A MiniGuide for Python Packaging with Poetry

Jianhua Liu

6/27/2024

Contents

1 Introduction

This MiniGuide is a revised version of Dependency Management With Python Poetry – Real Python, aiming to suit our workflow. Due to the significant amount of borrowing from the original article, this MiniGuide cannot be widely distributed without a significant revision.

1.1 Definition of Pythion packaging

As discussed in Terminology of Python packaging and management, we mean the following when we discuss Python packaging:

- Python version management
- Environment management
- Package management
- Package building
- Package publishing

Poetry is able to do everything except Python version management. It is designed to make it easier for developers to manage Python projects and their associated libraries or dependencies.

Note that although Poetry can create and manage virtual environments, we prefer to use conda for this work.

So, we will many use Poetry to simplify package management and build workflows for Python applications and libraries.

1.2 Interesting features of Poetry

We are mostly interested in the following features of Poetry:

- **Dependency Management**: Poetry provides a clear and concise way to declare, manage, and install dependencies. It ensures that you have the right versions of the libraries needed for your project. It is a good replacement of pip.
- **Dependency Resolution**: Poetry has a sophisticated dependency resolver that helps prevent version conflicts and ensures that you have a coherent set of dependencies.

  When your Python project relies on external packages, you need to make sure you're using the right version of each package. After an update, the updated package might not work as it did before. A **dependency manager** like Poetry can help you specify, install, and resolve external packages in your projects. This way, you can be sure that you always work with the correct dependency version on every machine.
- **Package Creation**: It helps you easily package and distribute your Python projects. You can define your package's properties, dependencies, and more, all in one place using `pyproject.toml`. This file is designed to replace the old version of `setup.py`.

  Poetry helps you create new projects or maintain existing projects while taking care of **dependency management** for you, all based on the `pyproject.toml` file.
- **Reproducible Builds**: It locks the versions of the entire dependency tree, ensuring that your project is reproducible and eliminating "it works on my machine" problems.

Poetry can also help you build a distribution package for your project. If you want to share your work, then you can use Poetry to publish your project on the Python Packaging Index (PyPI).

1.3 Coverage of this MiniGuide

- Creating a **new project** using Poetry.
- Adding Poetry to an **existing project.**
- Configuring the project through **`pyproject.toml`**.
- Pinning the project's **dependency versions**.
- Installing dependencies from a **`poetry.lock`** file.
- Running basic Poetry commands using the **Poetry CLI**.

1.4 Installing Poetry

Detailed instructions to installing Poetry are provided in the Installation of Poetry section.

1.5 Online documentation

https://python-poetry.org/docs/.

2 Command reference

Poetry is command-line based, and we need to use commands to perform our work.

2.1 Most commonly used Poetry commands

Poetry provides various commands to manage projects, and the following ones are most used:

- `poetry new <project-name>`: Create a new project.
- `poetry add <package-name>`: Add a new dependency to `pyproject.toml` and install it.
- `poetry remove <package-name>`: Remove a dependency.
- `poetry install`: Install all dependencies of a project as specified in `pyproject.toml` and install the project as a standard Python package. Note that the project is editable, which means when we change the code, the changes will be reflected immediately.
- `poetry build`: Build your project for distribution.

2.2 A full list of Poetry commands

Poetry Command	Explanation
`$ poetry --version`	Show the version of Poetry installation.
`$ poetry check`	Validate `pyproject.toml`.
`$ poetry config --list`	Show the Poetry configurations.
`$ poetry env use <a_python>`	Specify a Python interpreter used to create and manage an env for the project.
`$ poetry env info`	Display the info of the current virtual environment of your project.
`$ poetry env list`	List the virtual environments of your project.
`$ poetry export`	Export `poetry.lock` to other formats.
`$ poetry init`	Add Poetry to an existing project.
`$ poetry lock`	Pin the latest version of the dependencies into `poetry.lock`.
`$ poetry lock --no-update`	Refresh the `poetry.lock` file without updating any dependency version.
`$ poetry run`	Execute a command within a virtual environment managed by Poetry.
`$ poetry show`	List installed packages.
`$ poetry show --tree`	List installed packages in the form of direct and transitive packages.
`$ poetry update`	Update the project's dependencies.

You can check out the Poetry CLI documentation to learn more about the commands above and many other commands that Poetry offers. You can also run `poetry --help` to see information right in your terminal!

2.3 `poetry shell` vs `poetry run`

- `poetry shell` is used to activate the virtual environment and start an interactive shell session where we can run multiple commands, start a Python interpreter, or manage our project's environment interactively.

- `poetry run` is used to execute a specific command or script within the virtual environment without starting a new shell session. This is useful for running scripts, tests, or any single command that requires access to the project's dependencies.

Below is an example of using `poetry run` to run a script named `script.py`.

```
poetry run python script.py
```

3 Understanding the `pyproject.toml` file

One of the most important files for working with Poetry is the `pyproject.toml` file, which is a **configuration file** defined in PEP 518.

The authors of PEP 518 considered several file formats for the new configuration file, and they've finally chosen to use the **TOML** format, the Tom's Obvious Minimal Language. Note that the authors have also considered other configuration options, including YAML, JSON, CFG, and INI. TOML was choose because it is flexible with good readability.

Note that Poetry wanted to preserve the format of the `pyproject.toml` file, so they developed a powerful `tomlkit` library, which will be discussed later.

3.1 A simple `pyproject.toml` file

```
[tool.poetry]
name = "rp-poetry"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"

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

- There are three sections denoted with square brackets
- Each section is known as a table.
- Tables contain declarative instructions, with which tools like Poetry can recognize and use for **managing dependencies**, **building the project**, or performing other tasks.

3.2 Metadata and dependencies of the project

- `[tool.poetry]`. This table is used to define your project's metadata, such as the name and version.
- `[tool.poetry.dependencies]`. This table is used to specify external libraries managed by Poetry for your project.

These two are examples of TOML's **subtables**, which represent a hierarchy of configuration options. Here's their equivalent JSON representation:

```
"tool": {
    "poetry": {
        "name": "rp-poetry",
        "version": "0.1.0",
        ...
        "dependencies": {
            "python": "^3.12"
        }
    }
}
```

Note that the dot character (`.`) in a TOML table's name is a delimiter, separating the different levels of the hierarchy, much like nested objects in JSON.

In this case, the two subtables belong to only one external tool—Poetry. But you'll often find examples like `[tool.black]`, `[tool.isort]`, `[tool.mypy]`, or `[tool.pytest.ini_options]` for their corresponding tools. While many Python tools are moving their configuration to `pyproject.toml` files, there are still notable exceptions. For example, `flake8` didn't support them at the time of writing, so you might need a few separate configuration files.

3.3 Required fields

The `pyproject.toml` file starts with the `[tool.poetry]` subtable, where you can store general information about your project. Poetry defines a few table keys that are valid in this subtable. While some of them are optional, there are four required ones:

- **`name`**: The name of your distribution package that will appear on PyPI.
- **`version`**: The version of your package, ideally following semantic versioning.
- **`description`**: A short description of your package.
- **`authors`**: A list of authors, in the format `name <email>`.

3.4 Dependency management

The subtable `[tool.poetry.dependencies]` on line 8 is essential for your dependency management, which will be discussed below in more details.

3.4.1 Python version

When you create a new project with the `poetry new` command, the tool assumes the minimum Python version supported by your project based on the virtual environment Poetry itself was installed in. For example, if you installed Poetry using `pipx` on top of Python 3.12, then that interpreter version will appear in the resulting `pyproject.toml` file. Naturally, you can change this by editing the file.

3.5 The table for the build system

The last table, `[build-system]` on line 11 in the `pyproject.toml` file, defines metadata that Poetry and other build tools can work with. As mentioned earlier, this table isn't tool-specific, as it doesn't have a prefix. It has two keys:

- **`requires`**: A list of dependencies required to build your package, making this key mandatory

- **`build-backend`**: The specific backend tool used to perform the build process.

When you create a new project with Poetry, this is the `pyproject.toml` file that you start with. Over time, you'll add more configuration details about your package and the tools you use. As your project grows, your `pyproject.toml` file will grow with it. That's particularly true for the `[tool.poetry.dependencies]` subtable.

In the next section, you'll find out how to expand your project by adding third-party dependencies using Poetry.

4 Basic operations with Poetry

Below, we will use a project named `rp-poetry` to illustrate the operations.

4.1 Creating a new Poetry project

To create a new Poetry project from scratch, use the `poetry new` command followed by the desired project name.

There are a number of different cases when creating the new project.

4.1.1 Creating a project with flat layout

Go to the parent folder of the project and run the following command:

```
$ poetry new rp-poetry
```

This command creates a new folder named `rp-poetry/`. When creating the folder structure for a new project, Poetry follows the **flat layout** by default. In this layout, your Python package, `rp_poetry`, resides at the root level of the project folder, as shown below.

```
rp-poetry/
|
|-- rp_poetry/
|   `-- __init__.py
|
|-- tests/
|   `-- __init__.py
|
|-- README.md
`-- pyproject.toml
```

Note that Poetry has translated the dash (`-`) in the provided name into an underscore (`_`) in the corresponding `rp_poetry/` subfolder? This ensures you'll be able to import them as Python packages, as `rp-poetry` is not a valid identifier with the hyphen, the minus operator, in the Python syntax.

4.1.2 Creating a flat project with specified package name

To have more control over the resulting **package name**, you can optionally pass the `--name` argument, which lets you specify a different name for the Python package than your project folder:

```
$ poetry new rp-poetry --name realpoetry
```

Now, the **package** name will be changed from `rp_poetry` to `realpeotry`. The folder name of `rp-poetry` stays the same.

Note that Poetry will also normalize the package name provided through this argument should it contain a dash.

4.1.3 Creating a project with src layout

Another popular choice in Python project layout is the **src layout**, which keeps the code in an additional `src/` parent folder. If you prefer this layout instead, then pass the `--src` flag to Poetry when creating a new project:

```
$ poetry new rp-poetry --src
```

Adding this flag will result in a slightly different folder structure:

```
rp-poetry/
|
|-- src/
|   |
|   `-- rp_poetry/
|       `-- __init__.py
|
|-- tests/
|   `-- __init__.py
|
|-- README.md
`-- pyproject.toml
```

Now, your `rp_poetry` package is nested in the `src/` folder.

4.1.4 Projects with more complex structure

Both the flat and src layouts have their pros and cons, as shown in Python Packaging Guide.

Note: It's possible to specify a custom folder structure and even have multiple Python packages in one Poetry project. This can be useful in larger projects, which need to be organized into multiple, separate components. For example, the tic-tac-toe demo project comes with an engine library and several front-end components in a single distribution package.

To achieve this in Poetry, it requires manual updates in pyproject.toml.

4.2 Inspecting the project structure

Now, we only have minimum files in the project folder.

- The `rp_poetry/` subfolder has an empty `__init__.py` file, which turns the folder into an importable Python package.
- The `tests/` subfolder is meant to contain the unit tests code.
- In the root folder, an empty README file with an .md extension is created as default. The other two options are plain text or reStructuredText.
- In the root folder, `pyproject.toml` is also created with the minimum required metadata for your project.

4.3 Using virtual environments

We delegate this work to conda. No need to read the remainder of this section.

If you are interested in the creation of virtual environment using Poetry, keep reading. For more info, see the corresponding sections in Dependency Management With Python Poetry – Real Python or the locally-saved one py-pkg-4-ref-poetry.

4.3.1 Using multiple Python versions

Poetry can create a number virtual envs for the project, each with a different version of Python, as shown in Using multiple versions of Python for a project. This is one of the selling points of Poetry.

To check the virtual env, use the following command.

```
(venv) $ cd rp-poetry/ (venv)
$ poetry env info --path
/Users/Philipp/.virtualenvs/venv
```

If you want to have better control over the creation of a virtual environment, then you may tell Poetry explicitly which Python version you want to use up front:

```
$ poetry env use python3
```

With this command, you specify the path to a desired Python interpreter on your disk. Using bare `python3` will work as long as you have the corresponding Python executable in your `PATH`. You can also use the **minor Python version** like 3.12 or just 3.

When Poetry creates a new virtual environment, for example, after you've run the `poetry env use <python>` command for the first time, you'll see a message similar to this one:

```
$ poetry env use python3
Creating virtualenv rp-poetry-Dod5cRxq-py3.12 in
   /Users/Philipp/Library/Caches/pypoetry/virtualenvs
Using virtualenv:
   /Users/Philipp/Library/Caches/pypoetry/virtualenvs/rp-poetry-Dod5cRxq-py3.12
```

Note: To switch between existing environments, you can issue the `poetry env use <python>` command again. To quickly remove all environments associated with your project, run the `poetry env remove --all` command.

With that skill under your belt, you're ready to add some dependencies to your project and see Poetry shine.

4.4 Declaring runtime dependencies

When you created your `rp-poetry` project using Poetry, it scaffolded the minimal `pyproject.toml` file with the `[tool.poetry.dependencies]` subtable. So far, it only contains a declaration of the Python interpreter version, but it's also where you can specify the external Python packages that your project requires.

Editing this file by hand can become tedious, though, and it doesn't actually install anything into the project's virtual environment. That's where Poetry's CLI comes into play again.

With Poetry, we will not use pip to install packages any more; we will use Poetry instead.

Running the `poetry add` command will automatically update your `pyproject.toml` file with the new dependency and install the package at the same time. You can even specify multiple packages in one go:

```
$ poetry add requests beautifulsoup4
```

This will find the latest versions of both dependencies on PyPI, install them in the corresponding virtual environment, and insert two declarations in your `pyproject.toml` file:

```
[tool.poetry]
name = "rp-poetry"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
requests = "^2.31.0"
beautifulsoup4 = "^4.12.3"

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

The order of these declarations reflects the order in which you specified those packages in the command line.

Note the caret symbol (`^`) before the version specifiers indicates that Poetry is free to install any version matching the leftmost non-zero digit of the version string. For example, if the Requests library releases a new version `2.99.99`, then Poetry will consider it an acceptable candidate for your project. However, version `3.0` wouldn't be allowed.

Note: The idea behind this follows the semantic versioning scheme, `major.minor.patch`, where minor and patch updates shouldn't introduce backward-incompatible changes. Still, this is just an assumption that could break your builds, and some notable Python developers disagree with such a choice of defaults on behalf of Poetry.

If you'd like to add a particular version of an external package or define custom version constraints, then Poetry lets you do that on the command line:

```
$ poetry add requests==2.25.1 "beautifulsoup4<4.10"
```

When you run this command, Poetry will first remove any previously installed versions of these packages and downgrade their indirect or transitive dependencies as needed. It'll then determine the most suitable versions of these packages, taking into account other existing constraints to resolve potential conflicts.

Note: You may sometimes need to surround your package names with quotes when their version specifiers involve symbols with a special meaning. In this case, quoting the `beautifulsoup4` package prevents the shell from interpreting the less-than symbol (`<`) as a redirection operator.

If you want to remove one or more dependencies from your project, then Poetry provides the related `poetry remove` command:

```
$ poetry remove requests
Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 0 updates, 5 removals

  - Removing certifi (2023.11.17)
  - Removing chardet (4.0.0)
  - Removing idna (2.10)
  - Removing requests (2.25.1)
  - Removing urllib3 (1.26.18)

Writing lock file
```

As you can see, it'll remove the given dependency along with its transitive dependencies, so you don't need to worry about **leftover packages** that are no longer needed by your project. This is an advantage of Poetry over plain pip, which can only uninstall the individual packages.

Notice that Poetry informs you about *writing to the lock file* whenever you add or remove a dependency. Lock files will be discussed later.

So far, you've been adding runtime dependencies that were necessary to make your program work correctly. However, you might only need some dependencies at specific stages of development, such as during testing. Next up, you'll see what tools Poetry gives you to manage those kinds of dependencies.

4.5 Grouping dependencies and adding extras

Another neat feature in Poetry, which is missing from pip, is the ability to manage **groups of dependencies**, allowing you to keep logically related dependencies separate from your runtime dependencies. For example, during development, you'll often want additional packages, such as linters, type checkers, or testing frameworks, which would only bloat the final distribution package. Your users don't need those, after all.

4.5.1 Using groups

With Poetry, you can group dependencies under arbitrary names so that you can selectively install those groups later on when needed. Here's how to add a few dependencies to a group called `dev` and some dependencies to another group called `test`:

```
$ poetry add --group dev black flake8 isort mypy pylint
$ poetry add --group test pytest faker
```

Running these commands will cause Poetry to install the listed packages into the project's virtual environment and also add two additional subtables in your `pyproject.toml` file, which look as follows:

```
[tool.poetry]
name = "rp-poetry"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
```

```
beautifulsoup4 = "<4.10"

[tool.poetry.group.dev.dependencies]
black = "^24.1.1"
flake8 = "^7.0.0"
isort = "^5.13.2"
mypy = "^1.8.0"
pylint = "^3.0.3"

[tool.poetry.group.test.dependencies]
pytest = "^8.0.0"
faker = "^22.6.0"

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

The new subtables start with the name `tool.poetry.group` and end with the word `dependencies`. The part in the middle must be a unique group name.

Normally, when you don't provide any group name, Poetry puts the specified packages in an implicit **main group**, which corresponds to the `[tool.poetry.dependencies]` subtable. Therefore, you can't use the name `main` for one of your groups because it's been reserved.

Note: Before Poetry 1.2.0, you could use the `-D` or `--dev` option as a shortcut for adding dependencies to a development group. This option has been deprecated, however, as it became incompatible with regular dependency groups. The deprecated feature used to create a separate subtable in the `pyproject.toml` file.

4.5.2 Defining optional groups

You can define optional groups by setting the corresponding attribute in the `pyproject.toml` file. For example, this declaration will turn your `test` group into an optional one:

```
# ...

[tool.poetry.group.test]
optional = true

[tool.poetry.group.test.dependencies]
pytest = "^8.0.0"
faker = "^22.6.0"

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

Poetry won't install dependencies belonging to such a group unless you explicitly instruct it to by using the `--with` option. Note that you must declare another TOML subtable to mark a group as optional because the group's configuration is kept separately from its dependency list.

4.5.3 Defining optional packages

In addition to this, you can add the individual **packages as optional** to let the user choose whether to install them:

```
$ poetry add --optional mysqlclient psycopg2-binary
```

Optional dependencies are meant to be available at runtime when explicitly requested by the user during installation. It's common to mark packages as optional when they're platform-specific or when they provide features, such as a particular database adapter, that only some users will need.

In `pyproject.toml`, optional dependencies look slightly more verbose:

```
[tool.poetry]
name = "rp-poetry"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
beautifulsoup4 = "^4.12.3"
mysqlclient = {version = "^2.2.1", optional = true}
psycopg2-binary = {version = "^2.9.9", optional = true}

# ...
```

The `mysqlclient` and `psycopg2-binary` dependencies have their `optional` flag set to `true`, while their version string is kept in another attribute.

4.5.4 Defining the extras group

However, this isn't enough to expose such optional dependencies to the user. You must also define extras in your `pyproject.toml` file, which are sets of optional dependencies that your users can install together:

```
# ...

[tool.poetry.extras]
databases = ["mysqlclient", "psycopg2-binary"]
mysql = ["mysqlclient"]
pgsql = ["psycopg2-binary"]

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

Depending on your particular needs, you can opt in to the desired features by selecting one or more extras during installation. As you can see, it's also possible to combine a few extras, which might be tailored to a specific use case, such as testing.

With the above discussion, you're already well-equipped to implement Poetry in your projects. In the next section, you'll learn how to work with a Poetry project from a user's perspective.

4.6 Installing your package with Poetry

Imagine that you've just cloned a Git repository with the `rp-poetry` project from GitHub and are starting fresh with no virtual environment. To simulate that, you can remove some of Poetry's metadata and any virtual

environments associated with the project:

```
$ rm poetry.lock
$ poetry env remove --all
```

If you're a developer who wants to contribute to this project, then you can execute the `poetry install` command inside the `rp-poetry/` folder to get the ball rolling:

```
$ poetry install
```

If other contributors have committed the `poetry.lock` file to the remote repository, which they generally should, then Poetry will read that file. It'll reproduce the **exact same environment** on your machine with identical versions of all the dependencies listed in the most recent snapshot of `poetry.lock`.

Otherwise, Poetry will fall back to reading the **top-level dependencies** outlined in the `pyproject.toml` file and will resolve the set of packages satisfying the version constraints. As a result, you'll have a new local `poetry.lock` file. This may potentially lead to a different state of dependencies in your virtual environment than those of other contributors who've installed the project at a different time.

The **dependency resolution** becomes essential when you have many dependencies that require several third-party packages with different versions of their own. Before installing any packages, Poetry figures out which version of a package fulfills the version constraints that other packages set as their requirements. That's not a trivial task. In rare cases, a solution may not even exist!

By default, Poetry installs dependencies from the implicit **main group** as well as all **dependency groups**, unless you marked them as optional.

Poetry also installs your own `rp-poetry` distribution package in editable mode to allow for changes in your source code to be immediately reflected in the environment without the need for reinstallation.

In contrast, Poetry won't automatically install **extra sets of dependencies** and **optional groups** of dependencies. To get those, you must use some of the following parameters:

- `--all-extras`
- `--extras {extra}`
- `--with {optional groups}`

You can install a few extras at the same time by repeating the `--extras` parameter for each desired set of optional dependencies. However, Poetry treats the individual extras as mutually exclusive until you say otherwise. So, it'll only install those specified on the command line while removing all other extras from your virtual environment if needed. In particular, it'll remove all extras when you don't select at least one during installation.

Apart from that, you have a few more options, allowing you to cherry-pick exactly which dependencies to install, including:

Option	Meaning
`--no-root`	Install dependencies without the package itself.

Option	Meaning
`--only-root`	Install your package without its dependencies.
`--only {groups}`	Install only these dependency groups.
`--without {groups}`	Don't install these dependency groups.

When you specify the `--only {groups}` option, Poetry will ignore the `--with {optional groups}` and `--without {groups}` options.

Resolving dependencies results in updating or producing a new `poetry.lock` file. It's where Poetry keeps track of all packages and their exact versions that your project uses. You're going to have a closer look at that file now.

5 Managing dependencies using Poetry

Whenever you interact with Poetry through its command-line interface, it updates the `pyproject.toml` file and pins the resolved versions in the `poetry.lock` file. However, you don't have to let Poetry do all the work. You can manually modify dependencies in the `pyproject.toml` file and lock them afterward.

5.1 Manually locking dependencies

Poetry generates and refreshes the `poetry.lock` file when needed as long as you stick to its command-line interface. While the lock file is *not* meant to be changed by hand, you can edit the related `pyproject.toml` file at will. Unfortunately, this may sometimes cause both files to become out of sync.

Suppose you wanted to bring back the Requests library that you removed from the `rp-poetry` project earlier in this tutorial. You can open the `pyproject.toml` file in your text editor and type the necessary declaration in the main group of dependencies:

```
[tool.poetry]
name = "rp-poetry"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
requests = "*"
beautifulsoup4 = "<4.10"
mysqlclient = {version = "^2.2.1", optional = true}
psycopg2-binary = {version = "^2.9.9", optional = true}

# ...
```

By using the asterisk ($\star$) as the version constraint, you indicate that you're not specifying any particular version of the Requests library, and that any version will be acceptable. But this library isn't installed yet.

If you now open your terminal and navigate to the project's parent directory, then you can tell Poetry to install the manually added dependencies into the associated virtual environment and update the lock file:

```
$ poetry install
Installing dependencies from lock file
Warning: poetry.lock is not consistent with pyproject.toml.
   You may be getting improper dependencies.
   Run `poetry lock [--no-update]` to fix it.

Because rp-poetry depends on requests (*)
   which doesn't match any versions,
   version solving failed.
```

In this case, Poetry refuses to install the dependencies because your `poetry.lock` file doesn't currently mention the Requests library present in the companion `pyproject.toml` file. Conversely, if you removed a declaration of a resolved and installed dependency from `pyproject.toml`, then you'd face a similar complaint. Why's this happening?

Remember that Poetry always installs the **resolved dependencies** from the `poetry.lock` file, where it pinned down the exact package versions. It'll only consider the dependencies that you've listed in the `pyproject.toml` file when it needs to update or regenerate a missing lock file.

Therefore, to fix such a discrepancy, you could delete the lock file and run `poetry install` again to let Poetry resolve all dependencies from scratch. That's not the best approach, though. It's potentially time-consuming. But even worse, it disregards the specific versions of previously resolved dependencies, removing the guarantee of reproducible builds.

A far better approach to align the two files is by **manually locking** the new dependencies with the `poetry lock` command:

```
$ poetry lock
Updating dependencies
Resolving dependencies... (1.0s)

Writing lock file
```

This will update your `poetry.lock` file to match the current `pyproject.toml` file without installing any dependencies.

Poetry processes all dependencies in your `pyproject.toml` file, finds packages that satisfy the declared constraints, and pins their exact versions in the lock file. But Poetry doesn't stop there. When you run `poetry lock`, it also recursively traverses and locks all dependencies of your direct dependencies.

Note: The `poetry lock` command also updates your existing dependencies if newer versions that still fit your version constraints are available. If you don't want to update any dependencies that are already in the `poetry.lock` file, then add the `--no-update` option to the `poetry lock` command:

```
$ poetry lock --no-update
Resolving dependencies... (0.1s)
```

In this case, Poetry only resolves the new dependencies but leaves any existing dependency versions inside the `poetry.lock` file untouched.

It's important to note that dependency locking is only about two things:

- **Resolving:** Finding packages that satisfy all version constraints.
- **Pinning:** Taking a snapshot of the resolved versions in the `poetry.lock` file.

Poetry doesn't actually install the resolved and pinned dependencies for you after running `poetry lock`. To confirm this, try importing the locked Requests library from the associated virtual environment, which Poetry manages for you:

```
$ poetry run python -c "import requests"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'requests'
```

Here, you specify a one-liner program by providing the `-c` option to the Python interpreter in your virtual environment. The program tries to import the Requests library but fails due to a missing module, which wasn't installed.

Now that you've pinned all dependencies, it's time to install them so that you can use them in your project.

5.2 Synchronizing your environment

When the `poetry.lock` file agrees with its `pyproject.toml` counterpart, then you can finally install dependencies that Poetry locked for you:

```
$ poetry install
Installing dependencies from lock file

Package operations: 5 installs, 0 updates, 0 removals

  - Installing certifi (2023.11.17)
  - Installing charset-normalizer (3.3.2)
  - Installing idna (3.6)
  - Installing urllib3 (2.2.0)
  - Installing requests (2.31.0)

Installing the current project: rp-poetry (0.1.0)
```

By running `poetry install`, you make Poetry read the `poetry.lock` file and install all dependencies listed there. In this case, your virtual environment already had most of the required dependencies in place, so Poetry only installed the missing ones.

When you run the same command again, Poetry won't have much left to do anymore:

```
$ poetry install
Installing dependencies from lock file

No dependencies to install or update

Installing the current project: rp-poetry (0.1.0)
```

As a result, the Requests library should already be available for grabs when you import it within the interactive Python REPL session started through Poetry:

```
$ poetry run python -q
>>> import requests
>>> requests.__version__
'2.31.0'
```

This time, you can import `requests` without any trouble, which is great! You may now exit the interpreter by typing `exit()` and hitting Enter.

Note: The `-q` option runs Python in quiet mode, suppressing the welcome message with version and copyright information.

What if your virtual environment contains **leftover packages** that you previously installed but no longer need? Poetry doesn't mind such packages, even if they're not formally declared in `pyproject.toml` or `poetry.lock`. You could've installed them as optional dependency groups some time ago or completely outside of Poetry's control by, for instance, using `pip` directly:

```
$ poetry run python -m pip install httpie
```

The `httpie` package indirectly brings ten additional dependencies, which take up space and could potentially interfere with your project's actual dependencies. Besides, external packages might sometimes create security holes if you don't keep them up-to-date.

To synchronize your **virtual environment** with the locked packages pinned in `poetry.lock`, you can pass the optional `--sync` flag to the `poetry install` command:

```
$ poetry install --sync
Installing dependencies from lock file

Package operations: 0 installs, 0 updates, 10 removals

  - Removing defusedxml (0.7.1)
  - Removing httpie (3.2.2)
  - Removing markdown-it-py (3.0.0)
  - Removing mdurl (0.1.2)
  - Removing multidict (6.0.4)
  - Removing pygments (2.17.2)
  - Removing pysocks (1.7.1)
  - Removing requests-toolbelt (1.0.0)
  - Removing rich (13.7.0)
  - Removing setuptools (69.0.3)

Installing the current project: rp-poetry (0.1.0)
```

This ensures that your virtual environment only contains the packages specified in your `pyproject.toml` and `poetry.lock` files, preventing potential conflicts caused by unnecessary or outdated dependencies.

In the next section, you'll learn how to update your project's dependencies using Poetry.

5.3 Updating and upgrading dependencies

Suppose you added the Requests library to your project in December 2020, when the latest version of this library was the 2.25.1 release. By default, Poetry configured a **permissive version constraint** in

your `pyproject.toml` file, which involves a caret (`^2.25.1`) to allow future non-breaking updates to pass through.

Over the years, you've been adding many other dependencies to your project with `poetry add`, and Poetry automatically picked up more recent releases of Requests that still satisfied the original version constraint. Poetry then updated the lock file accordingly and installed new versions of dependencies into your virtual environment. Now, you have the 2.29.0 release of Requests pinned in the `poetry.lock` file and installed in your environment.

Fast forward to the time of writing, when the library's 2.31.0 release has become a cutting-edge version. To verify this, you can ask Poetry to compare your locked dependencies against their **latest releases on PyPI** by running this command:

```
$ poetry show --latest --top-level
beautifulsoup4     4.9.3      4.12.3     Screen-scraping library
requests           2.29.0     2.31.0     Python HTTP for Humans.
```

This new release continues to satisfy your version constraint for Requests, so Poetry should accept it. However, when you try to run `poetry install` again, it doesn't do anything:

```
$ poetry install
Installing dependencies from lock file

No dependencies to install or update

Installing the current project: rp-poetry (0.1.0)
```

Recall that the `poetry install` command prioritizes the `poetry.lock` file over your version constraints declared in `pyproject.toml` to ensure reproducible environments. If you want to update dependencies with compatible versions, then you have the following choices:

- Remove `poetry.lock` and run `poetry install`
- Run `poetry lock` followed by `poetry install`
- Run `poetry update`

As mentioned earlier, the first option has its drawbacks because it forces Poetry to recalculate all dependencies from scratch. The other two options are essentially equivalent, so you might choose the latter to handle both steps at once.

Updating dependencies always carries some risk. To better understand what's about to happen, you can ask Poetry to perform a **dry run** before taking the dive:

```
$ poetry update --dry-run
Updating dependencies
Resolving dependencies... (0.7s)

Package operations: 0 installs, 2 updates, 0 removals, 3 skipped

  - Updating urllib3 (1.26.18 -> 2.2.0)
  - Installing certifi (2023.11.17): Skipped (...) Already installed
  - Installing charset-normalizer (3.3.2): Skipped (...) Already installed
  - Installing idna (3.6): Skipped (...) Already installed
```

```
  - Updating requests (2.29.0 -> 2.31.0)
```

The highlighted lines indicate which dependencies will be updated and in which direction.

The `poetry update` command will lock and update *all* packages along with their dependencies to their latest compatible versions. If you want to update one or more specific packages, then you can list them as arguments:

```
$ poetry update requests beautifulsoup4
```

With this command, Poetry will search for a new version of `requests` and a new version of `beautifulsoup4` that fulfill the version constraints listed in your `pyproject.toml` file. Then, it'll resolve all dependencies of your project and pin their versions in your `poetry.lock` file. At the same time, your `pyproject.toml` file won't change because the listed constraints remain valid.

Normally, to **upgrade a dependency** to a version that's outside of the version constraints declared in your `pyproject.toml` file, you must adjust that file beforehand. Alternatively, you can forcefully upgrade a dependency to its **latest version** by running the `poetry add` command with the at operator (`@`) and a special keyword, `latest`:

```
$ poetry add requests@latest
```

When you run the `poetry add` command with the `latest` keyword, Poetry will ignore your current version constraint in `pyproject.toml` and replace it with a new constraint based on the latest version found. It's as if you've never added that dependency to your project before. Use this option with caution, as an incompatible version of one of your dependencies could break the project.

If you now open your `pyproject.toml` file, then the version constraint for the Requests library will look as follows:

```
[tool.poetry]
name = "rp-poetry"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
requests = "^2.31.0"
beautifulsoup4 = "<4.10"
mysqlclient = {version = "^2.2.1", optional = true}
psycopg2-binary = {version = "^2.9.9", optional = true}

# ...
```

Without using the `latest` keyword or an explicit version constraint in the `poetry add` command, Poetry would conclude that the requested package is already present in your project and would do nothing.

Now you've gotten a handle on how Poetry uses `pyproject.toml` and `poetry.lock`. Next up, you'll have a final look at both files.

5.4 Comparing `pyproject.toml` and `poetry.lock`

The version constraints of the dependencies declared in your `pyproject.toml` file can be fairly loose. This allows for some level of flexibility when incorporating **bug fixes** or resolving **version conflicts**. By having more package versions to choose from, Poetry is more likely to find a combination of compatible dependencies.

On the other hand, Poetry tracks the exact versions of the dependencies that you're actually using in the `poetry.lock` file. This improves Poetry's **performance** by caching the resolved package versions so that it doesn't have to resolve them again every time you install or update your dependencies.

To ensure **reproducible environments** across your team, you should consider committing the `poetry.lock` file to your version control system like Git. By keeping this file tracked in a version control system, you ensure that all developers will use identical versions of required packages. However, there's one notable exception.

When you develop a library rather than an application, it's common practice *not* to commit the `poetry.lock` file. Libraries typically need to remain compatible with multiple versions of their dependencies rather than with a single, locked-down set.

When you come across a repository that contains a `poetry.lock` file, it's a good idea to use Poetry to manage its dependencies. On the other hand, if other developers on your team aren't using Poetry yet, then you should coordinate with them about the potential adoption of Poetry across the board before making any decisions. You must always maintain compatibility with other dependency management tools that the team might be using.

6 Adding Poetry to an existing project

Chances are, you already have some projects that didn't start their life with the `poetry new` command. Or maybe you inherited a project that wasn't created with Poetry, but now you want to use Poetry for your dependency management. In these types of situations, you can add Poetry to existing Python projects.

6.1 Convert a Folder Into a Poetry Project

Say you have an `rp-hello/` folder with a `hello.py` script inside:

```
print("Hello, World!")
```

It's the classic `Hello, World!` program, which prints the famous string on the screen. But maybe this is just the beginning of a grand project, so you decide to add Poetry to it. Instead of using the `poetry new` command from before, you'll use the `poetry init` command inside your project folder:

```
$ cd rp-hello/
$ poetry init
```

This command will guide you through creating your `pyproject.toml` config.

```
Package name [rp-hello]:
Version [0.1.0]:
Description []: My Grand Project
Author [Philipp Acsany <philipp@realpython.com>, n to skip]:
License []:
Compatible Python versions [^3.12]:
```

```
Would you like to define your main dependencies interactively? (yes/no) [yes]
    no
Would you like to define your development dependencies interactively? (yes/no)
    [yes] no

(...)
```

The `poetry init` command collects the necessary information to generate a `pyproject.toml` file by asking you questions interactively. It gives you recommendations with sensible defaults for most of the configurations that you need to set up, and you can press Enter to accept them. When you don't declare any dependencies, the `pyproject.toml` file that Poetry creates looks something like this:

```
[tool.poetry]
name = "rp-hello"
version = "0.1.0"
description = "My Grand Project"
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"

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

This content resembles the examples that you went through in the previous sections.

Now you can use all the commands that Poetry offers. With a `pyproject.toml` file present, you can now run your script from an isolated virtual environment:

```
$ poetry run python hello.py
Creating virtualenv rp-hello-aVGSp4kH-py3.12 in
   /Users/Philipp/Library/Caches/pypoetry/virtualenvs
Hello, World!
```

Because Poetry didn't find any virtual environments to use, it created a new one before executing your script. After doing this, it displays your `Hello, World!` message without any errors. That means you now have a working Poetry project.

6.2 Importing dependencies to Poetry

Sometimes you have a project that already comes with its own requirements file, which lists all the external dependencies. Take a look at the `requirements.txt` file of this Python web scraper project:

```
beautifulsoup4==4.9.3
certifi==2020.12.5
chardet==4.0.0
idna==2.10
requests==2.25.1
soupsieve==2.2.1
urllib3==1.26.4
```

This project requires seven third-party packages to run properly. Granted, only the Requests and Beautiful Soup libraries are directly referenced in the source code, while the rest are pulled in as their transitive dependencies.

Poetry doesn't deal with such requirements files out of the box, as it keeps your project dependencies elsewhere. However, once you've created a Poetry project with `poetry init`, you can quickly import an existing `requirements.txt` file into the project by using this command:

```
$ poetry add $(cat requirements.txt)
Creating virtualenv web-scraping-bs4-MsO9mbrq-py3.12 in
   /Users/Philipp/Library/Caches/pypoetry/virtualenvs

Updating dependencies
Resolving dependencies... (0.9s)

Package operations: 7 installs, 0 updates, 0 removals

  - Installing certifi (2020.12.5)
  - Installing chardet (4.0.0)
  - Installing idna (2.10)
  - Installing soupsieve (2.2.1)
  - Installing urllib3 (1.26.4)
  - Installing beautifulsoup4 (4.9.3)
  - Installing requests (2.25.1)

Writing lock file
```

The `cat` utility reads the specified file and writes its content to the standard output stream. In this case, you pass that content to the `poetry add` command with the help of the shell's command substitution syntax. That, in turn, installs each dependency listed in the `requirements.txt` file into your Poetry project.

When a requirements file is straightforward like this, using `poetry add` and `cat` can save you some manual work. However, this isn't ideal because all dependencies end up being listed in your `pyproject.toml` file:

```
[tool.poetry]
name = "web-scraping-bs4"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
beautifulsoup4 = "4.9.3"
certifi = "2020.12.5"
chardet = "4.0.0"
idna = "2.10"
requests = "2.25.1"
soupsieve = "2.2.1"
urllib3 = "1.26.4"

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

Normally, you only want to list those packages that your project *directly* depends on. Apart from that, you should generally avoid using exact version numbers, as it can lead to conflicts with existing packages or prevent you from getting the latest features and security updates. Instead, you should specify a version range that allows for some flexibility while still adhering to semantic versioning principles.

In a perfect world, your `pyproject.toml` file should contain only these two dependencies with slightly more permissive version specifiers:

```
[tool.poetry]
name = "web-scraping-bs4"
version = "0.1.0"
description = ""
authors = ["Philipp Acsany <philipp@realpython.com>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.12"
beautifulsoup4 = "^4.9.3"
requests = "^2.25.1"

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

Poetry will resolve the remaining dependencies and lock them in the corresponding `poetry.lock` file.

Now, when you show your project's dependencies as a tree, you'll know exactly which of them are used by your project directly and which are their transitive dependencies:

```
$ poetry show --tree
beautifulsoup4 4.12.3 Screen-scraping library
`-- soupsieve >1.2
requests 2.31.0 Python HTTP for Humans.
|-- certifi >=2017.4.17
|-- charset-normalizer >=2,<4
|-- idna >=2.5,<4
`-- urllib3 >=1.21.1,<3
```

Okay. You know how to import dependencies into a Poetry project. What about exporting them the other way around?

6.3 Exporting dependencies from Poetry

In some situations, you must have a `requirements.txt` file. For example, you may want to host your Django project on Heroku, which expects the `requirements.txt` file to determine what dependencies to install. Poetry gives you a few options to go about creating this file.

6.3.1 Using `pip freeze`

As long as your project's virtual environment is up-to-date, you can pin your dependency versions using the traditional `pip freeze` command:

```
$ poetry run python -m pip freeze > requirements.txt
```

To execute the command within the context of your project's virtual environment managed by Poetry, you must use `poetry run`. Alternatively, you could've activated the virtual environment in an interactive shell and then run the same command from there. Either way, you must first ensure that all the required dependencies are installed in their expected versions by, for example, issuing the `poetry install` command.

```
# by poetry-plugin-export
poetry export --output requirements.txt --without-hashes
```

6.3.2 Using Poetry's export plugin

We can also use Poetry's export plugin to export the `requirements.txt` file.

```
poetry export --output requirements.txt --without-hashes
```

It essentially lets you export dependencies from `poetry.lock` to various file formats, including `requirements.txt`, which is the default.

If we used the `env4pkg_proj.yml` file (discussed at The `env4pkg_proj.yml` file) to create a conda env, this plugin has been installed already. If not, there are several ways to install it:

- `poetry self add poetry-plugin-export`
- `pipx inject poetry poetry-plugin-export`
- `python -m pip install poetry-plugin-export`

Once the plugin is installed, you can use the following command to export dependencies from your Poetry-managed project to a `requirements.txt` file:

```
$ poetry export --output requirements.txt
```

The resulting file includes hashes and environment markers by default, meaning that you can work with very strict requirements that resemble the content of your `poetry.lock` file. As a matter of fact, the export plugin *needs* the lock file in order to generate the requirements file. If that file doesn't exist, then Poetry will create it after resolving and locking your dependencies.

Even though the plugin looks at the lock file to get a big picture of your project's dependencies, it'll only export dependencies from the implicit **main group** in your `pyproject.toml` file. If you'd like to include additional dependencies from **dependency groups**, including the optional and non-optional groups, then use the `--with` parameter followed by comma-separated names of those groups:

```
$ poetry export --output requirements.txt --with dev,test
```

Conversely, to generate a requirements file with dependencies belonging to only a few of your dependency groups while excluding the implicit `main` group, use the `--only` option:

```
$ poetry export --output requirements-dev.txt --only development
```

In all cases, the **extras** with optional dependencies won't be exported. To include them as well, you must either list them explicitly by repeating the `--extras` parameter or enable them all at once with the `--all-extras` flag.

You might have noticed that the export plugin shares a few options with the `poetry install` command that you saw before. That's no coincidence. To reveal the plugin's available options and their descriptions, run `poetry export --help` in your terminal.

7 Using additional envs in Poetry

Sometimes we need to test a project using different versions of Python in addition to the one coming with the conda env. In this case, we need to create new virtual envs using `venv`. For more info, see py-pkg-6-mg-a-few-tools.

8 The workflow for developing a project from scratch

The workflow for developing a project from scratch includes the following steps:

- Creating a new env using conda and install Poetry in it. Activate the env.
- Navigating to the parent folder of the project folder.
- Creating a **new project** using Poetry.
- Adding Poetry to an **existing project.**
- Configuring the project through **`pyproject.toml`**.
- Pinning the project's **dependency versions**.
- Installing dependencies from a **`poetry.lock`** file.
- Running basic Poetry commands using the **Poetry CLI**.

Step 1. Creating a new env using conda.

Step 2. Creating a new project using Poetry.

```
poetry new my_project --src
```

We can check the project folder using `cd my_project` and then `tree`.

Step 3. Installing the project in editable mode.

```
poetry install
```

Step 4. Editing the project code to make a minimum system

Step 5. Adding dependencies using

```
poetry add xxx
```

Step 6. Running the project

```
poetry run python my_project/your_script.py
```

9 Installation of Poetry

Poetry is distributed as a Python package itself, which means that you can install it into a virtual environment using `pip`, just like any other external package:

```
pip install poetry
```

This is fine if you just want to quickly try it out. However, the official documentation strongly advises *against* installing Poetry into your project's virtual environment, which the tool must manage. Because Poetry depends on several external packages itself, you'd run the risk of a **dependency conflict** between one of your project's dependencies and those required by Poetry. In turn, this could cause Poetry or your code to malfunction.

In practice, you always want to keep Poetry separate from any virtual environment that you create for your Python projects. You also want to install Poetry **system-wide** to access it as a stand-alone application regardless of the specific virtual environment or Python version that you're currently working in.

There are several ways to get Poetry running on your computer, including:

1. A tool called `pipx`
2. The official installer
3. Manual installation
4. Pre-built system packages

In most cases, the *recommended way to install Poetry* is with the help of `pipx`, which takes care of creating and maintaining isolated virtual environments for command-line Python applications. After installing `pipx`, you can install Poetry by issuing the following command in your terminal window:

```
pipx install poetry
```

While this command looks very similar to the one you saw previously, it'll install Poetry into a dedicated virtual environment that won't be shared with other Python packages.

Crucially, `pipx` will also set an alias to the `poetry` executable in your shell so that you can invoke Poetry from any directory without manually activating the associated virtual environment:

```
poetry --version Poetry (version 1.7.1)
```

When you type `poetry` in your terminal or PowerShell console, you'll always refer to the executable script installed in its isolated virtual environment. In fact, you'll often run the `poetry` command from within an active virtual environment of your project, and Poetry will correctly pick it up!

If, for some reason, you can't or don't want to use `pipx`, then the next best thing you can do is take advantage of the **official installer**, which you can download and execute with a command like this:

- Windows

  ```
  PS> (Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing
  ).Content | python -
  ```

  On Windows, you can use the Invoke-WebRequest PowerShell command along with the -UseBasicParsing option to download the content of a requested URL and print it onto the standard output stream (stdout). With the pipe operator (|), you can intercept the downloaded install-poetry.py script and pipe it into your Python interpreter from the standard input stream (stdin).

- Linux + macOS

  ```
  $ curl -sSL https://install.python-poetry.org | python3-
  ```

  The `curl` command downloads the content of a requested URL and prints it onto the standard output stream (stdout). When you use the pipe operator (|), you define a Unix pipeline, which intercepts the downloaded install-poetry.py script and pipes it into your Python interpreter from the standard input stream (stdin).

Underneath the `install.python-poetry.org` URL is a cross-platform Python script that, more or less, replicates what `pipx` does, but in a slightly different way.

If you're interested in the technical details, then take a peek at the underlying source code of the installation script. By following its steps **manually**, you'll have a chance to tweak the installation process if you ever need something specific for your setup. This can be particularly useful when you prepare a continuous integration environment.

Finally, some operating systems may bundle Poetry as a **native package**. For example, if you're using a Debian-based system like Ubuntu, then you might be able to install Poetry with `apt`, like so:

```
sudo apt install python3-poetry
```

Bear in mind that the Poetry version packaged like that might not be the latest one. Moreover, the native package may bring hundreds of megabytes of extra dependencies, such as another Python interpreter, which would be completely unnecessary if you installed Poetry using one of the earlier methods.

Once the Poetry installation is complete, you can confirm that it's working correctly by typing `poetry --version` at your command prompt. This should print out your current version of Poetry.

Note: Depending on how you installed Poetry, you can try one of these commands to upgrade it:

- `python3-m pip install -upgrade poetry`
- `pipx upgrade poetry`
- `poetry self update`
- `sudo apt install --only-upgrade python3-poetry`

All but the last one should generally ensure that you have the most recent version of Poetry installed on your computer.