JsonPreprocessor

v. 0.1.4

Mai Dinh Nam Son

14.09.2022

Contents

Chapter 1

Introduction

1.1 Json Preprocessor documentation

This is the documentation for JsonPreprocessor Python package

Json is a format used to represent data and becomes the universal standard of data exchange. Today many software projects are using Json format as a configuration file, for a big or a complex project, there is a need to have some enhanced format in json file such as adding the comments, importing other json files, etc.

Base on that needs, we develop JsonPreprocessor package which allows using the comments, importing other Json files, overwrite existing or add new parameters, and nested parameter.

Chapter 2

Description

2.1 Getting Started

The JsonPreprocessor package is a Python3 package which allows developers to handle additional features in json files such as:

- Adds the comments
- Imports other json files
- Overwrites existing or add new parameters
- Uses nested parameter
- Other features

These json files will be handled by JsonPreprocessor package which returns as result a dictionary object of the deserialized data.

2.2 How to install

Firstly, clone python-jsonpreprocessor repository to your machine.

Then go to python-jsonpreprocessor, using the 2 common commands below to build or install this package:

```
setup.py build      will build the package underneath 'build/'
setup.py install    will install the package
```

After the build processes is completed, the package is located in 'build/', and the generated package documentation is located in **build/lib/JsonPreprocessor**.

We can use `--help` to discover the options for `build` command, example:

```
setup.py build      will build the package underneath 'build/'
setup.py install    will install the package

Global options:
--verbose (-v)      run verbosely (default)
--quiet (-q)        run quietly (turns verbosity off)
--dry-run (-n)      don't actually do anything
--help (-h)         show detailed help message
--no-user-cfg       ignore pydistutils.cfg in your home directory
--command-packages  list of packages that provide distutils commands

Information display options (just display information, ignore any commands)
--help-commands     list all available commands
--name              print package name
```

```
--version (-V)      print package version
--fullname          print <package name>-<version>
--author            print the author's name
--author-email      print the author's email address
--maintainer        print the maintainer's name
--maintainer-email  print the maintainer's email address
--contact           print the maintainer's name if known, else the author's
--contact-email     print the maintainer's email address if known, else the
                    author's
--url               print the URL for this package
--license           print the license of the package
--licence           alias for --license
--description       print the package description
--long-description  print the long package description
--platforms         print the list of platforms
--classifiers       print the list of classifiers
--keywords          print the list of keywords
--provides          print the list of packages/modules provided
--requires          print the list of packages/modules required
--obsoletes         print the list of packages/modules made obsolete

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
 or: setup.py --help [cmd1 cmd2 ...]
 or: setup.py --help-commands
 or: setup.py cmd --help
```

2.3 Features

2.3.1 Adding the comments

With a big or a complex project, it requires a lot of configuration parameters. So adding comments to json files is useful in case of more and more content is added, e.g. because of a json file has to hold a huge number of configuration parameters for different features. Comments can be used here to clarify the meaning of these parameters or the differences between them.

Every line starting with "//", is commented out. Therefore a comment is valid for singles lines only.

Comment out a block of several lines with only one start and one end comment string, is currently not supported.

Example:

```
//************************************************************************
//  Author: ROBFW-AIO Team
//
//  This file defines all common global parameters and will be included to all
//  test config files
//************************************************************************
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  // Version control information.
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "1"
  },
  "params": {
    // Global parameters
    "global": {
      "gGlobalIntParam" : 1,
      "gGlobalFloatParam" : 1.332,  // This parameter is used to configure for ....
      "gGlobalString"   : "This is a string",
```

```
      "gGlobalStructure": {
        "general": "general"
      }
    }
  },
  "preprocessor": {
    "definitions": {
      // FEATURE switches
      "gPreprolIntParam" : 1,
      "gPreproFloatParam" : 1.332,
   // The parameter for feature ABC
      "gPreproString"   : "This is a string",
      "gPreproStructure": {
                           "general": "general"
                          }
      }
  },
  "TargetName" : "gen3flex@dlt"
}
```

2.3.2 Imports other json files

This import feature enables developers to take over the content of other json files into the current json file. A json file that is imported into another json file, can contain imports also (allows nested imports).

A possible usecase for nested imports is to handle similar configuration parameters of different variants of a feature or a component within a bunch of several smaller files, instead of putting all parameter into only one large json file.

Example:

Suppose we have the json file `params_global.json` with the content:

```
//*****************************************************************************
//  Author: ROBFW-AIO Team
//
//  This file defines all common global parameters and will be included to all
//  test config files
//*****************************************************************************
//
//  This is to distinguish the different types of resets
{
  "gGlobalIntParam" : 1,

  "gGlobalFloatParam" : 1.332, // This parameter is used to configure for ....

  "gGlobalString"   : "This is a string",

  "gGlobalStructure": {
    "general": "general"
  }
}
```

And other json file `preprocessor_definitions.json` with content:

```
//*****************************************************************************
//  Author: ROBFW-AIO Team
//
//  This file defines all common global parameters and will be included to all
//  test config files
//*****************************************************************************
{
  "gPreprolIntParam" : 1,
```

```
  "gPreproFloatParam" : 1.332,
  // The parameter for feature ABC
  "gPreproString"   : "This is a string",

  "gPreproStructure": {
                        "general": "general"
                      }
}
```

Then we can import these 2 files above to the json file `config.json` with content:

```
//************************************************************************
//  Author: ROBFW-AIO Team
//
//  This file defines all common global parameters and will be included to all
//  test config files
//************************************************************************
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  // Version control information.
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "1"
  },
  "params": {
    // Global parameters
    "global": {
       "[import]": "<path_to_the_imported_file>/params_global.json"
      }
    },
  "preprocessor": {
    "definitions": {
      // FEATURE switches
        "[import]": "<path_to_the_imported_file>/preprocessor_definitions.json"
      }
  },
  "TargetName" : "gen3flex@dlt"
}
```

The `config.json` file is handled by JsonPreprocessor package, then return the dictionary object for a program like below:

```
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "1"
  },
  "params": {
    "global": {
      "gGlobalIntParam" : 1,
      "gGlobalFloatParam" : 1.332,
      "gGlobalString"   : "This is a string",
      "gGlobalStructure": {
        "general": "general"
        }
```

```
    }
  },
  "preprocessor": {
    "definitions": {
      "gPreprolIntParam" : 1,
      "gPreproFloatParam" : 1.332,
      "gPreproString"   : "This is a string",
      "gPreproStructure": {
                          "general": "general"
                         }
    }
  },
  "TargetName" : "gen3flex@dlt"
}
```

2.3.3 Overwrites existing or add new parameters

This JsonPreprocessor package also provides developers ability to overwrite or update as well as add new parameters. Developers can update parameters which are already declared and add new parameters or new element into existing parameters. The below example will show the way to do these features.

In case we have many different variants, and each variant requires a different value assigned to the parameter. This feature could help us update new value for existing parameters, it also supports to add new parameters to existing configuation object.

Example:

Suppose we have the json file `params_global.json` with the content:

```
{
  "gGlobalIntParam" : 1,

  "gGlobalFloatParam" : 1.332, // This parameter is used to configure for ....

  "gGlobalString"   : "This is a string",

  "gGlobalStructure": {
    "general": "general"
  }
}
```

Then we import `params_global.json` to json file `config.json` with content:

```
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  // Version control information.
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "1"
  },
  "params": {
    // Global parameters
    "global": {
       "[import]": "<path_to_the_imported_file>/params_global.json"
      }
    },
  "TargetName" : "gen3flex@dlt",
  // Overwrite parameters
  "${params}['global']['gGlobalFloatParam']": 9.999,
  "${version}['patchversion']": "2",
```

```
  "${params}['global']['gGlobalString']": "This is the new value for the already
↪  existing parameter.",
  // Add new parameters
  "${newParam}": {
              "abc": 9,
              "xyz": "new param"
  },
  "${params}['global']['gGlobalStructure']['newGlobalParam']": 123
}
```

The `config.json` file is handled by JsonPreprocessor package, then return the dictionary object for a program like below:

```
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "2"
  },
  "params": {
    "global": {
      "gGlobalIntParam" : 1,
      "gGlobalFloatParam" : 9.999,
      "gGlobalString"   : "This is the new value for the already existing
↪  parameter.",
      "gGlobalStructure": {
        "general": "general",
       "newGlobalParam": 123
        }
      }
  },
  "TargetName": "gen3flex@dlt",
  "newParam": {
     "abc": 9,
     "xyz": "new param"
  }
}
```

2.3.4 Uses nested parameter

With JsonPreprocessor package, user can also use nested parameters as example below:

Example:

Suppose we have the json file `config.json` with the content:

```
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  // Version control information.
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "1"
  },
  "params": {
    // Global parameters
    "global": {
      "gGlobalIntParam" : 1,
```

```
      "gGlobalFloatParam" : 1.332, // This parameter is used to configure for ....
      "gGlobalString"   : "This is a string",
      "gGlobalStructure": {
        "general": "general"
        }
    }
  },
  "preprocessor": {
    "definitions": {
      "gPreprolIntParam" : 1,
      "gPreproFloatParam" : 9.664,
     "ABC": "checkABC",
      "gPreproString"   : "This is a string",
      "gPreproStructure": {
                           "general": "general"
                          }
    }
  },
  "TargetName" : "gen3flex@dlt",
  // Nested parameter
  "${params}['global'][${preprocessor}['definitions']['ABC']]": true,
  "${params}['global']['gGlobalFloatParam']":
↪  ${preprocessor}['definitions']['gPreproFloatParam']
}
```

The `config.json` file is handled by JsonPreprocessor package, then return the dictionary object for a program like below:

```
{
  "Project": "G3g",
  "WelcomeString": "Hello... ROBFW is running now!",
  "version": {
    "majorversion": "0",
    "minorversion": "1",
    "patchversion": "1"
  },
  "params": {
    "global": {
      "gGlobalIntParam" : 1,
      "gGlobalFloatParam" : 9.664,
      "gGlobalString"   : "This is a string",
      "gGlobalStructure": {
        "general": "general"
        },
     "checkABC": true
    }
  },
  "preprocessor": {
    "definitions": {
      "gPreprolIntParam" : 1,
      "gPreproFloatParam" : 9.664,
     "ABC": "checkABC",
      "gPreproString"   : "This is a string",
      "gPreproStructure": {
                           "general": "general"
                          }
    }
  },
  "TargetName" : "gen3flex@dlt"
}
```

2.3.5 Other features

To facilitate the json usage, the Python data types such as **“True“**, **“False“**, and **“None“** will be accepted as json syntax elements while using JsonPreprocessor package.

Chapter 3

CJsonPreprocessor.py

3.1 Class: CSyntaxType

Imported by:

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

3.2 Class: CPythonJSONDecoder

Imported by:

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

Class: PythonJSONDecoder

Add python data types and syntax to json. `True`, `False` and `None` will be a accepted as json syntax elements.

Args:

json.JSONDecoder (*object*)

Decoder object provided by `json.loads`

3.2.1 Method: custom_scan_once

3.3 Class: CJsonPreprocessor

Imported by:

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

Class: CJsonPreprocessor

CJsonPreprocessor extends the syntax of json.

Features are

- Allow c/c++-style comments within json files.
  // single line or part of single line and /* */ multline comments are possible
- Allow to import json files into json files
  `"[import]" : "relative/absolute path"`, imports another json file to exactly this location.
  `%envariable%` and `${envariable}` can be used, too.

- Allow use of variables within json files
  In any place the syntax ${basenode.subnode. ... nodename} allows to reference an already existing variable.
  - Example:

```
{
    "basenode" : {
                    subnode : {
                                    "myparam" : 5
                                },

                },

    "myVar" : "${basenode.subnode.myparam}"
}
```

- Allow python data types `True`, `False` and `None`

3.3.1 Method: jsonLoad

Method: jsonLoad

This function is the entry point of JsonPreprocessor.

It loads the json file, preprocesses it and returns the preprocessed result as data structure.

Args:

jFile (*string*)

Relative/absolute path to main json file.
`%envvariable%` and `${envvariable}` can be used, too in order to access environment variables.

Returns:

oJson (*dict*)

Preprocessed json file(s) as data structure

Chapter 4

Appendix

About this package:

Table 4.1: Package setup

Setup parameter	Value
Name	JsonPreprocessor
Version	0.1.4
Date	14.09.2022
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 5

History

0.1.0	01/2022
Initial version	
0.1.4	09/2022
Updated documentation	