

GenPackageDoc

v. 0.44.0

Holger Queckenstedt

05.06.2026

Contents

1	Introduction	1
2	Repository Structure	2
3	Documentation Build Process	3
4	Command Line	5
5	Documents Structure	7
5.1	Example: reST file	8
5.2	Example: Python module	11
6	Interface and Module Descriptions	13
7	Runtime Variables	15
8	Syntax Aspects	16
8.1	Common rules	16
8.2	Syntax extensions	17
9	Listings	20
10	Pictures and Diagrams	26
10.1	Pictures Import	26
10.2	Diagrams Rendering and Import	26
10.3	Example: Sequence diagram	27
10.4	Example: JSON diagram	28
11	CDocBuilder.py	29
11.1	genpackagedoc-cdocbuilder-cdocbuilder	29
11.1.1	genpackagedoc-cdocbuilder-cdocbuilder-build	29
12	CInterface.py	30
12.1	genpackagedoc-cinterface-cinterface	30
12.1.1	genpackagedoc-cinterface-cinterface-getlatexstyles	30
13	CPackageDocConfig.py	31
13.1	genpackagedoc-cpackagedocconfig-cpackagedocconfig	31
13.1.1	genpackagedoc-cpackagedocconfig-cpackagedocconfig-printconfig	31
13.1.2	genpackagedoc-cpackagedocconfig-cpackagedocconfig-printconfigkeys	31
13.1.3	genpackagedoc-cpackagedocconfig-cpackagedocconfig-get	31

13.1.4	genpackagedoc-cpackagedocconfig-cpackagedocconfig-getconfig	31
14	CPatterns.py	32
14.1	genpackagedoc-cpatterns-cpatterns	32
14.1.1	genpackagedoc-cpatterns-cpatterns-getheader	32
14.1.2	genpackagedoc-cpatterns-cpatterns-getchapter	32
14.1.3	genpackagedoc-cpatterns-cpatterns-getfooter	33
14.1.4	genpackagedoc-cpatterns-cpatterns-getautodefinedheader	33
15	CSourceParser.py	34
15.1	genpackagedoc-csourceparser-csourceparser	34
15.1.1	genpackagedoc-csourceparser-csourceparser-parsesourcefile	34
16	Appendix	35
17	History	36

Chapter 1

Introduction

What is the meaning of this Python package?

The Python package **GenPackageDoc** generates the documentation of Python modules. The content of this documentation is taken out of the docstrings of functions, classes and their methods. All docstrings have to be written in reStructuredText (reST) format, that is a certain markdown dialect.

It is possible to extend the documentation by the content of additional files either in reST format or in LaTeX format. The documentation is generated in the following way:

1. Files in LaTeX format are taken over immediately.
2. Files in reST format are converted to LaTeX files and to HTML files.
3. All docstrings of all Python modules in the package are converted to LaTeX files and to HTML files.
4. All LaTeX files together are converted to a single PDF document. This requires a separately installed LaTeX distribution (recommended: TeX Live). A LaTeX distribution is **not** part of **GenPackageDoc** and has to be installed separately!
5. All HTML files are collected in a separate folder together with a `index.html` file as entry point.

Caution: LaTeX files are not converted to HTML files! Therefore, the content of LaTeX files are not part of the Documentation in html format.

The sources of **GenPackageDoc** are available in the following GitHub repository:

[python-genpackagedoc](#)

The repository **python-genpackagedoc** uses its own functionality to document itself and the contained Python package **GenPackageDoc**.

Therefore, the complete repository can be used as an example about writing a package documentation.

It has to be considered, that the main goal of **GenPackageDoc** is to provide a toolchain to generate documentation out of Python sources that are stored within a repository, and therefore we have dependencies to the structure of the repository. For example: Configuration files with values that are specific for a repository, should not be installed. Such a specific configuration value is e.g. the name of the package or the name of the PDF document.

The impact is: There is a deep relationship between the repository containing the sources to be documented, and the sources and the configuration of **GenPackageDoc** itself. Therefore some manual preparations are necessary to use **GenPackageDoc** also in other repositories.

How to do this is explained in detail in the next chapters.

The outcome of all preparations of **GenPackageDoc** in your own repository is a document like the one you are currently reading.

Chapter 2

Repository Structure

What is the structure of the application repository?

- Folder `GenPackageDoc`
Contains the package code.
This folder is specific for the package.
- Folder `packagedoc`
Contains all package documentation related files, e.g. the **GenPackageDoc** configuration, additional input files and the generated documentation itself.
This folder is specific for the documentation.
- Repository root folder
 - `genpackagedoc.py`
Python script to start the documentation build
 - `build_backend.py`
Custom build backend to execute additional installation steps. Currently this is a pattern only reserved for future development.
 - `pyproject.toml`
Main configuration file for the installation of the component
 - `dump_repository_config.py`
Little helper to dump the repository configuration to console

Chapter 3

Documentation Build Process

How do the files and folders listed above relate to each other? What is the flow of information when the documentation is generated?

- The process starts with the execution of `genpackagedoc.py` within the repository root folder.
- `genpackagedoc.py` creates a repository configuration object

```
config/CRepositoryConfig.py
```

- The repository configuration object reads the static repository configuration values out of the TOML file

```
pyproject.toml
```

Exception: The component version is taken from the Python source code

```
GenPackageDoc/version.py
```

- The repository configuration object adds dynamic values (like operating system specific settings and paths) to the repository configuration.
- The configuration file `packagedoc_config.json` contains settings like
 - Paths to Python packages to be documented
 - Paths and names of additional reST files
 - Path and name of output folder (LaTeX files and output PDF file)
 - User defined parameter (that can be defined here as global runtime variables and can be used in any reST code)
 - Basic settings related to the output PDF file (like document name, name of author, ...)
 - Path to LaTeX compiler/ (*a LaTeX distribution is not part of **GenPackageDoc***)

Be aware of that the within `packagedoc_config.json` specified output folder

```
"OUTPUT" : "./build"
```

will be deleted at the beginning of the documentation build process! Make sure that you do not have any files inside this folder opened when you start the process. In case of the path is relative, the reference is the position of `genpackagedoc.py`. The complete path is created recursively.

Further details are explained within the json file itself.

- `genpackagedoc.py` also creates an own configuration object

```
GenPackageDoc/CPackageDocConfig.py
```

`CPackageDocConfig.py` takes over all repository configuration values, reads in the static **GenPackageDoc** configuration (`packagedoc_config.json`) and adds dynamically computed values like the full absolute paths belonging to the documentation build process. Also all command line parameters are resolved and checked.

The reference for all relative paths is the position of `genpackagedoc.py` (that is the repository root folder).

After the execution of `genpackagedoc.py` the resulting PDF document can be found under the specified name within the specified output folder (`"OUTPUT"`). This folder also contains all temporary files generated during the documentation build process.

Because the output folder is a temporary one, the PDF document is copied to the folder containing the package sources and therefore is included in the package installation. This is defined in the **GenPackageDoc** configuration, section `"PDFDEST"`.

Chapter 4

Command Line

Which command line parameters are available to influence the behavior of the application?

Some configuration parameter predefined within `packagedoc_config.json`, can be overwritten in command line.

`--output`

Path and name of folder containing all output files.

Caution: GenPackageDoc deletes this folder!

`--pdfdest`

Path and name of folder in which the generated PDF file will be copied to (after this file has been created within the output folder).

Caution: The generated PDF file will per default be copied to the package folder within the repository. This is defined in `packagedoc_config.json`. The version of the PDF file within the package folder will be part of the installation. When you change the PDF destination, then you get this file at another location - but this file will not be part of the installation any more. Installed will be the version, that is still present within the package folder of the repository.

`--htmldest`

Path and name of folder in which the generated HTML files will be copied to (after this file has been created within the output folder).

Caution: GenPackageDoc deletes this folder!

`--configdest`

Path and name of folder in which a dump of the current configuration will be copied to.

The configuration dump is part of the build output (section `'OUTPUT'`) and available in txt and in json format. It might be useful for further processes to have access to all details regarding the current documentation build.

`--strict`

If `True`, a missing LaTeX compiler aborts the process, otherwise the process continues.

```
--simulateonly
```

If `True`, the LaTeX compiler is switched off. No new PDF output will be generated. Already existing PDF output will not be updated. This is not handled as error and also not handled as warning. Only the source files will be parsed. This switch is useful to do a pre check for possible syntax issues within the source files without spending time for rendering PDF files.

Example

```
genpackagedoc.py --output="../any/other/location" --pdfdest="../any/other/location" --↵  
↵ htmldest="../any/other/location" --configdest="../any/other/location" --strict=True
```

Chapter 5

Documents Structure

How are the generated documents structured? What causes an entry within the table of content and how does the table of content look like?

In the following we use terms taken over from the LaTeX world: *chapter*, *section* and *subsection*.

A *chapter* is the top level within the PDF document; a *section* is the level below *chapter*, a *subsection* is the level below *section*.

The following assignments happen during the generation of a PDF document:

- The content of every additionally included separate reST file is a *chapter*.
 - In case of you want to add another chapter to your documentation, you have to include another reST file.
 - The headline of the chapter is the name of the reST file (automatically).
Therefore, it is not necessary to repeat the headline inside the file.
- The content of every included Python module is also a *chapter*.
 - The headline of the chapter is the name of the Python module (automatically).
This means also that within the PDF document structure every Python module is at the same level as additionally included reST files.
- Within additionally included separate reST files sections and subsections can be defined by the usual reST syntax elements for headings:
 - A line underlined with " `=` "
 - A line underlined with " `-` "
- Within the docstrings of Python modules the headings are added automatically (for functions, classes and methods)
 - Classes and functions are listed at section level (both classes and functions are assumed to be at the same level).
 - Class methods are listed at subsection level.

Further nestings of headings are not supported (because we do not want to overload the table of content).

5.1 Example: reST file

This can be the content of a file `Example Chapter.rst` in reST format:

```

.. highlight::

   This is an example chapter to demonstrate the structure of a PDF document.

Example section 1
=====

*text text text text text text text text text text text text text text*

Example subsection 1.1
-----

*text text text text text text text text text text text text text text*

Example subsection 1.2
-----

*text text text text text text text text text text text text text text*

Example section 2
=====

*text text text text text text text text text text text text text text*

Example subsection 2.1
-----

*text text text text text text text text text text text text text text*

```

In PDF this causes an entry in the table of content:

2	Example Chapter	11
2.1	Example section 1	11
2.1.1	Example subsection 1.1	11
2.1.2	Example subsection 1.2	11
2.2	Example section 2	11
2.2.1	Example subsection 2.1	11

And this is the content itself:

Chapter 2

Example Chapter

This is an example chapter to demonstrate the structure of a PDF document.

2.1 Example section 1

text text text text text text text text text text text text text

2.1.1 Example subsection 1.1

text text text text text text text text text text text text text

2.1.2 Example subsection 1.2

text text text text text text text text text text text text text

2.2 Example section 2

text text text text text text text text text text text text text

2.2.1 Example subsection 2.1

text text text text text text text text text text text text text

The table of contents of the documentation in HTML format is currently limited to the top level (*chapters*):

GenPackageDoc	<i>This is an example chapter to demonstrate the structure of a PDF document.</i>
Examples	
Example Chapter	
Introduction	<i>text text text text text text text text text text text text text text text</i>
Repository Structure	Example subsection 1.1
Documentation Build Process	<i>text text text text text text text text text text text text text text text</i>
Command Line	Example subsection 1.2
Documents Structure	<i>text text text text text text text text text text text text text text text</i>
Interface and Module Descriptions	Example section 2
Runtime Variables	<i>text text text text text text text text text text text text text text text</i>
Syntax Aspects	Example subsection 2.1
Listings	<i>text text text text text text text text text text text text text text text</i>

5.2 Example: Python module

This can be the content of some docstrings within a Python module `example_module.py`:

```

"""
.. highlight::

   The example module is a Python module containing example docstrings
"""

def example_function():
    """
    :acontent:`example_function` is a simple example function.
    """

class CExample():
    """
    :acontent:`CExample` is the main class of the example module.
    """

    def example_method_1(self):
        """
        This is the example method :acontent:`example_method_1` of the example module :fssystem:`↔
        ↪ example_module.py`.
        """
        pass

    def example_method_2(self):
        """
        This is the example method :acontent:`example_method_2` of the example module :fssystem:`↔
        ↪ example_module.py`.
        """
        pass

```

In PDF this causes an entry in the table of content:

18	example_module.py	42
18.1	Function: example_function	42
18.2	Class: CExample	42
18.2.1	Method: example_method_1	42
18.2.2	Method: example_method_2	42

And this is the content itself:

Chapter 18

example_module.py

The example module is a Python module containing example docstrings

18.1 Function: example_function

`example_function` is a simple example function.

18.2 Class: CExample

`CExample` is the main class of the example module.

18.2.1 Method: example_method_1

This is the example method `example_method_1` of the example module `example_module.py`.

18.2.2 Method: example_method_2

This is the example method `example_method_2` of the example module `example_module.py`.

The name of the Python module is placed at *chapter* level. The names of functions and classes are placed at *section* level. The names of class methods are placed at *subsection* level.

Caution: Headings within docstrings of Python module are not supported! Nested function definitions are skipped also.

The table of contents of the documentation in HTML format is currently limited to the top level (that is the name of the Python module). The functions classes and methods are not yet included in the HTML table of content.

Chapter 6

Interface and Module Descriptions

How to describe an interface of a function or a method? How to describe a Python module?

To have a unique look and feel of all interface descriptions, the following style is recommended:

Example

```

def ExampleInterface(self):
    """
Description of ExampleInterface

**Arguments:**

* :acontent:`arg1`

  / *Condition*: mandatory / *Type*: :acontent:`int` /

  arg1 description

* :acontent:`arg2`

  / *Condition*: optional / *Type*: :acontent:`bool` / *Default*: :acontent:`True` /

  arg2 description

**Returns:**

* :acontent:`ack`

  / *Type*: :acontent:`bool` /

  Indicates if the computation was successful or not.

* :acontent:`result`

  / *Type*: :acontent:`str` /

  Returned value
    """

```

Within the rendered documents this interface description looks like this:

7.3.2 Method: ExampleInterface

Description of ExampleInterface

Arguments:

- `arg1`
/ *Condition*: mandatory / *Type*: `int` /
arg1 description
- `arg2`
/ *Condition*: optional / *Type*: `bool` / *Default*: `True` /
arg2 description

Returns:

- `ack`
/ *Type*: `bool` /
Indicates if the computation was successful or not.
- `result`
/ *Type*: `str` /
Returned value

The docstrings containing the description, have to be placed directly in the next line after the `def` or `class` statement.

It is also possible to place a docstring at the top of a Python module. The exact position doesn't matter - but it has to be the first constant expression within the code. Within the documentation the content of this docstring is placed before the interface description and should contain general information belonging to the entire module.

The usage of such a docstring is an option.

Chapter 7

Runtime Variables

What are "runtime variables" and how to use them in reST content?

All configuration parameters of **GenPackageDoc** are taken out of four sources:

1. the static repository configuration

```
pyproject.toml
```

2. the dynamic repository configuration

```
config/CRepositoryConfig.py
```

3. the static **GenPackageDoc** configuration

```
packagedoc/packagedoc_config.json
```

4. the dynamic **GenPackageDoc** configuration

```
GenPackageDoc/CPackageDocConfig.py
```

Some of them are runtime variables and can be accessed within reST content (within docstrings of Python modules and also within separate reST files).

This means it is possible to add configuration values automatically to the documentation.

This happens by encapsulating the runtime variable name in triple hashes. This "triple hash" syntax is introduced to make it easier to distinguish between the json syntax (mostly based on curly brackets) and additional syntax elements used within values of json keys.

The name of the repository e.g. can be added to the documentation with the following reST content:

```
The name of the repository is ###REPOSITORYNAME###.
```

This document contains a chapter "Appendix" at the end. This chapter is used to make the repository configuration a part of this documentation and can be used as example.

Additionally to the predefined runtime variables a user can add own ones.

See **"PARAMS"** within `packagedoc_config.json`.

All predefined runtime variables are written in capital letters. To make it easier for a developer to distinguish between predefined and user defined runtime variables, all user defined runtime variables have to be written in small letters completely.

Also the **"DOCUMENT"** keys within `packagedoc_config.json` are runtime variables.

Also within `packagedoc_config.json` the triple hash syntax can be used to access repository configuration values.

With this mechanism it is e.g. possible to give the output PDF document automatically the name of the package:

```
"DOCUMENT" : {
    "OUTPUTFILENAME" : "###PACKAGENAME###.tex",
```

Chapter 8

Syntax Aspects

What specific syntax rules must be observed when writing separate documents and the docstrings of Python modules in reST format?

8.1 Common rules

Important to know about the syntax of Python and reST is:

- In both Python and reST the indentation of text is part of the syntax!
- The indentation of the triple quotes indicating the beginning and the end of a docstring has to follow the Python syntax rules.
- The indentation of the content of the docstring (= the interface description in reST format) has to follow the reST syntax rules. To avoid a needless indentation of the text within the resulting PDF document and to avoid further unwanted side effects caused by improper indentations, it is strongly required to start at least the first line of a docstring text within the first column! And this first line is the reference for the indentation of further lines of the current docstring. The indentation of these further lines depends on the reST syntax element that is used here.
- In reSTreST also blank lines are part of the syntax!

Why is a proper indentation of the docstrings so much important?

The contents of all docstrings of a Python module will be merged to one single reST document (internally by **GenPackageDoc**). In this single reST document we do not have separated docstring lines any more. We have one text! And we have a relationship between previous lines and following lines in this text. The indentation of these previous and following lines must fit together – accordingly to the reST syntax rules. Otherwise we either get syntax issues during computation or we get text with a layout that does not fit to our expectation.

8.2 Syntax extensions

Like mentioned above, **GenPackageDoc** converts the documentation content from reST format to LaTeX format and to HTML format. Every format defines its own special characters as part of the syntax. For example: *underscores* in reST, *backslashes* in LaTeX and horizontal and vertical distances in HTML.

It is not possible, with acceptable effort, to use all special characters directly as literals in reST in such a way that they are displayed as literals in both LaTeX and HTML. The differences between the involved formats and the resulting requirements for escaping are simply too great. To enable the use of these special characters nonetheless, **GenPackageDoc** provides its own syntax elements. This allows **GenPackageDoc** to handle these special cases specifically for each output format.

Every syntax element starts with a slash, followed by an abbreviation in capital letters.

The following syntax elements are available:

RST	LaTeX/PDF	HTML
/NL (<i>newline</i>)	\newline	
/NP (<i>newpage</i>)	\newpage	""
/VS (<i>vertical space 1</i>)	\vspace{1ex}	display:block; height:1ex;
/VVS (<i>vertical space 2</i>)	\vspace{2ex}	display:block; height:2ex;
/VVVS (<i>vertical space 3</i>)	\vspace{3ex}	display:block; height:3ex;
/BS (<i>backslash</i>)	\textbackslash{}	\\
/IBS (<i>inline backslash</i>)	\\	\\
/HS (<i>horizontal space</i>)	{\ttfamily\hspace{0.6em}}	
/IHS (<i>inline horizontal space</i>)	{\ttfamily\hspace{0.6em}}	
/US (<i>underscore</i>)	_	_

The following example demonstrates how the syntax extensions work:

```

**Distances:**

This Robot Framework code prints a string to log: :acontent:`log Hello`.
This Robot Framework code prints another string to log: :acontent:`log World`.

This Robot Framework code prints a string to log: :acontent:`log/IHS/IHS/IHS/IHSHello`./NL
This Robot Framework code prints another string to log: :acontent:`log/IHS/IHS/IHS/IHSWorld`./VVVS

Some distances between strings/HS/HSstrings/HS/HS/HS/HSstrings/VS
and lines/VS
and lines/VVS
and lines/VVVS
and lines

```

Output in HTML format:

Distances:

This Robot Framework code prints a string to log: `log Hello`. This Robot Framework code prints another string to log: `log World`.

This Robot Framework code prints a string to log: `log Hello`.
This Robot Framework code prints another string to log: `log World`.

Some distances between strings strings strings

and lines

and lines

and lines

and lines

Insights:

- A line break in editor does not cause a line break in output files.
- A line break in output files must be set explicitly by `/NL` (= *newline*) at the end of a line.
- Without additional measures, multiple spaces in standard text and in inline literals are each reduced to a single space.
- A space that is to be preserved is defined by `/HS` (= *horizontal space*) in standard text and by `/IHS` (= *inline horizontal space*) in inline literals.
- Additional to the possibility to use blank lines for vertical distances in standard text, several *vertical space* elements (`/VS`, `/VVS`, `/VVVS`) are available to define vertical distances in higher granularity.
- User defined vertical distances are not possible in inline literals (and makes no sense here).

Recommendations:

Like shown in the example above, the usage of syntax elements should follow these rules:

- Prefer to put the syntax element for a line break at the end of the line - and combine this with a line break of the text in editor. This eases the readability of the text.
- Prefer to put syntax elements for vertical distances at the end of a line.
- Prefer to put syntax elements for horizontal distances immediately between strings inside a line.

Reasons:

This would increase the vertical distance between lines caused by additional blank lines:

```
.. anycontent::

    text text text text

    /VS

    text text text text
```

This would increase the horizontal distance between two strings within a line by additional blanks:

```
.. anycontent::  
  
   text /HS text
```

This style should be avoided (but is for sure not forbidden).

In block literals (e.g. like `anycontent`), the extended syntax is not allowed - and not necessary. The content is printed out **as is** (*without any interpretation*).

Hint: It is good practise to put a `/NP` at the end of each section. Every following section will start at a new page in PDF output (in HTML output, `/NP` has no effect).

Chapter 9

Listings

How to realize listings of code written in Python, Robot Framework and JSON? How to realize listings of console content and log file content?

GenPackageDoc supports several code listings and console listings. These listings are available as text block and as inline text also. You can use these listings in reST files and also in LaTeX files immediately. The reST elements for listings are mapped to the corresponding LaTeX commands when **GenPackageDoc** internally converts the reST files in LaTeX files.

Text blocks are realized by reST directives. Inline text is realized by reST roles.

Examples:

This is a reST directive `pythoncode` code example:

```
for index in list:
    print("index")
```

Written in reST format:

```
.. pythoncode::

    for index in list:
        print("index")
```

Written in LaTeX format:

```
\begin{pythoncode}
  for index in list:
    print("index")
\end{pythoncode}
```

★

This is a reST directive `robotcode` code example:

```
*** Test Cases ***
Example

FOR     ${index}     IN RANGE     0     ${max}
  log    index: ${index}     console=yes
END
```

Written in reST format:

```
.. robotcode::

    *** Test Cases ***

    Example

    FOR     ${index}    IN RANGE    0    ${max}
        log     index: ${index}    console=yes
    END
```

Written in LaTeX format:

```
\begin{robotcode}
    *** Test Cases ***

    Example

    FOR     ${index}    IN RANGE    0    ${max}
        log     index: ${index}    console=yes
    END
\end{robotcode}
```

★

This is a reST directive `jsoncode` code example:

```
{
    "param_1" : "ABC",
    "param_2" : true,
    "param_3" : [1,2,3],
    "param_4" : {"A" : 1, "B" : 2}
}
```

Written in reST format:

```
.. jsoncode::

    {
        "param_1" : "ABC",
        "param_2" : true,
        "param_3" : [1,2,3],
        "param_4" : {"A" : 1, "B" : 2}
    }
```

Written in LaTeX format:

```
\begin{jsoncode}
    {
        "param_1" : "ABC",
        "param_2" : true,
        "param_3" : [1,2,3],
        "param_4" : {"A" : 1, "B" : 2}
    }
\end{jsoncode}
```

★

This is a reST directive `anycontent` code example:

```
Any content, no matter if Python, Robot Framework, JSON or something else.
Any content, no matter if Python, Robot Framework, JSON or something else.
Any content, no matter if Python, Robot Framework, JSON or something else.
```

Written in reST format:

```
.. anycontent::

    Any content, no matter if Python, Robot Framework, JSON or something else.
    Any content, no matter if Python, Robot Framework, JSON or something else.
    Any content, no matter if Python, Robot Framework, JSON or something else.
```

Written in LaTeX format:

```
\begin{anycontent}
    Any content, no matter if Python, Robot Framework, JSON or something else.
    Any content, no matter if Python, Robot Framework, JSON or something else.
    Any content, no matter if Python, Robot Framework, JSON or something else.
\end{anycontent}
```

★

This is a reST directive `consolelog` code example:

```
log index: 0
log index: 1
log index: 2
```

Written in reST format:

```
.. consolelog::

    log index: 0
    log index: 1
    log index: 2
```

Written in LaTeX format:

```
\begin{consolelog}
    log index: 0
    log index: 1
    log index: 2
\end{consolelog}
```

★

This is a reST directive `filesystem` code example:

```
C:\folder_1\file_1.txt
C:\folder_2\file_2.txt
C:\folder_3\file_3.txt
```

Written in reST format:

```
.. filesystem::

    C:\folder_1\file_1.txt
    C:\folder_2\file_2.txt
    C:\folder_3\file_3.txt
```

Written in LaTeX format:

```
\begin{filesystem}
  C:\folder_1\file_1.txt
  C:\folder_2\file_2.txt
  C:\folder_3\file_3.txt
\end{filesystem}
```

Hint: `consolelog` and `filesystem` currently have the same layout. Nevertheless, using individual and specific names eases the readability. Maybe later different layouts will be realized.

This is inline Python code: `print("Hello Python")`

Written in reST format:

```
:pcode:`print("Hello Python")`
```

Written in LaTeX format:

```
\pcode{print("Hello Python")}
```

This is inline Robot Framework code: `log Hello RobotFramework AIO`

Written in reST format:

```
:rcode:`log/IHS/IHS/IHS/IHSHello RobotFramework AIO`
```

Written in LaTeX format:

```
\rcode{log\ \ \ \ Hello RobotFramework AIO}
```

Hint: Blanks must be announced explicitly in inline code, because under normal circumstances multiple blanks are collapsed to a single blank (in LaTeX and in HTML also).

This is inline JSON code: `"param" : [1,2,3]`

Written in reST format:

```
:jcode:`"param" : [1,2,3]`
```

Written in LaTeX format:

```
\jcode{"param" : [1,2,3]}
```

This is any inline content: `any content`

Written in reST format:

```
:accontent:`any content`
```

Written in LaTeX format:

```
\accontent{any content}
```

The horizontal separator line with the star in the middle can be inserted with the reST directive `.. hrstar::`. This is a nice possibility to separate smaller parts of the documentation from each other - without introducing a new headline for every part.

When we use all of these reST syntax elements to realize a concrete documentation - how does this look like? The following is a full example:

In the system configuration file, the parameter `:acontent:`aparam`` and `:acontent:`bparam`` ↩ ↪ are defined in the following way:

```
.. jsoncode::

    {
        // system parameter
        "aparam" : 1,
        "bparam" : "hello world"
    }
```

This configuration file can be found here:

```
.. filesystem::

    ${systemroot}/packages/labels/alabel/config.jsonp
```

Make sure to select the proper label: `:acontent:`alabel``.

The **Robot Framework** keyword `:acontent:`compute_system_parameter`` computes these ↩ ↪ system parameters in the following way:

- * The parameter `:acontent:`aparam`` is increased by value `:acontent:`1``.
- * The parameter `:acontent:`bparam`` is concatenated with string `:acontent:`today`` (with a ↩ ↪ blank in between).

This is the corresponding Python implementation of the keyword `:acontent:`` ↩ ↪ `compute_system_parameter``:

```
.. pythoncode::

    system_config = system.get_config()
    system_config['aparam'] = system_config['aparam'] + 1
    system_config['bparam'] = system_config['bparam'] + " today"
    number_of_parameters = len(system_config)
    return system_config, number_of_parameters
```

Additionally to the system configuration `:acontent:`system_config``, another variable ↩ ↪ `:acontent:`number_of_parameters`` containing the number of system parameters, is returned.

Within a **Robot Framework** test you call the keyword `:acontent:`` ↩ ↪ `compute_system_parameter`` in the following way:

```
.. robotcode::

    *** Test Cases ***

    SystemTest

        ${system_config}    ${number_of_parameters}    compute_system_parameter
        log    Number of parameters: ${number_of_parameters}
        log    aparam.....: ${system_config}[aparam]
```

In the log file it can be observed that the value of parameter `:acontent:`aparam`` is ↩ ↪ `:acontent:`2`` now.

The value of parameter `:acontent:`bparam`` is `:acontent:`hello world today`` now. With 2 parameters in total.

Result:

★

In the system configuration file, the parameter `aparam` and `bparam` are defined in the following way:

```
{
  // system parameter
  "aparam" : 1,
  "bparam" : "hello world"
}
```

This configuration file can be found here:

```
${systemroot}/packages/labels/alabel/config.jsonp
```

Make sure to select the proper label: `alabel`.

The **Robot Framework** keyword `compute_system_parameter` computes these system parameters in the following way:

- The parameter `aparam` is increased by value `1`.
- The parameter `bparam` is concatenated with string `today` (with a blank in between).

This is the corresponding Python implementation of the keyword `compute_system_parameter`:

```
system_config = system.get_config()
system_config['aparam'] = system_config['aparam'] + 1
system_config['bparam'] = system_config['bparam'] + " today"
number_of_parameters = len(system_config)
return system_config, number_of_parameters
```

Additionally to the system configuration `system_config`, another variable `number_of_parameters` containing the number of system parameters, is returned.

Within a **Robot Framework** test you call the keyword `compute_system_parameter` in the following way:

```
*** Test Cases ***
SystemTest

    ${system_config}    ${number_of_parameters}    compute_system_parameter
    log    Number of parameters: ${number_of_parameters}
    log    aparam.....: ${system_config}[aparam]
```

In the log file it can be observed that the value of parameter `aparam` is `2` now. The value of parameter `bparam` is `hello world today` now. With 2 parameters in total.

★

Chapter 10

Pictures and Diagrams

How to import pictures? How to render and import diagrams?

10.1 Pictures Import

Pictures can be imported in the following way:

1. Import in reST content:

```
.. image:: ./pictures/AnyPicture.png
```

2. Import in LaTeX content:

```
\includegraphics[scale=0.7]{./pictures/AnyPicture.png}
```

The user needs to adapt the scaling to make the rendered diagrams fit to a page of a PDF document in best way. But this scaling only works in LaTeX code, but not in reST code.

10.2 Diagrams Rendering and Import

A *diagram* in this context is a picture that is rendered out of source code. **GenPackageDoc** supports **PlantUML** that supports a wide range of diagrams.

To use the **PlantUML** functionality with **GenPackageDoc**, some preconditions have to be fulfilled:

1. **PlantUML** is installed (either as stand-alone installation or as VSCodium extension)
2. **GenPackageDoc** is configured (`packagedoc_config.json`):
 - a. In section `"DIAGRAMS"` a path to a diagrams folder is defined (containing the diagrams source code).
 - b. In section `"JAVA"` path and name of the java interpreter is defined (because **PlantUML** is a java application).
 - c. In section `"PLANT_UML"` path and name of the **PlantUML** application is defined.

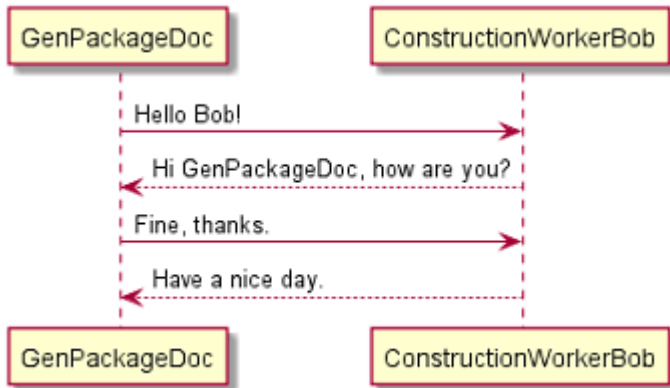
All **PlantUML** source code files within the `"DIAGRAMS"` folder need to have the extension `puml`.

10.3 Example: Sequence diagram

The following code of a `puml` file produces a sequence diagram:

```
@startuml
GenPackageDoc -> ConstructionWorkerBob: Hello Bob!
ConstructionWorkerBob --> GenPackageDoc: Hi GenPackageDoc, how are you?
GenPackageDoc -> ConstructionWorkerBob: Fine, thanks.
ConstructionWorkerBob --> GenPackageDoc: Have a nice day.
@enduml
```

Result:



10.4 Example: JSON diagram

The following code of a `puml` file produces a sequence diagram:

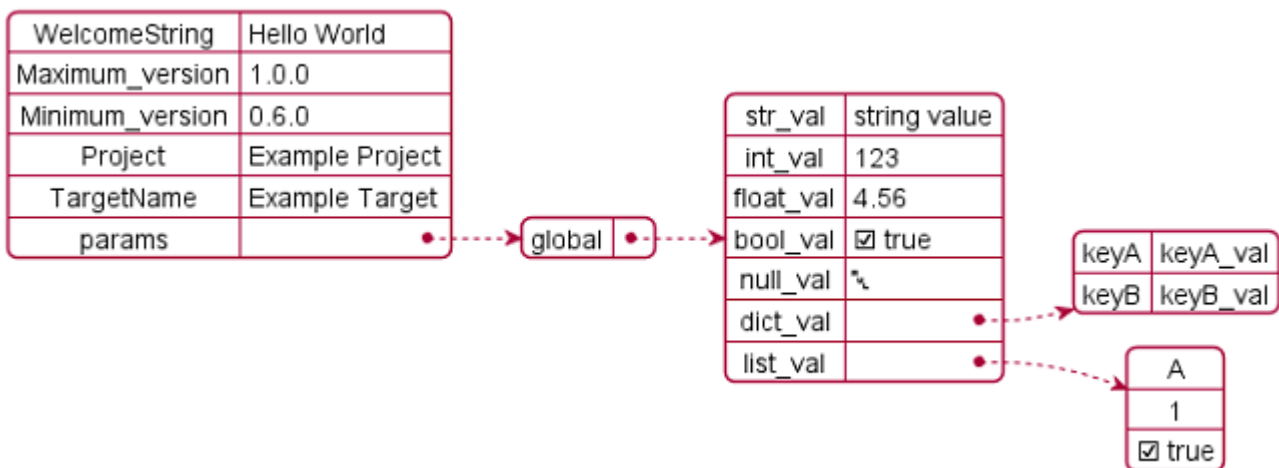
```
@startjson
{
  "WelcomeString" : "Hello World",

  "Maximum_version" : "1.0.0",
  "Minimum_version" : "0.6.0",

  "Project" : "Example Project",
  "TargetName" : "Example Target",

  "params" : {
    "global" : {
      "str_val" : "string value",
      "int_val" : 123,
      "float_val" : 4.56,
      "bool_val" : true,
      "null_val" : null,
      "dict_val" : {"keyA" : "keyA_val", "keyB" : "keyB_val"},
      "list_val" : ["A", 1, true]
    }
  }
}
@endjson
```

Result:



In case of **PlantUML** is configured in the **GenPackageDoc** configuration and in case of `puml` files are available within the diagrams folder, **GenPackageDoc** automatically calls **PlantUML** to render the diagrams. They can be imported in the following way:

1. Import in reST content:

```
.. image:: ./diagrams/SequenceDiagram.png
```

2. Import in LaTeX content:

```
\includegraphics[scale=0.7]{./diagrams/SequenceDiagram.png}
```

Chapter 11

CDocBuilder.py

CDocBuilder is a Python module containing all methods to generate files in TEX and HTML format.

11.1 genpackagedoc-cdocbuilder-cdocbuilder

`CDocBuilder` is the main class to build TEX sources out of docstrings of Python modules and separate text files in reST format.

This depends on a JSON configuration file, provided by a `oPackageDocConfig` object (that includes the Repository configuration).

Method to execute: `Build()`

11.1.1 genpackagedoc-cdocbuilder-cdocbuilder-build

Arguments:

(no arguments)

Returns:

- `bSuccess`
/ *Type*: bool /
Indicates if the computation of the method `sMethod` was successful or not.
- `sResult`
/ *Type*: str /
The result of the computation of the method `sMethod`.

Chapter 12

CInterface.py

Python module containing an interface for **GenPackageDoc**. This interface can be used to get access to the LaTeX stylesheets that are part of the **GenPackageDoc** installation.

12.1 genpackagedoc-cinterface-cinterface

12.1.1 genpackagedoc-cinterface-cinterface-getlatexstyles

The LaTeX stylesheets are part of the installation of **GenPackageDoc**. In case of anyone else than **GenPackageDoc** needs these stylesheets, this method can be used to copy them to any other folder.

Arguments:

- `sDestination`
/ *Condition*: required / *Type*: str /
Path and name of a folder in which the styles folder from **GenPackageDoc** will be copied.

Returns:

- `bSuccess`
/ *Type*: bool /
Indicates if the computation of the method `sMethod` was successful or not.
- `sResult`
/ *Type*: str /
The result of the computation of the method `sMethod`.

Chapter 13

CPackageDocConfig.py

Python module containing the configuration for **GenPackageDoc**. This includes the repository configuration and command line values.

13.1 genpackagedoc-cpackagedocconfig-cpackagedocconfig

13.1.1 genpackagedoc-cpackagedocconfig-cpackagedocconfig-printconfig

Prints all configuration values to console.

13.1.2 genpackagedoc-cpackagedocconfig-cpackagedocconfig-printconfigkeys

Prints all configuration key names to console.

13.1.3 genpackagedoc-cpackagedocconfig-cpackagedocconfig-get

Returns the configuration value belonging to a key name.

13.1.4 genpackagedoc-cpackagedocconfig-cpackagedocconfig-getconfig

Returns the complete configuration dictionary.

Chapter 14

CPatterns.py

Python module containing source patterns used to generate the tex file output.

14.1 genpackagedoc-cpatterns-cpatterns

The CPatterns class provides a set of LaTeX source patterns used to generate the tex file output.

All source patterns are accessible by corresponding Get methods. Some source patterns contain placeholder that will be replaced by input parameter of the Get method.

14.1.1 genpackagedoc-cpatterns-cpatterns-getheader

Defines the header of the main tex file.

Arguments:

- sTitle
/ *Condition*: required / *Type*: str /
The title of the output document (name of the described package)
- sVersion
/ *Condition*: required / *Type*: str /
The version of the output document (version of the described package)
- sAuthor
/ *Condition*: required / *Type*: str /
The author of the output document (author of the described package)
- sDate
/ *Condition*: required / *Type*: str /
The date of the output document (date of the described package)

Returns:

- sHeader
/ *Type*: str /
LaTeX code containing the header of main tex file.

14.1.2 genpackagedoc-cpatterns-cpatterns-getchapter

Defines single chapter of the main tex file.

A single chapter is equivalent to an additionally imported text file in rst format or equivalent to a single Python module within a Python package.

Arguments:

- `sHeadline`
/ *Condition*: required / *Type*: str /
The chapter headline (that is either the name of an additional rst file or the name of a Python module).
- `sLabel`
/ *Condition*: required / *Type*: str /
The chapter label (to enable linking to this chapter)
- `sDocumentName`
/ *Condition*: required / *Type*: str /
The name of a single tex file containing the chapter content. This file is imported in the main text file after the chapter headline that is set by `sHeadline`.

Returns:

- `sHeader`
/ *Type*: str /
LaTeX code containing the headline and the input of a single tex file.

14.1.3 genpackagedoc-cpatterns-cpatterns-getfooter

Defines the footer of the main tex file.

Arguments:

(no arguments)

Returns:

- `sFooter`
/ *Type*: str /
LaTeX code containing the footer of the main tex file.

14.1.4 genpackagedoc-cpatterns-cpatterns-getautodefinedheader

Defines the header of the autodefined LaTeX sty file.

Arguments:

(no arguments)

Returns:

- `sAutodefinedHeader`
/ *Type*: str /
LaTeX code containing the header of the autodefined LaTeX sty file.

Chapter 15

CSourceParser.py

Python module containing all methods to parse the documentation content of Python source files.

15.1 genpackagedoc-csourceparser-csourceparser

The CSourceParser class provides a method to parse the functions, classes and their methods together with the corresponding docstrings out of Python modules. The docstrings have to be written in rst syntax.

15.1.1 genpackagedoc-csourceparser-csourceparser-parsesourcefile

The method ParseSourceFile parses the content of a Python module.

Arguments:

- `source_file`
/ *Condition*: required / *Type*: str /
Path and name of a single Python module.
- `include_private`
/ *Condition*: optional / *Type*: bool / *Default*: False /
If False: private methods are skipped, otherwise they are included in documentation.
- `include_undocumented`
/ *Condition*: optional / *Type*: bool / *Default*: True /
If True: also classes and methods without docstring are listed in the documentation (together with a hint that information is not available), otherwise they are skipped.

Returns:

- `dictContent`
/ *Type*: dict /
A dictionary containing all the information parsed out of `source_file`.
- `bSuccess`
/ *Type*: bool /
Indicates if the computation of the method `sMethod` was successful or not.
- `sResult`
/ *Type*: str /
The result of the computation of the method `sMethod`.

Chapter 16

Appendix

About GenPackageDoc:

Author

Holger Queckenstedt (Holger.Queckenstedt@de.bosch.com)

Version

0.44.0 (05.06.2026)

Short Description

Documentation builder for Python packages

Homepage

[python-genpackagedoc](#)

Documentation

[Readme](#)
[Main Documentation](#)

Sources

[Repository](#)

Issues

[Issues](#)

Python required

≥ 3.11

License

Apache-2.0

Chapter 17

History

Version 0.44.0 (05.06.2026)

Added:

- Search function
- GenPackageDoc decorator
- Tags included in PDF and HTML output

Maintenance:

- LaTeX preamble (`xcolor` fix)
- Robot Framework decorator parsing

Version 0.43.0 (18.05.2026)

Maintenance:

- Usage of **Setuptools** replaced by usage of **PIP/TOML/Setuptools**
- History reworked (youngest version on top now) and migrated from LaTeX format to RST format

Added:

- Possibility to exclude interface files from computation
- Markdown directives to support Python and **Robot Framework** code listings
- Markdown directives to support JSONP code listings
- Markdown directives to support common console and log listings
- Markdown directives to support listings of file system contents
- Markdown directives to support a simple highlighting in gray color
- Syntax extensions for vertical and horizontal distances
- Support of **async** functions and methods
- Output of documentation in HTML-Format

Version 0.42.0 (12.01.2026)

Maintenance:

- Usage of **pandoc/py pandoc** replaced by usage of **docutils** (conversion of RST content to LaTeX format)

Version 0.41.0 (11.07.2023)

Added:

- Text macro `\q{}` to set double quotes

Version 0.40.0 (09.05.2023)

Added:

- **PlantUML** support
- Possibility to render diagrams directly from text source code

Version 0.39.0 (06.01.2023)

Added:

- Masking of underlines in case of the content of `\repo` or `\pkg` contains underlines (masking required by LaTeX)

Version 0.38.0 (30.11.2022)

Removed:

- Harming ligatures that were added by **Pandoc** automatically to LaTeX code in case of multiple minus characters in names

Version 0.37.0 (21.11.2022)

Added:

- LaTeX commands `pythonlog` and `plog`
- keyword decorator detection

Maintenance:

- LaTeX style adaptations
- `INCLUDEPRIVATE` temporarily switched off (requires bugfixes)

Version 0.36.0 (17.11.2022)

Maintenance:

- Brightness of all colors of listings reduced to 45% (both text boxes and inline)

Version 0.35.0 (16.11.2022)

Maintenance:

- Layout settings of some LaTeX commands adapted
- Repository name and package name in bold
- Inline code and inline listings in clearer colors

Version 0.34.0 (07.11.2022)

Added:

- Auto defined LaTeX style file containing mnemotechnical commands to type the repository name and the package name

Version 0.33.0 (19.09.2022)

Maintenance:

- Rework of label mechanism (to enable unique links to functions, classes and methods with names that are not unique over all Python modules within a package)

Version 0.32.0 (16.09.2022)*Added:*

- Labels at chapter level

Maintenance:

- Partial rework of label mechanisms

Version 0.31.0 (12.09.2022)*Fixes:*

- Import path of a module in PDF file

Version 0.30.0 (31.08.2022)*Added:*

- `simulateonly` mode (command line switch to skip the PDF generation)

Version 0.29.0 (24.08.2022)*Changes:*

- Layout of import path of a module (in PDF file)

Version 0.28.0 (23.08.2022)*Added:*

- LaTeX environment variable `GENDOC_LATEXPATH`

Maintenance:

- File `robotframeworkaio.sty` aligned to version of this file in `GenMainDoc`

Version 0.27.0 (17.08.2022)*Added:*

- LaTeX style definition for Python syntax highlighting

Version 0.26.0 (27.07.2022)*Maintenance:*

- History reworked

Added:

- File `common.sty` (common LaTeX commands e.g. to support the history)

Version 0.25.0 (27.07.2022)*Maintenance:*

- Layout of `RobotFramework AIO` syntax highlighting (in `.sty` files)

Version 0.24.0 (25.07.2022)*Maintenance:*

- Line breaks in listings (in `robotframeworkaio.sty`)

Version 0.23.0 (13.07.2022)*Maintenance:*

- File `preamble.tex`

Version 0.22.0 (13.07.2022)*Maintenance:*

- File `preamble.tex`
- Folder `styles`

Fix:

- `setup.py` installs tex files also

Version 0.21.0 (12.07.2022)*Maintenance:*

- Separated file `preamble.tex`

Version 0.20.0 (29.06.2022)*Fix:*

- Added missing masking of underlines in PDF document title (required for LaTeX)

Version 0.19.0 (28.06.2022)*Added:*

- Method `GetLaTeXStyles`

Maintenance:

- `PythonExtensionsCollection` updated to version 0.8.0

Version 0.18.0 (20.06.2022)*Added:*

- Parameter to define an output folder for a dump of final configuration

Version 0.17.0 (17.06.2022)*Added:*

- Possibility to dump the configuration

Maintenance:

- Code and error handling

Version 0.16.0 (01.06.2022)

Maintenance:

- Path computation reworked

Version 0.15.0 (31.05.2022)*Added:*

- **GenPackageDoc** command line
- Separate **GenPackageDoc** configuration class

Version 0.14.0 (27.05.2022)*Added:*

- LaTeX compiler check
- Control parameter **STRICT** added to **GenPackageDoc** configuration

Version 0.13.0 (24.05.2022)*Maintenance:*

- LaTeX style definitions moved to a separate folder

Version 0.12.0 (19.05.2022)*Added:*

- Admonitions based on LaTeX environment `tcolorbox`

Maintenance:

- Layout adaptations in titlepage

Fix:

- Page numbering in TOC

Version 0.11.0 (18.05.2022)*Added:*

- Possibility to import **tex** files

Version 0.10.0 (17.05.2022)*Added:*

- Postprocessing for **rst** and **tex** source files added

Fix:

- multiply-defined labels

Version 0.9.2 (16.05.2022)*Fix:*

- Automated line breaks within code blocks

Version 0.9.1 (11.05.2022)*Maintenance:*

- Documentation

Version 0.9.0 (10.05.2022)

Maintenance:

- Layout maintenance and syntax extensions for newline, newpage and vspace

Version 0.8.0 (10.05.2022)

Changes:

- Bugfixes and code maintenance

Added:

- Version history

Version 0.7.0 (10.05.2022)

Added:

- Setup process and file **README.rst**

Maintenance:

- Code

Version 0.6.0 (09.05.2022)

Added:

- Parameter `INCLUDEUNDOCUMENTED`

Version 0.5.0 (09.05.2022)

Added:

- Parameter `INCLUDEPRIVATE`

Version 0.4.0 (06.05.2022)

Added:

- Possibility to describe complete Python modules

Version 0.3.0 (06.05.2022)

Added:

- Automated headings for functions, classes and methods

Version 0.2.0 (05.05.2022)

Added:

- Python syntax highlighting within code blocks

Version 0.1.0 (04/2022)

Initial version

[GenPackageDoc in GitHub](#)

[GenPackageDoc in PyPi](#)

GenPackageDoc.pdf

Created at 22.06.2026 - 03:50:23

by GenPackageDoc v. 0.44.0 / 05.06.2026
