How To Generate API Documentation with Sphinx

Hans Petter Langtangen[1,2]

[1]Simula Research Laboratory
[2]University of Oslo

May 2, 2012

The current standard tool for documenting Python software is Sphinx. This tool was created to support hand-written documentation files in the reStructuredText (reST) format, but Sphinx also supports automatic generation of module or package documentation based on parsing function headers and extracting doc strings. We refer to such documentation as *API documentation*. For an example, see the documentation of the `numpy.polyfit` function http://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html.

There are two principal steps in making API documentation. First, write doc strings in all key classes, methods, and functions using the formatting described below. Second, copy the script make.py (view) to the directory where you have the source code, adapt the script by changing a few lines in the top of it, and run the code to generate the documentation (in the `API` subdirectory). The script automates the various steps in running Sphinx and preparing Sphinx files as described later.

We shall now briefly describe the reST format and show step by step how to create API documentation of Python modules. We follow the documentation rules of the `numpy` package. Sphinx version 1.1 or higher is then required. In addition, the `numpydoc` Sphinx extension must be installed. Download the `numpy` source tree, go to the top directory and perform

```
cd doc/sphinxext
sudo python setup.py install
```

0.1 Simple Formatting Rules

The reST format used by Sphinx and other popular tools in the Python community is a lightly tagged markup language, much less tagged than LaTeX and HTML. There is a Quick Start Guide for reST that gives a much broader overview than the brief description below. The Sphinx Quick Reference is also handy.

Paragraphs are separated by blank lines. Words in running text can be *emphasized*. Furthermore, text in double backquotes is typeset as code: `s = sin(r)`. Bullet lists start with a dash (-) and are indented, with a blank line before and after:

```
* a is the first parameter.

* b is the second parameter. An item can
  occupy multiple lines.

* c is the third parameter.
```

In description lists, where each item starts with a keyword, an item starts with the keyword, followed by a colon, and the text appears indented on the next line:

```
a:
  the first parameter

b:
  the second parameter. An item can
  occupy multiple
  lines.

c:
  the third parameter.
```

To make a section heading, just write the heading and use equal signs, on the line below the heading, for sections, and simple dashes for subsections (other choices of characters are also possible).

Mathematics. Mathematical formulas are typeset in LaTeX style inline. For example, $ax^2 + bx + c$ is written like

```
:math:`ax^2 + bx + c`
```

Two write an equation on a separate line, write

```
`.. math:: ax^2 + bx + c = 0
```

or

```
.. math::

   ax^2 + bx + c = 0
```

Remember to end the equation block with a blank line. Several equations can be aligned below each other by using `&` as alignment character:

```
.. math::

   ax^2 + bx + c &= 0,\\
   dx + e &= 0.
```

Code Snippets. To include a piece of code like

```
def roots(a, b, c):
    q = b**2 - 4*a*c
    root1 = (-b + sqrt(q))/float(2*a)
    root2 = (-b - sqrt(q))/float(2*a)
    return root1, root2
```

you can write it as

```
Here is an example::

        def roots(a, b, c):
            q = b**2 - 4*a*c
            root1 = -b + sqrt(q)/float(2*a)
            root2 = -b - sqrt(q)/float(2*a)
            return root1, root2
```

The code block must be intented, and the preceding line must end with a double colon. To specify the type of programming language and associated formatting (via the Pygments package), write

```
.. code-block:: python

        def roots(a, b, c):
            q = b**2 - 4*a*c
            root1 = -b + sqrt(q)/float(2*a)
            root2 = -b - sqrt(q)/float(2*a)
            return root1, root2
```

Interactive sessions and doctests can be inserted without colon and indentation of the code, but a blank line is needed before and after the interactive block.

```
Here is an example in an interactive Python shell.

>>> a = 1
>>> b = 2
>>> a + b
3

Note: the result is correct.
```

How to Format Doc Strings. Here is a function with a typical doc string formatted in `numpy` style.

```
# This is Python code
from numpy.lib.scimath import sqrt  # handles real and complex args

def roots(a, b, c, verbose=False):
    """
    Return the two roots in the quadratic equation::

      a*x**2 + b*x + c = 0

    or written with math typesetting
```

```
.. math:: ax^2 + bx + c = 0

The returned roots are real or complex numbers,
depending on the values of the arguments ‘a‘, ‘b‘,
and ‘c‘.

Parameters
----------
a: int, real, complex
   coefficient of the quadratic term
b: int, real, complex
   coefficient of the linear term
c: int, real, complex
   coefficient of the constant term
verbose: bool, optional
   prints the quantity ‘‘b**2 - 4*a*c‘‘ and if the
   roots are real or complex

Returns
-------
root1, root2: real, complex
    the roots of the quadratic polynomial.

Raises
------
ValueError:
    when ‘a‘ is zero

See Also
--------
:class:‘Quadratic‘: which is a class for quadratic polynomials
    that also has a :func:‘Quadratic.roots‘ method for computing
    the roots of a quadratic polynomial. There is also a class
    :class:‘~linear.Linear‘ in the module :mod:‘linear‘
    (i.e., :class:‘linear.Linear‘).

Notes
-----
The algorithm is a straightforward implementation of
a very well known formula [1]_.

References
----------
.. [1] Any textbook on mathematics or
       ‘Wikipedia <http://en.wikipedia.org/wiki/Quadratic_equation>‘_.

Examples
--------
>>> roots(-1, 2, 10)
(-5.3166247903553998, 1.3166247903553998)
>>> roots(-1, 2, -10)
((-2-3j), (-2+3j))

Alternatively, we can in a doc string list the arguments and
return values in a table

==========   =============   ================================
Parameter    Type            Description
==========   =============   ================================
a            float/complex   coefficient for quadratic term
b            float/complex   coefficient for linear term
c            float/complex   coefficient for constant term
r1, r2       float/complex   return: the two roots of
                             the quadratic polynomial
==========   =============   ================================
"""
if abs(a) < 1E-14:
    raise ValueError('a=%g is too close to zero' % a)

q = b**2 - 4*a*c
```

```
    if verbose:
        print 'q=%g: %s roots' % (q, 'real' if q>0 else 'complex')

    root1 = (-b + sqrt(q))/float(2*a)
    root2 = (-b - sqrt(q))/float(2*a)
    return root1, root2
```

Note the following:

1. Arguments to the functions and other variables are typeset in single backticks (normally translated to an italic font by Sphinx).
2. The headings `Parameters` (for function arguments), `Returns`, etc., are standard names and lead to a certain formatting of the doc string in HTML. The text following these headings are description lists. Sometimes a simpler formatting is convenient, e.g., a table or just running text explaining what the arguments and return values are.
3. One can make links to the documentation of other classes and functions as demonstrated under "See Also" (a tilde strips off the module prefix in the output).

0.2 Running Sphinx

We have made a complete example on making API documentation with Sphinx. The module files quadratic.py (view) and linear.py (view) contain examples of classes and a stand-alone functions with doc strings formatted as described above. The file make.py (view) runs (automatically) all the steps described below and creates HTML documentation of the two modules.

Make Sphinx Module Files. For each module file `module.py` you want include in the documentation, prepare a file `module.txt` containing

```
:mod:`module`
=============

.. automodule:: module
   :members:
   :undoc-members:
   :special-members:
   :inherited-members:
   :show-inheritance:
```

This specifications imply that the documentation will contain all member functions (not starting with an underscore) with doc strings (`:members:`), and those without doc strings (`:undoc-members:`), as well as all special methods (`:special-members:`), and all methods inherited from super classes (`:inherited-members:`). For the worked example we need to make the module files `src-sphinx_api/api/quadratic.txt` (view) and `src-sphinx_api/api/linear.txt` (view).

The name of modules in a subpackages must be listed with the full package path. For example, module `mod` in subpackage `s2` of subpackage `s1` is listed as

```
:mod:‘s1.s2.mod‘
================

.. automodule:: s1.s2.mod
```

in the file `mod.txt`. The `index.txt` file has a corresponding line with `mod` (which actually is the basename of the file `mod.txt` where the module `s1.s2.mod` is defined). For each of the `__init__.py` files in the packages one will normally make a `.txt` file with the package name, say `s2.txt`, where the first lines are:

```
:mod:‘s1.s2‘
============

.. automodule:: s1.s2
```

Create Sphinx Directory Tree. Sphinx needs a series of files that can be automatically generated by running

Terminal

```
sphinx-quickstart
```

and answering the questions. Specify a directory name as ”rooth path for the documentation”, say `api`, give the documentation a title, author, and version number. Make sure the extension of sphinx files is `.txt` and not `.rst`. If you make a fresh version of the documentation, remember to first delete the `api` directory. Move all the `module.txt` files to the `api` directory.

Make Index File. In the recently generated `api` directory, you must make an index file `index.txt` that lists the modules for which there exist `.txt` files . The `index.txt` file is automatically generated by `sphinx-quickstart`, but no modules are listed. Here is the typical look when it contains two modules `quadratic` and `linear`:

```
.. Docex Example documentation master file, created by
   sphinx-quickstart on Thu Feb 16 10:50:28 2022.
   ...

Welcome to Sphinx API Example’s documentation!
==============================================

Contents:

.. toctree::
   :maxdepth: 2
```

```
   quadratic
   linear

Indices and tables
==================

* :ref:‘genindex‘
* :ref:‘modindex‘
* :ref:‘search‘
```

Recall that each module listed in this file must have a corresponding `.txt` file as described above.

Edit the Configuration File. The `api` directory contains a file `conf.py` which allows you to configure a lot of features. You need to find the line with

```
#sys.path.insert(0, os.path.abspath('.'))
```

Uncomment this line and insert the directory where the modules reside, in this case the parent directory

```
sys.path.append(os.path.join(os.path.abspath(os.pardir)))
```

We also recommend to make use of more Sphinx extension modules. Find the line with `extensions =` and edit it to

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.mathjax',
    'sphinx.ext.viewcode',
    'numpydoc',
    'sphinx.ext.autosummary',
    'sphinx.ext.doctest',
    'sphinx.ext.inheritance_diagram']
```

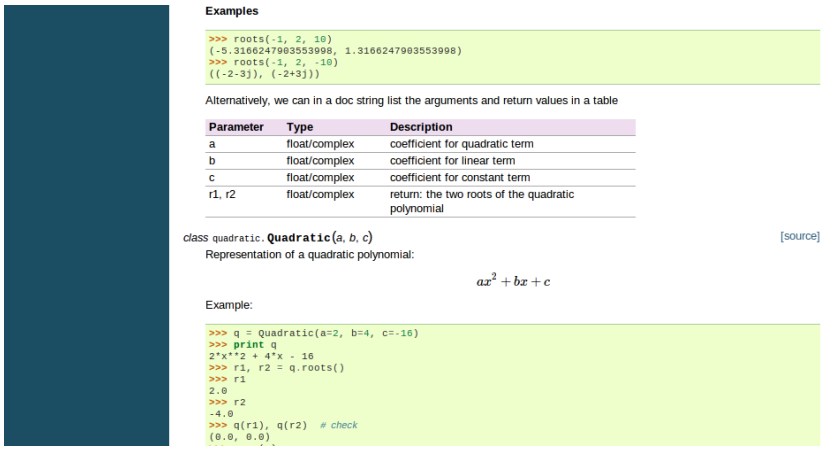
If not the `numpydoc` Sphinx extension is enabled, headings like `Parameters`, `Returns`, etc., are ignored and give rise to error messages ("Unexpected section title").

You may also want to add

```
extensions += [
    'matplotlib.sphinxext.only_directives',
    'matplotlib.sphinxext.plot_directive',
    'matplotlib.sphinxext.ipython_directive',
    'matplotlib.sphinxext.ipython_console_highlighting']
```

if `matplotlib` is installed.

Compile the Sphinx Document. You are now ready to compile an HTML version of the Sphinx documentation:



Examples

```
>>> roots(-1, 2, 10)
(-5.3166247903553998, 1.3166247903553998)
>>> roots(-1, 2, -10)
((-2-3j), (-2+3j))
```

Alternatively, we can in a doc string list the arguments and return values in a table

Parameter	Type	Description
a	float/complex	coefficient for quadratic term
b	float/complex	coefficient for linear term
c	float/complex	coefficient for constant term
r1, r2	float/complex	return: the two roots of the quadratic polynomial

class quadratic.**Quadratic**(*a*, *b*, *c*) [source]

Representation of a quadratic polynomial:

$$ax^2 + bx + c$$

Example:

```
>>> q = Quadratic(a=2, b=4, c=-16)
>>> print q
2*x**2 + 4*x - 16
>>> r1, r2 = q.roots()
>>> r1
2.0
>>> r2
-4.0
>>> q(r1), q(r2)  # check
(0.0, 0.0)
```

Figure 1: Snapshot of HTML documentation automatically generated by Sphinx.

Terminal

```
make html
```

This command results in a directory `_build/html` with a file `index.html` that can be loaded into a web browser for inspection.

The Python script make.py (view) automatically generates `.txt` index files for each `.py` file, runs `sphinx-quickstart`, copies index files to the new generated directory, edits `conf.py`, and runs `make html` to create the API documentation in HTML format. Examining the `make.py` script provides a complete recipe for getting started with Sphinx for automatically generating module and package documentation. The script can easily be applied to your own projects (it works without modifications if you want to document all `.py` files in a directory). in a directory)

To see the result of the generated documentation, invoke `api/_build/html/index.html`. Click around to see the various features, like the index, for instance. The layout and colors can be customized through different Sphinx *themes*, see the `api/conf.py` file. Several examples are provided in the examples directory.

Our example with the `quadratic` and `linear` modules is minimalistic. An excellent large-scale example on documenting a packing using Sphinx is found in the Matplotlib source (subdirectory `doc`). SciTools also applies Sphinx for documentation, and the file `doc/api/sphinx-src/00README` in the SciTools source tree explains the necessary steps in detail. Before diving into the documentation details of Matplotlib or SciTools, it will be advantageous to have digested some of the official Sphinx documentation, reached from `http://sphinx.pocoo.org/`.

0.3 Doconce Doc String Format

A disadvantage with the Sphinx format in doc strings is that it has quite some tagging that can be annoying when reading the doc strings directly, as done when invoking `pydoc` on the command line or `help(...)` or `object.__doc__` in interactive Python sessions. By writing the doc strings in Doconce format, one can transform the text both to Sphinx and to plain ASCII. That is, the doc strings looks nice in `pydoc` and in HTML.