An overview of package management in Python in 2018

2018-09-07

First of all, this article will only talk about package management in the context of backend applications. I do not maintain any Python library so I am not able to give insights on that side of the package management story.

I will begin by quickly explaining the most common way of setting up Python projects and handling dependencies to make sure everyone is on the same page.

A basic setup

Pretty much all the Django/Flask projects I have seen use the same structure: a `requirements` folder containing several `.txt` files listing the dependencies and their versions.

The folder typically looks like the following:

```
.
├── base.txt
├── local.txt
└── prod.txt
```

`base.txt` contains all the essential dependencies:

```
boto3==1.6.10

Django==2.0.3
django-cors-headers==2.2.0
django-filter==1.1.0
django-floppyforms==1.7.0
django-nested-inline==0.3.7
django-storages==1.6.5
djangorestframework==3.7.7
djangorestframework-jsonp==1.0.2
social-auth-app-django==2.1.0
social-auth-core==1.7.0
django-widget-tweaks==1.4.1
```

While it is not required, every requirement files I have seen were using `==`, also called requirement pinning, to depend on the exact versions to ensure reproducibility. Well, as much reproducibility as a mutable package index allows anyway as we will see later.

Other requirements files will import the `base.txt` file and then add environment specific dependencies such as testing libraries or developer tools for `local.txt` for example.

```
-r base.txt

ipython==6.2.1
flake8==3.5.0
django-extensions==2.0.6
mypy==0.570
factory_boy==2.10.0
```

But wait! Before being able to install the dependencies, you first need to create a virtual environment if you are not using Docker. Python 3 has built-in support for virtualenvs and you can create one simply by running `python3 -m` `venv tutorial-env`. To activate it, you need to execute the `activate` script in the `bin` folder of the virtualenv.

As this feature is somewhat recent, many virtualenv helpers exist to make it easier to use. I use virtualenvwrapper combined with the `zsh` plugin with the same name to automatically activate the right virtualenv when I'm `cd`ing into a folder that has one.

Now that your virtualenv is setup and activated, you can run `pip install -r requirements/local.txt` to install your dependencies.

You can list the currently installed packages by running `pip freeze` in the virtualenv. The downside is that it will also show all indirect dependencies as well. You can order that list with the `-r my_file.txt` argument to have your explicit dependencies at the top but it isn't great.

This process works but is quite manual:

- beginners might not know of virtualenvs and how to use them
- packages and their versions are edited manually so it is easy to make mistakes by forgetting to add/remove one
- not really secure: the package index Pypi is mutable so 2.1.0 one day can be different from 2.1.0 another day and there will no way to know about it. While you can add hashes to requirements files but I have never seen anyone go through the hassle of doing that themselves

We can see that when dealing with package management in Python there are two issues: dealing with virtualenvs and actually managing the packages.

There are plenty of tools that have been built to fix one or both issues; let's look at a few of them now. I will use the list of packages of an actual Django project with about 28 dependencies to test them. While speed is not THAT important for a package manager, I will only mention it if there is something to be said about it.

pip-tools

pip-tools only tries to handle the package management part, meaning you still need to use `virtualenvwrapper` or something similar to handle virtualenvs.

`pip-tools` gives you two command line tools: `pip-compile` and `pip-sync`.

The easiest way to get started with `pip-tools` is to create a `requirements.in` file listing your dependencies:

```
boto3
Django
django-cors-headers
django-filter
django-floppyforms
django-nested-inline
django-storages
djangorestframework
djangorestframework-jsonp
social-auth-app-django
social-auth-core
django-widget-tweaks
```

The `requirements.in` file uses the same requirements syntax as `requirements.txt`: I could indicate for example that I want `Django==2.0.8` instead of the latest version.

Running `pip-compile` in the same folder will create a `requirements.txt` file:

```
#
# This file is autogenerated by pip-compile
# To update, run:
#
#    pip-compile --output-file requirements.txt requirements.in
#
boto3==1.8.7
botocore==1.11.7          # via boto3, s3transfer
certifi==2018.8.24        # via requests
chardet==3.0.4            # via requests
```

```
defusedxml==0.5.0         # via python3-openid, social-auth-core
django-cors-headers==2.4.0
django-filter==2.0.0
django-floppyforms==1.7.0
django-nested-inline==0.3.7
django-storages==1.7
django-widget-tweaks==1.4.2
django==2.1.1

... more lines
```

`pip-compile` automatically grabbed the latest versions and each line not present in your `requirements.in` has a comment indicating why it is here. This is already a big improvement over `pip freeze > requirements.txt`.

Remember the packages hashes mentioned before? `pip-compile --generate-hashes` can automatically computes the hashes and put them in the generated file:

```
#
# This file is autogenerated by pip-compile
# To update, run:
#
#    pip-compile --generate-hashes --output-file requirements.txt requ
#
boto3==1.8.7 \
    --hash=sha256:7b54cc29dde1d4833b082b8ef4062872297cf652ed20b2d485e
    --hash=sha256:b181fb87661a4268174ba29cb5efbc9e728202f3f80e00356e8
botocore==1.11.7 \
    --hash=sha256:4d36cb3a0b6308eeb56f69b964b92b9f4609423b07a4979c298
    --hash=sha256:86d0bf23b5071c6a956ad2a1d5cd8d1d7c997e2c04151f7c8ff
    # via boto3, s3transfer
certifi==2018.8.24 \
    --hash=sha256:376690d6f16d32f9d1fe8932551d80b23e9d393a8578c5633a2
    --hash=sha256:456048c7e371c089d0a77a5212fb37a2c2dce1e24146e3b7e02
    # via requests
chardet==3.0.4 \
    --hash=sha256:84ab92ed1c4d4f16916e05906b6b75a6c0fb5db821cc65e70cb
    --hash=sha256:fc323ffcaeaed0e0a02bf4d117757b98aed530d9ed4531e3e15
    # via requests

... more lines
```

The initial run takes a bit of time but subsequent ones take under a second.

The last thing we need for package management is an easy way to upgrade the packages and `pip-compile` provides that out of the box:

- `pip-compile --upgrade` will update all packages
- `pip-compile -P PACKAGE_NAME` will update only that package

An astute reader might have noticed that so far we have only managed to write and update a text file and have not actually installed anything. That's where `pip-sync` comes into play.

Running `pip-sync requirements.txt` will compare the packages listed in the file with what is currently installed, installing the missing packages and uninstalling the packages not listed. This ensures your virtualenv doesn't have extra dependencies and is representing accurately your requirements file.

The main con I have is pretty silly: why is it `pip-compile` and `pip-sync` instead of `piptools compile` and `piptools sync`?

Pipenv

Pipenv is developed by the Python Packaging Authority, responsible for developing pip so I had big expectations.

It is inspired by Cargo/Bundler/Yarn and introduces 2 files that you will recognize if you used any of them before:

- a `Pipfile`: a TOML file listing packages as well as a minimal Python version
- a `Pipfile.lock`: a lockfile in JSON format listing every dependencies version with their hash

Anytime you want to install the dependencies, Pipenv will read `Pipfile.lock` and install everything in it. To make it easier for users, Pipenv also wraps virtualenv, getting rid of the need to have `virtualenvwrapper`. Pipenv seems to be only targeting applications, it doesn't help if you are packaging a library.

Let's see how it works in practice.

The first step is to create a new project: `pipenv --python 3.7`. This is a pretty strange command as `pipenv` only displays the help so I would have expected the command to be `pipenv new` rather than some flags. Typically flags in this situation are only there to display information, like `pipenv --version`. Pipenv will also create a new virtualenv when running `pipenv install` if there isn't one.

To activate the newly created virtualenv, run `pipenv shell`. To exit it, simply type `exit`.

Running `pipenv install django` will do two things:

- insert `django = "*"` in the `packages` table of the `Pipfile`
- download the latest django version, install it in the virtual env and add the hash/version in `Pipfile.lock`

Putting `"*"` as a version number is not great: I need to read the `Pipfile.lock` to see which version was installed and `*` is never a good default for a version number, even with a lockfile. Changing it to `django = "==2.1.1"` which was the version it installed did the trick.

You can run `pipenv sync` to do the equivalent of `pip-sync` from `pip-tools`, but using `Pipfile.lock` as the truth this time. Pipenv actually uses `pip-tools` for some of its commands under the hood.

A cool feature of Pipenv is `pipenv check` that will check for security issues in your packages. Running a slightly outdated version of Django for example gives the following:

```
Checking PEP 508 requirements...
Passed!
```

```
Checking installed package safety...
36368: django >=2.0.0, <2.0.8 resolved (2.0.5 installed)!
django.middleware.common.CommonMiddleware in Django 1.11.x before 1.1
```

> *[PEP 508](#) is the grammar defining how to write dependencies versions, i.e. the `==` for example you have seen before in that article.*

Pipenv seems extremely slow: running `pipenv install -d` the test project with already pinned versions took 166s. Almost 3 minutes. Running it again right after took 55s. Any action dealing with packages takes a minimum of 30s on my machine. I did say in the introduction that speed is not too important but Pipenv does test the limit of that affirmation.

Lastly, I found the UX of the whole tool to be pretty awkward or buggy:

- `pipenv` shows an help menu with the various actions but `pipenv check --help` for example will not tell you what `pipenv check` actually does, only what flags are available
- `pipenv uninstall` is said to "Un-installs a provided package and removes it from Pipfile." but somehow `pipenv uninstall --three` will destroy the current virtualenv and create a new one? It looks like all subcommands have the same flags available to create/recreate a virtualenv instead of having a `pipenv new` command and avoid this pitfall.
- errors are very cryptic: I made a typo while adding a dependency in `Pipfile` resulting in an invalid TOML variable and all I got was two tracebacks for a total of 70 lines with the reason on the last line. How about only showing me the reason instead?
- `pipenv update` doesn't tell what was actually updated...?
- `pipenv update django` updates all the packages instead of only django....?

poetry

I believe I first heard of poetry while looking up Black and noticing a `pyproject.toml`.

PEP 518 introduced `pyproject.toml`:

> *This PEP specifies how Python software packages should specify what build dependencies they have in order to execute their chosen build system. As part of this specification, a new configuration file is introduced for software packages to use to specify their build dependencies (with the expectation that the same configuration file will be used for future configuration details).*

The interesting part of that PEP for the purpose of this article is the tool section.

Rather than having one config file for each tool like Flake8 or Black, having all of them consolidated in one file would be very valuable.

Poetry uses this file to list the dependencies, keeping everything neatly in place.

There are two ways to get started with poetry:

- `poetry new SOME_NAME`: will create a folder named SOME_NAME with some basic structure setup for a Python project
- `poetry init`: an interactive way to setup your project and some dependencies

The downside of the `poetry init` is that the package search is not that great: searching for `django` will return a list of 99 packages, all of them being `django-something` and not the actual django. I don't know what Pypi offers as an API so maybe it is a limitation of the API but even then, showing 99 results is not very user friendly and if there is an exact match it should show it to you. The help text says I can enter the exact package name but typing `django` tries to autocomplete with the package names from the list and results in an error:

```
 > django
Value "django" is invalid

Enter package # to add, or the complete package name if it is not list
 [ 0] django-bagou
 [ 1] django-maro
 ... 99 packages
```

Poetry will also create virtualenv automatically when running `install`, `add` or `remove` with the current version of `python`: you cannot currently create a virtualenv with a different version.

You are supposed to activate a shell with `poetry shell` but it breaks if you using `$WORKON_HOME` for other tools: https://github.com/sdispater/poetry/issues/214 so I ended up ignoring the virtualenv related commands. I didn't spend time looking into it so there might be an easy fix but it should work out of the box with the rest of the ecosystem.

If you try to install a package using `poetry add django`, you will notice that a `pyproject.lock` has been created and that some dependencies were added to `pyproject.toml`:

```
[tool.poetry.dependencies]
python = "3.7"
django = "^2.1"

[tool.poetry.dev-dependencies]
pytest = "^3.0"
```

As opposed to the other tools, `poetry` uses the semantic versioning to refer to dependencies versions. If you have used Cargo in Rust, you will feel at home. The issue is that many Python libraries do not use semantic versioning so it does feel a bit weird to use in this context.

While trying to clean its cache to test the speed, I ran into a small UX issue. Poetry offers a command to clean the cache (`poetry cache:clear`) but it takes a path/string to a cache as an argument. A new user like me will not know what argument to give, especially since the command is not in the documentation. Looking into issues, we can see that we can run `poetry cache:clear --all pypi` but the documentation doesn't even mention `cache:clear` so it's not obvious.

As expected, the initial `poetry install` takes some time but subsequent runs take under a second. The UI is very clean and clear about what was installed and their versions.

Poetry also has `poetry update` to update dependencies and once again makes it clear what has changed.

While this post is focused on backend applications, it is worth mentioning that Poetry apparently handles building and publishing libraries as well which is a big plus if you are a maintainer: one tool for all purposes!

My opinion

I have only mentioned 3 tools but there are way more! I wouldn't be surprised if Python had the highest number of tools dedicated to package and environment management of any programming language. Since all those tools are written in Python themselves you still need to use Pip first to install them, globally sometimes. A better solution in my opinion would be to write the package manager in a language compiling down to a binary, solving the bootstrapping issue and being a good example of https://xkcd.com/927/.

To go back to the three tools mentioned in this article, I would recommend either pip-tools or poetry. My limited usage of Pipenv had too many WTFs to be considered ready for actual use, not even taking into account its extreme slowness. I'm not entirely sure why it is the recommended tool other than the virtualenv setup was easy to do because it feels like an alpha version. If you are using Docker containers instead of virtualenvs, I don't see any advantage of Pipenv over pip-tools or Poetry.

I like the simplicity of `pip-tools`. If you are currently writing `requirements.txt` by hand, using `pip-tools` to get some extra features like hashes is a pretty small step to take which I would recommend right now.

On the other hand, if `pyproject.toml` gets adopted by other tools (Flake8, mypy etc), having everything in one file would be very appealing and poetry would become the best choice at that time. I expect `poetry` to work with my current use of `virtualenvwrapper` though, I don't want to convert all my projects to `poetry` just to make it happy.