Dependency management with Poetry

Poetry is a tool for dependency management and packaging in Python. It helps you declare, manage, and install dependencies, ensuring that your project remains isolated and that dependency versions are compatible. Here’s a guide to using Poetry for dependency management:

## 1. Installation

First, you need to install Poetry. You can do this by running:

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

Or, if you prefer using pipx:

pipx install poetry

## 2. Configure Poetry to create the env in the local folder

Navigate the To create a new pipx install poetry

## 2. Creating a New Project

To create a new project with Poetry, navigate to your desired directory and run:

poetry new my\_project

This will create a new directory named my\_project with a basic structure and a pyproject.toml file.

## 3. Existing Project

For an existing project, you can initialize Poetry by running:

poetry init

This command will guide you through a series of prompts to set up your pyproject.toml file.

## 4. Configuring python to use

poetry env use /c/Users/ramak/AppData/Local/Programs/Python/Python311/python.exe

## 4. Adding Dependencies

You can add dependencies to your project using the add command. For example, to add requests:

poetry add requests

OR

poetry add requests --path C:\RSD\Poetry\_Trial\poet\_01

To add development dependencies (e.g., for testing or linting), use the --group flag:

poetry add --group dev pytest

## 5. List the added packages

poetry show

OR

poetry show -v

## 5. Installing Dependencies

To install the dependencies listed in your pyproject.toml file, simply run:

poetry install

## 6. Using Virtual Environments

Poetry automatically creates and manages a virtual environment for your project. You can activate the virtual environment using:

poetry shell

To run a command within the virtual environment without activating it, use:

poetry run python your\_script.py

## 7. Updating Dependencies

To update all dependencies to their latest versions according to the constraints in your pyproject.toml file:

poetry update

To update a specific dependency:

poetry update requests

## 8. Publishing a Package

If you want to publish your project as a package, you can do so by first configuring your pyproject.toml file with the necessary metadata and then running:

poetry publish --build

## 9. Handling Multiple Environments

Poetry supports multiple dependency groups, which can be useful for managing dependencies across different environments (e.g., development, testing, production). For example:

[tool.poetry.group.dev.dependencies]  
pytest = "^6.0"

[tool.poetry.group.test.dependencies]  
coverage = "^5.0"

To install dependencies for a specific group, use:

poetry install –with dev

poetry install –without dev –with test

## 10. Lock File

Poetry uses a poetry.lock file to ensure consistent installs. This file locks the exact versions of the dependencies. Make sure to commit this file to your version control system.

Example pyproject.toml

Here's an example of what your pyproject.toml might look like:

[tool.poetry]  
name = "my\_project"  
version = "0.1.0"  
description = ""  
authors = ["Your Name <you@example.com>"]

[tool.poetry.dependencies]  
python = "^3.8"  
requests = "^2.25.1"

[tool.poetry.group.dev.dependencies]  
pytest = "^6.2"

[build-system]  
requires = ["poetry-core>=1.0.0"]  
build-backend = "poetry.core.masonry.api"

By using Poetry, you can simplify dependency management, ensure consistency, and improve the overall maintainability of your Python projects.

#### Types of Version Constraints

1. Exact Version:

requests = "2.25.1"

Only version 2.25.1 will be considered valid.

1. Version Range:

requests = ">=2.0,<3.0"

Any version of requests starting from 2.0 up to, but not including, 3.0 will be considered valid.

1. Caret (^) Constraints:

requests = "^2.25.1”

Allows updates that do not modify the left-most non-zero digit. This is a common way to specify that you want bug fixes and new features, but not breaking changes. For example, ^2.25.1 includes versions 2.25.1 up to but not including 3.0.0.

1. Tilde (~) Constraints:

requests = "~2.25"

Allows updates that are compatible with the specified version, generally minor and patch updates. For example, ~2.25 includes versions 2.25.0 up to but not including 2.26.0.

1. Wildcards:

requests = "2.\*"

Allows any version where the major version is 2, i.e., 2.0.0, 2.1.0, etc.

A more detailed explanation about the poetry.lock file and how it fits into the dependency management process with Poetry.

# Understanding the poetry.lock File

The poetry.lock file is crucial for ensuring reproducible builds and consistent environments. Here’s a breakdown of its purpose and how it works:

## Purpose of the poetry.lock File:

Locking Dependencies: It records the exact versions of dependencies (and their dependencies) that were installed when you ran poetry install or poetry update. This ensures that every time you or someone else installs the project, the same versions are used, preventing the "it works on my machine" problem.

Consistency: By locking dependencies, you ensure that your project behaves the same way on different machines and environments.

## How It Works:

Initial Creation: When you first add a dependency using poetry add <dependency>, Poetry resolves the dependency tree and records the exact versions in poetry.lock.

Installing Dependencies: When you run poetry install, Poetry reads from poetry.lock to install the exact versions specified. If poetry.lock doesn’t exist, Poetry generates it.

Updating Dependencies: When you run poetry update, Poetry updates the dependencies in poetry.lock to their latest compatible versions based on your pyproject.toml constraints and updates the lock file accordingly.

### Example Scenario:

Suppose you add a dependency requests to your project. Your pyproject.toml might specify:

[tool.poetry.dependencies]  
requests = "^2.25.1"

When you run poetry add requests, Poetry resolves and locks the exact version (e.g., requests 2.25.1) along with all transitive dependencies (dependencies of requests).

Example poetry.lock File

Here is a simplified example of what a poetry.lock file might look like after adding requests:

[[package]]  
name = "requests"  
version = "2.25.1"  
description = "Python HTTP for Humans."  
category = "main"  
optional = false  
python-versions = ">=2.7, !=3.0.\*, !=3.1.\*, !=3.2.\*, !=3.3.\*, !=3.4.\*, <4"

[metadata]  
lock-version = "1.1"  
python-versions = "^3.8"  
content-hash = "abc123..."

[metadata.files]  
requests = [  
 {file = "requests-2.25.1.tar.gz", hash = "sha256:abc123..."},  
 {file = "requests-2.25.1-py2.py3-none-any.whl", hash = "sha256:def456..."}  
]

## Workflow with poetry.lock

### Adding a Dependency:

poetry add requests

This updates both pyproject.toml and poetry.lock.

### Installing Dependencies:

poetry install

This installs the exact versions from poetry.lock.

### Updating Dependencies:

poetry update

This updates to the latest compatible versions and updates poetry.lock.

### Importance of Committing poetry.lock to Version Control

**Reproducibility**: Ensures that all contributors and environments use the same dependency versions.

**Traceability**: Makes it easier to track changes in dependencies over time.

**Debugging**: Helps in debugging issues related to specific dependency versions.

By committing both pyproject.toml and poetry.lock to your version control system, you ensure that your project remains stable and that others can recreate your environment exactly.