Style guide

This document describes the syntax and best practices for docstrings used with the numpydoc extension for [Sphinx](#).

> **ⓘ Note**
>
> For an accompanying example, see [example.py](#).
>
> Some features described in this document require a recent version of `numpydoc`. For example, the [Yields](#) section was added in `numpydoc` 0.6.

Overview

We mostly follow the standard Python style conventions as described here:

- [Style Guide for C Code](#)
- [Style Guide for Python Code](#)
- [Docstring Conventions](#)

Additional PEPs of interest regarding documentation of code:

- [Docstring Processing Framework](#)
- [Docutils Design Specification](#)

Use a code checker:

- [pylint](#): a Python static code analysis tool.
- [pyflakes](#): a tool to check Python code for errors by parsing the source file instead of importing it.
- [pycodestyle](#): (formerly `pep8`) a tool to check Python code against some of the style conventions in PEP 8.
- [flake8](#): a tool that glues together `pycodestyle`, `pyflakes`, `mccabe` to check the style and quality of Python code.
- [vim-flake8](#): a `flake8` plugin for Vim.

Import conventions

The following import conventions are used throughout the NumPy source and documentation:

```python
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Do not abbreviate `scipy`. There is no motivating use case to abbreviate it in the real world, so we avoid it in the documentation to avoid confusion.

Docstring Standard

A documentation string (docstring) is a string that describes a module, function, class, or method definition. The docstring is a special attribute of the object (`object.__doc__`) and, for consistency, is surrounded by triple double quotes, i.e.:

```python
"""This is the form of a docstring.

It can be spread over several lines.

"""
```

NumPy, [SciPy](#), and the scikits follow a common convention for docstrings that provides for consistency, while also allowing our toolchain to produce well-formatted reference guides. This document describes the current

📄 v: latest ▾

community consensus for such a standard. If you have suggestions for improvements, post them on the .

Our docstring standard uses re-structured text (reST) syntax and is rendered using Sphinx (a pre-processor that understands the particular documentation style we are using). While a rich set of markup is available, we limit ourselves to a very basic subset, in order to provide docstrings that are easy to read on text-only terminals.

A guiding principle is that human readers of the text are given precedence over contorting docstrings so our tools produce nice output. Rather than sacrificing the readability of the docstrings, we have written pre-processors to assist Sphinx in its task.

The length of docstring lines should be kept to 75 characters to facilitate reading the docstrings in text terminals.

Sections

The docstring consists of a number of sections separated by headings (except for the deprecation warning). Each heading should be underlined in hyphens, and the section ordering should be consistent with the description below.

The sections of a function's docstring are:

1. Short summary

A one-line summary that does not use variable names or the function name, e.g.

```
def add(a, b):
    """The sum of two numbers.

    """
```

The function signature is normally found by introspection and displayed by the help function. For some functions (notably those written in C) the signature is not available, so we have to specify it as the first line of the docstring:

```
"""
add(a, b)

The sum of two numbers.

"""
```

2. Deprecation warning

A section (use if applicable) to warn users that the object is deprecated. Section contents should include:

- In what NumPy version the object was deprecated, and when it will be removed.
- Reason for deprecation if this is useful information (e.g., object is superseded, duplicates functionality found elsewhere, etc.).
- New recommended way of obtaining the same functionality.

This section should use the `deprecated` Sphinx directive instead of an underlined section header.

```
.. deprecated:: 1.6.0
        `ndobj_old` will be removed in NumPy 2.0.0, it is replaced by
        `ndobj_new` because the latter works also with array subclasses.
```

3. Extended Summary

A few sentences giving an extended description. This section should be used to clarify *functionality*, not to discuss implementation detail or background theory, which should rather be explored in the Notes section below. You may refer to the parameters and the function name, but parameter descriptions still belong in the Parameters section.

4. Parameters

Description of the function arguments, keywords and their respective types.

```
Parameters
----------
x : type
    Description of parameter `x`.
y
    Description of parameter `y` (with type not specified).
```

Enclose variables in single backticks. The colon must be preceded by a space, or omitted if the type is absent.

For the parameter types, be as precise as possible. Below are a few examples of parameters and their types.

```
Parameters
----------
filename : str
copy : bool
dtype : data-type
iterable : iterable object
shape : int or tuple of int
files : list of str
```

If it is not necessary to specify a keyword argument, use `optional`:

```
x : int, optional
```

Optional keyword parameters have default values, which are displayed as part of the function signature. They can also be detailed in the description:

```
Description of parameter `x` (the default is -1, which implies summation
over all axes).
```

or as part of the type, instead of `optional`. If the default value would not be used as a value, `optional` is preferred. These are all equivalent:

```
copy : bool, default True
copy : bool, default=True
copy : bool, default: True
```

When a parameter can only assume one of a fixed set of values, those values can be listed in braces, with the default appearing first:

```
order : {'C', 'F', 'A'}
    Description of `order`.
```

When two or more input parameters have exactly the same type, shape and description, they can be combined:

```
x1, x2 : array_like
    Input arrays, description of `x1`, `x2`.
```

When documenting variable length positional, or keyword arguments, leave the leading star(s) in front of the name:

```
*args : tuple
    Additional arguments should be passed as keyword arguments
**kwargs : dict, optional
    Extra arguments to `metric`: refer to each metric documentation for a
    list of all possible arguments.
```

5. Returns

Explanation of the returned values and their types. Similar to the Parameters section, except the name of each return value is optional. The type of each return value is always required:

```
Returns
-------
int
    Description of anonymous integer return value.
```

If both the name and type are specified, the Returns section takes the same form as the Parameters section:

```
Returns
-------
err_code : int
    Non-zero value indicates error code, or zero on success.
err_msg : str or None
    Human readable error message, or None on success.
```

6. Yields

Explanation of the yielded values and their types. This is relevant to generators only. Similar to the Returns section in that the name of each value is optional, but the type of each value is always required:

```
Yields
------
int
    Description of the anonymous integer return value.
```

If both the name and type are specified, the Yields section takes the same form as the Returns section:

```
Yields
------
err_code : int
    Non-zero value indicates error code, or zero on success.
err_msg : str or None
    Human readable error message, or None on success.
```

Support for the Yields section was added in numpydoc version 0.6.

7. Receives

Explanation of parameters passed to a generator's `.send()` method, formatted as for Parameters, above. Since, like for Yields and Returns, a single object is always passed to the method, this may describe either the single parameter, or positional arguments passed as a tuple. If a docstring includes Receives it must also include Yields.

8. Other Parameters

An optional section used to describe infrequently used parameters. It should only be used if a function has a large number of keyword parameters, to prevent cluttering the Parameters section.

9. Raises

An optional section detailing which errors get raised and under what conditions:

```
Raises
------
LinAlgException
    If the matrix is not numerically invertible.
```

This section should be used judiciously, i.e., only for errors that are non-obvious or have a large chance of getting raised.

10. Warns

An optional section detailing which warnings get raised and under what conditions, formatted similarly to Raises.

11. Warnings

An optional section with cautions to the user in free text/reST.

12. See Also

An optional section used to refer to related code. This section can be very useful, but should be used judiciously. The goal is to direct users to other functions they may not be aware of, or have easy means of discovering (by looking at the module docstring, for example). Routines whose docstrings further explain parameters used by this function are good candidates.

As an example, for `numpy.mean` we would have:

```
See Also
--------
average : Weighted average.
```

When referring to functions in the same sub-module, no prefix is needed, and the tree is searched upwards for a match.

Prefix functions from other sub-modules appropriately. E.g., whilst documenting the `random` module, refer to a function in `fft` by

```
fft.fft2 : 2-D fast discrete Fourier transform.
```

When referring to an entirely different module:

```
scipy.random.norm : Random variates, PDFs, etc.
```

Functions may be listed without descriptions, and this is preferable if the functionality is clear from the function name:

```
See Also
--------
func_a : Function a with its description.
func_b, func_c_, func_d
func_e
```

If the combination of the function name and the description creates a line that is too long, the entry may be written as two lines, with the function name and colon on the first line, and the description on the next line, indented four spaces:

```
See Also
--------
package.module.submodule.func_a :
    A somewhat long description of the function.
```

13. Notes

An optional section that provides additional information about the code, possibly including a discussion of the algorithm. This section may include mathematical equations, written in [LaTeX](#) format:

```
Notes
-----
The FFT is a fast implementation of the discrete Fourier transform:

.. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}
```

Equations can also be typeset underneath the math directive:

```
The discrete-time Fourier time-convolution property states that

.. math::

    x(n) * y(n) \Leftrightarrow X(e^{j\omega } )Y(e^{j\omega } )\\
    another equation here
```

Math can furthermore be used inline, i.e.

```
The value of :math:`\omega` is larger than 5.
```

Variable names are displayed in typewriter font, obtained by using `\mathtt{var}`:

variable names are displayed in typewriter font, obtained by using `\mathtt{var}`.

```
We square the input parameter `alpha` to obtain
:math:`\mathtt{alpha}^2`.
```

Note that LaTeX is not particularly easy to read, so use equations sparingly.

Images are allowed, but should not be central to the explanation; users viewing the docstring as text must be able to comprehend its meaning without resorting to an image viewer. These additional illustrations are included using:

```
.. image:: filename
```

where filename is a path relative to the reference guide source directory.

14. References

References cited in the Notes section may be listed here, e.g. if you cited the article below using the text [1]_, include it as in the list as follows:

```
.. [1] O. McNoleg, "The integration of GIS, remote sensing,
   expert systems and adaptive co-kriging for environmental habitat
   modelling of the Highland Haggis using object-oriented, fuzzy-logic
   and neural-network techniques," Computers & Geosciences, vol. 22,
   pp. 585-588, 1996.
```

which renders as [1]:

[1]   O. McNoleg, "The integration of GIS, remote sensing, expert systems and adaptive co-kriging for environmental habitat modelling of the Highland Haggis using object-oriented, fuzzy-logic and neural-network techniques," Computers & Geosciences, vol. 22, pp. 585-588, 1996.

Referencing sources of a temporary nature, like web pages, is discouraged. References are meant to augment the docstring, but should not be required to understand it. References are numbered, starting from one, in the order in which they are cited.

> ⚠️ **Warning**
>
> **References will break tables**
>
> Where references like [1] appear in a tables within a numpydoc docstring, the table markup will be broken by numpydoc processing. See numpydoc issue #130

15. Examples

An optional section for examples, using the doctest format. This section is meant to illustrate usage, not to provide a testing framework — for that, use the `tests/` directory. While optional, this section is very strongly encouraged.

When multiple examples are provided, they should be separated by blank lines. Comments explaining the examples should have blank lines both above and below them:

```
Examples
--------
>>> np.add(1, 2)
3

Comment explaining the second example.

>>> np.add([1, 2], [3, 4])
array([4, 6])
```

The example code may be split across multiple lines, with each line after the first starting with '... ':

```
>>> np.add([[1, 2], [3, 4]],
...        [[5, 6], [7, 8]])
```

```
array([[ 6,  8],
       [10, 12]])
```

For tests with a result that is random or platform-dependent, mark the output as such:

```
>>> import numpy.random
>>> np.random.rand(2)
array([ 0.35773152,  0.38568979])  #random
```

You can run examples as doctests using:

```
>>> np.test(doctests=True)
>>> np.linalg.test(doctests=True)  # for a single module
```

In IPython it is also possible to run individual examples simply by copy-pasting them in doctest mode:

```
In [1]: %doctest_mode
Exception reporting mode: Plain
Doctest mode is: ON
>>> %paste
 import numpy.random
 np.random.rand(2)
## -- End pasted text --
array([ 0.8519522 ,  0.15492887])
```

It is not necessary to use the doctest markup `<BLANKLINE>` to indicate empty lines in the output. Note that the option to run the examples through `numpy.test` is provided for checking if the examples work, not for making the examples part of the testing framework.

The examples may assume that `import numpy as np` is executed before the example code in *numpy*. Additional examples may make use of *matplotlib* for plotting, but should import it explicitly, e.g., `import matplotlib.pyplot as plt`. All other imports, including the demonstrated function, must be explicit.

When matplotlib is imported in the example, the Example code will be wrapped in [matplotlib's Sphinx `plot` directive](). When matplotlib is not explicitly imported, .. *plot*:: can be used directly if *matplotlib.sphinxext.plot_directive* is loaded as a Sphinx extension in `conf.py`.

Documenting classes

Class docstring

Use the same sections as outlined above (all except `Returns` are applicable). The constructor (`__init__`) should also be documented here, the [Parameters]() section of the docstring details the constructor's parameters.

An **Attributes** section, located below the [Parameters]() section, may be used to describe non-method attributes of the class:

```
Attributes
----------
x : float
    The X coordinate.
y : float
    The Y coordinate.
```

Attributes that are properties and have their own docstrings can be simply listed by name:

```
Attributes
----------
real
imag
x : float
    The X coordinate.
y : float
    The Y coordinate.
```

In general, it is not necessary to list class methods. Those that are not part of the public API have names that start with an underscore. In some cases, however, a class may have a great many methods, of which only a few are relevant (e.g., subclasses of ndarray). Then, it becomes useful to have an additional **Methods** section:

```
class Photo(ndarray):
    """
    Array with associated photographic information.

    ...

    Attributes
    ----------
    exposure : float
        Exposure in seconds.

    Methods
    -------
    colorspace(c='rgb')
        Represent the photo in the given colorspace.
    gamma(n=1.0)
        Change the photo's gamma exposure.

    """
```

If it is necessary to explain a private method (use with care!), it can be referred to in the [Extended Summary](#) or the [Notes](#) section. Do not list private methods in the **Methods** section.

Note that *self* is *not* listed as the first parameter of methods.

Method docstrings

Document these as you would any other function. Do not include `self` in the list of parameters. If a method has an equivalent function (which is the case for many ndarray methods for example), the function docstring should contain the detailed documentation, and the method docstring should refer to it. Only put brief summary and [See Also](#) sections in the method docstring. The method should use a [Returns](#) or [Yields](#) section, as appropriate.

Documenting class instances

Instances of classes that are part of the NumPy API (for example *np.r_*, *np.c_*, *np.index_exp*, etc.) may require some care. To give these instances a useful docstring, we do the following:

- Single instance: If only a single instance of a class is exposed, document the class. Examples can use the instance name.
- Multiple instances: If multiple instances are exposed, docstrings for each instance are written and assigned to the instances' `__doc__` attributes at run time. The class is documented as usual, and the exposed instances can be mentioned in the [Notes](#) and [See Also](#) sections.

Documenting generators

Generators should be documented just as functions are documented. The only difference is that one should use the [Yields](#) section instead of the [Returns](#) section. Support for the [Yields](#) section was added in [numpydoc](#) version 0.6.

Documenting constants

Use the same sections as outlined for functions where applicable:

```
1. summary
2. extended summary (optional)
3. see also (optional)
4. references (optional)
5. examples (optional)
```

Docstrings for constants will not be visible in text terminals (constants are of immutable type, so docstrings can not be assigned to them like for for class instances), but will appear in the documentation built with Sphinx.

Documenting modules

Each module should have a docstring with at least a summary line. Other sections are optional, and should be used in the same order as for documenting functions when they are appropriate:

```
1. summary
2. extended summary
3. routine listings
4. see also
5. notes
6. references
7. examples
```

Routine listings are encouraged, especially for large modules, for which it is hard to get a good overview of all functionality provided by looking at the source file(s) or the `__all__` dict.

Note that license and author info, while often included in source files, do not belong in docstrings.

Other points to keep in mind

- Equations : as discussed in the <u>Notes</u> section above, LaTeX formatting should be kept to a minimum. Often it's possible to show equations as Python code or pseudo-code instead, which is much more readable in a terminal. For inline display use double backticks (like `y = np.sin(x)`). For display with blank lines above and below, use a double colon and indent the code, like:

  ```
  end of previous sentence::

      y = np.sin(x)
  ```

- Notes and Warnings : If there are points in the docstring that deserve special emphasis, the reST directives for a note or warning can be used in the vicinity of the context of the warning (inside a section). Syntax:

  ```
  .. warning:: Warning text.

  .. note:: Note text.
  ```

  Use these sparingly, as they do not look very good in text terminals and are not often necessary. One situation in which a warning can be useful is for marking a known bug that is not yet fixed.

- array_like : For functions that take arguments which can have not only a type *ndarray*, but also types that can be converted to an ndarray (i.e. scalar types, sequence types), those arguments can be documented with type *array_like*.

- Links : If you need to include hyperlinks in your docstring, note that some docstring sections are not parsed as standard reST, and in these sections, numpydoc may become confused by hyperlink targets such as:

  ```
  .. _Example: http://www.example.com
  ```

  If the Sphinx build issues a warning of the form `WARNING: Unknown target name: "example"`, then that is what is happening. To avoid this problem, use the inline hyperlink form:

  ```
  `Example <http://www.example.com>`_
  ```

Common reST concepts

For paragraphs, indentation is significant and indicates indentation in the output. New paragraphs are marked with a blank line.

Use `*italics*`, `**bold**` and ` ``monospace`` ` if needed in any explanations (but not for variable names and doctest code or multi-line code). Variable, module, function, and class names should be written between single back-ticks (`` `numpy` ``).

A more extensive example of reST markup can be found in <u>this example document</u>; the <u>quick reference</u> is useful while editing.

Line spacing and indentation are significant and should be carefully followed.

Conclusion

This document itself was written in ReStructuredText. [An example](#) of the format shown here is available.

Created using [Sphinx](#) 5.1.1.