# pipenv Documentation

*Release 2018.05.18*

**Kenneth Reitz**

**Jul 03, 2018**

# Contents

**Pipenv** — the officially recommended Python packaging tool from Python.org, free (as in freedom).

Pipenv is a tool that aims to bring the best of all packaging worlds (bundler, composer, npm, cargo, yarn, etc.) to the Python world. *Windows is a first-class citizen, in our world.*

It automatically creates and manages a virtualenv for your projects, as well as adds/removes packages from your `Pipfile` as you install/uninstall packages. It also generates the ever-important `Pipfile.lock`, which is used to produce deterministic builds.

Pipenv is primarily meant to provide users and developers of applications with an easy method to setup a working environment. For the distinction between libraries and applications and the usage of `setup.py` vs `Pipfile` to define dependencies, see *Pipfile vs setup.py*.

The problems that Pipenv seeks to solve are multi-faceted:

- You no longer need to use `pip` and `virtualenv` separately. They work together.
- Managing a `requirements.txt` file can be problematic, so Pipenv uses `Pipfile` and `Pipfile.lock` to separate abstract dependency declarations from the last tested combination.
- Hashes are used everywhere, always. Security. Automatically expose security vulnerabilities.
- Strongly encourage the use of the latest versions of dependencies to minimize security risks arising from outdated components.
- Give you insight into your dependency graph (e.g. `$ pipenv graph`).
- Streamline development workflow by loading `.env` files.

CHAPTER 1

# Install Pipenv Today!

Just use pip:

```
$ pip install pipenv
```

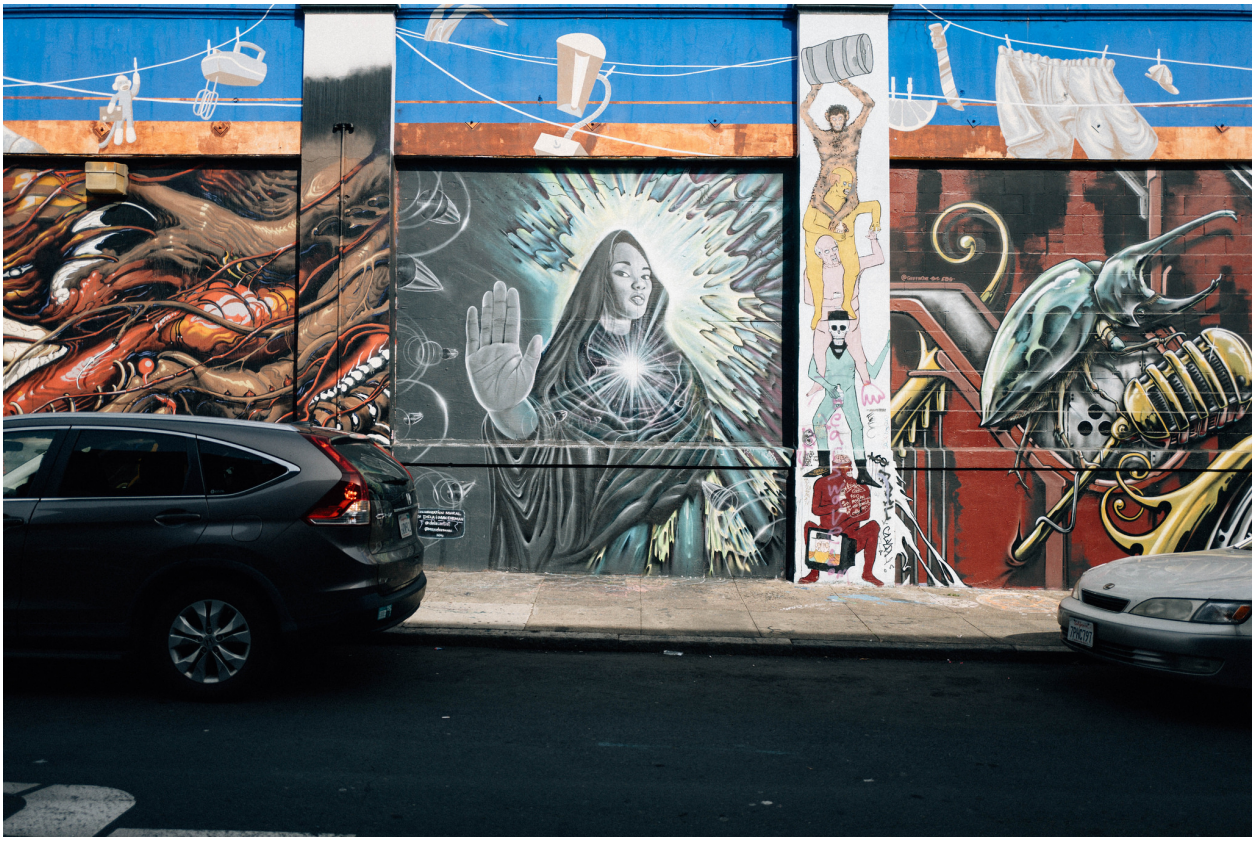
Or, if you're using Ubuntu 17.10:

```
$ sudo apt install software-properties-common python-software-properties
$ sudo add-apt-repository ppa:pypa/ppa
$ sudo apt update
$ sudo apt install pipenv
```

Otherwise, if you're on MacOS, you can install Pipenv easily with Homebrew:

```
$ brew install pipenv
```

## 1.1 Pipenv & Virtual Environments



This tutorial walks you through installing and using Python packages.

It will show you how to install and use the necessary tools and make strong recommendations on best practices. Keep in mind that Python is used for a great many different purposes, and precisely how you want to manage your dependencies may change based on how you decide to publish your software. The guidance presented here is most directly applicable to the development and deployment of network services (including web applications), but is also very well suited to managing development and testing environments for any kind of project.

---

**Note:** This guide is written for Python 3, however, these instructions should work fine on Python 2.7—if you are still using it, for some reason.

---

### 1.1.1 Make sure you've got Python & pip

Before you go any further, make sure you have Python and that it's available from your command line. You can check this by simply running:

```
$ python --version
```

You should get some output like `3.6.2`. If you do not have Python, please install the latest 3.x version from python.org or refer to the Installing Python section of *The Hitchhiker's Guide to Python*.

---

**Note:** If you're newcomer and you get an error like this:

```
>>> python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
```

It's because this command is intended to be run in a *shell* (also called a *terminal* or *console*). See the Python for Beginners getting started tutorial for an introduction to using your operating system's shell and interacting with Python.

Additionally, you'll need to make sure you have pip available. You can check this by running:

```
$ pip --version
pip 9.0.1
```

If you installed Python from source, with an installer from python.org, or via **'Homebrew'_** you should already have pip. If you're on Linux and installed using your OS package manager, you may have to install pip separately.

If you plan to install pipenv using Homebrew you can skip this step. The Homebrew installer takes care of pip for you.

### 1.1.2 Installing Pipenv

Pipenv is a dependency manager for Python projects. If you're familiar with Node.js' npm or Ruby's bundler, it is similar in spirit to those tools. While pip can install Python packages, Pipenv is recommended as it's a higher-level tool that simplifies dependency management for common use cases.

Use `pip` to install Pipenv:

```
$ pip install --user pipenv
```

**Note:** This does a user installation to prevent breaking any system-wide packages. If `pipenv` isn't available in your shell after installation, you'll need to add the user base's binary directory to your `PATH`.

On Linux and macOS you can find the user base binary directory by running `python -m site --user-base` and adding `bin` to the end. For example, this will typically print `~/.local` (with `~` expanded to the absolute path to your home directory) so you'll need to add `~/.local/bin` to your `PATH`. You can set your `PATH` permanently by modifying ~/.profile.

On Windows you can find the user base binary directory by running `py -m site --user-site` and replacing `site-packages` with `Scripts`. For example, this could return `C:\Users\Username\AppData\Roaming\Python36\site-packages` so you would need to set your `PATH` to include `C:\Users\Username\AppData\Roaming\Python36\Scripts`. You can set your user `PATH` permanently in the Control Panel. You may need to log out for the `PATH` changes to take effect.

### 1.1.3 Installing packages for your project

Pipenv manages dependencies on a per-project basis. To install packages, change into your project's directory (or just an empty directory for this tutorial) and run:

```
$ cd myproject
$ pipenv install requests
```

Pipenv will install the excellent Requests library and create a `Pipfile` for you in your project's directory. The Pipfile is used to track which dependencies your project needs in case you need to re-install them, such as when you share your project with others. You should get output similar to this (although the exact paths shown will vary):

```
Creating a Pipfile for this project...
Creating a virtualenv for this project...
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/
↪Versions/3.6'
New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
Installing setuptools, pip, wheel...done.

Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
Installing requests...
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, chardet, certifi, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4␣
↪urllib3-1.22

Adding requests to Pipfile's [packages]...
P.S. You have excellent taste!
```

### 1.1.4 Using installed packages

Now that Requests is installed you can create a simple `main.py` file to use it:

```
import requests

response = requests.get('https://httpbin.org/ip')

print('Your IP is {0}'.format(response.json()['origin']))
```

Then you can run this script using `pipenv run`:

```
$ pipenv run python main.py
```

You should get output similar to this:

```
Your IP is 8.8.8.8
```

Using `$ pipenv run` ensures that your installed packages are available to your script. It's also possible to spawn a new shell that ensures all commands have access to your installed packages with `$ pipenv shell`.

### 1.1.5 Next steps

Congratulations, you now know how to install and use Python packages!

## 1.2 Homebrew Installation of Pipenv

Homebrew is a popular open-source package management system for macOS.

Installing pipenv via Homebrew will keep pipenv and all of its dependencies in an isolated virtual environment so it doesn't interfere with the rest of your Python installation.

Once you have installed Homebrew simply run:

```
$ brew install pipenv
```

To upgrade pipenv at any time:

```
$ brew upgrade pipenv
```

## 1.3 Pragmatic Installation of Pipenv

If you have a working installation of pip, and maintain certain "toolchain" type Python modules as global utilities in your user environment, pip user installs allow for installation into your home directory. Note that due to interaction between dependencies, you should limit tools installed in this way to basic building blocks for a Python workflow like virtualenv, pipenv, tox, and similar software.

To install:

```
$ pip install --user pipenv
```

For more information see the user installs documentation, but to add the installed cli tools from a pip user install to your path, add the output of:

```
$ python -c "import site; import os; print(os.path.join(site.USER_BASE, 'bin'))"
```

To upgrade pipenv at any time:

```
$ pip install --user --upgrade pipenv
```

## 1.4 Crude Installation of Pipenv

If you don't even have pip installed, you can use this crude installation method, which will bootstrap your whole system:

```
$ curl https://raw.githubusercontent.com/kennethreitz/pipenv/master/get-pipenv.py |␣
↪python
```

Congratulations, you now have pip and Pipenv installed!

CHAPTER 2

# User Testimonials

**Jannis Leidel, former pip maintainer—** *Pipenv is the porcelain I always wanted to build for pip. It fits my brain and mostly replaces virtualenvwrapper and manual pip calls for me. Use it.*

**David Gang—** *This package manager is really awesome. For the first time I know exactly what my dependencies are which I installed and what the transitive dependencies are. Combined with the fact that installs are deterministic, makes this package manager first class, like cargo.*

**Justin Myles Holmes—** *Pipenv is finally an abstraction meant to engage the mind instead of merely the filesystem.*

CHAPTER 3

# Pipenv Features

- Enables truly *deterministic builds*, while easily specifying *only what you want*.
- Generates and checks file hashes for locked dependencies.
- Automatically install required Pythons, if `pyenv` is available.
- Automatically finds your project home, recursively, by looking for a `Pipfile`.
- Automatically generates a `Pipfile`, if one doesn't exist.
- Automatically creates a virtualenv in a standard location.
- Automatically adds/removes packages to a `Pipfile` when they are un/installed.
- Automatically loads `.env` files, if they exist.

The main commands are `install`, `uninstall`, and `lock`, which generates a `Pipfile.lock`. These are intended to replace `$ pip install` usage, as well as manual virtualenv management (to activate a virtualenv, run `$ pipenv shell`).

## 3.1 Basic Concepts

- A virtualenv will automatically be created, when one doesn't exist.
- When no parameters are passed to `install`, all packages `[packages]` specified will be installed.
- To initialize a Python 3 virtual environment, run `$ pipenv --three`.
- To initialize a Python 2 virtual environment, run `$ pipenv --two`.
- Otherwise, whatever virtualenv defaults to will be the default.

## 3.2 Other Commands

- `graph` will show you a dependency graph, of your installed dependencies.

- `shell` will spawn a shell with the virtualenv activated.
- `run` will run a given command from the virtualenv, with any arguments forwarded (e.g. `$ pipenv run python` or `$ pipenv run pip freeze`).
- `check` checks for security vulnerabilities and asserts that PEP 508 requirements are being met by the current environment.

CHAPTER 4

# Further Documentation Guides

## 4.1 Basic Usage of Pipenv

This document covers some of Pipenv's more basic features.

### 4.1.1 Example Pipfile & Pipfile.lock

Here is a simple example of a `Pipfile` and the resulting `Pipfile.lock`.

#### Example Pipfile

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

[packages]
requests = "*"


[dev-packages]
pytest = "*"
```

#### Example Pipfile.lock

```
{
    "_meta": {
        "hash": {
            "sha256":
↪"8d14434df45e0ef884d6c3f6e8048ba72335637a8631cc44792f52fd20b6f97a"
        },
        "host-environment-markers": {
            "implementation_name": "cpython",
            "implementation_version": "3.6.1",
            "os_name": "posix",
            "platform_machine": "x86_64",
            "platform_python_implementation": "CPython",
            "platform_release": "16.7.0",
            "platform_system": "Darwin",
            "platform_version": "Darwin Kernel Version 16.7.0: Thu Jun 15 17:36:27␣
↪PDT 2017; root:xnu-3789.70.16~2/RELEASE_X86_64",
            "python_full_version": "3.6.1",
            "python_version": "3.6",
            "sys_platform": "darwin"
        },
        "pipfile-spec": 5,
        "requires": {},
        "sources": [
            {
                "name": "pypi",
                "url": "https://pypi.python.org/simple",
                "verify_ssl": true
            }
        ]
    },
    "default": {
        "certifi": {
            "hashes": [

↪"sha256:54a07c09c586b0e4c619f02a5e94e36619da8e2b053e20f594348c0611803704",
```

```
↪"sha256:40523d2efb60523e113b44602298f0960e900388cf3bb6043f645cf57ea9e3f5"
            ],
            "version": "==2017.7.27.1"
        },
        "chardet": {
            "hashes": [

↪"sha256:fc323ffcaeaed0e0a02bf4d117757b98aed530d9ed4531e3e15460124c106691",

↪"sha256:84ab92ed1c4d4f16916e05906b6b75a6c0fb5db821cc65e70cbd64a3e2a5eaae"
            ],
            "version": "==3.0.4"
        },
        "idna": {
            "hashes": [

↪"sha256:8c7309c718f94b3a625cb648ace320157ad16ff131ae0af362c9f21b80ef6ec4",

↪"sha256:2c6a5de3089009e3da7c5dde64a141dbc8551d5b7f6cf4ed7c2568d0cc520a8f"
            ],
            "version": "==2.6"
        },
        "requests": {
            "hashes": [

↪"sha256:6a1b267aa90cac58ac3a765d067950e7dbbf75b1da07e895d1f594193a40a38b",

↪"sha256:9c443e7324ba5b85070c4a818ade28bfabedf16ea10206da1132edaa6dda237e"
            ],
            "version": "==2.18.4"
        },
        "urllib3": {
            "hashes": [

↪"sha256:06330f386d6e4b195fbfc736b297f58c5a892e4440e54d294d7004e3a9bbea1b",

↪"sha256:cc44da8e1145637334317feebd728bd869a35285b93cbb4cca2577da7e62db4f"
            ],
            "version": "==1.22"
        }
    },
    "develop": {
        "py": {
            "hashes": [

↪"sha256:2ccb79b01769d99115aa600d7eed99f524bf752bba8f041dc1c184853514655a",

↪"sha256:0f2d585d22050e90c7d293b6451c83db097df77871974d90efd5a30dc12fcde3"
            ],
            "version": "==1.4.34"
        },
        "pytest": {
            "hashes": [

↪"sha256:b84f554f8ddc23add65c411bf112b2d88e2489fd45f753b1cae5936358bdf314",

↪"sha256:f46e49e0340a532764991c498244a60e3a37d7424a532b3ff1a6a7653f1a403a"
```

```
            ],
            "version": "==3.2.2"
        }
    }
}
```

### 4.1.2 General Recommendations & Version Control

- Generally, keep both `Pipfile` and `Pipfile.lock` in version control.
- Do not keep `Pipfile.lock` in version control if multiple versions of Python are being targeted.
- Specify your target Python version in your *Pipfile*'s `[requires]` section. Ideally, you should only have one target Python version, as this is a deployment tool.
- `pipenv install` is fully compatible with `pip install` syntax, for which the full documentation can be found here.

### 4.1.3 Example Pipenv Workflow

Clone / create project repository:

```
...
$ cd myproject
```

Install from Pipfile, if there is one:

```
$ pipenv install
```

Or, add a package to your new project:

```
$ pipenv install <package>
```

This will create a `Pipfile` if one doesn't exist. If one does exist, it will automatically be edited with the new package your provided.

Next, activate the Pipenv shell:

```
$ pipenv shell
$ python --version
...
```

### 4.1.4 Example Pipenv Upgrade Workflow

- Find out what's changed upstream: `$ pipenv update --outdated`.
- **Upgrade packages, two options:**
    1. Want to upgrade everything? Just do `$ pipenv update`.
    2. Want to upgrade packages one-at-a-time? `$ pipenv update <pkg>` for each outdated package.

### 4.1.5 Importing from requirements.txt

If you only have a `requirements.txt` file available when running `pipenv install`, pipenv will automatically import the contents of this file and create a `Pipfile` for you.

You can also specify `$ pipenv install -r path/to/requirements.txt` to import a requirements file.

If your requirements file has version numbers pinned, you'll likely want to edit the new `Pipfile` to remove those, and let `pipenv` keep track of pinning. If you want to keep the pinned versions in your `Pipfile.lock` for now, run `pipenv lock --keep-outdated`. Make sure to *upgrade* soon!

### 4.1.6 Specifying Versions of a Package

To tell pipenv to install a specific version of a library, the usage is simple:

```
$ pipenv install requests==2.13.0
```

This will update your `Pipfile` to reflect this requirement, automatically.

### 4.1.7 Specifying Versions of Python

To create a new virtualenv, using a specific version of Python you have installed (and on your `PATH`), use the `--python VERSION` flag, like so:

Use Python 3:

```
$ pipenv --python 3
```

Use Python3.6:

```
$ pipenv --python 3.6
```

Use Python 2.7.14:

```
$ pipenv --python 2.7.14
```

When given a Python version, like this, Pipenv will automatically scan your system for a Python that matches that given version.

If a `Pipfile` hasn't been created yet, one will be created for you, that looks like this:

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

[dev-packages]

[packages]

[requires]
python_version = "3.6"
```

Note the inclusion of `[requires] python_version = "3.6"`. This specifies that your application requires this version of Python, and will be used automatically when running `pipenv install` against this `Pipfile` in the future (e.g. on other machines). If this is not true, feel free to simply remove this section.

If you don't specify a Python version on the command–line, either the `[requires] python_full_version` or `python_version` will be selected automatically, falling back to whatever your system's default `python` installation is, at time of execution.

### 4.1.8 Editable Dependencies (e.g. `-e .`)

You can tell Pipenv to install a path as editable — often this is useful for the current working directory when working on packages:

```
$ pipenv install --dev -e .

$ cat Pipfile
...
[dev-packages]
"e1839a8" = {path = ".", editable = true}
...
```

Note that all sub-dependencies will get added to the `Pipfile.lock` as well.

**Note:** Sub-dependencies are **not** added to the `Pipfile.lock` if you leave the `-e` option out.

### 4.1.9 Environment Management with Pipenv

The three primary commands you'll use in managing your pipenv environment are `$ pipenv install`, `$ pipenv uninstall`, and `$ pipenv lock`.

#### $ pipenv install

`$ pipenv install` is used for installing packages into the pipenv virtual environment and updating your Pipfile.

Along with the basic install command, which takes the form:

```
$ pipenv install [package names]
```

The user can provide these additional parameters:

- `--two` — Performs the installation in a virtualenv using the system `python2` link.
- `--three` — Performs the installation in a virtualenv using the system `python3` link.
- `--python` — Performs the installation in a virtualenv using the provided Python interpreter.

**Warning:** None of the above commands should be used together. They are also **destructive** and will delete your current virtualenv before replacing it with an appropriately versioned one.

**Note:** The virtualenv created by Pipenv may be different from what you were expecting. Dangerous characters (i.e. `$` `` ` `` `! * @ "` as well as space, line feed, carriage return, and tab) are converted to underscores. Additionally, the full path to the current folder is encoded into a "slug value" and appended to ensure the virtualenv name is unique.

- `--dev` — Install both `develop` and `default` packages from `Pipfile.lock`.
- `--system` — Use the system `pip` command rather than the one from your virtualenv.
- `--ignore-pipfile` — Ignore the `Pipfile` and install from the `Pipfile.lock`.
- `--skip-lock` — Ignore the `Pipfile.lock` and install from the `Pipfile`. In addition, do not write out a `Pipfile.lock` reflecting changes to the `Pipfile`.

#### $ pipenv uninstall

`$ pipenv uninstall` supports all of the parameters in *pipenv install*, as well as two additional options, `--all` and `--all-dev`.

- `--all` — This parameter will purge all files from the virtual environment, but leave the Pipfile untouched.
- `--all-dev` — This parameter will remove all of the development packages from the virtual environment, and remove them from the Pipfile.

#### $ pipenv lock

`$ pipenv lock` is used to create a `Pipfile.lock`, which declares **all** dependencies (and sub-dependencies) of your project, their latest available versions, and the current hashes for the downloaded files. This ensures repeatable, and most importantly *deterministic*, builds.

### 4.1.10 About Shell Configuration

Shells are typically misconfigured for subshell use, so `$ pipenv shell --fancy` may produce unexpected results. If this is the case, try `$ pipenv shell`, which uses "compatibility mode", and will attempt to spawn a subshell despite misconfiguration.

A proper shell configuration only sets environment variables like `PATH` during a login session, not during every subshell spawn (as they are typically configured to do). In fish, this looks like this:

```
if status --is-login
    set -gx PATH /usr/local/bin $PATH
end
```

You should do this for your shell too, in your `~/.profile` or `~/.bashrc` or wherever appropriate.

---

**Note:** The shell launched in interactive mode. This means that if your shell reads its configuration from a specific file for interactive mode (e.g. bash by default looks for a `~/.bashrc` configuration file for interactive mode), then you'll need to modify (or create) this file.

---

### 4.1.11 A Note about VCS Dependencies

Pipenv will resolve the sub–dependencies of VCS dependencies, but only if they are installed in editable mode:

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

$ cat Pipfile
[packages]
requests = {git = "https://github.com/requests/requests.git", editable=true}
```

If editable is not true, sub–dependencies will not be resolved.

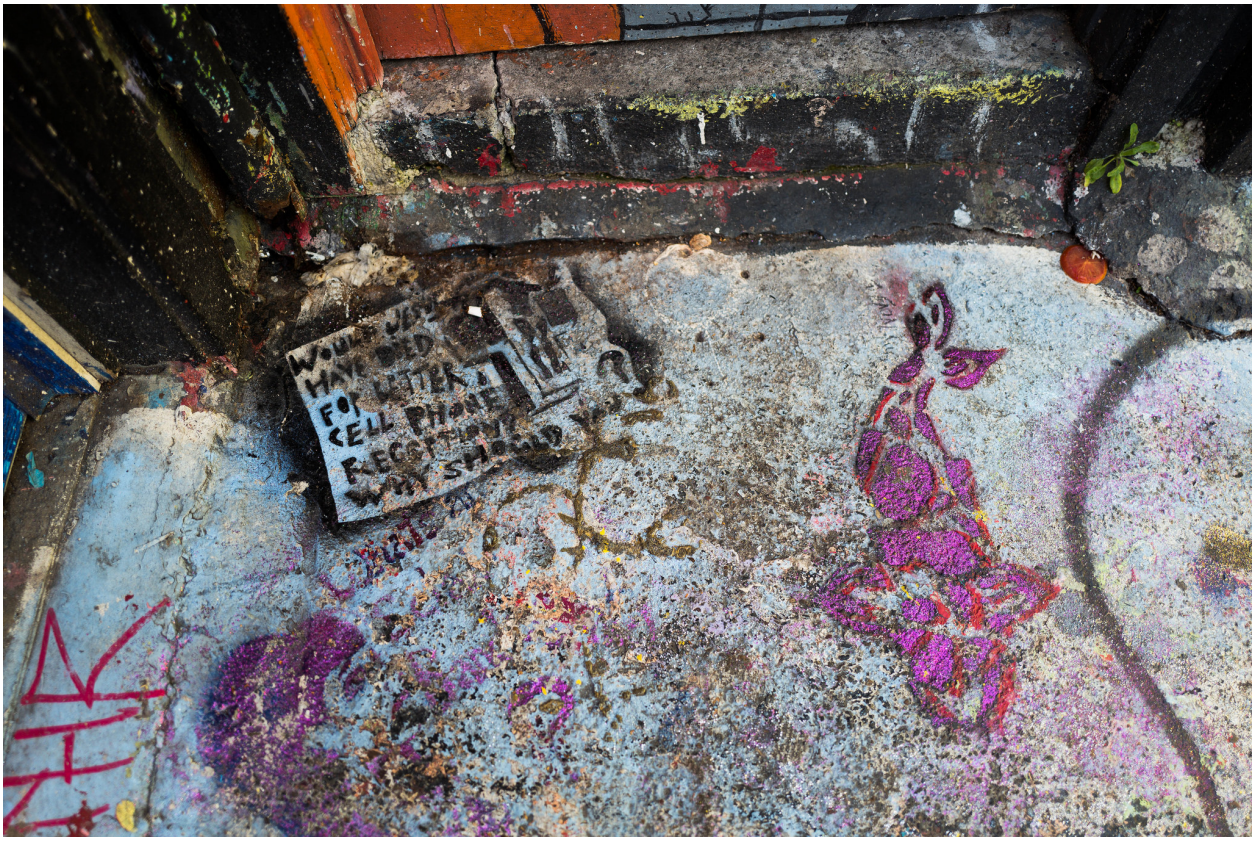
For more information about other options available when specifying VCS dependencies, please check the Pipfile spec.

### 4.1.12 Pipfile.lock Security Features

`Pipfile.lock` takes advantage of some great new security improvements in `pip`. By default, the `Pipfile.lock` will be generated with the sha256 hashes of each downloaded package. This will allow `pip` to guarantee you're installing what you intend to when on a compromised network, or downloading dependencies from an untrusted PyPI endpoint.

We highly recommend approaching deployments with promoting projects from a development environment into production. You can use `pipenv lock` to compile your dependencies on your development environment and deploy the compiled `Pipfile.lock` to all of your production environments for reproducible builds.

## 4.2 Advanced Usage of Pipenv




This document covers some of Pipenv's more glorious and advanced features.

### 4.2.1 Caveats

- Dependencies of wheels provided in a `Pipfile` will not be captured by `$ pipenv lock`.
- There are some known issues with using private indexes, related to hashing. We're actively working to solve this problem. You may have great luck with this, however.

- Installation is intended to be as deterministic as possible — use the `--sequential` flag to increase this, if experiencing issues.

### 4.2.2 Specifying Package Indexes

If you'd like a specific package to be installed with a specific package index, you can do the following:

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

[[source]]
url = "http://pypi.home.kennethreitz.org/simple"
verify_ssl = false
name = "home"

[dev-packages]

[packages]
requests = {version="*", index="home"}
maya = {version="*", index="pypi"}
records = "*"
```

Very fancy.

### 4.2.3 Injecting credentials into Pipfiles via environment variables

Pipenv will expand environment variables (if defined) in your Pipfile. Quite useful if you need to authenticate to a private PyPI:

```
[[source]]
url = "https://$USERNAME:${PASSWORD}@mypypi.example.com/simple"
verify_ssl = true
name = "pypi"
```

Luckily - pipenv will hash your Pipfile *before* expanding environment variables (and, helpfully, will substitute the environment variables again when you install from the lock file - so no need to commit any secrets! Woo!)

### 4.2.4 Specifying Basically Anything

If you'd like to specify that a specific package only be installed on certain systems, you can use PEP 508 specifiers to accomplish this.

Here's an example `Pipfile`, which will only install `pywinusb` on Windows systems:

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

[packages]
requests = "*"
pywinusb = {version = "*", sys_platform = "== 'win32'"}
```

Voilà!

Here's a more complex example:

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

[packages]
unittest2 = {version = ">=1.0,<3.0", markers="python_version < '2.7.9' or (python_
↪version >= '3.0' and python_version < '3.4')"}
```

Magic. Pure, unadulterated magic.

### 4.2.5 Deploying System Dependencies

You can tell Pipenv to install a Pipfile's contents into its parent system with the `--system` flag:

```
$ pipenv install --system
```

This is useful for Docker containers, and deployment infrastructure (e.g. Heroku does this).

Also useful for deployment is the `--deploy` flag:

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

This will fail a build if the `Pipfile.lock` is out–of–date, instead of generating a new one.

### 4.2.6 Pipenv and Other Python Distributions

To use Pipenv with a third-party Python distribution(e.g. Anaconda), you simply provide the path to the Python binary:

```
$ pipenv install --python=/path/to/python
```

Anaconda uses Conda to manage packages. To reuse Conda–installed Python packages, use the `--site-packages` flag:

```
$ pipenv --python=/path/to/python --site-packages
```

### 4.2.7 Generating a `requirements.txt`

You can convert a `Pipfile` and `Pipfile.lock` into a `requirements.txt` file very easily, and get all the benefits of extras and other goodies we have included.

Let's take this `Pipfile`:

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

[packages]
requests = {version="*"}
```

And generate a `requirements.txt` out of it:

```
$ pipenv lock -r
chardet==3.0.4
requests==2.18.4
certifi==2017.7.27.1
idna==2.6
urllib3==1.22
```

If you wish to generate a `requirements.txt` with only the development requirements you can do that too! Let's take the following `Pipfile`:

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

[dev-packages]
pytest = {version="*"}
```

And generate a `requirements.txt` out of it:

```
$ pipenv lock -r --dev
py==1.4.34
pytest==3.2.3
```

Very fancy.

### 4.2.8 Detection of Security Vulnerabilities

Pipenv includes the safety package, and will use it to scan your dependency graph for known security vulnerabilities!

Example:

```
$ cat Pipfile
[packages]
django = "==1.10.1"

$ pipenv check
Checking PEP 508 requirements...
Passed!
Checking installed package safety...

33075: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django before 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3,␣
↪when settings.DEBUG is True, allow remote attackers to conduct DNS rebinding␣
↪attacks by leveraging failure to validate the HTTP Host header against settings.
↪ALLOWED_HOSTS.

33076: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3 use a␣
↪hardcoded password for a temporary database user created when running tests with an␣
↪Oracle database, which makes it easier for remote attackers to obtain access to the␣
↪database server by leveraging failure to manually specify a password in the␣
↪database settings TEST dictionary.

33300: django >=1.10,<1.10.7 resolved (1.10.1 installed)!
CVE-2017-7233: Open redirect and possible XSS attack via user-supplied numeric␣
↪redirect URLs
==========================================================================================
```

```
Django relies on user input in some cases  (e.g.
:func:`django.contrib.auth.views.login` and :doc:`i18n </topics/i18n/index>`)
to redirect the user to an "on success" URL. The security check for these
redirects (namely ``django.utils.http.is_safe_url()``) considered some numeric
URLs (e.g. ``http:999999999``) "safe" when they shouldn't be.

Also, if a developer relies on ``is_safe_url()`` to provide safe redirect
targets and puts such a URL into a link, they could suffer from an XSS attack.

CVE-2017-7234: Open redirect vulnerability in ``django.views.static.serve()``
================================================================================

A maliciously crafted URL to a Django site using the
:func:`~django.views.static.serve` view could redirect to any other domain. The
view no longer does any redirects as they don't provide any known, useful
functionality.

Note, however, that this view has always carried a warning that it is not
hardened for production use and should be used only as a development aid.
```

**Note:** In order to enable this functionality while maintaining its permissive copyright license, *pipenv* embeds an API client key for the backend Safety API operated by pyup.io rather than including a full copy of the CC-BY-NC-SA licensed Safety-DB database. This embedded client key is shared across all *pipenv check* users, and hence will be subject to API access throttling based on overall usage rather than individual client usage.

### 4.2.9 Community Integrations

There are a range of community-maintained plugins and extensions available for a range of editors and IDEs, as well as different products which integrate with Pipenv projects:

- Heroku (Cloud Hosting)
- Platform.sh (Cloud Hosting)
- PyUp (Security Notification)
- Emacs (Editor Integration)
- Fish Shell (Automatic `$ pipenv shell`!)
- VS Code (Editor Integration)

Works in progress:

- Sublime Text (Editor Integration)
- PyCharm (Editor Integration)
- Mysterious upcoming Google Cloud product (Cloud Hosting)

### 4.2.10 Open a Module in Your Editor

Pipenv allows you to open any Python module that is installed (including ones in your codebase), with the `$ pipenv open` command:

```
$ pipenv install -e git+https://github.com/kennethreitz/background.git#egg=background
Installing -e git+https://github.com/kennethreitz/background.git#egg=background...
...
Updated Pipfile.lock!

$ pipenv open background
Opening '/Users/kennethreitz/.local/share/virtualenvs/hmm-mGOawwm_/src/background/
↪background.py' in your EDITOR.
```

This allows you to easily read the code you're consuming, instead of looking it up on GitHub.

**Note:** The standard `EDITOR` environment variable is used for this. If you're using VS Code, for example, you'll want to `export EDITOR=code` (if you're on macOS you will want to install the command on to your `PATH` first).

### 4.2.11 Automatic Python Installation

If you have pyenv installed and configured, Pipenv will automatically ask you if you want to install a required version of Python if you don't already have it available.

This is a very fancy feature, and we're very proud of it:

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

[dev-packages]

[packages]
requests = "*"

[requires]
python_version = "3.6"

$ pipenv install
Warning: Python 3.6 was not found on your system...
Would you like us to install latest CPython 3.6 with pyenv? [Y/n]: y
Installing CPython 3.6.2 with pyenv (this may take a few minutes)...
...
Making Python installation global...
Creating a virtualenv for this project...
Using /Users/kennethreitz/.pyenv/shims/python3 to create virtualenv...
...
No package provided, installing all dependencies.
...
Installing dependencies from Pipfile.lock...
    5/5 -- 00:00:03
To activate this project's virtualenv, run the following:
 $ pipenv shell
```

Pipenv automatically honors both the `python_full_version` and `python_version` PEP 508 specifiers.

### 4.2.12 Automatic Loading of `.env`

If a `.env` file is present in your project, `$ pipenv shell` and `$ pipenv run` will automatically load it, for you:

```
$ cat .env
HELLO=WORLD

$ pipenv run python
Loading .env environment variables...
Python 2.7.13 (default, Jul 18 2017, 09:17:00)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.environ['HELLO']
'WORLD'
```

This is very useful for keeping production credentials out of your codebase. We do not recommend committing `.env` files into source control!

If your `.env` file is located in a different path or has a different name you may set the `PIPENV_DOTENV_LOCATION` environment variable:

```
$ PIPENV_DOTENV_LOCATION=/path/to/.env pipenv shell
```

To prevent pipenv from loading the `.env` file, set the `PIPENV_DONT_LOAD_ENV` environment variable:

```
$ PIPENV_DONT_LOAD_ENV=1 pipenv shell
```

### 4.2.13 Custom Script Shortcuts

Pipenv supports to customize shortcuts in the `scripts` section. `pipenv run` will automatically load it and find the correct command to replace with. Given the `Pipfile`:

```
[scripts]
printfoo = "python -c \"print('foo')\""
```

You can type in your terminal to run:

```
$ pipenv run printfoo
foo
```

### 4.2.14 Support for Environment Variables

Pipenv supports the usage of environment variables in values. For example:

```
[[source]]
url = "https://${PYPI_USERNAME}:${PYPI_PASSWORD}@my_private_repo.example.com/simple"
verify_ssl = true
name = "pypi"

[dev-packages]

[packages]
requests = {version="*", index="home"}
```

```
maya = {version="*", index="pypi"}
records = "*"
```

Environment variables may be specified as `${MY_ENVAR}` or `$MY_ENVAR`. On Windows, `%MY_ENVAR%` is supported in addition to `${MY_ENVAR}` or `$MY_ENVAR`.

### 4.2.15 Configuration With Environment Variables

Pipenv comes with a handful of options that can be enabled via shell environment variables. To activate them, simply create the variable in your shell and pipenv will detect it.

- `PIPENV_DEFAULT_PYTHON_VERSION` — Use this version of Python when creating new virtual environments, by default (e.g. `3.6`).
- `PIPENV_SHELL_FANCY` — Always use fancy mode when invoking `pipenv shell`.
- `PIPENV_VENV_IN_PROJECT` — If set, use `.venv` in your project directory instead of the global virtualenv manager `pew`.
- `PIPENV_COLORBLIND` — Disable terminal colors, for some reason.
- `PIPENV_NOSPIN` — Disable terminal spinner, for cleaner logs. Automatically set in CI environments.
- `PIPENV_MAX_DEPTH` — Set to an integer for the maximum number of directories to recursively search for a Pipfile.
- `PIPENV_TIMEOUT` — Set to an integer for the max number of seconds Pipenv will wait for virtualenv creation to complete. Defaults to 120 seconds.
- `PIPENV_INSTALL_TIMEOUT` — Set to an integer for the max number of seconds Pipenv will wait for package installation before timing out. Defaults to 900 seconds.
- `PIPENV_IGNORE_VIRTUALENVS` — Set to disable automatically using an activated virtualenv over the current project's own virtual environment.
- `PIPENV_PIPFILE` — When running pipenv from a $PWD other than the same directory where the Pipfile is located, instruct pipenv to find the Pipfile in the location specified by this environment variable.
- `PIPENV_CACHE_DIR` — Location for Pipenv to store it's package cache.
- `PIPENV_HIDE_EMOJIS` — Disable emojis in output.
- `PIPENV_DOTENV_LOCATION` — Location for Pipenv to load your project's .env.
- `PIPENV_DONT_LOAD_ENV` — Tell Pipenv not to load the .env files automatically.

If you'd like to set these environment variables on a per-project basis, I recommend utilizing the fantastic direnv project, in order to do so.

Also note that pip itself supports environment variables, if you need additional customization.

For example:

```
$ PIP_INSTALL_OPTION="-- -DCMAKE_BUILD_TYPE=Release" pipenv install -e .
```

### 4.2.16 Custom Virtual Environment Location

Pipenv's underlying `pew` dependency will automatically honor the `WORKON_HOME` environment variable, if you have it set — so you can tell pipenv to store your virtual environments wherever you want, e.g.:

```
export WORKON_HOME=~/.venvs
```

In addition, you can also have Pipenv stick the virtualenv in `project/.venv` by setting the `PIPENV_VENV_IN_PROJECT` environment variable.

### 4.2.17 Testing Projects

Pipenv is being used in projects like Requests for declaring development dependencies and running the test suite.

We've currently tested deployments with both Travis-CI and tox with success.

#### Travis CI

An example Travis CI setup can be found in Requests. The project uses a Makefile to define common functions such as its `init` and `tests` commands. Here is a stripped down example `.travis.yml`:

```
language: python
python:
    - "2.6"
    - "2.7"
    - "3.3"
    - "3.4"
    - "3.5"
    - "3.6"
    - "3.7-dev"

# command to install dependencies
install: "make"

# command to run tests
script:
    - make test
```

and the corresponding Makefile:

```
init:
    pip install pipenv
    pipenv install --dev

test:
    pipenv run py.test tests
```

#### Tox Automation Project

Alternatively, you can configure a `tox.ini` like the one below for both local and external testing:

```
[tox]
envlist = flake8-py3, py26, py27, py33, py34, py35, py36, pypy

[testenv]
deps = pipenv
commands=
    pipenv install --dev
    pipenv run py.test tests
```

```
[testenv:flake8-py3]
basepython = python3.4
commands=
    pipenv install --dev
    pipenv run flake8 --version
    pipenv run flake8 setup.py docs project test
```

Pipenv will automatically use the virtualenv provided by `tox`. If `pipenv install --dev` installs e.g. `pytest`, then installed command `py.test` will be present in given virtualenv and can be called directly by `py.test tests` instead of `pipenv run py.test tests`.

You might also want to add `--ignore-pipfile` to `pipenv install`, as to not accidentally modify the lock-file on each test run. This causes Pipenv to ignore changes to the `Pipfile` and (more importantly) prevents it from adding the current environment to `Pipfile.lock`. This might be important as the current environment (i.e. the virtualenv provisioned by tox) will usually contain the current project (which may or may not be desired) and additional dependencies from `tox`'s `deps` directive. The initial provisioning may alternatively be disabled by adding `skip_install = True` to tox.ini.

This method requires you to be explicit about updating the lock-file, which is probably a good idea in any case.

A 3rd party plugin, tox-pipenv is also available to use Pipenv natively with tox.

### 4.2.18 Shell Completion

To enable completion in fish, add this to your config:

```
eval (pipenv --completion)
```

Alternatively, with bash or zsh, add this to your config:

```
eval "$(pipenv --completion)"
```

Magic shell completions are now enabled!

### 4.2.19 Working with Platform-Provided Python Components

It's reasonably common for platform specific Python bindings for operating system interfaces to only be available through the system package manager, and hence unavailable for installation into virtual environments with *pip*. In these cases, the virtual environment can be created with access to the system *site-packages* directory:

```
$ pipenv --three --site-packages
```

To ensure that all *pip*-installable components actually are installed into the virtual environment and system packages are only used for interfaces that don't participate in Python-level dependency resolution at all, use the *PIP_IGNORE_INSTALLED* setting:

```
$ PIP_IGNORE_INSTALLED=1 pipenv install --dev
```

### 4.2.20 Pipfile vs setup.py

There is a subtle but very important distinction to be made between **applications** and **libraries**. This is a very common source of confusion in the Python community.

Libraries provide reusable functionality to other libraries and applications (let's use the umbrella term **projects** here). They are required to work alongside other libraries, all with their own set of subdependencies. They define **abstract dependencies**. To avoid version conflicts in subdependencies of different libraries within a project, libraries should never ever pin dependency versions. Although they may specify lower or (less frequently) upper bounds, if they rely on some specific feature/fix/bug. Library dependencies are specified via `install_requires` in `setup.py`.

Libraries are ultimately meant to be used in some **application**. Applications are different in that they usually are not depended on by other projects. They are meant to be deployed into some specific environment and only then should the exact versions of all their dependencies and subdependencies be made concrete. To make this process easier is currently the main goal of Pipenv.

To summarize:

- For libraries, define **abstract dependencies** via `install_requires` in `setup.py`. The decision of which version exactly to be installed and where to obtain that dependency is not yours to make!
- For applications, define **dependencies and where to get them** in the *Pipfile* and use this file to update the set of **concrete dependencies** in `Pipfile.lock`. This file defines a specific idempotent environment that is known to work for your project. The `Pipfile.lock` is your source of truth. The `Pipfile` is a convenience for you to create that lock-file, in that it allows you to still remain somewhat vague about the exact version of a dependency to be used. Pipenv is there to help you define a working conflict-free set of specific dependency-versions, which would otherwise be a very tedious task.
- Of course, `Pipfile` and Pipenv are still useful for library developers, as they can be used to define a development or test environment.
- And, of course, there are projects for which the distinction between library and application isn't that clear. In that case, use `install_requires` alongside Pipenv and `Pipfile`.

You can also do this:

```
$ pipenv install -e .
```

This will tell Pipenv to lock all your `setup.py`–declared dependencies.

### 4.2.21 Changing Pipenv's Cache Location

You can force Pipenv to use a different cache location by setting the environment variable `PIPENV_CACHE_DIR` to the location you wish. This is useful in the same situations that you would change `PIP_CACHE_DIR` to a different directory.

### 4.2.22 Changing Where Pipenv Stores Virtualenvs

By default, Pipenv stores all of your virtualenvs in a single place. Usually this isn't a problem, but if you'd like to change it for developer ergonomics, or if it's causing issues on build servers you can set `PIPENV_VENV_IN_PROJECT` to create the virtualenv inside the root of your project.

### 4.2.23 Changing Default Python Versions

By default, Pipenv will initialize a project using whatever version of python the python3 is. Besides starting a project with the `--three` or `--two` flags, you can also use `PIPENV_DEFAULT_PYTHON_VERSION` to specify what version to use when starting a project when `--three` or `--two` aren't used.

## 4.3 Frequently Encountered Pipenv Problems

Pipenv is constantly being improved by volunteers, but is still a very young project with limited resources, and has some quirks that needs to be dealt with. We need everyone's help (including yours!).

Here are some common questions people have using Pipenv. Please take a look below and see if they resolve your problem.

**Note: Make sure you're running the newest Pipenv version first!**

### 4.3.1 Your dependencies could not be resolved

Make sure your dependencies actually *do* resolve. If you're confident they are, you may need to clear your resolver cache. Run the following command:

```
pipenv run pipenv-resolver --clear
```

and try again.

If this does not work, try manually deleting the whole cache directory. It is usually one of the following locations:

- `~/Library/Caches/pipenv` (macOS)
- `%LOCALAPPDATA%\pipenv\pipenv\Cache` (Windows)
- `~/.cache/pipenv` (other operating systems)

Pipenv does not install prereleases (i.e. a version with an alpha/beta/etc. suffix, such as *1.0b1*) by default. You will need to pass the `--pre` flag in your command, or set

```
[pipenv]
allow_prereleases = true
```

in your Pipfile.

### 4.3.2 No module named <module name>

This is usually a result of mixing Pipenv with system packages. We *strongly* recommend installing Pipenv in an isolated environment. Uninstall all existing Pipenv installations, and see *Homebrew Installation of Pipenv* to choose one of the recommended way to install Pipenv instead.

### 4.3.3 My pyenv-installed Python is not found

Make sure you have `PYENV_ROOT` set correctly. Pipenv only supports CPython distributions, with version name like `3.6.4` or similar.

### 4.3.4 Pipenv does not respect pyenv's global and local Python versions

Pipenv by default uses the Python it is installed against to create the virtualenv. You can set the `--python` option, or `$PYENV_ROOT/shims/python` to let it consult pyenv when choosing the interpreter. See *Specifying Versions of a Package* for more information.

If you want Pipenv to automatically "do the right thing", you can set the environment variable `PIPENV_PYTHON` to `$PYENV_ROOT/shims/python`. This will make Pipenv use pyenv's active Python version to create virtual environments by default.

### 4.3.5 ValueError: unknown locale: UTF-8

macOS has a bug in its locale detection that prevents us from detecting your shell encoding correctly. This can also be an issue on other systems if the locale variables do not specify an encoding.

The workaround is to set the following two environment variables to a standard localization format:

- `LC_ALL`
- `LANG`

For Bash, for example, you can add the following to your `~/.bash_profile`:

```
export LC_ALL='en_US.UTF-8'
export LANG='en_US.UTF-8'
```

For Zsh, the file to edit is `~/.zshrc`.

**Note:** You can change both the `en_US` and `UTF-8` part to the language/locale and encoding you use.

### 4.3.6 /bin/pip: No such file or directory

This may be related to your locale setting. See *ValueError: unknown locale: UTF-8* for a possible solution.

### 4.3.7 `shell` does not show the virtualenv's name in prompt

This is intentional. You can do it yourself with either shell plugins, or clever `PS1` configuration. If you really want it back, use

```
pipenv shell -c
```

instead (not available on Windows).

### 4.3.8 Pipenv does not respect dependencies in setup.py

No, it does not, intentionally. Pipfile and setup.py serve different purposes, and should not consider each other by default. See *Pipfile vs setup.py* for more information.

CHAPTER 5

# Pipenv Usage

## 5.1 pipenv

```
pipenv [OPTIONS] COMMAND [ARGS]...
```

### Options

**--where**
Output project home information.

**--venv**
Output virtualenv information.

**--py**
Output Python interpreter information.

**--envs**
Output Environment Variable options.

**--rm**
Remove the virtualenv.

**--bare**
Minimal output.

**--completion**
Output completion (to be eval'd).

**--man**
Display manpage.

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--site-packages**
Enable site-packages for the virtualenv.

**--version**
Show the version and exit.

### 5.1.1 check

```
pipenv check [OPTIONS] [ARGS]...
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--system**
Use system Python.

**--unused** <unused>
Given a code path, show potentially unused dependencies.

#### Arguments

**ARGS**
Optional argument(s)

### 5.1.2 clean

```
pipenv clean [OPTIONS]
```

#### Options

**-v, --verbose**
Verbose mode.

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--dry-run**
Just output unneeded packages.

### 5.1.3 graph

```
pipenv graph [OPTIONS]
```

#### Options

**--bare**
Minimal output.

**--json**
Output JSON.

**--reverse**
Reversed dependency graph.

### 5.1.4 install

```
pipenv install [OPTIONS] [PACKAGE_NAME] [MORE_PACKAGES]...
```

#### Options

**-d, --dev**
Install package(s) in [dev-packages].

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--system**
System pip management.

**-r, --requirements** <requirements>
Import a requirements.txt file.

**-c, --code** <code>
Import from codebase.

**-v, --verbose**
Verbose mode.

**--ignore-pipfile**
Ignore Pipfile when installing, using the Pipfile.lock.

**--sequential**
Install dependencies one-at-a-time, instead of concurrently.

**--skip-lock**
Ignore locking mechanisms when installing—use the Pipfile, instead.

**--deploy**
Abort if the Pipfile.lock is out–of–date, or Python version is wrong.

**--pre**
Allow pre–releases.

**--keep-outdated**
Keep out–dated dependencies from being updated in Pipfile.lock.

**--selective-upgrade**
Update specified packages.

#### Arguments

**PACKAGE_NAME**
Optional argument

**MORE_PACKAGES**
Optional argument(s)

### 5.1.5 lock

```
pipenv lock [OPTIONS]
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**-v, --verbose**
Verbose mode.

**-r, --requirements**
Generate output compatible with requirements.txt.

**-d, --dev**
Generate output compatible with requirements.txt for the development dependencies.

**--clear**
Clear the dependency cache.

**--pre**
Allow pre–releases.

**--keep-outdated**
Keep out–dated dependencies from being updated in Pipfile.lock.

### 5.1.6 open

```
pipenv open [OPTIONS] MODULE
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

#### Arguments

**MODULE**
Required argument

### 5.1.7 run

```
pipenv run [OPTIONS] COMMAND [ARGS]...
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

#### Arguments

**COMMAND**
Required argument

**ARGS**
Optional argument(s)

### 5.1.8 shell

```
pipenv shell [OPTIONS] [SHELL_ARGS]...
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--fancy**
Run in shell in fancy mode (for elegantly configured shells).

**--anyway**
Always spawn a subshell, even if one is already spawned.

#### Arguments

**SHELL_ARGS**
Optional argument(s)

### 5.1.9 sync

```
pipenv sync [OPTIONS]
```

#### Options

**-v, --verbose**
Verbose mode.

**-d, --dev**
Additionally install package(s) in [dev-packages].

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--bare**
Minimal output.

**--clear**
Clear the dependency cache.

**--sequential**
Install dependencies one-at-a-time, instead of concurrently.

### 5.1.10 uninstall

```
pipenv uninstall [OPTIONS] [PACKAGE_NAME] [MORE_PACKAGES]...
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**--system**
System pip management.

**-v, --verbose**
Verbose mode.

**--lock**
Lock afterwards.

**--all-dev**
Un-install all package from [dev-packages].

**--all**
Purge all package(s) from virtualenv. Does not edit Pipfile.

**--keep-outdated**
Keep out–dated dependencies from being updated in Pipfile.lock.

#### Arguments

**PACKAGE_NAME**
Optional argument

**MORE_PACKAGES**
Optional argument(s)

### 5.1.11 update

```
pipenv update [OPTIONS] [MORE_PACKAGES]... [PACKAGE]
```

#### Options

**--three, --two**
Use Python 3/2 when creating virtualenv.

**--python** <python>
Specify which version of Python virtualenv should use.

**-v, --verbose**
Verbose mode.

**-d, --dev**
Install package(s) in [dev-packages].

**--clear**
Clear the dependency cache.

**--bare**
Minimal output.

**--pre**
Allow pre–releases.

**--keep-outdated**
Keep out–dated dependencies from being updated in Pipfile.lock.

**--sequential**
Install dependencies one-at-a-time, instead of concurrently.

**--outdated**
List out–of–date dependencies.

**--dry-run**
List out–of–date dependencies.

### Arguments

**MORE_PACKAGES**
Optional argument(s)

**PACKAGE**
Optional argument

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Index

## Symbols

## S