

# How to publish python code

Kale Kundert

February 20, 2017

Once you get some code to work, it's easy to not bother making that code easy for other people to use. Usually you have more pressing things to worry about, and it's hard to even know what you could do to help other people use your code anyway. However, especially as scientists, we have a responsibility to make the code we write easily accessible to anyone who wants or needs to use it.

In this tutorial, we'll walk through all the steps required to publish some python code. The example code we'll be working with will manipulate DNA sequences in simple ways. Specifically, it'll calculate lengths, GC contents, and reverse complements. We'll expose this functionality both as a library and as a command-line tool.

Broadly speaking, there are three things we need to do to publish code. These are making the code available, testing the code, and documenting the code. Each of these larger objectives can be broken down into a handful of very concrete steps which this tutorial will describe:

## **Making the code available**

Organizing your files . . . . .	2
Choosing a license . . . . .	5
Hosting your code on Github . . . . .	9
Writing a setup.py file . . . . .	10
Uploading your package to PyPI . . . . .	12

## **Testing the code**

Writing tests with pytest . . . . .	13
Running tests on TravisCI . . . . .	14

## **Documenting the code**

Writing a README.rst file . . . . .	3
Providing command-line help with docopt . . . . .	7
Writing documentation with Sphinx . . . . .	15
Uploading documentation to ReadTheDocs . . . . .	16

## **Other topics**

Writing some example functions . . . . .	6
Starting new projects with cookiecutter . . . . .	17

## Organizing your files

Most python projects organize their files in the same way. This standard directory layout is pretty simple and works well in every situation I've ever encountered. By adhering to it, you make your code easier for others to grok and you avoid getting tied into knots by bad organizational decisions.

The first step is to choose a name for your project. This name has to be unique (i.e. there can't be any other python packages with the same name) and it should somehow reflect what your code actually does. The name I'll use for this tutorial is kbkdna, because my initials are kbk and the code will deal with DNA. You should use your own initials instead, because you won't be able to publish a package with the same name as mine.

```
# Make a directory for the project and move into it.
```

```
$ mkdir kbkdna
```

```
$ cd kbkdna
```

```
# Make a README file.
```

```
$ touch README.rst
```

```
# Make a LICENSE file.
```

```
$ touch LICENSE.txt
```

```
# Make a directory for your python code.
```

```
$ mkdir kbkdna
```

```
$ touch kbkdna/__init__.py
```

```
# Make a directory for your documentation.
```

```
$ mkdir docs
```

```
# Make a directory for your tests.
```

```
$ mkdir tests
```

```
# Make sure everything was created properly.
```

```
$ ls
```

```
docs/  kbkdna/  tests/  README.rst  LICENSE.txt
```

## Writing a **README.rst** file

A `README` file should briefly explain what the project is supposed to do, how to install it, and how to use it. I also like to include a few “badges” to advertise that the project can be installed using the standard tools, that it passes its tests, and that its documentation is available online (which are all things we’ll get to in this tutorial).

The `*.rst` suffix means that the file is reStructuredText. This is a simple file format is supposed to look good as raw text while still being easy to convert to HTML. The two important things to know are that you can start a new section by underlining the title of that section with equal signs or dashes, and you can enter a block of code by ending the previous paragraph with two colons and indenting the code block.

```

`kbkdna` --- Simple tools for working with DNA
=====

When doing biology, sometimes you just need to quickly know how long a
DNA sequence is, or what its reverse complement is. There are lots of
tools out there that can tell you these things, but they often do a
lot more than that too and can be overkill for really quick questions.
In contrast, this package provides an easy-to-use command-line app
that's just designed to give simple answers to simple questions.


.. image:: https://img.shields.io/pypi/v/kbkdna.svg
   :target: https://pypi.python.org/pypi/kbkdna
.. image:: https://img.shields.io/pypi/pyversions/kbkdna.svg
   :target: https://pypi.python.org/pypi/kbkdna
.. image:: https://img.shields.io/travis/kalekundert/kbkdna.svg
   :target: https://travis-ci.org/kalekundert/kbkdna
.. image:: https://readthedocs.org/projects/kbkdna/badge/?version=latest
   :target: http://kbkdna.readthedocs.io/en/latest/


Installation
-----

You can install ``kbkdna`` from PyPI using ``pip``:

    $ pip install kbkdna


Usage
-----

The command-line application that gets installed is called ``dna``.
You can use the ``--help`` flag to get information on the kinds of
things it can calculate::

    $ dna --help


You can use the ``len`` command to get the length of a DNA sequence::

    $ dna len CATCTAATTCAACAAGAATT
    20


You can use the ``rc`` command to get the reverse complement of a DNA
sequence::

    $ dna rc CATCTAATTCAACAAGAATT
    AATTCTTGTTGAATTAGATG


You can use the ``gc`` command to calculate the GC content of a DNA
sequence::

    $ dna gc CATCTAATTCAACAAGAATT
    25.0%

```

Listing 1: README.rst

## Choosing a license

Choosing a license is important because it lets other people know how they can use your code. The website [choosealicense.com](http://choosealicense.com) is a good resource where you can compare a number common licenses and find one you like. The two most common choices for academic code are the MIT license and the GPLv3 license:

### MIT License

No restrictions on how your code can be used. This allows your code to be included with commercial software.

### GPLv3 License

Requires that any derivatives of your code be made publicly available. This effectively forbids your code from being included with commercial software.

Despite my general intellectual preference for the GPLv3 license, I'll use the MIT license for kbkdna because it's short enough to fit on one page.

MIT License

Copyright (c) 2016, Kale Kundert

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Listing 2: LICENSE.txt

## Writing some example functions

Below are the four functions making up the library we want to publish. Nothing about this code is particular to how we will publish it; it's just regular python code. Copy it into `kbkdna/dna.py`:

```
#!/usr/bin/env python2

def reverse(seq):
    """Return the reverse of the given sequence (i.e. 3' to
    5')."""
    return seq[::-1]

def complement(seq):
    """Return the complement of the given sequence (i.e. G=>C,
    A=>T, etc.)"""
    from string import maketrans
    complements = maketrans('ACTGactg', 'TGACTgac')
    return seq.translate(complements)

def reverse_complement(seq):
    """Return the reverse complement of the given sequence
    (e.g. the opposite strand)."""
    return reverse(complement(seq))

# This function contains a bug. Do you see it?
def gc_content(seq):
    """Return the GC content of the given sequence (e.g. the
    fraction of nucleotides that are either G or C)."""
    return sum(x in 'GC' for x in seq) / len(seq)
```

Listing 3: `kbkdna/dna.py`

Also add the following code to `kbkdna/__init__.py`:

```
#!/usr/bin/env python2

from .dna import *
```

Listing 4: `kbkdna/__init__.py`

## Providing command-line help with `docopt`

Along with our library, we want to provide a program to manipulate DNA sequences from the command-line. This program will read a sequence and a desired manipulation (i.e. length, GC content, reverse complement) from the command line, and will print out the result of that manipulation. Documenting the arguments that this program expects is a crucial part of making this program usable for other people. This should be done in two ways:

1. If the program receives arguments that don't make sense, it should print a brief usage hint.
2. If the user provides the `-h` or `--help` flag, the program should print a detailed help message. This message should include a few sentences saying what the program is supposed to do and a description of every single argument the program takes.

There are several ways to do both these things in python, but the `docopt`<sup>†</sup> module is by far my favorite. We just need to write a help message at the top of our script describing the arguments we expect. `docopt` will read that message and use it to parse the command-line arguments for us. If there are any problems, it will print a usage hint and end the program. If not, it will give us a dictionary containing the parsed arguments.

The thing I love about `docopt` is that it combines a good help message, which is good for users, with easy command-line argument parsing, which is good for me. I've found that I'm much more likely to write documentation when there's an immediate benefit in it for me!

There's an example on the following page. Copy it into `kbkdna/cli.py`. There are parts of this code that are particular to how we will publish it, specifically the relative import and the main function that's never called, so don't bother trying to run it yet.

---

<sup>†</sup><http://docopt.org/>

```
#!/usr/bin/env python2

"""\
Perform various simple manipulations on DNA sequences.

Usage:
    dna len <seq>
    dna rc <seq>
    dna gc <seq> [--fraction]

Commands:
    len:  Print the length of the given DNA sequence.
    rc:   Print the reverse-complement of the given DNA sequence.
    gc:   Print the GC content of the given DNA sequence.

Arguments:
    <seq>: The DNA sequence to work with.

Options:
    --fraction
        For the 'gc' command, display the result as a raw fraction
        (e.g. a number between 0 and 1) rather than a percentage.
"""

from . import dna

def main():
    import docopt
    args = docopt.docopt(__doc__)
    seq = args['<seq>']

    if args['len']:
        print len(seq)
    if args['rc']:
        print dna.reverse_complement(seq)
    if args['gc']:
        gc = dna.gc_content(seq)
        if args['--fraction']: print gc
        else: print '{:.2f}%'.format(gc / 100.)
```

Listing 5: kbkdna/cli.py



## Hosting your code on Github

Version control is useful for many reasons, but two in particular are worth mentioning here. First, it allows you to go back to old versions of your code if you have to, which allows you to tinker with working code more fearlessly. Second, it makes it easy to share your code with other people and for other people to contribute to your project. This is especially true if you host your repository on a popular site like GitHub<sup>†</sup>.

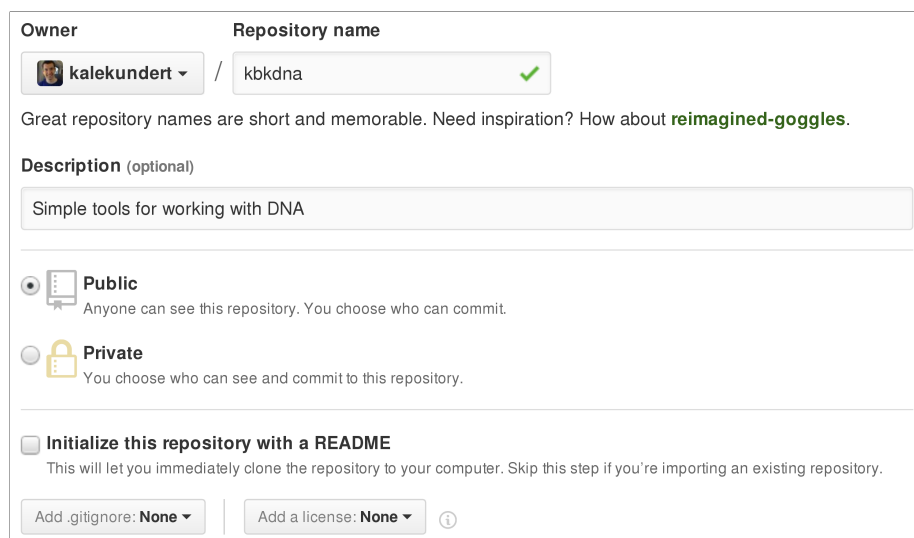
The first step is to turn your project directory into a git repository and to commit all the files you've made so far:

```
$ git init
$ git commit -a -m "Initial commit"
```

Next, log into GitHub. You'll have to create an account if you don't already have one. Then find the "New repository" button and click it.

New repository

Give the repository the same name as your project and provide a brief description for it. The defaults are fine for the rest of the options.

The image shows the 'New repository' form on GitHub. At the top, there are two fields: 'Owner' with a dropdown menu showing 'kalekundert' and 'Repository name' with a text input containing 'kbkdna' and a green checkmark. Below these is a note: 'Great repository names are short and memorable. Need inspiration? How about reimaged-goggles.' The 'Description (optional)' field contains the text 'Simple tools for working with DNA'. There are two radio button options: 'Public' (selected) with the description 'Anyone can see this repository. You choose who can commit.' and 'Private' with the description 'You choose who can see and commit to this repository.' Below these is a checkbox labeled 'Initialize this repository with a README' with the subtext 'This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.' At the bottom, there are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None', followed by an information icon.

The next page will tell you the commands you need to run to connect the repository you made for your project with `git init` with this new one being hosted by GitHub. Here are the commands I ran:

```
$ git remote add origin git@github.com:kalekundert/kbkdna.git
$ git push -u origin master
```

---

<sup>†</sup><https://github.com/>

## Writing a `setup.py` file

The best way to install python software that other people have written is to use a tool called `pip`<sup>†</sup>. You just need to give the name of the software you want to install. For example, here is the command to install `docopt`:

```
$ pip install docopt
```

To make this work for our project, we need to specify some additional information in a file called `setup.py`.

```
#!/usr/bin/env python2

from distutils.core import setup

setup(
    name='kbkdna',
    version='0.0.0',
    author='Kale Kundert',
    author_email='kale.kundert@ucsf.edu',
    url='http://github.com/kalekundert/kbkdna',
    packages=[
        'kbkdna',
    ],
    install_requires=[
        'docopt',
    ],
    entry_points={
        'console_scripts': [
            'dna=kbkdna.cli:main',
        ],
    },
    description='Simple tools for working with DNA',
    long_description=open('README.rst').read(),
    classifiers=[
        'Programming Language :: Python :: 2',
        'Intended Audience :: Science/Research',
        'Topic :: Scientific/Engineering :: Bio-Informatics',
    ],
)
```

Listing 6: `setup.py`

The `name` field is the name that other people will eventually use to refer to our project. The `packages` field is a list of directories containing the python code to install. The `install_requires` field is a list of other packages that our project makes use of. Whenever our project is installed, these will automatically be installed as well. The `entry_points` field has a lot going on, but it says

---

<sup>†</sup><https://packaging.python.org/installing/>

we want to create a command-line application called `dna` to run our `main()` function. The rest of the fields are optional metadata and are hopefully pretty self-explanatory.

We can now use `pip` to install our project for ourselves:

```
$ pip install -e .
```

The `-e` flag tells `pip` to install our code in “editable” mode. In this mode, our files will be linked (rather than copied) to the installation destination, so any changes we make will immediately be part of the installation. Without the `-e` flag, we’d have to rerun the `install` command every time we wanted to update the installed code.

The advantage of writing a `setup.py` file and using `pip` like this is that it makes your code usable from anywhere on your computer. No matter what directory you’re in, you can import and run your code. This may seem like a small thing, but it makes organizing your scripts dramatically easier. It also makes managing a set of general-purpose utility functions easier, and reusing code between projects like this is a great way to save time and effort.

## Uploading your package to PyPI

In order for other people to be able to install our software using `pip`, we need to upload it to the Python Package Index (PyPI)<sup>†</sup>, a global repository of python software that anyone can contribute to. The first step is to register an account on PyPI:

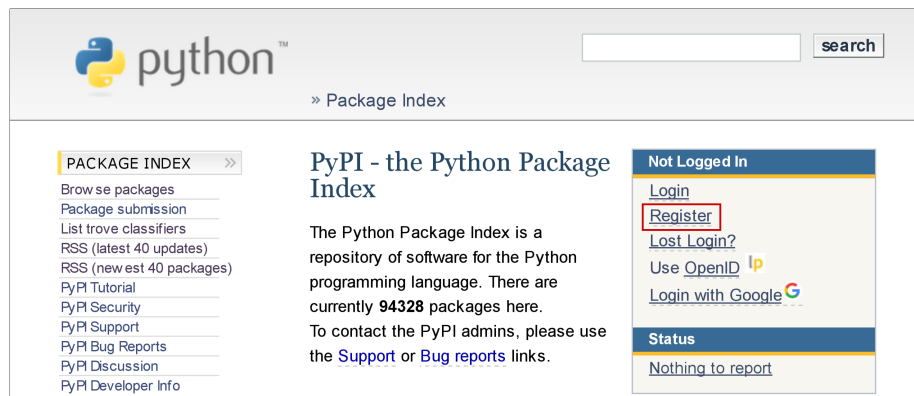


Figure 1: <https://pypi.python.org/pypi>

Once you've done that, run the following two commands to register and upload your project. You only need to run the `register` command one time for each project. You can run the `sdist upload` command whenever you want to upload a new version of your project, but you will need to increment the version number in `setup.py` each time.

```
$ python setup.py register
$ python setup.py sdist upload
```

Now anyone should be able to install your code just as they would install any other code:

```
$ pip install kbdna
```

---

<sup>†</sup><https://pypi.python.org/pypi>

## Writing tests with `pytest`

Published code should be tested, both to convince yourself and to convince your users that the code works. There are many ways to test your code, but `pytest`<sup>†</sup> is my favorite because it requires very little boilerplate, produces clear output, and integrates nicely with the debugger. Here are what some tests for our project might look like. Copy this file into `tests/test_dna.py`:

```
#!/usr/bin/env python2

import pytest
import kbkdna

def test_reverse_complement():
    assert kbkdna.reverse_complement('ATGC') == 'GCAT'

def test_gc_content():
    assert kbkdna.gc_content('ATGC') == 0.5
```

Listing 7: `tests/test_dna.py`

`pytest` will look for tests in files whose name begins with `test_`. Within those files, each test is just a normal function whose name also begins with `test_`. Within those functions, use python's `assert` statement to test your code.

Install and run `pytest` using the following commands:

```
$ pip install pytest
$ py.test
```

This should print a big red error message, explaining that the GC content of ATGC was expected to be 50%, but was incorrectly calculated to be 0%. This bug is due to integer division in the `gc_content()` function, and we can fix it by adding the following line<sup>‡</sup> to the beginning of `kbkdna/dna.py`:

```
from __future__ import division
```

If you run the tests again, they should pass and you should get a nice green success message.

---

<sup>†</sup><http://doc.pytest.org/en/latest/>

<sup>‡</sup>It's a good idea to put this line in all of your python2 scripts.

## Running tests on TravisCI

Travis CI<sup>†</sup> is a service that runs tests for open-source projects. Running your tests on Travis CI is a good way to guarantee that your code can be installed and run on other people's computers, and that you didn't accidentally forget to commit a file or list a dependency.

To use Travis CI, we need to make a file called `.travis.yml` that tells it how to install our project and run our tests:

```
language: python
python:
  - '2.6'
  - '2.7'
install:
  - pip install .
script:
  - py.test
```

Listing 8: `.travis.yml`

The `python` section specifies which versions of python should be used to test our code. The `install` section lists the commands Travis CI needs to run to install our project. It will automatically clone our repository and `cd` into it, so `pip install .` is all we need to do. The `script` section lists the commands needed to run our tests, which in this case is simply `py.test`.

The next step is to register your project for testing. You can log into Travis CI with your GitHub account, so just go to <https://travis-ci.org> and click the green button that says “Sign in with Github”.

A green rectangular button with the text "Sign in with GitHub" and a small GitHub Octocat logo to the right.

Then click on your avatar in the top right corner and you'll be brought to a list of your public repositories on GitHub. Find `kalekundert/kbkdna` and flick its switch.



This tells Travis CI to test your code every time you push changes to GitHub. You can trigger your first test by committing `.travis.yml` and pushing.

```
$ git add .travis.yml
$ git commit -m "Configure Travis CI"
$ git push
```

---

<sup>†</sup><https://travis-ci.org/>

## Writing documentation with Sphinx

Sphinx<sup>†</sup> is a tool for generating HTML documentation. HTML documentation is good because it's ready to be put online, and it supports markup like images and links that make the documentation easier to understand.

To get started with Sphinx, install it and run the `sphinx-quickstart` command given below. This will fill in the `docs` directory with boilerplate files and ask you a number of questions about how you want your documentation configured. Provide a project name, an author name, and a version number, and say yes when it asks if you want to use the `autodoc` extension. Otherwise, accept the defaults.

```
$ pip install sphinx
$ sphinx-quickstart docs
```

You can run the following commands to generate and view your documentation, although you'll just see some generic text since we haven't written anything yet.

```
$ cd docs
$ make html
$ firefox _build/html/index.html
```

Writing Sphinx documentation is not hard, but the details are beyond the scope of this tutorial. Refer to <http://www.sphinx-doc.org/en/1.5/tutorial.html> for a gentle introduction. You can also draw inspiration from the official python documentation, since it's written using Sphinx and every page has a "Show source" link. I'll also leave you with a simple example that shows how to include your README file in the documentation and how to automatically document the functions we wrote<sup>‡</sup>. The latter feature is what the `autodoc` extension is for. Copy this into `docs/index.rst`.

```
.. include:: ../README.rst
```

```
Library Functions
```

```
-----
```

```
.. automodule:: kbkdna.dna
   :members:
   :member-order: bysource
```

Listing 9: `docs/index.rst`

Regenerate the documentation and refresh the page in your browser. You should see the text from your README file, nicely formatted, followed by descriptions of the four functions that we wrote.

---

<sup>†</sup><http://www.sphinx-doc.org/>

<sup>‡</sup>This example is not very representative, in large part because Sphinx projects usually span multiple files and have more levels of organization. But for such a simple project, this is good enough.

## Uploading documentation to ReadTheDocs

ReadTheDocs<sup>†</sup> is a service that hosts documentation for your projects online. Using ReadTheDocs is a great way to make your documentation widely available without having to run your own web server. First, you need to make an account:

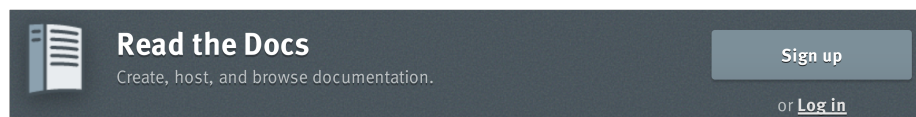


Figure 2: <https://readthedocs.org/>

Once you’ve logged in, click “Import a Project” and then “Connect to GitHub”. You’ll be asked to log into GitHub and to confirm that you want ReadTheDocs to have certain access to your repositories. When you’ve done that, go back to the “Import a project” page, find kbkdna in your list of repositories, and click it’s big plus-sign button.



ReadTheDocs will now automatically generate and update the documentation for your project whenever you push to GitHub.

```
$ git add docs/index.rst docs/conf.py docs/Makefile
$ git commit -m "Add Sphinx documentation."
$ git push
```

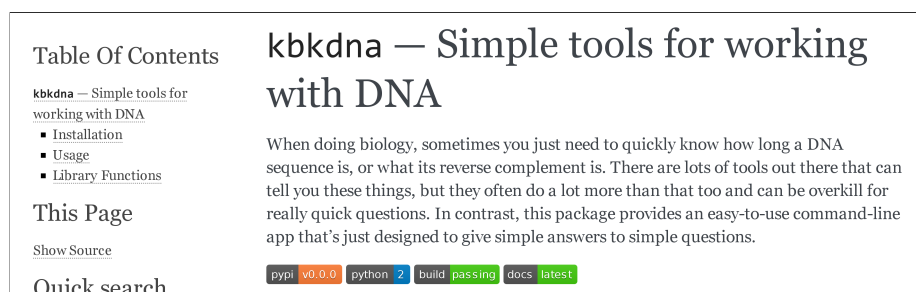


Figure 3: <http://kbkdna.readthedocs.io/en/latest/>

---

<sup>†</sup><https://readthedocs.org/>



## Starting new projects with **cookiecutter**

We've done a lot to make our code easy for other people to use, but we also had to write a lot of boilerplate code to do it. This might make the prospect of starting a new project seem daunting. Fortunately, we can use a tool called `cookiecutter` <https://github.com/audreyr/cookiecutter> to write all this boilerplate for us:

```
$ pip install cookiecutter
$ cookiecutter https://github.com/Kortemme-Lab/python_cookiecutter.git
```

When you run this command, it'll ask for a few pieces of information including your name and the name of your project. It will then use that information to create a new project directory from the template found at the specified URL. The template referenced above mostly follows this tutorial and should be appropriate for any python project. It includes a `setup.py` file, a `main()` function that will be turned into a command-line app, a `README` file with badges, the GPLv3 license, a Travis CI configuration file, and a ready-to-go Sphinx directory.

Writing python code that's easy to share with others is important, and it shouldn't be hard. In fact, I've truly found that writing code this way has made my life easier, because doing so builds on robust design patterns that work for the whole community. I encourage you to put in the time it takes to learn how to do this. It'll be worth the effort.