# Python dependency management is a dumpster fire

2024-11-04 - Python

## How a fire starts

Let's walk through a common scenario.

You have a repetitive task which you want to automate with minimal effort. Naturally, you decide to whip up a quick Python script. With a few handy libraries you pip installed, you can write the logic in fewer than 100 lines of code. Everything goes well.

*The sun is shining. With all the time you saved with your automation efforts, you decide to go camping and enjoy a bonfire.*

After some time you realize you can re-use functionality in your script to solve other problems. Your script evolves into multiple scripts, modules, and eventually a library. All the while, you continue to install new dependencies with pip and your system package manager. Everything goes well.

*The sky is clear and the stars are bright. Some embers from your bonfire were blown into a patch of dry grass by the wind. It begins to smoulder.*

Others are interested in using your library. You happily share your code. Unfortunately, they can't get it to work on their computer. They have to set up an environment like the one on your machine. But how can they do that? You don't remember. After days of trial and error, they manage to get something to run, but there are still obscure bugs which you can't reproduce on your machine.

*There's a thick white smoke in the air. The trees are on fire.*

Eventually you install a few innocuous packages in your own environment and everything breaks. You don't know what caused the breakage, and you don't know how to go back to a working environment. If you did not use a virtual environment: congratulations, you now bricked your system's Python installation. Critical system packages don't work anymore.

*The sky is red. The inferno encroaches on your house.*

You decide to re-install your operating system and vow to go back to doing everything in Excel.

*The earth is scorched. All your possessions are in ashes. You have to start over.*

---

Dependencies are like a bonfire: comfortable as long as you take proper precautions. Unfortunately, Python has two problems when it comes to dependencies:

- you will need them to write any useful program. Python is mostly a "glue" language that provides convenient wrappers for libraries written in more performant languages like C, C++, Fortran, Java or Rust. If you don't want to write your own C extensions, you will rely on libraries written and published by other people.
- the standard tooling makes it almost guaranteed you do the wrong thing. "Pip install package" and you are off to the races, right? This is especially problematic given that Python is advertised as a "beginner friendly" programming language. Managing Python dependencies well is all but simple and intuitive.

In short, venturing into the Python ecosystem is like camping in a dry forest full of dry grass and dead trees on a cold night. You'll need to light a bonfire and there's a very good chance it will turn into a full scale forest fire.

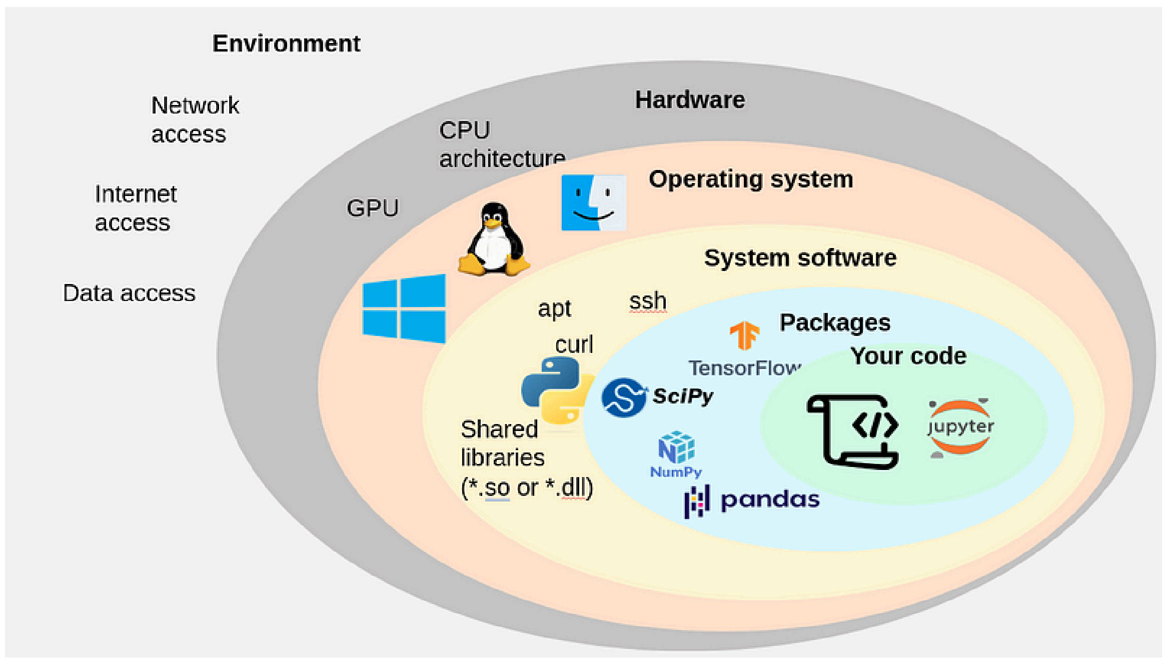
This article is all about fire safety techniques and tools. It's about how you should think about dependency management, which tools you should consider for different scenarios, and what trade offs you'll have to make. Finally, it exposes the complexity and lingering problems in the ecosystem.

## What are dependencies?

Let's begin by defining terms. What is a dependency? You might propose it's another package or library you use in your code. It's what you pip install.

Unfortunately, this is only a subset of all dependencies. Ultimately, **a dependency is anything external to your own code that is required to get your code to run as expected.** We can divide these up into a number of layers:

1. **Project specific packages.** These are your typical Python libraries you would install with your Python package manager.
2. **System packages.** These are global packages or libraries (e.g. .*so or* .dll files) installed system wide, using your system package manager (e.g. homebrew, apt, pacman, ...). These are shared among all users and all projects.
3. **Operating system.** You might think that Python runs on any operating system. This is only true because Python itself and libraries written in C are compiled for different platforms. Packages that are only compiled for Linux will not work on Windows. More subtly, any operation that makes low level system calls, e.g. allocating memory or writing a file to disk, underneath the hood may behave slightly differently depending on which operating system you run the code on.
4. **Hardware.** CPUs have different architectures, like X86, amd64, or arm64. Code that is compiled for one architecture will not run on another. Your Python code does not need to be compiled, but the underlying libraries and Python itself do. Additionally, your code may depend on the availability of hardware accelerators like GPUs.
5. **The environment.** This is everything outside your computer and relates to network access. Maybe your code needs to fetch information from a database, from a website or an API. Those external resources can also be considered dependencies.



The point here is to recognize that your "simple" script or Jupyter notebook often relies on a tall and complex stack of systems, libraries, and environmental conditions in order to function. When you develop some code on your machine you rarely think of this complexity. But it becomes essential once you want to run your code on other machines or work together on the code with others.

## Why should we "manage dependencies" and what does that even mean?

The key reason for adopting good dependency management practices is **reproducibility**. Reproducibility means we can follow a set of steps and always get exactly the same result. Reproducibility makes the behavior of the code **deterministic**: everyone that interacts with the code sees the same behavior and finds the same bugs. As soon as a single step in the chain of instructions is not reproducible, the behavior of code becomes non-deterministic.

There are three stages in which reproducibility is important:

1. **The development stage:** while you develop, you want to ensure you can always easily re-create the environment needed for your code to run. This is important even if you develop by yourself. If you accidentally break your environment, you always have the ability to start over. It is critical when you develop as part of a team. Ideally everyone develops in identical environments. The development stage may require additional development dependencies that are not required at run time, for example linting and formatting tools.
2. **The build stage:** when you want to publish your code or deploy your application, you may have to perform some additional steps. If you are publishing a pure Python library, maybe that is simply putting your code in a zip-file or tarball and uploading it somewhere. If you have written extensions in other languages, you may need to compile them. If you are deploying a web service, you may have to build a Docker container.  The end result of a build stage is one or more artifacts. Reproducibility means that you can always produce identical artifacts by re-running the build i.e. you guarantee that there is a one-to-one correspondence between your code and whatever it is you publish. The build stage may also require additional dependencies, for example a compiler, a container build tool-chain, or even just a zip-file creation tool.
3. **The deployment stage / run time:** when your code or application is running out there in the wild, you want to guarantee predictable behavior. You don't want the behavior of your code to change because you were not carefully tracking your dependencies. Ideally, your code behaves the same as during the development stage.

Good dependency management means that:

- **all** dependencies for the development, build and deployment stages are **explicitly declared and tracked together with the code in version control.** This is simply reflects the fact that **your application = your code + all your dependencies.**
- there is an **automated process** that is able to turn these declared dependencies into a working environment.
- it is possible to safely **evolve and update** your environment when new versions of dependencies become available.

In most circumstances, this is possible to do up to the "operating system" dependency layer. If you work in the cloud or on a hypervisor, where hardware can be declared in code, it is possible to expand your control over the dependency stack even further. In most cases you must handle assumptions about the environment, like availability of hardware or network connections, in the application code.

In the rest of this article, we will mainly deal with software dependencies up to the operating system level.

## What is the best way to manage dependencies?

There are two prerequisites before we can even begin to talk about managing dependencies:

- **Version control**
- **Environment isolation**

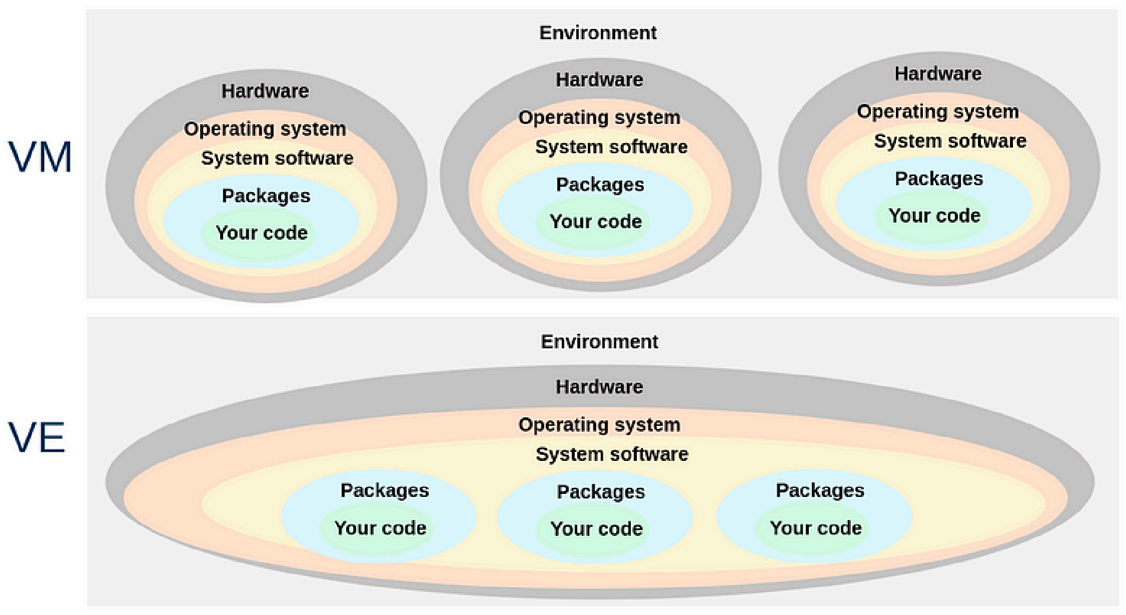
You should be tracking your code with a version control system. There are no exceptions. The industry standard is git. At the most basic level, a version control system ensures that you can always roll your code back to any recorded point in history (i.e. a "commit"), so you can make changes fearlessly. We require the version control system to also track the files in which we declare our dependencies.

Secondly, you should have a mechanism of isolating the environment for developing and running your project from the host machine. Otherwise, the dependencies required for the project could break the host environment or other projects on the same machine. Depending on the level of isolation, the host environment can still influence the behavior of all projects.

The optimal level of isolation is achieved by using a separate machine per project; practical examples of this approach include virtual machines or development containers.

These options are not always convenient, so many developers opt for a more lightweight form of isolation: the virtual environment. There are multiple kinds of virtual environments for Python projects, and depending on their implementation they either only isolate pip-installable Python packages, or they also include some system level dependencies. We will compare these options when we discuss specific tooling.

The downside of virtual environments is that there may be implicit dependencies, i.e. extra requirements on the host, that live outside the isolated environment. In practice, virtual environments usually offer sufficient isolation from the host for many Python projects.



There is also a more niche and exotic option that sits between virtual environments and virtual machines: Nix. Nix tries to take the virtual environment idea to the extreme and applies it to all software on the system, to guarantee that your environment does not leak to the host and vice versa. We will touch on the benefits and drawbacks of Nix near the end of this article.

Once you have version control and environment isolation figured out, good dependency management boils down to a few steps:

- Creating a **definition file**
- Generating a **lock file**
- **Syncing** your environment with the lock file
- Tracking **both** the definition file and the lock file in **version control**

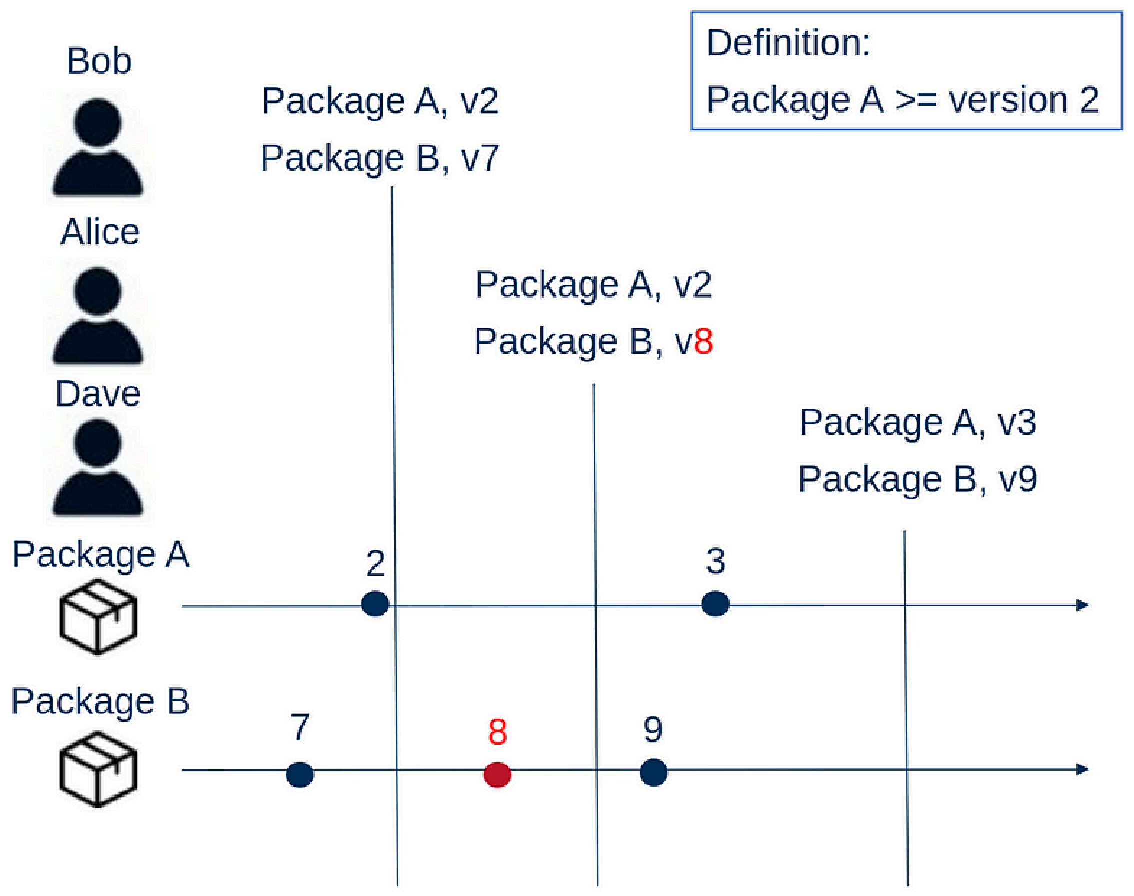
Why? Let's break it down.

A definition file is a file where you **state the dependencies you need and the minimal version constraints.** It's all the packages you directly use in your code (check the import statements). For example, you need pandas, at least version 1.5, but less than 2.0. The idea is that if someone else creates an environment based on your definition file, the code *should* work.

However, in practice, it's impossible to guarantee that the code will work for all possible valid combinations of package versions. Suppose you have defined *n* packages. These packages don't depend on each other and don't have dependencies of their own. For each package you provide a range of acceptable versions that span *m* releases. In this scenario, there are $m^n$ valid environments. With only a handful of packages and versions, it becomes entirely infeasible to test all possible environments. There's a good chance your code will not work for at least some combination of packages.

In reality, the situation is even worse: most packages require their own dependencies, which count towards the exponent *n*. These **transitive dependencies** are implicit and not declared anywhere in your project. The projects you depend on have exactly the same problem as you: their definition file can produce a broken environment. Therefore, it's possible that some version of some transitive dependency breaks your environment. Sounds unlikely, but I have seen it happen.

What does this mean? It means that if I create an environment from a definition file, and you create an environment from exactly the same file at some other point in time, we could end up with different environments. Mine could work fine, while yours could be broken. And this could be caused by a broken release from some transitive dependency we did not even know we needed. Consider the following illustration:



We have a definition file that requires package A with at least version 2. Package A requires package B. When Bob creates his environment he installs package A version 2 and package B version 7. A bit later, package B releases version 8, but it contains a massive bug. Alice happens to create her environment just after this release and she gets package A version 2 and package B version 8. Her environment is broken. The bug in package B is fixed after some time and version 9 is released. Also package A releases a new version. Dave creates his environment and gets package A version 3 and package B version 9. His environment works fine. Nothing changed in the definition file, yet environments went from working to broken and back to working.

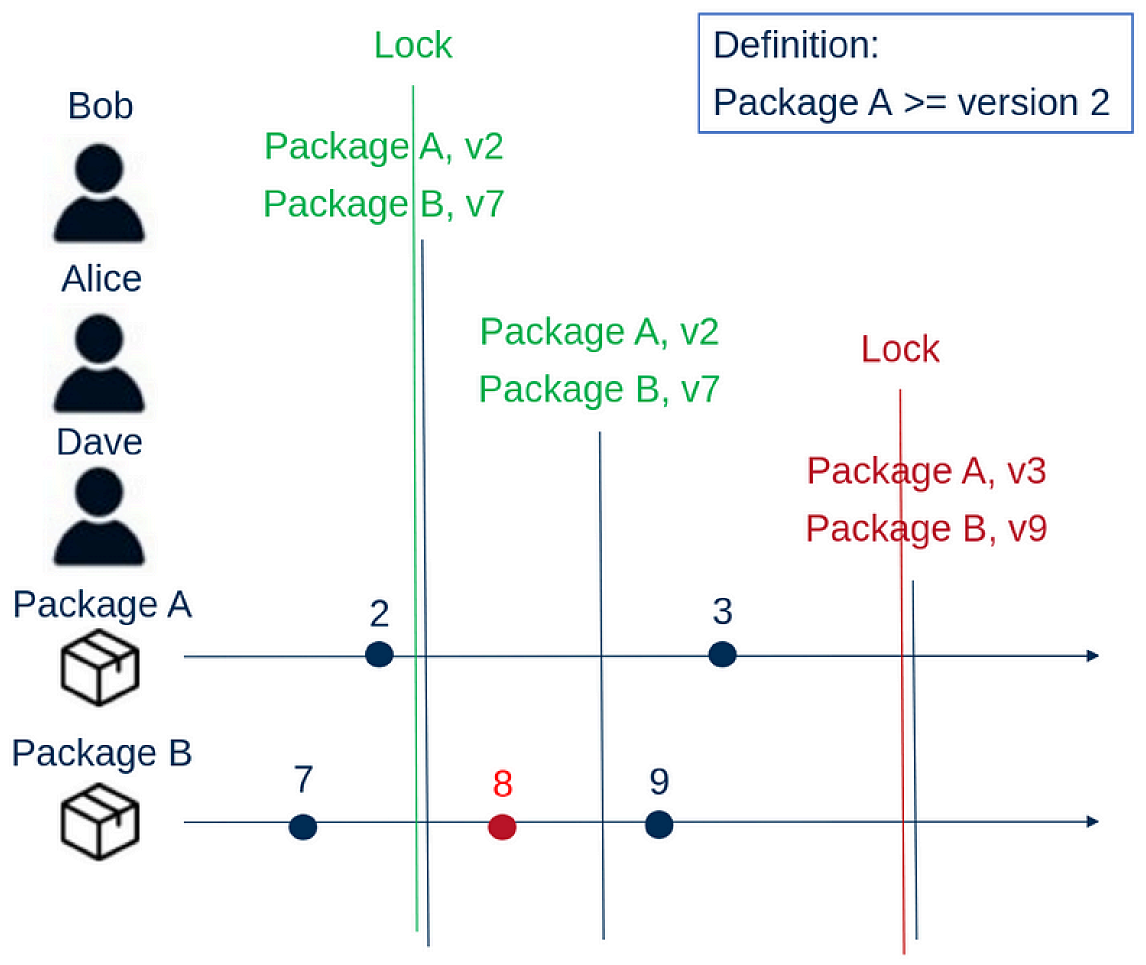
The point of the example is that environments created from the definition file are a function of time. With hundreds of implicit and explicit dependencies, it is impossible to track the release schedules and change logs of all these projects. Consequently, you are playing Russian roulette every time you set up an environment.

While it is impossible to guarantee that environments are not broken, it IS possible to remove time as a variable and guarantee that we can always re-create exactly the same environment. This requires pinning both our required dependencies and our transitive dependencies. This is what the **lock file** is for.

Instead of creating an environment directly from the definition file, we allow our **dependency resolver** (typically embedded in our package manager) to figure out what packages and versions it would install if we were to create an environment at that moment in time. Instead of creating this environment, we write all the packages and fixed versions to a file - the lock file. Good lock files also include hashes, so that we are absolutely sure everyone is downloading the same artifacts.

In this way, we have inserted the explicit step of "locking" our dependencies into the process.

The lock file represents a fully reproducible environment (as long as packages and versions remain available on repositories). What happens to our example when Bob, Alice and Dave used a lock file?



Suppose Bob is first. He first creates a lock file, which will look like his environment in the original example. He then creates his environment based on the lock file. For Bob nothing changes except that he had an additional intermediate step. Bob commits the lock file to version control.

In the revised scenario, Alice, can install exactly the same environment as Bob because she gets the lock file. Since Bob's environment works, she decides to create her environment from the lock file. She does not suffer the issues associated with installing v8 of package B.

Dave could also install exactly the same environment. However, he read somewhere that a new version of package A came out, and he would like to use some of the new features. He refreshes the lock file and creates his environment. He experiences no issues, so he commits the new lock file to version control. The next time Bob and Alice work on the project, they update their environment to reflect the new lock file. With the lock file we guarantee that the environment evolves deterministically together with the code. What's more: if our environment ever breaks, for example if Alice had refreshed the lock file, we can always go back to an earlier working environment because it is saved in version control.

It's a quite simple and yet ingenious idea to give you *some* grip on the uncountable number of possible environments.

Of course, this approach doesn't solve all problems related to dependencies. You might commit a lock file which you thought was working, but only much later discover the environment is in fact broken. By that time your code has evolved so much it may not be so easy to turn back the clock. You may still have difficult troubleshooting sessions to figure out where exactly in your dependency stack issues arise. If you pinpoint the issue, you may have to perform a manual hacky patch until things get fixed upstream. All these reasons and many more is why some in the industry suggest projects should avoid dependencies altogether.

Unfortunately it is nearly impossible to avoid dependencies with Python. We must deal with them. What makes this extra frustrating is that the process is made unnecessarily difficult.

# Why is managing dependencies in Python hard?

## Default tooling does not encourage best practices

For many modern programming languages, the associated tooling has the lock-file based dependency management mechanism baked in. For a great example, consider Rust's Cargo.

Not so with Python.

The default package manager for Python is pip. The default instruction to install a package is to run `pip install package`. Unfortunately, this imperative approach for creating your environment is entirely divorced from the versioning of your code. You very quickly end up in a situation where you have 100's of packages installed. You no longer know which packages you explicitly asked to install, and which packages got installed because they were a transitive dependency. You no longer know which version of the code worked in which environment, and there is no way to roll back to an earlier version of your environment. Installing any new package could break your environment.

Even worse is that if you run `pip` from your system's Python, you can break apps that all users on the system rely on. You can mitigate this danger by running `pip install --user package`, which installs the package only for your user in the `~/.local` directory, or by using virtual environments. Unfortunately virtual environments are not often taught to beginning Python programmers as it's often deemed an advanced topic. Additionally, default virtual environments come with limitations, which we will discuss.

## The system software and project package problem

Because Python is mostly a "glue" language, you often end up in a situation where you need non-Python dependencies in your environment. Unfortunately, those are usually not installable through pip, so you need additional tooling to bootstrap your environment.

The non-Python-package dependency that everyone has to deal with is the Python interpreter itself. You can use the system package manager to install it, but then you are stuck with one version of Python for all other projects on your machine. As a consequence, all kinds of tooling has been developed specifically to manage multiple Python versions on the same system.

The more tools you need to declare and create an environment, the more difficult it is to automate the setup and guarantee reproducibility.

## Ecosystem fragmentation

Because Python's default tooling is somewhat lackluster, a plethora of third party tools have been developed to patch the gaps. Each of these tools are developed with particular use cases in mind, and none are appropriate to deal with all possible project cases nor the entire development lifecycle.

When tools are mutually compatible, interoperable and synergistic, a large number of options to choose from isn't a problem. Unfortunately, in the case of Python, the landscape is split between two ecosystems: Pypi.org + pip and anaconda.org + Conda (more on this later). These tools can be made to work together, but this can cause issues down the line.

The end result is that you have to make an informed decision on which of the many tools you need for your specific project. In this process, you must deal with the noise of people on the internet who vouch for whatever tool they like as the end-all-be-all tool.

### Third party tooling is often written in Python

The ultimate irony is that many of the Python developer tools are themselves written in Python. That means, to install them, you first need to have a Python installation and an environment. If you install these tools inside the same environment as your project, the dependencies of your tools can start to conflict with the dependencies you need for your application. Very soon you reach toolception: you will need tools, to manage dependencies for tools that manage the dependencies of your application. Fortunately, there has been a recent paradigm shift, where many of the new tools for Python are now written in Rust and distributed as statically linked binaries.

## Available tooling for Python

Now that we have an appreciation for the concepts, let's survey the tooling landscape for Python environment and dependency management. Specifically, let's look at the capabilities, advantages and drawbacks of different tools.

I meanly focus on breadth rather than depth. I'm not an expert user for ALL these tools, so if I write something about your favorite tool that is not true please point this out in the comments.

This section is about getting informed and making up your own mind. In the next section, I'll give you my personal advice on what tools I would use in different situations.

### pip

Pip is the default package manager that comes with Python. This is a big advantage: you don't need to install anything else. By default it installs packages from the Pypi.org repository where a huge number of packages are available (500K+). It can also be configured to install from other (private) package repositories or directly from git repositories that follow a specific structure.

Since wheels have become an accepted distribution format in 2013, pip is fast and simple to use. Before wheels, Python packages were distributed strictly as zipped source files. For pure Python packages this doesn't matter very much: you just unzip the archive and use the code. But for packages with extension modules in other languages, these need to be compiled on the system of the user. This requires build dependencies, like a build system and a compiler. This could get complicated, error prone, and slow for packages that are mainly written in C or C++ like scientific libraries. Instead, wheels are binary distributions; extension modules are compiled beforehand by the publisher. Compiled extension modules are platform specific, so it is up to the publisher for making wheels for multiple platforms.

Until quite recently (2020), pip did not have a robust dependency resolution algorithm, meaning that you could very easily break an environment. Depending on the order in which you installed packages, you could get a different and even inconsistent environment. This is no longer the case, as long as you use pip v20.3 or higher.

The downside of pip is that it's a Python tool. You first need a Python installation in order to use it, and pip is confined to that Python installation.  Pip can not manage Python itself, nor any other non-Python package. For that you need separate tooling. However, non-Python code can still be packaged *as if* it were a Python package and made available on Pypi: consider many of the packages in the science or data space.

Additionally, pip encourages imperative use, which is not great for reproducibility. Essentially, you run pip install commands until you have the environment that you want. You can declare your dependencies in a `requirements.txt` file (your definition file) and create your environments from this. However, there is no lock file. You could manually generate a kind of lock file using something like `pip freeze >> environment.lock`, as this will look at all the packages currently installed in your environment and write them to a file. However, such a manual process is error prone and most people prefer to use a dedicated tool to manage lock files.

So, to summarize pip:

### Capabilities:

- Install Python packages

### Advantages:

- Included in Python since Python 3.4
- Fast installs since the wheel packaging format introduction in 2013
- Since pip 20.3 decent and fast dependency resolution algorithm

### Disadvantages:

- It's a Python tool
- No installation of non-Python packages
- No lock files

## venv

One could say that venv is the pip equivalent for virtual environments. It's a built-in tool that serves to create virtual environments. Inside the virtual environment you can install packages with pip.

The way it works is that it creates a folder, typically within your project directory, which contains the packages you need in your environment. A small shell script inside this folder "activates" the environment by setting the `PYTHONHOME` and `PATH` environment variable, which tells Python where to look for packages and executables.

A potential downside of this virtual environment approach is that if you have a lot of packages with similar dependencies, you will have to duplicate them for each project. In principle you don't have to redownload them, since pip implements caching.

Just like with pip, it's a Python tool, so you need a Python installation. There is a hard link from the venv to the Python version you used to create the virtual environment; you can not change the Python interpreter.

Venv also does not modify the shared library paths, and does not deal with non pip-installable packages, so a lot of system dependencies and other ways to install non-python packages may still be required for your environment to work.

So, to summarize venv:

### Capabilities:

- Manage virtual environments for Python packages

### Advantages:

- Included in Python since Python 3.3

### Disadvantages:

- It's a Python tool
- All environments must use the same Python interpreter
- No installation of non-Python packages

## virtualenv

Virtualenv is the O.G. venv. Before venv became part of Python, virtualenv could be used to create virtual environments. It must be installed through pip. The main differences between venv and virtualenv is that with virtualenv you have the option to point to a different Python interpreter to create the virtual environment. So you might have a Python 3.9 install with virtualenv. If you also have Python 3.12 installed, you can use this virtualenv to create a virtual environment that uses Python 3.12. This is not possible with venv.

### Capabilities:

- Manage virtual environments for Python packages

### Advantages:

- Can point to a different Python interpreter

### Disadvantages:

- All the same ones as venv
- It's a third party package that must be pip installed

## pip-tools

pip-tools is a light-weight tool that introduces the lock file mechanism to Python. Instead of directly writing your `requirements.txt`, you write a `requirements.in` file, which is your definition file. You then use the `pip-compile` command to generate the `requirements.txt` file, which functions as the lock file. You can then use `requirements.txt` with pip to set up your environment, or if you already have an environment you can use the `pip-sync` command provided by pip-tools to sync your environment with what is defined in the lock file.

It's lightweight, simple, and it's the only thing you need besides the default tooling.

You can have an arbitrary number of `requirements.in` files, which can be useful if you have different types of environments you'd like to be able to create. For example, it's common that for development, you might need some additional dependencies for testing and linting. These can go in the `dev-requirements.in`. For each definition file, you must maintain a lock file.

The downside is that it is again a Python tool, so you need to install it into your own environment. It might not be compatible with the Python version you need or its dependencies might conflict with the ones you need in your project.

A small inconvenience is that you have to manually maintain the `requirements.in` files. More advanced tools provide users with CLI utilities to facilitate this process.

### Capabilities:

- Manage lock files

### Advantages:

- Light weight, simple, interoperable with basic pip/venv tooling

### Disadvantages:

- It's a Python tool
- Only able to deal with packages that can be installed with pip
- Managing the definition files is a manual process

## Pipenv

Pipenv is as if pip + virtualenv + pip-tools were combined into one tool. It essentially provides a command line tool that wraps `pip` and `virtualenv`, and implements its own locking mechanism. Instead of a `requirements.in` and `requirements.txt` , pipenv maintains a `Pipfile` and `Pipfile.lock`. The idea behind these files is similar, but the format is completely different. Using the pipenv command, you can install packages and create and maintain virtual environments. The Pipfile and Pipfile.lock will be updated automatically based on the commands you run.

Again, it's a third party tool written in Python with all the associated drawbacks. Another minor drawback is that the tool only supports normal dependencies and "dev" dependencies, so it's not possible to define environments in a more granular way. As it just wraps pip and virtualenv, all the same limitations are inherited.

### Capabilities:

- Install packages
- Manage virtual environments
- Manage definition files
- Manage lock files

### Advantages:

- Light weight, simple, wraps basic pip/venv tooling

### Disadvantages:

- It's a Python tool
- Has its own format for definition and lock files
- Only able to deal with packages that can be installed with pip
- Only able to distinguish between dev and non-dev dependencies

## Poetry

Poetry is the first tool on our list so far that aims to capture the entire development flow for a Python project, from project bootstrapping, virtual environments, to dependency management, to even building and publishing packages. Poetry is often praised in the blogosphere as the final evolution in Python tooling. Unfortunately, it also has its drawbacks, especially when you deal with more complex packages.

Concerning the development aspect (virtual environments and dependency management) poetry could be seen as a competitor to pipenv. For Poetry, the definition file is the `pyproject.toml` file. You can use the Poetry CLI to install new dependencies (`poetry add`), which automatically adds them to `pyproject.toml` and installs them in your environment. Poetry maintains a `poetry.lock` file for reproducibility. Poetry manages virtual environments for you, and has a feature that it automatically installs the current project into the environment in "editable mode" (pip supports this with `pip install -e .`). If you use Conda environments (see later), Poetry can detect this and use this environment as virtual environment.

Besides managing the development workflow, Poetry also handles the build and deploy workflow. Basically, it provides similar functionality as setuptools + twine out of the box, without using those packages as dependencies.

However, the build back-end of Poetry seems quite basic as the documentation states it supports building only pure Python wheels. That means it does not natively support building extension modules in other languages like C, which setuptools does support. Of course there exist workarounds.

A nice feature of Poetry is that it supports grouping dependencies, which allows you to have more control over which dependencies you want to install in your environment. This is similar to having multiple definition files, except everything is organized nicely in the `pyproject.toml` file.

The downside of the single file approach is that all dependencies you declare must be mutually compatible. So if want to manage a set of environments that are not compatible, poetry becomes problematic. For example, I recently worked on an NLP project, where I wanted to be able to create an environment with GPU enabled pytorch (for training) and another environment with only CPU enabled pytorch (for deployment and inference). CPU pytorch and GPU pytorch can not be simultaneously declared in the `pyproject.toml`. Hence, I had to create all kinds of hackyness to work around the problem of maintaining different incompatible environments with poetry.

Poetry does not wrap pip. It has its own dependency resolution algorithm, which online seems to receive mixed reviews with some complaining it can be rather slow.

Poetry is again a tool written in Python, with all the associated drawbacks. Compared to pipenv, it is more heavyweight with more dependencies (41 versus only 6). This means it can become extremely problematic to install directly inside your development environment.

In summary, Poetry is great for simple pure Python projects that only require pip-installable dependencies. The fact that it is an all in one tool is simultaneously its greatest benefit as its greatest drawback.

### Capabilities:

- Install packages
- Manage virtual environments
- Manage definition files
- Manage lock files
- Build packages
- Publish packages

### Advantages:

- All-in-one opinionated tool for the entire development lifecycle of a Python project
- Convenient CLI
- Dependency grouping

### Disadvantages:

- It's a Python tool
- More heavy tool, more dependencies
- Less interoperable with other tooling, no support for other build backends
- No support for maintaining mutually incompatible environments
- Has its own format for dependency definition and lock files
- Only able to deal with packages that can be installed with pip
- Does not support complex package builds

## PDM (Edit 14/12/2024)

When I shared this article online, I was asked why I did not mention PDM. The honest reason was because I had not heard of it. I also have not used PDM. PDM seems to be in many ways like Poetry, but better. Being newer, it does not carry the baggage of Poetry. It adheres to PEP standards with regards to the `pyproject.toml` file. But it is still a tool written in Python with quite some dependencies, which comes with some drawbacks. For completeness I mention it here, please let me know in the comments if I've missed something critical about PDM.

### Capabilities:

- Mostly the same as Poetry

### Advantages:

- Adheres to PEP standards
- Can use `uv` (see later) for dependency resolution and installation
- You can use the `PDM` build back-end independently from PDM

### Disadvantages:

- Mostly the same as Poetry

## pyenv

Until now, all of our tooling assumed Python was already installed on the system. All these tools used the virtual environment as isolation level. The virtual environment includes pip- or poetry-installed python packages and a symlink to a Python interpreter. But how do we install these Python interpreters and ideally manage multiple python versions on our system? None of the previously mentioned tools provide a solution.

This is what pyenv does. Pyenv is a simple shell utility that can be used to install different versions of Python, and "activate" them globally or at a project level. That means that the `python` command in the shell can be easily configured to point to the right executable.

Pyenv is the first tool that does NOT require a Python installation up front. The default installation method is through a system package manager like homebrew, which can be annoying for users that do not have admin/sudo access on a system. However, it can be also be installed at a user level through other means with a bit more effort. Initial set-up may also require modification of the shell configuration file.

Despite what the name might suggest, pyenv does not manage virtual environments. It only manages python versions. This is a good thing, as pyenv can be used in combination with all the previously mentioned tools.

Something which many users may find a drawback is that pyenv installs Python from source. Each time you install a new Python version you will have to wait 5–15 min, depending on what kind of hardware you are working with, for Python to compile and install.

Compiling from source also means that your system must have a C compiler installed and a few other system dependencies. If you don't have those and don't have admin/sudo access to your system you are back at square one. In either case, setting this up the first time can be a hassle depending on your operating system.

Finally, pyenv only works on Unix-like operating systems. It can not be installed on Windows. There is a separate project called pyenv-win which aims to port pyenv to the OS. Installing all the relevant dependencies on Windows, like a C compiler, can be quite a nightmare.

### Capabilities:

- Install and manage different Python versions

### Advantages:

- Pure shell scripts, no Python dependency
- Follows Unix philosophy: does one thing well

### Disadvantages:

- Installing a new Python version requires downloading and compiling the source code
- Can require a bit of set-up the first time to get it to work. Quite a few build dependencies must be installed.
- No Windows support.

### Edit 14/12/2024

The reddit user AndydeCleyre informed me of mise. Mise does everything pyenv does and more. You can use it to install different versions of not only Python, but also other language runtimes like Node. In addition, it's a task runner like Make and allows you to set environment variables automatically based on your current directory like direnv. I have not used the tool myself, but it seems like a worthwhile tool to mention here.

## pipx

Most of the tools we have discussed previously suffered from being written in Python themselves. In order to install them you need a Python installation first. Then where do you install the tools?

If you install it inside your project's virtual environment, the tool's dependencies may conflict with the ones you require for your code. If you install it with pip at the user level, every project you work on must use the same version of the tool. Worse, all tools will share the same environment.

You could try to create separate virtual environments inside your project for your tools, but you may have to perform some magic to get the tools to install packages in the right environment.

pipx is a tool that installs pip packages at the user level, each in a separate virtual environment, and symlinks the entrypoints to locations on the `PATH` like `~/.local/bin`. In this way, poetry, pipenv, and other tools can be installed at the user level without conflicting with each other. All of the packages can be independently updated using the pipx CLI. This is a much better approach than just pip installing the tools at the user level, since all of the tools are isolated and don't have to share dependencies.

Unfortunately, it seems it is not possible to install multiple versions of the same tool and switch dynamically between them. For example, one project may be working with Poetry v1.5 while another uses v1.8. It is possible to use pipx to temporarily use a specific version of a tool using `pipx run`, but it seems it's not possible to keep all versions permanently and switch between them.

Additionally, somewhat ironically, pipx is also written in Python, and must be installed at the user or system level. The virtual environments that it provisions links to a Python version, which by default is the same Python that is used to run pipx. Fortunately this can be overridden using the `--python` flag. `pyenv` should be used in tandem to install the different Python interpreters.

### Capabilities:

- Install pip packages as executables in isolated virtual environments

### Advantages:

- Better than pip installing a tool directly at the user level → isolated dependencies and can use different Python interpreters

### Disadvantages:

- It is a tool written in Python
- No installing of multiple versions of the same tool, so the tool version must be shared among all projects

## uv

uv is relatively new tooling, developed by the same people who created ruff, that aims to be the all-in-one python project and package manager. As it says in the README:

> A single tool to replace `pip`, `pip-tools`, `pipx`, `poetry`, `pyenv`, `virtualenv`, and more.

It indeed delivers on this promise and more. uv feels very much like a tool that aims to do everything, and manages to do it better. Let's dive in.

Firstly, it's written in Rust and is distributed as a single binary without external dependencies. Just download it and you can use uv. That is a whole lot less hassle than installing and using the previously mentioned python-based tools. A binary is available in the GitHub releases for all the most common platforms.

Secondly, it's fast. 10–100x faster than pip for resolving dependencies.

Just like Poetry, uv handles the entire development flow: installing of packages, managing virtual environments, building and publishing. It does dependency management correctly, with dependency definitions in the `pyproject.toml` and a lock file in `uv.lock`. But it does many things better. uv follows Python standards (PEP 508 and PEP 621) for defining dependencies, while Poetry does not. uv allows you to choose any package build back-end; Poetry does not. uv maintains a global package cache which is useful when dependencies are duplicated across projects and environments.

In addition, Poetry does not manage the Python version in your environment. For that you have to rely on pyenv. With uv, you no longer need pyenv either; uv installs and manages Python interpreters. While pyenv installs Python from source, uv downloads and installs binaries from their own repositories. Much more convenient and no build dependencies required on the host. The only potential downside is that the Python interpreter may be less optimized for your specific CPU.

Furthermore, uv also does what pipx does: you can install a pip package as an executable in its own isolated environment. So you can use it to manage Python tools. uv can also create isolated environments for simple scripts.

uv explicitly aims to be compatible with existing and default tooling as much as possible, especially pip and pip-tools. It has a pip interface and a built-in mechanism to convert its lock file to a `requirements.txt` format.

Until very recently, uv did not support arbitrary grouping of dependencies. There were only normal dependencies and `dev` dependencies. This was a slight edge Poetry had over uv. However, since version 4.2.7 this is now also supported in uv.

There are a few limitations to be aware of.

Firstly, just like Poetry, there seems to be no simple built-in way to manage mutually incompatible environments for the same project. There is only one `pyproject.toml` and one lock file; all dependencies need to be consistent.

Secondly, while it's already a great step forward to be able to manage both Python interpreters and pip packages with one tool, this doesn't cut it for some projects. Of course this is a limitation not only of uv, but also all the previously mentioned tooling.

## Capabilities:

- Install packages
- Install pip packages as executables in isolated environments ("tools")
- Manage Python versions
- Manage virtual environments

- Manage definition files
- Manage lock files
- Build packages (provided there is a build backend)
- Publish packages

### Advantages:

- Written in Rust: *blazingly fast*™, single small binary, no external dependencies
- Multi-platform support (most CPU architectures + all major OS's: Linux, MacOS, Windows)
- All-in-one opinionated tool for the entire development lifecycle of a Python project
- Convenient CLI
- Global package cache
- Downloads Python interpreter binaries
- Interoperable with PEP and Python standards, e.g. standard format for dependency definition in `pyproject.toml`
- Can select any build back-end for complex builds
- Arbitrary dependency grouping

### Disadvantages:

- No support for maintaining multiple mutually incompatible environments
- Only able to deal with packages that can be installed with pip (and Python interpreters)

## Conda

With Conda, we come to the bifurcation in the Python ecosystem. The "main" Python ecosystem centers around pip-installable packages from pypi.org. All the previously mentioned tooling orbited in this system. Conda is a completely different package manager developed by Anaconda, a private company. It primarily serves to install packages from anaconda.org, a repository that has no link with pypi.org. Unlike pip, Conda is also able to create virtual environments.

So why have this parallel ecosystem of packages?

The main reason Conda was developed, was to solve the problems faced by the fields of scientific research and data science.

Most of the packages in these fields are Python wrappers over libraries written in C, C++ and Fortran. Back when Python packages were distributed as sources (up until 2013), the burden was on the user to compile these extension modules. The process to do this is encoded in the setup.py script, but builds still rely on external dependencies (like a compiler) being available on the user's system. Configuring a system to get a build to work can be a painful process. Complex packages like Matplotlib can take ages to build, depending on what hardware you are working with.

Additionally, while Python code works on any OS (because the CPython interpreter is compiled for all platforms), extension modules may not. Different operating systems have different C/C++ compiler toolchains, and C++ code that compiles with g++ may not compile with clang or Visual Studio C++. Packages may also have to be compiled against specific versions of shared libraries that need to exist on the user's system. Many developers of scientific Python packages work on Linux or Unix-like, but a majority of the users may work on Windows. If the developer did not consider all possible platforms their code may run on, users could be left stranded.

The Conda ecosystem places the burden of building and compiling code on developers and package maintainers. Packages need to be distributed to users as binaries (i.e. all code that needs to be compiled is already compiled), so installation is fast, easy and reproducible for users. No undeclared build dependencies required for compiling stuff on the side of the user.

You might think that, with the introduction of wheels as distribution format on Pypi, Conda is entirely obsolete. For installing most Python packages or statically linked binaries (e.g. DuckDB) this is mostly true. However, for anything else, Conda remains far superior, which is why it still shines in the data science and scientific Python space.

Compared to pip, Conda has a much broader definition of what constitutes "a package". It can be a Python package, but also a shared library, header files, or an executable. In this sense, Conda is much more similar to a system package manager like apt or homebrew. That means that Conda can, for example, be used to install a specific version of NVIDIA's CUDA Toolkit in a Conda environment. This resource is shared among the packages in the environment.

This is not possible with wheels. Wheels expect that all non-Python dependencies, like shared libraries, are packaged with the wheel or are available at the system level. If you install numpy with pip, the wheel ships with almost everything it needs, but very low level libraries may still be expected to be present on the user's system. With Conda, almost all these libraries can be installed as separate packages.

Conda virtual environments are therefore much more like a mini isolated machine. Your Python interpreter lives inside the environment, because this is just another package you can install with Conda. All the packages you would otherwise have to install system wide can live there. Of course your Python packages can live there as well, and you can even use pip inside the environment to install Python packages from Pypi.org (though this is not recommended if some of your Python packages come from anaconda.org).

There are advantages and disadvantages to both the wheel and the Conda approach.

Wheels are isolated from each other, so Matplotlib and Numpy can live in the same virtual environment while being built against and shipped with different shared libraries. This is not possible in a Conda environment, where Numpy and Matplotlib have to use the same shared libraries, like BLAS and LAPACK, that are distributed as separate packages.

On the other hand, being able to share dependencies is in some cases practical or even essential. Packaging the CUDA Toolkit into each wheel that runs code on an NVIDIA GPU would balloon their size to a ridiculous extent. The consequence is that wheels still have implicit dependencies on things installed at the system level. This can be mostly avoided with Conda. Still, Conda expects some packages at the system level, mainly very low level libraries like libc and libm.

Because packages tend to be more granular, packages in the Conda ecosystem have more dependencies, and dependency resolution becomes a lot harder and more time consuming. Conda ships with a much more robust algorithm than pip, a very important point in favor of Conda before pip's algorithm was improved in version 20.3. However, Conda is implemented in Python so it is excruciatingly slow.

Some other noteworthy design differences between the Conda and pypi ecosystems:

- Conda environments are typically "global" and designed to be shared among multiple projects. You can "activate" them in your shell from any location on your system. This often leads to situations where Conda environments are even shared among multiple users. Normal virtual environments typically exist at the project level, and should only be activated when you are in the project directory.
- pypi.org has a single namespace for packages. Once a name is taken, you can no longer use that name to publish a package. Anaconda.org is split into different "channels" so everyone can publish their own version of numpy on their own channel. Mixing and matching packages from different channels is usually a bad idea as dependencies may not be compatible. The conda-forge channel is a community maintained channel that aims to make most software available using a consistent set of build tools. A few years ago, I made a long video on how you can contribute packages to conda-forge, you can check it out here.

**Edit 14/12/2024**: on Reddit, Peter Wang from Anaconda shared a relevant talk with regards to Python packaging underneath a thread where this article was shared, which relates to this paragraph and which I can highly recommend: < https://www.youtube.com/watch?v=qA7NVwmx3gw>.

### Capabilities:

- Install any type of software and libraries at the user level
- Manage Python versions
- Manage conda environments

### Advantages:

- Multi-platform support
- Global package cache
- Packages are distributed as compiled binaries
- Robust dependency resolution algorithm
- Can also use pip inside Conda environments
- Option for global and shared environments

### Disadvantages:

- Slow, written in Python (should no longer be true since 2022, version 22.11, as pointed out by Kevin Markham, see this post)
- Serial downloads of packages (should no longer be true since 2022, version 22.11, as pointed out by Kevin Markham, see this post)
- Somewhat intrusive installation process (modifies shell config)
- Limited interoperability with the "main" Python ecosystem
- No lock file
- Building and distributing packages for Conda is painful (but this is also the case for wheels with extension modules)

## Mamba

Mamba is a strictly better Conda. It's a tool that aims to be a near drop-in replacement for Conda, but solve its biggest pain points: slow dependency resolution and parallel downloads. To speed up dependency resolution, it is implemented in C++ and uses a different algorithm.

The way to install Mamba used to be dodgy and has evolved significantly since its introduction. It used to be you first needed Conda, then you could install mamba into an environment with Conda from the conda-forge channel. The recommended approach is now to entirely sidestep Conda and install miniforge, or alternatively use micromamba.

### Capabilities:

- Same as Conda

### Advantages:

- Fast (dependency resolution + parallel downloads)
- Micromamba is distributed as single statically linked executable

### Disadvantages:

- Same as Conda except it is fast (if you use conda >= 22.11, then it uses the mamba resolver)

## conda-lock

conda-lock is the pip-tools of the Conda ecosystem. It introduces the much needed lock file mechanism to Conda environments. It can use both the Conda and Mamba resolvers to generate the lock file.

A very nice feature is that pip installable packages, defined in the `environment.yml`, are also included in the lock file. For this it uses the Poetry resolver.

A downside: it's written in Python, so again you need a good strategy for installing it. Luckily it is pip-installable, so you could use pipx or uv to install it as a stand-alone "tool" for your user.

A second minor downside: maintenance of the definition file is manual. There are no CLI facilities to manage your `environment.yml` file.

### Capabilities:

- Manage lock files

### Advantages:

- Simple, interoperable with Conda/Mamba
- Can handle multiple mutually exclusive environments
- Can also handle pip-installable packages in environment.yml

### Disadvantages:

- It's a Python tool
- Managing the definition files is a manual process

## Pixi

Pixi can be regarded as the uv of the Conda ecosystem. It's written in Rust and is distributed as a statically linked binary executable. Because the Conda ecosystem is in principle language agnostic, Pixi can even be used to manage dependencies for C++ projects, as long as dependencies are available on Anaconda.org. Pixi does fast dependency resolution and has a built-in lock-file mechanism. You can specify all your configuration and dependencies in the `pyproject.toml` (if you are building a Python project) or the custom `pixi.toml` (if you are building another project). It has a convenient CLI to manage your environment (add, remove, update packages).

Unlike Conda or Mamba, Pixi favors the "per project" environment approach. There is no base environment, but there is a global package cache. A cool concept is that your project can have multiple environments with different sets of dependencies, for example a default and a dev environment. In this way you can run your code against different environments. Of course on one machine you are constrained to a single operating system. Still, the configuration allows you to specify for which operating systems all dependencies should be resolved. You can even specify OS specific dependencies. It's also possible to specify python dependencies from pypi.org in your `pyproject.toml`, in exactly the same way as uv expects them, and these are also taken up in the lock file.

Another nice feature of Pixi is that it allows you to specify system dependencies expected by your package, for example glibc or Linux kernel versions. You won't be able to install those with Pixi, but at least it's explicit about these additional dependencies, and will refuse to create your environment if the system dependencies don't match.

Instead of being able to install Python packages as tools or run them as scripts like `uv`, Pixi has a concept of "tasks", which are similar to what you might define in a `Makefile`.

Unfortunately, Pixi does not help you build packages; it would be cool if Pixi could somehow facilitated building packages for conda-forge. If you are building a Python package, you can rely on typical build back-ends like setuptools or hatchling.

### Capabilities:

- Install packages
- Manage Python versions
- Manage (multiple) virtual environments
- Manage definition files
- Manage lock files
- Makefile-like project automation with "tasks"

### Advantages:

- Written in Rust: *blazingly fast*™, single binary, no external dependencies
- Multi-platform support (most CPU architectures + all major OS's: Linux, MacOS, Windows)
- Convenient CLI
- Global package cache
- Downloads Python binaries and any non-Python package available on anaconda.org (or private repositories)
- Can use `pyproject.toml` but also `pixi.toml` to configure Pixi and define dependencies
- Can select any build back-end

### Disadvantages:

- Limited compatibility with other tooling, though import and export of `environment.yml` is supported
- No global environments which deviates from Conda's philosophy

## Meta-issue of most tools: self-versions

An issue with most tools is that they do not a priori consider compatibility issues between older and newer versions of themselves. It would seem that tool developers assume that their configuration language will never change, but this is almost never the case. New versions of tools may deprecate some configuration and/or add new configuration syntax. The trouble is that the configuration is not labeled with any version, so a user has no idea which versions of the tool should be used to manage the project.

I have personally experienced issues that stem from this flaw when using Poetry. The `pyproject.toml` could not be parsed with the version of Poetry I was using because some configuration had been deprecated and removed. The `pyproject.toml` file itself does not indicate which versions of Poetry it is compatible with. Hence, quite some trial and error was required to find a suitable older Poetry version with which I could parse the file.

This is not meant to be a knock on Poetry. It is a systematic issue presented in most tools.

Tool developers should consider labeling all configuration and lock files with schema versions. An example is the `apiVersion` which is a part of every Kubernetes manifest. With the schema version declared, a tool can check whether it is compatible. If it is not, it can report to the user which older versions are compatible, or whether a newer version is required.

## Tooling beyond the Python ecosystem

We've talked about Python specific tooling for managing a project, but here I want to mention some additional tooling that is language agnostic and can be employed within a Python project.

### Containers (e.g. Docker)

At my previous place of employment, Conda was a dirty word. Instead, the preferred tooling for managing system dependencies and isolating environments were containers. Inside the container, basic Python tooling like pip, venv and pip-tools were used, although some teams also used Poetry.

Containers have become the industry standard for deploying services. Very crudely, it's a mini virtual machine that ships applications with all their dependencies in a single artifact. The only required dependencies are a Linux kernel and a container runtime. Once you have a container image, running it is a breeze and fully reproducible. Properly managed virtual environments can come close to reproducibility, but will always fall short due to undeclared dependencies outside the environment.

The main downsides of containers are the chore of building images and shipping them around (push/pull). Containers are built based on a file with build instructions, e.g. a Dockerfile for Docker containers. Writing this file so that a container builds quickly, correctly and in a reproducible way is an art. I've written at length about some of the limitations of containers here.

While containers primarily serve as a deployment mechanism, some people also advocate for its use as a tool for development. By isolating all dependencies from the host and developing in a container, we can guarantee that all devs see exactly the same environment. In this way, dev containers can serve as a replacement for virtual environments or Conda environments.

Personally, I was never really sold on development containers, mainly because rebuilding containers to add a single package can be a pain. Maybe a matter of personal taste.

While containers are fully reproducible at runtime, building them is often not reproducible. The build steps are imperative commands and often include `apt install` steps; this installs "system dependencies" with versions that depend on when the container is built.

### Capabilities:

- Run or develop software in isolation from the host. A lightweight VM.

### Advantages:

- Fully self-contained
- Fully reproducible at runtime
- Standardized
- High industry adoption and large amount of available tooling
- No external dependencies except for a container runtime

### Disadvantages:

- Building containers can be a pain
- You may need root access to the host to build and/or run containers
- Selecting the right base image for your project can get you stuck between a rock and a hard place
- Container images take up a lot of disk space
- Shipping images around can take up a lot of bandwidth
- Linux only inside containers (yes, Docker containers on Windows or Mac actually run on a hidden Linux virtual machine)
- Making containers interact with things outside the container is a pain
- It is very hard to make container builds reproducible

## Nix

As promised, I also briefly touch on Nix. Nix can be seen as a somewhat exotic alternative to all other tooling.  It's hard to describe in only a few paragraphs what Nix is. Nix was originally created as part of a PhD project, and the PhD thesis manuscript gives the best overview of what problems Nix aims to address. Essentially, it is all the issues that have been mentioned throughout this article. Nix was created with the specific aim of **reproducible, declarative software builds and deployments**.

For our purposes, Nix can be viewed like a package manager unlike any other. Installing packages imperatively with Nix is impossible; there is no `nix install` command. Instead, you must declare the "state" you wish to achieve using the Nix language, a domain specific functional programming language. The "state" can refer to some output of a build process, an environment, but can also be your entire operating system. The later is actually the idea behind NixOS, a Linux distribution that uses the Nix package manager. Your "state" code then describes your settings and which packages you want to have installed.

All this may sound weird, but it is a powerful idea. By describing your desired state in a declarative way and explicitly declaring all inputs, "build" output reproducibility is guaranteed. In this way, you can track your state in version control, always rebuild everything from scratch, and easily roll back to any point in history.

When everyone on the project uses Nix, you don't need any other package managers or project management tools. You maintain the dependencies, environment and build process all in Nix code, which is tracked together with the project code in version control. With this approach, everyone is guaranteed to have the exact same environment. You can use Nix with a project in any programming language. Nix does not distinguish between Python packages or system packages: everything is just software. There are no more implicit dependencies.

The main downside is that most other people don't use Nix for their projects. The Nix language is arcane and somewhat difficult. Since Nix is build on the core idea of reproducibility, it expects things to be done a certain way. That means it is probably the least interoperable with any other tool. To install a Python package that is not available on nixpkgs, you must first create a Nix expression for that packages. Documentation is scattered and often not up to date.

Despite all these drawbacks, Nix is a very interesting idea that puts reproducibility and dependency management front and center. Here is a very nice read on the revolutionary ideas behind Nix and also some of the challenges it faces.

Practically, you can use Nix in multiple ways to manage a Python project. If you've got an existing Python project that uses poetry or Conda, you can give conda-shell or poetry2nix a try. At this moment I would not advise Nix to the average Python developer, but it is certainly something to keep an eye on.

### Capabilities:

- Create fully isolated and reproducible environments with any software
- Create fully reproducible builds of artifacts

### Advantages:

- Laser focus on reproducibility of built artifacts
- Declarative approach to building packages and environments
- Lock-file mechanism for any project when using flakes
- Much more enjoyable dev experience for reproducible environments than dev containers

### Disadvantages:

- An ecosystem on its own, very poor interoperability
- Small community, poor documentation
- Arcane domain specific functional language to write configuration

### Edit 14/12/2024

Brandon Maier reached out to me via e-mail and pointed out that an important downside of Nix is that you must first be root to set it up. This is because the `/nix/store` needs to be initialized. He also informed me of two interesting projects in the `Nix` ecosystem: Nix-portable which aims to make Nix work without admin privileges, and devenv which aims to make Nix more accessible for managing a project lifecycle.

## What should you use?

Ok, but can't I just tell you what you should use? That's what these types of articles are all about right?

Unfortunately, I have to remain flaky: it depends. More specifically, it depends on:

- **what tools are already in use in existing projects?** There is value in standardization and familiarity, even if another tool might be objectively better for specific cases.
- **what are you allowed to do on your computer?** Are you root/admin? Can you install arbitrary software at the system level? Do you have privileges to run and build containers on your system?
- **what is your operating system, Windows or Unix-like?**
- **what you are building?** Do you need non-Python dependencies? Do you need complex or niche packages? What domain are you developing for: data science, sysadmin scripting, web development, back-end, research software, ...?

Based on these considerations, this would be my advice in the following scenarios, as of the time of writing (November 2024):

### Admin on system, only require Python dependencies

You can do anything on your system. You can install packages at the system level with homebrew, apt, ... and you can run and build Docker containers. You are building a web app, an API, a basic data science project, a data pipeline. All your dependencies are available on pypi.org - or a compatible artifact repository - and can be pip installed.

This scenario covers a large majority of Python projects. **For 95% of these types of projects I would recommend uv** as an all in one tool to deal with the entire project lifecycle

In the 5% case, if you have the very specific requirement of being able to maintain multiple mutually incompatible environments which can not be expressed in a single `pyproject.toml` file using environment markers, I would instead recommend **pip + venv + pip-tools** to manage dependencies and environments, and **pyenv/pyenv-win** to manage Python versions. **On Windows, conda, mamba or pixi may be the easier alternative** to creating virtual environments and managing the Python version.

You can develop directly on your system using a virtual environment, or if you prefer a higher level of isolation leverage a dev container. Your deployment target will likely be a container.

### Non admin on system, only require Python dependencies

You are trying to build the same type of project but in a more constrained environment. In this scenario, **I would recommend using uv or Pixi**. Both uv and Pixi can be downloaded as a binary, are available for most platforms, and don't require admin/root privileges to run them.

I would pick uv if you are absolutely sure you don't have to install any system dependencies with a different package manager.

Otherwise, I would recommend Pixi but with all python package dependencies declared in the `project.dependencies` table of the `pyproject.toml` file. Basically, this gets you a Conda virtual environment but with all packages installed from pypi.org. The benefit is that, as long as you have defined your dependencies in the standard way in `pyproject.toml`, you should still be able to switch to uv to manage your project. Pixi gives you the escape hatch of tapping into the Conda ecosystem for installing "system dependencies" at a user level, without any requirement for privilege escalation.

Additionally, by installing from pypi.org, you are guaranteed to have the latest versions and widest selection of packages. The Pixi documentation does state that resolving the environment becomes much slower if you declare "pip" dependencies, so do some testing to see whether this is a bottleneck.

Again, if you need to maintain **mutually incompatible environments**, you may instead look into the more basic tools: **pip + venv + pip-tools + pyenv** or **Mamba + conda-lock**.

If you do not wish to use the conda ecosystem, you can also explore uv + dev containers. This of course requires that you can build and run containers on your system.

### Admin on system, require complex dependencies

You can do anything on your system. You are building an application that requires niche dependencies that can not be pip-installed. You are working in scientific research, deep learning and/or are using GPUs and require the CUDA Toolkit.

In this situation **I would advise either uv or Pixi.** You can either go with uv to manage Python dependencies, and install the other dependencies with the system package manager or inside a dev container (prefered), or you can use Pixi to manage almost all dependencies (including CUDA but excluding the NVIDIA driver).  My personal preference would be to use Pixi, as I then don't need to deal with building containers, I don't need to rely on the system package manager, and more of my dependencies are explicit. It's very likely all dependencies are available on conda-forge. If packages are not available on conda-forge and if I would have to compile them from source, only then would I favor dev containers.

### Non admin on system, require complex dependencies

In this case, I would **highly recommend Pixi** for all the aforementioned reasons. The conda ecosystem allows you to install everything you would need without needing priviledged access to the system.

## Wrapping up and looking ahead

It's been a long journey. If you've read this far, congrats! I hope you now appreciate the nuance and complexity behind managing "a simple Python project". I hope I gave you a reasonable overview of tools you can look at, and you now understand which tools will serve you under which circumstances.

Tooling for managing Python projects has come very far, and I think tools like uv and Pixi will mostly replace all other tooling for almost every new project. In choosing between the two, the main choice that has to be made is whether one uses the Conda package ecosystem or the pypi ecosystem. Each has their reasons for existing. There are still a few cases where simpler tools that do fewer things are preferred, like if you have to maintain multiple mutually incompatible environments.

We also discussed Nix and containers, both of which aim to provide a framework for creating reproducible and isolated environments. However, both these tools can also be part of the deployment process. Containers are at this date far more popular and mature than Nix, and they are more interoperable with other tooling. However, I find that Nix provides a much more integrated and less frustrating development experience.

In this article, I did not discuss packaging, build systems and deployment of Python projects. I may cover this in another post.

---

#dependencies #conda #pip #pixi #uv

Nickname

Email (optional)

Reply...