scenario

Alexis Royer <alexis.royer@gmail.com>

Apr 19, 2023

CONTENTS:

CHAPTER

ONE

PURPOSE OF THE *SCENARIO* TESTING FRAMEWORK

The *scenario* library is a framework for writing and executing full campaigns of tests, with human-readable documentation.

A *scenario* test case is a sequence of *steps*, executed one after the others, defining a *story* by the way.

—

One of the main interests of *scenario* is its ability to *reuse test code*:

- *Step objects*: Instanciate steps one after the others, just like bricks, and quickly write different versions of a story (like a nominal test scenario, then alternative scenarios).
- *Subscenarios*: Reuse existing test cases as subscenario utils, a fair way to set up initial conditions for instance.

Another strength of the *scenario* framework is its *documentation facilities*:

- Tie the test documentation (actions, expected results) right next to the related test code (see *quickstart* for an overview). By the way, the code is more understandable, and the whole easier to maintain.
- Easily collect test *evidence*, just by using the *assertion API* provided.
- Use *execution reports* to generate deliverable documentation in the end.

—

scenario also comes with a set of high quality features, making tests easier to write and maintain:

- Rich *assertion API*, with *evidence* collection (as introduced above).
- Powerful *logging system*, with class loggers, indentation and colorization.
- Handful *configuration facilities*.
- *Campaign* definition and execution.
- *Scenario* and *campaign reports*.
- *Stability* investigation tools.
- Flexible *known issue* and test workaround tracking.
- …

CHAPTER

TWO

INSTALLATION

2.1 Prerequisites

Mandatory:

- Python 3.6 or later (https://www.python.org/downloads/)

Recommended:

- mypy (https://pypi.org/project/mypy/), tested with version 0.931
- pytz (https://pypi.org/project/pytz/)
- PyYAML (https://pypi.org/project/PyYAML/)

Documentation generation (optional):

- Sphinx (https://pypi.org/project/Sphinx/)
- Java

2.2 From sources

Clone the project sources:

```
$ git clone https://github.com/alxroyer/scenario
```

Use the 'bin/run-test.py' or 'bin/run-campaign.py' launchers directly. Let's say you had cloned the project in '/path/to/scenario':

```
$ /path/to/scenario/bin/run-test.py --help
$ /path/to/scenario/bin/run-campaign.py --help
```

CHAPTER

THREE

QUICK START

3.1 Create your first test scenario

The example below shows how to describe a test scenario with step methods.

```
# -*- coding: utf-8 -*-

import scenario


class CommutativeAddition(scenario.Scenario):

    SHORT_TITLE = "Commutative addition"
    TEST_GOAL = "Addition of two members, swapping orders."

    def __init__(self, a=1, b=3):
        scenario.Scenario.__init__(self)
        self.a = a
        self.b = b
        self.result1 = 0
        self.result2 = 0

    def step000(self):
        self.STEP("Initial conditions")

        if self.ACTION(f"Let a = {self.a}, and b = {self.b}"):
            self.evidence(f"a = {self.a}")
            self.evidence(f"b = {self.b}")

    def step010(self):
        self.STEP("a + b")

        if self.ACTION("Compute (a + b) and store the result as result1."):
            self.result1 = self.a + self.b
            self.evidence(f"result1 = {self.result1}")

    def step020(self):
        self.STEP("b + a")

        if self.ACTION("Compute (b + a) and store the result as result2."):
```

(continues on next page)

(continued from previous page)

```
            self.result2 = self.b + self.a
            self.evidence(f"result2 = {self.result2}")

    def step030(self):
        self.STEP("Check")

        if self.ACTION("Compare result1 and result2."):
            pass
        if self.RESULT("result1 and result2 are the same."):
            self.assertequal(self.result1, self.result2)
            self.evidence(f"{self.result1} == {self.result2}")
```

Start with importing the *scenario* module:

```
# -*- coding: utf-8 -*-

import scenario
```

Within your module, declare a class that extends the base *scenario.Scenario* class:

```
class CommutativeAddition(scenario.Scenario):
```

Depending on your configuration (see *ScenarioConfig.expectedscenarioattributes()*), define your scenario attributes:

```
    SHORT_TITLE = "Commutative addition"
    TEST_GOAL = "Addition of two members, swapping orders."
```

Optionally, define an initializer that declares member attributes, which may condition the way the scenario works:

```
    def __init__(self, a=1, b=3):
        scenario.Scenario.__init__(self)
        self.a = a
        self.b = b
        self.result1 = 0
        self.result2 = 0
```

Then, define the test steps. Test steps are defined with methods starting with the `step` pattern:

```
    def step000(self):
```

```
    def step010(self):
```

```
    def step020(self):
```

```
    def step030(self):
```

The steps are executed in their alphabetical order. That's the reason why regular steps are usually numbered.

Give the step descriptions at the beginning of each step method by calling the *StepUserApi.STEP()* method:

```
        self.STEP("Initial conditions")
```

Define actions by calling the *StepUserApi.ACTION()* method:

```
        if self.ACTION(f"Let a = {self.a}, and b = {self.b}"):
```

Define expected results by calling the *StepUserApi.RESULT()* method:

```
        if self.RESULT("result1 and result2 are the same."):
```

Actions and expected results shall be used as the condition of an `if` statement. The related test script should be placed below these `if` statements:

```
        if self.ACTION("Compute (a + b) and store the result as result1."):
            self.result1 = self.a + self.b
```

This makes it possible for the *scenario* library to call the step methods for different purposes:

1. **to peak all the action and expected result descriptions, without executing the test script:**
   in that case, the *StepUserApi.ACTION()* and *StepUserApi.RESULT()* methods return `False`, which prevents the test script from being executed.
2. **to execute the test script:**
   in that case, these methods return `True`, which lets the test script being executed.

The expected result test script sections may usually use assertion methods provided by the *scenario.Assertions* class:

```
        if self.RESULT("result1 and result2 are the same."):
            self.assertequal(self.result1, self.result2)
```

Eventually, the *StepUserApi.evidence()* calls register *test evidence* with the test results. This kind of call may be used under an action or expected result `if` statement.

```
        if self.ACTION("Compute (a + b) and store the result as result1."):
            self.result1 = self.a + self.b
            self.evidence(f"result1 = {self.result1}")
```

```
        if self.RESULT("result1 and result2 are the same."):
            self.assertequal(self.result1, self.result2)
            self.evidence(f"{self.result1} == {self.result2}")
```

Your scenario is now ready to execute.

3.2 Scenario execution

A scenario must be executed with a launcher script.

A default launcher script is provided within the 'bin' directory (from the main directory of the *scenario* library):

```
$ ./bin/run-test.py --help
```

```
usage: run-test.py [-h] [--config-file CONFIG_PATH] [--config-value KEY VALUE]
                   [--debug-class DEBUG_CLASS] [--doc-only]
                   [--issue-level-error ISSUE_LEVEL]
                   [--issue-level-ignored ISSUE_LEVEL]
                   [--json-report JSON_REPORT_PATH]
```

(continues on next page)

(continued from previous page)

```
                  [--extra-info ATTRIBUTE_NAME]
                  SCENARIO_PATH [SCENARIO_PATH ...]

Scenario test execution.

positional arguments:
  SCENARIO_PATH         Scenario script(s) to execute.

optional arguments:
  -h, --help            Show this help message and exit.
  --config-file CONFIG_PATH
                        Input configuration file path. This option may be
                        called several times.
  --config-value KEY VALUE
                        Single configuration value. This option may be called
                        several times.
  --debug-class DEBUG_CLASS
                        Activate debugging for the given class.
  --doc-only            Generate documentation without executing the test(s).
  --issue-level-error ISSUE_LEVEL
                        Define the issue level from and above which known
                        issues should be considered as errors. None by
                        default, i.e. all known issues are considered as
                        warnings.
  --issue-level-ignored ISSUE_LEVEL
                        Define the issue level from and under which known
                        issues should be ignored. None by default, i.e. no
                        known issue ignored by default.
  --json-report JSON_REPORT_PATH
                        Save the report in the given JSON output file path.
                        Single scenario only.
  --extra-info ATTRIBUTE_NAME
                        Scenario attribute to display for extra info when
                        displaying results. Applicable when executing several
                        tests. This option may be called several times to
                        display more info.
```

Tip: See the *launcher script extension* section in order to define your own launcher if needed.

Give your scenario script as a positional argument to execute it:

```
$ ./bin/run-test.py ./demo/commutativeaddition.py
```

```
SCENARIO 'demo/commutativeaddition.py'
-------------------------------------------------


STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.step000)
-------------------------------------------------
    ACTION: Let a = 1, and b = 3
```

(continues on next page)

(continued from previous page)

```
  EVIDENCE:   -> a = 1
  EVIDENCE:   -> b = 3

STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
-----------------------------------------------
    ACTION: Compute (a + b) and store the result as result1.
  EVIDENCE:   -> result1 = 4

STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
-----------------------------------------------
    ACTION: Compute (b + a) and store the result as result2.
  EVIDENCE:   -> result2 = 4

STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
-----------------------------------------------
    ACTION: Compare result1 and result2.
    RESULT: result1 and result2 are the same.
  EVIDENCE:   -> 4 == 4

END OF 'demo/commutativeaddition.py'
-----------------------------------------------
             Status: SUCCESS
    Number of STEPs: 4/4
  Number of ACTIONs: 4/4
  Number of RESULTs: 1/1
               Time: HH:MM:SS.mmmmmm
```

Note: The output presented above is a simplified version for documentation concerns. By default, test outputs are colored, and log lines give their timestamp (see *log colors* and *log date/time* sections).

3.3 Test code reuse

In order to quickly get a first test case running, the example before defines a scenario with *step methods*.

As introduced in the *purpose section*, the *scenario* framework is better being used with *step objects* for test code reuse.

If you're interested in test code reuse, go straight away to *step object* or *subscenario* sections.

Otherwise, take a dive in the *advanced menu* for further information on *scenario* features.

CHAPTER

FOUR

ADVANCED USAGE

4.1 Assertions

The *scenario* framework comes with a rich set of assertion methods, dealing with:

- Equalities, inequalities and comparisons,
- `None` values, object references and types,
- Strings and regular expressions,
- Sets of values (lists or tuples),
- Times v/s step executions,
- JSON data,
- Files and directories.

For the full list of assertion methods, please refer to the detailed documentation of the *Assertions* class which defines them.

It constitutes a base class for common classes like *ScenarioDefinition* and *StepDefinition*.

All assertion methods generally have the following parameters:

err
: The `err` message is the same as the optional message of `unittest` assertion methods.

  It gives the opportunity to explicitize the error message when the assertion fails.

evidence
: The `evidence` parameter may be either a boolean or a string value.

  When this parameter is set with a `True`-like value, *evidence* is automatically stored with the data used for the assertion when it succeeds:

  - Set it to `True` to use the default evidence message only.
  - Set it with a string to specialize the evidence message.

`unittest` assertions

scenario assertions take great inspiration from the well known `unittest` module.

All `unittest` assertions methods may not have their equivalent in the *Assertions* class.

In case you need one of these, the *unittest* object makes it available.

4.2 Logging

Logging is one of the first useful means that help investigating on test executions. That's the reason why a particular attention is paid to providing efficient logging facilities.

For the purpose, the *scenario* framework makes use and extends the standard `logging` library.

4.2.1 Log levels

As the system `logging` module proposes it, log levels are defined by `int` values. The *ERROR*, *WARNING*, *INFO* and *DEBUG* log levels are mostly to be considered.

ERROR, *WARNING* and *INFO* log records are always displayed. *DEBUG* log records may be displayed or not, depending on the context (see *main logger* and *class loggers* below).

Log levels are displayed within the log lines (usually after the *date/time*). They are also commonly emphased with *colors* in the console, in order to facilitate hot analyses.

4.2.2 Date / time

When analyzing test logs, the timing question is usually essential.

Log date/time is displayed by default at the beginning of the log lines, with a ISO8601 pattern: `YYYY-MM-DDTHH:MM:SS.mmmuuu+HH:MM`.

It may disabled through the *scenario.log_date_time* configuration value.

4.2.3 Main logger

The *scenario* framework defines a main logger. It uses a regular `logging.Logger` instance with `'scenario'` for name[1].

It is accessible through the *scenario.logging* variable.

Debugging is enabled by default with this main logger.

```
    def step010(self):
        self.STEP("Logging with the main logger")

        if self.ACTION("Log messages of different log levels with the main logger."):
            scenario.logging.error("This is an error!!!")
            scenario.logging.warning("This is a warning!")
            scenario.logging.info("This is information.")
            scenario.logging.debug("This is debug.")
```

```
STEP#1: Logging with the main logger (demo/loggingdemo.py:22:LoggingScenario.step010)
------------------------------------------------
    ACTION: Log messages of different log levels with the main logger.
              ERROR    This is an error!!!
              WARNING  This is a warning!
```

(continues on next page)

[1] In case you need to manipulate `logging` instance directly, the `logging.Logger` instances are available through the *Logger.logging_instance* property.
The *Logger.logging_instance* property is available to both main logger and class loggers.

(continued from previous page)

```
            INFO     This is information.
            DEBUG    This is debug.
```

Implementation details

The main logger carries the `logging.Handler` instances. It ownes up to two handlers:

1. A first one for the console output, always set.
2. A second one, optional, for file logging when applicable (see *file logging*).

4.2.4 Class loggers

Class loggers make it possible to classify sets of log lines with a given subject: the *log class*. The *log class* is displayed within the log lines, between square brackets, and furthermore makes it possible to enable or disable debug log lines for it.

A *Logger* instance may be created directly.

```
_logger = scenario.Logger("My logger")
```

But a common pattern is to inherit from *Logger*, either directly (see *test libraries*) or through an intermediate class. A couple of *scenario* classes inherit from the *Logger* class, among others:

- *scenario.Scenario*,
- *scenario.Step*.

```
class LoggingScenario(scenario.Scenario):

    def __init__(self):
        scenario.Scenario.__init__(self)
```

Todo: Example needed for inheriting *scenario.Step*.

A *scenario.Scenario* instance is a class logger with the path of the Python script defining it as its *log class*.

A *scenario.Step* instance takes the fully qualified name of the class or method defining it for *log class*.

Debugging is enabled by default for such user-defined scenario and step instances:

```
class LoggingScenario(scenario.Scenario):

    def step020(self):
        self.STEP("Logging with the scenario instance")

        if self.ACTION("Log messages of different log levels with the scenario itself."):
            self.error("This is an error!!!")
            self.warning("This is a warning!")
            self.info("This is information.")
            self.debug("This is debug.")
```

```
STEP#2: Logging with the scenario instance (demo/loggingdemo.py:31:LoggingScenario.
↪step020)
------------------------------------------------
    ACTION: Log messages of different log levels with the scenario itself.
              ERROR    [demo/loggingdemo.py] This is an error!!!
              WARNING  [demo/loggingdemo.py] This is a warning!
              INFO     [demo/loggingdemo.py] This is information.
              DEBUG    [demo/loggingdemo.py] This is debug.
```

Otherwise, debugging is disabled by default for class loggers.

```
class LoggingScenario(scenario.Scenario):

    def __init__(self):
        scenario.Scenario.__init__(self)
        self.class_logger = MyLogger()

    def step030(self):
        self.STEP("Logging with a class logger")

        if self.ACTION("Log messages of different log levels with the class logger␣
↪instance."):
            self.class_logger.error("This is an error!!!")
            self.class_logger.warning("This is a warning!")
            self.class_logger.info("This is information.")
            self.class_logger.debug("This is debug.")
```

```
STEP#3: Logging with a class logger (demo/loggingdemo.py:40:LoggingScenario.step030)
------------------------------------------------
    ACTION: Log messages of different log levels with the class logger instance.
              ERROR    [My logger] This is an error!!!
              WARNING  [My logger] This is a warning!
              INFO     [My logger] This is information.
```

Class logger debugging can be activated on-demand, either 1) programmatically, ...:

```
        if self.ACTION("Activate debugging for the class logger instance."):
            self.class_logger.enabledebug(True)

        if self.ACTION("Log a debug message again with the class logger instance."):
            self.class_logger.debug("This is debug again.")
```

```
    ACTION: Activate debugging for the class logger instance.
    ACTION: Log a debug message again with the class logger instance.
              DEBUG    [My logger] This is debug again.
```

... 2) through the `--debug-class` command line option, ...:

```
$ ./bin/run-test.py --debug-class "My logger" ./demo/loggingdemo.py
```

... or 3) through the *scenario.debug_classes* configuration value.

—

Acces to `logging.Logger` instances

4.2.5 Colors

The *scenario* framework manages colorization in the console, which facilitates hot analyses of the log flow.

Log colors may be disabled through the *scenario.log_color* configuration value.

Log level colorization

The basic usage of log colorization is to highlight log levels:

- *ERROR*: colored in red,
- *WARNING*: colored in yellow,
- *INFO*: colored in white,
- *DEBUG*: colored in grey.

These default colors may be overriden with the *scenario.log_%(level)_color* configuration values.

The log message also takes the color of its log level by default.

Class logger colorization

Log colors may also be applied for every log line of a given *log class*, which is particularly useful when different class loggers generate interleaved log lines:

```
        self.class_logger.setlogcolor(scenario.Console.Color.LIGHTBLUE36)
```

4.2.6 Indentation

Log indentation also contributes in facilitating log analyses.

The *scenario* framework provides several indentation mechanisms.

Scenario stack indentation

When *sub-scenarios* are executed, lines of '|' characters highlight the nestings of scenario executions.

Example of output from the commutativeadditions.py sample test.

```
SCENARIO 'demo/commutativeadditions.py'
----------------------------------------------


STEP#1: Both positive members (demo/commutativeadditions.py:16:CommutativeAdditions.
↪step010)
------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with 4 and 5 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
```

(continues on next page)

(continued from previous page)

```
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = 4, and b = 5
      |   EVIDENCE:   -> a = 4
      |   EVIDENCE:   -> b = 5
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -> result1 = 9
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -> result2 = 9
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -> 9 == 9
      |
      | END OF 'demo/commutativeaddition.py'

STEP#2: Positive and negative members (demo/commutativeadditions.
↪py:23:CommutativeAdditions.step020)
------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with -1 and 3 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = -1, and b = 3
      |   EVIDENCE:   -> a = -1
      |   EVIDENCE:   -> b = 3
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -> result1 = 2
      |
```

(Output truncated...)

If a sub-scenario executes another sub-scenario, the '|' indentation is doubled, and so on.

Class logger indentation

Additional indentation may be useful when the test makes verifications in a recursive way.

It may be set using the following methods:

- *Logger.pushindentation()*,
- *Logger.popindentation()*,
- *Logger.resetindentation()*.

When these calls are made on a class logger, the logging lines of this class logger are indented the way below.

```
    def step110(self):
        self.STEP("Class logger indentation")

        if self.ACTION("Log something with the class logger."):
            self.class_logger.info("Hello")
        try:
            for _ in range(3):
                if self.ACTION("Push indentation to the class logger."):
                    self.class_logger.pushindentation()
                if self.ACTION("Log something with the class logger."):
                    self.class_logger.info("Hello")
            if self.ACTION("Pop indentation from the class logger."):
                self.class_logger.popindentation()
            if self.ACTION("Log something with the class logger."):
                self.class_logger.info("Hello")
        finally:
            if self.ACTION("Reset the class logger indentation."):
                self.class_logger.resetindentation()
            if self.ACTION("Log something with the class logger."):
                self.class_logger.info("Hello")
```

```
STEP#4: Class logger indentation (demo/loggingdemo.py:55:LoggingScenario.step110)
-------------------------------------------------
    ACTION: Log something with the class logger.
              INFO     [My logger] Hello
    ACTION: Push indentation to the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]     Hello
    ACTION: Push indentation to the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]         Hello
    ACTION: Push indentation to the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]             Hello
    ACTION: Pop indentation from the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]         Hello
    ACTION: Reset the class logger indentation.
    ACTION: Log something with the class logger.
              INFO     [My logger] Hello
```

Additional indentation pattern

The *Logger.pushindentation()* and *Logger.popindentation()* methods have a `indentation` parameter that lets you change the 4-space default pattern by what you need.

When removing indentation, the indentation pattern passed on must be the same as the one added in regards.

```
self.class_logger.pushindentation("1> ")
self.class_logger.pushindentation("2> ")
self.class_logger.pushindentation("3> ")
self.class_logger.popindentation("3> ")
self.class_logger.popindentation("2> ")
self.class_logger.popindentation("1> ")
```

Main logger indentation

When *Logger.pushindentation()*, *Logger.popindentation()* and *Logger.resetindentation()* calls are made on the main logger, it takes effect on every log lines:

- main logger and class logger loggings (from *DEBUG* to *ERROR* log levels),
- but also actions, expected results and evidence texts.

```
    def step120(self):
        self.STEP("Main logger indentation")

        if self.ACTION("Log something with the main logger."):
            scenario.logging.info("Hello")
        try:
            for _ in range(3):
                if self.ACTION("Push indentation to the main logger."):
                    scenario.logging.pushindentation()
                if self.ACTION("Log something with the main logger."):
                    scenario.logging.info("Hello")
            if self.ACTION("Pop indentation from the main logger."):
                scenario.logging.popindentation()
            if self.ACTION("Log something with the main logger."):
                scenario.logging.info("Hello")
        finally:
            if self.ACTION("Reset the main logger indentation."):
                scenario.logging.resetindentation()
            if self.ACTION("Log something with the main logger."):
                scenario.logging.info("Hello")
```

```
STEP#5: Main logger indentation (demo/loggingdemo.py:76:LoggingScenario.step120)
------------------------------------------------
    ACTION: Log something with the main logger.
              INFO     Hello
    ACTION: Push indentation to the main logger.
    ACTION:     Log something with the main logger.
                  INFO     Hello
    ACTION:     Push indentation to the main logger.
```

(continues on next page)

(continued from previous page)

```
    ACTION:         Log something with the main logger.
                      INFO     Hello
    ACTION:         Push indentation to the main logger.
    ACTION:             Log something with the main logger.
                          INFO     Hello
    ACTION:             Pop indentation from the main logger.
    ACTION:         Log something with the main logger.
                      INFO     Hello
    ACTION:         Reset the main logger indentation.
    ACTION: Log something with the main logger.
              INFO     Hello
```

Scenario stack v/s user identation

Even though main logger indentation applies to every log lines, it does not break the *scenario stack indentation* presentation: the '|' characters remain aligned, the main logger indentation applies after.

4.2.7 Debugging

Depending on the *logger configuration*, debug lines may be discarded. By the way, formatting the whole logging message prior to discarding is a waste of time. Depending on the amount of debugging information generated along the code, this can slow down the tests in a sensible way.

Such useless formatting processing can be saved by:

1. passing *format arguments as positional arguments*,
2. *delaying string building*.

Formatting & positional arguments

When logging a debug line, one can write:

```
self.debug("Hello I'm %s." % name)  # Option 1: `%` operator.
self.debug(f"Hello I'm {name}")     # Option 2: f-string.
self.debug("Hello I'm %s.", name)   # Option 3. positional arguments.
```

The second option is preferrable to the first one in as much as as it is easier to maintain (main point for f-strings), and f-strings are around 10% more efficient.

Still, with f-strings, the resulting string is computed before it is passed to the *Logger.debug()* method, and possibly discarded after being computed.

That's the reason why, the third option is even more efficient for debug logging: a useless message will be discarded before the formatting arguments are applied to it.

Delayed strings

Even when passing format arguments as positionals, some of them may take a while being computed by themselves.

That's the reason why the *scenario.debug* package gathers a couple of functions and classes that enable delaying more string computations:

Function	Class	Description	Example
	DelayedStr	Abstract class that defines a string which computation may be delayed. You may inherit from this base class for specific needs.	
	FmtAndArgs	Describes a delayed string that should be built with format and arguments. The string can be prepared step by step, thanks to the *FmtAndArgs.push()* method. The application of the arguments is delayed on time when needed.	`_str = scenario.debug.FmtAndArgs("Hello, I'm %s", name)` `if profession:` `  _str.push("and I'm a %s", profession)` `_str.push(".")` `self.debug(_str)`
saferepr()	*SafeRepr*	Computes a *repr*-like string, but ensures a *not-too-long* string, possibly focused on certain parts, such computation being delayed as for the others.	`self.debug(` `  "%s in %s",` `  scenario.debug.saferepr(searched),` `  scenario.debug.saferepr(longtext, focus=searched),` `)`
jsondump()	*JsonDump*	Delays the dump computation for JSON data.	`self.debug(` `  "JSON data: %s", scenario.debug.jsondump(data, indent=2),` `  extra=self.longtext(max_lines=10),` `)` **Tip:** *jsondump()* may basically be displayed as *long texts*.
callback()	*CallbackStr*	Delays the execution of a string builder callback. Possibly set with a lambda, this function makes it possible to delay quite everything.	`self.debug(` `  "Very special data: %s",` `  scenario.debug.callback(lambda x, y, z: ..., arg1, arg2, arg3),` `)`

Long texts

The *scenario* logging feature provides a way to log long texts on several lines.

To do so, set the `extra` parameter using the *Logger.longtext()* method when logging some text:

```
self.debug(scenario.jsondump(_json_data, indent=2),
           extra=self.longtext(max_lines=10))
```

This feature has primarily been designed for debugging, but it works with the *Logger.info()*, *Logger.warning()* and *Logger.error()* methods as well.

The `max_lines` parameter may be set to `None` in order to display the full text.

4.2.8 File logging

The *scenario* logging feature provides the ability to save the test log output into a file.

To do so, set the *scenario.log_file* configuration value, either with the `--config-value` command line option, or within a configuration file.

The command line example below disables the output in the console, but saves it into the 'doc/data/commutativeaddition.log' file (from the main directory of the *scenario* library):

```
$ mkdir -p ./doc/data
$ ./bin/run-test.py \
    --config-value "scenario.log_console" "0" \
    --config-value "scenario.log_file" "./doc/data/commutativeaddition.log" \
    ./demo/commutativeaddition.py
$ cat ./doc/data/commutativeaddition.log
```

```
SCENARIO 'demo/commutativeaddition.py'
----------------------------------------------


STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.step000)
----------------------------------------------
    ACTION: Let a = 1, and b = 3
  EVIDENCE:   -> a = 1
  EVIDENCE:   -> b = 3

STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
----------------------------------------------
    ACTION: Compute (a + b) and store the result as result1.
  EVIDENCE:   -> result1 = 4

STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
----------------------------------------------
    ACTION: Compute (b + a) and store the result as result2.
  EVIDENCE:   -> result2 = 4

STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
----------------------------------------------
    ACTION: Compare result1 and result2.
```

(continues on next page)

(continued from previous page)

```
    RESULT: result1 and result2 are the same.
  EVIDENCE:   -> 4 == 4

END OF 'demo/commutativeaddition.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 4/4
  Number of ACTIONs: 4/4
  Number of RESULTs: 1/1
               Time: HH:MM:SS.mmmmmm
```

Tip: The *scenario.log_file* configuration value may also be set programmatically through the *ConfigDatabase.set()* method, as illustrated in the *launcher script extension* section.

4.2.9 Extra flags

The *LogExtraData* define a set of flags that can be set to specialize the behaviour of each *Logger*.

For instance, the *ScenarioRunner* and *ScenarioStack* classes disable the *LogExtraData.ACTION_RESULT_MARGIN* flag, so that their related log lines remain aligned on the left whatever the current action / expected result context is.

Please, refer the following links for details on extra flags:

- *LogExtraData*
- *Logger.setextraflag()*

Warning: Setting extra flags on class loggers, or even worse on the main logger, may lead to unpredictable behaviours.

Nevertheless, this advanced feature is provided as is. To be used with parsimony.

4.3 Test evidence

Storing test evidence with the test results might be a good practice.

When one reads test results, and only knows about action and expected result texts, he/she has to trust that the test script actually did what is written in the texts given.

In order to tackle this uncertainty, evidence may be stored with test results. Doing so reinforces the credibility of the results, in as much as a human could check manually that the automatic test script did the right thing.

As introduced in the *quickstart guide*, the *StepUserApi.evidence()* method, available in *ScenarioDefinition* and *StepDefinition* classes, lets you save evidence while the test is executed.

Assertion routines defined in the *Assertions* class can be used to collect evidence as well. Set the optional `evidence` parameter to either `True` or a string describing what is being checked.

Test evidence is saved with the scenario JSON reports in the 'evidence' list of each action or expected result execution.

4.4 Error management

Todo: Documentation needed for error management:

- By default, errors break the test execution (*assertions* or any exception).
- Except for *known errors*.
- Except when the `--continue-on-error` option or *ScenarioConfig.Key.CONTINUE_ON_ERROR* configuration is set.

4.5 Stability tracking

Todo: Documentation needed for stability tracking facilities.

Feature coming soon (#63).

4.6 Known issues

One dilemma we commonly have to face when managing tests is to deal with known issues.

On the one hand, as long as a known issue exists, the feature that the test verifies cannot be said to be completely fulfilled. Thus, putting the test in the *ExecutionStatus.FAIL* status is a formal way to mark the corresponding feature as not fully supported yet.

On the other hand, from the continuous integration point of view, seeing the test in the *ExecutionStatus.FAIL* status from day to day, apparently because of that known issue, may occult useful information on other possible regressions.

That's the reason why the *scenario* framework provides an API in order to register known issues in the tests (see *StepUserApi.knownissue()* and *KnownIssue*).

4.6.1 Default behaviour

By default, known issues are handled as simple warnings, making the tests fall to the intermediate *ExecutionStatus.WARNINGS* status. The warnings are logged in the console, and saved in *test reports*.

This way, a regression will be highlighted as soon as it occurs by the continuous integration process, in as much as the test will turn to the *ExecutionStatus.FAIL* status.

This way, one can safely implement a workaround in a test affected by a known issue, but track it formally in the same time. Once the known issue has been fixed in the *software/system under test* (SUT), the workaround and the known issue reference can be removed from the test, hopefully turning the latter into the *ExecutionStatus.SUCCESS* status.

4.6.2 Issue levels

Issues may exist for various reasons, representing various criticities:

1. As introduced in the section before, an issue may be related to the *software/system under test* (SUT).
2. But talking about incremental development, it may also be because the SUT has not fully implemented a given feature yet, this being planned in a later release, which is a less critical situation than (1).
3. The issue may also be related to defects of the test environment, which is obviously less critical than defects of the SUT itself (1).
4. But talking about the test environment, it may also be related to the context in which the tests are being executed (Internet not ready, ability of the test platform, ...) which may be less critical again than real defects of the test environment (3).
5. *Other reasons...*

In order to discriminate the various situations, known issues may be registered with an issue level (*IssueLevel*).

Issue levels are basically integer values. The higher, the more critical. Programmatically, they can be described by an `enum.IntEnum`.

They can be associated with meaningful names, with the help of the *IssueLevel.definenames()* and/or *IssueLevel.addname()* methods, or *scenario.issue_levels* configurations. These names make it easier to read in the console, and maintain in the test code.

Enum-defined issue levels

If issue levels are defined with an `enum.IntEnum`, this `enum.IntEnum` class can be passed on as is to the *IssueLevel.definenames()* method.

```
import enum
import scenario


# Define issue levels.
class CommonIssueLevel(enum.IntEnum):
    SUT = 40
    TEST = 30
    CONTEXT = 20
    PLANNED = 10
scenario.IssueLevel.definenames(CommonIssueLevel)


class MyStep(scenario.Step):

    def step():
        self.STEP("...")

        # Track a known issue, with issue level *PLANNED=10*.
        # By default, this known issue is logged in the console, and saved in JSON␣
↪reports, as warning.
        self.knownissue(
            level=CommonIssueLevel.PLANNED,
            message="Waiting for feature XXX to be implemented",
        )
```

(continues on next page)

(continued from previous page)

```
        # Do not proceed with the following test actions and expected results until␣
↪feature XXX is implemented.
        # if self.ACTION("..."):
        #     ...
```

4.6.3 Error / ignored issue level thresholds

Once issue levels are set, two issue level thresholds may be used when launching the test or campaign in order to tell which issue levels should be considered as errors, warnings, or simply ignored.

Table 1: Error and ignored issue levels

	Configuration	Effect
Error issue level	`--issue-level-error` option or *scenario.issue_level_error* configuration	Known issues with issue level greater than or equal to the given value are considered as errors. **Note:** When the *error issue level* is set, known issues without issue level turn to errors by default. Known errors don't break the test execution as *test exceptions* do. By the way, several errors may be logged and saved in *test reports*.
Ignored issue level	`--issue-level-ignored` option or *scenario.issue_level_ignored* configuration	Known issues with issue level less than or equal to the given value are ignored.

This way, without changing the test code, permissive executions can be launched for continuous integration purpose, but stricter executions can still be launched to constitute official test results.

4.6.4 Issue identifiers

Known issues may be registered with an issue identifier, refering to a tier bugtracker tool.

Optionally, a URL builder handler may be installed (see *KnownIssue.seturlbuilder()*), in order to build URLs to the tier bugtracker tool from issue identifiers. These URLs are then displayed in the console and saved in *test reports*, and are usually directly clickable from both contexts.

```
import scenario
import typing


# Define and install a URL builder handler.
def _urlbuilder(issue_id):  # type: (str) -> typing.Optional[str]
    if issue_id.startswith("#"):
        return f"https://repo/issues/{issue_id.lstrip('#')}"
    # Unexpected issue id format, return `None` for no URL.
```

(continues on next page)

(continued from previous page)

```
    return None
scenario.KnownIssue.seturlbuilder(_url_builder)


class MyStep(scenario.Step):

    def step():
        self.STEP("...")

        # Track issue #10.
        # Thanks to the URL builder handler, the 'https://repo/issues/10' URL is␣
↪displayed in the console and saved in JSON reports.
        self.knownissue(
            id="#10",
            message="Waiting for feature #10 to be implemented",
        )

        # Do not proceed with the following test actions and expected results until␣
↪feature #10 is implemented.
        # if self.ACTION("..."):
        #     ...
```

Tip: *Issue levels* and *issue identifiers* can be used in the same time when registering known issues.

4.6.5 Registration: definition v/s execution level

It is generally preferrable to register known issues at the definition level (i.e. outside action / result blocks). Doing so, even though an error occurs during a test execution, known issues are still saved with the test results.

Nevertheless, certain known issues can't be registered at the definition level (issues related to the test execution context for instance). For such situations, it remains possible to register known issues at the execution level (i.e. inside action / result blocks), but there is no guarantee that the known issue will be saved with the test results, since it depends on the test execution.

Known issues from test libraries

The *ScenarioStack.knownissue()* is provided in order to register known issues from anywhere in *test libraries*.

4.7 User test libraries

Todo: Documentation needed for user test libraries.

Create test libraries by inheriting both *Assertions* and *Logger*.

```
class MyLogger(scenario.Logger):

```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        scenario.Logger.__init__(self, "My logger")
```

```
# -*- coding: utf-8 -*-

import scenario


class HtmlControl(scenario.Assertions, scenario.Logger):
    def __init__(
            self,
            name,  # type: str
    ):  # type: (...) -> None
        scenario.Assertions.__init__(self)
        scenario.Logger.__init__(self, name)

    def type(
            self,
            text,  # type: str
    ):  # type: (...) -> None
        self.info(f"Typing text {text!r}")

    def click(self):  # type: (...) -> None
        self.info("Clicking on the button")

    def gettext(self):  # type: (...) -> str
        return "<p>Hello john!</p>"


def body():  # type: (...) -> HtmlControl
    return HtmlControl("body")


def getedit(
        id,  # type: str
):  # type: (...) -> HtmlControl
    return HtmlControl(f"edit[@id={id!r}]")


def getbutton(
        id,  # type: str
):  # type: (...) -> HtmlControl
    return HtmlControl(f"edit[@id={id!r}]")
```

Memo: Debugging is disabled by default for class loggers.

4.8 Handlers

The *scenario* framework provides a service that triggers handlers on events.

4.8.1 Handler registration

First of all, declare your handler function:

```
def _error(event, data):
    assert isinstance(data, scenario.EventData.Error)
    scenario.logging.debug(f"{event!r} handler called with error {data.error!r}")
```

Then the handlers may be installed by calling the *Handlers.install()* method on the *scenario.handlers* manager:

```
        scenario.handlers.install(scenario.Event.ERROR, _error)
```

Tip: The event may be passed as an enum constant or a string.

Using enums is usually a better option in as much as they can be type checked in camparison with simple strings.

The *Handlers.install()* method has a couple of parameters that specialize the way the handlers live and are triggered:

scenario
: Related scenario, if any.

  When this reference is set, the handler will be triggered only if the current scenario is the given scenario.

once
: *Once* flag.

  Makes this scenario be triggered once, and then uninstalled.

first
: Make this handler be called prior to other handlers when the event is met, otherwise the handler is called after the other handlers already registered for the given event.

The handlers may be uninstalled thanks to the *Handlers.uninstall()* method.

4.8.2 Scenario events

The following tables describes the *scenario* events that can be used to register handlers for.

Tip: Use the *scenario.Event* shortcut to the internal *ScenarioEvent* enum from *scenario* user code.

Table 2: Events raised during a scenario execution

Event	Description	Data type
ScenarioEvent.BEFORE_TEST = "scenario.before-test"	*Before test* handlers: handlers that are executed at the beginning of the scenario.	*ScenarioEventData.Scenario*
ScenarioEvent.BEFORE_STEP = "scenario.before-step"	*Before step* handlers: handlers that are executed before each step.	*ScenarioEventData.Step*
ScenarioEvent.ERROR = "scenario.error"	Error handler: handlers that are executed on test errors.	*ScenarioEventData.Error*
ScenarioEvent.AFTER_STEP = "scenario.after-step"	*After step* handlers: handlers that are executed after each step.	*ScenarioEventData.Step*
ScenarioEvent.AFTER_TEST = "scenario.after-test"	*After test* handlers: handlers that are executed at the end of the scenario.	*ScenarioEventData.Scenario*

Table 3: Events raised during a campaign execution

Event	Description	Data type
ScenarioEvent.BEFORE_CAMPAIGN = "scenario.before-campaign"	*Before campaign* handlers: handlers that are executed at the beginning of the campaign.	*ScenarioEventData.Campaign*
ScenarioEvent.BEFORE_TEST_SUITE = "scenario.before-test-suite"	*Before test suite* handlers: handlers that are executed at the beginning of each test suite.	*ScenarioEventData.TestSuite*
ScenarioEvent.BEFORE_TEST_CASE = "scenario.before-test-case"	*Before test case* handlers: handlers that are executed at the beginning of each test case.	*ScenarioEventData.TestCase*
ScenarioEvent.ERROR = "scenario.error"	Error handler: handlers that are executed on test errors.	*ScenarioEventData.Error*
ScenarioEvent.AFTER_TEST_CASE = "scenario.after-test-case"	*After test case* handlers: handlers that are executed after each test case.	*ScenarioEventData.TestCase*
ScenarioEvent.AFTER_TEST_SUITE = "scenario.after-test-suite"	*After test suite* handlers: handlers that are executed after each test suite.	*ScenarioEventData.TestSuite*
ScenarioEvent.AFTER_CAMPAIGN = "scenario.after-campaign"	*After campaign* handlers: handlers that are executed after the campaign.	*ScenarioEventData.Campaign*

4.8.3 User events

Even though the handler service is basically provided to let user code react on *scenario events*, it is made as a general feature so that it can be used for other purposes. This way, you may define your own set of events within your test environment for instance.

In order to do so, a good practice is to define your set of events with an enum, so that they can be type checked.

```
class UserEvent(enum.Enum):
    FOO = "foo"
```

Then use the *Handlers.callhandlers()* method to make the registered handlers (matching their additional conditions in option) be called. Pass on event data as a single objet, which can be whatever you want.

```
scenario.handlers.callhandlers(UserEvent.FOO, {"a": 1, "b": "bar"})
```

Tip: Considering evolutivity concerns, event data should rather be set with:

- dedicated objects, like *ScenarioEventData* proposes,
- or more informal dictionaries, like the 'demo/handlers.py' sample does.

4.9 Configuration database

The *scenario* framework provides a general configuration database.

It is available through the *scenario.conf* attribute.

4.9.1 Configuration nodes

The database configuration is a tree of sections, sub-sections, sub-sub-sections, ... ending with final values.

The *ConfigNode* class describes a node in the resulting configuration tree, either a section or a final value.

4.9.2 Configuration tree & keys

Configuration keys are a simplified form of JSONPath: they are dot-separated strings, with the ability to index a single list item with a number between square brackets.

With the following sample data:

```
a:
  b:
    c: 55
x:
  y:
  - z: 100
  - z: 101
  - z: 102
  - z: 104
```

- *"a.b.c"* points to the 55 value,
- *"x.y[2].z"* points to the 102 value,
- *"a.b"* points to the so-named sub-section (the corresponding data being a `dict`),
- *"x.y"* points to the so-named list (the corresponding data being a `list`),
- *""* (empty string) points to root node.

Tip: Configuration keys may be passed as strings or string enums.

4.9.3 Loading and setting configurations through the command line

Configuration values are basically set through the command line, with the `--config-file` and/or `--config-value` options of test and campaign launchers.

```
usage: run-test.py [-h] [--config-file CONFIG_PATH] [--config-value KEY VALUE]
                   [--debug-class DEBUG_CLASS] [--doc-only]
                   [--issue-level-error ISSUE_LEVEL]
                   [--issue-level-ignored ISSUE_LEVEL]
                   [--json-report JSON_REPORT_PATH]
                   [--extra-info ATTRIBUTE_NAME]
                   SCENARIO_PATH [SCENARIO_PATH ...]

Scenario test execution.

positional arguments:
  SCENARIO_PATH         Scenario script(s) to execute.

optional arguments:
  -h, --help            Show this help message and exit.
  --config-file CONFIG_PATH
                        Input configuration file path. This option may be
                        called several times.
  --config-value KEY VALUE
                        Single configuration value. This option may be called
                        several times.
  --debug-class DEBUG_CLASS
                        Activate debugging for the given class.
  --doc-only            Generate documentation without executing the test(s).
  --issue-level-error ISSUE_LEVEL
                        Define the issue level from and above which known
                        issues should be considered as errors. None by
                        default, i.e. all known issues are considered as
                        warnings.
  --issue-level-ignored ISSUE_LEVEL
                        Define the issue level from and under which known
                        issues should be ignored. None by default, i.e. no
                        known issue ignored by default.
  --json-report JSON_REPORT_PATH
                        Save the report in the given JSON output file path.
                        Single scenario only.
  --extra-info ATTRIBUTE_NAME
                        Scenario attribute to display for extra info when
                        displaying results. Applicable when executing several
                        tests. This option may be called several times to
                        display more info.
```

Configuration files can be in one of the following formats:

Table 4: Configuration file formats

Format	File extensions
INI (as handled by configparser)	*.ini*, *.INI*
JSON	*.json*, *.JSON*
YAML (requires PyYAML to be installed)	*.yaml*, *.yml*, *.YAML*, *.YML*

Several files may be loaded consecutively by repeating the `--config-file` option.

This makes it possible to split configuration files on various purposes:

- the kind of software / system under test,
- the environment used to execute the tests,
- the identity of the person who launches the tests,
- ...

The configuration data from the different files is merged all together in the resulting tree, the values set from the latter files overloading the ones already set by the formers files.

Then, the single values set by the `--config-value` options finally update the configuration tree.

Boolean value conversions

When configuration values are boolean values, they may be passed as strings in one of the usual forms recognized:

***True* values**
: any non-zero integer or integer string, strings like "True", "TRUE", "true", "Yes, "YES", "yes", "Y", "y"

***False* values**
: 0 (zero) or "0", strings like "False", "FALSE", "false", "No", "NO", "no", "N", "n"

4.9.4 Manipulating configurations from the code

The code can then access configuration values (resp. *ConfigNode* instances) through the *ConfigDatabase.get()* method (resp. *ConfigDatabase.getnode()*).

```
# Access a final value (`None` if the value does not exist).
_any = scenario.conf.get("a.b.c")  # type: typing.Optional[typing.Any]
# Access a final value of the given type (`None` if the value does not exist).
_int1 = scenario.conf.get("a.b.c", type=int)  # type: typing.Optional[int]
# Access a final value, or fallback to a default value.
_int2 = scenario.conf.get("a.b.c", type=int, default=100)  # type: int
# Access a whole section as a JSON dictionary (`None` if the section does not exist).
_section = scenario.conf.get("a", type=dict)  # type: typing.Optional[typing.Dict[str,␣
↪typing.Any]]
# Access a whole list as a JSON list (`None` if the list does not exist).
_list = scenario.conf.get("x.y", type=list)  # type: typing.Optional[typing.List[typing.
↪Any]]
```

The configuration keys available can be listed with the *ConfigDatabase.getkeys()* method.

Configuration files can be loaded from the code (see *ConfigDatabase.loadfile()*).

```
# Load a configuration file.
scenario.conf.loadfile("demo/conf.yml")
```

Configuration data can also be set (either sections or lists or single values, see *ConfigDatabase.set()*).

```
# Set a single value.
scenario.conf.set("a.b.c", 150)
# Update a whole section (possibly with sub-sections).
scenario.conf.set("a.b", {"c": 200})
scenario.conf.set("a", {"b": {"c": 200}})
```

Automatic configuration data conversions

When setting data from the code, the configuration database applies the following conversions:

Table 5: Automatic configuration data conversions

Input data type	Storage
Path-likes	`str` form of the path, using `os.fspath()`
`enum.EnumMeta`	`list`
`enum.IntEnum`	`int` form of the enum value
Other `enum.Enum`	`str` form of the enum value

Configuration nodes can be accessed directly from the code, and provide an API that can be used from the user code (see *ConfigNode*).

```
# Access a configuration node (`None` if the node does not exist).
_node = scenario.conf.getnode("a.b.c")  # type: typing.Optional[scenario.ConfigNode]
```

4.9.5 Configuration origins

In case configurations lead to some erroneous situation, the configuration database keeps memory of *configuration origins* (see *ConfigNode.origins* and *ConfigNode.origin*).

This information can help a user fix his/her configuration files when something goes wrong.

For the purpose, the *ConfigNode.errmsg()* method builds error messages giving the representative origin of the given configuration node.

The *ConfigDatabase.show()* and *ConfigNode.show()* methods also display the configuration tree with origins.

```
$ ./demo/run-demo.py --config-file demo/conf.json --config-value x.y[0].z 0 --show-
↪configs demo/htmllogin.py
```

```
INFO     Main path: '/path/to/scenario'
INFO     [scenario.ConfigDatabase]   a:
INFO     [scenario.ConfigDatabase]     a.b:
INFO     [scenario.ConfigDatabase]       a.b.c: 55  # from demo/conf.json
INFO     [scenario.ConfigDatabase]   x:
INFO     [scenario.ConfigDatabase]     x.y[0]:
INFO     [scenario.ConfigDatabase]       x.y[0].z: '0'  # from <args>
INFO     [scenario.ConfigDatabase]     x.y[1]:
INFO     [scenario.ConfigDatabase]       x.y[1].z: 100  # from demo/conf.json
INFO     [scenario.ConfigDatabase]     x.y[2]:
INFO     [scenario.ConfigDatabase]       x.y[2].z: 101  # from demo/conf.json
INFO     [scenario.ConfigDatabase]     x.y[3]:
```

(continues on next page)

(continued from previous page)

```
INFO     [scenario.ConfigDatabase]       x.y[3].z: 102  # from demo/conf.json
INFO     [scenario.ConfigDatabase]     x.y[4]:
INFO     [scenario.ConfigDatabase]       x.y[4].z: 103  # from demo/conf.json
```

4.9.6 *scenario* configurable keys and values

The following table describes the *scenario* configurable keys & values.

Tip: Use the *scenario.ConfigKey* shortcut to the internal *ScenarioConfig.Key* enum from *scenario* user code.

Table 6: Scenario configurable keys and values

Key		Type	Description	Default
ScenarioConfig.Key.TIMEZONE	`scenario.timezone`	String	Timezone specification. Possible values: 'UTC', 'Z', or numerical forms like '+01:00', '-05:00'. More options *when pytz is installed*: 'CET', 'US/Pacific', 'Japan', ... Execute the following Python code for the complete list: `import pytz print("\n".join(pytz.all_timezones))`	Not set, i.e. use of the local timezone
ScenarioConfig.Key.LOG_DATETIME	`scenario.log_date_time`	Boolean	Should the log lines include a timestamp?	Enabled
ScenarioConfig.Key.LOG_CONSOLE	`scenario.log_console`	Boolean	Should the log lines be displayed in the console?	Enabled
ScenarioConfig.Key.LOG_COLOR_ENABLED	`scenario.log_color`	Boolean	Should the log lines be colored?	Enabled
ScenarioConfig.Key.LOG_COLOR	`scenario.log_%(level)_color`, `%(level)` being one of (`error`, `warning`, `info`, `debug`)	Integer	Console color code per log level. See *Console.Color* for a list useful color codes.	scenario.log_error_color: red(91), scenario.log_warning_color: yellow(33), scenario.log_info_color: white(1), scenario.log_debug_color: dark grey(2)
ScenarioConfig.Key.LOG_FILE	`scenario.log_file`	File path string	Should the log lines be written in a log file?	Not set, i.e. no file logging
ScenarioConfig.Key.DEBUG_CLASSES	`scenario.debug_classes`	List of strings (or comma-separated string)	Which debug classes to display?	Not set
ScenarioConfig.Key.EXPECTED_ATTRIBUTES	`scenario.expected_attributes`	List of strings (or comma-separated string)	Expected scenario attributes.	Not set
ScenarioConfig.Key.CONTINUE_ON_ERROR	`scenario.continue_on_error`	Boolean	Should the scenarios continue on error? If set to `True`, an error ends the current step, but following steps are still executed. The same behaviour may	Disabled

4.10 Step objects: instanciate steps and sequence them as scenarios

The *quickstart* showed how to quickly write a first test scenario using `step...()` methods.

However, test code reuse can hardly be achieved with step methods. In order to be able to reuse steps between different scenarios, it is better defining them as classes, inheriting from *scenario.Step*.

Todo: Documentation needed for steps as objects.

4.10.1 Alternative scenarios

Todo: Documentation needed for alternative scenarios.

4.11 Subscenarios: reuse existing scenarios in other scenarios

Scenarios can be reused as subscenarios in other ones.

Executing existing scenarios as sub-scenarios are particularly useful for the following purposes:

- define alternative scenarios (error scenarios) from a nominal one,
- reuse a nominal scenario as the initial condition of other ones, in order to bring the system or software under test in the expected initial state,
- repeat a base scenario with varying input data.

4.11.1 Initial conditions

Todo: Documentation needed for initial conditions.

4.11.2 Varying input data

Todo: Improve subscenario documentation with a better example.

In order to illustrate this use case of subscenarios, let's get back to the previous `CommutativeAddition` scenario defined *previously*.

The `CommutativeAddition` scenario can be called multiple times, with different inputs, in a super `CommutativeAdditions` scenario:

```
1 # -*- coding: utf-8 -*-
2
3 import pathlib
4 import scenario
```

(continues on next page)

(continued from previous page)

```
import sys

sys.path.append(str(pathlib.Path(__file__).parent))
from commutativeaddition import CommutativeAddition  # noqa: E402


class CommutativeAdditions(scenario.Scenario):

    SHORT_TITLE = "Commutative additions"
    TEST_GOAL = "Call the CommutativeAddition scenario with different inputs."

    def step010(self):
        self.STEP("Both positive members")

        if self.ACTION("Launch the CommutativeAddition scenario with 4 and 5 for inputs.
↪"):
            _scenario = CommutativeAddition(4, 5)
            scenario.runner.executescenario(_scenario)

    def step020(self):
        self.STEP("Positive and negative members")

        if self.ACTION("Launch the CommutativeAddition scenario with -1 and 3 for inputs.
↪"):
            _scenario = CommutativeAddition(-1, 3)
            scenario.runner.executescenario(_scenario)

    def step030(self):
        self.STEP("Both negative members")

        if self.ACTION("Launch the CommutativeAddition scenario with -1 and -7 for␣
↪inputs."):
            _scenario = CommutativeAddition(-1, -7)
            scenario.runner.executescenario(_scenario)
```

To do so, start with loading your base scenario with a regular `import` statement:

```
from commutativeaddition import CommutativeAddition  # noqa: E402
```

Instanciate it with the appropriate values:

```
            _scenario = CommutativeAddition(4, 5)
```

And execute it as a subscenario:

```
            scenario.runner.executescenario(_scenario)
```

Executing this super scenario produces the following output:

```
$ ./bin/run-test.py ./demo/commutativeadditions.py
```

```
SCENARIO 'demo/commutativeadditions.py'
-----------------------------------------------
```

(continues on next page)

(continued from previous page)

```
STEP#1: Both positive members (demo/commutativeadditions.py:16:CommutativeAdditions.
↪step010)
-------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with 4 and 5 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = 4, and b = 5
      |   EVIDENCE:   -> a = 4
      |   EVIDENCE:   -> b = 5
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -> result1 = 9
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -> result2 = 9
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -> 9 == 9
      |
      | END OF 'demo/commutativeaddition.py'

STEP#2: Positive and negative members (demo/commutativeadditions.
↪py:23:CommutativeAdditions.step020)
-------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with -1 and 3 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = -1, and b = 3
      |   EVIDENCE:   -> a = -1
      |   EVIDENCE:   -> b = 3
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -> result1 = 2
```

(continues on next page)

(continued from previous page)

```
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -> result2 = 2
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -> 2 == 2
      |
      | END OF 'demo/commutativeaddition.py'

STEP#3: Both negative members (demo/commutativeadditions.py:30:CommutativeAdditions.
↪step030)
------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with -1 and -7 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = -1, and b = -7
      |   EVIDENCE:   -> a = -1
      |   EVIDENCE:   -> b = -7
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -> result1 = -8
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -> result2 = -8
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -> -8 == -8
      |
      | END OF 'demo/commutativeaddition.py'

END OF 'demo/commutativeadditions.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 3/3
  Number of ACTIONs: 3/3
  Number of RESULTs: 0/0
```

(continues on next page)

(continued from previous page)

```
                Time: HH:MM:SS.mmmmmm
```

Each subscenario execution appears indented with a pipe character.

Subscenario nestings

A subscenario may call other subscenarios.

For each subscenario in the execution stack, a pipe indentation is inserted in the log lines, in order to highlight the scenario and subscenario execution nestings.

4.12 Goto

Todo: Documentation needed for the goto feature.

4.13 Multiple scenario executions

As described by the scenario launcher help message, several scenarios may be executed with a single command line.

```
usage: run-test.py [-h] [--config-file CONFIG_PATH] [--config-value KEY VALUE]
                   [--debug-class DEBUG_CLASS] [--doc-only]
                   [--issue-level-error ISSUE_LEVEL]
                   [--issue-level-ignored ISSUE_LEVEL]
                   [--json-report JSON_REPORT_PATH]
                   [--extra-info ATTRIBUTE_NAME]
                   SCENARIO_PATH [SCENARIO_PATH ...]

Scenario test execution.

positional arguments:
  SCENARIO_PATH         Scenario script(s) to execute.

optional arguments:
  -h, --help            Show this help message and exit.
  --config-file CONFIG_PATH
                        Input configuration file path. This option may be
                        called several times.
  --config-value KEY VALUE
                        Single configuration value. This option may be called
                        several times.
  --debug-class DEBUG_CLASS
                        Activate debugging for the given class.
  --doc-only            Generate documentation without executing the test(s).
  --issue-level-error ISSUE_LEVEL
                        Define the issue level from and above which known
                        issues should be considered as errors. None by
```

(continues on next page)

(continued from previous page)

```
                        default, i.e. all known issues are considered as
                        warnings.
  --issue-level-ignored ISSUE_LEVEL
                        Define the issue level from and under which known
                        issues should be ignored. None by default, i.e. no
                        known issue ignored by default.
  --json-report JSON_REPORT_PATH
                        Save the report in the given JSON output file path.
                        Single scenario only.
  --extra-info ATTRIBUTE_NAME
                        Scenario attribute to display for extra info when
                        displaying results. Applicable when executing several
                        tests. This option may be called several times to
                        display more info.
```

For example:

```
$ ./bin/run-test.py demo/commutativeaddition.py demo/loggingdemo.py
```

Option restriction

When executing several scenarios in the same command line, a couple of options come to be not applicable, such as `--json-report`.

The tests are executed one after the other, in the order given by the command line.

A summary of the scenario executions is given in the end.

```
-----------------------------------------------
INFO     TOTAL                                  Status      Steps     Actions    Results ␣
↪  Time
INFO     2 tests, 0 failed, 0 with warnings                   9/9       31/31        1/1 ␣
↪  HH:MM:SS.mmmmmm
-----------------------------------------------
INFO     demo/commutativeaddition.py            SUCCESS       4/4         4/4        1/1 ␣
↪  HH:MM:SS.mmmmmm
INFO     demo/loggingdemo.py                    SUCCESS       5/5       27/27        0/0 ␣
↪  HH:MM:SS.mmmmmm
```

4.14 Campaigns

Campaigns shall be launched with the 'run-campaign.py' script.

```
usage: run-campaign.py [-h] [--config-file CONFIG_PATH]
                       [--config-value KEY VALUE] [--debug-class DEBUG_CLASS]
                       [--doc-only] [--issue-level-error ISSUE_LEVEL]
                       [--issue-level-ignored ISSUE_LEVEL]
                       [--outdir OUTDIR_PATH] [--dt-subdir]
                       [--extra-info ATTRIBUTE_NAME]
                       TEST_SUITE_PATH [TEST_SUITE_PATH ...]
```

(continues on next page)

(continued from previous page)

```
Scenario campaign execution.

positional arguments:
  TEST_SUITE_PATH       Test suite file(s) to execute.

optional arguments:
  -h, --help            Show this help message and exit.
  --config-file CONFIG_PATH
                        Input configuration file path. This option may be
                        called several times.
  --config-value KEY VALUE
                        Single configuration value. This option may be called
                        several times.
  --debug-class DEBUG_CLASS
                        Activate debugging for the given class.
  --doc-only            Generate documentation without executing the test(s).
  --issue-level-error ISSUE_LEVEL
                        Define the issue level from and above which known
                        issues should be considered as errors. None by
                        default, i.e. all known issues are considered as
                        warnings.
  --issue-level-ignored ISSUE_LEVEL
                        Define the issue level from and under which known
                        issues should be ignored. None by default, i.e. no
                        known issue ignored by default.
  --outdir OUTDIR_PATH  Output directory to store test results into. Defaults
                        to the current directory.
  --dt-subdir           Do not store the test results directly in OUTDIR_PATH,
                        but within a subdirectory named with the current date
                        and time.
  --extra-info ATTRIBUTE_NAME
                        Scenario attribute to display for extra info when
                        displaying results. This option may be called several
                        times to display more info.
```

4.14.1 Test suite files

Test suite files are text files that describe the scenario files to execute, or not to execute.

Example from the demo.suite test suite file:

```
# This is a sample campaign description file.

# White-list:
+ *.py

# Black-list:
- htmltestlib.py
- run-demo.py
```

Table 7: Test suite files syntax

Type of line	Syntax	Effects
Comment	Starts with a '#' character.	No effect.
White list	Starts with a '+' character, or no starter character, followed by a path or a glob-style pattern.	Describes one (or several) script path(s) of scenario(s) to execute. When the path is relative, it is computed from the test suite file directory.
Black list	Starts with a '-' character, followed by a path or a glob-style pattern.	Describes one (or several) script path(s) to remove from the white list constituted by the preceding lines. When the path is relative, it is computed from the test suite file directory.

Tip: White-list lines after a black-list line may restore script paths avoided by the latter.

4.14.2 Campaign execution

Test suite files are executed one after the others, in the order given by the command line.

A summary of the tests executed is displayed in the end.

```
$ ./bin/run-campaign.py demo/demo.suite
```

```
CAMPAIGN
------------------------------------------------
  TEST SUITE 'demo/demo.suite'
  ---------------------------------------------
    Executing 'demo/commutativeaddition.py'
      DEBUG    Log file:    'out/commutativeaddition.log'
      DEBUG    JSON report: 'out/commutativeaddition.json'
    Executing 'demo/commutativeadditions.py'
      DEBUG    Log file:    'out/commutativeadditions.log'
      DEBUG    JSON report: 'out/commutativeadditions.json'
    Executing 'demo/handlers.py'
      DEBUG    Log file:    'out/handlers.log'
      DEBUG    JSON report: 'out/handlers.json'
    Executing 'demo/htmllogin.py'
      DEBUG    Log file:    'out/htmllogin.log'
      DEBUG    JSON report: 'out/htmllogin.json'
    Executing 'demo/loggingdemo.py'
      DEBUG    Log file:    'out/loggingdemo.log'
      DEBUG    JSON report: 'out/loggingdemo.json'

  END OF TEST SUITE 'demo/demo.suite'
  ---------------------------------------------
             Number of test cases: 5
         Number of tests in error: 0
    Number of tests with warnings: 0
```

(continues on next page)

(continued from previous page)

```
                  Number of steps: 15/15
                Number of actions: 39/39
                Number of results: 2/2
                             Time: HH:MM:SS.mmmmmm

END OF CAMPAIGN
-----------------------------------------------
          Number of test suites: 1

DEBUG    JUnit report: 'out/campaign.xml'
-----------------------------------------------
INFO     TOTAL                                  Status      Steps     Actions    Results ␣
↪  Time
INFO     5 tests, 0 failed, 0 with warnings                 15/15       39/39        2/2 ␣
↪  HH:MM:SS.mmmmmm
-----------------------------------------------
INFO     demo/commutativeaddition.py           SUCCESS        4/4         4/4        1/1 ␣
↪  HH:MM:SS.mmmmmm
INFO     demo/commutativeadditions.py          SUCCESS        3/3         3/3        0/0 ␣
↪  HH:MM:SS.mmmmmm
INFO     demo/handlers.py                      SUCCESS        2/2         2/2        0/0 ␣
↪  HH:MM:SS.mmmmmm
INFO     demo/htmllogin.py                     SUCCESS        1/1         3/3        1/1 ␣
↪  HH:MM:SS.mmmmmm
INFO     demo/loggingdemo.py                   SUCCESS        5/5       27/27        0/0 ␣
↪  HH:MM:SS.mmmmmm
```

4.14.3 Campaign reports

The `--outdir` option specifies the directory where the execution reports should be stored.

`--dt-subdir` option

In conjonction with it, the `--dt-subdir` option tells the 'run-campaign.py' launcher to create a date/time subdirectory in the output directory.

The date/time subdirectory is formed on the 'YYYY-MM-DD_HH-MM-SS' pattern.

For each scenario executed, a *JSON report* is stored in the output directory.

Eventually, a campaign report is generated in the XML JUnit format.

```
<?xml version="1.0" encoding="utf-8"?>
<testsuites actions-executed="39" actions-total="39" disabled="0" errors="0" failures="0
↪" results-executed="2" results-total="2" steps-executed="15" steps-total="15" tests="5
↪" time="SSS.mmmmmm">
        <testsuite actions-executed="39" actions-total="39" disabled="0" errors="0"␣
↪failures="0" id="0" name="demo/demo.suite" results-executed="2" results-total="2"␣
↪skipped="0" steps-executed="15" steps-total="15" tests="5" time="SSS.mmmmmm" timestamp=
↪"YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM">
                <testcase actions-executed="4" actions-total="4" classname="demo/
```

(continues on next page)

(continued from previous page)

```
↪commutativeaddition.py" name="demo/commutativeaddition.py" results-executed="1"␣
↪results-total="1" status="SUCCESS" steps-executed="4" steps-total="4" time="SSS.mmmmmm
↪">
                        <link href="out/commutativeaddition.log" rel="log" type="text/
↪plain"/>
                        <link href="out/commutativeaddition.json" rel="report" type=
↪"application/json"/>
                        <system-out>SCENARIO 'demo/commutativeaddition.py'
------------------------------------------------


STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.step000)
------------------------------------------------
    ACTION: Let a = 1, and b = 3
  EVIDENCE:   -&gt; a = 1
  EVIDENCE:   -&gt; b = 3

STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
------------------------------------------------
    ACTION: Compute (a + b) and store the result as result1.
  EVIDENCE:   -&gt; result1 = 4

STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
------------------------------------------------
    ACTION: Compute (b + a) and store the result as result2.
  EVIDENCE:   -&gt; result2 = 4

STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
------------------------------------------------
    ACTION: Compare result1 and result2.
    RESULT: result1 and result2 are the same.
  EVIDENCE:   -&gt; 4 == 4

END OF 'demo/commutativeaddition.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 4/4
  Number of ACTIONs: 4/4
  Number of RESULTs: 1/1
               Time: HH:MM:SS.mmmmmm

</system-out>
                </testcase>
                <testcase actions-executed="3" actions-total="3" classname="demo/
↪commutativeadditions.py" name="demo/commutativeadditions.py" results-executed="0"␣
↪results-total="0" status="SUCCESS" steps-executed="3" steps-total="3" time="SSS.mmmmmm
↪">
                        <link href="out/commutativeadditions.log" rel="log" type="text/
↪plain"/>
                        <link href="out/commutativeadditions.json" rel="report" type=
↪"application/json"/>
                        <system-out>SCENARIO 'demo/commutativeadditions.py'
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------


STEP#1: Both positive members (demo/commutativeadditions.py:16:CommutativeAdditions.
↪step010)
------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with 4 and 5 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = 4, and b = 5
      |   EVIDENCE:   -&gt; a = 4
      |   EVIDENCE:   -&gt; b = 5
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -&gt; result1 = 9
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -&gt; result2 = 9
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -&gt; 9 == 9
      |
      | END OF 'demo/commutativeaddition.py'

STEP#2: Positive and negative members (demo/commutativeadditions.
↪py:23:CommutativeAdditions.step020)
------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with -1 and 3 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = -1, and b = 3
      |   EVIDENCE:   -&gt; a = -1
      |   EVIDENCE:   -&gt; b = 3
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
```

(continues on next page)

(continued from previous page)

```
      |   EVIDENCE:   -&gt; result1 = 2
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -&gt; result2 = 2
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -&gt; 2 == 2
      |
      | END OF 'demo/commutativeaddition.py'

STEP#3: Both negative members (demo/commutativeadditions.py:30:CommutativeAdditions.
↪step030)
------------------------------------------------
    ACTION: Launch the CommutativeAddition scenario with -1 and -7 for inputs.
      | SCENARIO 'demo/commutativeaddition.py'
      | ------------------------------------------------
      |
      | STEP#1: Initial conditions (demo/commutativeaddition.py:18:CommutativeAddition.
↪step000)
      | ------------------------------------------------
      |     ACTION: Let a = -1, and b = -7
      |   EVIDENCE:   -&gt; a = -1
      |   EVIDENCE:   -&gt; b = -7
      |
      | STEP#2: a + b (demo/commutativeaddition.py:25:CommutativeAddition.step010)
      | ------------------------------------------------
      |     ACTION: Compute (a + b) and store the result as result1.
      |   EVIDENCE:   -&gt; result1 = -8
      |
      | STEP#3: b + a (demo/commutativeaddition.py:32:CommutativeAddition.step020)
      | ------------------------------------------------
      |     ACTION: Compute (b + a) and store the result as result2.
      |   EVIDENCE:   -&gt; result2 = -8
      |
      | STEP#4: Check (demo/commutativeaddition.py:39:CommutativeAddition.step030)
      | ------------------------------------------------
      |     ACTION: Compare result1 and result2.
      |     RESULT: result1 and result2 are the same.
      |   EVIDENCE:   -&gt; -8 == -8
      |
      | END OF 'demo/commutativeaddition.py'

END OF 'demo/commutativeadditions.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 3/3
  Number of ACTIONs: 3/3
```

(continues on next page)

(continued from previous page)

```
  Number of RESULTs: 0/0
                Time: HH:MM:SS.mmmmmm

</system-out>
                </testcase>
                <testcase actions-executed="2" actions-total="2" classname="demo/
↪handlers.py" name="demo/handlers.py" results-executed="0" results-total="0" status=
↪"SUCCESS" steps-executed="2" steps-total="2" time="SSS.mmmmmm">
                        <link href="out/handlers.log" rel="log" type="text/plain"/>
                        <link href="out/handlers.json" rel="report" type="application/
↪json"/>
                        <system-out>SCENARIO 'demo/handlers.py'
------------------------------------------------

DEBUG    'scenario.before-test' handler called with test &lt;Handlers 'demo/handlers.py'&
↪gt;
DEBUG    'scenario.before-step' handler called with step &lt;StepDefinition 'Handlers.
↪step010'&gt;

STEP#1: `UserEvent.FOO` event triggering (demo/handlers.py:55:Handlers.step010)
------------------------------------------------
    ACTION: Trigger the `UserEvent.FOO` event, with the following parameters: a=1 and b=
↪'bar'.
              DEBUG    'foo' handler called with {'a': 1, 'b': 'bar'}
DEBUG    'scenario.after-step' handler called with step &lt;StepDefinition 'Handlers.
↪step010'&gt;
DEBUG    'scenario.before-step' handler called with step &lt;StepDefinition 'Handlers.
↪step020'&gt;

STEP#2: `UserEvent.FOO` event triggering (demo/handlers.py:61:Handlers.step020)
------------------------------------------------
    ACTION: Trigger the `UserEvent.FOO` event, with the following parameters: a=2 and b=
↪'baz'.
              DEBUG    'foo' handler called with {'a': 2, 'b': 'baz'}
DEBUG    'scenario.after-step' handler called with step &lt;StepDefinition 'Handlers.
↪step020'&gt;
DEBUG    'scenario.after-test' handler called with test &lt;Handlers 'demo/handlers.py'&
↪gt;

END OF 'demo/handlers.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 2/2
  Number of ACTIONs: 2/2
  Number of RESULTs: 0/0
                Time: HH:MM:SS.mmmmmm

</system-out>
                </testcase>
                <testcase actions-executed="3" actions-total="3" classname="demo/
↪htmllogin.py" name="demo/htmllogin.py" results-executed="1" results-total="1" status=
↪"SUCCESS" steps-executed="1" steps-total="1" time="SSS.mmmmmm">
```

(continues on next page)

(continued from previous page)

```
                        <link href="out/htmllogin.log" rel="log" type="text/plain"/>
                        <link href="out/htmllogin.json" rel="report" type="application/
↪json"/>
                        <system-out>SCENARIO 'demo/htmllogin.py'
------------------------------------------------


STEP#1: Login screen (demo/htmllogin.py:13:TestLoginPage.step010_loginscreen)
------------------------------------------------
    ACTION: Type the login.
              INFO     [edit[@id='login']] Typing text 'john'
    ACTION: Type the password.
              INFO     [edit[@id='password']] Typing text '0000'
    ACTION: Click on the OK button.
              INFO     [edit[@id='submit']] Clicking on the button
    RESULT: The login page says hello to the user.

END OF 'demo/htmllogin.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 1/1
  Number of ACTIONs: 3/3
  Number of RESULTs: 1/1
               Time: HH:MM:SS.mmmmmm

</system-out>
                </testcase>
                <testcase actions-executed="27" actions-total="27" classname="demo/
↪loggingdemo.py" name="demo/loggingdemo.py" results-executed="0" results-total="0"␣
↪status="SUCCESS" steps-executed="5" steps-total="5" time="SSS.mmmmmm">
                        <link href="out/loggingdemo.log" rel="log" type="text/plain"/>
                        <link href="out/loggingdemo.json" rel="report" type="application/
↪json"/>
                        <system-out>SCENARIO 'demo/loggingdemo.py'
------------------------------------------------


STEP#1: Logging with the main logger (demo/loggingdemo.py:22:LoggingScenario.step010)
------------------------------------------------
    ACTION: Log messages of different log levels with the main logger.
              ERROR    This is an error!!!
              WARNING  This is a warning!
              INFO     This is information.
              DEBUG    This is debug.

STEP#2: Logging with the scenario instance (demo/loggingdemo.py:31:LoggingScenario.
↪step020)
------------------------------------------------
    ACTION: Log messages of different log levels with the scenario itself.
              ERROR    [demo/loggingdemo.py] This is an error!!!
              WARNING  [demo/loggingdemo.py] This is a warning!
              INFO     [demo/loggingdemo.py] This is information.
```

(continues on next page)

(continued from previous page)

```
              DEBUG    [demo/loggingdemo.py] This is debug.

STEP#3: Logging with a class logger (demo/loggingdemo.py:40:LoggingScenario.step030)
-------------------------------------------------
    ACTION: Log messages of different log levels with the class logger instance.
              ERROR    [My logger] This is an error!!!
              WARNING  [My logger] This is a warning!
              INFO     [My logger] This is information.
    ACTION: Activate debugging for the class logger instance.
    ACTION: Log a debug message again with the class logger instance.
              DEBUG    [My logger] This is debug again.

STEP#4: Class logger indentation (demo/loggingdemo.py:55:LoggingScenario.step110)
-------------------------------------------------
    ACTION: Log something with the class logger.
              INFO     [My logger] Hello
    ACTION: Push indentation to the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]     Hello
    ACTION: Push indentation to the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]         Hello
    ACTION: Push indentation to the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]             Hello
    ACTION: Pop indentation from the class logger.
    ACTION: Log something with the class logger.
              INFO     [My logger]         Hello
    ACTION: Reset the class logger indentation.
    ACTION: Log something with the class logger.
              INFO     [My logger] Hello

STEP#5: Main logger indentation (demo/loggingdemo.py:76:LoggingScenario.step120)
-------------------------------------------------
    ACTION: Log something with the main logger.
              INFO     Hello
    ACTION: Push indentation to the main logger.
    ACTION:     Log something with the main logger.
                  INFO     Hello
    ACTION:     Push indentation to the main logger.
    ACTION:         Log something with the main logger.
                      INFO     Hello
    ACTION:         Push indentation to the main logger.
    ACTION:             Log something with the main logger.
                          INFO     Hello
    ACTION:             Pop indentation from the main logger.
    ACTION:         Log something with the main logger.
                      INFO     Hello
    ACTION:         Reset the main logger indentation.
    ACTION: Log something with the main logger.
              INFO     Hello
```

(continues on next page)

(continued from previous page)

```
END OF 'demo/loggingdemo.py'
------------------------------------------------
             Status: SUCCESS
    Number of STEPs: 5/5
  Number of ACTIONs: 27/27
  Number of RESULTs: 0/0
               Time: HH:MM:SS.mmmmmm

</system-out>
                </testcase>
        </testsuite>
</testsuites>
```

XML JUnit format

A reference documentation could not be found for the XML JUnit format.

In spite of, the [CUBIC] page can be noted as one of the best resources on that topic.

4.15 Reports

Reports may be generated when executing a single scenario, with the `--json-report` option:

```
$ ./bin/run-test.py ./demo/commutativeaddition.py --json-report ./demo/
↪commutativeaddition.json
```

Below, the JSON output file for the *quickstart* `CommutativeAddition` sample scenario:

```
{
  "$schema": "https://github.com/alxroyer/scenario/blob/master/schema/scenario-report-v1.
↪schema.json",
  "name": "demo/commutativeaddition.py",
  "href": "demo/commutativeaddition.py",
  "attributes": {},
  "steps": [
    {
      "location": "demo/commutativeaddition.py:18:CommutativeAddition.step000",
      "description": "Initial conditions",
      "executions": [
        {
          "number": 1,
          "time": {
            "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
            "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
            "elapsed": SSS.mmmmmm
          },
          "errors": [],
          "warnings": []
        }
      ],
```

(continues on next page)

(continued from previous page)


```
      "actions-results": [
        {
          "type": "ACTION",
          "description": "Let a = 1, and b = 3",
          "executions": [
            {
              "time": {
                "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "elapsed": SSS.mmmmmm
              },
              "evidence": [
                "a = 1",
                "b = 3"
              ],
              "errors": [],
              "warnings": [],
              "subscenarios": []
            }
          ]
        }
      ]
    },
    {
      "location": "demo/commutativeaddition.py:25:CommutativeAddition.step010",
      "description": "a + b",
      "executions": [
        {
          "number": 2,
          "time": {
            "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
            "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
            "elapsed": SSS.mmmmmm
          },
          "errors": [],
          "warnings": []
        }
      ],
      "actions-results": [
        {
          "type": "ACTION",
          "description": "Compute (a + b) and store the result as result1.",
          "executions": [
            {
              "time": {
                "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "elapsed": SSS.mmmmmm
              },
              "evidence": [
                "result1 = 4"
              ],
```


(continues on next page)

(continued from previous page)


```
            "errors": [],
            "warnings": [],
            "subscenarios": []
          }
        ]
      }
    ]
  },
  {
    "location": "demo/commutativeaddition.py:32:CommutativeAddition.step020",
    "description": "b + a",
    "executions": [
      {
        "number": 3,
        "time": {
          "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
          "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
          "elapsed": SSS.mmmmmm
        },
        "errors": [],
        "warnings": []
      }
    ],
    "actions-results": [
      {
        "type": "ACTION",
        "description": "Compute (b + a) and store the result as result2.",
        "executions": [
          {
            "time": {
              "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
              "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
              "elapsed": SSS.mmmmmm
            },
            "evidence": [
              "result2 = 4"
            ],
            "errors": [],
            "warnings": [],
            "subscenarios": []
          }
        ]
      }
    ]
  },
  {
    "location": "demo/commutativeaddition.py:39:CommutativeAddition.step030",
    "description": "Check",
    "executions": [
      {
        "number": 4,
        "time": {
```


(continues on next page)

(continued from previous page)


```
            "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
            "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
            "elapsed": SSS.mmmmmm
          },
          "errors": [],
          "warnings": []
        }
      ],
      "actions-results": [
        {
          "type": "ACTION",
          "description": "Compare result1 and result2.",
          "executions": [
            {
              "time": {
                "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "elapsed": SSS.mmmmmm
              },
              "evidence": [],
              "errors": [],
              "warnings": [],
              "subscenarios": []
            }
          ]
        },
        {
          "type": "RESULT",
          "description": "result1 and result2 are the same.",
          "executions": [
            {
              "time": {
                "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
                "elapsed": SSS.mmmmmm
              },
              "evidence": [
                "4 == 4"
              ],
              "errors": [],
              "warnings": [],
              "subscenarios": []
            }
          ]
        }
      ]
    }
  ],
  "status": "SUCCESS",
  "errors": [],
  "warnings": [],
  "time": {
```


(continues on next page)

(continued from previous page)

```
    "start": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
    "end": "YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM",
    "elapsed": SSS.mmmmmm
  },
  "stats": {
    "steps": {
      "executed": 4,
      "total": 4
    },
    "actions": {
      "executed": 4,
      "total": 4
    },
    "results": {
      "executed": 1,
      "total": 1
    }
  }
}
```

Note: Dates are ISO-8601 encoded, and elapsed times are given in seconds. They are figured with the respective patterns 'YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM' and 'SSS.mmmmmm' above.

Todo: Documentation needed for campaign reports

4.16 Scenario attributes

Todo: Documentation needed for scenario attributes:

- May commonly be used to manage additional info like: test title, test objective, names of features tested.
- Expected scenario attributes.
- Best practice: use of enums.
- Best practice: overload *scenario.Scenario* with an initializer that requires your own scenario attributes.
- `--extra-info` option, applicable for test and campaign launchers.
- Extra info may be configured by default in your own launcher.

4.17 Launcher script extension

It is common that a user test environment needs to do a couple of things like:

- prepare the environment before the test execution,
- execute additional actions after the test execution,
- offer configurable features.

To do so, the user test environment may define its own launcher script, as illustrated by the demo/run-demo.py file.

4.17.1 Command line argument extension

About configurable features, *configuration files* come as a straight forward solution. Nevertheless, it is sometimes faster in use to provide command line options to the test launcher script also. To do so, our 'demo/run-demo.py' first overloads the *ScenarioArgs* class:

- The final program description is set with the *Args.setdescription()* method.
- Extra arguments may be defined thanks to the *Args.addarg()* then *ArgInfo.define()* methods.

```
class DemoArgs(scenario.ScenarioArgs):
    def __init__(self):
        scenario.ScenarioArgs.__init__(self)

        self.setdescription("Demo test launcher.")

        self.welcome_message = "Hello you!"
        self.bye_message = "Bye!"
        self.addarg("Name", "welcome_message", str).define(
            "--welcome", metavar="NAME",
            action="store", type=str,
            help="User name.",
        )

        self.show_config_db = False
        self.addarg("Show configuration database", "show_config_db", bool).define(
            "--show-configs",
            action="store_true",
            help="Show the configuration values with their origin, then stop.",
        )
```

The *Args._checkargs()* method may be overloaded in order to check additional constraints, after the arguments have been parsed, and the *Args* attributes have been updated:

- Start or finish with calling the mother class's *ScenarioArgs._checkargs()* method.
- This method is expected to return `True` or `False` whether an error has been detected or not.

```
    def _checkargs(self, args):
        if not super()._checkargs(args):
            return False

        if not self.welcome_message:
            scenario.logging.error(f"Wrong name {self.welcome_message!r}")
```

(continues on next page)

(continued from previous page)

```
            return False
        if not self.welcome_message.startswith("Hello"):
            _name = self.welcome_message
            self.welcome_message = f"Hello {_name}!"
            self.bye_message = f"Bye {_name}!"

        return True
```

Then, in the *main* part, prior to calling the *ScenarioRunner.main()* method:

- Set an instance of our DemoArgs class with the *Args.setinstance()* method.
- Call the *Args.parse()* method to parse the command line arguments.

```
    # Command line arguments.
    scenario.Args.setinstance(DemoArgs())
    if not scenario.Args.getinstance().parse(sys.argv[1:]):
        sys.exit(int(scenario.Args.getinstance().error_code))
```

At this point, the user test environment can use the extra arguments added with the DemoArgs class, but regular arguments as well.

```
    # --show-configs option.
    if DemoArgs.getinstance().show_config_db:
        scenario.conf.show(logging.INFO)
        sys.exit(int(scenario.ErrorCode.SUCCESS))

    # Welcome message.
    scenario.logging.info(DemoArgs.getinstance().welcome_message)

    # File logging: use the first scenario file name to determine the output log file␣
↪name.
    _outpath = DemoArgs.getinstance().scenario_paths[0].with_suffix(".log")
    scenario.conf.set("scenario.log_file", _outpath)
    scenario.logging.info(f"Test log saved in '{_outpath}'")
```

Using the --help option displays both:

- the usual *ScenarioArgs* options,
- and the extra options added by the DemoArgs class.

```
$ ./demo/run-demo.py --help
```

```
usage: run-demo.py [-h] [--config-file CONFIG_PATH] [--config-value KEY VALUE]
                   [--debug-class DEBUG_CLASS] [--doc-only]
                   [--issue-level-error ISSUE_LEVEL]
                   [--issue-level-ignored ISSUE_LEVEL]
                   [--json-report JSON_REPORT_PATH]
                   [--extra-info ATTRIBUTE_NAME] [--welcome NAME]
                   [--show-configs]
                   SCENARIO_PATH [SCENARIO_PATH ...]

Demo test launcher.
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  SCENARIO_PATH         Scenario script(s) to execute.

optional arguments:
  -h, --help            Show this help message and exit.
  --config-file CONFIG_PATH
                        Input configuration file path. This option may be
                        called several times.
  --config-value KEY VALUE
                        Single configuration value. This option may be called
                        several times.
  --debug-class DEBUG_CLASS
                        Activate debugging for the given class.
  --doc-only            Generate documentation without executing the test(s).
  --issue-level-error ISSUE_LEVEL
                        Define the issue level from and above which known
                        issues should be considered as errors. None by
                        default, i.e. all known issues are considered as
                        warnings.
  --issue-level-ignored ISSUE_LEVEL
                        Define the issue level from and under which known
                        issues should be ignored. None by default, i.e. no
                        known issue ignored by default.
  --json-report JSON_REPORT_PATH
                        Save the report in the given JSON output file path.
                        Single scenario only.
  --extra-info ATTRIBUTE_NAME
                        Scenario attribute to display for extra info when
                        displaying results. Applicable when executing several
                        tests. This option may be called several times to
                        display more info.
  --welcome NAME        User name.
  --show-configs        Show the configuration values with their origin, then
                        stop.
```

4.17.2 Pre & post-operations

As introduced above, extending the launcher script gives you the opportunity to add pre-operations, as soon as the command line arguments have been parsed, and post-operations after the test execution.

Our demo/run-demo.py script gives examples of pre & post-operations:

- a welcome message displayed before the test is executed:

```
    # Welcome message.
    scenario.logging.info(DemoArgs.getinstance().welcome_message)
```

- a bye message displayed just before the command line ends:

```
    # Bye message.
    scenario.logging.info(DemoArgs.getinstance().bye_message)
```

- optional display of the configuration database:

```
    # --show-configs option.
    if DemoArgs.getinstance().show_config_db:
        scenario.conf.show(logging.INFO)
        sys.exit(int(scenario.ErrorCode.SUCCESS))
```

- *configuration value settings* that enable *file logging*:

```
    # File logging: use the first scenario file name to determine the output log file␣
↪name.
    _outpath = DemoArgs.getinstance().scenario_paths[0].with_suffix(".log")
    scenario.conf.set("scenario.log_file", _outpath)
    scenario.logging.info(f"Test log saved in '{_outpath}'")
```

4.17.3 Base launcher execution

The call to the *ScenarioRunner.main()* method will not analyze command line arguments twice, and use the values given by our `DemoArgs` instance already set.

```
    # Scenario execution.
    _res = scenario.runner.main()
```

4.17.4 Return code

Eventually, convert the enum value returned by *ScenarioRunner.main()* into a simple integer value, so that the error can be handled in the shell that launched the command line.

```
    # Error code.
    sys.exit(int(_res))
```

4.17.5 Campaign launcher script extension

Extending the campaign launcher script works the same, except that:

- the *CampaignArgs* class may be overloaded to add extra command line arguments,
- the *CampaignRunner.main()* must be called in the end.

4.17.6 Setting the main path (optional)

Another thing that a launcher script may do is to set the *main path* (see *Path.setmainpath()*).

A *main path* shall be set for the current test projet. This way, all paths displayed during the tests may be nicely displayed as *pretty path* from this *main path*, whatever the current working directory (see *Path.prettypath*).

```
    # Main path.
    scenario.Path.setmainpath(scenario.Path(__file__).parents[1])
```

Tip: For display purpose, it is advised to set the *main path* after the program arguments have been analyzed.

4.18 Scenario stack

Todo: Documentation needed for `scenario.stack`.

- Current test case, step… being built / executed.
- Inspection facility.
- May be combined with *handlers*.

CHAPTER

FIVE

DEVELOPMENT

This page describes how to develop the *scenario* library.

5.1 Development environment

This section describes the tools constituting the development environment.

5.1.1 Launch the tests

Todo: Documentation needed for testing:

- `./test/run-unit-campaign.py`
- `./test/run-unit-test.py`

5.1.2 Type checking

Todo: Documentation needed for type checking:

- `./tools/checktypes.py`
- Adjust `files` and `namespace_packages` configurations in `mypy.conf` depending on mypy#9393 returns.

5.1.3 Check encodings and file permissions

Todo: Documentation needed for encoding checking:

- `repo-checkfiles`

5.1.4 Check license headers

Todo: Documentation needed for license headers:

- `repo-checklicense`

5.1.5 Build documentation

Todo: Documentation needed for building the documentation:

- `./tools/mkdoc.py`

5.2 Design

This page describes the design of the *scenario* library.

5.2.1 Architecture

Scenario execution

The *ScenarioDefinition*, *StepDefinition* and *ActionResultDefinition* classes are the base classes for the definition of scenarios, steps, actions and expected results respectively.

The *ScenarioRunner* instance handles the execution of them.

Its *ScenarioRunner.main()* method is the entry point that a *launcher script* should call. This method:

1. analyzes the command line arguments and loads the configuration files (see the *related design section*),
2. builds a scenario instance from the given scenario script, with reflexive programming,
3. proceeds with the scenario execution.

The *ScenarioRunner* class works with a couple of helper classes.

The *ScenarioExecution*, *StepExecution* and *ActionResultExecution* classes store the execution information related to definition classes cited above.

Table 1: Definition v/s execution classes

	Definition	Execution
Scenario level	*ScenarioDefinition* • Describes the list of *StepDefinition* that define the scenario. • Gives the related *ScenarioExecution* instance when executed.	*ScenarioExecution* • Tells which step is currently being executed. • Stores the execution error, if any. • Stores execution statistics. • Gives access to the related *ScenarioDefinition* instance.
Step level	*StepDefinition* • Describes the list of *ActionResultDefinition* that define the step. • Gives the related *StepExecution* instances when executed.	*StepExecution* • Tells wich action or expected result is currently being executed. • Stores the execution error, if any. • Stores execution statistics. • Gives access to the related *StepDefinition* instance.
Action and expected result level	*ActionResultDefinition* • Describes an action or an expected result, with its text. • Gives the related *ActionResultExecution* instances when executed.	*ActionResultExecution* • Stores *evidence*. • Stores the execution error, if any. • Stores execution statistics. • Gives access to the related *ActionResultDefinition* instance.

Note: Due to the *goto* feature, steps, actions and expected results may be executed several times within a single scenario execution.

The *ScenarioStack* also is a helper class for *ScenarioRunner*:

- It stores the current stack of scenarios being executed (see *sub-scenarios*.
- It also provides a couple of accessors to the current step, action or expected result being executed.

The *ScenarioRunner* class remains the conductor of all:

1. The *ScenarioRunner.main()* method is called.
2. For each script path given in the command line:
    1. A main *ScenarioDefinition* instance is created[2] from the scenario class in the script[8]. A *ScenarioExecution* instance is created as well, and pushed to the *ScenarioStack* instance[4].
    2. *ScenarioRunner._execution_mode* is set to *ScenarioRunner.ExecutionMode.BUILD_OBJECTS*:

[2] See *ScenarioRunner.executepath()*.
[8] See *ScenarioDefinitionHelper*.
[4] See *ScenarioRunner._beginscenario()*.

1. In case the steps are defined with `step...()` methods, the *ScenarioDefinition* is fed using reflexive programmation (the same for scenario attributes defined with class members)[4Page 65, 8].
2. Each step is executed a first time[Page 65, 45] in order to build its *ActionResultDefinition* instances for each *StepUserApi.ACTION()* and *StepUserApi.RESULT()* call[6]. During this first execution of the step, the two latter methods return `False`[1], which prevents the test from being executed at this point.

3. *ScenarioRunner._execution_mode* is set to *ScenarioRunner.ExecutionMode.EXECUTE* or *ScenarioRunner.ExecutionMode.DOC_ONLY*[3]. For each step[35]:
   1. A *StepExecution* instance is created[5].
   2. The user test code is called[5].
   3. For each *StepUserApi.ACTION()* and *StepUserApi.RESULT()* call[6]:
      1. A *ActionResultExecution* instance is created[6].
      2. If a sub-scenario is executed, then it is pushed to the *ScenarioStack* instance[Page 65, 4], built[Page 65, 4Page 65, 85], executed[35], and eventually popped from the *ScenarioStack* instance[7].
   4. The main scenario is eventually popped from the *ScenarioStack* instance[7].
3. If there were several scenarios executed, the final results are displayed[9].

—

Subscenarios

Todo: Documentation needed: Architecture - Subscenarios

Assertions, error management & execution locations

Todo: Documentation needed: Architecture - Error management

Campaign execution

Todo: Documentation needed: Architecture - Campaign execution

- *CampaignRunner*
- *CampaignExecution*, *TestSuiteExecution*, *TestCaseExecution* classes.
- Test suite files.
- Test cases executed in separate processes.

[5] See *ScenarioRunner._execstep()*.
[6] See *ScenarioRunner.onactionresult()*.
[1] See *ScenarioRunner._execution_mode*.
[3] See *ScenarioRunner.executescenario()*.
[7] See *ScenarioRunner._endscenario()*.
[9] See *ScenarioResults*.

Logging

Todo: Documentation needed: Architecture - Logging

Configuration

Todo: Documentation needed: Architecture - Configuration

Path management

Todo: Documentation needed: Architecture - Path

5.2.2 Source documentation

The detailed documentation extracted from the Python sources.

scenario package

scenario package definition.

Package information

info

Alias of *PKG_INFO*.

Gives the package information: version, ...

Base classes

Classes to inherit from in order to describe test scenarios and libraries.

Scenario

Alias of *ScenarioDefinition*.

Base class to inherit from in order to define a test scenario.

Step

Alias of *StepDefinition*.

Base class to inherit from in order to define a test step.

ActionResult

Alias of *ActionResultDefinition*.

Assertions

Make verifications on data.

Assertions
: Alias of *Assertions*.

  Library of static assertion methods.

  Can be sub-classes. *Scenario* and *Step* inherit from this class.

assertionhelpers
: Alias of *assertionhelpers*.

  Helper functions and types when you want to write your own assertion routines.

Logging

Logging management.

Logger
: Alias of *Logger*.

  Object with logging capabilities.

  Scenario and *Step* inherit from this class.

logging
: Main logger instance.

Console
: Alias of *Console*.

  Console colors.

LogExtraData
: Alias of *LogExtraData*.

  Logging extra data management.

debug
: Alias of *debugutils*.

  Helper functions and types for debugging.

Configuration

Configuration management.

conf
: Configuration manager instance.

ConfigNode
: Alias of *ConfigNode*.

ConfigKey
: Alias of *ScenarioConfig.Key*.

  scenario configuration keys.

Launchers

Classes to launch the test scenarios and campaigns from custom launcher scripts.

runner

Scenario runner instance.

Call from your own scenario launcher script with:

```
scenario.runner.main()
```

campaign_runner

Campaign runner instance.

Call from your own campaign launcher script with:

```
scenario.campaign_runner.main()
```

Args

Alias of *Args*.

Base class for *ScenarioArgs* and *CampaignArgs*.

ScenarioArgs

Alias of *ScenarioArgs*.

Inherit from this class in order to extend *ScenarioRunner* arguments with your own launcher script ones.

CampaignArgs

Alias of *CampaignArgs*.

Inherit from this class in order to extend *CampaignRunner* arguments with your own launcher script ones.

ErrorCode

Alias of *ErrorCode*.

Error codes returned by the `main()` methods of *ScenarioRunner* and *CampaignRunner*.

Handlers (advanced)

Add reactive code.

handlers

Handler manager instance.

Event

Alias of *ScenarioEvent*.

EventData

Alias of *ScenarioEventData*.

Execution result classes (advanced)

Sometimes, you may need to access information about the test execution itself.

ExecutionStatus

Alias of *ExecutionStatus*.

Describes the final status of a scenario or campaign execution.

ScenarioExecution

Alias of *ScenarioExecution*.

StepExecution

Alias of *StepExecution*.

ActionResultExecution

Alias of *ActionResultExecution*.

TestError

Alias of *TestError*.

Describes an error that occurred during the tests.

ExceptionError

Alias of *ExceptionError*.

Describes an error due to an exception that occurred during the tests.

KnownIssue

Alias of *KnownIssue*.

Describes an error due to an exception that occurred during the tests.

IssueLevel

Alias of *IssueLevel*.

Provides methods to define named issue levels.

TimeStats

Alias of *TimeStats*.

Describes execution time statistics.

ExecTotalStats

Alias of *ExecTotalStats*.

Describes count statistics: number of items executed, out of the total number of items.

stack

Scenario stack instance.

Reports (advanced)

The following objects give you the opportunity to read and write scenario and campaign reports.

report

Scenario report manager.

campaign_report

Campaign report manager.

Miscellaneous

Path

Alias of *Path*.

AnyPathType

Alias of `path.AnyPathType`.

SubProcess

Alias of *SubProcess*.

Eases the way to prepare a sub-process, execute it, and then retrieve its results.

VarSubProcessType

Alias of `subprocess.VarSubProcessType`.

CodeLocation

Alias of *CodeLocation*.

datetime

Alias of *datetimeutils*.

Date/time utils.

tz

Alias of *timezoneutils*.

Timezone utils.

enum

Alias of *enumutils*.

Enum utils.

Submodules

scenario.actionresultdefinition module

Action / expected result definition.

class ActionResultDefinition

Bases: `object`

This class describes both an action or an expected result.

class Type

Bases: *StrEnum*

Enum that tells whether a user text defines an action or an expected result.

ACTION = 'ACTION'

Action type.

RESULT = 'RESULT'

Expected result type.

__init__(*type*, *description*)

Parameters

- **type** – Action/result type.
- **description** – User description for this action/result.

Note: As it makes the API convenient, we deliberately shadow the built-in with the `type` parameter.

type

Action/result type.

description

Action/result textual description.

step

Owner step.

Initially set with a void reference. Fixed when `stepdefinition.StepDefinition.addactionsresults()` is called.

executions

Executions.

__repr__()

Canonical string representation.

__str__()

Printable string representation.

scenario.actionresultexecution module

Action / expected result execution management.

class ActionResultExecution

Bases: `object`

Action/result execution tracking object.

__init__(*definition*)

Sets the start time automatically.

definition

Owner action/result reference.

time
Time statistics.

evidence
Evidence items.

subscenarios
Sub-scenario executions.

errors
Errors.

warnings
Warnings.

__repr__()
Canonical string representation.

scenario.args module

Base module for program arguments management.

Args.getinstance() gives the only only instance of program arguments, May actually be a `scenarioargs.ExecArgs` or a *CampaignArgs* instance.

class Args
Bases: *Logger*, *CommonConfigArgs*, *CommonLoggingArgs*

Common program arguments management.

Handles:

- `--help` option,
- Configuration file options,
- Logging options.

_instance
Main instance of *Args*.

static setinstance(*instance*, *warn_reset=True*)
Sets the main instance of *Args*.

Parameters

- **instance** – *Args* instance.
- **warn_reset** – Set to `False` in order to avoid the warning to be logged.

When consecutive calls occur, the latest overwrites the previous, and a warning is displayed unless `warn_reset` is set to `False`.

classmethod getinstance()
Singleton.

Returns
Main *Args* instance.

> **Warning:** The main *Args* instance is not created automatically by this method, and should be set with *setinstance()* prior to any *getinstance()* call.

classmethod isset()

Checks whether the single instance of *Args* is set and of the `cls` type.

Parameters
cls – Expected type.

Returns
`True` if the single *Args* instance is of the given type, `False` otherwise.

__init__(*class_debugging*)

Defines common program arguments.

Parameters
class_debugging – See *CommonLoggingArgs*.

__arg_parser

`argparse` parser object.

__arg_infos

Arguments information.

parsed

Parsed flag. Tells whether arguments have been successfully parsed yet or not.

error_code

Argument parsing error code.

setprog(*name*)

Overwrites program name.

Parameters
name – Program name to be displayed with usage info.

setdescription(*description*)

Overwrites program description.

Parameters
description – Program description to be displayed with usage info.

addarg(*member_desc*, *member_name*, *member_type*)

Adds a program argument.

Parameters

- **member_desc** – Textual description of the program argument(s).
- **member_name** – Corresponding member name in the owner *Args* instance.
- **member_type** – Type of the program argument, or base type of the program arguments list, or conversion handler. When defined as a 2 items tuple, the argument feeds a dictionary: the first item of the tuple shall be `str` (for the dictionary keys), and the second item gives the type of the dictionary values.

Returns
`Args.ArgInfo` instance whose *ArgInfo.define()* should be called onto.

ArgInfo.define() should be called on the *ArgInfo* object returned:

```
self.addarg("Configuration files", "config_paths", Path).define(
    "--config-file", metavar="CONFIG_PATH",
    action="append", type=str, default=[],
    help="Input configuration file path. "
         "This option may be called several times.",
)
```

parse(*args*)

Parses program arguments.

Parameters
args – Argument list, without the program name.

Returns
`True` for success, `False` otherwise.

_checkargs(*args*)

Handler for special verifications on program arguments.

Parameters
args – The untyped object returned by `argparse.ArgumentParser.parse_args()`.

Returns
`True` for success, `False` otherwise.

Shall be overridden in subclasses.

class ArgInfo

Bases: `object`

Class that describes a single program argument (single value, list or dictionary).

Whether the program argument is a single value, or a list of value, depends on the `argparse` definition made through *ArgInfo.define()*.

__init__(*arg_parser*, *member_desc*, *member_name*, *member_type*)

Parameters

- **arg_parser** – Related `argparse.ArgumentParser` instance.
- **member_desc** – Textual description of the program argument(s).
- **member_name** – Corresponding member name in the owner *Args* instance.
- **member_type** – Base type of the program argument(s). See *Args.addarg()* for a detailed description of this parameter.

`Args.ArgInfo.define()` should be called onto each `Args.ArgInfo` instance newly created.

See also:

Args.addarg(), `Args.ArgInfo.define()`

arg_parser

Related `argparse.ArgumentParser` instance.

member_desc

Textual description of the program argument(s).

member_name

Corresponding member name in the owner *Args* instance.

key_type

Key type, when the argument feeds a dictionary.

value_type

Base type of the program argument(s).

parser_arg

`argparse.Action` instance defined by the `Args.ArgInfo.define()` method.

define(**args*, ***kwargs*)

Defines the `argparse` command line argument.

Parameters

- **args** – List of positional arguments.
- **kwargs** – Dictionary of named arguments.

Refer to the regular `argparse` documentation, except for the `dest` parameter which should not be set. The `Args.ArgInfo.member_name` member will be used for the purpose.

Should be called on the `Args.ArgInfo` returned the *Args.addarg()* method.

See also:

Args.addarg()

process(*args_instance*, *parsed_args*)

Process the argument value once parsed by `argparse` and feed the *Args* instance.

Parameters

- **args_instance** – *Args* instance to feed.
- **parsed_args** – Opaque parsed object returned by the `argparse` library.

Returns

`True` when the operation succeeded, `False` otherwise.

scenario.assertionhelpers module

Assertion helpers.

Functions, types and constants for the *Assertions* class.

unittest

`unittest.TestCase` instance used to call `unittest` assertion functions.

safecontainer(*obj*)

Ensures working with a string or list-like object.

Parameters

obj – Input iterable object.

Returns

String or list-like object:

- may be used as is, in order to check its emptiness,
- may be applied `len()` on it,
- has a `count()` method,

- …

errmsg(*optional*, *standard*, **args*)

Formats an error message: the optional and/or the regular one.

Parameters

- **optional** – Optional assertion message, if set.
- **standard** – Standard assertion message.
- **args** – Standard assertion message arguments.

Returns
Error message.

ctxmsg(*context*, *err*, **args*)

Builds an contextual assertion message.

Parameters

- **context** – Context pattern, basically the methods name (e.g.: `"assertisinstance()"`).
- **err** – Detailed assertion message.
- **args** – Detailed assertion message arguments

Returns
Assertion message.

isnonemsg(*context*, *what*)

Builds an assertion message indicating that an element in unexpectedly `None`.

Parameters

- **context** – Context pattern, basically the methods name (e.g.: `"assertisinstance()"`).
- **what** – Name of the parameter, or element, unexpectedly `None` (e.g.: `"obj"` for a `obj` parameter).

Returns
Assertion message.

evidence(*evidence_enabled*, *regular*, **args*)

Tracks assertion data, depending on the current scenario configuration

Parameters

- **evidence_enabled** – Proof message activation and/or specialization (see the *dedicated note*).
- **regular** – Regular proof message.
- **args** – Proof message arguments.

getstepexecution(*step_execution_specification*)

Retrieves the (last) *StepExecution* instance corresponding to the given specification.

Parameters
step_execution_specification – Step execution specification (see `assertionhelpers.StepExecutionSpecType`).

Returns
Step execution corresponding to the given specification.

Raise
Exception when the step execution could not be found.

scenario.assertions module

Assertion methods.

The *Assertions* class defines a collection of assertion methods.

class Assertions

Bases: object

The *Assertions* class gathers static assertion methods.

It can be subclasses by classes that onboard these assertion methods, like the base *ScenarioDefinition* and *StepDefinition* classes.

See the *assertion documentation* for details.

static fail(*err*)

Makes the test fail with the given message.

Parameters
err – Error message.

static todo(*err*)

Makes the test fail because it is not completely implemented.

Parameters
err – Error message.

static assertequal(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks member equality.

Parameters

- **obj1** – First member.
- **obj2** – Second member.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnotequal(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks member inequality.

Parameters

- **obj1** – First member.
- **obj2** – Second member.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisnone(*obj*, *err=None*, *evidence=False*)

Checks a given value is None.

Parameters

- **obj** – Value expected to be None.

- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisnotnone(*obj*, *err=None*, *evidence=False*)

Checks a given value is not `None`.

Parameters

- **obj** – Value expected to be not `None`.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Returns

The value `obj`, ensured not to be `None`.

static assertisinstance(*obj*, *type*, *err=None*, *evidence=False*)

Checks whether the object is of the given type, or one of the given types.

Parameters

- **obj** – Object to check.
- **type** – Type or list of types to check the object against.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Returns

The value `obj`, ensured not to be of type `type`.

Note: As it makes the API convenient, we deliberately shadow the built-in with the `type` parameter.

---

static assertisnotinstance(*obj*, *type*, *err=None*, *evidence=False*)

Checks whether the object is not of the given type, or none of the given types.

Parameters

- **obj** – Object to check.
- **type** – Type or list of types to check the object against.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Note: As it makes the API convenient, we deliberately shadow the built-in with the `type` parameter.

---

static assertsameinstances(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks two Python instances are the same.

Parameters

- **obj1** – First instance to check.
- **obj2** – Second instance to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnotsameinstances(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks two Python instances are not the same.

Parameters

- **obj1** – First instance to check.
- **obj2** – Second instance to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static asserttrue(*value*, *err=None*, *evidence=False*)

Checks a value is `True`.

Parameters

- **value** – Value to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertfalse(*value*, *err=None*, *evidence=False*)

Checks a value is `False`.

Parameters

- **value** – Value to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertless(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks a value is strictly less than another.

Parameters

- **obj1** – Value expected to be below.
- **obj2** – Value expected to be above.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertlessequal(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks a value is less than or equal to another.

Parameters

- **obj1** – Value expected to be below.
- **obj2** – Value expected to be above.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertgreater(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks a value is strictly greater than another.

Parameters

- **obj1** – Value expected to be above.

- **obj2** – Value expected to be below.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertgreaterequal(*obj1*, *obj2*, *err=None*, *evidence=False*)

Checks a value is greater than or equal to another.

Parameters

- **obj1** – Value expected to be above.
- **obj2** – Value expected to be below.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertstrictlybetween(*between*, *low*, *high*, *err=None*, *evidence=False*)

Checks a value is strictly between two others.

Parameters

- **between** – Value expected to be between the others.
- **low** – Low value.
- **high** – High value.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertbetweenorequal(*between*, *low*, *high*, *err=None*, *evidence=False*)

Checks a value is between or equal to two others.

Parameters

- **between** – Value expected to be between the others.
- **low** – Low value.
- **high** – High value.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnear(*obj1*, *obj2*, *margin*, *err=None*, *evidence=False*)

Checks a value is near another oe.

Parameters

- **obj1** – Value to check.
- **obj2** – Reference value.
- **margin** – Margin of error.
- **err** – Optional error message.
- **evidence** –

Returns

Evidence activation (see the *dedicated note*).

static assertstartswith(*string*, *start*, *err=None*, *evidence=False*)

Checks a string (or bytes) starts with a given pattern

Parameters

- **string** – String (or bytes) to check.
- **start** – Expected start pattern.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnotstartswith(*string*, *start*, *err=None*, *evidence=False*)

Checks a string (or bytes) does not start with a given pattern.

Parameters

- **string** – String (or bytes) to check.
- **start** – Unexpected start pattern.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertendswith(*string*, *end*, *err=None*, *evidence=False*)

Checks a string (or bytes) ends with a given pattern.

Parameters

- **string** – String (or bytes) to check.
- **end** – Expected end pattern.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnotendswith(*string*, *end*, *err=None*, *evidence=False*)

Checks a string (or bytes) does not end with a given pattern.

Parameters

- **string** – String (or bytes) to check.
- **end** – Unexpected end pattern.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertregex(*regex*, *string*, *err=None*, *evidence=False*)

Checks a string (or bytes) matches a regular expression.

Parameters

- **regex** – Regular expression to match with.
- **string** – String (or bytes) to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Returns

`re` match object.

Note: The `regex` and `string` parameters follow the usual order of `re` functions (contrary to `unittest` `assertRegex()`).

`static assertnotregex`(*regex*, *string*, *err=None*, *evidence=False*)

Checks a string (or bytes) does not match a regular expression.

Parameters

- **regex** – Regular expression to match with.
- **string** – String (or bytes) to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Note: The `regex` and `string` parameters follow the usual order of `re` functions (contrary to `unittest` `assertNotRegex()`).

`static asserttimeinstep`(*time*, *step*, *err=None*, *evidence=False*, *expect_end_time=True*)

Checks the date/time is within the given step execution times.

Parameters

- **time** – Date/time to check.
- **step** – Step specification (see `assertionhelpers.StepExecutionSpecType`).
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).
- **expect_end_time** – `True` when the step execution is expected to be terminated.

Returns

Step execution that matched the specification.

`static asserttimeinsteps`(*time*, *start*, *end*, *err=None*, *evidence=False*, *expect_end_time=True*)

Checks the date/time is in the execution times of a given range of steps.

Parameters

- **time** – Date/time to check.
- **start** – Specification of the first step of the range (see `assertionhelpers.StepExecutionSpecType`).
- **end** – Specification of the last step of the range (see `assertionhelpers.StepExecutionSpecType`).
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).
- **expect_end_time** – `True` when the `end` step execution is expected to be terminated.

Returns

Step execution that matched the `start` and `end` specifications.

static asserttimebeforestep(*time*, *step*, *err=None*, *evidence=False*)

Checks the date/time is (strictly) before a given step executime time.

Parameters

- **time** – Date/time to check.
- **step** – Step specification (see `assertionhelpers.StepExecutionSpecType`).
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Returns

Step execution that matched the specification.

static asserttimeafterstep(*time*, *step*, *err=None*, *evidence=False*)

Checks the date/time is (strictly) after a given step executime time.

Parameters

- **time** – Date/time to check.
- **step** – Step specification (see `assertionhelpers.StepExecutionSpecType`).
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

Returns

Step execution that matched the specification.

static assertisempty(*obj*, *err=None*, *evidence=False*)

Checks that a container object (string, bytes, list, dictionary, set, ...) is empty.

Parameters

- **obj** – Container object to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisnotempty(*obj*, *err=None*, *evidence=False*)

Checks that a container object (string, bytes, list, dictionary, set, ...) is not empty.

Parameters

- **obj** – Container object to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertlen(*obj*, *length*, *err=None*, *evidence=False*)

Checks the length of a container object (string, bytes, list, dictionary, set, ...).

Parameters

- **obj** – Container object which length to check.
- **length** – Expected length.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertin(*obj*, *container*, *err=None*, *evidence=False*)

Checks a pattern or item is in a container object (string, bytes, list, dictionary, set, ...).

Parameters

- **obj** – Pattern or item to check in `container`.
- **container** – Container object.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnotin(*obj*, *container*, *err=None*, *evidence=False*)

Checks a pattern or item is not in a container object (string, bytes, list, dictionary, set, ...).

Parameters

- **obj** – Pattern or item to check not in `container`.
- **container** – Container object.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertcount(*container*, *obj*, *count*, *err=None*, *evidence=False*)

Checks a string (or bytes), contains the expected number of patterns, or a list, dictionary or set contains the expected number of a given item.

Parameters

- **container** – String (or bytes), list, dictionary or set that should contain `obj` `count` times.
- **obj** – Pattern or item to check `count` times in `container`.
- **count** – Expected number of `obj` in `container`.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertjson(*json_data*, *jsonpath*, *err=None*, *evidence=False*, *type=None*, *value=None*, *ref=None*, *count=1*, *len=None*)

Checks JSON content.

Parameters

- **json_data** – Input JSON dictionary.
- **jsonpath** – JSONPath.

  Currently a subset of the full syntax (see https://goessner.net/articles/JsonPath/).
- **err** – Optional error message.
- **evidence** – Evidence activation (see `scenario.Assertions`'s documentation).
- **type** – Expected type for the matching elements.
- **value** – Expected value for the matching elements.
- **ref** – Reference JSON dictionary giving the expected value for the given path.
- **count** – Expected number of matching elements.

  1 by default. May be set to `None`.

- **len** – Expected length.

  It assumes `len()` can be applied on the only searched item, which means that when using `len`:

  - `count` must not be set to anything else but 1 (by default),
  - it is a good practice to specify the expected `type` as well (`list` usually).

Returns
The matching element, when `count` is 1, list of matching elements otherwise.

Note: As it makes the API convenient, we deliberately shadow the built-in with the `type` parameter.

---

static assertexists(*path*, *err=None*, *evidence=False*)

Checks whether a path exists.

Parameters

- **path** – Path to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertnotexists(*path*, *err=None*, *evidence=False*)

Checks whether a path does not exist.

Parameters

- **path** – Path to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisfile(*path*, *err=None*, *evidence=False*)

Checks whether a path is a regular file.

Parameters

- **path** – Path to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisdir(*path*, *err=None*, *evidence=False*)

Checks whether a path is a directory.

Parameters

- **path** – Path to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertsamepaths(*path1*, *path2*, *err=None*, *evidence=False*)

Checks whether two paths are actually the same, even though they may be absolute or relative, or accessed through a symbolic link…

Parameters

- **path1** – First path to check.

- **path2** – Second path to check.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisrelativeto(*path*, *dir*, *err=None*, *evidence=False*)

Checks whether a path is a sub-path of a directory.

Parameters

- **path** – Path to check.
- **dir** – Container directory candidate.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

static assertisnotrelativeto(*path*, *dir*, *err=None*, *evidence=False*)

Checks whether a path is not a sub-path of a directory.

Parameters

- **path** – Path to check.
- **dir** – Directory expected not to be a container directory for `path`.
- **err** – Optional error message.
- **evidence** – Evidence activation (see the *dedicated note*).

scenario.campaignargs module

Campaign runner program arguments.

class CampaignArgs

Bases: *Args*, *CommonExecArgs*

Campaign runner program arguments.

__init__(*positional_args=True*, *default_outdir_cwd=True*)

Defines program arguments for *CampaignRunner*.

Parameters

- **positional_args** – `False` to disable the scenario path positional arguments definition. Useful for user programs that wish to redefine it.
- **default_outdir_cwd** – `False` to disable the use of the current directory by default.

_default_outdir_cwd

Current directory as the default output directory flag.

_outdir

Output directory path.

Inner attribute. `None` until actually set, either with the –outdir option, or programmatically in sub-classes.

create_dt_subdir

`True` when an output subdirectory in *CampaignArgs.outdir* named with the campaign execution date and time should be created.

extra_info

Attribute names to display for extra info. Applicable when executing several tests.

test_suite_paths

Campaign file path.

property outdir

Output directory path as a public property.

_checkargs(*args*)

Check campaign arguments once parsed.

Returns
True for success, False otherwise.

scenario.campaignexecution module

Campaign execution results.

The *CampaignExecution* class stores the execution results of a campaign. It owns a list of `TestSuite` instances (actually one, called 'All'), which owns a list of `TestCase` instances (one test case per scenario).

class CampaignExecution

Bases: `object`

Main campaign result object.

__init__(*outdir*)

Parameters
outdir – Output directory path.

`None` initializes the output directory path with the current working directory.

outdir

Output directory path.

test_suite_executions

Test suite results.

time

Time statistics.

__repr__()

Canonical string representation.

property junit_path

JUnit path.

property steps

Step statistics.

property actions

Action statistics.

property results

Expected result statistics.

property counts

Campaign statistics.

class TestSuiteExecution

Bases: `object`

Test suite execution object.

__init__(*campaign_execution*, *test_suite_path*)

Parameters

- **campaign_execution** – Owner *CampaignExecution* object.
- **test_suite_path** – Test suite file path.

  None initializes the *test_suite_file* member with a *void* file path, which makes the *test_suite_file* instance *void* as well. This path can be fixed programmatically later on.

campaign_execution

Owner campaign execution.

test_suite_file

Campaign file.

test_case_executions

Test cases.

time

Time statistics.

__repr__()

Canonical string representation.

property steps

Step statistics.

property actions

Action statistics.

property results

Expected result statistics.

property counts

Campaign statistics.

class TestCaseExecution

Bases: `object`

Test case (i.e. test scenario) execution object.

__init__(*test_suite_execution*, *script_path*)

Parameters

- **test_suite_execution** – Owner `TestSuite` object.
- **script_path** – Scenario script path.

  None initializes the *script_path* member with a *void* file path. This path can be fixed programmatically later on.

`test_suite_execution`
: Owner test suite execution.

`script_path`
: Scenario script path.

`time`
: Time statistics.

`log`
: Test case log output.

`json`
: Test case JSON output.

`__repr__()`
: Canonical string representation.

`property scenario_execution`
: Scenario execution data.

`property name`
: Test case name.

`property status`
: Scenario execution status.

`property errors`
: Test errors.

`property warnings`
: Warnings.

`property steps`
: Step statistics.

`property actions`
: Action statistics.

`property results`
: Expected result statistics.

`class CampaignStats`
: Bases: `object`

JUnit compatible statistics.

Failures v/s errors

According to https://stackoverflow.com/questions/3425995/whats-the-difference-between-failure-and-error-in-junit:

- tests are considered that they have "failed" because of an assertion,
- tests are said to be in "error" but of an unexpected error.

`__init__()`
: Initializes all counters with `0`.

total

Total number of test cases.

disabled

Number of test cases disabled.

skipped

Number of skipped test cases.

For test suites.

warnings

Number of tests that terminated with warnings.

failures

Number of test cases that failed due to assertions.

errors

Number of test cases that failed unexpectedly.

class LogFileReader

Bases: `object`

Log file path and content.

__init__()

Initializes *path* and *content* attributes with `None`.

path

Test case log file path.

content

Test case log file content.

read()

Read the log file.

Returns
`True` when the log file could be read successfully, `False` otherwise.

class JsonReportReader

Bases: `object`

JSON file path and content.

__init__()

Initializes *path* and *content* attributes with `None`.

path

Test case JSON file path.

content

Scenario execution data read from the test case JSON file.

read()

Read the JSON report.

Returns
`True` when the JSON report file could be read and parsed successfully, `False` otherwise.

scenario.campaignlogging module

Campaign execution logging.

CAMPAIGN_LOGGING

Main instance of *CampaignLogging*.

class CampaignLogging

Bases: `object`

Campaign execution logging management.

class _Call

Bases: *StrEnum*

CampaignLogging call identifiers.

BEGIN_CAMPAIGN = 'begincampaign'

BEGIN_TEST_SUITE = 'begintestsuite'

BEGIN_TEST_CASE = 'begintestcase'

END_TEST_CASE = 'endtestcase'

END_TEST_SUITE = 'endtestsuite'

END_CAMPAIGN = 'endcampaign'

__init__()

Initializes private instance members.

_calls

History of this class's method calls.

Makes it possible to adjust the display depending on the sequence of information.

begincampaign(*campaign_execution*)

Displays the beginning of the campaign.

Parameters
campaign_execution – Campaign being executed.

begintestsuite(*test_suite_execution*)

Displays the beginning of a test suite.

Parameters
test_suite_execution – Test suite being executed.

begintestcase(*test_case_execution*)

Displays the beginning of a test case.

Parameters
test_case_execution – Test case being executed.

endtestcase(*test_case_execution*)

Displays the end of a test case.

:param test_case_execution:Test case being executed.

endtestsuite(*test_suite_execution*)

Displays the end of a test suite.

:param test_suite_execution:Test suite being executed.

endcampaign(*campaign_execution*)

Displays the end of the campaign.

Parameters
campaign_execution – Campaign being executed.

Displays the campaign statistics

scenario.campaignreport module

Campaign reports.

CAMPAIGN_REPORT

Main instance of *CampaignReport*.

class CampaignReport

Bases: *Logger*

Campaign report management.

JUnit XML reporting file format:

- Refer to: https://llg.cubic.org/docs/junit/ [CUBIC]
- Other useful resource: https://stackoverflow.com/questions/442556/spec-for-junit-xml-output

__init__()

Configures logging for the *CampaignReport* class.

_junit_path

JUnit report path being written or read.

writejunitreport(*campaign_execution*, *junit_path*)

Generates a JUnit XML report output file.

Parameters

- **campaign_execution** – Campaign execution to generate the report for.
- **junit_path** – Path to write the JUnit report into.

Returns
`True` for success, `False` otherwise.

readjunitreport(*junit_path*)

Reads the JUnit report.

Parameters
junit_path – Path of the JUnit file to read.

Returns
Campaign execution data read from the JUnit file. `None` when the file could not be read, or its content could not be parsed successfully.

_campaign2xml(*xml_doc*, *campaign_execution*)

Campaign JUnit XML generation.

Parameters

- **xml_doc** – XML document.
- **campaign_execution** – Campaign execution to generate the JUnit XML for.

Returns
Campaign JUnit XML.

_xml2campaign(*xml_doc*)

Campaign execution reading from JUnit report.

Parameters
xml_doc – JUnit XML document to read from.

Returns
Campaign execution data.

_testsuite2xml(*xml_doc*, *test_suite_execution*, *test_suite_id*)

Test suite JUnit XML generation.

Parameters

- **xml_doc** – XML document.
- **test_suite_execution** – Test suite execution to generate the JUnit XML for.
- **test_suite_id** – Test suite identifier.

Returns
Test suite JUnit XML.

_xml2testsuite(*campaign_execution*, *xml_test_suite*)

Test suite reading from JUnit report.

Parameters

- **campaign_execution** – Owner campaign execution instance.
- **xml_test_suite** – JUnit XML to read from.

Returns
Test suite execution data.

_testcase2xml(*xml_doc*, *test_case_execution*)

Test case JUnit XML generation.

Parameters

- **xml_doc** – XML document.
- **test_case_execution** – Test case execution to generate the JUnit XML for.

Returns
Test case JUnit XML.

_xml2testcase(*test_suite_execution*, *xml_test_case*)

Test case reading from JUnit XML.

Parameters

- **test_suite_execution** – Owner test suite execution instance.

- **xml_test_case** – JUnit XML to read from.

Returns
Test case execution data.

_safestr2xml(*string*)

Safe string conversion before it is used in the JUnit report.

Removal of colors.

Parameters
string – String to convert.

Returns
String safely converted.

_path2xmlattr(*xml_node*, *attr_name*, *path*)

Sets a path XML attribute.

Sets either a relative or absolute path depending on the given file location compared with this JUnit file location.

Parameters

- **xml_node** – XML node to set the attribute for.
- **attr_name** – Attribute name.
- **path** – Path object to use to set the attribute value.

_xmlattr2path(*xml_node*, *attr_name*)

Path computation from an XML attribute.

When the attribute describes a relative path, the path is computed from the JUnit file. When it describes an absolute path, the path is taken as is.

Parameters

- **xml_node** – XML node which attribute to read from.
- **attr_name** – Attribute name to read.

Returns
Path computed.

_xmlcheckstats(*xml_node*, *attr_name*, *objects*)

Statistics consistency checking between an upper level and its children.

Parameters

- **xml_node** – Upper XML node which statistics to check.
- **attr_name** – Statistics attribute to check.
- **objects** – Execution objects to check statistics with.

Displays warnings when the statistics mismatch.

scenario.campaignrunner module

Campaign execution management.

CAMPAIGN_RUNNER

Main instance of *CampaignRunner*.

class CampaignRunner

Bases: *Logger*

Campaign execution engine: runs test scenarios from input files.

Only one instance, accessible through the *CAMPAIGN_RUNNER* singleton.

This class works with the following helper classes, with their respected purpose:

- *CampaignArgs*: command line arguments,
- *CampaignExecution*: object that describes a campaign execution,
- *CampaignLogging*: campaign execution logging,
- *CampaignReport*: campaign report generation.

__init__()

Configures logging for the *CampaignRunner* class.

main()

Campaign runner main function, as a member method.

Returns
Error code.

_exectestsuitefile(*campaign_execution*, *test_suite_path*)

Executes a test suite file.

Parameters

- **campaign_execution** – CampaignExecution object to store results into.
- **test_suite_path** – Test suite file to execute.

Returns
Error code.

_exectestsuite(*test_suite_execution*)

Executes a test suite.

Parameters
test_suite_execution – Test suite to execute.

Returns
Error code.

_exectestcase(*test_case_execution*)

Executes a test case.

Parameters
test_case_execution – Test case to execute.

Returns
Error code.

scenario.configargs module

Common configuration program arguments.

class CommonConfigArgs

Bases: object

Base class for argument parser classes that embed common configuration program arguments.

__init__()

Installs common configuration program arguments.

config_paths

Configuration files.

config_values

Additional configuration values.

scenario.configdb module

Configuration database management.

CONFIG_DB

Main instance of *ConfigDatabase*.

class ConfigDatabase

Bases: *Logger*

Configuration management.

This class loads a list of configuration files, and aggregates all configuration read in a single configuration tree.

See the *configuration database* documentation.

class FileFormat

Bases: *StrEnum*

Configuration file formats.

INI = 'INI'

INI configuration file format.

JSON = 'JSON'

JSON configuration file format.

YAML = 'YAML'

YAML configuration file format.

__init__()

Initializes instance attributes and configures logging for the *ConfigDatabase* class.

_root

Configuration tree.

loadfile(*path*, *format=None*, *root=''*)

Loads a configuration file.

Parameters

- **path** – Path of the configuration file to load.

- **format** – File format.

  Determined automatically from the file extension when not specified.

- **root** – Root key to load the file from.

savefile(*path*, *format=None*, *root=''*)

Saves a configuration file.

Parameters

- **path** – Path of the configuration file to save.
- **format** – File format.

  Determined automatically from the file extension when not specified.

- **root** – Root key to save the file from.

set(*key*, *data*, *origin=None*)

Sets a configuration value of any type.

Parameters

- **key** – Configuration key.
- **data** – Configuration data.

  Can be a single value, a dictionary or a list.

  When a `os.PathLike` is given, it is automatically converted in its string form with `os.fspath()`.

  When `None` is given, it is equivalent to calling *remove()* for the given `key`.

- **origin** – Origin of the configuration data: either a simple string, or the path of the configuration file it was defined in.

  Defaults to code location when not set.

remove(*key*)

Removes a configuration key (if exists).

Parameters
key – Configuration key to remove.

show(*log_level*)

Displays the configuration database with the given log level.

Parameters
log_level – `logging` log level.

getkeys()

Returns the list of keys.

Returns
Configuration keys.

getnode(*key*)

Retrieves the configuration node for the given key.

Parameters
key – Searched configuration key.

Returns
Configuration node when the configuration could be found, or `None` otherwise.

`get`(*key*)

`get`(*key*, *type*)

`get`(*key*, *type*, *default*)

`get`(*key*, *type*, *default*)

Returns a configuration value of any type.

Parameters

- **key** – Configuration key.
- **type** – Expected value type.
- **default** – Default value.

Returns

Configuration value if set, or default value if set, or `None` otherwise.

scenario.configini module

INI configuration file management.

`class ConfigIni`

Bases: `object`

INI configuration file management.

`static loadfile`(*path*, *root=''*)

Loads a INI configuration file.

Parameters

- **path** – Path of the INI file to load.
- **root** – Root key to load the INI file from.

`static savefile`(*path*, *root=''*)

Saves a INI configuration file.

Parameters

- **path** – Path of the INI file to save.
- **root** – Root key to save data from.

> **Warning:** Works only for *Dict[str, Dict[str, Union[str, int, bool, float]]]* dictionaries (i.e. *[section]/key = value* structures).

scenario.configjson module

JSON configuration file management.

`class ConfigJson`

Bases: `object`

JSON configuration file management.

`static loadfile`(*path*, *root=''*)

Loads a JSON configuration file.

Parameters

- **path** – Path of the JSON file to load.
- **root** – Root key to load the JSON file from.

`static savefile`(*path*, *root=''*)

Saves a JSON configuration file.

Parameters

- **path** – Path of the JSON file to save.
- **root** – Root key to save data from.

scenario.configkey module

Configuration key management.

`class ConfigKey`

Bases: `object`

Configuration key utils.

`static join`(*a*, *b*)

Joins key parts.

Parameters

- **a** – First key part to join.
- **b** – Second key part to join.

Returns

Concatenation of the two key parts.

scenario.confignode module

Configuration node management.

`class ConfigNode`

Bases: `object`

Considering that configurations are organized in a tree structure, this class represents a node of the tree, with either:

- a final item,
- a dictionary of *ConfigNode*,
- or a list of *ConfigNode*.

`__init__`(*parent*, *key*)

Parameters

- **parent** – Parent node. `None` for the root node.
- **key** – Key of the configuration node.

parent

Parent node.

`None` for the root node, as well as for removed nodes.

key

Configuration key.

_data

Configuration data.

Either:

- a final item,
- a dictionary of *ConfigNode*,
- or a list of *ConfigNode*.

origins

Origins of the configuration value: either a string or the path of the configuration file it was defined in.

__repr__()

Canonical string representation.

Gives the configuration key and type of data.

set(*data*, *subkey=None*, *origin=None*)

Sets configuration data.

Parameters

- **data** – Configuration data: dictionary, list or single value.

  When `None` is given and no `subkey` is provided, it is equivalent to calling *remove()* on the current node.

- **subkey** – Relative key from this node to store the data in.
- **origin** – Origin of the configuration data: either a simple string, or the path of the configuration file it was defined in. Defaults to code location when not set.

_setdata(*data*)

Sets the node's data, applying conversions when applicable, and displays debug info on the data stored.

Parameters
data – Node's data being set.

remove()

Removes the node from its parent.

Note: Does nothing on the root node (no parent for the root node, by definition).

show(*log_level*)

Displays the configuration database with the given log level.

Parameters
log_level – `logging` log level.

getkeys()

Retrieves the list of full keys from this node.

Returns
List of full keys.

`getsubkeys`()

Retrieves the list of sub-keys from this node.

Returns
List of sub-keys.

`get`(*subkey*)

Finds a sub-node from this node.

Parameters
subkey – Sub-key from this node.

Returns
Sub-node if found, `None` otherwise.

`_getsubnode`(*subkey*, *create_missing=False*, *origin=None*)

Finds or creates a sub-node from this node.

Parameters

- **subkey** – Sub-key from this node.
- **create_missing** – `True` to create missing sub-nodes.
- **origin** – Origin info to set for each sub-node walked through or created, starting from this one.

Returns
Sub-node if found, `None` otherwise.

`property data`

Retrieves the node data as a JSON-like structure, or value as given.

Returns
JSON-like structure or value.

`cast`(*type*)

Ensures the retrieval of the node data with the expected type.

Parameters
type – Expected type.

Returns
JSON-like structure or value of the expected type.

When the configuration data is not of the expected type, a `ValueError` is raised.

`property origin`

Representative origin for the current node.

`errmsg`(*msg*, *origin=None*)

Builds an error message giving the context of the current node.

Parameters

- **msg** – Detailed message.
- **origin** – Specific origin info. Use of *origins* by default.

Returns
Error message.

scenario.configtypes module

Configuration management types.

scenario.configyaml module

YAML configuration file management.

class ConfigYaml

Bases: object

YAML configuration file management.

static loadfile(*path*, *root=''*)

Loads a YAML configuration file.

Parameters

- **path** – Path of the YAML file to load.
- **root** – Root key to load the YAML file from.

static savefile(*path*, *root=''*)

Saves a YAML configuration file.

Parameters

- **path** – Path of the YAML file to save.
- **root** – Root key to save data from.

scenario.console module

Console management.

class Console

Bases: object

Console management.

class Color

Bases: IntEnum

Log colors.

Color numbers as they are used in the console.

RESET = 0

Code color to reset colors.

WHITE01 = 1

White.

DARKGREY02 = 2

Dark grey.

BLACK30 = 30

Black.

RED31 = 31
Red.

GREEN32 = 32
Green.

YELLOW33 = 33
Yellow.

DARKBLUE34 = 34
Dark blue.

PURPLE35 = 35
Purple.

LIGHTBLUE36 = 36
Light blue.

LIGHTGREY37 = 37
Light grey.

DARKGREY90 = 90
Another dark grey.

RED91 = 91
Another red.

GREEN92 = 92
Another green.

YELLOW93 = 93
Another yellow.

DARKBLUE94 = 94
Another dark blue.

PURPLE95 = 95
Another purple.

LIGHTBLUE96 = 96
Another light blue.

WHITE97 = 97
Another white.

LIGHTGREY98 = 98
Another light grey.

disableconsolebuffering()
Disables stdout & stderr buffering.

scenario.datetimeutils module

Date/time conversions from timestamp to ISO8601.

Note: This modules intends to centralize date/time conversions from timestamp to ISO8601, which remains a pain in Python. Indeed, full support of timezones with their 'Zoulou' or '+/-00:00' forms is not provided by default.

`DURATION_REGEX`

Regular expression matching a duration as displayed by *scenario* (i.e. last part of ISO8601).

`ISO8601_REGEX`

Regular expression matching ISO8601 date/times.

`toiso8601`(*timestamp*, *timezone=None*)

Formats a timestamp to a ISO8601 string.

Parameters

- **timestamp** – Input timestamp.
- **timezone** – Optional timezone specification. `None` stands for the local timezone.

Returns
ISO8601 string.

Raises
ValueError – When the operation could not be completed.

`fromiso8601`(*iso8601*)

Parses a ISO8601 string in a timestamp.

Parameters
iso8601 – Input ISO8601 string.

Returns
Timestamp.

Raises
ValueError – When the operation could not be completed.

`f2strtime`(*timestamp*)

Computes a string representation for the given timestamp.

Parameters
timestamp – Timestamp to convert.

Returns
String representation of the timestamp.

`f2strduration`(*duration*)

Computes a string representation for a time duration.

Parameters
duration – Time duration to convert.

Returns
String representation of the duration.

str2fduration(*duration*)

Parses a time duration from its string representation as computed by *f2strduration()*.

Parameters
duration – String representation of the time duration as computed by *f2strduration()*.

Returns
Time duration.

scenario.debugclasses module

scenario debugging.

The *DebugClass* enum defines the *scenario* debug classes (see: *Logger*).

class DebugClass

Bases: *StrEnum*

scenario debug classes.

ARGS = 'scenario.Args'

Program arguments debugging.

CAMPAIGN_REPORT = 'scenario.CampaignReport'

Campaign report debugging.

CAMPAIGN_RUNNER = 'scenario.CampaignRunner'

Campaign runner debugging.

CONFIG_DATABASE = 'scenario.ConfigDatabase'

Configuration database debugging.

EXECUTION_LOCATIONS = 'scenario.ExecutionLocations'

Execution location debugging.

HANDLERS = 'scenario.Handlers'

Handlers.

LOG_STATS = 'scenario.LogStats'

Logging statistics.

REFLEX = 'scenario.reflex'

Reflexive programmation debugging.

SCENARIO_REPORT = 'scenario.ScenarioReport'

Scenario report debugging.

SCENARIO_RESULTS = 'scenario.ScenarioResults'

Scenario results debugging.

SCENARIO_RUNNER = 'scenario.ScenarioRunner'

Scenario runner debugging.

SCENARIO_STACK = 'scenario.ScenarioStack'

Scenario stack debugging.

TEST_SUITE_FILE = 'scenario.TestSuiteFile'

Test suite file debugging.

EXECUTION_TIMES = 'scenario.#65.exec-times'

Issue#65 debugging: execution times.

scenario.debugloggers module

Extra debugging loggers.

class ExecTimesLogger

Bases: *Logger*

Issue#65 logger.

__init__(*context*)

Creates an execution times logger for the given context.

Parameters
context – Context, usually a function/method name.

context

Debug logger context. Usually a function/method name.

t0

Starting time for this debug logger.

_last_tick

Last tick time.

tick(*message*)

Logs intermediate time information.

Parameters
message – Object of this tick.

finish()

Terminates logging for the given context.

scenario.debugutils module

Functions and classes for debugging.

class DelayedStr

Bases: ABC

Abstract class that defines a string which computation can be delayed.

The main interest of it is to postpone heavy processing, so that if ever useless, it is not executed at all.

__init__()

Initializes the string computation result cache.

__str

Cached string computation result.

__repr__()

Canonical string representation.

This method may be useless. Whatever, let's return the canonical representation of the string defined by this object.

__str__()

Triggers the string computation on the first call, and cache it for later calls.

abstract _computestr()

String computation handler.

Returns

String computed for the object.

Will be cached by *__str__()*.

_abc_impl = <_abc_data object>

class FmtAndArgs

Bases: *DelayedStr*

Makes it possible to prepare a string format with its corresponding arguments, as usual with the % operator, and have it computed if needed.

__init__(*fmt=''*, **args*)

Prepares the string format and arguments, possibly with initial values.

Parameters

- **fmt** – Inital string format.
- **args** – Initial string arguments.

fmt

String format.

args

Format arguments.

push(*fmt*, **args*)

Pushes additional format and arguments.

Makes it possible to prepare the string step by step, and/or conditionally.

Parameters

- **fmt** – String format to appens.
- **args** – Corresponding arguments.

Returns

self

_computestr()

String computation handler.

Returns

String computed for the object.

Will be cached by __str__().

_abc_impl = <_abc_data object>

class SafeRepr

Bases: *DelayedStr*

Delays the computation of the safe canonical representation of an object.

Same as `unittest`, safe representation means that the string computed while not exceed a given length, so that it remains human readable.

__init__(*obj*, *max_length=256*, *focus=None*)

Stores the object reference for later safe representation computation.

Parameters

- **obj** – Object to represent.
- **max_length** – Maximum length for the resulting string.
- **focus** – Data to focus on.

obj

Object to represent.

max_length

Maximum length for the resulting string.

focus

Data to focus on.

_computestr()

String computation handler.

Returns

String computed for the object.

Will be cached by `__str__()`.

_abc_impl = <_abc_data object>

saferepr(*obj*, *max_length=256*, *focus=None*)

Safe representation of an object.

Parameters

- **obj** – Object to represent.
- **max_length** – Maximum length for the resulting string.
- **focus** – Data to focus on.

Returns

SafeRepr delayed computation object.

class JsonDump

Bases: *DelayedStr*

Delays the dump of JSON data.

__init__(*json_data*, ***kwargs*)

Stores the JSON data for later dump.

Parameters

- **json_data** – JSON data to dump.

- **kwargs** – `json.dumps()`-like arguments.

json_data

JSON data to dump.

kwargs

`json.dumps()`-like arguments.

_computestr()

String computation handler.

Returns

String computed for the object.

Will be cached by `__str__()`.

_abc_impl = <_abc_data object>

jsondump(*json_data*, ***kwargs*)

Dump of JSON data.

Parameters

- **json_data** – JSON data to dump.
- **kwargs** – `json.dumps()`-like arguments.

Returns

JsonDump delayed computation object.

class CallbackStr

Bases: *DelayedStr*

String builder callback manager.

__init__(*callback*, **args*, ***kwargs*)

Stores the callback with its arguments for later execution.

Parameters

- **callback** – String builder callback.
- **args** – Callback positional arguments.
- **kwargs** – Callback named arguments.

callback

String builder callback.

args

Callback positional arguments.

kwargs

Callback named arguments.

_computestr()

String computation handler.

Returns

String computed for the object.

Will be cached by `__str__()`.

_abc_impl = <_abc_data object>

callback(*callback*, **args*, ***kwargs*)

Stores a string builder callback with its arguments for later execution.

Parameters

- **callback** – String builder callback.
- **args** – Callback positional arguments.
- **kwargs** – Callback named arguments.

Returns

CallbackStr delayed computation object.

scenario.enumutils module

enum augmentations.

class StrEnum

Bases: str, Enum

String enum.

_member_names_ = []

_member_map_ = {}

_member_type_

alias of str

_value2member_map_ = {}

enum2str(*value*)

Ensures a string value from a string/enum union.

Parameters

value – String already, or string enum.

Returns

String.

Note: value if given as an enum is basically expected to be a string enum. Whether this is not the case, the value is converted as a string anyways.

---

scenario.errcodes module

Command line error codes.

Error codes returned by the *ScenarioRunner* or *CampaignRunner* programs.

class ErrorCode

Bases: IntEnum

Error codes enum.

Note: Codes inspired from HTTP status codes, but with error codes less than 256.

- 20-29: Normal errors.
- 40-49: Input related errors.
- 50-59: Processing and output related errors.

SUCCESS = 0

Success.

TEST_ERROR = 21

When a test failed.

ENVIRONMENT_ERROR = 40

Errors due to the environnement.

ARGUMENTS_ERROR = 41

Errors due to invalid arguments.

INPUT_MISSING_ERROR = 42

Errors due to missing inputs.

INPUT_FORMAT_ERROR = 43

Errors due to invalid input format.

INTERNAL_ERROR = 50

Internal error.

static worst(*error_codes*)

Returns the worst error code from the list.

The higher the error value, the worse.

Parameters
error_codes – List to find the worst error code from.

Returns
Worst error code.

scenario.executionstatus module

Execution status management.

class ExecutionStatus

Bases: *StrEnum*

Scenario & campaign execution status.

SUCCESS = 'SUCCESS'

Success.

WARNINGS = 'WARNINGS'

Success with warnings.

FAIL = 'FAIL'

Failure.

SKIPPED = 'SKIPPED'

Test skipped.

UNKOWN = 'UNKNOWN'

Unknown status.

scenario.handlers module

Handler management.

HANDLERS

Main instance of *Handlers*.

class Handler

Bases: `object`

Handler storage.

__init__(*event*, *handler*, *scenario_definition*, *once*)

Parameters

- **event** – Event triggered.
- **handler** – Handler function.
- **scenario_definition** – Related scenario, if any.
- **once** – *Once* flag.

event

Event triggered.

handler

Handler function.

scenario_definition

Related scenario, if any.

once

Once flag.

class Handlers

Bases: *Logger*

Handler manager.

__init__()

Initializes an empty handler list.

_handlers

Installed handlers.

Dictionary that associates events with their related handler list.

install(*event*, *handler*, *scenario=None*, *once=False*, *first=False*)

Installs a handler.

Parameters

- **event** – Event triggered.

- **handler** – Handler function.
- **scenario** – Related scenario definition, if any.
- **once** – *Once* flag.
- **first** – `True` to install this handler at the head of the list attached with the event given.

> **Warning:** Does not prevent a later handler to be installed before this one.

uninstall(*event*, *handler*)

Removes the handler.

Parameters

- **event** – Event triggered.
- **handler** – Handler function.

callhandlers(*event*, *data*)

Calls applicable handlers for the given event.

Parameters

- **event** – Event met.
- **data** – Event data to pass on when calling each handler.

scenario.issuelevels module

Issue levels.

class IssueLevel

Bases: `ABC`

Abstract class that gathers useful issue level methods.

_named = {}

Named issue levels.

static definenames(*named_issue_levels*)

Defines the named issue level list.

Parameters
named_issue_levels – New issue level definition.

Resets names previously defined if any.

static addname(*__issue_level*)

static addname(*__name*, *__issue_level*)

Add an issue level name.

See overloads for argument details.

static getnamed()

Retrieves the current list of named issue levels.

Returns
Dictionary of {`str` name => `int` issue level}.

static getnameddesc(*reverse=False*)

Retrieves a textual description for the current list of named issue levels.

Parameters
reverse – `True` to sort names by descending issue levels, `False` by default.

Returns
'<name>=<int>' comma separated string, sorted depending on `reverse`.

_abc_impl = <_abc_data object>

static getdesc(*level*)

Retrieves a textual description for the given issue level.

Parameters
level – Issue level to describe.

Returns
'<name>=<int>' or '<int>' description depending on whether `level` is an `enum.IntEnum` or an `int`.

static parse(*level*)

Converts an optional `str` or `int` value to a `enum.IntEnum` if given in the named issue levels, or a simple `int`.

Parameters
level – `str` or `int` data to parse.

Returns
`enum.IntEnum` or `int` value.

Logs a warning if named issue levels are set but the given issue level number does not match with any.

scenario.knownissues module

Known issues.

exception KnownIssue(*message*, *level=None*, *id=None*, *url=None*)

Bases: *TestError*

Known issue object.

May be:

- considered as an error,
- considered as a warning,
- ignored.

_url_builder

URL builder handler configured.

static seturlbuilder(*url_builder*)

Sets or unsets a URL builder handler.

Parameters
url_builder – URL builder handler to set, or `None` to unset.

This handler shall return a URL string when it succeeded in building an URL for a given issue identifier, or `None` otherwise.

__init__(*message*, *level=None*, *id=None*, *url=None*)

Creates a known issue instance from the info given and the current execution stack.

Parameters

- **message** – Error or warning message to display with.
- **level** – Issue level. Optional.
- **id** – Issue identifier. Optional.
- **url** – Issue URL. Optional.

level

Issue level.

id

Issue identifier.

_url

Issue URL set, or computed from *id* and *_url_builder*.

location

Redefinition of `TestError.location` in order to explicitize it cannot be `None` for *KnownIssue* instances.

__str__()

Short representation of the known issue.

'Issue(({level-name}=){level})({id})! {message}'.

property url

Issue URL getter.

Returns

Issue URL if set, `None` otherwise.

static fromstr(*string*)

Builds a *KnownIssue* instance from its string representation.

Parameters

string – String representation, as computed by *__str__()*.

Returns

New *KnownIssue* instance.

__eq__(*other*)

Known isue equality operator.

Parameters

other – Candidate object.

Returns

`True` when known issues hold the same information, `False` otherwise.

iserror()

Tells whether this error object is actually an error.

Returns

`True` for a real error, `False` for a simple warning (see *iswarning()*) or when the error should be ignored (see *isignored()*).

`iswarning`()

Tells whether this error object is just a warning.

Returns
`True` for a simple warning, `False` for a real error (see *iserror()*) or when the error should be ignored (see *isignored()*).

`isignored`()

Tells whether this error object should be ignored.

Returns
`True` when the error should be ignored, `False` for a real error (see *iserror()*) or a warning (see *iswarning()*).

`logerror`(*logger*, *level=30*, *indent=''*)

TestError.logerror() override in order to display the issue URL on a second line (if any).

`tojson`()

Converts the `TestError` instance into a JSON dictionary.

Returns
JSON dictionary.

`static fromjson`(*json_data*)

Builds a *KnownIssue* instance from its JSON representation.

Parameters
json_data – JSON dictionary.

Returns
New *KnownIssue* instance.

scenario.locations module

Execution location management.

Execution locations may be used:

- to locate a class / function / method definition (see *ScenarioDefinition* and *StepDefinition*),
- to locate the place of the current execution, or where an exception occurred.

`EXECUTION_LOCATIONS`

Main instance of *ExecutionLocations*.

`class CodeLocation`

Bases: `object`

Class that describes a code location, i.e. a point where an element is defined, or the test execution takes place.

`static fromtbitem`(*tb_item*)

Computes an *CodeLocation* based on a traceback item.

Parameters
tb_item – Traceback item.

Returns
CodeLocation instance.

static frommethod(*method*)

Computes an ExecutionLocation based on a method.

Parameters
method – Method to locate.

Returns
CodeLocation instance.

static fromclass(*cls*)

Computes an ExecutionLocation based on a class.

Parameters
cls – Class to locate.

Returns
CodeLocation instance.

__init__(*file*, *line*, *qualname*)

Inititializes the *CodeLocation* instance with the given values.

Parameters

- **file** – File path where the execution takes place.
- **line** – Line in the file where the execution takes place.
- **qualname** – Qualified name of the class/function pointed.

file

File path.

Set as a *Path* when file is passed on as a *Path*. Set as a pathlib.Path otherwise, possibly a relative path in that case.

line

Line number in the file.

qualname

Method name.

__eq__(*other*)

Compares the *CodeLocation* instance with another object.

Parameters
other – Candidate object.

Returns
True if the objects are similar, False otherwise.

tolongstring()

Long text representation.

static fromlongstring(*long_string*)

Computes an ExecutionLocation from its long text representation.

Parameters
long_string – Long text, as returned by *tolongstring()*.

Returns
CodeLocation instance.

class ExecutionLocations

Bases: *Logger*

Methods to build execution location stacks.

__init__()

Sets up logging for the *ExecutionLocations* class.

fromcurrentstack(*limit=None*, *fqn=False*)

Builds a stack of *CodeLocation* from the current call stack.

Parameters

- **limit** – Maximum number of backward items.
- **fqn** – `True` to ensure fully qualified names.

Returns

Stack of *CodeLocation*.

fromexception(*exception*, *limit=None*, *fqn=False*)

Builds a stack of *CodeLocation* from an exception.

Parameters

- **exception** – Exception to build the stack from.
- **limit** – Maximum number of backward items.
- **fqn** – `True` to ensure fully qualified names.

Returns

Stack of *CodeLocation*.

_fromtbitems(*tb_items*, *limit=None*, *fqn=False*)

Builds a stack of *CodeLocation* from traceback items.

Parameters

tb_items – Traceback items to build the stack from.

Returns

Stack of *CodeLocation*.

scenario.logextradata module

Logging extra data handling.

class LogExtraData

Bases: *StrEnum*

Keys to can be used in the dictionary passed on in the `extra` parameter of the standard `logging` functions.

CURRENT_LOGGER = '_logger_'

Current logger reference.

Stores a class:.*logger.Logger* instance reference.

Automatically set by *LoggerLogFilter* so that *LogFormatter* knows about the current *Logger* instance when processing the log event.

LONG_TEXT_MAX_LINES = '_long_text_max_lines_'

Long text mode option.

When set, activates the *long text mode*.

`int` that gives the maximum number of lines to display.

DATE_TIME = '_date_time_'

Extra flag: Should date/time be displayed?

`True` by default.

Still depends on usual `logging` configurations.

COLOR = '_color_'

Extra flag: May color be used? (console only)

`True` by default.

Still depends on usual `logging` configurations.

LOG_LEVEL = '_log_level_'

Extra flag: Should the log level be displayed?

`True` by default.

SCENARIO_STACK_INDENTATION = '_scenario_stack_indentation_'

Extra flag: Should the scenario stack indentation be displayed?

`True` by default.

MAIN_LOGGER_INDENTATION = '_main_logger_indentation_'

Extra flag: Should the main logger indentation be displayed?

`True` by default.

CLASS_LOGGER_INDENTATION = '_class_logger_indentation_'

Extra flag: Should the main logger indentation be displayed?

`True` by default.

ACTION_RESULT_MARGIN = '_action_result_margin_'

Extra flag: Should the log be applied a margin that makes it indented within the action/result block it belongs to?

`True` by default.

static extradata(*extra*)

Translates a *{ExtraFlag: Any}* dictionary into a `logging` compatible dictionary.

The resulting dictionary basically deserves the `extra` parameter of `logging` functions.

Parameters
extra – Enum dictionary to translate.

Returns
`logging` compatible dictionary.

static get(*record*, *key*)

Retrieves extra data from a record.

Parameters

- **record** – Record to look for extra data in.

- **key** – Extra data name to look for.

Returns
Extra data value if set, or `None` otherwise.

`static set`(*record*, *key*, *value*)

Sets extra data with a record.

Parameters

- **record** – Record to store extra data in.
- **key** – Extra data name to set.
- **value** – Extra data value.

scenario.logfilters module

Log filtering.

`class LoggerLogFilter`

Bases: `Filter`

Log filter attached to a *Logger* instance.

Filters log records depending on the log level and the associated logger configuration.

`__init__`(*logger*)

Parameters
logger – Attached *Logger* instance.

`_logger`

Attached *scenario* *Logger* instance.

`filter`(*record*)

Filtering hook implementation.

Parameters
record – Log record to check for filtering.

Returns

See `logging.Filter.filter()`: "Is the specified record to be logged? Returns 0 for no, nonzero for yes."

Nevertheless, we can see from the code that booleans are actually returned.

Checks whether the log record should be filtered out due to the attached `Logger` configuration.

`class HandlerLogFilter`

Bases: `Filter`

Log filter attached to a `logging.Handler` instance.

Filters log records depending on *scenario* configurations: *ScenarioConfig.Key.LOG_CONSOLE* and *ScenarioConfig.Key.LOG_FILE*.

`__init__`(*handler*)

Parameters
handler – Attached `logging.Handler`.

`_handler`

Attached `logging.Handler`.

`filter`(*record*)

Filtering hook implementation.

Parameters
record – Log record to check for filtering.

Returns

See `logging.Filter.filter()`: "Is the specified record to be logged? Returns 0 for no, nonzero for yes."

Nevertheless, we can see from the code that booleans are actually returned.

Checks the *ScenarioConfig.Key.LOG_CONSOLE* or *ScenarioConfig.Key.LOG_FILE* configurations, depending on the handler attached.

scenario.logformatter module

Log record formatting.

`class LogFormatter`

Bases: `Formatter`

Formats log records.

Log record formatting includes the following aspects:

Date/time display
See *log date/time* documentation.

Displayed by default, unless it is disabled through the *ScenarioConfig.Key.LOG_DATETIME* configuration, or the `ExtraFlag.DATE_TIME` extra flag.

Log level display
See *log levels* documentation.

Log level is always displayed, unless it is disabled through the `ExtraFlag.LOG_LEVEL` extra flag.

Date/time display
See *log date/time* documentation.

Displayed by default, unless it is disabled through the *ScenarioConfig.Key.LOG_DATETIME* configuration, or the `ExtraFlag.DATE_TIME` extra flag.

Log level display
See *log levels* documentation.

Log level is always displayed, unless it is disabled through the `ExtraFlag.LOG_LEVEL` extra flag.

Log class display
See *class loggers* documentation.

Indentation
See *log indentation* documentation.

Colorization
See *log colors* documentation.

Console log colorization may be disabled through the *ScenarioConfig.Key.LOG_CONSOLE* configuration, or the `ExtraFlag.COLOR` extra flag.

`__init__`(*handler*)

Parameters
handler – Attached `logging.Handler`.

`_handler`

Attached `logging.Handler`.

`format`(*record*)

`logging` method overload that implements most of the *scenario* log formatting expectations.

Parameters
record – Log record to format for printing.

Returns
Log string representation.

`_with`(*record*, *extra_flag*, *default=True*)

Tells whether the logging aspect described by `extra_flag` is on or off for the given record.

Parameters

- **extra_flag** – Extra flag / logging aspect to check.
- **default** – Default return value. May be set to `False` when required.

Returns
`True` if the logging aspect described by `extra_flag` in on for the current record, `False` otherwise.

Depends on :

1. The extra flags set in the log record,
2. The scenario configuration,
3. The current execution state.

`static _levelcolor`(*level*)

Determines log color out from log level.

Parameters
level – Log level which respective color to find out.

Returns
Log color corresponding to the given log level.

`static nocolor`(*string*)

Removes color control characters from a string.

Parameters
string – String to remove color control characters from.

Returns
String without color control characters.

scenario.logger module

Logger class definition.

_main_loggers

Number of main loggers already created.

Constitutes a guard against the creation of several main loggers, i.e. loggers without a *log class*.

class Logger

Bases: `object`

scenario logger base class for the main logger and sub-loggers.

The *Logger* class enables you to make your log lines be controlled by a *log class*. This will make the log lines be prefixed with the given log class, and give you the opportunity to activate or deactivate the corresponding debug log lines programmatically (see *enabledebug()*) or by configuration (see *ScenarioConfig.debugclasses()*).

__init__(*log_class*)

Parameters
log_class – Log class.

Empty string for the main logger (for the main logger only!).

See also:

enabledebug() and *setlogcolor()*.

log_class

Log class.

_logger

`logging.Logger` instance as a member variable.

_debug_enabled

`True` to enable log debugging. `None` lets the configuration tells whether debug log lines should be displayed for this logger.

_log_color

Optional log color configuration.

_indentation

Logger indentation.

_extra_flags

Extra flags configurations.

property logging_instance

Provides the reference of the `logging.Logger` instance attached with this *Logger* instance.

enabledebug(*enable_debug*)

Debug log enabling / disabling.

Parameters
enable_debug – `True` for debug log enabling, `False` otherwise.

Returns
`self`

See the *main logger* and *class loggers* sections to learn more about debugging with *Logger* instances.

`isdebugenabled`()

Tells whether debug logging is currently enabled for this *Logger* instance.

Returns
`True` when debug logging is enabled, `False` otherwise.

`setlogcolor`(*color*)

Sets or clears a log line color specialized for the logger.

Parameters
color – Log line color. `None` to reset to default.

Log class colorization offers the possibilty to differenciate log lines betwwen different loggers running at the same time, each one having its own color. See the *log class colorization* section for detailed information.

`getlogcolor`()

Returns the specialized log line color for this logger, if any.

Returns
Log line color. `None` when not set.

`pushindentation`(*indentation=' '*)

Adds indentation for this *Logger* instance.

Parameters
indentation – Optional indentation pattern.

See the dedicated sections to learn more about the differences between calling this method *on the main logger* on the one hand, and *on a class logger* on the other hand.

`popindentation`(*indentation=' '*)

Removes indentation for the *Logger* instance.

Parameters
indentation – Optional indentation pattern. Must be the same as the indentation pattern passed on with the matching *pushindentation()* call on a LIFO basis (Last-In First-Out).

`resetindentation`()

Resets the indentation state attached with this *Logger* instance.

`getindentation`()

Returns the current indentation attached with this *Logger* instance.

Returns
Current indentation.

`setextraflag`(*extra_flag*, *value*)

Sets or unsets an extra flag configuration.

Parameters

- **extra_flag** – Extra flag name.
- **value** – Extra flag configuration. `None` to unset the extra flag configuration.

`getextraflag`(*extra_flag*)

Returns the extra flag configuration set (or not).

Parameters
extra_flag – Extra flag name.

Returns
`True` or `False` when the configuration is set, or `None` otherwise.

error(*msg*, **args*, ***kwargs*)

Logs an error message with this logger.

warning(*msg*, **args*, ***kwargs*)

Logs a warning message with this logger.

info(*msg*, **args*, ***kwargs*)

Logs an informational message with this logger.

debug(*msg*, **args*, ***kwargs*)

Logs a debug message with this logger.

The processing of the message depends on the *_debug_enabled* configuration (see *enabledebug()*).

log(*level*, *msg*, **args*, ***kwargs*)

Logs a message with a configurable severity.

_log(*level*, *msg*, *args*, ***kwargs*)

`logging.Logger._log()` method indirection.

Parameters

- **self** – In as much as `self` is bound with the method, even though the call was made from a `logging.Logger` instance, `self` remains a *scenario* *Logger* when we arrive here.
- **level** – Log level.
- **msg** – Log message.
- **args** – Other positional arguments as a tuple.
- **kwargs** – Named parameter arguments.

Handles appropriately the optional `exc_info` parameter.

_torecord(*level*, *msg*, *args*, ***kwargs*)

After the *_log()* indirection, eventually sends the log data to the base `logging` module to create a log record.

Parameters

- **level** – Log level.
- **msg** – Log message.
- **args** – Other positional arguments as a tuple.
- **kwargs** – Named parameter arguments.

longtext(*max_lines*)

Builds the *long text extra* option in order to display the log message as several lines.

Parameters
max_lines – Maximum number of lines.

Returns
long text extra option.

See the *long text logging* section for more details.

_loglongtext(*level*, *msg*, *args*, *max_lines*, ***kwargs*)

Logs the beginning of a long text on multiple lines.

Parameters

- **level** – Log level.
- **msg** – Log message.
- **args** – Other positional arguments as a tuple.
- **max_lines** – Maximum number of lines to display. All lines when set to None.
- **kwargs** – Named parameter arguments.

scenario.loggermain module

MainLogger class definition with *MAIN_LOGGER* singleton.

MAIN_LOGGER

Main logger instance.

class MainLogger

Bases: *Logger*

Main logger augmentation of *Logger*.

__init__()

Enables debugging by default and makes console initializations.

rawoutput(*message*)

Logs a line with automatic formattings disabled.

Helper function for the *ScenarioLogging* and *CampaignLogging* classes.

Parameters
message – Log message to output.

scenario.loggingargs module

Common logging program arguments.

class CommonLoggingArgs

Bases: object

Base class for argument parser classes that embed common logging program arguments.

__init__(*class_debugging*)

Installs common logging program arguments.

Parameters
class_debugging – True to enable per-class debugging, False for unclassed debugging only.

When per-class debugging is enabled, the main logger debugging is enabled by default.

debug_main

Main logger debugging.

debug_classes

Debug classes.

scenario.loggingservice module

Logging service.

LOGGING_SERVICE

Main instance of *LoggingService*.

class LoggingService

Bases: object

Logging service management class.

start()

Starts logging features.

stop()

Stops logging features.

scenario.loghandler module

Log handling.

class LogHandler

Bases: object

Log handler management.

console_handler = <StreamHandler <stdout> (NOTSET)>

Console handler instance.

Created with the main logger.

file_handler

File handler instance, when started.

Created when the logging service is started and file logging is required.

scenario.path module

Path management.

AnyPathType

Type for path-like data: either a simple string or a os.PathLike instance.

class Path

Bases: object

Helper class for path management.

This class really looks like pathlib.Path, but differs from it in that:

1. it ensures persistent paths, even though initialized from a relative path and the current directory changes afterwards,
2. it provides a *prettypath* display from a main directory set for the current project (see *setmainpath()*),
3. it does not describe the current working implicitely when initialized from nothing, but a *void* path.

The *Path* class supports the os.PathLike interface.

_main_path

Main path, used to compute the relative *prettypath*. Unset by default.

static setmainpath(*path*, *log_level=20*)

Sets the main path, used to compute the relative *prettypath*.

Parameters

- **path** – New main path.
- **log_level** – Log level (as defined by the standard `logging` package) to use for the related log line.

static getmainpath()

Returns
Main path, i.e. base path for *prettypath* computations.

static cwd()

Computes a *Path* instance representing the current working directory.

Returns
Current working directory.

static home()

Computes a *Path* instance representing the current user's home directory.

Returns
Current user's home directory.

static tmp()

Computes a *Path* instance representing the temporary directory.

Returns
Temporary directory.

__init__(*path=None*, *relative_to=None*)

Ensures the management of an absolute path.

Parameters

- **path** – File or directory path as a path-like.

  Makes the *Path* instance *void* when not set.
- **relative_to** – Base directory or file to consider as the root, when the path given is a relative path.

  Giving a file path as `relative_to` is equivalent to giving its owner directory.

If the path given is relative, it is transformed in its absolute form from the current working directory.

_abspath

`pathlib.Path` instance used to store the absolute path described by this *Path* instance.

as_posix

Shortcut to `pathlib.PurePath.as_posix()`.

as_uri

Shortcut to `pathlib.PurePath.as_uri()`.

is_absolute
Shortcut to pathlib.PurePath.is_absolute().

is_reserved
Shortcut to pathlib.PurePath.is_reserved().

match
Shortcut to pathlib.PurePath.match().

stat
Shortcut to pathlib.Path.stat().

chmod
Shortcut to pathlib.Path.chmod().

exists
Shortcut to pathlib.Path.exists().

expanduser
Shortcut to pathlib.Path.expanduser().

group
Shortcut to pathlib.Path.group().

is_dir
Shortcut to pathlib.Path.is_dir().

is_file
Shortcut to pathlib.Path.is_file().

is_mount
Shortcut to pathlib.Path.is_mount().

is_symlink
Shortcut to pathlib.Path.is_symlink().

is_socket
Shortcut to pathlib.Path.is_socket().

is_fifo
Shortcut to pathlib.Path.is_fifo().

is_block_device
Shortcut to pathlib.Path.is_block_device().

is_char_device
Shortcut to pathlib.Path.is_char_device().

lchmod
Shortcut to pathlib.Path.lchmod().

lstat
Shortcut to pathlib.Path.lstat().

mkdir
Shortcut for pathlib.Path.mkdir().

open
Shortcut to `pathlib.Path.open()`.

owner
Shortcut to `pathlib.Path.owner()`.

read_bytes
Shortcut to `pathlib.Path.read_bytes()`.

read_text
Shortcut to `pathlib.Path.read_text()`.

readlink
Shortcut to `pathlib.Path.readlink()`.

rmdir
Shortcut to `pathlib.Path.rmdir()`.

symlink_to
Shortcut to `pathlib.Path.symlink_to()`.

hardlink_to
Shortcut to `pathlib.Path.hardlink_to()`.

link_to
Shortcut to `pathlib.Path.link_to()`.

touch
Shortcut to `pathlib.Path.touch()`.

unlink
Shortcut to `pathlib.Path.unlink()`.

write_bytes
Shortcut to `pathlib.Path.write_bytes()`.

write_text
Shortcut to `pathlib.Path.write_text()`.

__fspath__()
`os.PathLike` interface implementation.

__repr__()
Canonical string representation.

__str__()
Human readable string representation (same as *prettypath*).

__hash__()
Hash computation.

Makes it possible to use *Path* objects as dictionary keys.

property parts
See pathlib.PurePath.parts.

property drive
See pathlib.PurePath.drive.

property root

See pathlib.PurePath.root.

property anchor

See pathlib.PurePath.anchor.

property parents

Gives the list of parent directories as *Path* objects.

See pathlib.PurePath.parents.

property parent

Gives the parent directory as a *Path* object.

See pathlib.PurePath.parent.

property name

Base name of the path.

See pathlib.PurePath.name.

property suffix

Gives the extension of the file (or directory name), with its leading dot, if any, or an empty string if no extension.

See pathlib.PurePath.suffix.

property suffixes

Gives the list of consecutive extensions, with their leading dot character.

See pathlib.PurePath.suffixes.

property stem

Gives the basename of the path, without the final extension if any.

See pathlib.PurePath.stem.

property abspath

Absolute form of the path in the POSIX style.

property prettypath

Gives the pretty path.

The pretty path is actually a relative path from the main path if set (see *setmainpath()*), or the current working directory otherwise, and presented in the POSIX style.

resolve()

Retrieves another *Path* instance similar to this one.

Returns
New *Path* instance.

__eq__(*other*)

Checks whether `other` equals to this path.

Parameters
other – Path to checks against.

Returns
`True` when the paths equal, `False` otherwise.

samefile(*other*)

Returns `True` when `other` describes the same path as this one.

Parameters
other – Other path (or anything that is not a path at all).

Returns
`True` when `other` is the same path.

__truediv__(*other*)

Joins this directory path with a sub-path.

Parameters
other – Sub-path to apply from this directory path.

Returns
New *Path* instance.

joinpath(**other*)

Joins this directory path with a list of sub-paths.

Parameters
other – Sub-paths to apply from this directory path.

Returns
New *Path* instance.

with_name(*name*)

See pathlib.PurePath.with_name().

with_stem(*stem*)

See pathlib.PurePath.with_stem().

with_suffix(*suffix*)

See pathlib.PurePath.with_suffix().

is_void()

Tells whether this path is void.

Returns
`True` when the path is void, `False` otherwise.

is_relative_to(*other*)

Tells whether this path is a sub-path of the candidate parent directory.

Parameters
other – Candidate parent directory.

Returns
`True` when this path is a sub-path of `other`.

See pathlib.PurePath.is_relative_to().

relative_to(*other*)

Computes a relative path.

Parameters
other – Reference path to compute the relative path from.

Returns
Relative path from `other` in the POSIX style.

Note: The behaviour of this method differs from the one of `pathlib.PurePath.relative_to()`.

`pathlib.PurePath.relative_to()` raises a `ValueError` as soon as this path is not a sub-path of `other`. In order te be able compute relative paths beginning with "../", we use `os.path.relpath()` instead.

See pathlib.PurePath.relative_to().

iterdir()

Lists this directory path.

Returns
Paths iterator.

See pathlib.Path.iterdir().

glob(*pattern*)

Returns the list of files that match the given pattern.

Parameters
pattern – Path pattern (see `glob.glob()`). May be either a relative or an absolute path specification.

Returns
List of paths that match the pattern.

See pathlib.Path.glob().

rglob(*pattern*)

See pathlib.Path.rglob().

rename(*target*)

Moves this file or directory as `target`.

Parameters
target – Target path.

Returns
New target *Path* instance.

See pathlib.Path.rename().

replace(*target*)

See pathlib.Path.replace().

scenario.pkginfo module

scenario package information.

PKG_INFO

Main *PackageInfo* instance.

class PackageInfo

Bases: `object`

Package information.

property version

Current version of the *scenario* package as a semver string.

Returns
"x.y.z" version string.

property version_tuple

Current version of the *scenario* package as a semver tuple.

Returns
(x, y, z) version tuple.

scenario.reflex module

Reflexive programmation tools and Python augmentations.

REFLEX_LOGGER

Logger instance for reflexive programming.

qualname(*obj*)

Returns the qualified name of an object.

Parameters
obj – Object to retrieve the qualified name for.

Returns
Qualified name.

Note: Accessing directly the *__qualname__* attribute makes mypy generate errors like "'…' has no attribute '__qualname__'".

---

isiterable(*obj*)

Tells whether an object is iterable or not.

Parameters
obj – Object to check.

Returns
True when the object is iterable, False otherwise.

Inspired from https://stackoverflow.com/questions/1952464/in-python-how-do-i-determine-if-an-object-is-iterable#1952481.

importmodulefrompath(*script_path*)

Imports a module from its Python script path.

Parameters
script_path – Python script path.

Returns
Module loaded.

getloadedmodulefrompath(*script_path*)

Retrieves a module already loaded corresponding to the given path.

Parameters
script_path – Python script path.

Returns
Corresponding module if already loaded.

`checkfuncqualname`(*file*, *line*, *func_name*)

Tries to retrieve the fully qualified name of a function or method.

Parameters

- **file** – Path of the file the function is defined int.
- **line** – Line number inside the function.
- **func_name** – Short name of the function.

Returns
Fully qualified name of the function, or `func_name` as is by default.

`codelinecount`(*code*)

Retrieves the number of lines of the given code object.

Parameters
code – Code object which lines to count.

Returns
Number of lines.

Apparently, inspect does give the straight forward information.

The *co_lnotab* attribute, being an "encoded mapping of line numbers to bytecode indices", is our best chance for the purpose.

The https://svn.python.org/projects/python/branches/pep-0384/Objects/lnotab_notes.txt resource gives complementary information on how to parse this *co_lnotab* attribute.

***co_lnotab* depreciation.**

According to https://www.python.org/dev/peps/pep-0626/#backwards-compatibility: "The co_lnotab attribute will be deprecated in 3.10 and removed in 3.12."

---

scenario.scenarioargs module

Scenario runner program arguments.

`class CommonExecArgs`

Bases: `object`

Base class for argument parser classes that embed common test execution program arguments.

`__init__`()

Installs common test execution program arguments.

`doc_only`

`True` when the test(s) is(are) executed for documentation generation only, i.e. the test script(s) for actions and verifications should not be executed.

`issue_level_error`

Error issue level.

issue_level_ignored

Ignored issue level.

_checkargs(*args*)

Check common test execution program arguments once parsed.

Returns

True for success, False otherwise.

static reportexecargs(*args*, *subprocess*)

Report execution program arguments from an argument parser instance to the arguments of a sub-process being built.

Parameters

- **args** – Argument parser instance to report program arguments from to.
- **subprocess** – Sub-process being built to report program arguments to.

class ScenarioArgs

Bases: *Args*, *CommonExecArgs*

Scenario runner program argument management.

Provides arguments for the *ScenarioRunner* execution.

Arguments given through the command line prevail on the configurations in the configuration files (see *ScenarioConfig*).

__init__(*positional_args=True*)

Declares the scenario runner program arguments, and binds them to the member fields.

Parameters

positional_args – `False` to disable the scenario path positional arguments definition. Useful for user programs that wish to redefine it.

json_report

JSON report output file path. No JSON report when `None`.

extra_info

Attribute names to display for extra info. Applicable when executing several tests.

scenario_paths

Path of the scenario Python script to execute.

_checkargs(*args*)

Checks scenario runner arguments once parsed.

Returns

`True` for success, `False` otherwise.

scenario.scenarioconfig module

scenario framework configurations.

ScenarioConfigKey

Shortcut to the *ScenarioConfig.Key* enum in order to make it possible to import it without the *ScenarioConfig* class.

SCENARIO_CONFIG

Main instance of *ScenarioConfig*.

class ScenarioConfig

Bases: `object`

scenario configuration management.

This class defines static methods that help reading *scenario* configurations: from the program arguments (see: *Args*), and the configuration database (see: *ConfigDatabase*).

class Key

Bases: *StrEnum*

scenario configuration keys.

TIMEZONE = 'scenario.timezone'

Should a specific timezone be used? String value. Default is the local timezone.

LOG_DATETIME = 'scenario.log_date_time'

Should the log lines include a timestamp? Boolean value.

LOG_CONSOLE = 'scenario.log_console'

Should the log lines be displayed in the console? Boolean value.

LOG_COLOR_ENABLED = 'scenario.log_color'

Should the log lines be colored? Boolean value.

LOG_COLOR = 'scenario.log_%s_color'

Log color per log level. Integer value.

LOG_FILE = 'scenario.log_file'

Should the log lines be written in a log file? File path string.

DEBUG_CLASSES = 'scenario.debug_classes'

Which debug classes to display? List of strings, or comma-separated string.

EXPECTED_ATTRIBUTES = 'scenario.expected_attributes'

Expected scenario attributes. List of strings, or comma-separated string.

CONTINUE_ON_ERROR = 'scenario.continue_on_error'

Should the scenario continue on error? Boolean value.

DELAY_BETWEEN_STEPS = 'scenario.delay_between_steps'

Should we wait between two step executions? Float value.

RUNNER_SCRIPT_PATH = 'scenario.runner_script_path'

Runner script path. Default is 'bin/run-test.py'.

SCENARIO_TIMEOUT = 'scenario.scenario_timeout'

Maximum time for a scenario execution. Useful when executing campaigns. Float value.

RESULTS_EXTRA_INFO = 'scenario.results_extra_info'

Scenario attributes to display for extra info when displaying scenario results, after a campaign execution, or when executing several tests in a single command line. List of strings, or comma-separated string.

ISSUE_LEVEL_NAMES = 'scenario.issue_levels'

Issue level names. Dictionary of names (`str`) => `int` values.

ISSUE_LEVEL_ERROR = 'scenario.issue_level_error'

Issue level from and above which known issues should be considered as errors.

ISSUE_LEVEL_IGNORED = 'scenario.issue_level_ignored'

Issue level from and under which known issues should be ignored.

__init__()

Initializes the timezone cache information.

__timezone

Timezone cache information.

timezone()

Gives the timezone configuration, if set.

Returns

Timezone configuration if set, `None` otherwise.

When not set, the local timezone is used.

invalidatetimezonecache()

Invalidates the timezone cache information.

logdatetimeenabled()

Determines whether the Log line should include a timestamp.

Configurable through *Key.LOG_DATETIME*.

logconsoleenabled()

Determines whether the log should be displayed in the console.

Configurable through *Key.LOG_CONSOLE*.

logoutpath()

Determines whether the log lines should be written in a log file.

Returns

Output log file path if set, `None` indicates no file logging.

Configurable through *Key.LOG_FILE*.

logcolorenabled()

Determines whether log colors should be used when displayed in the console.

Configurable through *Key.LOG_COLOR_ENABLED*.

logcolor(*level*, *default*)

Retrieves the expected log color for the given log level.

Parameters

- **level** – Log level name.

- **default** – Default log color.

Returns
Log color configured, or default if not set.

Configurable through *Key.LOG_COLOR*.

debugclasses()

Retrieves the debug classes configured.

Returns
List of debug classes.

Adds debug classes defined by the program arguments (see `args.Args.debug_classes`) plus those defined by the configurations (see *Key.DEBUG_CLASSES*).

expectedscenarioattributes()

Retrieves the user scenario expected attributes.

Configurable through *Key.EXPECTED_ATTRIBUTES*.

continueonerror()

Determines whether the test should continue when a step ends in error.

Configurable through *Key.CONTINUE_ON_ERROR*.

delaybetweensteps()

Retrieves the expected delay between steps.

Checks in configurations only (see *Key.DELAY_BETWEEN_STEPS*).

runnerscriptpath()

Gives the path of the scenario runner script path.

Useful when executing campaigns.

scenariotimeout()

Retrieves the maximum time for a scenario execution.

Useful when executing campaigns.

Checks in configurations only (see *Key.SCENARIO_TIMEOUT*).

resultsextrainfo()

Retrieves the list of scenario attributes to display for extra info when displaying test results.

Returns
List of scenario attribute names.

Applicable when displaying campaign results or the result of several tests executed in a single command line.

loadissuelevelnames()

Loads the issue level names configured through configuration files.

issuelevelerror()

Retrieves the issue level from and above which known issues should be considered as errors.

Returns
Error issue level if set, `None` otherwise.

issuelevelignored()

Retrieves the issue level from and under which known issues should be ignored.

Returns

Ignored issue level if set, None otherwise.

_readstringlistfromconf(*config_key*, *outlist*)

Reads a string list from the configuration database, and feeds an output list.

Parameters

- **config_key** – Configuration key for the string list.

  The configuration node pointed by config_key may be either a list of strings, or a comma-separated string.

- **outlist** – Output string list to feed.

  Values are prevented in case of duplicates.

_warning(*node*, *msg*)

Logs a warning message for the given configuration node.

Parameters

- **node** – Configuration node related to the warning.
- **msg** – Warning message.

ScenarioConfigKey

alias of *Key*

scenario.scenariodefinition module

Scenario definition.

class MetaScenarioDefinition

Bases: ABCMeta

Meta-class for *ScenarioDefinition*.

So that it can be a meta-class for *ScenarioDefinition*, *MetaScenarioDefinition* must inherit from abc.ABCMeta (which makes it inherit from type by the way) because the *StepUserApi* base class inherits from abc.ABC.

static __new__(*mcs*, *name*, *bases*, *attrs*, ***kwargs*)

Overloads class definition of *ScenarioDefinition* class and sub-classes.

Sets *MetaScenarioDefinition.InitWrapper* instances in place of __init__() methods, in order to have *ScenarioDefinition* initializers enclosed with *BuildingContext.pushscenariodefinition()* / *BuildingContext.popscenariodefinition()* calls.

Parameters

- **name** – New class name.
- **bases** – Base classes for the new class.
- **attrs** – New class attributes and methods.
- **kwargs** – Optional arguments.

class InitWrapper

Bases: object

Wrapper for __init__() methods of *ScenarioDefinition* instances.

Encloses the initializer's execution with *BuildingContext.pushscenariodefinition()* / *BuildingContext.popscenariodefinition()* calls, so that the building context of scenario stack knows about the scenario definition being built.

__init__(*init_method*)

Stores the original __init__() method.

Parameters

init_method – Original __init__() method.

init_method

Original __init__() method.

__get__(*obj*, *objtype=None*)

Wrapper descriptor: returns a __init__() bound method with obj.

Parameters

- **obj** – Optional instance reference.
- **objtype** – Unused.

Returns

Bound initializer callable (as long as obj is not None).

Inspired from: - https://docs.python.org/3/howto/descriptor.html - https://github.com/dabeaz/python-cookbook/blob/master/src/9/multiple_dispatch_with_function_annotations/example1.py

__call__(**args*, ***kwargs*)

__init__() wrapper call.

Parameters

- **args** – Positional arguments.

  First item should normally be the *ScenarioDefinition* instance the initializer is executed for.
- **kwargs** – Named arguments.

Pushes the scenario definition to the building context of the scenario stack before the initializer's execution, then removes it out after the initializer's execution.

class ScenarioDefinition

Bases: *StepUserApi*, *Assertions*, *Logger*

Base class for any final test scenario.

See the *quickstart guide*.

classmethod getinstance()

Expects and retrieves the current scenario definition with its appropriate type.

Returns

The current scenario definition instance, typed with the final user scenario definition class this method is called onto.

The "current" scenario is actually the one being executed or built.

Makes it possible to easily access the attributes and methods defined with a user scenario definition.

location

Definition location.

script_path

Script path.

name

Scenario name: i.e. script pretty path.

continue_on_error

Continue on error option.

Local configuration for the current scenario.

Prevails on *ScenarioConfig.Key.CONTINUE_ON_ERROR* (see *ScenarioRunner._shouldstop()*).

Not set by default.

__attributes

Scenario attributes (see *ScenarioConfig.expectedscenarioattributes()*).

__step_definitions

List of steps that define the scenario.

execution

Scenario execution, if any.

__repr__()

Canonical string representation.

__str__()

Human readable string representation of the scenario definition.

setattribute(*name*, *value*)

Defines an attribute for the scenario.

Parameters

- **name** – Attribute name.
- **value** – Attribute value.

Returns
`self`

getattribute(*name*)

Retrieves an attribute value defined with the scenario.

Parameters
name – Attribute name.

Returns
Attribute value.

Raises
KeyError – When the attribute name is not defined.

getattributenames()

Retrieves all attribute names defined with the scenario.

Returns
List of attribute names, sorted in alphabetical order.

section(*section_description*)

Adds a step section.

Parameters
section_description – Description for the section.

Returns
The section step just added.

addstep(*step_definition*)

Adds steps to the step list defining the scenario.

Parameters
step_definition – Step definition to add.

Returns
The step just added.

getstep(*step_specification=None*, *index=None*)

Finds a step definition.

Parameters

- **step_specification** – Step specification (see stepdefinition.StepSpecificationType), or None.
- **index** – Step index in the matching list. Last item when not specified.

Returns
Step definition found, if any.

expectstep(*step_specification=None*, *index=None*)

Expects a step definition.

When the step cannot be found, an exception is raised.

Parameters

- **step_specification** – Step specification (see stepdefinition.StepSpecificationType), or None.
- **index** – Step index in the matching list. Last item when not specified.

Returns
Expected step.

Raises
KeyError – When the step definition could not be found.

property steps

Step list.

class ScenarioDefinitionHelper

Bases: object

Scenario definition helper methods.

Avoids the public exposition of methods for internal implementation only.

static getscenariodefinitionclassfromscript(*script_path*)

Retrieves the scenario definitions classes from a Python script.

Parameters
script_path – Path of a Python script.

Returns
Scenario definition classes, if any.

__init__(*definition*)

Instanciates a helper for the given scenario definition.

Parameters
definition – Scenario definition instance this helper works for.

definition

Related scenario definition.

_logger

Make this class log as if it was part of the `ScenarioRunner` execution.

buildsteps()

Reads the scenario step list by inspecting the user scenario class, and feeds the scenario definition step list.

scenario.scenarioevents module

Scenario events.

class ScenarioEvent

Bases: *StrEnum*

Events described by the *scenario* framework.

BEFORE_TEST_CASE differs from *BEFORE_TEST* in that *BEFORE_TEST_CASE* is triggered within the context of a campaign execution while *BEFORE_TEST* is triggered within the context of a scenario execution.

The same for *AFTER_TEST_CASE* compared with *AFTER_TEST*.

BEFORE_CAMPAIGN = 'scenario.before-campaign'

Before campaign event: triggers handlers at the beginning of the campaign.

BEFORE_TEST_SUITE = 'scenario.before-test-suite'

Before test suite event: triggers handlers at the beginning of each test suite.

BEFORE_TEST_CASE = 'scenario.before-test-case'

Before test case event: triggers handlers at the beginning of each test case.

BEFORE_TEST = 'scenario.before-test'

Before test event: triggers handlers at the beginning of the scenario.

BEFORE_STEP = 'scenario.before-step'

Before step event: triggers handlers before each regular step.

ERROR = 'scenario.error'

Error event: triggers handlers on test errors.

AFTER_STEP = 'scenario.after-step'

After step event: triggers handlers after each regular step.

AFTER_TEST = 'scenario.after-test'

After test event: triggers handlers at the end of the scenario.

AFTER_TEST_CASE = 'scenario.after-test-case'

After test case event: triggers handlers after each test case.

AFTER_TEST_SUITE = 'scenario.after-test-suite'

After test suite event: triggers handlers after each test suite.

AFTER_CAMPAIGN = 'scenario.after-campaign'

After campaign event: triggers handlers after the campaign.

class ScenarioEventData

Bases: ABC

Container classes associated with *ScenarioEvent* events.

class Campaign

Bases: object

ScenarioEvent.BEFORE_CAMPAIGN and *ScenarioEvent.AFTER_CAMPAIGN* data container.

__init__(*campaign_execution*)

Parameters
campaign_execution – Campaign notified.

campaign

Campaign notified.

class TestSuite

Bases: object

ScenarioEvent.BEFORE_TEST_SUITE and *ScenarioEvent.AFTER_TEST_SUITE* data container.

__init__(*test_suite_execution*)

Parameters
test_suite_execution – Test suite notified.

test_suite

Test suite notified.

class TestCase

Bases: object

ScenarioEvent.BEFORE_TEST_CASE and *ScenarioEvent.AFTER_TEST_CASE* data container.

__init__(*test_case_execution*)

Parameters
test_case_execution – Test case notified.

test_case

Test case notified.

class Scenario

Bases: object

ScenarioEvent.BEFORE_TEST and *ScenarioEvent.AFTER_TEST* data container.

__init__(*scenario_definition*)

Parameters
scenario_definition – Scenario notified.

scenario

Scenario notified.

class Step

Bases: object

ScenarioEvent.BEFORE_STEP and *ScenarioEvent.AFTER_STEP* data container.

__init__(*step_definition*)

Parameters
step_definition – Step notified.

step

Step notified.

class Error

Bases: object

ScenarioEvent.ERROR data container.

__init__(*error*)

Parameters
error – Error notified.

error

Error notified.

_abc_impl = <_abc_data object>

scenario.scenarioexecution module

Scenario execution management.

class ScenarioExecution

Bases: object

Object that gathers execution information for a scenario.

The scenario execution information is not stored in the base *ScenarioDefinition* class for user scenario definitions. In order to avoid confusion, the dedicated members and methods are implemented in a separate class: *ScenarioExecution*.

__init__(*definition*)

Parameters
definition – Related scenario definition under execution. May be None when the *ScenarioExecution* instance is created as a data container only.

definition

Related scenario definition.

__current_step_definition

Current step reference in the scenario step list.

__next_step_definition

Next step reference in the step list. Used when a scenariodefinition.ScenarioDefinition.goto() call has been made.

time

Time statistics.

errors

Errors.

warnings

Warnings.

_logger

Make this class log as if it was part of the `ScenarioRunner` execution.

__repr__()

Canonical string representation.

startsteplist()

Initializes the step iterator.

Once the step iterator has been initialized, the *current_step_definition* attribute gives the current step. If the scenario has no step of the kind, *current_step_definition* is `None` as a result.

The *nextstep()* method then moves the iterator forward.

nextstep()

Moves the step iterator forward.

Returns
`True` when a next step is ready, `False` otherwise.

When *nextstep()* returns `False`, the *current_step_definition* is `None` as a result.

setnextstep(*step_definition*)

Arbitrary sets the next step for the step iterator.

Useful for the *goto feature*.

Parameters
step_definition – Next step definition to execute.

property current_step_definition

Current step definition being executed.

Depends on the current active step iterator (see `backtosteptype()`).

property status

Scenario execution status.

ExecutionStatus.FAIL when an exception is set, *ExecutionStatus.SUCCESS* otherwise.

Returns
Scenario execution status.

property step_stats

Step statistics computation.

Returns
Number of steps executed over the number of steps defined.

property action_stats

Action statistics computation.

Returns
Number of actions executed over the number of actions defined.

property result_stats

Expected result statistics computation.

Returns
Number of expected results executed over the number of expected results defined.

__cmp(*other*)

Scenario execution comparison in terms of result criticity.

Parameters
other – Other *ScenarioExecution* instance to compare with.

Returns

- -1 if `self` is less critical than `other`,
- 0 if `self` and `other` have the same criticity,
- 1 if `self` is more critical than `other`.

__lt__(*other*)

Checks whether `self` < `other`, i.e. `self` strictly less critical than `other`.

__le__(*other*)

Checks whether `self` <= `other`, i.e. `self` less critical than or as critical as``other``.

__gt__(*other*)

Checks whether `self` > `other`, i.e. `self` strictly more critical than `other`.

__ge__(*other*)

Checks whether `self` >= `other`, i.e. `self` more critical than or as critical as``other``.

scenario.scenariologging module

Scenario logging.

SCENARIO_LOGGING

Main instance of *ScenarioLogging*.

class ScenarioLogging

Bases: `object`

Scenario logging management.

ACTION_RESULT_MARGIN = 12

Actions, expected results and evidence lines are right-aligned with the longest 'EVIDENCE: ' pattern.

SCENARIO_STACK_INDENTATION_PATTERN = ' | '

The scenario stack indentation pattern ensures that the '|' lines are presented the 'ACTION: ' or 'RESULT: ' pattern they relate to.

class _Call

Bases: *StrEnum*

ScenarioLogging call identifiers.

BEGIN_SCENARIO = 'beginscenario'

BEGIN_ATTRIBUTES = 'beginattributes'

ATTRIBUTE = 'attribute'

END_ATTRIBUTES = 'endattributes'

STEP_DESCRIPTION = 'stepdescription'

ACTION = 'action'

RESULT = 'result'

END_SCENARIO = 'endscenario'

__init__()

Initializes the last call history.

_calls

History of this class's method calls.

Makes it possible to adjust the display depending on the sequence of information.

_known_issues

Known issues already displayed.

beginscenario(*scenario_definition*)

Displays the beginning of a scenario execution.

Parameters
scenario_definition – Scenario being executed.

beginattributes()

Marks the beginning of scenario attributes.

attribute(*name*, *value*)

Display the value of a scenario attribute.

Parameters

- **name** – Scenario attribute name.
- **value** – Scenario attribute value.

See also:

ScenarioConfig.expectedscenarioattributes()

endattributes()

Marks the end of scenario attributes, and the beginning of the test steps by the way.

stepsection(*step_section*)

Displays a step section.

Parameters
step_section –

Returns

stepdescription(*step_definition*)

Displays a step being executed.

Parameters
step_definition – Step definition being executed.

actionresult(*actionresult*, *description*)

Displays an action or an expected result being executed.

Parameters

- **actionresult** – Action or expected result being executed.
- **description** – Action/result description.

error(*error*)

Displays the test exception.

Parameters
error – Error to display.

evidence(*evidence*)

Displays an evidence.

Evidence being saved with the test results shall also be printed out in the console.

Parameters
evidence – Evidence text.

endscenario(*scenario_definition*)

Displays the end of a scenario execution.

Parameters
scenario_definition – Scenario which execution has just finished.

Resets the *_known_issues* history for the main scenario.

displaystatistics(*scenario_execution*)

Displays the scenario statistics.

Parameters
scenario_execution – Scenario which execution has just finished.

scenario.scenarioreport module

Statistics class module.

SCENARIO_REPORT

Main instance of *ScenarioReport*.

class ScenarioReport

Bases: *Logger*

JSON report generator.

__init__()

Configures logging for the *ScenarioReport* class.

_json_path

JSON report path being written or read.

writejsonreport(*scenario_definition*, *json_path*)

Generates the JSON report output file for the given scenario execution.

Parameters

- **scenario_definition** – Scenario to generate the JSON report for.

- **json_path** – Path to write the JSON report into.

Returns
`True` for success, `False` otherwise.

readjsonreport(*json_path*)

Reads the JSON report file.

Parameters
json_path – JSON file path to read.

Returns
Scenario data read from the JSON report file. `None` when the file could not be read, or its content could not be parsed successfully.

_scenario2json(*scenario_definition*, *is_main*)

Scenario report JSON generation.

Parameters

- **scenario_definition** – Scenario to generate the JSON report for.
- **is_main** – True for the main scenario, False otherwise.

Returns
JSON report object.

_json2scenario(*json_scenario*)

Scenario data reading from JSON report.

Parameters
json_scenario – Scenario JSON report to read.

Returns
Scenario data.

_step2json(*step_definition*)

Generates the JSON report for a step.

Parameters
step_definition – Step definition (with execution) to generate the JSON report for.

Returns
JSON report object.

_json2step(*json_step_definition*)

Step reading from JSON report.

Parameters
json_step_definition – Step definition JSON report to read.

Returns
StepDefinition data.

_actionresult2json(*action_result_definition*)

Generates the JSON report for an action / expected result.

Parameters
action_result_definition – Action or expected result to generate the JSON report for.

Return JSON
JSON report object.

_json2actionresult(*json_action_result_definition*)

Action / expected result reading from JSON report.

Parameters
json_action_result_definition – Action / expected result JSON report to read.

Returns
ActionResultDefinition data.

scenario.scenarioresults module

Scenario results management.

SCENARIO_RESULTS

Main instance of *ScenarioResults*.

class ScenarioResults

Bases: *Logger*

List of scenario execution results.

__init__()

Initializes an empty list.

_results

List of `ScenarioResult` instances.

add(*scenario_execution*)

Adds a `ScenarioResult` instance in the list.

Parameters
scenario_execution – Scenario execution instance.

property count

Returns
Number of scenario execution results in the list.

display()

Displays the results of the scenario executions in the list.

Designed to display convient information after *ScenarioLogging* and *CampaignLogging* outputs.

classmethod _displayscenarioline(*log_level*, *fmt*, *scenario_execution*)

Displays a scenario line.

Parameters

- **log_level** – Log level to use.
- **fmt** – Format to use.
- **scenario_execution** – Scenario to display.

classmethod _displayerror(*log_level*, *error*)

Displays a test error.

Parameters

- **log_level** – Log level to use.
- **error** – Test error to display.

scenario.scenariorunner module

Scenario execution management.

SCENARIO_RUNNER

Main instance of *ScenarioRunner*.

class ScenarioRunner

Bases: *Logger*

Test execution engine: runs scenarios, i.e. instances derived from the *ScenarioDefinition* class.

Only one instance, accessible through the *SCENARIO_RUNNER* singleton.

Implements the *main()* function for scenario executions.

This class works with the following helper classes, with their respected purpose:

- *ScenarioArgs*: command line arguments,
- *ScenarioExecution*: object that describes a scenario execution,
- *ScenarioStack*: stack of *ScenarioExecution*,
- *ScenarioLogging*: scenario execution logging,
- *ScenarioReport*: scenario report generation.

class ExecutionMode

Bases: *StrEnum*

Execution mode enum.

Tells whether the scenario runner is currently:

- building the objects,
- generating the documentation,
- or executing the test.

BUILD_OBJECTS = 'build'

The scenario runner is currently building the test objects.

DOC_ONLY = 'doc-only'

The scenario runner is currently generating the test documentation.

EXECUTE = 'execute'

The scenario runner is currently executing the final test script.

__init__()

Sets up logging for the *ScenarioRunner* class, and member variables.

main()

Scenario runner main function, as a member method.

Returns

Error code.

property _execution_mode

Current execution mode.

Depends on 1) the scenario stack building context, and 2) the scenario args –doc-only option.

executepath(*scenario_path*)

Executes a scenario from its script path.

Parameters
scenario_path – Scenario Python script path.

Returns
Error code, but no `errcodes.ErrorCode.TEST_ERROR`.

Feeds the *SCENARIO_RESULTS* instance.

executescenario(*scenario_definition*, *start_time=None*)

Executes a scenario or subscenario.

Parameters

- **scenario_definition** – Scenario to execute.
- **start_time** – Optional starting time specification.

  May be set in order to save the most accurate info on the starting time of the scenario.

Returns
Error code, but no `errcodes.ErrorCode.TEST_ERROR`.

_buildscenario(*scenario_definition*)

Builds a scenario definition.

Parameters
scenario_definition – *ScenarioDefinition* instance to populate with steps, actions and expected results definitions.

Returns
Error code.

_beginscenario(*scenario_definition*)

Begins a scenario or sub-scenario execution.

Parameters
scenario_definition – Scenario or subscenario which execution to start.

Returns
Error code.

_endscenario(*scenario_definition*)

Ends a scenario or subscenario execution.

Parameters
scenario_definition – Scenario or subscenario which execution to end.

Returns
Error code.

_execstep(*step_definition*)

Executes the step.

Parameters
step_definition – Step definition to execute.

onstepdescription(*description*)

Call redirection from `scenariodefinition.ScenarioDefinition.STEP()`.

Parameters
description – Step description.

_notifyknownissuedefinitions(*step_user_api*, *known_issues=None*)

Notifies the known issues declared at the definition level for the given scenario or step definition.

Parameters

- **step_user_api** – Scenario or step definition to process known issues for.
- **known_issues** – Specific known issue list to process. Defaults to *StepUserApi.known_issues* when not set.

onactionresult(*action_result_type*, *description*)

Call redirection from `scenariodefinition.ScenarioDefinition.ACTION()` or `scenariodefinition.ScenarioDefinition.RESULT()`.

Parameters

- **action_result_type** – ACTION or RESULT.
- **description** – Action or expected result description.

_endcurrentactionresult()

Ends the current action or expected result section.

onevidence(*evidence*)

Call redirection from `scenariodefinition.ScenarioDefinition.EVIDENCE()`.

Parameters
evidence – Evidence text.

doexecute()

Tells whether the test script shall be executed.

Returns
`True` when the test script shall be executed.

onerror(*error*, *originator=None*)

Called when an error occurs.

Parameters

- **error** – Error that occurred.
- **originator** – Scenario or step definition that made the call to *onerror()*, set in :meth:.stepuserapi.StepUserApi.knownissue()`..

_shouldstop()

Tells whether the scenario execution should stop.

Returns
`True` when the scenario execution should stop, `False` when the scenario execution should continue on.

goto(*to_step_specification*)

Call redirection from `scenariodefinition.ScenarioDefinition.goto()`.

Parameters
to_step_specification – Specification of the next step to execute.

exception GotoException

Bases: Exception

Breaks execution in a step method when scenariodefinition.ScenarioDefinition.goto() is called.

scenario.scenariostack module

Scenario execution stack.

SCENARIO_STACK

Main instance of *ScenarioStack*.

class BuildingContext

Bases: object

Storage of instances under construction.

__init__()

Declares references of objects under construction.

Objects under construction:

- scenario definition,
- but no step definition!

> **Warning:** We do not store a step definition reference here, fed from *StepDefinition.__init__()* especially, for the reason that we cannot guarantee that the reference of a volatile step definition would not take the place of a real step definition being built.
>
> See *step_definition* and *fromoriginator()* for further details on the step definition reference management.

__scenario_definitions

Scenario definitions being built.

pushscenariodefinition(*scenario_definition*)

Pushes the reference of the scenario definition being built to the building context of the scenario stack.

Parameters
scenario_definition – Scenario definition being built.

popscenariodefinition(*scenario_definition*)

Pops the reference of the scenario definition being built from the building context of the scenario stack.

Parameters
scenario_definition – Scenario definition being built.

property scenario_definition

Main scenario definition being built (i.e. the first), if any.

property step_definition

Step definition being built, if any.

The step definition being built is the one currently executed in scenariorunner.ScenarioRunner.ExecutionMode.BUILD_OBJECTS execution mode by *ScenarioRunner._buildscenario()*.

> **Warning:** This does not cover the case of method calls directly made in a step object initializer, and especially for:
>
> - *StepUserApi.knownissue()*
>
> In order to cover such cases, theses methods shall set an `originator` parameter when calling *ScenarioRunner* methods, to let the latter identify the appropriate instance being built with the help of the *fromoriginator()* method.

fromoriginator(*originator*)

Determines the actual object being built.

Parameters
originator – *StepUserApi* instance that made a call.

Returns
StepUserApi actually being built.

Fixes the `originator` reference from the current scenario definition being built to the current step definition being built if any.

Lets the `originator` reference as is otherwise:

- either a *StepDefinition* reference directly,
- or *ScenarioDefinition* reference.

class ScenarioStack

Bases: *Logger*

Scenario execution stack management.

This class acts as a helper for the *ScenarioRunner* class.

It also determines the *scenario stack logging indentation*.

Note: The fact that the *ScenarioStack* class is a helper for the *ScenarioRunner* one explains the definition of the additional methods and properties:

- `checkcurrentscenario()`,
- `current_step`,
- `current_action_result`.

By the way, this makes this class being a bit more than just a *scenario stack manager*, but rather a *scenario execution context manager*.

Whatever, the name of this class is convenient as is, even though it is labelled as a "stack" only.

---

exception ContextError

Bases: `Exception`

Notifies a scenario stack error.

__init__()

Defines the error message.

__init__()

Initializes an empty scenario execution stack.

building

Instances under construction.

__scenario_executions

Scenario execution stack.

The first item defines the `main_scenario`. The subscenarios (if any) then follow.

history

History of scenario executions.

Main scenario executions only. In the chronological order.

pushscenarioexecution(*scenario_execution*)

Adds a scenario execution instance in the scenario execution stack.

Parameters
scenario_execution – Scenario execution to set on top of the scenario execution stack.

popscenarioexecution()

Removes the last scenario execution from the scenario execution stack.

Returns
Scenario execution removed.

property size

Returns the size of the scenario execution stack.

Returns
Number of scenario executions currently stacked.

property main_scenario_definition

Main scenario definition under execution.

Returns the reference of the top scenario definition under execution, whether subscenarios are being executed or not.

Almost equivalent to *main_scenario_execution*, but retrieves the scenario definition instance.

property main_scenario_execution

Main scenario execution instance.

Returns the reference of the top scenario execution instance, whether subscenarios are being executed or not.

Almost equivalent to *main_scenario_definition*, but retrieves the scenario execution instance.

ismainscenario(*scenario*)

Tells whether the given scenario corresponds to the main one under execution.

Parameters
scenario – Scenario definition or scenario execution to check.

Returns
`True` if the scenario corresponds to the main scenario, `False` otherwise.

property current_scenario_definition

Current scenario definition under execution.

The latest unterminated subscenario if any, i.e. the main scenario if no current sub-scenario.

Almost equivalent to *current_scenario_execution*, but retrieves the scenario definition instance.

property current_scenario_execution

Current scenario execution instance.

The latest unterminated subscenario if any, i.e. the main scenario if no current sub-scenario.

Almost equivalent to *current_scenario_definition*, but retrieves the scenario execution instance.

iscurrentscenario(*scenario*)

Tells whether the given scenario corresponds to the one on top of the scenario stack.

Parameters
scenario – Scenario definition or scenario execution to check.

Returns
`True` if the scenario corresponds to the main scenario, `False` otherwise.

property current_step_definition

Current step definition under execution.

Out of the current scenario.

Compared with *current_step_execution*, this method returns the step definition whatever the execution mode of the *ScenarioRunner*.

`None` if no current step definition under execution.

property current_step_execution

Current step execution instance.

Out of the `current_step`.

Compared with *current_step_definition*, this method may not return a step execution instance when the *ScenarioRunner* is building objects.

`None` if no current step execution instance.

property current_action_result_definition

Current action or expected result definition under execution.

Out of the current step definition or step execution.

`None` current action / expected result definition.

property current_action_result_execution

Current action or expected result execution instance.

Out of the current step execution.

`None` if no current action / expected result execution instance.

knownissue(*__id*, *__message*)

knownissue(*message*, *level=None*, *id=None*)

Registers a known issue in the current context.

raisecontexterror(*error_message*)

Raises an error about the scenario stack execution.

Displays error information about the current status of the stack for investigation purpose.

Parameters
error_message – Error message.

Raises
ScenarioStack.ContextError – Systematically.

scenario.stats module

Common statistics.

class TimeStats

Bases: object

Common time statistics.

__init__()

Initializes the time statistics with None values.

_start

Start time, if specified.

_elapsed

Elapsed time, if specified.

_end

End time, if specified.

__str__()

Computes a string representation of the time interval in the '[%s - %s]' form.

Returns
String representation of the time interval.

property start

Start time getter.

property elapsed

Elapsed time getter.

property end

End time getter.

setstarttime()

Starts the time statistics with the current time.

setendtime()

Ends the time statistics with the current time.

tojson()

Converts the *TimeStats* instance into a JSON dictionary.

Returns
JSON dictionary, with optional 'start', 'end' and 'elapsed' float fields, when the values are set.

static fromjson(*json_data*)

Builds a *TimeStats* instance from its JSON representation.

Parameters
json_data – JSON dictionary, with optional 'start', 'end' and 'elapsed' float fields.

Returns
New *TimeStats* instance.

class ExecTotalStats

Bases: object

Executable item statistics: number of executed items over a total count.

__init__()

Initializes the count statistics with 0.

total

Total count of executable items.

executed

Count of items executed.

__str__()

Computes a '%d' or '%d/%d' string representation of the statistics, depending on the args.Args.doc_only parameter.

Returns
String representation of the statistics.

add(*stats*)

Integrates a tier *ExecTotalStats* instance into this one.

Parameters
stats – Tier *ExecTotalStats* instance.

Returns
Self (named parameter idiom).

Increments both *executed* and *total* counts with the tier's values.

tojson()

Converts the *ExecTotalStats* instance into a JSON dictionary.

Returns
JSON dictionary, with 'executed' and 'total' int fields.

static fromjson(*json_data*)

Builds a *ExecTotalStats* instance from its JSON representation.

Parameters
json_data – JSON dictionary, with 'executed' and 'total' int fields.

Returns
New *ExecTotalStats* instance.

scenario.stepdefinition module

Step definition.

class StepDefinition

Bases: *StepUserApi*, *Assertions*, *Logger*

Step definition management.

classmethod getinstance(*index=None*)

Expects and retrieves a step with its appropriate type.

Parameters
index – Optional step index of the kind. Optional. See *ScenarioDefinition.getstep()* for more details.

Returns
The expected step for the current scenario, typed with the final user step definition class this method is called onto.

The "current" scenario is actually the one being executed or built.

Makes it possible to easily access the attributes and methods defined with a user step definition.

__init__(*method=None*)

Parameters
method – Method that defines the step, when applicable. Optional.

scenario

Owner scenario.

Initially set with a void reference. Fixed when `scenariodefinition.ScenarioDefinition.addsteps()` is called.

method

Step method, if any.

location

Definition location.

description

Step description.

__action_result_definitions

List of actions and expected results that define the step.

executions

Step executions.

__repr__()

Canonical string representation.

__str__()

Returns a human readable representation of the step definition.

property name

Step name, i.e. the fully qualified name of the class or method defining it.

property number

Step definition number.

Number of this step definition within the steps defining the related scenario. Starting from 1, as displayed to the user.

addactionresult(**action_result_definitions*)

Adds actions / expected results to the list defining the step.

Parameters
action_result_definitions – Action / expected result definitions to add.

Returns
`self`

property actions_results

Action / expected result list.

getactionresult(*index*)

Retrieves an *ActionResultDefinition* instance from its location.

Parameters
index – Action/result definition index.

Returns
Action/result definition instance.

step()

Calls *method*, when not overloaded.

This method should be overloaded by user step definition classes.

Otherwise, this base implementation of this method expects the *method* attribute to be set, and invokes it.

_abc_impl = <_abc_data object>

class StepDefinitionHelper

Bases: object

Step definition helper methods.

Avoids the public exposition of methods for internal implementation only.

__init__(*definition*)

Instanciates a helper for the given step definition.

Parameters
definition – Step definition instance this helper works for.

matchspecification(*step_specification*)

Determines whether the given step specification matches the related step definition.

Parameters
step_specification – Step specification to check.

Returns
True when the specification matches the related step definition.

static specificationdescription(*step_specification*)

Returns a human readable representation of the step specification.

Parameters
step_specification – Step specification to compute a string representation for.

Returns
String representation.

saveinitknownissues()

Saves *init* known issues for the related step definition.

I.e. the known issues declared at the definition level, before the :meth:StepDefinition.step()` method has been called.

The appropriate call to this method is made in *ScenarioRunner._buildscenario()*.

`getinitknownissues()`

Retrieves the known issue list saved by `stashinitknownissues()` for the related step definition.

Returns
Init known issue list.

`class StepMethods`

Bases: `object`

Collection of static methods to help manipulating methods.

`static _hierarchycount(logger, method)`

Returns the number of classes in class hierarchy that have this method being declared.

Parameters

- **logger** – Logger to use for debugging.
- **method** – Method to look for accessibility in class hierarchy.

Returns
Count. The higher, the upper class the method is defined into.

Used by the *sortbyhierarchythennames()* and *sortbyreversehierarchythennames()* methods.

`static _dispmethodlist(methods)`

Computes a debug representation of a method list.

Parameters
methods – Array of methods to debug.

Returns
Debug representation.

`static sortbynames(logger, methods)`

Sorts an array of methods by method names.

Parameters

- **logger** – Logger to use for debugging.
- **methods** – Array of methods to sort.

`static sortbyhierarchythennames(logger, methods)`

Sorts an array of methods by hierarchy at first, then by method names.

Parameters

- **logger** – Logger to use for debugging.
- **methods** – Array of methods to sort.

Makes the methods defined in the higher classes be executed prior to those defined in the lower classes, i.e. makes the most specific methods be executed at last.

Formerly used by *before-test* and *before-step* steps.

`static sortbyreversehierarchythennames(logger, methods)`

Sorts an array of methods by reverse hierarchy first, then by method names.

Parameters

- **logger** – Logger to use for debugging.
- **methods** – Array of methods to sort.

Makes the methods defined in the lower classes be executed prior to those defined in the upper classes, i.e. makes the most specific methods be executed at first.

Formerly used by *after-test* and *after-step* steps.

scenario.stepexecution module

Step execution management.

class StepExecution

Bases: object

Step execution information.

Note: Due to the **goto* feature*, a step may be executed several times. By the way, a *StepDefinition* instance may own multiple instances of *StepExecution*.

__init__(*definition*, *number*)

Initializes a new step execution for the given step definition.

Starts the execution time with the current date.

Parameters

- **definition** – Step definition this instance describes an execution for.
- **number** – Execution number. See *number*.

definition

Owner step reference.

number

Step execution number.

Execution number of this step execution within the steps executions of the related scenario. Starting from 1, as displayed to the user.

current_action_result_definition

Current action or expected result under execution.

Differs from *__current_action_result_definition_index* in that this reference can be set to None when the action / expected result execution is done.

time

Time statistics.

errors

Error.

warnings

Warnings.

__current_action_result_definition_index

Current action or expected result index under execution.

__repr__()

Canonical string representation.

getnextactionresultdefinition()

Retrieves the next action/result definition to execute.

Returns
Next *ActionResultDefinition* instance to execute.

Sets the *current_action_result_definition* reference by the way.

getstarttime()

Retrieves the starting time of the step execution.

Returns
Step execution start time.

getendtime(*expect*)

Retrieves the ending time of the step execution.

Parameters
expect – True when this step execution is expected to be terminated. Otherwise, the current time is returned.

Returns
Step execution end time, or current time.

static actionstats(*definition*)

Computes action statistics for the given step definition.

Parameters
definition – Step definition to compute action statistics for.

Returns
Action statistics of the step.

static resultstats(*definition*)

Computes expected result statistics for the given step definition.

Parameters
definition – Step definition to compute expected result statistics for.

Returns
Expected result statistics of the step.

scenario.stepsection module

Step section management.

class StepSection

Bases: *StepDefinition*

Step section definition.

Overloads *StepDefinition* but does not act as a regular step.

ScenarioRunner actually recognizes *StepSection* instances and skips their execution.

ScenarioReport also recognizes *StepSection* instances, and therefore does not generate 'executions' nor 'actions-results' sections for them.

__init__(*description*)

Parameters
description – Step section description.

description

Step section description.

step()

Calls `method`, when not overloaded.

This method should be overloaded by user step definition classes.

Otherwise, this base implementation of this method expects the `method` attribute to be set, and invokes it.

_abc_impl = <_abc_data object>

scenario.stepuserapi module

User API methods for user *ScenarioDefinition* or *StepDefinition* overloads.

class StepUserApi

Bases: `ABC`

Base class that defines the methods made available for user *ScenarioDefinition* or *StepDefinition* overloads.

__init__()

Initializes an empty known issue list.

known_issues

Known issues at the definition level.

STEP(*description*)

Defines the short description of a step.

Parameters
description – Step description.

Note: We deliberately deviate from PEP8 namings in order to highlight *STEP()* calls in the final test code.

ACTION(*action*)

Describes a test action.

Parameters
action – Action description.

Returns
`True` when the test script shall be executed, `False` otherwise (documentation generation).

Note: We deliberately deviate from PEP8 namings in order to highlight *ACTION()* calls in the final test code.

RESULT(*result*)

Describes an expected result.

Parameters
result – Expected result description.

Returns
`True` when the test script shall be executed, `False` otherwise (documentation generation).

Note: We deliberately deviate from PEP8 namings in order to highlight *RESULT()* calls in the final test code.

---

doexecute()

Tells whether test script should be executed.

Returns
`True` for test execution, `False` for documentation generation only, exactly the same as the *ACTION()* and *RESULT()* methods do, but without generating any texts.

evidence(*evidence*)

Saves an evidence for the current action or expected result.

Parameters
evidence – Evidence text.

goto(*to_step_specification*)

Makes the execution jump to the given step.

Parameters
to_step_specification – Step specification of the step to jump to (see `stepdefinition.StepSpecificationType`).

_abc_impl = <_abc_data object>

knownissue(*__id*, *__message*)

knownissue(*message*, *level=None*, *id=None*)

General implementation for related overloads.

scenario.subprocess module

SubProcess class definition.

class SubProcess

Bases: `object`

Sub-process execution.

__init__(**args*)

Parameters
args – Command line arguments. May be the first arguments only, then rely on the *addargs()* method to add others.

cmd_line

Sub-process command line arguments.

See *addargs()*.

env
: See *setenv()*.

cwd
: See *setcwd()*.

_logger
: See *setlogger()*.

_stdout_line_handler
: Handler to call on each stdout line.

_stderr_line_handler
: Handler to call on each stderr line.

_exit_on_error_code
: See *exitonerror()*.

returncode
: Sub-process return code.

stdout
: Standard output as a string.

stderr
: Standard error as a string.

time
: Time statistics.

_popen
: `subprocess.Popen` instance.

_async
: Tells whether the *run()* method should wait for the end of the sub-process.

_stdout_reader
: Stdout reader thread routine.

_stderr_reader
: Stderr reader thread routine.

__repr__()
: Canonical string representation.

__str__()
: Human readable string representation.

tostring()
: Human readable full string representation.

addargs(**args*)
: Extra arguments addition.

  Parameters
  args – Extra arguments.

  Returns
  `self`

hasargs(**args*)

Determines whether the command line contains the given sequence of consecutive arguments.

Parameters
args – Sequence of arguments being searched.

Returns
`True` when the arguments have been found, `False` otherwise.

setenv(***kwargs*)

Sets extra environment variables.

Parameters
kwargs – Extra environment variables.

Returns
`self`

setcwd(*cwd*)

Sets the current working directory.

Parameters
cwd – Current working directory.

Returns
`self`

setlogger(*logger*)

Directs log lines to the given logger instance.

Parameters
logger – Logger instance to use.

Returns
`` self``

onstdoutline(*handler*)

Installs a handler to be called on each stdout line.

Parameters
handler – Handler to call on each stdout line.

Returns
`self`

onstderrline(*handler*)

Installs a handler to be called on each stderr line.

Parameters
handler – Handler to call on each stderr line.

Returns
`self`

exitonerror(*exit_on_error_code*)

Tells whether the main program should stop (`sys.exit()`) in case of an error.

Parameters
exit_on_error_code – Set to `None` to keep executing in case of an error (default behaviour). Set to a *ErrorCode* value to make the main program stop with the given error code. `True` is an equivalent for `errcodes.ErrorCode.INTERNAL_ERROR`, `False` is an equivalent for `None`.

Returns
`self`

The return code is available through the *returncode* attribute.

run(*timeout=None*)

Sub-process execution.

Parameters
timeout – Waiting timeout, in seconds.``None`` to wait infinitely.

Returns
`self`

The sub-process return code is available through the *returncode* attribute.

runasync()

Launches the sub-process asynchronously.

Contrary to *run()*, this method launches the sub-process, then returns without waiting for the end of it.

_readstdoutthread()

Stdout reader thread routine.

_readstderrthread()

Stderr reader thread routine.

isrunning()

Tells whether the sub-process is currently running.

Returns
`True` when the sub-process is still running. `False` otherwise.

wait(*timeout=None*)

Waits for the sub-process to terminate.

Parameters
timeout – Waiting timeout, in seconds. `None` to wait infinitely.

Returns
`self`

Raises
TimeoutError – When the sub-process did not terminate within `timeout` seconds.

kill()

Kills the sub-process.

Returns
`self`

_onerror(*error_message*, **args*)

Error management.

Optionally logs the error and terminates the main process.

Parameters

- **error_message** – Error message.
- **args** – Error message arguments.

`_log`(*level*, *message*, **args*)

Pushes a log line to the attached logger, if any.

Parameters

- **level** – Log level.
- **message** – Log message.
- **args** – Format arguments.

scenario.testerrors module

Test errors.

exception `TestError`(*message*, *location=None*)

Bases: `Exception`

Base test error object.

Stores information about an error that occurred during the test.

Declared as an exception so that it can be propagated as is.

`__init__`(*message*, *location=None*)

Parameters

- **message** – Error message.
- **location** – Error location.

`message`

Error message.

`location`

Error location.

`__str__`()

Short representation of the error.

`__repr__`()

Programmatic representation of the error.

`iserror`()

Tells whether this error object is actually an error.

Returns

`True` for a real error, `False` for a simple warning (see *iswarning()*) or when the error should be ignored (see *isignored()*).

`iswarning`()

Tells whether this error object is just a warning.

Returns

`True` for a simple warning, `False` for a real error (see *iserror()*) or when the error should be ignored (see *isignored()*).

isignored()

Tells whether this error object should be ignored.

Returns
`True` when the error should be ignored, `False` for a real error (see *iserror()*) or a warning (see *iswarning()*).

logerror(*logger*, *level=40*, *indent=''*)

Logs the error info.

Parameters

- **logger** – Logger to use for logging.
- **level** – Log level.
- **indent** – Indentation to use.

tojson()

Converts the *TestError* instance into a JSON dictionary.

Returns
JSON dictionary.

static fromjson(*json_data*)

Builds a *TestError* instance from its JSON representation.

Parameters
json_data – JSON dictionary.

Returns
New *TestError* instance.

exception ExceptionError(*exception=None*)

Bases: *TestError*

Test error related to an exception.

__init__(*exception=None*)

Parameters
exception – Root cause exception, if available.

exception

The root cause exception, if any.

exception_type

Type of the exception, if any.

__str__()

Short representation of the exception error.

Exception type + message.

logerror(*logger*, *level=40*, *indent=''*)

TestError.logerror() override in order to log the exception traceback, if an exception is stored.

Defaults to *TestError.logerror()* if no exception.

tojson()

Converts the *TestError* instance into a JSON dictionary.

Returns
JSON dictionary.

`static fromjson(json_data)`

Builds a *ExceptionError* instance from its JSON representation.

Parameters
json_data – JSON dictionary.

Returns
New *ExceptionError* instance.

scenario.testsuitefile module

Test suite file management.

`class TestSuiteFile`

Bases: *Logger*

Test suite file reader.

`__init__(path)`

Initializes a test suite file reader from its path.

Parameters
path – Test suite file path.

`path`

Test suite file path.

`script_paths`

Script paths describes by the test suite file.

Filled once the test suite file has been successfully read.

See also:

read().

`__repr__()`

Canonical string representation.

`read()`

Reads and parses the test suite file.

Returns
`True` for success, `False` otherwise.

scenario.textfile module

Text file management.

`class TextFile`

Bases: `object`

Wrapper for reading and writing text files.

This class doesn't aim to be a faithful IO class, but rather a wrapper to common file operations.

`__init__`(*path*, *mode*, *encoding=None*)

Parameters

- **path** – File path.
- **mode** – “r” or “w”.
- **encoding** – Explicit file encoding when specified. Will be guessed from the input file when reading otherwise. UTF-8 by default.

`_file`

Python file instance.

`encoding`

File encoding.

`_guessencoding`()

Tries to guess the file encoding from a dedicated comment in the first lines of it.

Stores the result in the *encoding* attribute.

`read`(*size=-1*)

Reads a string from the file.

Parameters
size – Size to read.

Returns
Content string.

`readlines`()

Reads all lines from a file.

Returns
File lines.

`write`(*text*)

Writes a string to the file.

Parameters
text – String to write.

Returns
Number of bytes written. May not equal the string length depending on the encoding.

`close`()

Closes the file.

`guessencoding`(*path*)

Return the encoding guessed from a text file.

Parameters
path – Path of the file to guess encoding for.

Returns
Encoding guessed from the file. UTF-8 by default.

See also:

TextFile._guessencoding() and *TextFile.encoding*.

scenario.timezoneutils module

Timezone handling.

UTC

UTC timezone constant.

local(*ref_timestamp*)

Returns the local timezone for the given timestamp.

Parameters
ref_timestamp – Reference timestamp to compute the local timezone for.

Returns
Local timezone (fixed time offset, non DST-aware).

> **Warning:** The timezone returned is a fixed time offset, i.e. it does not handle DST (Daylight Saving Time) shifts.
>
> That's the reason why the `ref_timestamp` must be set appropriately in order to avoid errors between summer and winter times.

fromstr(*tz_desc*)

Computes a timezone information from an optional string.

Parameters
tz_desc – Optional timezone description.

Returns
Timezone information when `tz_desc` is set, `None` otherwise.

scenario.typing module

Type definitions to help type hints.

scenario.xmlutils module

XML DOM utils.

class Xml

Bases: `object`

Because:

1. there seems to be no obvious portable choice for parsing and writing XML in Python (see https://stackoverflow.com/questions/1912434/how-to-parse-xml-and-count-instances-of-a-particular-node-attribute),
2. standard libraries such as *xml.dom.minidom* are sometimes untyped or partially untyped,

let's define a wrapper that gives us the opportunity to abstract the final library used and work around typing issues.

class Document

Bases: object

XML document.

__init__()

Instanciates an XML document, either for reading or writing.

_xml_doc

Underlying library document reference.

property root

Returns
Retrieves the root node of the document.

static read(*path*)

Reads from an XML file.

Parameters
path – File to read from.

Returns
XML document read from the file.

write(*path*)

Writes the document into a file.

Parameters
path – File to write to.

createnode(*tag_name*)

Create a node with the given tag name.

Parameters
tag_name – Tag name.

Returns
New node.

createtextnode(*text*)

Create a text node.

Parameters
text – Initial text for the new node.

Returns
New text node.

class INode

Bases: ABC

Abstract interface for regular nodes and text nodes.

_abc_impl = <_abc_data object>

class Node

Bases: *INode*

Regular XML node.

__init__(*xml_element*)

Parameters
xml_element – Underlying library node reference.

_xml_element

Underlying library node reference.

property tag_name

Returns
Tag name of the node.

hasattr(*name*)

Tells whether the node has an attribute of the given name.

Parameters
name – Attribute name.

Returns
`True` when the node has an attribute of the given name, `False` otherwise.

getattr(*name*)

Retrives the attribute value of the given name.

Parameters
name – Attribute name.

Returns
Attribute value, or possibly an empty string if the attribute does not exist.

setattr(*name*, *value*)

Set an attribute.

Parameters
- **name** – Attribute name.
- **value** – Attribute value.

Returns
`self`

getchildren(*tag_name*)

Retrieves direct children with the given tag name.

Parameters
tag_name – Children tag name.

Returns
List of children nodes.

gettextnodes()

Retrieves direct children text nodes.

Returns
List of children text nodes.

appendchild(*child*)

Adds a child to the node.

Parameters
child – New node or text node to set as a child.

Returns
The child just added.

_abc_impl = <_abc_data object>

class TextNode

Bases: *INode*

Text node.

__init__(*xml_text*)

Parameters
xml_text – Underlying library text node reference.

_xml_text

Underlying library text node reference.

property data

Text content.

append(*data*)

Adds some text to the text node.
Parameters
data – Additional text.

_abc_impl = <_abc_data object>

5.3 Coding rules

A few coding rules are defined for the project.

5.3.1 Files

File permissions

File permissions are stored apropriately in the git database, so that:

- regular files remain with 644 permissions,
- executable scripts and binaries get the 755 permissions.

As long as the `chmod` command is available in the development environment, the `tools/checkrepo.py` script checks this rule over the files of the git repository.

Encodings

Encoding is utf-8 for all files.

The encoding is explicitely specified in the first lines of Python scripts through:

```
# -*- coding: utf-8 -*-
```

The `tools/checkrepo.py` script checks this rule over the files of the git repository.

5.3.2 Git

Branching strategy

Todo: Documentation needed: Describe the branching strategy.

- Possibly *git-flow* once we have tested it? Not sure…
- **branch names:**

  - feature/#xxx/detailed-description
  - bugfix/#xxx/detailed-description
  - hotfix/#xxx/detailed-description
  - enhancement/#xxx/detail-description
  - int/vX.Y.Z+

Commit messages

Use common git message format guidelines, such as:

- https://www.freecodecamp.org/news/writing-good-commit-messages-a-practical-guide/
- https://chris.beams.io/posts/git-commit/

5.3.3 Python coding

Strings

Todo: Documentation needed for string coding rules:

- **Differenciate strings and byte-strings:**
  - **Use of `""` / `r""` / `f""` (double quote) to enclose `str` strings**
    - Except for strings in f-string {}-blocks.
  - Use of `b''` / `rb''` (simple quotes) to enclose `bytes` strings
- **Use f-strings**
  - Except for debugging (for optimization concerns)
  - Except for assertion errors and evidence (for optimization concerns)
  - Except for regex (because of '{}' escaping)
  - Except for bytes (f-strings not available)
- Use *repr* specification (`f"{...!r}"` / `"%r"`) instead of calling `repr()` (for optimization concerns)

Namings

Todo: Documentation needed for namings:

- PEP8 compatible
- Packages
- Modules
- Classes
- Attributes

- Methods & functions
- Getters (properties) and setters, same as attributes
- Constants

Presentation

Todo: Documentation needed for code presentation

- Indentation:
  - Avoid right-aligned comments (hard to maintain when the names change)
  - Functions and methods (same purpose):
    * New line for the first parameter
    * Parameters indented with 2 tabs (as proposed by PyCharm by default). Makes it more readable by differenciating the parameters from the function / body.
- Trailing commas (refer to PEP8 https://www.python.org/dev/peps/pep-0008/#when-to-use-trailing-commas)
- New line after the opening parenthesis of the function declarations

Packages

Todo: Documentation needed for packages, file names:

- *scenario* package
- '__init__.py' that exports symbols from the given package

Note: `scenario.test`, `scenario.tools` subpackages are implemented at different locations, out of the main 'src/' directory.

Static & class methods

Do not use the `@staticmethod` or `@classmethod` whenever a method could be converted so. It is preferrable to rely on the meaning at first, in order to make the code more stable along the time.

By default, PyCharm "detects any methods which may safely be made static". This coding rule goes against these suggestions. By the way, we do not want to set `# noinspection PyMethodMayBeStatic` pragmas everywhere a method could "be made static" in the code. Thus, please disable the "Method may be static" inspection rule in your PyCharm settings.

Typings

The code uses Python 2 typings.

Even though Python 2 compatibility is not expected, it has proved that comment type hints did not suffer a couple of limitations compared with Python 3 type hints:

- impossible for a method to return an instance of the current class, without a prior `import __future__ import annotations` statement, which is available from Python 3.8 only,

```
# Python 3 type hints

from __future__ import annotations  # << Python 3.8 only

class A:
    def meth() -> A:  # << NameError: name 'A' is not defined, if `__future__.
↪annotations` not imported
        return A()
```

```
# Python 2 type hints

class A:
    def meth():  # type: (...) -> A
        return A()
```

- impossible to define the type of the iterating variable in a *for* loop statement,

```
# Python 3 type hints

for _i: int in range(10):  # << SyntaxError: invalid syntax
    pass
```

```
# Python 2 type hints

for _i in range(10):  # type: int
    pass
```

- impossible to define the type of a couple of variables initialized from a function returning a tuple.

```
# Python 3 type hints

_a: int, _b: str = func()  # << SyntaxError: invalid syntax
```

```
# Python 2 type hints

_a, _b = func()  # type: int, str
```

Furthermore, we use the multi-line style, that makes the code diffs shorter, and the maintainance simpler by the way.

Imports

Order of Imports

1. Place system imports at first:
   - One system import per `import` line.
   - Sort imports in the alphabetical order.
2. Then *scenario* imports:
   - Use the relative form `from .modulename import ClassName`.
   - Sort *scenario* imports in the alphabetical order of module names.
   - For a given module import statement, sort the symbols imported in their alphabetical order as well.
3. Then test imports (for test modules only).

Justification for ordering imports

Giving an order for imports is a matter of code stability.

The alphabetical order works in almost any situation (except on vary rare occasion). It's simple, and easy to read through. That's the reason why it is chosen as the main order for imports.

The fewer project imports at the top level

We call project imports imports from modules located in the same package. As a consequence, these imports are usually the relative ones.

In order to avoid cyclic module dependencies in the package, the fewer project imports shall be placed at the top level of modules. Postpone as much as possible the imports with local imports in function or methods where the symbol is actually used.

Nevertheless, this does not mean that local imports should be repeated every time that a basic object is used in each function of method (like *Path* for instance).

In order to ensure that a top level project import is legitimate, it shall be justified with a comment at the end of the `import` line. `typing.TYPE_CHECKING` import statements shall be justified as well (see the *typing.TYPE_CHECKING* section below).

The 'tools/checkdeps.py' script may help visualizing *scenario* module dependencies:

```
$ ./tools/checkdeps.py
```

```
INFO     <13>               __init__.py => [actionresultdefinition.py,␣
↪actionresultexecution.py, args.py, assertionhelpers.py,
INFO                                        assertions.py, campaignargs.py,␣
↪campaignexecution.py, campaignreport.py,
INFO                                        campaignrunner.py, configdb.py, confignode.
↪py, console.py, datetimeutils.py,
INFO                                        debugutils.py, enumutils.py, errcodes.py,␣
↪executionstatus.py, handlers.py,
INFO                                        issuelevels.py, knownissues.py, locations.py,
↪ logextradata.py, logger.py,
```

(continues on next page)

(continued from previous page)

```
INFO                                          loggermain.py, path.py, pkginfo.py,␣
↪scenarioargs.py, scenarioconfig.py,
INFO                                          scenariodefinition.py, scenarioevents.py,␣
↪scenarioexecution.py, scenarioreport.py,
INFO                                          scenariorunner.py, scenariostack.py, stats.
↪py, stepdefinition.py, stepexecution.py,
INFO                                          subprocess.py, testerrors.py, timezoneutils.
↪py]
INFO     <12>        campaignlogging.py => [campaignexecution.py, enumutils.py]
INFO     <12>         campaignreport.py => [campaignexecution.py, logger.py, path.py,␣
↪xmlutils.py]
INFO     <12>         campaignrunner.py => [campaignexecution.py, errcodes.py, logger.py]
INFO     <12>         scenarioevents.py => [campaignexecution.py, enumutils.py,␣
↪scenariodefinition.py, stepdefinition.py,
INFO                                          testerrors.py]
INFO     <11>      campaignexecution.py => [executionstatus.py, path.py,␣
↪scenarioexecution.py, stats.py, testerrors.py]
INFO     <11>        scenariologging.py => [actionresultdefinition.py, enumutils.py,␣
↪scenariodefinition.py,
INFO                                          scenarioexecution.py, stepdefinition.py,␣
↪stepsection.py, testerrors.py]
INFO     <11>         scenarioreport.py => [actionresultdefinition.py, logger.py,␣
↪scenarioexecution.py, stepdefinition.py]
INFO     <11>        scenarioresults.py => [logger.py, scenarioexecution.py, testerrors.
↪py]
INFO     <11>          scenariostack.py => [actionresultdefinition.py,␣
↪actionresultexecution.py, logger.py,
INFO                                          scenariodefinition.py, scenarioexecution.py,␣
↪stepdefinition.py, stepexecution.py,
INFO                                          stepuserapi.py]
INFO     <10>               handlers.py => [logger.py, scenariodefinition.py]
INFO     <10>      scenarioexecution.py => [executionstatus.py, scenariodefinition.py,␣
↪stats.py, stepdefinition.py]
INFO     <10>         scenariorunner.py => [actionresultdefinition.py, enumutils.py,␣
↪errcodes.py, knownissues.py, logger.py,
INFO                                          scenariodefinition.py, stepdefinition.py,␣
↪stepuserapi.py, testerrors.py]
INFO     <09>     scenariodefinition.py => [assertions.py, logger.py, stepdefinition.py,␣
↪stepsection.py, stepuserapi.py]
INFO     <08>          stepexecution.py => [actionresultdefinition.py, stats.py,␣
↪stepdefinition.py]
INFO     <08>            stepsection.py => [stepdefinition.py]
INFO     <07>         stepdefinition.py => [actionresultdefinition.py, assertions.py,␣
↪knownissues.py, locations.py, logger.py,
INFO                                          stepuserapi.py]
INFO     <06>  actionresultexecution.py => [actionresultdefinition.py]
INFO     <06>           campaignargs.py => [args.py, path.py, scenarioargs.py]
INFO     <06>            knownissues.py => [logger.py, testerrors.py]
INFO     <05> actionresultdefinition.py => [enumutils.py, locations.py]
INFO     <05>           scenarioargs.py => [args.py, subprocess.py]
INFO     <05>             testerrors.py => [locations.py, logger.py]
INFO     <04>                   args.py => [configargs.py, logger.py, loggingargs.py]
```

(continues on next page)

(continued from previous page)

```
INFO     <04>              configdb.py => [configmode.py, enumutils.py, logger.py]
INFO     <04>           debugloggers.py => [debugutils.py, logger.py]
INFO     <04>              locations.py => [logger.py]
INFO     <04>            logfilters.py => [logger.py]
INFO     <04>            loggermain.py => [logger.py]
INFO     <04>                reflex.py => [debugclasses.py, logger.py]
INFO     <04>            subprocess.py => [errcodes.py, logger.py]
INFO     <04>          testsuitefile.py => [logger.py]
INFO     <03>           logformatter.py => [console.py, logextradata.py]
INFO     <03>                 logger.py => [console.py, logextradata.py]
INFO     <02>            assertions.py => [assertionhelpers.py]
INFO     <02>           debugclasses.py => [enumutils.py]
INFO     <02>        executionstatus.py => [enumutils.py]
INFO     <02>           logextradata.py => [enumutils.py]
INFO     <02>         scenarioconfig.py => [configmode.py, console.py, enumutils.py,␣
↪path.py]
INFO     <01>       assertionhelpers.py => []
INFO     <01>             configargs.py => []
INFO     <01>              configini.py => []
INFO     <01>             configjson.py => []
INFO     <01>              configkey.py => []
INFO     <01>             configmode.py => []
INFO     <01>            configtypes.py => []
INFO     <01>             configyaml.py => []
INFO     <01>                console.py => []
INFO     <01>          datetimeutils.py => []
INFO     <01>             debugutils.py => []
INFO     <01>              enumutils.py => []
INFO     <01>                errcodes.py => []
INFO     <01>            issuelevels.py => []
INFO     <01>            loggingargs.py => []
INFO     <01>         loggingservice.py => []
INFO     <01>             loghandler.py => []
INFO     <01>                   path.py => []
INFO     <01>                pkginfo.py => []
INFO     <01>                  stats.py => []
INFO     <01>            stepuserapi.py => []
INFO     <01>               textfile.py => []
INFO     <01>          timezoneutils.py => []
INFO     <01>                 typing.py => []
INFO     <01>               xmlutils.py => []
```

`typing.TYPE_CHECKING` pattern

See https://stackoverflow.com/questions/39740632/python-type-hinting-without-cyclic-imports for some information on this so-called pattern in this section.

Use the `typing.TYPE_CHECKING` pattern for two reasons only:

1. for types declared only when `typing.TYPE_CHECKING` is `True` of course (otherwise the code execution will fail),
2. but then, **for cyclic type hint dependencies only**.

Risks with the `if typing.TYPE_CHECKING:` pattern

Sometimes, a class `A` requires another class `B` (in one of its method signatures, or possibly because it inherits from `B`), and so does the other class `B` with class `A` as well.

From the execution point of view, this situation can usually be handled with local imports in some of the methods involved.

Still, from the type hint point of view, a cyclic dependency remains between the two modules. The `typing.TYPE_CHECKING` pattern makes it possible to handle such cyclic dependencies.

But caution! the use of this pattern generates a risk on the execution side. Making an import under an `if typing.TYPE_CHECKING:` condition at the top of a module makes the type checking pass. Nevertheless, the same import should not be forgotten in the method(s) where the cyclic dependency is actually used, otherwise it fails when executed, which is somewhat counterproductive regarding the type checking goals!

Let's illustrate that point with an example.

Let the `a.py` module define a super class `A` with a `getb()` method returning a `B` instance or `None`:

```
if typing.TYPE_CHECKING:
    from .b import B

class A:
    def getb(self, name):  # type: (str) -> typing.Optional[B]
        for _item in self.items:
            if isinstance(_item, B):
                return _item
        return None
```

Let the `b.py` module define `B`, a subclass of `A`:

```
from .a import A

class B(A):
    def __init__(self):
        A.__init__(self)
```

The `B` class depends on the `A` class for type hints *and* execution. So the `from .a import A` import statement must be set at the top of the `b.py` module.

The `A` class needs the `B` class for the signature of its `A.getb()` method only. Thus, the `from .b import B` import statement is set at the top of the `a.py` module, but under a `if typing.TYPE_CHECKING:` condition.

This makes type checking pass, but fails when the `A.getb()` method is executed. Indeed, in `a.py`, as the `B` class is imported for type checking only, the class is not defined when the `isinstance()` call is made. By the way, the import statement must be repeated as a local import when the `B` class is actually used in the `A.getb()` method:

```
class A:
    def getb(self, name):  # type: (str) -> typing.Optional[B]
        # Do not forget the local import!
        from .b import B

        for _item in self.items:
            if isinstance(_item, B):
                return _item
        return None
```

That's the reason why the `typing.TYPE_CHECKING` pattern shall be used as less as possible, i.e. when cyclic dependencies occur because type hints impose it.

Python compatibility

The code supports Python versions from 3.6.

The 'tools/checktypes.py' scripts checks code amongst Python 3.6.

Python versions

No need to handle Python 2 anymore, as long as its end-of-line was set on 2020-01-01 (see PEP 373).

As of 2021/09, Python 3.6's end-of-life has not been declared yet (see https://devguide.python.org/devcycle/#end-of-life-branches), while Python 3.5's end-of-life was set on 2020-09-30 (see PEP 478).

5.3.4 Documentation

Publication

HTML documentation, if saved in a github repository, can be displayed using a `https://htmlpreview.github.io/?https://github.com/...` indirection (inspired from https://stackoverflow.com/questions/8446218/how-to-see-an-html-page-on-github-as-a-normal-rendered-html-page-to-see-preview#12233684). The display however is not optimal.

The documentation is published on the https://docs.readthedocs.io/ platform.

Docstrings

Python docstrings follow the *ReStructured Text* format.

PyCharm configuration

In order to make PyCharm use the *ReStructured Text* format for docstrings, go through: "File" > "Settings" > "Tools" > "Python Integrated Tools" > "Docstrings" > "Docstring format" (as of PyCharm 2021.1.1)

Select the "reStructured Text" option.

The 'Initializer' word in `__init__()` docstrings should be avoided. `__init__()` docstrings should be more specific on what the initializers do for the object.

Sphinx accepts a couple of keywords for a same meaning (see stackoverflow.com#34160968 and github.com). Let's choose of them:

Table 2: Preferred ReStructured Text tags

Preferred tag	Unused tags	Justification
`:return:`	`:returns:`	`:return:` is the default tag used by PyCharm when generating a docstring pattern.
`:raise:`	`:raises:`	Consistency with `:return:`.

The `:raise:` syntax is the following:

```
"""
:raise: Unspecified exception type.
:raise ValueError: System exception.
:raise .neighbourmodule.MyException: Project defined exception.
"""
```

The exception type can be specified:

- It must be set before the second colon (Sphinx handles it a makes an dedicated presentation for it).
- It can be either a system exception type, or a project exception defined in the current or a neighbour module (same syntax as within a `:class:`MyException`` syntax).

Admonitions: notes, warnings...

The `.. admonition::` directive makes it possible to define a title for a "message box" block (see https://docutils.sourceforge.io/docs/ref/rst/directives.html#generic-admonition). Eg:

```
.. admonition:: Message box title
    :class: tip

    Message box content...
```

Message box title

Message box content...

The `:class:` attribute shall be set with one of the following classes (see https://docutils.sourceforge.io/docs/ref/rst/directives.html#specific-admonitions):

- `tip` (do not use `hint`)
- `note`
- `important`
- `warning` (do not use `attention`, `caution` nor `danger`)
- `error`

When no title is needed, the directive with names corresponding to the selected classes above may be used. Eg:

```
.. tip:: Short tip text, without title,
         which may be continued on the next line.
```

Tip: Short tip text, without title, which may be continued on the next line.

ReStructured Text indentation

ReStructured Text directives could lead to use indentations of 3 spaces.

Considering that this is hard to maintain with regular configurations of editors, 4 space indentations shall be preferred in docstrings and *.rst* files.

Domains

Default domain

Unless the `.. default-domain::` directive is used, the Python domain is the default domain.

We do not use the `:py` domain specification in the Python docstrings, in as much as it is implicit.

However, we use the `:py` domain specification in *.rst* files in order to be explicit for cross referencing python objects.

Cross references

Use relative imports as much as possible to reference symbols out of the current module.

In as much as *Sphinx* does not provide a directive to cross-reference them, use double backquotes to highlight function and method parameters.

Cross referencing parameters

There is no current cross reference directive for function and method parameters (see sphinx#538).

From the documentation of the python domain, the best existing directive would be `:obj:` but it is not really clear (`:attr:` is for data attributes of objects).

Other useful resources on that topic:

- https://stackoverflow.com/questions/11168178/how-do-i-reference-a-documented-python-function-parameter-using-sphinx-mark
- https://pypi.org/project/sphinx-paramlinks/

Module attributes

Module attributes should be documented using the `.. py:attribute::` pragma, extending the `__doc__` variable.

```
__doc__ += """
.. py:attribute:: MY_CONST

    Attribute description.
"""
MY_CONST = 0  # type: int
```

Otherwise, they may not be cross-referenced from other modules.

Property return type hint

sphinx.ext.autodoc does not make use of property return type hints in the output documentation.

Nevertheless, we do not make use of the `:type:` directive, which would be redundant with the return type hint already set. The sphinx#7837 enhancement request has been opened for that purpose.

5.4 Guidelines

This section describes guidelines that shall be followed.

Bash commands are assumed to be executed from the root directory of the repository.

5.4.1 Deliver on an integration branch

1. Check licence headers:

   ```
   repo-checklicenseheaders
   ```

   There should be no error. Fix headers if needed. If files have been modified, commit them (probably with the `--amend` option).

2. Check typings:

   ```
   ./tools/checktypes.py
   ```

   There should be no error. Fix things if needed. If files have been modified, commit them (probably with the `--amend` option).

3. Check tests:

   Check test data is up-to-date:

   ```
   ./tools/updatetestdata.py
   ```

   ```
   ./test/run-unit-campaign.py
   ```

   There may be warnings, but no errors.

4. Check documentation:

   a. Generation the documentation:

```
rm -rf ./doc/html/
./tools/mkdoc.py
```

Check the 'mkdoc.py' output log errors and warnings:

- There may be TODO warnings, especially for sections that still have to be documented.
- There may be warnings for "duplicate object" (see issue #25)

There shall be no other errors.

b. Check the HTML output in 'doc/html/':

Check the pages that use the `.. literalinclude::` directive with the `:lines:` option (following list established as of version v0.2.0):

- advanced.config-db.html
- advanced.handlers.html
- advanced.launcher.html
- advanced.logging.html
- advanced.subscenarios.html
- advanced.test-libs.html
- quickstart.html

5. Check files encoding:

Check all files use utf-8 encoding and unix end-of-line characters, and have the appropriate permissions:

```
repo-checkfiles --all
```

If files have been modified, this should be minor modifications. Check line encoding modifications with `git diff -b`. Commit the modifications (probably with the `--amend` option).

5.4.2 Deliver a new version

0. Merge on the master branch:

```
git checkout master
git merge --squash --ff-only int/vX.Y.Z+
```

Update the commit message, then:

```
git commit
```

1. Check the scenario version stored in the code:

Check the version tuple defined in 'src/pkginfo.py'.

If files have been modified, commit them (probably with the `--amend` option).

2. Apply *delivery checking* as described before.
3. Update the documentation:

Check the `copyright` and `version` variables in 'tools/conf/sphinx/conf.py'.

Regenerate the documentation:

```
rm -rf doc/html/
./tools/mkdoc.py
```

Commit modifications (with the `--amend` option).

4. Add a tag on the final node:

```
git tag vX.Y.Z
```

5. Push on the github repository:

```
git push
git push vX.Y.Z
```

6. Configure readthedocs:

Go to the readthedocs project page.

Configure a build for the new version, and set it as the default.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y