GenPackageDoc

v. 0.33.1

Holger Queckenstedt

17.10.2022

Contents

Chapter 1

Introduction

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.

It is possible to extend the documentation by the content of additional text files. The docstrings and also the additional text files have to be written in rst syntax (rst is the abbreviation for "**r**e **s**tructured **t**ext", that is a certain markdown dialect).

The documentation is generated in two steps:

1. The rst sources are converted into LaTeX sources
2. The LaTeX sources are converted into a PDF document. This requires a separately installed LaTeX distribution (recommended: MiKTeX), that is **not** part of `GenPackageDoc`.

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

python-genpackagedoc

The repository `python-genpackagedoc` uses it's 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 document 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 PDF document like the one you are currently reading.

Chapter 2

Description

2.1 Repository content

What is the content of the repository `python-genpackagedoc`*?*

- Folder `GenPackageDoc`

  Contains the package code.

  This folder is specific for the package.

- Folder `config`

  Contains the repository configuration (e.g. the name of the package, the name of the repository, the author, and more ...).

  This folder is specific for the repository.

- Folder `additions`

  Contains additionally needed sources like setup related class definitions and sources, that are imported from other repositories - to make this repository stand alone

- 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
  - `setup.py`
    Python script to install the package sources. This includes the execution of `genpackagedoc.py`. Therefore building the documentation is part of the installation process.
  - `dump_repository_config.py`
    Little helper to dump the repository configuration to console

2.2 Documentation build process

How do the files and folders listed above, belong together? What is the way, the information flows when the documentation is generated?

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

  `genpackagedoc.py` can be used stand alone - but this script is also called by `setup.py`. The impact is that every installation includes an update of the documentation.

- `genpackagedoc.py` creates a repository configuration object

  `config/CRepositoryConfig.py`

- The repository configuration object reads the static repository configuration values out of a separate json file

  `config/repository_config.json`

- The repository configuration object adds dynamic values (like operating system specific settings and paths) to the repository configuration. Not all of them are required for the documentation build process, but the repository configuration also supports the setup process (`setup.py`).

  There is one certain setting in the repository configuration file

  `config/repository_config.json`,

  that is essential for the documentation build process:

  `"PACKAGEDOC" : "./packagedoc"`

  This is the path to a folder, in which all further documentation related files are placed. In case of the path is relative, the reference is the position of `genpackagedoc.py`. It is required that within this folder the configuration file for the documentation build process

  `packagedoc_config.json`

  can be found. The name of this json file is fix!

- The configuration file `packagedoc_config.json` contains settings like

  - Paths to Python packages to be documented
  - Paths and names of additional rst files
  - Path and name of output folder (tex files and output PDF file)
  - User defined parameter (that can be defined here as global runtime variables and can be used in any rst 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"`.

Command line

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

`--output`

Path and name of folder containing all output files.

`--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 using `setup.py`). 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. Please try to get the bottom of your motivation when you change this setting.

`--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"
↪  --configdest="../any/other/location" --strict=True
```

All listed parameters are optional. GenPackageDoc creates the complete output path (`--output`) recursively. Other destination folder (`--pdfdest` and `--configdest`) have to exist already.

2.3 PDF document structure

How is the resulting PDF document structured? What causes an entry within the table of content of the PDF document?

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 rst file is a *chapter*.
  - In case of you want to add another chapter to your documentation, you have to include another rst file.
  - The headline of the chapter is the name of the rst file (automatically).
    Therefore the heading within an rst file has to start at section level!
- 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 rst files.
- Within additionally included separate rst files sections and subsections can be defined by the following rst syntax elements for headings:
  - A line underlined with "=" characters is a section
  - A line underlined with "-" characters is a subsection
- 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 subesction level.

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

2.4 Examples

2.4.1 Example 1: rst file

The text of this chapter is taken over from an rst file named `Description.rst`.

This rst file contains the following headlines:

```
Repository content
==================

Documentation build process
===========================

PDF document structure
======================

Examples
========

Example 1: rst file
-------------------

Example 2: Python module
------------------------
```

Because `Description.rst` is the second imported rst file, the chapter number is 2. The chapter headline is "Description" (the name of the rst file). The top level headlines *within* the rst file are at *section* level. The fourth section (Examples) contains two subsections.

The outcome is the following part of the table of content:

2.4.2 Example 2: Python module

Part of this documentation is a Python module with name `CDocBuilder.py` (listed in table of content at *chapter* level). This module contains a class with name `CDocBuilder` (listed in table of content at *section* level). The class `CDocBuilder` contains a method with name `Build` (listed in table of content at *subsection* level).

This causes the following entry within the table of contents:

2.5 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

```
    """
Description of function or method.

**Arguments:**

* ``input_param_1``

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

  Description of input_param_1.

* ``input_param_2``

  / *Condition*: optional / *Type*: bool / *Default*: False /

  Description of input_param_2.

**Returns:**

* ``return_param``

  / *Type*: str /

  Description of return_param.
    """
```

Some of the special characters used within the interface description, are part of the rst syntax. They will be explained in one of the next sections.

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.

2.6 Runtime variables

What are "runtime variables" and how to use them in rst text?

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

1. the static repository configuration
   `config/repository_config.json`
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 rst text (within docstrings of Python modules and also within separate rst 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 rst text:

```
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",
```

2.7 Syntax aspects

2.7.1 Common rules

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

- In both Python and rst 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 rst format) has to follow the rst 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 rst syntax element that is used here.
- In rst also blank lines are part of the syntax!

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

The contents of all doctrings of a Python module will be merged to one single rst document (internally by GenPackageDoc). In this single rst 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 rst syntax rules. Otherwise we either get syntax issues during computation or we get text with a layout that does not fit to our expectation.

Please be attentive while typing your documentation in rst format!

2.7.2 Syntax extensions

`GenPackageDoc` extends the rst syntax by the following topics:

- *newline*

  A newline (line break) is realized by a slash ('`/`') at the end of a line containing any other rst text (this means: the slash must **not** be the only character in line).
  Internally this slash is mapped to the LaTeX command `\newline`.

- *vspace*

  An additional vertical space (size: the height of the '`x`' character - depending on the current type and size of font) is realized by a single slash ('`/`'). This slash must be the only character in line!
  Internally this slash is mapped to the LaTeX command `\vspace{1ex}`.

- *newpage*

  A newpage (page break) is realized by a double slash ('`//`'). These two slashes must be the only characters in line!
  Internally this double slash is mapped to the LaTeX command `\newpage`.

These syntax extensions can currently be used in separate rst files only and are not available within docstrings of Python modules.

Chapter 3

CDocBuilder.py

Python module containing all methods to generate tex sources.

3.1 Class: CDocBuilder

Imported by:

```
from GenPackageDoc.CDocBuilder import CDocBuilder
```

Main class to build tex sources out of docstrings of Python modules and separate text files in rst format.

Depends on a json configuration file, provided by a `oPackageDocConfig` object (this includes the Repository configuration).

Method to execute: `Build()`

3.1.1 Method: 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 4

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.

4.1 Class: CInterface

Imported by:

```
from GenPackageDoc.CInterface import CInterface
```

4.1.1 Method: 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 5

CPackageDocConfig.py

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

5.1 Class: CPackageDocConfig

Imported by:

```
from GenPackageDoc.CPackageDocConfig import CPackageDocConfig
```

5.1.1 Method: PrintConfig

Prints all cofiguration values to console.

5.1.2 Method: PrintConfigKeys

Prints all cofiguration key names to console.

5.1.3 Method: Get

Returns the configuration value belonging to a key name.

5.1.4 Method: GetConfig

Returns the complete configuration dictionary.

Chapter 6

CPatterns.py

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

6.1 Class: CPatterns

Imported by:

```
from GenPackageDoc.CPatterns import 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.

6.1.1 Method: 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.

6.1.2 Method: 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.

6.1.3 Method: 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.

Chapter 7

CSourceParser.py

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

7.1 Class: CSourceParser

Imported by:

```
from GenPackageDoc.CSourceParser import 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.

7.1.1 Method: ParseSourceFile

The method `ParseSourceFile` parses the content of a Python module.

Arguments:

- `sFile`
  / *Condition*: required / *Type*: str /
  Path and name of a single Python module.
- `bIncludePrivate`
  / *Condition*: optional / *Type*: bool / *Default*: False /
  If `False`: private methods are skipped, otherwise they are included in documentation.
- `bIncludeUndocumented`
  / *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 `sFile`.
- `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 8

Appendix

About this package:

Table 8.1: Package setup

Setup parameter	Value
Name	GenPackageDoc
Version	0.33.1
Date	17.10.2022
Description	Documentation builder for Python packages
Package URL	python-genpackagedoc
Author	Holger Queckenstedt
Email	Holger.Queckenstedt@de.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 :: 3 - Alpha
Intended audience	Intended Audience :: Developers
Topic	Topic :: Software Development

Chapter 9

History

0.1.0	04/2022
Initial version	
0.2.0	05.05.2022
Python syntax highlighting within code blocks added	
0.3.0	06.05.2022
Automated headings for functions, classes and methods	
0.4.0	06.05.2022
Possibility to describe complete Python modules added	
0.5.0	09.05.2022
Parameter `INCLUDEPRIVATE` added	
0.6.0	09.05.2022
Parameter `INCLUDEUNDOCUMENTED` added	
0.7.0	10.05.2022
Setup process introduced and README.rst added; code maintenance	
0.8.0	10.05.2022
Bugfixes and code maintenance; history added	
0.9.0	10.05.2022
Layout maintenance and syntax extensions for `newline`, `newpage` and `vspace` reworked	
0.9.1	11.05.2022
Documentation maintenance	
0.9.2	16.05.2022
Fix: automated line breaks within code blocks	
0.10.0	17.05.2022
Postprocessing for rst and tex sources added; 'multiply-defined labels' fix	
0.11.0	18.05.2022
Import of tex files enabled	
0.12.0	19.05.2022
- Admonitions added, based on LaTeX environment `tcolorbox` *- Layout adaptions in titlepage* *- Page numbering fix in TOC*	
0.13.0	24.05.2022
LaTeX style definitions moved to separate folder	
0.14.0	27.05.2022

- *LaTeX compiler check added* - *Control parameter* `STRICT` *added to* `packagedoc_config`	
0.15.0	31.05.2022
- *Command line added* - *Separate GenPackageDoc configuration class added*	
0.16.0	01.06.2022
Path computation reworked	
0.17.0	17.06.2022
- *Configuration dump added* - *Code maintenance* - *Error handling extended*	
0.18.0	20.06.2022
Added parameter to define an output folder for a dump of final configuration	
0.19.0	28.06.2022
- *Method* `GetLaTeXStyles` *added* - *PythonExtensionsCollection updated to version 0.8.0*	
0.20.0	29.06.2022
Document title bugfix: Added missing masking of underlines (required for LaTeX)	
0.21.0	12.07.2022
Separated file `preamble.tex`	
0.22.0	13.07.2022
- *Maintenance of* `preamble.tex` *and* `styles` *folder* - `setup.py` *fix (install tex files also)*	
0.23.0	13.07.2022
Maintenance of `preamble.tex`	
0.24.0	25.07.2022
Maintenance of `robotframeworkaio.sty` *(line breaks in listings)*	
0.25.0	27.07.2022
Layout maintenance of RobotFramework AIO syntax highlighting (`.sty` *files)*	
0.26.0	27.07.2022
History reworked; common.sty introduced	
0.27.0	17.08.2022
Added LaTeX style definition for Python syntax highlighting	
0.28.0	23.08.2022
- *Introduced new LaTeX environment variable* `GENDOC_LATEXPATH` - `robotframeworkaio.sty` *aligned to version in* `GenMainDoc`	
0.29.0	24.08.2022
Changed the way the import path of a module is printed out to PDF file	
0.30.0	31.08.2022
Introduced `simulateonly` *mode (command line switch to skip the PDF generation)*	
0.31.0	12.09.2022
Fix of import path of a module in PDF file	
0.32.0	16.09.2022
- *Added labels at chapter level* - *partial rework of label mechanisms*	

0.33.0	19.09.2022
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)	