Poetry, pyenv and other rascals

On modern Python versions, environments and dependencies management.

PIP

Handy little tool for installing packages

Installing packages? PIP to the rescue.

Nanana

PIP

Handy little tool for installing packages

Installing packages? PIP to the rescue.

Pip is a tool most of you should already know. It's used to install packages used in python development and since a couple of version already, is shipped with Python by default.
But what does it exactly mean to install packages?

In short pip just provides tooling around downloading packages from Python Package Index – PYPI. It's a default index of Python packages where almost anyone can add packages.

Default is a good word, because pip allows us to use different indexes. So for example your company could have it's self-hosted version of packages and then use it as a private version of pypi. This allows for example better packages verification, only private network calls during CI/CD/development processes.

It's quite interesting option especially given latest malicious attacks on popular Python opensource packages.

What exactly is a **Package Index?**

Actually nothing complex. **It's just a http server,** let's say, that provides **a list of bundles of Python code – packages** and **some metadata** about them. Nothing more.

Fun take-home assignment to experiment with something new: **try to implement your own version of pypi and add there certain features like token-protected package access or even more tokens with granular permission feature.**

PIP

Handy little tool for installing packages

Useful pip commands for daily work

```
pip install package
pip install "package>=1.0"
pip install package==1.0
pip install –r requirements.txt

pip uninstall package

pip list #  lists all packages globally
pip list –-outdated

pip freeze  #  lists packages globally but without dependencies of pip and build stuff
pip freeze > requirements.txt

pip show package

pip search "query"
```

So, we install these packages, where to exactly?

Default package installation

Usually the case is that we have one, maximum two backwards incompatible versions of Python installed on our machine. In the past it used to be Python2 & Python3, Nowadays most of the time just Python3 is supported, installed as Python2 reached EoL.

Anyhow. This means that in the Dark Ages or by default one would install the packages globally, for the whole system. That's is bad for multitude of reasons. As for what installing package means, in a very big summary, it's nothing more than downloading a bundle of python code structured in a certain way, that gets downloaded and put in a given directory of python installation, with additional steps possible inbetween.

What if project A requires package Z in version 1.0.0, but project B requires package Z in version 2.0.0? Would you reinstall this package every time you switch to different projects?

virtualenv

Any python dev’s favourite pet

(Virtual) Environment management? Virtualenv to the rescue.

To combat the problem described in the previous slide -> packages getting installed globally, **virtualenv** came around.

In short, it’s something that allows us to **„create” another „instance” of installation of Python**. Eg. For a given project.

This way we can have various python package versions for various project.

Subset of virtualenv comes integrated with **default CPython installation from version 3.3 onwards.**

All is golden now, we can live happily ever after?

No.

poetry

Everyone's gotta be a little fragile and vulnerable sometimes…

What if pip and virtualenv had a love child that was also doing steroids?

The problem with pip is usually dependency version management. So.

Even if we know, our project A, requires package Z in version 1.0.0, usually at the first glance, pip doesn't tell us about the dependencies of this Z package.

It introduces the possibility of problems when your project reaches a point where it has a little bit more packages installed. Because these packages also have dependencies and their dependencies also have them.

Usually it's not a dependency hell like in the JS worlds, but at some point it can also get a bit tricky if you only lock the dependencies at the top level.

And at some point, when you reach corporate-level size of a project, it's almost guaranteed to have problems with this.

Also if the versions of these dependencies aren't guaranteed by default what about debugging?

I mean one build could have versions 1.2.3 of some dependency of a dependency, but another build, done 10 minutes earlier could have 1.2.2 if the versions aren't resolved in a deterministic, guaranteed way. It enables nasty bugs to appear.

This also is a security risk, because if you do not know what version of the dependency exactly you have, **a malicious version might** find their way in without our explicit knowledge, which is a **vulerability introduction opportunity.**

poetry

Everyone’s gotta be a little fragile and vulnerable sometimes…

Here comes the bo- Poetry.

We have something called **dependency resolving** and **dependency locking.**
Basically it’s just a proces of making sure that **we know the dependencies of our dependencies and their dependencies.**
And also we have a clear account of their **versions, usually signed with a hash.**

This allows something called deterministic builds which is one of the keys of **modern CI/CDs** and apps that adhere to **the Twelve-factor app pattern**.

This is exactly what Poetry does and it does this well.

Other than that, while we are at it, poetry **also makes projecet management easier**, takes care of **creating and managing virtualenvs for you** and enables easier, more **centralised project configuration by introducing pyproject.toml.**

pyproject.toml is usually the new standard python package config file.

Oh also, it makes building the packages easier as **it can bundle your python code and publish it to the package index of your choice.**

Overall poetry is neat. Very neat.

poetry

Everyone's gotta be a little fragile and vulnerable sometimes...

Example pyproject.toml

```
[tool.poetry]
name = "django-boilerplate"
version = "0.1.0"
description = ""
authors = ["Your Name <you@example.com>"]

[tool.poetry.dependencies]
python = "^3.10"
django = "^4.0.6"
psycopg2-binary = "^2.9.3"
celery = {extras = ["redis"], version = "^5.2.7"}

[tool.poetry.dev-dependencies]
coverage = "^6.4.2„
isort = "^5.10.1"
flake8 = "^4.0.1"
black = "^22.6.0"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Now all is dandy for sure, right? We got isolation of python environment, we got dependency resolvement, dependency locking, packaging, project configuration. Now for sure we can live happily ever after, yes?

No.

pyenv

Me and my N different skins.

Changing skins

Python is a peculiar little animal that sheds its skin from time to time. Meaning Python itself, other than our dependencies, also has it’s own versions. Each version contains new features, various improvements. Some of them are sometimes even backwards incompatibile.

By default it’s not trivial to install various python versions and have them working properly on the same machine.

Why would you need that? Well, same as with dependencies. One project could depend on Python 3.10, the other on 2.7 and some other on 3.12.

We need something like virtualenv, that would provide isolation, but instead of a project-level for python dependnecies, instead for the system on the python version level.

How do we do that?

With pyenv. Juiced up pyenv with a neat plugin that let’s us create virtualenvs from different python versions/interpreter implementations.

Pyenv + pyenv-virtualenv is also nice in regard of integration with poetry.

Anyhow. So we have pyenv-virtualenv which is a virtualenv-like wrapper for pyenv, which in turn is a a wrapper around python versions management, working on poetry which is a wrapper for pip and pip-tools, integrated with virtualenv which is also kind of a wrapper.

So we have wrapper of a wrapper working on a wrapper of a wrapper. Wrapper-ception.

Piptools

Lightweight approach

Version management done lighter

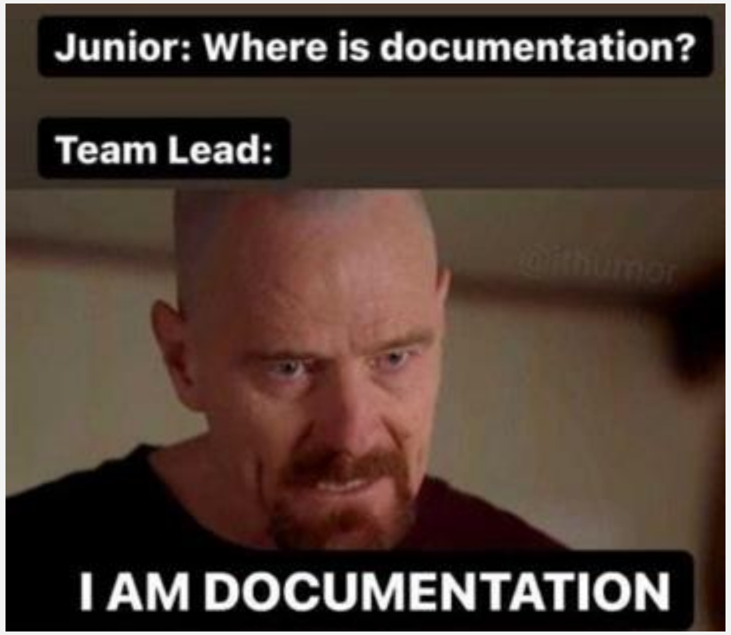
If your project is simple enough or you do not want to be bother will all of the previous things you can use pip-tools to pin your dependencies and all that.

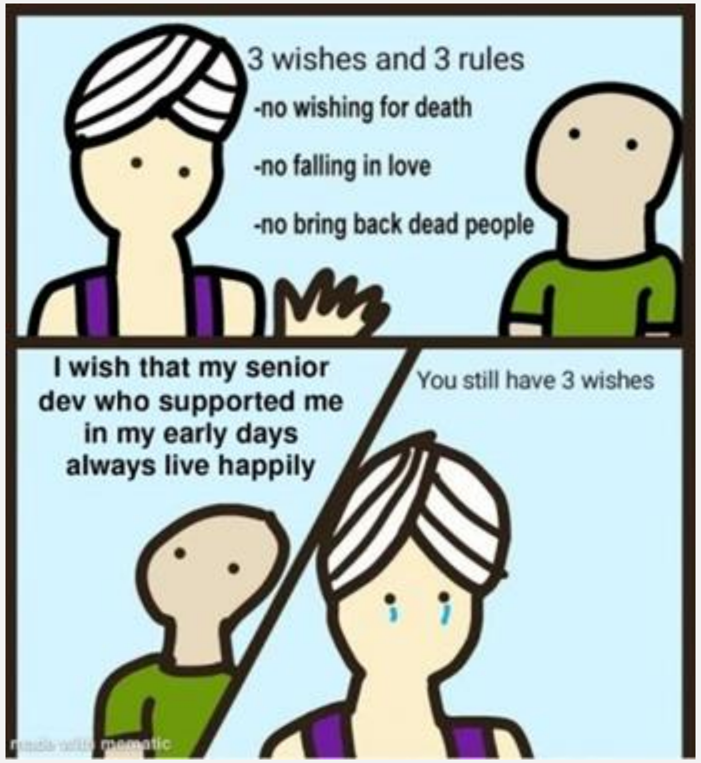
All is golden now, we can live happily ever after this time, right?

No.

Happy ever after is a myth and then you die, but at least we can have nice python versions, environment and dependency management, which is quite close in my books.

Serious self-righteous stuff aside,
here are some memes I enjoy to lighten up the mood




3 wishes and 3 rules
-no wishing for death
-no falling in love
-no bring back dead people
I wish that my senior dev who supported me in my early days always live happily
You still have 3 wishes
made with mematic

These were actually made by my former teammates. 10/10

Developers at the beginning of a project with Olaf
vs.
Developers at the end of a project.

Feedback? Questions?AMA.

Let’s get to know each other.

Thank you!