The Art and Craft of Python Packaging

Organizing your Python Code for Others

Introduction

Software Engineering Team

Department of Biomedical Informatics
SCHOOL OF MEDICINE
UNIVERSITY OF COLORADO **ANSCHUTZ MEDICAL CAMPUS**

Visit for more info: https://cu-dbmi.github.io/set-website/

Gratitude

Big ***thank you*** for attending!

Presentation Outline

1. ✍️ Publishing
2. 📦 Python Packaging
3. 🛠️ Packaging Understanding, Trust, and Connection

Why?

> “Your job as a developer is not just to create code that you can work with easily, but to create code that others can also work with easily.” — John Ousterhout, A Philosophy of Software Design

Why?

Other people working with your code:

- **Developers (*changing the code*)**
- **Users (*applying the code*)**

Why?

Developer Experience

> “DevEx refers to the systems, technology, process, and culture that influence the effectiveness of software development.” - GitHub Blog: Developer experience: What is it and why should you care?

Why?

Developer Experience can become User Experience

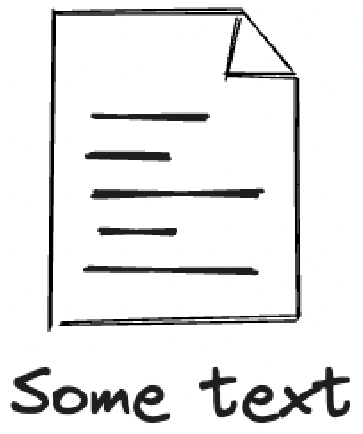
> “[O]rganizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.” - (Wikipedia: Conway’s Law)

Why?

🤷‍♂️ “I’m not convinced. My work is only for *[internal | quick | throwaway]* use so consistent organization doesn’t matter.”

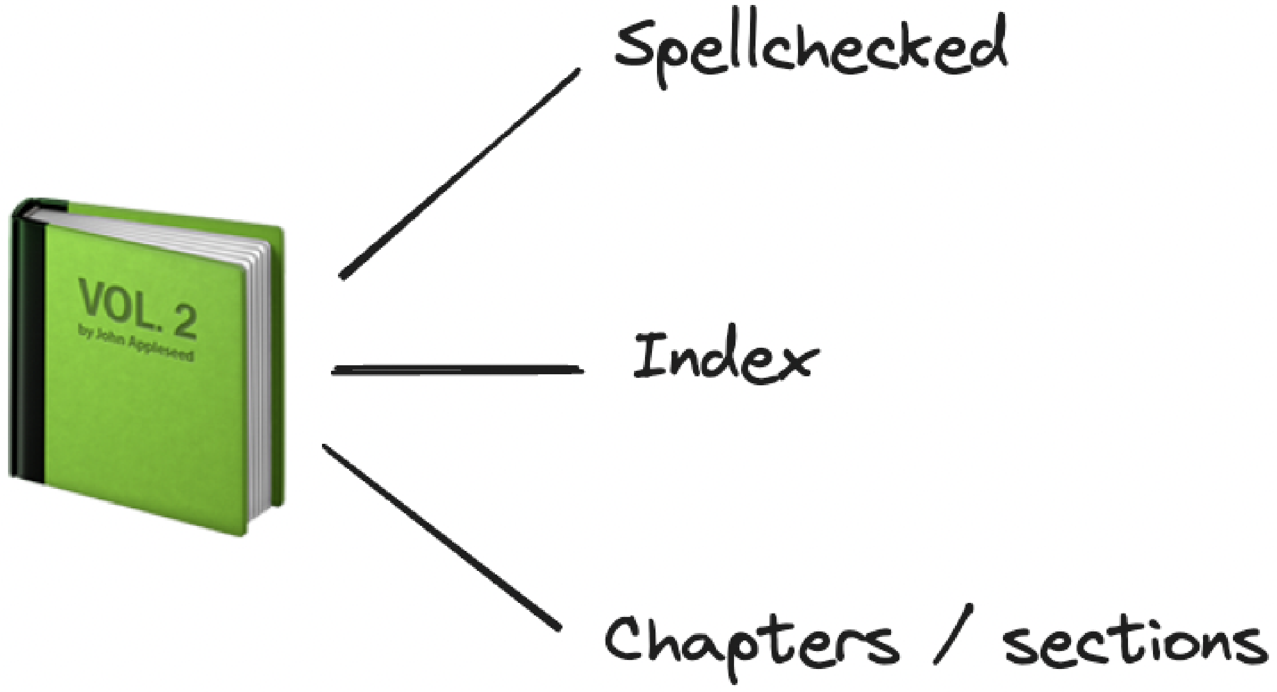
- Avoiding common practice and understanding weakens one of our collective superpowers: collaboration.
- Packaging practices **increase your development velocity** through reduced time necessary to understand and implement the code (or reuse it in the future)!
- Good packaging practices work just as well for inner source collaboration (private or proprietary development within organizations).
- How much time will you or others spend on packaging vs providing human impact from the work?

Publishing - Scratch Paper vs Book



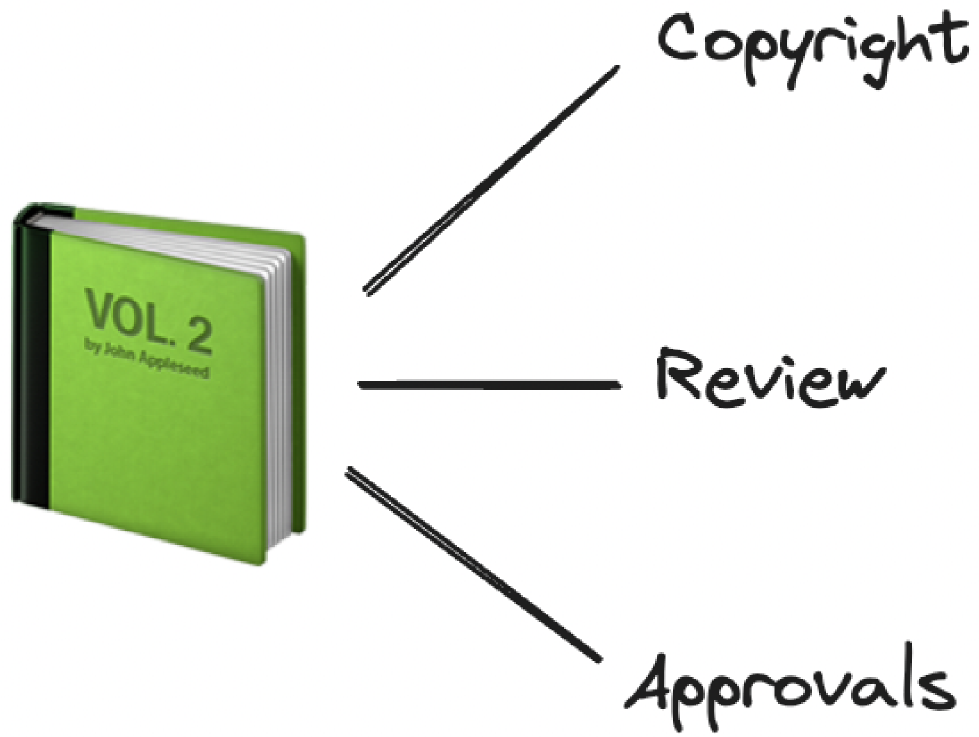
How are these different?

Publishing - Understanding



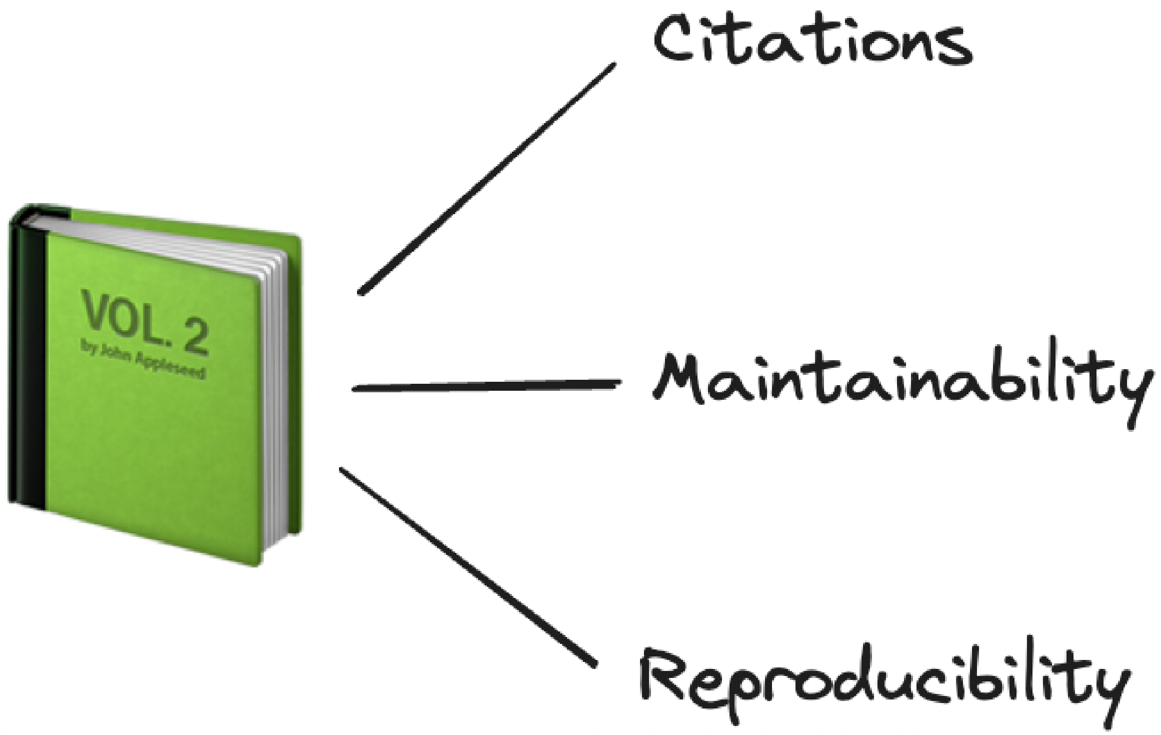
- Unsurprising formatting for **understanding** (sections, cadence, spelling, etc.)

Publishing - Trust



- Sense of **trust** from familiarity, connection, and authority (location, review, or style)

Publishing - Connection



- **Connection** to a wider audience (citations, maintainability, reproducibility)

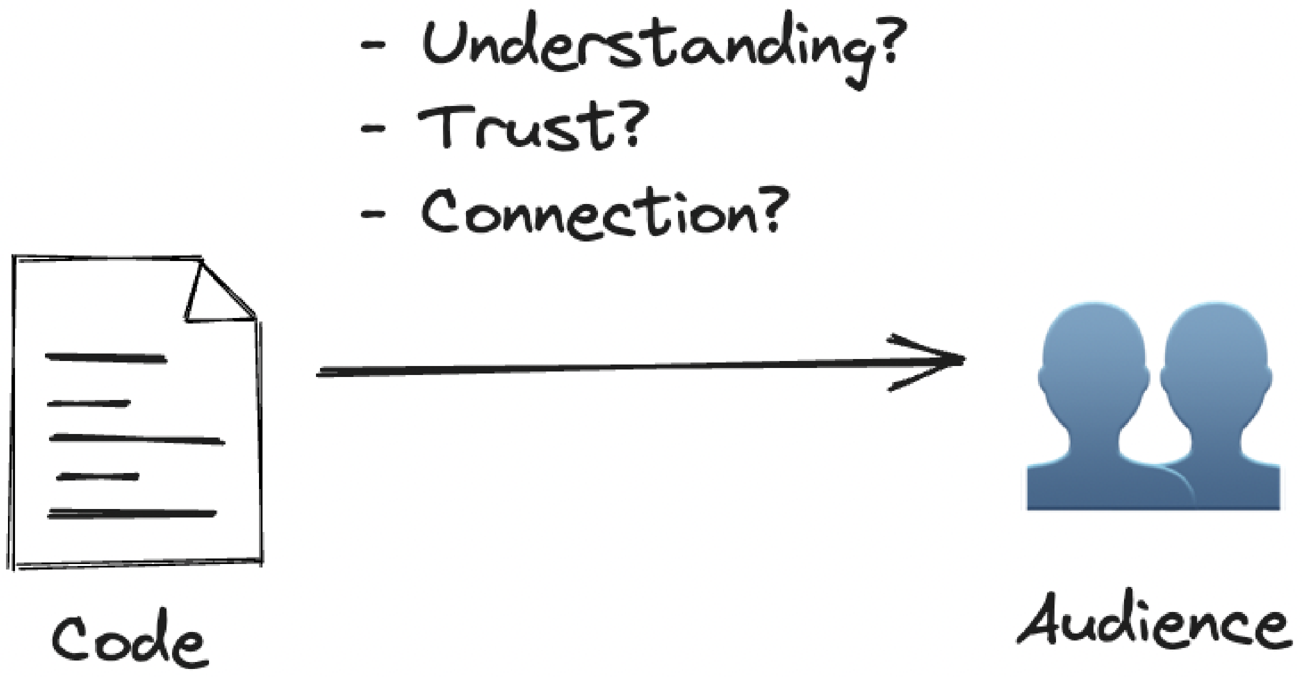
Publishing - Code as Language

Code is another kind of written language.

Ignoring language conventions can often result in poor grammar, or ***code smell***.

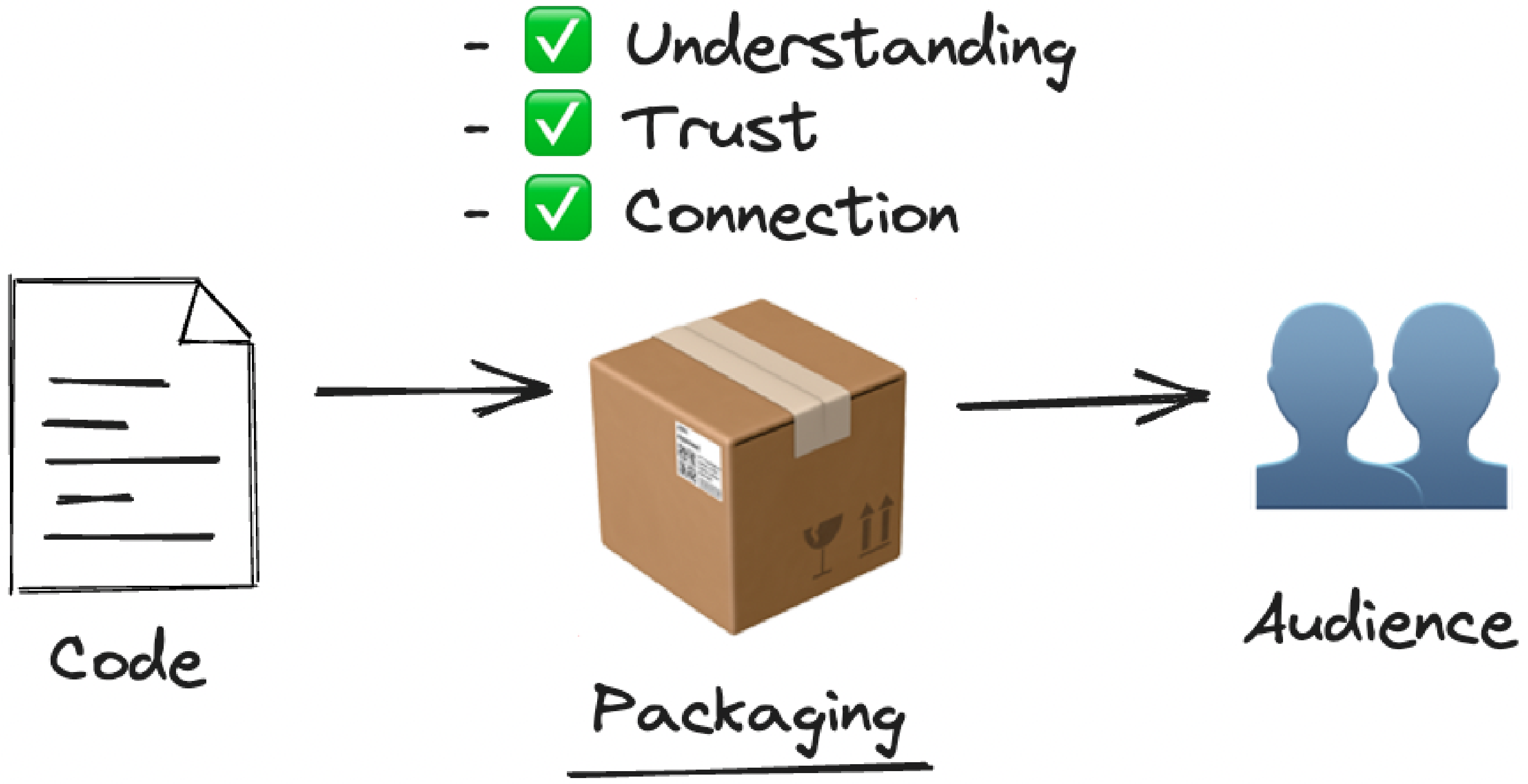
Code smells are indications that something might be going wrong. They generally reduce the understanding, trust, and connection for your code.

Publishing - Code as Language



Who are you writing for? Do they understand, trust, and connect with your code?

Publishing - Python



“**Packaging**” is the craft of preparing for and reaching distribution of your Python work.

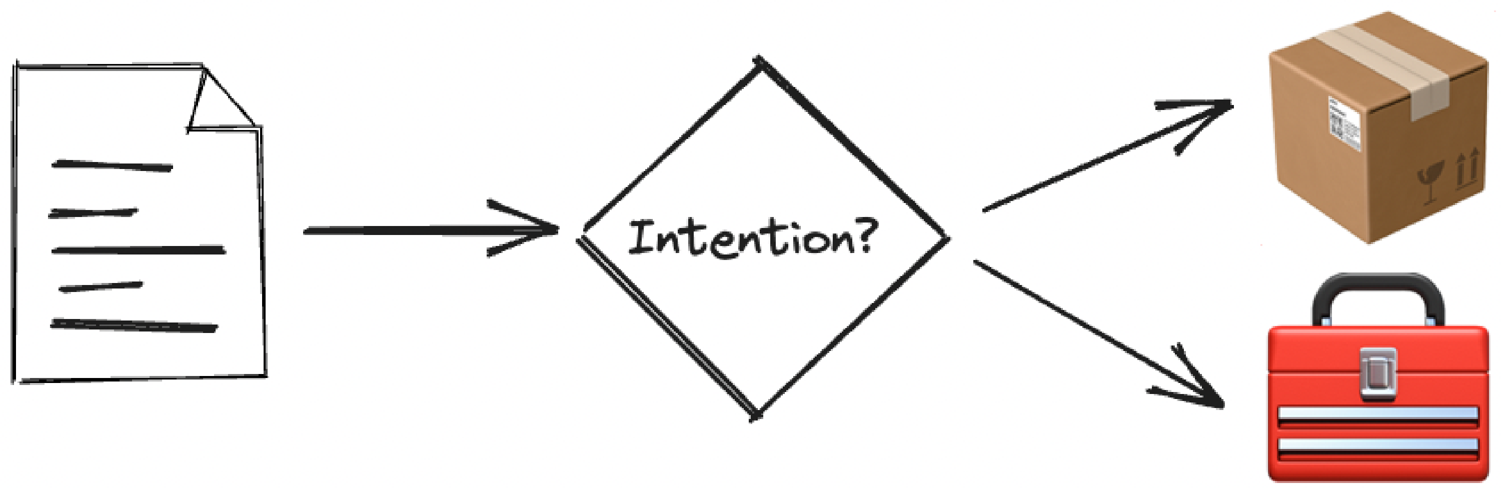
Publishing - Python

This presentation will focus on **preparations** for distribution.

(Creating a “book”, not yet sending it to “bookstores”.)

Package distribution is I feel another talk! 🙂

Publishing - Python



Python packaging is a practice which requires adjustment based on intention (there's no one-size fits all forever solution here).

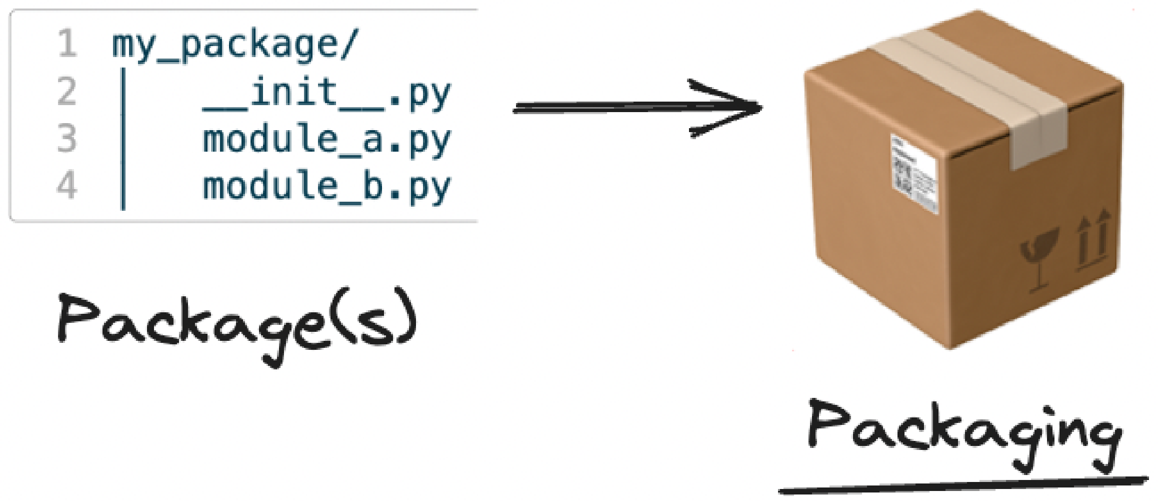
- For example: we'd package Python code differently for a patient bedside medical device use vs a freeware desktop videogame.

Python Packaging - Definitions

```
my_package/
│    __init__.py
│    module_a.py
│    module_b.py
```

- A Python **package** is a collection of modules (`.py` files) that usually include an “initialization file” `__init__.py`.

Python Packaging - Definitions



```
my_package/
│    __init__.py
│    module_a.py
│    module_b.py
```

- Python ***“packaging”*** is a broader term indicating formalization of code with publishing intent.

Python Packaging - Definitions

- Python packages are commonly installed from **PyPI** (Python Package Index, https://pypi.org).

> For example: `pip install pandas` references PyPI by default to install for the `pandas` package.

Python Packaging - Definitions

Environment

> "... a runtime system or runtime **environment** is a sub-system that exists both in the computer where a program is created, as well as in the computers where the program is intended to be run."

- Wikipedia: Runtime system

Python Packaging - Understanding

```
project_directory
├── README.md
├── LICENSE.txt
├── pyproject.toml
├── docs
│   └── source
│       └── index.md
├── src
│   └── package_name
│       └── __init__.py
│       └── module_a.py
└── tests
    └── __init__.py
    └── test_module_a.py
```

Python Packaging today generally assumes a specific directory design. Following this convention generally improves the **understanding** of your code.

Python Packaging - Understanding

- Note: Each file and directory has a specific purpose.
- Beware of dead code
- Each file should justify existence within the project.
  - Consider each file as a **“cost”**
    (Someone, someday will have to figure it out.)

Python Packaging - README.md

```
project_directory
├── README.md # used for documentation
...
```

The **`README.md`** file is a markdown file with documentation including project goals and other short notes about installation, development, or usage.

- The `README.md` file is akin to a book jacket blurb which quickly tells the audience what the book will be about.

Python Packaging - LICENSE.txt

```
project_directory
├── README.md
├── LICENSE.txt # indicates usage permissions and protections
...
```

The **`LICENSE.txt`** file is a text file which indicates licensing details for the project. It often includes information about how it may be used and protects the authors in disputes.

- The `LICENSE.txt` file can be thought of like a book's copyright page.
- See https://choosealicense.com/ for more details on selecting an open source license.

Python Packaging - pyproject.toml

```
project_directory
├── README.md
├── LICENSE.txt
├── pyproject.toml # outlines the project organization (and much more)
...
```

The **`pyproject.toml`** file is a Python-specific TOML file which helps organize how the project is used and built for wider distribution. More here later!

- The `pyproject.toml` file is similar to a book's table of contents, index, and printing or production specification.

Python Packaging - Docs Dir

```
project_directory
│...
├── docs # directory for in-depth documentation and docs build code
│   └── source
│       └── index.md
...
```

The **`docs`** directory is used for in-depth documentation and related documentation build code (for example, when building documentation websites, aka “docsites”).

- The `docs` directory includes information similar to a book’s “study guide”, providing content surrounding how to best make use of and understand the content found within.

Python Packaging - Source Code Dir

```
project_directory
|...
├── src # isolates source code for use in project
│   └── package_name
│       └── __init__.py
│       └── module_a.py
...
```

The `src` directory includes primary source code for use in the project. Python projects generally use a nested package directory with modules and sub-packages.

- The `src` directory is like a book's body or general content (perhaps thinking of modules as chapters or sections of related ideas).

Python Packaging - Test Code Dir

```
project_directory
│...
├── src
│   └── package_name
│       └── __init__.py
│       └── module_a.py
│
└── tests # organizes the validation of source code
    └── __init__.py
    └── test_module_a.py
```

The **`tests`** directory includes testing code for validating functionality of code found in the `src` directory. The above follows pytest conventions.

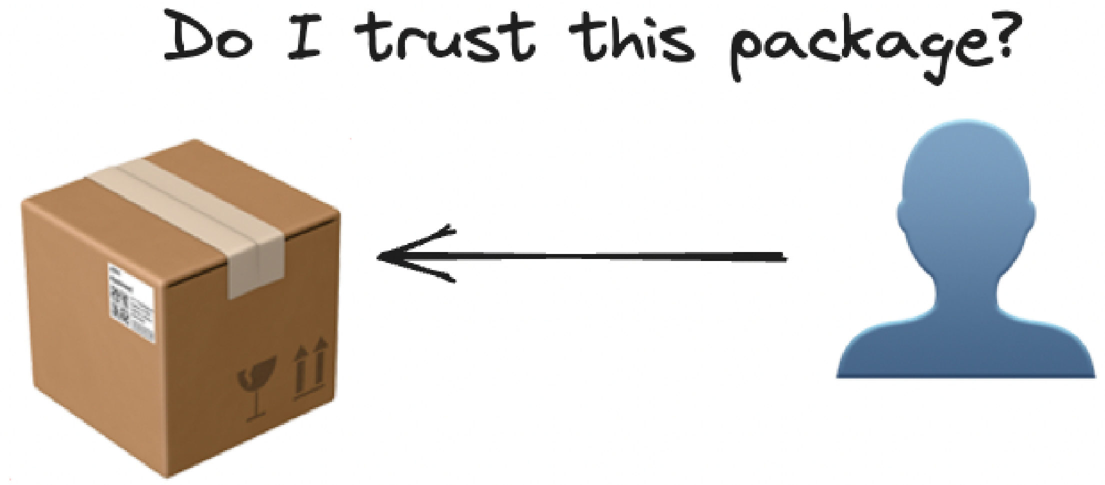
- The `tests` directory is for code which acts like a book's early reviewers or editors, making sure that if you change things in `src` the impacts remain as expected.

Python Packaging - Examples in the wild

The described Python directory structure can be witnessed in the wild from the following resources:

- `pypa/sampleproject`
- `microsoft/python-package-template`
- `scientific-python/cookie`
- … and so many more!

Python Packaging - Trust



Building an understandable body of content helps tremendously with audience trust.

- What else can we do to enhance project trust?

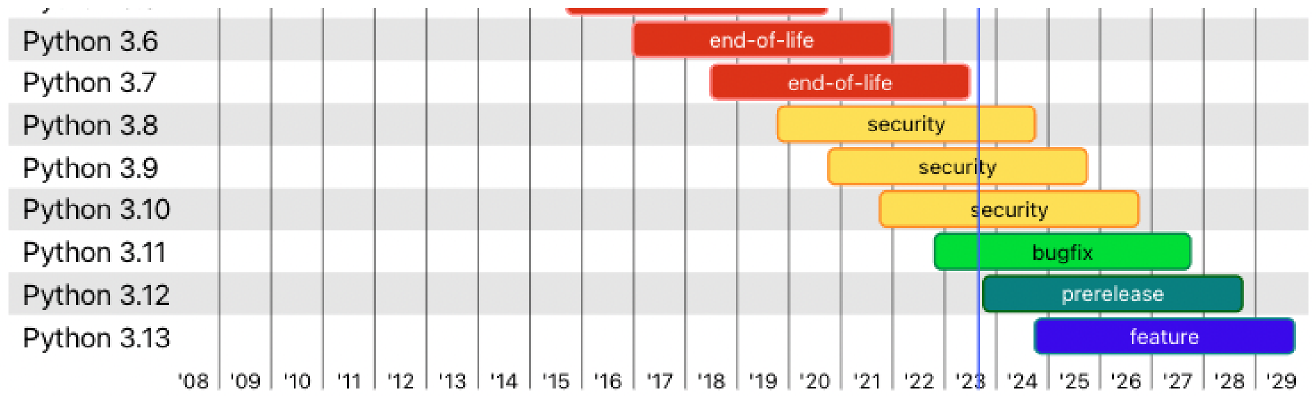
Python Packaging - Be authentic

user123
vs.
BioMagician

Be authentic! Fill out your profile to help your audience know the author and why you do what you do. See here for GitHub's documentation on filling out your profile.

- Add a profile picture of yourself or something fun.
- Make it customized and unique to you!

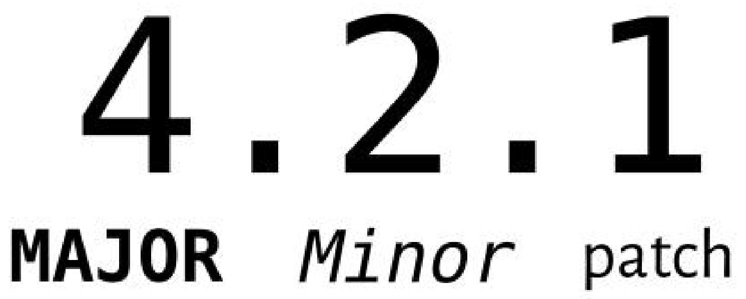
Python Packaging - Python versions



Use Python versions which are supported (this changes over time). Python versions which are end-of-life may be difficult to support and are a sign of code decay for projects.

- See here for updated information on Python version status.

Python Packaging - Your versions



Your software can also implement versioning.

One way: Semantic Versioning.

What version is your software today? What will it be tomorrow? Does the previous one still work?

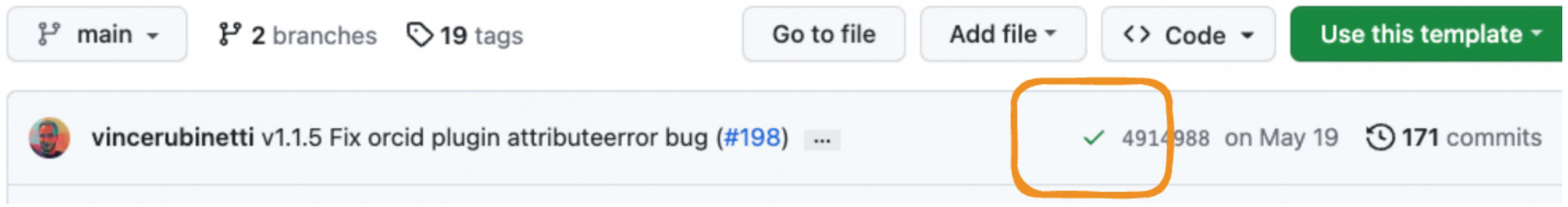
- Image from Wikimedia Commons: Maxime Lathuilière

Python Packaging - Security linters

Use security vulnerability linters to help prevent undesirable or risky processing for your audience. Doing this both practical to avoid issues and conveys that you care about those using your package!

- PyCQA/bandit: checks Python code
- pyupio/safety: checks Python dependencies
- gitleaks: checks for sensitive passwords, keys, or tokens

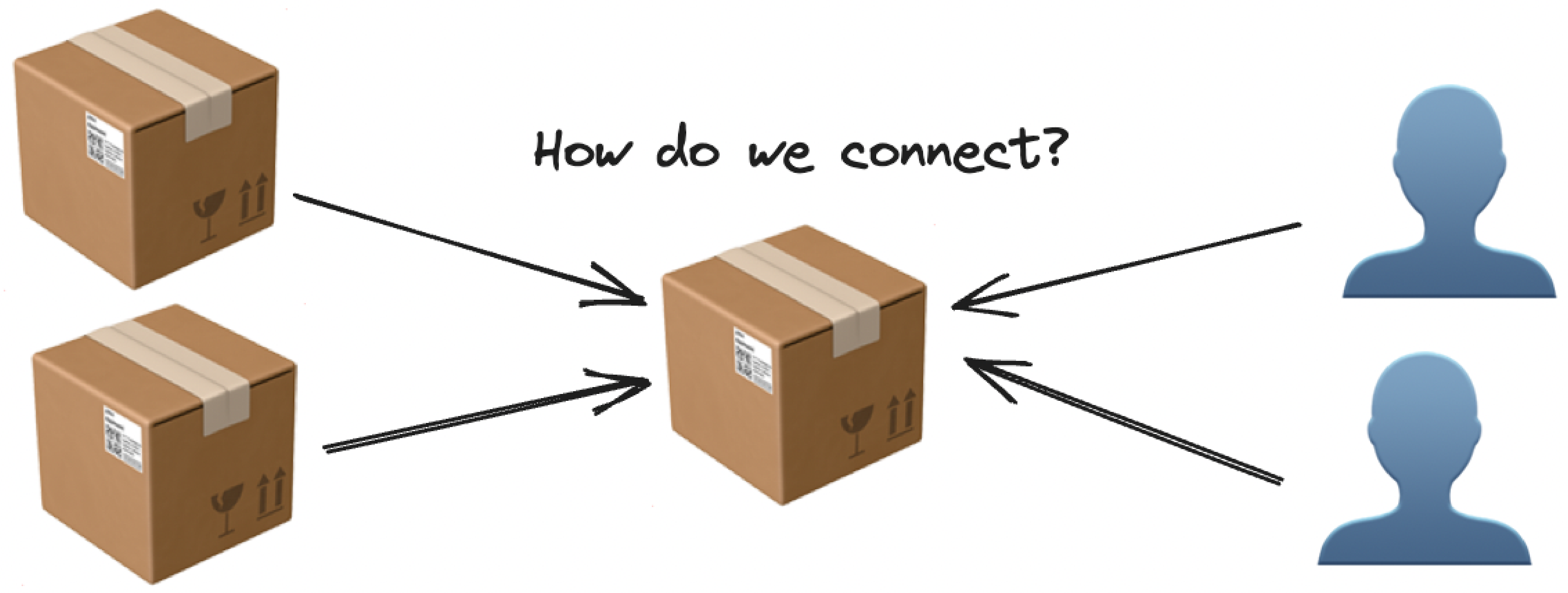
Python Packaging - GitHub Actions



Combining GitHub actions with security linters and tests from your software validation suite can add an observable ✅ for your project. This provides the audience with a sense that you're transparently testing and sharing results of those tests.

- See GitHub's documentation on this topic for more information.
- See also DBMI SET's blog post on "Automate Software Workflows with Github Actions"

Python Packaging - Connection



Understandability and trust set the stage for your project's **connection** to other people and projects.

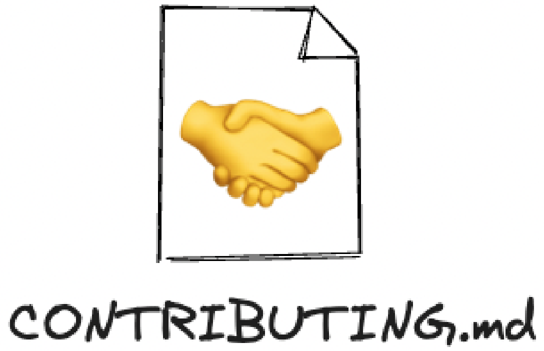
- What can we do to facilitate connection with our project?

Python Packaging - CITATION.cff

Add a `CITATION.cff` file to your project root in order to describe project relationships and acknowledgements in a standardized way. The CFF format is also GitHub compatible, making it easier to cite your project.

- This is similar to a book's credits, acknowledgements, dedication, and author information sections.
- See here for a `CITATION.cff` file generator (and updater).

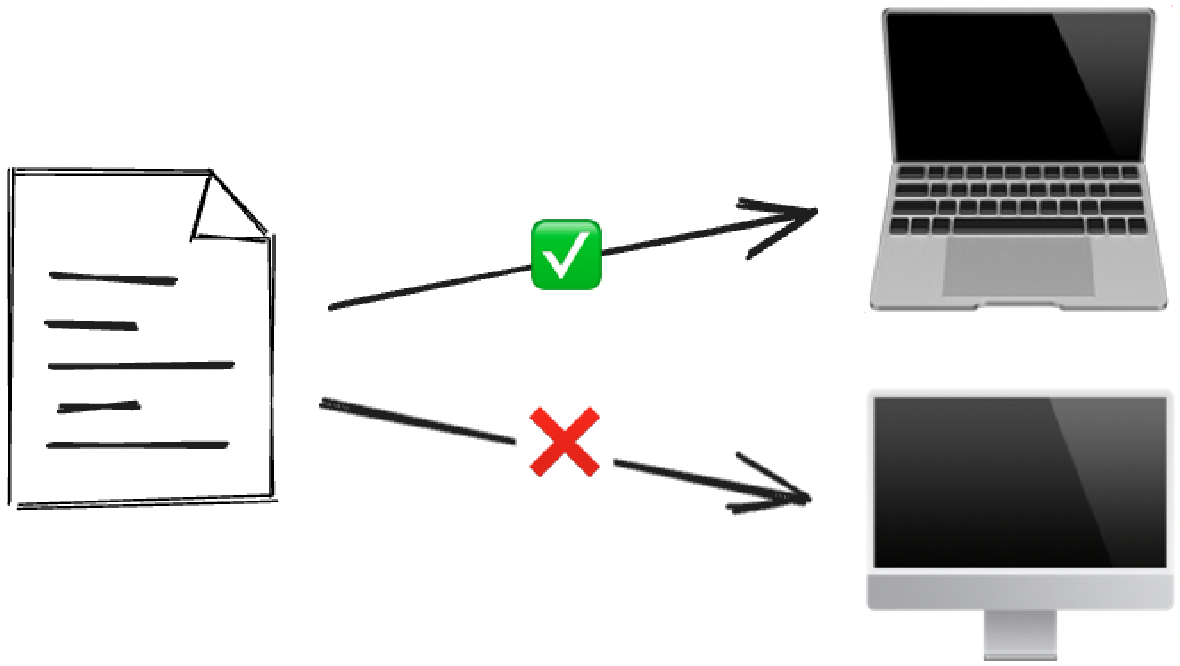
Python Packaging - CONTRIBUTING.md



Provide a `CONTRIBUTING.md` file to your project root so as to make clear support details, development guidance, code of conduct, and overall documentation surrounding how the project is governed.

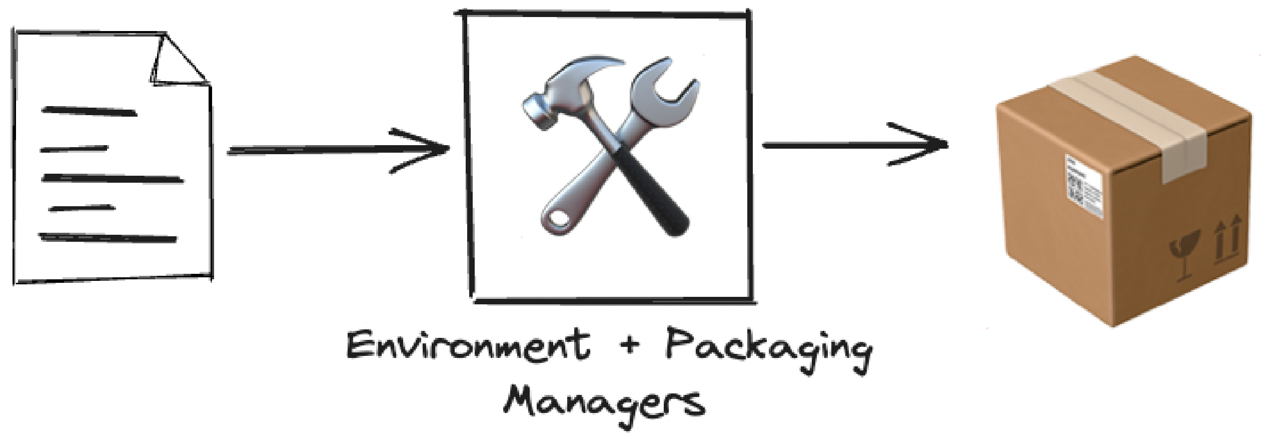
- See GitHub's documentation on "Setting guidelines for repository contributors"
- See opensource.guide's section on "Writing your contributing guidelines"

Python Packaging - Reproducibility



Code without an environment specification is difficult to run in a consistent way. This can lead to “works on my machine” scenarios where different things happen for different people, reducing the chance that people can connect with your code.

Python Packaging - Environments



Use **Python environment and packaging managers** to help unify how developers use or maintain your project. These tools commonly extend `pyproject.toml` files to declare environment and packaging metadata.

- Environment/dependency management facilitate how code is ***processed***.
- Packaging management facilitates how code is ***built*** for distribution.

Python Packaging - Brief history

“But why do we have to switch the way we do things?”

We’ve always been switching approaches! A brief history of Python environment and packaging tooling:

1. `distutils`, `easy_install` + `setup.py` (primarily used during 1990’s - early 2000’s)
2. `pip`, `setup.py` + `requirements.txt` (primarily used during late 2000’s - early 2010’s)
3. `poetry` + `pyproject.toml` (began use around late 2010’s - ongoing)

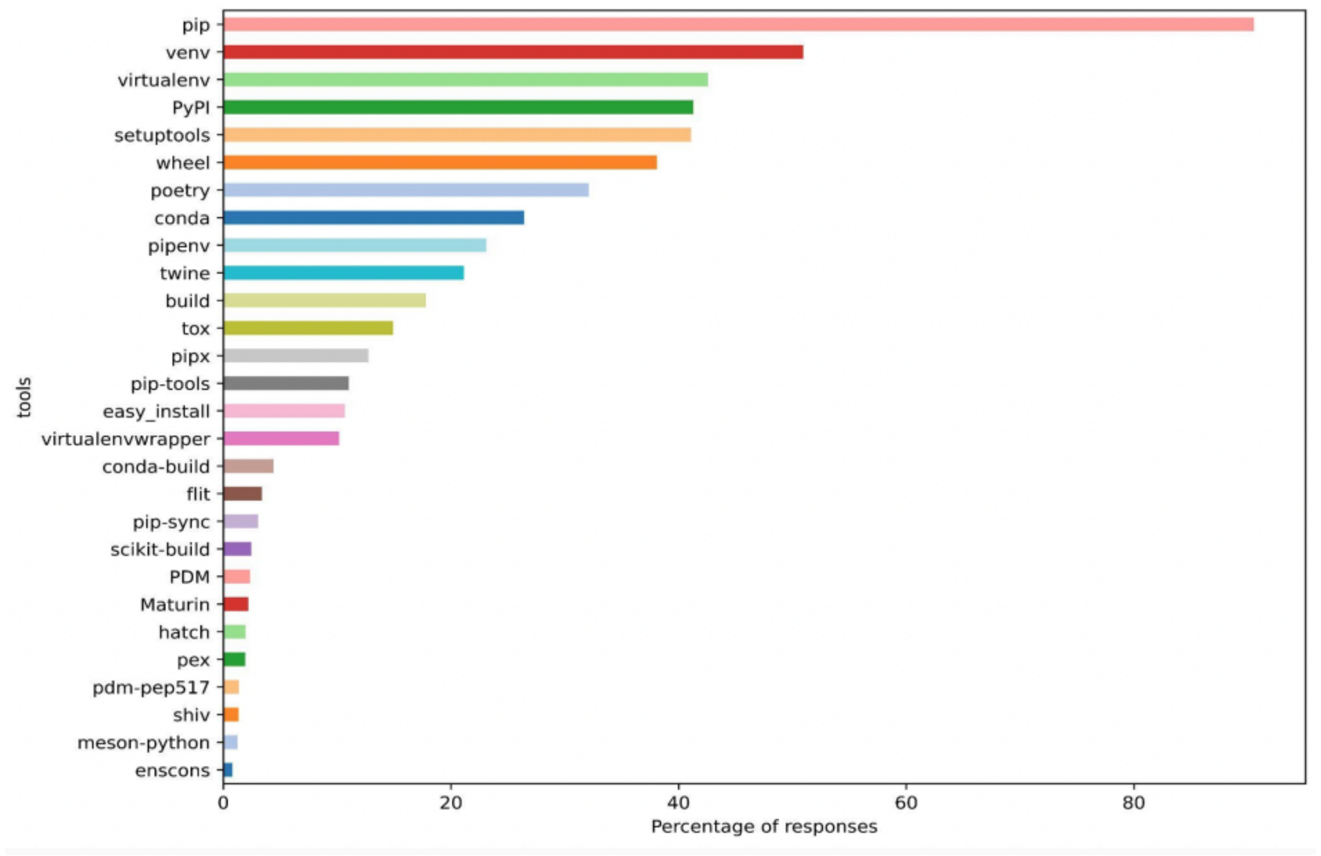
Python Packaging - Poetry

Poetry is one Pythonic environment and packaging manager which can help increase reproducibility using `pyproject.toml` files.

- It's one of many other alternatives such as `hatch` and `pipenv`.

Python Packaging - Poetry

Which Python packaging tools do you use? (survey responses, October 2022)



Poetry in context. (Source: Shamika Mohanan, Python Packaging User Survey, Oct 2022)

Python Packaging - Poetry

pyOpenSci has a great review of package build tools!

Python Packaging - Poetry

```
user@machine % poetry new --name=package_name --src .
Created package package_name in .

user@machine % tree .
.
├── README.md
├── pyproject.toml
├── src
│   └── package_name
│       └── __init__.py
└── tests
    └── __init__.py
```

After installation, Poetry gives us the ability to initialize a directory structure similar to what we presented earlier by using the `poetry new ...` command.

Python Packaging - Poetry

```toml
# pyproject.toml
[tool.poetry]
name = "package-name"
version = "0.1.0"
description = ""
authors = ["username <email@address>"]
readme = "README.md"
packages = [{include = "package_name", from = "src"}]

[tool.poetry.dependencies]
python = "^3.9"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Using this command also initializes the content of our `pyproject.toml` file with opinionated details.

Python Packaging - Poetry

```
user@machine % poetry add pandas

Creating virtualenv package-name-1STl06GY-py3.9 in /pypoetry/virtualenvs
Using version ^2.1.0 for pandas

...

Writing lock file
```

We can add dependencies directly using the `poetry add ...` command.

- A local virtual environment is managed for us automatically.
- A `poetry.lock` file is written when the dependencies are installed to help ensure the version you installed today will be what's used on other machines.

Python Packaging - Poetry

```
% poetry run python -c "import pandas; print(pandas.__version__)"

2.1.0
```

We can invoke the virtual environment directly using `poetry run ....`

- This allows us to quickly run code through the context of the project's environment.
- Poetry can automatically switch between multiple environments based on the local directory structure.

Python Packaging - Poetry

```
% poetry build

Building package-name (0.1.0)
  - Building sdist
  - Built package_name-0.1.0.tar.gz
  - Building wheel
  - Built package_name-0.1.0-py3-none-any.whl
```

Poetry readies source-code and pre-compiled versions of our code for distribution platforms like PyPI by using the `poetry build ...` command.

Python Packaging - Distribution

```
% pip install git+https://github.com/project/package_name
```

Even if we don’t reach wider distribution on PyPI or elsewhere, source code managed by `pyproject.toml` and the other techniques mentioned in this presentation can be used for “manual” distribution (with reproducible results).

Python Packaging - Poetry

What about Makefiles?

We often include `Makefiles` to help run frequently used development commands (for example, the `poetry build` or related).

Makefile-like definitions can be included in Poetry files `pyproject.toml` files using **Poe the Poet**.

Python Packaging - Poetry

`Makefile`

```
...

.PHONY: build

build:
    poetry run pytest
    poetry build

...
```

`pyproject.toml`

```
...

[tool.poe.tasks]
test   = "pytest"
_build = "poetry build"
build  = ["test", "_build"]

...
```

One fewer file and syntax to manage.
(Again, think about the cost of each file.)

Thank you!

Thank you for attending! Questions / comments?

Please don’t hesitate to reach out!

Question & Answer

Question: How can you know what's best for Python and packaging over time?

The adage `"Change is the only constant."` holds true for Python and packaging over time. Elements shared through this presentation will eventually no longer be best practices or be outright wrong. Software development in general often entails continuous new "tool and approach deluge," with many different things appearing during short periods of time. Building opinion and forming taste about these new tools and approaches can take time. Try things you see and find out how they work (or don't) for you!

If you notice something unpleasant about your developer or user experience which is related to packaging, it may be that someone has already felt the same way as you. One way to stay tuned into best practices within the scope of Python is to read Python Enhancement Proposals (PEPs) and relatedly, read Python discussion board posts (https://discuss.python.org/). Searching through these resources can help you stay informed.

Keep in mind there's no absolute perfection here. "Breaking" an accepted PEP doesn't mean that the software won't work. It might be that you need to use technologies or techniques which don't abide PEPs. However, it could be important to learn what others like you have run into and how they opted to solve it by reading their thoughts, especially if it was accepted by a wider community.

Question & Answer

Question: How does standard Python packaging apply to data science projects (where goals or environments may differ from a Python package)?

Data science Python projects may entail unique aspects that can change the way development best takes place. This might mean, for instance, that Poetry or what was presented here won't be the best fit. If that's the case, there likely are other tools which can enhance the project's ability to achieve outcomes (explore what's out there!).

The practice of "packaging" and following an understandable organization of files and content for a Python project can still be immensely helpful, especially when considering the cycle of onboarding and offboarding members of a team. The more bespoke or unique the file organization, the more time and complexity will be associated with learning *how* to develop for or use the project rather than creating impact or value from it.

Sometimes data science projects emerge with software organization as a "secondary focus". Refactoring can take time and be difficult to undertake after a project has already achieved outcomes. Maintaining a data science project long-term can be improved by following software best practices like packaging. Packaging as a practice helps increase the sustainability of the project through common understanding (aligning to existing conventions for somewhat unified perspective), increased trust (for ex.sometimes as scientific reproducibility), and connection (for ex. illustrating how the conclusions were derived through citation).

Question & Answer

Question: How can we engage others from the community using `CONTRIBUTING.md` and other community health files?

The `CONTRIBUTING.md`, `CODE_OF_CONDUCT.md`, and `LICENSE.txt` are the starting point for opening the doors to your project for others. These files stipulate the introduction and mutual protections for people interested in engaging with the project as well as the repercussions for misbehavior. Without these files it might seem a bit mysterious or uninviting for someone to engage; they might assume it's a solely private endeavor or fear their contributions as being not valued (or deleted entirely!). Providing these files upfront sends an open and understood message to others about how the project works and how contributions are handled.

Ensuring project contributors abide what is outlined in community health files is often manual but may be enhanced through various checks or linters.

- For example, alex (https://alexjs.com/) is a tool which can be used to "lint" for considerate writing.
- Blocklint (https://github.com/PrincetonUniversity/blocklint) is another similar tool.