A Better Pip Workflow™

2016

2 min read • 550 words

Update: I developed Pipenv to solve these problems. Check it out.

When developing Python applications today, it's standard practice to have a `requirements.txt` file in the root of your repository.

This file can be used in different ways, and typically takes one of these two forms:

1. A list of top-level dependencies a project has, often without versions specified.
2. A complete list of all dependencies a project has, each with exact versions specified.

Method #1: Simple Requirements

> A list of top-level dependencies a a project has, often without versions specified.

```
$ cat requirements.txtrequests[security]flaskgunicorn==19.4.5
```

Method #1 is very simple, and is effectively the user experience that everyone using requirements files wants. However, when a `requirements.txt` file like this is used to deploy to production, unexpected consequences can occur. Effectively, because versions haven't been pinned, running `$ pip install` will give you different results today than it will tomorrow.

This is bad. As different versions of sub-dependencies are released, the result of a fresh `$ pip install -r requirements.txt` will result in different packages being installed, and potentially, your application failing for unknown and hidden reasons.

Method #2: Exact Requirements

> A complete list of all dependencies a project has, each with exact package versions specified.

```
$ cat requirements.txtcffi==1.5.2cryptography==1.2.2enum34==1.1.2Flask==0.10.1gunicorn=
```

Method #2 is best-practice for deploying applications, and ensures an explicit runtime environment with deterministic builds.

All dependencies, including sub-dependencies, are listed, each with an exact version specified.

This type of `requirements.txt` is generated from the output of running `$ pip freeze` from within a current working runtime environment for the application. This encourages dev/prod parity, and encourages you to treat code within external packages with the same level of respect as your application code (because it is your application code).

The Frustrations

While the **Method #2** format for `requirements.txt` is best practice, it is a bit cumbersome. Namely, if I'm working on the codebase, and I want to `$ pip install --upgrade` some/all of the packages, I am unable to do so easily.

My previous method for doing so was to simply pick out the top-level dependencies with my eyes and manually type out `$ pip install requests[security] flask --upgrade`. This is not a good experience.

I thought long and hard about building a tool to solve this problem. Others, like pip-tools, already have. But, I don't want another tool in my toolchain; this should be possible with the tools available.

Eventually, I figured out a nice way to have the best of both worlds in my Python projects, with the tools I already use. I've been using this workflow in my projects for a while now, and I'm very happy with the results.

The Workflow

It's very simple: instead of having one requirements file, you have two:

- `requirements-to-freeze.txt`
- `requirements.txt`

https://gist.github.com/kennethreitz/1ae33e0f6744a5ae1976\.js
`requirements-to-freeze.txt` uses **Method #1**, and is used to specify your top-level dependencies, and any explicit versions you need to specify.

`requirements.txt` uses **Method #2**, and contains the output of `$ pip freeze` after `$ pip install requirements-to-freeze.txt` has been run.

Basic Usage

```
$ cd project-repo$ pip install -r requirements-to-freeze.txt --upgradeInstalling collec
```

The best of both worlds.

I encourage you to give this workflow a try. There's a good chance that it'll save you from some failing builds and scratching heads in the future :)
