Se connecter
Proposer un contenu
Identifiant
Identifiant
Mot de passe
Mot de passe
Connexion automatique
Se connecter
Pas de compte ? S'inscrire...

# Journal : Gestion de dépendances en python

Posté par **une**flanche (site web personnel, Mastodon) le 11 novembre 2026 à 18:34. Licence CC By-SA.
Étiquettes : python, pip

## Sommaire

- Intro
- Zig
- Dépendances Python à la Zig
  - Récupération des dépendances
  - Configuration à partir d'une archive
  - Quelques liens utiles
- Conclusion

Je vous partage ici mon retour d'expérience sur la gestion des dépendances en python.

L'article original est disponible ici.

## Intro

Dans le cadre de mon travail, j'ai régulièrement besoin d'écrire des scripts en python ou bien des notebook Jupyter. Je n'ai pas particulièrement d'affinité avec ce langage, disons qu'il s'est imposé à moi :-) En revanche, il y a quelque chose qui me chagrine, c'est la gestion des dépendances. C'est particulièrement agaçant de reprendre un script et de voir que les dépendances ne sont pas indiquées ou bien il manque leur version et loi de Murphy oblige, l'API a bougé... Bref, je pense lister les problématiques suivantes dans la gestion des dépendances de mes projets python:

- lister les dépendances
- fixer la version des dépendances
- fixer la version des dépendances des dépendances
- Obtenir les dépendances, comment travailler hors-ligne, que faire si `pypi` tombe

On peut déjà indiquer quelques solutions qui font consensus pour gérer une partie de ces problèmes. En particulier, on peut commencer par lister nos dépendances dans un fichier texte. Si vous utilisez `pip` pour installer vos dépendances, il dispose d'une solution standardisée: `requirements.txt`.

Cette solution permet de résoudre nos deux premiers problèmes:

- lister les dépendances: `scipy matplotlib numpy`
- fixer la version des dépendances: `scipy == 1.18.* matplotlib >= 3.11.2 numpy == 2.5.4`
- fixer la version des dépendances des dépendances ---> _ßOß_/
- Obtenir les dépendances, comment travailler hors-ligne, que faire si `pypi` tombe --> _ßOß_/

Cependant lorsqu'on reprend un vieux script, il arrive parfois que la version indiquée dans le fichier `requirements.txt` ne soit plus distribué sous forme binaire par pypi. Il faut alors compiler nous-même cette dépendance. Un cas concret: si je reprends mon exemple précédent mais que je fixe `numpy == 2.0.*`, une version qui a un peu plus de deux ans lorsque j'écris ces lignes. Pypi ne propose plus de téléchargeur directement les fichiers pré-compilés (`wheels`). `pip` va donc télécharger les sources et compiler le projet sur votre ordinateur. C'est à ce moment-là que ça peut se compliquer. En effet, certain projet ont la réputation d'être difficile à compiler. Maintenant extrapolons cette situation sur toute une cascade de dépendances, et vous apercevez la profondeur du gouffre...

Rappelons-le, à la base on souhaitait simplement réutiliser un script sur un vieux projet ;-)

## Zig

Un autre langage que j'utilise régulièrement: zig, je l'apprécie pour bien des raisons et l'une d'entre elles est sa gestion des dépendances. Pour résumer trivialement (explications valable pour la version 0.16.0 et 0.17.x, ces caractéristiques sont susceptibles d'évoluer):

- les dépendances sont listées dans un fichier texte: `build.zig.zon`
- chaque dépendance est fixée sur une révision de commit ou une version donnée
- les sources des dépendances sont téléchargées dans le répertoire `zig-pkg` (on peut si nécessaire les ajouter au VCS)
- On peut spécifier un fork local d'une dépendance (c'est toujours utile en phase de dev)

Pour plus d'informations, n'hésitez pas à lire ce post: Two Package Management Workflow Enhancements.

Pourquoi je vous parle de ça ? Parque j'aimerai bien retrouver un comportement similaire pour mes scripts en python...

## Dépendances Python à la Zig

La première fonctionnalité est sûrement la plus simple, la gestion d'un fork. `pip` gère nativement ce problème, il vous suffit d'installer la dépendance avec la commande suivante:

```
pip install -e path/to/project
```

Si votre paquet était déjà présent dans votre installation alors `pip` va le supprimer et ajouter le répertoire de votre projet à la liste des chemins d'import. Et c'est tout. Maintenant lorsque vous modifiez votre fork en local, les changements sont directement pris en compte dans votre projet.

### Récupération des dépendances

Passons maintenant à la partie récupération en local des dépendances. On commence par créer un répertoire `py-pkg` à la racine de votre projet. Maintenant, on va se servir de `pip` pour télécharger l'ensemble des dépendances dans ce répertoire:

```
py -m pip download --destination-directory py-
```

`pip` va télécharger les fichiers wheels de l'ensemble des dépendances dans ce dossier, mais aussi des dépendances de

> **Attention**
> Les fichiers `wheels` sont des fichiers binaires dépendant de la plateforme cible (car compilés). On peut toujours télécharger les sources avec la commande `py -m pip download --no-binary :all:` mais là dos le déconseille, car certain paquet sont difficiles à compiler... De plus, après avoir testé, ça ne télécharge que la dépendance et non toutes l'arborescence de dépendances...

### Configuration à partir d'une archive

Il n'est pas conseillé d'ajouter ces versions à votre VCS, car ce sont des fichiers binaires dépendants de votre plateforme, mais vous pouvez au moins les ajouter à votre archive lorsque vous taguez une nouvelle version de votre script.

Maintenant, on peut installer les dépendances de notre projet à partir de leur archive:

```
py -m venv .env
source .env/bin/activate
py -m pip install --no-index --find-links=py-g
```

### Quelques liens utiles

- pip local package
- pypi proxy: devpi

## Conclusion

Cette solution ne permet pas d'obtenir un fonctionnement calqué sur celui de zig. Cependant, elle permet de pérenniser l'archivage de vos scripts et de ses dépendances.

En effet, en archivant les fichiers `wheel` de votre projet vous vous assurez de pouvoir le réutiliser sans souci le jour venu. Enfin, si vous disposez toujours d'un interpréteur python compatible...

Ces problèmes n'en finiront jamais ;-)

(11 commentaires) Markdown EPUB

---

### # Soluce ultime

Posté par Sylvia Mobin (site web personnel) le 11 novembre 2026 à 20:59. Évalué à -2 (+4/-6).

### # zipper le virtual env ?

Posté par steph1978 le 11 novembre 2026 à 21:11. Évalué à 3 (+1/-0).

Merci pour ce journal.

Si on veut pouvoir fournir un script python en local, archiver le virtual env (zip, tgz) devrait suffire : toutes les dépendances `venv/lib/python3.x/site-packages`. Pour le réutiliser il faudra peut être ajuster les liens symboliques qui se trouvent dans `venv/bin`.

Répondre

#### [^] # Re: zipper le virtual env ?

Posté par **une**flanche (site web personnel, Mastodon) le 11 novembre 2026 à 21:43. Évalué à 1 (+0/-0).



Au départ je cherchais une solution pour gérer mes dépendances comme avec zig et notamment pouvoir facilement travailler hors-ligne. De ce fait je n'ai pas cherché du côté du `venv` mais vu qu'au bout du compte j'obtiens plus une solution d'archivage que de gestion de dépendances, ça vaudrait sans doute le coup que j'y jette un œil ^

Répondre

---

### # pyproject.toml et UV

Posté par **bind_magic** le 11 novembre 2026 à 21:28. Évalué à 10 (+10/-0). Dernière modification le 11 novembre 2026 à 21:33.

`pyproject.toml` s'est imposé comme le standard officiel (PEP 517/518/621) pour déclarer la configuration et les dépendances d'un projet Python. Pour la gestion des environnements virtuels et l'installation des dépendances et de l'outillage, uv offre un vrai gain (un peu comme cargo pour Rust). Accessoirement il est beaucoup plus rapide que pip.

Répondre

#### [^] # Re: pyproject.toml et UV

Posté par **une**flanche (site web personnel, Mastodon) le 11 novembre 2026 à 22:03. Évalué à 2 (+1/-0).



Je ne connaissais pas uv, merci pour la suggestion.

Répondre

#### [^] # Re: pyproject.toml et UV

Posté par **steph1978** le 12 novembre 2026 à 11:07. Évalué à 2 (+0/-0). Dernière modification le 12 novembre 2026 à 11:08.

Je trouve que UV est largement sur vendu par ses promoteurs. Y a pas plus par rapport à pip, mais rien qui ne le rende indispensable.
Et de se retrouver avec un outil dont la gouvernance est assurée par une boîté privée.

> A single tool to replace `pip`, `pip-tools`, `pipx`, `poetry`, `pyenv`, `twine`, `virtualenv`, and more.

- pip => donc pip
- pip-tools => un wrapper au dessus de pip, s'installe en module
- pipx => idem ; très pratique
- poetry => une alternative à pip, donc pip
- pyenv => j'aime mieux `asdf`
- twine => utile seulement pour celleux qui publient des packages
- virtualenv => `python -m venv`, trop dur :)
- and more => je vous le mets quand même ?

- 10-100x faster than `pip`.

Surement. Dans le benchmark, on gagne 4 seconds sur l'installation des dépendances. Mais ce n'est pas quelque chose qui se fait très souvent. Peut être pour un outil de CI qui fait ça en masse.

- Provides comprehensive project management, with a universal lockfile.

Subjectif.

- Runs scripts, with support for inline dependency metadata.

Ça c'est plutôt cool. Comme Mix install pour Elixir. Je ne sais pas si le format qu'ils utilisent est standard cependant.

- Installs and manages Python versions.

Se fait très bien avec asdf

- Runs and installs tools published as Python packages.

"similar to pipx", donc pas grand chose à en dire

- Includes a pip-compatible interface for a performance boost with a familiar CLI.

rétro compatibilité pip, donc pip

- Supports Cargo-style workspaces for scalable projects.

Pour ma part, je fais pas de scalable projects

- Disk-space efficient, with a global cache for dependency deduplication.

À creuser, ça peut être intéressant quand on multiplie les projets en python : pip pourrait peut être s'y mettre. Le soucis ce sera le garbage collection (= comment savoir qu'un package n'est plus utilisé nulle part).

- Installable without Rust or Python via `curl` or `pip`.

Résoud une problématique spécifique à uv, pas à pip

- Supports macOS, Linux, and Windows.

Comme pip.

Chez Astral, je préfère `ruff` le `ty` le type checker. Car 1./ ils sont vraiment rapide pour des tâches qui sont effectuées très très souvent, voire, en permanence 2./ ils n'ont pas d'impact sur la structure du projet : à tout moment, ils peuvent être remplacés par un équivalement (pyslp, mypy).

Répondre

**[^] # Re: pyproject.toml et UV**
Posté par **flagos** le 12 novembre 2026 à 14:42.
Évalué à 2 (+1/-1).

> Pour bon nombre de projets mono-repo, la vitesse apportée par uv n'est pas juste un benchmark :-) Si vous souffrez des lenteurs de pip, poetry ou ses copains, essayez uv, c'est vraiment bien.

Répondre

**[^] # Re: pyproject.toml et UV**
Posté par **steph1978** le 13 novembre 2026 à 00:01.
Évalué à 2 (+0/-0).

> Mono repo ou pas, je vois pas ce que ça change. Ce qui compte, c'est combien de fois on fait un `pip install`. Sur un poste de développeur : pas tant que ça. Sur une CI, oui, souvent.

Répondre

**# Corrections**
Posté par **42nodal** le 12 novembre 2026 à 10:52.
Évalué à 2 (+1/-0).

> Maintenance lorsque vous modifier votre fork en local, les changements sont directement prient en compte dans votre projet.

Maintenant [...] pris en compte [...]

non ?

Répondre

**[^] # Re: Corrections**
Posté par **unSanchix** (site web personnel, Mastodon) le 12 novembre 2026 à 12:42. Évalué à 1 (+0/-0).

> Ce sont bien des erreurs. Merci. Si un admin peut corriger :-)



Répondre

**[^] # Re: Corrections**
Posté par **Benoît Sibaud** (site web personnel) le 12 novembre 2026 à 15:15. Évalué à 4 (+1/-0). Dernière modification le 12 novembre 2026 à 15:15.

> Corrigé, merci.

Répondre

Envoyer un commentaire

Suivre le flux des commentaires

**Note** : les commentaires appartiennent à celles et ceux qui les ont postés. Nous n'en sommes pas responsables.

Revenir en haut de page