add custom functionality and keywords to your test automation projects.

---

3. **Cross-Platform:** Robot Framework is platform-independent and can be used on various operating systems like Windows, Linux, and macOS.

💡 One of the notable features of Robot Framework is its cross-platform compatibility. Below is an example of a simple Robot Framework test case that demonstrates cross-platform functionality by opening a web browser on different operating systems:

```
*** Settings ***
Library           SeleniumLibrary

*** Variables ***
${BROWSER}        Chrome

*** Test Cases ***
Open Browser Test
    [Setup]    Open Browser    <http://www.amazon.com>
${BROWSER}
    Do Something Platform Specific
    [Teardown]    Close Browser

*** Keywords ***
Do Something Platform Specific
    ${platform} =    Get Environment Variable    OS
    Log    Running on platform: ${platform}
    Run Keyword If    '${platform}' == 'Windows'    Window
s Specific Keyword
    Run Keyword If    '${platform}' == 'Linux'      Linux
Specific Keyword
    Run Keyword If    '${platform}' == 'MyOS'       MacOS
Specific Keyword

Windows Specific Keyword
    Log    Executing Windows-specific functionality
    # Add your Windows-specific test steps here

Linux Specific Keyword
```

```
    Log    Executing Linux-specific functionality
    # Add your Linux-specific test steps here

MacOS Specific Keyword
    Log    Executing MacOS-specific functionality
    # Add your MacOS-specific test steps here
```

In this example:

- The test case named `Open Browser Test` opens a web browser using the SeleniumLibrary's `Open Browser` keyword.
- The test case then calls the `Do Something Platform Specific` keyword.
- Inside the `Do Something Platform Specific` keyword, the operating system (Windows, Linux, or MacOS) is determined using the `Get Environment Variable` keyword.
- Based on the detected platform, different platform-specific keywords (`Windows Specific Keyword`, `Linux Specific Keyword`, and `MacOS Specific Keyword`) are executed.

This example demonstrates how Robot Framework allows you to write platform-independent test cases and handle platform-specific functionalities based on the operating system on which the tests are running.

---

4. **Data-Driven Testing:** It supports data-driven testing, allowing users to parameterize test cases and run them with different sets of input data.

> 💡 Data-Driven Testing in Robot Framework allows you to parameterize your test cases and run them with different sets of input data. Here's an example of a simple data-driven test case using the Robot Framework:

```
*** Settings ***
Library    SeleniumLibrary
```

```
*** Variables ***
${BROWSER}    Chrome
${URL}        <http://www.amazon.com>

*** Test Cases ***
Data-Driven Test
    [Template]    Test With Data
    Username    Password
    user1       pass1
    user2       pass2
    user3       pass3

*** Keywords ***
Test With Data
    [Arguments]    ${username}    ${password}
    Open Browser    ${URL}    ${BROWSER}
    Input Text      id=username    ${username}
    Input Text      id=password    ${password}
    Click Button    xpath=//button[@type='submit']
    # Add your validation steps here
    Close Browser
```

In this example:

- The `Data-Driven Test` test case is defined as a template with the `[Template]` keyword.
- The `Test With Data` keyword is defined to take two arguments, `${username}` and `${password}`.
- The actual test cases are then written under the `Data-Driven Test` section, providing different sets of data for each run.

When you run this test case, Robot Framework will execute the `Test With Data` keyword for each set of input data provided under the `Data-Driven Test` section. This allows you to reuse the same test logic with different input values.

You can add validation steps or assertions within the `Test With Data` keyword based on your specific testing requirements.

5. **Built-In Test Libraries:** Robot Framework comes with built-in libraries for common tasks, such as SeleniumLibrary for web testing, DatabaseLibrary for database testing, and RequestsLibrary for working with HTTP requests.

> 💡 Here's an example of using a built-in test library, specifically the `BuiltIn` library, in a Robot Framework test case:

```
*** Settings ***
Library    BuiltIn

*** Test Cases ***
Example Test Case
    Log    This is a log message    # Logging a message
    ${result}    Evaluate    2 + 2    # Using the Evaluate
keyword
    Should Be Equal As Numbers    ${result}    4    # Asse
rting equality
    ${random_string}    Generate Random String    8    # G
enerating a random string
    Log    Random String: ${random_string}
```

In this example:

- The `BuiltIn` library is imported with the `Library` setting.
- The `Log` keyword is used to print a log message to the console.
- The `Evaluate` keyword is used to perform a mathematical calculation and store the result in the `${result}` variable.
- The `Should Be Equal As Numbers` keyword is used to assert that the result is equal to 4.
- The `Generate Random String` keyword is used to generate a random string of length 8, and the result is stored in the `${random_string}` variable.

- Another log message is printed, displaying the randomly generated string.

This is just a simple example, and the `BuiltIn` library provides many more keywords for various purposes, such as string manipulation, list operations, variable handling, and more. You can explore the Robot Framework BuiltIn library documentation for a comprehensive list of keywords and their usage.

---

6. **Parallel Execution:** Test cases can be executed in parallel, improving the efficiency of test execution.

> 💡 Parallel execution in Robot Framework can be achieved using the `pabot` (Parallel Robot) tool, which is designed for running test cases in parallel. This example is, how you can structure your project and run tests in parallel using `pabot`.

Assuming you have a directory structure like this:

```
project_directory/
|-- tests/
|   |-- test_suite_1.robot
|   |-- test_suite_2.robot
|-- results/
```

How you can run test suites in parallel:

1. **Install `pabot`:**
   - You need to install the `pabot` tool. You can install it using the following command:

     ```
     pip install robotframework-pabot
     ```

2. **Run Tests in Parallel:**
   - Use `pabot` to run your test suites in parallel. For example, if you have two test suites (`test_suite_1.robot` and `test_suite_2.robot`), you can run them in

parallel as follows:

```
pabot --outputdir results tests/test_suite_1.robot t
ests/test_suite_2.robot
```

This command will execute the test suites concurrently and generate output files in the `results` directory.

3. **View Results:**
   - After the execution is complete, you can view the results in the `results` directory. Each test suite will have its own log and report files.

Here is a simple example of what a test suite (`test_suite_1.robot`) might look like:

```
*** Test Cases ***
Test Case 1
    [Documentation]    This is the first test case
    Log    Running Test Case 1
    # Your test steps go here

Test Case 2
    [Documentation]    This is the second test case
    Log    Running Test Case 2
    # Your test steps go here
```

You can have similar test cases in `test_suite_2.robot`.

The `pabot` tool helps parallelize the test execution, and it is suitable for scenarios where you want to run multiple test suites or test cases concurrently. Adjust the directory paths and test suite names according to your project structure.

---

7. **HTML and XML Reporting:** Robot Framework generates detailed HTML and XML reports, providing insights into test execution results, including pass/fail status, log messages, and screenshots.

💡 Below is an example of a simple Robot Framework test case along with HTML and XML reporting:

```
*** Settings ***
Documentation   This is a sample test case.
Library         SeleniumLibrary

*** Variables ***
${URL}          <http://www.amazon.com>
${Username}     myusername
${Password}     mypassword

*** Test Cases ***
Example Test Case
    [Documentation]    A simple test case to log in to a w
ebsite.
    Open Browser    ${URL}    chrome
    Input Text      id=username    ${Username}
    Input Text      id=password    ${Password}
    Click Button    xpath=//button[@type='submit']
    Close Browser

*** Keywords ***
Custom Keyword
    Log    This is a custom keyword.

*** Test Teardown ***
    Custom Keyword
```

Now, let's execute the test case and generate HTML and XML reports. In the terminal or command prompt, run the following command:

```
robot --outputdir results your_test_file.robot
```

Replace `your_test_file.robot` with the actual name of your test file. This command will generate HTML and XML reports in the specified output directory (`results` in this example).

After running the command, you should see HTML and XML reports in the specified output directory. Open the HTML report in a web browser to view detailed information about the test execution.

The HTML report typically includes information such as test case names, pass/fail status, log messages, and screenshots (if any). The XML report is machine-readable and can be used for further analysis or integration with other tools.

---

8. **Integration with Continuous Integration (CI) Tools:** It can be integrated with CI tools such as Jenkins, allowing for automated and continuous testing.

> 💡 Below is an example of how you can integrate Robot Framework tests with a Continuous Integration (CI) tool. In this case, I'll provide an example using Jenkins, a popular CI/CD tool.

Let's assume you have a Jenkins job set up to run your Robot Framework tests. Here's a basic Jenkinsfile (Pipeline script) example:

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                // Checkout your source code from version
control (e.g., Git)
                checkout scm
            }
        }

        stage('Install Dependencies') {
```

```
            steps {
                // Install Python and required dependencie
s
                sh 'pip install -r requirements.txt'
            }
        }

        stage('Run Robot Framework Tests') {
            steps {
                // Run Robot Framework tests
                sh 'robot path/to/your/tests'
            }
        }
    }

    post {
        always {
            // Publish Robot Framework test results
            robot publisher: [
                outputPath: 'output.xml',
                disableArchiveOutput: false,
                passThreshold: 90,
                unstableThreshold: 80,
                otherFiles: '**/*.png'
            ]
        }
    }
}
```

Explanation:

- The `Checkout` stage checks out your source code from your version control system (e.g., Git).
- The `Install Dependencies` stage installs Python and any required dependencies specified in your `requirements.txt` file.

- The `Run Robot Framework Tests` stage executes your Robot Framework tests using the `robot` command.
- The `post` section includes a step to publish Robot Framework test results. The `output.xml` file is generated by Robot Framework and contains test execution results. You can customize the thresholds for pass, unstable, and fail conditions.

Make sure to adapt this script according to your project's specific structure, dependencies, and requirements. Also, ensure that Jenkins has the necessary plugins installed to support Robot Framework integration. The `Jenkinsfile` is typically stored in your version control system alongside your source code, and Jenkins will use it to configure and execute your build pipeline.

I hope this helps you better understand and use Robot Framework. See you!