

---

# **testium**

***Release 0.3.2***

**François Dausseur**

**Jun 17, 2026**



# CONTENTS:

<b>1 Overview</b>	<b>1</b>
<b>2 Command Line Interface</b>	<b>3</b>
2.1 -h, --help	3
2.2 -b, --batch-execution	4
2.3 -o, --no-color	4
2.4 -c, --config-file	4
2.5 -r, --run-and-close	4
2.6 -l, --log-file	4
2.7 -d, --define	4
2.8 -p, --report-file	4
2.9 -t, --report-type	4
2.10 -n, --report-pattern	5
2.11 -i, --include-path	5
2.12 -g, --debug	5
<b>3 Modes of operation</b>	<b>7</b>
3.1 Graphical mode	7
3.2 Batch mode	7
3.3 Language server (editor support)	7
3.3.1 Installing the VSCode / VSCodium extension	8
<b>4 TUM file syntax</b>	<b>9</b>
4.1 Configuration files	9
4.1.1 Files loading	10
4.2 Global variables	10
4.2.1 Built-in values	11
4.2.2 Debug mode	11
4.2.3 Test items entries	12
4.2.4 References	12
4.2.5 Dialog values	12
4.2.6 Paramers passing, variable expansion and evaluations	12
4.3 Test Items	13
4.3.1 Items common attributes	13
4.3.2 Container items GUI default folding	17
4.3.3 <b>check</b> test item	17
4.3.4 <b>console</b> test item	17
4.3.5 <b>dialog_choices</b> test item	20
4.3.6 <b>dialog_image</b> test item	22
4.3.7 <b>dialog_message</b> test item	22
4.3.8 <b>dialog_note</b> test items	23
4.3.9 <b>dialog_question</b> test item	23
4.3.10 <b>dialog_references</b> test item	24
4.3.11 <b>dialog_value</b> test items	25

4.3.12	<b>py_func</b> test item	25
4.3.13	<b>git</b> test item	28
4.3.14	<b>group</b> test item	28
4.3.15	<b>jsonrpc</b> test item	29
4.3.16	<b>let</b> test item	32
4.3.17	<b>loop</b> test items	32
4.3.18	<b>lua_func</b> test item	33
4.3.19	<b>parallel</b> test item	36
4.3.20	<b>plot</b> test item	37
4.3.21	<b>report</b> test item	40
4.3.22	<b>run</b> test item	41
4.3.23	<b>sleep</b> test item	42
4.3.24	<b>unittest</b> test item	42
4.3.25	<b>pytest</b> test item	43
4.4	Includes	44
4.5	Templates	45
4.5.1	In the main test file	45
4.5.2	In <code>!include</code> directive	45
4.6	Reports	46
4.6.1	Built-in formats	46
4.6.2	Custom export formats (plugins)	47
4.7	Test outputs	47
4.8	Post execution	48
4.9	Sub-sequence references	48
4.10	Test documentation	48
4.10.1	Unittest	49
<b>5</b>	<b>Python helper library</b>	<b>51</b>
5.1	Global variables helper functions	51
5.2	Plot helper functions	52
5.3	Other helper functions	52
5.4	Debug mode	52

## OVERVIEW

*testium* is an automated test framework developed in python by François Dausseur. This software is developed in python and it implements the Qt6 graphical framework.

It has been developed since 2013 with production and development testing in mind.

It's function is to automate the execution of tests. It can be invoked either as command line application or as a graphical interface application.

Tests reports generation and customization are also in this tool's scope.

Its main features are:

- YAML test description,
- Test configuration files in YAML,
- Full range of pre-existing Test items,
- Test steps, loops,
- Dynamic variables expansion at test runtime,
- Conditional test step execution,
- Modularity of tests (reusable test sequences),
- etc.

All these features give the ability to the test engineer to perform efficient and robust testings.

Each test is described with the help of a [YAML](https://yaml.org/)<sup>1</sup> file having .tum as extension. This file is analyzed and then displayed as a tree in a graphical way in the GUI (see Figure above).

---

<sup>1</sup> <https://yaml.org/>

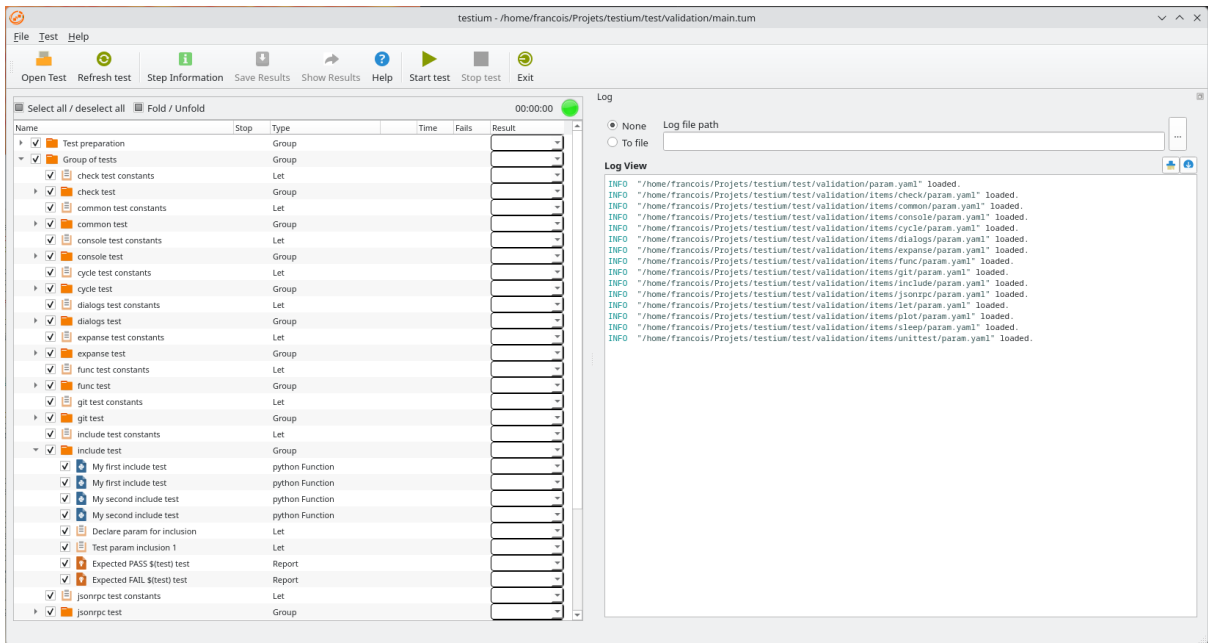


Fig. 1.1: testium

## COMMAND LINE INTERFACE

```
usage: testium.pyw [-h] [--version] [-b] [-c CONFIG_FILE [CONFIG_FILE ...
→]] [-r] [-l LOG_FILE]
                    [-d DEFINE [DEFINE ...]] [-p REPORT_FILE] [-t
→{sqlite,json,junit,html,text}]
                    [-n REPORT_PATTERN [REPORT_PATTERN ...]] [-i
→INCLUDE_PATH [INCLUDE_PATH ...]] [-o] [-g]
                    [test_file]

positional arguments:
test_file              the test script file

optional arguments:
-h, --help            show this help message and exit
--version             Returns the version of testium
-b, --batch-execution
                    Executes the test in batch mode
-c CONFIG_FILE [CONFIG_FILE ...], --config-file CONFIG_FILE [CONFIG_FILE ..
→.]
-o, --no-color        Deactivates stdout colors in batch mode
                    Configuration file
-r, --run-and-close  Runs the test then closes the application
-l LOG_FILE, --log-file LOG_FILE
                    log file name
-d DEFINE [DEFINE ...], --define DEFINE [DEFINE ...]
                    Configuration passed to the executed tests.
-p REPORT_FILE, --report-file REPORT_FILE
                    report file name
-t {sqlite,json,junit,html,text}, --report-type
→{sqlite,json,junit,html,text}
                    report file type
-n REPORT_PATTERN [REPORT_PATTERN ...], --report-pattern REPORT_PATTERN
→[REPORT_PATTERN ...]
                    report file pattern
-i INCLUDE_PATH [INCLUDE_PATH ...], --include-path INCLUDE_PATH
→[INCLUDE_PATH ...]
                    Python modules search path
-g, --debug          GUI debug mode
```

### 2.1 -h, --help

Returns what's in the previous section.

## 2.2 -b, --batch-execution

Executes the test in text mode. No need to have QT installed in that case.

## 2.3 -o, --no-color

Switch allowing to disable the colored output in batch mode.

## 2.4 -c, --config-file

This option allows to provide configuration file(s) from the command line. The configuration files format and content is detailed in the *config files* section.

If this parameter is not given while calling *testium*, the default configuration files will be used.

## 2.5 -r, --run-and-close

This parameter makes testium to close immediately after running the `test_file` argument passed during its call.

If there is no `test_file` argument passed, this option is ignored.

## 2.6 -l, --log-file

Path of the log file where to store the log of the test execution. Goes in a temporary folder if not provided.

## 2.7 -d, --define

Defines one or more variables in the form `VARIABLE1=value1 VARIABLE2=value2 ...`. Then, these variables are available from the test scripts, using the *global variables testium* feature.

## 2.8 -p, --report-file

Path of the report file, stored during the test execution.

This option is only useful in *batch mode*.

## 2.9 -t, --report-type

This option is used in conjunction with option `-p` and is defining the type of report to be generated.

Please read the *reports* section for more details on the possible types of report.

## 2.10 -n, --report-pattern

This option is used in conjunction with option *-p* and is defining the report pattern(s) used to filter the report results which will be included in the report file.

More details in *reports* section.

## 2.11 -i, --include-path

Additional python paths. These paths are appended to the `sys.path`<sup>2</sup> python variable.

## 2.12 -g, --debug

This option is only usefull while debugging *testium* in vscode in *graphical mode*.

---

<sup>2</sup> <https://docs.python.org/3/library/sys.html?highlight=sys%20path#sys.path/>



## MODES OF OPERATION

### 3.1 Graphical mode

*testium* tool has been initially designed to have Graphical User's interface.

The way to call it is simply by executing the `testium` command. It is the normal mode.

### 3.2 Batch mode

The batch mode allows to execute a test in text mode. In this mode, the test does not start any graphical interface.

Listing 3.1: call a test in batch mode

```
testium -b test/my_test/main.tum
```

### 3.3 Language server (editor support)

*testium* ships a [Language Server Protocol](#)<sup>3</sup> server so that `.tum` files get editor assistance — completion of test item types, hover documentation of their parameters, and an outline view — in any LSP-capable editor.

The server speaks LSP over standard input/output and is started with:

Listing 3.2: start the language server

```
testium lsp
```

It is not meant to be launched directly by the user: an editor's LSP client spawns it and drives the exchange. A VSCode / VSCodeium client extension, *testium\_assist*, is provided for that purpose; any other LSP client (Neovim, Emacs `lsp-mode`, ...) can be pointed at `testium lsp` as well.

The information the server exposes is the test item schema, which can also be dumped as JSON for inspection or tooling:

---

<sup>3</sup> <https://microsoft.github.io/language-server-protocol/>

Listing 3.3: dump the item / parameter schema

```
testium schema
```

Because the schema is built from *testium* itself, every new item type or parameter becomes available in the editor on the next *testium* upgrade, with no change to the client.

The language server is included in the pre-built binary, Flatpak and Applmage releases. For a source or wheel installation, pull the optional `lsp` dependencies:

Listing 3.4: enable the language server for a wheel / source install

```
pip install 'testium[lsp]'
```

### 3.3.1 Installing the VSCode / VSCodium extension

The *testium\_assist* client extension is published on [Open VSX<sup>4</sup>](#), the registry used by VS-Codium, Cursor, Windsurf, Eclipse Theia and code-server. In those editors, open the Extensions view and search `testium-assist`, or install it from the command line:

Listing 3.5: install in VSCodium and other Open VSX editors

```
codium --install-extension testium.testium-assist
```

Microsoft *VSCode* uses a different marketplace that does not list Open VSX extensions, so install the packaged `.vsix` by hand. Download it from the Open VSX page linked above, then either choose *Extensions* → ... → *Install from VSIX...* in the UI, or run:

Listing 3.6: install the `.vsix` in Microsoft VSCode

```
code --install-extension testium-assist-0.1.0.vsix
```

The extension launches `testium lsp`, so the `testium` command must be on the `PATH`. If *testium* is installed elsewhere — a specific binary or an Applmage — point the `testium.serverPath` setting at it instead.

Once installed, open a `.tum` file: completion of item types, hover documentation and the outline view become available. If nothing happens, check that no `files.associations` entry forces `*.tum` to another language (it must stay the `tum` language the extension provides).

---

<sup>4</sup> <https://open-vsx.org/extension/testium/testium-assist>

## TUM FILE SYNTAX

*testium* is a python-based tool which uses a YAML based description file to operate tests: the TUM file.

The description of tests is based on the definition of test sequences. There is a main YAML element which is the *testium* tool entry point.

All other steps are listed in the step list of the main. Some steps are themselves list of steps as well, such as loop or console items.

YAML is an indented language and parameter encapsulation is defined by their indentation.

YAML language auto detects data type so that it is not necessary to cast the element type explicitly. See [YAML home page](#)<sup>5</sup> for further information on the language.

The example below shows a basic implementation of the TUM description file:

Listing 4.1: main test file

```
main:
  name: Test example
  steps:
    - test_item1:
      name: test_1
    - loop :
      name: test cyle
      iterator: 5
      steps:
        - test_item:
          name: test_2
        - test_item:
          name: test_3
```

### 4.1 Configuration files

A configuration file can be specified in the *.tum* file or by the command line. This configuration file is optional and must be a YAML file.

The configuration files must have the *.yaml* or *.yml* file name extension.

During the test script loading process, the values defined in these configuration files are added to the global variables and are then accessible from the test items and scripts (cf. [global variables](#)).

The parameter file can be specified in the *.tum* file root:

---

<sup>5</sup> <https://yaml.org/>

Listing 4.2: configuration files definition in the main *.tum* test file

```
config_file:
  config1.yaml
  config2.yaml

main:
  name: Test example
  [ ... ]
```

Listing 4.3: example of configuration file: *param.yaml*

```
parameter1: value1
parameter2: 1234
parameter3: <| 12.34 * 2 |>
parameter4:
  - $(parameter1)
  - $(parameter3)
parameter5:
  sub_param1: sub_value1
  sub_param2: $(parameter4)
```

If nothing is specified, the *param.yaml* is automatically loaded, if present in the test directory.

### 4.1.1 Files loading

The YAML configuration files variables are evaluated directly and accessible from TUM tests description files and also from *python* and *lua* function test items.

See more details *below*.

## 4.2 Global variables

Global variables feature is adding the possibility for test items and test scripts to access a common and global variables database.

The global variables dataset is populated from various sources:

- the *command line*,
- *built-in values*,
- the *configuration files*,
- some test items results,
- the *helper library API*, accessible from python scripts.

Theses global variables are used for variable expansion in scripts (see *variables expansion*).

Another possible usage of the global variables is to share persistent data between test steps.

A library allowing python functions to access global variables is available from the python scripts. See details in section *helper library*.

Apart from the value obtained from the default *param.yaml* or defined configuration files, the global variables entries contains also

- built-in specific value (see *below*),

- values returned by test items.

### 4.2.1 Built-in values

The following keys are automatically accessible through the testium library API (see [helper library](#))

- `test_directory`: the absolute path of the directory of the main .tum file,
- `test_main_file`: the main .tum file,
- `os`: the name of the platform which is used. Can be Linux or Windows,
- `host_name`: The name of the host on which testium is running,
- `home`: home directory of the current user,
- `testrun_date`: The date when the test has started (as a string) in format YYYY-MM-DD,
- `testrun_time`: The time when the test has started (as a string) in format HH:MM:SS,
- `test_name`: The name of the file being executed without extension,
- `home`: the path of the current user's home directory,
- `test_outputs`: list of the paths of the test log and test report (if any),
- `last_step_result`: test result of the last step (see [Items common attributes](#)),
- `ts_start_<item_name>`: timestamp at the start of test item execution (see [Items common attributes](#)),
- `ts_end_<item_name>`: timestamp at the end of test item execution (see [Items common attributes](#)),
- `ts_duration_<item_name>`: duration of test item execution in seconds (see [Items common attributes](#)),
- `cn_<test_name>`: console test item result (see section [console test item](#)),
- `pf_<func_name>`: py\_func test item result (see section [py\\_func test item](#)),
- `lf_<func_name>`: lua\_func test item result (see section [lua\\_func test item](#)),
- `cs_<test_name>`: dialog\_choices test item result (see section [dialog\\_choices test item](#)),
- `loop_param`: loop iterator (available from within a loop item, see [loop test items](#)),
- `loop_index`: loop index (available from within a loop item, see [loop test items](#)),
- `loop_count`: loop number (available from within a loop item, see [loop test items](#)). If the loop number its value is the python constant `inf`.

### 4.2.2 Debug mode

Debug mode can be enabled by defining the global variable `test_debug`.

For example, it can be defined in the configuration file as:

Listing 4.4: example of configuration file: param.yaml

```
[...]
test_debug: True
[.]
```

It can also be defined from the command line with the option `-d test_debug`.

When debug mode is enabled, additional information are displayed in the log window.

Some *helper library functions* are available to give the state of the debug mode.

### 4.2.3 Test items entries

All test items attributes can be global variable entries; when using the entry `$(<global>)` before a key value, the corresponding key entry is searched within the global variables dataset.

### 4.2.4 References

If the `dialog_references` test item has been included (see *dialog\_reference test item*), the global dict will contain the result of this test item in the key `tested_items`.

### 4.2.5 Dialog values

All dialog returned values are inserted in the global variables entries with the key value being the test item name attribute (see *dialog\_value test item*).

### 4.2.6 Paramers passing, variable expansion and evaluations

One of the most useful functionalities for scalability and flexibility of the `.tum` files is the ability to expanse variables at test runtime.

It is done by replacing any occurrence of `$(my_global)` with the content of the variable in the global variables entries (see *global variables*).

The variable substitution is recursive and checks all the occurrences of the `$(x)` pattern in a string.

It is also possible to perform evaluation of python substrings during parameters passing. It is done by using the `<| expr |>` pattern in a string. `expr` may then be a correct python expression.

Below are illustrated simple and more complicated cases of expansion and evaluation depending on their pattern.

Listing 4.5: variables expansion and evaluation

```
- let:
  name: Dynamic variables expansion
  key: $(test)_PASS
  values:
    - expanse_select: <|"$(expanse_select)".replace("o", "a")|>
    - expanse_index: $(expanse_index_$(expanse_select))
    - expanse_table: $(expanse_table_$(expanse_select))
    - expanse_eval: <|$(expanse_index) == 1|>
```

## 4.3 Test Items

All *testium* steps are described in sequence as test items in the step list of the main test item (and eventually of the loop test item).

TUM file main item is itself a variant of test items with a name and an step list attributes.

### Note

Each test item declares the parameters it accepts. When a `.tum` file uses a key the item does not know, *testium* emits a warning listing the accepted parameter names (catching typos such as `param_filee` for `param_file`); a missing **required** parameter aborts loading with an error pointing at the source `.tum` file. Valid existing tests are unaffected.

### 4.3.1 Items common attributes

All test items have common attributes independently of their types, which are listed in next table, those are all optional parameters and their default value if not provided is given in the table as well.

Table 4.1: test items common attributes

Parameter name	Default value	Description
name	test item type	This is the test item name as displayed in the test tree window of the testium. This attribute is also supported by actions of console, jsonrpc or plot test items. Default value, if not provided, is the test item type.
stop_on_failure	False	If stop_on_failure is set to True, the test sequence execution stops on test item failure and no further test items are executed, except those with execute_on_stop attribute set (see below) It depends on the test item to take it into account or not. For example it makes sense to use it for unittest test type because it can contain many sub-tests, but not for sleep test type. In cycles, it means that the child sequence execution is stopped at first failure. It also means that the remaining loops are not executed.
execute_on_stop	False	When this attribute is set to True, the test item is always run, even on test failure of any test before. This feature is useful, to end the test sequence properly on test failure (switch off power supplies, climatic chamber temperature set to ambient temperature....)
skipped	False	The test item execution is to be skipped during test sequence execution if set to True. It will be displayed as failed in the report.
no_fail	False	The result of the test step is forced to PASS if this attribute is set to true.
doc	""	Documentation for the test item that appears in the test doc field and the contextual text window in the testium GUI.
Key	/	This attribute defines a key which will be attached to the test result and which will allow to be filtered during the report generation.
report	/	This attribute defines values (a dictionary) which will be added in the data field of the report.
condition	/	The test item is executed if its condition attribute content is a boolean True. see <a href="#">Conditional execution</a> .
process_result	/	Process an evaluation of the process_result and store it in the result see <a href="#">Process result</a> for details.
expected_result	/	Expected result value or string. see <a href="#">Expected result</a> for details.
store_result	/	Store the test result in a global variable. see <a href="#">Store result</a> for details.

### last test result

The global variable last\_step\_result is automatically set at the end of a test item execution.

If the corresponding test item does not return any actual, the content of the last\_step\_result variable will be the test success (PASS, FAIL or SKIP).

If the test item returns a value, the last\_step\_result variable will contain the returned

value.

The main test items returning a value are:

- *console* test item,
- *jsonrpc* test item,
- *dialog references* test item,
- *dialog value* test item.

### Test timings

After the execution of a test step, the following global variables are set :

- `ts_start_<item_name>`
- `ts_end_<item_name>`

and

- `duration: ts_duration_<item_name>`

See *global variables* for more detail on how to access to global variables from test items and scripts.

### Skipped test items

A variable named `skipped_test_item` can be defined in the global variable entries or in configuration file (see *config files*) as a list of item to be skipped.

### Conditional execution

The condition attribute content must be a boolean value (if not True, the condition is considered not met).

### Process result

The `process_result` attribute can be applied to all the test items. However, it's behavior is different depending if the test item is returning a value or not.

The `process_result` attribute content is evaluated as a python line.

The special `$(result)` variable is replaced in the `process_result` attribute content with the test result value.

The process result is done before the `expected_result`

If the result of the evaluation is a boolean, the test will be *PASSED* if True, and *FAIL* otherwise.

### Expected result

The `expected_result` attribute can be applied to all the test items. However, it's behavior is different depending if the test item is returning a value or not.

The test items returning a value are:

- *dialog\_references test item*
- *dialog\_value test items*
- *py\_func test item*
- *dialog\_choices test item*

- *json\_rpc test item*

For test items which don't return a value, the `expected_result` attribute content is compared to `PASS` or `FAIL`.

The `expected_result` attribute content is a simple comparison with `$(result)`.

If the result and the `expected_result` is equal, the test will be *PASSED* if `True`, and *FAIL* otherwise.

The special `$(result)` variable is replaced in the `expected_result` attribute content with the test result value.

### Store result

The `store_result` attribute stores the test result into a named global variable, making it available to subsequent test items via `$(variable_name)`.

If the test item returns a value (e.g. `py_func`, `json_rpc`), that value is stored. If `process_result` is also specified, the stored value is the post-processed result.

If the test item produces no value (result is `None`), the stored value is the test status string: `"PASS"` or `"FAIL"`, evaluated after `expected_result` but **before** `no_fail`. This ensures the real outcome is captured even when `no_fail: True` would otherwise mask a failure.

Listing 4.6: Store a function return value

```
- py_func:
  name: Read sensor
  func_name: read_temperature
  store_result: temperature

- py_func:
  name: Check temperature in range
  func_name: check_range
  param: [$(temperature), 20, 30]
```

Listing 4.7: Store a post-processed value

```
- py_func:
  name: Get firmware version string
  func_name: get_version
  process_result: "$$(result)'.split('.')[0]"
  store_result: fw_major
```

Listing 4.8: Store the pass/fail status of a test with no return value

```
- console:
  name: Send command
  console_name: device
  steps:
    - writeln: reboot
    - read_until: {expected: "ready", timeout: 10}
  store_result: reboot_status

- py_func:
  name: Use reboot status
  func_name: log_status
  param: [$(reboot_status)]
```

## Export attribute

Listing 4.9: Example of export common attribute usage

```
- check:
  name: Example of result specific to the step 001
  values:
    -  $\$(last\_step\_result) == PASS$ 
  key:
    - GID-1510554_step_1
  report:
    reported_list: <| random.sample(range(0,20), k=10) |>
    reported_float: <| math.sqrt(float(1)) |>
    reported_str: This is my reported sentence
```

### 4.3.2 Container items GUI default folding

The container items are items which are the parent of other test items. For example loops and groups are container test items.

In the GUI, if the user wants that a container test item is folded when he opens a test, the `.` character has to be placed before the test item declaration.

See an example below:

Listing 4.10: example of loop folded by default in the GUI

```
- .loop:
  doc: An example loop
  name: An example loop
  ...
```

### 4.3.3 check test item

The check test item returns the result of a python string evaluation:

Listing 4.11: example of check test item usage

```
- check:
  name: check test item example
  values:
    - "tictactoe" in  $\$(my\_global\_string)$ "
```

### 4.3.4 console test item

The console test item is of the form:

Listing 4.12: example of console test item usage

```
- console:
  name: test name in GUI
  console_name: console name in dict
  steps:
    - open:
      protocol: telnet
      telnet_host:  $\$(target\_ip)$ 
      telnet_port:  $\$(target\_port)$ 
```

(continues on next page)

(continued from previous page)

```
- writeln: reset
- read_until: {expected: U-Boot, timeout: 50}
- write: $(boot_vxworks_1)
- writeln: $(boot_vxworks_2)
- read_until:
  expected: U-Boot
  timeout: 15
- read_until:
  expected: Something that will never occurs
  timeout: 5
  no_fail: True
  mute: True
- close:
```

### Attributes

Beside common test items attributes, console test item has specific attributes:

- `console_name`: console instance name
- `write_delay`: optional parameter giving the delay to wait in milliseconds between each character sent.
- `steps`: a sequence of actions to be applied to the console, as listed above.

The console test item steps accept the parameters and configurations defined in the next sections.

All the following actions support the name attribute. The name is concatenated with the step type in the *testium* GUI, and recalled in the test log and reports.

### open action

The open action initializes the console with the attributes defined as described below. The console instance is then added to the `console_instances` entry of the global variables (cf *global variables*).

Open step accepts the following attribute:

- `protocol`: Setting of the console protocol, supported protocol are listed in table below
- Other attributes are dependent of the protocol in used and are listed in table below

Table 4.2: console protocols

Protocol	protocol parameter	Description
telnet	telnet_host	hostname of the target.
	telnet_port	port of the telnet server of the target.
ssh	ssh_host	Hostname or IP address of the target.
	ssh_user	port of the telnet server of the target.
serial	ssh_pwd	Password (optional).
	serial_port	Serial port to the target.
	serial_baudrate	Baud rate of the serial connection.
rawtcp	buffered	Optinal boolean parameter. If False, it forces the console to read directly the device. Default: True.
	tcp_host	hostname of the target.
terminal	tcp_port	port of the rawtcp server of the target.
	terminal_path	Path of the terminal console.
	shell	Shell to execute in the terminal Default: /usr/bin/env bash

- log: is available only for Telnet and Serial console and is a path to a folder or a file, where the log will be stored.

### close action

The close action closes the console devices and removes its instance from the console\_instances list accessible in the global variables (cf *global variables*).

No parameters required for this action.

### write action

write action takes as parameter the string to be written on the console.

### writeln action

writeln function is similar to the write function except that a 'n' (newline) character is sent at the end of the string to be written.

### read\_until action

The read\_until action is waiting for a string pattern from the console, its parameter are listed below

- expected: the pattern(s) to wait for. It accepts either a **single value** or a **list of values**; when a list is given the action succeeds as soon as **any** of the values is seen.
- regex: Boolean value (True or False, default False). When True every expected entry is interpreted as a Python regular expression (searched in the incoming stream, not anchored) instead of a literal string.
- timeout: Timeout setting for the action (in seconds)
- no\_fail: Boolean value (True or False) leading to no error reported if the expected input is not read
- mute: Boolean value (True or False) does not log any readen data

Listing 4.13: matching several values, and with a regular expression

```
# succeeds as soon as one of the three strings is received
- read_until:
  expected: [login:, "Password:", "$ "]
  timeout: 10

# regex: wait for "version X.Y.Z" with any numbers
- read_until:
  expected: 'version \d+\.\d+\.\d+'
  regex: True
  timeout: 5
```

The text read by the `read_until` action is stored in the global variable named `cn_<test_name>` (See [global variables](#) for more detail on accessing global variables from test items and scripts). When a list of values is given, the report also records, under the matched key, which pattern actually matched.

### Note

regex matching scans a bounded tail of the received stream (`Console.REGEX_WINDOW` characters), so a pattern that could only match after a very large amount of output — or across more than that window — may not be detected. Literal matching (the default) has no such limit.

In the example above, the global variable `$(cn_test name in GUI)` would be created at the end of the step. It would contain the resulting data of the read.

### 4.3.5 dialog\_choices test item

This test item displays a dialog asking a question and waiting for a selection to be done among defined list of items.

These selectable items can be passed as a tree.

The [Figure 4.1](#) displays an example of this item.

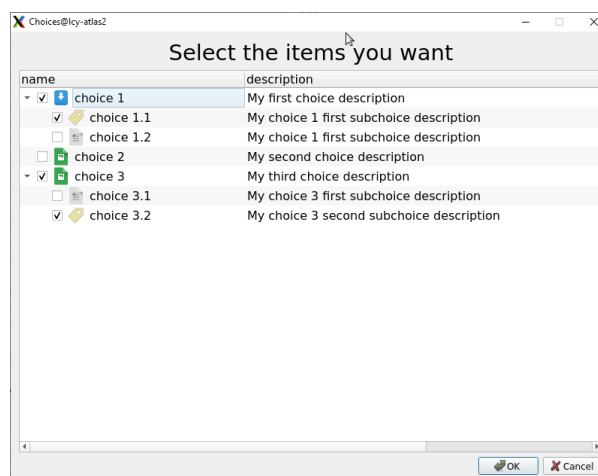


Fig. 4.1: choices dialog

The item parameters corresponding to [Figure 4.1](#) is shown below.

Listing 4.14: example of choices\_dialog test item usage

```

- dialog_choices:
  name: Choices
  question: Select the items you want
  icon: $(test_directory)/document.png
  choices:
    - name: choice 1
      description: My first choice description
      icon: $(test_directory)/document-save.png
      choices:
        - name: choice 1.1
          description: My choice 1 first subchoice description
          icon: $(test_directory)/Label.png

        - name: choice 1.2
          description: My choice 1 first subchoice description

    - name: choice 2
      description: My second choice description
      icon: $(test_directory)/image.png

    - name: choice 3
      description: My third choice description
      icon: $(test_directory)/image.png
      choices:
        - name: choice 3.1
          description: My choice 3 first subchoice description

        - name: choice 3.2
          description: My choice 3 second subchoice description
          icon: $(test_directory)/Label.png

```

### Attributes

The supported attributes of the dialog\_choices test item are:

- **question:** Question to be displayed in the dialog box.
- **choices:** List of the choices presented to the user.
- **icon:** Optional. Path of the icon used in the selection tree, for all the items by default.

### Choices attribute content

Each choice element is a dictionary which can have the following attributes

- **name:** name of the choice to be done.
- **description:** description of the choice to be done.
- **icon:** Optional. Path of the icon displayed in the selection tree in front of the corresponding choice.
- **choices:** List of sub-choices presented to the user (recursive).

### Feature

The dialog references test item creates the `cs_<name of test item>` entry in the global dictionary.

In the example above, the global variable name containing the result of the test item would be `cs_Choices`, and it would contain an object of this form:

Listing 4.15: example of result of the `dialog_choices` test item

```
[
  { 'name': 'choice 1',
    'checked': True,
    'choices': [ { 'name': 'choice 1.1', 'checked': True},
                  { 'name': 'choice 1.2', 'checked': False} ]
  },
  { 'name': 'choice 2',
    'checked': False},
  { 'name': 'choice 3',
    'checked': True,
    'choices': [ { 'name': 'choice 3.1', 'checked': False},
                  { 'name': 'choice 3.2', 'checked': True} ]
  }
]
```

See *global variables* for more detail on how to access to global variables from test items and scripts.

### 4.3.6 dialog\_image test item

This test item displays an image within a dialog box.

`dialog_image` test item has the following description format

Listing 4.16: example of `dialog_image` test item usage

```
- dialog_image:
  name: dialog image test item
  question: operator question
  filename: imageToBeDisplayed.jpg
```

### Attributes

`dialog_image` has the following specific attributes:

- `question`: Question to be displayed in the dialog box
- `filename`: File name of the image to be displayed in the dialog box.

### Feature

The test returns a FAIL if the answer is No and PASS if yes.

### 4.3.7 dialog\_message test item

This test item displays a simple dialog asking a question and returning the entered value. `dialog_message` test item has the following description format

Listing 4.17: example of dialog\_message test item usage

```
- dialog_message:  
  name: dialog value test item  
  question: operator question
```

### Attributes

dialog\_message has the following specific attribute:

- question: Sentence to be displayed in the dialog box

### Feature

Just display the message.

## 4.3.8 dialog\_note test items

This test item displays a simple dialog allowing to enter some text and printing the entered value in logs.

dialog\_note test item has the following description format

Listing 4.18: example of dialog\_note test item usage

```
- dialog_note:  
  name: dialog value test item  
  question: operator question
```

### Attributes

dialog\_note has the following specific attribute:

- question: Question to be displayed in the dialog box

### Feature

Prints the entered text in the log.

## 4.3.9 dialog\_question test item

This test item displays a simple dialog asking a question and returning the entered value.

dialog\_question test item has the following description format

Listing 4.19: example of dialog\_question test item usage

```
- dialog_question:  
  name: dialog value test item  
  question: operator question
```

### Attributes

dialog\_question has the following specific attribute:

- question: Question to be asked in the dialog box

## Feature

The test returns a FAIL if the answer is No and PASS if yes.

### 4.3.10 dialog\_references test item

This test item displays a dialog asking a question and waiting for references of the devices under test.

This test item has the following format:

Listing 4.20: example of dialog\_references test item usage

```
- dialog_references:  
  name: ask for a reference  
  question: Please give the reference of the product  
  reference:  
    - ref 1  
    - ref 2/rev
```

The example above displays the dialog box in Figure 4.2.

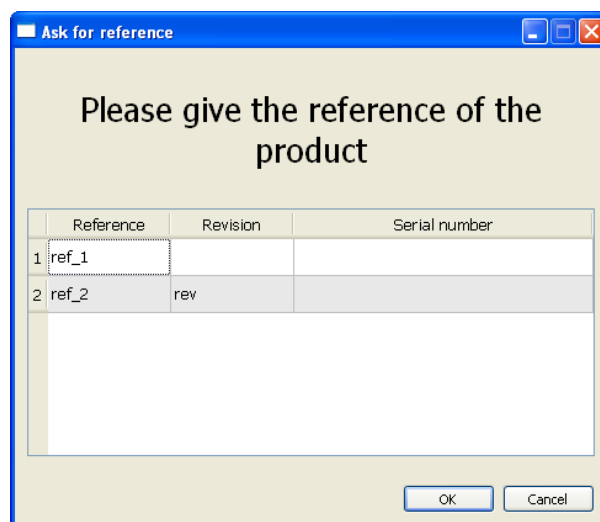


Fig. 4.2: dialog reference

## Attributes

All the following attributes are mandatory.

- question: Question to be displayed in the dialog box.
- reference: For each of this parameter in the test correspond to a row to fill in the dialog.

Every field for a reference can be pre-filled using separating each filed with an '/' (cf Figure 4.2).

## Feature

The dialog references test item creates the tested\_items entry in the global\_dict global variable. This entry is a list of dictionaries of this form:

Listing 4.21: example of `tested_items` global variable result of `dialog_reference` test item

```
[{'reference': 'XXXXX', 'revision': 'YYYYY', 'serial': 'ZZZZZ'}, ...]
```

### 4.3.11 dialog\_value test items

This test item displays a simple dialog asking a question and returning the entered value. `dialog_value` test item has the following description format

Listing 4.22: example of `dialog_value` test item usage

```
- dialog_value:
  name: dialog value test item
  question: operator question
```

#### Attributes

`dialog_value` has the following specific attribute:

- `question`: Question to be displayed in the dialog box
- `default`: default value to place in the dialog form (optional)

#### Feature

The returned value is added in the global variable entry with the key being the `dialog_value` test item name.

### 4.3.12 py\_func test item

The `py_func` test item is used to execute custom python scripts with the given input parameters.

There are two modes for executing a `py_func` item. The *class* mode and the *function* mode.

#### class py\_func item

This is the normal way of calling some custom python code.

A class must be defined and derived from `FunctionItem` from the `py_func.tm` module.

From this class it is possible to define some custom reported values with the following API

- `reportValue(key, value)`: This `FunctionItem` method is adding a value added to the report,
- `reportedValues()`: This `FunctionItem` method is retrieving the current report values.

Listing 4.23: `py_func` test item implementation example

```
import py_func.tm as tm

class TestItemPyFunc(tm.FunctionItem)

    def exec(param1, param2, param4, param4):
        ...
        self.reportValue('my_reported_value', reported_value)
```

(continues on next page)

(continued from previous page)

```
print(self.reportedValues())  
return 10
```

The `exec` method of the `FunctionItem` derived class is executed while running the `py_func` test item.

Listing 4.24: legacy `py_func` test item implementation

```
- py_func:  
  name: function test item  
  file: scriptTestFile.py  
  func_name: TestItemPyFunc  
  param:  
    - 123  
    - 0.123  
    - True  
    - $(global_dict_key)  
  expected_result: 10
```

### legacy `py_func`

The legacy `py_func` test item is of the form:

Listing 4.25: legacy `py_func` python function example

```
def dummy_func(param1, param2, param4, param4):  
    ...  
    return 10
```

There is no possibility to access the report features in that mode.

Listing 4.26: corresponding `py_func` tum extract

```
- py_func:  
  name: function test item  
  file: scriptTestFile.py  
  func_name: funcToBeExecuted  
  param:  
    - 123  
    - 0.123  
    - True  
    - $(global_dict_key)  
  expected_result: 10
```

### Attributes

Beside common test items attributes, `py_func` item has specific attribute, some of which being mandatory.

- `file`: the script file name that contains the function to be executed. Only python script format is supported.
- `func_name`: The function name to be executed.
- `param`: This is a list of parameters that are passed to the function in the order they are presented in the script. These parameters are not mandatory and are highly dependent of the function prototype.
- `context_id`: Optional. When set, all `py_func` items sharing the same `context_id` value run inside the same persistent Python subprocess for the duration of the test. See [py\\_func context](#) for details.

Listing 4.27: py\_func test item example of usage

```
- py_func:
  file: script_name.py
  func_name: methodName
  param:
    - $(my_param)
```

The result of the function (after eventual post treatment) is stored in the global variable named `pfn_<func_name>` (See [global variables](#) for more detail on how to access to global variables from test items and scripts).

In the example above, the global variable `$(pfn_function test item)` would be created at the end of the item execution. It would contain the resulting value of the `funcToBeExecuted` python function.

The `py_func` will always result PASS, except if the called function raises an exception or if the `expected_result` attribute is used.

### Sharing state between py\_func calls

Each `py_func` item without a `context_id` runs in a dedicated subprocess that is started and stopped around the call. State cannot be shared between two such items using module-level variables.

Inside a `py_func` script, `tm.setgd` and `tm.gd` read and write the testium global dictionary. Values stored this way are accessible from any subsequent test item, including other `py_func` items, without requiring a shared subprocess.

Listing 4.28: sharing a value via the global dictionary

```
import py_func.tm as tm

def produce(val):
    tm.setgd("my_shared_value", val)
    return val

def consume():
    return tm.gd("my_shared_value", None)
```

When `context_id` is set, all `py_func` items that share the same identifier reuse the same persistent subprocess. This allows sharing any Python object across calls — including objects that cannot be transmitted to other processes.

Listing 4.29: sharing an object via context\_id

```
import py_func.tm as tm

def open_connection():
    tm.setgd("conn", MyConnection())
    return "ok"

def use_connection():
    conn = tm.gd("conn")
    return conn.status()
```

Listing 4.30: py\_func items sharing a persistent subprocess

```
- py_func:
  name: open connection
  file: my_script.py
  func_name: open_connection
  context_id: my_context
  expected_result: ok

- py_func:
  name: use connection
  file: my_script.py
  func_name: use_connection
  context_id: my_context
  expected_result: open
```

The shared subprocess is automatically stopped at the end of the test run.

### Python Interpreter environment setup

Some global variables have an impact on the py\_func test item behavior:

- `python_bin`: This optional global variable can be used to define the python executable path. If not defined, the python interpreter is searched in at the default places in the system.
- `python_env`: This global variable can be used to define environment variables for the lua script execution environment. Only `PATH` and `PYTHONPATH` are supported.

Listing 4.31: example of configuration file: param.yaml

```
[...]
python_env:
  PATH: "/my/path/"
  PYTHONPATH: "/my/python/modules/"
[...]
```

### 4.3.13 git test item

Git test item allows this item has the following description format

Listing 4.32: git test item usage example

```
- git:
  name: git test item
  repo: [$(test_directory), "/path_to/another/repo"]
```

#### Attributes

- `repo`: a string or list of string path to the root of the git repository(ies) to follow.

### 4.3.14 group test item

This element is of the following form:

Listing 4.33: group test item usage example

```

- group:
  name: Group Item
  condition: <| "$(OS)" == "Linux" |>
  steps:
    - unittest:
      test_file: test_prod_alpha_13.py
      test_method:
        ...
    - sleep:
      timeout: 10

```

The group element is used to manage a sequence of item as a group.

### Attributes

- The steps list describes the sequence executed in the group. It is a list of any of the *testium* test items,

### 4.3.15 jsonrpc test item

The *jsonrpc* test item is used to access jsonrpc servers, by sending queries and analysing the answers. It supports JSONRPC v1.0<sup>6</sup> or v2.0<sup>7</sup>.

This test item can access the jsonrpc server by using an existing *console* or directly using a UDP protocol. Two low level *adapters* can be then chosen: *udp* or *console*.

Example of jsonrpc test item with the console adapter:

Listing 4.34: json\_rpc test item usage example

```

- json_rpc:
  name: JSONRPC console Query
  doc: JSONRPC console Query not waiting (only send)
  console:
    name : jsonrpc_server
    prompt: "@@>"
  timeout: 1
  version: "2.0"
  steps:
    - query:
      method: echo
      params:
        a: Hello world
        b: [0, 1, 2, 3]
      id: 3095372
      no_wait: True

- [...]

- json_rpc:
  name: JSONRPC console Reception
  doc: JSONRPC console reception of the previous request
  console: {name : jsonrpc_server}

```

(continues on next page)

<sup>6</sup> [https://www.jsonrpc.org/specification\\_v1](https://www.jsonrpc.org/specification_v1)

<sup>7</sup> <https://www.jsonrpc.org/specification>

(continued from previous page)

```
timeout: 1
steps:
  - receive:
      name: console reception
      id: 3095372
      timeout: 0.5
```

### Attributes

the jsonrpc attributes are:

- `timeout`: global communication timeout in seconds. It is a floating point number.
- `version`: "1.0" or "2.0" (as a string) depending on the version of the JSONRPC standard which is supported.
- `mute`: a boolean giving the verbosity of the jsonrpc exchanges on the log output.
- An *Adapter* is to be chosen between:
  - Console,
  - UDP,
- `steps`: a sequence of actions as described in the sections below.

### Steps

the jsonrpc steps can be of the following:

- `open`: used by UDP to open the socket explicitly,
- `close`: used by UDP adapter to close the socket explicitly,
- `query`: performs a complete or partial JSONRPC call,
- `receive`: used to receive the JSONRPC result of call previously done by the query action.

If no `expected_value` attribute is defined for query or receive actions, the success of the step will depend on the value returned by the JSONRPC frame. Indeed, this protocol defines a mean to notify if the remote procedure has succeeded or failed.

All the actions support the `name` attribute. The name is concatenated with the action type in the *testium* GUI, and recalled in the test log and reports.

### adapter attributes

The adapters attributes are listed in the table below.

Table 4.3: jsonrpc adapters

adapter	attribute attribute	type	Description
Console	console	<i>dictionary</i>	The console adapter configuration
	console.name	<i>string</i>	The name of the console which will be retrieved from the <i>global variables</i> . See also the <i>console test item</i> .
UDP	console.prompt	<i>string</i>	the eventual enclosing suffix of the jsonrpc frame.
	udp	<i>dictionary</i>	The UDP adapter configuration
	udp.server	<i>string</i>	UDP server hostname or IP address.
	udp.snd_port	<i>integer</i>	UDP server listening port
	udp.rcv_port	<i>integer</i>	UDP answer reception port (on client side)
	bufsize	<i>integer</i>	the maximum expected size of the buffer received while waiting for a jsonrpc frame.

### open action

The open jsonrpc action is only used with the *UDP adapter*<sec\_jsonrpc\_adapters> but is mandatory before any query action.

No parameter is required.

### close action

The close jsonrpc action is only used with the *UDP adapter*<sec\_jsonrpc\_adapters> but is mandatory after JSONRPC transfers are finished.

No parameter is required.

### query action

The query jsonrpc action has the following attributes:

- **method**: JSONRPC method to be called,
- **params**: JSONRPC param (must be conforming to the version defined above), by default it is an empty list.
- **id**: JSONRPC id. If not defined or starts with rand, it is chosen randomly. Otherwise it must be an integer value,
- **timeout**: reception timeout in seconds. It is a floating point number. It is by default the jsonrpc timeout.
- **no\_wait**: Optional boolean. False by default. This attribute defines if the reception is performed in this step (reception can be done appart, in the receive action described below),

### receive action

The receive jsonrpc action has the following attributes:

- **id**: JSONRPC id as an integer value,

- **timeout:** reception timeout in seconds. It is a floating point number, It is by default the jsonrpc timeout.

### 4.3.16 let test item

This element is of the following form:

Listing 4.35: let test item usage example

```
- let:  
  name: Let Item  
  values:  
    key1: value1  
    key2: value2  
    key3: <| $(variable)[$(loop_index)] |>
```

The let element is used to set values in the global directory.

#### Attributes

- The values list gives the {<key>, <value>} couples to set in the global directory,

### 4.3.17 loop test items

This element is of the following form:

Listing 4.36: loop test item usage example

```
- loop:  
  name: Cycle Temperature  
  iterator: 10  
  steps:  
    - unittest:  
      test_file: test_prod_rio6_8093.py  
    - py_func:  
      name: function test item  
      file: scriptTestFile.py  
      func_name: funcToBeExecuted  
      param:  
        - $(loop_param)  
  exit_condition:  
    file: script_name.py  
    func_name: methodName
```

The loop element executes repeatedly the steps sequence of items.

The configuration of the iteration process is done according to the iterator cycle sub-item. As described later in this chapter the iterator is configurable per cycle and allows to call a python function at each cycle loop.

#### Attributes

Below are described loop test item specific attributes.

- **Iterator:** giving the number of loop iteration (see dedicated chapter below).
- **steps:** describes the sequence executed at each cycle; it is a list of any of the testium test items.

- `exit_condition`: allows to exit the loop. If True is returned loop continues otherwise it breaks. `exit_condition` attributes are:
  - `time`: the loop stops after the time (in minutes) is elapsed (optional)
  - `value`: the loop stops when the content of the value attribute is True (optional)
  - `file`: the loop the script file name that contains a function to be executed on each loop. Only python script format is supported (optional if another `exit_condition` attribute is defined)
  - `func_name`: the function to execute on each loop when the file attribute is defined. The function referenced by the `func_name` attribute must have two parameters: the current loop iterator value and the report, even if they are not used. This attribute is mandatory if the file attribute is defined.
  - `eval`: optional parameter allowing post treatment of the function result. It is a python evaluable string in which the `$(result)` keyword is replaced by the actual function call result (see exemple below).

Listing 4.37: Loop exit condition

```
- loop:
  ...
  exit_condition:
    file: script_name.py
    func_name: methodName
    eval: $(result) < 2
```

### Iterator

The iterator attribute can be of the following types:

- An integer giving the cycle loop number,
- A list. The number of elements of the list gives the loop number, and the list member are the consecutive loop parameters,
- Undefined. Then cycle loops until the exit condition is reached.

### Loop variables

The following loop variables are automatically defined:

- `$(loop_param)`: parameter of the loop. It contains the iterator value.
- `$(loop_index)`: index of the loop, starting with 0 and incremented at each cycle.
- `$(loop_index_inverse)`: inverse of index of the loop, starting from cycle length -1 and decremented at each cycle.
- `$(loop_count)`: loop total iteration number. If the number of loops is undefined its value is the python `inf`.

When these variables are found in a parameter, an attribute, etc, a loop is searched recursively in the test hierarchy. And the variable value is replaced by the corresponding loop value.

If more than one loop exists in the test item hierarchy, the lowest level loop iterator is used.

#### 4.3.18 lua\_func test item

The `lua_func` test item is used to execute custom lua 5.4 scripts with the given input parameters.

The `lua_func` test item is of the form:

Listing 4.38: lua\_func python function example

```
local module = {}  
  
function module.dummy_func(param1, param2, param4, param4):  
    ...  
    return 10  
  
return module
```

Listing 4.39: corresponding lua\_func tum extract

```
- lua_func:  
  name: lua function test item  
  file: script_file.lua  
  func_name: dummy_func  
  param:  
    - 123  
    - 0.123  
    - True  
    - $(global_dict_key)  
  expected_result: 10
```

### Attributes

Beside common test items attributes, lua\_func item has specific attribute, some of which being mandatory.

- file: the script file name that contains the function to be executed. Only Lua script format is supported.
- func\_name: The function name to be executed.
- param: This is a list of parameters that are passed to the function in the order they are presented in the script. These parameters are not mandatory and are highly dependent of the function prototype.
- context\_id: Optional. When set, all lua\_func items sharing the same context\_id value run inside the same persistent Lua subprocess for the duration of the test. See [lua\\_func context](#) for details.

Listing 4.40: lua\_func test item example of usage

```
- lua_func:  
  name: activity  
  file: script_name.lua  
  func_name: methodName  
  param:  
    - $(my_param)
```

The result of the function (after eventual post treatment) is stored in the global variable named lfn\_<item\_name> (See [global variables](#) for more detail on how to access to global variables from test items and scripts).

In the example above, the global variable \$(lfn\_activity) would be created at the end of the item execution. It would contain the resulting value of the methodName Lua function.

The lua\_func will always result PASS, except if the called function raises an exception or if the expected\_result attribute is used.

### Sharing state between lua\_func calls

Each `lua_func` item without a `context_id` runs in a dedicated subprocess that is started and stopped around the call. Module-level variables are not preserved between two such items.

Inside a `lua_func` script, the `tm` module exposes `tm.setgd` and `tm.gd` to read and write the testium global dictionary of the test process. Values stored this way are accessible from any subsequent test item without requiring a shared subprocess.

Listing 4.41: sharing a value via the global dictionary

```
local tm = require("tm")
local module = {}

function module.produce(val)
    tm.setgd("my_shared_value", val)
    return val
end

function module.consume()
    return tm.gd("my_shared_value")
end

return module
```

When `context_id` is set, all `lua_func` items that share the same identifier reuse the same persistent subprocess. This allows Lua-side state (upvalues, module cache) to be retained across calls beyond what `tm.setgd` persists.

Listing 4.42: lua\_func items sharing a persistent subprocess

```
- lua_func:
  name: produce value
  file: my_script.lua
  func_name: produce
  context_id: my_context
  param:
    - hello

- lua_func:
  name: consume value
  file: my_script.lua
  func_name: consume
  context_id: my_context
  expected_result: hello
```

The shared subprocess is automatically stopped at the end of the test run.

### Lua Interpreter environment setup

Some global variables have an impact on the `lua_func` test item behavior:

- `lua_bin`: This optional global variable can be used to define the lua executable path. If not defined, the lua interpreter is searched in at the default place in the system.
- `lua_env`: This global variable can be used to define environment variables for the lua script execution environment. Only `PATH`, `LUA_PATH`, and `LUA_CPATH` are supported.

Listing 4.43: example of configuration file: param.yaml

```
[...]  
lua_env:  
  PATH: "/my/path/"  
  LUA_PATH: "/my/luamodules/?.lua;;"  
  LUA_CPATH: "/my/luamodules/?.so;;"  
[...]
```

### 4.3.19 parallel test item

This element is of the following form:

Listing 4.44: parallel test item usage example

```
- parallel:  
  name: My parallel block  
  sync: all  
  branches:  
    - name: Branch A  
      steps:  
        - py_func:  
          name: Long operation  
          file: long_op.py  
          func_name: do_work  
    - name: Branch B  
      wait_for:  
        condition: <| "$(ready_flag)" == "True" |>  
        timeout: 30  
      steps:  
        - let:  
          name: Mark done  
          values:  
            - branch_b_done: true
```

The parallel element runs several sequences of items concurrently. Each inner sequence is called a *branch* and runs in its own thread. The parent test item waits for branches to finish according to the sync policy.

#### Attributes

- **branches:** required. A list of branches to execute concurrently. Each branch has a name and a steps list (same structure as a group item). It can also declare a `wait_for` precondition (see below).
- **sync:** optional, defaults to `all`.
  - `all`: the parallel item completes when every branch has finished. The result is PASS if no branch returned FAIL (skipped or disabled branches are ignored, like in group); otherwise FAIL.
  - `any`: the parallel item completes as soon as the *first* branch finishes. The remaining branches are stopped (their next test items are not executed). The result is PASS if at least one branch succeeded.
- **no\_fail:** optional. When true, a FAIL result is forced to PASS for the parallel item itself (same semantics as for any test item). Branches keep their own result.

## Branch attributes

Each entry of branches is a dict with the following attributes:

- **name:** required. The branch name. Used in reports and as a prefix in the live log output (each line printed by the branch is prefixed with the branch name in square brackets, e.g. [Branch A], so concurrent branches stay readable).
- **steps:** required. The list of test items executed sequentially inside the branch.
- **wait\_for:** optional. Forces the branch to wait until a condition is met before running its steps. If the timeout elapses, the branch returns FAIL (the steps are not run). Sub-attributes:
  - **condition:** a testium expression evaluated repeatedly (every 100 ms) until it returns True.
  - **timeout:** maximum wait, in seconds. Defaults to 30.

## Reporting

Each branch produces its own row in the SQLite report (with type `Parallel branch`), in addition to the parent `Parallel` row. The `log` column of each row contains only the output emitted from that branch's thread, so logs are never mixed between concurrent branches.

In the live (terminal / GUI) output, lines emitted from a branch are prefixed with the branch name in square brackets (e.g. [Branch A]). The prefix is not stored in the SQLite log column.

## Notes

- A `sleep` item inside a branch is interruptible: if another `sync: any` branch wins the race, slow `sleep` items are aborted within ~50 ms.
- A `py_func` or `console` item inside a branch is **not** interruptible: a `sync: any` stop will only take effect after the current item returns. The branch will then skip its remaining steps.
- When a user disables a branch in the GUI tree, the branch returns `SKIP` instantly without affecting the others (it does *not* win a `sync: any` race).

### 4.3.20 plot test item

This test item is used to display runtime values of tests variables or any evolving value in a independent external window.

The plot window is defined using the `plot` test item:

Listing 4.45: plot test item usage example

```
- plot:
  name: test name in GUI
  plot_name: plot identifier
  steps:
    - open:
    - add:
    ...
```

### Attributes

In addition to common test items attributes, console test item has specific attributes:

- `plot_name`: plot window instance name.
- `steps`: a sequence of actions to be applied to the plot window. More than one action can be executed in a plot item.

The plot test item can accept the actions described in further sections.

All the following actions support the name attribute. The name is concatenated with the action type in the *testium* GUI, and recalled in the test log and reports.

### open action

This action initializes and opens the plot window with the corresponding attributes as defined below.

This action accepts one optional path parameter defining a path where are stored the plot lines values in csv format.

Listing 4.46: plot open action

```
- plot:
  name: Open the plot window
  plot_name: plot identifier
  steps:
    - open:
      name: open the plot
      log_path: $(test_directory)/tmp
```

### close action

The close action closes the plot window and removes its from the managed instances of *testium*.

This action does not have mandatory parameters. However, close optional action parameters are:

- `wait_dialog_exit`: Boolean value. If set to True, the window is kept opened until the user closes it manually.
- `timeout`: Value expressed in seconds. It is active if the `wait_dialog_exit` is set to True. If this parameter is defined, and if not closed manually, the dialog window is kept opened until the timeout elapses.

Listing 4.47: plot close action

```
- plot:
  name: Closes the plot
  plot_name: plot identifier
  steps:
    - close:
      wait_dialog_exit: True
      timeout: 600
```

#### Note

When the close action is entered, the periodic plots are stopped.

### add action

The add action is used to add a single data to the plot window.

Listing 4.48: plot add action

```

- plot:
  name: Add to the plot
  plot_name: plot identifier
  steps:
    - add:
      name: add value 1 & 2
      value1: $(loop_index)
      value2: $(loop_index)+2

```

The parameter of the add action is a dictionary of (*key*, *values*) pairs where the *key* is the plot line name and *value* is the numeric value to add to the plot line.

The *value* content is evaluated as a python statement if not a number, but a string.

### periodic action

This action allows to specify a python function to be called and which result is used to update the plot.

periodic plots are updated automatically and don't require further steps in a test sequence, once executed.

periodic action parameters are:

- *period*: period of the automatic value update.
- *file*: python file containing the function to call.
- *func\_name*: the name of the python function to be periodically called.
- *eval*: optional parameter allowing post treatment of the function result.

The *eval* parameter of the periodic action is a python evaluable string in which the *\$(result)* keyword is replaced by the actual function call result.

The result of the action must be a dictionary of (*key*, *values*) pairs where the *key* is the plot line name and *value* is the numeric value to add to the plot line.

Listing 4.49: plot periodic action

```

- plot:
  name: Add periodic to the plot
  plot_name: plot identifier
  steps:
    - periodic:
      period:          1
      file:            $(test_path)$(psep)plot.py
      func_name:       random_value
      eval:            '{"periodic": $(result)}'

```

### last\_value action

The *last\_value* action returns the last values added to the plot (periodically or not) into the global variables entries.

*last\_value* action parameters are:

- name: Optional parameter giving the list of measures to be returned. If it is not defined, all the measures are returned.

Listing 4.50: plot last\_value action

```
- plot:
  name: Plot measure_1 value
  plot_name: plot identifier
  steps:
    - last_value:
      name: [measure_1]
```

The result of the action is stored in the global variable named `plv_<item_name>` in the example above, it would be `$(plv_Plot measure_1 value)`. See [global variables](#) for more detail on how to access to global variables from test items and scripts.

### export action

The export action saves the plot window data in various formats to the filesystem.

Listing 4.51: plot export action

```
- plot:
  name: Plot export
  plot_name: plot identifier
  steps:
    - export: $(my_custom_path)/plot_export.pdf
    - export: $(my_custom_path)/plot_export.csv
```

At the time of writing of this documentation, `.pdf` and `.csv` files are supported.

### 4.3.21 report test item

This test item exports a report file.

To have this functionality activated, a `report` section must be defined at the root of the test file. The root report section is described in [report](#) section.

report test item has the following description format

Listing 4.52: report test item usage example

```
- report:
  name: Intermediate report
  export:
    - junit:
      path: $(home)/reports/report-key-1.junit
      pattern:
        - Unittest%
      key: report-key-1
    - text:
      file_name: report-key-1.txt
      path: $(home)/reports
      key:
        - report-key-1Attributes
```

This item is useful to generate intermediate reports in any format other than `sqlite`. Nevertheless, if `sqlite` export is defined, It won't generate anything.

## Attributes

report test item has the following specific attributes:

- **export**: reports to be exported. It is a list of the reports exports to be executed. The supported exports are:
  - junit
  - json
  - html
  - text

The export sub-attributes (see example above) may contain the following attributes.

- **path**: path of the report files directory,
- **filename**: report file name,
- **Pattern**: list of the patterns (applied on test names) used to select the tests to export into the report,
- **Key**: list of selected keys which are used to select the tests to export into the report.

### 4.3.22 run test item

This test item executes a new instance of testium with the specified .tum file.

- In **batch mode** (-b): the sub-instance is started with -b.
- In **GUI mode**: the sub-instance opens its own window with -r (run and close).
- **batch**: true forces the sub-instance to run headless (-b) even in the GUI.

A sub-instance started with -b is **captured**: its output is streamed into this test's log and report, and kept as the item's result value, so it can be stored with `store_result` and post-processed (`expected_result`, `process_result`, or a `py_func` reading the global variable).

The item result is **PASS** if the sub-instance launched and ran to completion, regardless of whether the sub-tests passed or failed. It is **FAIL** if the file could not be found, the sub-instance could not be launched, or the time window was not reached (see `start_time` / `end_time`).

Listing 4.53: run test item usage example

```
- run:
  name: Execute TUM
  tum: example_cycle.tum
  log_file: $(home)/reports/test.log
  report_file: $(home)/reports/test.rep
```

## Attributes

run test item has the following specific attributes:

- **tum**: mandatory, the path of the file to execute. Can be relative to the current execution folder.
- **param\_file** (optional): the path of the parameter file to use; otherwise the default parameter file is used.
- **batch** (optional): true to run the sub-instance headless (-b) and capture its output even in GUI mode (see above).

- `log_file` (optional): the path of the log file. In GUI mode, if not provided, a file is created with a timestamp next to the `.tum` file. Not used in batch mode (the output is captured instead).
- `report_file` (optional): the path of the report file to create.
- `start_time` (optional): earliest time to execute the sub-instance, in HH:MM format.
- `end_time` (optional): latest time for execution within a time frame, in HH:MM format.
- `wait_for_exec` (optional): true to wait until the time window defined by `start_time` and `end_time` is reached before running. Requires both `start_time` and `end_time`.

### 4.3.23 sleep test item

sleep test item has the following description format

Listing 4.54: sleep test item example usage

```
- sleep:
  name: sleep test item
  timeout: 10
  dialog: True
```

#### Attributes

- `timeout`: sleep duration in second or in relative date format like “2d 5h 31m 3s”, which translate into 2 days, 5 hours, 31 minutes and 3 seconds.
- `dialog`: If set to True, a window showing the remaining time to wait is displayed (optional parameter set to False by default)

### 4.3.24 unittest test item

unittest test item allows the execution of unittest test script which is part of python standard libraries.

The tum file prototype is as followed:

Listing 4.55: unittest test item usage example

```
- unittest:
  name: unitTest test item
  test_file: unitTestScript.py
  test_method:
    - test_1
    - test_2
```

#### Attributes

Beside common test items attributes, unittest test item has specific attribute, some of which being mandatory.

- `test_file`: it is the name (and eventually path) of the unittest file to be processed.
- `test_method`: it is an optional unittest test sub-item. If one or more elements are present, the unittest python script file is parsed and only the corresponding methods are included in the test tree. Otherwise, all the test methods are included in the test tree.

### Access to global variables entries

unittest file tests instances have access to the testium global variables by using the *helper's library*.

### Report value from unittest

Value can be added to the test report from unittest test at runtime.

Listing 4.56: example of unittest test item python function

```
from unittest import (TestCase)

class DummyTests(TestCase):
    def test_01_report(self):
        self.reported_values['key reported'] = 'value_reported'
```

### Console use example with unittest item

Here is an example how to use the console module from python unittest.

Listing 4.57: example of a *testium* console usage from a unittest python function

```
from unittest import (TestCase)
import console

class DummyTests(TestCase):
    @classmethod
    def setUpClass(cls):
        cls.consA0 = console.TelnetConsole('cons name', '192.168.98.123',
→7001)
        cls.consA0.open()
        cls.promptA0 = 'test-computer>'

    def test_01_console(self):
        self.consA0.write('config')
        self.assertEqual(self.consA0.read_until(self.promptA0, 10), 0)
        self.consA0.write('lsusb && echo "Done."\n')
        status, read_data = self.consA0.read_until('Done.',
                                                    10, return_data=True)

        self.assertEqual(status, 0)
        if read_data.find('ID 04f2:b684 Chicony Electronics Co.') != -1:
            index=0

    @classmethod
    def tearDownClass(cls):
        cls.consA0.close()
```

### 4.3.25 pytest test item

The pytest test item runs a [pytest](https://docs.pytest.org)<sup>8</sup> test file and turns every collected test into a child item, each with its own PASS / FAIL / SKIP result, duration and failure message. It is the pytest counterpart of the unittest test item.

The tum file prototype is as followed:

<sup>8</sup> <https://docs.pytest.org>

Listing 4.58: pytest test item usage example

```
- pytest:
  name: pytest test item
  test_file: test_device.py
  test_method:
    - test_boot
    - test_version
```

### Attributes

Beside common test items attributes, the pytest test item has specific attributes:

- `test_file`: the name (and eventually path) of the pytest file to run.
- `test_method`: optional list of test function names. When present, only the matching tests are included in the test tree (the name is matched against the function part of each pytest *node id*, the parametrisation suffix being ignored). Otherwise every collected test in the file is run.

### Host execution

Unlike unittest (which runs in *testium's* own interpreter), the pytest item runs pytest in a **subprocess on the host interpreter** — the same one used by `py_func / lua_func` (overridable with the `python_bin` global variable). As a consequence:

- pytest (and the test's own dependencies) must be installed on that host interpreter — e.g. `pip install pytest`. It is not bundled with *testium*.
- the tests run isolated from *testium's* in-process API; they are meant to be self-contained (they exercise the device/software under test, not the *testium* tree).

The item is invoked with a controlled pytest configuration (`--capture=no`, user adopts neutralised, cache disabled); a small built-in pytest plugin reports the collected tests and their results back to *testium*.

## 4.4 Includes

It is possible to include TUM files from another one by using the `!include` tag before the included file.

This feature is a testium specific implementation and is not part of the YAML language, although it is based on the tagging feature of the language and the customization possibility offered by the python pyYaml package.

Here is a basic example of file inclusion:

Listing 4.59: included\_file.tum

```
- test_item:
  name: test_2
- test_item:
  name: test_3
```

Listing 4.60: main.tum

```
#include with the sub-sequence reference mechanism
sequence: &included_sequence
```

(continues on next page)

(continued from previous page)

```

!include included_file.tum

main:
  name: Test example
  steps:
    - test_item1:
      name: test_1
    - *included_sequence

#include can also be inserted directly within the steps list
    - !include included_file.tum

```

## 4.5 Templates

*testium* embeds the [jinja2](#) template engine. It allows a great customization of the test files, and enforces reusability of test scripts.

### 4.5.1 In the main test file

The *testium* main test files are systematically passed through the jinja template engine.

The parameters passed to jinja are all the variables contained into the *configuration files* plus the *built-in values*.

### 4.5.2 In !include directive

Along with the basic inclusion capability, there is the possibility to use file inclusion parameters. These parameters are replacing corresponding keywords in bracket in the included file.

See examples below.

Listing 4.61: including a template

```

main:
  name: Test example
  steps:
    - test_item1:
      name: test_1

#include can also be inserted directly within the steps list
    - !include
      file: included_template_file.tum
      inclusion_parameter_1: param1
      inclusion_parameter_2: param2

```

Listing 4.62: included template

```

- test_item:
  name: {{ inclusion_parameter_1 }}
- {{ inclusion_parameter_2 }}:
  name: test_3
# The following construction is not allowed and will fail to load:
- test_item:
  name: {{ $(inclusion)_parameter_3 }}

```

## 4.6 Reports

If a report is required (in addition to the log), the report YAML element must be added at the root of the TUM main test file.

The report element accepts a single export or a list of them under the export key. Each export entry uses the format name as its key:

Listing 4.63: reports global settings — multiple exports

```
report:
  enabled: True
  log_stored: True
  export:
    - sqlite:
      path: $(home)/reports
      file_name: $(test_name).db
    - junit:
      path: $(home)/reports
      file_name: $(test_name).xml
    - html:
      path: $(home)/reports
      file_name: $(test_name).html
```

Table 4.4: report attributes

Attribute	default value	Description
enabled	True	Report activated
log_stored	False	When True, captures stdout per test item so exports (html, json) can include the log of each item.
export	/	One export entry or a list of them. Each entry's key is the format name (see below).

Each export entry supports the following sub-attributes:

Table 4.5: export attributes

Attribute	default value	Description
path	\$(report_path)	Output directory.
file_name	/	Output file name. May include \$(...) global-dict expansions.
pattern	/	One or more SQL LIKE patterns applied on the test name.
key	/	One or more SQL LIKE patterns applied on the test key (the per-item key attribute).

### 4.6.1 Built-in formats

- `sqlite` — raw SQLite database (storage layer; selecting it persists the run).
- `text` — simple indented text dump of the test tree.
- `json` — full report as JSON: `{"header": {...}, "tests": [...]}`.
- `junit` — JUnit XML (requires the `junit_xml` Python package).
- `html` — single HTML page with header, results table and per-item logs (requires `lxml`).

If a format is unknown or its optional dependency is missing, the export is skipped with an [report] Export skipped: ... info line on stdout — the test run is **not** interrupted.

### 4.6.2 Custom export formats (plugins)

A third-party Python package can register additional export formats via the testium.exporters setuptools entry point group. Once installed in the same Python environment as testium, the format is auto-detected at startup and can be referenced from the YAML by its declared name.

Plugin contract — a class with this constructor signature:

Listing 4.64: minimal exporter contract

```
class MyExporter:
    def __init__(self, name, con, path, pats, keys, no_header=False):
        # name      : str - report name
        # con       : sqlite3.Connection (read) - tables: header, tests
        # path      : str - output file path (already expanded)
        # pats      : list[str] - LIKE filters on test_name (may be empty)
        # keys      : list[str] - LIKE filters on report_key (may be empty)
        # no_header : bool - skip header section (set by the inline
        #             `report` test item)
        ... # do the work in __init__ and write to `path`
```

Tables and columns of the SQLite report:

- header(key TEXT, value TEXT) — keys: report\_version, test\_file, test\_name, test\_result, test\_revision, testium\_version, testrun\_date, testrun\_time, test\_duration.
- tests — 12 columns: timestamp\_start, test\_id, parent\_id, level, test\_name, test\_type, report\_key, result (PASS/FAIL/SKIP), message, duration (ms), log (captured stdout when log\_stored: True), data (JSON of values reported via self.reportValue(...)).

Declaration in the plugin's pyproject.toml:

Listing 4.65: registering an exporter via entry-points

```
[project.entry-points."testium.exporters"]
my_format = "my_pkg:MyExporter"
```

The plugin is then usable in any .tum report block as my\_format: — no testium configuration change required.

## 4.7 Test outputs

A list of test result outputs is automatically updated by *testium*.

This is a member of global variables dataset which key is test\_outputs.

This global\_dict member contains the log file path and, if configured, the report path as a list.

Other custom logged files may be added by user updated this global variables entry.

## 4.8 Post execution

A post execution script can be run for example to copy the output files.

For that, a `post_execution` element can be defined in the `.tum` file.

If the test set execution succeeded the `post_exec` function of `file_name` module is run else the `post_exec_fail` is run.

If the `post_execution` element is not defined, the `post_execution.py` file in the test directory is used by default if existing.

Listing 4.66: custom post execution python file

```
post_execution:
  file_name: test_report_text.py
```

## 4.9 Sub-sequence references

It is possible to alias any part of the TUM description file (typically a sequence of steps to be executed) to be inserted within another sequence.

This feature uses the anchor/alias mechanism of the [YAML language](#)<sup>9</sup>.

Here is an implementation example of a reference to a sub-sequence in a TUM file:

Listing 4.67: sub-sequences call

```
sequence: &temperature_step_sequence
  - test_item:
    name: test_2
  - test_item:
    name: test_3

main:
  name: Test example
  steps:
    - test_item1:
      name: test_1
    - *temperature_step_sequence
```

### Note

The entry before the alias (`sequence:` in the example above) is needed mandatorily by YAML language syntax. Nevertheless, its value is not used by *testium* and thus can be any value.

## 4.10 Test documentation

It is possible to display some explicative text user in the GUI.

The `doc` attribute of test items is used for that purpose and is displayed as a tooltip on the test row.

---

<sup>9</sup> <https://yaml.org/>

Listing 4.68: tests documentation

```

main:
  name: Test example
  steps:

    - unittest:
      name: unittest item
      doc: |
        The purpose of this unittest test item is to demonstrate
        its various features.
      test_file: dummy/dummy.py
      test_method: test_01_pass

```

See illustration in Figure 4.3.

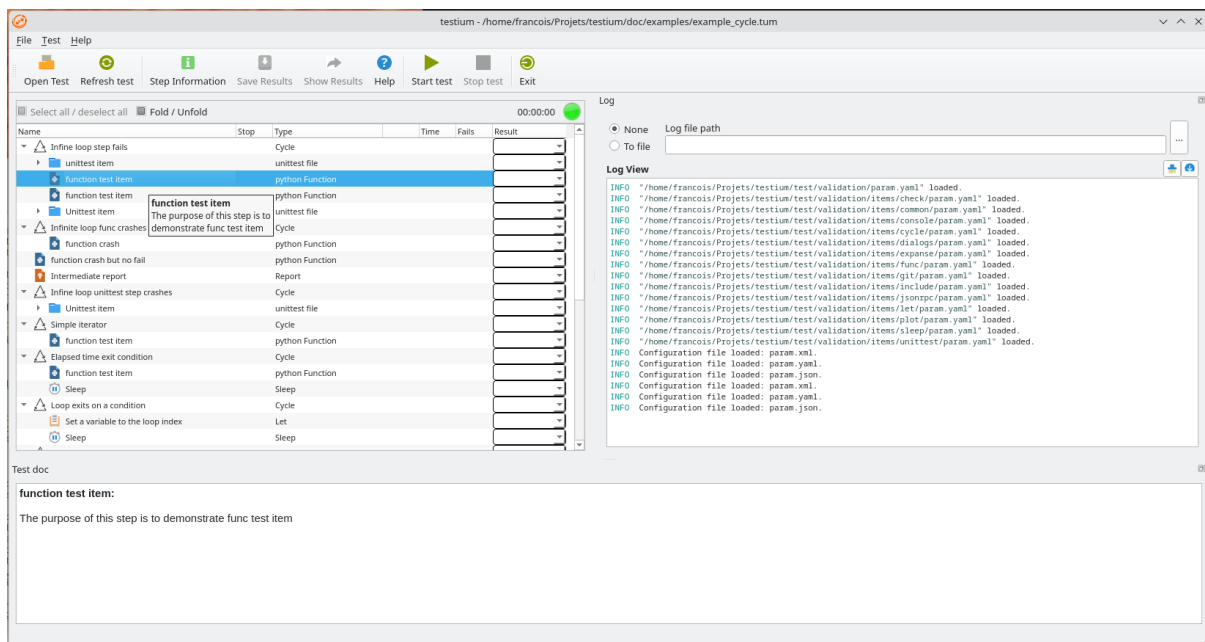


Fig. 4.3: Illustration of the doc attribute effect in the GUI.

### 4.10.1 Unittest

For `unittest` type test items, the python docstring of the test method is used as documentation.



## PYTHON HELPER LIBRARY

A python library including helper function for python modules called from testium py\_func items.

User scripts run inside the py\_func subprocess and interact with testium through a JSON-RPC bridge — the py\_func.tm module. They must **not** import api.testium or interpreter.\* directly: those are main-process modules and may not even be reachable in a packaged build (PyInstaller, .deb).

To include the support of this library in a python script, the following line must be included in the script header:

Listing 5.1: testium helper library import

```
import py_func.tm as tm
```

### 5.1 Global variables helper functions

To manage values in the global variables dataset:

**delgd**(*name*)

Remove an entry from the testium global dictionary.

**Parameters**

**name** (*str*) - Name of the entry to remove.

**gd**(*name, default=None*)

Return a value from the testium global dictionary.

The value is accessible from any test item and from any py\_func subprocess, regardless of the context\_id used.

**Parameters**

- **name** (*str*) - Name of the entry to retrieve.
- **default** - Value returned when the key is absent. Defaults to None.

**Returns**

The stored value, or *default* if not found.

**setgd**(*name, value*)

Store a value in the testium global dictionary.

The stored value is accessible from any subsequent test item and from any py\_func subprocess via gd().

When context\_id is used on the py\_func item, any Python object (including those that cannot be transmitted to other processes) can be stored and shared between calls running in the same subprocess.

### Parameters

- **name** (*str*) - Name of the entry to set.
- **value** - Value to store.

## 5.2 Plot helper functions

Add values to a running plot or read the last value from it:

**add\_plot\_values**(\**params*)

**last\_plot\_value**(\**params*)

Console and plot **lifecycle** management (add\_console, remove\_console, console, add\_plot, remove\_plot, plot) is performed by the console and plot test items themselves — not from user py\_func scripts. Use those test items to open/close consoles and plots.

## 5.3 Other helper functions

**OS**(\**params*)

**get\_main\_dir**(\**params*)

**init\_timestamp**(\**params*)

**text\_mode**(\**params*)

**timestamp**(\**params*)

**timestamp\_as\_sec**(\**params*)

## 5.4 Debug mode

The test\_debug global variable controls debug-only output. Read or write it via tm.gd("test\_debug") / tm.setgd("test\_debug", True).