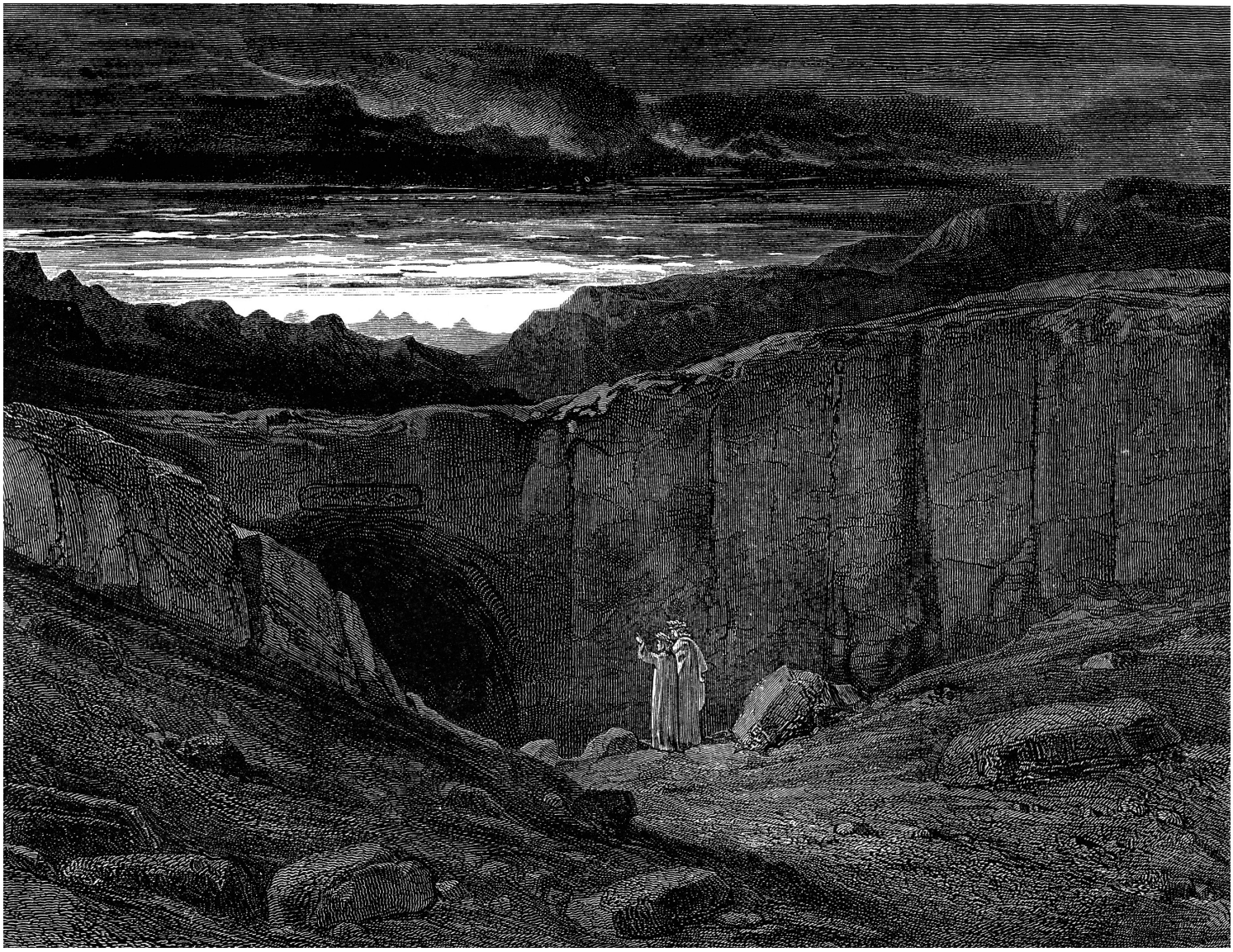
A Guide to Python's Virtual Environments

What they are, how to use them, and how they really work.

Matthew Sarmiento Follow
May 23, 2019 · 8 min read ★



The Gate of Dependency Hell: "Abandon all hope, ye who enter here." Illustration by Gustave Doré.

Python's virtual environments make life easier. *A lot* easier.

In this guide we'll cover the basics of virtual environments and how to use them. Then we'll peek under the hood and take a closer look at how virtual environments actually work.

⚠️ **Note**: We'll be using the latest version of Python 3.7.x on macOS Mojave throughout this guide.

Table of Contents

. . .

Why Use Virtual Environments?

Virtual environments provide a simple solution to a host of potential problems. In particular, they help you to:

- **Resolve dependency issues** by allowing you to use different versions of a package for different projects. For example, you could use Package A v2.7 for Project X and Package A v1.3 for Project Y.
- Make your project **self-contained** and **reproducible** by capturing all package dependencies in a requirements file.
- Install packages on a host on which you do not have admin privileges.

- Keep your global `site-packages/` directory tidy by removing the need to install packages system-wide which you might only need for one project.

Sounds pretty handy, no? As you begin to build more sophisticated projects and collaborate with others you'll come to see just how *essential* virtual environments are. And if you're a data scientist like me, you'll also want to familiarize yourself with their multilingual cousins, Conda environments.

But first thing's first.

. . .

What the Virtualenv?!

What exactly *is* a virtual environment?

A virtual environment is a Python tool for **dependency management** and **project isolation**. They allow Python **site packages** (third party libraries) to be installed locally in an isolated directory for a particular project, as opposed to being installed globally (i.e. as part of a system-wide Python).

Great. That all sounds nice, but what is a virtual environment *really*? Well, a virtual environment is just **a directory** with three important components:

- A `site-packages/` folder where third party libraries are installed.
- Symlinks to Python executables installed on your system.
- Scripts that ensure executed Python code uses the Python interpreter and site packages installed inside the given virtual environment.

The last bit is where all the s*** goes down. We'll take a closer look later on, but for now let's just see how we actually use virtual environments.

. . .

Virgil appeases Cerberus — Canto VI. Illustration by Gustave Doré.

Using Virtual Environments

Creating Environments

Say we wanted to create a virtual environment for a project we're working on called `test-project/`, which has the following directory tree.

```
test-project/
├── data
├── deliver           # Final analysis, code, & presentations
├── develop           # Notebooks for exploratory analysis
├── src               # Scripts & local project modules
└── tests
```

All we need to do is execute the `venv` module, which is part of the Python standard library.

```
% cd test-project/
% python3 -m venv venv/       # Creates an environment called venv/
```

⚠️ **Note:** You can replace "venv/" with a different name for your environment.

Voilà! A virtual environment has been born. Now our project looks like this:

```
test-project/
├── data
├── deliver
├── develop
├── src
├── tests
└── venv                  # There it is!
```

♻️ **Reminder:** A virtual environment is itself a directory.

The only thing left to do is to "activate" our environment by running the scripts we mentioned earlier.

```
% source venv/bin/activate
(venv) %                              # Fancy new command prompt
```

We're now inside an active virtual environment (indicated by the command prompt prefixed with the name of the active environment).

At this point we would work on our project as usual, safe in the knowledge that our project is completely isolated from the rest of our system. Inside our environment we cannot access system-wide site packages and any packages we install will not be accessible outside of our environment.

When we're done working on our project, we can exit the environment with

```
(venv) % deactivate
%                                   # Old familiar command prompt
```

. . .

Installing Packages

By default, only `pip` and `setuptools` are installed inside a new environment.

```
(venv) % pip list                    # Inside an active environment

Package    Version
---------- -------
pip        19.1.1
setuptools 40.8.0
```

If we want to install a specific version of a third party library, say v1.15.3 of `numpy`, we can just use `pip` as usual.

```
(venv) % pip install numpy==1.15.3
(venv) % pip list

Package    Version
---------- -------
numpy      1.15.3
pip        19.1.1
setuptools 40.8.0
```

Now we can import `numpy` in a script or active Python shell. For instance, say our project contains a script `tests/imports-test.py` with the following lines.

```
#!/usr/bin/env python3

import numpy as np
```

When we run this script directly from the command-line, we get:

```
(venv) % tests/imports-test.py
(venv) %                                # Look, Ma, no errors!
```

Success. Our script imported `numpy` without a hitch ☐.

. . .

Dante and Virgil cross the river Styx — Canto VIII. Illustration by Gustave Doré.

Managing Environments

Requirements Files

The easiest way to make our work reproducible by others is to include a requirements file in our project's **root directory** (top directory). To do so, we'll run `pip freeze`, which lists installed third party packages along with their version numbers,

```
(venv) % pip freeze
numpy==1.15.3
```

And write the output to a file, which we'll call `requirements.txt` .

```
(venv) % pip freeze > requirements.txt
```

We can use this same command to rewrite our requirements file whenever we update a package or install a new one.

Now anyone we share our project with will be able to run our project on their system by duplicating our environment using our `requirements.txt` file.

. . .

Duplicating Environments

Wait — how exactly do we do that?

Imagine our teammate Sara has pulled down our `test-project/` from our team's GitHub repository. On her system the project's directory tree looks like:

```
test-project/
├── data
├── deliver
├── develop
├── requirements.txt
├── src
└── tests
```

Notice anything slightly — *unusual*? Yep, that's right. There's no `venv/` folder. We've excluded it from our team's GitHub repository because including it can cause headaches.

This is one reason having a `requirements.txt` file is *essential* to reproducing your project's code.

To run our `test-project/` on her machine, all Sara needs to do is to create a virtual environment inside the project's root directory

```
Sara% cd test-project/
Sara% python3 -m venv venv/
```

And install the project's dependencies inside an active virtual environment with the incantation `pip install -r requirements.txt`.

```
Sara% source venv/bin/activate
(venv) Sara% pip install -r requirements.txt

Collecting numpy==1.15.3 (from -r i (line 1))
Installing collected packages: numpy
Successfully installed numpy-1.15.3                    # Woohoo! 🎊
```

Now the project's environment on Sara's system is *exactly* the same as on our system. Pretty neat, huh?

. . .

Troubleshooting

Sadly, things don't always go according to plan. Eventually you *will* run into problems. Maybe you've updated a particular site package by mistake and now find yourself in the ninth level of Dependency Hell, unable to run a single line of your project's code. Then again, maybe it's not that bad. Maybe you only find yourself in the seventh level.

Whatever level you find yourself in, the easiest way to escape the flames and see the sun shine again is to **re-create** your project's virtual environment.

```
% rm -r venv/                          # Nukes the old environment
% python3 -m venv venv/                # Makes a blank new one
% pip install -r requirements.txt      # Re-installs dependencies
```

That's it. Thanks to your `requirements.txt` file you're back in business. Yet another reason to *always* include a requirements file in your projects.

. . .

Dante speaks with the traitors in the ice — Canto XXXII. Illustration by Gustave Doré.

How Virtual Environments Do Their Thing

So you want to know more about virtual environments, eh? Like how an active environment *knows* how to use the right Python interpreter and how to find the right third party libraries.

echo $PATH

It all comes down to the value of PATH, which tells your shell what instance of Python to use and where to look for site packages. In your base shell PATH will look more or less like this.

```
% echo $PATH
/usr/local/bin:/usr/bin:/usr/sbin:/bin:/sbin
```

When you invoke a Python interpreter or run a `.py` script, your shell searches the directories listed in PATH **in order** until it encounters a Python instance. To see which Python instance PATH finds first, run `which python3`.

```
% which python3
/usr/local/bin/python3                # Your output may differ
```

It's also easy to see where this Python instance looks for site packages with the `site` module, which is part of the Python standard library.

```
% python3                             # Activates a Python shell
>>> import site
>>> site.getsitepackages()            # Points to site-packages folder

['/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Version
s/3.7/lib/python3.7/site-packages']
```

Running the script `venv/bin/activate` modifies PATH so that our shell searches through our project's local binaries *before* searching through our system's global binaries.

```
% cd ~/test-project/
% source venv/bin/activate
(venv) % echo $PATH

~/test-project/venv/bin:/usr/local/bin:/usr/bin:/usr/sbin:/bin:/sbin
```

Now our shell knows to use our project's local Python instance

```
(venv) % which python3
~/test-project/venv/bin/python3
```

And where to find our project's local site packages.

```
(venv) % python3
>>> import site
>>> site.getsitepackages()

['~/test-project/venv/lib/python3.7/site-packages']    # Ka-ching 🤑
```

. . .

A Sanity Check

Remember our `tests/imports-test.py` script from before? It looked like this.

```
#!/usr/bin/env python3

import numpy as np
```

We were able to run this script from inside our active environment with no problems because our environment's Python instance was able to access our project's local site packages.

What happens if we run the same script from *outside* our project's virtual environment?

```
% tests/imports-test.py                # Look, no active environment

Traceback (most recent call last):
  File "tests/imports-test.py", line 3, in <module>
    import numpy as np

ModuleNotFoundError: No module named 'numpy'
```

Yep, we get an error — **as we should**. If we didn't, it would mean we were able to access our project's local site packages from outside our project, defeating the entire purpose of having a virtual environment. The fact that we get an error is proof our project is **completely isolated** from the rest of our system.

. . .

The Directory Tree of an Environment

One thing that helps me keep all of this information organised in my head is having a clear picture of what an environment's directory tree looks like.

```
test-project/venv/                  # Our environment's root directory
├── bin
│   ├── activate                          # Scripts to activate
│   ├── activate.csh                      # our project's
│   ├── activate.fish                     # virtual environment.
│   ├── easy_install
│   ├── easy_install-3.7
│   ├── pip
│   ├── pip3
│   ├── pip3.7
│   ├── python -> /usr/local/bin/python   # Symlinks to system-wide
│   └── python3 -> python3.7              # Python instances.
├── include
├── lib
│   └── python3.7
│       └── site-packages              # Stores local site packages
└── pyvenv.cfg
```

. . .

Dante and Virgil return to the mortal realm — Canto XXXIV. Illustration by Gustave Doré.

Further Reading

If your curiosity hasn't been satisfied and you still want to know more about virtual environments, I highly recommend Real Python's terrific primer on virtual environments. And if you find yourself in thrall to the remarkable illustrations of Gustave Doré, I highly recommend reading Dante's *Inferno*.

Other than that, that about does it for us. If you'd like to stay up to date with my data science-y postings, feel free to follow me on twitter.

Cheers, and happy reading.

Thanks to Ramiro Sarmiento.