



Daniel van Flymen

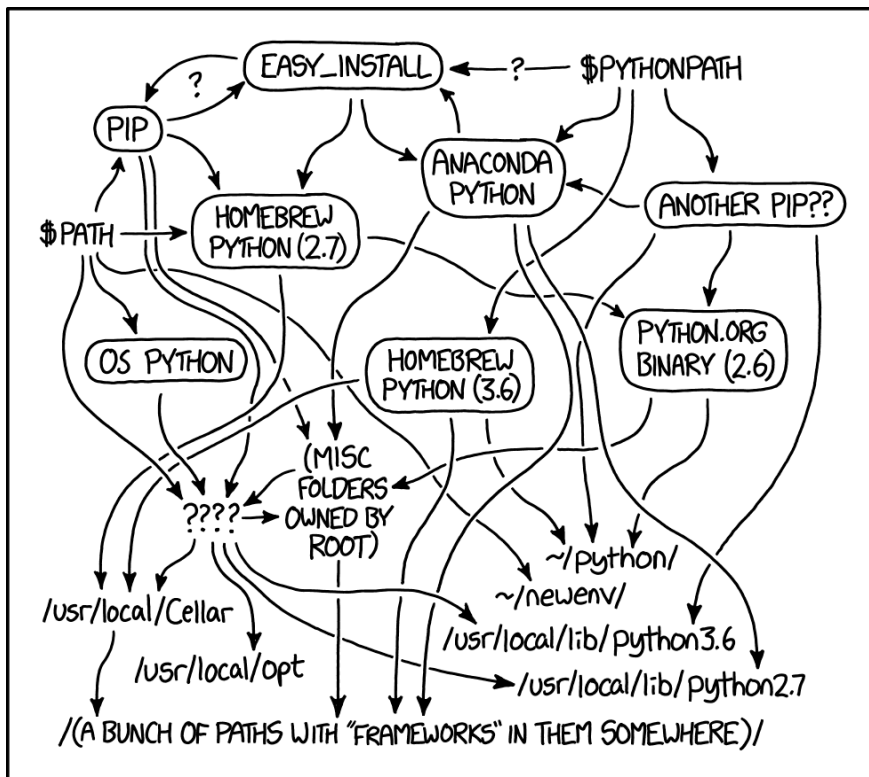
Follow

twitter.com/van_flymen • blockchain enthusiast & photographer • http://dvf.nyc • south african in nyc 🙌

Oct 23 · 5 min read

Use pyenv + Pipenv for local Python development

The missing guide for setting up a great local development workflow for your Python projects.



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

"Python Environment" by xkcd

This is an **opinionated** way of developing with Python locally. You've probably discovered that it's a pain in the ass to manage different projects with dependencies targeting different Python versions on your local machine.

To complicate things, there are multiple ways of installing Python too:

- Preinstallation by the OS 🙄
- Using a package manager like `brew` or `apt` 🙄

- Using the binaries from www.python.org 🙄
- Using [pyenv](#)—easy way to install and manage Python installations 😊

This guide uses [pyenv](#) to manage Python installations, and [Pipenv](#) to manage project dependencies (instead of raw pip).

. . .

Installing pyenv

Let's install via `brew` :

```
$ brew install pyenv
```

If you're not on Mac, please see [pyenv's installation instructions](#).

Add the following to your `~/.bash_profile` , or `~/.bashrc` (depending on your shell) to automatically initialize pyenv when your terminal loads:

```
eval "$(pyenv init -)"
```

How does pyenv work?

See all available Python versions:

```
$ pyenv install --list
```

Let's install Python 3.6.6

```
$ pyenv install 3.6.6
```

```
Installed Python-3.6.6 to
/Users/dvf/.pyenv/versions/3.6.6
```

pyenv won't change your global interpreter unless you tell it to:

```
$ python --version
```

```
Python 2.7.14
```

```
$ pyenv global 3.6.6
```

```
Python 3.6.6
```

pyenv allows you to install different versions of Python **local** to a directory. Let's create a project targeting Python 3.7.0:

```
$ pyenv install 3.7.0
```

```
Installed Python-3.7.0 to
/Users/dvf/.pyenv/versions/3.7.0
```

```
$ mkdir my_project && cd my_project
$ python --version
```

```
Python 3.6.6
```

```
$ pyenv local 3.7.0
$ python --version
```

```
Python 3.7.0
```

Now whenever you find yourself in `my_project` you'll automatically use the Python 3.7.0 interpreter.

👉 **Did that make sense?** If not, stop here and take some time to play around with `pyenv`—it works by installing all Python interpreters in `~/.pyenv` and dynamically adjusting your `$PATH` depending on your current directory.

What is Pipenv and how does it work?

Pipenv is the officially recommended way of managing project dependencies. Instead of having a `requirements.txt` file in your project, and managing virtualenvs, you'll now have a `Pipfile` in your project that does all this stuff automatically.

Start off by installing it via `pip`, it's a rapidly evolving project so make sure you have the latest version (2018.10.13 at the time of writing):

```
$ pip install -U pipenv
```

Using Pipenv for the first time

Let's set up Pipenv in your project:

```
$ cd my_project  
$ pipenv install
```

```
Creating a virtualenv for this project...
```

```
Pipfile: /Users/dvf/my_project/Pipfile
```

```
Using /Users/dvf/.pyenv/versions/3.7.0/bin/python3.7  
(3.7.0) to create virtualenv...
```

You'll find two new files in your project: `Pipfile` and `Pipfile.lock`.

If you're installing in a pre-existing project, Pipenv will convert your old `requirements.txt` into a `Pipfile`. How cool is that?

This is what your `Pipfile` should look like for a fresh project:

```
[[source]]  
url = "https://pypi.org/simple"  
verify_ssl = true  
name = "pypi"
```

```
[packages]
```

```
[dev-packages]

[requires]
python_version = "3.7"
```

Notice that we didn't activate any virtual environments here, Pipenv takes care of virtual environments for us. So, installing new dependencies is simple:

```
$ pipenv install django

Installing django
...

Installing collected packages: pytz, django
Successfully installed django-2.1.2 pytz-2018.5

Adding django to Pipfile's [packages]...
Pipfile.lock (4f9dd2) out of date, updating to
(a65489)...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (4f9dd2)!

Installing dependencies from Pipfile.lock (4f9dd2)...
🐍 ████████████████████████████████████████████████████████████████████████████ 2/2 - 00:00:01

To activate this project's virtualenv, run pipenv
shell.
Alternatively, run a command inside the virtualenv
with pipenv run.
```

If you inspect your Pipfile you'll notice it now contains django = "*" as a dependency.

If we wanted to install dev dependencies for use during development, for example YAPF, you'd add --dev to the install step:

```
$ pipenv install --dev yapf
```

What is Pipfile.lock?

`Pipfile.lock` is super important because it does two things:

1. Provides good security by keeping a hash of each package installed.
2. Pins the versions of all dependencies and sub-dependencies, giving you replicable environments.

Let's see what it currently looks like:

```
{
  "_meta": {
    "hash": {
      "sha256": "627ef89...64f9dd2"
    },
    "pipfile-spec": 6,
    "requires": {
      "python_version": "3.7"
    },
    "sources": [
      {
        "name": "pypi",
        "url": "https://pypi.org/simple",
        "verify_ssl": true
      }
    ]
  },
  "default": {
    "django": {
      "hashes": [
        "sha256:acdcc1...ab5bb3",
        "sha256:efbcad...d16b45"
      ],
      "index": "pypi",
      "version": "==2.1.2"
    },
    "pytz": {
      "hashes": [
        "sha256:a061aa...669053",
        "sha256:ffb9ef...2bf277"
      ],
      "version": "==2018.5"
    }
  },
  "develop": {}
}
```

Notice that the versions of each dependency are pinned. Without a *very* good reason, you would always want this file committed to your source control.

Custom Indexes

Until Pipenv it was difficult to use *private* Python repositories, for example if you'd like to host private Python libraries within your organization. Now all you need to do is define them as an additional sources in the `Pipfile` :

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"


[[source]]
url = "https://www.example.com"
verify_ssl = true
name = "some-repo-name"

[packages]
django = "*"
my-private-app = {version="*", index="some-repo-name"}

[dev-packages]

[requires]
python_version = "3.7"
```

Notice that we told `my-private-app` to use the private repo. If omitted, Pipenv will cycle through indexes until it finds the package.

 **Pipenv will also consume any environment variables in values**, which is useful if you have sensitive credentials you don't want sitting in source control (this was my contribution [here](#))

Deploying

When deploying it's important that your deploy fails if there's a mismatch between installed dependencies and the `Pipfile.lock` . So you should append `--deploy` to your install step which does just that:

```
$ pipenv install --deploy
```

You could also check which dependencies are mismatched:

```
$ pipenv check
```

And see which sub-dependencies are installed by packages:

```
$ pipenv graph --reverse  
  
pip==18.1  
pytz==2018.5  
  - Django==2.1.2 [requires: pytz]  
setuptools==40.4.3  
wheel==0.32.2  
yapf==0.24.0
```

Once-off commands, scripts and activating venvs

If you're actively developing a project, it's helpful to activate the virtual environment:

```
$ pipenv shell  
  
Launching subshell in virtual environment...  
(my_project) → my_project
```

Or, if you'd like to execute a command inside the venv:

```
$ pipenv run python manage.py runserver
```

You can also add scripts to `Pipfile` similar to `npm` `package.json` :

```
[[source]]  
url = "https://pypi.org/simple"  
verify_ssl = true  
name = "pypi"  
  
[packages]  
django = "*"  
  
[dev-packages]  
yapf = "*"
```



```
[scripts]
server = "python manage.py runserver"
```

```
[requires]
python_version = "3.7"
```

Now you can execute the script:

```
$ pipenv run server
```

We've just touched the tip of the iceberg. If you've like to learn more about Pipenv, I encourage you to read the [great documentation](#).

. . .

I hope this was helpful to you. And I'd love to hear any thoughts or suggestions you have in the comments!



