Basic usage

- Project setup
  - Setting a Python Version
  - Initialising a pre-existing project
  - Operating modes
  - Specifying dependencies
- Using your virtual environment
  - Using `poetry run`
  - Activating the virtual environment

- Version constraints
- Installing dependencies
  - Installing without `poetry.lock`
  - Installing with `poetry.lock`
  - Committing your `poetry.lock` file to version control
  - Installing dependencies only
- Updating dependencies to their latest versions

For the basic usage introduction we will be installing `pendulum`, a datetime library. If you have not yet installed Poetry, refer to the Introduction chapter.

Project setup

First, let's create our new project, let's call it `poetry-demo`:

```
poetry new poetry-demo
```

This will create the `poetry-demo` directory with the following content:

```
poetry-demo
├── pyproject.toml
├── README.md
├── src
│   └── poetry_demo
│       └── __init__.py
└── tests
    └── __init__.py
```

The `pyproject.toml` file is what is the most important here. This will orchestrate your project and its dependencies. For now, it looks like this:

```toml
[project]
name = "poetry-demo"
version = "0.1.0"
description = ""
authors = [
    {name = "Sébastien Eustace", email = "sebastien(
]
readme = "README.md"
requires-python = ">=3.9"
dependencies = [
]

[build-system]
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"
```

Poetry assumes your package contains a package with the same name as `project.name` located in the root of your project. If this is not the case, populate `tool.poetry.packages` to specify your packages and their locations.

Similarly, the traditional `MANIFEST.in` file is replaced by the `project.readme`, `tool.poetry.include`, and `tool.poetry.exclude` sections. `tool.poetry.exclude` is additionally implicitly populated by your `.gitignore`. For full documentation on the project format, see the pyproject section of the documentation.

Setting a Python Version

> Unlike with other packages, Poetry will not automatically install a python interpreter for you. If you want to run Python files in your package like a script or application, you must *bring your own* python interpreter to run them.

Poetry will require you to explicitly specify what versions of Python you intend to support, and its universal locking will guarantee that your project is installable (and all dependencies claim support for) all supported Python versions. Again, it's important to remember that – unlike other dependencies – setting a Python version is merely specifying which versions of Python you intend to support.

For example, in this `pyproject.toml` file:

```toml
[project]
requires-python = ">=3.9"
```

we are allowing any version of Python 3 that is greater or equal than `3.9.0`.

When you run `poetry install`, you must have access to some version of a Python interpreter that satisfies this constraint available on your system. Poetry will not install a Python interpreter for you.

Initialising a pre-existing project

Instead of creating a new project, Poetry can be used to 'initialize' a pre-populated directory. To interactively create a `pyproject.toml` file in directory `pre-existing-project`:

```
cd pre-existing-project
poetry init
```

Operating modes

Poetry can be operated in two different modes. The default mode is the **package mode**, which is the right mode if you want to package your project into an sdist or a wheel and perhaps publish it to a package index. In this mode, some metadata such as `name` and

`version`, which are required for packaging, are mandatory. Further, the project itself will be installed in editable mode when running `poetry install`.

If you want to use Poetry only for dependency management but not for packaging, you can use the **non-package mode**:

```
[tool.poetry]
package-mode = false
```

In this mode, metadata such as `name` and `version` are optional. Therefore, it is not possible to build a distribution or publish the project to a package index. Further, when running `poetry install`, Poetry does not try to install the project itself, but only its dependencies (same as `poetry install --no-root`).

> In the pyproject section you can see which fields are required in package mode.

Specifying dependencies

If you want to add dependencies to your project, you can specify them in the `project` section.

```
[project]
# ...
dependencies = [
    "pendulum (>=2.1,<3.0)"
]
```

As you can see, it takes a mapping of **package names** and **version constraints**.

Poetry uses this information to search for the right set of files in package "repositories" that you register in the `tool.poetry.source` section, or on PyPI by default.

Also, instead of modifying the `pyproject.toml` file by hand, you can use the `add` command.

```
$ poetry add pendulum
```

It will automatically find a suitable version constraint **and install** the package and sub-dependencies.

Poetry supports a rich dependency specification syntax, including caret, tilde, wildcard, inequality and multiple constraints requirements.

Using your virtual environment

By default, Poetry creates a virtual environment in `{cache-dir}/virtualenvs`. You can change the `cache-dir` value by editing the Poetry configuration. Additionally, you can use the `virtualenvs.in-project` configuration variable to create virtual environments within your project directory.

There are several ways to run commands within this virtual environment.

External virtual environment management

Poetry will detect and respect an existing virtual environment that has been externally activated. This is a powerful mechanism that is intended to be an alternative to Poetry's built-in, simplified environment management.

To take advantage of this, simply activate a virtual environment using your preferred method or tooling, before running any Poetry commands that expect to manipulate an environment.

Using `poetry run`

To run your script simply use `poetry run python your_script.py`. Likewise if you have command line tools such as `pytest` or `black` you can run them using `poetry run pytest`.

If managing your own virtual environment externally, you do not need to use `poetry run` since you will, presumably, already have activated that virtual environment and made available the correct python instance. For example, these commands should output the same python path:

```
conda activate your_env_name
which python
poetry run which python
eval "$(poetry env activate)"
which python
```

Activating the virtual environment

See Activating the virtual environment.

Version constraints

In our example, we are requesting the `pendulum` package with the version constraint `>=2.1.0 <3.0.0`. This means any version greater or equal to 2.1.0 and less than 3.0.0.

Please read Dependency specification for more in-depth information on versions, how versions relate to each other, and on the different ways you can specify dependencies.

How does Poetry download the right files?

When you specify a dependency in `pyproject.toml`, Poetry first takes the name of the package that you have requested and searches for it in any repository you have registered using the `repositories` key. If you have not registered any extra repositories, or it does not find a package with that name in the repositories you have specified, it falls back to PyPI.

When Poetry finds the right package, it then attempts to find the best match for the version constraint you have specified.

Installing dependencies

To install the defined dependencies for your project, just run the `install` command.

```
poetry install
```

When you run this command, one of two things may happen:

Installing without `poetry.lock`

If you have never run the command before and there is also no `poetry.lock` file present, Poetry simply resolves all dependencies listed in your `pyproject.toml` file and downloads the latest version of their files.

When Poetry has finished installing, it writes all the packages and their exact versions that it downloaded to the `poetry.lock` file, locking the project to those specific versions. You should commit the `poetry.lock` file to your project repo so that all people working on the project are locked to the same versions of dependencies (more below).

Installing with `poetry.lock`

This brings us to the second scenario. If there is already a `poetry.lock` file as well as a `pyproject.toml` file when you run `poetry install`, it means either you ran the `install` command

before, or someone else on the project ran the `install` command and committed the `poetry.lock` file to the project (which is good).

Either way, running `install` when a `poetry.lock` file is present resolves and installs all dependencies that you listed in `pyproject.toml`, but Poetry uses the exact versions listed in `poetry.lock` to ensure that the package versions are consistent for everyone working on your project. As a result you will have all dependencies requested by your `pyproject.toml` file, but they may not all be at the very latest available versions (some dependencies listed in the `poetry.lock` file may have released newer versions since the file was created). This is by design, it ensures that your project does not break because of unexpected changes in dependencies.

Committing your `poetry.lock` file to version control

As an application developer

Application developers commit `poetry.lock` to get more reproducible builds.

Committing this file to VC is important because it will cause anyone who sets up the project to use the exact same versions of the

dependencies that you are using. Your CI server, production machines, other developers in your team, everything and everyone runs on the same dependencies, which mitigates the potential for bugs affecting only some parts of the deployments. Even if you develop alone, in six months when reinstalling the project you can feel confident the dependencies installed are still working even if your dependencies released many new versions since then. (See note below about using the update command.)

> If you have added the recommended `[build-system]` section to your project's pyproject.toml then you *can* successfully install your project and its dependencies into a virtual environment using a command like `pip install -e .`. However, pip will not use the lock file to determine dependency versions as the poetry-core build system is intended for library developers (see next section).

As a library developer

Library developers have more to consider. Your users are application developers, and your library will run in a Python environment you don't control.

The application ignores your library's lock file. It can use whatever dependency version meets the constraints in your `pyproject.toml`. The application will probably use the latest compatible dependency version. If your library's `poetry.lock` falls behind some new dependency version that breaks things for your users, you're likely to be the last to find out about it.

A simple way to avoid such a scenario is to omit the `poetry.lock` file. However, by doing so, you sacrifice reproducibility and performance to a certain extent. Without a lockfile, it can be difficult to find the reason for failing tests, because in addition to obvious code changes an unnoticed library update might be the culprit. Further, Poetry will have to lock before installing a dependency if `poetry.lock` has been omitted. Depending on the number of dependencies, locking may take a significant amount of time.

If you do not want to give up the reproducibility and performance benefits, consider a regular refresh of `poetry.lock` to stay up-to-date and reduce the risk of sudden breakage for users.

Installing dependencies only

The current project is installed in editable mode by default.

If you want to install the dependencies only, run the `install` command with the `--no-root` flag:

```
poetry install --no-root
```

Updating dependencies to their latest versions

As mentioned above, the `poetry.lock` file prevents you from automatically getting the latest versions of your dependencies. To update to the latest versions, use the `update` command. This will fetch the latest matching versions (according to your `pyproject.toml` file) and update the lock file with the new versions. (This is equivalent to deleting the `poetry.lock` file and running `install` again.)

> Poetry will display a **Warning** when executing an install command if `poetry.lock` and `pyproject.toml` are not synchronized.

Python packaging and dependency management made easy.


DOCUMENTATION

Introduction

Basic usage

Managing dependencies

Libraries

Commands

Configuration

Repositories

Managing environments

Dependency specification

Plugins

The pyproject.toml file

Contributing to Poetry

Community

FAQ

pre-commit hooks

Building extension modules

GITHUB

Project

Issues

Discussions

OTHER PROJECTS

poetry-core

install.python-poetry.org

Bundle plugin

Export plugin
