RefactorDoc Documentation

Release 0.2

Ioannis Tziakos

February 10, 2016

Contents

[![Build Status](https://travis-ci.org/enthought/refactordoc.svg?branch=master)](https://travis-ci.org/enthought/refactordoc) [![Coverage Status](https://img.shields.io/coveralls/enthought/refactordoc.svg)](https://coveralls.io/r/entho

The RefactorDoc extension parses the function and class docstrings as they are retrieved by the autodoc extension and refactors 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.

Key aims of RefactorDoc are:

- Do not change the order of sections.
- Allow sphinx directives between (and inside) section blocks.
- Easier to debug (native support for debugging) and extend (future versions).

CHAPTER 1

Repository

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

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

CHAPTER 2

Installation

1. Install `refactordoc` from pypi using pip:

```
$ pip install reafactordoc
```

2. Add refactor-doc to the extensions variable of your sphinx `conf.py`:

```
extensions = [
    ...,
    'refactordoc',
    ...,
]
```

CHAPTER 3

Contents

3.1 Default refactoring

The base implementation of RefactorDoc provides refactoring for class and function doc-strings. A number of known (i.e. predefined) sections are processed by the ClassDoc and FunctionDoc classes and all unknown sections are re-factored using the `.. rubric::` directive by default.

For class objects the *ClassDoc* includes code to re-factor three types of sections.

Heading	Description	Item	Max	Rendered as
Methods	Class methods with summary	MethodItem	–	Table with links to the method
Attributes	Class attributes and their usage	Attribute	–	Sphinx attributes
Notes	Useful notes	paragraph	1	Note admonition

For function objects the *FunctionDoc* includes code to re-factor three types of sections.

Heading	Description	Item	Max	Rendered as
Arguments	function arguments and type	ArgumentItem	–	Parameters field list
Returns	Return value	ListItem	–	Unordered list
Raises	Raised exceptions	ListItem	–	Unordered list
Notes	Useful notes	paragraph	1	Note admonition

3.2 Usage rules

To be able to re-factor the sections properly the doc-strings should follow theses rules:

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 doc-string of the specific definition item class to have more information regarding the valid item header format.

3.3 Examples

3.3.1 Argument sections

```
Arguments
---------
new_lines : list
    The list of lines to insert

index : int
    Index to start the insertion
"""
```

BaseDoc.**insert_lines**(*lines*, *index*)

Insert refactored lines

Parameters

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

3.3.2 Method sections

```
Methods
-------
_refactor_attributes(self, header):
    Re-factor the attributes section to sphinx friendly format.

_refactor_methods(self, header):
    Re-factor the methods section to sphinx friendly format.

_refactor_notes(self, header):
    Re-factor the note section to use the rst ``.. note`` directive.
```

Note: The table that is created in this example does not have the links enabled because the methods are not rendered by autodoc (the `:no-members` option is set).

class refactordoc.class_doc.**ClassDoc**(*lines*, *headers=None*)

Docstring refactoring for classes.

The class provides the following refactoring methods.

Method	Description
`_refactor_attributes(self, header)`	Refactor the attributes section to sphinx friendly format.
`_refactor_methods(self, header)`	Refactor the methods section to sphinx friendly format.
`_refactor_notes(self, header)`	Refactor the note section to use the rst `.. note` directive.

Attribute sections

```
Attributes
----------
docstring : list
    A list of strings (lines) that holds doc-strings

index : int
    The current zero-based line number of the doc-string that is currently
    processed.

headers : dict
    The sections that the class re-factors. Each entry in the
    dictionary should have as key the name of the section in the
    form that it appears in the doc-strings. The value should be
    the postfix of the method, in the subclasses, that is
    responsible for refactoring (e.g. {'Methods': 'method'}).
```

class refactordoc.base_doc.**BaseDoc**(*lines*, *headers=None*)

Base abstract docstring refactoring class.

The class' main purpose is to parse the docstring and find the sections that need to be refactored. Subclasses should provide the methods responsible for refactoring the sections.

docstring = list

A list of strings (lines) that holds docstrings

index = int

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

headers = dict

The sections that the class will refactor. Each entry in the dictionary should have as key the name of the section in the form that it appears in the docstrings. The value should be the postfix of the method, in the subclasses, that is responsible for refactoring (e.g. {'Methods': 'method'}).

BaseDoc also provides a number of methods that operate on the docstring to help with the refactoring. This is necessary because the docstring has to change inplace and thus it is better to live the docstring manipulation to the class methods instead of accessing the lines directly.

Returns sections

```
Returns
-------
result : list
    A new list of left striped strings.
```

refactordoc.line_functions.**remove_indent**(*lines*)

Remove all indentation from the lines.

Returns **result** (*list*) – A new list of left striped strings.

Raises section

Notes

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

refactordoc.line_functions.**add_indent**(*lines*, *indent=4*)

Add spaces to indent a list of lines.

Parameters

- **lines** (*list*) – The list of strings to indent.
- **indent** (*int*) – The number of spaces to add.

Returns **lines** (*list*) – The indented strings (lines).

Note: Empty strings are not changed.

3.4 Architecture

The are three different parts in the pipeline of **refactordoc**.

1. The autodoc event hook and object refactoring dispatch;
2. The docstring section detection and method dispatching and;
3. The second component parsing and refactor of the detected sections;

3.4.1 The entry function

The entry function `setup` is located in the `__init__.py` file. Through the setup function refactor doc is loading the autodoc extention and hooks the `refactor_docstring()` function to the `autodoc-process-docstring` event.

The `refactor_docstring()` function receives the list of lines that compose the dostrings and based on the `object` initializes a new class instance to do the main work. The final item in the process is to execute the parse method of the created class.

3.4.2 The refactoring class

The refactoring classes are responsible for doing the actual work. These classes are derived from the `BaseDoc` class. After initialization refactoring takes place by executing the `parse()` method. The method looks for section headers by parsing the lines of the docstring. For each section that is discovered the `_refactor()` method is called with the name of the discovered section to dispatch processing to the associated refactoring method. The dispatcher constructs the name of the refactoring function by looking up the `headers` dictionary for a key equal to the header string found. If a key is found then the refactoring method name is composed from the prefix `_header_` and the retrieved value. If a key with the header name is not found then the default `_refactor_header()` is used.

3.4.3 The refactoring methods

Depending on the section the associated method parses and extracts the section definition block using the provided by the `BaseDoc` class utility methods.

When the definition block is a paragraph the `extract_paragraph()` will return the paragraph for further processing. When the definition block is a list of definition items. These items are parsed and extracted (i.e removed from the docstring) with the help of the `extract_items()` and a `DefintionItem` (or a subclass). The list of items that is returned holds all the information to produce a sequence of sphinx friendly rst.

After collecting the information in the section the refactoring method is ready to produce the updated rst and return a list of lines to the dispatching method so that they can be re-inserted in the docstring.

3.5 Section components

Each section is composed into a number of components these components are described below.

3.5.1 Section header

The start of the section is designated with the section header, which is a standard rst header. The underline is however restricted to using only − or =:

```
Section
-------
```

and:

```
Section
=======
```

Each section header is followed by a section definition block which can be either a list of items or one or more definition items. In general, The number and format of these items depends on the type of section that is currently parsed.

3.5.2 Definition list

Two of the most common formats are described

bellow:

The *standard definition item* format is based on the item of a variation of the definition list item as it defined in restructured text

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

where `<term>` is the single word (e.g. my_field) and `<definition>` is a indented block of rst code. The item header can optionally include the `<classifier>` attribute. This type of item is commonly used to describe class attributes and function arguments. In this documentation we will refer to this format as *variable* item to avoid confusion with sphinx directives.

A similar definition item format is the method item where the item header is composed of a function signature:

```
+----------------------------+
| term "(" [  classifier ] ")" |
+--+-------------------------+---+
   | definition                   |
```

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

This item is commonly used to describe provided functions (or methods) and thus is referred to as the *method item*. The `<classifier>` in this case is a list of arguments as it appears in the signature and `<definition>` the method summary (one sentence). All *method* fields should be separated by a single empty line.

3.5.3 Paragraph

Instead of a list of items the section can contain a paragraph:

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

This type of field is used for information sections like `Notes`.

Note: Currently the `<paragraph>` is a single unindented block with no empty lines. However, this will probably should change in future versions of RefactorDoc.

3.6 Building your own suite

While the default refactoring suite is enough for most cases. The user might need to extent the section repertoire, process other object types, allow more freedom in defining the definition list or restrict the docstring style to improve consinstancy through his code.

> **Warning:** All the methods below require to change the refactordoc code and even thought the changes might be small it is not considered the best way since updating refactordoc becomes non-trivial. Future version will remove this shortcoming.

3.7 Adding sections

New sections to be refactored can be simply added to the `headers` dictionary when an appropriate refactoring method exists. For example in the default suite that is shipped with refactordoc the *`FunctionDoc`* class sets the *Returns Raises* and *Yields* section to use the *_refactor_as_item_list* method in the class:

```
if headers is None:
    headers = {'Returns': 'as_item_list', 'Arguments': 'arguments',
               'Parameters': 'arguments', 'Raises': 'as_item_list',
               'Yields': 'as_item_list', 'Notes':'notes'}
```

When such a method does not

exist then the user has to augment the related class with that will parse and extract the section definition block(s) and return the refactored lines as a list of strings to replace the section in the docstring. The signature of the method should be `_header_<name>(self, header)`

Where `<name>` is the value in the `headers` that corresponds to the `header` string that is found in the docstring.

Note: More to come

3.8 Library Reference

The extension is separated into three main parts.

3.8.1 Sphinx extension

3.8.2 Refactor classes

class refactordoc.base_doc.**BaseDoc**(*lines*, *headers=None*)

Base abstract docstring refactoring class.

The class' main purpose is to parse the docstring and find the sections that need to be refactored. Subclasses should provide the methods responsible for refactoring the sections.

docstring = list

A list of strings (lines) that holds docstrings

index = int

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

headers = dict

The sections that the class will refactor. Each entry in the dictionary should have as key the name of the section in the form that it appears in the docstrings. The value should be the postfix of the method, in the subclasses, that is responsible for refactoring (e.g. {'Methods': 'method'}).

BaseDoc also provides a number of methods that operate on the docstring to help with the refactoring. This is necessary because the docstring has to change inplace and thus it is better to live the docstring manipulation to the class methods instead of accessing the lines directly.

bookmark()

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

docstring

Get the docstring lines.

eod

End of docstring.

extract_items(*item_class=None*)

Extract the definition items from a docstring.

Parse the items in the description of a section into items of the provided class time. Given a Definition-Item or a subclass defined by the `item_class` parameter. Staring from the current index position, the method checks if in the next two lines a valid header exists. If successful, then the lines that belong to the item description block (i.e. header + definition) are popped out from the docstring and passed to the `item_class` parser and create an instance of `item_class`.

The process is repeated until there is no compatible `item_class` found or we run out of docstring. Then the method returns a list of item_class instances.

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_class** (DefinitionItem) – A DefinitionItem or a subclass. This argument is used to check if a line in the docstring is a valid item and to parse the individual list items in the section. When None (default) the base DefinitionItem class is used.

Returns **parameters** (*list*) – List of the parsed item instances of item_class 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 DefinitionItem header and the following lines to belong to the definition:

```
<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 are found 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 my 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 refactored lines and move current index to the end.

insert_lines(*lines*, *index*)
Insert refactored lines

Parameters

- **new_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.

The docstring is parsed for sections. If a section is found then the corresponding refactoring 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.

class refactordoc.function_doc.**FunctionDoc**(*lines*, *headers=None*)
Docstring refactoring for functions

The class provides the following refactoring methods.

Method	Description
_refactor_arguments(self, header)	Refactor the Arguments and Parameters section to sphinx friendly format.
_refactor_as_items_list(self, header)	Refactor the Returns, Raises and Yields sections to sphinx friendly format.
_refactor_notes(self, header)	Refactor the note section to use the rst `.. note` directive.

_refactor_arguments(*header*)
Refactor the argument section to sphinx friendly format.

Parameters **header** (*unused*) – This parameter is ignored in thi method.

_refactor_as_item_list(*header*)
Refactor the a section to sphinx friendly item list.

Parameters **header** (*str*) – The header name that is used for the fields (i.e. `:<header>:`).

_refactor_notes(*header*)
Refactor the notes section to sphinx friendly format.

Parameters **header** (*unused*) – This parameter is ignored in this method.

class refactordoc.class_doc.**ClassDoc**(*lines*, *headers=None*)
Docstring refactoring for classes.

The class provides the following refactoring methods.

Method	Description
_refactor_attributes(self, header)	Refactor the attributes section to sphinx friendly format.
_refactor_methods(self, header)	Refactor the methods section to sphinx friendly format.
_refactor_notes(self, header)	Refactor the note section to use the rst `.. note` directive.

_get_column_lengths(*items*)
Helper function to estimate the column widths for the refactoring of the `Methods` section.

The method finds the index of the item that has the largest function name (i.e. self.term) and the largest signature. If the indexes are not the same then checks to see which of the two items have the largest string sum (i.e. self.term + self.signature).

_refactor_attributes(*header*)
Refactor the attributes section to sphinx friendly format.

_refactor_methods(*header*)
Refactor the methods section to sphinx friendly format.

_refactor_notes(*header*)
Refactor the note section to use the rst `.. note` directive.

3.8.3 Definition items

class refactordoc.definition_items.**ArgumentItem**
A definition item for function argument sections.

to_rst()
Render ArgumentItem in sphinx friendly rst using the `:param:` role.

Example

```
>>> item = ArgumentItem('indent', 'int',
... ['The indent to use for the description block.',
     ''
     'This is the second paragraph of the argument definition.'])
>>> item.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 refactordoc.definition_items.**AttributeItem**
Definition that renders the rst output using the attribute directive.

to_rst()
Return the attribute info using the attribute sphinx markup.

Examples

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

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

```
>>> item = AttributeItem('indent', '',
... ['The indent to use for the description block.'])
>>> 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 refactordoc.definition_items.**DefinitionItem**

A docstring definition item

Syntax diagram:

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

The Definition class is based on the nametuple class and is responsible to check, parse and refactor a docstring definition item into sphinx friendly rst.

term = str

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

classifier: str The classifier of the definition. Commonly used to reflect the type of an argument or the signature of a function.

Note: Currently only one classifier is supported.

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

Note: A 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).

classmethod is_definition(*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 subclass to 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.
```

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

Returns **definition** (*DefinitionItem*)

to_rst(***kwards*)

Outputs the Definition in sphinx friendly rst.

The method renders the definition into a list of lines that follow the rst markup. The default behaviour is to render the definition as an sphinx definition item:

```
<term>

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

Subclasses will usually override the method to provide custom made behaviour. However the signature of the method should hold only keyword arguments which have default values. The keyword arguments can be used to pass addition rendering information to subclasses.

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

Example

```
        >>> item = DefinitionItem('lines', 'list',
                                  ['A list of string lines rendered in rst.'])
        >>> item.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.

class refactordoc.definition_items.**ListItem**

A definition item that is rendered as an ordered/unordered list

to_rst(*prefix=None*)

Outputs ListItem in rst using as items in an list.

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

Example

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

```
        >>> item = ListItem('indent', 'int',
        ... ['The indent to use for'
             'the description block.'])
        >>> item.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 refactordoc.definition_items.**MethodItem**

A TableLineItem subclass to parse and render class methods.

classmethod **is_definition**(*line*)

Check if the definition header is a function signature.

classmethod **parse**(*lines*)

Parse a method definition item from a set of lines.

The class method parses the method signature and definition from the list of docstring lines and produces 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 method definition item is assumed to be as follows:

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

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

Returns **definition** (*MethodItem*)

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 strings attributes are clipped to the column width.

Example

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

class refactordoc.definition_items.**TableLineItem**

A Definition 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, classifier and definition fields of the TableLineItem. When the column width is 0 then the field

Note:

•The strings attributes are clipped to the column width.

Example

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

refactordoc.definition_items.**max_attribute_index** (*items*, *attr*)

Find the index of the attribute with the maximum length in a list of DefinitionItems.

Parameters

- **items** (*list*) – The list of the DefinitionItems (or subclasses).
- **attr** (*str*) – Attribute to look at.

refactordoc.definition_items.**max_attribute_length**(*items*, *attr*)
Find the max length of the attribute in a list of DefinitionItems.

Parameters

- **items** (*list*) – The list of the DefinitionItem instances (or subclasses).
- **attr** (*str*) – Attribute to look at.

3.8.4 Line functions

refactordoc.line_functions.**add_indent**(*lines*, *indent=4*)
Add spaces to indent a list of lines.

Parameters

- **lines** (*list*) – The list of strings to indent.
- **indent** (*int*) – The number of spaces to add.

Returns **lines** (*list*) – The indented strings (lines).

Note: Empty strings are not changed.

---

refactordoc.line_functions.**fix_backspace**(*word*)
Replace \ with \\ so that it will printed properly in the documentation.

refactordoc.line_functions.**fix_star**(*word*)
Replace * with * so that is will be parse properly by docutils.

refactordoc.line_functions.**fix_trailing_underscore**(*word*)
Replace the trailing _ with _ so that it will printed properly in the documentation.

refactordoc.line_functions.**get_indent**(*line*)
Return the indent portion of the line.

refactordoc.line_functions.**remove_indent**(*lines*)
Remove all indentation from the lines.

Returns **result** (*list*) – A new list of left striped strings.

refactordoc.line_functions.**replace_at**(*word*, *line*, *index*)
Replace the text in-line.

The text in line is replaced (not inserted) with the word. The replacement starts at the provided index. The result is cliped to the input length

Parameters

- **word** (*str*) – The text to copy into the line.
- **line** (*str*) – The line where the copy takes place.
- **index** (*int*) – The index to start coping.

Returns **result** (*str*) – line of text with the text replaced.

refactordoc.line_functions.**trim_indent**(*lines*)
Trim global intention level from lines.

3.9 Authors

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

3.9.1 Historical notes:

The refactor_doc 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.

3.10 Todos

Enhancements:

- Add error or warning messages when formating is wrong.
- Add conf.py configuration variable to define objects and the corresponding refactoring class
- Move DefinitionItem and subclasses to use templates similar to AttributeItem
- Allow DefinitionItem templates to be described in jinja2

3.11 Change Log

3.11.1 Version 0.3.1

23/05/2014

- Fix support and tests on Python 2.6 (#8)

3.11.2 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)

3.11.3 Version 0.2

31/01/2012

- First 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.

3.11.4 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.

3.12 License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

r

Symbols

A

B

C

D

E

F

G

H

I

L

M

P

R

S

T