# Virtual Environments in Python

## 1. What Is a Virtual Environment?

A **virtual environment** is an isolated Python environment created for a specific project.

It allows each project to have its own:

- Python packages
- package versions
- dependencies
- project configuration

Instead of installing every package globally on your computer, you create a separate environment for the project.

### Simple Example

Imagine you have two Python projects:

```
Project A
   Django 4.2

Project B
   Django 5.2
```

If both projects use your system-wide Python installation, different package versions can cause conflicts.

With virtual environments:

```
Project A
   □□□ .venv
      □□□ Django 4.2
```

```
Project B
   □□□ .venv
      □□□ Django 5.2
```

Each project gets its own dependency space.

---

## 2. Why Do We Need Virtual Environments?

Without virtual environments, packages are commonly installed globally.

For example:

```
pip install requests
pip install flask
pip install django
```

Over time, your computer might have:

```
requests
flask
django
numpy
pandas
...
```

Different projects may require different versions.

That creates problems.

### Problem 1: Version Conflicts

Project A requires:

`requests 2.31`

Project B requires:

`requests 2.32`

A single global environment may not satisfy both projects cleanly.

### Problem 2: Updating One Project Can Break Another

You upgrade a package for Project A:

`pip install --upgrade package`

Project B may suddenly stop working.

### Problem 3: Unnecessary Packages

A global environment can become full of packages that only belong to old projects.

### Problem 4: Reproducibility

Someone else may clone your project but have different package versions.

The project may work on your computer but fail on theirs.

---

## 3. The Main Idea

Think of a virtual environment as a **private Python workspace for one project**.

```
Computer
□
□□□ Project A
□   □□□ .venv
□       □□□ Python packages
□       □□□ project dependencies
□
□□□ Project B
□   □□□ .venv
□       □□□ Python packages
□       □□□ project dependencies
□
□□□ Global Python
```

Each project can manage its dependencies independently.

---

## 4. What Is `venv`?

Python includes a built-in module called:

`venv`

It is used to create virtual environments.

You normally do not need to install `venv` separately.

The basic command is:

```
python -m venv .venv
```

Here:

`python`

runs Python.

`-m`

tells Python to run a module.

`venv`

is the virtual environment module.

`.venv`

is the directory where the environment will be created.

---

## 5. Creating Your First Virtual Environment

Suppose your project looks like this:

```
my_project/
   app.py
```

Open your terminal inside the project directory.

Run:

```
python -m venv .venv
```

You will now have:

```
my_project/
□
□□□ .venv/
□□□ app.py
```

The .venv directory contains the isolated environment.

---

## 6. Why Is .venv Usually Used?

You could create an environment like:

```
python -m venv environment
```

or:

```
python -m venv my_environment
```

But:

```
.venv
```

is a common convention.

The dot also makes it a hidden-style directory on many systems.

A typical Python project therefore looks like:

```
my_project/
□□□ .venv/
□□□ app.py
□□□ README.md
```

---

## 7. Activating a Virtual Environment

Creating an environment does not automatically activate it.

You need to activate it before using it.

The command depends on your operating system and shell.

### Windows Command Prompt

```
.venv\Scripts\activate
```

### Windows PowerShell

```
.venv\Scripts\Activate.ps1
```

### macOS / Linux

```
source .venv/bin/activate
```

After activation, your terminal usually shows something like:

```
(.venv)
```

For example:

```
(.venv) C:\Projects\my_project>
```

This tells you that the virtual environment is currently active.

---

## 8. What Does Activation Actually Do?

Activation changes which Python and package tools your terminal uses by default.

Before activation:

`python`

may point to your system Python.

After activation:

`python`

points to the Python executable associated with the virtual environment.

Similarly:

`pip`

points to the environment's package installer.

This is why you can install packages without affecting unrelated projects.

---

## 9. Installing Packages Inside a Virtual Environment

Once the environment is active:

`pip install requests`

The package is installed into the active environment.

You can then use it:

```
import requests

response = requests.get("https://example.com")

print(response.status_code)
```

The important idea is:

```
Project
  
Activated .venv
  
pip install
  
Package belongs to this environment
```

---

## 10. Checking Installed Packages

Run:

```
pip list
```

You may see:

```
Package    Version
---------- -------
pip        25.x
```

```
requests   2.x
```

You can also use:

```
pip show requests
```

This gives information about a specific package.

For example:

```
Name: requests
Version: ...
Location: ...
```

---

# 11. Checking Which Python Is Being Used

This is extremely useful when debugging environments.

### Windows

```
where python
```

### macOS / Linux

```
which python
```

You can also run:

```
python -c "import sys; print(sys.executable)"
```

This shows the exact Python executable being used.

For example:

```
C:\Projects\my_project\.venv\Scripts\python.exe
```

If you see `.venv` in the path, you are using the project's virtual environment.

---

## 12. Checking the Python Version

Run:

```
python --version
```

Example:

```
Python 3.13.2
```

You can also check from Python:

```
import sys

print(sys.version)
```

Or:

```
import sys

print(sys.executable)
```

---

## 13. Installing a Specific Package Version

Suppose your project needs a specific version of `requests`.

You can write:

```
pip install requests==2.31.0
```

You can also specify a minimum version:

```
pip install "requests>=2.31"
```

Or a compatible range:

```
pip install "requests>=2.31,<3"
```

Version constraints are important when projects depend on specific package behavior.

---

## 14. Updating a Package

To upgrade a package:

```
pip install --upgrade requests
```

Or:

```
pip install -U requests
```

Both commands can be used for upgrading.

---

## 15. Uninstalling a Package

To remove a package:

```
pip uninstall requests
```

`pip` will normally ask for confirmation.

---

## 16. `pip` and Virtual Environments

It is important to understand the relationship.

`venv` creates the environment.

`pip` installs and manages packages inside that environment.

Think of it like:

```
venv
 
Creates isolated environment

pip
 
Manages packages inside environment
```

They solve different problems.

---

## 17. What Is `requirements.txt`?

A virtual environment is local to your machine.

But other developers need to know which packages your project requires.

This is where:

`requirements.txt`

is useful.

Example:

```
requests==2.32.3
flask==3.1.0
python-dotenv==1.1.0
```

It describes the project's Python dependencies.

---

## 18. Creating `requirements.txt`

With your virtual environment activated, run:

```
pip freeze > requirements.txt
```

This writes installed package versions into the file.

Example:

```
requests==2.32.3
flask==3.1.0
```

Now your project might look like:

```
my_project/
□□□ .venv/
□□□ app.py
□□□ requirements.txt
```

---

## 19. Installing From `requirements.txt`

Another developer can create a new environment:

```
python -m venv .venv
```

Activate it.

Then install the dependencies:

```
pip install -r requirements.txt
```

Now the environment contains the project's required packages.

The basic workflow is:

```
Clone project
    □
Create .venv
    □
Activate .venv
```

 
Install requirements
 
Run project

---

## 20. Should .venv Be Committed to Git?

Usually:

**No.**

You normally do not commit the entire .venv directory to Git.

Why?

Because it can contain:

- many files
- installed packages
- platform-specific files
- unnecessary generated data
- a large amount of project-specific environment data

Instead, commit:

requirements.txt

and your source code.

Then other developers recreate the environment.

---

## 21. Add `.venv` to `.gitignore`

Create or update:

`.gitignore`

Add:

```
.venv/
```

Your project might look like:

```
my_project/
□□□ .venv/
□□□ .gitignore
□□□ requirements.txt
□□□ app.py
□□□ README.md
```

Git ignores `.venv`.

---

## 22. Virtual Environment vs Global Environment

| Global Environment | Virtual Environment |
| --- | --- |
| Shared across projects | Usually isolated per project |
| | |

| | |
|---|---|
| Can cause version conflicts | Reduces dependency conflicts |
| Packages affect other projects | Packages stay project-specific |
| Harder to reproduce | Easier to reproduce |
| Can become cluttered | Easier to manage |

The goal is not to eliminate global Python.

The goal is to avoid using one shared package environment for unrelated projects.

---

## 23. Virtual Environment vs `requirements.txt`

These are not replacements for each other.

### Virtual environment

Provides:

`Isolation`

### `requirements.txt`

Provides:

`Dependency information`

You often use both:

```
Project
□
□□□ .venv
□   □□□ isolated environment
□
□□□ requirements.txt
    □□□ dependency list
```

---

## 24. Virtual Environment vs Python Installation

A virtual environment does not mean installing Python from scratch every time.

It creates an environment using a Python installation available on your system.

For example:

```
System Python
    □
    □□□ Project A □ .venv
    □□□ Project B □ .venv
    □□□ Project C □ .venv
```

The exact implementation details can vary by operating system and Python setup, but conceptually each project gets an isolated environment.

---

## 25. Deactivating a Virtual Environment

When you are finished working:

```
deactivate
```

Your terminal returns to the normal environment.

For example:

```
(.venv) C:\Projects\my_project>
```

becomes:

```
C:\Projects\my_project>
```

---

## 26. Do You Need to Activate Every Time?

Activation is a convenience for your shell.

You can also directly execute the environment's Python.

For example on Windows:

```
.venv\Scripts\python.exe app.py
```

On macOS/Linux:

```
.venv/bin/python app.py
```

However, activation is usually easier during normal development.

---

## 27. Recreating a Deleted Virtual Environment

Suppose you accidentally delete:

```
.venv/
```

Your source code is still safe.

Recreate it:

```
python -m venv .venv
```

Activate it:

```
.venv\Scripts\activate
```

Then reinstall dependencies:

```
pip install -r requirements.txt
```

This demonstrates an important principle:

> Your environment is disposable. Your project dependency definition should be reproducible.

---

## 28. A Complete Project Setup

A common Python project setup looks like this:

```
mkdir my_project
cd my_project

python -m venv .venv
```

Activate it.

Windows:

```
.venv\Scripts\activate
```

macOS/Linux:

```
source .venv/bin/activate
```

Install packages:

```
pip install requests python-dotenv
```

Create the dependency file:

```
pip freeze > requirements.txt
```

Now:

```
my_project/
□□□ .venv/
□□□ .gitignore
□□□ requirements.txt
□□□ app.py
```

---

## 29. Using Virtual Environments With VS Code

VS Code can detect Python environments.

A typical workflow is:

1. Open your project folder.
2. Create `.venv`.
3. Open the Command Palette.
4. Select **Python: Select Interpreter**.
5. Choose the Python interpreter inside `.venv`.
6. Open a new terminal.
7. Install project dependencies.

Your selected interpreter might look like:

```
.venv\Scripts\python.exe
```

Once selected, VS Code uses that environment for Python features such as running and debugging code.

---

## 30. Virtual Environments and Jupyter

Jupyter can also work with virtual environments.

The important idea is that your notebook kernel should use the same environment that contains your project's packages.

Otherwise you may see confusing errors such as:

```
ModuleNotFoundError
```

even though you installed the package somewhere else.

This usually happens because:

`Package installed in Environment A`

`Notebook running Environment B`

The package is therefore unavailable to the notebook.

---

## 31. The Most Common Virtual Environment Mistake

A very common problem is:

```
pip install requests
```

followed by:

```
import requests
```

and Python says:

```
ModuleNotFoundError: No module named 'requests'
```

Why?

You may have installed the package into one Python environment while running your program with another.

Check:

```
python -c "import sys; print(sys.executable)"
```

Then check:

```
python -m pip --version
```

This helps confirm which Python and `pip` are being used.

---

## 32. Why `python -m pip` Can Be Safer

Instead of:

```
pip install requests
```

you can use:

```
python -m pip install requests
```

This explicitly asks the selected Python interpreter to run its associated `pip`.

That makes it easier to avoid situations where:

```
python → Environment A
pip → Environment B
```

Using:

```
python -m pip
```

keeps them connected.

A useful habit is:

```
python -m pip install package_name
```

---

## 33. PowerShell Activation Problems

On Windows PowerShell, you may encounter an execution policy error when running:

```
.venv\Scripts\Activate.ps1
```

The issue is related to PowerShell's script execution policy.

Rather than randomly changing security settings, first understand the error and your organization's/device policy.

If activation is restricted, you can still run the environment's Python directly:

```
.venv\Scripts\python.exe app.py
```

You can also use Command Prompt if appropriate.

---

## 34. Virtual Environment Naming

Common names include:

```
.venv
venv
env
```

`.venv` is a common modern convention.

Choose one convention and use it consistently.

For example:

```
project/
□□□ .venv/
```

---

## 35. One Environment Per Project

A strong default practice is:

```
project-a/
□□□ .venv/

project-b/
□□□ .venv/

project-c/
□□□ .venv/
```

This keeps dependencies isolated.

You do not necessarily need a separate environment for every tiny script, but separate environments are especially useful for real projects.

---

## 36. What Happens When You Clone a Project?

Imagine you clone a Python project:

```
git clone project-url
cd project
```

You may see:

```
app.py
requirements.txt
.gitignore
```

but no `.venv`.

That is normal.

Create your own:

```
python -m venv .venv
```

Activate it.

Then:

```
python -m pip install -r requirements.txt
```

Now your local environment is ready.

---

## 37. A Real World Example: haas.dev

Imagine a Python service for a resource platform:

```
haas_resource_service/
□□□ .venv/
```

```
□□□ app/
□   □□□ models.py
□   □□□ services.py
□   □□□ routes.py
□□□ requirements.txt
□□□ .gitignore
□□□ README.md
```

The application may depend on:

```
fastapi
uvicorn
python-dotenv
```

The environment keeps these dependencies isolated from unrelated Python projects.

A developer setting up the project could run:

```
python -m venv .venv
```

Activate it and then:

```
python -m pip install -r requirements.txt
```

The source code and dependency definition are shared.

The local `.venv` is recreated on each machine.

---

## 38. The Environment Setup Mental Model

When starting a Python project, think:

1. Choose Python
   
2. Create .venv
   
3. Activate it
   
4. Install dependencies
   
5. Save dependency versions
   
6. Build the project

When another developer joins:

Clone project
   
Create .venv
   
Activate
   
Install requirements
   
Run project

---

# 39. Common Mistakes

## Mistake 1: Installing Everything Globally

pip install package

without using a project environment.

### Better

Create and activate `.venv` first.

---

## Mistake 2: Forgetting to Activate

You think you are working inside:

`.venv`

but the terminal is actually using global Python.

### Check

`python -c "import sys; print(sys.executable)"`

---

## Mistake 3: Using the Wrong `pip`

You install with one Python and run with another.

### Better

`python -m pip install package`

---

## Mistake 4: Committing `.venv`

Avoid putting the entire environment into Git.

Use:

`.venv/`

inside `.gitignore`.

---

### Mistake 5: Sharing Only `.venv`

A virtual environment is not a portable dependency definition.

Share:

`requirements.txt`

or another appropriate dependency configuration.

---

### Mistake 6: Forgetting to Update Dependencies

You install a new package but forget to update your project's dependency file.

Another developer then cannot reproduce your environment.

---

## 40. Best Practices

### 1. Use one environment per real project

`project/`

□□□ .venv/

## 2. Use a consistent name

Prefer:

.venv

## 3. Do not commit .venv

Add:

.venv/

to .gitignore.

## 4. Use python -m pip

python -m pip install package

## 5. Keep dependencies reproducible

Maintain an appropriate dependency file such as:

requirements.txt

## 6. Verify your interpreter

python -c "import sys; print(sys.executable)"

## 7. Recreate environments instead of manually repairing broken ones

Often the fastest solution is:

Delete .venv

 
Create .venv again
 
Install dependencies

---

## 41. venv vs virtualenv

Python provides:

`venv`

as part of the standard library.

There is also a third-party tool called:

`virtualenv`

For most beginners and many standard projects, `venv` is enough.

You can create an environment with:

`python -m venv .venv`

You do not need an additional tool just to start using virtual environments.

---

## 42. Virtual Environments and Dependency Management

Virtual environments solve **isolation**.

Dependency management can involve additional tools and files.

For example:

Virtual environment
 Where packages are installed

requirements.txt
 Which packages the project needs

pyproject.toml
 Modern project configuration and dependency definition

As Python projects become more advanced, you may encounter tools such as Poetry, Pipenv, or uv.

The important foundation is understanding what problem each tool is solving.

---

## 43. When Should You Create a Virtual Environment?

Create one when:

- starting a real Python project
- using third-party packages
- working on multiple Python projects
- collaborating with other developers
- building APIs
- building web applications
- working with data science libraries
- creating automation tools
- developing CLI applications

For a simple one-file script using only the standard library, a virtual environment may not always be necessary.

---

## 44. 5 Minute Challenge

Create a project called:

`weather_app`

### Step 1

Create the folder:

```
mkdir weather_app
cd weather_app
```

### Step 2

Create the environment:

```
python -m venv .venv
```

### Step 3

Activate it.

### Step 4

Install:

```
python -m pip install requests
```

### Step 5

Check:

```
python -m pip list
```

### Step 6

Create:

```
requirements.txt
```

using:

```
python -m pip freeze > requirements.txt
```

### Step 7

Check the project:

```
weather_app/
□□□ .venv/
□□□ requirements.txt
```

Then add:

```
.venv/
```

to `.gitignore`.

---

## 45. Mini Project: Reproducible Python Project Setup

Create:

```
task_manager/
□□□ .venv/
□□□ .gitignore
□□□ requirements.txt
□□□ app.py
```

Your goal is to create an environment that another developer can reproduce.

## Requirements

Install at least:

```
requests
python-dotenv
```

Create:

```
python -m pip freeze > requirements.txt
```

Then delete the environment:

```
.venv
```

Recreate it:

```
python -m venv .venv
```

Activate it.

Finally:

```
python -m pip install -r requirements.txt
```

If the application can run again, you have successfully created a reproducible environment setup.

---

# 46. Exercises

## Exercise 1

Create a virtual environment called:

`.venv`

for a project called:

`expense_tracker`

---

## Exercise 2

Install:

`requests`

inside the environment.

Verify it using:

`python -m pip show requests`

---

## Exercise 3

Find the Python executable:

`python -c "import sys; print(sys.executable)"`

Explain what the output tells you.

---

## Exercise 4

Create:

`requirements.txt`

from your environment.

---

## Exercise 5

Deactivate the environment and explain what changed.

---

## Exercise 6

Delete the environment and recreate it.

Install dependencies again using:

`python -m pip install -r requirements.txt`

---

# 47. Mini Quiz

## Question 1

What is the main purpose of a virtual environment?

A. To make Python faster
B. To isolate project dependencies
C. To replace Python
D. To create databases

**Answer: B**

---

## Question 2

Which command creates a virtual environment?

A.

```
pip create .venv
```

B.

```
python -m venv .venv
```

C.

```
python install venv
```

D.

`venv create python`

**Answer: B**

---

## Question 3

Should `.venv` normally be committed to Git?

A. Yes
B. No

**Answer: B**

---

## Question 4

What file can describe project dependencies?

A. `requirements.txt`
B. `python.txt`
C. `packages.py`
D. `environment.py`

**Answer: A**

---

## Question 5

Why can `python -m pip install requests` be useful?

**Answer:** It runs `pip` through the selected Python interpreter, reducing confusion between different Python installations and environments.

---

## 48. Interview Questions

### Beginner

1. What is a virtual environment?
2. Why do Python projects use virtual environments?
3. What is `venv`?
4. How do you create a virtual environment?
5. How do you activate it on Windows?
6. How do you activate it on macOS/Linux?
7. How do you deactivate it?
8. What is `pip`?
9. What is `requirements.txt`?
10. Should `.venv` be committed to Git?

### Intermediate

11. Why can two projects require different virtual environments?
12. What happens when you install a package inside an active environment?
13. How can you check which Python executable is being used?
14. Why might `ModuleNotFoundError` occur after installing a package?
15. Why is `python -m pip` often safer than simply using `pip`?
16. How do you recreate a deleted virtual environment?
17. What is the difference between a virtual environment and `requirements.txt`?
18. What is the difference between `venv` and `virtualenv`?
19. How do IDEs such as VS Code use virtual environments?
20. Why should environments generally be treated as disposable?

---

# 49. Quick Cheat Sheet

## Create

```
python -m venv .venv
```

## Activate on Windows CMD

```
.venv\Scripts\activate
```

## Activate on Windows PowerShell

```
.venv\Scripts\Activate.ps1
```

## Activate on macOS/Linux

```
source .venv/bin/activate
```

## Install package

```
python -m pip install requests
```

## Install exact version

```
python -m pip install requests==2.32.3
```

## Upgrade

```
python -m pip install --upgrade requests
```

## Uninstall

```
python -m pip uninstall requests
```

## List packages

```
python -m pip list
```

### Show package

```
python -m pip show requests
```

### Create requirements file

```
python -m pip freeze > requirements.txt
```

### Install requirements

```
python -m pip install -r requirements.txt
```

### Check Python executable

```
python -c "import sys; print(sys.executable)"
```

### Deactivate

```
deactivate
```

### Ignore environment in Git

```
.venv/
```

---

## 50. Key Takeaways

- A virtual environment isolates Python project dependencies.
- Python provides the built-in `venv` module.
- Create one with `python -m venv .venv`.
- Activate it before normal development.
- Install packages inside the active environment.
- `python -m pip` helps ensure you are using the correct `pip`.

- `requirements.txt` records project dependencies.
- `.venv` should normally not be committed to Git.
- Environments can be deleted and recreated.
- A reproducible project should allow another developer to create a fresh environment and install the required dependencies.
- Virtual environments become especially important as projects and dependency trees become more complex.

---

## The Developer Habit

Before installing a package for a Python project, think:

```
Am I inside the correct project?
        ↓
Is the correct .venv active?
        ↓
Is this the Python interpreter I expect?
        ↓
Install the dependency
        ↓
Record the dependency
```

That small habit prevents many dependency and environment problems later.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app