Python venv: How To Create, Activate, Deactivate, And Delete

python.land/virtual-environments/virtualenv

August 23, 2024

Python virtual environments allow you to install Python packages in a location isolated from the rest of your system instead of installing them system-wide. Let’s look at how to use Python venv’s, short for Python virtual environments, also abbreviated as *virtualenv*.

In this article, you will learn:

- The advantages of using virtual environments
- How to create a venv
- How to activate and deactivate it
- Different ways to delete or remove a venv
- How a venv works internally

Follow the course

I’m convinced this article gives you the solution to your problem, but how annoying is it to not exactly understand what’s going on? Stop feeling like a voodoo coder and learn this stuff properly once and for all. Our Python Fundamentals course extensively explains **Modules and packages**, **Virtual environments,** and **Package managers**. Give it a try; I assure you that you’ll like it!

Why you need virtual environments

There are multiple reasons why virtual environments are a good idea, and this is also why I’m telling you about them before we continue to the part where we start installing 3rd party packages. Let’s go over them one by one.

Preventing version conflicts

You could argue that installing third-party packages system-wide is very efficient. After all, you only need to install it once and can use the package from multiple Python projects, saving you precious time and disk space. There’s a problem with this approach that may start to unfold weeks or months later, however.

Suppose your project, `Project A`, is written against a specific version of `library X`. In the future, you might need to upgrade library X. Say, for example, you need the latest version for another project you started, called Project B. You upgrade library X to the latest version, and project B works fine. Great! But once you did this, it turns out your `Project A` code broke badly. After all, APIs can change significantly on major version upgrades.

A virtual environment fixes this problem by isolating your project from other projects and system-wide packages. You install packages inside this virtual environment specifically for the project you are working on.

Easy to reproduce and install

Virtual environments make it easy to define and install the packages specific to your project. Using a *requirements.txt* file, you can define exact version numbers for the required packages to ensure your project will always work with a version tested with your code. This also helps other users of your software since a virtual environment helps others reproduce the exact environment for which your software was built.

Works everywhere, even when not administrator (root)

If you're working on a shared host, like those at a university or a web hosting provider, you won't be able to install system-wide packages since you don't have the administrator rights to do so. In these places, a virtual environment allows you to install anything you want locally in your project.

Virtual environments vs. other options

There are other options to isolate your project:

1. In the most extreme case, you could buy a second PC and run your code there. Problem fixed! It was a bit expensive, though!
2. A virtual machine is a much cheaper option but still requires installing a complete operating system—a bit of a waste as well for most use cases.
3. Next in line is containerization, with the likes of Docker and Kubernetes. These can be very powerful and are a good alternative.

Still, there are many cases when we're just creating small projects or one-off scripts. Or perhaps you just don't want to containerize your application. It's another thing you need to learn and understand, after all. Whatever the reason is, virtual environments are a great way to isolate your project's dependencies.

How to create a Python venv

There are several ways to create a Python virtual environment, depending on the Python version you are running.

Before you read on, I want to point you to two other tools, Python Poetry and Pipenv. Both these tools combine the functionality of tools you are about to learn: virtualenv and pip. On top of that, they add several extras, most notably their ability to do proper dependency resolution.

To better understand virtual environments, I recommend you learn the basics first though, using this article. I just want to ensure that you know there are nicer ways to manage your packages, dependencies, and virtual environments.

Python 3.4 and above

If you are running Python 3.4+, you can use the venv module baked into Python:

```
python -m venv <directory>
```

This command creates a venv in the specified directory and copies pip into it as well. If you're unsure what to call the directory: `venv` is a commonly seen option; it doesn't leave anyone guessing what it is. So the command, in that case, would become:

```
python -m venv venv
```

A little further in this article, we'll look closely at the just-created directory. But let's first look at how to activate this virtual environment.

All other Python versions

The alternative that works for any Python version is using the virtualenv package. You may need to install it first with pip install:

```
pip install virtualenv
```

Once installed, you can create a virtual environment with:

```
virtualenv [directory]
```

Python venv activation

How you activate your virtual environment depends on the OS you're using.

Windows venv activation

To activate your venv on Windows, you need to run a script that gets installed by venv. If you created your venv in a directory called `myenv`, the command would be:

```
# In cmd.exe
venv\Scripts\activate.bat
# In PowerShell
venv\Scripts\Activate.ps1
```

Linux and MacOS venv activation

On Linux and MacOS, we activate our virtual environment with the source command. If you created your venv in the `myvenv` directory, the command would be:

```
$ source myvenv/bin/activate
```

That's it! We're ready to rock! You can now install packages with pip, but I advise you to keep reading to understand the venv better first.

How a Python venv works

When you activate a virtual environment, your `PATH` variable is changed. On Linux and MacOS, you can see it for yourself by printing the path with `echo $PATH`. On Windows, use `echo %PATH%` (in cmd.exe) or `$Env:Path` (in PowerShell). In my case, on Windows, it looks like this:

```
C:\Users\erik\Dev\venv\Scripts;C:\Program Files\PowerShell\7;C:\Program
Files\AdoptOpen....
```

It's a big list, and I only showed the beginning of it. As you can see, the Scripts directory of my venv is put in front of everything else, effectively overriding all the system-wide Python software.

So what does this PATH variable do?

When you enter a command that can't be found in the current working directory, your OS starts looking at all the paths in the PATH variable. It's the same for Python. When you import a library, Python looks in your PATH for library locations. And that's where our venv-magic happens: if your venv is there in front of all the other paths, the OS will look there first before looking at system-wide directories like /usr/bin. Hence, anything installed in our venv is found first, and that's how we can override system-wide packages and tools.

What's inside a venv?

If you take a look inside the directory of your venv, you'll see a folder structure like this on Windows:

```
.
├── Include
├── Lib
│   └── site-packages
├── pyvenv.cfg
└── Scripts
├── activate
├── activate.bat
├── Activate.ps1
├── deactivate.bat
├── pip3.10.exe
├── pip3.exe
├── pip.exe
├── python.exe
└── pythonw.exe
```

And on Linux and MacOS:

```
.
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.7
│   ├── pip
│   ├── pip3
│   ├── pip3.7
│   ├── python -> python3
│   └── python3 -> /usr/local/bin/python3
├── include
├── lib
│   └── python3.7
│       └── site-packages
└── pyvenv.cfg
```

Virtualenv directory tree

You can see that:

- The Python command is made available as both python and python3 (on Linux and MacOS), and the version is pinned to the version with which you created the venv by creating a symlink to it.
- On Windows, the Python binary is copied over to the scripts directory.
- All packages you install end up in the site-packages directory.
- We have activation scripts for multiple shell types (bash, csh, fish, PowerShell)
- Pip is available under pip and pip3, and even more specifically under the name `pip3.7` because I had a Python 3.7 installation at the time of writing this.

Deactivate the Python venv

Once you have finished working on your project, it’s a good habit to deactivate its venv. By deactivating, you leave the virtual environment. Without deactivating your venv, all other Python code you execute, even if it is outside your project directory, will also run inside the venv.

Luckily, deactivating your virtual environment couldn’t be simpler. Just enter this: `deactivate`. It works the same on all operating systems.

Deleting a Python venv

You can completely remove a virtual environment, but how you do that depends on what you used to create the venv. Let’s look at the most common options.

Delete a venv created with Virtualenv or python -m venv

There's no special command to delete a virtual environment if you used `virtualenv` or `python -m venv` to create your virtual environment, as is demonstrated in this article. When creating the virtualenv, you gave it a directory to create this environment in.

If you want to delete this virtualenv, deactivate it first and then remove the directory with all its content. On Unix-like systems and in Windows Powershell, you would do something like this:

```
deactivate
# If your virtual environment is in a directory called 'venv':
rm -r venv
```

Delete a venv with Pipenv

If you used Pipenv to create the venv, it's a lot easier. You can use the following command to delete the current venv:

```
pipenv --rm
```

Make sure you are inside the project directory. In other words, the directory where the `Pipenv` and `Pipenv.lock` files reside. This way, pipenv knows which virtual environment it has to delete.

If this doesn't work, you can get a little nastier and manually remove the venv. First, ask pipenv where the actual virtualenv is located with the following command:

```
pipenv --env
/home/username/.local/share/virtualenvs/yourproject-IogVUtsM
```

It will output the path to the virtual environment and all of its files and look similar to the example above. The next step is to remove that entire directory, and you're done.

Delete a venv with Poetry

If you created the virtualenv with Poetry, you can list the available venvs with the following command:

```
poetry env list
```

You'll get a list like this:

```
test-O3eWbxRl-py2.7
test-O3eWbxRl-py3.6
test-O3eWbxRl-py3.7 (Activated)
```

You can remove the environment you want with the `poetry env remove` command. You need to specify the exact name from the output above, for example:

```
poetry env remove test-O3eWbxRl-py3.7
```

Follow the course

Stop feeling like a voodoo coder and learn this stuff properly once and for all. Our Python Fundamentals course extensively explains **Modules and packages**, **Virtual environments,** and **Package managers**. Give it a try; I assure you that you'll like it!

Learn more

This article is part of a free Python Tutorial. You can browse the tutorial with the navigation buttons at the top and bottom of the article or use the navigation menu. Want to learn more? Here are some great follow-up reads:

- Next up: how to install packages with pip inside your venv
- Pipenv is a better way of managing your venv and packages.
- Learn the most common Linux commands (like cd, mkdir, pwd, etcetera)
- Official venv documentation: If you want to know all the details and command-line options

Conclusion

You learned how to create, activate, deactivate, and delete virtual environments. We also looked behind the curtains to see why and how a venv works. Now that you know how to create a venv, you need to learn how to install packages inside it. After that, I strongly recommend you to learn about Pipenv or Poetry. These tools combine the management of your virtual environment with proper package and dependency management.