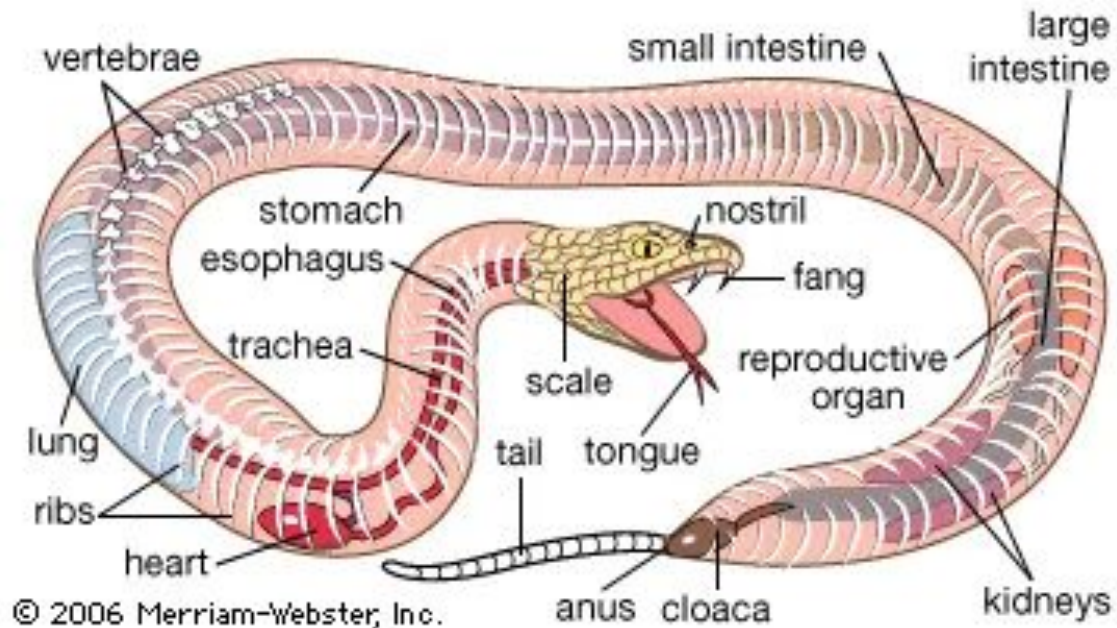

Endgame Essentials

Olivia Lynn

LSST-DA Data Science Fellowship Program Session 21

June 3, 2024

Anatomy of a Python



Anatomy of a Python Package

`project_name/`

`pyproject.toml`

`LICENSE`

`src/`

`package_name/`

`.github/workflows/`

`tests/`

`docs/`

project_name/

 **pyproject.toml**

LICENSE

src/

package_name/

.github/workflows/

tests/

docs/

pyproject.toml

pyproject.toml is a configuration file for managing Python project builds.

Key Sections:

[project] Contains project metadata, such as name, version, authors, and dependencies.

[build-system] Specifies the build backend and dependencies needed to build the project.

[tool] Configuration for various tools like linters, formatters, and test frameworks.

```
[project]
name = "pz-rail-base"
requires-python = ">=3.9"
license = {file = "LICENSE"}
readme = "README.md"
authors = [
    { name = "LSST Dark Energy Science Collabora"
]
classifiers = [
    "Development Status :: 4 - Beta",
    "License :: OSI Approved :: MIT License",
    "Intended Audience :: Developers",
    "Intended Audience :: Science/Research",
    "Operating System :: OS Independent",
    "Programming Language :: Python",
]
```

```
[build-system]
requires = [
    "setuptools>=62", # Used to build an
    "setuptools_scm>=6.2", # Gets releas
]
build-backend = "setuptools.build_meta"
```

```
[tool.pylint]
disable = [
    "abstract-method",
    "invalid-name",
```

pyproject.toml vs setup.py

- **setup.py** script used to be the standard way to set up Python packages.
- However, it's an executable file that you need to run in order to identify the dependencies that you'll need to install in order to execute this executable file that you need to run in order to identify the dependencies that you'll need to install in order to execute this executable file that you need to run in order to identify the dependencies...
- General switch towards pyproject.toml
- Special cases (legacy code and complex/custom builds–eg, with C++) do require keeping the setup.py (or having both)

project_name/

pyproject.toml

 **LICENSE**

src/

package_name/

.github/workflows/

tests/

docs/

Licenses

What are Licenses?

- Legal documents that grant permissions for using, modifying, and distributing software.

Why Do We Use Them?

- To define how code can be used by others.
- To protect the rights of the creator.
- To foster collaboration and sharing within the community.

How Do They Work?

- They specify the terms and conditions under which the software can be used, modified, and shared.
- Different licenses have different levels of permissions and restrictions.

Licenses, Pt. 2

Most Common Licenses:

- MIT License: Permissive, minimal restrictions on reuse.
- GPL (General Public License): Copyleft, requires derivatives to also be open source.
- Apache License 2.0: Permissive, with explicit grant of patent rights.
- BSD License: Permissive, similar to MIT but with a few more restrictions.

What Happens If You Don't Have One?

- Using Your Own Code: You retain all rights, but it's not clear what others can do.
- Others Using Your Code: Without a license, others technically don't have permission to use, modify, or distribute your code.

choose
alicense.com

Choose an open source license

An open source license protects contributors and users. Businesses and savvy developers won't touch a project without this protection.

{ Which of the following best describes your situation? }



I need to work in a community.

Use the [license preferred by the community](#) you're contributing to or depending on. Your project will fit right in.

If you have a dependency that doesn't have a license, ask its maintainers to [add a license](#).



I want it simple and permissive.

The [MIT License](#) is short and to the point. It lets people do almost anything they want with your project, like making and distributing closed source versions.

[Babel](#), [.NET](#), and [Rails](#) use the MIT License.



I care about sharing improvements.

The [GNU GPLv3](#) also lets people do almost anything they want with your project, *except* distributing closed source versions.

[Ansible](#), [Bash](#), and [GIMP](#) use the GNU GPLv3.

{ What if none of these work for me? }

My project isn't software.

I want more choices.

I don't want to choose a license.

project_name/

pyproject.toml

LICENSE

 **src/**

package_name/

.github/workflows/

tests/

docs/

Code Style

What's the point of consistent style?

- **Readability:** Consistent style makes code easier to read and understand.
- **Visual Cues:** Using conventions like PascalCase for class names helps you instantly recognize classes.
- **Collaboration:** With the increasing use of linters, maintaining consistent style is crucial for teamwork. Linters enforce style rules, making it difficult to ignore style guidelines.
- **Build Good Habits:** Start practicing consistent style early to make it a natural part of your coding routine.

Style: Linters

* A “code smell” is when something isn’t exactly wrong, but could be an indicator that your code is inefficient or could be refactored to be cleaner.

What are linters, and what do they do?

- Perform static program checking (analyze code without running it)
- Check for errors, coding standard violations, and code smells*
- Help reviewers focus on the code’s logic, not style issues

Linters

- pylint: Slow, thorough, highly customizable.
- ruff: Fast, good performance, customizable, can also auto-format.

Style: Autoformatters

Auto-formatters:

- Automatically changes source files to adhere to a consistent style.
- Do not interpret code for errors or code smells.
- **Tools:**
 - **black:** General code formatting.
 - **isort:** Sorts and organizes import statements.
 - **ruff format:** General code formatting.

Style: That's ruff, buddy

```
> pip install ruff
```

```
> ruff check .
```

```
> ruff format .
```

Ignore specific rules by adding them to your `pyproject.toml`:

```
[tool.ruff]
```

```
select = ["E", "F", "W"]
```

```
ignore = ["E501", "W503"]
```

Or use inline comments to ignore specific lines:

```
print("This is a long line") # noqa: E501
```



Style: PEP 8

Indentation: Use 4 spaces per indentation level.

Blank Lines:

- Use blank lines to separate top-level function and class definitions.
- Use blank lines to separate methods within a class.
- Use two blank lines before and after function definitions.

Line Length: Limit all lines to a maximum of 79 characters.

If you get really pedantic in the comments of a GitHub issue, you might meet one of your heroes (who will tell you to stop).



Line Length: Limit all lines to a maximum of 79 characters.

Style: PEP 8, Pt. 2

Imports:

- Import all modules at the top of the file.
- Group imports into three categories, in the following order:
 - Standard library imports.
 - Related third-party imports.
 - Local application/library-specific imports.
- Use absolute imports where possible.
- Avoid using wildcard imports (from module import *).

*isort is really handy
for all this!*

String Quotes:

Use single quotes (') or double quotes (") consistently throughout your code.

Style: PEP 8, Pt. 3

Whitespace: Avoid extraneous whitespace in the following situations:

Immediately inside parentheses, brackets, or braces

```
# Correct:
spam(ham[1], {eggs: 2})

# Incorrect:
spam( ham[ 1 ], { eggs: 2 } )
```

Immediately before a comma, semicolon, or colon

```
# Correct:
if x == 4: print(x, y)

# Incorrect:
if x == 4 : print(x , y)
```

More than one space around an assignment (or other) operator to align it with another

```
# Correct:
x = 1
y = 2

# Incorrect:
x  = 1
y  = 2
```

Style: PEP8, Pt. 4

Comments:

- Use comments to explain why something is done, not what is done.
- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code.
- Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement.

Naming conventions:

- Function and variable names should be in snake_case.
- Class names should be in PascalCase.
- Constants should be ALL_UPPERCASE (with underscores for spaces).

Style: PEP 8 Guidelines vs. Scientific Coding Needs

Line Length:

- PEP 8 recommends limiting lines to 79 characters.
- Exception for Equations:
 - Long equations should remain intact for readability.
 - Breaking equations can make them harder to understand.

Imports at top:

- “Import all modules at the top of the file” is logical for scripts, less so for notebooks. Practical to maintain context within a notebook.

Style: PEP 8 vs. Science, Pt. 2

Descriptive Variable Names: PEP 8 emphasizes meaningful, descriptive names.

Scientific Exceptions: Astronomical Conventions:

- ra (right ascension) and dec (declination) are clear to astronomers and can be exceptions.
- Filters like u, g, r, i, z, y would be acceptable as well.

Context Matters: Just because z is fine amidst a bunch of filters → is z fine to denote redshift? Is this clear and expected within the code?

Style: Other exceptions

While it is ideal to follow PEP 8 guidelines, the most important thing is to stay consistent.

This means you should **seek to maintain the same style choices** in both your own contributions to a codebase, as well as maintaining the style choices that had already been made to whatever codebase you're joining.

project_name/

pyproject.toml

LICENSE

src/

package_name/

 **.github/workflows/**

tests/

docs/

GitHub Actions

What are GitHub Actions?

- An automation tool provided by GitHub
- Automate workflows (perform a set of tasks defined in a given file)

How They're Coded:

- GitHub Actions are defined in YAML files
- Usually located in the `.github/workflows/` directory of your repository

```
name: CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run tests
        run: pytest
```

GitHub Actions: Common Uses

Continuous Integration (CI): Automatically build and test your code every time you push changes.

Continuous Deployment (CD): Deploy your code to production automatically after tests pass.

Code Linting and Formatting: Ensure code quality by automatically running linters and formatters.

Issue and PR Management: Automate tasks like labeling, commenting, and closing issues or PRs.

project_name/

pyproject.toml

LICENSE

src/

package_name/

.github/

workflows/

pull_request_template.md, etc



tests/

docs/

Issue/PR Templates

Predefined forms that users fill out when opening a new issue or PR

Add template: select ▼

Bug report

Standard bug report template

Feature request

Standard feature request template

Custom template

Blank template for other issue types

Issue: Bug Report

File a bug report. If this doesn't look right, [choose a different type](#).



[Bug]:

Thanks for taking the time to fill out this bug report!

Contact Details

How can we get in touch with you if we need more info?

ex. email@example.com

What happened?

Also tell us, what did you expect to happen?

A bug happened!

Version

What version of our software are you running?

None

What browsers are you seeing the problem on?

Selections: ▼

Issue/PR templates

Benefits:

- Ensures users provide necessary details to reported problems
- Automatically applies relevant tags to issues (if configured)
- Reminders for best practices:
 - Clearing notebook outputs
 - Linking issue number in the PR description
 - Writing detailed descriptions of the problem/solution

project_name/

pyproject.toml

LICENSE

src/

package_name/

.github/workflows/

 **tests/**

docs/

Unit Tests: What are they?

- Unit tests are automated tests that verify the functionality of a specific section of code (typically individual functions or methods)
- They isolate the smallest testable parts of an application to ensure they work as intended

Purpose:

- Ensure that each unit of the codebase performs as expected
- Help catch bugs early in the development process
- Facilitate refactoring by providing a safety net

Unit Tests: How do I use them?

pytest is a popular testing framework for Python that makes it easy to write simple and scalable test cases. (Other tools include unittest (built-in) and nose).

Setting up pytest:

```
> pip install pytest
```

Organizing Your Tests:

- Put test files in the tests/ directory
- Prefix test files with test_
- Prefix test functions with test_

Running all tests:

```
> pytest
```

Running tests in a specific file:

```
> pytest tests/test_my_module.py
```

Run a specific test function:

```
> pytest -k test_example
```

Other flags:

```
-v (verbose)
```

```
-s (show print statements)
```


Unit Tests: GitHub Actions

Automate Your Tests:

Use GitHub Actions to run your unit tests automatically on every push or pull request.

Further Details:

<https://docs.github.com/en/actions/examples/using-scripts-to-test-your-code-on-a-runner>

```
name: CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run tests
        run: pytest
```

Unit Tests: How do I write them?

```
def test_addition():  
    assert add(2, 3) == 5  
  
def test_divide_by_zero():  
    with pytest.raises(ZeroDivisionError):  
        divide(1, 0)  
  
def test_addition_edge():  
    assert add(-1, 1) == 0  
    assert add(1e10, 1e10) == 2e10
```

Unit Tests: How do I write them? Pt. 2

Using conftest.py:

- Centralize fixtures and hooks used across multiple test files.
- Define reusable components like setup and teardown actions.
- Gets stored in your tests/ directory.

```
import pytest

@pytest.fixture
def sample_data():
    return {"key": "value"}
```

a conftest.py with a fixture

Unit Tests: How do I write them? Pt. 3

Common Pytest Decorators:

- **@pytest.mark.parametrize:** Run a test with multiple sets of parameters.
- **@pytest.fixture:** Create reusable test data or setup.

```
@pytest.mark.parametrize("a, b, expected", [(1, 2, 3), (-1, -1, -2), (0, 0, 0)])
def test_add(a, b, expected):
    assert add(a, b) == expected

@pytest.fixture
def sample_data():
    return {"key": "value"}

def test_with_fixture(sample_data):
    assert sample_data["key"] == "value"
```

Unit Tests: How do I write them? Pt. 3.5

What is Pytest-Timeout?

- A pytest plugin that allows you to set time limits on test execution
- Helps detect and handle tests that take too long to complete

Install:

> pip install pytest-timeout

Basic Usage:

> pytest --timeout=10 # using a global timeout of 10 seconds for each test

Setting timeout per test:

```
@pytest.mark.timeout(5)
def test_with_timeout():
    import time
    time.sleep(6) # Fail
```

Unit Tests: How do I write them? Pt. 4

Pytest Hooks:

- Special functions that allow you to customize the behavior of pytest
- Enable you to extend/modify the test run at different stages of the testing process
- Hooks are defined in a conftest.py file or in a pytest plugin.
- Perform actions like modifying test collection, customizing test reports, or adding new command-line options.

```
def pytest_configure(config):  
    config.option.some_custom_option = True  
  
def pytest_runtest_setup(item):  
    print(f"Setting up for test: {item.name}")  
  
def pytest_runtest_teardown(item, nextitem):  
    print(f"Tearing down after test: {item.name}")
```

Test-driven development (TDD)

What is TDD?

- A software development approach where you write tests before writing the actual code
- Focuses on defining the desired behavior of the code upfront

The TDD Cycle:

1. Write a Test:
 - a. Write a test for a new feature/ functionality.
 - b. Test should fail initially (code isn't written yet)
2. Write the Code:
 - a. Write the minimal amount of code necessary to make the test pass.
3. Refactor:
 - a. Refactor the code to improve its structure and readability without changing its behavior.
 - b. Ensure all tests still pass after refactoring.

What if I need data in my tests?

Storing Toy Data:

- Store small, static datasets in a tests/data/ directory.
- Reference the data in your tests using conftest.py for easy access.

Large or Dynamic Data:

- If you need more data or data that must be downloaded, consider if it's suitable for a unit test.
- Perhaps, use smoke tests or integration tests for data that requires downloading or is large.
- Consider caching artifacts in CI/CD pipelines like GitHub Actions.

```
# conftest.py
@pytest.fixture
def sample_data_path():
    return os.path.join(
        os.path.dirname(__file__),
        'data',
        'sample_data.csv'
    )

# test.py
def test_with_data(sample_data_path):
    with open(sample_data_path, 'r') as f:
        data = f.read()
        assert data is not None
```


Mocking test data

Why Use Mock Data?

- Using mock data can simplify and speed up your tests by avoiding the need for real data
- Frees you from dependencies on external systems or large datasets
- Can mock functions, methods, objects, classes, side effects

How to Mock Data?

- Use the unittest.mock library
- Can use decorator or context manager

```
from unittest.mock import patch

@patch('lephare.some_module.some_function')
def test_with_mocked_function_decorator(mock_function):
    mock_function.return_value = 'mocked result'
    result = some_function()
    assert result == 'mocked result'

def test_with_mocked_function_context():
    with patch('lephare.some_module.some_function') as mock_function:
        mock_function.return_value = 'mocked result'
        result = some_function()
        assert result == 'mocked result'
```

How do I know I wrote enough tests?

“Testing can only show the presence of faults. It cannot determine their absence.”

–Dijkstra

Codecov can help

What is Codecov?

- Codecov is a tool that provides code coverage reports to help you understand which parts of your codebase are tested and which are not.
- It integrates with CI/CD pipelines to automatically generate and update coverage reports.

How It Works:

Setup:

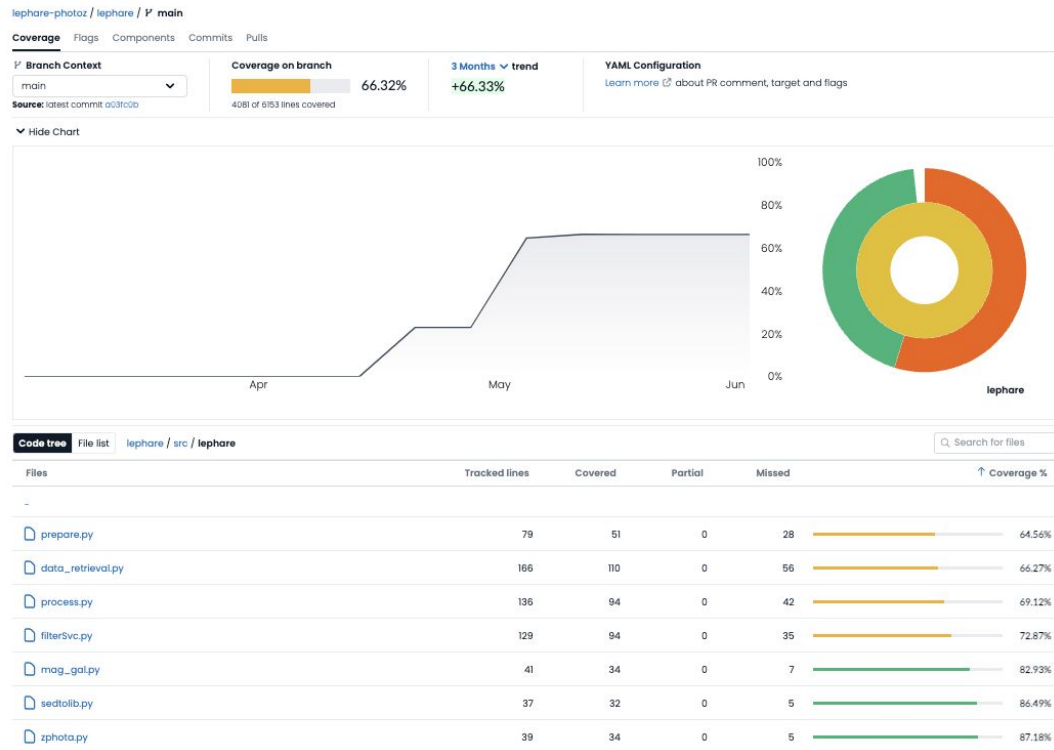
- Add Codecov to your project and configure it in your CI/CD pipeline.
- Run your tests to generate coverage data.
- Codecov collects this data and generates a coverage report.

Coverage Report:

- Shows the percentage of code covered by tests.
- Highlights uncovered lines and files.

Codecov, Pt. 2

- <https://about.codecov.io/>
- (Free for open source)



codecov bot commented last month • edited

Codecov Report

All modified and coverable lines are covered by tests ✓

Project coverage is 63.48%. Comparing base (afe58d2) to head

► Additional details and impacted files

[View full report in Codecov by Sentry.](#)

Have feedback on the report? [Share it here.](#)

project_name/

pyproject.toml

LICENSE

src/

package_name/

.github/workflows/

tests/

 **docs/**

Documentation

Documentation is crucial for understanding, using, and maintaining a codebase.

It comes in various forms, each serving different purposes.

Demos:

- Provide practical examples and use cases.
- Great for showing how to use specific features or functions.

Official Docs:

- Comprehensive and formal documentation.
- Includes detailed explanations, usage guidelines, and API references.

Docstrings:

- Embedded in the code.
- Describe the purpose, usage, and behavior of functions, classes, and modules.

ReadTheDocs and Sphinx


ReadTheDocs: <https://docs.readthedocs.io/en/stable/>

- A platform that hosts documentation for your projects
- Automatically builds and updates it as you push changes to your code

Sphinx:

- A documentation generator that converts reStructuredText (reST) files into various output formats, such as HTML and PDF, making it easy to create well-structured and readable documentation

ReadTheDocs and Sphinx, Pt. 2

 RAIL

latest

Search docs

GETTING STARTED

[Overview](#)

[Installation](#)

[Citing RAIL](#)

CONTRIBUTING

[Contribution Overview](#)

[Fix an Issue](#)

[Adding a new Rail Stage](#)

[Adding a new algorithm](#)

[Sharing a Rail Pipeline](#)

API

[API Documentation](#)

USAGE DEMOS

[Overview of Demonstrations](#)

[Core Notebooks](#)

[Creation Notebooks](#)

[Estimation Notebooks](#)

 / RAIL: Redshift Assessment Infrastructure Layers

RAIL: Redshift Assessment Infrastru

RAIL is a flexible open-source software library providing tools to pi redshift data products, including uncertainties and summary statist realistically complex systematics.

RAIL serves as the infrastructure supporting many extragalactic ap of Space and Time (LSST) on the Vera C. Rubin Observatory, includ activities. RAIL was initiated by the Photometric Redshifts (PZ) Wc Dark Energy Science Collaboration (DESC) as a result of the lesson Challenge 1 (DC1) experiment to enable the PZ WG Deliverables i Roadmap (see Sec. 5.18), aiming to guide the selection and impleme DESC analysis pipelines.

RAIL is developed and maintained by a diverse team comprising DI international in-kind contributors, LSST Interdisciplinary Collabora Frameworks software engineers, and other volunteers, but all are v regardless of LSST data rights. To get involved, chime in on the iss described in the Overview section.

Getting Started

- [Overview](#)

```
=====
RAIL: Redshift Assessment Infrastructure Layers
=====

RAIL is a flexible open-source software library providing tools

RAIL serves as the infrastructure supporting many extragalactic
RAIL was initiated by the Photometric Redshifts (PZ) Working G

RAIL is developed and maintained by a diverse team comprising
To get involved, chime in on the issues in any of the RAIL rep

.. toctree::
   :maxdepth: 1
   :caption: Getting Started

   source/overview
   source/installation
   source/citing

.. toctree::
   :maxdepth: 1
   :caption: Contributing

   source/contributing
   source/fix-an-issue
```


Docstrings

What Are Docstrings?

- String literals used to document modules, classes, functions, and methods directly in the code.
- Describe the purpose, usage, and behavior of the code elements they document.

Different Styles; we'll focus on NumPy Style:

- Widely used in scientific and data analysis communities.
- Provides a clear and structured format.
- Other Styles: Google Style, reStructuredText (reST) style.

Docstrings, Pt. 2

See official documentation:

<https://numpydoc.readthedocs.io/en/latest/format.html>

```
def add(a, b):  
    """  
    Add two numbers.  
  
    Parameters  
    -----  
    a : int or float  
        First number to add.  
    b : int or float  
        Second number to add.  
  
    Returns  
    -----  
    int or float  
        The sum of `a` and `b`.  
    """  
    return a + b
```

Docstrings: Why they're useful

```
1 def foo(x):
2     """A simple function that
3     return x
✓ 0.0s
```

```
1 foo??
✓ 0.0s
```

Signature: foo(x)
Source:
def foo(x):
 """A simple function that return
 return x
File: /var/folders/jz/hxf8drjs3
Type: function

Human Readability:

- Enhances the readability of the code by providing inline documentation.

IDE and Editor Support:

- Allows for hover-over tooltips in IDEs and code editors, showing the docstring content.

Interactive Notebooks:

- Use a ? after a method to see its docstring.
- Use ?? to see both the docstring and the source code.

ReadTheDocs Autodoc:

- Integrates with Sphinx to automatically generate documentation from docstrings.

Docstrings: Autodoc

API Documentation

Information on specific functions, classes, and methods.

- [creation namespace](#)
 - [Subpackages](#)
 - [creation.degraders namespace](#)
 - [Subpackages](#)
 - [Submodules](#)
 - [rail.creation.degraders.addRandom module](#)
 - `AddColumnOfRandom`
 - `AddColumnOfRandom.config_options`
 - `AddColumnOfRandom.name`
- [rail.creation.degraders.grid_selection module](#)
 - `GridSelection`

rail.creation.degraders.addRandom module

Add a column of random numbers to a dataframe.

```
class rail.creation.degraders.addRandom.AddColumnOfRandom(args, comm=None) \[source\]
```

Bases: `Noisifier`

Add a column of random numbers to a dataframe

Configuration Parameters: output_mode [str]: What to do with the outputs (default=default)
seed [type not specified]: Set to an *int* to force reproducible results. (default=None) col_name [str]: Name of the column with random numbers (default=chaos_bunny)

config_options

```
= {'col_name': <cec.config.StageParameter object>, 'output_mode': <cec.config.StageParameter object>,  
'seed': <cec.config.StageParameter object>}
```

```
name= 'AddColumnOfRandom'
```

Diagrams

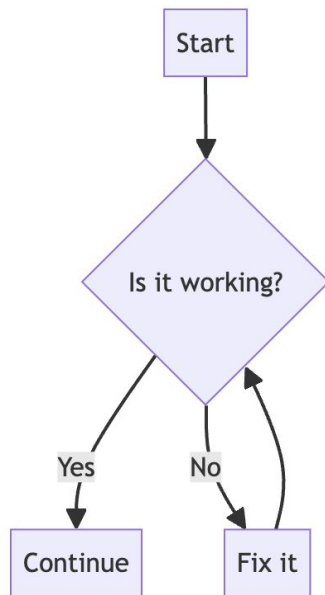
Including Diagrams in Documentation:

- You can use Mermaid to generate UML diagrams directly in your documentation.

What is Mermaid?

- Mermaid is a JavaScript-based diagramming and charting tool.
- It allows you to create diagrams and visualizations using a simple, markdown-like syntax.

```
graph TD
  A[Start] --> B{Is it working?}
  B -->|Yes| C[Continue]
  B -->|No| D[Fix it]
  D --> B
```



project_name/

pyproject.toml

LICENSE

src/

package_name/

.github/workflows/

tests/

docs/

LINCC Frameworks Python Project Template

<https://github.com/lincc-frameworks/python-project-template>

This project template codifies LINCC-Framework's best practices for python code organization, testing, documentation, and automation.

It is meant to help new python projects get started quickly, letting the user focus on writing code.

The template takes care of the minutiae of directory structures, tool configurations, and automated testing until the user is ready to take over.

PPT: All Features

Just list them all here, for reference

Maybe circle the ones that we'll jump into in the next few slides

PPT: CI Benchmarking

- So, discussing benchmarking in general. This actually sounds very useful! Fuck, I wish I knew more. Let's see what I can piece together from the docs:

- What is benchmarking

- What are major tools

- PPT uses ASV

- How does it use ASV? what does this produce?

- show an example: maybe the demo from the docs[:](<https://lincc-ppt.readthedocs.io/en/latest/>)

- <https://github.com/lincc-frameworks/benchmarking-asv>

PPT: CI Pre-commit

- Maybe add some of the verbiage from here:

<https://lincc-ppt.readthedocs.io/en/latest/practices/precommit.html>

- and reference that page for the list of our hooks
- Continuous integration Pre-Commit is the practice of running a set of hooks that enforce code styling consistency whenever new changes are proposed.
- General overview of how it works
 - Check this, but: you run git commit, and the pre-commit hook is like, hold on! lemme check some stuff. then it runs the checks, and it's either like aight, we good, or it's like, try again later s8r. Some hooks, like trivial linter errors, will go ahead and modify the file for you and (I believe) commit it (but print a

PPT: CI Pre-commit, Pt. 2

- As examples, the hooks we use are:
 - [] list here
- This is convenient for these reasons:
 - [] list
- But note, you can run into problems when you set up hooks that say, check all your notebooks can run without crashing, but you have a 30 minute ML training that happens in one. This blocks you from making the commit until it's trained! So, be aware. Street smarts! (I believe you can set them up to work on other conditions—like a hook only for PRs? or, I guess this is more of a github action that would be run).

PPT: Code coverage

This is also a good thing to cover post-unit test descriptions. Maybe I should have already covered this as a general topic? Probably.

Here, i should just briefly outline how PPT supports this

PPT: Git LFS

- General overview of what it is, how it works, why we use it (mention maybe that I've seen it in game dev and XR, tho tools like perforce (should probably have the definition handy—centralized version control, vs git's decentralized? TODO look this up) are widely popular)
- Anyway. Can be costly at their caps (100GB upload/download per month for free, iirc?) so there are alternatives to consider:
 - Zenodo pros and cos
 - OSF pros and cons
 - Mention how RAIL stores some data on a shared drive at NERSC
 - Lephare stores some data in a sep github repo (following precedent by sk-image, I believe, but TODO check this)

PPT: PyPI publishing

- what is this: (reword: [PyPI](https://pypi.org/) is a repository for packaged python software. When published via PyPI, a package can be installed on anyone's development environment with `pip` or `conda` commands. PyPI makes versioning and sharing your software products easy.)

- you can also share packages software with conda, though that's a little different. namely, conda can do non-python packages, but needs to be installed through conda, which can have really long wait times for dependency resolution esp in complex packages with many dependencies. this can be slightly sped up with mamba, which accesses conda channels but has a far more optimized resolver.

- TODO: look up any other key differences

PPT: PyPI publishing, Pt. 2

- what does our support for PyPI look like? glad you asked:
 - explain tagged releases, which maybe needs a moment in overviewing the main kind of writing release versions
 - semantic versioning explanation
 - then (and look up the details at play in PPT), if you release a tag that clears certain requirements, a GitHub Action will automatically publish that new version of the package to PyPI, so any users who run `pip your-package` (TODO are hyphens ok here?) will get that latest version
 - Maybe: a cool thing about making a new release is it'll automatically bundle all the PRs between the last release and now into a changelist for you