Documenting Your Project Using Sphinx

This covers just a few of the many many commands available via sphinx. For more, visit http://sphinx.pc

Also, another great site with just an overview of more cc
http://docs.geoserver.org/trunk/en/docguide/sphinx.html.

Installing Sphinx

Try:

```
easy_install -U sphinx
```

Sphinx QuickStart

To get started, `cd` into the documentation directory and type:

```
$ sphinx-quickstart

Please enter values for the following settings
(just press Enter to accept a default value, if
one is given in brackets).
```

Here is a list of the default used in this project:

Prompt	**Choice**
> Root path for the documentation [.]:	<ENTER>
> Separate source and build directories (y/N) [n]:	y
> Name prefix for templates and static dir [_]:	<ENTER>
> Project name:	an_example_pypi_project
> Author name(s):	Andrew Carter
> Project version:	0.0.1
> Project release [0.0.1]:	<ENTER>
> Source file suffix [.rst]:	<ENTER>
> Name of your master document (without suffix) [index]:	<ENTER>
> autodoc: automatically insert docstrings from modules (y/N) [n]:	y
> doctest: automatically test code snippets in doctest blocks (y/N) [n]:	n
> intersphinx: link between Sphinx documentation of different projects (y/N) [n]:	y
> todo: write “todo” entries that can be shown or hidden on build (y/N) [n]:	n
> coverage: checks for documentation coverage (y/N) [n]:	n
> pngmath: include math, rendered as PNG images (y/N) [n]:	n
> jsmath: include math, rendered in the browser by JSMath (y/N) [n]:	n
> ifconfig: conditional inclusion of content based on config values (y/N) [n]:	y
> Create Makefile? (Y/n) [y]:	n
> Create Windows command file? (Y/n) [y]:	n

Then you should get:

```
Finished: An initial directory structure has been created.

You should now populate your master file .\source\index.rst and create other documentation
source files. Use the sphinx-build command to build the docs, like so:
   sphinx-build -b builder .\source .\build
where "builder" is one of the supported builders, e.g. html, latex or linkcheck.
```

`conf.py`

In your `doc/source` directory is now a python file called `conf.py`.

This is the file that controls the basics of how sphinx runs when you run a build. Here you can do this lik

- Change the version/release number by setting the `version` and `release` variables.
- Set the project name and author name.
- Setup a project logo.
- Set the default style to `sphinx` or `default`. Default is what the standard python docs use.

and much much more. Browsing through this file will give you an understanding of the basics.

reStructured Text (reST) Resources

Sphinx is built of reStructured text and, when using sphinx most of what you type is reStructured te
below (and most examples are ripped out of these pages):

- http://docutils.sourceforge.net/rst.html
- http://docutils.sourceforge.net/docs/user/rst/quickref.html
- http://docutils.sourceforge.net/docs/user/rst/cheatsheet.txt

Bold/Italics

Bold and italics are done like this:

```
**bold** and *italics*
```

which render like **bold** and *italics*.

Lists

You can do:

```
* A thing.
* Another thing.

or

1. Item 1.
2. Item 2.
3. Item 3.

or

- Some.
- Thing.
- Different.
```

which render as:

- A thing.
- Another thing.

or

1. Item 1.
2. Item 2.
3. Item 3.

or

- Some.
- Thing.
- Different.

Headers

It’s up to you to pick a convention for headers and just stick with it – Sphinx will pick up on your convent

You can do whatever header stragetgy you want, but I think a good one is:

```
H1 -- Top of Page Header
************************
There should only be one of these per page and this will also -- when
converting to pdf -- be used for the chapters.
```

```
H2 -- Page Sections
===================

H3 -- Subsection
----------------

H4 -- Subsubsection
+++++++++++++++++++
```

So:

```
A Subpoint
----------
This is my idea.

A subsubpoint
+++++++++++++
This is a smaller idea.
```

Is rendered like this:

A Subpoint

This is my idea.

A subsubpoint

This is a smaller idea.

Tables

Basic tables are done like this:

```
COMPLEX TABLE:

+------------+------------+-----------+
| Header 1   | Header 2   | Header 3  |
+============+============+===========+
| body row 1 | column 2   | column 3  |
+------------+------------+-----------+
| body row 2 | Cells may span columns.|
+------------+------------+-----------+
| body row 3 | Cells may  | - Cells   |
+------------+ span rows. | - contain |
| body row 4 |            | - blocks. |
+------------+------------+-----------+

SIMPLE TABLE:

=====  =====  ======
   Inputs     Output
------------  ------
  A      B    A or B
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
```

Which render like this:

COMPLEX TABLE:

Header 1	Header 2	Header 3
body row 1	column 2	column 3
body row 2	Cells may span columns.	
body row 3	Cells may span rows.	• Cells • contain • blocks.
body row 4		

SIMPLE TABLE:

Inputs		Output
A	**B**	**A or B**
False	False	False

True	False	True
False	True	True
True	True	True

Links

Urls are automatically linked, like http://packages.python.org/an_example_pypi_project/

For other links, you basically use the _ operator.

To add a text with a hyperlink, I like using this format:

```
`Docs for this project <http://packages.python.org/an_example_pypi_project/>`_
```

which renders as Docs for this project.

To create an anchor link that jumps to another section in the same .rst file, use this syntax:

```
`Table of Contents`_
```

Which renders like this: Table of Contents

Images

Images syntax is like this:

```
.. figure::  images/sweat.jpg
   :align:   center

   Proof that getting rich is mostly luck.
```

Which renders like this:

Proof that getting rich is mostly luck.

You can also add an anchor point for an image like this:

```
.. _is_sweaty:
.. figure::  images/sweat.jpg
   :align:   center

   Proof that getting rich is mostly luck.
```

Which renders like this:

Proof that getting rich is mostly luck.

Now you can reference this anchor like this:

```
This picture is_sweaty_.
```

which renders like this:

This picture is_sweaty.

Documents

To download documents you use the syntax:

```
:download:`An Example Pypi Project<docs/examplepypi.pdf>`
```

which renders like **An Example Pypi Project**.

Substitutions

Substitutions syntax is

```
.. |biohazard| image:: images/biohazard.png

The |biohazard| symbol must be used on containers used to dispose of medical waste.
```

Or if you want to do a literal text replacement use:

```
.. |doctest| replace:: :mod:`doctest`

I really like |doctest|.
```

Which renders like this:

The ☣ symbol must be used on containers used to dispose of medical waste.

I really like `doctest`.

Note: Substitutions are really useful, especially when put into a `global.rst` and included at the top o
below for more.

Includes

The syntax:

```
.. include myfile.rst
```

Will 'inline' the given file. A common convention I use is create a global .rst file called `global.rst` and
page. Very useful for links to common images or common files links, etc.

Table of Contents

Using the syntax:

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

   setuptools
   buildanduploadsphinx
```

renders like this:

- Getting Started With `setuptools` and `setup.py`
  - Installing setuptools and easy install
  - Setting up `setup.py`
  - Using `setup.py`
  - Intermezzo: .pypirc file and gpg
  - Registering Your Project
  - Uploading Your Project
  - Putting It All Together With The Full Windows Script

where `setuptools`, `sphinx`, etc. correspond to `setuptools.rst` and `sphinx.rst` files in the local path.

Paragraph Markup

To bring attention to a section of text, use paragraphs level markups.

Important constructs include:

```
.. note::

.. warning::

.. versionadded:: version

.. versionchanged:: version

.. seealso::
```

The way you would use this is as follows:

```
This is a statement.

.. warning::

   Never, ever, use this code!

.. versionadded:: 0.0.1

It's okay to use this code.
```

Which would render like this:

This is a statement.

Warning: Never, ever, use this code!

Newin version 0.0.1.

Now it is okay to use this code.

For full Sphinx docs on paragraph markup, check out http://sphinx.pocoo.org/markup/para.html.

Code

Python code in sphinx is easy. Because we turned **`pygments`** on, all we need to do is use the `::` operato

```
Here is something I want to talk about::

    def my_fn(foo, bar=True):
        """A really useful function.

        Returns None
        """
```

Renders like this:

Here is something I want to talk about:

```
def my_fn(foo, bar=True):
    """A really useful function.

    Returns None
    """
```

Also you can use the `::` operator to basically render any text exactly using fixed width fonts and by
engine. This is useful for ascii art:

```
       .;;.
     ,;;*;;;;;,
    .-'``;-');;.
   /'  .-.  /*;;
 .'    \d    \;;               .;;;,
/ o      `    \;    ,__.     ,;*;;;*;,
\__, _.__,'   \_.-') __)--.;;;;;*;;;;,
 `""`;;;\       /-')_) __)  `\' ';;;;;;
    ;*;;;        -') `)_)  |\ |  ;;;;*;
    ;;;;|        `---`    O | | ;;*;;;
    *;*;\|                 O  / ;;;;;*
   ;;;;;/|    .-------\      / ;*;;;;;
  ;;;*;/ \    |        '.   (`. ;;;*;;;
  ;;;;;'. ;   |          )   \ | ;;;;;;
  ,;*;;;;\/   |.        /   /` | ';;;*;
   ;;;;;;/    |/       /   /__/   ';;;
   '*jgs/     |       /    |     ;*;
        `"""""`       `"""""`    ;'
```

Also, you can add “inline” code by using the fixed-font operator.

By using two ` `` ` marks like this:

```
This is inline ``if __name__ == '__main__':``
```

you get:

This is inline `if __name__ == '__main__':`

Python Cross Referencing Syntax

Sphinx makes it easy to quickly link to other code definitions that *are in the python path* or can be four

The major ones I use all the time are:

- `:mod:`
- `:func:`
- `:class:`

Which is used like this:

```
I really like the :mod:`threading` module which has the
:class:`threading.Thread` class.

Here is a link :func:`time.time`.
```

which renders like this:

I really like the **`threading`** module which has the **`threading.Thread`** class.

Here is a link **`time.time()`**.

For more, visit http://sphinx.pocoo.org/markup/inline.html.

‘Auto’ Directives

From the sphinx docs directly:

`sphinx.ext.autodoc` – Include documentation from docstrings

This extension can import the modules you are documenting, and pull in documentation from docstring

Note: For Sphinx (actually, the Python interpreter that executes Sphinx) to find your module, it must b that the module or the package must be in one of the directories on `sys.path` – adapt your `sys.path` in accordingly.

For this to work, the docstrings must of course be written in correct reStructuredText. You can then use in the docstrings, and it will end up correctly in the documentation. Together with hand-written docum the pain of having to maintain two locations for documentation, while at the same time avoiding au documentation.

For more on autodoc see http://sphinx.pocoo.org/ext/autodoc.html.

The main autodoc features I use are:

- `.. automodule:: <module_name>`
- `.. autoclass:: <class_name>` and
- `.. autofunction:: <function_name>`

The key to using these features is the `:members:` attribute. If:

- You don't include it at all, only the docstring for the object is brought in:
- You just use `:members:` with no arguments, then all public functions, classes, and methods are bro
- If you explictly list the members like `:members: fn0, class0, _fn1` those explict members are brou

We'll examine these points in the full example Full Code Example.

Function Definitions

Function doc strings deserve a mention. The sphinx syntax (taken from http://sphinx.pocoo.org/markup

```
.. function:: format_exception(etype, value, tb[, limit=None])

   Format the exception with a traceback.

   :param etype: exception type
   :param value: exception value
   :param tb: traceback object
   :param limit: maximum number of stack frames to show
   :type limit: integer or None
   :rtype: list of strings
```

which renders like this:

`format_exception`(*etype*, *value*, *tb*[, *limit=None*])

Format the exception with a traceback.

Parameters:
- *etype* – exception type
- *value* – exception value
- *tb* – traceback object
- *limit* (integer or None) – maximum number of stack frames to show

Return type: list of strings

However, this can be a bit hard to read in the docstring itself (one line of thinking is that the sphinx ma that a user might read through the `__doc__` attribute). In fact the google styleguide.googlecode.com/svn/trunk/pyguide.html says doc string should look like this:

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}
        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
```

which I think is a lot cleaner when you just look at the docstring. It won't have the pretty sphinx formatti
example in Full Code Example below.

Full Code Example

The `an_example_pypi_project` contains

- An `__init__` file for the module.
- `useful_1.py` and `useful_2.py`. These files are IDENTICAL so I'll only reprint one here.
- The `code.rst` file which pulls it all together. This file lives in the doc directory.

Note: The idea behind the `auto` directives is to keep as much documentation in the code docstrings
Sphinx still aims to give you control not found when using real auto tools like doxygen or epydoc.

Therefore, that is why you need the small stub file `code.rst` to bascially act as a director for pulling the

Contents of `an_example_pypi_project.__init__`:

```
"""A pypi demonstration vehicle.

.. moduleauthor:: Andrew Carter <andrew@invalid.com>

"""

import useful_1
import useful_2


def start():
    "This starts this module running ..."
```

Contents of `an_example_pypi_project.useful_1`:

```
"""
.. module:: useful_1
   :platform: Unix, Windows
   :synopsis: A useful module indeed.

.. moduleauthor:: Andrew Carter <andrew@invalid.com>


"""

def public_fn_with_googley_docstring(name, state=None):
    """This function does something.

    Args:
       name (str):  The name to use.

    Kwargs:
       state (bool): Current state to be in.

    Returns:
       int.  The return code::

          0 -- Success!
          1 -- No good.
          2 -- Try again.

    Raises:
       AttributeError, KeyError

    A really great idea.  A way you might use me is

    >>> print public_fn_with_googley_docstring(name='foo', state=None)
    0

    BTW, this always returns 0.  **NEVER** use with :class:`MyPublicClass`.

    """
    return 0
```

```python
def public_fn_with_sphinxy_docstring(name, state=None):
    """This function does something.

    :param name: The name to use.
    :type name: str.
    :param state: Current state to be in.
    :type state: bool.
    :returns:  int -- the return code.
    :raises: AttributeError, KeyError

    """
    return 0

def public_fn_without_docstring():
    return True

def _private_fn_with_docstring(foo, bar='baz', foobarbas=None):
    """I have a docstring, but won't be imported if you just use ``:members:``.
    """
    return None


class MyPublicClass(object):
    """We use this as a public class example class.

    You never call this class before calling :func:`public_fn_with_sphinxy_docstring`.

    .. note::

       An example of intersphinx is this: you **cannot** use :mod:`pickle` on this class.

    """

    def __init__(self, foo, bar='baz'):
        """A really simple class.

        Args:
           foo (str): We all know what foo does.

        Kwargs:
           bar (str): Really, same as foo.

        """
        self._foo = foo
        self._bar = bar

    def get_foobar(self, foo, bar=True):
        """This gets the foobar

        This really should have a full function definition, but I am too lazy.

        >>> print get_foobar(10, 20)
        30
        >>> print get_foobar('a', 'b')
        ab

        Isn't that what you want?

        """
        return foo + bar

    def _get_baz(self, baz=None):
        """A private function to get baz.

        This really should have a full function definition, but I am too lazy.

        """
        return baz
```

And finally, contents of `code.rst` which pulls it all together:

```
Documentation for the Code
**************************

.. automodule:: an_example_pypi_project


useful #1 -- auto members
=========================

This is something I want to say that is not in the docstring.

.. automodule:: an_example_pypi_project.useful_1
   :members:

useful #2 -- explicit members
=============================

This is something I want to say that is not in the docstring.
```

```
.. automodule:: an_example_pypi_project.useful_2
   :members: public_fn_with_sphinxy_docstring, _private_fn_with_docstring

.. autoclass:: MyPublicClass
   :members: get_foobar, _get_baz
```

When you’re done, you get *Documentation for the Code*.