




Packaging as Publishing

Python Code Formalizations

Presentation Outline

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

Why?

Why does this matter?

Following publishing conventions increases:


- Understanding
- Trust
- Reach (connection)

Why?

🙋 “I’m not convinced. My work is only for *[internal | quick | throwaway]* use so packaging / publishing for others doesn’t matter.”

- Good packaging practices work just as well for **inner source** collaboration (private or proprietary development within organizations).
- It also **increases your development velocity** through reduced time necessary to understand and implement the code (or reuse it in the future)!
- Internal practice can be a great way to ready yourself for the needs of publicly available open source software maintenance.

Why?

 “I’d rather use a bespoke method of packaging my work. It’s not my responsibility to help others understand my customizations.”

- Avoiding common practice and understanding weakens one of our collective superpowers: collaboration.
- How much time will you or others spend on packaging vs providing human impact from the work?

Publishing - Text vs Book



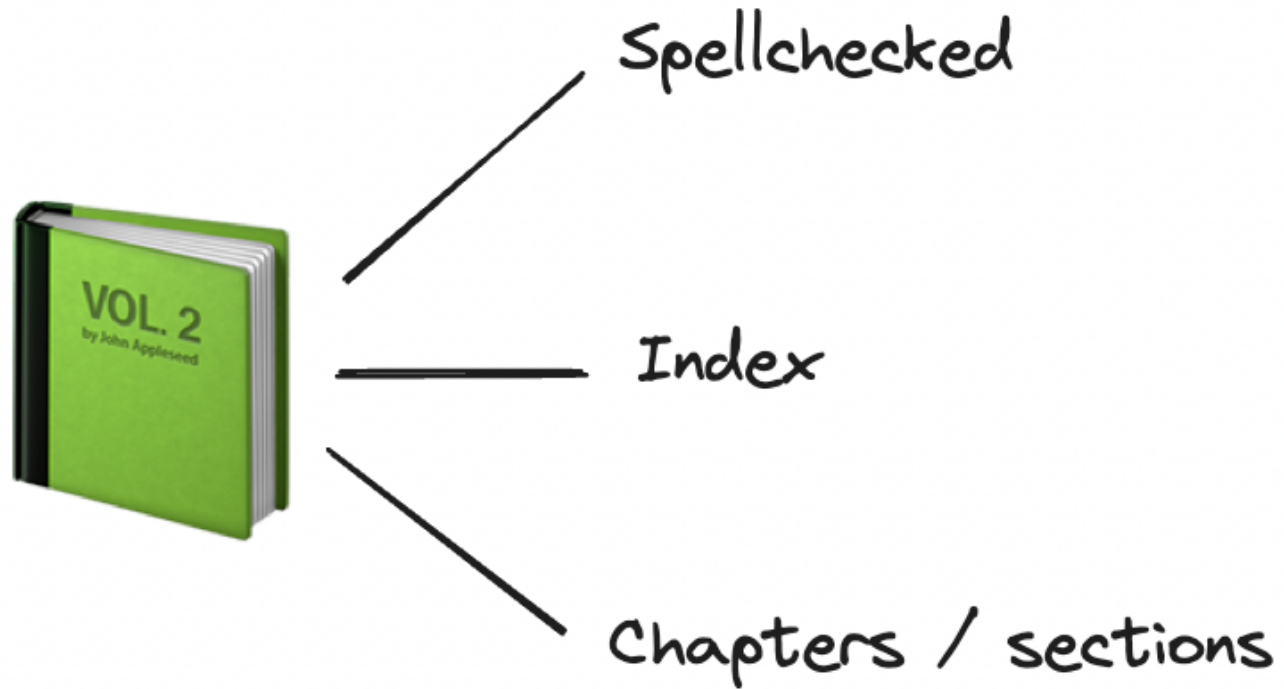
Some text



Book

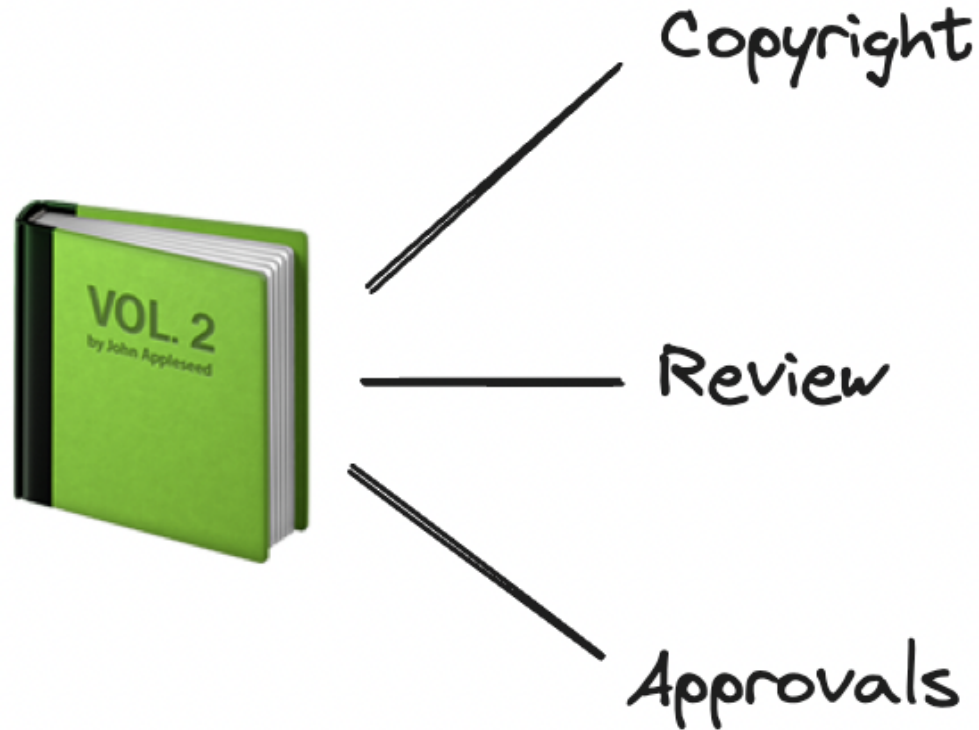
How are these two different?

Publishing - Understanding



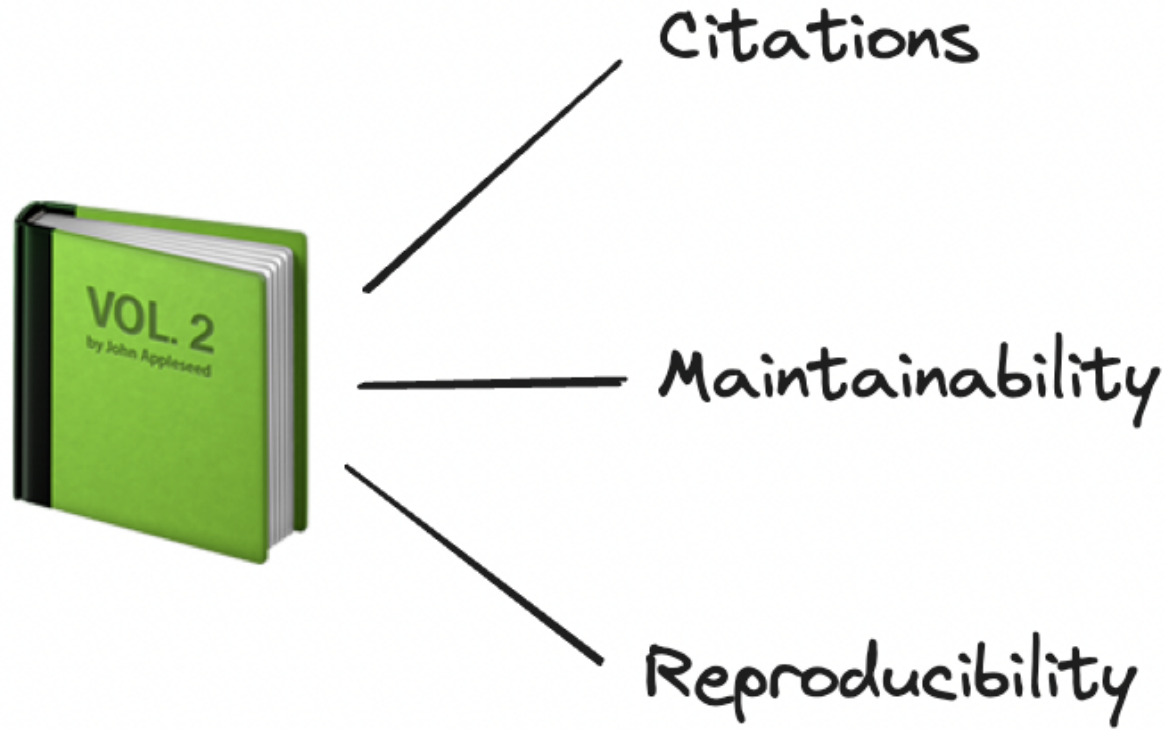
- Unsurprising formatting for **understanding** (sections, cadence, spelling, etc.)

Publishing - Trust



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

Publishing - Connection



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

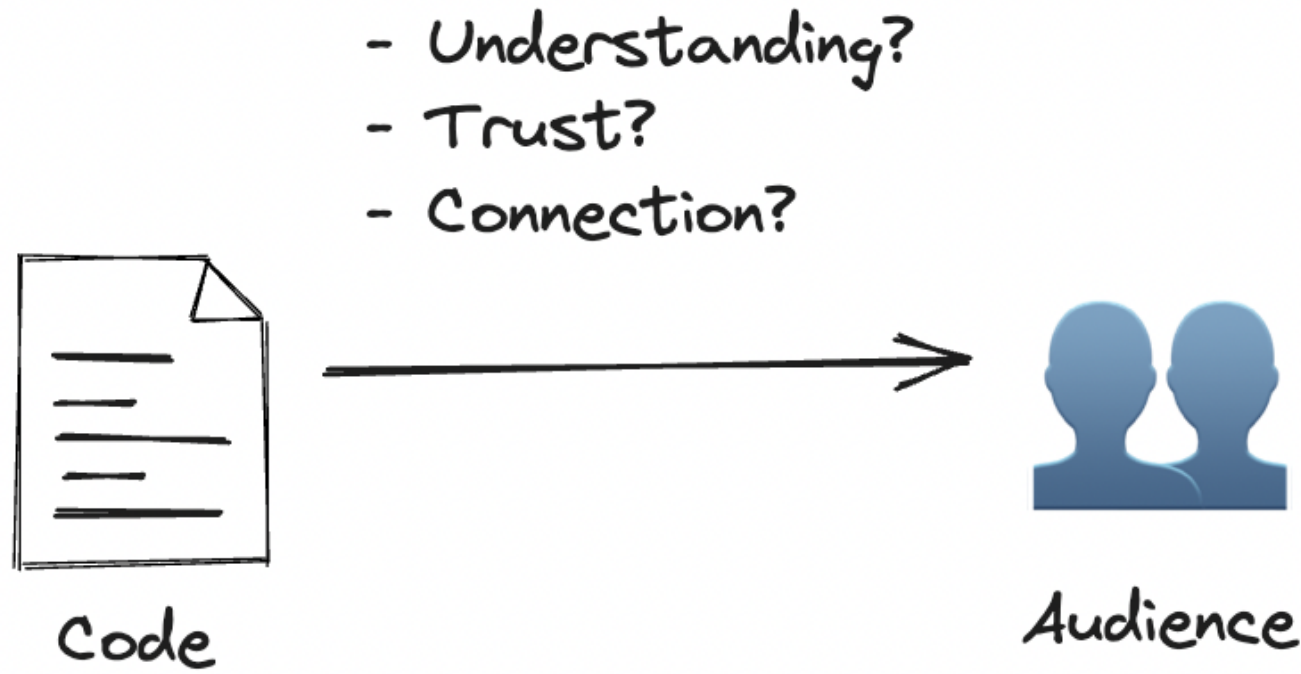
Publishing - Code as Language

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

-  Understanding
-  Trust
-  Connection

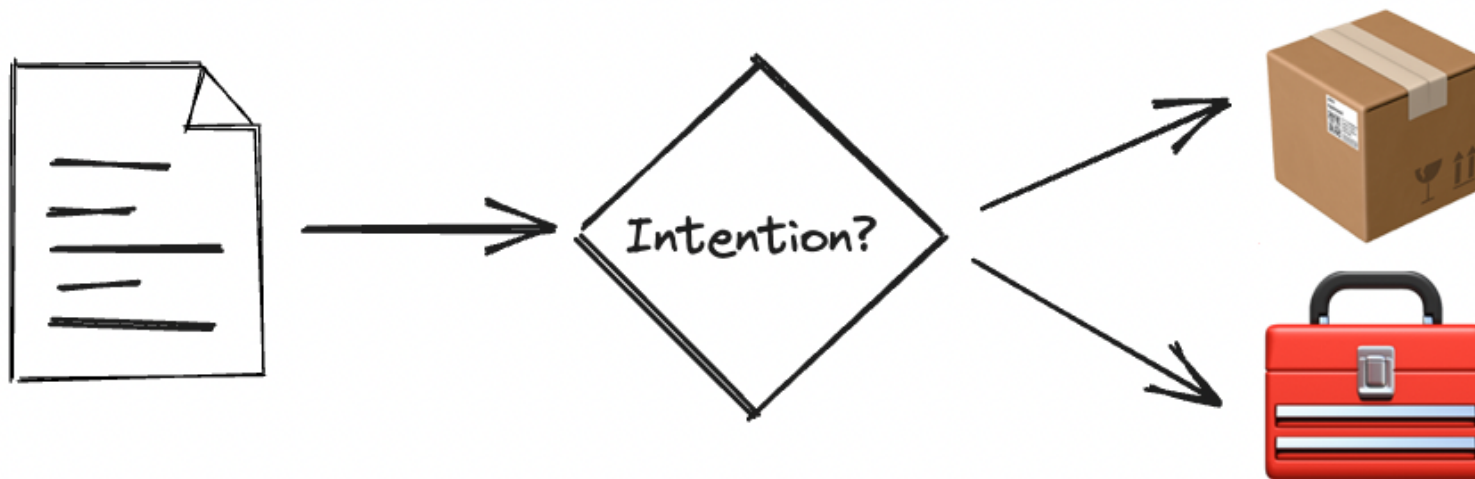


“**Packaging**” is the craft of preparing for and reaching distribution of your Python work.

Publishing - Python

This presentation will focus on preparations for distribution.

Publishing - Python



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

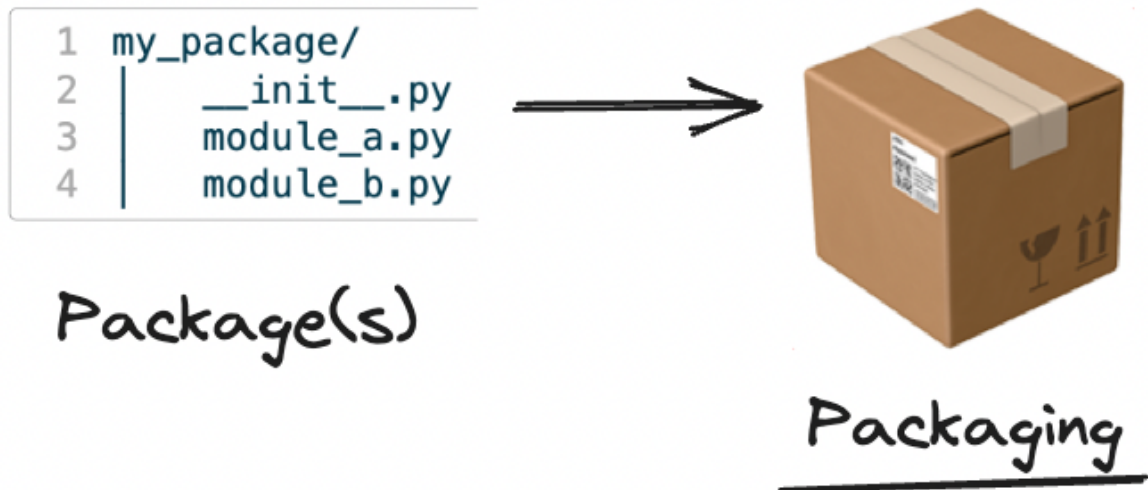
- For example: we'd package Python code differently for a patient bedside medical device use vs a freeware desktop videogame.

Python Packaging - Definitions

```
1 my_package/  
2 |   __init__.py  
3 |   module_a.py  
4 |   module_b.py
```

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

Python Packaging - Definitions



- 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 - Understanding

```
1 project_directory
2 |— README.md
3 |— LICENSE.txt
4 |— pyproject.toml
5 |— docs
6 |   |— source
7 |       |— index.md
8 |— src
9 |   |— package_name
10 |       |— __init__.py
11 |       |— module_a.py
12 |— tests
13 |   |— __init__.py
14 |   |— 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 - README.md

```
1 project_directory
2 |— README.md # used for documentation
3 ...
```

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

```
1 project_directory
2 |— README.md
3 |— LICENSE.txt # indicates usage permissions and protections
4 ...
```

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

```
1 project_directory
2 |— README.md
3 |— LICENSE.txt
4 |— pyproject.toml # outlines the project organization (and much more)
5 ...
```

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

```
1 project_directory
2 | ...
3 | — docs # directory for in-depth documentation and docs build code
4 |   | — source
5 |   |   | — index.md
6 | ...
```

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

```
1 project_directory
2 | ...
3 | — src # isolates source code for use in project
4 |   | — package_name
5 |   |   | — __init__.py
6 |   |   | — module_a.py
7 | ...
```

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

```
1 project_directory
2   ...
3   └── src
4       ├── package_name
5           ├── __init__.py
6           └── module_a.py
7
8   └── tests # organizes the validation of source code
9       ├── __init__.py
10      └── 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.

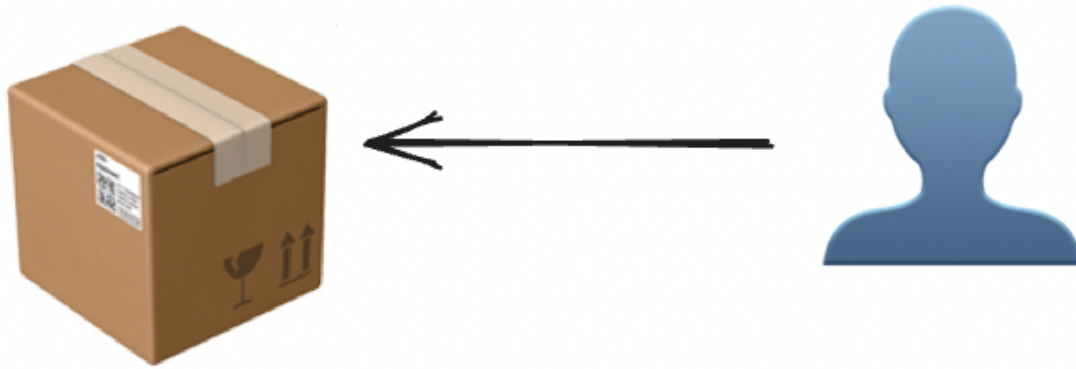
Python Packaging - Examples in the wild

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

Do I trust this package?



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.

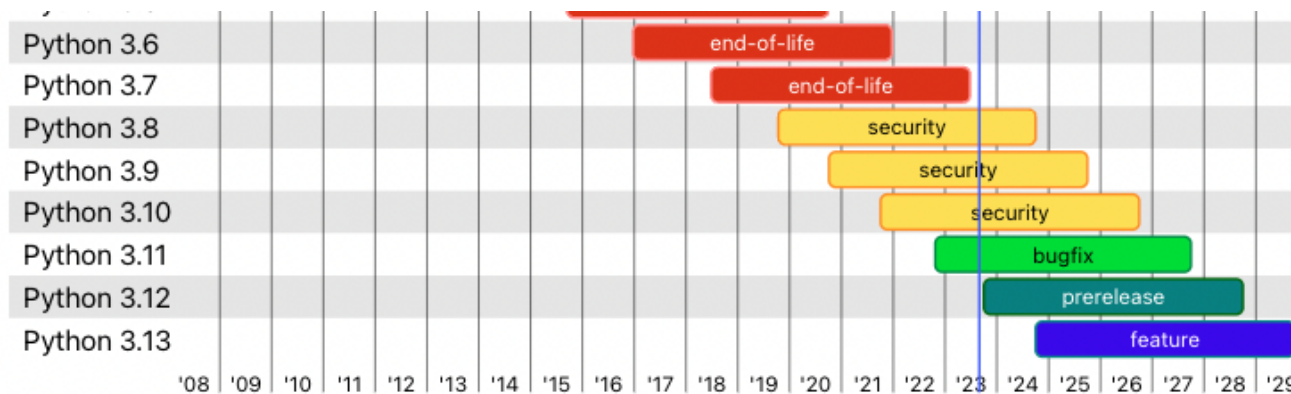


BioMagician

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](#).

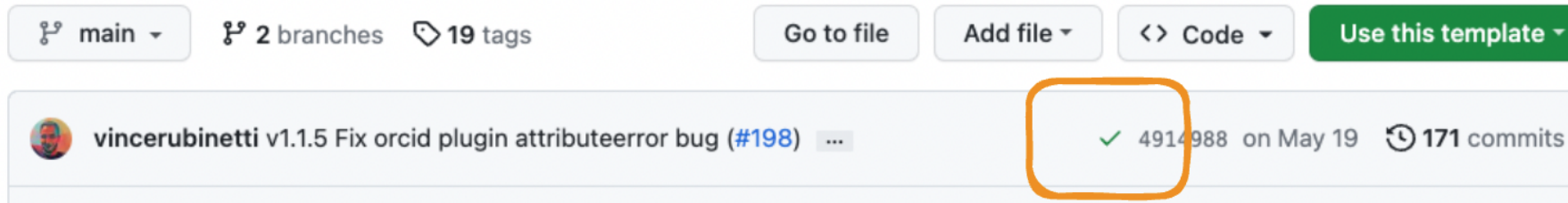
Python Packaging - Security linters




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

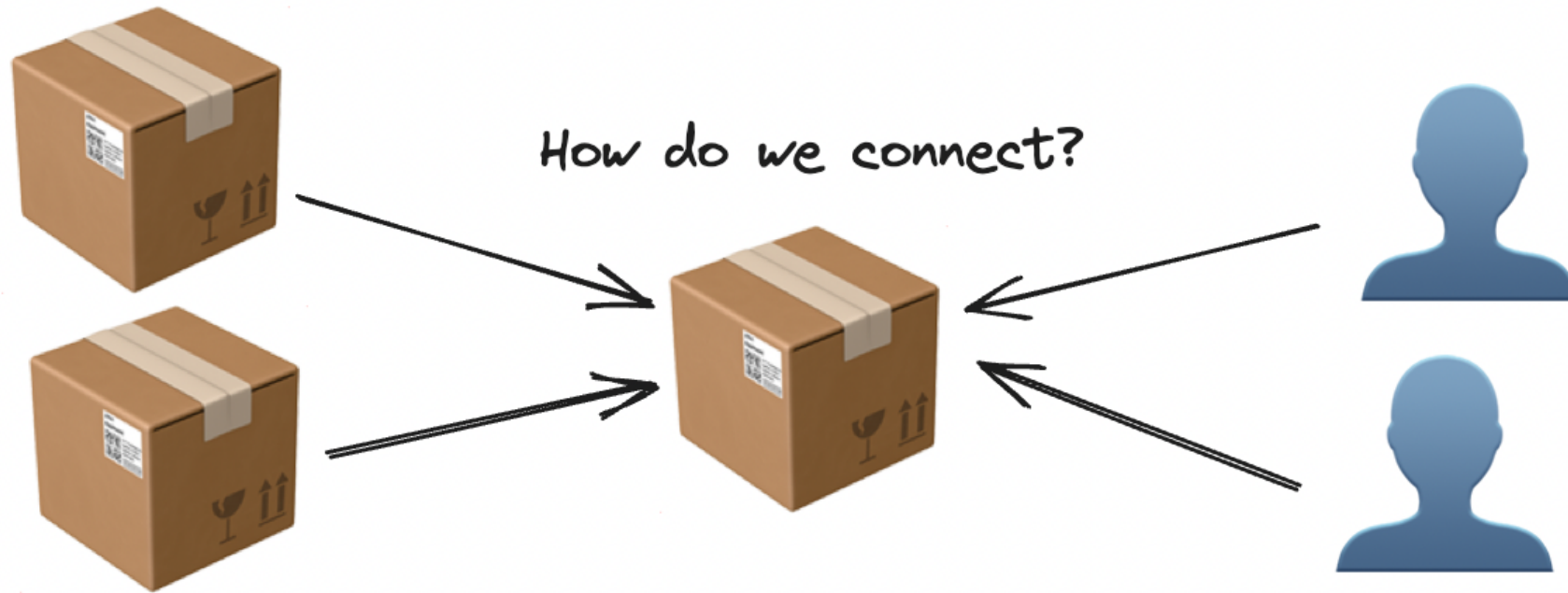
Python Packaging - GitHub Actions



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?

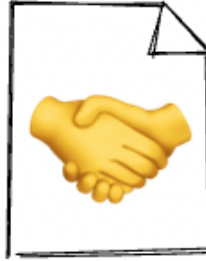
Python Packaging - CITATION.cff



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

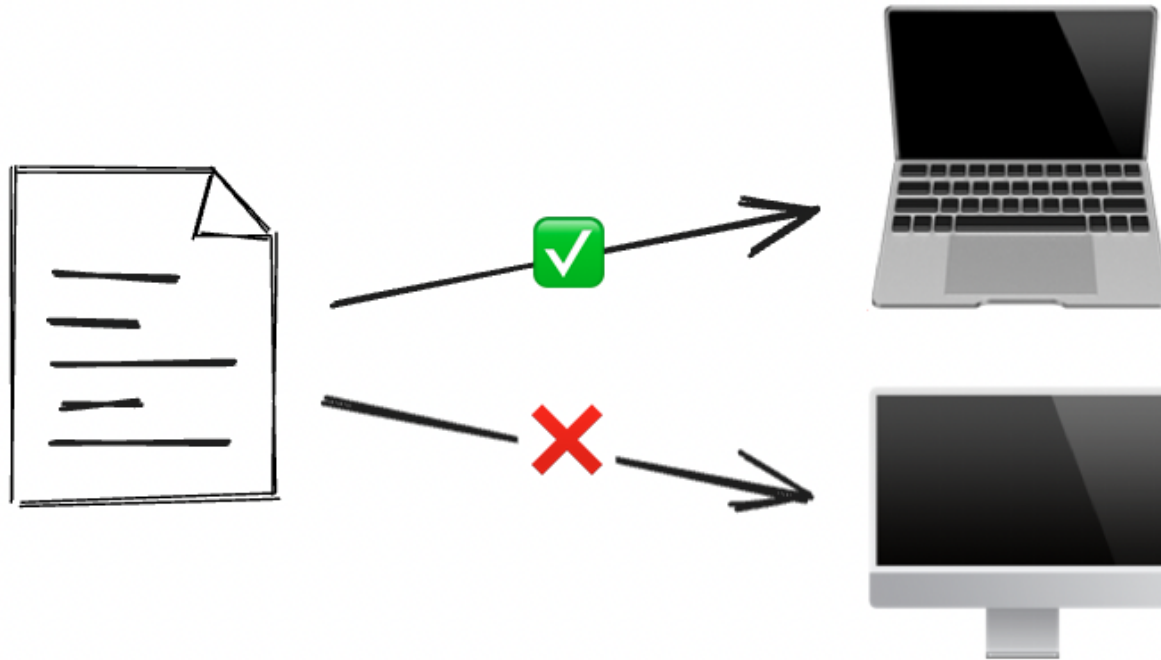


CONTRIBUTING.md

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.

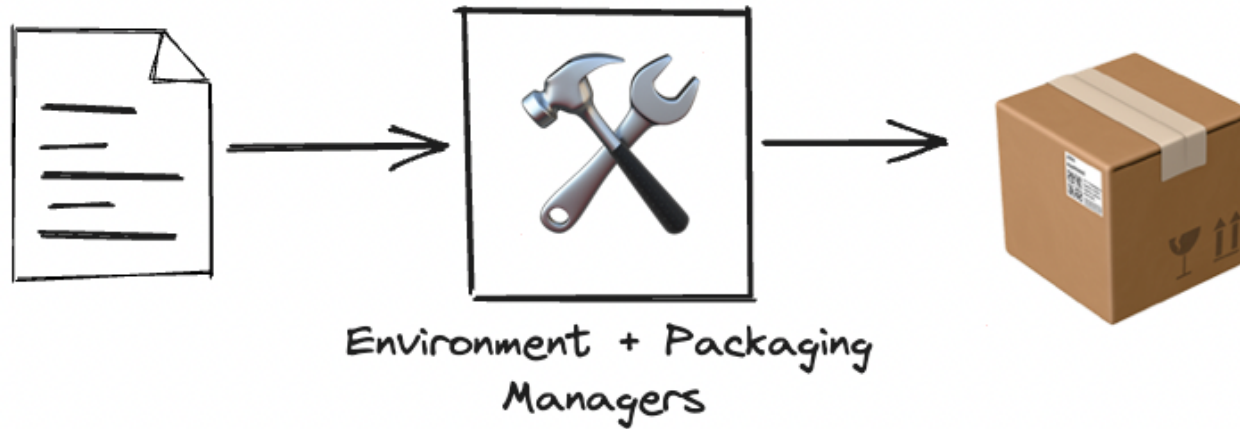
- See GitHub's documentation on "[Setting guidelines for repository contributors](#)"
- See opensource.guide's section on "[Writing your contributing guidelines](#)"

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

```
1 user@machine % poetry new --name=package_name --src .
2 Created package package_name in .
3
4 user@machine % tree .
5 .
6 |— README.md
7 |— pyproject.toml
8 |— src
9 |   |— package_name
10 |     |— __init__.py
11 |— tests
12 |   |— __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

```
1 # pyproject.toml
2 [tool.poetry]
3 name = "package-name"
4 version = "0.1.0"
5 description = ""
6 authors = ["username <email@address>"]
7 readme = "README.md"
8 packages = [{include = "package_name", from = "src"}]
9
10 [tool.poetry.dependencies]
11 python = "^3.9"
12
13 [build-system]
14 requires = ["poetry-core"]
15 build-backend = "poetry.core.masonry.api"
```

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

Python Packaging - Poetry

```
1 user@machine % poetry add pandas
2
3 Creating virtualenv package-name-1STl06GY-py3.9 in /pypoetry/virtualenvs
4 Using version ^2.1.0 for pandas
5
6 ...
7
8 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

```
1 % poetry run python -c "import pandas; print(pandas.__version__)"  
2  
3 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

```
1 % poetry build
2
3 Building package-name (0.1.0)
4   - Building sdist
5   - Built package_name-0.1.0.tar.gz
6   - Building wheel
7   - 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

```
1 % 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).

Thank you

Thank you! Questions / comments?