please follow installation instructions on poster at right now

(I will teach this during the lecture ☺)

Organizing, Documenting, and Distributing your Code

Jenni Rinker

DTU Wind Energy

Who I am.

- Researcher, Loads and Control, Wind Energy, Denmark Technical University. Engineering B.S., Civil Engineering M.S., Mechanical Engineering Ph.D.
- Matlab to Python during dissertation. Student last year. Not an expert.
- Recent team development experience. Focus on documentation and examples.
- User-oriented philosophy. Windows. Documentation, examples, and testing.
- I hate talking over people. If I ask you to stop talking, I do it with love. <3

Scenarios.

- You spend hours and hours writing beautiful Python code that follows all PEP conventions and does everything your lab could ever want. You happily introduce your repo to your colleagues. Three weeks later, they're still not using it. Even worse, they're writing code that duplicates functions in your files.
- You've got a bunch of functions and files and you want to share it with your colleagues. But how?
- You've struggled through learning sphinx and you've managed to structure all the source files so the generated html files look nice. But where can you put those files so new people can read them and learn to use your code?
- Your team just did a massive API restructure. After days of work, all your tests finally pass again. You drink. Weeks later, someone submits an issue…your examples are all broken.
- You've written a lot of beautiful tests, but you keep forgetting to run them every time you make changes to master. Things are constantly broken.

Surely there is a better way to do this…

This morning's lecture.

- Objective:
  - Bird's eye overview (and some training) into what you can do to get others to use your code.

- Demo:
  - GitHub repo: https://github.com/jennirinker/code-for-the-world.
  - Corresponding GitHub pages: https://jennirinker.github.io/code-for-the-world.

- You need to:
  1. Fork the repo to your GitHub account.
  2. Clone the forked repo to your machine.
  3. Half-close your laptop lid so I know you're ready to proceed.

What to expect.

- 09:00 – 10:30.
  - Package and repo structure.
  - Importing and installation.
- 10:30 – 11:00
  - covfefe.
- 11:00 – 11:30
  - Documenting with sphinx.
- 11:30 – 12:00
  - Continuous integration.
  - Further topics.

What to expect.

- 09:00 – 10:30.
  - Package and repo structure.
  - Importing and installation.
- 10:30 – 11:00
  - covfefe.
- 11:00 – 11:30
  - Documenting with sphinx.
- 11:30 – 12:00
  - Continuous integration.
  - Further topics.

Importing functions/modules in mypack.

- Package structure.

```
|----  mypack/
    |----  __init__.py
    |----  calcs.py ———————> def get_lift(…)
    |----  utils/
        |----  __init__.py
        |---- io.py ———————> def read_selig(…)
        |---- plotting.py ——> def plot_airfoil(…)
```

- Exercise.
  - What is the python code for the following cases:
    - Import the mypack module.
    - Import the utils module and assign it the name “mp_utils”.
    - Import the read_selig function directly into the namespace. (In other words, next line in script is “read_selig (…)”.)
    - Import the calcs submodule without assigning a custom name. Call the get_lift function from this submodule (no arguments).

What import actually does.

- import mypack
  1. Searches for the module (“finders”).
  2. Binds it to a name in the local scope (“loaders”).

Python executes the code on binding.

- Search order:
  1. sys.modules.
  2. Other places (e.g., sys.path).

Beware naming your package with the same name as a built-in.

Path search order is nuanced.

- Read more on the Python docs or Chris Yeh’s nice blog post.

Exercise. Top-down.

- Open file navigators at the mypack/ and utils/ directories.
- cd into code-for-the-world.
- In a Python 3 interpreter…
  - “import mypack”. What happens in the two directories?
  - “mypack.utils”. Expect it to work? Result?
  - “import mypack.utils”. What happens in the two directories?
  - “mypack.calcs”. Expect it to work? Result?
  - “mypack.utils.io”. Expect it to work? Result?
  - “import mypack.utils.io as my_io”, then “my_io.read_selig()”. Expect it to work? Result?

- **RULE**: When importing a package or module, Python will not import any other packages/modules at the same level or lower. Unless non-empty __init__.py.

Exercise. Bottom-up, pycache.

- Exit your Python interpreter. Delete the two pycache directories.
- In a Python 3 interpreter…
  - “import mypack.utils.io”. What happens in the file directories?
  - “mypack.utils.io.read_selig()”. Expect it to work? Result?
  - “mypack”. Expect it to work? Result?
  - “mypack.calcs”. Expect it to work? Result?
- Delete the pycache directories. “mypack.utils.io.read_selig()”. Expect it to work? Result?

- **RULE**: Bottom-level import also imports *necessary* packages in higher levels.
- **RULE**: Once package loaded into namespace, pycache files are not used until next import/reload.

Make your package discoverable everywhere. Install it.

- Add a setup.py file to the your repo directory (same level as your package).
  - Reverse-engineer it.

- "python setup.py [install/develop]" vs. "pip install [-e] ."
  - pip installs dependencies, then runs "python setup.py [install/develop]". Very nice.

pip install .	pip install –e .	pip install git+<git_url>
• Non-editable installation. • Copies files to Python directory.	• Editable (developer) installation. • Repo location is added to sys.path.	• Non-editable installation. • Source code directly to Python directory. • Can specify commit, branch, tag, issue, …

- For this lecture at ASPP 2018: replace "pip" and "python" with "pip3" and "python3": "pip3 install -e .".

Recommended practice, fuzzy areas.

- Making an environment.
  - A world where your code can live and not mess up/be messed up by others.
  - Example with Anaconda: “conda create --name new_env” then “[source] activate new_env” before pip install.
  - Running spyder terminals from an environment is possible, but not obvious.

- Dependency management.
  - “install_requires” in setup.py vs. “pip install -r requirements.txt”. Jury is still out. I prefer the former.
  - Use pip to install non-PyPI dependencies directly from git. Specify tag/branch/commit/etc.

- Install.bat files. For complex installations, I like them.

What to expect.

- 09:00 – 10:30.
  - Package and repo structure.
  - Importing and installation.
- 10:30 – 11:00
  - covfefe.
- 11:00 – 11:30
  - Documenting with sphinx.
- 11:30 – 12:00
  - Continuous integration.
  - Further topics.

What to expect.

- 09:00 – 10:30.
  - Package and repo structure.
  - Importing and installation.
- 10:30 – 11:00
  - covfefe.
- 11:00 – 11:30
  - Documenting with sphinx.
- 11:30 – 12:00
  - Continuous integration.
  - Further topics.

Sphinx. Using Python to make pretty things.

- Source files (text or images) to html or pdf.
  - html files can then be hosted on a private website, GitHub/GitLab pages, etc.
- Static content and/or auto-generated content from docstrings.
- Examine source files and website for this repo.
  - Note “View source”.
- Ironically, sphinx documentation is not intuitive.

Getting started with sphinx in your own repo.

1. Run sphinx-quickstart.
   - Makefiles (make.bat and Makefile), source directory with conf.py.

2. Change the conf.py settings as desired.
   - Notepad++ compare plugin with a demo conf.py.

3. Change your source files as desired.
   - Again, look to demo files.

4. Generate locally: make html.
   - Generated html files placed in a build directory.

My sphinx preferences.

- Alabaster is ugly. Sphinx “Read the Docs” all the way.
  - Requires sphinx_rtd_theme as a dependency, changes in conf.py.
- Numpy-style docstrings.
  - Requires numpydoc as a dependency, list it as an extension in conf.py.
- Redone index.rst file.
  - No “indices”, “tables”, stuff.
- Mixture of static and autogenerated content.
  - Listing sphinx.ext.autodoc as an extension in conf.py, adding module to sys.path. Alternatively, apidoc.
- Hosted examples.

Exercise. Locally (re)generating documentation.

- Generate the documentation locally and inspect it.
  - Must first generate output from examples using pytest: “python3 -m pytest examples/”.
  - From docs directory, “make html”. Main page is build/html/index.html.

- In mypack, make a new top-level module (same as calcs.py) with a single function, both with numpy-style docstrings.

- Add content for your new module/function in the existing api.rst file. Regenerate the html files. Does the web page look like you expected?

- Create a new custom .rst file. Any content you want. Anywhere. Figure out how to link it to the existing docs. (Hint: table of contents.) Regenerate html files. Result?

- If you got this far…figure out how to add images. Or change the sphinx theme.

What to expect.

- 09:00 – 10:30.
  - Package and repo structure.
  - Importing and installation.
- 10:30 – 11:00
  - covfefe.
- 11:00 – 11:30
  - Documenting with sphinx.
- 11:30 – 12:00
  - Continuous integration.
  - Further topics.

Let’s become automatons.

- Continuous integration (CI).
  - Integrate your code often and check it with an automatic build.
  - You push to master (or other specified branch), and stuff happens.

- Travis CI is a free option integrated with GitHub.
  - Control what happens in the build using a configuration file (.travis.yml).

- My current build:
  - Test examples. (Generates images and output text used by sphinx.)
  - Run tests and calculate coverage.
  - Build html pages using sphinx and push them to GitHub pages.

Exercise. Start messing with Travis.

- Enable Travis CI on your fork of my repo. Instructions on the "Configuring Travis CI" page.
- Comment out the deploy section in .travis.yml. Commit everything (master is fine) and push it.
- Monitor your build at travis-ci.com. When the build is done, answer these questions:
  - Did the build pass?
  - What OS is travis running your build with? (Important.)
  - What version of sphinx-rtd-theme? (Important.)
  - How many examples passed?
  - What is the test coverage?
  - Any warnings?
- Add a new test for your new functions/module. Recommit, repush. New coverage?

Exercise. If time. Deploy to GitHub pages.

- Follow the instructions in "Configuring Travis CI" to configure your GitHub token.
- Uncomment the deploy section, commit and push to master.
- Compare your pages to mine. Is your new content there?
  - Link to your pages: https://<user>.github.io/<repo_name>.
- If you're bored, do something cool.
  - Find a repo whose docs you like, and mimic something.
  - Look into codecov.
  - Try "make latexpdf" instead. Or look into using rinohtype.

Random thoughts and things.

- Compiling on Windows (f2py/cython) can be painful.
  - Be aware of this when passing code from Linux/Unix to Windows.
  - Configuration instructions with Anaconda: repo / website.

- Can pass version restrictions into install_requires in setup.py.

- Use argparse to pass in command-line arguments.

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

Go forth. Disseminate. Thanks.