

Pipenv: A Guide to the New Python Packaging Tool

by Alexander VanTol ⌚ Apr 24, 2018 💬 53 Comments

🔑 **intermediate** **tools**

Mark as Completed 📌

🐦 Tweet 📱 Share ✉ Email

Table of Contents

- [Problems that Pipenv Solves](#)
 - [Dependency Management with requirements.txt](#)
 - [Development of Projects with Different Dependencies](#)
 - [Dependency Resolution](#)
- [Pipenv Introduction](#)
 - [Example Usage](#)
 - [Pipenv’s Dependency Resolution Approach](#)
 - [The Pipfile](#)
 - [The Pipfile.lock](#)
 - [Pipenv Extra Features](#)
- [Package Distribution](#)
 - [Yes, I need to distribute my code as a package](#)
 - [I don’t need to distribute my code as a package](#)
- [I already have a requirements.txt. How do I convert to a Pipfile?](#)
- [What’s next?](#)
- [Is Pipenv worth checking out?](#)
- [References, further reading, interesting discussions, and so forth](#)

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You

pythonistacafe.com

🔧 Remove ads

▶ Watch Now

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Working With Pipenv](#)

Help

Pipenv is a packaging tool for Python that solves some common problems associated with the typical workflow using `pip`, `virtualenv`, and the good old `requirements.txt`.

In addition to addressing some common issues, it consolidates and simplifies the development process to a single command line tool.

This guide will go over what problems Pipenv solves and how to [manage your Python dependencies](#) with Pipenv. Additionally, it will cover how Pipenv fits in with previous methods for [package](#) distribution.

Free Bonus: [Click here to get access to a free 5-day class](#) that shows you how to avoid common dependency management issues with tools like Pip, PyPI, Virtualenv, and requirements files.

Problems that Pipenv Solves

To understand the benefits of Pipenv, it’s important to walk through the current methods for packaging and dependency management in Python.

Let’s start with a typical situation of handling third-party packages. We’ll then build our way towards deploying a complete Python application.



[Remove ads](#)

Dependency Management with `requirements.txt`

Imagine you’re working on a [Python project](#) that uses a third-party package like `flask`. You’ll need to specify that requirement so that other developers and automated systems can run your application.

So you decide to include the `flask` dependency in a `requirements.txt` file:

Python Requirements

`flask`

Great, everything works fine locally, and after hacking away on your app for a while, you decide to move it to production. Here’s where things get a little scary...

The above `requirements.txt` file doesn’t specify which version of `flask` to use. In this case, `pip install -r requirements.txt` will install the latest version by default. This is okay unless there are interface or behavior changes in the newest version that break our application.

For the sake of this example, let’s say that a new version of `flask` got released. However, it isn’t backward compatible with the version you used during development.

Now, let’s say you deploy your application to production and do a `pip install -r requirements.txt`. [Pip](#) gets the latest, not-backward-compatible version of `flask`, and just like that, your application breaks... in production.

“*But hey, it worked on my machine!*”—I’ve been there myself, and it’s not a great feeling.

At this point, you know that the version of `flask` you used during development worked fine. So, to fix things, you try to be a little more specific in your `requirements.txt`. You add a *version specifier* to the `flask` dependency. This is also called *pinning* a dependency:

Python Requirements

`flask==0.12.1`

Pinning the `flask` dependency to a specific version ensures that a `pip install -r requirements.txt` sets up the exact version of `flask` you used during development. But does it really?

Keep in mind that `flask` itself has dependencies as well (which `pip` installs automatically). However, `flask` itself doesn't specify exact versions for its dependencies. For example, it allows any version of `Werkzeug>=0.14`.

Again, for the sake of this example, let's say a new version of `Werkzeug` got released, but it introduces a show-stopper bug to your application.

When you do `pip install -r requirements.txt` in production this time, you will get `flask==0.12.1` since you've pinned that requirement. However, unfortunately, you'll get the latest, buggy version of `Werkzeug`. Again, the product breaks in production.

The real issue here is that **the build isn't deterministic**. What I mean by that is that, given the same input (the `requirements.txt` file), `pip` doesn't always produce the same environment. At the moment, you can't easily replicate the exact environment you have on your development machine in production.

The typical solution to this problem is to use `pip freeze`. This command allows you to get exact versions for all 3rd party libraries currently installed, including the sub-dependencies `pip` installed automatically. So you can freeze everything in development to ensure that you have the same environment in production.

Executing `pip freeze` results in pinned dependencies you can add to a `requirements.txt`:

Python Requirements

```
click==6.7
Flask==0.12.1
itsdangerous==0.24
Jinja2==2.10
MarkupSafe==1.0
Werkzeug==0.14.1
```

With these pinned dependencies, you can ensure that the packages installed in your production environment match those in your development environment exactly, so your product doesn't unexpectedly break. This "solution," unfortunately, leads to a whole new set of problems.

Now that you've specified the exact versions of every third-party package, you are responsible for keeping these versions up to date, even though they're sub-dependencies of `flask`. What if there's a security hole discovered in `Werkzeug==0.14.1` that the package maintainers immediately patched in `Werkzeug==0.14.2`? You really need to update to `Werkzeug==0.14.2` to avoid any security issues arising from the earlier, unpatched version of `Werkzeug`.

First, you need to be aware that there's an issue with the version you have. Then, you need to get the new version in your production environment before someone exploits the security hole. So, you have to change your `requirements.txt` manually to specify the new version `Werkzeug==0.14.2`. As you can see in this situation, the responsibility of staying up to date with necessary updates falls on you.

The truth is that you really don't care what version of `Werkzeug` gets installed as long as it doesn't break your code. In fact, you probably want the latest version to ensure that you're getting bug fixes, security patches, new features, more optimization, and so on.

The real question is: **"How do you allow for deterministic builds for your Python project without gaining the responsibility of updating versions of sub-dependencies?"**

Spoiler alert: The easy answer is using Pipenv.

A Peer-to-Peer Learning Community for
Python Enthusiasts...Just Like You

pythonistacafe.com



 [Remove ads](#)

Development of Projects with Different Dependencies

Let's switch gears a bit to talk about another common issue that arises when you're working on multiple projects. Imagine that `ProjectA` needs `django==1.9`, but `ProjectB` needs `django==1.10`.

By default, Python tries to store all your third-party packages in a system-wide location. This means that every time you want to switch between ProjectA and ProjectB, you have to make sure the right version of django is installed. This makes switching between projects painful because you have to uninstall and reinstall packages to meet the requirements for each project.

The standard solution is to use a [virtual environment](#) that has its own Python executable and third-party package storage. That way, ProjectA and ProjectB are adequately separated. Now you can easily switch between projects since they’re not sharing the same package storage location. PackageA can have whatever version of django it needs in its own environment, and PackageB can have what it needs totally separate. A very common tool for this is virtualenv (or venv in Python 3).

Pipenv has virtual environment management built in so that you have a single tool for your package management.

Dependency Resolution

What do I mean by dependency resolution? Let’s say you’ve got a requirements.txt file that looks something like this:

Python Requirements

```
package_a
package_b
```

Let’s say package_a has a sub-dependency package_c, and package_a requires a specific version of this package: package_c>=1.0. In turn, package_b has the same sub-dependency but needs package_c<=2.0.

Ideally, when you try to install package_a and package_b, the installation tool would look at the requirements for package_c (being >=1.0 and <=2.0) and select a version that fulfills those requirements. You’d hope that the tool resolves the dependencies so that your program works in the end. This is what I mean by “dependency resolution.”

Unfortunately, pip itself doesn’t have real dependency resolution at the moment, but there’s an [open issue](#) to support it.

The way pip would handle the above scenario is as follows:

1. It installs package_a and looks for a version of package_c that fulfills the first requirement (package_c>=1.0).
2. Pip then installs the latest version of package_c to fulfill that requirement. Let’s say the latest version of package_c is 3.1.

This is where the trouble (potentially) starts.

If the version of package_c selected by pip doesn’t fit future requirements (such as package_b needing package_c<=2.0), the installation will fail.

The “solution” to this problem is to specify the range required for the sub-dependency (package_c) in the requirements.txt file. That way, pip can resolve this conflict and install a package that meets those requirements:

Python Requirements

```
package_c>=1.0, <=2.0
package_a
package_b
```

Just like before though, you’re now concerning yourself directly with sub-dependencies (package_c). The issue with this is that if package_a changes their requirement without you knowing, the requirements you specified (package_c>=1.0, <=2.0) may no longer be acceptable, and installation may fail... again. The real problem is that once again, you’re responsible for staying up to date with requirements of sub-dependencies.

Ideally, your installation tool would be smart enough to install packages that meet all the requirements without you explicitly specifying sub-dependency versions.

Pipenv Introduction

Now that we’ve addressed the problems, let’s see how Pipenv solves them.

First, let's install it:


Shell

```
$ pip install pipenv
```

Once you've done that, you can effectively forget about `pip` since Pipenv essentially acts as a replacement. It also introduces two new files, the `Pipfile` (which is meant to replace `requirements.txt`) and the `Pipfile.lock` (which enables deterministic builds).

Pipenv uses `pip` and `virtualenv` under the hood but simplifies their usage with a single command line interface.



 [Remove ads](#)

Example Usage

Let's start over with creating your awesome Python application. First, spawn a shell in a virtual environment to isolate the development of this app:

Shell

```
$ pipenv shell
```

This will create a virtual environment if one doesn't already exist. Pipenv creates all your virtual environments in a default location. If you want to change Pipenv's default behavior, there are some [environmental variables for configuration](#).

You can force the creation of a Python 2 or 3 environment with the arguments `--two` and `--three` respectively. Otherwise, Pipenv will use whatever default `virtualenv` finds.

Sidenote: If you require a more specific version of Python, you can provide a `--python` argument with the version you require. For example: `--python 3.6`

Now you can install the 3rd party package you need, `flask`. Oh, but you know that you need version `0.12.1` and not the latest version, so go ahead and be specific:

Shell

```
$ pipenv install flask==0.12.1
```

You should see something like the following in your terminal:

Shell

```
Adding flask==0.12.1 to Pipfile's [packages]...
Pipfile.lock not found, creating...
```

You'll notice that two files get created, a `Pipfile` and `Pipfile.lock`. We'll take a closer look at these in a second. Let's install another 3rd party package, `numpy`, for some number-crunching. You don't need a specific version so don't specify one:

Shell

```
$ pipenv install numpy
```

If you want to install something directly from a version control system (VCS), you can! You specify the locations similarly to how you'd do so with `pip`. For example, to install the `requests` library from version control, do the following:

Shell

```
$ pipenv install -e git+https://github.com/requests/requests.git#egg=requests
```

Note the `-e` argument above to make the installation editable. Currently, [this is required](#) for Pipenv to do sub-dependency resolution.

Let's say you also have some unit tests for this awesome application, and you want to use `pytest` for running them. You don't need `pytest` in production so you can specify that this dependency is only for development with the `--dev` argument:

Shell

```
$ pipenv install pytest --dev
```

Providing the `--dev` argument will put the dependency in a special `[dev-packages]` location in the `Pipfile`. These development packages only get installed if you specify the `--dev` argument with `pipenv install`.

The different sections separate dependencies needed only for development from ones needed for the base code to actually work. Typically, this would be accomplished with additional requirements files like `dev-requirements.txt` or `test-requirements.txt`. Now, everything is consolidated in a single `Pipfile` under different sections.

Okay, so let's say you've got everything working in your local development environment and you're ready to push it to production. To do that, you need to lock your environment so you can ensure you have the same one in production:

Shell

```
$ pipenv lock
```

This will create/update your `Pipfile.lock`, which you'll never need to (and are never meant to) edit manually. You should always use the generated file.

Now, once you get your code and `Pipfile.lock` in your production environment, you should install the last successful environment recorded:

Shell

```
$ pipenv install --ignore-pipfile
```

This tells Pipenv to ignore the `Pipfile` for installation and use what's in the `Pipfile.lock`. Given this `Pipfile.lock`, Pipenv will create the exact same environment you had when you ran `pipenv lock`, sub-dependencies and all.

The lock file enables deterministic builds by taking a snapshot of all the versions of packages in an environment (similar to the result of a `pip freeze`).

Now let's say another developer wants to make some additions to your code. In this situation, they would get the code, including the `Pipfile`, and use this command:

Shell

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

This installs all the dependencies needed for development, which includes both the regular dependencies and those you specified with the `--dev` argument during `install`.

When an exact version isn't specified in the `Pipfile`, the `install` command gives the opportunity for dependencies (and sub-dependencies) to update their versions.

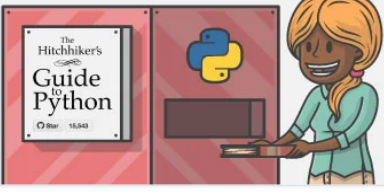
This is an important note because it solves some of the previous problems we discussed. To demonstrate, let's say a new version of one of your dependencies becomes available. Because you don't need a specific version of this dependency, you don't specify an exact version in the `Pipfile`. When you `pipenv install`, the new version of the dependency will be installed in your development environment.

Now you make your changes to the code and run some tests to verify everything is still working as expected. (You do have unit tests, right?) Now, just as before, you lock your environment with `pipenv lock`, and an updated `Pipfile.lock` will be generated with the new version of the dependency. Just as before, you can replicate this new environment in production with the lock file.

As you can see from this scenario, you no longer have to force exact versions you don't truly need to ensure your development and production environments are the same. You also don't need to stay on top of updating sub-dependencies you "don't care about." This workflow with Pipenv, combined with your excellent testing, fixes the issues of manually doing all your dependency management.

A Python Best Practices Handbook

python-guide.org



 [Remove ads](#)

Pipenv's Dependency Resolution Approach

Pipenv will attempt to install sub-dependencies that satisfy all the requirements from your core dependencies. However, if there are conflicting dependencies (package_a needs package_c>=1.0, but package_b needs package_c<1.0), Pipenv will not be able to create a lock file and will output an error like the following:

```
Warning: Your dependencies could not be resolved. You likely have a mismatch in your sub-dependencies
You can use $ pipenv install --skip-lock to bypass this mechanism, then run $ pipenv graph to inspect
Could not find a version that matches package_c>=1.0,package_c<1.0
```

As the warning says, you can also show a dependency graph to understand your top-level dependencies and their sub-dependencies:

Shell

```
$ pipenv graph
```

This command will print out a tree-like structure showing your dependencies. Here's an example:

```
Flask==0.12.1
- click [required: >=2.0, installed: 6.7]
- itsdangerous [required: >=0.21, installed: 0.24]
- Jinja2 [required: >=2.4, installed: 2.10]
  - MarkupSafe [required: >=0.23, installed: 1.0]
- Werkzeug [required: >=0.7, installed: 0.14.1]
numpy==1.14.1
pytest==3.4.1
- attrs [required: >=17.2.0, installed: 17.4.0]
- funcsigns [required: Any, installed: 1.0.2]
- pluggy [required: <0.7,>=0.5, installed: 0.6.0]
- py [required: >=1.5.0, installed: 1.5.2]
- setuptools [required: Any, installed: 38.5.1]
- six [required: >=1.10.0, installed: 1.11.0]
requests==2.18.4
- certifi [required: >=2017.4.17, installed: 2018.1.18]
- chardet [required: >=3.0.2,<3.1.0, installed: 3.0.4]
- idna [required: >=2.5,<2.7, installed: 2.6]
- urllib3 [required: <1.23,>=1.21.1, installed: 1.22]
```

From the output of `pipenv graph`, you can see the top-level dependencies we installed previously (Flask, numpy, pytest, and requests), and underneath you can see the packages they depend on.

Additionally, you can reverse the tree to show the sub-dependencies with the parent that requires it:

Shell

```
$ pipenv graph --reverse
```

This reversed tree may be more useful when you are trying to figure out conflicting sub-dependencies.

The Pipfile

[Pipfile](#) intends to replace `requirements.txt`. Pipenv is currently the reference implementation for using Pipfile. It seems very likely that [pip itself will be able to handle these files](#). Also, it's worth noting that [Pipenv is even the official package management tool recommended by Python itself](#).

The syntax for the Pipfile is [TOML](#), and the file is separated into sections. `[dev-packages]` for development-only packages, `[packages]` for minimally required packages, and `[requires]` for other requirements like a specific version of Python. See an example file below:

Config File

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

[dev-packages]
pytest = "*"

[packages]
flask = "==0.12.1"
numpy = "*"
requests = {git = "https://github.com/requests/requests.git", editable = true}

[requires]
python_version = "3.6"
```

Ideally, you shouldn't have any sub-dependencies in your Pipfile. What I mean by that is you should only include the packages you actually import and use. No need to keep `chardet` in your Pipfile just because it's a sub-dependency of `requests`. (Pipenv will install it automatically.) The Pipfile should convey the top-level dependencies your package requires.

The Pipfile.lock

This file enables deterministic builds by specifying the exact requirements for reproducing an environment. It contains exact versions for packages and hashes to support more secure verification, which [pip itself now supports](#) as well. An example file might look like the following. Note that the syntax for this file is JSON and that I've excluded parts of the file with `...`:

JSON

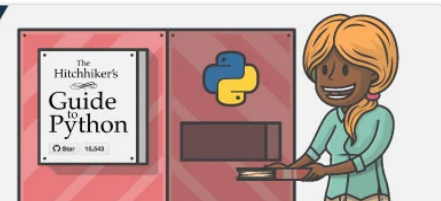
```
{
  "_meta": {
    ...
  },
  "default": {
    "flask": {
      "hashes": [
        "sha256:6c3130c8927109a08225993e4e503de4ac4f2678678ae211b33b519c622a7242",
        "sha256:9dce4b6bfbb5b062181d3f7da8f727ff70c1156cbb4024351eafd426deb5fb88"
      ],
      "version": "==0.12.1"
    },
    "requests": {
      "editable": true,
      "git": "https://github.com/requests/requests.git",
      "ref": "4ea09e49f7d518d365e7c6f7ff6ed9ca70d6ec2e"
    },
    "werkzeug": {
      "hashes": [
        "sha256:d5da73735293558eb1651ee2fddc4d0dedcfa06538b8813a2e20011583c9e49b",
        "sha256:c3fd7a7d41976d9f44db327260e263132466836cef6f91512889ed60ad26557c"
      ],
      "version": "==0.14.1"
    }
    ...
  },
  "develop": {
    "pytest": {
      "hashes": [
        "sha256:8970e25181e15ab14ae895599a0a0e0ade7d1f1c4c8ca1072ce16f25526a184d",
        "sha256:9ddcb879c8cc859d2540204b5399011f842e5e8823674bf429f70ada281b3cc6"
      ],
      "version": "==3.4.1"
    }
    ...
  }
}
```

Note the exact version specified for every dependency. Even the sub-dependencies like `werkzeug` that aren't in our `Pipfile` appear in this `Pipfile.lock`. The hashes are used to ensure you're retrieving the same package as you did in development.

It's worth noting again that you should never change this file by hand. It is meant to be generated with `pipenv lock`.

Your Guide to the Python Programming Language and a Best Practices Handbook

python-guide.org



[Remove ads](#)

Pipenv Extra Features

Open a third-party package in your default editor with the following command:

Shell

```
$ pipenv open flask
```

This will open the `flask` package in the default editor, or you can specify a program with an `EDITOR` environmental variable. For example, I use [Sublime Text](#), so I just set `EDITOR=subl`. This makes it super simple to dig into the internals of a package you're using.

You can run a command in the virtual environment without launching a shell:

Shell

```
$ pipenv run <insert command here>
```

Check for security vulnerabilities (and [PEP 508](#) requirements) in your environment:

Shell

```
$ pipenv check
```

Now, let’s say you no longer need a package. You can uninstall it:

Shell

```
$ pipenv uninstall numpy
```

Additionally, let’s say you want to completely wipe all the installed packages from your virtual environment:

Shell

```
$ pipenv uninstall --all
```

You can replace `--all` with `--all-dev` to just remove dev packages.

Pipenv supports the automatic loading of environmental variables when a `.env` file exists in the top-level directory. That way, when you `pipenv shell` to open the virtual environment, it loads your environmental variables from the file. The `.env` file just contains key-value pairs:

Text

```
SOME_ENV_CONFIG=some_value
SOME_OTHER_ENV_CONFIG=some_other_value
```

Finally, here are some quick commands to find out where stuff is. How to find out where your virtual environment is:


Shell

```
$ pipenv --venv
```

How to find out where your project home is:

Shell

```
$ pipenv --where
```



 [The Real Python Podcast »](#)

 [Remove ads](#)

Package Distribution

You may be asking how this all works if you intend to distribute your code as a package.

Yes, I need to distribute my code as a package

How does Pipenv work with `setup.py` files?

There are a lot of nuances to that question. First, a `setup.py` file is necessary when you're using `setuptools` as your build/distribution system. This has been the de facto standard for a while now, but [recent changes](#) have made the use of `setuptools` optional.

This means that projects like [flit](#) can use the new [pyproject.toml](#) to specify a different build system that doesn't require a `setup.py`.

All that being said, for the near future `setuptools` and an accompanying `setup.py` will still be the default choice for many people.

Here's a recommended workflow for when you are using a `setup.py` as a way to distribute your package:

- `setup.py`
- `install_requires` keyword should include whatever the package ["minimally needs to run correctly."](#)
- `Pipfile`
- Represents the concrete requirements for your package
- Pull the minimally required dependencies from `setup.py` by installing your package using Pipenv:
 - Use `pipenv install '-e .'`
 - That will result in a line in your `Pipfile` that looks something like `"e1839a8" = {path = ".", editable = true}`.
- `Pipfile.lock`
- Details for a reproducible environment generated from `pipenv lock`

To clarify, put your minimum requirements in `setup.py` instead of directly with `pipenv install`. Then use the `pipenv install '-e .'` command to install your package as editable. This gets all the requirements from `setup.py` into your environment. Then you can use `pipenv lock` to get a reproducible environment.

I don't need to distribute my code as a package

Great! If you are developing an application that isn't meant to be distributed or installed (a personal website, a desktop application, a game, or similar), you don't really need a `setup.py`.

In this situation, you could use `Pipfile/Pipfile.lock` combo for managing your dependencies with the flow described previously to deploy a reproducible environment in production.

I already have a `requirements.txt`. How do I convert to a `Pipfile`?

If you run `pipenv install` it should automatically detect the `requirements.txt` and convert it to a `Pipfile`, outputting something like the following:

```
requirements.txt found, instead of Pipfile! Converting...
Warning: Your Pipfile now contains pinned versions, if your requirements.txt did.
We recommend updating your Pipfile to specify the "*" version, instead.
```

Take note of the above warning.

If you have pinned exact versions in your `requirements.txt` file, you'll probably want to change your `Pipfile` to only specify exact versions you truly require. This will allow you to gain the real benefits of transitioning. For example, let's say you have the following but really don't need that exact version of `numpy`:

Config File

```
[packages]
numpy = "==1.14.1"
```

If you don't have any specific version requirements for your dependencies, you can use the wildcard character `*` to tell Pipenv that any version can be installed:

Config File

```
[packages]
numpy = "*"
```

If you feel nervous about allowing any version with the `*`, it's typically a safe bet to specify greater than or equal to the version you're already on so you can still take advantage of new versions:

Config File

```
[packages]
numpy = ">=1.14.1"
```

Of course, staying up to date with new releases also means you're responsible for ensuring your code still functions as expected when packages change. This means a test suite is essential to this whole Pipenv flow if you want to ensure functioning releases of your code.

You allow packages to update, run your tests, ensure they all pass, lock your environment, and then you can rest easy knowing that you haven't introduced breaking changes. If things do break because of a dependency, you've got some regression tests to write and potentially some more restrictions on versions of dependencies.

For example, if `numpy==1.15` gets installed after running `pipenv install` and it breaks your code, which you hopefully either notice during development or during your tests, you have a couple options:

1. Update your code to function with the new version of the dependency.

If backward compatibility with previous versions of the dependency isn't possible, you'll also need to bump your required version in your Pipfile:

Config File

```
[packages]
numpy = ">=1.15"
```

2. Restrict the version of the dependency in the Pipfile to be `<` the version that just broke your code:

Config File

```
[packages]
numpy = ">=1.14.1,<1.15"
```

Option 1 is preferred as it ensures that your code is using the most up-to-date dependencies. However, Option 2 takes less time and doesn't require code changes, just restrictions on dependencies.

You can also install from requirement files with the same `-r` argument `pip` takes:

Shell

```
$ pipenv install -r requirements.txt
```

If you have a `dev-requirements.txt` or something similar, you can add those to the Pipfile as well. Just add the `--dev` argument so it gets put in the right section:

Shell

```
$ pipenv install -r dev-requirements.txt --dev
```

Additionally, you can go the other way and generate requirements files from a Pipfile:

Shell

```
$ pipenv lock -r > requirements.txt
$ pipenv lock -r -d > dev-requirements.txt
```


Your Weekly Dose of All Things Python!

pycoders.com



 [Remove ads](#)

What's next?

It appears to me that a natural progression for the Python ecosystem would be a build system that uses the `Pipfile` to install the minimally required dependencies when retrieving and building a package from a package index (like PyPI). It is important to note again that the [Pipfile design specification](#) is still in development, and Pipenv is just a reference implementation.

That being said, I could see a future where the `install_requires` section of `setup.py` doesn't exist, and the `Pipfile` is referenced for minimal requirements instead. Or the `setup.py` is gone entirely, and you get metadata and other information in a different manner, still using the `Pipfile` to get the necessary dependencies.

Is Pipenv worth checking out?

Definitely. Even if it's just as a way to consolidate the tools you already use (`pip` & `virtualenv`) into a single interface. However, it's much more than that. With the addition of the `Pipfile`, you only specify the dependencies you truly need.

You no longer have the headache of managing the versions of everything yourself just to ensure you can replicate your development environment. With the `Pipfile.lock`, you can develop with peace of mind knowing that you can exactly reproduce your environment anywhere.

In addition to all that, it seems very likely that the `Pipfile` format will get adopted and supported by official Python tools like `pip`, so it'd be beneficial to be ahead of the game. Oh, and make sure you're updating all your code to Python 3 as well: [2020 is coming up fast](#).


References, further reading, interesting discussions, and so forth

- [Official Pipenv documentation](#)
- [Official Pipfile Project](#)
- [Issue addressing `install_requires` in regards to `Pipfile`](#)
- [More discussion on `setup.py` vs `Pipfile`](#)
- [Post talking about PEP 518](#)
- [Post on Python packaging](#)
- [Comment suggesting Pipenv usage](#)

Free Bonus: [Click here to get access to a free 5-day class](#) that shows you how to avoid common dependency management issues with tools like `Pip`, `PyPI`, `Virtualenv`, and `requirements` files.

Mark as Completed



 **Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Working With Pipenv](#)



Python Tricks 