Virtual Environments

When you work in Python projects you probably should use a **virtual environment** (or a similar mechanism) to isolate the packages you install for each project.

Info

If you already know about virtual environments, how to create them and use them, you might want to skip this section. 🤓

Tip

A **virtual environment** is different than an **environment variable**.

An **environment variable** is a variable in the system that can be used by programs.

A **virtual environment** is a directory with some files in it.

Info

This page will teach you how to use **virtual environments** and how they work.

If you are ready to adopt a **tool that manages everything** for you (including installing Python), try uv [↪].

Create a Project

First, create a directory for your project.

What I normally do is that I create a directory named `code` inside my home/user directory.

And inside of that I create one directory per project.

bash

```
```

Create a Virtual Environment

When you start working on a Python project **for the first time**, create a virtual environment **inside your project**.

Tip

You only need to do this **once per project**, not every time you work.

venv

To create a virtual environment, you can use the `venv` module that comes with Python.

```bash
```

What that command means

- `python`: use the program called `python`
- `-m`: call a module as a script, we'll tell it which module next
- `venv`: use the module called `venv` that normally comes installed with Python
- `.venv`: create the virtual environment in the new directory `.venv`

uv

If you have `uv` [↪] installed, you can use it to create a virtual environment.

```
bash

$ uv venv

restart ↻
```

Tip

By default, `uv` will create a virtual environment in a directory called `.venv`.

But you could customize it passing an additional argument with the directory name.

That command creates a new virtual environment in a directory called `.venv`.

`.venv` or other name

You could create the virtual environment in a different directory, but there's a convention of calling it `.venv`.

Activate the Virtual Environment

Activate the new virtual environment so that any Python command you run or package you install uses it.

Tip

Do this **every time** you start a **new terminal session** to work on the project.

Linux, macOS

```
bash
```

Windows PowerShell

```
bash
                                                    fast →
$ .venv\Scripts\Activate.
```

Windows Bash

Or if you use Bash for Windows (e.g. Git Bash [↪]):

```
bash
                                                    fast →
$ source .venv/Scripts/ac
```

> **Tip**
>
> Every time you install a **new package** in that environment, **activate** the environment again.
>
> This makes sure that if you use a **terminal (CLI) program** installed by that package, you use the one from your virtual environment and not any other that could be installed globally, probably with a different version than what you need.

Check the Virtual Environment is Active

Check that the virtual environment is active (the previous command worked).

> **Tip**
>
> This is **optional**, but it's a good way to **check** that everything is working as expected and you are using the virtual environment you intended.

Linux, macOS, Windows Bash

```
bash
```

```
```

If it shows the `python` binary at `.venv/bin/python`, inside of your project (in this case `awesome-project`), then it worked. 🎉

Windows PowerShell

```bash
fast →

$ Get-Command python

C:\Users\user\code\awesome-project\.venv\Scripts\python
```

If it shows the `python` binary at `.venv\Scripts\python`, inside of your project (in this case `awesome-project`), then it worked. 🎉

Upgrade `pip`

Tip

If you use `uv` [↪] you would use it to install things instead of `pip`, so you don't need to upgrade `pip`. 😎

If you are using `pip` to install packages (it comes by default with Python), you should **upgrade** it to the latest version.

Many exotic errors while installing a package are solved by just upgrading `pip` first.

Tip

You would normally do this **once**, right after you create the virtual environment.

Make sure the virtual environment is active (with the command above) and then run:

```bash
```

Add `.gitignore`

If you are using **Git** (you should), add a `.gitignore` file to exclude everything in your `.venv` from Git.

Tip

If you used `uv` [↪] to create the virtual environment, it already did this for you, you can skip this step. 😎

Tip

Do this **once**, right after you create the virtual environment.

bash

What that command means

- `echo "*"`: will "print" the text `*` in the terminal (the next part changes that a bit)
- `>`: anything printed to the terminal by the command to the left of `>` should not be printed but instead written to the file that goes to the right of `>`
- `.gitignore`: the name of the file where the text should be written

And `*` for Git means "everything". So, it will ignore everything in the `.venv` directory.

That command will create a file `.gitignore` with the content:

```
*
```

Install Packages

After activating the environment, you can install packages in it.

Tip

Do this **once** when installing or upgrading the packages your project needs.

If you need to upgrade a version or add a new package you would **do this again**.

Install Packages Directly

If you're in a hurry and don't want to use a file to declare your project's package requirements, you can install them directly.

Tip

It's a (very) good idea to put the packages and versions your program needs in a file (for example `requirements.txt` or `pyproject.toml`).

`pip`

```bash
```

`uv`

If you have `uv` [↪]:

```bash
                                                        fast →
$ uv pip install "fastapi
```

Install from `requirements.txt`

If you have a `requirements.txt`, you can now use it to install its packages.

`pip`

```
bash
```

`uv`

If you have `uv` [↪]:

```
bash

                                                          fast →
   $ uv pip install -r requi ▌
```

`requirements.txt`

A `requirements.txt` with some packages could look like:

```
fastapi[standard]==0.113.0
pydantic==2.8.0
```

Run Your Program

After you activated the virtual environment, you can run your program, and it will use the Python inside of your virtual environment with the packages you installed there.

```
bash
```

Configure Your Editor

You would probably use an editor, make sure you configure it to use the same virtual environment you created (it will probably autodetect it) so that you can get autocompletion and inline errors.

For example:

- VS Code [↪]
- PyCharm [↪]

Tip

You normally have to do this only **once**, when you create the virtual environment.

Deactivate the Virtual Environment

Once you are done working on your project you can **deactivate** the virtual environment.

```bash
```

This way, when you run `python` it won't try to run it from that virtual environment with the packages installed there.

Ready to Work

Now you're ready to start working on your project.

Tip

Do you want to understand what's all that above?

Continue reading. 👇 🤓

Why Virtual Environments

To work with FastAPI you need to install Python [↪].

After that, you would need to **install** FastAPI and any other **packages** you want to use.

To install packages you would normally use the `pip` command that comes with Python (or similar alternatives).

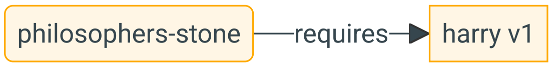
Nevertheless, if you just use `pip` directly, the packages would be installed in your **global Python environment** (the global installation of Python).

The Problem

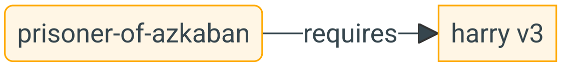
So, what's the problem with installing packages in the global Python environment?

At some point, you will probably end up writing many different programs that depend on **different packages**. And some of these projects you work on will depend on **different versions** of the same package. 😱

For example, you could create a project called `philosophers-stone`, this program depends on another package called **`harry`, using the version `1`**. So, you need to install `harry`.

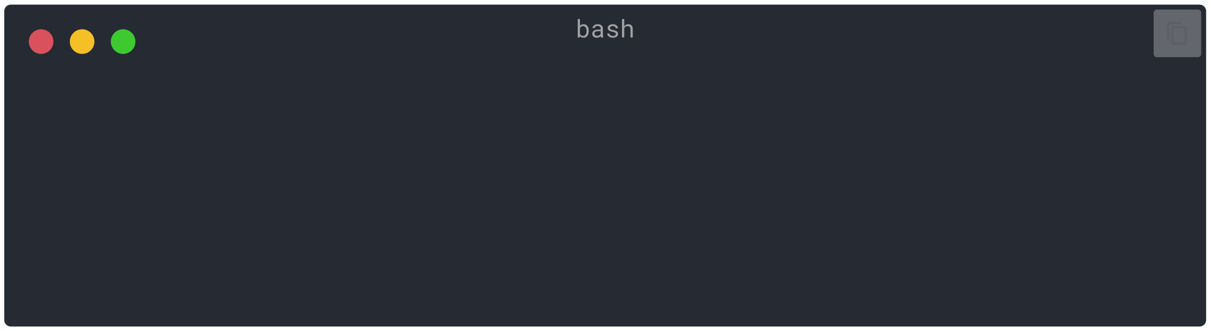


Then, at some point later, you create another project called `prisoner-of-azkaban`, and this project also depends on `harry`, but this project needs **`harry` version `3`**.

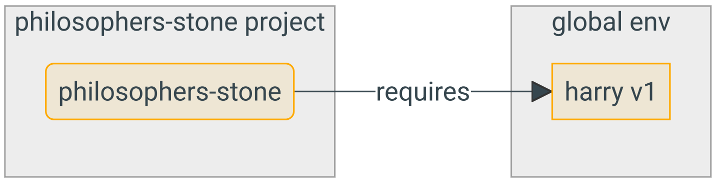


But now the problem is, if you install the packages globally (in the global environment) instead of in a local **virtual environment**, you will have to choose which version of `harry` to install.

If you want to run `philosophers-stone` you will need to first install `harry` version `1`, for example with:



And then you would end up with `harry` version `1` installed in your global Python environment.

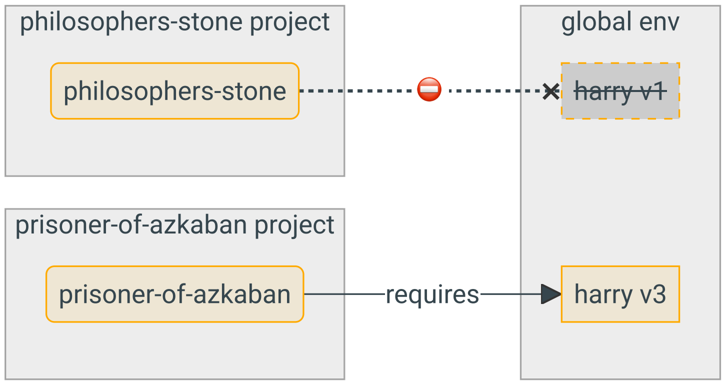


But then if you want to run `prisoner-of-azkaban`, you will need to uninstall `harry` version `1` and install `harry` version `3` (or just installing version `3` would automatically uninstall version `1`).

```bash
```

And then you would end up with `harry` version `3` installed in your global Python environment.

And if you try to run `philosophers-stone` again, there's a chance it would **not work** because it needs `harry` version `1`.



Tip

It's very common in Python packages to try the best to **avoid breaking changes** in **new versions**, but it's better to be safe, and install newer versions intentionally and when you can run the tests to check everything is working correctly.

Now, imagine that with **many** other **packages** that all your **projects depend on**. That's very difficult to manage. And you would probably end up running some projects with some **incompatible versions** of the packages, and not knowing why something isn't working.

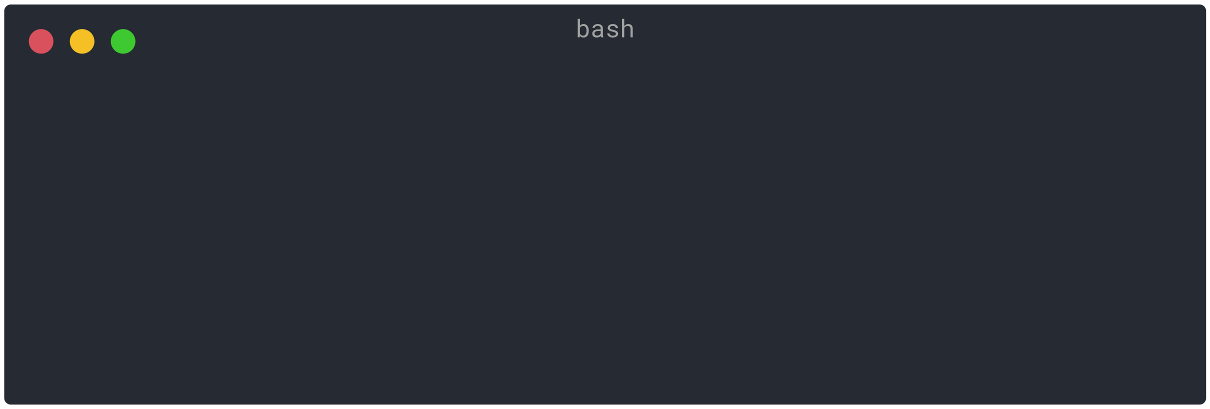
Also, depending on your operating system (e.g. Linux, Windows, macOS), it could have come with Python already installed. And in that case it probably had some packages pre-installed with some specific versions **needed by your system**. If you install packages in the global Python environment, you could end up **breaking** some of the programs that came with your operating system.

Where are Packages Installed

When you install Python, it creates some directories with some files in your computer.

Some of these directories are the ones in charge of having all the packages you install.

When you run:



That will download a compressed file with the FastAPI code, normally from PyPI [↪].

It will also **download** files for other packages that FastAPI depends on.

Then it will **extract** all those files and put them in a directory in your computer.

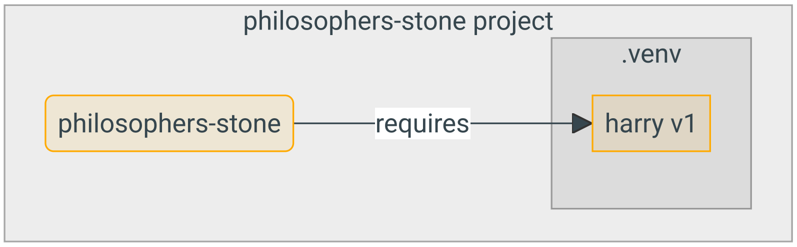
By default, it will put those files downloaded and extracted in the directory that comes with your Python installation, that's the **global environment**.

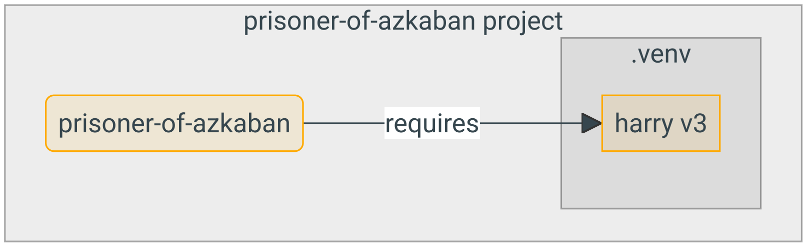
What are Virtual Environments

The solution to the problems of having all the packages in the global environment is to use a **virtual environment for each project** you work on.

A virtual environment is a **directory**, very similar to the global one, where you can install the packages for a project.

This way, each project will have its own virtual environment (`.venv` directory) with its own packages.





What Does Activating a Virtual Environment Mean

When you activate a virtual environment, for example with:

Linux, macOS

```
bash
```

Windows PowerShell

```
bash

                                                        fast →
$ .venv\Scripts\Activate.
```

Windows Bash

Or if you use Bash for Windows (e.g. Git Bash [↪]):

```
bash

                                                        fast →
$ source .venv/Scripts/ac
```

That command will create or modify some environment variables ↪ that will be available for the next commands.

One of those variables is the `PATH` variable.

Tip

You can learn more about the `PATH` environment variable in the Environment Variables ↪ section.

Activating a virtual environment adds its path `.venv/bin` (on Linux and macOS) or `.venv\Scripts` (on Windows) to the `PATH` environment variable.

Let's say that before activating the environment, the `PATH` variable looked like this:

Linux, macOS

```
/usr/bin:/bin:/usr/sbin:/sbin
```

That means that the system would look for programs in:

- `/usr/bin`
- `/bin`
- `/usr/sbin`
- `/sbin`

Windows

```
C:\Windows\System32
```

That means that the system would look for programs in:

- `C:\Windows\System32`

After activating the virtual environment, the `PATH` variable would look something like this:

Linux, macOS

```
/home/user/code/awesome-project/.venv/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

That means that the system will now start looking first for programs in:

```
/home/user/code/awesome-project/.venv/bin
```

before looking in the other directories.

So, when you type `python` in the terminal, the system will find the Python program in

```
/home/user/code/awesome-project/.venv/bin/python
```

and use that one.

Windows

```
C:\Users\user\code\awesome-project\.venv\Scripts;C:\Windows\System32
```

That means that the system will now start looking first for programs in:

```
C:\Users\user\code\awesome-project\.venv\Scripts
```

before looking in the other directories.

So, when you type `python` in the terminal, the system will find the Python program in

```
C:\Users\user\code\awesome-project\.venv\Scripts\python
```

and use that one.

An important detail is that it will put the virtual environment path at the **beginning** of the `PATH` variable. The system will find it **before** finding any other Python available. This way, when you run `python`, it will use the Python **from the virtual environment** instead of any other `python` (for example, a `python` from a global environment).

Activating a virtual environment also changes a couple of other things, but this is one of the most important things it does.

Checking a Virtual Environment

When you check if a virtual environment is active, for example with:

Linux, macOS, Windows Bash

```
bash
```

Windows PowerShell

```
bash

                                                           fast →
$ Get-Command python

C:\Users\user\code\awesome-project\.venv\Scripts\python
```

That means that the `python` program that will be used is the one **in the virtual environment**.

You use `which` in Linux and macOS and `Get-Command` in Windows PowerShell.

The way that command works is that it will go and check in the `PATH` environment variable, going through **each path in order**, looking for the program called `python`. Once it finds it, it will **show you the path** to that program.

The most important part is that when you call `python`, that is the exact "`python`" that will be executed.

So, you can confirm if you are in the correct virtual environment.

Tip

It's easy to activate one virtual environment, get one Python, and then **go to another project**.

And the second project **wouldn't work** because you are using the **incorrect Python**, from a virtual environment for another project.

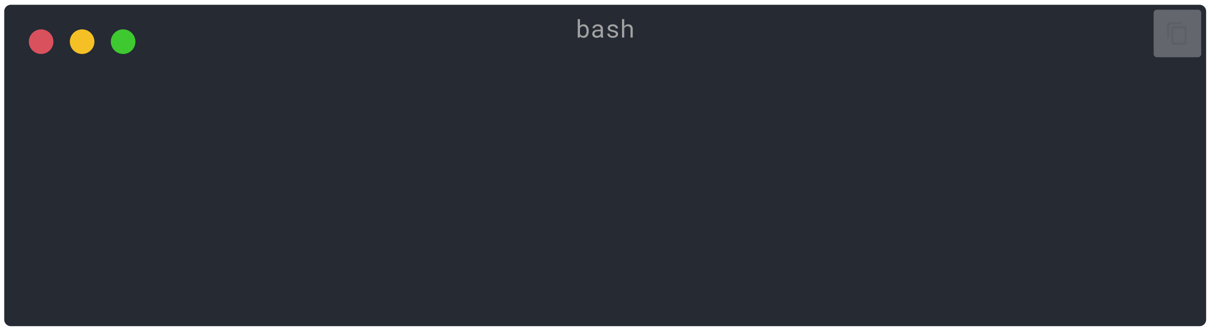
It's useful being able to check what `python` is being used. 🤓

Why Deactivate a Virtual Environment

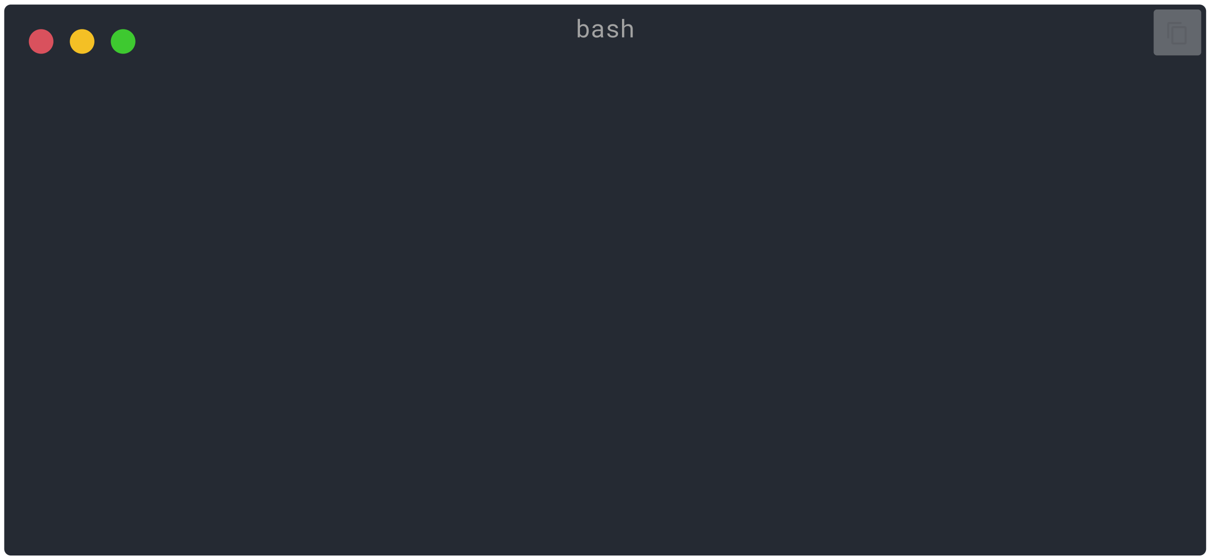
For example, you could be working on a project `philosophers-stone`, **activate that virtual environment**, install packages and work with that environment.

And then you want to work on **another project** `prisoner-of-azkaban`.

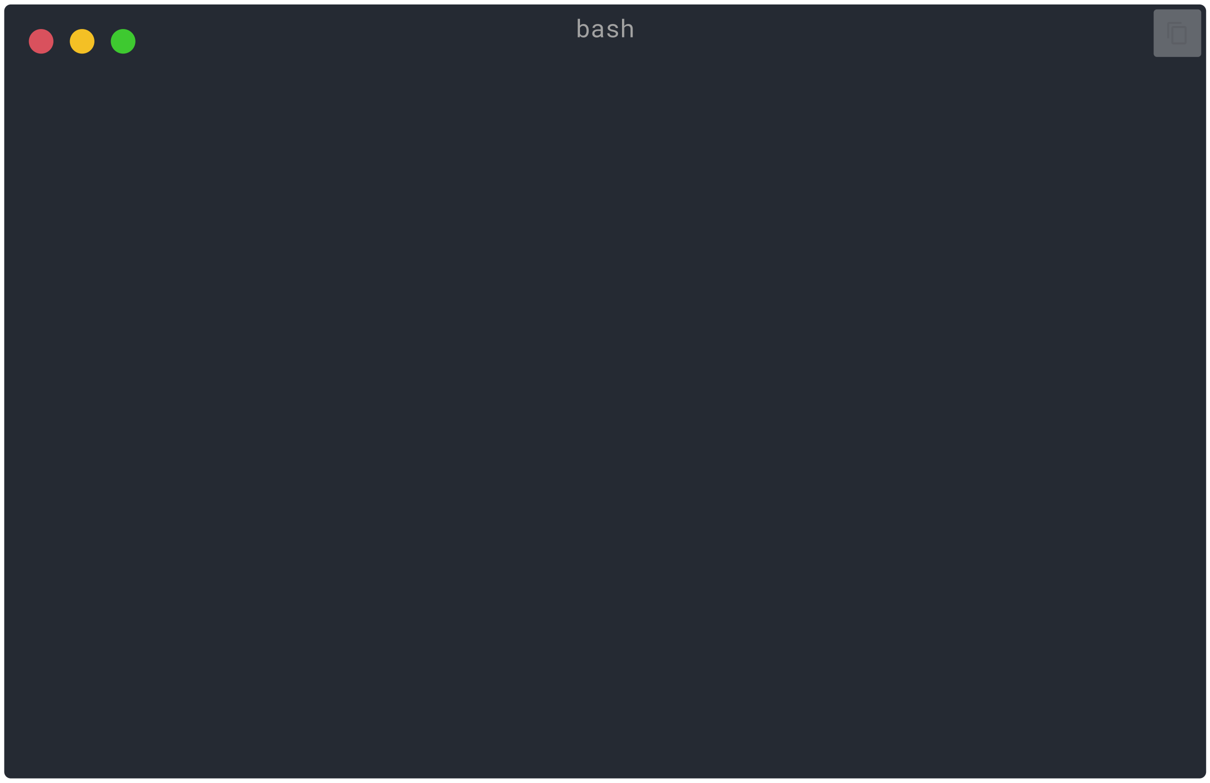
You go to that project:



If you don't deactivate the virtual environment for `philosophers-stone`, when you run `python` in the terminal, it will try to use the Python from `philosophers-stone`.



But if you deactivate the virtual environment and activate the new one for `prisoner-of-askaban` then when you run `python` it will use the Python from the virtual environment in `prisoner-of-azkaban`.



Alternatives

This is a simple guide to get you started and teach you how everything works **underneath**.

There are many **alternatives** to managing virtual environments, package dependencies (requirements), projects.

Once you are ready and want to use a tool to **manage the entire project**, packages dependencies, virtual environments, etc. I would suggest you try uv [↪].

`uv` can do a lot of things, it can:

- **Install Python** for you, including different versions
- Manage the **virtual environment** for your projects
- Install **packages**
- Manage package **dependencies and versions** for your project
- Make sure you have an **exact** set of packages and versions to install, including their dependencies, so that you can be sure that you can run your project in production exactly the same as in your computer while developing, this is called **locking**
- And many other things

Conclusion

If you read and understood all this, now **you know much more** about virtual environments than many developers out there. 🤓

Knowing these details will most probably be useful in a future time when you are debugging something that seems complex, but you will know **how it all works underneath**. 😎