Python Project Periphery:

All the small stuff they don't teach you

Jacob Wilkins

Scientific Computing Department, STFC

January 13, 2026

Before we start

- One of the most common things I hear from scientists is:

  "But my code isn't good enough to publish!"
- After contributing **many** LoC (and hopefully removing a few).
- The only **bad** coder is one who can't/doesn't take feedback.
- If:
  - You've written something useful.
  - You can describe what you intended to do.
  - You're willing to accept outside contributions.
  - You're willing to respond to and learn from those.
- **Your code will be fine and will be useful.**

Before we start

- This is not a guide to programming.
  - There will be references where needed.
- Instructions are written for a Linux terminal.
  - Works on Windows and GUI tools are available.
- I will be using the emacs editor for most things.
  - Save yourself! Don't fall into this trap.
- This is a basic introduction to give you the tools get started and the language to ask questions.
  - Sadly, it will not give you domain expertise (yet).

Before we start

- Don't dive immediately into writing a project.
- Don't reinvent the wheel!
- Ask/search around for pre-written tools that do what you want.
- Libraries have combined experience of $\sim$100+ person-years.
- If the library doesn't do **exactly** what you want consider:
  - Asking the library if they're willing/plan to implement what you want.
  - Using the library as a basis (dependency) for your work!
  - Contributing the feature back to the library.
    N.B. Make sure to read their guidelines!

Glossary

- Files
- Folders
- Commands
- Keywords

Before we start

Resources for this talk are available at:
`https://github.com/oerc0122/Python-Project-Periphery`

Before we start

- We need to make GitHub know it's us when we talk.
- Need to generate an ssh key.
- On linux/Gitbash run `ssh-keygen`
- You do not need to add a passphrase for this.
- On GitHub, go to your avatar (top-right)
- Settings → SSH and GPG keys → New SSH key
- Paste contents of `~/.ssh/id.rsa` into box.

Before we start

- You may need to create a virtual environment to be able to install things.

Set up Virtual environment

- Run `python -m venv <project>` (where project is the name you want to use).
- From `Scripts` in `<project>`, run `activate.bat`.

The code

Introducing florp

```
from cmath import sqrt
import math

from cowsay import cow

CBRT_UNITY_IM = sqrt(3)/2 * 1j

def florp(a, b, c):
    det = b**2 - (4*a*c)

    if math.isclose(det, 0):
        cow("Degenerate MOOoo-ts")

    return ((-b + sqrt(det)) / (2*a), (-b - sqrt(det)) / (2*a))

def florp2(a, b, c, d):
    q = (3*a*c - b**2) / (9*a**2)
    r = (9*a*b*c - 27*a**2*d - 2*b**3) / (54*a**3)

    s = (r + sqrt(q**3 + r**2))**(1/3)
    t = (r - sqrt(q**3 + r**2))**(1/3)

    x1 = s + t - (b/3*a)
    x2 = -(s + t)/2 - (b/3*a) + CBRT_UNITY_IM * (s - t)
    x3 = -(s + t)/2 - (b/3*a) - CBRT_UNITY_IM * (s - t)

    if any(x == x1 for x in (x2, x3)):
        cow("Degenerate MOOoo-ts")

    return (x1, x2, x3)
```

Florpulate it

- Florp is a very sophisticated library.
- It clearly performs florpulation.
- What is florpulation?
- Do you think this is a sensible name for this project?

$$\text{florp} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\text{florp2} = \Re(\sqrt[3]{1})(s+t) + \Im(\sqrt[3]{1})(s-t) + p$$

where

$$s = \left[r + \sqrt{q^3 + (r^2)}\right]^{\frac{1}{3}}, t = \left[r - \sqrt{q^3 + (r^2)}\right]^{\frac{1}{3}}$$

$$p = \frac{-b}{3a}, q = \frac{3ac - b^2}{9a^2}, r = \frac{9abc - 27a^2d - 2b^3}{54a^3}.$$

Sensible names for sensible projects

- Before anything else need to give the project usable names!
- Let's choose `solver.py` and rename functions accordingly.
- Next thing is to get it saved and tracked.

Sensible names

Your turn

- Rename `florp.py` to `solver.py`
- Rename `florp` to `quadratic`
- Rename `florp2` to `cubic`

GitHub

Using GitHub

- This assumes you have some familiarity with git and GitHub.
- Also requires you to have a GitHub account.
- Everyone set up?

Adjust to taste

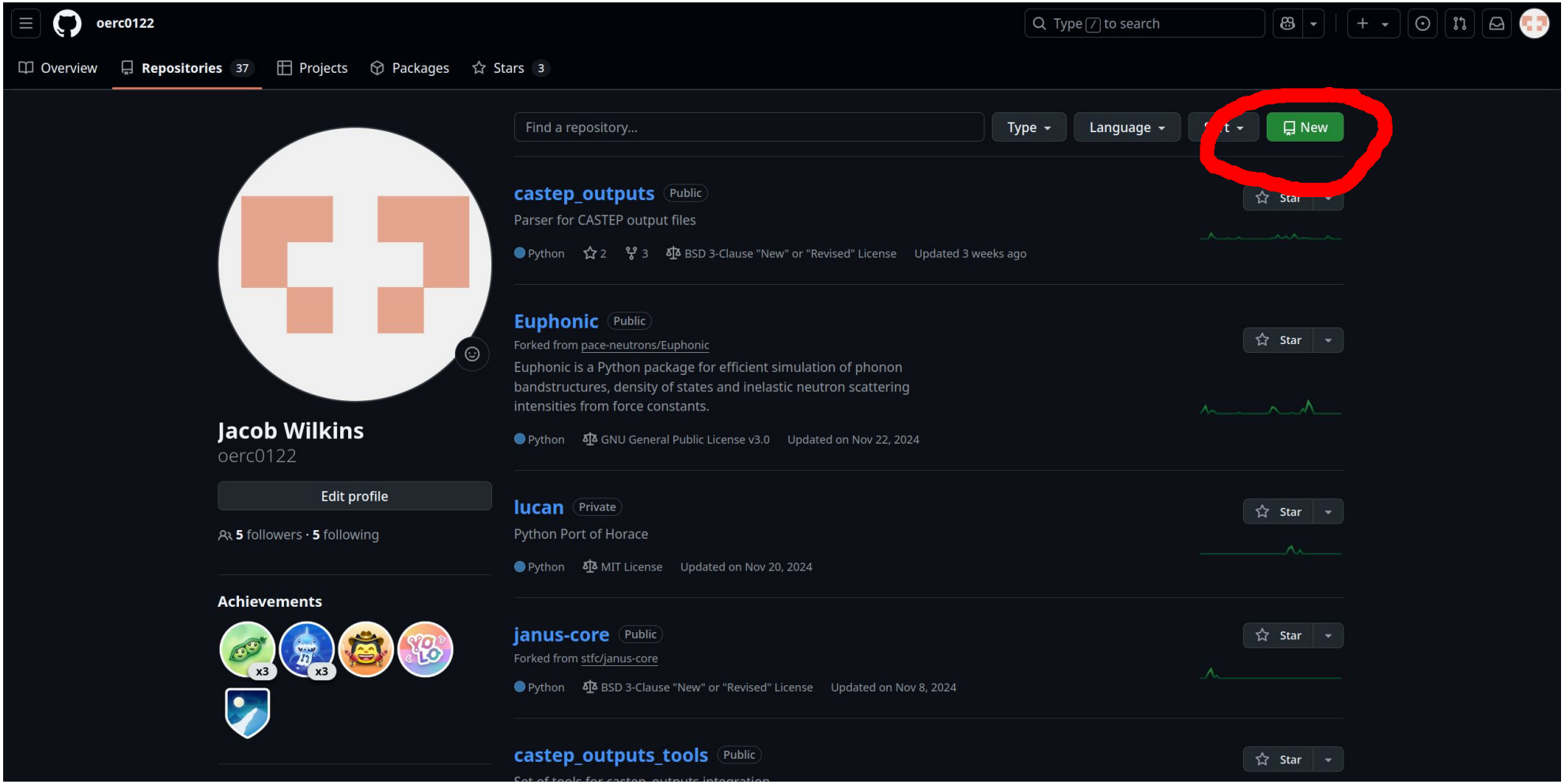
Other repositories do exist, such as GitLab, BitBucket, etc.

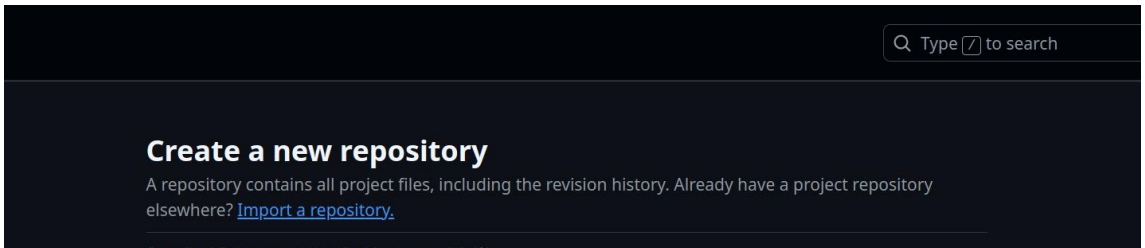
Using GitHub

Your turn

- Create a new repository with a new (sensible) name (PolysolveLib)!
- Add license, readme and set to ignore "python" extras.
  - The license choice is probably a topic for another talk.
  - The readme is displayed at the bottom of your GitHub page.
  - The ignore skips temporary files when using "`git add`".
- `git clone` the new repo and move `solver.py` into it.
- `git add solver.py`
- `git commit -m 'Add initial code'`
- `git push --set-upstream origin`

Example


oerc0122
Type / to search
Overview
Repositories 37
Projects
Packages
Stars 3
Find a repository...
Type
Language
New
castep_outputs Public
Parser for CASTEP output files
Python 2 3 BSD 3-Clause "New" or "Revised" License Updated 3 weeks ago
Euphonic Public
Forked from pace-neutrons/Euphonic
Euphonic is a Python package for efficient simulation of phonon bandstructures, density of states and inelastic neutron scattering intensities from force constants.
Python GNU General Public License v3.0 Updated on Nov 22, 2024
lucan Private
Python Port of Horace
Python MIT License Updated on Nov 20, 2024
janus-core Public
Forked from stfc/janus-core
Python BSD 3-Clause "New" or "Revised" License Updated on Nov 8, 2024
castep_outputs_tools Public
Star
Jacob Wilkins
oerc0122
Edit profile
5 followers · 5 following
Achievements
x3
x3


Type / to search
Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Using GitHub

Your turn

- Create a new repository with a new (sensible) name (PolysolveLib)!
- Add license, readme and set to ignore "python" extras.
  - The license choice is probably a topic for another talk.
  - The readme is displayed at the bottom of your GitHub page.
  - The ignore skips temporary files when using "`git add`".
- `git clone` the new repo and move `solver.py` into it.
- `git add solver.py`
- `git commit -m 'Add initial code'`
- `git push --set-upstream origin`

Using GitHub

- Double check on GitHub and your files should be on it.

Package

From script to project

- To start a project, we need to define what the project is.
- The first step is to change the structure to that of a project.
- Make a new folder "`polysolve`" and `git mv` "`solver.py`" into it.

Layout

This form of putting code in `<project>/...` is called flat-layout.
You can also put code in `src/<project>/...` this is called source-layout.

Making a package

Try it out!

- Make a new folder "`polysolve`"
- `git mv` "`solver.py`" into it.
- Create an empty file called `__init__.py` in `polysolve`
- You need to `pip install cowsay` to get this to run.
- From `PolysolveLib` run `python` and run the following code (**NOTE:** Do not type the >>>)

```
#          foldername          filename
>>> from polysolve import solver
>>> solver.quadratic(1, 2, 3)
#    import.function
```

NOTE: This is only accessible from our project folder (`PolysolveLib`), not the system, it's not installed yet.

Making a package

- This file, `__init__.py`, is a magic file.
- In Python it makes a folder accessible for `import`.
- All files in the folder with the `__init__.py` are accessible.
- Subfolders can be nested, each one needs an `__init__.py`.
- Code in an `__init__.py` is run **when the module is imported**.
- The is attached to the name of the module.
- This can be used for setup or our package's metadata.

I've never metadata who did

Your turn

- Add the following code to `__init__.py`

```
"""Module to compute quadratic/cubic roots."""
__author__ = "Me"
__version__ = "0.1"
```

- From `PolysolveLib` run `python` and run the following code:

```
>>> import polysolve
>>> polysolve.__author__
>>> polysolve.__version__
>>> help(polysolve)
```

Making a project

- Now that we have a package it's time to make this a project.
- We need a `pyproject.toml`.
- `pyproject.toml` defines the metadata our project[1].
- When you `pip install` this reads the `pyproject.toml`.

TOML History

TOML (Tom's Own Markup Language) is a standardised format designed to replace the non-standardised `.ini` format configurations.

Ancient (modern) History

Older projects used to use something called `setup.py`, this is being deprecated except where your project needs e.g. Cython or compiled C++, and even then...

[1] For more info on Python packaging take a look on the PyPA at: `https://packaging.python.org/en/latest/tutorials/packaging-projects/`

The pyproject.toml

```
[build-system]
requires = ["setuptools >= 61.0.0"]
build-backend = "setuptools.build_meta"

[project]
name = "polysolve"
authors = [{name = "", email = ""}]
requires-python = ">= 3.8"
readme = "README.md"
description = ""
license = {text = "BSD-3-Clause"}

keywords = []
dependencies = []
classifiers = []
dynamic = ["version"]

[project.urls]
Homepage="https://github.com/XXX/polysolve"
Repository="https://github.com/XXX/polysolve.git"

[tool.setuptools.dynamic]
version = {attr = "polysolve.__version__"}
```

Let's look at these individually.

Note

Keywords are arranged into "block"s and are order independent within blocks. Blocks are order independent too.

build-system

```
[build-system]
requires = ["setuptools >= 61.0.0"]
build-backend = "setuptools.build_meta"
```

- These are the Python tools `pip` will use to build your project.
- You may choose something else (info on PyPA[2]), but we'll just stick with `setuptools`.

[2] https://packaging.python.org/en/latest/tutorials/packaging-projects/

project

```
[project]
name = "polysolve"
authors = [{name = "", email = ""}]
requires-python = ">= 3.8"
readme = "README.md"
description = ""
license = {text = "BSD-3-Clause"}

keywords = []
dependencies = []
classifiers = []
```

- These define the properties which describe your project:
- `name` – The project's installed name.
- `authors` – The project's authors.*
- `requires-python` – The minimum version of python needed to run the project.
- `readme` – The readme file/content.*
- `description` – A brief summary of the project.*

Webpages

```
[project.urls]
Homepage="https://github.com/XXX/polysolve"
Repository="https://github.com/XXX/polysolve.
    git"
```

- PyPI will add these links in a sidebar if you upload your project.

Dynamic

```
[project]
dynamic = ["version"]

[tool.setuptools.dynamic]
version = {attr = "polysolve.__version__"}
```

- You may have spotted `dynamic` at the end of the `[project]` block.
- `dynamic` is a special keyword which tells `pip` the variable will come from somewhere else.
- We define our `version` as coming from our package.

Extra dynamicism

We can define several other properties as `dynamic` see PyPA for more info.

Connect the dots

Your turn

- Copy `pyproject.toml` to your project root directory.
- We can fill in the gaps in our `pyproject.toml` (**NOTE:** You must add non-blank authors)
- We also need to tell it that we have some dependencies (`cowsay`). Change the relevant line to `dependencies = ["cowsay"]`

Connect the dots

- Then we can see some magic happen.
- `pip` checks we have all the requirements, installs the dependencies, then our project.
- **NOTE:** It's now installed system-wide.

Try it out!

`pip uninstall cowsay` – Just proving a point

```
pip install .
cd ~
python
>>> from polysolve import solver
>>> solver.quadratic(3, 1, 2)
```

Developing

While developing you will want:

```
pip install -e .
```

which will link to the package so as you edit it the system version updates.

Get it gitted

To git

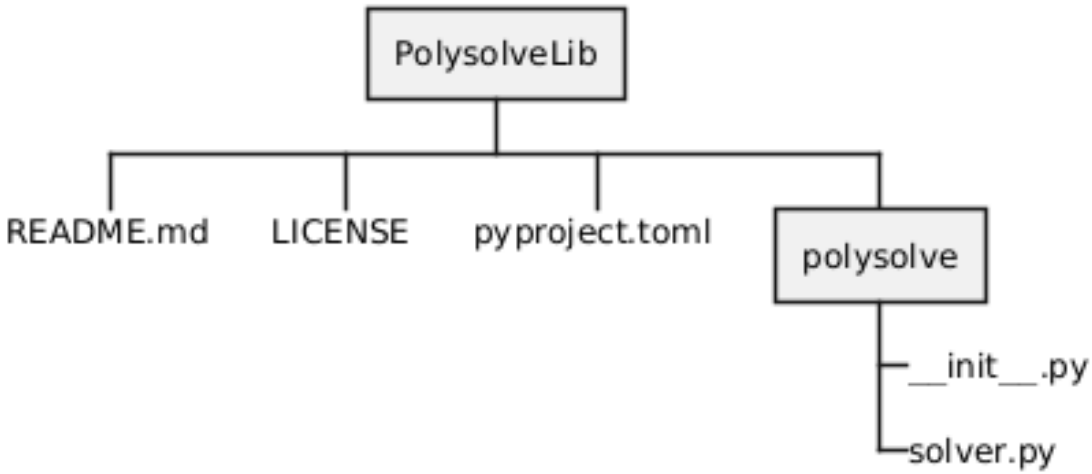
- Now `add` the files to `git`.
- `git commit`
- `push` it up to GitHub.

More magic!

```
pip install git+https://github.com/<owner_name>/polysolve.git
```

NOTE: PyPI is "easier", but requires accounts. This is convenient for small stuff.

Structure


PolysolveLib
README.md
LICENSE
pyproject.toml
polysolve
__init__.py
solver.py

Great, we have a project

- Now what?
- The next step is to make it usable.
- That means usable by other people.

Documentation

Documentation, documentation, documentation.

- We're going to begin by looking at documentation.
- Documentation tends to fall by the wayside.
- **However**, it's the most important thing in released software.

Documentation, documentation, documentation.

- Let's start with something simple.
- Our `README.md` basically says the project name.

Our first docs

- We know how to install it now, so let's add that.

```
pip install git+https://github.com/<owner_name>/polysolve.git
```

- Push it up to GitHub and see the glory of your hard work.

IDE Ahoy

- Anybody here used VSCode or another IDE[3]?
- When you start typing a function, it tells you what argument comes next.
- It also tells you the `type` it should be (`int`, `float`, etc.).

[3] Interactive Development Environment

IDE Ahoy

- The IDE isn't doing any **magic** to find out, we tell it!
- How do we tell it?
- We use "type-hints" or "annotations".

```
def quadratic(
    a: float,
    b: float,
    c: float
) -> tuple[float, float]:
    det = b**2 - (4 * a * c)

    out = ((-b + sqrt(det)) / (2 * a), (-b - sqrt(det)) /
        (2 * a))

    return out

from __future__ import annotations

def quadratic(
    a: float,
    b: float,
    c: float
) -> tuple[float, float]:
    det = b**2 - (4 * a * c)
```

Handy dandy

- These type-hints aren't just useful to users.
- They're useful to us as developers.
- We know when changing things what we're allowed to do.

```
def quadratic(
    a: float,
    b: float,
    c: float
) -> tuple[float, float]:
    det = b**2 - (4 * a * c)

    out = ((-b + sqrt(det)) / (2 * a), (-b - sqrt(det)) /
        (2 * a))

    return out
```

What are we doing again?

- So we know what we're feeding the black box.
- Wouldn't it be nice if the box told us what it did (or is trying to do)?
- Don't go rushing off to write in the `README.md` again!

What are we doing again?

- Python allows us to annotate further!
- Introducing the docstring!
- This is the minimal docstring.
- We can add more![4]

```
"""
Solves the roots of a quadratic equation.
"""
```

[4] See `quadexm.py`

What are we doing again?

- We can add more![5]

```
"""
Solves the roots of a quadratic equation.
"""

"""
Solves the roots of a quadratic equation.

Uses the quadratic formula. Result must be real.
"""

"""
Solves the roots of a quadratic equation.

Uses the quadratic formula. Result must be real.

Parameters
----------
a
    :math:`x^2` coefficient.
b
    :math:`x` coefficient.
c
    Constant value.
"""
```

Substance over style

- **Note:** what I've been showing you is **one** style of docs.
- This style is called `numpydoc` style after the `numpy` library.

Substance over style

The main styles are: numpydoc

numpydoc.readthedocs.io/en/latest/format.html

```
from math import sqrt

def quadratic(a: float, b: float, c: float) -> tuple[float, float]:
    """
    Solves the roots of a quadratic equation.

    Uses the quadratic formula. Result must be real.

    Parameters
    ----------
    a
        :math:`x^2` coefficient.
    b
        :math:`x` coefficient.
    c
        Constant value.

    Returns
    -------
    .. [1] O. McNoleg, "The integration of GIS, remote sensing,
```

Substance over style

The main styles are: google

google.github.io/styleguide/pyguide.html

```
def quadratic(a: float, b: float, c: float) -> tuple[float
    , float]:
    """Solves the roots of a quadratic equation.

    Uses the quadratic formula. Result must be real.

    Parameters:
        a: :math:`x^2` coefficient.
        b: :math:`x` coefficient.
        c: Constant value.

    Returns:
        Positive and negative roots of quadratic.
    """
```

Substance over style

The main styles are: sphinx

sphinx-rtd-tutorial.readthedocs.io/en/latest/docstrings.html

```
def quadratic(a: float, b: float, c: float) -> tuple[float
    , float]:
    """Solves the roots of a quadratic equation.

     Uses the quadratic formula. Result must be real.

    :param a: :math:`x^2` coefficient.
    :param b: :math:`x` coefficient.
    :param c: Constant value.

    :return: Positive and negative roots of quadratic.
    """
```

More magic

- Ok, we've got docstrings. Now time for a callback:
- Remember this?[5]
- What happens if this goes out of date or doesn't work?

Examples

```
>>> quadratic(1, 2, 0)
(0.0, -2.0)
>>> quadratic(3., 0., -1.)
(0.577350269189625 7, -0.577350269189625 7)
```

[5] See quadexm.py

More magic

- Thankfully, Python provides a way to use these as tests!
- (Already into tests and we're not out of the docs section yet! Sneak peek!)
- `https://docs.python.org/3/library/doctest.html`

```
Examples
--------
>>> quadratic(1, 2, 0)
(0.0, -2.0)
>>> quadratic(3., 0., -1.)
(0.577350269189625 7, -0.577350269189625 7)
```

Trying tests

Try it out!

- Add a sensible amount of documentation to `solver.py`. (Short summary, parameters, returns, examples will do. You can steal bits of `quadexm.py`).
- Try running python doctests on your source

```
python -m doctest polysolve.py
```

- We can make it so when we run our code we run the tests. Add the following to the end of `solver.py`.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

- Run `python polysolve.py`.

More magic

- Doctests are designed to imitate Python REPL.
- Designed for copying and pasting from REPL.
- Lines starting with "`>>>`" are run.
- Lines can be continued/indented with "`...`".
- Lines with neither are checked against the result.
- Need to import libraries if they're needed.

Example

```
>>> my_var = ["hello", "goodbye"]
>>> my_var
['hello', 'goodbye']
>>> for i in range(3):
...     print(i)
0
1
2
```

Example

Finally getting to docs

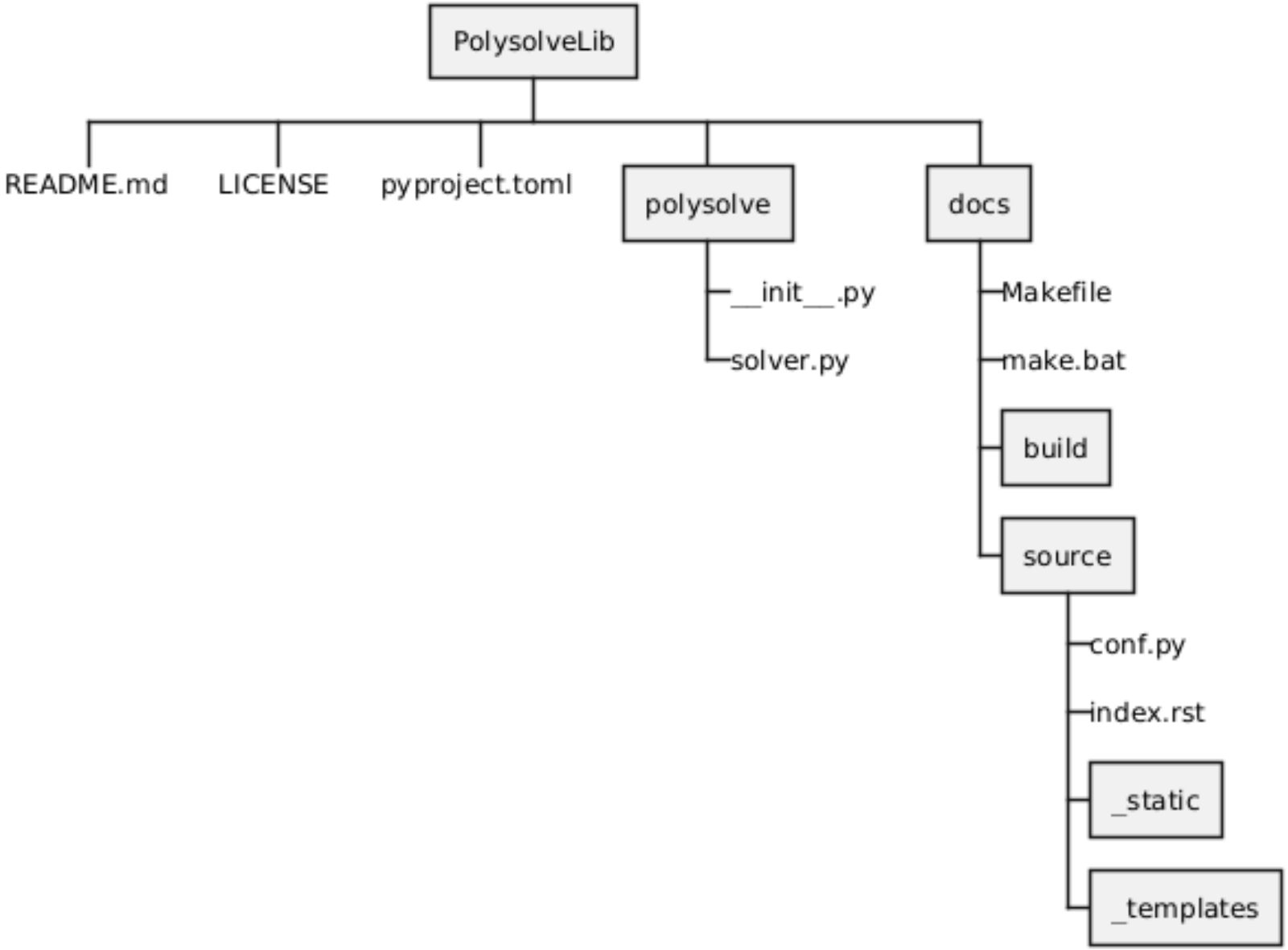
- Now after so long, it's time to finally write some docs!
- (or let the computer write some for us...)

Getting started

Your turn

- `pip install sphinx sphinx_rtd_theme`
- Make a directory in `PolysolveLib` called `docs`
- From the `docs` folder run `sphinx-quickstart` (answer `y` to the first question)

Structure


PolysolveLib
README.md
LICENSE
pyproject.toml
polysolve
__init__.py
solver.py
docs
Makefile
make.bat
build
source
conf.py
index.rst
_static
_templates

Build the docs

Build the docs

- From the `docs` folder run the appropriate `make` with `html` as the argument.
- This should build some docs.
- Open `build/html/index.html` in your browser.
- Bask in the glory of your docs.

Keys to the docs

- Key files in the new docs are:
  - `conf.py` – Configuration for docs.
  - `index.rst` – Main starting file for docs.
- Let's take a look at these.

conf.py

- `conf.py` is an auto-generated Python file with instructions for building the docs.
- It is a full Python file you can run code in, e.g. we can pull out information from our package.
- For example, we can use our defined metadata.
- `sphinx` is a fully extensible package. We'll be using some of these later.
- `exclude_patterns` allows us to exclude source files from our sphinx build.
- We can change the docs theme to render them differently.

```
# Configuration file for the Sphinx documentation builder.
#
# For the full list of built-in configuration values, see the documentation:
# https://www.sphinx-doc.org/en/master/usage/configuration.html

# -- Project information -----------------------------------------------------
# https://www.sphinx-doc.org/en/master/usage/configuration.html#project-
    information
import polysolve
from datetime import date

project = 'polysolve'
author = polysolve.__author__
copyright = f'{author}, {date.today().year}'
```

Fill in the config

Configure the docs

- Import `polysolve` and try setting the version and author from the metadata in the module. (**Note:** This will only work if `sphinx` can see your module, i.e. it's installed.)
- Since we installed `sphinx_rtd_theme` we can try that. Set the `html_theme` to `sphinx_rtd_theme`
- `make` the docs and see what's changed.

index.rst

- `sphinx` docs are written in REStructured Text (ReST/rst)[5].
- Text "marked-up" with formatting (like LaTeX or HTML).

```
.. polysolve documentation master file, created by
   sphinx-quickstart on Mon Oct 14 21:27:04 2024.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

polysolve documentation
=======================

Add your content using ``reStructuredText`` syntax. See the
`reStructuredText <https://www.sphinx-doc.org/en/master/usage/restructuredtext
    /index.html>`_
documentation for details.


.. toctree::
   :maxdepth: 2
   :caption: Contents:
```

[5] `www.sphinx-doc.org/en/master/usage/restructuredtext/index.html`

Writing our docs

Get some docs

- Now we need some files to actually to actually fill with docs!
- In `docs/source`, create a file called `usage.rst`.
- Write some documentation.
- In `docs/source/index.rst`, add it to our "table of contents tree" (`toctree`).
- `make html`

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   usage
```

Docstring Magic

- So now we can write about every single function in our project.
  - How many could there be?
  - What do you mean not every project has 20 lines?
- Remember our docstrings?
- Maybe there's a way to avoid writing everything twice.

Docstring magic

- What if we could extract all the docstrings we've already written?
- We're going to need to do a couple of things.
- Time to use some extensions.
- `sphinx.ext.autodoc` extracts docstrings from functions.
- `sphinx.ext.napoleon` converts our `numpydoc` to `sphinx`
- `sphinx.ext.autosummary` adds a summary to each page.

```
extensions = [
"sphinx.ext.autodoc",
"sphinx.ext.napoleon",
"sphinx.ext.autosummary",
]
```

Docstring magic

- We need to create all the infrastructure to extract our info.
- Just kidding, there's a tool for that!

It's magic!

- Add `sphinx.ext.autodoc`, `sphinx.ext.napoleon`, and `sphinx.ext.autosummary` to the `extensions` in `conf.py`
- From the `PolysolveLib` folder run `sphinx-apidoc -o docs/source/api polysolve`
- Add `api/modules` to the `toctree` in `docs/source/index.rst`.
- Run `make html`

Docstring magic

- But what if we add files?
- Wouldn't it be tedious to run this every time?
- Time to use another extension!

Try it now

- Add a new extensions `sphinx.ext.apidoc`.
- Add a new configuration variable which looks like the following.
- **Note:** Directories are relative to the `conf.py`

```
apidoc_modules = [
    {"path": "../../polysolve",
     "destination": "api"},
]
```

Type-hints

- But our typehints aren't with our params...
- If only there were some tool to extract those too.
- Alas, it's surely impossible...

Try it yourself

Add the type-hints

- Run `pip install sphinx-autodoc-typehints`
- Add `"sphinx_autodoc_typehints"` to the list of `extensions` in `docs/source/conf.py`

But what is a float really?

- But now I'm unhappy because when I click `float` it doesn't take me to the documentation of `float`.
- Some people, honestly.
- Introducing "`intersphinx`".
- Links your documentation against other `sphinx` documentation sites automatically.

Intersphinx

Your turn

- Add `"sphinx.ext.intersphinx"` to `conf.py`.
- Add an `intersphinx_mapping` to `conf.py` (see: below)
- This mapping tells `sphinx` where to search for external documentation.
- `make` the docs and see the new highlighted links.

```
intersphinx_mapping = {
    'python': ('https://docs.python.org/3/', None),
}
```

Adding it to the project

Add it in

- Now that we know what we need, we can add these to our `pyproject.toml`. Not everyone needs to install them, so let's add them as an optional dependency. (See below)
- Run `pip install -e ".[docs]"` from `PolysolveLib` to install with the `docs` extra.

```
[project.optional-dependencies]
docs = ["sphinx",
        "sphinx_rtd_theme",
        "sphinx-autodoc-typehints"]
```

Done with docs

- More extensions and tools are available for building docs.
- In particular things like:
  - Integrated Jupyter tutorials (`nbsphinx`).
  - Testing within documentation (`sphinx.ext.doctest`).
  - and many more...

Tests

Done with docs

- Now we have some documentation to back up our code.
- Now we're ready to check it works.
- We already have `doctest`s, which are good, but incomplete.

Types of tests

- We generally break testing down into 3–4 main types:
  - Unit tests – Tests of each function.
  - System tests / End-to-end tests – Small tests of the whole programs.
  - Benchmark tests – Tests real world cases.
  - Integration tests – Tests interfaces between program components.
- We split these into three types:
  - Science tests – Check the validity against a known result.
  - Regression tests – Check values haven't changed.
  - Fail-state tests – Intentionally check failure states.

Testing

- Our `doctest`s go some of the way towards unit-tests.
- But they aren't the be all and end all.
- Let's see how to do proper tests.

Testing

Setting up

- Install the `pytest`[a] library.
- Create a `tests` folder.
- In that folder, let's create a `test_quadratic.py` with the code below.

```
import math
from polysolve.solver import quadratic

def quad(a, b, c, x):
    return a * x**2 + b * x + c

def test_roots():
    """Tests that quadratic finds the root for a known problem."""
    params = (3.0, 0.0, -1.0)
    roots = quadratic(*params)

    assert all(math.isclose(quad(*params, root), 0.0) for root in roots)
```

- From `PolysolveLib`, run `pytest`. But what about our doctests? Try `pytest --doctest-modules`

---

[a] **NOTE:** Python ships with the `unittest` library, but rather than teaching two methods and confusing things, I'm sticking with one.

Testing

- By default, `pytest` scans files from where you are.
- `pytest` picks up files starting with `test_`.
- Runs all functions starting with `test_`.
- (and as mentioned with the `--doctest-modules` flag, runs those too)
- Collates them all and runs them together.

Multiple-Testing

- So we have our first test, but solving $3x^2 - 1$ shouldn't be all we try.
- Try to think of all the common cases...
  1. What other common cases might we try?
  2. What happens if $a = 0$?
  3. What happens if $b^2 - 4ac < 0$?
  4. What happens if I pass `int`s?

  ⋮ ...

Multiple-Testing

- Focussing on question 1...
- We want to run say: x^2, $x^2 + 14x + 49$, $3x^2 + 2x + 1$, ...
- Do we need to create a function for each one? No.

```
@pytest.mark.parametrize(
    "params, expected",
    [
        ([1.0, 0.0, 0.0], [0.0, 0.0]),
        ([1.0, 14.0, 49.0], [-7.0, -7.0]),
        ([3.0, 2.0, -1.0], [1 / 3, -1.0]),
    ],
)
def test_quadratic(params, expected):
    """Test quadratic meets expectations."""
    assert all(map(cmath.isclose, quadratic(*params),
        expected))

@pytest.mark.parametrize('a', [1, 2, 3])
@pytest.mark.parametrize('b', [1, 2, 3])
def test_example(a, b):
    """Example function taking 2 arguments."""
    assert np.product([a, b]) == a*b
```

Note

Setting up

- Have a go at writing a test with `parametrize`.
  Note: We now have to import `pytest`.
- Remember to run it with `pytest`.

```
@pytest.mark.parametrize(
    "params, expected",
    [
        ([1.0, 0.0, 0.0], [0.0, 0.0]),
        ([1.0, 14.0, 49.0], [-7.0, -7.0]),
        ([3.0, 2.0, -1.0], [1 / 3, -1.0]),
    ],
)
def test_quadratic(params, expected):
    """Test quadratic meets expectations."""
    assert all(map(cmath.isclose, quadratic(*params),
        expected))
```

Testing failures

- Now we need to fail spectacularly.
- Usually, providing a wrong answer is worse than exploding[6].
- It's good to make sure our failures fail and are helpful.

```
def test_quadratic_fails():
    """Check bad quadratic raises error."""
    with pytest.raises(ValueError,
                       match="not quadratic"):
        # There are infinite roots on this flat line.
        quadratic(0., 0., 0.)
```

[6] HCF - Halt and Catch Fire – Genuine assembly instruction

Failing

Trying to fail

- Add the following test to `tests/test_quadratic.py`

```
def test_quadratic_fails():
    """Check bad quadratic raises error."""
    with pytest.raises(ValueError,
                       match="not quadratic"):
        # There are infinite roots on this flat line.
        quadratic(0., 0., 0.)
```

- Does it pass? If not can you make it?

Testing as design

- We should know what we want our code to do before we write it.
- One way of writing software is:
  - Define tests which describe desired functionality.
  - Develop until tests pass.
- This can be useful for known problems.
- Roughly describing something called Behaviour-Driven Development.

Testing

- Ok, I've written a test which doesn't work (yet).
- We can skip the test if we know it doesn't work (yet).
- It is bad practice to skip tests because they don't work.
- It is worse practice to remove tests because they don't work.
- Remove tests only if they don't fit the design.

```
@pytest.mark.skip(reason="Beyond maths as we know it.")
def test_quintic():
    """Test quintic meets expectations."""
    assert np.isclose(quintic(1., 0., 0., 0., 0., 0., 0.),
                      expected)
```

Testing

- Testing helps you:
  - Prove the efficacy of your code.
  - Develop functionality defined by requirements.
  - Identify exactly when something went wrong.
  - Avoid adding broken code.
- **Hint:** Testing is good.
- Code without tests can be considered worthless.

Adding to pyproject

Adding it in

- Add it to your `pyproject.toml`
- Install it to check it works.
- `add` all appropriate files and get it on GitHub (`push`).

```
[project.optional-dependencies]
docs = [
    "sphinx",
    "sphinx-rtd-theme",
    "sphinx-autodoc-typehints"
]
test = ["pytest"]
```

Testing

- As discussed a few other times `pytest` is one of many testing frameworks. Others include:
- `unittest` – Basic test harness installed with `Python`.
- `cucumber` – Tests written in "English" rather than code.
- `hypothesis` – Tests with randomly generated values meeting requirements.

CI/CD

Automagic

- But running tests isn't fun.
- It'd be boring if we had to do it **every** time.
- If only there were a better way...
- CI/CD (continuous integration/continuous deployment)
- Fancy name which means automated testing & building.

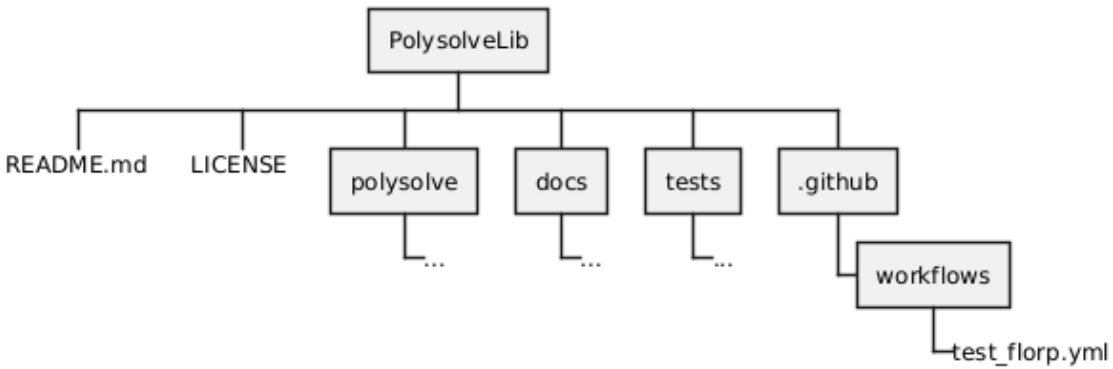
Automagic

- GitHub lets us run tests on their machines.
- with some minor caveats
- We just need to tell them what they need to do.
- We do this by adding a `yaml` file in the right place.
- GitHub provides actions where we just need to fill in values.

Starting off

Your turn

- Create a nested folder in `PolysolveLib` called `.github/workflows`
- Copy the `test_florp.yml` from the `Resources` folder into `.github/workflows`.
- `add` this to git, `commit` and `push` to github.
- Go to github and on the line by your commits, you should see either ✓, ○ or ×



Anatomy of the YAML

- Display name and script permissions.
- What will trigger the run.
- Main job description.
  - Which OS to run on.
  - Run on Ubuntu with each of the python versions
- Job stages using matrix defined previously.
  - Download the repo using git
  - Install Python
  - Install the project
  - Run the tests.

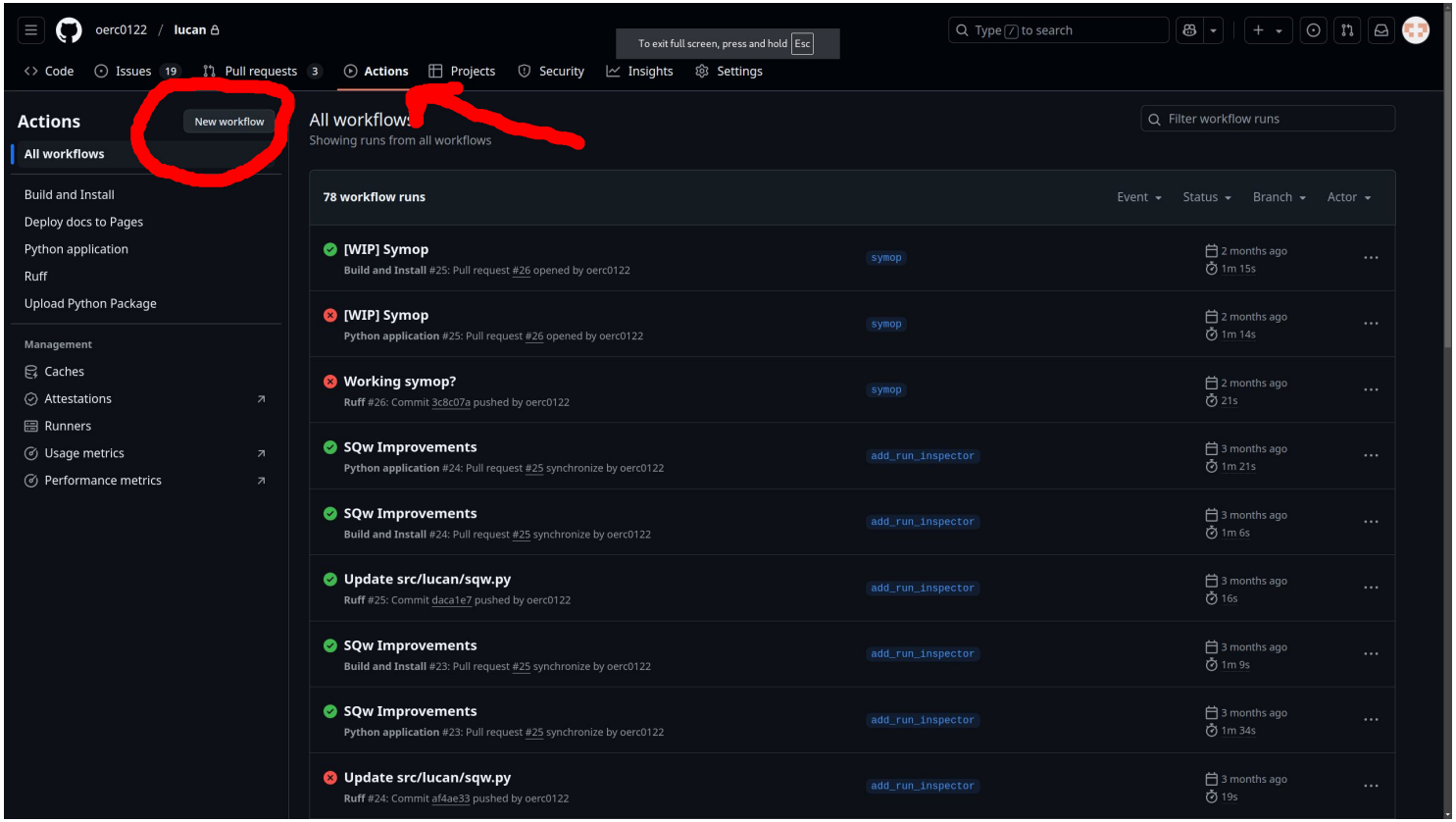
```
name: Python application

permissions:
  contents: read

on:
  push:
    branches: [ "main" ]
  pull_request:
    types:
      - opened
      - synchronize
      - reopened

jobs:
  build:

    runs-on: ubuntu-latest

    strategy:
      matrix:
        python-version: ["3.8", "3.9", "3.10"]

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
```

Testing

- For more info see
  `https://docs.github.com/en/actions`
- There is a bit of magic, using other people's scripts.
  - (The `actions/...@v3`)
- Besides that, it's just the commands you would run.
- GitHub offers Windows/Mac machines too!

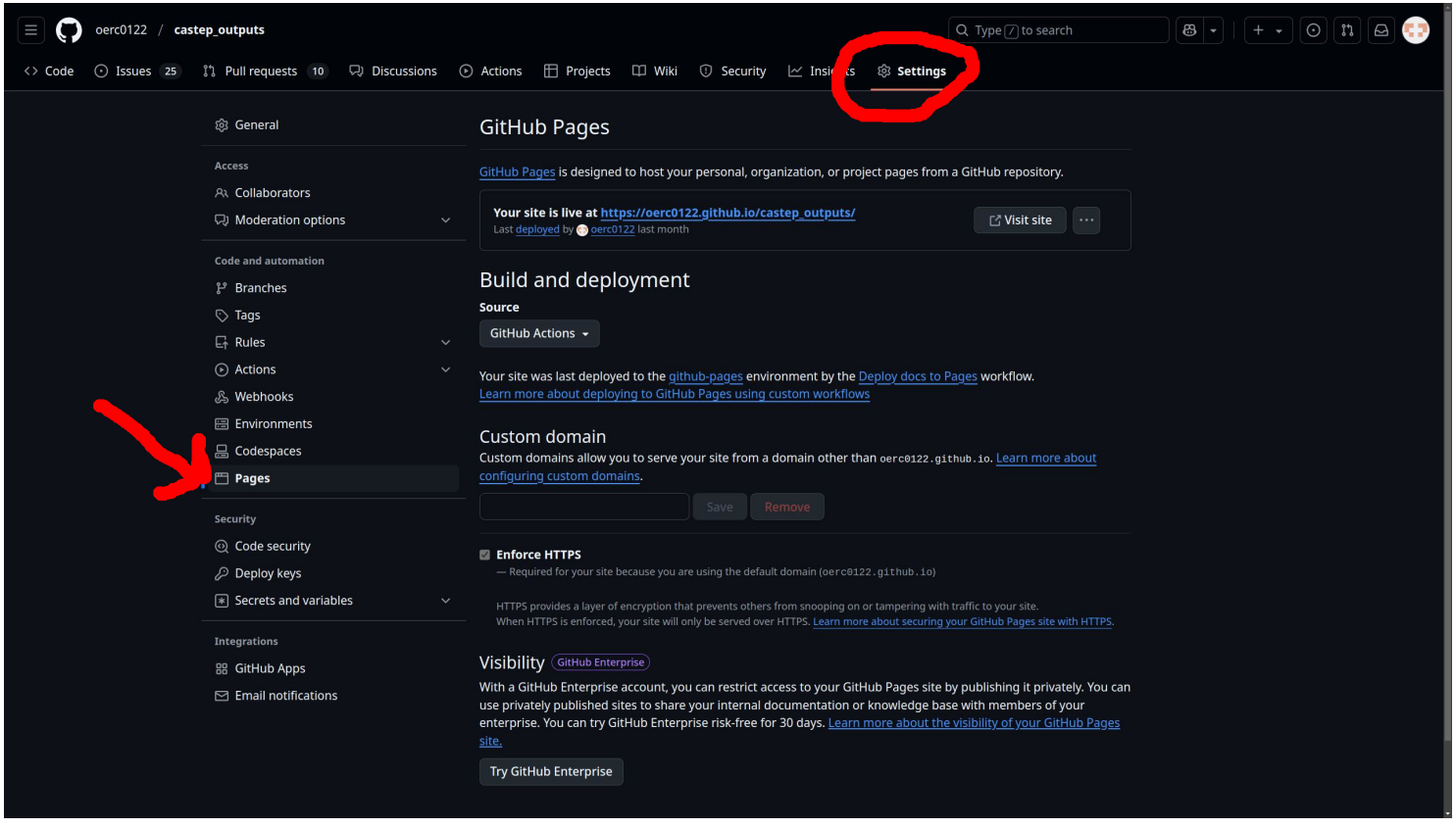
Action Economy

- GitHub contains a number of pre-written scripts for doing common jobs.
- These can be useful starting points for writing more complex scripts yourself.



Documentation in Action(s)

- But what about all the docs we've written?
- I don't want to host a website (but you can)!
- GitHub provides "GitHub pages"; sites for projects.
- These can point to a branch or be managed by actions.



Documentation in Actions(s)

Docs

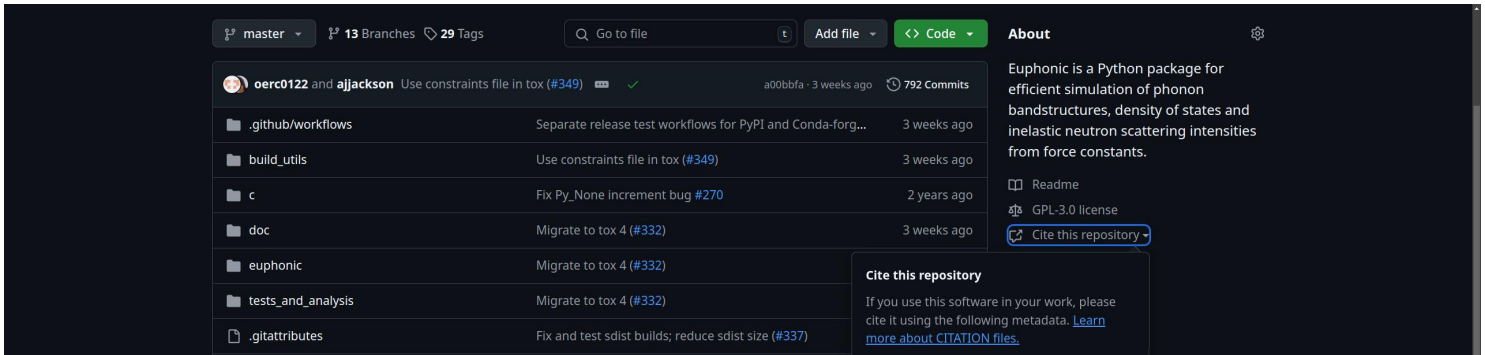
- Go to the marketplace and find an action doing (almost) what we want. (Static Pages is a good start)
- Add a bit of script to make it do (exactly) what we want.
  - Setup python.
  - Install the project. (**Note:** Remember the extras!)
  - Build the docs.
  - Upload the docs. (**Note:** part of our template)
- If you're struggling, take a look at `test_float.yml`
- If you're still struggling, `Resources/run_docs.yml` is a complete solution. See if you can work out what it's doing.
- When you're done, you should be able to go to `https://<username>.github.io/<repository>/`

Bonus

CITATION.cff

- Ok, so you've written the best project ever.
- At what point does the fame and glory start rolling in?
- Until people know how great you are that's not going to happen.
- `CITATION.cff` is a standard format for code attribution.
- `https://citation-file-format.github.io/`

```
cff-version: 1.2.0
message: "If you use this software, please cite it as below."
authors:
  - family-names: Example
    given-names: Stephen
    orcid: https://orcid.org/1234-5678-9101-1121
title: "My Research Software"
version: 2.0.4
identifiers:
  - type: doi
    value: 10.5281/zenodo.1234
date-released: 2021-08-11
```



DOIs

- Going a step further than just adding the citation file.
- Mint a DOI for a version of the software allowing it to be cited.
- STFC provides a DOI minting service.

Extra tools

- Remember we mentioned an IDE earlier in the talk?
- An IDE is just one of many tools which can be helpful in development.
  - IDE – Specialised editor to make development easier
    - (`VSCode`, `Spyder`).
  - Linting – Checks for stylistic errors
    - (`ruff`, `flake8`, `pylint`)
  - Type checking – Checks for passing the wrong data through
    - (`mypy`, `pydantic`, `beartype`)

Ruff times

- Ruff[7] is a tool to encourage (enforce) code standards.
- Modern replacement for half a dozen prior tools.
- Flags violations of code standards.
- Also handles formatting code
- Customisable through `pyproject.toml`

Adding it in

```
[project.optional-dependencies]
docs = ["sphinx",
        "sphinx-rtd-theme",
        "sphinx_autodoc_typehints"]
test = ["pytest"]
lint = ["ruff<0.13.0"]
```

Try it out!

```
ruff check
```

Try it out!

```
ruff format
```

Ruff times[8]

```yaml
name: Lint

on:
  push:
  pull_request:

lint_check_ruff:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: astral-sh/ruff-action@v3
      with:
        args: "check"

lint_format_ruff:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: astral-sh/ruff-action@v3
      with:
        args: "format --check"
```

[8] lint.yml

cookiecutter

- Possible to take out some of the boilerplate of setup.
- `cookiecutter.io` is a set of pre-configured project folders.
- Modules for many languages (including Python)

Checklist

- ☐ Sensible names
- ☐ On GitHub
- ☐ Project layout
  - ☐ `__init__.py`
  - ☐ `pyproject.toml`
- ☐ Documentation
  - ☐ Docstrings
  - ☐ Doctests
  - ☐ Typehints
  - ☐ `sphinx-quickstart`
  - ☐ `sphinx-apidoc`
- ☐ Tests
- ☐ CI/CD
  - ☐ Automated testing
  - ☐ Automatic documentation
- ☐ `CITATION.cff`