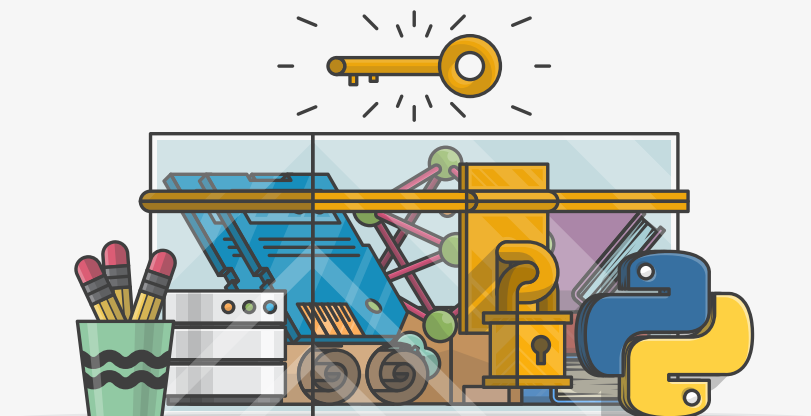




**Keep reading Real Python by  
creating a free account or  
signing in:**



[Continue »](#)

Already have an account? [Sign-In](#)

- Use the pyproject.toml File
- [Work With Python Poetry](#)
  - Use Poetry's Virtual Environment
  - Declare Your Dependencies
  - Install a Package With Poetry
- [Handle poetry.lock](#)

- [Pin Dependencies in poetry.lock](#)
- [Install Dependencies From poetry.lock](#)
- [Update Dependencies](#)
- [Distinguish Between pyproject.toml and poetry.lock](#)
- [Add Poetry to an Existing Project](#)
  - [Add pyproject.toml to a Scripts Folder](#)
  - [Use an Existing requirements.txt File](#)
  - [Create requirements.txt From poetry.lock](#)
- [Command Reference](#)
- [Conclusion](#)

platform.sh

The cloud hosting platform built to deploy  
tens of websites and web apps. Fast.



Start Your Free Trial Now

Remove ads

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, a package might not work as it did before the update. A dependency manager like Python **Poetry** helps you specify, install, and resolve external packages in your projects. This way, you can be sure that you always work with the right dependency version on every machine.

### In this tutorial, you'll learn how to:

- Start a **new** Poetry project
- Add Poetry to an **existing** project
- Use the `pyproject.toml` file
- Pin **dependencies**
- Install dependencies with `poetry.lock`
- Execute basic **Poetry CLI** commands

Using [Poetry](#) will help you start new projects, maintain existing ones, and master **dependency management**. You'll be prepared to work with `pyproject.toml` files, which will be the [standard](#) for defining build requirements in Python projects.

To complete this tutorial and get the most out of it, you should have

a basic understanding of [virtual environments](#), [modules and packages](#), and [pip](#).

While this tutorial focuses on dependency management, Poetry can also help you with [building and packaging](#) projects. If you want to share your work, then you can even [publish](#) your Poetry project to the [Python Packaging Index \(PyPI\)](#).

**Free Bonus:** Click here to get access to a free 5-day class that shows you how to avoid common dependency management issues with tools like Pip, PyPI, Virtualenv, and requirements files.

## Take Care of Prerequisites

Before diving into the nitty-gritty of Python Poetry, you'll take care of some prerequisites. First, you'll read a short overview of terminology you'll encounter in this tutorial. Next, you'll install Poetry itself.



Master **Real-World Python Skills**  
With a Community of Experts

Level Up With Unlimited Access to Our Vast Library  
of Python Tutorials and Video Lessons

[Watch Now »](#)

 Remove ads

## Relevant Terminology

If you've ever used an `import` statement in one of your Python scripts, then you've worked with **modules**. Some of these modules might have been Python files you wrote on your own. Others could have been **built-in** modules, like [datetime](#). However, sometimes what Python provides isn't enough. That's when you might turn to external, packaged modules. When your Python code relies on external modules, you can say that these **packages** are **dependencies** of your project.


You can find packages that aren't part of the [Python standard library](#) in [PyPI](#). Before seeing how this works, you need to install Poetry on


your system.

## Python Poetry Installation

To use Poetry in your command line, you should install it system-wide. If you just want to try it out, then you can install it into a virtual environment using [pip](#). But you should try this method with caution because Poetry will install its own dependencies, which can conflict with other packages you're using in your project.

The recommended way to [install Poetry](#) is by using the official `install-poetry` script. You can either download and run this [Python file](#) manually or select your operating system below to use the appropriate command:

 Windows

 Linux + macOS

Windows PowerShell

```
PS C:\> (Invoke-WebRequest -Uri https://raw.githubusercontent.com
```

If you're on Windows, then you can use the `Invoke-WebRequest` cmdlet with the `-UseBasicParsing` option to download the content of a requested URL to the **standard output stream (stdout)**. With the pipe character (`|`), you're handing over the output to the **standard input stream (stdin)** of python. In this case, you're *piping* the content of `install-poetry.py` to your Python interpreter.

**Note:** Some users [report errors on Windows 10](#) when they use the PowerShell command.

In the output, you should see a message that the installation is complete. You can run `poetry --version` in your terminal to see if poetry works. This command will display your current Poetry version. If you want to update Poetry, then you can run `poetry self update`.

## Get Started With Python Poetry

With Poetry installed, it's time to see how Poetry works. In this section, you'll learn how to start a fresh Poetry project and how to add Poetry to an existing project. You'll also see the project structure and inspect the `pyproject.toml` file.

## Create a New Poetry Project

You can create a new Poetry project by using the `new` command and a project name as an argument. In this tutorial, the project is called `rp-poetry`. Create the project, and then move into the newly created directory:

### Shell

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

By running `poetry new rp-poetry`, you create a new folder named `rp-poetry/`. When you look inside the folder, you'll see a structure:

```
rp-poetry/
├── rp_poetry/
│   └── __init__.py
├── tests/
│   ├── __init__.py
│   └── test_rp_poetry.py
├── README.rst
└── pyproject.toml
```

Poetry automatically normalizes package names for you. It transformed the dash (-) in your project name into an underscore (\_) in the folder name of `rp_poetry/`. Otherwise, the name wouldn't be allowed in Python, so you couldn't import it as a module. To have more control over creating the package name, you can use the `--name` option to name it differently than the project folder:

### Shell

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

If you prefer to store your source code in an additional `src/` parent folder, then Poetry lets you stick to that convention by using the `--src` flag:

#### Shell

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

By adding the `--src` flag, you've created a folder named `src/`, which contains your `rp_poetry/` directory:

```
rp-poetry/
├── src/
│   └── rp_poetry/
│       └── __init__.py
├── tests/
│   ├── __init__.py
│   └── test_rp_poetry.py
├── README.rst
└── pyproject.toml
```

When creating a new Poetry project, you'll receive a basic folder structure right away.



Your **Guided Tour** Through the **Python 3.9 Interpreter** »

 Remove ads

## Inspect the Project Structure

The `rp_poetry/` subfolder itself isn't very spectacular yet. Inside this directory, you'll find an `__init__.py` file with your package's version:

Python

```
# rp_poetry/__init__.py

__version__ = "0.1.0"
```

When you hop over to the `tests/` folder and open `test_rp_poetry.py`, you'll notice that `rp_poetry` is already importable:

```
Python

# tests/test_rp_poetry.py

from rp_poetry import __version__

def test_version():
    assert __version__ == "0.1.0"
```

Poetry has also added a first test to the project. The `test_version()` function checks whether the `__version__` variable of `rp_poetry/__init__.py` contains the expected version. However, the `__init__.py` file isn't the only place where you define the version of your package. The other location is the `pyproject.toml` file.

## Use the `pyproject.toml` File

One of the most important files for working with Poetry is the `pyproject.toml` file. This file isn't an invention of Poetry. It's a **configuration file** standard that was defined in PEP 518:

This PEP specifies how Python software packages should specify what build dependencies they have in order to execute their chosen build system. As part of this specification, a new configuration file is introduced for software packages to use to specify their build dependencies (with the expectation that the same configuration file will be used for future configuration details). ([Source](#))

The authors considered a few file formats for the “new configuration file” mentioned in the quote above. In the end, they decided on the

**TOML** format, which stands for [Tom's Obvious Minimal Language](#). In their opinion, TOML is flexible enough, with better readability and less complexity than the other options, which are YAML, JSON, CFG, or INI. To see how TOML looks, open the `pyproject.toml` file:

```
TOML

1  # pyproject.toml
2
3  [tool.poetry]
4  name = "rp-poetry"
5  version = "0.1.0"
6  description = ""
7  authors = ["Philipp <philipp@realpython.com>"]
8
9  [tool.poetry.dependencies]
10 python = "^3.9"
11
12 [tool.poetry.dev-dependencies]
13 pytest = "^5.2"
14
15 [build-system]
16 requires = ["poetry-core>=1.0.0"]
17 build-backend = "poetry.core.masonry.api"
```

You can see four sections in the `pyproject.toml` file. These sections are called **tables**. They contain instructions that tools like Poetry recognize and use for **dependency management** or **build routines**.

If a table name is tool-specific, it must be prefixed with `tool`. By using such a **subtable**, you can add instructions for different tools in your project. In this case, there is only `tool.poetry`. But you might see examples like `[tool.pytest.ini_options]` for [pytest](#) in other projects.

In the `[tool.poetry]` subtable on line 3 above, you can store general information about your Poetry project. Your available keys are [defined by Poetry](#). While some keys are optional, there are four that you must specify:

1. **name**: the name of your package
2. **version**: the version of your package, ideally following



## semantic versioning

3. **description:** a short description of your package
4. **authors:** a list of authors, in the format name <email>

The subtables `[tool.poetry.dependencies]` on line 9 and `[tool.poetry.dev-dependencies]` on line 12 are essential for your dependency management. You'll learn more about these subtables in the next section when you add dependencies to your Poetry project. For now, the important thing is to recognize that there *is* differentiation between package dependencies and development dependencies.

The last table of the `pyproject.toml` file is `[build-system]` on line 15. This table defines data that Poetry and other build tools can work with, but as it's not tool-specific, it doesn't have a prefix. Poetry created the `pyproject.toml` file with two keys in place:

1. **requires:** a list of dependencies that are required to build the package, making this key mandatory
2. **build-backend:** the Python object used to perform the build process

If you want to learn a bit more about this section of the `pyproject.toml` file, then you can find out more by reading about [source trees in PEP 517](#).

When you start a new project with Poetry, this is the `pyproject.toml` file you start with. Over time, you'll add configuration details about your package and the tools you're using. As your Python project grows, your `pyproject.toml` file will grow with it. This is particularly true for the subtables `[tool.poetry.dependencies]` and `[tool.poetry.dev-dependencies]`. In the next section, you'll find out how to expand these subtables.



[Online Python Training for Teams »](#)

Remove ads

# Work With Python Poetry

Once you've set up a Poetry project, the real work can begin. You can start coding once Poetry is in place. Along the way, you'll find out how Poetry provides you with a **virtual environment** and takes care of your dependencies.

## Use Poetry's Virtual Environment

When you start a new Python project, it's good practice to create a [virtual environment](#). Otherwise, you may confuse different dependencies from different projects. Working with virtual environments is one of Poetry's core features, and it'll never interfere with your global Python installation.

However, Poetry doesn't create a virtual environment right away when you start a project. You can confirm that Poetry hasn't created a virtual environment by having Poetry list all virtual environments connected to the current project. If you haven't already, `cd` into `rp-poetry/` and then run a command:

Shell

```
$ poetry env list
```

For now, there shouldn't be any output.

Poetry will create a virtual environment along the way when you run certain commands. If you want to have better control over the creation of a virtual environment, then you might decide to tell Poetry explicitly which Python version you want to use for it and go from there:

Shell

```
$ poetry env use python3
```

With this command, you're using the same Python version that you used to install Poetry. Using `python3` works when you have the Python executable in your `PATH`.

**Note:** Alternatively, you can pass an absolute path to a Python executable. It should match the Python version constraint you can find in your `pyproject.toml` file. If it doesn't, then you may run into trouble because you're working with a different Python version than the version your project requires. Code that works in your environment could be buggy on another machine.

Even worse, external packages often rely on specific Python versions. Consequently, a user installing your package might get an error because your dependency versions are incompatible with their Python version.

When you run `env use`, you'll see a message:

#### Shell

```
Creating virtualenv rp-poetry-AWdWY-py3.9 in ~/Library/Caches/pypoetry/virtualenvs/rp-poetry-AWdWY-py3.9
Using virtualenv: ~/Library/Caches/pypoetry/virtualenvs/rp-poetry-AWdWY-py3.9
```

As you can see, Poetry constructed a unique name for your project's environment. The name contains the project name and the Python version. The seemingly random string in the middle is a hash of your parent directory. With this unique string in the middle, Poetry can handle multiple projects with the same name and the same Python version on your system. That's important because, by default, Poetry creates all your virtual environments in the same folder.

Without any other configuration, Poetry creates the virtual environments in the `virtualenvs/` folder of Poetry's **cache directory**:

Operating System	Path
macOS	~/Library/Caches/pypoetry
Windows	C:\Users\<username>\AppData\Local\pypoetry\Cache

Operating System	Path
Linux	~/.cache/pypoetry

If you want to change the default cache directory, then you can edit [Poetry's configuration](#). This can be useful when you're already using [virtualenvwrapper](#) or another third-party tool for managing your virtual environments. To see the current configuration, including the configured `cache-dir`, you can run a command:

#### Shell

```
$ poetry config --list
```

Usually, you don't have to change this path. If you want to learn more about interacting with Poetry's virtual environments, then the Poetry documentation contains a chapter about [managing environments](#).

As long as you're inside your project folder, Poetry will use the virtual environment associated with it. If you're ever in doubt, you can check whether the virtual environment is activated by running the `env list` command again:

#### Shell

```
$ poetry env list
```

This will display something like `rp-poetry-AWdWY-py3.9 (Activated)`. With an activated virtual environment, you're ready to start managing some dependencies and see Poetry shine.



[Real Python for Teams »](#)

Remove ads

# Declare Your Dependencies

A key element of Poetry is its handling of your dependencies. Before you get the ball rolling, take a look at the two dependency tables in the `pyproject.toml` file:

## TOML

```
# rp_poetry/pyproject.toml (Excerpt)
```

```
[tool.poetry.dependencies]
```

```
python = "^3.9"
```

```
[tool.poetry.dev-dependencies]
```

```
pytest = "^5.2"
```

There are currently two dependencies declared for your project. One is Python itself. The other is [pytest](#), a widely used testing framework. As you've seen before, your project contains a `tests/` folder and a `test_rp_poetry.py` file. With **pytest** as a dependency, Poetry can run your tests immediately after installation.

**Note:** At the time of writing this tutorial, running `pytest` with Poetry using [Python 3.10](#) doesn't work. Poetry installs a `pytest` version that is incompatible with Python 3.10.

The Poetry developers are [aware of this issue](#), and it will be fixed with the release of Poetry 1.2.

Make sure that you're inside the `rp-poetry/` project folder and run a command:

## Shell

```
$ poetry install
```

With the `install` command, Poetry checks your `pyproject.toml` file for dependencies then resolves and installs them. The resolving part is especially important when you have many dependencies that require different 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.

Besides `pytest` and its requirements, Poetry also installs the project itself. This way, you can import `rp_poetry` into your tests right away:

#### Python

```
# tests/test_rp_poetry.py

from rp_poetry import __version__

def test_version():
    assert __version__ == "0.1.0"
```

With your project's package installed, you can import `rp_poetry` into your tests and check for the `__version__` string. With `pytest` installed, you can use the `poetry run` command to execute the tests:

#### Shell

```
1 $ poetry run pytest
2 ===== test session starts =====
3 platform darwin -- Python 3.9.1, pytest-5.4.3, py-1.10.0, p
4 rootdir: /Users/philipp/Real Python/rp-poetry
5 collected 1 item
6
7 tests/test_rp_poetry.py .
8
9 ===== 1 passed in 0.01s =====
```

Your current test is running successfully, so you can confidently continue coding. However, if you look closely at line 3, something looks a bit odd. It says `pytest-5.4.3`, not `5.2` like stated in the `pyproject.toml` file. Good catch!

To recap, the `pytest` dependency in your `pyproject.toml` file looks like this:

#### TOML

```
# rp_poetry/pyproject.toml (Excerpt)
```

```
[tool.poetry.dev-dependencies]
pytest = "^5.2"
```

The caret (^) in front of 5.2 has a specific meaning, and it's one of the [versioning constraints](#) that Poetry provides. It means that Poetry can install any version that matches the leftmost non-zero digit of the version string. This means that using 5.4.3 is allowed. Version 6.0 wouldn't be allowed.

A symbol like the caret will become important when Poetry tries to resolve the dependency versions. If there are only two requirements, this isn't too hard. The more dependencies you declare, the more complicated it gets. Let's see how Poetry handles this by installing new packages into your project.

## Install a Package With Poetry

You may have used [pip](#) before to install packages that aren't part of the Python standard library. If you run `pip install` with the package name as an argument, `pip` looks for packages on the [Python Package Index](#). You can use Poetry the same way.

If you want to add an external package like [requests](#) to your project, then you can run a command:

Shell

```
$ poetry add requests
```

By running `poetry add requests`, you're adding the latest version of the `requests` library to your project. You can use version constraints like `requests<=2.1` or `requests==2.24` if you want to be more specific. When you don't add any constraints, Poetry will always try to install the latest version of the package.

Sometimes there are packages that you only want to use in your development environment. With `pytest`, you spotted one of them already. Another common library includes a code formatter like [Black](#), a documentation generator like [Sphinx](#), and a static analysis

tool like [Pylint](#), [Flake8](#), [mypy](#), or [coverage.py](#).

To explicitly tell Poetry that a package is a development dependency, you run `poetry add` with the `--dev` option. You can also use a shorthand `-D` option, which is the same as `--dev`:

#### Shell

```
$ poetry add black -D
```

You added `requests` as a project dependency and `black` as a development dependency. Poetry has done a few things for you in the background. For one thing, it added your declared dependencies to the `pyproject.toml` file:

#### TOML

```
# rp_poetry/pyproject.toml (Excerpt)
```

```
[tool.poetry.dependencies]
```

```
python = "^3.9"
```

```
requests = "^2.26.0"
```

```
[tool.poetry.dev-dependencies]
```

```
pytest = "^5.2"
```

```
black = "^21.9b0"
```

Poetry added the `requests` package as a project dependency to the `tool.poetry.dependencies` table, while it added `black` as a development dependency to `tool.poetry.dev-dependencies`.

Differentiating between project dependencies and development dependencies prevents installing requirements that a user doesn't need to run the program. The development dependencies are only relevant for other developers of your package who want to run tests with `pytest` and make sure the code is properly formatted with `black`. When users install your package, they only install `requests` with it.

**Note:** You can go even further and declare **optional dependencies**. This can be handy when you want to give



users the option to install a specific database adapter that isn't required but enhances your package. You can learn more about optional dependencies in the [Poetry documentation](#).

Besides the changes to the `pyproject.toml` file, Poetry also created a new file named `poetry.lock`. In this file, Poetry keeps track of all packages and the exact versions you're using in the project.



 Remove ads

## Handle `poetry.lock`

When you run the `poetry add` command, Poetry automatically updates `pyproject.toml` 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 add dependencies to the `pyproject.toml` file and lock them afterward.

## Pin Dependencies in `poetry.lock`

If you want to [build a web scraper with Python](#), then you may want to use [Beautiful Soup](#) to parse your data. Add it to the `tool.poetry.dependencies` table in the `pyproject.toml` file:

### TOML

```
# rp_poetry/pyproject.toml (Excerpt)

[tool.poetry.dependencies]
python = "^3.9"
requests = "^2.26.0"
beautifulsoup4 = "4.10.0"
```

By adding `beautifulsoup4 = "4.10.0"`, you're telling Poetry that it should install exactly this version. When you add a requirement to the `pyproject.toml` file, it's not installed yet. As long as there's no `poetry.lock` file present in your project, you can run `poetry install`

after manually adding dependencies, because Poetry looks for a `poetry.lock` file first. If it doesn't find one, Poetry resolves the dependencies listed in the `pyproject.toml` file.

As soon as a `poetry.lock` file is present, Poetry will rely on this file to install dependencies. Running only `poetry install` would trigger a warning that both files are out of sync and would produce an error because Poetry wouldn't know of any `beautifulsoup4` versions in the project yet.

To pin manually added dependencies from your `pyproject.toml` file to `poetry.lock`, you must first run the `poetry lock` command:

#### Shell

```
$ poetry lock
Updating dependencies
Resolving dependencies... (1.5s)

Writing lock file
```

By running `poetry lock`, Poetry processes all dependencies in your `pyproject.toml` file and locks them into the `poetry.lock` file. And Poetry doesn't stop there. When you run `poetry lock`, Poetry 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 fit your version constraints are available. If you don't want to update any dependencies that are already in the `poetry.lock` file, then you have to add the `--no-update` option to the `poetry lock` command:

#### Shell

```
$ 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.
```

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

## Install Dependencies From `poetry.lock`

If you followed the steps from the previous section, then you've already installed `pytest` and `black` by using the `poetry add` command. You've also locked `beautifulsoup4`, but you haven't installed Beautiful Soup yet. To verify that `beautifulsoup4` isn't installed yet, open the **Python interpreter** with the `poetry run` command:

Shell


```
$ poetry run python3
```

Executing `poetry run python3` will open an interactive [REPL](#) session in Poetry's environment. First, try to import `requests`. This should work flawlessly. Then try importing `bs4`, which is the module name for Beautiful Soup. This should throw an error because Beautiful Soup isn't installed yet:

Python

>>>

```
>>> import requests
>>> import bs4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'bs4'
```

As expected, you can import `requests` without any trouble, and the module `bs4` can't be found. Exit the interactive Python interpreter by typing `exit()` and hitting `Enter` .

After locking dependencies with the `poetry lock` command, you have to run the `poetry install` command so that you can actually use them in your project:

Shell

```
$ poetry install
```

```
Installing dependencies from lock file
```

```
Package operations: 2 installs, 0 updates, 0 removals
```

- Installing soupsieve (2.2.1)
- Installing beautifulsoup4 (4.10.0)

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

By running `poetry install`, Poetry reads the `poetry.lock` file and installs all dependencies that are declared in it. Now, `bs4` is ready for you to use in your project. To test this, enter `poetry run python3` and import `bs4` into the Python interpreter:

Python

>>>

```
>>> import bs4
>>> bs4.__version__
'4.10.0'
```

Perfect! No errors this time, and you have the exact version you declared. This means that Beautiful Soup is pinned correctly in your `poetry.lock` file, is installed in your project, and is ready to use. To list available packages in your project and inspect their details, you can use the `show` command. When you run it with the `--help` flag, you'll see how to use it:

Shell

```
$ poetry show --help
```

To inspect a package, you can use `show` with the package name as an argument, or you can use the `--tree` option to list all dependencies as a tree. This will help you see the nested requirements of your project.



[Learn Python](#) »

# Update Dependencies

For updating your dependencies, Poetry provides different options depending on two scenarios:

1. Update a dependency inside your version constraints.
2. Update a dependency outside your version constraints.

You can find your version constraints in your `pyproject.toml` file. When a new version of a dependency still fulfills your version constraints, you can use the update command:

Shell

```
$ poetry update
```

The update command will update all your packages and their dependencies within their version constraints. Afterward, Poetry will update your `poetry.lock` file.

If you want to update one or more specific packages, then you can list them as arguments:

Shell

```
$ 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 the versions into your `poetry.lock` file. Your `pyproject.toml` file will stay the same because the listed constraints remain valid.

If you want to update a dependency with a version that's higher than the defined version in the `pyproject.toml` file, you need to adjust the `pyproject.toml` file beforehand. Another option is to run the add command with a version constraint or the `latest` tag:

Shell

```
$ poetry add pytest@latest --dev
```

When you run the `add` command with the `latest` tag, it looks for the latest version of the package and updates your `pyproject.toml` file. Including the `latest` tag or a version constraint is critical in using the `add` command. Without it, you'd get a message that the package is already present in your project. Also, don't forget to add the `--dev` flag for development dependencies. Otherwise, you'd add the package to your regular dependencies.

After adding a new version, you must run the `install` command you learned about in the section above. Only then are your updates locked into the `poetry.lock` file.

If you're not sure which version-based changes an update would introduce to your dependencies, you can use the `--dry-run` flag. This flag works for both the `update` and the `add` commands. It displays the operations in your terminal without executing any of them. This way, you can spot version changes safely and decide which update scenario works best for you.

## Distinguish Between `pyproject.toml` and `poetry.lock`

While the version requirement in the `pyproject.toml` file can be loose, Poetry locks the versions you're actually using in the `poetry.lock` file. That's why you should commit this file if you're using [Git](#). By providing a `poetry.lock` file in a **Git repository**, you ensure that all developers will use identical versions of required packages. When you come across a repository that contains a `poetry.lock` file, it's a good idea to use Poetry for it.

With `poetry.lock`, you can make sure that you use exactly the versions the other developers are using. And if the other developers aren't using Poetry, you can add it to an existing project that wasn't set up with Poetry.

# Add Poetry to an Existing Project

Chances are, you have projects that you didn't start 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.

## Add `pyproject.toml` to a Scripts Folder

If your project only contains some Python files, then you can still add Poetry as a foundation for future builds. In this example, there's only one file, `hello.py`:

Python

```
# rp-hello/hello.py

print("Hello World!")
```

The only thing this script does is to output the string "Hello World!". But maybe this is just the beginning of a grand project, so you decide to add Poetry to your project. Instead of using the `poetry new` command from before, you'll use the `poetry init` command:

Shell

```
$ poetry init
```

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

```
Package name [rp-hello]: rp-hello
```

```
Version [0.1.0]:
```

```
Description []: My Hello World Example
```

```
Author [Philipp <philipp@realpython.com>, n to skip]:
```

```
License []:
```

```
Compatible Python versions [^3.9]:
```

```
Would you like to define your main dependencies interactively?
```

```
Would you like to define your development dependencies interactively?
```

```
Generated file
```

The `poetry init` command will start an [interactive session](#) to create a `pyproject.toml` file. Poetry gives you recommendations for most of the configurations you need to set up, and you can press `Enter ↵` to use them. When you don't declare any dependencies, the `pyproject.toml` file that Poetry creates looks something like this:

#### TOML

```
# rp-hello/pyproject.toml

[tool.poetry]
name = "rp-hello"
version = "0.1.0"
description = "My Hello World Example"
authors = ["Philipp <philipp@realpython.com>"]

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]

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

The content looks similar to the examples that you went through in the previous sections.

Now you can use all the commands that a Poetry project offers. With a `pyproject.toml` file present, you can now run scripts:

#### Shell

```
$ poetry run python3 hello.py
Creating virtualenv rp-simple-UCsI2-py3.9 in ~/Library/Caches/poetry
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.



## Use an Existing requirements.txt File

Sometimes you have projects that already have a requirements.txt file. Take a look at the requirements.txt file of this [Python web scraper](#):

### Shell

```
$ cat requirements.txt
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
```

With the [cat utility](#), you can read a file and write the content to the **standard output**. In this case, it shows the dependencies of the web scraper project. Once you've created the Poetry project with `poetry init`, you can combine the cat utility with the `poetry add` command:

### Shell

```
$ poetry add `cat requirements.txt`
Creating virtualenv rp-require-0ubvZ-py3.9 in ~/Library/Caches/

Updating dependencies
Resolving dependencies... (6.2s)

Writing lock file

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)
```

When a requirements file is straightforward like this, using `poetry add` and `cat` can save you some manual work.

Sometimes `requirements.txt` files are a bit more complicated, however. In those cases, you can either execute a test run and see how it turns out or add requirements by hand to the `[tool.poetry.dependencies]` table in the `pyproject.toml` file. To see if the structure of your `pyproject.toml` is valid, you can run `poetry check` afterward.

## Create requirements.txt From poetry.lock

In some situations, you must have a `requirements.txt` file. For example, maybe you want to [host your Django project on Heroku](#). For cases like this, Poetry provides the [export command](#). If you have a Poetry project, you can create a `requirements.txt` file from your `poetry.lock` file:

### Shell

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

Using the `poetry export` command in this way creates a `requirements.txt` file that includes [hashes](#) and [environment markers](#). This means that you can be sure to work with very strict requirements that resemble the content of your `poetry.lock` file. If you also want to include your development dependencies, you can add `--dev` to the command. To see all available options, you can check `poetry export --help`.

## Command Reference

This tutorial has introduced you to Poetry's dependency management. Along the way, you've used some of Poetry's command-line interface (CLI) commands:

Poetry Command	Explanation
\$ poetry --version	Show the version of your Poetry installation.
\$ poetry new	Create a new Poetry project.
\$ poetry init	Add Poetry to an existing project.
\$ poetry run	Execute the given command with Poetry.
\$ poetry add	Add a package to <code>pyproject.toml</code> and install it.
\$ poetry update	Update your project's dependencies.
\$ poetry install	Install the dependencies.
\$ poetry show	List installed packages.
\$ poetry lock	Pin the latest version of your dependencies into <code>poetry.lock</code> .
\$ poetry lock --no-update	Refresh the <code>poetry.lock</code> file without updating any dependency version.
\$ poetry check	Validate <code>pyproject.toml</code> .
\$ poetry config --list	Show the Poetry configuration.
\$ poetry env list	List the virtual environments of your project.
\$ poetry export	Export <code>poetry.lock</code> to other formats.

Poetry Command	Explanation
----------------	-------------

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

## Conclusion

In this tutorial, you explored how to create a new Python Poetry project and how to add Poetry to an existing one. A key part of Poetry is the `pyproject.toml` file. In combination with `poetry.lock`, you can ensure that you install the exact version of each package that your project requires. When you track the `poetry.lock` file in your Git repository, you also make sure that all other developers in the project install the same dependency versions on their machines.

### In this tutorial, you learned how to:

- Start a **new** Poetry project
- Add Poetry to an **existing** project
- Use the `pyproject.toml` file
- Pin **dependencies**
- Install dependencies with `poetry.lock`
- Execute basic **Poetry CLI** commands

This tutorial focused on the basics of Poetry's dependency management, but Poetry can also help you **build and upload** your package. If you want to get a taste of this capability, then you can read about how to use Poetry when [publishing an open source Python package to PyPI](#)).

Mark as Completed



Python Tricks

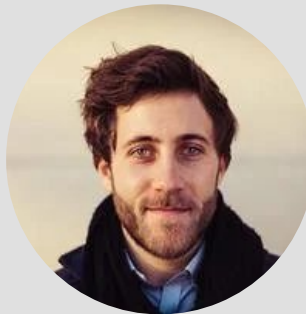


Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Send Me Python Tricks »

## About **Philipp Acsany**



Philipp is a Berlin-based software engineer with a graphic design background and a passion for full-stack web development.

» [More about Philipp](#)

*Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:*



Aldren



Bartosz



Geir  
Arne



Kate

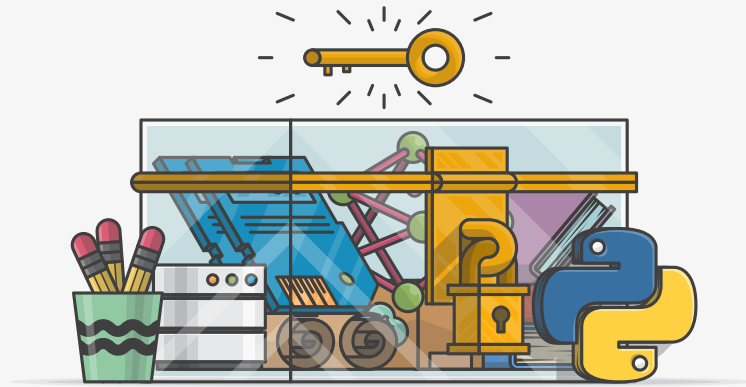


Martin



Sadie

# Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

[Level Up Your Python Skills »](#)

## What Do You Think?

Rate this article:



Tweet

Share

Share

Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

**Commenting Tips:** The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).



Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next “[Office Hours](#)” [Live Q&A Session](#). Happy Pythoning!

## Keep Learning

Related Tutorial Categories: [best-practices](#) [devops](#) [intermediate](#)

[tools](#)

— FREE Email Series —

 Python Tricks 

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Get Python Tricks »](#)

 No spam. Unsubscribe any time.

## All Tutorial Topics

[advanced](#) [api](#) [basics](#) [best-practices](#) [community](#) [databases](#)  
[data-science](#) [devops](#) [django](#) [docker](#) [flask](#) [front-end](#) [gamedev](#) [gui](#)  
[intermediate](#) [machine-learning](#) [projects](#) [python](#) [testing](#) [tools](#)  
[web-dev](#) [web-scraping](#)

platform.sh

**Deploy tens of  
websites and web  
apps in the languages  
and frameworks of  
your choice. Fast.**



[Start Your Free Trial Now](#)

## Table of Contents

- [Take Care of Prerequisites](#)
- [Get Started With Python Poetry](#)
- [Work With Python Poetry](#)
- [Handle poetry.lock](#)
- [Add Poetry to an Existing Project](#)
- [Command Reference](#)
- [Conclusion](#)

[Mark as Completed](#)





[Tweet](#)[Share](#)[Email](#)

Join Real Python and Unlock  
Learning Paths, Courses, Live  
Q&As, and More:

**Become a Python Expert »**

## Find Your Dream Python Job

[pythonjobshq.com](https://pythonjobshq.com)



[i](#) [Remove ads](#)

© 2012–2022 Real Python · [Newsletter](#) ·  
[Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) ·  
[Instagram](#) · [Python Tutorials](#) · [Search](#) · [Privacy](#)  
[Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)  
♡ Happy Pythoning!