3 Documentation

As I've already touched upon, documentation is one of the most important parts of writing software. Unfortunately, there are still a lot of projects out there that doesn't provide proper documentation. Writing documentation is seen as a complicated and daunting task, but it doesn't have to be: with the tools that are available to Python programmers, documenting your code can be just as easy as writing it in the first place.

One of the biggest culprits behind why documentation is either sparse or nonexistent is that many people assume that the only way to document code is by hand. Even if you have multiple people working on the same project, this means that one or more of them is going to end up having to juggle contributing code with maintaining documentation – and if you ask any developer which job they'd prefer, you can be sure they'll tell you they'd rather write software than write about software. Sometimes the documentation process is even completely separate from the development process, meaning that the documentation is written by people who have never written so much as a line of the actual code. Furthermore, any documentation produced this way is likely to be out-of-date: whether the documentation is handled by the programmers themselves or by dedicated writers, it's almost impossible for manual documentation to keep up with the pace of development.

The bottom line is, the more degrees of separation there are between your code and your documentation, the harder it will be to keep the latter properly maintained.

So why keep your code and documentation separate at all? It's not only possible to put your documentation directly in your code itself, but it's also easy to convert that documentation into easy-to-read HTML and PDF files.

The *de facto* standard documentation format for Python is *reStructuredText*, or *reST* for short. It's a lightweight markup language (like the famous *Markdown*) that's as easy to read and write for humans as it is for computers. Sphinx is the most commonly used tool for working with this format: it can read *reST*-formatted content and output documentation in a variety of other formats.

Your project documentation should include:

- The problem your project is intended to solve, in one or two sentences.
- The license your project is distributed under. If your software is open source, you should also include this information in a header in each code file: just because you've uploaded your code to the Internet doesn't mean that people will know what they're allowed to do with it.
- A small example of how it works.
- Installation instructions.
- Links to community support, mailing list, IRC, forums, etc.
- A link to your bug tracker system.
- A link to your source code so that developers can download and start delving into it right away.

You should also include a `README.rst` file that explains what your project does. This README will be displayed on your GitHub or PyPI project page; both sites know how to handle *reST* formatting.

Tip

If you're using GitHub, you can also add a `CONTRIBUTING.rst` file that will be displayed when someone creates a pull request. It should provide a checklist for them to follow before they submit the request, e.g. follow PEP 8 or don't forget to run the unit tests.

Tip

Read The Docs allows you to build and publish your documentation online automatically. Signing up and configuring a project is a straightforward process: it searches for your *Sphinx* configuration file, builds your documentation, and makes it available for your users to access. It's a great companion to code hosting sites.

3.1 Getting started with Sphinx and reST

First of all, you should run `sphinx-quickstart` in your project's top-level directory. This will create the directory structure *Sphinx* expects to find, along with two files in the `doc/source` folder: `conf.py`, which contains *Sphinx*'s configuration settings (and is absolutely required for *Sphinx* to work), and `index.rst`, which will serve as the front page of your documentation.

You can then build your documentation in HTML format by calling `sphinx-build` with your source directory and output directory as arguments:

```
$ sphinx-build doc/source doc/build
  import pkg_resources
Running Sphinx v1.2b1
loading pickled environment... done
No builder selected, using default: html
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
```

```
preparing documents... done
writing output... [100%] index
writing additional files... genindex search
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.
```

Now you can open `doc/build/index.html` in your favorite browser and read your documentation.

Tip

If you are using *setuptools* or *pbr* (see Section 4.2) for packaging, *Sphinx* extends them to support the command `setup.py build_sphinx`, which will run `sphinx-build` automatically. The *pbr* integration of *Sphinx* has some saner defaults, such as outputting the documentation in the `doc` subdirectory.

`index.rst` is where your documentation begins, but it doesn't have to end there: *reST* supports includes, so there's nothing stopping you from dividing your documentation up into multiple files. Don't worry too much about syntax and semantics to start with: it's true that *reST* offers a lot of formatting possibilities, but you'll have plenty of time to dive into the reference later. The complete reference explains how to create titles, bulleted lists, tables, and more.

3.2 Sphinx modules

Sphinx is highly extensible: its basic functionality only supports manual documentation, but it comes with a number of useful modules which enable automatic documentation and other features. For example, `sphinx.ext.autodoc` extracts *reST*-formatted docstrings from your modules and generates `.rst` files for inclusion. `sph` `inx-quickstart` will ask you if you want to activate this module when you run it – alternately, you can edit your `conf.py` file and add it as an extension:

```
extensions = ['sphinx.ext.autodoc']
```

Note that `autodoc` will **not** automatically recognize and include your modules. You need to explicitly indicate which modules you want to be documented by adding something like this to one of your `.rst` files:

```
.. automodule:: foobar
   :members: ❶
   :undoc-members: ❷
   :show-inheritance: ❸
```

❶ Request that all documented members be printed (optional)

❷ Request that all undocumented members be printed (optional)

❸ Show inheritance (optional)

Also note:

- If you don't include any directives, Sphinx won't generate any output.
- If you only specify `:members:`, undocumented nodes on your module/class/method tree will be skipped, even if all their members are documented. For example, if you document the methods of a class but not the class itself, `:members:` will exclude both the class and its methods entirely. To keep this from happening, you'd either have to write a docstring for the class or specify `:undoc-members:` as well.
- Your module needs to be where Python can import it. Adding `.`, `..`, and/or `../..` to `sys.path` can help with this.

autodoc gives you the power to include most of your documentation in your actual source code. You can even pick and choose which modules and methods to document – it's not an "all-or-nothing" solution. By maintaining your documentation directly alongside your source code, you can easily ensure it stays up-to-date.

If you're writing a Python library, you'll usually want to format your API documentation with a table of contents containing links to individual pages for each module. The `sphinx.ext.autogen` module was created specifically to handle this common use case. First, you need to enable it in `conf.py`:

```
extensions = ['sphinx.ext.autodoc', 'sphinx.ext.autosummary']
```

Now you can add something like the following to an `.rst` file to automatically generate a TOC for the specified modules:

```
.. autosummary::


   mymodule
   mymodule.submodule
```

This will create files called `generated/mymodule.rst` and `generated/mymodule.submodule.rst` containing the `autodoc` directives described earlier. Using this same format, you can specify which parts of your module API you want included in your documentation.

Tip

In large projects, it can be tedious to add modules to this list by hand. Just remember that `conf.py` is an ordinary Python source file: there's nothing stopping you from writing your own code in it, including code that automatically builds `.rst` files indicating which modules to document.

Another useful feature of *Sphinx* is the ability to run *doctest* on your examples automatically when you build your documentation. *doctest* is a standard Python module which searches your documentation for code snippets and runs them to test whether they accurately reflect what your code actually does. Every paragraph starting with >>> (i.e. the primary prompt) is treated as a code snippet to test:

```
To print something to the standard output, use the :py:func:`print` ↩
   function.


    >>> print("foobar")
    foobar
```

It's easy to end up leaving your examples unchanged as your API evolves; *doctest* helps you make sure this doesn't happen. If your documentation includes a step-by-step tutorial, *doctest* will help you keep it up-to-date throughout development. You can also use *doctest* for *Documentation-Driven Development (DDD)*: write your documentation and examples first, and then write your code to match your documentation.

Taking advantage of this feature is as simple as running `sphinx-build` with the special `doctest` builder:

```
    $ sphinx-build -b doctest doc/source doc/build
    Running Sphinx v1.2b1
    loading pickled environment... done
    building [doctest]: targets for 1 source files that are out of date
    updating environment: 0 added, 0 changed, 0 removed
    looking for now-outdated files... none found
    running tests...

    Document: index
    ---------------
    1 items passed all tests:
       1 tests in default
    1 tests in 1 items.
```

```
1 passed and 0 failed.
Test passed.

Doctest summary
===============
    1 test
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
build succeeded.
```

Sphinx also provides a bevy of other features, either out-of-the-box or through extension modules, including:

- Link between projects using
- HTML themes
- Diagrams and formulas
- Output to Texinfo and EPUB format
- Linking to external documentation

You might not need all this functionality right away, but if you ever need it in the future, it's good to know in advance that there are modules that can provide it.

3.3 Extending Sphinx

Sometimes the off-the-shelf solutions just aren't enough. It's one thing if you're writing an API that's going to be used from within Python, but what if you're writing, say, an HTTP REST API? *Sphinx* will only document the Python side of your API, forcing you to write your REST API documentation by hand with all the problems that entails.

The creators of WSME had other ideas. They developed a *Sphinx* extension called *sphinxcontrib-pecanwsme* which analyzes docstrings and actual Python code to generate REST API documentation automatically. You can do the same thing for your own projects: if you can extract information from your code that could be useful in your documentation, it only makes sense to automate the process.

Tip

You can use *sphinxcontrib.httpdomain* for other HTTP frameworks such as Flask, Bottle, and Tornado.

My point here is that whenever you know that you could extract information from your code that could help to build documentation, you should really do that and automatize it. It is better than trying to maintain a manually written documentation, especially when you can leverage it with auto-publication tools like Read The Docs.

To write a *Sphinx* extension, first you need to write a module, preferably as a submodule of `sphinxcontrib` (as long as your module is generic enough), and pick a name for it. *Sphinx* expects this module to have one predefined function called `setup(app)`. The `app` object will contain the methods you'll use to connect your code to *Sphinx* events and directives. The full list of methods is available in the Sphinx extension API.

For example, *sphinxcontrib-pecanwsme* adds a single directive called `rest-controller` using the `setup(app)` function. This added directive needs a fully qualified WSME controller class name to generate documentation for.

Example 3.1 Code from `sphinxcontrib.pecanwsme.rest.setup`

```
def setup(app):
    app.add_directive('rest-controller', RESTControllerDirective)
```

`RESTControllerDirective` is a directive class which has to have certain properties and methods as described in the Sphinx extension API. The main method, `run()`, will do the actual work of extracting documentation from your code.

The *sphinx-contrib* repository has a bunch of small modules that can help you develop your own.

Note

Even though *Sphinx* is written in Python and targets it by default, there are extensions available that allow it to support other languages as well. You can use *Sphinx* to document your project in full even if it uses multiple languages at once.