The original source for these docstring standards is the NumPy project, and the associated numpydoc tools. The most up-to-date version of these standards can be found at numpy's github site. The guidelines below have been adapted to the Astropy package.

Overview

In general, we 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
- pyflakes
- pep8.py

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

```
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.

It is not necessary to do `import numpy as np` at the beginning of an example. However, some sub-modules, such as `fft`, are not imported by default, and you have to include them explicitly:

```
import numpy.fft
```

after which you may use it:

```
np.fft.fft2(...)
```

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.:

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

It can be spread over several lines.

"""
```

NumPy and SciPy have defined a common convention for docstrings that provides for consistency, while also allowing our toolchain to produce well-formatted reference guides. This format should be used for Astropy docstrings.

This 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 sections of the 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 Astropy 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 note Sphinx directive instead of an underlined section header.

   ```
   .. note:: Deprecated in Astropy 1.2
             `ndobj_old` will be removed in Astropy 2.0, 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`.
```

   Enclose variables in single backticks.

   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).
```

   When a parameter can only assume one of a fixed set of values, those values can be listed in braces:

```
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`.
```

5. **Returns**

   Explanation of the returned values and their types, of the same format as **parameters**.

6. **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.

7. **Raises**

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

```
Raises
------
InvalidWCSException
    If the WCS information is invalid.
```

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

8. **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 a hypothetical function `astropy.wcs.world2pix` converting sky to pixel coordinates, we would have:

```
See Also
--------
pix2world : Convert pixel to sky coordinates
```

   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 a hypothetical `astropy.vo` module, refer to a function in `table` by

```
table.read : Read in a VO table
```

   When referring to an entirely different module:

```
astropy.coords : Coordinate handling routines
```

   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
```

9. **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:

```
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}`:

```
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.

10. **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] 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.

11. **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:

    ```
    >>> astropy.wcs.world2pix(233.2, -12.3)
    (134.5, 233.1)

    Comment explaining the second example

    >>> astropy.coords.fk5_to_gal("00:42:44.33 +41:16:07.5")
    (121.1743, -21.5733)
    ```

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

    ```
    >>> astropy.coords.randomize_position(244.9, 44.2, radius=0.1)
    (244.855, 44.13)  # random
    ```

    It is not necessary to use the doctest markup `<BLANKLINE>` to indicate empty lines in the output. The examples may assume that `import numpy as np` is executed before the example code.

Documenting classes

Class docstrings

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 constructors parameters.

An `Attributes` section, located below the `Parameters` section, may be used to describe class variables:

```
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 Table(ndarray):
    """
    A class to represent tables of data

    ...

    Attributes
    ----------
    columns : list
        List of columns

    Methods
    -------
    read(filename)
        Read a table from a file
    sort(column, order='ascending')
        Sort by `column`
    """
```

If it is necessary to explain a private method (use with care!), it can be referred to in the **extended summary** or the **notes**. Do not list private methods in the Methods section.

Do not list `self` as the first parameter of a method.

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, 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.

Documenting class instances

Instances of classes that are part of the Astropy API 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 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

- 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.

- Questions and Answers : For general questions on how to write docstrings that are not answered in this document, refer to http://docs-old.scipy.org/numpy/Questions+Answers/.

- `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`.

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 `courier` if needed in any explanations (but not for variable names and doctest code or multi-line code). Variable, module and class names should be written between single back-ticks (`` `astropy` ``).

A more extensive example of reST markup can be found in this example document; the quick reference is useful

while editing.

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

Conclusion

An example of the format shown here is available. Refer to How to Build API/Reference Documentation on how to use Sphinx to build the manual.

Created using Sphinx 1.3.5. Last built 10 Mar 2016.