Napoleon - *Marching toward legible docstrings*

Note

As of Sphinx 1.3, the napoleon extension will come packaged with Sphinx under *sphinx.ext.napoleon*. The *sphinxcontrib.napoleon* extension will continue to work with Sphinx <= 1.2.

Are you tired of writing docstrings that look like this:

```
:param path: The path of the file to wrap
:type path: str
:param field_storage: The :class:`FileStorage` instance to wrap
:type field_storage: FileStorage
:param temporary: Whether or not to delete the file when the File
   instance is destructed
:type temporary: bool
:returns: A buffered writable file descriptor
:rtype: BufferedFileStorage
```

ReStructuredText is great, but it creates visually dense, hard to read docstrings. Compare the jumble above to the same thing rewritten according to the Google Python Style Guide:

```
Args:
    path (str): The path of the file to wrap
    field_storage (FileStorage): The :class:`FileStorage` instance to wrap
    temporary (bool): Whether or not to delete the file when the File
       instance is destructed

Returns:
    BufferedFileStorage: A buffered writable file descriptor
```

Much more legible, no?

Napoleon is a Sphinx extension that enables Sphinx to parse both NumPy and Google style docstrings - the style recommended by Khan Academy.

Napoleon is a pre-processor that parses NumPy and Google style docstrings and converts them to reStructuredText before Sphinx attempts to parse them. This happens in an intermediate step while Sphinx is processing the documentation, so it doesn't modify any of the docstrings in your actual source code files.

Getting Started

1. Install the napoleon extension:

```
$ pip install sphinxcontrib-napoleon
```

2. After setting up Sphinx to build your docs, enable napoleon in the Sphinx *conf.py* file:

```
# conf.py

# Add autodoc and napoleon to the extensions list
extensions = ['sphinx.ext.autodoc', 'sphinxcontrib.napoleon']
```

3. Use *sphinx-apidoc* to build your API documentation:

```
$ sphinx-apidoc -f -o docs/source projectdir
```

Docstrings

Napoleon interprets every docstring that Sphinx autodoc can find, including docstrings on: `modules`, `classes`, `attributes`, `methods`, `functions`, and `variables`. Inside each docstring, specially formatted Sections are parsed and converted to reStructuredText.

All standard reStructuredText formatting still works as expected.

Docstring Sections

All of the following section headers are supported:

- `Args` *(alias of Parameters)*
- `Arguments` *(alias of Parameters)*
- `Attributes`
- `Example`
- `Examples`
- `Keyword Args` *(alias of Keyword Arguments)*
- `Keyword Arguments`
- `Methods`
- `Note`
- `Notes`
- `Other Parameters`
- `Parameters`
- `Return` *(alias of Returns)*
- `Returns`
- `Raises`
- `References`
- `See Also`
- `Todo`
- `Warning`
- `Warnings` *(alias of Warning)*
- `Warns`
- `Yield` *(alias of Yields)*
- `Yields`

Google vs NumPy

Napoleon supports two styles of docstrings: Google and NumPy. The main difference between the two styles is that Google uses indention to separate sections, whereas NumPy uses underlines.

Google style:

```python
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value

    """
    return True
```

NumPy style:

```python
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

NumPy style tends to require more vertical space, whereas Google style tends to use more horizontal space. Google style tends to be easier to read for short and simple docstrings, whereas NumPy style tends be easier to read for long and in-depth docstrings.

The Khan Academy recommends using Google style.

The choice between styles is largely aesthetic, but the two styles should not be mixed. Choose one style for your project and be consistent with it.

See also

For complete examples:

- Example Google Style Python Docstrings
- Example NumPy Style Python Docstrings

Type Annotations

PEP 484 introduced a standard way to express types in Python code. This is an alternative to expressing types directly in docstrings. One benefit of expressing types according to PEP 484 is that type checkers and IDEs can take advantage of them for static code analysis.

Google style with Python 3 type annotations:

```
def func(arg1: int, arg2: str) -> bool:
    """Summary line.

    Extended description of function.

    Args:
        arg1: Description of arg1
        arg2: Description of arg2

    Returns:
        Description of return value

    """
    return True
```

Google style with types in docstrings:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value

    """
    return True
```

Note

Python 2/3 compatible annotations aren't currently supported by Sphinx and won't show up in the docs.

Configuration

class `sphinxcontrib.napoleon.Config`(***settings*) [source]

Sphinx napoleon extension settings in *conf.py*.

Listed below are all the settings used by napoleon and their default values. These settings can be changed in the Sphinx *conf.py* file. Make sure that both “sphinx.ext.autodoc” and “sphinxcontrib.napoleon” are enabled in *conf.py*:

```
# conf.py

# Add any Sphinx extension module names here, as strings
extensions = ['sphinx.ext.autodoc', 'sphinxcontrib.napoleon']

# Napoleon settings
napoleon_google_docstring = True
napoleon_numpy_docstring = True
napoleon_include_init_with_doc = False
napoleon_include_private_with_doc = False
napoleon_include_special_with_doc = False
napoleon_use_admonition_for_examples = False
napoleon_use_admonition_for_notes = False
napoleon_use_admonition_for_references = False
napoleon_use_ivar = False
napoleon_use_param = True
napoleon_use_rtype = True
napoleon_use_keyword = True
```

`napoleon_google_docstring`

`bool` (Defaults to True) – True to parse Google style docstrings. False to disable support for Google style docstrings.

`napoleon_numpy_docstring`

`bool` (Defaults to True) – True to parse NumPy style docstrings. False to disable support for NumPy style docstrings.

`napoleon_include_init_with_doc`

`bool` (Defaults to False) – True to list `__init___` docstrings separately from the class docstring. False to fall back to Sphinx’s default behavior, which considers the `__init___` docstring as part of the class documentation.

If True:

```
def __init__(self):
    """
    This will be included in the docs because it has a docstring
    """

def __init__(self):
    # This will NOT be included in the docs
```

napoleon_include_private_with_doc

`bool` (Defaults to False) – True to include private members (like `_membername`) with docstrings in the documentation. False to fall back to Sphinx's default behavior.

If True:

```
def _included(self):
    """
    This will be included in the docs because it has a docstring
    """
    pass

def _skipped(self):
    # This will NOT be included in the docs
    pass
```

napoleon_include_special_with_doc

`bool` (Defaults to False) – True to include special members (like `__membername__`) with docstrings in the documentation. False to fall back to Sphinx's default behavior.

If True:

```
def __str__(self):
    """
    This will be included in the docs because it has a docstring
    """
    return unicode(self).encode('utf-8')

def __unicode__(self):
    # This will NOT be included in the docs
    return unicode(self.__class__.__name__)
```

napoleon_use_admonition_for_examples

`bool` (Defaults to False) – True to use the `.. admonition::` directive for the **Example** and **Examples** sections. False to use the `.. rubric::` directive instead. One may look better than the other depending on what HTML theme is used.

This NumPy style snippet will be converted as follows:

```
Example
-------
This is just a quick example
```

If True:

```
.. admonition:: Example

   This is just a quick example
```

If False:

```
.. rubric:: Example

This is just a quick example
```

`napoleon_use_admonition_for_notes`

`bool` (Defaults to False) – True to use the `.. admonition::` directive for **Notes** sections. False to use the `.. rubric::` directive instead.

> ❗ Note
>
> The singular **Note** section will always be converted to a `.. note::` directive.

> ❗ See also
>
> `napoleon_use_admonition_for_examples`

`napoleon_use_admonition_for_references`

`bool` (Defaults to False) – True to use the `.. admonition::` directive for **References** sections. False to use the `.. rubric::` directive instead.

> ❗ See also
>
> `napoleon_use_admonition_for_examples`

`napoleon_use_ivar`

`bool` (Defaults to False) – True to use the `:ivar:` role for instance variables. False to use the `.. attribute::` directive instead.

This NumPy style snippet will be converted as follows:

```
Attributes
----------
attr1 : int
    Description of `attr1`
```

If True:

```
:ivar attr1: Description of `attr1`
:vartype attr1: int
```

If False:

```
.. attribute:: attr1

   *int*

   Description of `attr1`
```

`napoleon_use_param`

`bool` (Defaults to True) – True to use a `:param:` role for each function parameter. False to use a single `:parameters:` role for all the parameters.

This NumPy style snippet will be converted as follows:

```
Parameters
----------
arg1 : str
    Description of `arg1`
arg2 : int, optional
    Description of `arg2`, defaults to 0
```

If True:

```
:param arg1: Description of `arg1`
:type arg1: str
:param arg2: Description of `arg2`, defaults to 0
:type arg2: int, optional
```

If False:

```
:parameters: * **arg1** (*str*) --
               Description of `arg1`
             * **arg2** (*int, optional*) --
               Description of `arg2`, defaults to 0
```

`napoleon_use_keyword`

`bool` (Defaults to True) – True to use a `:keyword:` role for each function keyword argument. False to use a single `:keyword arguments:` role for all the keywords.

This behaves similarly to `napoleon_use_param`. Note unlike docutils, `:keyword:` and `:param:` will not be treated the same way - there will be a separate "Keyword Arguments" section, rendered in the same fashion as "Parameters" section (type links created if possible)

See also

`napoleon_use_param`

`napoleon_use_rtype`

`bool` (Defaults to True) – True to use the `:rtype:` role for the return type. False to output the return type inline with the description.

This NumPy style snippet will be converted as follows:

```
Returns
-------
bool
    True if successful, False otherwise
```

If True:

```
:returns: True if successful, False otherwise
:rtype: bool
```

If False:

```
:returns: *bool* -- True if successful, False otherwise
```

Index