docrep Documentation

Release 0.2.4

Philipp Sommer

Oct 23, 2018

Contents

1 What's this? **1**

2 Installation **5**

3 API Reference **7**

4 Changelog **21**

4.1 v0.2.4 21
4.2 v0.2.3 21
4.2.1 Changed 21
4.2.2 Added 21
4.3 v0.2.2 22
4.3.1 Added 22
4.4 v0.2.1 22
4.4.1 Changed 22
4.5 v0.2.0 22
4.5.1 Added 22
4.5.2 Changed 23

5 Indices and tables **25**

Python Module Index **27**

CHAPTER 1

What's this?

Welcome to the **doc**umentation **rep**etition module **docrep**! This module targets developers that develop complex and nested Python APIs and helps them to create a well-documented piece of software.

The motivation is simple, it comes from the don't repeat yourself principle and tries to reuse already existing documentation code.

Suppose you have a well-documented function

```
In [1]: def do_something(a, b):
   ...:     """
   ...:     Add two numbers
   ...:
   ...:     Parameters
   ...:     ----------
   ...:     a: int
   ...:         The first number
   ...:     b: int
   ...:         The second number
   ...:
   ...:     Returns
   ...:     -------
   ...:     int
   ...:         `a` + `b`"""
   ...:     return a + b
   ...:
```

and you have another function that builds upon this function

```
In [2]: def do_more(a, b):
   ...:     """
   ...:     Add two numbers and multiply it by 2
   ...:
   ...:     Parameters
   ...:     ----------
   ...:     a: int
```

(continues on next page)

(continued from previous page)

```
   ...:         The first number
   ...:     b: int
   ...:         The second number
   ...:
   ...:     Returns
   ...:     -------
   ...:     int
   ...:         (`a` + `b`) * 2"""
   ...:     return do_something(a, b) * 2
   ...:
```

Here for `do_more` we use the function `do_something` and actually we do not even care about `a` anymore. So we could even say

```
In [3]: def do_more(*args, **kwargs):
   ...:     """...long docstring..."""
   ...:     return do_something(*args, **kwargs) * 2
   ...:
```

because we only care about the result from `do_something`. However, if we want to change something in the parameters documentation of `do_something`, we would have to change it in `do_more`. This can become a severe error source in large and complex APIs!

So instead of copy-and-pasting the entire documentation of `do_something`, we want to automatically repeat the given docstrings and that's what this module is intended for. Hence, The code above could be rewritten via

```
In [4]: import docrep

In [5]: docstrings = docrep.DocstringProcessor()

In [6]: @docstrings.get_sectionsf('do_something')
   ...: @docstrings.dedent
   ...: def do_something(a, b):
   ...:     """
   ...:     Add two numbers
   ...:
   ...:     Parameters
   ...:     ----------
   ...:     a: int
   ...:         The first number
   ...:     b: int
   ...:         The second number
   ...:
   ...:     Returns
   ...:     -------
   ...:     int
   ...:         `a` + `b`"""
   ...:     return a + b
   ...:

In [7]: @docstrings.dedent
   ...: def do_more(*args, **kwargs):
   ...:     """
   ...:     Add two numbers and multiply it by 2
   ...:
   ...:     Parameters
   ...:     ----------
```

(continues on next page)

(continued from previous page)

```
   ...:     %(do_something.parameters)s
   ...:
   ...:     Returns
   ...:     -------
   ...:     int
   ...:         (`a` + `b`) * 2"""
   ...:     return do_something(*args, **kwargs) * 2
   ...:

In [8]: help(do_more)
Help on function do_more in module __main__:

do_more(*args, **kwargs)
    Add two numbers and multiply it by 2

    Parameters
    ----------
    a: int
        The first number
    b: int
        The second number

    Returns
    -------
    int
        (`a` + `b`) * 2
```

You can do the same for any other section in the objects documentation and you can even remove or keep only specific parameters or return types (see *keep_params()* and *delete_params()*). The module intensively uses pythons `re` module so it is very efficient. The only restriction is, that your Python code has to be documented following the numpy conventions (i.e. it should follow the conventions from the sphinx napoleon extension).

If your docstring does not start with an empty line as in the example above, you have to use the *DocstringProcessor.with_indent()* method. See for example

```
In [9]: @docstrings.get_sectionsf('do_something')
   ...: def second_example_source(a, b):
   ...:     """Summary is on the first line
   ...:
   ...:     Parameters
   ...:     ----------
   ...:     a: int
   ...:         The first number
   ...:     b: int
   ...:         The second number
   ...:
   ...:     Returns
   ...:     -------
   ...:     int
   ...:         `a` + `b`"""
   ...:     return a + b
   ...:

In [10]: @docstrings.with_indent(4)  # we indent the replacements with 4 spaces
    ....: def second_example_target(*args, **kwargs):
    ....:     """Target docstring with summary on the first line
    ....:
```

(continues on next page)

(continued from previous page)

```
   ....:     Parameters
   ....:     ----------
   ....:     %(do_something.parameters)s
   ....:
   ....:     Returns
   ....:     -------
   ....:     int
   ....:         (`a` + `b`) * 2"""
   ....:     return second_example_source(*args, **kargs) * 2
   ....:

In [11]: help(second_example_target)
Help on function second_example_target in module __main__:

second_example_target(*args, **kwargs)
    Target docstring with summary on the first line

    Parameters
    ----------
    a: int
        The first number
    b: int
        The second number

    Returns
    -------
    int
        (`a` + `b`) * 2
```

CHAPTER 2

Installation

Installation simply goes via pip:

```
$ pip install docrep
```

or via conda:

```
$ conda install -c conda-forge docrep
```

or from the source on github via:

```
$ python setup.py install
```

Note: When using docrep in python 2.7, there is to mention that the `__doc__` attribute of classes is not writable, so something like

```
In [12]: @docstrings
    ....: class SomeClass(object):
    ....:     """An awesome class
    ....:
    ....:     Parameters
    ....:     ----------
    ....:     %(repeated.parameters)s
    ....:     """
    ....:
```

would raise an error. There are several workarounds (see the issue on github) but the default for python 2.7 is, to simply not modify class docstrings. You can, however, change this behaviour using the *DocstringProcessor.python2_classes* attribute.

CHAPTER 3

API Reference

Classes

DocstringProcessor(*args, **kwargs)	Class that is intended to process docstrings

Functions

safe_modulo(s, meta[, checked, ...])	Safe version of the modulo operation (%) of strings

class docrep.**DocstringProcessor**(**args*, ***kwargs*)

Bases: object

Class that is intended to process docstrings

It is, but only to minor extends, inspired by the `matplotlib.docstring.Substitution` class.

Examples

Create docstring processor via:

```
>>> from docrep import DocstringProcessor
>>> d = DocstringProcessor(doc_key='My doc string')
```

And then use it as a decorator to process the docstring:

```
>>> @d
... def doc_test():
...     '''That's %(doc_key)s'''
...     pass

>>> print(doc_test.__doc__)
That's My doc string
```

Use the *get_sectionsf()* method to extract Parameter sections (or others) form the docstring for later

usage (and make sure, that the docstring is dedented):

```
>>> @d.get_sectionsf('docstring_example',
...                  sections=['Parameters', 'Examples'])
... @d.dedent
... def doc_test(a=1, b=2):
...     '''
...     That's %(doc_key)s
...
...     Parameters
...     ----------
...     a: int, optional
...         A dummy parameter description
...     b: int, optional
...         A second dummy parameter
...
...     Examples
...     --------
...     Some dummy example doc'''
...     print(a)

>>> @d.dedent
... def second_test(a=1, b=2):
...     '''
...     My second function where I want to use the docstring from
...     above
...
...     Parameters
...     ----------
...     %(docstring_example.parameters)s
...
...     Examples
...     --------
...     %(docstring_example.examples)s'''
...     pass

>>> print(second_test.__doc__)
My second function where I want to use the docstring from
above

Parameters
----------
a: int, optional
    A dummy parameter description
b: int, optional
    A second dummy parameter

Examples
--------
Some dummy example doc
```

Another example uses non-dedented docstrings:

```
>>> @d.get_sectionsf('not_dedented')
... def doc_test2(a=1):
...     '''That's the summary
...
...     Parameters
```

(continues on next page)

(continued from previous page)

```
...     ----------
...     a: int, optional
...         A dummy parameter description'''
...     print(a)
```

These sections must then be used with the `with_indent()` method to indent the inserted parameters:

```
>>> @d.with_indent(4)
... def second_test2(a=1):
...     '''
...     My second function where I want to use the docstring from
...     above
...
...     Parameters
...     ----------
...     %(not_dedented.parameters)s'''
...     pass
```

Methods

`dedent`(func)	Dedent the docstring of a function and substitute with `params`
`dedents`(s[, stacklevel])	Dedent a string and substitute with the `params` attribute
`delete_kwargs`(base_key[, args, kwargs])	Deletes the `*args` or `**kwargs` part from the parameters section
`delete_kwargs_s`(s[, args, kwargs])	Deletes the `*args` or `**kwargs` part from the parameters section
`delete_params`(base_key, *params)	Method to delete a parameter from a parameter documentation.
`delete_params_s`(s, params)	Delete the given parameters from a string
`delete_types`(base_key, out_key, *types)	Method to delete a parameter from a parameter documentation.
`delete_types_s`(s, types)	Delete the given types from a string
`get_extended_summary`(s[, base])	Get the extended summary from a docstring
`get_extended_summaryf`(*args, **kwargs)	Extract the extended summary from a function docstring
`get_full_description`(s[, base])	Get the full description from a docstring
`get_full_descriptionf`(*args, **kwargs)	Extract the full description from a function docstring
`get_sections`(s, base[, sections])	Method that extracts the specified sections out of the given string if (and only if) the docstring follows the numpy documentation guidelines[1].
`get_sectionsf`(*args, **kwargs)	Decorator method to extract sections from a function docstring
`get_summary`(s[, base])	Get the summary of the given docstring
`get_summaryf`(*args, **kwargs)	Extract the summary from a function docstring
`keep_params`(base_key, *params)	Method to keep only specific parameters from a parameter documentation.
`keep_params_s`(s, params)	Keep the given parameters from a string
`keep_types`(base_key, out_key, *types)	Method to keep only specific parameters from a parameter documentation.

Continued on next page

Table 3 – continued from previous page

keep_types_s(s, types)	Keep the given types from a string
save_docstring(key)	Descriptor method to save a docstring from a function
with_indent([indent])	Substitute in the docstring of a function with indented `params`
with_indents(s[, indent, stacklevel])	Substitute a string with the indented `params`

Attributes

param_like_sections	sections that behave the same as the *Parameter* section by defining a
params	`dict`. Dictionary containing the parameters that are used in for
patterns	`dict`. Dictionary containing the compiled patterns to identify
python2_classes	The action on how to react on classes in python 2
text_sections	sections that include (possibly not list-like) text

Parameters and **kwargs (**args*) – Parameters that shall be used for the substitution. Note that you can only provide either `*args` or `**kwargs`, furthermore most of the methods like *get_sectionsf* require `**kwargs` to be provided.

dedent (*func*)

Dedent the docstring of a function and substitute with *params*

Parameters func (*function*) – function with the documentation to dedent and whose sections shall be inserted from the *params* attribute

dedents (*s*, *stacklevel=3*)

Dedent a string and substitute with the *params* attribute

Parameters

- **s** (*str*) – string to dedent and insert the sections of the *params* attribute
- **stacklevel** (*int*) – The stacklevel for the warning raised in `safe_module()` when encountering an invalid key in the string

delete_kwargs (*base_key*, *args=None*, *kwargs=None*)

Deletes the `*args` or `**kwargs` part from the parameters section

Either *args* or *kwargs* must not be None. The resulting key will be stored in

base_key + 'no_args' if *args* is not None and *kwargs* is None

base_key + 'no_kwargs' if *args* is None and *kwargs* is not None

base_key + 'no_args_kwargs' if *args* is not None and *kwargs* is not None

Parameters

- **base_key** (*str*) – The key in the *params* attribute to use
- **args** (*None or str*) – The string for the args to delete
- **kwargs** (*None or str*) – The string for the kwargs to delete

[1] https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt

Notes

The type name of *args* in the base has to be like ``*<args>`` (i.e. the *args* argument preceeded by a '*' and enclosed by double '``'). Similarily, the type name of *kwargs* in *s* has to be like ``**<kwargs>``

classmethod delete_kwargs_s (*s*, *args=None*, *kwargs=None*)

Deletes the *args or **kwargs part from the parameters section

Either *args* or *kwargs* must not be None.

Parameters

- **s** (*str*) – The string to delete the args and kwargs from
- **args** (*None or str*) – The string for the args to delete
- **kwargs** (*None or str*) – The string for the kwargs to delete

Notes

The type name of *args* in *s* has to be like ``*<args>`` (i.e. the *args* argument preceeded by a '*' and enclosed by double '``'). Similarily, the type name of *kwargs* in *s* has to be like ``**<kwargs>``

delete_params (*base_key*, **params*)

Method to delete a parameter from a parameter documentation.

This method deletes the given *param* from the *base_key* item in the *params* dictionary and creates a new item with the original documentation without the description of the param. This method works for the 'Parameters' sections.

The new docstring without the selected parts will be accessible as base_key + '.no_' + '|'.join(params), e.g. 'original_key.no_param1|param2'.

See the *keep_params()* method for an example.

Parameters

- **base_key** (*str*) – key in the *params* dictionary
- ***params** – str. Parameter identifier of which the documentations shall be deleted

See also:

delete_types(), *keep_params()*

static delete_params_s (*s*, *params*)

Delete the given parameters from a string

Same as *delete_params()* but does not use the *params* dictionary

Parameters

- **s** (*str*) – The string of the parameters section
- **params** (*list of str*) – The names of the parameters to delete

Returns The modified string *s* without the descriptions of *params*

Return type str

delete_types (*base_key*, *out_key*, **types*)

Method to delete a parameter from a parameter documentation.

This method deletes the given *param* from the *base_key* item in the `params` dictionary and creates a new item with the original documentation without the description of the param. This method works for `'Results'` like sections.

See the `keep_types()` method for an example.

Parameters

- **base_key** (`str`) – key in the `params` dictionary
- **out_key** (`str`) – Extension for the base key (the final key will be like `'%s.%s' % (base_key, out_key)`
- ***types** – str. The type identifier of which the documentations shall deleted

See also:

`delete_params()`

static delete_types_s(*s*, *types*)

Delete the given types from a string

Same as `delete_types()` but does not use the `params` dictionary

Parameters

- **s** (`str`) – The string of the returns like section
- **types** (`list of str`) – The type identifiers to delete

Returns The modified string *s* without the descriptions of *types*

Return type str

get_extended_summary(*s*, *base=None*)

Get the extended summary from a docstring

This here is the extended summary

Parameters

- **s** (`str`) – The docstring to use
- **base** (`str or None`) – A key under which the summary shall be stored in the `params` attribute. If not None, the summary will be stored in `base + '.summary_ext'`. Otherwise, it will not be stored at all

Returns The extracted extended summary

Return type str

get_extended_summaryf(**args*, ***kwargs*)

Extract the extended summary from a function docstring

This function can be used as a decorator to extract the extended summary of a function docstring (similar to `get_sectionsf()`).

Parameters and **kwargs (`*args`) – See the `get_extended_summary()` method. Note, that the first argument will be the docstring of the specified function

Returns Wrapper that takes a function as input and registers its summary via the `get_extended_summary()` method

Return type function

get_full_description(*s*, *base=None*)

Get the full description from a docstring

This here and the line above is the full description (i.e. the combination of the `get_summary()` and the `get_extended_summary()`) output

Parameters

- **s** (*str*) – The docstring to use
- **base** (*str or None*) – A key under which the description shall be stored in the `params` attribute. If not None, the summary will be stored in `base + '.full_desc'`. Otherwise, it will not be stored at all

Returns The extracted full description

Return type str

get_full_descriptionf(**args*, ***kwargs*)

Extract the full description from a function docstring

This function can be used as a decorator to extract the full descriptions of a function docstring (similar to `get_sectionsf()`).

Parameters and **kwargs (**args*) – See the `get_full_description()` method. Note, that the first argument will be the docstring of the specified function

Returns Wrapper that takes a function as input and registers its summary via the `get_full_description()` method

Return type function

get_sections(*s, base, sections=['Parameters', 'Other Parameters']*)

Method that extracts the specified sections out of the given string if (and only if) the docstring follows the numpy documentation guidelines[1]. Note that the section either must appear in the `param_like_sections` or the `text_sections` attribute.

Parameters

- **s** (*str*) – Docstring to split
- **base** (*str*) – base to use in the `sections` attribute
- **sections** (*list of str*) – sections to look for. Each section must be followed by a newline character ('n') and a bar of '-' (following the numpy (napoleon) docstring conventions).

Returns The replaced string

Return type str

References

See also:

`delete_params()`, `keep_params()`, `delete_types()`, `keep_types()`, `delete_kwargs()`

save_docstring() for saving an entire docstring

get_sectionsf(**args*, ***kwargs*)

Decorator method to extract sections from a function docstring

Parameters and **kwargs (**args*) – See the `get_sections()` method. Note, that the first argument will be the docstring of the specified function

Returns Wrapper that takes a function as input and registers its sections via the `get_sections()` method

Return type function

get_summary(*s*, *base=None*)

Get the summary of the given docstring

This method extracts the summary from the given docstring *s* which is basicly the part until two newlines appear

Parameters

- **s** (*str*) – The docstring to use
- **base** (*str or None*) – A key under which the summary shall be stored in the `params` attribute. If not None, the summary will be stored in `base + '.summary'`. Otherwise, it will not be stored at all

Returns The extracted summary

Return type str

get_summaryf(**args*, ***kwargs*)

Extract the summary from a function docstring

Parameters and **kwargs (**args*) – See the `get_summary()` method. Note, that the first argument will be the docstring of the specified function

Returns Wrapper that takes a function as input and registers its summary via the `get_summary()` method

Return type function

keep_params(*base_key*, **params*)

Method to keep only specific parameters from a parameter documentation.

This method extracts the given *param* from the *base_key* item in the `params` dictionary and creates a new item with the original documentation with only the description of the param. This method works for `'Parameters'` like sections.

The new docstring with the selected parts will be accessible as `base_key + '.' + '|'.join(params)`, e.g. `'original_key.param1|param2'`

Parameters

- **base_key** (*str*) – key in the `params` dictionary
- ***params** – str. Parameter identifier of which the documentations shall be in the new section

See also:

`keep_types()`, `delete_params()`

Examples

To extract just two parameters from a function and reuse their docstrings, you can type:

```
>>> from docrep import DocstringProcessor
>>> d = DocstringProcessor()
>>> @d.get_sectionsf('do_something')
... def do_something(a=1, b=2, c=3):
...     '''
...     That's %(doc_key)s
...
...     Parameters
...     ----------
...     a: int, optional
...         A dummy parameter description
...     b: int, optional
...         A second dummy parameter that will be excluded
...     c: float, optional
...         A third parameter'''
...     print(a)

>>> d.keep_params('do_something.parameters', 'a', 'c')

>>> @d.dedent
... def do_less(a=1, c=4):
...     '''
...     My second function with only `a` and `c`
...
...     Parameters
...     ----------
...     %(do_something.parameters.a|c)s'''
...     pass

>>> print(do_less.__doc__)
My second function with only `a` and `c`

Parameters
----------
a: int, optional
    A dummy parameter description
c: float, optional
    A third parameter
```

Equivalently, you can use the `delete_params()` method to remove parameters:

```
>>> d.delete_params('do_something.parameters', 'b')

>>> @d.dedent
... def do_less(a=1, c=4):
...     '''
...     My second function with only `a` and `c`
...
...     Parameters
...     ----------
...     %(do_something.parameters.no_b)s'''
...     pass
```

---

static keep_params_s(*s*, *params*)

Keep the given parameters from a string

Same as `keep_params()` but does not use the `params` dictionary

Parameters

- **s** (*str*) – The string of the parameters like section
- **params** (*list of str*) – The parameter names to keep

Returns The modified string *s* with only the descriptions of *params*

Return type str

keep_types(*base_key*, *out_key*, **types*)

Method to keep only specific parameters from a parameter documentation.

This method extracts the given *type* from the *base_key* item in the *params* dictionary and creates a new item with the original documentation with only the description of the type. This method works for the `'Results'` sections.

Parameters

- **base_key** (*str*) – key in the *params* dictionary
- **out_key** (*str*) – Extension for the base key (the final key will be like `'%s.%s' % (base_key, out_key)`
- ***types** – str. The type identifier of which the documentations shall be in the new section

See also:

delete_types(), *keep_params()*

Examples

To extract just two return arguments from a function and reuse their docstrings, you can type:

```
>>> from docrep import DocstringProcessor
>>> d = DocstringProcessor()
>>> @d.get_sectionsf('do_something', sections=['Returns'])
... def do_something():
...     '''
...     That's %(doc_key)s
...
...     Returns
...     -------
...     float
...         A random number
...     int
...         A random integer'''
...     return 1.0, 4

>>> d.keep_types('do_something.returns', 'int_only', 'int')

>>> @d.dedent
... def do_less():
...     '''
...     My second function that only returns an integer
...
...     Returns
...     -------
...     %(do_something.returns.int_only)s'''
...     return do_something()[1]

>>> print(do_less.__doc__)
```

(continues on next page)

(continued from previous page)

```
My second function that only returns an integer

Returns
-------
int
    A random integer
```

Equivalently, you can use the `delete_types()` method to remove parameters:

```
>>> d.delete_types('do_something.returns', 'no_float', 'float')

>>> @d.dedent
... def do_less():
...     '''
...     My second function with only `a` and `c`
...
...     Returns
...     ----------
...     %(do_something.returns.no_float)s'''
...     return do_something()[1]
```

---

static keep_types_s(*s*, *types*)

Keep the given types from a string

Same as `keep_types()` but does not use the `params` dictionary

Parameters

- **s** (*str*) – The string of the returns like section
- **types** (*list of str*) – The type identifiers to keep

Returns The modified string *s* with only the descriptions of *types*

Return type str

param_like_sections = ['Parameters', 'Other Parameters', 'Returns', 'Raises']

sections that behave the same as the *Parameter* section by defining a list

params = {}

`dict`. Dictionary containing the parameters that are used in for substitution.

patterns = {}

`dict`. Dictionary containing the compiled patterns to identify the Parameters, Other Parameters, Warnings and Notes sections in a docstring

python2_classes = 'ignore'

The action on how to react on classes in python 2

When calling:

```
>>> @docstrings
... class NewClass(object):
...     """%(replacement)s"""
```

This normaly raises an AttributeError, because the `__doc__` attribute of a class in python 2 is not writable. This attribute may be one of `'ignore', 'raise' or 'warn'`

save_docstring(*key*)

Descriptor method to save a docstring from a function

Like the *get_sectionsf()* method this method serves as a descriptor for functions but saves the entire docstring

text_sections = ['Warnings', 'Notes', 'Examples', 'See Also', 'References']
sections that include (possibly not list-like) text

with_indent(*indent=0*)
Substitute in the docstring of a function with indented *params*

Parameters **indent** (*int*) – The number of spaces that the substitution should be indented

Returns Wrapper that takes a function as input and substitutes it's `__doc__` with the indented versions of *params*

Return type function

See also:

with_indents(), *dedent()*

with_indents(*s*, *indent=0*, *stacklevel=3*)
Substitute a string with the indented *params*

Parameters

- **s** (*str*) – The string in which to substitute
- **indent** (*int*) – The number of spaces that the substitution should be indented
- **stacklevel** (*int*) – The stacklevel for the warning raised in `safe_module()` when encountering an invalid key in the string

Returns The substituted string

Return type str

See also:

with_indent(), *dedents()*

docrep.**safe_modulo**(*s*, *meta*, *checked=''*, *print_warning=True*, *stacklevel=2*)
Safe version of the modulo operation (%) of strings

Parameters

- **s** (*str*) – string to apply the modulo operation with
- **meta** (*dict or tuple*) – meta informations to insert (usually via `s % meta`)
- **checked** (*{'KEY', 'VALUE'}, optional*) – Security parameter for the recursive structure of this function. It can be set to 'VALUE' if an error shall be raised when facing a TypeError or ValueError or to 'KEY' if an error shall be raised when facing a KeyError. This parameter is mainly for internal processes.
- **print_warning** (*bool*) – If True and a key is not existent in *s*, a warning is raised
- **stacklevel** (*int*) – The stacklevel for the `warnings.warn()` function

---

Examples

The effects are demonstrated by this example:

```
>>> from docrep import safe_modulo
>>> s = "That's %(one)s string %(with)s missing 'with' and %s key"
>>> s % {'one': 1}          # raises KeyError because of missing 'with'
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'with'
>>> s % {'one': 1, 'with': 2}        # raises TypeError because of '%s'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
>>> safe_modulo(s, {'one': 1})
"That's 1 string %(with)s missing 'with' and %s key"
```

CHAPTER 4

Changelog

4.1 v0.2.4

This new minor release has an improved documentation considering the `keep_params` and `keep_types` section and triggers new builds for python 3.7.

4.2 v0.2.3

This minor release contains some backward incompatible changes on how to handle the decorators for classes in python 2.7. Thanks @lesteve and @guillaumeeb for your input on this.

4.2.1 Changed

- When using the decorators for classes in python 2.7, e.g. via:

```
>>> @docstrings
... class Something(object):
...     "%(replacement)s"
```

  it does not have an effect anymore. This is because class docstrings cannot be modified in python 2.7 (see issue #5). The original behaviour was to raise an error. You can restore the old behaviour by setting *DocstringProcessor.python2_classes = 'raise'*.

- Some docs have been updated (see PR #7)

4.2.2 Added

- the *DocstringProcessor.python2_classes* to change the handling of classes in python 2.7

4.3 v0.2.2

4.3.1 Added

- We introduce the `DocstringProcessor.get_extended_summary()` and `DocstringProcessor.get_extended_summaryf()` methods to extract the extended summary (see the numpy documentation guidelines).
- We introduce the `DocstringProcessor.get_full_description()` and `DocstringProcessor.get_full_descriptionf()` methods to extract the full description (i.e. the summary plus extended summary) from a function docstring

4.4 v0.2.1

4.4.1 Changed

- Minor bug fix in the get_sections method

4.5 v0.2.0

4.5.1 Added

- Changelog
- the *get_sectionsf* and *get_sections* methods now also support non-dedented docstrings that start with the summary, such as:

```
>>> d = DocstringProcessor()
>>> @d.get_sectionsf('source')
... def source_func(a=1):
...     '''That's the summary
...
...        Parameters
...        ----------
...        a: int, optional
...            A dummy parameter description'''
...     pass
```

- the new *with_indent* and *with_indents* methods can be used to replace the argument in a non-dedented docstring, such as:

```
>>> @d.with_indent(4)
... def target_func(a=1):
...     """Another function using arguments of source_func
...
...     Parameters
...     ----------
...     %(source.parameters)s"""
...     pass

>>> print(target_func.__doc__)
```

(continues on next page)

(continued from previous page)

```
Another function using arguments of source_func

    Parameters
    ----------
    a: int, optional
        A dummy parameter description
```

4.5.2 Changed

- the *get_sectionsf* and *get_sections* method now always uses the dedented version of the docstring. Thereby it first removes the summary.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

docrep, 7

Index

D

dedent() (docrep.DocstringProcessor method), 10
dedents() (docrep.DocstringProcessor method), 10
delete_kwargs() (docrep.DocstringProcessor method), 10
delete_kwargs_s() (docrep.DocstringProcessor class method), 11
delete_params() (docrep.DocstringProcessor method), 11
delete_params_s() (docrep.DocstringProcessor static method), 11
delete_types() (docrep.DocstringProcessor method), 11
delete_types_s() (docrep.DocstringProcessor static method), 12
docrep (module), 7
DocstringProcessor (class in docrep), 7

G

get_extended_summary() (docrep.DocstringProcessor method), 12
get_extended_summaryf() (docrep.DocstringProcessor method), 12
get_full_description() (docrep.DocstringProcessor method), 12
get_full_descriptionf() (docrep.DocstringProcessor method), 13
get_sections() (docrep.DocstringProcessor method), 13
get_sectionsf() (docrep.DocstringProcessor method), 13
get_summary() (docrep.DocstringProcessor method), 14
get_summaryf() (docrep.DocstringProcessor method), 14

K

keep_params() (docrep.DocstringProcessor method), 14
keep_params_s() (docrep.DocstringProcessor static method), 15
keep_types() (docrep.DocstringProcessor method), 16
keep_types_s() (docrep.DocstringProcessor static method), 17

P

param_like_sections (docrep.DocstringProcessor attribute), 17
params (docrep.DocstringProcessor attribute), 17
patterns (docrep.DocstringProcessor attribute), 17
python2_classes (docrep.DocstringProcessor attribute), 17

S

safe_modulo() (in module docrep), 18
save_docstring() (docrep.DocstringProcessor method), 17

T

text_sections (docrep.DocstringProcessor attribute), 18

W

with_indent() (docrep.DocstringProcessor method), 18
with_indents() (docrep.DocstringProcessor method), 18