ChatGPT

Modern Python Data Science Project Setup (2025)

Project Bootstrapping and Initialization

Begin by initializing a new Python project with tools like **Poetry** or **Hatch** to create a standard structure. For example:

```
poetry new my_data_project
# Creates src/my_data_project/, pyproject.toml, tests/, etc.
```

```
hatch new "My Data Project"
# Creates a project with src/, LICENSE, README, pyproject.toml, etc [1].
```

These commands auto-generate a `src/` package directory, a `pyproject.toml`, a basic README, and test scaffolding. Use **Git** from day one:

```
git init
echo "__pycache__/" >> .gitignore
echo "venv/" >> .gitignore
git add .
git commit -m "Initial commit"
```

Always add a suitable license early. For enterprises, prefer a **permissive license** (e.g. MIT or Apache 2.0) to allow broad reuse; Apache 2.0 adds explicit patent grants that many companies require [2] [3]. Avoid copyleft (e.g. GPL) if the code may be used in proprietary projects.

Leverage project templates to automate setup. For instance, the *cookiecutter-poetry* template includes GitHub Actions CI, pre-commit hooks (Black, Ruff, MyPy, etc.), and docs with MkDocs [4]. Similarly, the **Cookiecutter Data Science** template sets up `data/`, `notebooks/`, `src/`, and `docs/` directories with best practices pre-configured [5] [6]. Such templates save time and ensure consistency.

Environment Management and Dependency Handling

Use **virtual environments** to isolate dependencies. The built-in `venv` (PEP 405) is always available:

```
python3.12 -m venv venv
source venv/bin/activate
```

However, tools like **Poetry** and **Hatch** unify env & dependency management. Poetry creates its own venv and generates a lockfile for reproducible installs [7]. For example, after `poetry install`, Poetry resolves dependencies and records exact versions in `poetry.lock` [7]. Hatch can similarly manage multiple named environments and even *download specific Python versions on demand* [8] [9].

Compare tools: `pip` + `venv` is simple but has no lockfile by default [10]. **Conda** offers rich data science packages (MKL, CUDA), but uses its own ecosystem and licensing (Anaconda's license may restrict commercial use [11]). Poetry (or PDM, Hatch) leverages PyPI and PEP 621 standards (via `pyproject.toml`) [12], making it easier to switch tools later. Notably, Poetry 2.0 (Jan 2025) now fully supports the standard `[project]` table in `pyproject.toml` [13], aligning with PEP 621.

For testing across Python versions, use tools like **Tox**, **Nox**, or Hatch's environments. Tox reads a static `tox.ini`, while Nox uses a `noxfile.py` for flexibility. For example, a Nox session can create a fresh env, install pytest, and run tests [14]:

```
import nox

@nox.session
def tests(session):
    session.install('pytest')
    session.run('pytest')
```

Running `nox` executes all defined sessions. Hatch can also run tests in isolated environments and even auto-download required Python versions if needed [8].

Secrets management: Never hardcode credentials. For local dev, keep secrets in a `.env` file loaded with `python-dotenv`. Add `.env` to `.gitignore` and do *not* commit it [15]. Example usage in code:

```
from dotenv import load_dotenv
import os

load_dotenv()  # loads variables from .env into os.environ
db_pass = os.getenv("DB_PASSWORD")
```

In CI/CD or production, use secure stores: e.g. **GitHub Secrets**, AWS Secrets Manager, or **HashiCorp Vault**. Vault is widely used in enterprises as an open-source KMS that "manages secrets and keys across multiple environments" with strong controls [16]. (See Vault's docs or hvac library for integration.) Inject secrets at runtime (via environment variables or injected files) rather than storing them in code.

Project Structure and Packaging Standards

Adopt the **src-layout** pattern: put your package code under a `src/` directory. For example:

```
my_project/
├── LICENSE
├── README.md
├── pyproject.toml
```

```
├── src/
│   └── my_project/
│       ├── __init__.py
│       └── module.py
├── tests/
│   └── test_module.py
└── Dockerfile
```

This avoids Python import quirks: as the Python Packaging User Guide notes, src-layout "helps prevent accidental usage of the in-development copy of the code" [17]. In a flat layout, the root directory appears on `sys.path` and can mask installed packages; the src-layout enforces that you must install (even as editable) to import your package [18] [17]. The pyOpenSci guide similarly *"strongly suggest[s]"* using the src layout for reliability [19].

Put test code in a separate `tests/` directory (mirroring the `src/` layout) so that test imports are explicit. Data-science projects often include top-level folders for: `data/` (raw, processed datasets), `notebooks/` (Jupyter notebooks), `models/` (trained model artifacts), `docs/` or `mkdocs.yml` for documentation, etc. For example, the Cookiecutter Data Science template has `data/raw`, `data/processed`, `notebooks/`, `models/`, and even `reports/` and `references/` [5] [6]. Always `.gitignore` large or sensitive data; consider DVC or cloud storage for big datasets.

Define all metadata and build config in `pyproject.toml`. Include a `[build-system]` table (PEP 518) specifying the build-backend (e.g. `hatchling` or `setuptools.build_meta`) [12]. Use the `[project]` table (PEP 621) for name, version, dependencies, etc. (Poetry 2.0 now supports this standard table alongside `[tool.poetry]` [13].) Tool configurations go under `[tool.*]` in the same file: e.g. `[tool.black]`, `[tool.ruff]`, `[tool.hatch.env]`, etc., centralizing setup. Avoid legacy `setup.py` unless needed for custom build scripts [13].

If containerizing, include a `Dockerfile` (and optionally `docker-compose.yml`). Use a specific base image tag (e.g. `python:3.13-slim`) rather than `python:latest` [20], to ensure reproducibility. A typical Dockerfile might multi-stage build to install dependencies and copy only the `src/` code. Also add a `.dockerignore` to exclude source artifacts (e.g. `__pycache__`, `.git`) for smaller images. Document any dev-environments (e.g. `.devcontainer/` for VSCode Dev Containers).

Quality Assurance Toolchain

Formatters: Use opinionated auto-formatters to enforce style. **Black** is a de facto standard; **Ruff** (a Rust-based tool) now offers compatible formatting (`ruff --fix`) at much higher speed [21]. Ruff can replace Black entirely if desired. For example, `ruff --select=F` lints and `ruff --fix` formats code. Also use **isort** (for imports) — Ruff can auto-sort imports too. Add these to a **pre-commit** config. Example `.pre-commit-config.yaml`:

```
repos:
  - repo: https://github.com/psf/black
    rev: stable
    hooks:
      - id: black
  - repo: https://github.com/astral-sh/ruff-pre-commit
```

```
    rev: v0.XX.0
    hooks:
      - id: ruff
        args: ["--fix"]
  - repo: https://github.com/PyCQA/isort
    rev: 5.11.4
    hooks:
      - id: isort
```

Linters: Ruff also subsumes many Flake8 checks and can enforce style/quality in one tool [21]. Still consider **pylint** or **pydocstyle** for specific checks. Run linters in CI to catch issues early.

Type Checkers: Adopt static typing. Use **MyPy** or **Pyright** to validate types. MyPy is mature, while Pyright (Microsoft) is very fast and now popular. Add `mypy` (or `pyright`) to pre-commit and CI. Type hints improve code robustness, especially for complex data pipelines.

Pre-commit Hooks: Integrate all checks as pre-commit hooks so code is formatted and linted on each commit (before pushes). Tools like cookiecutter-poetry include pre-commit configs for Ruff, MyPy, etc. [22]. Run `pre-commit run --all-files` in CI to enforce them.

Testing & Coverage: Use **pytest** as the testing framework. Write tests in `tests/` and name them `test_*.py`. Example command to run tests:

```
pytest --maxfail=1 --disable-warnings -q
```

Aim for high coverage but balance with meaningful tests. Use `pytest-cov` or `coverage.py`:

```
coverage run -m pytest && coverage report --fail-under=80
```

to enforce a minimum (e.g. 80%) coverage. Continuous Integration should fail if coverage is too low.

Task Runners: Tox, Nox, or Hatch tasks can automate sequences like linting, testing, building docs. Tox uses `tox.ini` with an `[env]` matrix; Nox uses Python `noxfile.py` sessions [14]. For simple projects, you might skip Tox/Nox and rely on GitHub Actions matrix. Hatch has a built-in concept of *environments* with named scripts: for instance, you can define a "test" environment with custom commands for unit vs integration tests [23].

Security Scanning: Integrate security checks in CI. Use **Bandit** to analyze code for common security issues (e.g. shell injection, unsafe deserialization) [24]. Use **Safety** (or `pip-audit`) to scan Python dependencies against vulnerability databases [25]. For example, `safety check -r requirements.txt`. Google's **OSV-Scanner** (CLI) can also check project lockfiles against the OSV database [26]. These should run periodically or on pull requests to catch known CVEs in libraries.

Documentation Approaches

Compare **Sphinx** vs **MkDocs-Material** for docs. Sphinx (RST-based) is traditional for Python libs; it supports autodoc to pull docstrings into docs. MkDocs (Markdown) with the Material theme is simpler to set up. Both support API docs: Sphinx's `autodoc` or MkDocs' `mkdocstrings` plugin. Cookiecutter-

poetry, for instance, defaults to MkDocs [4], and the data science template provides a `docs/` folder configured for MkDocs. Choose one style for consistency.

For docstrings, pick a style and use it project-wide. Google style or NumPy style are popular for readability; Sphinx autodoc can parse either if configured (Sphinx has `napoleon` to read Google/NumPy docstrings). Google style is concise and often easier for simple functions, while NumPy style (used by numpy, scipy) handles complex parameters well. The key is consistency. Docstring linters (e.g. `pydocstyle`, `darglint`) can enforce presence and style.

Host the docs on **GitHub Pages** or **ReadTheDocs**. MkDocs can easily be built and pushed to GitHub Pages (e.g. via `mkdocs gh-deploy`). Sphinx docs integrate seamlessly with ReadTheDocs (auto-trigger on tags). Automate documentation builds in CI: e.g. a GitHub Action that installs docs dependencies and runs `mkdocs build` or `sphinx-build`.

Measure docs completeness: use coverage tools. Sphinx's `sphinx.ext.coverage` can report undocumented items [27]. You can also track docstring coverage (what percent of functions have docs) with plugins or by ensuring `:undoc-members:` is disabled so that missing docs cause warnings.

CI/CD Implementation

Use **GitHub Actions** (or similar) for CI/CD. A typical workflow (`.github/workflows/ci.yml`) tests on pushes/PRs with a Python matrix. For example:

```
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.12, 3.13]   # Test latest Python versions [28]
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install poetry
          poetry install
      - name: Lint & Format Check
        run: |
          poetry run ruff src/ tests/ --exit-zero --show-source --show-fixes
          poetry run black --check .
      - name: Type Check
        run: poetry run mypy src/
      - name: Test Suite
```

```
        run: |
          poetry run pytest --maxfail=1 --disable-warnings -q
      - name: Coverage Report
        run: |
          poetry run coverage run -m pytest
          coverage report --fail-under=80
```

The `strategy.matrix` key runs jobs across Python versions (and OS, if needed) [28] . Adjust steps for your toolchain (pip, pipenv, etc.). Include steps for building docs, e.g. `poetry run mkdocs build`, if desired.

For **release automation**, use actions like `actions/create-release@v1` or **python-semantic-release**. One pattern is using Conventional Commits for versioning, then a GitHub Action that bumps `pyproject.toml` and publishes to PyPI/Artifactory on merge to `main`. The `python-semantic-release` tool can automate changelogs and PyPI releases when a PR is merged.

Container CI/CD: If deploying as a container, include steps to build and push Docker images. For example, use docker/build-push-action in your workflow:

```
  - name: Build and Push Docker Image
    uses: docker/build-push-action@v4
    with:
      context: .
      push: true
      tags: ghcr.io/your-org/your-app:${{ github.sha }}
```

This builds an image (based on your `Dockerfile`) and pushes it to a registry. You can use the Python matrix and a separate job to test inside containers as needed. Finally, automate deployment (e.g. to Kubernetes, AWS ECS) in another workflow or stage once builds pass.

Production Readiness Considerations

Logging: For cloud-native apps, use structured logging (JSON). Libraries like **structlog** make this easy. In dev, you might pretty-print logs, but in production output JSON. As structlog docs say, "in production you should emit structured output (like JSON)" for easy aggregation [29] . A simple config:

```
import structlog, logging
logging.basicConfig(format="%(message)s", stream=sys.stdout,
level=logging.INFO)
structlog.configure(processors=[
    structlog.processors.KeyValueRenderer()
])
log = structlog.get_logger()
log.info("app_startup", version="1.0", env="prod")
```

This ensures each log line is JSON with keys like `timestamp`, `level`, and custom data, which can be ingested by log systems (ELK, Splunk, etc.). Include context (correlation IDs) if microservices are used.

Monitoring & Observability: Instrument metrics with Prometheus (use the `prometheus_client` Python package). For example:

```
from prometheus_client import start_http_server, Counter

c = Counter('requests_total', 'Total HTTP requests')
start_http_server(8000)  # Exposes /metrics
def handle_request():
    c.inc()
    # handle request...
```

Use **OpenTelemetry** for distributed tracing and metrics; many frameworks have OTEL support. For example, use the OpenTelemetry Python SDK to generate traces and export to Jaeger or Zipkin. Monitoring dashboards (Grafana) and alerting should be set up around these metrics.

Backup and Disaster Recovery: Follow the **3-2-1 backup rule** [30] : keep 3 copies of data, on 2 different media, with 1 off-site. For data science, back up raw and processed data, model artifacts, and code. Use automated backups: e.g. nightly AWS S3 sync or database snapshots. Test recovery procedures. Document backup schedules and retention policies. For example, in AWS, you might enable daily snapshots of RDS and S3 replication to a different region.

Deployment Documentation: Maintain runbooks or README sections describing deployment steps, environment variables, and rollback procedures. For container apps, document how to apply Helm charts or Kubernetes manifests. Include architecture diagrams and config examples. Version these docs alongside the code so that they match each release. This ensures on-call engineers can reliably redeploy or recover the system.

Long-term Maintenance Planning

Dependency Updates: Keep libraries up-to-date automatically. Use bots like **Renovate** or **Dependabot**. These create PRs for new versions of dependencies (including transitive updates) according to your schedule. As Astral's docs note, "It is considered best practice to regularly update dependencies, to avoid being exposed to vulnerabilities, limit incompatibilities, and avoid complex upgrades later" [31] . Configure the bot to update `pyproject.toml` and lockfiles; for example, Renovate can refresh a `poetry.lock` or `hatch.lock` file and bump versions. Review and merge these PRs promptly.

License Compliance: Periodically scan dependencies' licenses. Tools like `pip-licenses` or `pip-license-checker` list all licenses of installed packages [32] . Ensure none violate corporate policy (e.g. avoid GPL if prohibited). GitHub's licensee or FOSS compliance tools can help audit. If using OSS in an enterprise, maintain a list of approved licenses and run these scanners in CI as a check.

Copyright Maintenance: If you include code with copyright notices, update the year or contributors file as needed. Many teams use a `CONTRIBUTORS.md` and rely on Git history. For each release, update `CHANGELOG.md` with authorship. Some use pre-commit hooks (e.g. licenseheaders) to ensure each source file has correct header info.

Deprecation Policy: When removing or changing features, mark them as deprecated in docs and code. In Python, a common pattern is:

```
import warnings

def old_function(...):
    warnings.warn("old_function is deprecated; use new_function instead",
DeprecationWarning, stacklevel=2)
    # ...
```

Additionally, use decorators (e.g. the `deprecated` library) to annotate deprecated functions or classes. Document the deprecation timeline (which release it was deprecated in, when it will be removed) in the docstring. Keep a `DEPRECATION.md` in docs outlining overall deprecation strategy (e.g., support for Python versions or APIs). Update this documentation as part of maintenance.

Version & Policy Updates: Lastly, stay aligned with evolving best practices. For example, by 2025 the community has largely moved to `pyproject.toml`-based builds and tools like Black/Ruff for formatting. Reflect major ecosystem shifts in your project's template and docs so new team members and developers follow current standards.

Sources: Modern Python tooling and practices are documented in the PyPA guides and community templates [33] [12]. Examples include the *Python Packaging Guide* and cookiecutter templates [4] [5], which illustrate these patterns in practice. The advice above incorporates up-to-date best practices and tools (as of 2025) to ensure a robust, maintainable data-science Python project.

---

[1] Introduction - Hatch
https://hatch.pypa.io/1.9/intro/

[2] Blog - Exygy | Exygy Recognized as One of the Fastest-Growing Companies by Inc. 5000
https://www.exygy.com/blog/which-license-should-i-use-mit-vs-apache-vs-gpl

[3] Apache License 2.0 Explained | Apache 2.0 Uses, Benefits & Requirements | Snyk
https://snyk.io/articles/apache-license/

[4] [22] GitHub - fpgmaas/cookiecutter-poetry: A modern cookiecutter template for Python projects that use Poetry for dependency management
https://github.com/fpgmaas/cookiecutter-poetry

[5] [6] Cookiecutter Data Science
https://cookiecutter-data-science.drivendata.org/

[7] An unbiased evaluation of environment management and packaging tools
https://alpopkes.com/posts/python/packaging_tools/

[8] Python Environment Management with Hatch - Earthly Blog
https://earthly.dev/blog/python-hatch/

[9] [23] Why Hatch? - Hatch
https://hatch.pypa.io/1.13/why/

[10] [11] Comparing the best Python project managers | by Digital Power | Medium
https://medium.com/@digitalpower/comparing-the-best-python-project-managers-46061072bc3f

[12] [13] Writing your pyproject.toml - Python Packaging User Guide
https://packaging.python.org/en/latest/guides/writing-pyproject-toml/

14 Welcome to Nox — Nox 2025.2.9 documentation
https://nox.thea.codes/

15 16 How to Handle Secrets in Python
https://blog.gitguardian.com/how-to-handle-secrets-in-python/

17 18 33 src layout vs flat layout - Python Packaging User Guide
https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/

19 Python Package Structure for Scientific Python Projects — Python Packaging Guide
https://www.pyopensci.org/python-package-guide/package-structure-code/python-package-structure.html

20 Best practices for containerizing Python applications with Docker | Snyk
https://snyk.io/blog/best-practices-containerizing-python-docker/

21 Black vs Ruff - What's the difference?
https://www.packetcoders.io/whats-the-difference-black-vs-ruff/

24 Welcome to Bandit — Bandit documentation
https://bandit.readthedocs.io/

25 safety · PyPI
https://pypi.org/project/safety/

26 GitHub - google/osv-scanner: Vulnerability scanner written in Go which uses the data provided by https://osv.dev
https://github.com/google/osv-scanner

27 sphinx.ext.coverage – Collect doc coverage stats — Sphinx documentation
https://www.sphinx-doc.org/en/master/usage/extensions/coverage.html

28 Building and testing Python - GitHub Docs
https://docs.github.com/en/actions/use-cases-and-examples/building-and-testing/building-and-testing-python

29 Logging Best Practices - structlog 25.3.0 documentation
https://www.structlog.org/en/stable/logging-best-practices.html

30 What is the 3-2-1 backup rule?
https://www.veeam.com/blog/321-backup-rule.html

31 Using uv with dependency bots | uv
https://docs.astral.sh/uv/guides/integration/dependency-bots/

32 GitHub - CodeScoring/awesome-open-source-licensing: Cool links, tools & papers related to Open Source Licensing
https://github.com/CodeScoring/awesome-open-source-licensing