Organising Your Code Like a Pro

A Friendly Guide to Folder Management and Project Structure for Master's Students

Hey there, master coders-to-be! Whether you're diving into a solo project or joining forces with a team, keeping your code clean and organized is one of the *best* things you can do for your sanity and future self. Let's face it: messy folders, misplaced files, and cryptic file names aren't a vibe. So let's break down how to build a repo that's functional, professional, and even a little fun to use.

Why Organization Matters

Imagine you're searching for a needle in a haystack… of spaghetti code. 🌀 Chaos isn't cute, especially when it comes to coding. Good organisation:

- Saves **time** (because you don't have to hunt for "final_final_revised2.py").
- Helps others (and future you!) understand your work.
- Makes teamwork smoother and way less stressful.
- Ensures reproducibility for your results—critical for academic projects.

Your Ultimate Repository Blueprint

Picture your repository as a well-organized backpack. Everything should have its place and purpose. Here's what your repo might look like:

```
project-name/
├── data/
├── docs/
├── src/
├── tests/
├── notebooks/
├── results/
├── scripts/
├── requirements.txt
├── README.md
├── LICENSE
└── .gitignore
```

Let's unpack what goes into each folder and file, and why it's important.

---

1. The Root Folder: Your Project's Welcome Mat

This is the first thing anyone will see when they open your repo, so make it **shine**.

📖 README.md

The **MVP** of your repo. Think of this as the "About Me" for your project. Include:

- **What this project does:** A brief description.
- **How to run it:** Installation and usage instructions.
- **Dependencies:** What tools or libraries are required.
- **Anything cool to share:** Fun features? Future plans? Add it here!

Example:

```
# CoolProject
This is a machine learning project that predicts pizza topp
ing combinations you'll love. 🍕

## Installation
```

📜 LICENSE

This is the legal stuff. Think of it as your project's permission slip.

Not sure which license to pick? The MIT license is simple and widely used.

🙅‍♂️ .gitignore

Keeps your repo clean by telling Git what *not* to track.

Common things to ignore:

- Temporary files (`.tmp`, `.DS_Store`).

- Secrets (`.env`).
- Compiled files (`.pyc`).
- Large datasets (`data/raw/`).

---

2. Folder Structure: The Magic Behind the Scenes

Folders make your project look pro *and* keep you organized. Here's the lowdown:

📂 data/

For all things *data*.

- **raw/**: Original, untouched data files.
- **processed/**: Cleaned-up, analysis-ready data.

> Pro Tip: Never edit raw data directly! Use scripts to transform it and keep things reproducible.

📂 docs/

Documentation makes your project's heart sing. Store:

- **Manuals**: How to use your project.
- **Diagrams**: Fancy visuals like flowcharts.
- **API Docs**: If you're building something interactive.

📂 src/

The brain of your project—your source code lives here.

- **Modules:** Break your code into logical chunks (e.g., `src/data_processing.py`).
- **Main entry point:** A `main.py` file to run the project.

> Pro Tip: Write clean, reusable code and document it with helpful comments. Future-you will thank you.

📂 tests/

Keep your project bug-free with tests.

Use tools like `pytest` or `unittest` to ensure everything works as expected.

Example:

- `tests/test_data_processing.py` for your data-cleaning scripts.
- `tests/test_model.py` for your ML model.

> Pro Tip: Automate your tests with CI tools like GitHub Actions. It's like having a robot buddy who catches mistakes!

📁 notebooks/

Exploration zone! This is where your Jupyter notebooks go.

- Name them clearly: `01_data_cleaning.ipynb`, `02_model_training.ipynb`.
- Use markdown to explain what you're doing.

> Pro Tip: Keep notebooks for experimentation. Turn polished code into scripts and move them to src/.

📁 results/

The fruit of your labor. 🍎

Store:

- **Figures**: Charts, plots, and visualizations.
- **Models**: Saved files from your ML models (e.g., `.pkl`, `.h5`).

📁 scripts/

Utility belt time. 🛠️

Store handy scripts that automate tasks, like:

- Data downloads: `download_data.py`.
- Preprocessing pipelines: `preprocess_data.py`.

---

3. Supporting Files: The Glue That Holds It All Together

📜 requirements.txt

A simple file listing all your dependencies.

Example:

```
numpy==1.21.0
pandas==1.3.0
matplotlib==3.4.2
```

> Pro Tip: Use pip freeze > requirements.txt to generate this automatically.

📜 setup.py *(Optional)*

If you want to turn your project into a package, this is your go-to.

Basic Example: setup.py

```
python
Copy code
from setuptools import setup, find_packages

setup(
    name="coolproject",  # Name of your package
    version="0.1.0",  # Version of your package
    description="A project that predicts pizza topping comb
inations you'll love.",  # Short description
    long_description=open("README.md").read(),  # Detailed
description (usually from README.md)
    long_description_content_type="text/markdown",  # Speci
fy the format of the long description
    author="Your Name",  # Your name (or your team’s name)
    author_email="your.email@example.com",  # Your email
    url="https://github.com/username/coolproject",  # Proje
ct's homepage (GitHub, etc.)
    packages=find_packages(where="src"),  # Automatically f
ind packages in 'src/' folder
    package_dir={"": "src"},  # Tell setuptools where to lo
ok for packages
    include_package_data=True,  # Include files specified i
n MANIFEST.in
    install_requires=[
        "numpy>=1.21.0",
        "pandas>=1.3.0",
        "matplotlib>=3.4.2"
    ],  # Dependencies needed to run your package
    python_requires=">=3.7",  # Specify the Python versions
supported
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],  # Metadata to make your project easier to find
    entry_points={
        "console_scripts": [
            "coolproject-cli=coolproject.main:main_functio
n"
        ]
    },  # Optional: Create CLI commands
)
```

What Each Part Does

1. Basic Information

- `name`: The name of your package. This is what people will use to install it (e.g., `pip install coolproject`).
- `version`: A semantic version number. Follow the format `major.minor.patch`.
- `description` and `long_description`: These provide a short and detailed explanation of your project.

2. Specifying Packages

- `packages=find_packages(where="src")`: Automatically finds all Python modules and sub-packages in the specified folder (e.g., `src/`).

- `package_dir={"": "src"}`: Tells setuptools to look for the code in the `src` folder.

3. Dependencies

- `install_requires`: A list of libraries your project depends on. Specify exact versions or ranges for compatibility.

4. Python Compatibility

- `python_requires`: Ensures users install the package only if they have a compatible Python version.

5. Metadata

- `classifiers`: A list of tags that make your package easier to find on PyPI or in search engines.

6. CLI Support (Optional)

- `entry_points`: Allows you to define custom command-line interface (CLI) tools that run your code. For example, `coolproject-cli` would run the `main_function` in `coolproject/main.py`.

To Install Locally

Once you have this `setup.py`, you can install your project locally by running:

```
pip install -e .
```

📜 environment.yml *(Optional)*

For Conda fans, this file defines your environment.

Setting Up and Using a Conda Environment with a `.yml` File

A **Conda environment** is like a mini workspace that keeps all your project dependencies and tools neatly organized and separate from other projects. Using a `.yml` file to define your environment makes it easy for you (and others!) to recreate the exact setup with a single command.

---

1. Create a Conda Environment `.yml` File

The `.yml` file defines the environment, including the Python version, dependencies, and any optional tools. Here's an example:

`environment.yml`

```
yaml
Copy code
name: coolproject  # Name of your environment
channels:
  - defaults       # Default Conda channels for finding pac
kages
dependencies:
  - python=3.9     # Python version
  - numpy=1.21.0   # Specify exact versions if needed
  - pandas=1.3.0
  - matplotlib=3.4.2
  - scikit-learn
  - jupyterlab     # Optional: For Jupyter Notebooks
  - pip            # Ensure pip is available
  - pip:
      - seaborn    # Additional dependencies installed via
pip
```

---

2. Create the Environment

To create the environment from the `.yml` file:

```
bash
Copy code
conda env create -f environment.yml
```

- This command will:
  1. Read the `environment.yml` file.
  2. Create an environment named `coolproject`.
  3. Install all the specified dependencies.

Best Practices for Happy Repos

Let’s sprinkle some magic dust on your projects with these tips:

1. Keep It Clean

- Name files descriptively (`train_model.py` > `new.py`).
- Avoid cluttering folders with random files.

2. Commit Like a Pro

- Commit often, but only meaningful changes.
- Write clear commit messages:
  ✅
  `Fix bug in data processing script`
  ❌
  `fixed stuff`.

3. Stay Consistent

- Follow Python’s PEP 8 style guide.
- Use tools like `black` to auto-format your code.

4. Document, Document, Document

- Write clear docstrings for functions.
- Keep your README.md updated as the project evolves.

Tools to Make Your Life Easier

- **Git:** For version control.
- **Virtual Environments:** Use `venv` or `conda` to avoid dependency headaches.
- **Linting:** Use `flake8` or `pylint` to catch errors before running your code.
- **Automated Testing:** Tools like `pytest` paired with CI pipelines make testing painless.

Final Thoughts

Organizing your code doesn’t just make your project look good—it helps you work faster, smarter, and with way less stress. By following these tips, you’ll create repositories that are easy to navigate, fun to work on, and impressive to collaborators.

Happy coding! 🎉