**Virtual Environment in Python**

A **virtual environment** in Python is a self-contained directory that includes a specific version of Python and all the dependencies needed for a project. This means you can create different virtual environments for different projects, each with its own dependencies, without conflicts.

**🔹 Why Do We Need a Virtual Environment?**

Imagine you have two projects:

* **Project A** needs Django 3.2
* **Project B** needs Django 4.0

If you install Django globally, one project might break when you install a different version. A **virtual environment** solves this problem by keeping dependencies isolated.

**Check if Python is Installed**

Open a terminal or command prompt and run:

python --version

or

python3 --version

If Python is installed, it will show something like:

Python 3.10.12

**Create a Virtual Environment**

Run the following command:

python -m venv my\_env

This creates a folder named my\_env in your current directory.

📌 **What happens when you create a virtual environment?**

* It generates a folder containing:
  + A **Python interpreter** specific to that environment.
  + A **bin (or Scripts on Windows) folder** with executables.
  + A **lib folder** that contains installed packages.

**Activate the Virtual Environment**

Before installing packages, you need to **activate** the environment.

* **On Windows (Command Prompt):**

my\_env\Scripts\activate

* **On macOS/Linux:**

source my\_env/bin/activate

After activation, you’ll see (my\_env) before your terminal prompt, indicating that the environment is active:

(my\_env) user@machine:~$

**Install Packages in the Virtual Environment**

Once activated, install packages as usual:

pip install numpy

To check installed packages:

pip list

**Deactivate the Virtual Environment**

When you're done working, deactivate the virtual environment by running:

deactivate

Now, you're back to your system’s global Python environment.

**Remove a Virtual Environment**

If you no longer need a virtual environment, you can delete it:

rm -rf my\_env

rd /s /q my\_env

**🔹 Example: Using Virtual Environments in Projects**

Let's say you're working on two projects:

1. **Project A (using Flask)**
2. **Project B (using Django)**

Without a virtual environment, installing Flask for one project and Django for another might cause conflicts.

✅ Solution:

# Create separate environments for each project

python -m venv flask\_env

python -m venv django\_env

# Activate and install Flask in one

source flask\_env/bin/activate

pip install flask

deactivate

# Activate and install Django in another

source django\_env/bin/activate

pip install django

deactivate

Now, each project has its own dependencies!

**🔹 Bonus: Managing Environments with virtualenv and pipenv**

* **venv** is built into Python (Python 3.3+).
* **virtualenv** is an alternative tool with more features:

pip install virtualenv

virtualenv my\_env

* **pipenv** combines pip and virtualenv for better package management:

pip install pipenv

pipenv install flask

pipenv shell

**🔹 Summary:**

|  |  |
| --- | --- |
| **Command** | **Purpose** |
| python -m venv my\_env | Create a virtual environment |
| source my\_env/bin/activate (Linux/macOS)  or  my\_env\Scripts\activate (Windows) | Activate environment |
| pip install package\_name | Install a package |
| pip list | Check installed packages |
| deactivate | Exit the environment |
| rm -rf my\_env or rd /s /q my\_env | Delete the environment |