JsonPreprocessor

v. 0.8.0

Mai Dinh Nam Son

12.08.2024

Contents

Chapter 1

Introduction

Java**S**cript **O**bject **N**otation (**JSON**) is a text-based format for storing any user defined data and can also be used for data interchange between different applications.

But this format has some limitations and the **JsonPreprocessor** has been introduced to fill the gaps.

The **JsonPreprocessor** extends the JSON format by the following features:

1. Parts of a JSON file can be commented out
2. A JSON file can import other JSON files (nested imports)
3. Parameter can be defined, referenced and overwritten (follow up definitions in configuration files overwrite previous definitions of the same parameter)
4. Also Python specific keywords like `True` , `False` and `None` can be used (additionally to the corresponding JSON keywords `true` , `false` and `null`)

The main goal of the **JsonPreprocessor** is to support huge sets of parameters for complex projects. And the features of the **JsonPreprocessor** support this complexity:

1. Like in usual programming languages code comments are useful to explain the meaning of the defined parameters.
2. Splitting all required parameters into several JSON files - that can import each other - enables to distinguish e.g. between local and global parameters or between specific and common parameters. Another advantage of a file split is: Smaller files with a more specific content are easier to maintain than a huge single file that contains all.
3. A possible use case for a file split would be to have a software containing several different components with each component requires an individual set of parameters - and therefore an own JSON file. Additionally all components also require a common set of parameters. In this case all common parameters can be defined within an own JSON file that is imported into all other JSON files containing the specific values. This procedure avoids redundancy in parameter definitions.
4. Parameters can be initialized in common JSON files and overwritten in specific JSON files that import the common ones.

But this has consequences: The new features cause some deviations from JSON standard.

These deviations harm the syntax highlighting of editors and also cause invalid findings of JSON format related static code checkers.

To avoid conflicts between the standard JSON format and the extended JSON format described here, the **JsonPreprocessor** uses the alternative file extension `.jsonp` for all JSON files.

References:

The **JsonPreprocessor** is hosted in PyPi (recommended for users) and in GitHub (recommended for developers):

- JsonPreprocessor in PyPi
- JsonPreprocessor in GitHub

Details about how to get the **JsonPreprocessor** can be found in the README.

For the development environment **VSCodium** an extension is available to support the extended JSON format of the **JsonPreprocessor**: vscode-jsonp

Chapter 2

Description

2.1 How to execute

The **JsonPreprocessor** is implemented in Python3 and therefore requires a Python3 installation.

A basic Python script to use the **JsonPreprocessor** can look like this:

```
from JsonPreprocessor.CJsonPreprocessor import CJsonPreprocessor
import pprint

json_preprocessor = CJsonPreprocessor()
try:
   values = json_preprocessor.jsonLoad("./file.jsonp")
   pprint.pprint(values)
except Exception as reason:
   print(f"'{reason}'")
```

The main method of the **JsonPreprocessor** is: `jsonLoad` . Input is the path and the name of a JSON file. Output is a dictionary containing all values parsed from this JSON file.

In case of any errors while computing the JSON file, the **JsonPreprocessor** throws an exception. Therefore it is required to call the method `jsonLoad` inside a `try/except` block.

`pprint` is used in this example to give the output a better readability in console.

In chapter The JSONP format the format of JSON files used by the **JsonPreprocessor**, is described in detail. All discussed JSON files can be tested with the example script listed above.

2.2 VSCodium support

In the introduction we mentioned that the JSON syntax extensions introduced by the **JsonPreprocessor**, harm the syntax highlighting of editors.

Either we give the JSON files the extension `.json`, then an editor expects a JSON file in standard syntax, or we change the extension to `.jsonp`, but in this case an editor usually does not know how to display a file of such type.

In case you use VSCodium, you can install a jsonp extension.

With this extension the VSCodium editor will be able to display `.jsonp` files properly.

Some Impressions:

```
{
    // 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"
}
```

```
// 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" : ${params.0.dict_1_key_2.1},
// -> 'The variable '${params}['0']['dict_1_key_2']['1']' is not available!'
//
// c) standard notation
"dict_2_A_key_2_value_2_standard" : ${params}[1]['dict_2_key_2']['dict_2_A_key_2'][1]
```

Chapter 3

The JSONP format

This chapter explains the format of JSON files used by the **JsonPreprocessor** 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 **JsonPreprocessor** 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}}
```

This output is of a certain dictionary type (named *dotdict*) that allows to access elements also with an object oriented dot notation (details about this format can be found in section dotdict notation).

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 **JsonPreprocessor** is a Python application and therefore the returned content is required to be formatted Python compatible, the **JsonPreprocessor** 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 **JsonPreprocessor**. The output of this example is the same than in the previous example.

Also block comments and inline comments are possible, realized by a pair of `/* */` :

```
{
   /*
   "param1" : 1,
   "param2" : "A",
   */

   "testlist" : ["A1", /*"B2", "C3",*/ "D4"]
}
```

Outcome:

```
{'testlist': ['A1', 'D4']}
```

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 **JsonPreprocessor** 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.

We introduce a new file `componentB.2.jsonp` that imports `componentB.jsonp` and creates new parameters based on already existing parameters:

```
{
   // 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}
}
```

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'}
```

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 converted to type `str` . This implicit string conversion is limited to parameters of simple data types like integers and floats. Composite data types like lists and dictionaries cannot be used for that.

In more detail:

The dollar operator keeps the data type of the referenced parameter. In case of `int_val` is of type `int` , also `int_val_b` is of type `int` :

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

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` is of type `str` .

Further content can be added between the double quotes. This can be used to create composite strings:

```
"str_val"    : "ABC",
"int_val"    : 1,
"float_val"  : 2.3,
"bool_val"   : True,
"none_val"   : None,
"list_val"   : [1,2,3],
"dict_val"   : {"A" : "B"},
"newparam1" : "prefix_${str_val}_suffix",
"newparam2" : "prefix_${int_val}_suffix",
"newparam3" : "prefix_${float_val}_suffix",
"newparam4" : "prefix_${bool_val}_suffix",
"newparam5" : "prefix_${none_val}_suffix"
```

Outcome:

```
{'bool_val': True,
 'dict_val': {'A': 'B'},
 'float_val': 2.3,
 'int_val': 1,
 'list_val': [1, 2, 3],
 'newparam1': 'prefix_ABC_suffix',
 'newparam2': 'prefix_1_suffix',
 'newparam3': 'prefix_2.3_suffix',
 'newparam4': 'prefix_True_suffix',
 'newparam5': 'prefix_None_suffix',
 'none_val': None,
 'str_val': 'ABC'}
```

Using composite data types inside strings is not supported:

```
"newparam6" : "prefix_${listval}_suffix"
```

or:

```
"newparam7" : "prefix_${dictval}_suffix"
```

Result for `"newparam6"` :

```
The substitution of parameter '${listval}' inside the string value ↩
    ↪ 'prefix_${listval}_suffix' is not supported! Composite data types like lists and ↩
    ↪ dictionaries cannot be substituted inside strings.
```

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'}}
```

Dictionary keys and indices as parameter

In all code examples above the indices of lists and the key names of dictionaries have been hard coded strings. It is also possible to use parameters:

```
{
   "index1"                : 0,
   "index2"                : 1,
   "key1"                  : "keyA",
   "key2"                  : "keyB",
   "testlist"              : ["A", "B"],
   "testdict"              : {"keyA" : "A", "keyB" : "B"},
   "tmp1"                  : ${testlist}[${index1}],
   "tmp2"                  : ${testdict}[${key1}],
   ${testlist}[${index1}] : ${testlist}[${index2}],
   ${testdict}[${key1}]   : ${testdict}[${key2}],
   ${testlist}[${index2}] : ${tmp1},
   ${testdict}[${key2}]   : ${tmp2}
}
```

Outcome:

```
{'index1': 0,
 'index2': 1,
 'key1': 'keyA',
 'key2': 'keyB',
 'testdict': {'keyA': 'B', 'keyB': 'A'},
 'testlist': ['B', 'A'],
 'tmp1': 'A',
 'tmp2': 'A'}
```

Meaning of single quotes in square brackets

Single quotes are used to convert the content inside to a string.

- In case of the parameter `param` is of type `str` , the expressions `[${param}]` and `['${param}']` have the same outcome: The content inside the square brackets is a string. The single quotes have no meaning in this case (because the parameter is already of type `str`).
- In case of the parameter `param` is of type integer, the quotes in `['${param}']` convert the integer value to a string. Without the quotes (`[${param}]`), the content inside the square brackets is an integer.

In the context of **JsonPreprocessor** JSON files, only strings and integers are expected to be inside square brackets (except the brackets are used to define a list). Other data types are not supported here.

Whether a string or an integer is expected, depends on the data type of the parameter, the square bracket expression belongs to. Dictionaries require a string (a key name), lists require an integer (an index). Deviations will cause an error.

Summarized the following combinations are valid (on both the left hand side of the colon and the right hand side of the colon):

```
${listparam}[${intparam}]
${listparam}[1]
${dictparam}['${intparam}']
${dictparam}[${stringparam}]
${dictparam}['${stringparam}']
${dictparam}['keyname']
```

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 **JsonPreprocessor**.
- 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.

These are the JSON files:

- `project.jsonp`

```
{
   // define 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']['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']['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']['featureC_param_1'] : "featureC param 1 value",
   ${project_values}['featureC_params']['featureC_param_2'] : "featureC param 2 value"
}
```

It is not required to start the code listed above, with dictionary initializations like

```
   "project_values" : {},
   ${project_values}['featureA_params'] : {},
   ${project_values}['featureB_params'] : {},
   ${project_values}['featureC_params'] : {},
```

These initializations are done implicitly by the **JsonPreprocessor**. Further details about the implicit creation of dictionaries can be found in section Implicit creation of dictionaries.

It is for sure still possible to do the initialization of a dictionary explicitly with `{}` . But keep in mind: This deletes all already existing keys in 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 **JsonPreprocessor** 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 Dynamic key names

In section Overwrite parameters we mentioned the possibility to define the value of string parameters dynamically, e.g. in this way:

```
"str_val"   : "ABC",
"newparam1" : "prefix_${str_val}_suffix",
```

The value of `newparam1` is defined by an expression that is encapsulated in quotes and contains - beneath hard coded parts - a dollar operator expression (that is the dynamic part).

The same is also possible on the left hand side of the colon. In this case the name of a parameter is created dynamically.

Example:

```
"strval"  : "A",
"dictval" : {"A_2" : 1},
${dictval}['${strval}_2'] : 2
```

In second line a new dictionary with key `A_2` is defined. In third line we overwrite the initial value of this key with another value. The name of this key is defined with the help of parameter `strval` .

Outcome:

```
{'dictval': {'A_2': 2}, 'strval': 'A'}
```

The same in dotdict notation:

```
"strval"  : "A",
"dictval" : {"A_2" : 1},
${dictval.${strval}_2} : 3
```

The precondition for using dynamic key names is that a key with the resulting name (here `A_2`) does exist already. Therefore this mechanism can be used to overwrite the value of existing keys, but cannot be used to create new keys!

This will not work (because of a key with name `A_2` does not yet exist):

```
"strval"  : "A",
"dictval" : {"${strval}_2" : 1}
```

Outcome:

```
A substitution in key names is not allowed! Please update the key name "${strval}_2"
```

3.8 Implicit 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 **JsonPreprocessor** 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 implicitly. 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.

Key names

The implicit creation of data structures is only possible with *hard coded* key names. Parameters are not supported.

Example:

```
{
   "paramA" : "ABC",
   "subKey" : "ABC",
   ${testdict.subKey.subKey.paramA} : "DEF"
}
```

All sub key levels within the expression `${testdict.subKey.subKey.paramA}` are interpreted as hard coded strings, even in case of parameters with the same name do exist.

For example: The name of the implicitly created key at bottom level is `"paramA"` , and not the value `"ABC"` of the parameter with the same name (`"paramA"`).

Therefore the outcome is:

```
{'paramA': 'ABC', 'subKey': 'ABC', 'testdict': {'subKey': {'subKey': {'paramA': 'DEF'}}}}
```

Reference to existing keys

It is possible to use parameters to refer to *already existing* keys.

```
{
   // data structure created implicitly
   ${testdict.subKey_1.subKey_2.subKey_3} : "ABC",

   // string parameter with name of an existing key
   "keyName_3" : "subKey_3",

   // parameter used to refer to an existing key
   ${testdict.subKey_1.subKey_2.${keyName_3}} : "XYZ"
}
```

Outcome:

```
{'keyName_3': 'subKey_3',
 'testdict': {'subKey_1': {'subKey_2': {'subKey_3': 'XYZ'}}}}
```

Parameters cannot be used to create new keys.

```
{
   // data structure created implicitly
   ${testdict.subKey_1.subKey_2.subKey_3} : "ABC",

   // string parameter with name of a not existing key
   "keyName_4" : "subKey_4",

   // usage of keyName_4 is not possible here
   ${testdict.subKey_1.subKey_2.subKey_3.${keyName_4}} : "XYZ"
}
```

Outcome is the following error:

```
"The implicit creation of data structures based on nested parameter is not supported ..."
```

The same error will happen in case of the standard notation is used:

```
{
   // usage of keyName_4 is not possible here
   ${testdict}['subKey_1']['subKey_2']['subKey_3'][${keyName_4}] : "XYZ"
}
```

Chapter 4

CJsonPreprocessor.py

4.1 Class: CSyntaxType

Imported by:

```
from JsonPreprocessor.CJsonPreprocessor import CSyntaxType
```

4.2 Class: CNameMangling

Imported by:

```
from JsonPreprocessor.CJsonPreprocessor import CNameMangling
```

4.3 Class: CPythonJSONDecoder

Imported by:

```
from JsonPreprocessor.CJsonPreprocessor import CPythonJSONDecoder
```

Extends the JSON syntax by the Python keywords `True`, `False` and `None`.
Arguments:

- `json.JSONDecoder`
  / *Type*: object /
  Decoder object provided by `json.loads`

4.3.1 Method: custom_scan_once

4.4 Class: CJsonPreprocessor

Imported by:

```
from JsonPreprocessor.CJsonPreprocessor import CJsonPreprocessor
```

CJsonPreprocessor extends the JSON syntax by the following features:

- Allow c/c++-style comments within JSON files
- Allow to import JSON files into JSON files
- Allow to define and use parameters within JSON files
- Allow Python keywords `True`, `False` and `None`

4.4.1 Method: getVersion

Returns the version of JsonPreprocessor as string.

4.4.2 Method: getVersionDate

Returns the version date of JsonPreprocessor as string.

4.4.3 Method: jsonLoad

This method is the entry point of JsonPreprocessor.

`jsonLoad` loads the JSON file, preprocesses it and returns the preprocessed result as Python dictionary.

Arguments:

- `jFile`
  / *Condition*: required / *Type*: str /
  Path and name of main JSON file. The path can be absolute or relative and is also allowed to contain environment variables.
- `masterFile`
  / *Condition*: required / *Type*: bool /
  Identifies the entry level when loading JSONP file in comparison with imported files levels. Default value is True

Returns:

- `oJson`
  / *Type*: dict /
  Preprocessed JSON file(s) as Python dictionary

4.4.4 Method: jsonLoads

`jsonLoads` loads the JSONP content, preprocesses it and returns the preprocessed result as Python dictionary.

Arguments:

- `sJsonpContent`
  / *Condition*: required / *Type*: str /
  The JSONP content.
- `referenceDir`
  / *Condition*: required / *Type*: str /
  A reference path for loading imported files.
- `firstLevel`
  / *Condition*: required / *Type*: bool /
  Identifies the entry level when loading JSONP content in comparison with imported files levels.

Returns:

- `oJson`
  / *Type*: dict /
  Preprocessed JSON content as Python dictionary

4.4.5 Method: jsonDump

This method writes the content of a Python dictionary to a file in JSON format and returns a normalized path to this JSON file.

Arguments:

- `oJson`

  / *Condition*: required / *Type*: dict /

- `outFile` (*string*)

  / *Condition*: required / *Type*: str /

  Path and name of the JSON output file. The path can be absolute or relative and is also allowed to contain environment variables.

Returns:

- `outFile` (*string*)

  / *Type*: str /

  Normalized path and name of the JSON output file.

Chapter 5

Appendix

About this package:

Table 5.1: Package setup

Setup parameter	Value
Name	JsonPreprocessor
Version	0.8.0
Date	12.08.2024
Description	Preprocessor for json files
Package URL	python-jsonpreprocessor
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 6

History

0.1.0	01/2022
Initial version	
0.1.4	09/2022
Documentation updated	
0.2.3	05/2023
dotdict format added	
0.2.4	06/2023
Maintenance of dotdict format and log output	
0.3.0	09/2023
- Implicit creation of data structures *- Dotdict feature bug fixing* *- Update nested parameters handling in key name and value* *- Nested parameter feature bug fixing* *- Nested parameters substitution and overwriting improvement* *- Jsonp file path computation improvement*	
0.3.1	12/2023
- Add jsonDump method to write a file in JSON format *- Improve nested parameter format* *- Improve error message log* *- Fix bugs of data structures implicitly* *- Improve index handling together with nested parameters*	
0.3.3	01/2024
- Some bugs fixed in implicitly created data structures *- Improved index handling together with nested parameters* *- Improved format of nested parameters; improved error messages* *- Added getVersion and getVersionDate methods to get current version and the date of the version*	
0.4.0	03/2024
- Optimized regular expression patterns *- Improved duplicated parameters handling* *- Added mechanism to prevent Python application freeze* *- Removed globals scope out of all exec method executions* *- Optimized errors handling while loading nested parameters* *- Fixed bugs*	
0.5.0	04/2024
Extended debugging support. In case of JSON syntax errors, the JsonPreprocessor exception contains an extract of the JSON content nearby the position, where the error occurred.	

0.6.0	05/2024
- JsonPreprocessor returns a dotdict *- Blocked dynamic key names* *- Improved error messages* *- Fixed bugs*	
0.6.1	05/2024
Added pydotdict package to installation dependencies	
0.7.0	06/2024
- Added jsonLoads method that allows users to directly parse JSONP content from strings *- Improved error messages* *- Fixed bugs*	
0.7.1	06/2024
Maintained release workflow	
0.8.0	08/2024
- Implemented a naming convention check for key names within .jsonp files processed by the JsonPreprocessor *- Fixed issues related to error handling deviation* *- Updated error messages log*	

JsonPreprocessor.pdf

Created at 12.08.2024 - 15:32:42

by GenPackageDoc v. 0.41.0