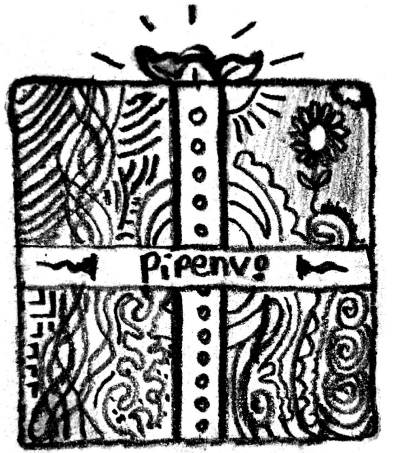

# Python的依赖管理及Pipenv

@frostming


# 我是谁？

- 研发效能·持续集成@Tencent
- 从 2014 年开始用 Python，前测试，现开发
- 开源爱好者
- Pipenv 维护者之一




队长别开枪，是我！

# 目录 CONTENTS



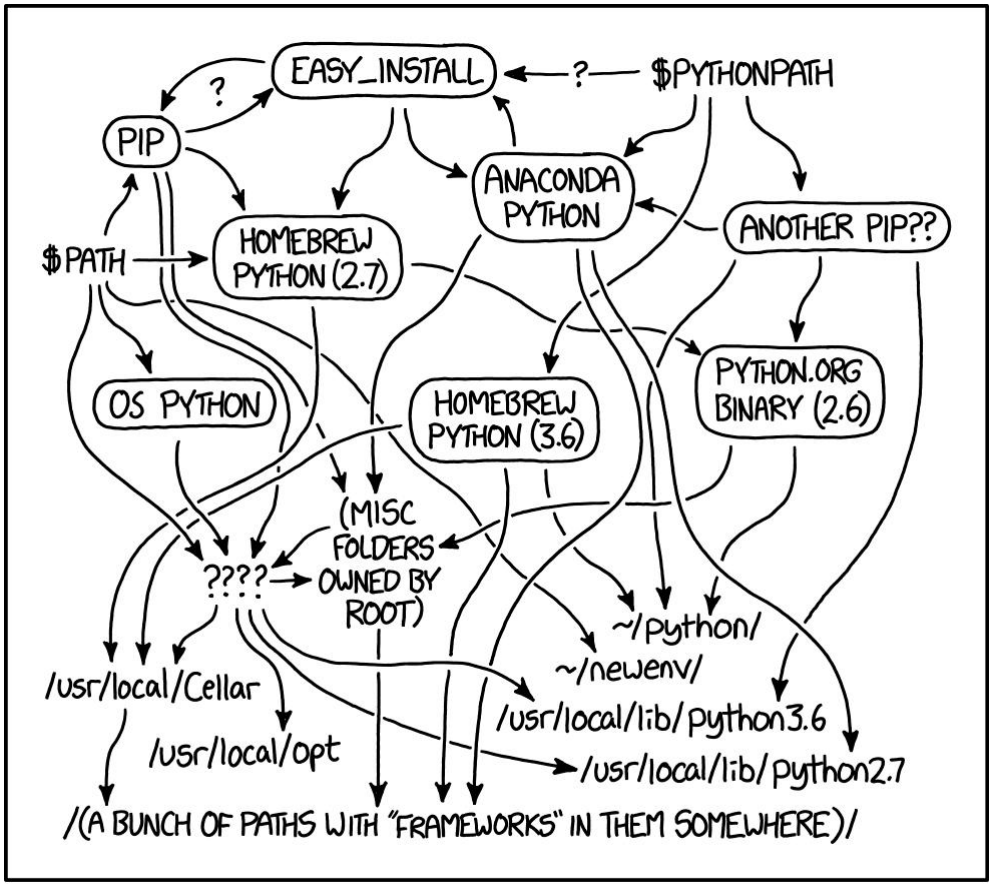

# 为什么需要虚拟环境

你是否遇到过这些问题：

- 安装了包却提示导入失败
  - 我的包安装到哪了？
- Python 版本升级了，原有的命令行程序全体罢工
  - 对，说的就是Homebrew
- 升级了某个库的版本，导致别的应用（库）挂了
  - 可能需要隔离环境了

# 混乱的 Python 环境






MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

# Python 寻找包的机制



```
In [1]: import sys

In [2]: sys.prefix
Out[2]: '/home/frostming/.pyenv/versions/3.7.4'

In [3]: import click

In [4]: click
Out[4]: <module 'click' from '/home/frostming/.pyenv/versions/3.7.4/lib/python3.7/site-packages/click/__init__.py'>
```

# Python 寻找包的机制

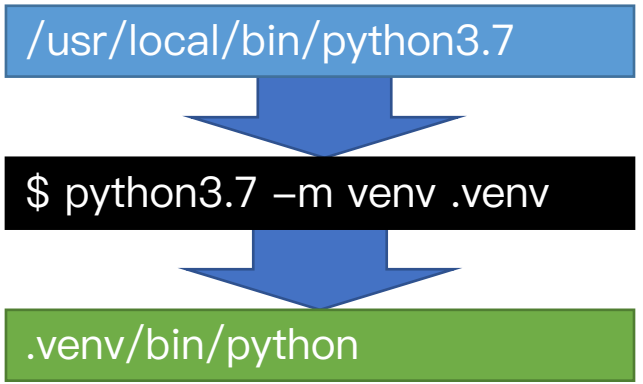


- 到 sys.prefix 下面对应的 Lib 目录寻找包
- PYTHONPATH 增加额外搜索路径
- 所有 CLI 应用程序均隐式对应一个 Python 解释器

File: /home/frostming/.pyenv/versions/3.7.4/bin/pip

```
1  #!/home/frostming/.pyenv/versions/3.7.4/bin/python3.7
2  # -*- coding: utf-8 -*-
3  import re
4  import sys
5
6  from pip._internal import main
7
8  if __name__ == '__main__':
9      sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$', '', sys.argv[0])
10     sys.exit(main())
```

# 使用虚拟环境隔离




sys.prefix 决定寻找包的路径及安装包的路径

每个环境都有不同的 sys.prefix

# 不佳实践

sudo pip install <package>

sudo ln –sf /path/to/my/python /usr/bin/python

使用 homebrew 安装的 Python 安装命令行程序

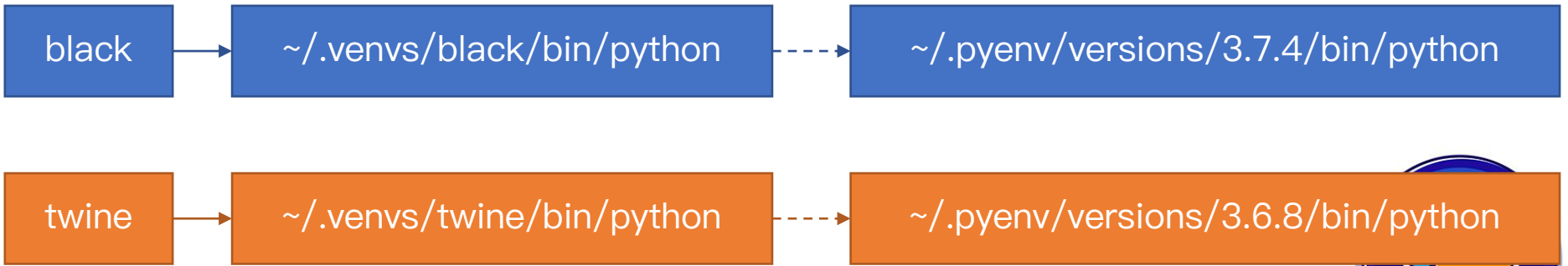
export PYTHONPATH=$PYTHONPATH:/my/path/to/libs

## 最佳实践

每个命令行程序使用自己单独的虚拟环境，然后将 executable 软链到 PATH 中

尽量使用 Python 3 自带的 venv 模块

pipx install <package>



# 虚拟环境工具

- –m venv
- virtualenv
- virtualenvwrapper
- pew
- pyenv–venv
- conda
- direnv
- ...

\+ pip ?


# 我们为什么需要依赖管理

pip 是否已经足够使用？

它的依赖管理有什么问题？

碰到这些问题，该怎么办？

# 依赖冲突

django-celery 依赖 celery>=3.1.15,<4.0

PyPI 上 celery 最新版 4.3.0

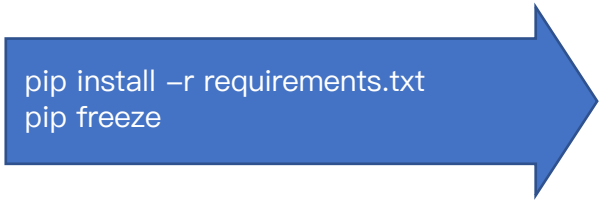
$ pip install celery django-celery


$ pip install celery

$ pip install django-celery

# requirements.txt 呢？

Flask
requests



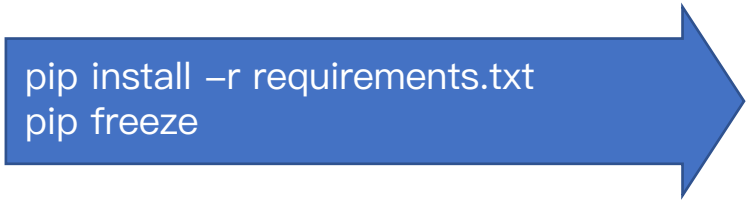

```
Flask==1.0.2
itsdangerous==1.1.0
Click==7.0
Jinja2==2.10.3
MarkupSafe==1.1.1
werkzeug==0.16.0
requests==2.22.0
idna==2.8
certifi==2019.9.11
chardet==3.0.4
urllib3==1.25.6
```

- 抽象依赖
- 开发环境使用
- 倾向使用最新版本
- 方便变更

- 具体依赖
- 生产环境使用
- 使用指定版本
- 不经常变更

## requirements.txt 呢？

pywin32; os_name==‘nt’

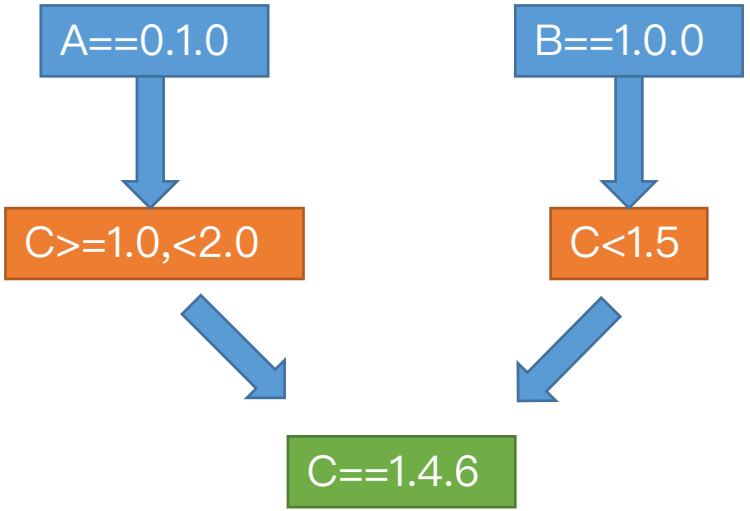



?

具体依赖不确定

不能保证环境可复制

# 为什么需要依赖解析




A==0.1.0
B==1.0.0
C==1.4.6

## 依赖解析需要什么

- 满足所有依赖限制
- 跨平台、跨 Python 版本
- 若找不到合适的版本，需要报告冲突

## 可用工具

- Pipenv (piptools)
- Poetry
- 没了


# Pipenv 的简单使用

- 自动创建、管理虚拟环境，与项目绑定
- 依赖解析、锁定
- 新的requirements.txt规范，基于TOML格式
- 优美的命令行界面

# Pipenv Demo

```
frostming@VM_52_123_centos~/w/pycon2019>
```


# Python 包管理的未来

# Python 包管理的未来

## PEP 517 – 与构建系统无关的配置文件格式

## PEP 518 – 在指定Python构建依赖

setup.py

```
from setuptools import setup

setup(
    name=NAME,
    version=VERSION,
    description=DESCRIPTION,
    install_requires=["reqests"]
)
```

与 setuptools 强耦合，文件内容难以解析，
依赖运行时确定
构建工具仅此一家别无分号

pyproject.toml

```
[build-system]
requires = ["flit"] # Defined by PEP517
build-backend = "local_backend"
backend-path = ["backend"]
```

方便工具解析内容，赋能更多构建工具

PEP 582 – Python 局部包目录

```
foo
 ├── __pypackages__
 │       └── 3.8
 │           └── lib
 │               └── bottle
 └── myscript.py
```

**摆脱虚拟环境**

pyflow – PEP 517 + PEP 582 + 依赖解析


# THANK YOU

frostming

知 frostming

frostming90

frostming