Python Virtual Environments: A Quick Guide

Rahul Pandare
July 28, 2025

Purpose: This article provides an introduction to Python virtual environments and explains how to install and use them locally for project-specific development.

What Is a Virtual Environment?

A **virtual environment** is an isolated Python environment that allows you to manage dependencies for a specific project without affecting other projects or the system Python.The Python executable is copied to a new location, allowing it to function independently from its original installation and environment. This is especially useful when working on multiple projects with different dependency requirements.

Benefits of using virtual environments:

- Avoid conflicts between package versions.
- Keep your global Python installation clean.
- Ensure reproducibility for collaborators or deployments.

Tools for Managing Python Environments

There are a few tools commonly used to manage virtual environments:

- **`venv`**: Built-in Python module to create lightweight virtual environments.
- **`pyenv`**: Manages multiple versions of Python on your machine.
- **`pyenv-virtualenv`**: A `pyenv` plugin for managing virtual environments for specific Python versions.

Installing `pyenv` (macOS/Homebrew)

Homebrew is a package manager for macOS and is the easiest way to install `pyenv`.

```
brew update
brew install pyenv
brew install pyenv-virtualenv
```

Add the following to your shell configuration file (e.g., ~/.zshrc, ~/.bash_profile, or ~/.bashrc):

```
# Set up pyenv
export PYENV_ROOT="$HOME/.pyenv"
export PATH="$PYENV_ROOT/bin:$PATH"

# Load pyenv and pyenv-virtualenv
eval "$(pyenv init --path)"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

We add these lines to .zshrc so that every time a new terminal opens, `pyenv` and `pyenv-virtualenv` are automatically initialized, making Python version and virtual environment management seamless. Then apply changes `source ~/.zshrc` (or source the appropriate file)

Creating Python Virtual Environments

There are two primary ways to create virtual environments:

1. Using a Local Python Installation with `venv`

Step 1: Create and Use the Virtual Environment

```
/opt/homebrew/bin/python3 -m venv myenv   # Create the environment
source myenv/bin/activate                 # Activate the
environment
```

To deactivate:

```
deactivate
```

Installed libraries location:`myenv/lib/python3.x/site-packages/`

Tip: Use name `.myenv` to make the virtual environment folder hidden.

2. Using `pyenv + pyenv-virtualenv`

Step 1: Install a Python Version

```
pyenv install 3.10.11 # or any other required version
```

Step 2: Create a Virtual Environment

```
pyenv virtualenv 3.10.11 myenv
```

Step 3: List Available Environments

```
pyenv versions
```

Example output:

```
  system
  3.10.11
  3.10.11/envs/myenv
* myenv (set by ~/.pyenv/version)
```

Step 4: Activate and Deactivate

```
pyenv activate myenv     # Activate environment
pyenv deactivate         # Deactivate environment
```

Check: Once the virtual environment is activated, only the libraries installed within that environment are available for use, ensuring complete isolation from the global Python environment. To verify that you're using the correct Python interpreter, run the command `which python`. This should return the path to the Python executable associated with your virtual environment, for example: `/Users/rahul/.pyenv/shims/python`

Tip: To auto-activate a specific environment when entering a project directory, create a .python-version file:

```
echo "myenv" > .python-version
```

To stop the auto-activation of a specific environment when entering a project directory remove the python-version file:

```
rm -r .python-version
```

Using Jupyter Notebook in a Virtual Environment

Jupyter Notebook is a powerful and user-friendly tool for working with Python, especially in contexts where visualization, interactive coding, and exploratory data analysis are essential. It offers a browser-based interface that allows you to combine code, output, and rich text documentation in one place, making it ideal for tasks like machine learning, data science, and research workflows. While other editors such as VS Code or PyCharm provide interactive Python capabilities, Jupyter is often more intuitive for iterative experimentation in my opinion. After setting up your virtual environment, you need to make it accessible within Jupyter by installing the appropriate kernel and registering it. Use the following commands:

```
pyenv activate myenv        # Activate the virtual environment
pip install ipykernel       # Install Jupyter kernel support in this
environment

# Register the environment as a Jupyter kernel
python -m ipykernel install --user
  --name=myenv
  --display-name "Python (pyenv-myenv)"
```

Now you can launch jupyter notebook by:

```
jupyter notebook
```

This will open a Jupyter Notebook in your browser on a local host port, using the current Python environment.

Best Practice: Saving and Reusing Dependencies

Once you're done with development, freeze the package versions to a file:

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

To reinstall the same dependencies in another environment:

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