Main menu

Why Python devs should use Pipenv

Only a year old, Pipenv has become the official Python-recommended resource for managing package dependencies.

28 Feb 2018 | Lacey Williams Henschel (/users/laceynwilliams) | 30 | 3 comments

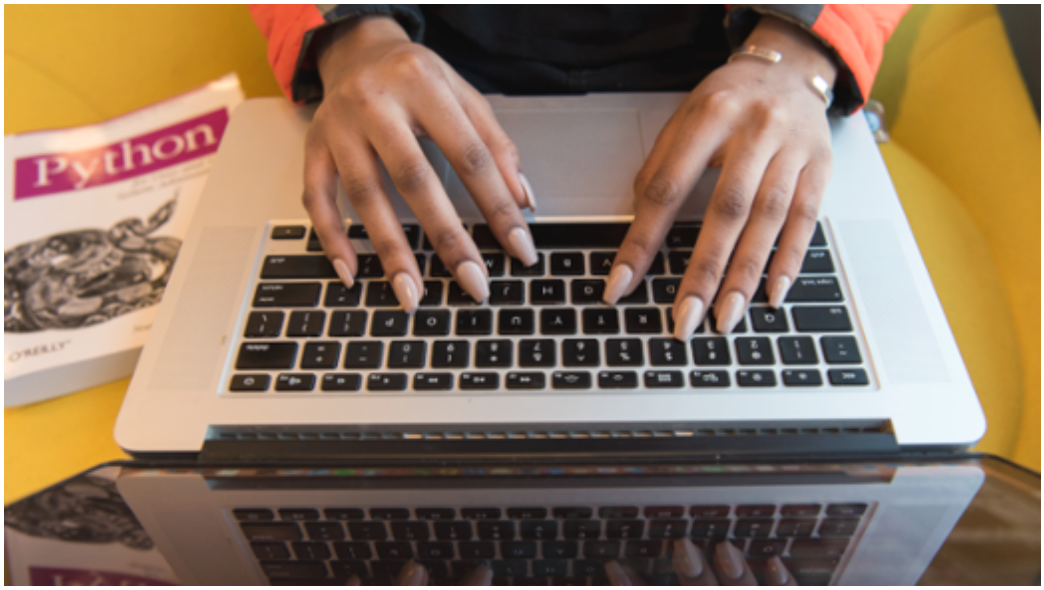

Image by :
WOCinTech Chat (https://www.flickr.com/photos/wocintechchat/25926664911/). Modified by Opensource.com. CC BY-SA 4.0 (https://creativecommons.org/licenses/by/4.0/)

This article was co-written with Jeff Triplett (https://opensource.com/users/jefftriplett).

Pipenv, the "Python Development Workflow for Humans" created by Kenneth Reitz a little more than a year ago, has become the official Python-recommended resource (https://packaging.python.org/tutorials/managing-dependencies/#managing-dependencies) for managing package dependencies. But there is still confusion about what problems it solves and how it's more useful than the standard workflow using `pip` and a `requirements.txt` file. In this month's Python column, we'll fill in the gaps.

A brief history of Python package installation

To understand the problems that Pipenv solves, it's useful to show how Python package management has evolved.

Take yourself back to the first Python iteration. We had Python, but there was no clean way to install packages.

Then came Easy Install (http://peak.telecommunity.com/DevCenter/EasyInstall), a package that installs other Python packages with relative ease. But it came with a catch: it wasn't easy to uninstall packages that were no longer needed.

Enter pip (https://packaging.python.org/tutorials/installing-packages/#use-pip-for-installing), which most Python users are familiar with. `pip` lets us install and uninstall packages. We could specify versions, run `pip freeze > requirements.txt` to output a list of installed packages to a text file, and use that same text file to install everything an app needed with `pip install -r requirements.txt`.

But `pip` didn't include a way to isolate packages from each other. We might work on apps that use different versions of the same libraries, so we needed a way to enable that. Along came virtual environments (https://packaging.python.org/tutorials/installing-packages/#creating-virtual-environments), which enabled us to create small, isolated environments for each app we worked on. We've seen many tools for managing virtual environments: virtualenv (https://virtualenv.pypa.io/en/stable/), venv (https://docs.python.org/3/library/venv.html), virtualenvwrapper (https://virtualenvwrapper.readthedocs.io/en/latest/), pyenv (https://github.com/pyenv/pyenv), pyenv-virtualenv (https://github.com/pyenv/pyenv-virtualenv), pyenv-virtualenvwrapper (https://github.com/pyenv/pyenv-virtualenvwrapper), and even more. They all play well with `pip` and `requirements.txt` files.

The new kid: Pipenv

Pipenv aims to solve several problems.

More Python Resources

- What is Python? (https://opensource.com/resources/python?intcmp=7016000000127cYAAQ)
- Top Python IDEs (https://opensource.com/resources/python/ides?intcmp=7016000000127cYAAQ)
- Top Python GUI frameworks (https://opensource.com/resources/python/gui-frameworks?intcmp=7016000000127cYAAQ)
- Latest Python content (https://opensource.com/tags/python?intcmp=7016000000127cYAAQ)
- More developer resources (https://developers.redhat.com/?intcmp=7016000000127cYAAQ)

First, the problem of needing the `pip` library for package installation, plus a library for creating a virtual environment, plus a library for managing virtual environments, plus all the commands associated with those libraries. That's a lot to manage. Pipenv ships with package management and virtual environment support, so you can use one tool to install, uninstall, track, and document your dependencies and to create, use, and organize your virtual environments. When you start a project with it, Pipenv will automatically create a virtual environment for that project if you aren't already using one.

Pipenv accomplishes this dependency management by abandoning the `requirements.txt` norm and trading it for a new document called a Pipfile (https://github.com/pypa/pipfile). When you install a library with Pipenv, a `Pipfile` for your project is automatically updated with the details of that installation, including version information and possibly the Git repository location, file path, and other information.

Second, Pipenv wants to make it easier to manage complex interdependencies. Your app might depend on a specific version of a library, and that library might depend on a specific version of another

library, and it's just dependencies and turtles all the way down. When two libraries your app uses have conflicting dependencies, your life can become hard. Pipenv wants to ease that pain by keeping track of a tree of your app's interdependencies in a file called `Pipfile.lock`. `Pipfile.lock` also verifies that the right versions of dependencies are used in production.

Also, Pipenv is handy when multiple developers are working on a project. With a `pip` workflow, Casey might install a library and spend two days implementing a new feature using that library. When Casey commits the changes, they might forget to run `pip freeze` to update the requirements file. The next day, Jamie pulls down Casey's changes, and suddenly tests are failing. It takes time to realize that the problem is libraries missing from the requirements file that Jamie doesn't have installed in the virtual environment.

Because Pipenv auto-documents dependencies as you install them, if Jamie and Casey had been using Pipenv, the `Pipfile` would have been automatically updated and included in Casey's commit. Jamie and Casey would have saved time and shipped their product faster.

Finally, using Pipenv signals to other people who work on your project that it ships with a standardized way to install project dependencies and development and testing requirements. Using a workflow with `pip` and requirements files means that you may have one single `requirements.txt` file, or several requirements files for different environments. It might not be clear to your colleagues whether they should run `dev.txt` or `local.txt` when they're running the project on their laptops, for example. It can also create confusion when two similar requirements files get wildly out of sync with each other: Is `local.txt` out of date, or is it really supposed to be that different from `dev.txt`? Multiple requirements files require more context and documentation to enable others to install the dependencies properly and as expected. This workflow has the potential to confuse colleagues and increase your maintenance burden.

Using Pipenv, which gives you `Pipfile`, lets you avoid these problems by managing dependencies for different environments for you. This command will install the main project dependencies:

```
pipenv install
```

Adding the `--dev` tag will install the dev/testing requirements:

```
pipenv install --dev
```

There are other benefits to using Pipenv: It has better security features, graphs your dependencies in an easier-to-understand format, seamlessly handles `.env` files, and can automatically handle differing dependencies for development versus production environments in one file. You can read more in the documentation (https://docs.pipenv.org/).

Pipenv in action

The basics of using Pipenv are detailed in the Managing Application Dependencies (https://packaging.python.org/tutorials/managing-dependencies/) section of the official Python packaging tutorial. To install Pipenv, use `pip`:

```
pip install pipenv
```

To install packages to use in your project, change into the directory for your project. Then to install a package (we'll use Django as an example), run:

```
pipenv install django
```

You will see some output that indicates that Pipenv is creating a `Pipfile` for your project.

If you aren't already using a virtual environment, you will also see some output from Pipenv saying it is creating a virtual environment for you.

Then, you will see the output you are used to seeing when you install packages.

To generate a `Pipfile.lock` file, run:

```
pipenv lock
```

You can also run Python scripts with Pipenv. To run a top-level Python script called `hello.py`, run:

```
pipenv run python hello.py
```

And you will see your expected result in the console.

To start a shell, run:

```
pipenv shell
```

If you would like to convert a project that currently uses a `requirements.txt` file to use Pipenv, install Pipenv and run:

```
pipenv install requirements.txt
```

This will create a Pipfile and install the specified requirements. Consider your project upgraded!

Learn more

Check out the Pipenv documentation, particularly Basic Usage of Pipenv (https://docs.pipenv.org/basics/), to take you further. Pipenv creator

Kenneth Reitz gave a talk on Pipenv, "The Future of Python Dependency Management (https://www.pytennessee.org/schedule/presentation/158/)," at a recent PyTennessee event. The talk wasn't recorded, but his slides (https://speakerdeck.com/kennethreitz/the-future-of-python-dependency-management) are helpful in understanding what Pipenv does and the problems it solves.

Topics :

Programming (/tags/programming) **Python** (/tags/python)

Python column (/tags/python-column)

(/users/laceynwilliams)

About the author

Lacey Williams Henschel - Lacey Williams Henschel is a software engineer with REVSYS and part of the organizing team for DjangoCon US. In the past, she's chaired DjangoCon US, organized several Django Girls workshops, taught courses for Treehouse, and written about accessibility at tech events.

• More about me (/users/laceynwilliams)

- Learn how you can contribute (/participate)

Contributors

(/users/jefftriplett)

Jeff Triplett

(/users/jefftriplett)

Recommended reading

What the pandas Python data analysis library and SQL taught me about taking an average (/article

Bring JavaScript to your Java enterprise with Vert.x (/article/18/4/benefits-javascript-vertx)

An introduction to the Flask Python web app framework (/article/18/4/flask)

OpenGL bindings for Bash (/article/18/4/opengl-bindings-bash)

Just say no to root (in containers) (/article/18/3/just-say-no-root-containers)

Python ChatOps libraries: Opsdroid and Errbot (/article/18/3/python-chatops-libraries-opsdroid-and-errbot)

3 Comments

VSumo (/users/viralsumo) on 28 Feb 2018 1

Looks promising will give it a try !

Nayiotus (/users/nayiotus) on 06 Mar 2018 0

This sounds rather intriguing and definately something worth giving a try! I'm all for

anything that streamlines dependencies :)

Thanks for the great article

Seth Kenlon (/users/seth) on 13 Mar 2018 0

Great article. I hadn't heard about this yet, but I'll be trying it immediately. Thanks!

Comment now

Login or Register (/user/login?destination=node/42601) to earn points for your comments.

Your name *

E-mail *

The content of this field is kept private and will not be shown publicly.

Accept the Terms of Use (/legal) to continue. You are licensing your contribution(s) as CC-BY-SA. *

CAPTCHA

This question is for testing whether or not you are a human visitor and to prevent automated spam submissions.

I'm not a robot reCAPTCHA Privacy - Terms

Join us on IRC

Connect with Opensource.com editors, community moderators, and writers on Freenode IRC in the #opensource.com channel.

Join us on IRC

Find us: