Eggplant Monitoring Insights

Script Editing User and Reference Guide

September 2021

Document Control

Proprietary Information

The content of this document is considered proprietary information.

Eggplant gives permission to copy this review for the purposes of disseminating information within its clients or any regulatory agency.

Document Version Control

Data Classification	**Client Confidential**
Client Name	All Eggplant Monitoring Insights Clients
Report Reference	Final
Document Title	Script Editing
Author	Philip Vandenberg
QA and Approval	Paul Bianciardi

Document History

Issue No.	Issue Date	Issued By	Change Description
0.1	15 May 2019	Philip Vandenberg	1st Draft
0.2	20 May 2019	Philip Vandenberg	Technical Review
1.0	22 May 2019	Philip Vandenberg	1st Release
1.1	14 June 2019	Paul Bianciardi	Additional methods Formatting updates
1.2	27 June 2019	Paul Bianciardi	Updated copy
1.3	17 July 2019	Paul Bianciardi	Additional methods
1.4	23 July 2019	Paul Bianciardi	Updated images
1.5	19 February 2020	Priya Sundararajan	Additional methods
1.6	09 September 2021	Mayooraj Murugathasan	Script editing improvements
1.7	23 September 2021	Mayooraj Murugathasan	Updated images and incorporated review feedback

Contents

1 Scope

1.1 Document Scope

This document provides basic user and reference information to support the script editing feature of Eggplant Monitoring Insights as released in Agent 10.

The objective is to provide a background of how the feature can be accessed and used, outline best practices and document the available additional programming functions provided by Eggplant through the scripting interface.

This document does not provide guidance on programming or scripting language, methods and practices except where it is appropriate to do so.

1.2 Document Structure

The document is divided into several sections, describing how to modify scripts and a reference guide for the different functions available to you.

2 Script Editing User Guide

This section provides an overview of script editing mode.

2.1 Accessing the Administration Portal

Access to the script editing functions is gained through the Administration Portal of Monitoring Insights. Figure 1 shows how to access the Administration Portal for a user journey.

Click on the monitor from the main Monitoring Insights portal (where you see all your monitors listed). This will show a menu of options for that monitor. Select the **Monitor Configuration** option:

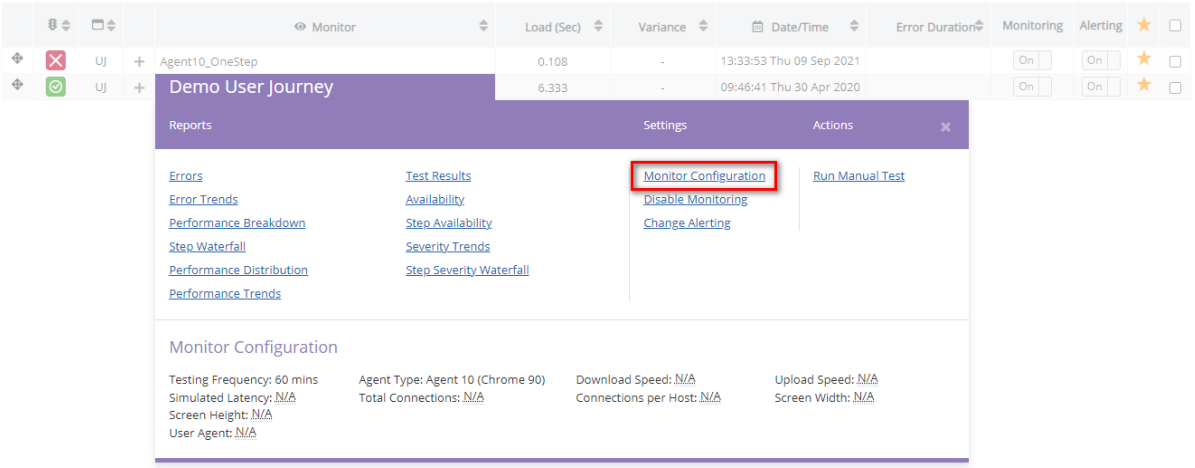

Figure 1: Gaining access to the Administration Portal.

2.2 User Journey Settings

Clicking **Monitor Configuration** will open the Administration Portal in the View/Edit User Journey Settings as shown in Figure 2.

Many administration functions can be performed in this portal to define how the monitor will work and the tests it needs to perform.

Each of the steps of the user journey are listed and access control provides either viewing and/or editing capabilities, depending on your role.

Authorisation must have been granted to enable access to the script editing functions.

If you are authorised to do this, the **Edit Script Code** option will be available in the top panel of the View/Edit User Journey Settings, as shown in Figure 2.

To access the script editor, click the highlighted icon:

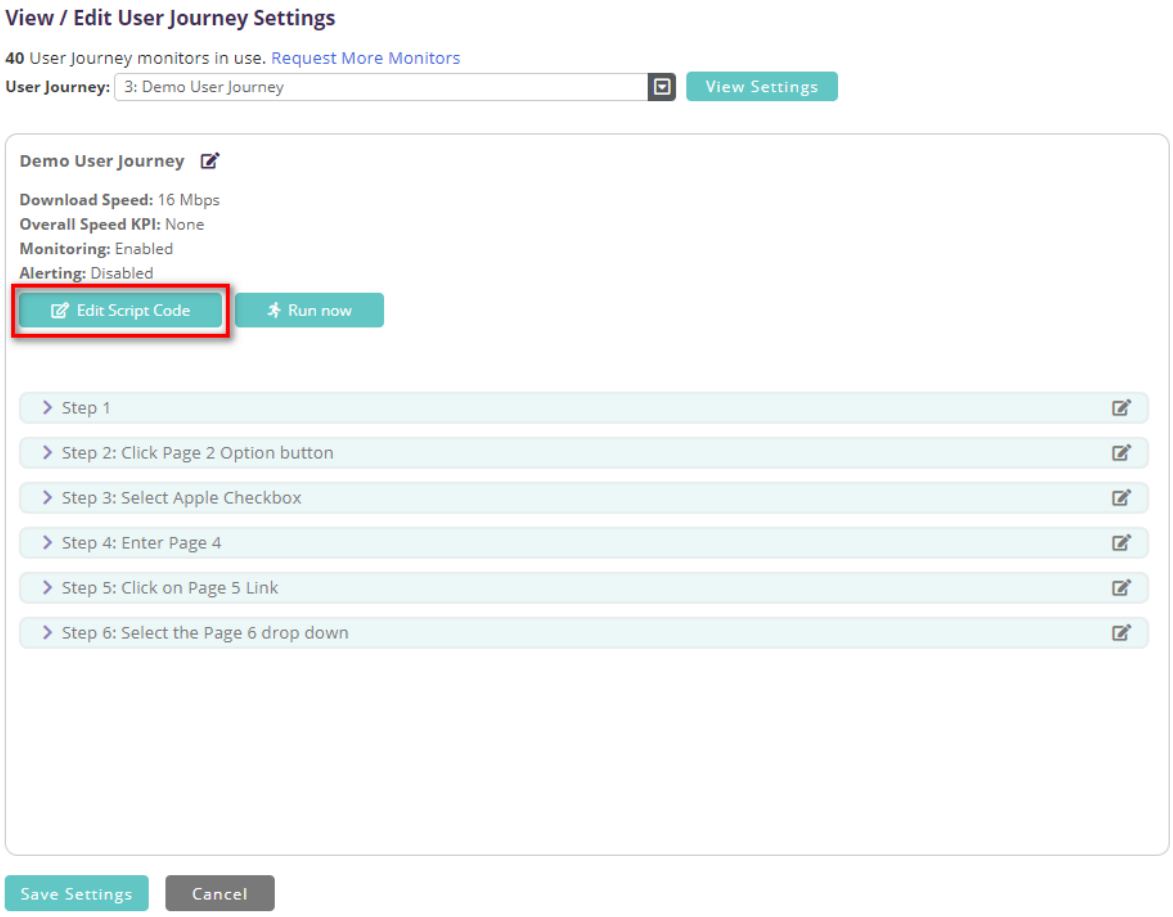

Figure 2: User Journey settings in the administration portal

2.3 Accessing the Script Editor

Figure 3 shows the interface for the script editor for the example script. The script is presented in blocks of code for each step of the journey. There is also a specialist block at the beginning and at the end for advance scripting - see section 4 'Advanced Scripting' below.

Each block can be edited separately but the script is saved and updated as a complete entity using the Save option at the bottom of all the script blocks.

Figure 3: Script code editor

2.4 Viewing the version history

The version history of the code can be viewed as shown in Figure 4.

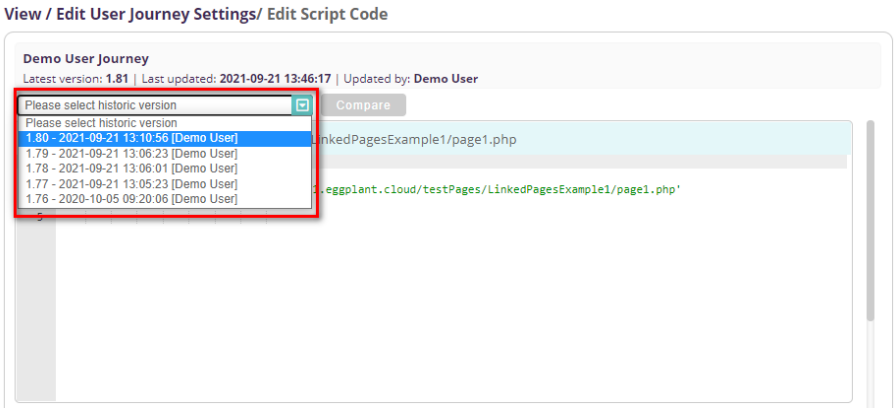

Figure 4: Version history of the code

2.5 Comparing versions

A previous version can be compared with the current version by selecting the relevant previous version from the dropdown and clicking on the Compare button, as shown in Figure 5.

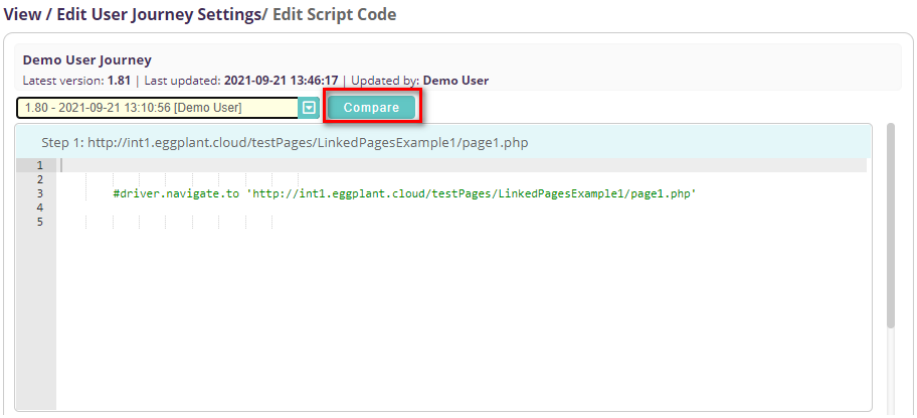

Figure 5: Compare versions

The code differences between the two versions will then be displayed side-by-side as shown in Figure 6.

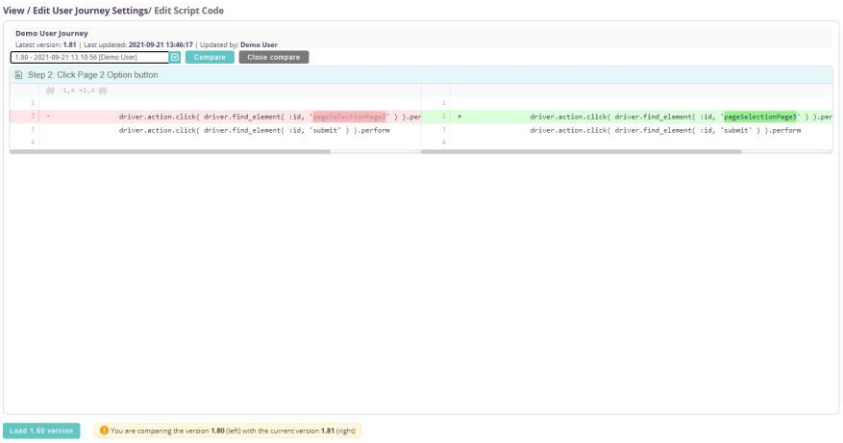

Figure 6: View the code differences

The historic version can be loaded into the script editor by clicking on the 'Load version' button as shown in Figure 7.

Figure 7: Load historic version

The historic version will then be displayed.

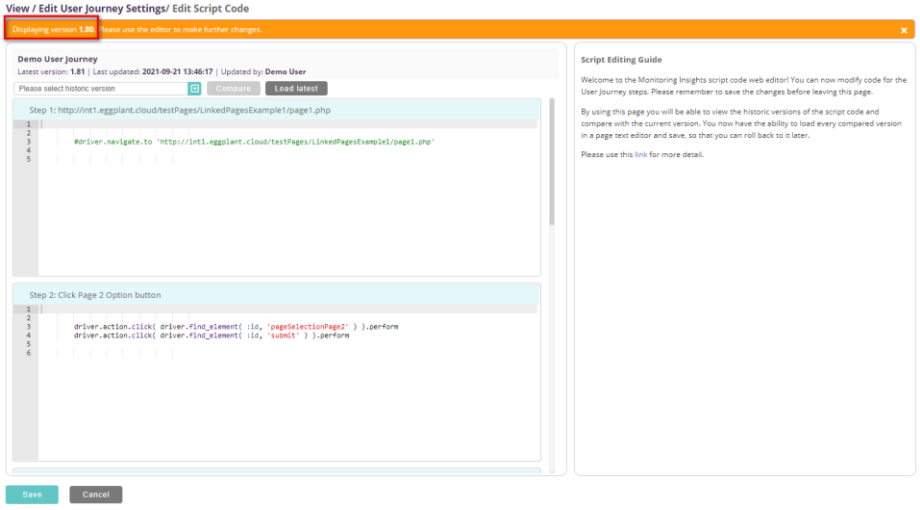

Figure 8: View historic version

2.6 Rollback of code

If desired, the historic version can be saved as the latest version (i.e. rollback) by clicking the 'Save' button, as shown in Figure 9. Changes can also be made to the historic version before saving it as the latest version.

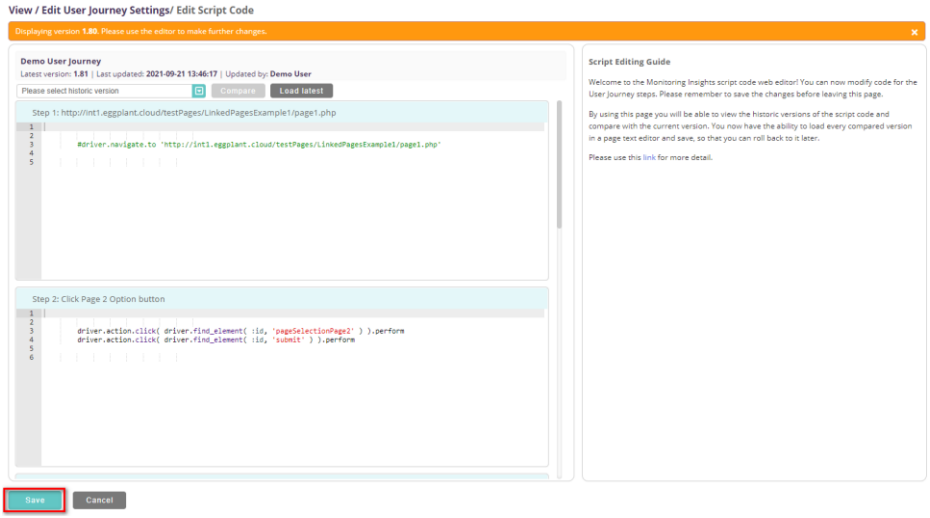

Figure 9: Save historic version

After saving the script a confirmation message will be displayed. The new latest version number of the script will also be shown along with the updated date/time and the name of the user who performed the update. See figure 10.

Figure 10: Display confirmation message

3 Script Editing Reference Guide

Scripts are written in Ruby, with the associated Selenium binding. The following sections document various functions that can be used within the Monitoring Insights scripts.

Care should be taken, as changes will be activated in your monitor as soon as you save them.

It is recommended that when applying changes, you first comment out any existing lines (with a `#`) and then add a new line with the modification, to make it easier to undo the change if necessary. Additionally, it is recommended you add a comment to each change.

If in any doubt, please contact Eggplant Customer Support first before making any changes.

3.1 Script Behaviour Functions

This section describes various methods available and inbuilt within Agent 10 that can be used within any of the steps of a user journey script. Some methods are more advanced and should only be changed with caution.

The methods in this section are used to control the behaviour of the user journey and exist in the `EggPlant::Script::` namespace.

Method/Attribute with Example	Return	Description
`EggPlant::Script::getCustomString()` e.g: `EggPlant::Script::getCustomString()`	string	Gets the custom string for the currently executing script

`EggPlant::Script::getCustomString(ScriptID, shard)`
e.g:
`EggPlant::Script::getCustomString(123, 4)`
	string	Do not modify this function without reference to Eggplant Customer Support first.
		This allows you to get the custom string from different scripts.

`EggPlant::Script::getCustomNumber()` e.g:	int	Gets the custom number for the currently executing script

Method/Attribute with Example	Return	Description
`EggPlant::Script::getCustomNumber()`		

`EggPlant::Script::getCustomNumber(ScriptID, shard)`
e.g:
`EggPlant::Script::getCustomNumber(123, 4)`
	int	Do not modify this function without reference to Eggplant Customer Support first.
		This allows you to get the custom number from different scripts.

`EggPlant::Script::setCustomString(String)`
e.g.:
`EggPlant::Script::setCustomString('hello world')`
	n/a	Sets the custom string for the currently executing script.

`EggPlant::Script::setCustomString(String, ScriptID, shard)`
e.g:
`EggPlant::Script::setCustomString('hello world', 123, 4)`
	n/a	Do not modify this function without reference to Eggplant Customer Support first.
		This allows you to set the custom string for different scripts.

`EggPlant::Script::setCustomNumber(int)`
e.g:
`EggPlant::Script::setCustomNumber(123456)`
	n/a	Sets the custom number for the currently executing script.

`EggPlant::Script::setCustomNumber(int, ScriptID, shard)`
e.g:
`EggPlant::Script::setCustomNumber(123456, 123, 4)`
	n/a	Do not modify this function without reference to Eggplant Customer Support first.
		This allows you to set the custom number for different scripts

Method/Attribute with Example	Return	Description

`EggPlant::Script::setResultCode(resultCode)`
e.g:
`EggPlant::Script::setResultCode(23)`
	n/a	Do not modify this function without reference to Eggplant Customer Support first.
		
		This will force a specified result code for the step.
		
		This function requires an integer parameter, and will return a script error (RC=103) if no parameter is passed, or if the passed parameter is not an integer.
		
		Notes:
		• This will FORCE the result code for the step (irrespective of the success/failure of other checks), so use with caution.
		• Is useful run time checks (such as no stock available) on an otherwise dummy step.

`EggPlant::Script::setRequiredText(String)`
e.g:
`EggPlant::Script::setRequiredText("Condition not met")`
	n/a	Sets the expected phrase to the given string for the current step.
		The phrase is set in the running script and all future runs (as soon as the method is called).

`EggPlant::Script::setRequiredText(String, step)`
e.g:
`EggPlant::Script::setRequiredText("Condition not met", 2)`
	n/a	Sets the expected phrase to the given string for the specified step.
		The step number is indexed from 1 and refers to all steps irrespective of whether or not the step is enabled.

`EggPlant::Script::getFormVariable(String)`
e.g:
`EggPlant::Script::getFormVariable("foo")`
	string	Gets the value of the form variable for the given variable name from the current step
		If the name does not exist, then an empty string is returned.

Method/Attribute with Example	Return	Description

`EggPlant::Script::getFormVariable(String, step)`
e.g:
`EggPlant::Script::getFormVariable("foo", 2)`
	string	Gets the value of the form variable for the given variable name from the specified step.
		The step number is indexed from 1 and refers to steps irrespective of whether or not the step is enabled.

`EggPlant::Script::addExtraInfo()`
e.g:
`EggPlant::Script::addExtraInfo("Selected item 4")`
	string	Creates/appends the given string to ExtraInfo for the given run.
		Multiple calls will continually append to ExtraInfo.

`EggPlant::Script::getUploadFile()` e.g: `EggPlant::Script::getUploadFile()`	string	Gets the path location for the file to be uploaded for the corresponding step being run in. There needs to be a file uploaded against the step.
`EggPlant::Script::getCustomHeaders()` e.g: `EggPlant::Script::getCustomHeaders()`	array	Lists all custom headers currently applied to all requests for this step. It does not include permanent custom headers, use `getPermamentCustomHeaders` as well if you also need to list headers that persistent over multiple steps.

`EggPlant::Script::addCustomHeader(headerName, headerValue)`
e.g:
`EggPlant::Script::addCustomHeader("x-header", "abc")`
	n/a	Adds a custom header for all requests, for this step only. Use `addPermanentCustomHeaders` to persist over multiple steps. Multiple calls can be used to add multiple custom headers.
		NOTE that these headers will automatically be removed after each step.

Method/Attribute with Example	Return	Description

`EggPlant::Script::removeCustomHeader("x-header")`
e.g:
`EggPlant::Script::removeCustomHeader(name)`
	n/a	Removes the custom header for all requests, for this step only. Use `removePermanentCustomHeaders` to remove permanent custom headers that persist over multiple steps.
		NOTE that only custom headers that have been added method can be removed (not standard headers added by the browser).

`EggPlant::Script::getPermanentCustomHeaders()` e.g: `EggPlant::Script::getPermanentCustomHeaders()`	array	Lists all permanent custom headers currently applied to all requests at that point in time. It does not include custom headers for the current step only, use `getCustomHeaders` as well if you also need those.

`EggPlant::Script::addPermanentCustomHeader(name, value)`
e.g:
`EggPlant::Script::addPermanentCustomHeader("x-header", "abc")`
	n/a	Adds a custom header for all requests, for all steps from that point onwards. Multiple calls can be used to add multiple custom headers.

`EggPlant::Script::removePermanentCustomHeader(name)`
e.g:
`EggPlant::Script::removePermanentCustomHeader("x-header")`
	n/a	Removes the custom header for all requests, for all steps from that point onwards.
		NOTE that only custom headers that have been added method can be removed (not standard headers added by the browser).

3.2 Current State of the Run

This section describes various methods available for determining the current state of the monitor run. These all exist in the `EggPlant::Run::` namespace.

Use or change these with caution, please contact Eggplant Customer Support if in any doubt.

Method/Attribute with Example	Return	Description

`EggPlant::Run::finalStepNumber`
e.g:
`if EggPlant::Run::finalStepNumber == 5…`
	int	Returns the final step number for the user journey run, at the point at which it is called.
		Steps are always indexed from 1 and include any disabled steps.
		If used on a step, `finalStepNumber` will always return the number of that step (e.g., `4`), however it is rarely used like this.
		`finalStepNumber` will generally be used in the **finalise** block (see the following section for more details on finalise).
		When called in the finalise block, `finalStepNumber` will return the step number that the user journey got to when completed. This is particularly useful to determine what step number the user journey got to, as it could fail at any step.
		For example:
		• If a 5 step journey runs to completion, `finalStepNumber` called in the finalise block, will return `5`.
		• If a 5 step journey fails at step 3, `finalStepNumber` called in the finalise block, will return `3`.

`EggPlant::Run:: overallResultCode` e.g:	int	Returns the latest result code for the user journey run, at the point at which it is called in the script.

Method/Attribute with Example	Return	Description
`if EggPlant::Run::overallResultCode == 23 && EggPlant::Run::finalStepNumber == 5…`		This will always be the result code, up to the last **completed** step. For example, if called in Step 2, `overallResultCode` will be the result code of Step 1. Note that the `overallResultCode` will always reflect the 'worst' result code up to that point, which may be from several steps prior. Calls to `overallResultCode` are generally made in the **finalise** block (see the following section for more details on finalise).

`EggPlant::Run::severity`
e.g:
`if ["PROBLEM", "ERROR"].include? EggPlant::Run::severity…`
	string	Returns the severity of the running user journey corresponding to the `EggPlant::Run::overallResultCode`, as one of the strings: `"NULL"`, `"OK"`, `"WARNING"`, `"PROBLEM"` or `"DOWN"`.

`EggPlant::Run::testType`
e.g:
`if EggPlant::Run::testType == "RETEST"…`
	string	Returns the test type of the running user journey as one of the strings: `"SCHEDULED"`, `"RETEST"` or `"MANUAL"`.

`EggPlant::Run::getRawContent()`	string	Returns a single string that is a concatenation of all HTTP bodies for textual responses (e.g., HTML, CSS and AJAX responses), for the current step.

3.3 Helper Functions

There are two types of helper functions available within the scripts. Those for simplifying Selenium interactions, and those that provide more generic convenience helpers.

3.3.1 Selenium Helpers

These provide convenience helpers for interacting with websites, implemented using Selenium. These are all extensions to the `Selenium::WebDriver::Driver` interface, so are referenced from the `driver` object.

Method/Attribute	Return	Description
`driver.find_text(text, tag = "*")`	element or nil	Returns a visible element containing the `text` string based on the type of `tag` given (e.g. you can just search within `"div"` elements if you know that text is of that type). If there is no match, or the only match is not visible, then `nil` will be returned. This ensures you can interact with the returned element. If there are multiple visible matches, then the first will be returned.
`driver.find_texts(text, tag = "*")`	[element] or [ ]	The same as `find_text` but will return all matching, visible elements, not just the first. If there is no match, or the only match is not visible, then an empty array will be returned.

`driver.scroll_to(selector, distance = 100, type = :css)`	n/a	Scrolls to the element matching the given selector and then an additional `distance` of pixels to ensure that the selector is fully visible. The `type` of selector can be specified, or the default `css` will be used. By default, this will scroll an additional 100 pixels, but this can be changed by defining the `distance` parameter. If the selector does not exist, then an exception of type `Selenium::WebDriver::Error::NoSuchElementError` will be raised (ideally, use `selector_exists` before scrolling).
`driver.selector_exists(selector, type = :css)`	boolean	Returns whether the given `selector` of the given `type`, exists. If the `selector` exists and is visible, this will return `true`. If the `selector` exists but it not visible, or does not exist at all, this will return `false`.
`driver.selectors_exist(selector, type = :css)`	[element] or [ ]	The same as `selector_exists` but will return all matching, visible elements, not just the first. If there is no match, or the only match is not visible, then an empty array will be returned.
`driver.text_exists(text, tag = "*")`	boolean	Returns whether the given `text` based on the type of `tag` given, exists and is visible. If the `text` exists and is visible, this will return `true`. If the `text` exists but it not visible, or does not exist at all, this will return `false`.

`driver.wait_for_selector(selector, timeout = 30, type = :css)`	boolean	This will wait for up to `timeout` seconds for the `selector` of the given `type` to exist and be visible. If the `selector` is found within the `timeout`, this will return `true`, otherwise `false` will be returned.
`driver.wait_for_text(text, tag = "*", timeout = 30)`	boolean	This will wait for up to `timeout` seconds for the `text` based on the type of `tag` given, to exist and be visible. If the `text` is found and is visible within the `timeout`, this will return `true`, otherwise `false` will be returned.
`driver.wait_while_selector(selector, timeout = 30, type = :css)`	boolean	This will wait for up to `timeout` seconds for the `selector` of the given `type` to no longer exist on the page. If the `selector` no longer exists within the `timeout`, this will return `true`, otherwise `false` will be returned. This is useful for example when additional content is loading behind a spinner and you want to wait until the spinner has gone.
`driver.wait_while_visible(selector, timeout = 30, type = :css)`	boolean	This will wait for up to `timeout` seconds for the `selector` of the given `type` to no longer be visible (but still exists on the page). If the `selector` is no longer visible within the `timeout`, this will return `true`, otherwise `false` will be returned. This is useful for example when additional content is loading behind a spinner and you want to wait until the spinner has gone.

3.3.2 Convenience Helpers

These are generic Ruby helpers for some commonly used functions. These all exist in the `EggPlant::Helper::` namespace.

Method/Attribute with Example	Return	Description

`EggPlant::Helper::generateDate(numDays, format)`
e.g:
`EggPlant::Helper::generateDate(7, "Next week is %d/%m/%Y")`
`EggPlant::Helper::generateDate(-7, "Last week was: %d/%m/%Y")`
	string	Returns a date relative to now, `numDays` in the future (if positive), or `numDays` in the past (if negative).
		The `format` is how you want the date returned, as defined in the `strftime` Ruby function.

`EggPlant::Helper::getContentBetween(src, from, start, end, look_forward=true)`
e.g:
`EggPlant::Helper::getContentBetween('abcdef', '', 'b', 'e')`
`# will return 'cd'`
`EggPlant::Helper::getContentBetween('abcdef', '', 'w', 'e')`
`# will return 'NOTFOUND'`
`EggPlant::Helper::getContentBetween('z1a3y1b3', '', '1', '3')`
`# will return 'a'`
`EggPlant::Helper::getContentBetween('1a3z1b3z', 'z', '1', '3')`
`# will return 'b'`
`EggPlant::Script::getContentBetween('1a3z1b3z', 'z', '1', '3', false)`
`# will return 'a'`
	string	Returns a substring between the `start` and `end` points from the given `src` string, starting from the `from` string (i.e. `from` acts as an anchor to find first).
		If there is no match or the `from` string is not found, then the string `'NOTFOUND'` will be returned.
		By default, this will search for the `from` string and then search to the right (forward) for the `start` string. Set `look_forward` to `false` to search backwards from the `from` string for the `start` string).
		If you do not want an anchor for the `from` string, then set this to an empty string (`''`).

4 Advanced Scripting

This section provides some detail about more advanced scripting, that may or may not apply to your scripts.

Apply these scripts with caution. Contact Eggplant Customer Support if you have any questions.

4.1 Initialise and Finalise Blocks

It is sometimes necessary to run some script code before the initial URL is loaded. A common use case for example is to set permanent custom HTTP headers, which need to be set for all steps, including step 1.

To achieve this, use the **initialise** code block as shown in Figure 11.

Similarly, it is sometimes necessary to run some code when the user journey run has finished. This code always executes at the end of the run, irrespective of the success or failure of the user journey, or which step it finished at. A common use case is to update or reset a custom string or number, ready for the next run.

To achieve this, use the **finalise** code block as shown in Figure 11.

Note that the **finalise** code block is effectively executed after the run has finished. As such, you can only perform certain operations in this block. You cannot for instance, change the result code of the run as it has already finished.

You will often use the methods defined in the “Current State of the Run” section above in the **finalise** block to determine which step or condition the user journey finished at. For example, you may only want to update the custom string if the user journey finished with a specific failure.

The **initialise** and **finalise** code blocks will only be available if they have already been set up in the script. Contact Eggplant Customer Support for more details.

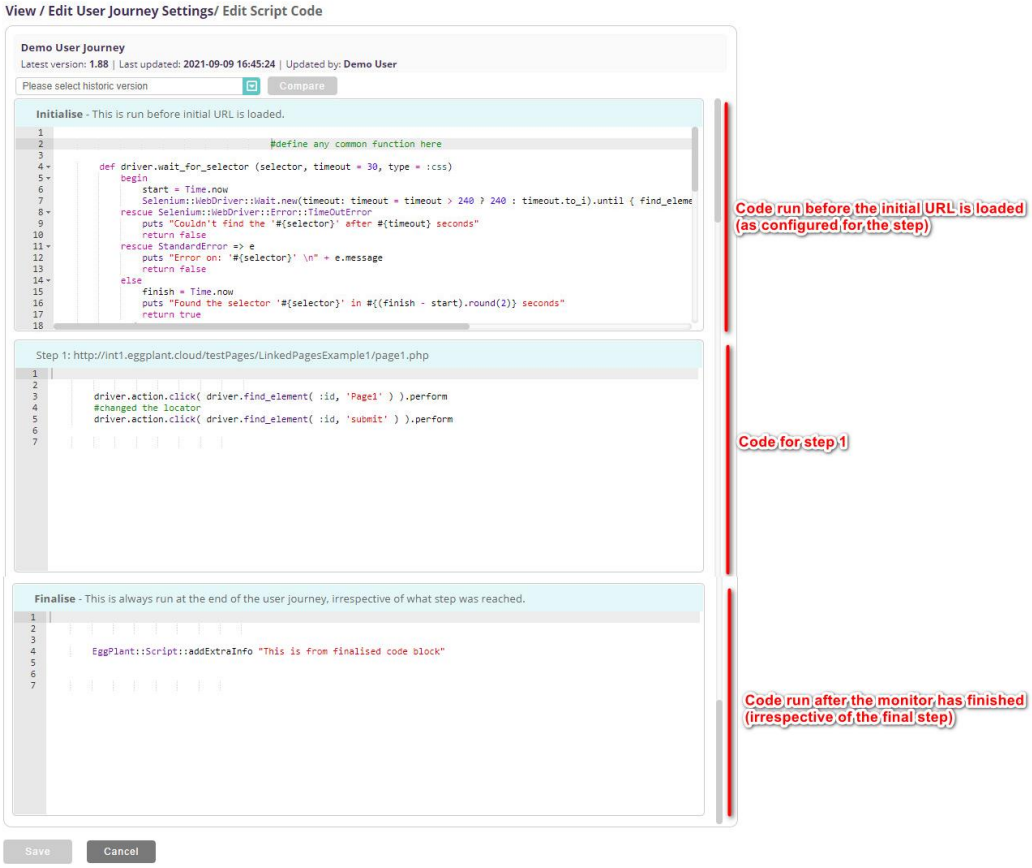

Figure 11: Script code editor with initialise and finalise code block

Script code in the **initialise** block will run before the initial URL is loaded. Any code in the **finalise** block will execute after the run has finished:

4.2 Scope

The code is scoped in such a way that it will be available in subsequent step code blocks.

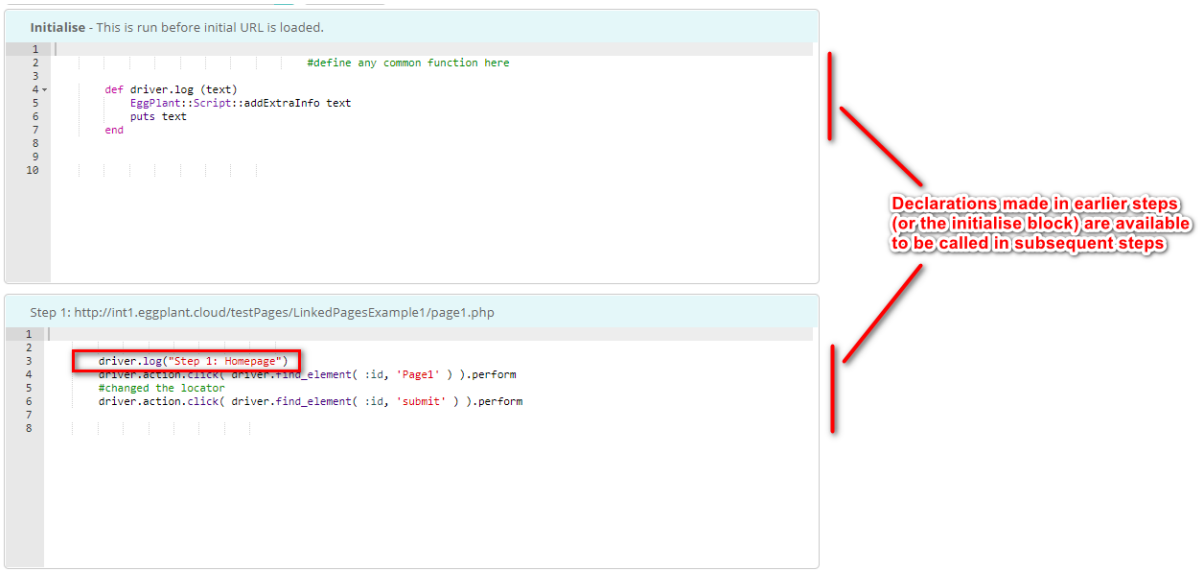

Figure 12: Declaring functions for use in later steps

This is often used to declare functions in the initialise block, or in step 1, and then call those functions in subsequent steps.

If you need to declare a variable for use between steps, then ensure that it is defined with the `@` scope, or it will not work and you will get a result code 103, script error.

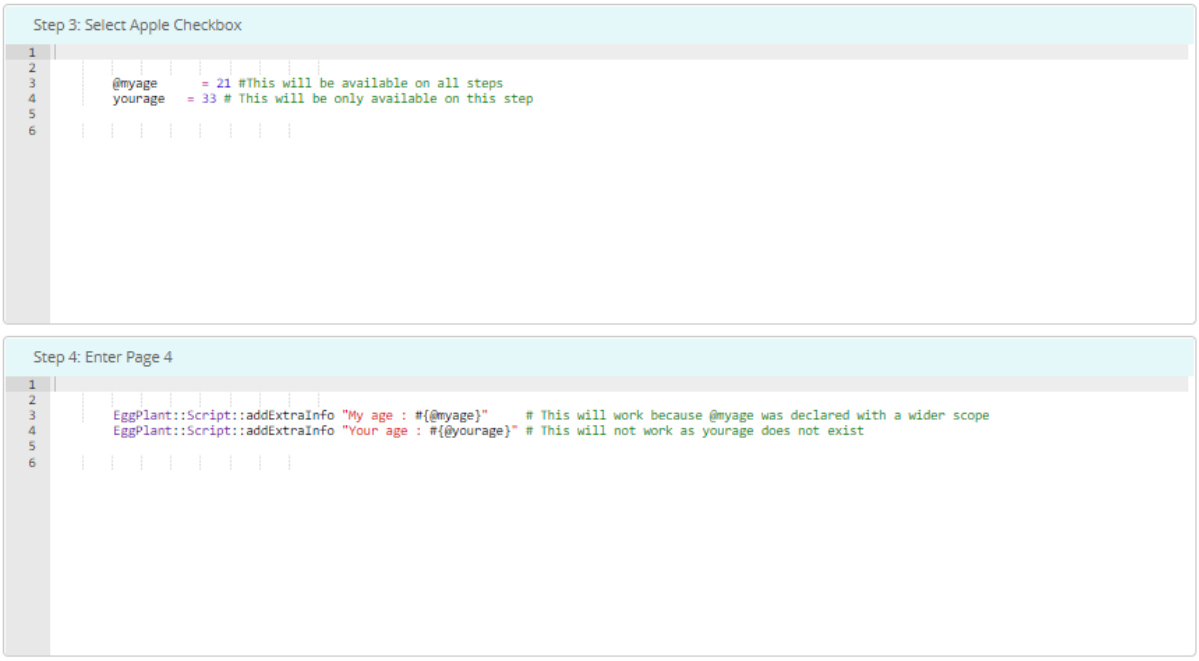

Figure 13: Variable scope