RobotFramework_TestsuitesManagement

v. 0.7.3

Mai Dinh Nam Son

18.09.2023

Contents

Chapter 1

Introduction

The **RobotFramework_TestsuitesManagement** enables users to define dynamic configuration values within separate configuration files in JSON format.

These configuration values are available during test execution - but under certain conditions that can be defined by the user (e.g. to realize a variant handling). This means: Not all parameter values are available during test execution - only the ones that belong to the current test scenario.

To realize this, the **RobotFramework_TestsuitesManagement** provides the following features:

1. Split all possible configuration values into several JSON configuration files, with every configuration file contains a specific set of values for configuration parameter
2. Use nested imports of JSON configuration files
3. Follow up definitions in configuration files overwrite previous definitions (of the same parameter)
4. Select between several criteria to let the Robot Framework use a certain JSON configuration file

How to install

The **RobotFramework_TestsuitesManagement** can be installed in two different ways: via PyPi (recommended for users) and via GitHub (recommended for developers).

Installation details can be found in the README.

Further links

For self-study a tutorial is available containing lots of examples. Here you find the rendered tutorial documentation.

For the development environment **VSCodium** an extension is available to support the features of the **RobotFramework_TestsuitesManagement**: vscode-jsonp. This extension adapts e.g. the syntax highlighting of the editor.

Chapter 2

Description

2.1 Meaning of "Test Suites Management"

In the scope of the Robot Framework a test suite is either a single robot file containing one or more test cases, or a set of several robot files.

Usually all test cases of a test suite run under the same conditions - but these conditions may be different. For example the same test case is used to test several different variants of a system under test. Every variant requires individual values for certain configuration parameters.

Tests are carried out at several test benches. All test benches have different hardware configurations. Also the different test benches may require individual values for configuration parameters used in the tests.

Therefore the same tests have to run under different conditions!

The Robot Framework provides several places to define parameters: robot files, resource files, parameter files. But these parameters are fixed. Therefore we need a more dynamic way of accessing parameters. And we postulate the following: When switching between tests of several variants and test executions on several test benches, no changes shall be required within the test code.

The outcome is that another position has to be introduced to store values for variant and test bench specific parameters. And a possibility has to be provided to dynamically make either the one or the other set of values vailable during the execution of tests - depending on outer circumstances like "*which variant?*" and "*which test bench?*". Those dynamic configuration values are stored within separate configuration files in JSON format and the **RobotFramework_TestsuitesManagement** makes the values available globally during the test execution.

Two different kinds of JSON configuration files are involved:

1. *parameter configuration files*

   These configuration files contain all parameter definitions (can be more than one configuration file in a project)

2. *variant configuration file*

   This is a single configuration file containing the mapping between the several parameter configuration files and a name (usually the name of a variant). This name can be used in command line to select a certain parameter configuration file containing the values for this variant.

   Background: It's easier simply to use a name for referencing a certain variant instead of having the need always to mention the path and name of a configuration file.

To realize a concrete test suites management for your project, you need to

- identify the parameters that are variant specific, depending on the number of variants in your project,
- identify the parameters that are test bench specific, depending on the number of test benches in your project,
- identify the parameters that are both: variant specific and test bench specific,
- identify the parameters that have the same value in all variants and test benches.

After this

- for every set of parameters (variant specific and bench specific) you have to introduce a certain parameter configuration file,
- in the variant configuration file you have to define for every variant a variant name together with the path to the corresponding parameter configuration file.

Basically all configuration files of the **RobotFramework_TestsuitesManagement** are implemented in JSON format. This format is extended by some useful features like code comments and imports (nested configuration files). This is explained in more detail in the following chapters. These features cause deviations from standard JSON format. To give applications like editors or syntax checkers a chance to handle these deviations (without invalid findings), all JSON configurations files of the **RobotFramework_TestsuitesManagement** have the extension `.jsonp` , instead of `.json` .

The content of the configuration files is described in the next section.

2.2 Content of configuration files

1. variant configuration file

This file configures the access to all variant dependent `robot_config*.jsonp` files.

```
{
  "default": {
               "name": "robot_execution_config.jsonp",
               "path": ".../config/"
             },
  "variant_1": {
                 "name": "robot_config_variant_1.jsonp",
                 "path": ".../config/"
               },
  "variant_2": {
                 "name": "robot_config_variant_2.jsonp",
                 "path": ".../config/"
               },
  "variant_3": {
                 "name": "robot_config_variant_3.jsonp",
                 "path": ".../config/"
               }
}
```

The example above contains definitions for three variants with names: `variant_1` , `variant_2` and `variant_3` . Additionally a variant named `default` is defined. This default configuration becomes active in case of no certain variant name is provided when the test suite is being executed.

Another aspect is important: the **three dots**. The path to the `robot_config*.jsonp` files depends on the test file location. A different number of `../` is required dependent on the directory depth of the test case location.

Therefore we use here three dots to tell the **RobotFramework_TestsuitesManagement** to search from the test file location up till the `robot_config*.jsonp` files are found:

```
./config/robot_config.jsonp
../config/robot_config.jsonp
../../config/robot_config.jsonp
../../../config/robot_config.jsonp
```

and so on.

Hint: The paths to the `robot_config*.jsonp` files are relative to the position of the test suite - **and not relative to the position of the mapping file in which they are defined!** You are free to move your test suites one or more level up or down in the file system, but using the three dots notation enables you to let the position of the `config` folder unchanged.

It is of course still possible to use the standard notation for relative paths:

```
"path": "./config/"
```

2. parameter configuration files

In these configuration files all parameters are defined, that shall be available globally during test execution.

Some parameters are required. Optionally the user can add own ones. The following example shows the smallest version of a parameter configuration file containing only the most important parameters. This version is a default version and part of the **RobotFramework_TestsuitesManagement** installation.

```
{
  "WelcomeString"   : "Hello... Robot Framework is running now!",
  "Maximum_version" : "1.0.0",
  "Minimum_version" : "0.6.0",
  "Project"         : "RobotFramework Testsuites",
  "TargetName"      : "Device_01"
}
```

`Project` , `WelcomeString` and `TargetName` are simple strings that can be used anyhow. `Maximum_version` and `Minimum_version` are part of a version control mechanism: In case of the version of the currently installed software is outside the range between `Minimum_version` and `Maximum_version` , the test execution stops with an error message.

What is the meaning of "currently installed software"?

- The first possibility is that the **RobotFramework_TestsuitesManagement** runs stand-alone, that means, it is not part of a larger bundle (like the RobotFramework AIO). The installation from PyPi or GitHub causes such a stand-alone installation. In this case the component version of the **RobotFramework_TestsuitesManagement** itself is used for a version control against `Minimum_version` and `Maximum_version` .
- The second possibility is that the **RobotFramework_TestsuitesManagement** runs as part of the RobotFramework AIO. In this case the version of the entire RobotFramework AIO is used for a version control instead.

The version control mechanism is optional. In case you do not need to have your tests under version control, you can set the versions to the value `null` .

```
"Maximum_version" : null,
"Minimum_version" : null,
```

As an alternative it is also possible to remove `Minimum_version` and `Maximum_version` completely.

In case you define only one single version number, only this version number is considered. The following combination makes sure, that the installed software at least is of version 0.6.0, but there is no upper version limit:

```
"Maximum_version" : null,
"Minimum_version" : "0.6.0",
```

Hint: The parameters are keys of an internal configuration dictionary. They have to be accessed in the following way:

```
Log    Maximum_version : ${CONFIG}[Maximum_version]
Log    Project : ${CONFIG}[Project]
```

The following example is an extended version of a configuration file containing also some user defined parameters.

```
{
  "WelcomeString"   : "Hello... Robot Framework is running now!",
  "Maximum_version" : "1.0.0",
  "Minimum_version" : "0.6.0",
  "Project"         : "RobotFramework Testsuites",
  "TargetName"      : "Device_01"
  "params": {
              // global parameters
              "global" : {
                           "param1" : "ABC",
                           "param2" : 25
                         }
            }
}
```

User defined parameters have to be placed inside `params:global` . The intermediate level `global` is introduced to enable further parameter scopes than `global` in future.

All user defined parameters have the scope `params:global` per default. Therefore they can be accessed directly:

```
Log    param1 : ${param1}
```

And another feature can be seen in the example above:
In the context of the **RobotFramework_TestsuitesManagement** the JSON format is an extended one. Deviating from JSON standard it is possible to comment out lines with starting them with a double slash `//` . This allows to add explanations about the meaning of the defined parameters already within the JSON file.

2.3 Access to configuration files

With an installed **RobotFramework_TestsuitesManagement** every test execution requires a configuration - that is the accessibility of a configuration file in JSON format. The **RobotFramework_TestsuitesManagement** provides four different possibilities - also called *level* - to realize such an access. These possibilities are sorted and the **RobotFramework_TestsuitesManagement** tries to access the configuration file in a certain order: Level 1 has the highest priority and level 4 has the lowest priority.

Level 1

Path and name of a parameter configuration file is provided in command line of the Robot Framework.

Level 2 (recommended)

The name of the variant is provided in command line of the Robot Framework.

This level requires that a variant configuration file is passed to the suite setup of the **RobotFramework_TestsuitesManagement**.

Level 2 includes the automated selection of a default variant (in case of no variant name is provided in command line). Also this default variant has to be defined within the variant configuration file.

Level 3

The **RobotFramework_TestsuitesManagement** searches for parameter configuration files within a folder `config` in current test suite folder. In case of such a folder exists and parameter configuration files are inside, they will be used.

Level 4 (unwanted, fallback solution only)

The **RobotFramework_TestsuitesManagement** uses the default configuration file that is part of the installation.

Summary

- With highest priority a parameter configuration file provided in command line, is considered - even in case of also other configuration files (level 2 - level 4) are available.
- If a parameter configuration file is not provided in command line, but a variant name, then the configuration belonging to this variant, is loaded - even in case of also other configuration files (level 3 - level 4) are available.
- If nothing is specified in command line, then the **RobotFramework_TestsuitesManagement** tries to find parameter configuration files within a `config` folder and take them if available - even in case of also the level 4 configuration file is available.
- In case of the user does not provide any information about parameter configuration files to use, the **RobotFramework_TestsuitesManagement** loads the default configuration from installation folder (fallback solution; level 4).

In this context two aspects are important to know for users:

1. *Which parameter configuration file is selected for the test execution?*
   To answer this question the log file contains the path and the name of the selected parameter configuration file.
2. *For which reason is this parameter configuration file selected?*
   To answer this question the log file also contains the level number. The level number indicates the reason.

With these log file entries the test execution is clearly understandable, traceable and scales for huge test suites.

Why is level 2 the recommended one?

Level 2 is the most flexible and extensible solution. Because the robot files contain a link to a variants configuration file, the possible sets of parameter values can already be taken out of the code.

The values selected by level 1, you only see in the log files, but not in the code, because the selection happens in command line only.

Level 3 has a rather strong binding between robot files and configuration files. If you start the test implementation based on level 3 and after this want to have a variant handling, then you have to switch from level 3 to level 2 - and this causes effort in implementation.

Wherease if you start with level 2 immediately and need to consider another set of configuration values for the same tests, then you only have to add another parameter configuration file and another entry in the variants configuration file, without changing any test implementation.

We strongly recommend not to mix up several different configuration levels in one project!

2.4 Activation of "Test Suites Management"

To activate the test suites management you have to import the **RobotFramework_TestsuitesManagement** library in the following way:

```
Library    RobotFramework_TestsuitesManagement    WITH NAME    tm
```

We recommend to use the `WITH NAME` option to shorten the robot code a little bit.

The next step is to call the `testsuite_setup` of the **RobotFramework_TestsuitesManagement** within the `Suite Setup` of your test:

```
Suite Setup    tm.testsuite_setup
```

As long as you

- do not provide a parameter configuration file in command line when executing the test suite (level 1),
- do not provide a variants configuration file as parameter of the `testsuite_setup` (level 2),
- do not have a `config` folder containing parameter configuration files in your test suites folder (level 3),

the **RobotFramework_TestsuitesManagement** falls back to the default configuration (level 4).

In case you want to realize a variant handling you have to provide the path and the name of a variants configuration file to the `testsuite_setup` :

```
Suite Setup    tm.testsuite_setup    ./config/exercise_variants.jsonp
```

To ease the analysis of a test execution, the log file contains informations about the selected level and the path and the name of the used configuration file, for example:

```
Running with configuration level: 2
CfgFile Path: ./config/exercise_config.jsonp
```

Please consider: The `testsuite_setup` requires a variants configuration file (in the example above: `exercise_variants.jsonp`) - whereas the log file contains the resulting parameter configuration file (in the example above: `exercise_config.jsonp`), that is selected depending on the name of the variant provided in command line of the Robot Framework.

2.5 Variants selection

In a previous section the level concept for configuration files has been explained. This section contains corresponding code examples.

1. Selection of a certain parameter configuration file in command line

```
--variable config_file:"(path to parameter configuration file)"
```

2. Selection of a certain variant per name in command line

```
--variable variant:"(variant name)"
```

3. Parameter configuration taken from `config` *folder*

This `config` folder has to be placed in the same folder than the test suites.

Parameter configuration files within this folder are considered under two different conditions:

- The configuration file has the name `robot_config.jsonp` . That is a fix name predefined by the **Robot-Framework_TestsuitesManagement**.
- The configuration file has the same name than a robot file inside the test suites folder, e.g.:
  - Name of test suite file: `example.robot`
  - Path and name of corresponding parameter configuration file: `./config/example.jsonp`

  With this rule it is possible to give every test suite in a certain folder an own individual configuration.

2.6 Local configuration

It might be required to execute tests on several different test benches with every test bench has it's own individual hardware that might require configuration parameter values that are test bench specific. This can be related to common configuration parameters and also to parameters that are variant specific. In the second case a configuration parameter is both variant specific *and* test bench specific.

The *local configuration* feature of the **RobotFramework_TestsuitesManagement** provides the possibility to define test bench specific configuration parameter values.

The meaning of *local* in this context is: placed on a certain test bench - and valid for this bench only.

Also this local configuration is based on configuration files in JSON format. These files are the last ones that are considered when the configuration is loaded. The outcome is that it is possible to define default values for test bench specific parameters in other configuration files - to be also test bench independent. And it is possible to use the local configuration to overwrite these default values with values that are specific for a certain test bench.

Important:

- Local configuration files are fragments only - and not a full configuration! Even so they need to follow the JSON syntax rules. This means, at least they have to start with an opening curly bracket and they have to end with a closing curly bracket.
- Local configuration files must not contain the mandatory top level parameters like the `WelcomeString` and others.

Using the local configuration feature is an option and the **RobotFramework_TestsuitesManagement** provides two ways to realize it:

1. *per command line*

   Path and name of the local parameter configuration file is provided in command line of the Robot Framework with the following syntax:

   ```
   --variable local_config:"(path to local configuration file)"
   ```

2. *per environment variable*

   An environment variable named `ROBOT_LOCAL_CONFIG` exists and contains path and name of a local parameter configuration file.

   The user has to create this environment variable!

   This mechanism allows a user - without any command line extensions - automatically to refer on every test bench to an individual local configuration, simply by giving on every test bench this environment variable an individual value.

The command line has a higher priority than the environment variable. If both is available the local configuration is taken from command line.

Recommendation: *To avoid an accidental overwriting of local configuration files in version control systems we recommend to give those files names that are test bench specific.*

2.7 Priority of configuration parameters

In previous sections the level concept has been explained. This concept introduces four levels of priority that define, which of the possible sources of configuration parameters are processed. But there are other rules involved that influence the priority:

- The local configuration has higher priority than other parameter configurations
- The command line has higher priority than definitions within configuration files

Already in command line we have several possibilities to make settings:

- Set a parameter configuration file (with **RobotFramework_TestsuitesManagement** command line variable `config_file` , *level 1*)
- Set a variant name (with **RobotFramework_TestsuitesManagement** command line variable `variant` , *level 2*)
- Set a local configuration (with **RobotFramework_TestsuitesManagement** command line variable `local_config`)
- Set any other variables (directly with Robot Framework command line variable `--variable`)

And it is possible that in all four use cases the same parameters are used. Or in other words: It is possible to use the `--variable` mechanism to define a parameter that is also defined within a parameter configuration or within a local configuration - or in both together.

Finally this is the order of processing (with highest priority first):

1. Single command line variable (`--variable`)
2. Local configuration (`local_config`)
3. Variant specific configuration (`config_file` or `variant`)

Meaning:

1. Variant specific configuration is overwritten by local configuration
2. Local configuration is overwritten by single command line variable

What happens in case of a command line contains both a `config_file` and a `variant` ?

`config_file` is level 1 and `variant` is level 2. Level 1 has higher priority than level 2. Therefore `config_file` is the valid one. This does **not** mean that `config_file` overwrites `variant` ! In case of a certain level is identified (here: level 1), all other levels are ignored. The outcome is that - in this example - the `variant` has no meaning. Between different levels there is an *either or* relationship. And that is the reason for that it makes no sense to define both in command line, a `config_file` and a `variant` . The **RobotFramework_TestsuitesManagement** throws an error in this case.

But when additionally `--variable` is used to define a new value for a parameter that is already defined in one of the involved configuration files, then the configuration file value is overwritten by the command line value.

And even this is not all. The Robot Framework provides further possibilities to define parameters in command line, e.g. by `--variablefile` . `--variable` and `--variablefile` are Robot Framework mechanisms to define parameters, whereas `config_file` and `local_config` are corresponding **RobotFramework_TestsuitesManagement** mechanisms.

The rules behind all are: `--variable` overrules `--variablefile` . Robot Framework mechanisms overrule **RobotFramework_TestsuitesManagement** mechanisms.

To avoid the things becoming too much complicated, we urgently recommend not to mixup both mechanisms to define *different values for the same parameters* (but to overwrite only a single variable with `--variable` might be OK).

2.8 Nested configuration files

In case of a project requires more and more parameters, it makes sense to split the growing configuration file into smaller ones.

This means, at first we have to split all configuration parameters in

1. parameters that are specific for a certain variant,
2. common parameters that have the same value for all variants

Placing those common parameters in every single variant specific parameter configuration file would create a lot of redundancy. This would also complicate the maintenance.

The solution is to use the variant specific configuration files only for variant specific parameters and to put all common parameters in a separate configuration file. This common parameter file has to be imported in every variant specific parameter file.

The outcome is that still with the selection of a certain variant specific parameter file both types of parameters are available: the variant specific ones and the common ones.

This can be done in the following way:

For example we have the following variant specific configuration files:

```
config/config_variant1.jsonp
config/config_variant2.jsonp
```

Additionaly we have a configuration file with common parameters:

```
config/config_common.jsonp
```

The import of `config_common.jsonp` into `config_variant1.jsonp` and into `config_variant2.jsonp` is possible in the following way:

```
"params" : {
            "global": {
                        "[import]"    : "./config_common.jsonp",
                        "teststring" : "variant specific value"
                       }
           }
```

The key `[import]` indicates the import of another configuration file. The value of the key is the path and name of this file.

Imports can be nested. An imported configuration file is allowed to contain imports also.

The content of the importing file and the content of all imported files are merged. In case of duplicate parameter names follow up definitions overwrite previous definitions of the same parameter!

Important:

- All imported configuration files are fragments only - and not a full configuration! Even so they need to follow the JSON syntax rules. This means, at least they have to start with an opening curly bracket and they have to end with a closing curly bracket.
- Imported configuration files must not contain the mandatory top level parameters like the `WelcomeString` and others.

2.9 Overwritten parameters

Summarized the **RobotFramework_TestsuitesManagement** provides three different types of parameter configuration files to define parameters:

1. A full standard parameter configuration file containing at least the mandatory parameters and - as option - also user defined parameters
2. A parameter configuration file fragment that is imported in other configuration files by the `[import]` key
3. A local parameter configuration file that is also a fragment only, and accessed either by command line or environment variable

All types of configuration file can be used

1. to define new parameters
2. to overwrite already existing parameters

This possibility only belongs to user defined parameters with scope `params:global` !

Example:

1. Define a new parameter:

```
"params" : {
            "global": {
                       "teststring" : "initial value"
                      }
           }
```

2. Overwrite an already existing parameter:

To overwrite a parameter is - after the initial definition - possible at any follow up position

- in the same configuration file or
- in other configuration files like the imported ones or
- in a local configuration file

With the following syntax:

```
${params}['global']['teststring'] : "new value"
```

The resulting value of a parameter at the end depends on the priority (computation order) described in previous sections of this description.

Chapter 3

The JSONP format

This chapter explains the format of JSON files used by the **RobotFramework_TestsuitesManagement** in detail. We concentrate here on the content of the JSON files and the corresponding results, available in Python dictionary format.

3.1 Standard JSON format

The **RobotFramework_TestsuitesManagement** supports JSON files with standard extension `.json` and standard content.

- JSON file:

```
{
   "param1" : "value1",
   "param2" : "value2"
}
```

Outcome:

```
{'param1': 'value1', 'param2': 'value2'}
```

A JSON file with extension `.jsonp` and same content will produce the same output.

We recommend to give every JSON file the extension `.jsonp` to have a strict separation between the standard and the extended JSON format.

The following example still contains standard JSON content, but with parameters of several different data types (simple and composite).

```
{
   "param_01" : "string",
   "param_02" : 123,
   "param_03" : 4.56,
   "param_04" : ["A", "B", "C"],
   "param_05" : {"A" : 1, "B" : 2, "C" : 3}
}
```

This content produces the following output:

```
{'param_01': 'string',
 'param_02': 123,
 'param_03': 4.56,
 'param_04': ['A', 'B', 'C'],
 'param_05': {'A': 1, 'B': 2, 'C': 3}}
```

3.2 Boolean and null values

JSON supports the boolean values `true` and `false`, and also the null value `null`.

In Python the corresponding values are different: `True`, `False` and `None`.

Because the **RobotFramework_TestsuitesManagement** is a Python application and therefore the returned content is required to be formatted Python compatible, the **RobotFramework_TestsuitesManagement** does a conversion automatically.

Accepted in JSON files are both styles:

```
{
   "param_06" : true,
   "param_07" : false,
   "param_08" : null,
   "param_09" : True,
   "param_10" : False,
   "param_11" : None
}
```

The output contains all keywords in Python style only:

```
{'param_06': True,
 'param_07': False,
 'param_08': None,
 'param_09': True,
 'param_10': False,
 'param_11': None}
```

3.3 Comments

Comments can be added to JSON files with `//` :

```
{
   // JSON keywords
   "param_06" : true,
   "param_07" : false,
   "param_08" : null,
   // Python keywords
   "param_09" : True,
   "param_10" : False,
   "param_11" : None
}
```

All lines starting with `//` , are ignored by the **RobotFramework_TestsuitesManagement**. The output of this example is the same than in the previous example.

3.4 Import of JSON files

We assume the following scenario:

A software component *A* requires a set of configuration parameters. A software component *B* that belongs to the same main software or to the same project, requires another set of configuration parameters. Additionally both components require a common set of parameters (with the same values).

The outcome is that at least we need two JSON configuration files:

1. A file `componentA.jsonp` containing all parameters required for component *A*
2. A file `componentB.jsonp` containing all parameters required for component *B*

But with this solution both JSON files would contain also the common set of parameters. This is unfavorable, because the corresponding values need to be maintained at two different positions.

Therefore we extend the list of JSON files by a file containing the common part only:

1. A file `common.jsonp` containing all parameters that are the same for component *A* and component *B*
2. A file `componentA.jsonp` containing remaining parameters (with specific values) required for component *A*
3. A file `componentB.jsonp` containing remaining parameters (with specific values) required for component *B*

Finally we use the import mechanism of the **RobotFramework_TestsuitesManagement** to import the file `common.jsonp` in file `componentA.jsonp` and also in file `componentB.jsonp` .

This can be the content of the JSON files:

- `common.jsonp`

```
{
   // common parameters
   "common_param_1" : "common value 1",
   "common_param_2" : "common value 2"
}
```

- `componentA.jsonp`

```
{
   // common parameters
   "[import]" : "./common.jsonp",
   //
   // component A parameters
   "componentA_param_1" : "componentA value 1",
   "componentA_param_2" : "componentA value 2"
}
```

- `componentB.jsonp`

```
{
   // common parameters
   "[import]" : "./common.jsonp",
   //
   // component B parameters
   "componentB_param_1" : "componentB value 1",
   "componentB_param_2" : "componentB value 2"
}
```

Explanation:

JSON files are imported with the key `"[import]"` . The value of this key is the path and name of the JSON file to be imported.

A JSON file can contain more than one import. Imports can be nested: An imported JSON file can import further JSON files also.

Outcome:

The file `componentA.jsonp` produces the following output:

```
{'common_param_1': 'common value 1',
 'common_param_2': 'common value 2',
 'componentA_param_1': 'componentA value 1',
 'componentA_param_2': 'componentA value 2'}
```

The file `componentB.jsonp` produces the following output:

```
{'common_param_1': 'common value 1',
 'common_param_2': 'common value 2',
 'componentB_param_1': 'componentB value 1',
 'componentB_param_2': 'componentB value 2'}
```

It can be seen that the returned dictionary contains both the parameters from the loaded JSON file and the parameters imported by the loaded JSON file.

3.5 Overwrite parameters

We take over the scenario from the previous section: We still have a JSON file `componentA.jsonp` containig the parameters for component A, a JSON file `componentB.jsonp` for component B and a JSON file `common.jsonp` for both components.

But now component B requires a different value of a common parameter: Within a JSON file we need to change the value of a parameter that is initialized within an imported file. That is possible.

This is now the content of the JSON files:

- `common.jsonp`

```
{
   // common parameters
   "common_param_1" : "common value 1",
   "common_param_2" : "common value 2"
}
```

- `componentA.jsonp`

```
{
   // common parameters
   "[import]" : "./common.jsonp",
   //
   // component A parameters
   "componentA_param_1" : "componentA value 1",
   "componentA_param_2" : "componentA value 2"
}
```

- `componentB.jsonp`

```
{
   // common parameters
   "[import]" : "./common.jsonp",
   //
   // component B parameters
   "componentB_param_1" : "componentB value 1",
   "componentB_param_2" : "componentB value 2",
   // overwrite parameter initialized by imported file
   "common_param_2" : "common componentB value 2"
}
```

Explanation:

With

```
    "common_param_2" : "common componentB value 2"
```

in `componentB.jsonp` the initial definition

```
    "common_param_2" : "common value 2"
```

in `common.jsonp` is overwritten.

Outcome:

The file `componentB.jsonp` produces the following output:

```
{'common_param_1': 'common value 1',
 'common_param_2': 'common componentB value 2',
 'componentB_param_1': 'componentB value 1',
 'componentB_param_2': 'componentB value 2'}
```

Important: *The value a parameter has finally, depends on the order of definitions, redefinitions and imports!*

In file `componentB.jsonp` we move the import of `common.jsonp` to the bottom:

```
{
   // component B parameters
   "componentB_param_1" : "componentB value 1",
   "componentB_param_2" : "componentB value 2",
   "common_param_2" : "common componentB value 2"
   //
   // common parameters
   "[import]" : "./common.jsonp",
}
```

Now the imported file overwrites the value initialized in the importing file.

Outcome:

```
{'common_param_1': 'common value 1',
 'common_param_2': 'common value 2',
 'componentB_param_1': 'componentB value 1',
 'componentB_param_2': 'componentB value 2'}
```

Up to now we considered simple data types only. In case we want to overwrite a parameter that is part of a composite data type, we need to extend the syntax. This is explained in the next examples.

Again we take over the scenario from the previous section: We still have a JSON file `componentA.jsonp` containig the parameters for component *A*, a JSON file `componentB.jsonp` for component *B* and a JSON file `common.jsonp` for both components.

But now all values are part of composite data types like lists and dictionaries.

This is the content of the JSON files:

- `common.jsonp`

```
{
   // common parameters
   "common_param_1" : ["common value 1.1", "common value 1.2"],
   "common_param_2" : {"common_key_2_1" : "common value 2.1" ,
                       "common_key_2_2" : "common value 2.2"}
}
```

- `componentA.jsonp`

```
{
   // common parameters
   "[import]" : "./common.jsonp",
   //
   // component A parameters
   "componentA_param_1" : ["componentA value 1.1", "componentA value 1.2"],
   "componentA_param_2" : {"componentA_key_2_1" : "componentA value 2.1" ,
                           "componentA_key_2_2" : "componentA value 2.2"}
}
```

- `componentB.jsonp`

```
{
   // common parameters
   "[import]" : "./common.jsonp",
   //
   // component B parameters
   "componentB_param_1" : ["componentB value 1.1", "componentB value 1.2"],
   "componentB_param_2" : {"componentB_key_2_1" : "componentB value 2.1" ,
                           "componentB_key_2_2" : "componentB value 2.2"}
}
```

Like in previous examples, the outcome is a merge of the imported JSON file and the importing JSON file, e.g. for `componentA.jsonp` :

```
{'common_param_1': ['common value 1.1', 'common value 1.2'],
 'common_param_2': {'common_key_2_1': 'common value 2.1',
                    'common_key_2_2': 'common value 2.2'},
 'componentA_param_1': ['componentA value 1.1', 'componentA value 1.2'],
 'componentA_param_2': {'componentA_key_2_1': 'componentA value 2.1',
                        'componentA_key_2_2': 'componentA value 2.2'}}
```

Now the following questions need to be answered:

1. How to get the value of an already existing parameter?
2. How to get the value of a single element of a parameter of nested data type (list, dictionary)?
3. How to overwrite the value of a single element of a parameter of nested data type?
4. How to add an element to a parameter of nested data type?

We introduce another JSON file `componentB.2.jsonp` in which we import the JSON file `componentB.jsonp` . In this file we also add content to work with simple and composite data types to answer the questions above.

This is the initial content of `componentB.2.jsonp` :

```
{
   // import of componentB parameters
   "[import]" : "./componentB.jsonp",
   //
   // some additional parameters of simple data type
   "string_val" : "ABC",
   "int_val"    : 123,
   "float_val"  : 4.56,
   "bool_val"   : true,
   "null_val"   : null,

   // access to existing parameters
   "string_val_b"         : ${string_val},
   "int_val_b"            : ${int_val},
   "float_val_b"          : ${float_val},
   "bool_val_b"           : ${bool_val},
   "null_val_b"           : ${null_val},
   "common_param_1_b"     : ${common_param_1},
   "componentB_param_2_b" : ${componentB_param_2}
}
```

The rules for accessing parameters are:

- Existing parameters are accessed by a dollar operator and a pair of curly brackets (`${...}`) with the parameter name inside.
- If the entire expression of the right hand side of the colon is such a dollar operator expression, it is not required any more to encapsulate this expression in quotes.
- Without quotes, the dollar operator keeps the data type of the referenced parameter. If you use quotes, the value of the used parameter will be of type `str` .

Outcome:

```
{'bool_val': True,
 'bool_val_b': True,
 'common_param_1': ['common value 1.1', 'common value 1.2'],
 'common_param_1_b': ['common value 1.1', 'common value 1.2'],
 'common_param_2': {'common_key_2_1': 'common value 2.1',
                    'common_key_2_2': 'common value 2.2'},
 'componentB_param_1': ['componentB value 1.1', 'componentB value 1.2'],
 'componentB_param_2': {'componentB_key_2_1': 'componentB value 2.1',
                        'componentB_key_2_2': 'componentB value 2.2'},
 'componentB_param_2_b': {'componentB_key_2_1': 'componentB value 2.1',
                          'componentB_key_2_2': 'componentB value 2.2'},
 'float_val': 4.56,
 'float_val_b': 4.56,
 'int_val': 123,
 'int_val_b': 123,
 'null_val': None,
 'null_val_b': None,
 'string_val': 'ABC',
 'string_val_b': 'ABC'}
```

Let's take a deeper look at the following line:

```
"int_val_b"            : ${int_val},
```

Like mentioned in the rules above, the dollar operator keeps the data type of the referenced parameter: In case of `int_val` is of type `int` , also `int_val_b` will be of type `int` .

Like mentioned in the rules above, it is not required any more to encapsulate dollar operator expressions at the right hand side of the colon in quotes. But nevertheless it is possible to use quotes. In case of:

```
"int_val_b"            : "${int_val}",
```

the parameter `int_val_b` would be of type `string` .

Value of a single element of a parameter of nested data type

To access an element of a list and a key of a dictionary, we change the content of file `componentB.2.jsonp` to:

```
{
   // import of componentB parameters
   "[import]" : "./componentB.jsonp",
   //
   "list_element_0" : ${componentB_param_1}[0],
   "dict_key_2_2"   : ${common_param_2}['common_key_2_2']
}
```

Outcome:

```
{'common_param_1': ['common value 1.1', 'common value 1.2'],
 'common_param_2': {'common_key_2_1': 'common value 2.1',
                    'common_key_2_2': 'common value 2.2'},
 'componentB_param_1': ['componentB value 1.1', 'componentB value 1.2'],
 'componentB_param_2': {'componentB_key_2_1': 'componentB value 2.1',
                        'componentB_key_2_2': 'componentB value 2.2'},
 'dict_key_2_2': 'common value 2.2',
 'list_element_0': 'componentB value 1.1'}
```

Overwrite the value of a single element of a parameter of nested data type

In the next example we overwrite the value of a list element and the value of a dictionary key.

Again we change the content of file `componentB.2.jsonp` :

```
{
   // import of componentB parameters
   "[import]" : "./componentB.jsonp",
   //
   ${componentB_param_1}[0]            : "componentB value 1.1 (new)",
   ${common_param_2}['common_key_2_1'] : "common value 2.1 (new)"
}
```

The dollar operator syntax at the left hand side of the colon is the same than previously used on the right hand side. The entire expression at the left hand side of the colon must *not* be encapsulated in quotes in this case.

Outcome:

The single elements of the list and the dictionary are updated, all other elements are unchanged.

```
{'common_param_1': ['common value 1.1', 'common value 1.2'],
 'common_param_2': {'common_key_2_1': 'common value 2.1 (new)',
                    'common_key_2_2': 'common value 2.2'},
 'componentB_param_1': ['componentB value 1.1 (new)', 'componentB value 1.2'],
 'componentB_param_2': {'componentB_key_2_1': 'componentB value 2.1',
                        'componentB_key_2_2': 'componentB value 2.2'}}
```

Add an element to a parameter of nested data type

Adding further elements to an already existing list is not possible in JSON! But it is possible to add keys to an already existing dictionary.

The following example extends the dictionary `common_param_2` by an additional key `common_key_2_3` :

```
{
   // import of componentB parameters
   "[import]" : "./componentB.jsonp",
   //
   ${common_param_2}['common_key_2_3'] : "common value 2.3"
}
```

Outcome:

```
{'common_param_1': ['common value 1.1', 'common value 1.2'],
 'common_param_2': {'common_key_2_1': 'common value 2.1',
                    'common_key_2_2': 'common value 2.2',
                    'common_key_2_3': 'common value 2.3'},
 'componentB_param_1': ['componentB value 1.1', 'componentB value 1.2'],
 'componentB_param_2': {'componentB_key_2_1': 'componentB value 2.1',
                        'componentB_key_2_2': 'componentB value 2.2'}}
```

Use of a common dictionary

The last example in this section covers the following use case:

- We have several JSON files, each for a certain purpose within a project (e.g. for every feature of this project a separate JSON file).
- They belong together and therefore they are all imported into a main JSON file that is the file that is handed over to the **RobotFramework_TestsuitesManagement**.
- Every imported JSON file introduces a certain bunch of parameters. All parameters need to be a part of a common dictionary.
- Outcome is that finally only one single dictionary is used to access the parameters from all JSON files imported in the main JSON file.

To realize this, it is necessary to separate the initialization of the dictionary from all positions where keys are added to this dictionary.

These are the JSON files:

- `project.jsonp`

```
{
   // initialization
   "project_values" : {},
   //
   // add some common values
   ${project_values}['common_project_param_1'] : "common project value 1",
   ${project_values}['common_project_param_2'] : "common project value 2",
   //
   // import feature parameters
   "[import]" : "./featureA.jsonp",
   "[import]" : "./featureB.jsonp",
   "[import]" : "./featureC.jsonp"
}
```

- `featureA.jsonp`

```
{
   // parameters required for feature A
   ${project_values}['featureA_params'] : {},
   ${project_values}['featureA_params']['featureA_param_1'] : "featureA param 1 value",
   ${project_values}['featureA_params']['featureA_param_2'] : "featureA param 2 value"
}
```

- `featureB.jsonp`

```
{
   // parameters required for feature B
   ${project_values}['featureB_params'] : {},
   ${project_values}['featureB_params']['featureB_param_1'] : "featureB param 1 value",
   ${project_values}['featureB_params']['featureB_param_2'] : "featureB param 2 value"
}
```

- `featureC.jsonp`

```
{
   // parameters required for feature C
   ${project_values}['featureC_params'] : {},
   ${project_values}['featureC_params']['featureC_param_1'] : "featureC param 1 value",
   ${project_values}['featureC_params']['featureC_param_2'] : "featureC param 2 value"
}
```

Explanation:

Every `feature*.jsonp` file refer to the dictionary inizialized within `project.jsonp` .

Important is that the initialization `"project_values" : {},` happens at top level (`project.jsonp`), and not within the imported files (`feature*.jsonp`), otherwise follow up initializations would delete previously added keys of this dictionary!

Outcome:

```
{'project_values': {'common_project_param_1': 'common project value 1',
                    'common_project_param_2': 'common project value 2',
                    'featureA_params': {'featureA_param_1': 'featureA param 1 value',
                                        'featureA_param_2': 'featureA param 2 value'},
                    'featureB_params': {'featureB_param_1': 'featureB param 1 value',
                                        'featureB_param_2': 'featureB param 2 value'},
                    'featureC_params': {'featureC_param_1': 'featureC param 1 value',
                                        'featureC_param_2': 'featureC param 2 value'}}}
```

3.6 dotdict notation

Up to now we have accessed dictionary keys in this way (standard notation):

```
${dictionary}['key']['sub_key']
```

Additionally to this standard notation, the **RobotFramework_TestsuitesManagement** supports the so called *dotdict* notation where keys are handled as attributes:

```
${dictionary.key.sub_key}
```

In standard notation keys are encapsulated in square brackets and all together is placed *outside* the curly brackets. In dotdict notation the dictionary name and the keys are separated by dots from each other. All together is placed *inside* the curly brackets.

In standard notation key names are allowed to contain dots:

```
${dictionary}['key']['sub.key']
```

In dotdict notation this would cause ambiguities:

```
${dictionary.key.sub.key}
```

Therefore it is not possible to implement in this way! In case you need to have dots inside key names, you must use the standard notation. We recommend to prefer underlines as separator - like done in the examples in this document.

Do you really need dots inside key names?

Please keep in mind: The dotdict notation is a reduced one. Because of parts are missing (e.g. the single quotes around key names), the outcome can be code that is really hard to capture.

In the following example we create a composite data structure and demonstate how to access single elements in both notations.

- JSON file:

```
{
   // composite data structure
   "params" : [{"dict_1_key_1" : "dict_1_key_1 value",
                "dict_1_key_2" : ["dict_1_key_2 value 1", "dict_1_key_2 value 2"]},
               //
               {"dict_2_key_1" : "dict_2_key_1 value",
                "dict_2_key_2" : {"dict_2_A_key_1" : "dict_2_A_key_1 value",
                                  "dict_2_A_key_2" : ["dict_2_A_key_2 value 1", ↩
    ↪ "dict_2_A_key_2 value 2"]}}],
   //
   // access to single elements of composite data structure
   //
   // a) standard notation
   "dict_1_key_2_value_2_standard" : ${params}[0]['dict_1_key_2'][1],
   // b) dotdict notation
   "dict_1_key_2_value_2_dotdict" : ${params.0.dict_1_key_2.1},
   //
   // c) standard notation
   "dict_2_A_key_2_value_2_standard" : ${params}[1]['dict_2_key_2']['dict_2_A_key_2'][1]
   // d) dotdict notation
   "dict_2_A_key_2_value_2_dotdict" : ${params.1.dict_2_key_2.dict_2_A_key_2.1}
}
```

Outcome:

In case of the composite data structure becomes more and more nested (and if also the key names contain numbers), understanding the expressions (like `${params.1.dict_2_key_2.dict_2_A_key_2.1}`) becomes more and more challenging!

```
{'dict_1_key_2_value_2_standard': 'dict_1_key_2 value 2',
 'dict_2_A_key_2_value_2_standard': 'dict_2_A_key_2 value 2',
 'params': [{'dict_1_key_1': 'dict_1_key_1 value',
             'dict_1_key_2': ['dict_1_key_2 value 1', 'dict_1_key_2 value 2']},
            {'dict_2_key_1': 'dict_2_key_1 value',
             'dict_2_key_2': {'dict_2_A_key_1': 'dict_2_A_key_1 value',
                              'dict_2_A_key_2': ['dict_2_A_key_2 value 1',
                                                 'dict_2_A_key_2 value 2']}}]}
```

3.7 Substitution of dollar operator expressions

Like shown in previous examples, existing parameters are accessed by a dollar operator and a pair of curly brackets (`${...}`) with the parameter name inside.

We discussed use cases, where the entire expression of the left hand side of the colon and also the entire expression of the right hand side of the colon have been either hard coded strings, encapsulated in quotes, or dollar operator expressions, not encapsulated in quotes.

Also a mix of both is possible!

Outcome is a hard coded string, encapsulated in quotes, with parts represented by one or more than one dollar operator expression. This can be used to create new content very dynamically: On the right hand side of the colon new string values can be created; on the left hand side of the colon this mechanism generates dynamic parameter names.

• JSON file:

```
{
   "project"     : "Test",
   "version"     : 1.23,
   "item_number" : 1,
   "component"   : "componentA",
   //
   "${project}_message_${item_number}" : "Component '${component}' has version ${version}"
}
```

Outcome:

```
{'component': 'componentA',
 'item_number': 1,
 'project': 'Test',
 'Test_message_1': "Component 'componentA' has version 1.23",
 'version': 1.23}
```

We recommend to use simple data types for this kind of substitution only!

But nevertheless, on the right hand side of the colon also composite data types are possible. Here it might makes sense to use them. But on the left hand side of the colon only simple data types are allowed for substitution. Because it makes no sense to create parameter names based on composite data types. Most probably this would cause invalid key names.

To demonstrate this, we change the JSON file to:

```
{
   "component"      : "componentA",
   "composite_data" : ["AB", 12, True, null, {"kA" : "kAval", "kB" : "kBval"}],
   //
   "test_parameter" : "Key values of component '${component}' are: ${composite_data}"
}
```

!!! Caution: substitution bug: issues/92 **!!!**

Outcome:

```
{'component': 'componentA',
 'composite_data': ['AB', 12, True, None, {'kA': 'kAval', 'kB': 'kBval'}],
 'test_parameter': "Key values of component 'componentA' are: ['AB', 12, True, "
                   "None, {'kA': 'kAval', 'kB': 'kBval'}]"}
```

Now we try to do the same on the left hand side of the colon:

```
{
   "param"          : "string value",
   "composite_data" : ["AB", 12, True, null, {"kA" : "kAval", "kB" : "kBval"}],
   //
   "test_parameter_${composite_data}" : ${param}
}
```

Outcome:

```
!!! Error message expected; still under development !!!
```

!!! Caution: issues/69 issuecomment-1589581903 **!!!**

Overwriting vs. substitution

With the last two examples in this section we take a deeper look at the syntax difference between overwriting and substitution.

1. Overwriting:

With `${param2} : ${param1}` the initial value `"XYZ"` of parameter `param2` is overwritten with the value `"ABC"` of parameter `param1` .

```
{
   "param1"  : "ABC",
   "param2"  : "XYZ",
   //
   ${param2} : ${param1},
}
```

Result:

```
{"param1" : "ABC",
 "param2" : "ABC"}
```

2. Substitution:

With `"${param2}" : ${param1}` a new parameter with name `XYZ` (the value of `param2`) and value `"ABC"` is created.

```
{
   "param1"     : "ABC",
   "param2"     : "XYZ",
   //
   "${param2}" : ${param1},
}
```

Result:

```
{"param1" : "ABC",
 "param2" : "XYZ",
 "XYZ" : "ABC"}
```

3.8 Implicite creation of dictionaries

Up to now we have discussed two different ways of creating nested dictionaries.

The first one is "on the fly", like:

```
{
   "project_values" : {"keyA" : "keyA value",
                       "keyB" : {"keyB1" : "keyB1 value",
                                 "keyB2" : {"keyB21" : "keyB21 value",
                                            "keyB22" : "keyB22 value"}}}
}
```

In case of it is required to split the definition into several files, we have to add keys (and also the initialization) line by line:

```
{
   "project_values" : {},
   ${project_values}['keyA'] : "keyA value",
   ${project_values}['keyB'] : {},
   ${project_values}['keyB']['keyB1'] : "keyB1 value",
   ${project_values}['keyB']['keyB2'] : {},
   ${project_values}['keyB']['keyB2']['keyB21'] : "keyB21 value",
   ${project_values}['keyB']['keyB2']['keyB22'] : "keyB22 value"
}
```

The result will be the same as in the previous example.

It can be seen now that this way of creating nested dictionaries is rather long winded, because every inititialization of a dictionary requires a separate line of code (at every level).

To shorten the code, the **RobotFramework_TestsuitesManagement** supports an implicite creation of dictionaries.

This is the resulting code in standard notation:

```
{
   ${project_values}['keyA'] : "keyA value",
   ${project_values}['keyB']['keyB1'] : "keyB1 value",
   ${project_values}['keyB']['keyB2']['keyB21'] : "keyB21 value",
   ${project_values}['keyB']['keyB2']['keyB22'] : "keyB22 value"
}
```

And the same in dotdict notation (with precondition, that no key name contains a dot):

```
{
   ${project_values.keyA} : "keyA value",
   ${project_values.keyB.keyB1} : "keyB1 value",
   ${project_values.keyB.keyB2.keyB21} : "keyB21 value",
   ${project_values.keyB.keyB2.keyB22} : "keyB22 value"
}
```

Caution:

We urgently recommend *not* to mixup both styles in one line of code. In case of keys contain a list and also numerical indices are involved, we recommend to prefer the standard notation.

Please be aware of: In case of a missing level in between an expression like

```
{
   ${project_values.keyB.keyB2.keyB22} : "keyB22 value"
}
```

you will *not* get an error message! The entire data structure will be created implicitely. The impact is that this method is very susceptible to typing mistakes.

The implicite creation of data structures does not work with lists! In case you use a list index out of range, you will get a corresponding error message.

Chapter 4

CConfig.py

4.1 Function: bundle_version

This function prints out the package version which is:

- RobotFramework_TestsuitesManagement version when this module is installed stand-alone (via pip or directly from sourcecode)

- RobotFramework AIO version when this module is bundled with RobotFramework AIO package

Arguments:

- No input parameter is required

Returns:

- No return variable

4.2 Class: CConfig

Imported by:

```
from RobotFramework_TestsuitesManagement.Config.CConfig import CConfig
```

Defines the properties of configuration and holds the identified config files.

The loading configuration method is divided into 4 levels, level1 has the highest priority, Level4 has the lowest priority.

Level1: Handed over by command line argument

Level2: Read from content of json config file

```
{
   "default": {
      "name": "robot_config.jsonp",
      "path": ".../config/"
   },
   "variant_0": {
      "name": "robot_config.jsonp",
      "path": ".../config/"
   },
   "variant_1": {
      "name": "robot_config_variant_1.jsonp",
      "path": ".../config/"
   },
      ...
      ...
}
```

According to the `ConfigName`, RobotFramework_TestsuitesManagement will choose the corresponding config file. `".../config/"` indicats the relative path to json config file, RobotFramework_TestsuitesManagement will recursively find the `config` folder.

Level3: Read in testsuite folder: `/config/robot_config.jsonp`

Level4: Read from RobotFramework AIO installation folder:

`/RobotFramework/defaultconfig/robot_config.jsonp`

4.2.1 Method: loadCfg

This loadCfg method uses to load configuration's parameters from json files.

Arguments:

- No input parameter is required

Returns:

- No return variable

4.2.2 Method: verifyVersion

This verifyVersion validates the current package version with maximum and minimum version (if provided in the configuration file).

The package version is:

- RobotFramework_TestsuitesManagement version when this module is installed stand-alone (via pip or directly from sourcecode)

- RobotFramework AIO version when this module is bundled with RobotFramework AIO package

In case the current version is not between min and max version, then the execution of testsuite is terminated with "unknown" state

Arguments:

- No input parameter is required

Returns:

- No return variable

4.2.3 Method: bValidateMinVersion

This bValidateMinVersion validates the current version with required minimun version.

Arguments:

- `tCurrentVersion`
  / *Condition*: required / *Type*: tuple /
  Current package version.
- `tMinVersion`
  / *Condition*: required / *Type*: tuple /
  The minimum version of package.

Returns:

- `True` or `False`

4.2.4 Method: bValidateMaxVersion

This bValidateMaxVersion validates the current version with required minimun version.

Arguments:

- `tCurrentVersion`
  / *Condition*: required / *Type*: tuple /
  Current package version.
- `tMinVersion`
  / *Condition*: required / *Type*: tuple /
  The minimum version of package.

Returns:

- `True or False`

4.2.5 Method: bValidateSubVersion

This bValidateSubVersion validates the format of provided sub version and parse it into sub tuple for version comparision.

Arguments:

- `sVersion`
  / *Condition*: required / *Type*: string /
  The version of package.

Returns:

- `lSubVersion`
  / *Type*: tuple /

4.2.6 Method: tupleVersion

This tupleVersion returns a tuple which contains the (major, minor, patch) version.

In case minor/patch version is missing, it is set to 0. E.g: "1" is transformed to "1.0.0" and "1.1" is transformed to "1.1.0"

This tupleVersion also support version which contains Alpha (a), Beta (b) or Release candidate (rc): E.g: "1.2rc3", "1.2.1b1", ...

Arguments:

- `sVersion`
  / *Condition*: required / *Type*: string /
  The version of package.

Returns:

- `lVersion`
  / *Type*: tuple /
  A tuple which contains the (major, minor, patch) version.

4.2.7 Method: versioncontrol_error

Wrapper version control error log:

Log error message of version control due to reason and set to unknown state.

Arguments:

- `reason`
  / *Condition*: required / *Type*: string /
  `reason` can only be `conflict_min`, `conflict_max` and `wrong_minmax`.
- `version1`
  / *Condition*: required / *Type*: string /
- `version2`
  / *Condition*: required / *Type*: string /

Returns:

- No return variable

Chapter 5

COnFailureHandle.py

5.1 Class: COnFailureHandle

Imported by:

```
from RobotFramework_TestsuitesManagement.Keywords.COnFailureHandle import
↪  COnFailureHandle
```

5.1.1 Method: is_noney

Chapter 6

CSetup.py

6.1 Class: CSetupKeywords

Imported by:

```
from RobotFramework_TestsuitesManagement.Keywords.CSetup import CSetupKeywords
```

This CSetupKeywords class uses to define the setup keywords which are using in suite setup and teardown of robot test script.

`Testsuite Setup` keyword loads the RobotFramework AIO configuration, checks the version of RobotFramework AIO, and logs out the basic information of the robot run.

`Testsuite Teardown` keyword currently do nothing, it's defined here for future requirements.

`Testcase Setup` keyword currently do nothing, it's defined here for future requirements.

`Testcase Teardown` keyword currently do nothing, it's defined here for future requirements.

6.1.1 Keyword: testsuite_setup

This testsuite_setup defines the `Testsuite Setup` which is used to loads the RobotFramework AIO configuration, checks the version of RobotFramework AIO, and logs out the basic information of the robot run.

Arguments:

- `sTestsuiteCfgFile`

  / *Condition*: required / *Type*: string /

  `sTestsuiteCfgFile=''` and variable `config_file` is not set RobotFramework AIO will check for configuration level 3, and level 4.

  `sTestsuiteCfgFile` is set with a <json_config_file_path> and variable `config_file` is not set RobotFramework AIO will load configuration level 2.

Returns:

- No return variable

6.1.2 Keyword: testsuite_teardown

This testsuite_teardown defines the `Testsuite Teardown` keyword, currently this keyword does nothing, it's defined here for future requirements.

6.1.3 Keyword: testcase_setup

This testcase_setup defines the `Testcase Setup` keyword, currently this keyword does nothing, it's defined here for future requirements.

6.1.4 Keyword: testcase_teardown

This testcase_teardown defines the `Testcase Teardown` keyword, currently this keyword does nothing, it's defined here for future requirements.

6.2 Class: CGeneralKeywords

Imported by:

```
from RobotFramework_TestsuitesManagement.Keywords.CSetup import CGeneralKeywords
```

This CGeneralKeywords class defines the keywords which will be using in RobotFramework AIO test script.

`Get Config` keyword gets the current config object of robot run.

`Load Json` keyword loads json file then return json object.

In case new robot keyword is required, it will be defined and implemented in this class.

6.2.1 Keyword: get_config

This get_config defines the `Get Config` keyword gets the current config object of RobotFramework AIO.

Arguments:

- No parameter is required

Returns:

- `oConfig.oConfigParams`
  / *Type*: json /

6.2.2 Keyword: load_json

Loads a json file and returns a json object.

Arguments:

- `jsonfile`
  / *Condition*: required / *Type*: string /
  The path of Json configuration file.
- `level`
  / *Condition*: required / *Type*: int /
  Level = 1 -> loads the content of jsonfile.
  level != 1 -> loads the json file which is set with variant (likes loading config level2)

Returns:

- `oJsonData`
  / *Type*: json /

Chapter 7

CStruct.py

7.1 Class: CStruct

Imported by:

```
from RobotFramework_TestsuitesManagement.Utils.CStruct import CStruct
```

This `CStruct` class creates the given attributes dynamically at runtime.

Chapter 8

Event.py

8.1 Class: Event

Imported by:

```
from RobotFramework_TestsuitesManagement.Utils.Events.Event import Event
```

8.1.1 Method: trigger

Chapter 9

ScopeEvent.py

9.1 Class: ScopeEvent

Imported by:

```
from RobotFramework_TestsuitesManagement.Utils.Events.ScopeEvent import ScopeEvent
```

9.1.1 Method: trigger

9.2 Class: ScopeStart

Imported by:

```
from RobotFramework_TestsuitesManagement.Utils.Events.ScopeEvent import ScopeStart
```

9.3 Class: ScopeEnd

Imported by:

```
from RobotFramework_TestsuitesManagement.Utils.Events.ScopeEvent import ScopeEnd
```

Chapter 10

__init__.py

10.1 Function: on

10.2 Function: dispatch

10.3 Function: register_event

Chapter 11

LibListener.py

11.1 Class: LibListener

Imported by:

```
from RobotFramework_TestsuitesManagement.Utils.LibListener import LibListener
```

This `LibListener` class defines the hook methods.

- `_start_suite` hooks to every starting testsuite of robot run.
- `_end_suite` hooks to every ending testsuite of robot run.
- `_start_test` hooks to every starting test case of robot run.
- `_end_test` hooks to every ending test case of robot run.

Chapter 12

__init__.py

12.1 Class: RobotFramework_TestsuitesManagement

Imported by:

```
from RobotFramework_TestsuitesManagement.__init__ import
↪  RobotFramework_TestsuitesManagement
```

12.1.1 Method: run_keyword

12.1.2 Method: get_keyword_tags

12.1.3 Method: get_keyword_documentation

12.1.4 Method: failure_occurred

12.2 Class: CTestsuitesCfg

Imported by:

```
from RobotFramework_TestsuitesManagement.__init__ import CTestsuitesCfg
```

Chapter 13

Appendix

About this package:

Table 13.1: Package setup

Setup parameter	Value
Name	RobotFramework_TestsuitesManagement
Version	0.7.3
Date	18.09.2023
Description	Functionality to manage RobotFramework testsuites
Package URL	robotframework-testsuitesmanagement
Author	Mai Dinh Nam Son
Email	son.maidinhnam@vn.bosch.com
Language	Programming Language :: Python :: 3
License	License :: OSI Approved :: Apache Software License
OS	Operating System :: OS Independent
Python required	>=3.0
Development status	Development Status :: 4 - Beta
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

Chapter 14

History

0.1.0	06/2022
Initial version	
0.2.2	07/2022
Created documentation and updated message logs	
0.3.0	07/2022
Added local configuration feature; documentation rework	
0.4.0	03/2023
Maintenance of log output; maintenance of JSON schema validation of configuration file	
0.7.1	05/2023
Introduce package context which allows RobotFramework_TestsuitesManagement work in many contexts: *- stand-alone* *- as part of RobotFramework AIO package*	
0.7.3	09/2023
New RobotFramework 6.1 adaptation; Maintenance of log output	

RobotFramework_TestsuitesManagement.pdf

Created at 18.09.2023 - 16:30:03

by GenPackageDoc v. 0.41.0