Dependency Management

Dependency Management

Python's tooling for dependency management is evolving.

Current generation: `requirements.txt`

- Uses `pip` and `venv` in the standard library
- Widely used and mature support

New process: `Pipfile`

- Solves some important deficiencies
- High quality reference implementation, called `pipenv`, is available

We'll go over both.

Virtual Environments

Python lets you create a *virtual environment*. (A.k.a. "virtualenv" or "venv".)

Think of it as a lightweight container for a Python app:

- Dependencies precisely tracked and specified
- And kept in version control, for easy reproducability
- Two different Python applications with conflicting dependencies can peacefully coexist on the same machine.
- Python packages can be installed without requiring elevated system privileges.

Creating a VENV

For Python 3:

```
# The recommended method in Python 3.
$ python3 -m venv webappenv

# Does the same thing, but deprecated.
$ pyvenv webappenv
```

```
# Same as the above, in Python 2.
$ virtualenv webappenv
```

All create folder named `webappenv`.

Activating The VENV

The new folder contains lots of goodies.

To access them, activate the virtual environment:

```
# In macOS and Linux/Unix
$ source webappenv/bin/activate
(webappenv)$
```

On Windows:

```
C:\labs> webappenv\Scripts\activate.bat
(webappenv) C:\labs>
```

Notice your prompt has changed!

What's in the VENV?

With the virtual environment activated, you have your own local copy of the Python interpreter:

```
(webappenv)$ which python
/Users/sam/mywebapp/webappenv/bin/python
(webappenv)$ python -V
Python 3.6.0
```

And other tools, like `pip`:

```
(webappenv)$ which pip
/Users/sam/mywebapp/webappenv/bin/pip
```

Deactivate with `deactivate`:

```
(webappenv)$ deactivate
$ which python
/usr/bin/python
```

Multiple Environments

Each Python application can have its own virtual environment. Each can have a completely different version of Python.

```
$ cd /Users/sam/mediatag
$ virtualenv mediatagenv
$ source mediatagenv/bin/activate
(mediatagenv)$ which python
/Users/sam/mywebapp/mediatagenv/bin/python
(mediatagenv)$ python -V
Python 2.7.13
```

Using Pip

Modern Python provides an application called `pip`.

It lets you easily install 3rd-party Python libraries.

Your virtual environment has it:

```
$ source venv/bin/activate
(venv)$ python -V
Python 3.6.0
(venv)$ which pip
/Users/sam/myapp/venv/bin/pip
```

pip install

Use `pip install` to install opensource libraries:

```
pip install requests
```

These are fetched from PyPI: https://pypi.org

Upgrade an installed package, with `-U` or `--upgrade`:

```
pip install --upgrade requests
```

Or uninstall:

```
pip uninstall requests
```

Reproducible Builds

Don’t put the venv itself in version control. Some of those files are HUGE, and very platform-dependent.

Other problems:

- How to track exact version dependencies?
- How to specify the precise set of libraries, so that other devs can build it on their machines?
- How to manage upgrades (or even downgrades) over time?

`pip` provides a set of tools for this: `requirements.txt`.

Freeze

Start by using `pip freeze`:

```
(venv)$ pip freeze
requests==2.7.0
```

Prints what was installed from Pypi, one per line.
Place this in a file named `requirements.txt`:

```
(venv)$ pip freeze > requirements.txt
(venv)$ cat requirements.txt
requests==2.7.0
```

`requirements.txt` is what you actually check into version control!

Rebuild

You can pass these requirements to pip, using `-r`:

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv)$ pip install -r requirements.txt
Collecting requests==2.7.0 (from -r requirements.txt (line 1))
```

This allows the environment to be recreated on any server or machine.

Not a perfect solution, but often works very well.

Evolving Requirements

As dependencies evolve, update `requirements.txt` as you go along:

```
(venv)$ pip install django
Installing collected packages: pytz, django
Successfully installed django-1.11.7 pytz-2017.3

(venv)$ pip freeze > requirements.txt
(venv)$ git diff requirements.txt
diff --git a/requirements.txt b/requirements.txt
index ac2cf62..354542c 100644
--- a/requirements.txt
+++ b/requirements.txt
@@ -1 +1,3 @@
+Django==1.11.7
+pytz==2017.3
 requests==2.7.0

(venv)$ git add requirements.txt; git commit -m 'Install Django'
[master 8a3ec09] Install Django
 1 file changed, 2 insertions(+)
```

Naming the VENV

Two schools of thought. Pros and cons for each:

1) Pick a consistent name. "venv" is popular:

```
python3 -m venv venv
```

Upside: Consistency. Easy to make git ignore it.

2) Or pick a name that has to do with the application.

```
python3 -m venv mywebappenv
```

Upside: More descriptive prompt.

Downside: Inconsistent.

`requirements.txt`

Using `requirements.txt` brings some problems, especially for larger applications.

- It doesn't address different requirements for production; updating dependencies; and development (builds).
- Removing with `pip` does not remove dependencies; you have to do that manually. Error prone.
- You can do things with `pip` to handle all this… but it's manual, labor-intenstive, and easy to make mistakes.
- Pip has some extra security features, but the standard set-up doesn't take advantage of them.

These are solved by a new requirements file format called `Pipfile`.

The reference implementation for working with it is a tool called `pipenv`.

Pipfile

The `Pipfile` format greatly improves on `requirements.txt`.

- Separate sections for dependencies needed in production, vs. those needed during development.
- Also allows you to declare the install source (defaulting to PyPI), intended Python version, and more.
- Paired with a separate `Pipfile.lock` file, generated from `Pipfile`.
- `Pipfile.lock` specifies exact version numbers. Repeatable builds
- Incorporates hashes, for improved security
- Includes other build parameters

Both `Pipfile` and `Pipfile.lock` are put in version control. Normally you modify only `Pipfile`, then regenerate `Pipfile.lock`.

So how do you do that?

pipenv

Pipenv is a new tool that makes it easy to work with Pipfiles.

No venv setup needed. Just say:

```
pipenv install django
```

This will:

- Create a virtual environment
- Record the top-level `django` requirement in `Pipfile`
- Generate `Pipfile.lock`, with the exact version numbers of `django`, AND its dependency `pytz`
- And clean up intelligently if you remove `django` in the future.

Activate with `pipenv shell`. (Similar to `activate`.)

Great explanation of the benefits: https://goo.gl/9V2dmZ

Pipenv, or pip + venv?

`pipenv` is recommended by the Python packaging WG (PyPA).

`pip` and `venv` are part of the standard library; `pipenv` is not, and may never be. But `Pipfile` support is very likely to ship with Python in the future (maybe through upgrading `pip`).

My recommendation:

- Learn `pip` and `venv`. They're widely used, and even `pipenv` builds on them.
- Consider learning and using `pipenv` right now.
- You can start new projects with `pip` and a `requirements.txt`. Or if you are a fan of `pipenv`, start with that.
- Once a project's needs expand to multiple `requirements.txt` files, change to using `pipenv` for that project.
- In the future, `pip` may include `Pipfile` support; if and when it does, start new projects with that.