Your project structure for Civil\_Engineering\_Tools is quite well-organized and follows many Python best practices! Here's a breakdown of what looks good and a few minor suggestions for potential enhancements:

Strengths of your current structure:

- Root Directory: Contains essential files like .gitignore , LICENSE , README.md , pyproject.toml , and requirements.txt . This is standard and good.

- Source Code ( src/ layout): Placing your main package mat\_ceng under a src/ directory (i.e., `src` ) is a recommended practice. It helps avoid import issues and clearly separates your installable code from other project files.

- Tests ( tests/ ): Having a dedicated `tests` directory at the root for your tests is standard. The naming convention test\_\*.py (e.g., `test\_mat\_ceng.py` ) is also correct for test discovery by tools like pytest.

- Notebooks ( notebooks/ ): A `notebooks` directory is a common and sensible place for Jupyter notebooks used for experimentation, analysis, or examples. The sub-categorization within notebooks (e.g., `Calculating trib regions` , `modeling\_from\_revit` ) is excellent for organization.

- Streamlit Application ( streamlit\_app/ ): Separating your Streamlit application into its own directory ( `streamlit\_app` ) is clean. The use of a pages/ subdirectory within `ceng\_app` is the standard way to create multi-page Streamlit apps.

- Dependency Management: You have both `pyproject.toml` and `requirements.txt` , along with `uv.lock` . This indicates you're likely using modern Python packaging tools.

Potential suggestions (consider these if they fit your workflow):

1. Dependency Files Clarification:

- If your pyproject.toml (e.g., using Poetry, PDM, or Hatch) defines your project dependencies, requirements.txt might be redundant or could be generated from pyproject.toml for specific purposes (e.g., pip freeze > requirements.txt for a snapshot, or poetry export -f requirements.txt --output requirements.txt ).

- If uv is your primary package manager, it typically works with pyproject.toml or requirements.txt as input and uv.lock as the lock file. Ensure your dependency sources are consistent.

2. Module Naming in src/mat\_ceng/ :

- You have a file `mat\_ceng.py` inside the `mat\_ceng` package. While not strictly wrong, sometimes this can be slightly confusing (e.g., from mat\_ceng import mat\_ceng ).

- Consider if the contents of `mat\_ceng.py` could be moved into `\_\_init\_\_.py` if it's meant to expose the primary API of the mat\_ceng package (e.g., so users can do from mat\_ceng import some\_function ). Alternatively, if it's a specific submodule, its current naming is fine, or it could be named more descriptively if it serves a particular purpose within the mat\_ceng tools.

3. Documentation ( docs/ ):

- For more extensive documentation beyond the README.md , consider adding a docs/ directory. You could use tools like Sphinx to generate HTML documentation from reStructuredText or Markdown files, including auto-generating API docs from your Python docstrings.

4. Examples ( examples/ ):

- If you have standalone Python scripts that demonstrate how to use your mat\_ceng library (distinct from the more exploratory nature of notebooks), an examples/ directory at the root could be beneficial.

5. Data Files ( data/ ):

- You currently have data files like `floor\_data.json` within specific notebook directories, which is perfectly fine for data tied to a particular notebook.

- If you have more general data files that might be used by the library itself or multiple examples/notebooks, you could consider a top-level data/ directory. Remember to add this to .gitignore if the data files are large or shouldn't be version-controlled.

6. Scripts/CLI Tools:

- If your project includes command-line interface (CLI) tools, you might organize them in a scripts/ or cli/ directory. Alternatively, you can define entry points in your pyproject.toml which allows tools built with libraries like click or typer to be installed as commands.

Overall, your project is already in good shape. These suggestions are minor and depend on the scale and future direction of your project. Keep up the great work!

That's a great foresight to plan for organizing your submodules within `mat\_ceng` as your project grows! The most common and effective way to do this in Python is by creating sub-packages .

Essentially, you'll create new directories inside `mat\_ceng` for different categories of your civil engineering tools. Each of these new directories will also need an \_\_init\_\_.py file to make them recognizable as Python packages (or sub-packages).

Here's a conceptual example of how you could structure it:

Let's say your civil engineering tools can be broadly categorized into areas like 'structural analysis', 'geotechnical', 'hydraulics', and 'surveying'. You could structure `mat\_ceng` like this:

src/

└── mat\_ceng/

├── \_\_init\_\_.py # Main \_\_init\_\_.py for mat\_ceng

|

├── structural/

│ ├── \_\_init\_\_.py

│ ├── beam\_analysis.py

│ ├── column\_design.py # Your existing column.py might go here or be split

│ ├── slab\_design.py

│ └── ... (other structural modules)

|

├── geotechnical/

│ ├── \_\_init\_\_.py

│ ├── soil\_mechanics.py

│ ├── foundation\_design.py

│ └── ... (other geotechnical modules)

|

├── hydraulics/

│ ├── \_\_init\_\_.py

│ ├── open\_channel\_flow.py

│ ├── pipe\_networks.py

│ └── ... (other hydraulics modules)

|

├── surveying/

│ ├── \_\_init\_\_.py

│ ├── coordinate\_geometry.py

│ ├── leveling.py

│ └── ... (other surveying modules)

|

├── csi\_interop/ # For CSi-related tools like your csi.py

│ ├── \_\_init\_\_.py

│ └── etabs\_api.py # Potentially refactored from csi.py

│ └── sap2000\_api.py

|

└── utils/ # For common utility functions used across submodules

├── \_\_init\_\_.py

└── geometry\_helpers.py # e.g., column\_area.py might fit here or in structural

# Existing files like mat\_ceng.py would either be refactored into these

# submodules, or its content might be suitable for the main mat\_ceng/\_\_init\_\_.py

# if it's meant to be the primary API access point.

How imports would work:

With this structure, your imports would become more specific:

from mat\_ceng.structural import column\_design

from mat\_ceng.geotechnical import soil\_mechanics

from mat\_ceng.csi\_interop import etabs\_api

# If you define things in the \_\_init\_\_.py of a sub-package,

# you can make imports cleaner:

# e.g., in mat\_ceng/structural/\_\_init\_\_.py:

# from .column\_design import Column

# Then you could import as:

# from mat\_ceng.structural import Column

Steps to implement:

1. Identify Categories: Think about the logical groupings for your current and planned tools.

2. Create Directories: Inside `mat\_ceng` , create a new directory for each category (e.g., structural , geotechnical ).

3. Add \_\_init\_\_.py : Add an empty \_\_init\_\_.py file to each new directory to make it a sub-package.

4. Move and Refactor Modules:

- Move your existing .py files (like `column.py` , `column\_area.py` , `csi.py` ) into the appropriate new sub-package directory.

- You might need to refactor them if a single old file contained logic for multiple new sub-packages.

- Update import statements in your code and tests to reflect the new paths.

5. Organize mat\_ceng.py : Decide what to do with `mat\_ceng.py` . Its contents might be better suited to be distributed among the new submodules, or parts of it could go into the main `\_\_init\_\_.py` to provide a convenient top-level API for your library (e.g., from mat\_ceng import useful\_function ).

Benefits of this approach:

- Clarity: Makes it easier to understand what different parts of your library do.

- Maintainability: Easier to find and modify code related to a specific domain.

- Scalability: Accommodates a large number of modules without cluttering the main package directory.

- Reduced Namespace Collisions: Less chance of accidentally naming two different modules the same thing.

This approach provides a robust and scalable way to manage a growing collection of specialized tools within your mat\_ceng library.

Excellent question! Organizing your `tests` folder to mirror your src structure is a highly recommended practice. It makes it much easier to locate tests corresponding to specific modules and helps in maintaining a clean and understandable test suite, especially as your project grows.

Given the proposed structure for `src/mat\_ceng` :

src/

└── mat\_ceng/

├── \_\_init\_\_.py

├── structural/

│ ├── \_\_init\_\_.py

│ ├── beam\_analysis.py

│ └── column\_design.py

├── geotechnical/

│ ├── \_\_init\_\_.py

│ └── soil\_mechanics.py

# ... and so on for csi\_interop, surveying, utils

Your `tests` directory should ideally mirror this structure:

tests/

├── \_\_init\_\_.py # Makes 'tests' a package, useful for some test runners/plugins

|

├── mat\_ceng/ # Mirroring the main package name

│ ├── \_\_init\_\_.py

│ |

│ ├── structural/

│ │ ├── \_\_init\_\_.py

│ │ ├── test\_beam\_analysis.py

│ │ └── test\_column\_design.py

│ |

│ ├── geotechnical/

│ │ ├── \_\_init\_\_.py

│ │ └── test\_soil\_mechanics.py

│ # ... and so on for csi\_interop, surveying, utils

│

│ # You might also have tests for the top-level mat\_ceng module itself

│ └── test\_mat\_ceng\_general.py # Or tests for functionalities in mat\_ceng/\_\_init\_\_.py

|

├── conftest.py # Optional: For pytest fixtures shared across all tests

|

└── integration/ # Optional: For integration tests that span multiple modules

├── \_\_init\_\_.py

└── test\_structural\_geotechnical\_interaction.py

Key Considerations for Test Organization:

1. Mirror the Source Structure:

- For every sub-package in src/mat\_ceng (e.g., structural , geotechnical ), create a corresponding sub-directory in tests/mat\_ceng .

- For every module file in src/mat\_ceng/some\_subpackage/some\_module.py , create a corresponding test file tests/mat\_ceng/some\_subpackage/test\_some\_module.py .

- Your current files `test\_mat\_ceng.py` and `test\_mat\_ceng\_column.py` would be moved and potentially renamed. For example, `test\_mat\_ceng\_column.py` would likely become tests/mat\_ceng/structural/test\_column\_design.py (assuming column.py moves to src/mat\_ceng/structural/column\_design.py ).

2. \_\_init\_\_.py Files in Test Directories:

- Adding \_\_init\_\_.py files to your test directories (e.g., tests/\_\_init\_\_.py , tests/mat\_ceng/\_\_init\_\_.py , tests/mat\_ceng/structural/\_\_init\_\_.py ) makes them Python packages. This can be beneficial for:

- Helping test discovery for some tools or configurations.

- Allowing you to import helper functions or fixtures from other parts of your test suite using relative imports if needed.

3. Test File Naming:

- Continue using the test\_\*.py prefix for test files (e.g., test\_beam\_analysis.py ).

- Inside each test file, test methods/functions should also be prefixed with test\_ (e.g., def test\_calculate\_bending\_moment(): ). This is standard for tools like pytest and unittest to auto-discover tests.

4. Unit Tests vs. Integration Tests:

- Unit Tests: The structure described above is primarily for unit tests, where you test individual modules or functions in isolation. Each test\_\*.py file would typically focus on testing the corresponding \*.py file in your src directory.

- Integration Tests: For tests that verify interactions between multiple modules or sub-packages, you might consider a separate top-level directory within tests , such as tests/integration/ . These tests are often more complex and test broader workflows.

- For example, a test in tests/integration/test\_workflow\_A.py might import and use functions from mat\_ceng.structural and mat\_ceng.geotechnical to ensure they work together correctly.

5. conftest.py for Fixtures (pytest):

- If you're using pytest , you can place a conftest.py file in the root of your tests directory (or in subdirectories) to define fixtures, hooks, or plugins that are shared across multiple test files.

- For example, a conftest.py in tests/mat\_ceng/structural/ could define fixtures specific to testing structural components.

6. Test Data:

- If your tests require data files, you can place them alongside the relevant test files or in a dedicated test\_data subdirectory within the specific test package (e.g., tests/mat\_ceng/structural/test\_data/ ).

Benefits of This Mirrored Structure:

- Discoverability: Easy to find the tests for any given piece of source code.

- Maintainability: When you modify a source module, you know exactly where to find and update its tests.

- Clarity: The test suite's organization reflects the application's architecture.

- Scalability: Handles a growing number of tests and source modules gracefully.

By adopting this parallel structure, you'll create a more robust and maintainable testing environment for your Civil\_Engineering\_Tools project.

It's great that you've implemented the structural changes! Your project is looking much more organized. Based on the latest structure you've shared, here are a few more suggestions to further enhance your Civil\_Engineering\_Tools project:

1. Finalize Test File Locations:

- I notice `test\_mat\_ceng.py` and `test\_mat\_ceng\_column.py` are still directly under the `tests` directory.

- To fully align with the mirrored structure, you should move these:

- `test\_mat\_ceng\_column.py` would likely move to tests/mat\_ceng/structural/test\_column.py (or test\_column\_design.py if you renamed column.py to column\_design.py in src ).

- `test\_mat\_ceng.py` could be renamed to something like tests/mat\_ceng/test\_core.py if it tests general functionalities of the mat\_ceng package itself (perhaps from src/mat\_ceng/\_\_init\_\_.py or the old src/mat\_ceng/mat\_ceng.py ). If its tests are specific to submodules, then its contents should be split and moved into the respective test files within tests/mat\_ceng/structural/ , tests/mat\_ceng/geotechnical/ , etc.

2. Comprehensive Documentation:

- Docstrings: Ensure all your modules, classes, functions, and methods have clear and informative docstrings (e.g., following Google, NumPy, or reStructuredText style). This is invaluable for maintainability and for others (or your future self) to understand the code.

- docs/ Directory: For more extensive documentation, create a docs/ directory at the project root.

- Use tools like Sphinx ( Sphinx ) or MkDocs ( MkDocs ) to generate a user-friendly HTML documentation site.

- These tools can automatically pull in your docstrings to create API documentation.

- Include tutorials, usage examples, and conceptual explanations.

- Improve README.md : Make your `README.md` more comprehensive. Include:

- A clear project description and its purpose.

- Installation instructions.

- Quick start guide/basic usage examples.

- How to run tests.

- Contribution guidelines (if applicable).

- A link to the full documentation if you create one.

3. Code Quality & Consistency:

- Type Hinting: Add type hints to your Python code (Python 3.5+). This improves code readability, helps catch errors early (with tools like MyPy), and makes your code easier to understand and refactor.

def calculate\_area(length: float, width: float) -> float:

return length \* width

- Linters and Formatters: Integrate tools to enforce code style and catch potential errors:

- Ruff ( Ruff ): An extremely fast Python linter and formatter, written in Rust. It can replace Flake8, isort, and even parts of Black and others.

- Black ( Black ): An uncompromising code formatter that ensures uniform style.

- Flake8 ( Flake8 ): A popular linter that checks for PEP 8 compliance and other common issues.

- isort ( isort ): Sorts your imports automatically.

- Configure these in your pyproject.toml .

- Pre-commit Hooks: Use the pre-commit framework ( pre-commit ) to run linters, formatters, and other checks automatically before each commit. This helps maintain code quality consistently.

4. Dependency Management:

- You have `pyproject.toml` , `requirements.txt` , and `uv.lock` . Clarify your dependency management strategy:

- If pyproject.toml (with a tool like Poetry, PDM, or Hatch, or just build system and project metadata) is your primary source of truth for dependencies, ensure requirements.txt is either generated from it (e.g., poetry export > requirements.txt ) or serves a distinct, documented purpose (e.g., for a specific deployment environment).

- If uv is your main tool, it typically uses pyproject.toml or requirements.txt as input and uv.lock as the lockfile. Ensure this workflow is clear.

- Consider using dependency groups in pyproject.toml (if your build backend supports it, e.g., Poetry, PDM) for development dependencies (like pytest, linters) vs. runtime dependencies.

5. \_\_init\_\_.py and Public API:

- Carefully consider what you expose from your package's \_\_init\_\_.py files (e.g., `\_\_init\_\_.py` and the \_\_init\_\_.py in submodules like `structural` ).

- These files define the public API of your library. You can import specific classes/functions into them to make them easier to access for users of your library.

- Example in src/mat\_ceng/structural/\_\_init\_\_.py :

from .column\_design import Column # Assuming Column class in column\_design.py

from .beam\_analysis import calculate\_moment

\_\_all\_\_ = ["Column", "calculate\_moment"] # Explicitly define public API

- - This allows users to do from mat\_ceng.structural import Column instead of from mat\_ceng.structural.column\_design import Column .

6. Continuous Integration (CI):

- Set up a CI pipeline using services like GitHub Actions, GitLab CI, or Jenkins.

- Your CI pipeline should automatically:

- Run linters and formatters to check code style.

- Run your test suite (e.g., using pytest ) on every push or pull request.

- Optionally, build your documentation and check for type errors with MyPy.

- This ensures that changes don't break existing functionality and maintain code quality.

7. Examples and Notebooks:

- Your `notebooks` directory is great for exploration. Ensure these are well-commented and easy to follow.

- Your `examples` directory (which you created) is good for standalone, runnable scripts demonstrating how to use your library's features.

By systematically addressing these areas, you can significantly improve the robustness, maintainability, and usability of your Civil\_Engineering\_Tools project. Choose the suggestions that best fit your current needs and project goals!

You're on the right track with wanting an editable install for development! With uv , the process is very similar to pip , and it integrates well with pyproject.toml and uv.lock .

Here's how you can achieve an editable install and manage your dependencies with uv :

1. Editable Install with uv

The direct equivalent to pip install -e . using uv is:

uv pip install -e .

- This command tells uv to install your project (found in the current directory . ) in editable mode.

- It will typically look at your `pyproject.toml` to understand your project's metadata and its dependencies.

2. How uv Handles Dependencies and uv.lock

- Primary Source of Truth: Your `pyproject.toml` should be the primary source of truth for your project's direct dependencies. You define them in the [project.dependencies] table (for runtime dependencies) and often in [project.optional-dependencies] for development tools (like pytest , ruff , mypy ).

- uv.lock File:

- When you run uv pip install ... or uv sync ... , uv will try to use the `uv.lock` file if it exists and is consistent with your pyproject.toml (or requirements.txt if that's what you're using as the source).

- If `uv.lock` doesn't exist, or if you've changed dependencies in `pyproject.toml` , uv will resolve the dependencies and create/update `uv.lock` with the exact versions of all packages (direct and transitive).

- The uv pip install -e . command will also respect and potentially update the lock file based on the dependencies declared in your `pyproject.toml` for your project.

- uv sync for Reproducibility: For the most reproducible builds based exactly on your lock file, you'd typically use:

uv sync

- However, uv sync doesn't directly support the -e . editable install of the project itself in the same way uv pip install -e . does. uv sync is more about ensuring the environment matches the lockfile precisely.

For development, uv pip install -e . is generally what you want, as it sets up the editable install and ensures dependencies are met (updating the lockfile if needed).

3. Recommended Development Workflow with uv

Here's a typical workflow:

- Initial Setup / Cloning Repository:

1. Create and activate a virtual environment (highly recommended):

python -m venv .venv

source .venv/bin/activate # Linux/macOS

# .venv\Scripts\activate # Windows

Or, let uv manage the virtual environment (it can create one if it doesn't detect an active one, or you can use uv venv ):

uv venv # Creates a .venv if not present

source .venv/bin/activate # or .venv\Scripts\activate

1. Install your project in editable mode along with its dependencies (this will also generate/update uv.lock ):  
   uv pip install -e ".[dev]"

(Assuming you have a [project.optional-dependencies.dev] section in your `pyproject.toml` for development tools. If not, just uv pip install -e . for runtime dependencies).

Adding a New Dependency:

1. Add the new package to your pyproject.toml under [project.dependencies] or [project.optional-dependencies.dev] .

2. Then, re-run the install command. uv will detect the change, resolve the new dependency, install it, and update uv.lock .

uv pip install -e ".[dev]"

Alternatively, uv offers an add command (experimental as of early 2024 but likely to become stable):

uv pip install <new\_package\_name> # For runtime

uv pip install --group dev <new\_dev\_package\_name> # For dev (if using groups)

Or more directly, if uv add becomes the standard:

uv add <new\_package\_name>

uv add <new\_dev\_package\_name> --dev

- 1. These add commands would typically update your pyproject.toml and uv.lock automatically.

-

Updating Existing Dependencies: To update all packages to their latest allowed versions (respecting constraints in pyproject.toml ) and regenerate uv.lock :

uv pip install --upgrade -e ".[dev]"

Key Takeaways for Your Use Case:

- Use uv pip install -e . (and potentially uv pip install -e ".[dev]" ) for your development setup.

- This command will handle the editable installation of your project and ensure dependencies from `pyproject.toml` are installed, using/updating `uv.lock` as needed.

- You won't have problems with uv.lock if you let uv manage it by re-running install commands after changing pyproject.toml or by using uv add (when stable and if it fits your workflow).

This approach ensures that your development environment has your project installed editably and that your dependencies are managed consistently and reproducibly with uv .