Brandon’s Sphinx Tutorial

Release 2013.0

Brandon Rhodes

March 02, 2017

Contents

PyCon 2013
San Jose, California
Thursday morning
March 14th

```
 9:00pm - 10:30pm   First Half of Tutorial
                    Break (refreshments served)
10:50pm - 12:20pm   Conclusion of Tutorial
```

Welcome to my Sphinx tutorial, which is now in its fourth year at PyCon. Sphinx has come a long way since this tutorial was first offered back on a cold February day in 2010, when the most recent version available was 0.6.4. Sphinx has now reached 1.1.3, and I have worked to keep this tutorial up to date with all of the most recent features in Sphinx. I hope you enjoy it with me!

Chapter 1

Notes on Using Sphinx

Here are some quick notes on running Sphinx successfully. Each topic will be elaborated upon at the right point during our class.

Starting a Sphinx project

The wonder of a properly designed framework is that it begins by positioning you at a working starting point instead of leaving you to wander endlessly through a `README` that, after dozens of steps, leaves you guessing which step you did improperly to have wound up with a broken install.

Sphinx gets you started with `sphinx-quickstart`. Here is how a quick-start session will look, paying attention only to its prompts and how you should respond (which is mostly by pressing Return over and over), when you use it to create a new project:

```
$ sphinx-quickstart
Welcome to the Sphinx quickstart utility...

> Root path for the documentation [.]: doc
> Separate source and build directories (y/N) [n]:
> Name prefix for templates and static dir [_]:
> Project name: trianglelib
> Author name(s): Brandon
> Project version: 1.0
> Project release [1.0]:
> Project language [en]:
> Source file suffix [.rst]: .rst
> Name of your master document (without suffix) [index]: index
> Do you want to use the epub builder (y/N) [n]: n
> autodoc: automatically insert docstrings ... (y/N) [n]: y
> doctest: automatically test code snippets ... (y/N) [n]: y
> intersphinx: ... (y/N) [n]:
> todo: ... (y/N) [n]:
> coverage: ... (y/N) [n]:
> pngmath: ... (y/N) [n]:
> mathjax: ... (y/N) [n]: y
> ifconfig: ... (y/N) [n]:
> viewcode: include links to the source code ... (y/N) [n]: y
> githubpages: ... (y/n) [n]:
```

```
> Create Makefile? (Y/n) [y]: y
> Create Windows command file? (Y/n) [y]: y
```

Sphinx layout

After you have succeeded in quick-starting, your project should look something like this, if we imagine that you created your `doc` Sphinx directory right next to your `trianglelib` Python package directory:

```
your-project/
|-- doc/
|   |-- Makefile
|   |-- _build/
|   |-- _static/
|   |-- _templates/
|   |-- conf.py
|   |-- index.rst
|   `-- make.bat
|-- setup.py
`-- trianglelib/
    |-- __init__.py
    |-- shape.py
    `-- utils.py
```

The `index.rst` is your initial documentation file, whose table of contents you will expand as you add additional `.rst` files to this directory.

Hints

Here are a few adjustments you can make to a Sphinx project once you have its files laid out and set up.

- Sphinx is sensitive to indentation — blocks and code snippets end when the indentation level returns to its previous level — but Sphinx is usually forgiving about how far exactly you indent; you can generally choose freely how far to indent, so long as you are then consistent about sticking to that level. Sphinx is much like Python in this regard, but without a strong community preference for a particular number of spaces per indentation level.
- Build your documentation by changing directory to the directory that contains the `Makefile` and then running:

```
    make html
```

- You can view the documentation by running Python's built-in web server, opening `http://localhost:8000/` in your web browser, and navigating into the `_build/html` directory:

```
python -m SimpleHTTPServer
```

- Feel free to create directories beneath `doc` for groups of `.rst` files that do not belong at the top level. Simply refer to such files in your `index.rst` table of contents using a `dir/filename` syntax so that Sphinx can find them.
- You can get rid of the line that says "Contents:" in `index.rst` since it causes a bizarre blank page in the PDF output.
- To make prettier PDFs, remove the whole section in `index.rst` named "Indices and tables."
- Do not despair if you realize later that you need an extension that you failed to answer `y` to during the quick-start; simply go into the `conf.py` and add the extension to the `extensions` list and make any other changes the extension needs. You can always simply re-run the quick start to make a new project with the extension active, then run `diff` between your own `Makefile` and the new one to see what the differences are.
- You can use Semantic Newlines to be friendly to your version control.
- Here is a small diagram of how I think of documentation, which we will use as a rough guide during class:

```
                     developer              developer
new user          writing new code        reading code
   |                     |                      |
   v                     v                      v
tutorial  <-->   topical guides   <-->   API reference
```

Helping autodoc find your package

In order to run `autodoc` directives on your package, the Python interpreter that is running Sphinx for you needs to be able to import your package. You can test whether Python can see your package by testing whether this command returns without error:

```
python -c 'import your_package'
```

There are three general approaches to making your package available to `autodoc`.

1. Have your package's top-level directory sit right next to your Sphinx `Makefile` and `conf.py` and all of your top-level RST text files. When you type `make` inside this directory and it goes off and runs Sphinx, your package will be visible because it is sitting in the current working directory.

The downside: you usually want your package sitting out by itself in your source distribution, not mixed in or sitting beneath its own documentation.

2. Set the Python path environment variable to point to the directory containing your package. You can do this with an `export` statement that you run before you start building your documentation:

```
export PYTHONPATH=/home/brandon/trianglelib
```

Or you can set the Python path only for the build command itself, leaving your shell variable settings pristine:

```
PYTHONPATH=/home/brandon/triangle-project make html
```

The downside: You either have to remember to manually set this environment variable each time you run Sphinx, or you have to create and maintain a small shell script as a separate file that will remember to set the path and run Sphinx.

3. If you have installed Sphinx inside a virtual environment — which is a really, really great idea — then you can install your under-development package there too by using the pip `--editable` flag:

```
pip install -e /home/brandon/triangle-project
```

Once you have run this command, the Python running inside of this virtual environment is permanently able to `import trianglelib` without further ado. (Assuming that you do not remove the project from your filesystem!)

The downside: When you check the project out on to a fresh machine, you either have to always remember to manually set up the virtual environment the right way, or you have to keep a shell script in the repository that sets it up for you each time. (Even though that is a good idea anyway.)

4. Assuming that your package and its documentation are part of the same source repository — as they should be — they will always have the same relative position on the filesystem. In this case, you can simply edit the Sphinx `conf.py` so that its `sys.path` configuration entry points at the relative position of your package:

```
sys.path.append(os.path.abspath('../triangle-project'))
```

All upside: this is, in my opinion, the best approach, as it always goes along for the ride with your repository, and works immediately upon repository check-out without having to rely on any intermediate setup steps.

Deployment

We will discuss this topic in depth, but here are some links for your further reference when the class is complete:

- It should be possible to export the contents of `_build/html` to any file-system-based web service and serve it as static content.
- You can package the documentation in a ZIP file and upload it using the "edit" page for your Python package, and it will appear at the URL:

  http://pythonhosted.org/<project-name>

  Detailed instructions for this procedure live at:

  http://pythonhosted.org/
- The powerful and popular Read the Docs service lets you configure your GitHub repository so that every time you push a new version of your software, the documentation gets automatically rebuilt and made available at:

  https://readthedocs.org/projects/<project-name>/

  Read the Docs also supports custom host names if you want your documentation to appear beneath your own project sub-domain. More information is available at:

  https://readthedocs.org/
- Creating a PDF is nearly as simple as running:

```
make html
```

  Except that you have to have the Latex typesetting system installed, which is a daunting task on many platforms and operating systems. On my own Ubuntu Linux laptops, I need to install several packages before even attempting it:

```
texlive-fonts-recommended
texlive-latex-recommended
texlive-latex-extra
```

- See the Sphinx documentation for several other supported formats!
- We will tackle simple theming tasks during the tutorial's second half; remember that the PyEphem project is a good living example of how to completely replace the Sphinx HTML themes with one of your own, so that you are essentially using Sphinx to build your own web site.

Chapter 2

RST Quick Reference

```
Underline titles with punctuation
=================================

For subtitles, switch to another punctuation mark
-------------------------------------------------

*Italic* **bold** ``name`` ``function()`` ``expression = 3 + 3``
`Hyperlink <http://en.wikipedia.org/wiki/Hyperlink>`_ `Link`_

.. _Link: http://en.wikipedia.org/wiki/Link_(The_Legend_of_Zelda)
.. image:: images/python-logo.png
.. A comment block starts with two periods, can continue indented.

A paragraph is one or more lines of un-indented text, separated
from the material above and below by blank lines.

    “Block quotes look like paragraphs, but are indented with
    one or more spaces.”

| Because of the pipe characters, this will become one line,
| And this will become another line, like in poetry.

term
  Definition for the “term”, indented beneath it.
another term
  And its definition; any of these definitions can continue on for
  several lines by -- you guessed it! -- being similarly indented.

* Each item in a list starts with an asterisk (or “1.”, “a.”, etc).
* List items can go on for several lines as long as you remember to
  keep the rest of the list item indented.

Code blocks are introduced by a double-colon and are indented::

    import docutils
    print help(docutils)

>>> print 'But doctests start with ">>>" and need no indentation.'
```

Chapter 3

Sphinx Quick Reference

```
.. note::

   Your note should consist of one or more paragraphs, all indented
   so that they clearly belong to the note and not to the text or
   directive that follows.

   Many other directives are also supported, including: warning,
   versionadded, versionchanged, seealso, deprecated, rubric,
   centered, hlist, glossary, productionlist.

.. code-block:: c

   /* Or say "highlight::" once to set the language for all of the
      code blocks that follow it.  Options include ":linenos:",
      ":linenothreshold:", and ":emphasize-lines: 1,2,3". */

   char s[] = "You can also say 'python', 'ruby', ..., or 'guess'!";

.. literalinclude:: example.py
   :lines: 10-20
   :emphasize-lines: 15,16

.. module:: httplib

.. class:: Request

   Zero or more paragraphs of introductory material for the class.

   .. method:: send()

      Description of the send() method.

   .. attribute:: url

      Description of the url attribute.

   Many more members are possible than just method and attribute,
   and non-Python languages are supported too; see the Sphinx docs
   for more possibilities!
```

```
.. testcode::

   print 'The doctest extension supports code without >>> prompts!'

.. testoutput::

   The doctest extension supports code without >>> prompts!

.. _custom-label:
.. index:: single: paragraph, targeted paragraph, indexed paragraph

This paragraph can be targeted with :ref:`custom-label`, and will also
be the :index:`target` of several index entries!

.. index:: pair: copper, wire

This paragraph will be listed in the index under both “wire, copper”
and “copper, wire.”  See the Sphinx documentation for even more complex
ways of building index entries.

Many kinds of cross-reference can be used inside of a paragraph:

:ref:`custom-label`              :class:`~module.class`
:doc:`quick-sphinx`              :attr:`~module.class.method()`
:mod:`module`                    :attr: `~module.class.attribute`

(See the Sphinx “Inline Markup” chapter for MANY more examples!)
```

Chapter 4

Writing 'api.rst'

Your first assignment is to create your own `api.rst` document whose output looks just like the chapter "The trianglelib API reference" at the end of this handout!

Approach this task as three smaller steps. For your first try, just use three plain autodoc directives to generate your entire chapter from the contents of the `trianglelib` docstrings. Separate the three `autodoc` directives with sub-titles to make the chapter more organized:

```
The trianglelib API reference
=============================

.. automodule:: trianglelib
   :members:

The "shape" module
------------------

.. automodule:: trianglelib.shape
   :members:

The "utils" module
------------------

.. automodule:: trianglelib.utils
   :members:
```

Add `api` to the list of chapters in your table of contents in the `index.rst` file, and then build the resulting document. Note that you will get errors if Sphinx cannot find the `trianglelib` Python module, in which case you should consult the solutions in the Notes on Using Sphinx chapter of this document.

Once your build is successful, take a look at your output document. It should already contain most of the information that you need! Compare the output carefully with the corresponding example bound with this documentation. What changes will you have to learn how to make in order to get the two documents to match?

For example: where is the text describing how to instantiate `Triangle`? You will need to add it manually, because `autodoc` does not pull that information from your code. So you will need to take control of that class's presentation. To take control, switch `shape` from the using the `automodule::` directive to using the plain old `module::` directive, which produces *no output* of its own but makes you explicitly provide its contents yourself. This frees you up to build your own version of `Triangle` that includes some introductory text before getting to work on its members:

```
.. module:: trianglelib.shape

.. autoclass:: Triangle
   :members:

   <Describe instantiation here!>
```

Note that the name of the class `Triangle` does not have to be prefixed with `trianglelib.shape.` because the `module::` directive had already told Sphinx about where to look for the classes and methods that you describe in this file.

You are getting closer, but the documentation still does not look exactly the same. For one thing, the first method shown right now is `area()` since functions and attributes are shown in alphabetical order by default. Try each of the following directives in turn to achieve some other possible orderings:

```
:member-order: bysource

:member-order: groupwise

:members: is_equilateral, is_isosceles, ...
```

Note that you can also say `:exclude-members:` followed by the names of methods to leave out.

At that point your API chapter will begin to strongly resemble the printed one! Here are further bonus goals in case you finish early:

1. How can you display the attributes `a`, `b`, and `c` as presented in the printed document?
2. The printed chapter describes the triangle `__eq__()` method by actually showing two triangles compared with the == operator; can you get your version of the document to show the same thing?
3. If you have not done so already, add in the example doctest that stands just beneath the instantiation instructions in the printed version of the chapter.
4. Try running `make doctest` — are your code samples correct? Add some deliberate errors into the code to see what the output looks like when doctests fail.
5. Create example doctests for a few of the functions in `utils` by turning off `automodule` for the `utils` module, explicitly autodoc'ing each of its five functions to pull them back into your documentation, and adding example code beneath each one.

Chapter 5

Writing 'tutorial.rst'

Having written our low-level API documentation, we are now going to turn to the first thing a new user reads: your tutorial!

The tutorial is short to keep this tutorial from going too long, but it does try to both serve the real purpose of a tutorial, which is to show users how to get started with your library, and it also should help you practice reStructuredText.

Remember to add your tutorial document's filename to the table of contents in your `index.rst` file!

While plain `names` and `attributes` are simply displayed in a typewriter font, `functions()` and `methods()` should be followed by a pair of parenthesis to make them look right.

Your task is to make the tutorial's text look as pretty as it does in the printed copy here in this handout. Here are some bonus goals if you finish early:

1. Do you know whether the sample program really works? Add a deliberate mistake to it and try running `make doctest` to see if you really get presented with an error. If no error appears, then consult the previous Sphinx Quick Reference chapter and try to figure out how to mark up the code and output so that they get tested.
2. Whenever a module, class, or function is mentioned in the text, you should mark it to create a cross-reference into the API document you have already written. Try out the following two styles of cross-reference and see what difference they make in your formatted output:

```
the :class:`trianglelib.shape.Triangle` class
the :class:`~trianglelib.shape.Triangle` class
```

3. If you managed to get the quote from Euclid to appear at the top of your tutorial, how might you make it appear in the index under the entry "Euclid"?
4. In the sentence "Read *The trianglelib guide* to learn more", we want the phrase "The trianglelib guide" to become an actual hyperlink to the guide itself. Create a nearly empty `guide.rst` document that consists of just an underlined title for right now, add it to your table of contents, and see whether you can make the cross reference a clickable link.

Chapter 6

Writing 'guide.rst'

Given how much markup you have already learned, you will probably find marking up the Guide to be a more modest challenge than those you have enjoyed so far. Once you have it marked up and looking good, here is a list of bonus challenges you might try out:

1. Make all of the references to function, class, and method names turn into real hyperlinks into the API chapter.

2. In accomplishing the first goal, you probably had to use the package name `trianglelib` quite a lot, which involves a quite depressing amount of repetition. While you cannot use the `module::` directive here, since this is not the API documentation, try using the directive `currentmodule:: trianglelib` at the top of the Guide and verify that you can then remove the module name from the beginning of all of your cross references.

3. Are the doctests in the guide really running every time you type `make doctest`? Try ending the paragraph that precedes them with `:` and then with `::` and observe the difference in results.

4. Note the three periods that form an ellipsis `"..."` in the traceback shown in the guide. This is a standard established long ago by the Python Standard Library's doctest module that lets you avoid including the whole ugly traceback — which you would then have to edit and correct every time a line number changed in one of your source files! Try removing the ellipsis to confirm that `make doctest` indeed fails if it is asked to literally match the contents of the traceback.

5. I have grown to dislike doctests more and more over the years, most recently because users cannot easily cut-and-paste doctest code into their own programs without then having to manually backspace over all of the >>> prompts that get pasted in with the code. Convert the example in the *Triangle dimensions* section into a `testcode::` block and a `testoutput::` block instead, and confirm that the block is getting detected, executed, and tested when you run `make doctest`.

6. If you are able to access the Sphinx documentation during the tutorial, look up the `math` Sphinx extension and try to figure out how I made the inequality equation so very pretty. As a super bonus, see if you can replace the inequality with this equivalent expression:

$$\frac{a+b}{c} > 1$$

Chapter 7

Example: tutorial.rst — The trianglelib tutorial

"There is no royal road to geometry." — Euclid

This module makes triangle processing fun! The beginner will enjoy how the *utils* module lets you get started quickly.

```
>>> from trianglelib import utils
>>> utils.is_isosceles(5, 5, 7)
True
```

But fancier programmers can use the *Triangle* class to create an actual triangle *object* upon which they can then perform lots of operations. For example, consider this Python program:

```
from trianglelib.shape import Triangle
t = Triangle(5, 5, 5)
print 'Equilateral?', t.is_equilateral()
print 'Isosceles?', t.is_isosceles()
```

Since methods like *is_equilateral()* return Boolean values, this program will produce the following output:

```
Equilateral? True
Isosceles? True
```

Read Example: guide.rst — The trianglelib guide to learn more!

Warning: This module only handles three-sided polygons; five-sided figures are right out.

Chapter 8

Example: guide.rst — The trianglelib guide

Whether you need to test the properties of triangles, or learn their dimensions, `trianglelib` does it all!

Special triangles

There are two special kinds of triangle for which `trianglelib` offers special support.

Equilateral triangle All three sides are of equal length.

Isosceles triangle Has at least two sides that are of equal length.

These are supported both by simple methods that are available in the `trianglelib.utils` module, and also by a pair of methods of the main `Triangle` class itself.

Triangle dimensions

The library can compute triangle perimeter, area, and can also compare two triangles for equality. Note that it does not matter which side you start with, so long as two triangles have the same three sides in the same order!

```
>>> from trianglelib.shape import Triangle
>>> t1 = Triangle(3, 4, 5)
>>> t2 = Triangle(4, 5, 3)
>>> t3 = Triangle(3, 4, 6)
>>> print t1 == t2
True
>>> print t1 == t3
False
>>> print t1.area()
6.0
>>> print t1.scale(2.0).area()
24.0
```

Valid triangles

Many combinations of three numbers cannot be the sides of a triangle. Even if all three numbers are positive instead of negative or zero, one of the numbers can still be so large that the shorter two sides could not actually meet to make a closed figure. If c is the longest side, then a triangle is only possible if:

$$a + b > c$$

While the documentation for each function in the *utils* module simply specifies a return value for cases that are not real triangles, the *Triangle* class is more strict and raises an exception if your sides lengths are not appropriate:

```
>>> from trianglelib.shape import Triangle
>>> Triangle(1, 1, 3)
Traceback (most recent call last):
  ...
ValueError: one side is too long to make a triangle
```

If you are not sanitizing your user input to verify that the three side lengths they are giving you are safe, then be prepared to trap this exception and report the error to your user.

Chapter 9

Example: api.rst — The trianglelib API reference

Routines for working with triangles.

The two modules inside of this package are packed with useful features for the programmer who needs to support triangles:

shape This module provides a full-fledged *Triangle* object that can be instantiated and then asked to provide all sorts of information about its properties.

utils For the programmer in a hurry, this module offers quick functions that take as arguments the three side lengths of a triangle, and perform a quick computation without the programmer having to make the extra step of creating an object.

The "shape" module

class `trianglelib.shape.`**`Triangle`**`(`*a*, *b*, *c*`)`

A triangle is a three-sided polygon.

You instantiate a `Triangle` by providing exactly three lengths `a`, `b`, and `c`. They can either be intergers or floating-point numbers, and should be listed clockwise around the triangle. If the three lengths *cannot* make a valid triangle, then `ValueError` will be raised instead.

```
>>> from trianglelib.shape import Triangle
>>> t = Triangle(3, 4, 5)
>>> print t.is_equilateral()
False
>>> print t.area()
6.0
```

Triangles support the following attributes, operators, and methods.

a
b
c

The three side lengths provided during instantiation.

`triangle1 == triangle2`

Returns true if the two triangles have sides of the same lengths, in the same order. Note that it is okay if the two triangles happen to start their list of sides at a different corner; `3,4,5` is the same triangle as `4,5,3` but neither of these are the same triangle as their mirror image `5,4,3`.

area()
Return the area of this triangle.

is_equilateral()
Return whether this triangle is equilateral.

is_isosceles()
Return whether this triangle is isoceles.

is_similar(*triangle*)
Return whether this triangle is similar to another triangle.

perimeter()
Return the perimeter of this triangle.

scale(*factor*)
Return a new triangle, *factor* times the size of this one.

The "utils" module

Routines to test triangle properties without explicit instantiation.

`trianglelib.utils.`**compute_area**(*a*, *b*, *c*)
Return the area of the triangle with side lengths *a*, *b*, and *c*.

If the three lengths provided cannot be the sides of a triangle, then the area 0 is returned.

`trianglelib.utils.`**compute_perimeter**(*a*, *b*, *c*)
Return the perimeer of the triangle with side lengths *a*, *b*, and *c*.

If the three lengths provided cannot be the sides of a triangle, then the perimeter 0 is returned.

`trianglelib.utils.`**is_equilateral**(*a*, *b*, *c*)
Return whether lengths *a*, *b*, and *c* are an equilateral triangle.

`trianglelib.utils.`**is_isosceles**(*a*, *b*, *c*)
Return whether lengths *a*, *b*, and *c* are an isosceles triangle.

`trianglelib.utils.`**is_triangle**(*a*, *b*, *c*)
Return whether lengths *a*, *b*, *c* can be the sides of a triangle.

Python Module Index

Index