SectionDoc Documentation

Release 0.2

Ioannis Tziakos

Jun 13, 2017

Contents

The sectiondoc extension parses the function and class docstrings as they are retrieved by the autodoc extension and renders the section blocks into sphinx friendly rst. The extension shares similarities with alternatives (such as numpydoc) but aims at reflecting the original form of the docstring and support project specific customizations.

Key features of **sectiondoc** are:

- Do not change the order of sections.
- Allow sphinx directives between (and inside) section blocks.
- Custom rendering styles

Note: Sectiondoc should work with sphinx >= 0.4 that provides the `autodoc-process-docstring` hook.

CHAPTER 1

Repository

The sectiondoc extension lives at Github. You can clone the repository using:

```
$ git clone https://github.com/enthought/sectiondoc.git
```

CHAPTER 2

Installation

Install `sectiondoc` from pypi using pip:

```
$ pip install sectiondoc
```

Install the latest developing version using:

```
$ pip install git+https://github.com/enthought/sectiondoc.git#egg=sectiondoc
```

CHAPTER 3

Usage

Styles can be selected by referencing in `conf.py` the module they are defined:

```
extensions = [
    ...,
    'sectiondoc.styles.legacy',
    ...,
]
```

CHAPTER 4

Contents

Docstring rendering

The are five different parts in the pipeline of **sectiondoc** docstring rendering.

Style

The rendering `Style` maps the objects types provided by autodoc to `DocRender` factory instances which are responsible for rendering the provided docstring.

The DocRender

The DocRender is responsible for doing the actual work. At initialization the class receives a dictionary mapping section titles to a tuple containing a rendering function and (optionally) section item parsing and rendering classes. The actual rendering starts by executing *`parse()`* to detect sections in the docstring. For each section that is discovered the `_render()` is called with the name of the discovered section to further dispatch processing to the associated section rendering function. If an associated function to the section does not exist the default is to use *`rubric()`*.

Section rendering function

The rendering fuctions will use the utility methods of the the DocRender instance to extract the section block. Depedending on the implementation `extract_paragraph()` is called to return the paragraph for further processing or `extract_items()` is called to return the list of *`Item`* instances. When a list of *`Item`* is collected the section the rendering function will produce the updated rst docstring using the appropriate *`Renderer`*.

Item

`Item` instances contain the `term`, `classfier(s)` and `definition` information of items in a section. Each `Item` type knows how to parse a set of lines grouping and filtering the information ready to be rendered into sphinx friendly rst.

Renderer

The `Renderer` is used by the section renderer functions to render a previously contructed `Item` into sphinx friently rst.

Styles

SectionDoc comes with the following predefined rendering styles

Default

Default style is a stricter implementation of **legacy_** where the definition item description is parsed using the *`DefinitionItem`* which follows the `rst` definition more closely.

For class objects the **default** renders 5 types of sections:

Heading	Description	Parse as	Max	Rendered as
Attributes	Class attributes and their usage	DefinitionItem	–	Sphinx attributes
Arguments	function arguments and type	DefinitionItem	–	Parameters field list
Parameters	function arguments and type	DefinitionItem	–	Parameters field list
Methods	Class methods with summary	MethodItem	–	Table with links to the methods
Notes	Useful notes	paragraph	1	Note admonition

For functions the **default** renders six types of sections:

Heading	Description	Parse as	Max	Rendered as
Arguments	function arguments and type	DefinitionItem	–	Parameters field list
Parameters	function arguments and type	DefinitionItem	–	Parameters field list
Returns	Return value	DefinitionItem	–	Unordered list
Raises	Raised exceptions	DefinitionItem	–	Unordered list
Yields	Yield values	DefinitionItem	–	Unordered list
Notes	Useful notes	paragraph	1	Note admonition

Note: All other sections are rendered using the `.. rubric::` directive by default.

layout rules

To be able to detect and render the sections properly the docstrings should follow the following rules:

- Between the section header and the first section item there can be at most only one empty line.
- The end of the section is designated by one of the following:
  - The allowed number of items by the section has been parsed.
  - Two consecutive empty lines are found.
  - The line is not identified as a possible header of the section item.

Hint: Please check the docstring of the specific definition item class to have more information regarding the valid item header format.

Examples

Argument section

```
Arguments
---------
inputa : str
    The first argument holds the first input!.

    This is the second paragraph.

inputb : float : int
    The second argument is a float.
    the default value is 0.

    .. note:: this is an optional value.
```

arguments (*inputa*, *inputb*)

Parameters

- **inputa** (*str*) – The first argument holds the first input!.

  This is the second paragraph.

- **inputb** (*float or int*) – The second argument is a float. the default value is 0.

  Note: this is an optional value.

Attribute sections

```
Attributes
----------
docstring : list
    A list of strings (lines) that holds docstrings. The lines are
    changed inplace.

index : int
    The zero-based line number of the docstring that is currently
    processed.
```

class **Attributes**

docstring = list
A list of strings (lines) that holds docstrings

index = int
The current zero-based line number of the docstring that is proccessed.

Returns sections

```
Returns
-------
myvalue : list
    A list of important values.
    But we need to say more things about it.
```

returns()

Returns **myvalue** (*list*) – A list of important values. But we need to say more things about it.

Raises section

```
Raises
------
TypeError
    This is the first paragraph of the description.
    More description.

ValueError
    Description of another case where errors are raised.
```

raises()

Raises

- **TypeError** – This is the first paragraph of the description. More description.
- **ValueError** – Description of another case where errors are raised.

Method section

```
Methods
-------
extract_fields(indent='', field_check=None)
    Extract the fields from the docstring

get_field()
    Get the field description.

get_next_paragraph()
    Get the next paragraph designated by an empty line.
```

class **MyClass**

Method	Description
`extract_fields(indent='', field_check=None)`	Extract the fields from the docstring
`get_field()`	Get the field description.
`get_next_paragraph()`	Get the next paragraph designated by an empty line.

Notes

```
Notes
-----
Empty strings are not changed.
```

Note: Empty strings are not changed.

Legacy

Previous versions of Sectiondoc (and the even older refactordoc package) supported a single style for rendering sections in function/method doc-strings. The old style is still supported in recent versions as the **legacy** style.

For class objects the **legacy** renders five types of sections:

Heading	Description	Parse as	Max	Rendered as
Attributes	Class attributes and their usage	OrDefinitionItem	–	Sphinx attributes
Arguments	function arguments and type	OrDefinitionItem	–	Parameters field list
Parameters	function arguments and type	OrDefinitionItem	–	Parameters field list
Methods	Class methods with summary	MethodItem	–	Table with links to the methods
Notes	Useful notes	paragraph	1	Note admonition

For functions the **default** renders six types of sections:

Heading	Description	Parse as	Max	Rendered as
Arguments	function arguments and type	OrDefinitionItem	–	Parameters field list
Parameters	function arguments and type	OrDefinitionItem	–	Parameters field list
Returns	Return value	OrDefinitionItem	–	Unordered list
Raises	Raised exceptions	OrDefinitionItem	–	Unordered list
Yields	Yield values	OrDefinitionItem	–	Unordered list
Notes	Useful notes	paragraph	1	Note admonition

Note: All other sections are rendered using the `.. rubric::` directive by default.

layout rules

To be able to detect and render the sections properly the docstrings should follow the following rules:

- Between the section header and the first section item there can be at most only one empty line.
- The end of the section is designated by one of the following:
  - The allowed number of items by the section has been parsed.
  - Two consecutive empty lines are found.
  - The line is not identified as a possible header of the section item.

  Hint: Please check the docstring of the specific definition item class to have more information regarding the valid item header format.

Examples

Argument section

```
Arguments
---------
inputa : str
    The first argument holds the first input!.

    This is the second paragraph.

inputb : float or int
    The second argument is a float.
    the default value is 0.

    .. note:: this is an optional value.
```

`arguments` (*inputa*, *inputb*)

Parameters

- **`inputa`** (*`str`*) – The first argument holds the first input!.

  This is the second paragraph.

- **`inputb`** (*`float or int`*) – The second argument is a float. the default value is 0.

Note: this is an optional value.

Attribute sections

```
Attributes
----------
docstring : list
    A list of strings (lines) that holds docstrings. The lines are
    changed inplace.

index : int
    The zero-based line number of the docstring that is currently
    processed.
```

class **`Attributes`**

`docstring` = **list**

A list of strings (lines) that holds docstrings

`index` = **int**

The current zero-based line number of the docstring that is proccessed.

Returns sections

```
Returns
-------
myvalue : list
    A list of important values.
    But we need to say more things about it.
```

returns()

Returns **myvalue** (*list*) – A list of important values. But we need to say more things about it.

Raises section

```
Raises
------
TypeError :
    This is the first paragraph of the description.
    More description.

ValueError :
    Description of another case where errors are raised.
```

raises()

Raises

- **TypeError** – This is the first paragraph of the description. More description.
- **ValueError** – Description of another case where errors are raised.

Method section

```
Methods
-------
extract_fields(indent='', field_check=None)
    Extract the fields from the docstring

get_field()
    Get the field description.

get_next_paragraph()
    Get the next paragraph designated by an empty line.
```

class **MyClass**

Method	Description
`extract_fields(indent='', field_check=None)`	Extract the fields from the docstring
`get_field()`	Get the field description.
`get_next_paragraph()`	Get the next paragraph designated by an empty line.

Notes

```
Notes
-----
Empty strings are not changed.
```

Note: Empty strings are not changed.

Note:

- The default rendering style is currently `default`

Extending

Custom styles can be created by instanciating a `Style` to map a *`DocRender`* factory for each type of object rendered by autodoc. For example adding the following functions in you conf.py defines a rendering style for functions and methods:

```
def function_section(lines):
  return DocRender(
      lines,
      sections={
          'Returns': (item_list, ListItem, OrDefinitionItem),
          'Arguments': (arguments, Argument, OrDefinitionItem),
          'Parameters': (arguments, Argument, OrDefinitionItem),
          'Raises': (item_list, ListItem, OrDefinitionItem),
          'Yields': (item_list, ListItem, OrDefinitionItem),
          'Notes': (notes_paragraph, None, None)})


def setup(app):
    style = Style({
        'function': function_section,
        'method': function_section})
    app.setup_extension('sphinx.ext.autodoc')
    app.connect('autodoc-process-docstring', style.render_docstring)
```

Specifically the `Style` instance will map the `function` and `method` docstrings to the dostring rendering funtion `function_section`. The `DocRender` will then detect the sections `Returns`, `Arguments`, `Parameters`, `Raises`, `Yields`, `Notes` and use the mapped combination of section rendering function, Item description and item rendering type to render the detected section in-place.

The rendering styles can be further extented by implemeting new Item, Renderer instances or section rendering functions.

Library Reference

The extension is separated into three main parts.

Styles

class `sectiondoc.styles.`**`DocRender`**(*lines*, *sections=None*)

Docstring rendering class.

The class' main purpose is to parse the docstring and find the sections that need to be refactored. The operation take place in two stages:

- The class is instanciated with the appropriate section renderers
- The `parse` method is called to parse and render the sections inplace.

`docstring` **= list**

A list of strings (lines) that holds docstrings. The lines are changed inplace.

`index` **= int**

The zero-based line number of the docstring that is currently processed.

`sections` **= dict**

The sections that will be detected and rendered. The dictionary maps the section headers for detection to a tuple containing the section rendering function and optional values for the item renderer and parser.

Parameters

- **`lines`** (*`list`*) – The docstring as a list of strings where to render the sections
- **`sections`** (*`dict`*) – The sections that will be detected and rendered. The dictionary maps the section headers for detection to a tuple containing the section rendering function and optional values for the item renderer and parser. If on section rendering information is provided the default behaviour of the class is to render every section using the rubric rendering function.

`bookmark`()

append the current index to the end of the list of bookmarks.

`extract_items`(*item_type=None*)

Extract the section items from a docstring.

Parse the items in the description of a section into items of the provided item type. The method starts at the current line index position and checks if in the next two lines contain a valid item of the desired type. If successful, the lines that belong to the item description block (i.e. item header + item body) are popped out from the docstring and passed to the `item_type.parser` class method to get a new instance of `item_type`.

The process is repeated until there are no compatible `item_type` items found in the section or we run out of docstring lines, The collected item instances are returned

The exit conditions allow for two valid section item layouts:

1. No lines between items:

```
<header1>
    <description1>

    <more description>
<header2>
    <description2>
```

2. One line between items:

```
<header1>
    <description1>

    <more description>

<header2>
    <description2>
```

Parameters **`item_type`** (`Item`) – An Item type or a subclass. This argument is used to check if a line in the docstring is a valid item header and to parse the individual list items in the section. *`AnyItem`* will be used by default.

Returns **items** (*list*) – List of the collected item instances of *`Item`* type.

`get_next_block`()
Get the next item block from the docstring.

The method reads the next item block in the docstring. The first line is assumed to be the Item header and the following lines to belong to the definition body:

```
<header line>
    <definition>
```

The end of the field is designated by a line with the same indent as the field header or two empty lines in sequence.

`get_next_paragraph`()
Get the next paragraph designated by an empty line.

`goto_bookmark`(*bookmark_index=-1*)
Move to bookmark.

Move the current index to the docstring line given by the `self.bookmarks[bookmark_index]` and remove it from the bookmark list. Default value will pop the last entry.

Returns **bookmark** (*int*)

`insert_and_move`(*lines*, *index*)
Insert lines and move the current index to the end.

`insert_lines`(*lines*, *index*)
Insert lines in the docstring.

Parameters

- **`lines`** (*`list`*) – The list of lines to insert
- **`index`** (*`int`*) – Index to start the insertion

`is_section`()
Check if the current line defines a section.

`parse`()
Parse the docstring for sections.

The docstring is parsed for sections. If a section is found then the corresponding section rendering method is called.

`peek`(*ahead=0*)
Peek ahead a number of lines

The function retrieves the line that is ahead of the current index. If the index is at the end of the list then it returns an empty string.

Parameters **`ahead`** (*`int`*) – The number of lines to look ahead.

`pop`(*index=None*)
Pop a line from the dostrings.

`read`()
Return the next line and advance the index.

`remove_if_empty`(*index=None*)
Remove the line from the docstring if it is empty.

`remove_lines`(*index*, *count=1*)
Removes the lines from the docstring

`seek_to_next_non_empty_line`()
Goto the next non_empty line.

`docstring`
Get the docstring lines.

`eod`
End of docstring.

Sections

`sectiondoc.sections.`**`attributes`**(*doc*, *header*, *renderer=<class 'sectiondoc.renderers.attribute.Attribute'>*, *item_class=<class 'sectiondoc.items.definition_item.DefinitionItem'>*)
Render the attributes section to sphinx friendly format.

Parameters

- **`doc`** (`DocRender`) – The docstring container.
- **`header`** (*`string`*) – This parameter is ignored in this method.
- **`renderer`** (`Renderer`) – A renderer instance to render the items.
- **`item_class`** (*`type`*) – The item parser class to use. Default is `orDefinitionItem`.

`sectiondoc.sections.`**`notes_paragraph`**(*doc*, *header*, *renderer=None*, *item_class=None*)
Render the note section to use the rst `.. note` directive.

The section is expected to be given as a paragraph.

`sectiondoc.sections.`**`methods_table`**(*doc*, *header*, *renderer=<class 'sectiondoc.renderers.method.Method'>*, *item_class=<class 'sectiondoc.items.method_item.MethodItem'>*)
Render the methods section to sphinx friendly table format.

`sectiondoc.sections.`**`rubric`**(*doc*, *header*, *renderer=None*, *item_class=None*)
Refactor a header section using the rubric directive.

The method supports refactoring of single word headers, two word headers and headers that include a backslash ''''.

Parameters **`header`** (*`string`*) – The header string to use with the rubric directive.

`sectiondoc.sections.`**`arguments`**(*doc*, *header*, *renderer=<class 'sectiondoc.renderers.argument.Argument'>*, *item_class=<class 'sectiondoc.items.definition_item.DefinitionItem'>*)

Render the argument section to sphinx friendly format.

Parameters

- **`doc`** (`DocRender`) – The docstring container.
- **`header`** (*`string`*) – This parameter is ignored in this method.
- **`renderer`** (`Renderer`) – A renderer instance to render the items.
- **`item_class`** (*`type`*) – The item parser class to use. Default is `orDefinitionItem`.

`sectiondoc.sections.`**`item_list`**(*doc*, *header*, *renderer=<class 'sectiondoc.renderers.list_item.ListItem'>*, *item_class=<class 'sectiondoc.items.or_definition_item.OrDefinitionItem'>*)

Render the section to sphinx friendly item list.

Parameters

- **`doc`** (`DocRender`) – The docstring container.
- **`header`** (*`str`*) – The header name that is used for the fields (i.e. `:<header>:`).
- **`renderer`** (`Renderer`) – A renderer instance to render the items.
- **`item_class`** (*`type`*) – The item parser class to use. Default is *`OrDefinitionItem`*.

Items

class `sectiondoc.items.`**`OrDefinitionItem`**

A docstring definition section item.

In this section definition item there are two classifiers that are separated by `or`.

Syntax diagram:

```
+-------------------------------------------------+
| term [ " : " classifier [ " or " classifier] ]  |
+--+-------------------------------------------------+---+
   | definition                                       |
   | (body elements)+                                 |
   +--------------------------------------------------+
```

`term` = str

The term usually reflects the name of a parameter or an attribute.

`classifiers` = list

The classifiers of the definition. Commonly used to reflect the type of an argument or the signature of a function. Only two classifiers are accepted.

`definition` = list

The list of strings that holds the description the definition item.

Note: An Or Definition item is based on the item of a section definition list as it defined in restructured text (_http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#sections).

Create new instance of Item(term, classifiers, definition)

classmethod `is_item`(*line*)

Check if the line is describing a definition item.

The method is used to check that a line is following the expected format for the term and classifier attributes.

The expected format is:

```
+-------------------------------------------------+
| term [ " : " classifier [ " or " classifier] ]  |
+-------------------------------------------------+
```

Subclasses can restrict or expand this format.

classmethod `parse`(*lines*)

Parse a definition item from a set of lines.

The class method parses the definition list item from the list of docstring lines and produces a DefinitionItem with the term, classifier and the definition.

Note: The global indention in the definition lines is striped

---

The term definition is assumed to be in one of the following formats:

```
term
    Definition.
```

```
term
    Definition, paragraph 1.

    Definition, paragraph 2.
```

```
term : classifier
    Definition.
```

```
term : classifier or classifier
    Definition.
```

lines docstring lines of the definition without any empty lines before or after.

Returns definition (*OrDefinitionItem*)

class `sectiondoc.items.`**`DefinitionItem`**

A docstring definition section item.

In this section definition item, multiple classifiers can be provided as shown in the diagram below.

Syntax diagram:

```
+------------------------------+
| term [ " : " classifier ]*   |
+--+--------------------------+---+
   | definition                   |
   | (body elements)+             |
   +------------------------------+
```

term = str
The term usually reflects the name of a parameter or an attribute.

classifiers = list
The classifiers of the definition. Commonly used to reflect the type of an argument or the signature of a function. Multiple classifiers are allowed separated by colons ‘‘ : ‘‘.

definition = list
The list of strings that holds the description the definition item.

Create new instance of Item(term, classifiers, definition)

classmethod is_item(*line*)
Check if the line is describing a definition item.

The method is used to check that a line is following the expected format for the term and classifier attributes.

The expected format is:

```
+-----------------------------+
| term [ " : " classifier ]* |
+-----------------------------+
```

Subclasses can restrict or expand this format.

classmethod parse(*lines*)
Parse a definition item from a set of lines.

The class method parses the definition list item from the list of docstring lines and produces a DefinitionItem with the term, classifier and the definition.

Note: The global indention in the definition lines is striped

The term definition is assumed to be in one of the following formats:

```
term
    Definition.
```

```
term
    Definition, paragraph 1.

    Definition, paragraph 2.
```

```
term : classifier
    Definition.
```

```
term : classifier : classifier
    Definition.
```

lines docstring lines of the definition without any empty lines before or after.

Returns **definition** (*DefinitionItem*)

class sectiondoc.items.**MethodItem**
A MethodItem for method descriptions.

`term` **= str**
The term usually reflects the name of the method.

`classifiers` **= list**
The classifiers reflect the signature (i.e args and kwargs) of the method.

`definition` **= list**
The list of strings that holds the description the method item.

Create new instance of Item(term, classifiers, definition)

classmethod `is_item`(*line*)
Check if the definition header is a function signature.

The expected header has the following format:

```
+-------------------------------------------------+
| term "(" [ classifier [, classifier]* ] ")" |
+-------------------------------------------------+
```

classmethod `parse`(*lines*)
Parse a method definition item from a set of lines.

Parse the method signature and definition from the list of docstring lines and produce a MethodItem where the `term` is the method name and the classifier is arguments.

Note: The global indention in the definition lines is striped

The format of the method definition item is expected to be as follows:

```
+-------------------------------------------------+
| term "(" [ classifier [, classifier]* ] ")" |
+--+--------------------------------------+---+
   | definition                          |
   | (body elements)+                    |
   +--------------------------------------+
```

Parameters `lines` – docstring lines of the method definition item without any empty lines before or after.

Returns definition (*MethodItem*)

class `sectiondoc.items.`**`AnyItem`**
A docstring definition section item.

In this section item the are not restrictions on the classifier's section of the item header.

Syntax diagram:

```
+-------------------------------------------------+
| term [ " : " text ]                             |
+--+---------------------------------------------+---+
   | definition                                     |
   | (body elements)+                               |
   +-------------------------------------------------+
```

`term` **= str**
The term usually reflects the name of a parameter or an attribute.

classifiers = list
The classifiers of the definition. Commonly used to reflect the type of an argument or the signature of a function. Any text after the ‘ : ‘ till the end of the line is consider a single classifier.

definition = list
The list of strings that holds the description the definition item.

Note: AnyItem is probably closer to numpydoc on describing a section item.

Create new instance of Item(term, classifiers, definition)

classmethod is_item (*line*)
Check if the line is describing a definition item.

The method is used to check that a line is following the expected format for the term and classifier attributes.

The expected format is:

```
+-------------------------------------------------+
| term [ " : "  text ]                            |
+-------------------------------------------------+
```

Subclasses can restrict or expand this format.

classmethod parse (*lines*)
Parse a definition item from a set of lines.

The class method parses the definition list item from the list of docstring lines and produces a DefinitionItem with the term, classifier and the definition.

Note: The global indention in the definition lines is striped.

The term definition is assumed to be in one of the following formats:

```
term
    Definition.
```

```
term :
    Definition.
```

```
term
    Definition, paragraph 1.

    Definition, paragraph 2.
```

::

term: Definition, paragraph 1.

Definition, paragraph 2.

```
term : any text is valid here
    Definition.
```

lines docstring lines of the definition without any empty lines before or after.

Returns **definition** (*AnyItem*)

class `sectiondoc.items.`**`Item`**

A section item.

The Item class is responsible to check, parse a docstring item into a (term, classifiers, definition) tuple.

Format diagram:

```
+-------------------------------------------------+
| header                                          |
+--+----------------------------------------------+---+
   | definition                                       |
   | (body elements)+                                 |
   +--------------------------------------------------+
```

Depending only in the type of the list item the header is split into a term and one or more classifiers.

`term` = **str**

The term usually reflects the name of a parameter or an attribute.

`classifiers` = **list**

The classifier(s) of the term. Commonly used to reflect the type of an argument or the signature of a function.

`definition` = **list**

The list of strings that holds the description of the definition item.

Create new instance of Item(term, classifiers, definition)

classmethod **`is_item`**(*line*)

Check if the line is describing an item.

The method is used to check that a line is following the expected format for the `term` and `classifiers` attributes.

classmethod **`parse`**(*lines*)

Parse a definition item from a set of lines.

The class method parses the item from the list of docstring lines and produces a Item with the term, classifier and the definition.

Note: The global indention in the definition lines is striped

Parameters **`lines`** – docstring lines of the definition without any empty lines before or after.

Returns **item** (*Item*)

`mode`

Property (`string`), the operational mode of the item based on the available info. Possible values are `{'only_term', 'no_classifiers', 'no_definition', 'full'}`.

Renderers

class `sectiondoc.renderers.`**`Method`**(*item=None*)

Render method items as a table row.

to_rst (*columns=(0, 0)*)

Outputs definition in rst as a line in a table.

Parameters **columns** (*tuple*) – The two item tuple of column widths for the *:meth:* role column and the definition (i.e. summary) of the MethodItem

Note: The string attributes are clipped to the column width.

Example

```
>>> item = MethodItem('function', 'arg1, arg2',
... ['This is the best function ever.'])
>>> renderer = Method(item)
>>> renderer.to_rst(columns=(40, 20))
:meth:`function <function(arg1, arg2)>`  This is the best fun
```

class sectiondoc.renderers.**Argument** (*item=None*)

Render an item as a sphinx parameter role.

to_rst ()

Render an item as an argument using the `:param:` role.

Example

```
>>> item = Item('indent', 'int',
... ['The indent to use for the description block.',
     ''
     'This is the second paragraph of the argument definition.'])
>>> renderer = Argument(item)
>>> renderer.to_rst()
:param indent:
    The indent to use for the description block.
    This is the second paragraph of the argument definition.
:type indent: int
```

Note: There is no new line added at the last line of the `to_rst()` method.

class sectiondoc.renderers.**Renderer** (*item=None*)

An item renderer.

to_rst (***kwards*)

Outputs the *item* in sphinx friendly rst.

The method renders the passed into a list of lines that follow the rst markup.

Subclasses need to override the method to provide their custom made behaviour. However the signature of the method should hold only keyword arguments which always have default values.

Returns **lines** (*list*) – A list of string lines rendered in rst.

class sectiondoc.renderers.**Attribute** (*item=None*)

Render an Item instance using the sphinx attribute directive.

to_rst()

Return the attribute info using the attribute sphinx markup.

Examples

```
>>> item = Item('indent', 'int',
... ['The indent to use for the description block.'])
>>> Attribute(item).to_rst()
.. attribute:: indent
    :annotation: = `int`

    The indent to use for the description block
>>>
```

```
>>> item = Item('indent', '',
... ['The indent to use for the description block.'])
>>> Attribute(item).to_rst()
.. attribute:: indent

    The indent to use for the description block
>>>
```

Note: An empty line is added at the end of the list of strings so that the results can be concatenated directly and rendered properly by sphinx.

class sectiondoc.renderers.**ListItem**(*item=None*)

Rendered an item instance as an ordered/unordered list item.

to_rst(*prefix=None*)

Renders an item as items in an rst list.

Parameters **prefix** (*str*) – The prefix to use. For example if the item is part of an unnumbered list then `prefix='-'`.

Example

```
>>> item = Item('indent', 'int',
... ['The indent to use for the description block.'])
>>> renderer = ListItem(item)
>>> renderer.to_rst(prefix='-')
- **indent** (`int`) --
  The indent to use for the description block.
```

```
>>> item = Item('indent', 'int',
... ['The indent to use for'
     'the description block.'])
>>> renderer = ListItem(item)
>>> renderer.to_rst(prefix='-')
- **indent** (`int`) --
  The indent to use for
  the description block.
```

Note: An empty line is added at the end of the list of strings so that the results can be concatenated directly and rendered properly by sphinx.

class `sectiondoc.renderers.`**`TableRow`**(*item=None*)

Render an Item that represents a table line.

`to_rst`(*columns=(0, 0, 0)*)

Outputs definition in rst as a line in a table.

Parameters **`columns`** (*`tuple`*) – The three item tuple of column widths for the term, classifiers and definition fields of the TableLineItem. When the column width is 0 then the field is ignored.

Note:

- The strings attributes are clipped to the column width.

Example

```
>>> item = Item('function(arg1, arg2)', '',
... ['This is the best function ever.'])
>>> TableRow(item).to_rst(columns=(22, 0, 20))
function(arg1, arg2)   This is the best fun
```

class `sectiondoc.renderers.`**`Definition`**(*item=None*)

Render an Item instance as a sphinx definition term

`to_rst`(***kwards*)

Outputs the Item in sphinx friendly rst.

The method renders the `definition` into a list of lines that follow the rst markup of a sphinx definition item:

```
<term>

   (<classifier(s)>) --
   <definition>
```

Returns **lines** (*list*) – A list of string lines rendered in rst.

Example

```
>>> item = Item(
        'lines', 'list',
        ['A list of string lines rendered in rst.'])
>>> renderer = Definition(item)
>>> renderer.to_rst
lines

    *(list)* --
    A list of string lines rendered in rst.
```

Note: An empty line is added at the end of the list of strings so that the results can be concatenated directly and rendered properly by sphinx.

Authors

Ioannis Tziakos is the main developer and maintainer of the sectiondoc sphinx extension.

Historical notes:

The refactor_doc (original name of sectiondoc) extention started while working on the Enaml project with Chris Colbert, Robert Kern, Corran Webster, Tim Diller, and David Wyde at Enthought.

Many people at Enthought have provided feedback, given suggestions and fixes.

Change Log

Version 0.5.0dev

- The DefinitionItem now follows the rst description
- Implement a new AnyItem definition.
- Support rendering styles (#13)
- Fix documentation built
- Add *Parameters* section for class docstrings (#22)
- Rename package to sectiondoc (#16)
- Fix support and tests on Python 2.6 (#8)

Version 0.3.0

23/05/2014

- Support for Python 2.6 to 3.4 (#3, #4)
- Tests are run on TravisCI for all supported Python versions on Linux (#4)
- A setup.py file has been added to allow installable releases (#5)

Version 0.2

31/01/2012

- First draft of the documentation and rename to refactordoc
- Removed depedancy to docscrape.py
- Refactordoc is now a valid sphinx extention
- Factor out boilerplate code from refactoring methods to class methods.

- Factored out DefinitionItem class.
- Better test coverage.
- Code and Docstring cleanup.

Early Versions

26/10/2011

An early copy of the refactor_doc‘ can be found in the enaml documentation source directory. The module is named `enamldoc` and uses the Reader class that is in the docscrape.py file of the numpydoc package.

License

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index