Better Python dependency while packaging your project

Itech Gal Follow
Mar 1, 2018 · 6 min read

I have been cooking this blog topic idea for a long time. I did a lot of searching, reading and trying while working on different projects. But even today after publishing it I don't think I'm 100% satisfied with the provided solution how to manage python project dependencies efficiently.

What is package and dependency management?

Software released in bundled packages this way it's easier to manage installed programs.

The package manager is a collection of libraries that are packaged together, which makes it easier to download entire package rather than each library.

Almost every library in the package has a dependency managed by the d**ependency manager.**

Dependency management helps manage all the libraries required to make an application work. It's incredibly beneficial when you're dealing with complex projects and in a multi-environment. Dependency management also helps to keep track, update libraries faster and easier, as well as solve the problem then one package will depend on another package.

Every programming language has its flavor of dependency manager.

To summarize all above :

- The library is a collection of already pre-written code.
- The package is a collection of libraries that are built on each other or using each other one way or another.

Typical way of managing project dependency today

Today the most used Python package manager is **pip,** used to install and manage python software packages, found in the Python Package Index. Pip helps us, python developers, effortlessly "manually" control installation and lifecycle of publicly available Python packages from their online repositories.

Pip also can upgrade, show, uninstall project dependencies, etc.

To install the package, you can just run `pip install <somepackage>` that will build an extra Python library in your home directory.

Running `pip freeze,` can help to check installed packages and packages versions listed in case-insensitive sorted order.

Project setup

After building your application, you will need to perform some set of actions(steps) to make application dependencies available in the different environments.

Actions will be similar to the one below:

- Create a virtual environment `$ python3 -m venv /path/to/new/virtual/env`
- Install packages using `$pip install <package>` command
- Save all the packages in the file with `$ pip freeze > requirements.txt.` Keep in mind that in this case, requirements.txt file will list all packages that have been installed in virtual environment, regardless of where they came from
- Pin all the package versions. You should be pinning your dependencies, meaning every package should have a fixed version.
- Add requirements.txt to the root directory of the project. Done.

Install project dependencies

When if you're going to share the project with the rest of the world you will need to install dependencies by running `$pip install -r requirements.txt`

To find more information about individual packages from the requiements.txt you can use `$pip show <packagename>`. But how informative the output is?

```
~ $ pip show ansible
Name: ansible
Version: 2.3.1.0
Summary: Radically simple IT automation
Home-page: https://ansible.com/
Author: Ansible, Inc.
Author-email: info@ansible.com
License: GPLv3
Location: /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages
Requires: jinja2, PyYAML, paramiko, pycrypto, setuptools
~ $
```

Example pip show command

How can project dependencies be easily maintained?

Personally, I think above setup is not easy to maintain, for the variety of reasons:

1. Sometime requirements.txt files contain more than thousands of lines. When maintain and update package version is hard, it will be even more hard to automate it (for example: delete development dependencies, etc.).
2. If versions are not pinned in `requirements.txt` the result executing a fresh `$ pip install -r requirements.txt` will result in different packages being installed every time then new, different versions of sub-dependencies will be released.
3. **`Pip`** doesn't have dependency resolution.
4. Sometimes you may want to create requirements.txt as an empty file without modules (this will not work with `pip freeze` command.)

Are there any better alternatives?

Option 1 : multiple requirements.txt files ?

There are many examples of projects with multiple requirements.txt files. Developers having different versions of requirements.txt file for example for different environments (e.g., test or local) or files for different users (e.g., machines vs. people).

Multiple requirements.txt is a good solution for managing project dependencies? I disagree…managing manually various requirements.txt files not a good solution, and it will be not easy if they grow more than even ~50lines.

Option 2: can Pipreqs and Pipdeptre make it better?

I recently tried `pipreqs` utility, which generates `requirements.txt` file based on project imports. It's simple to use.

To generate a requirements.txt file you can run `pipreqs /your_project/path`

```
~ $ pipreqs project_example
INFO: Successfully saved requirements file in project_example/requirements.txt
~ $ ls project_example/requirements.txt
-rw-r--r--  1 itechgirl  staff     1B Oct 10 23:01 project_example/requirements.txt
```

Example pipreqs command

Pipdeptree

I thought of combining it with `pipdeptree` another cool and "handy" command line utility which will help to display the installed python packages in the form of a dependency tree.

After executing `pipdeptree` command in your terminal window in the virtualenv directory of the project, all the installed python packages of a dependency tree will be displayed:

```
~ $ pipdeptree
ansible==2.3.1.0
  - jinja2 [required: Any, installed: 2.9.6]
    - MarkupSafe [required: >=0.23, installed: 1.0]
  - paramiko [required: Any, installed: 2.1.2]
    - cryptography [required: >=1.1, installed: 1.9]
      - asn1crypto [required: >=0.21.0, installed: 0.22.0]
      - cffi [required: >=1.7, installed: 1.10.0]
        - pycparser [required: Any, installed: 2.17]
      - idna [required: >=2.1, installed: 2.5]
      - six [require`d: >=1.4.1, installed: 1.10.0]
    - pyasn1 [required: >=0.1.7, installed: 0.3.2]
  - pycrypto [required: >=2.6, installed: 2.6.1]
  - PyYAML [required: Any, installed: 3.12]
  - setuptools [required: Any, installed: 36.3.0]
asyncio==3.4.3
```

Example `pipdeptree` command

Cool bonus that `pipdeptree` will warns you when you have multiple dependencies where versions don't exactly match.

I found it's handy in some cases, like:

- if you want to create a requirements.txt for a git repository and you only want to list the packages required to run that script; packages that the script imports without any "extras"
- support option like `clean` command
- can be used with `pipdeptree` to verify project dependencies

There are some downsides too, `Pipreq` will not include the plugins required for specific projects, and you will end up adding plugins information manually in a requirement.txt. It's not yet a very mature utility.

Option 3: have you tried pip-compile?

`pip-compile` module provides two commands: `pip-compile` and `pip-sync` .

`pip-compile` command will generate `requirements.in` or `requirements.txt` of top-level summary of the packages and all (underlying dependencies) pinned. And you can store .in and .txt files in version control.How useful, right?

This means that we can get the same versions whenever we run `pip install` , no matter what the new version is.

`-generate-hashes` flag helps to generate-hashes. In this case `pip-compile` consults the PyPI index for each top level package required, looking up the package versions available.

To update all packages, periodically re-run `pip-compile --upgrade` .

To update a specific package to the latest or a particular version use the `--upgrade-package` or `-P` flag.

`pip-sync` command used to update your virtual environment to reflect exactly what's in there. Command will install/upgrade/uninstall everything necessary to match the `requirements.txt` contents.

Software dependencies are often the largest attack surface

To know what dependencies your project hasT is very useful to find out after the fact what packages were installed and what dependencies your project has.

Organizations usually assume most risks come from public-facing web applications. That has changed. With dozens of small components in every application, threats can come from anywhere in the codebase.

Currently using `pip freeze` will only show the final package list of dependencies.

I would recommend using `pipdeptree` module which helps to find possible dependency conflicts and displaying an actual dependency in the project.

`pipdeptree --reverse` leaf package installed first

Another good advice is start building applications using **Docker**. Every tool we run in Docker is one less tool we have to install locally, so get up-and-running phase will be much faster. But that is a different topic.

Happy Packaging .

Interesting reading:

Wiki about pip https://www.kennethreitz.org/essays/announcing-pipenv

Pinning packages: http://nvie.com/posts/pin-your-packages/

Docker: https://www.fullstackpython.com/docker.html

Python Dependencies