python-genpackagedoc

Holger Queckenstedt

10.05.2022

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:

https://github.com/test-fullautomation/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 to learn how to write a documentation.

Currently no `setup.py` is available. Later this setup script will install some libraries of `GenPackageDoc`. As long as there are no libraries installed, the local version within this repository are used for documentation build.

But independend from such installations 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 every other repository.

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 (currently active at this position, later will be installed)

  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
  - `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` 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 will support the setup process later.

  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. The path has to be relative! Reference is the position of `genpackagedoc.py`. This is hard coded internally. The relative path itself can be any one.
  But 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.

Further details are explained within the json file itself.

Now all required input and output parameter are known. After the execution of `genpackagedoc.py` the resulting PDF document can be found under the specified name within the specified output folder.

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 descriptions

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

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 within the next section.

Important to know about 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 it is recommended to start the docstring text within the first column (or rather use the first column as reference for further indentations of rst text).
- In rst also blank lines are part of the syntax!

Please be attentive while typing your documentation in rst format!

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 three 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`

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 extensions

This feature is in an experimental phase currently! And is only available in rst files but not in docstrings.

The question is how to use rst to cause a line break or a page break in the corresponding PDF document?

This is realized by the following additionally introduced syntax elements: `"//nl"` and `"//np"`.

- A `"//nl"` at the end of an rst text line causes a **line break** (without any further vertical space between the lines).
- A `"//np"` at the end of an rst text line causes a **page break.**

Internally `"//nl"` is mapped to the LaTeX command `"\newline"` and `"//np"` is mapped to the LaTeX command `"\newpage"`.

2.8 RST syntax

What is rst and how to use the rst syntax elements within rst files and doctrings of Python modules?

(to be continued)

Chapter 3

CDocBuilder.py

Python module containing all methods to generate tex sources.

3.1 Class: CDocBuilder

Import: `GenPackageDoc.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 `oRepositoryConfig` object.

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

CPatterns.py

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

4.1 Class: CPatterns

Import: `GenPackageDoc.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.

4.1.1 Method: GetHeader

Defines the header of the main tex file.

Arguments:

- `sAuthor`
  / *Condition*: required / *Type*: str /
  The author of the package
- `sTitle`
  / *Condition*: required / *Type*: str /
  The title of the output document
- `sDate`
  / *Condition*: required / *Type*: str /
  The date of the output document

Returns:

- `sHeader`

  / *Type*: str /

  LaTeX code containing the header of main tex file.

4.1.2 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).

- `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.

4.1.3 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 5

CSourceParser.py

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

5.1 Class: CSourceParser

Import: `GenPackageDoc.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.

5.1.1 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 6

Outlook

ToDo list:

- [01]

  Introduce `setup.py` including the execution of `genpackagedoc.py` and adding the generated PDF document to the installation.
  Introduce README.rst and README.md.

  10.05.2022: Setup process introduced and README.rst added

- [02]

  Currently it is hard coded, that private functions and methods are skipped. Therefore they are not part of the resulting PDF document.
  A configuration switch might be useful to give the user the ability to control this behavior.

  09.05.2022: Parameter 'INCLUDEPRIVATE' added

- [03]

  Currently it is implemented that also functions, classes and methods without docstrings are part of the resulting PDF document. They are listed together with the hint, that a docstring is not available.
  A configuration switch might be useful to give the user the ability to control this behavior.

  09.05.2022: Parameter 'INCLUDEUNDOCUMENTED' added

- [04]

  Currently it is implemented that for Python modules will be searched recursively within the given root folder. Maybe the algorithm also catches modules from which the user does not want `GenPackageDoc` to include them.
  A configuration exclude filter can be implemented to skip those files. Or maybe other way round: An include filter includes a subset of available files only.

  The same filter mechanism can be extended for the content of Python modules (= include/exclude functions, classes and methods).

- [05]

  Currently the configuration parameter for the documentation build process are taken from a json file `packagedoc_config.json`.
  It might be helpful to have the possibility to overwrite them in command line (e.g. for redirecting the path to the output folder without changing any code).

- [06]

  Introduce text boxes for warnings, errors and informations.

- [07]

  The documentation build process allows relative paths only
  (in `packagedoc_config.json`).
  Maybe a mechanism is useful to allow absolute paths and paths based on environment variables also.

- [08]

  Explore further rst syntax elements like the code directive. Some of them produces LaTeX code that requires the include of additional LaTeX packages. Sometimes this causes errors that have to be fixed.

  05.05.2022: Python syntax highlighting realized

- [09]

  The documentation has to be extended by a set of rst examples (rst best practices).

- [10]

  A postprocessing for LaTeX code needs to be implemented:

  - Enable proper line breaks
  - Resolve the ambiguity of labels created automatically when the LaTeX code is generated (for every input file separately)

- [11]

  Currently the docstrings of Python modules have to contain a heading for functions, classes and methods. The developer is responsible for that. Maybe it is not necessary to maintain these headings manually. It has to be investigated, if these headings can be added automatically by `GenPackageDoc`.

  06.05.2022: Headings are added automatically.

- [12]

  Currently the documentation of a single Python module starts at *function* or *class* level. This means it is not possible to provide common information about the Python module itself (placed **before** the content of the first function or class of the module). A way have to be found to add such content.

  06.05.2022: Implemented in version 0.4.0

- [13]
  The error handling needs to be extended!
- [14]
  Take over the description of

  `config/repository_config.json`

  from inside this json file (comment blocks) to the main PDF document.
- [15]
  Reference section with useful links
- [16]
  History
- [17]
  Debug switch to enable additional output
- [18]
  Parse decorators to identify Robot Framework keyword definitions
- [19]
  Selftest

Chapter 7

Appendix

About this package:

Package name:

GenPackageDoc

Package version

0.7.0

Package date

10.05.2022

Package description

This package provides a documentation builder for Python packages

Package URL on GitHub

https://github.com/test-fullautomation/python-genpackagedoc

Author of package

Holger Queckenstedt

Email of author

Holger.Queckenstedt@de.bosch.com

Package programming language

Programming Language :: Python :: 3

Package license

License :: OSI Approved :: Apache Software License

Supported operating systems

Operating System :: OS Independent

Python required

>=3.0

Development status

Development Status :: 3 - Alpha

Intended audience

Intended Audience :: Developers

Package topic

Topic :: Software Development

This PDF has been created at 10.05.2022 - 12:09:17
