ChatGPT

Python Packaging Project Setup Guide – *wellsfargo_1pii_kpi_automation*

Introduction

This document provides guidelines for setting up a Python packaging project in a Wells Fargo internal context, using the `wellsfargo_1pii_kpi_automation` project as an example. It covers two standard methods of packaging a Python project:

- **Setuptools with** `setup.cfg` **+** `pyproject.toml` – the traditional packaging approach using Setuptools.
- **Poetry with** `pyproject.toml` – a modern all-in-one approach using Poetry.

We compare these methods, highlight why **mixing both systems in one project is discouraged**, and present recommended configurations for each. Explanations are provided for each configuration section (metadata, options, dependencies, build system, etc.), along with notes on using Artifactory as a private PyPI repository. Finally, we outline best practices regarding versioning (semantic versioning), CI/CD pipeline compatibility, and dependency locking for reproducible environments.

Comparison of Packaging Methods

Both Setuptools and Poetry achieve the goal of making your Python project installable as a package, but they differ in configuration style and tooling. The table below summarizes key differences:

Aspect	Setuptools (setup.cfg)	Poetry (pyproject.toml)
Configuration Files	`setup.cfg` (for package metadata) + minimal `pyproject.toml` (PEP 517/518 build config). May also include `setup.py` (often minimal or none).	Single `pyproject.toml` file containing all metadata under `[tool.poetry]` and dependencies under `[tool.poetry.dependencies]`. No separate `setup.cfg` or `setup.py`.
Metadata Definition	Declared in `[metadata]` and `[options]` sections of `setup.cfg` (or in `setup.py` script). Uses an INI-like syntax.	Declared in the `[tool.poetry]` table in TOML format. Poetry requires fields like name, version, description, authors, etc. [1].

Aspect	Setuptools (setup.cfg)	Poetry (pyproject.toml)
Dependencies Definition	Listed as `install_requires` in `setup.cfg`'s `[options]` section (or `requirements.txt` for dev/test). No built-in lock file; rely on pip to resolve at install time.	Listed in `[tool.poetry.dependencies]` (and dev in `[tool.poetry.dev-dependencies]`). Poetry generates a `poetry.lock` lockfile to pin exact versions for reproducibility [2] [3].
Build System Backend	Setuptools (specified via `[build-system]` in pyproject). Example: `setuptools.build_meta` with requirements `setuptools` and `wheel` [4].	Poetry's core backend (`poetry-core`) (via `[build-system]` in pyproject). Example: `build-backend = "poetry.core.masonry.api"` with requirement `poetry-core>=1.0.0`.
Tooling & Workflow	Use pip or `build` to create distributions (wheel, sdist). E.g. `pip install .` or `python -m build` uses Setuptools. Dependencies installed via pip, possibly using a `requirements.txt` for dev.	Use Poetry CLI for most tasks (`poetry install`, `poetry add`, `poetry build`, `poetry publish`). Alternatively, pip can build using the Poetry backend since `poetry-core` is installed, but typically Poetry manages env and build.
Dev Dependencies	Managed outside of packaging (e.g., separate `dev-requirements.txt` or using pip-tools). Not included in the package metadata.	Declared in `[tool.poetry.dev-dependencies]` (or optional groups) in the same pyproject. Not included in built distribution, but captured in lockfile for dev environment.
Lockfile for Reproducibility	No native lockfile. Recommended to use `pip freeze` or `pip-compile` to create a requirements lock file for exact versions in CI.	Uses `poetry.lock` to lock dependency versions. This should be committed to version control for reproducible installs across environments.
Entry Points / Scripts	Defined under `[options.entry_points]` in `setup.cfg` (e.g. console_scripts for CLI).	Defined under `[tool.poetry.scripts]` in `pyproject.toml` to create console script entry points.
Repository Sources	By default uses PyPI or an index URL configured in pip. Private index (Artifactory) is configured via pip settings or environment.	Can specify additional package indexes in pyproject via `[[tool.poetry.source]]` (or use `poetry config` to add repositories). Supports authenticating to private repos (Artifactory) via config [5].

Aspect	Setuptools (setup.cfg)	Poetry (pyproject.toml)
CI/CD Integration	Requires installing build dependencies (setuptools, wheel). Use pip to install and test. Might require separate steps for resolving dev requirements.	Poetry can manage a venv in CI or you can install `poetry` to use it. Alternatively, in CI one can use `pip install .` which will invoke Poetry's build backend. Poetry's lockfile ensures consistent deps in CI.
Use Case Considerations	Well-suited for projects that prefer minimal dependencies (leveraging tools that come with pip). Many existing projects and tools expect setup.cfg.	Good for projects that want an all-in-one tool for dependency management and packaging. Facilitates easier dependency management and environment isolation, at the cost of introducing the Poetry tool.

Table: High-level comparison of Setuptools vs Poetry packaging approaches.

In summary, **Setuptools** relies on standard Python packaging tooling and separate configuration files, whereas **Poetry** provides a more integrated approach with a single config and built-in dependency management. **Mixing the two systems in one project is strongly discouraged**, as it can lead to conflicting configurations and tools ignoring each other's settings (detailed below).

Packaging with Setuptools (`setup.cfg` and minimal `pyproject.toml`)

In the Setuptools-based approach, we use the `setup.cfg` file to declaratively specify package metadata and options, and a minimal `pyproject.toml` to indicate the build system requirements. This is the "traditional" method aligning with standard setuptools usage and PEP 517/518 compliance.

Project Structure: It is recommended to use a structured layout for the project. For example:

```
wellsfargo_1pii_kpi_automation/
├── src/
│   └── wellsfargo_1pii_kpi_automation/
│       ├── __init__.py
│       └── ... (your Python modules)
├── tests/
│   └── ... (your test modules)
├── setup.cfg
├── pyproject.toml
├── README.md
└── LICENSE
```

In this structure, the package code resides in `src/wellsfargo_1pii_kpi_automation`. The `setup.cfg` and `pyproject.toml` live at the project root.

Example: `setup.cfg` **(Setuptools Configuration)**

```
[metadata]
name = wellsfargo_1pii_kpi_automation
version = 0.1.0
description = Automation of 1PII KPI calculations and reporting
author = Wells Fargo (Internal Team)
license = Proprietary
classifiers =
    Programming Language :: Python :: 3.9
    Programming Language :: Python :: 3.10
    Programming Language :: Python :: 3.11
    License :: Other/Proprietary
    Operating System :: OS Independent

[options]
packages = find:
package_dir =
    = src
python_requires = >=3.9
install_requires =
    pandas>=1.5,<2.0
    openpyxl>=3.0
include_package_data = True

[options.packages.find]
where = src

; Optional: include console script entry point
[options.entry_points]
console_scripts =
    kpi-auto = wellsfargo_1pii_kpi_automation.cli:main
```

Explanation:

- `[metadata]`: Contains package metadata:
- `name`: Distribution name of the package (should be unique).
- `version`: Package version (here `0.1.0` following semantic versioning).
- `description`: Short summary of the package.
- `author`: Name of the author or team (for internal projects, this could be a team or department).
- `license`: License identifier (e.g., "Proprietary" for internal code, or an OSI license if applicable).
- `classifiers`: A list of Trove classifiers for package indexing. These include programming language version support and license, etc.
- `[options]`: Main package configuration:

- `packages = find:` tells setuptools to automatically find sub-packages. We then specify `package_dir` to point to the `src` directory.
- `package_dir` mapping `'= src'` indicates that our package modules are under the `src/` directory.
- `python_requires`: Specifies the Python version compatibility (here, Python 3.9+).
- `install_requires`: Lists runtime dependencies (with version constraints). In this example, the project depends on `pandas` (>=1.5,<2.0) and `openpyxl` (>=3.0). These will be installed when someone installs the package.

- `include_package_data`: If `True`, includes files specified in MANIFEST.in or data files inside packages.

- `[options.packages.find]`: Additional options for package finding:

- `where = src` directs setuptools to look inside the `src` directory for packages. This pairs with the `packages = find:` directive above. (If the code was not in a `src` folder, this could be omitted or adjusted.)

- `[options.entry_points]` (optional): Defines console scripts or other entry points. In this example, we define a console script named `kpi-auto` that points to a `main` function in a `wellsfargo_1pii_kpi_automation/cli.py` module. This means after installation, running `kpi-auto` will execute that `main()` function. (This section is only needed if the project provides command-line tools.)

With `setup.cfg` configured, we also create a minimal `pyproject.toml` to specify Setuptools as the build backend.

Example: `pyproject.toml` (for Setuptools build system)

```
[build-system]
requires = [
    "setuptools>=61.0",
    "wheel"
]
build-backend = "setuptools.build_meta"
```

Explanation:

- `[build-system]`: This section is mandated by PEP 517/518 to declare the build system.
- `requires`: Lists the packages needed to build the project. Here we require setuptools (version 61.0 or higher, to support `setup.cfg` declarative config) and `wheel` (so that wheels can be built) [4]. This ensures that when tools like pip build the project, they first install these build requirements.
- `build-backend`: Specifies the build backend. `"setuptools.build_meta"` is the standard backend for Setuptools builds. This tells pip or build frontends to use Setuptools to build the package (using the config in `setup.cfg`).

This minimal pyproject.toml is enough for Setuptools. **No** `[tool.poetry]` **section is present**, since we are not using Poetry in this approach. Tools like pip will read this file, install Setuptools and wheel, then execute the build.

Usage and Workflow (Setuptools method)

- **Building the package**: You can build distribution artifacts using the standard PyPA tool `build` or pip. For example:
- `python -m build` will create a source distribution (`.tar.gz`) and a wheel (`.whl`) in the `dist/` directory.
- Alternatively, `pip install .` will build and install the package in one step.
- **Installing in development**: For dev work, you might use `pip install -e .` (editable install) to install the package in development mode. Setuptools will use the above configuration to install the package in a way that edits in code are reflected.
- **Managing dev dependencies**: Since Setuptools does not handle dev dependencies in `setup.cfg`, you might maintain a `requirements-dev.txt` or use a tool like `pip-tools` to pin dev requirements. These could include testing frameworks (e.g., `pytest`) or linters which are not needed in production installation.
- **Publishing to Artifactory**: To publish the built package to an internal PyPI (Artifactory), you can use tools like Twine. For example:
- Build the distributions as above, then run `twine upload --repository-url <Artifactory URL> -u <user> -p <api_key> dist/*`. The `<Artifactory URL>` would be the endpoint of your PyPI repository in Artifactory (e.g., `https://<artifactory>/artifactory/api/pypi/<repo>/`).
- Ensure your project's version is updated (in `setup.cfg`) before each release, following semantic versioning.

Packaging with Poetry (`pyproject.toml` only)

In the Poetry-based approach, all project configuration is consolidated in a single `pyproject.toml` file. Poetry manages package metadata, dependencies, and build configuration from this file, and it also maintains a lock file for precise dependency versions. **No** `setup.cfg` **or** `setup.py` **is needed** when using Poetry.

Start by ensuring Poetry is initialized for your project (e.g., via `poetry init` or `poetry new`). This will create a basic `pyproject.toml`. You can then edit it as needed. For our example project, the pyproject might look like:

```
[tool.poetry]
name = "wellsfargo_1pii_kpi_automation"
version = "0.1.0"
description = "Automation of 1PII KPI calculations and reporting"
authors = ["Wells Fargo Team <[email protected]>"]
license = "Proprietary"
readme = "README.md"
# If relevant, specify the package as a package (for src layout)
```

```
packages = [
    { include = "wellsfargo_1pii_kpi_automation", from = "src" }
]

[tool.poetry.dependencies]
python = ">=3.9,<4.0"
pandas = "^1.5.0"
openpyxl = "^3.0.0"
# Other dependencies can be listed with version constraints (Caret (^) means
compatible releases)

[tool.poetry.dev-dependencies]
pytest = "^7.0"
flake8 = "^6.0"
# Dev/test dependencies not included in the final package

[tool.poetry.scripts]
kpi-auto = "wellsfargo_1pii_kpi_automation.cli:main"

# Optional: If using an internal PyPI repository in Artifactory for some
dependencies
# [[tool.poetry.source]]
# name = "artifactory-internal"
# url = "https://<your_artifactory_domain>/artifactory/api/pypi/<repo-name>/
simple/"
# default = true  # or priority = "primary"

[build-system]
requires = ["poetry-core>=1.5.0"]
build-backend = "poetry.core.masonry.api"
```

Explanation:

- `[tool.poetry]`: This table holds the core metadata for the project (managed by Poetry) [6] [7]:
- `name`: Package name (same as the distribution name, and usually the Python import package). Poetry assumes you have a package directory matching this name in your project [8].
- `version`: Package version, ideally following semantic versioning (e.g., MAJOR.MINOR.PATCH). Poetry requires the version to be PEP 440 compliant [9].
- `description`: A short description of the project.
- `authors`: List of authors with name and email.
- `license`: License identifier (if applicable). For internal projects, this could be "Proprietary" or omitted.
- `readme`: Path to README file (to include as long description).
- `packages`: (Optional) If using a `src/` layout, this specifies which packages to include and from which path. In the example, we include the `wellsfargo_1pii_kpi_automation` package from

the `src` directory. (If not using `src` layout, Poetry can auto-detect the package, but it's clearer to specify.)

- `[tool.poetry.dependencies]`: Lists all runtime dependencies of the project:

- `python`: Specifies the Python versions supported (here we require >=3.9, <4.0).

- Other lines (e.g., `pandas`, `openpyxl`) specify dependencies and versions. Poetry allows version constraints like caret (`^1.5.0`) meaning "compatible with 1.5.0" (roughly >=1.5.0,<2.0.0 in SemVer terms). These will be used by Poetry to resolve and lock exact versions in the `poetry.lock` file.

- `[tool.poetry.dev-dependencies]`: Lists development dependencies (not needed at runtime for users of the package, but used in development and testing).

- E.g., `pytest` and `flake8` are included for testing and linting. These are only installed when you do `poetry install` in a development context (unless you add `--without dev`). They will not be installed when someone installs the package from PyPI/Artifactory, because they are not part of the published package requirements.

(Note: In Poetry 1.2+, you can also define groups for dev dependencies, but the `[tool.poetry.dev-dependencies]` section remains a simple way to list them.)

- `[tool.poetry.scripts]`: Defines command-line entry points (similar to console_scripts in setuptools).

- In this example, `kpi-auto` is a script name that will invoke `wellsfargo_1pii_kpi_automation.cli:main`. This allows users to run `kpi-auto` after installing the package, same effect as described in the setuptools example.

- `[[tool.poetry.source]]` (Optional, shown commented out): This section can be repeated to add external package indexes (e.g., an internal Artifactory repository):

- `name`: Identifier for the repository (e.g., "artifactory-internal").
- `url`: The base URL of the simple index API of Artifactory PyPI repository.
- `default = true` (or `priority = "primary"` in newer Poetry) indicates this is the primary source for packages (which disables the implicit default PyPI). Alternatively, one can set `priority = "supplemental"` to use Artifactory as a secondary source [10] [11].

Use this if some dependencies are only available internally, or if you want to ensure Poetry installs from Artifactory instead of public PyPI. **Do not include credentials in this file.** Instead, use `poetry config` to set up authentication (for example, using the CLI commands `poetry config repositories.<name> <url>` and `poetry config http-basic.<name> <username> <password>` to store credentials securely on the machine running Poetry [5]).

- `[build-system]`: Specifies Poetry's build backend:

- `requires`: We require `poetry-core` (the lightweight build backend for Poetry). Version `>=1.5.0` is indicated in this example; adjust to the Poetry core version appropriate (Poetry will usually set this for you on `poetry new`).
- `build-backend`: Set to `"poetry.core.masonry.api"`, which is the entry point for Poetry's build process. This allows PEP 517 build frontends (like pip) to build the project even without the Poetry CLI installed, by using the poetry-core library.

Once this pyproject.toml is configured, all packaging information is in place. Poetry will also generate or update `poetry.lock` whenever dependencies are added or updated.

Usage and Workflow (Poetry method)

- **Installing dependencies**: Run `poetry install` to create a virtual environment and install all dependencies (including dev if not using `--no-dev`). This uses the lock file to ensure exact versions.
- **Adding dependencies**: Use `poetry add <package>` or `poetry add --dev <package>` which updates the pyproject and lockfile.
- **Building the package**: Use `poetry build` to generate the wheel and sdist (output in the `dist/` directory).
- **Publishing**: Use `poetry publish` to upload to PyPI or a configured repository. For Artifactory, ensure you configured the repository in Poetry (`poetry config repositories.<repo>` as above, and credentials). Then `poetry publish -r <repo>` will upload the package to Artifactory.
- **CI/CD**: In a CI environment, you can either:
- Install Poetry on the runner and run `poetry install && poetry run pytest` (for tests) and `poetry build` / `poetry publish`.
- Or use PEP 517 via pip: e.g., `pip install .` will use the `pyproject.toml` to install build requirements (poetry-core) and install the package. (However, to install dev dependencies in CI, using the Poetry CLI is easier.)
- The `poetry.lock` file should be committed to your git repository. This ensures that all developers and CI pipelines use the same resolved versions of dependencies, improving reproducibility [12] [13].

Avoid Mixing Setuptools and Poetry in One Project

It is **not recommended to mix** the Setuptools (setup.cfg) and Poetry approaches in the same project. Each approach expects to manage the project configuration independently, and using both can cause conflicts and confusion:

- If a `pyproject.toml` contains a `[tool.poetry]` section, Poetry will treat the project as a Poetry-managed project and ignore other configuration sources. For example, **Poetry will ignore dependencies listed in** `setup.cfg`; it reads only pyproject for that information [14].
- You would end up maintaining duplicate metadata (version, description, dependencies) in two places (setup.cfg and pyproject), which is error-prone and defeats the purpose of a single source of truth.
- Build tools may become confused about which backend to use. A pyproject with Poetry config and also a build-system pointing to setuptools could lead to unintended behavior. In fact, as one Poetry maintainer noted: *"you cannot mix Poetry's way of defining package metadata/dependency with those of setuptools... you have to decide how you want to manage your project: Poetry or setuptools. Both doesn't work."* [15]. In other words, use one method or the other, not both.

Recommendation: Choose one packaging method at project initialization. If you are migrating from one to the other, remove the obsolete configuration files. For instance, if converting an existing setuptools project to Poetry, remove `setup.cfg` (and any `setup.py`) once pyproject.toml is set up with Poetry. Conversely, if extracting a Poetry project to standard setuptools, remove the `[tool.poetry]` sections from pyproject.toml.

By keeping the configuration single-sourced, you avoid contradictory definitions and simplify the build process.

Best Practices and Additional Notes

Finally, regardless of which packaging approach you use, consider the following best practices for an internal project:

Semantic Versioning and Release Management

Adopt **semantic versioning** for your project's `version`. This means using a MAJOR.MINOR.PATCH format: - Increment the **MAJOR** version for incompatible API changes, - Increment the **MINOR** version for adding functionality in a backwards-compatible manner, - Increment the **PATCH** version for backwards-compatible bug fixes.

For example, going from `1.2.3` to `1.3.0` indicates a new feature (non-breaking), whereas `2.0.0` would indicate breaking changes. This convention helps consumers of your package (even if internal) understand the impact of an upgrade. Note that Python packaging requires versions to conform to PEP 440 syntax, which is largely compatible with semantic versioning (Poetry, for instance, enforces PEP 440 compliance [9]). Stick to numeric versions and use pre-release tags (e.g., `1.0.0b1` for a beta, `1.0.0.dev1` for a development preview) if needed for testing.

Ensure the version in your configuration (be it `setup.cfg` or `pyproject.toml`) is bumped appropriately on each release. This version is what will be published to Artifactory and is used by pip for dependency resolution.

Dependency Management and Locking

For **reproducible builds and environments**, it's important to lock dependencies: - In Poetry, the `poetry.lock` file serves this purpose by pinning exact versions of all dependencies (including transitive dependencies). Commit this lock file to version control, and use it in CI to install deps. This guarantees that all developers and CI use the same dependency versions [12] [13]. - In the setuptools approach, consider generating a `requirements.txt` (or similar) with pinned versions for the development environment or deployment. Tools like `pip freeze` can capture the environment, or better, use `pip-compile` (from **pip-tools**) to derive a locked requirements file from your `install_requires`. This isn't used by pip when installing your library (which will use the version ranges in `install_requires`), but it's useful for your own testing and CI to ensure consistency. - Avoid overly broad dependency specifications. For example, pin a minimum version that you have tested, and if possible upper-bound if you know newer major releases might break compatibility. (Poetry's caret (`^`) and tilde (`~`) version operators can help constrain ranges in a maintainable way.) - Periodically review and update dependencies to incorporate security patches and

improvements, and update the lock file (Poetry's `poetry update`, or updating the `install_requires` and regen the requirements.txt for setuptools).

CI/CD Integration

Integrate the packaging and distribution steps into your CI/CD pipeline: - **Testing**: Ensure your CI runs tests (e.g., via pytest) in an environment where your package is installed just as it would be for a user. For setuptools, this could mean doing `pip install .` (or `pip install dist/yourpkg.whl`) before running tests. For Poetry, you can use `poetry install && poetry run pytest`. - **Building**: Have the CI produce build artifacts on each release (wheels and source distributions). The `python -m build` tool can be invoked in a pipeline for setuptools projects. For Poetry, `poetry build` does the same. - **Publishing to Artifactory**: Use pipeline steps to publish the built package to Artifactory PyPI. This might involve using an API key or credentials stored in the pipeline. For example, using Twine (for setuptools) or `poetry publish` (for Poetry). Make sure credentials are not hard-coded in config files, but rather injected via environment variables or CI secret management. - **Environment Isolation**: In CI, use a fresh virtual environment for builds to avoid contamination from previous runs. Poetry automatically creates an isolated venv for you. With pip/setuptools, create a venv or use containerized jobs. - **Verification**: Optionally, after publishing, the pipeline could verify the package can be installed from Artifactory (perhaps in a staging environment) to catch any issues with missing dependencies or files.

Additional Tips

- **Don't commit build artifacts**: Wheel and sdist files in `dist/` should be produced by CI or manually for releases, but not kept under version control.
- **Documentation**: Keep your README and docstrings up-to-date as they can be included in the package metadata (e.g., long_description on PyPI).
- **Consistent Tooling**: If your team uses both methods across different projects, ensure that each project's README or docs specify how to build and release it (so a developer knows whether to use Poetry commands or standard pip/setuptools commands). Consistency within a single project is key.
- **Artifactory considerations**: If using Artifactory, ensure the index URL and credentials are properly configured. For pip/setuptools users, this might be a pip `--index-url` or pip.ini/pip.conf configuration. For Poetry, use the aforementioned `poetry source` or global config. The goal is that a developer (or CI) doing `poetry install` or `pip install yourpkg` does so seamlessly using the internal repository.

By following these guidelines, the `wellsfargo_1pii_kpi_automation` project (and others like it) will have a clear, maintainable packaging setup. Whether using Setuptools or Poetry, sticking to one approach and adhering to best practices will make the project easier to build, test, and deploy within Wells Fargo's systems.

Note: To obtain this document as a Markdown file, you can copy the above content into a text editor and save it with a `.md` extension. For a Word document, you may convert the Markdown to `.docx` using a tool (e.g., Pandoc) or paste the content into MS Word (the formatting will render thanks to the structured headings, lists, and code blocks).

---

1 2 3 6 7 12 13 Dependency Management With Python Poetry – Real Python
https://realpython.com/dependency-management-python-poetry/

4 Making Sense of pyproject.toml, setup.cfg, and setup.py - DEV Community
https://dev.to/2320sharon/making-sense-of-pyprojecttoml-setupcfg-and-setuppy-2o6m

5 Set Up PyPI with a Poetry Client
https://jfrog.com/help/r/jfrog-artifactory-documentation/set-up-pypi-with-a-poetry-client

8 Basic usage | Documentation | Poetry - Python dependency ...
https://python-poetry.org/docs/basic-usage/

9 The pyproject.toml file | Documentation | Poetry - Python dependency management and packaging made easy
https://python-poetry.org/docs/pyproject/

10 11 python - How to add a private repository using poetry? - Stack Overflow
https://stackoverflow.com/questions/74323364/how-to-add-a-private-repository-using-poetry

14 15 poetry not installing dependencies from install_requires · Issue #4648 · python-poetry/poetry · GitHub
https://github.com/python-poetry/poetry/issues/4648