python_tooling_example Documentation

Release 0.0.7+0.g19e5e65.dirty

Bruno Beltran

Aug 28, 2020

Contents:

In this documentation, you will find a complete guide to taking a simple Python package and adding to it a useful set of continuous integration tools.

In addition, we will go over some documentation best practices and a step-by-step guide on how to incorporate your documentation into a useful website like ReadTheDocs.org.

Please visit this package's Github page to follow along with this guide using that package as an example.

CHAPTER 1

setup.py

The new Python documenation for creating a `setup.py` file and publishing code to PyPI is very thorough, and can be found on the official Python documentation site.

Here, we summarize the steps briefly and highlight the parts that are likely to be important to someone sharing scientific (e.g. Physics) code.

1.1 Setting up the environment

We will assume a POSIX-like environment in what follows, and a `Bash`-like shell.

```
$ python -m pip install setuptools wheel twine
```

on some machines, `pip` is not installed by default. One workaround is

```
$ python -m ensurepip --default-pip
```

Now that the tools are installed, all that's left is to make an account on PyPI.

To avoid typing your username all the time you can make a `~/.pypirc` file, whose contents need only look like:

```
[distutils]
index-servers=pypi

[pypi]
repository = https://upload.pypi.org/legacy/
username = bbeltr1

[pypitest]
repository = https://test.pypi.org/legacy/
username = bbeltr1
```

1.2 Pushing to PyPI

Copy the included `setup.py` file into your own project, and modify the fields as instructed in the comments.

Now, all that's needed is to run

```
$ python setup.py sdist
$ twine upload dist/*
```

and now anyone can `pip install` your package!

Note: Every time you run `python setup.py sdist`, setuptools creates a versioned package file in the folder `dist`. Twine will complain if you try to upload the same package version twice to PyPI, so it's more typical to actually run something like

```
$ python setup.py sdist
$ twine upload dist/package_name-X.Y.Z*
```

to specifically upload the version that was just created.

See the section on versioning for a quick tutorial on how to make the actual uploading to PyPI happen automatically whenever you bump the version number, once this initial setup is working.

1.3 Author's note

A common issue I see among scientists is that people are scared to put half-baked code onto Github or PyPI, and so they keep putting off publishing their code there until it never happens at all!

Some code is better than no code, as long as your documentation is clear about its limitations!

Publish your code!

CHAPTER 2

Documenting with Sphinx

Sphinx is a tool that helps you easily combine automatically-generated, docstring-based documentation with custom written tutorials and example scripts (usually in the form of Jupyter notebooks). It is the *de facto* standard for Python package documentation.

2.1 ReStructuredText

The markdown language used to combine the various types of documentation into a cohesive website is ReStructured-Text.

RST was originally designed to author complex documents, from books to webpages, and so is has all the complexity of other document creation languages such as LaTeX.

However, for the purposes of creating Python documentation, a very simple subset of the RST language is required.

1. *Titles*
2. *Labels & References*
3. *Extension commands (aka "Roles" and "Directives")*

All other details, such as italics, lists, links, tables, etc. can be easily looked up in the RST quickref guide.

2.1.1 Titles

Titles in Sphinx are handled by simply underlining (and/or overlining) the text with a symbol. Subsection titles are simply treated by choosing a different symbol.

Sphinx doesn't really care what symbol you use for what section heading level (it's inferred at document compilation time), but it's good to be consistent.

```
==========
Main title
==========
```

(continues on next page)

(continued from previous page)

```
Intro text goes here.

Section title
=============

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua.

Subsection title
----------------

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat.

Another Subsection
------------------

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore
eu fugiat nulla pariatur.

Another Section
===============

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.
```

You can use titles in your docstrings, and when Sphinx includes them in the documentation, they will be rendered appropriately.

2.1.2 Labels

Labels work similarly to LaTeX. They should be unique strings and will be used within your documentation for cross references, like links.

Two dots, a space, and an underscore mark the start of a label. A colon marks its end.:

```
.. _label-text:

Example title
-------------
```

This particular label would be referenced via a link like:

```
Anywhere in the text, you can make it so that `this text points to the
"Example title" above <label-text>`.
```

Note that any title can be given a label. In particular, this can often be useful for references specific sections of a long docstring from elsewhere in the documentation.

2.1.3 Roles and Directives

Anything that can't be achieved with vanilla restructured text is done via so-called "directives", which are just extension commands that are understood by the RST compiler.

The most common examples are things like

- `image`/`figure`: for including images
- `code-block`: for including code in various languages with syntax highlighting
- `math`: for including LaTeX-style "display math" using MathJax

The syntax for using a directive is similar to a label. Two dots and space mark the start of a directive (note no underscore!) and two colons demarcate the end of the directive's name:

```
.. directive-name:: arg1 arg2 arg3
    :named-parameter1: value1
    :named-parameter2: value2

    Text that the directive should be applied to. it
    must be indented. also note the blank link before
    the text starts.

    The text block can span multiple paragraphs, and
    ends when indentation returns to its previous level.

    .. note::

        Directives can be nested like this. Notice the
        simpler syntax for directives that do not
        require any arguments.
```

The above directive should be thought of as calling a python method with signature

```
def directive_name(a1, a2, a3, named_param1=None, named_param2=None):
```

In this syntax, the above directive call is equivalent to `directive_name(arg1, arg2, arg3, named_param1=value1, named_param2=value2)`. In the RST world, unnamed arguments are called "arguments" and named arguments are called "options".

There are also inline versions of directives called "Roles". The syntax for these is much simpler:

```
In the middle of a sentence, you can simply use the
:role-name:`text` syntax for simple things like inline
code or math.
```

The most common roles are:

- `code`: for inline code markup/syntax highlighting
- `math`: for inline math, analogous to latex's single dollar sign environment
- `ref` (Sphinx-specific): for creating links between different documents in your documentation. Prefer this over the usual `link <syntax>`.

The full list of standard RST roles can be found in the official docs. The extra roles provided by Sphinx are listed on the Sphinx website.

The full list of standard RST directives can be found at in the official documentation. The extensions added by Sphinx are documented in the Sphinx docs.

Note: In addition to forgetting indentation in a directive, another common gotcha with RST is that it uses two newlines to separate paragraphs (and also to separate different types of markup!). For example, it's easy to forget that you need

a blank line between a directive name and it's target. Or a blank line between the end of a list and the rest of the paragraph.

2.2 Sphinx Setup

Canonically, the folder structure of a Python repo looks like

```
git-repo-name
    ├── docs
    │   ├── build
    │   ├── make.bat
    │   ├── Makefile
    │   └── source
    │       ├── conf.py
    │       └── index.rst
    ├── LICENSE
    ├── README.md
    ├── requirements.txt
    └── package_name
        ├── __init__.py
        ├── module.py
        └── subpackage
            ├── __init__.py
            └── another_module.py
```

This section will cover how to create and populate the `docs` directory.

Thankfully, Sphinx comes with a quick setup tool that makes startup a breeze.

First, `pip install sphinx`. Then, in the top level directory (`git-repo-name` above), simply run

```
$ mkdir doc
$ cd doc
$ sphinx-quickstart
```

In response to the prompts:

1. Request a separate "build" and "source" directory to get the folder structure as shown above.
2. Fill in your project's information as requested.
3. When asked what extensions should be installed, I recommended selecting "yes" for everything **except** "imgmath". Instead wait for the next option, "mathjax", which is basically superior in every way.
4. Everything else can be safely left on the defaults.

2.3 Cleaning Up After Sphinx's Auto Setup

Sphinx needs the `conf.py` file to be able to import your package, so that it can use the docstrings in its modules' dicts' `__doc__` attributes to create the automatic documentation.

Unfortunately, the `sphinx-quickstart` script isn't very smart about this, so we often need to add a couple of lines near the top of the conf.py script

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

import package_name
```

These paths should point to your package's directory from the perspective of your "doc" and "source" directories.

I also recommended changing the "version" and "release" strings at the top of `conf.py` to point to the appropriate attributes in your module. If you don't know what that should be, simply use the following (and see the section on versioneer for how to set up your git repo to automatically track version numbers easily (for now, just make sure there's a `__version__ = 'X.Y.Z'` statement in your package's `__init__.py` file.

```
version = package_name.__version__
release = package_name.__version__
```

Finally, while we're here, let's tell Sphinx what type of docstrings we use, and ask it to automatically generate an index of all packages, modules, classes, functions, methods, etc.

```
# Autodoc settings
autosummary_generate = True

# Napoleon settings
napoleon_google_docstring = False
napoleon_numpy_docstring = True
```

This code can go anywhere below the `extensions` definition line in `conf.py`.

2.4 Using Sphinx

Sphinx's build system basically works by eval'ing your `conf.py` file, compiling `index.rst`, and then looking for any `.. toc::` directives and compiling the rst files that these point to.

For each such `*.rst` file we create, Sphinx will create a single webpage. `index.rst` will be our default landing page, but other than that, the sky is the limit in terms of what you can create!

Note: At this point you should be able to follow some examples of using Sphinx, for documentation. You can inspect the documentation of any popular Python package, like numpy, or scipy, but these packages are quite complex. For a more easy to understand example I recommend the seaborn plotting package. The code for its documentation can be found on Github.

In what follows, I'll outline the typical structure of my own personal documentation, which basically boils down to automatically generated API documentation, and Jupyter notebook (or raw RST pages) that contain tutorials for different parts of my package. For an example of what this looks like for a more "realistic" Python project, see the documentation for the multi_locus_analysis package.

2.4.1 Creating a "Page"

While normally label names and reference names are used as described above (and have no relationship to the name of the file containing the label), Sphinx finds documentation pages by looking for files whose file name matches a label referenced by the `.. toctree::` directive of your `index.rst` file.

In English, what that means is that to add a new page to your documentation website, you must first add an entry to the toctree, like `API reference <api>`, for example. Then you must create a file whose name EXACTLY matches the label (the bit in angle brackets, so in this case the file must be called `api.rst`).

Finally, in order for that file to be correctly linked to, make sure that it has a label on its main title. In the case above, the file would start with something like:

```
.. _api:

API reference
=============
```

Do this for each page you wish to create.

2.4.2 Layout of the website

I try to use a simple layout for my documentation websites. Namely, my toctree points to any tutorials that I might have set up for my package, and then has one final entry for the API documentation. See the code for this website for an example.

2.4.3 API Docs

The main strength of Sphinx is its ability to understand Python code and docstrings automagically. However, I would be lying if I didn't admit that most people find the "autosummary module" that comes with sphinx to be not quite... *auto* ...enough.

But that's no big deal! Some bright people have written an "autoautosummary" extension that is much easier to use. It's not included in mainline sphinx however, so we'll have to copy the code into our `conf.py` ourselves.

First copy the `Manual extensions` section of this package's `conf.py` file into your own project, and add `sphinx.ext.autosummary` to the list of required extensions in `conf.py`. Then, copy the `doc/source/_templates/autosummary_module.rst` file from this package into your own repo.

Now, simply copy the `api.rst` file from this repo and create one entry per module that you want to document!

2.5 ReadTheDocs.org

Once you have your Sphinx documentation building locally (i.e. `make html` works in the `doc` directory), then all that's left to get a beautiful website online for your package is to make an account on ReadTheDocs.org, then follow the instructions to link this account with your Github account!

CHAPTER 3

Continuous Integration Tools

3.1 Travis

Travis is a so-called "continuous integration" suite. Usually, that's fancy speak for "builds/tests your code when you push to Github".

In order to set up Travis to automatically build/test your code, simply make an account on their website and follow the instructions to link this with your Github account.

You can then simply copy the `.travis.yml` file from this package into your own git repo.

Note: In order for the provided `travis.yml` file to work, you'll either need to follow the instructions in the section on *automatically pushing to PyPI* below, or simply delete the "deploy" section of the `yml` file for now.

By default, this `travis.yml` file simply runs `pytest`.

3.2 Versioneer

My favorite way to keep track of versions is using the Versioneer package.

To get versioneer working, simply copy the `[versioneer]` section of this project's `setup.cfg` file, and change the "prefix" (i.e. if you want your git tags to look like "v-1.2.3" instead of "pte-v1.2.3", change "pte-v" to "v-") and change the folder the _version.py file should go into to your package's directory.

Now run

```
$ versioneer install
```

at your project's top level.

Modify your setup.py to contain the lines

```
import versioneer
setup(...
        version=versioneer.get_version(),
        cmdclass=versioneer.get_cmdclass(),
        ...)
```

and make sure that any lines like `__version__ = 'X.Y.Z'` have been removed from your main `__init__.py` file. Then commit the results.

You can now update your package's version in all the relevant places (there's like 2-6, depending on how complicated your package is) all at once by simply tagging the relevant git commit with the new version number.

See below for detailed instructions.

3.3 Automatically Pushing to PyPI

Once you have Versioneer setup, the .travis.yml file included in this directory will already be setup to push your code to PyPI any time you "tag" a git commit with an new version number.

If you've never used git tags before, simply make a commit and then execute

```
$ git tag  -a TAG-STRING -m "TAG DESCRIPTION"
```

so for example the commit with this documentation page in it was tagged

```
$ git tag -a pte-v0.0.1 -m "Added pytest, travis, and relevant docs."
```

You can then make sure the tags are pushed (so that travis sees them when building your commit), by pushing with the extra flag for tags

```
$ git push --tags
```

The only change to the `.travis.yml` file required is to follow the instructions on the Travis website for encrypting your PyPI password in the Travis configuration file (you'll see mine encrypted there by default, under `deploy -> password -> secure`.

3.4 Automatically Deploying Documentation

This is covered in the *Sphinx Autodoc* section.

CHAPTER 4

Table of Contents

4.1 Example Jupyter Notebook Tutorial

demonstrate the usage of our very advanced module. . . .

[1]:
```
import python_tooling_example
```

4.1.1 Numbers tutorial

[2]:
```
python_tooling_example.add_2(2)
```

[2]:
```
4
```

4.1.2 String tutorial

[3]:
```
python_tooling_example.concat_2("hi there number ") + "!"
```

[3]:
```
'hi there number 2!'
```

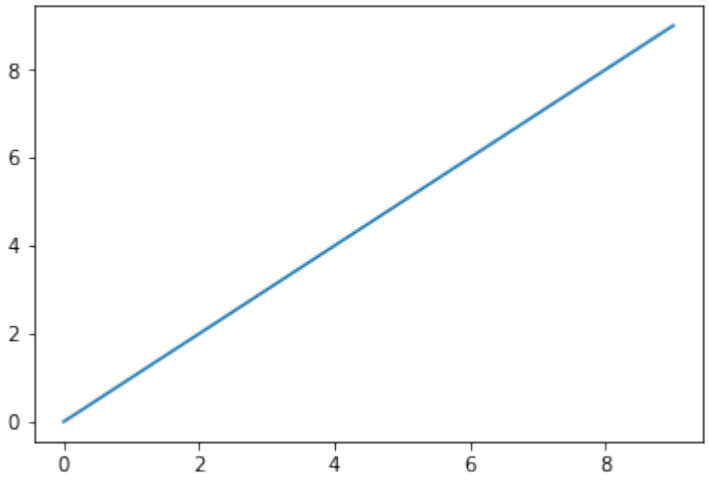
[10]:
```
%matplotlib inline
```

[11]:
```
import numpy as np
```

[12]:
```
import matplotlib.pyplot as plt
```

[13]:
```
plt.plot(np.arange(10))
```

[13]:
```
[<matplotlib.lines.Line2D at 0x7fa13320b908>]
```



[ ]:

CHAPTER 5

API reference

Click any module for detailed descriptions of available functions, usage, etc.

5.1 Adding 2 to a Number

example_module	This is an example module designed to demonstrate the convenenience of including documentation inline in your code.

5.1.1 python_tooling_example.example_module

This is an example module designed to demonstrate the convenenience of including documentation inline in your code.

python_tooling_example.example_module.**add_2**(*a*)

add integer two to a number

Parameters **a** (*float*) – the number that needs to be added to

Returns **a+2** – the result

Return type float

Notes

Uses the built-in "+" operator

5.2 Concatenating 2 to a String

example_subpackage.daughter_module	Another example module, this time inside of a subpackage.

5.2.1 python_tooling_example.example_subpackage.daughter_module

Another example module, this time inside of a subpackage.

python_tooling_example.example_subpackage.daughter_module.**concat_2**(*a*)

Concatenate the string "2" to to the end of an existing string object.

Parameters **a** (*string*) – The string to be concatenated to.

Returns **a+"2"** – The resulting string.

Return type string

Notes

Uses the built-in + operator.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

C

P