PACKAGING IN PYTHON?

DON'T ROLL THE DICE.

EFFECTIVE PACKAGE MANAGEMENT IS HARD.

WHAT SHOULD WE EXPECT FROM PACKAGE MANAGERS?

RECONCILES THE BEST VERSION OF EACH PACKAGE FOR ALL REQUIREMENTS.

```
  setuptools >= 36.2.1
+ setuptools == *
+ setuptools <= 37.0.0
= setuptools 36.7.2
```

WORKS TO OPERATING SYSTEM CONSTRAINTS.

(Like Window's default 260 character path limit)

ENSURES A PERFECTLY REPLICABLE DEPENDENCY TREE.

(Not just the packages you rely on — also the packages those packages rely upon, ad nauseum!)

SECURE.

... NOT OBVIOUSLY INSECURE?

Use different techniques to make sure the package you downloaded is the one you expected.

Doesn't help if you expected a malicious package.

Some package managers will automatically execute any scripts attached to a package as

pre/post-install.

The rest just show warnings that a user clicks through.

OPTIMAL DOWNLOAD EXPERIENCE.

Use a combination of techniques to make package downloads fast and reliable.

EMOJI.

Because `yarn` said so. 🎉

SO... PYTHON.

Python's current recommended package manager,

`pip`,

(which isn't generally installed by default if you're using a Linux distribution's package manager),

(and also isn't installed by default on macOS),

is available by default on Windows when you install Python!

RIGHT. EASY.

(USE PYENV EVERYWHERE, AND YOU'LL HAVE PIP, AT LEAST)

LINUX DISTRIBUTIONS PATCH PIP IN THEIR REPOSITORIES.

The default behaviour of trying to install packages to system-level directories is not helpful for the average Python developer.

`sudo pip install` is evil!

❌ WORKS TO OPERATING SYSTEM CONSTRAINTS.

(Though, the Linux distributions generally do the same thing in their own repositories in a slightly different way — the behaviour can be desirable!)

WHAT IF YOU NEED TO INSTALL TWO VERSIONS OF DJANGO AT THE SAME TIME?

```
$ pip3 install --user django==2.1

...
Installing collected packages: django
Successfully installed django-2.1
$ pip3 install --user django==1.11.15
...
Installing collected packages: django
Successfully installed django-2.1
```

THAT'S NOT DJANGO 1.11.15!

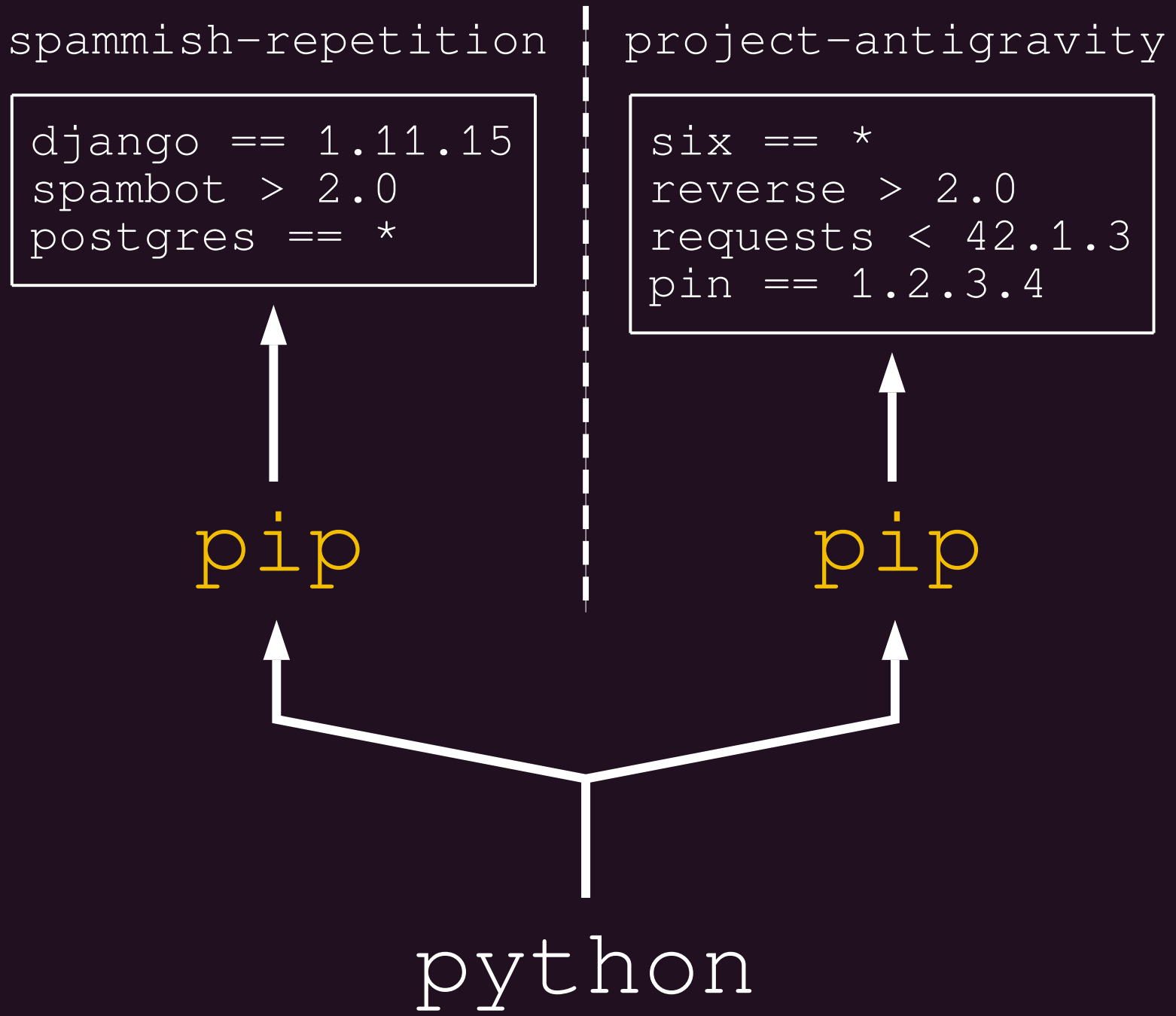
❌ ENSURES A PERFECTLY REPLICABLE DEPENDENCY TREE.

EMOJI ASIDE,

THAT'S TWO MAJOR ISSUES WITH DEFAULT PIP USAGE.

VIRTUALENV

virtualenv is a tool to create isolated Python environments.


spammish-repetition
project-antigravity
django == 1.11.15
spambot > 2.0
postgres == *
six == *
reverse > 2.0
requests < 42.1.3
pin == 1.2.3.4
pip
pip
python

THE MAGICAL VIRTUALENV ACTIVATE SCRIPT

`activate` sandboxes the session into using a specific (redirected) Python interpreter, and keeps packages inside the sandbox. It can optionally let you access packages installed elsewhere in the system, or restrict access entirely.

A VIRTUALENV PER PROJECT

If you use a `virtualenv` per project, each project can have completely independent dependencies, without any accidental upgrades or operations that affect another project.

PIP FREEZE

Using the `pip freeze` command, you can get a list of all packages installed by `pip`.

Inside a `virtualenv`, that's limited to packages you installed there.

```
autopep8==1.3.5
colorama==0.3.9
mock==2.0.0
nose==1.3.7
pycodestyle==2.4.0
pylint==1.9.2
rope==0.10.7
six==1.11.0
virtualenv==16.0.0
virtualenv-clone==0.3.0
```

ENSURES A PERFECTLY REPLICABLE DEPENDENCY TREE?

NOT QUITE.

TRANSITIVE DEPENDENCIES AREN'T LOCKED.

A package you rely upon might have a package **it** relies upon update, breaking compatibility.

Then your production deploy won't run, even though the "same dependencies" work on your machine. 😭

And as a side note, if you're forgetful like me, sometime you forget to `activate` your `virtualenv`.

Repeatedly.

PIPENV

Python Development Workflow for Humans

AUTOMATED VIRTUALENVS.

TRANSITIVE DEPENDENCY LOCKING.

ONE COMMAND TO USE AND REMEMBER.

SERIOUSLY, IT DOES MORE COOL THINGS THAN I REALISED!

```
$ pipenv --three
Using /home/liamdawson/.pyenv/versions/3.7.0/bin/python3 (3.7.0)
create virtualenv...
Virtualenv location: /home/liamdawson/.virtualenvs/wat-txLatJkZ
Creating a Pipfile for this project...
$ pipenv install django

Installing django...
Adding django to Pipfile's [packages]...
Pipfile.lock not found, creating...
Updated Pipfile.lock (4f9dd2)!
Installing dependencies from Pipfile.lock (4f9dd2)...
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv ru
```

```
$ pipenv run python -m 'django-admin' startproject helloworld
$ ls
helloworld/  Pipfile  Pipfile.lock
# or you could run pipenv shell, and then any old command.
```

NO NEED TO MAKE OR

ACTIVATE VIRTUALENVS.

They're still there and in use, though — `pipenv --venv`.

CHECK IN THE PIPFILE.LOCK

(Unless you're building a reusable library)

Now you have a locked dependency graph for fully replicable builds! 🙌

HOW HARD IS IT TO MIGRATE TO PIPENV?

```
$ git clone https://github.com/shalakhin/check_btc_tx.git
$ cd check_btc_tx
$ pipenv --three
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.
$ sed -i -Ee 's/==.+"$/*"/' Pipfile
$ pipenv install
$ pipenv run python check 1Archive1n2C579dMsAu3iC6tWzuQJz8dN
TXID:   a75fe75...
Status: Confirmed
Amount: 0.15997978
```

TESTS?

Handled natively as a "dev-only" dependency.

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

$ head --lines=-3 Pipfile | tail --lines=+5

[dev-packages]
nose = "*"

[packages]
click = "*"
requests = "*"
$ pipenv install # --dev
```

REVERSIBLE?

```
pipenv run pip freeze > requirements.txt
```

EMOJI?

Just not on Windows, where I ran my code.

(Time to write a PR!)

WHEN A NEW DEVELOPER STARTS ON YOUR PROJECT...

PIPENV INSTALL

- If they're using `pyenv`, it installs the right Python version for them.
- Sets up a `virtualenv` for the project.
- Installs the requirements (and dev requirements if you pass `--dev`).

- Can run scripts you set up, such as `pipenv run format`.
- Can still replicate `activate` behaviour from the `virtualenv` via `pipenv shell`

PREVIOUS EXAMPLE BECOMES

1. `git clone`
2. `pipenv install --dev`
3. `pipenv run check`
4. 🍻

WHERE DOES PIPENV FALL SHORT?

PIPENV IS A TOP-NOTCH TOOL FOR MANAGING AN APPLICATION WHICH:

- Runs on one version of Python.
- Isn't used as a dependency.
- Isn't packaged for PyPi etc.

PIPFILES DECLARE AN EXPECTED VERSION OF PYTHON

PIPFILE.LOCK RIGIDLY FIXES DEPENDENCY VERSIONS

(which doesn't suit usage as a dependency)

PIPENV WON'T HELP WITH PACKAGE CONFIGURATION

IF THOSE ASSUMPTIONS DON'T SUIT YOU?

TRY POETRY

(the tool)

POETRY IS A SOMEWHAT FULLER-

FEATURED TOOL FOR DEPENDENCY MANAGEMENT.

Poetry is a little more involved, and uses more metadata.

This equips it to support packaging and publishing, in addition to dependency management.

IF YOU USE PYENV TO MANAGE YOUR PYTHON VERSIONS

AND PIPENV OR POETRY TO MANAGE YOUR PROJECTS

DANGER NOODLE WILL BE KIND TO ALL.

FIN.

Want to follow up later? I'm on Twitter:

@LIAMDAWS

CREDITS

- Elephant photo by Inbetween Architects on Unsplash
- Original snake photo by Boris Smokrovic on Unsplash