



Selenium Open Source Test Automation Framework Tips and Tricks

Version 0.1

September 2009

DISCLAIMER

Verbatim copying and distribution of this entire article is permitted worldwide, without royalty, in any medium, provided this notice is preserved.

TABLE OF CONTENTS

| | | |
|-----------|--|----------|
| 1. | INTRODUCTION | 3 |
| 1.1. | Purpose | 3 |
| 1.2. | Scope | 3 |
| 2. | CONDITIONS | 4 |
| 2.1. | Problem Scenarios | 4 |
| 2.1.1. | Handling nested conditions..... | 4 |
| 2.1.2. | Changing start rows and end rows during script maintenance | 4 |
| 3. | RECOVERY | 6 |
| 4. | PUTS FUNCTION | 7 |
| 5. | LOOPS | 8 |
| 6. | 'VERIFYSELECT' KEYWORD | 9 |

1. Introduction

1.1. Purpose

This “Tips & Tricks” document provides an overview of handling frequently encountered scripting problems and some valuable dos and don'ts to maximize the productivity of the Open Source Test Automation Framework

1.2. Scope

This document is just for understanding and reference. There might be a need to tweak the solutions provided before implementation, depending on the actual scenario in hand.



This document requires prior knowledge and working experience with the Open Source Test Automation Framework. For understanding the keywords and syntaxes, please refer to the “Scripting Standards” document.

2. Conditions

Conditions are basically implemented when we need a specific code to be executed only when a condition is satisfied. For ex: We would want a 'Pass' report to be sent to the log if a variable 'A' is equal to variable 'B', and we would want a 'Fail' report otherwise.

Syntax - r |condition|<Var A>;<condition>;<Var B>|Start ROW;End ROW

2.1. Problem Scenarios

At times you may encounter the following two problem scenarios during the implementation of conditions using the Open Source Test Automation Framework.

2.1.1. Handling nested conditions

At times the flow might not be restricted to just one condition but instead require multiple conditions before code is executed. In other words, at times nested conditions might be required. In such cases, the second condition should be placed after the first condition and should fall within the range of the first condition.

The following example in 'Table 1' shows a simple way to do this. The first condition at line number 10 has the start and end rows as 11 and 13 respectively. The second condition is placed at line number 11 and has the start row as line number 12 where the third condition is placed. Therefore, if the first condition is satisfied only then the control will go to the second condition, and if the second condition is satisfied, the control would go to the third condition. Thus, during actual execution all three conditions would have to be satisfied before the code at line number 13 is executed.

| Line | A | B | C | D | Remarks |
|------|---|-----------|---------------------------------------|-------|------------------|
| 10 | r | Condition | VarA;equals;VarB | 11;13 | First condition |
| 11 | r | Condition | VarB;equals;VarC | 12;13 | Second condition |
| 12 | r | Condition | VarD;greaterthan;VarE | 13;13 | Third condition |
| 13 | r | Report | Pass;Expected result :: Actual Result | | |

Table 1: Nested Conditions

2.1.2. Changing start rows and end rows during script maintenance

One of the major problems that often go unnoticed during script development is the use of hard-coded values to specify the start and end rows of any condition. The row numbers pose a significant problem

during maintenance when changes are implemented in the script that involves the addition and deletion of rows. Whenever a row is added or , the start and end rows of a condition no longer hold true and thus need to be updated. This results in wasted time when the number of conditions is high.

To solve this problem, simple Excel functions can be used to specify the start and end rows. If the **ROW()** function available in Excel is used to specify the start and end rows, the line numbers get updated automatically whenever a new row is added or deleted in the script.

Example – In the script below in ‘**Table 2,**’ if we look at the ‘1st Condition’ the formula used to specify the start and end rows (i.e., 11;13 is “=ROW()+1&”;&ROW(A13)”)

The ROW() function returns the row number of a particular reference cell. If no parameter is passed, then it returns the current row number. The formula used can be split into three parts for a better understanding:

1. **ROW()+1**: Generally all the conditions have the start row as the very next line. Here we are fetching the current row by ‘ROW()’ and then adding ‘1’ to get the next row number.
2. **&”;&**: Since we have a semicolon ‘;’ as the separator between the start and end rows we have to concatenate it with the help of ‘&’ operator in Excel.
3. **ROW(A13)**: Here we are using a reference to get the end row. In the example shown, row (A13) returns the value ‘13.’ In case a line is inserted between rows 10 and 11 and 11 and 12, the end row would automatically be incremented by 1.

| Line | A | B | C | D | (Formula to be used) |
|------|---|-----------|----------------------------|-------|----------------------|
| 10 | r | Condition | VarA;equals;Va rB | 11;13 | =ROW()+1&”;&ROW(A13) |
| 11 | r | Condition | VarB;equals;Va rC | 12;13 | |
| 12 | r | Condition | VarD;greaterth an;VarE | 13;13 | |
| 13 | r | Report | Pass; Act Res:: Exp Res | | |

Table 2: Usage of Excel’s ROW Function



This formula works with the help of references. Hence, the end row to which there is a specific reference should not be deleted. It can be modified and any number of rows can be deleted or added in between, but the end row should not be deleted.

3. Recovery

During the execution of the keyword script, when there is a perform keyword and if the particular object is not identified, then the execution will not continue. To handle this issue, we have a method defined in the `functionlibrary.rb`. Whenever there is a perform keyword, before performing the action defined in the keyword there will be a check for the object if it exists. If the object does not exist then this method `recovery()` will be called. As per the current implementation, the execution will continue to the next line without exiting the run. However, the user has the privilege to modify the method to his or her requirement so as to continue the run, stop the execution, or bring the system to the base state.

4. puts function

Maintenance is an integral part of the automation lifecycle. Once we have developed the scripts, we have to maintain them to keep the script updated and in sync with the application.

The “puts” functionality proves to be very handy for maintaining scripts. You can use the “puts” keyword in the Ruby code to invoke the value of the dynamic variables during the execution of the script. This, to some extent, helps in pinpointing the error points during execution.

The example below shows the code and usage of the puts function,

Example 1:

```
def method()  
  puts "Entered Method"  
  @selenium.click "link=Back"  
  puts "Completed Method"  
end
```

In this instance, there will be a statement printed in the console saying "Entered Method" when the execution of this method starts, and there will be one more statement printed when the execution of the method is completed saying "Completed Method."

Example 2:

```
def method()  
  puts "Entered Method"  
  @selenium.click "link=Back"  
  a = '1'  
  puts a  
  puts "Completed Method"  
end
```

This also prints the same statements. But before printing the "Completed Method" statement, the value of the variable 'a' will be printed. This is how we can get the value of dynamic variable in the code.

5. Loops

Loops prove to be a very useful tool when we have to execute code repeatedly for a given set of data. This section describes how to use the looping keywords in the Open Source Test Automation Framework to construct a loop.

Scenario: In this scenario, we have to create a set of users for the AUT. This is a repetitive job. The loop keyword of the Open Source Test Automation Framework has been designed to handle such scenarios.

To use the loop keyword, we have to place the data in a separate data sheet. For example, in Table 6 the names of the users are to be entered in a column called 'NAME'.

| | NAME | B | C |
|---|--------|---|---|
| 1 | User 1 | | |
| 2 | User 2 | | |
| 3 | User 3 | | |

Table 6: Data Table

As shown in the example below, at line number 10 we have the loop keyword that has the start and end rows (11;13) as the argument. Therefore, lines number 11 and 13 would serve as the body of the loop. At line number 11, we are setting the context and in 12 we are entering the user name. Here, the value has to be retrieved from the 'NAME' column of the 'Action 1' sheet. Hence, we have used 'set:dt_NAME'. By default the number of iterations of the loop is defined by the number of start and end rows indicated in row 10, column C. In the current example, the loop would continue three times, one loop for 11, 12, and 13 as indicated by 11;13.

If we want to run it two times, then it could be done by inserting '2' in the 'D' column in line number 10.

| Line | A | B | C | D | Remarks |
|------|---|---------|--------------------|-------------|---|
| 10 | r | Loop | 11;13 | | Loop Keyword |
| 11 | r | Context | Window;Create_User | | |
| 12 | r | Perform | Textbox;UserName | Set:dt_NAME | Setting value from col 'NAME' of 'Action 1' sheet of QTP. |
| 13 | r | Perform | Button;Submit | | |

Table 7: Loops

6. 'VerifySelect' keyword

The VerifySelect keyword is used to verify the items in a 'weblis' or 'combobox'. We can use the 'VerifySelect' keyword along with the 'loop' keyword to verify that the values put in a specified column of the 'Action 1' sheet of the global sheet are a part of the combobox items list.

Scenario: In this scenario, we have a flight booking application and need to check that a given set of 'source' terminals are listed in the 'source' dropdown.

We need to place the given set of 'source' terminal names in the data sheet in a column called 'source' as shown in Table 8.

| | SOURCE | B | C |
|---|-----------|---|---|
| 1 | Frankfurt | | |
| 2 | Mumbai | | |
| 3 | Bangalore | | |
| 4 | New York | | |

Table 8: List of source terminals as input

We have to then write the script as shown below in Table 9 where 'Verify Select' is used in conjunction with the 'loop' keyword.

| Line | A | B | C | D | Remarks |
|------|---|---------|---------------------|----------------------------|--|
| 10 | r | Loop | 11;12 | | Loop Keyword |
| 11 | r | Context | Window;Booking_page | | |
| 12 | r | Perform | Combobox;Source | VerifySelect: dt_SOURCE | Each item in the source column would be verified |

Table 9: VerifySelect keyword

In the above example, we start a loop in line number 10, and in line number 12 we have used the 'VerifySelect' keyword that would verify if the item is present in the list of items of the 'source' combobox. As already mentioned in the 'loops' section, the loop would be executed until all the items listed in the source column of the Action 1 sheet have been checked for.

COPYRIGHT

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details.