QTP – Open Source Test Automation Framework Tips and Tricks

Version 1.0

April 2009

TABLE OF CONTENTS

List of Tables

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 equals to variable 'B' and 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 no. 10 has the start and end rows as 11 and 13 respectively. The second condition is placed at line no. 11 and has the start row as line no 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 the three conditions would have to be satisfied before the code at line no 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 row and end rows during script maintenance

One of the major problems that often goes unnoticed during script development is the usage 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 involve the addition and deletion of rows. Whenever a row is added or deleted 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 are 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.

For ex – 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;VarB	11;13	=ROW()+1&";"&ROW(A13)
11	r	Condition	VarB;equals;VarC	12;13	
12	r	Condition	VarD;greaterthan;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. BaseState

It is very important that the automation scripts developed should not have any dependency on the state the application is in. It should be able to execute from any screen in the application. The importance of this is that when these scripts run in a suite and a script fails, the following script might not find the application in the home screen. In such circumstances, the second script should not fail because it did not find the application in the required state.

The best way to resolve this is to have code in place that would take the application to the main screen before beginning the execution of any script. One of the ways of doing this is to use a Reusable Action and call this Reusable Action at the first and last line of each script. For best results, this code can also be included in the Error Handling functions.

The example below in **'Table 3'** shows the call to BaseState in the first and last lines of the script. The Base State function can be associated with certain parameters if required for additional flexibility.

Example:

Line	A	B	C	D	Remarks
1	r	Callaction	RA_BaseState	Param	**Base State function call.**
2	r	Keyword	Keyword	Keyword	**Script Body**
10	r	Keyword	Keyword	Keyword	
15	r	Callaction	RA_BaseState	Param	**Base State function call.**

Table 3: Base State

4. Message Box

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 "message box" functionality proves to be very handy for maintaining scripts. You can use the "MsgBox" keyword to invoke a message box with the value of the dynamic variables during the execution of the script. This, to some extent, helps in pinpointing the error points during execution.

In the example below in '**Table 4**' at line number 10, 'Text' property of a 'textbox' object with the name 'Name' is stored in a variable 'strValue' for later use at line 12 where it is being checked to be equal to "Smith". During maintenance, we can actually see what is getting stored in the variable '#strValue' by inserting a simple 'msgbox' keyword with '#strValue' as the argument.

Line	A	B	C	D	Remarks
10	r	Storevalue	Textbox;Name	Text:strValue	
11	r	Msgbox	#strValue		**'Msgbox' used.**
12	r	checkcondition	#strValue;equals;Smith	13;13	
13	r	Report	Pass; Act result :: Exp result		

Table 4: Message Box 'msgbox' Keyword

5. Descriptive Programming

Very often we encounter problems related to identification of objects. Many times QTP fails to identify the objects because of a change in some properties of the object. This requires an update to the properties of the object in the Shared/Global Object Repository. When we are in execution mode, these kind of failures do not project the exact picture because the functionality might be working fine but because of a property change in some of the objects the script would fail.

Such situations can be minimized by using descriptive programming in integration with the keyword scripts. Even though this approach won't ensure 100% elimination of such failures, it will significantly reduce the occurrence of such failures with minimal effort.

In the example below in '**Table 5**', line number 10 is for clicking on a button object. However, here we are referencing the object with descriptive programming and not through the shared repository. The main aim of this approach is to pass object properties and their values (which would make the object unique) as parameters. In line no 10 we are referencing an object that has a property "name" the value of which is "Login".

Even if some properties of the Login button change because of code changes, QTP wouldn't face problems in identifying the object as long as the "name" property has the value "Login" and this property is sufficient to make the object unique.

Line	A	B	C	D	Remarks
10	r	Perform	Button;name:=Login	Click	**Desc Prog used**
11	r	Check	Page;title:=ABC	exist	

Table 5: Descriptive Programming

6. 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 one of the columns of the 'Action 1' sheet of the data table. 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: Action 1 Sheet of QTP Data Table

As shown in the example below, at line no 10 we have the loop keyword that has the start and end rows (11;13) as the argument. Therefore, lines no 11 and 13 would serve as the body of the loop. At line no 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 row 10 in 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 for two times then it could be done by inserting '2' in the 'D' column in line no 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

7. ‘VerifySelect’ keyword

The VerifySelect keyword is used to verify the items in a ‘weblist’ 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 ‘Action 1’ sheet of the QTP Data Table 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 conjuction 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 no 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.