Robot Framework + Selenium automation basics

Overview

Robot Framework is a keyword-driven framework used for test case automation. It's implemented in Python, but it can also run on Jython (Python for the Java platform) or IronPython (Python for .NET).

Tests can be written in plain English, just as a text file (actually, the .robot extension can be used, but these are text files as well).

Robot Framework is a generic framework, so it relies on Selenium (the actual tool that allows controlling web browsers) to run tests for web applications. But it can also be extended with other libraries, like Android Library, Database Library, Appium Library, etc. so it's not limited to just web application testing.

Custom Robot Framework keywords can be created by writing higher level keywords (which don't require programming expertise) or by implementing new keywords (which do need coding abilities).

Test data

Tables

Robot Framework supports a variety of formats (like plain text, TSV or HTML), but only plain text will be addressed here.

In plain text files (or .robot files, which is but an alternative extension for plain text), tests are written with a case-insensitive, tabular syntax where columns are separated by two spaces or a pipe character surrounded with spaces (|). Tabs will be converted to two spaces in plain text files, and extra spaces will be ignored (as long as there are at least two spaces). And, although not enforced, the Gherkin syntax of "Given - When - Then" can also be used to write tests, but this is optional.

Robot Framework uses a table structure, where a table name must be preceeded by one or more asterisks and two spaces separate columns (leading and trailing whitespace in cells are automatically removed). It recognizes four types of tables:

- Settings: to import libraries and resources or to define metadata.
- Test Cases: where tests will be listed.
- Variables: to hold variable definitions.
- Keywords: where higher-level keywords will be defined.

If a column needs to be empty, then this should be indicated, either by a single backslash ("\") or by the ${EMPTY} variable.

Similarly, the backslash can be used to escape other special characters or sequences. For example, \\ means a single backslash (as in, C:\\automation), \n means a new line, \r means a carriage return and \t means a tab character, to name a few escape sequences.

It's also important to keep in mind that plain text files are always expected to use UTF-8 or ASCII encoding, so if there are characters that use a different encoding in the tests, they might fail.

Settings

This is going to be where libraries and other information related to tests will be defined.

A test suite (namely, a file containing tests) or a test directory (a directory containing test suites) can have an initialization file, which must be named __init__.extension (where extension needs to be a valid extension, such as .robot), to hold specific settings and variables. But it's important to be aware that variables and keywords defined or imported in initialization files will not be available in the lower level test suites. This is why there are also "resource files", which can be imported to test suites using the Resource setting in the Settings table.

The initialization file is where we'd put our suite or test setup/tear down actions (what must be done before a suite or test starts and what must be done after its execution finishes) or test timeouts.

Libraries

Since tests are written using keywords, these keywords must be available somewhere: it could be standard Robot Framework libraries, external libraries or even a custom library created by the user. Robot Framework provides a built-in library which is automatically imported, containing generic keywords that are normally used in test cases. And for web application testing, SeleniumLibrary is a good option to allow interaction with the browser.

Before being able to use a library, it must be imported in a "Settings" table, by using the "Library" keyword. As an example:

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

But, when using SeleniumLibrary, it doesn't work its wonders by itself: it needs a "driver" to be able to control a specific browser. These drivers are simple exe files that can be downloaded from Selenium's download page at http://docs.seleniumhq.org/download/ and imported into an automation project.

Test cases

All test cases contained in a file will be considered a "test suite", so initialization files in the same test suite will affect them all.

A test case will have a name and steps (where steps are indented, to follow the table formatting required by RF). They can also have optional documentation, indicated by the setting [Documentation], that will be ignored when compiling.

Also, some keywords have arguments (some of them are required, some are optional). This will be indicated by the library documentation. For instance, if we want to open a browser window then we can use the "Open Browser" keyword provided by SeleniumLibrary, which must take at least two arguments: the url to navigate to, and which browser we want to open. These arguments must be in a different table column, so at least two spaces are required to separate them..

An example test case could be as follows:

```
*** Test Cases ***
Navigate to Homepage
  [Documentation] Verifies the home page can be accessed.
  Open Browser                  http://www.google.com  chrome
  Maximize Browser Window
  Location Should Contain       http://www.google.com
```

Variables

When a test suite starts to grow and we need to manage a great deal of test cases and keywords (which also have to be maintained and updated over time), it's better to use variables instead of hard-coding values in arguments. This is where the "Variables" table comes in handy. Also, sometimes a keyword returns a value that then has to be used somewhere else, so this can be stored in a variable within the keyword definition itself.

In the above examples, the browser name and the page url could be stored in variables and then used as arguments.

Variables are referenced as ${VARIABLE_NAME} and their values separated by spaces or tabs. Also, they can have spaces in their names and, as mentioned before, they are case-insensitive.

```
*** Variables ***
${BROWSER}                      chrome
${HOMEPAGE URL}                 http://www.google.com
```

So now our test can be re-written as:

```
Navigate to Homepage
  [Documentation] Verifies the home page can be accessed.
  Open Browser                  ${HOMEPAGE URL}  ${BROWSER}
  Maximize Browser Window
  Location Should Contain       ${HOMEPAGE URL}
```

A variable name will be replaced by its string representation whenever it's found in a keyword or test. When string variables are used to evaluate expressions in keywords like "*Evaluate"*, "*Run Keyword If"* and "*Should Be True"*, they need to be quoted (single or double), and if they can contain newlines, they must be triple quoted, since these evaluations are made using Python functionality, so in the end it all comes down to comparing string literals. As an example, this would verify the URL in browser is exactly as we expect:

```
  ${page url}                   Get Location
  Should Be True                '${page url}'=='${HOMEPAGE URL}'
```

In this example, using the keyword "Location Should Be" would be the best way to go, but it's done this way to show how string comparison works. Also, notice how a variable was used to store the result returned by "Get Location" and then used in the next keyword execution.

The case is not the same when the variable contains a value that should not be interpreted as a string. As an example, let's say we have a page with a video and we have to validate that the video is paused at some point. For this, we can have javascript return the state of the video, by using the

"Execute Javascript" keyword, which will return a result that will be stored in a variable and then used to be evaluated as True or False. Example:

```
${result}                          Execute JavaScript return
document.getElementById("myVideoId").paused
Should Be True                     ${result} == False
```

Finding elements: locators

A web page can have a wide variety of elements. When a page loads, the browser generates a Document Object Model ("DOM") for it, which is a tree of nested elements that are present in that page. We should be able to identify these elements whenever they're involved in a test (e.g., to verify a specific element is visible, or to interact with it). This is done through a variety of methods and might require just a little bit of HTML/CSS/XPath knowledge (just enough to understand what's going on).

Robot Framework provides some different locator strategies, which can be explicit or implicit. Some attributes are common to all keywords, but some keywords can also find elements using additional attributes. As an example, the keyword "*Element Should Be Visible*" needs an argument that indicates which element exactly it is that should be visible when the test runs. This kind of argument is normally called a "locator".

It's good practice to store these locators in variables instead of just entering a hardcoded locator as a keyword argument. This also allows better maintainability should a locator change if the web application being tested changes.

These are the default locators listed by SeleniumLibrary:

- id: Element id
- name: Name attribute
- identifier: Either id or name
- class: Element class
- tag: Tag name
- xpath: XPath expression.
- css: CSS selector.
- dom: DOM expression
- link: Exact text a link has
- partial link: Partial link text
- sizzle: Sizzle selector provided by jQuery.
- jquery: Same as the above.
- default: Keyword specific default behavior.

Whatever the strategy used, we should make sure the element is correctly identified and that our locator matches that element only. That means the locators *must* be reliable. For example, if we use the class of an element, since classes can be applied to more than one element in the DOM, this could (and probably will) lead to identifying more than one element with one locator. On the other hand, since ids are unique (or should be) and independent of the element type and location in the DOM, an id might be a good choice to pinpoint an element.

It's not always easy to find a good balance between flexibility (to allow valid small changes in the DOM) and bug detection. If a locator is too strict, any change in the DOM will cause a failed test

when there might be no bug at all; but if it's too flexible, then it might end matching more than one element. Usually, ID, Name, CSS and XPath (in that order) are the preferred strategies to locate elements. However, XPath is the slowest and less flexible way to do it, so it should be used with caution (XPath can also get very complicated and it would take some time to master it). Writing reliable, readable, flexible locators is an art, and there are many good articles about it. Mostly, this knowledge will be acquired with experience.

When trying to identify an element, a good way is to open the web page and use some browser tool like "Inspect Element" (by right-clicking on the element we want to locate). The HTML code will be displayed and we'll be able to see what kind of element it is (a paragraph, a header, an image, etc.), if there are any available IDs that we can use, or if we must use some other strategy.

If there are no unique IDs or names that can be used, then we should try to build a CSS selector. Usually, the best way is to build the locator from that element up, looking at the element type first, then at the element one level above that contains it, and so on. Also, browser dev tools normally provide a way to just right-click on an element and copy the CSS selector (or the XPath). But preferrably we should know what this selector means in order to be able to maintain or improve it (doing something without understanding what it implies is never a good thing).

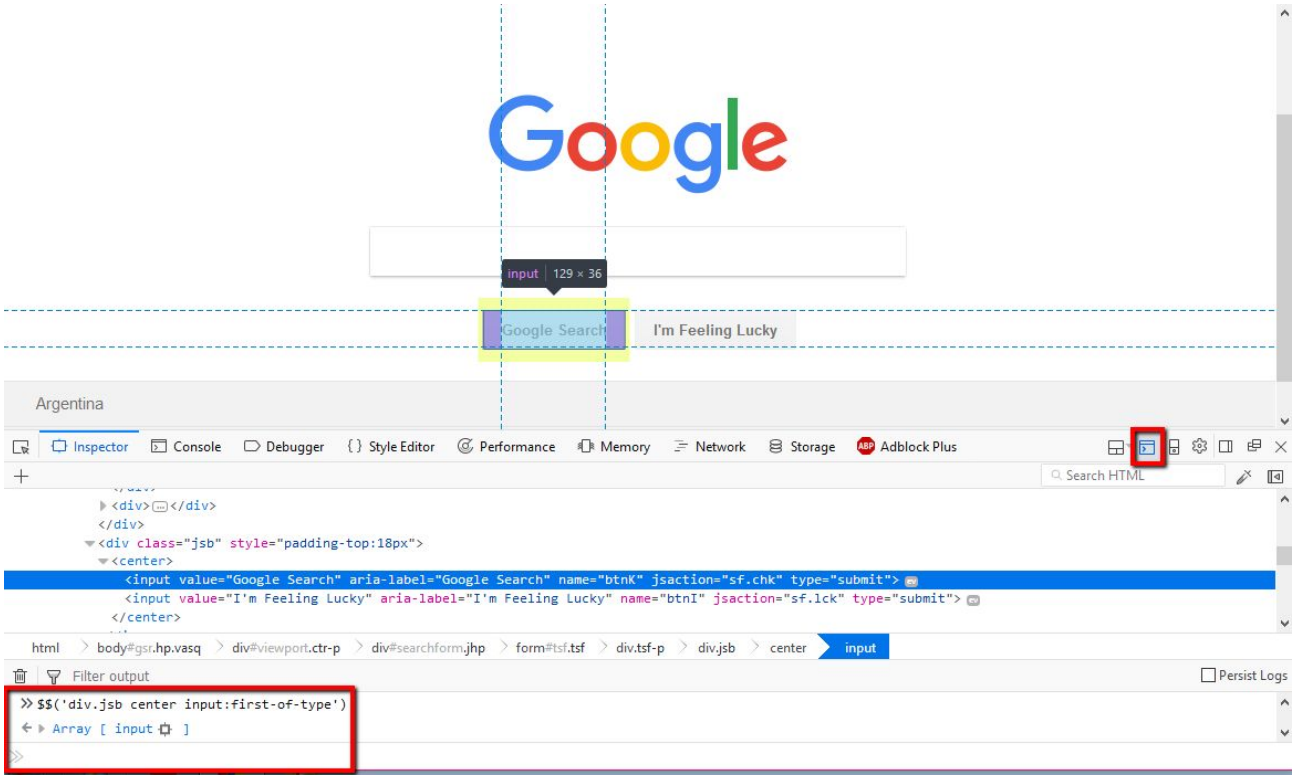
As an example, let's say we want to find the "Google Search" button in Google's home page. So, upon element inspection (right click and "inspect element" or Ctrl+Shift+C in Firefox or Chrome), we can see that this is an "input" element type that's contained in a "center" element inside a "div" within the DOM, and this "div" element has a class called "jsb". Also, this is not the only input element within this div (since there's another search button), so we must select only the one we care about. So the following CSS selector will match the desired button: `div.jsb center input:first-child`

This is where some CSS knowledge will be helpful, as we must indicate that something is a class applied to some element (with the '.' dot), or that something is a nested object (with the ' ' space character), or that we want to select only the first of the same kind elements that the div contains (with the :first-child selector). For this purpose it's always good to keep in mind CSS selectors (a good reference can be found at www.w3schools.com/cssref/css_selectors.asp).

To test that the selector we've come up with is right, we can use the browser's web console (pressing ESC in Firefox or Chrome, or the web console button in the web inspector). In this console, type the CSS locator wrapped by `$$('')` like this:

```
$$('div.jsb center input:first-child')
```

(As a reference, web console helpers in Firefox are listed here https://developer.mozilla.org/docs/Tools/Web_Console/Helpers)



An array with only one element will be returned. Clicking on that element we can verify it's the button we were trying to locate. If the array size is bigger than 1, then the locator is matching more than one element and we should try to find a way to identify just the one we want.

So now we're ready to store this css selector in a variable that we'll later use in our tests:

```
${GOOGLE SEARCH BUTTON}          css=div.jsb center input:first-child
```

Similarly, if XPath was used instead of a CSS selector, we would need to tell Robot Framework that the value being stored in the variable should be treated as such by replacing "css=" with "xpath="). For an XPath reference: https://www.w3schools.com/xml/xpath_syntax.asp.

It's also a good practice not to place the locators with tests, all in one file. It's always better if locators live in a different file that can be imported by tests with the "Resource" keyword, within the "Settings" table.

Element attributes

There are times when the presence of a specific attribute, class, id, etc. in an element needs to be verified. There will be keywords that allow us to add selectors in their arguments.

As an example, consider the case of a menu where the currently active element is indicated by having a class named "active" at some point of execution (but not always: only when it is indeed active), so we need to check that such element exists (using a locator) and also that at some point (e.g., after clicking on a button) gets the class "active" applied to it. So we could use a keyword like "*Element Should Be Visible*" or "*Wait Until Page Contains Element*" and then place the class by adding it to the element, preceeded by a dot (class selector):

```
Wait Until Page Contains Element      ${ELEMENT LOCATOR}.active
```

To tidy things up a little more, the class name could be stored in a variable, like ${CLASS ACTIVE} which can be then used in the keyword:

```
Wait Until Page Contains Element      ${ELEMENT LOCATOR}.${CLASS ACTIVE}
```

In a similar fashion, an attribute from an object can be used by adding a "@" character, like this: ${SOME ELEMENT}@attribute.

Higher-level keywords

By using keywords already available in the imported libraries, a user can create their own keywords. This is desirable as a good practice, to make tests more readable and to avoid having arguments, variables and other not-so-readable code in a test case. This way, even stakeholders could write tests in an easy way. So, wrapping all of these things in a high-level keyword would be the best way to go (and it's always better to follow good practices from the beginning, instead of just writing tests without proper structure to end up having a poor quality test suite).

Taking a look at the test case example above it's fairly easy to recognize that "Open Browser" with its arguments will open Chrome and navigate to www.google.com, but there are cases where some more intricate things must be passed as arguments (like a locator, or when javascript code is executed).

So let's consider a test case like this:

```
*** Test Cases ***
Main Items Displayed In Homepage
  [Documentation] Verifies the logo and search components are displayed.
  Open Google Home Page
  Main Google Logo Is Displayed
  Search Elements Are Displayed
```

In this test case all three keywords used are custom keywords created at a higher-level, using the "Keywords" table, as follows:

```
*** Keywords ***
Open Home Page
  Open Browser                    ${HOMEPAGE URL}  ${BROWSER}
  Maximize Browser Window
  Location Should Contain         ${HOMEPAGE URL}
  Title Should Be                 ${HOMEPAGE TITLE}

Main Google Logo Shows Up
  Wait Until Element Is Visible   ${GOOGLE LOGO}

Search Elements Are Displayed
  Wait Until Element Is Visible   ${SEARCH BOX}
  Wait Until Element Is Visible   ${GOOGLE SEARCH BUTTON}
  Wait Until Element Is Visible   ${FEELING LUCKY BUTTON}
```

For a good structure, test cases and higher-level keywords should be placed in different files. Using the “Resource” keyword within the “Settings” table, a file containing tests can use these higher-level keywords:

```
*** Settings ***
Resource                        Resources.robot
Resource                        PageObjects.robot
```

Notice that the file containing variables and locators (explained in the previous section) is also being imported here. This way, tests can use the keywords and variables in those two files as if they were all included in the same tests file.

Abstracting the logic

Sometimes a simple high-level keyword is not enough and we need to abstract some of the logic in new keywords that will then be executed from within a high-level keyword (which, in turn, will be executed by a test). So we can also create keywords that use parameters and return something.

Let’s say, as an example, that we have to do some string manipulation (using the “String” library from Robot Framework at http://robotframework.org/robotframework/latest/libraries/String.html) on an element to obtain the actual string we need to use (for example, trim leading and trailing spaces). We could then create a new keyword in the same “Resources” file where the high-level keywords live, and set it to get the unmodified string as an argument and return it after it’s been manipulated. For this, we will use [Arguments] right before the keyword definition and [Return] at the end:

```
Get Trimmed String
  [Documentation]               Trims leading and trailing spaces.
  [Arguments]                   ${unmodified string}
  ${new string}                 Strip String  ${unmodified string}
  [Return]                      ${new string}
```

So, whenever we use “Get Trimmed String” along with a string as an argument, it will return a new string so we should store that value in a variable to be used.

Another usual case would be when an action causes the page to refresh and before making any further checks, we need to make sure the page has actually been refreshed and it’s not showing outdated elements. A simple approach could be to create a keyword that waits for the element to be updated and then use this keyword combined with “Wait Until Keyword Succeeds” (from the “BuiltIn” library):

```
Verify Page Is Updated
  [Documentation]               Validates the element has been updated.
  [Arguments]                   ${expected value}
  ${element value}              Get Element Attribute  ${ELEMENT}@attr
  Should Be True                '${element value}'=='${expected value}'

Update Page
  [Documentation]               Click button and refresh page.
  Click Button                  ${BUTTON LOCATOR}
```

```
  Wait Until Keyword Succeeds    10  3s  Verify Page Is Updated  ${SOME
EXPECTED VALUE}
```

This way, the “Update Page” will click on some button that will cause the page to refresh (e.g.: a button that changes the application language) and then it will wait until the “Verify Page Is Updated” keyword succeeds, by executing it 10 times with a 3 second wait between one retry and the other. This helps us make sure the “Update Page” keyword didn’t fail because it checked for the expected value before the page had time to refresh. If it fails it will be because the value was never found after 10 retries.

Settings, keywords, tests, variables… how to organize it all?

There is not a unique way to organize a project, but there are good and bad practices. Yes, we can throw things in a file just so the project compiles and executes without issue, but this doesn’t mean it’s well done. Good practices are intended to make a project more readable and maintainable, so they should be followed as much as possible.

For instance, Robot Framework allows to import test libraries in test case files, resource files and test suite initialization files, and all keywords in the imported library will be available in that file (except for resource files, as their keywords will also be available in other files using them). However, it’s always best to keep some sort of well-defined structure. A proposed structure could be the following:

- Any custom keyword implementation should be in a library outside the whole Robot Framework structure Each keyword will be implemented by a method and the method name will define how the keyword will be called. Example: a keyword to set up the browser driver path.

- A file for resources to be globally used by all test suites: this is where we’ll import the libraries that are to be commonly used by all (or most) test suites. Let’s call it **CommonResources.robot** or something along those lines. This file will only contain a “Settings” table to import libraries and nothing else.

  Example: SeleniumLibrary and our custom implementations will be imported here using the “Library” keyword:

- One directory per test suite: each feature or part of the application under test should have its own test suite; even better: its own directory in which the test suite lives. For a website, this could mean a directory per each page (home page, sign up page, etc.). And better have each directory contain one test suite related to the feature the tests will be about. *Example: a “home_page” directory to test the website home page would include one file with several test cases that define the “home page” test suite.*

    - One initialization file per test suite: each test suite (directory) will have its own initialization file, that will get things ready to execute tests, as each feature might require different kind of setups. This is the **__init__.robot** file (file name should be exactly that) which will have a “Settings” table as well, for suite or test setup and teardown. This file gets automatically recognized and imported so it doesn’t need to be explicitly done. *Example: Setup could be to set the browser driver path and opening the browser, and tear down could be to close the browser.*

- One objects file per test suite (Page Object Model): variables that are local to a test suite, locators and other representation of objects will be placed in this file, to keep them separated from keyword definitions and test cases themselves. Let's call this one **PageObjects.robot**, and make it so it has a "Variables" table only. *Example: all texts from the test suite are stored in variables, as well as locators.*
- One resources file per test suite: this is where high-level keywords are to be defined, so it will have a "Keywords" table. Also, since a test suite could use some specific libraries that other test suites don't, this file could also have a "Settings" table to import them. Let's call this file **Resources.robot**. *Example: some keyword needs to manipulate strings, so we import the String library in the "Settings" table. Then the "Keywords" table will contain keyword implementation, using variables defined in PageObjects.robot and the variables that are local to each keyword definition (for instance, to store the return value of a keyword).*
- One tests file per test suite: this is where the actual test cases will live. It's also where the other files will be imported, so it will have a "Settings" table as well as a "Test Cases" table. Since there will be one of these files in each test suite, the test suite name could be left to be defined by the directory name and this file can be called just **Tests.robot**. *Example:* "Settings" table will import (using the "Resource" keyword) CommonResources.robot, Resources.robot and PageObjects.robot (remember __init__ gets automatically imported). Then the "Test Cases" table will have all the needed test cases, each one of them using the high-level keywords defined in Resources.robot.

One last note

This was just an overview of what Robot Framework can do for automators in order to keep their task simple and flexible. There's more to it, and probably when starting to create your own tests you'll run into many singularities that come with each specific project. But do not fret: in those obscure times you'll probably be able to solve most problems by reading through:

- The User Guide: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html
- Standard libraries (especially "built-in") documentation: http://robotframework.org/robotframework/
- Selenium library documentation: http://robotframework.org/SeleniumLibrary/SeleniumLibrary.html