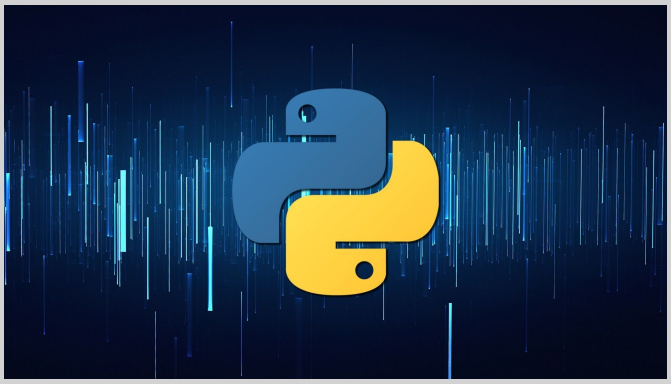
Environnements de programmation Python, et les outils dédiés



Krystof26

23 mai 2025

Préface

Ce petit guide est conçu pour fournir une compréhension approfondie des environnements virtuels en Python et des outils associés, essentiels si l'on souhaite optimiser son flux de travail. Les environnements virtuels permettent de créer des espaces isolés pour chaque projet, facilitant ainsi la gestion des dépendances et des versions de bibliothèques en évitant tout. Cette isolation est cruciale pour éviter les problèmes de compatibilité entre les différentes versions de bibliothèques utilisées par divers projets. En utilisant des environnements virtuels, nous pouvons nous assurer que chaque projet fonctionne dans un environnement propre et contrôlé, ce qui simplifie la gestion des dépendances et améliore la reproductibilité des projets.

Nous commencerons par une exploration du module intégré **venv** qui offre une manière simple et efficace de créer et de gérer des environnements virtuels. Ce module est fondamental, car il permet de maintenir un environnement propre et organisé pour chaque projet.

Ensuite, nous examinerons une variété d'outils qui complètent et étendent les fonctionnalités de **venv**. Parmi ceux-ci, **pip**, le gestionnaire de paquets par défaut de Python, est indispensable pour installer et gérer les bibliothèques Python. Nous aborderons également **pyenv**, un outil puissant pour gérer plusieurs versions de Python sur un même système, et **poetry**, un outil moderne pour la gestion des dépendances et des environnements virtuels qui simplifie la création et la gestion des projets Python.

Enfin, nous terminerons par une présentation exhaustive d'**uv**, un outil moderne et performant pour la gestion des environnements virtuels et des dépendances. **uv** se distingue par sa rapidité et son efficacité, offrant une alternative robuste aux outils traditionnels.

Ce guide est destiné à tout développeur Python, autant débutant qu'expérimenté, cherchant à améliorer sa compréhension et l'utilisation des environnements virtuels et des outils associés.

Il faut savoir que les environnements virtuels et les outils associés sont en constante évolution, et de nouvelles solutions émergent régulièrement pour répondre aux besoins changeants des développeurs. Ce guide ne peut donc être une finalité en soit, j'encourage à rester informés des dernières avancées et ainsi explorer de nouvelles technologies pour continuer à améliorer sa pratique de développement.

Table des matières

1

Les environnements virtuels
Isoler et optimiser ses projets

> *“L’essence de l’homme est d’être virtuel, parce qu’il ne peut se satisfaire de sa réalité passagère.”*
> — Philippe Quéau

L’utilisation d’environnements virtuels est une pratique courante et efficace dans le développement Python. Ils permettent de gérer les dépendances séparément pour différents projets, ce qui évite les conflits et permet de conserver des configurations plus propres. Ainsi, chaque fois que l’on travaille sur un projet Python qui utilise des dépendances externes que l’on installe avec **pip**, il est préférable de créer d’abord un environnement virtuel.

Les environnements virtuels sont des outils essentiels pour tout développeur Python, car ils offrent une solution simple et efficace pour isoler les dépendances de chaque projet. Cela signifie que nous pouvons travailler sur plusieurs projets simultanément, chacun ayant ses propres versions de bibliothèques, sans craindre les conflits de versions. Par exemple, un projet peut nécessiter **Pyside 6.4**, tandis qu’un autre projet peut nécessiter **Pyside 6.9**. Sans environnements virtuels, il serait difficile de gérer ces dépendances de manière propre et organisée.

En créant un environnement virtuel pour chaque projet, nous pouvons installer les dépendances spécifiques à ce projet sans affecter les autres projets ou le système global. Cela est particulièrement utile lorsque vous travaillons sur des projets de grande envergure collaboratifs. Les environnements virtuels permettent de s’assurer que tous les membres de l’équipe utilisent les mêmes versions de bibliothèques, ce qui facilite la collaboration et réduit les risques d’erreurs.

1.1 Le module venv

Pour une utilisation basique, **venv** est un excellent choix car il est déjà fourni lors de l'installation de Python. Le module **venv** est un outil intégré à Python qui permet de créer et de gérer des environnements virtuels de manière simple et efficace. Il est particulièrement utile pour les développeurs qui souhaitent isoler les dépendances de leurs projets sans avoir à installer des outils supplémentaires.

L'un des principaux avantages de **venv** est sa simplicité d'utilisation. Pour créer un nouvel environnement virtuel, il suffit d'exécuter la commande :

```
$ python -m venv nom_de_l_environnement
```

Cette commande crée un nouveau répertoire contenant une copie de l'interpréteur Python, ainsi que les répertoires nécessaires pour installer les bibliothèques spécifiques au projet.

Une fois l'environnement virtuel créé, nous pouvons l'activer en utilisant la commande appropriée (selon le système d'exploitation). Par exemple, sur un système Unix ou MacOS, nous utiliserons la commande :

```
$ source nom_de_l_environnement/bin/activate
```

Une fois activé, nous pouvons installer les dépendances spécifiques à notre projet en utilisant **pip**, le gestionnaire de paquets de Python.

Le module **venv** est également très flexible. Il permet de spécifier la version de Python à utiliser pour l'environnement virtuel, ce qui est particulièrement utile lorsque nous travaillons sur des projets nécessitant des versions spécifiques de Python. De plus, **venv** est compatible avec la plupart des outils et bibliothèques Python, ce qui en fait un choix polyvalent pour une large gamme de projets.

Installation de venv

L'installation de **venv** se réalise au niveau du système global. Sur un système **Debian GNU/Linux** :

```
user@machine: # apt install python3.13-venv
Installation de :
  python3.13-venv

Installation de dependances :
  python3-pip-whl  python3-setuptools-whl

[...]
```

```
Dépaquetage de python3.13-venv (3.13.3-2) ...
Paramétrage de python3-setuptools-whl (78.1.0-1.2) ...
Paramétrage de python3-pip-whl (25.1.1+dfsg-1) ...
Paramétrage de python3.13-venv (3.13.3-2) ...
```

Créer un environnement virtuel

Se rendre dans le répertoire du projet, et saisir la commande :

```
user@machine: $ python3 -m venv venv/
```

A noter que par convention l'environnement virtuel est souvent nommé **venv** ou **env** ou **.venv**. De plus, il n'est pas nécessaire d'utiliser une barre oblique à la fin du nom, mais elle est là pour rappeler que c'est un répertoire qui est créé.

Nous pouvons donc vérifier la présence du répertoire contenant notre environnement au sein de notre projet :

```
user@machine: $ ls -a
.  ..  .venv
```

Activation de cet environnement virtuel

```
user@machine: $ source .venv/bin/activate
(.venv)user@machine: $
```

Noter la modification du prompt qui est désormais précédé du nom de l'environnement virtuel.

Il est toutefois possible de travailler sur ses fichiers sans activer l'environnement virtuel mais l'activation sera nécessaire avant l'exécution du script.

Installer des dépendances

Après avoir créé et activé l'**environnement virtuel**, nous pouvons installer toutes les dépendances externes dont nous avons besoin pour notre projet. Installons par exemple le module **Pyside6** :

```
user@machine: $ python3 -m pip install pyside6
Collecting pyside6
```

```
  Using cached PySide6-6.9.0-cp39-abi3-manylinux_2_28_x86_64
    ↪ .whl.metadata (5.5 kB)
Collecting shiboken6==6.9.0 (from pyside6)
  Using cached shiboken6-6.9.0-cp39-abi3-
    ↪ manylinux_2_28_x86_64.whl.metadata (2.7 kB)
Collecting PySide6-Essentials==6.9.0 (from pyside6)
  Using cached PySide6_Essentials-6.9.0-cp39-abi3-
    ↪ manylinux_2_28_x86_64.whl.metadata (3.9 kB)
Collecting PySide6-Addons==6.9.0 (from pyside6)
  Using cached PySide6_Addons-6.9.0-cp39-abi3-
    ↪ manylinux_2_28_x86_64.whl.metadata (4.2 kB)
Using cached PySide6-6.9.0-cp39-abi3-manylinux_2_28_x86_64.
  ↪ whl (558 kB)
Using cached PySide6_Addons-6.9.0-cp39-abi3-
  ↪ manylinux_2_28_x86_64.whl (166.3 MB)
Using cached PySide6_Essentials-6.9.0-cp39-abi3-
  ↪ manylinux_2_28_x86_64.whl (94.2 MB)
Using cached shiboken6-6.9.0-cp39-abi3-manylinux_2_28_x86_64
  ↪ .whl (206 kB)
Installing collected packages: shiboken6, PySide6-Essentials
  ↪ , PySide6-Addons, pyside6
Successfully installed PySide6-Addons-6.9.0 PySide6-
  ↪ Essentials-6.9.0 pyside6-6.9.0 shiboken6-6.9.0
```

pip installe alors les paquets dans un endroit isolé (en dehors du système), et nous pouvons maintenant travailler sur notre projet Python sans vous soucier des conflits de dépendances.

Une fois terminé de travailler avec cet environnement virtuel nous pouvons le désactiver :

```
(.venv)user@machine: $ deactivate
user@machine: $
```

Le prompt ne fait plus alors mention de l'environnement virtuel.

1.2 Intérêt des environnements virtuels

Techniquement, Python est installé avec deux répertoires `site-packages` ou `dist-packages` :

— `purelib/` qui ne contient que des modules écrits en code Python pur.
— `platlib/` qui contient des binaires qui ne sont pas écrits en Python pur, par exemple des fichiers `.dll`, `.so` ou `.pydist`.

Pour visualiser les chemins de ces répertoires sur le système, se rendre dans l'interpréteur

Python puis :

```
>>> import sysconfig
>>> sysconfig.get_path("purelib")
'/usr/local/lib/python3.13/dist-packages'
>>> sysconfig.get_path("platlib")
'/usr/local/lib/python3.13/dist-packages'
```

Et avec notre environnement virtuel activé :

```
>>> import sysconfig
>>> sysconfig.get_path("purelib")
'/venv/lib/python3.13/site-packages'
>>> sysconfig.get_path("platlib")
'/venv/lib/python3.13/site-packages'
```

L'intérêt des environnements virtuels se situe à plusieurs niveaux :

— Évite la pollution du système.
— Évite les conflits dépendances (un même paquet avec des versions différentes, une version spécifique à chaque projet développé).
— Minimiser les problèmes de reproductivité : avec un environnement virtuel distinct pour chaque projet, il sera plus facile de lire les exigences du projet à partir des dépendances épinglées, et ainsi de partager ces exigences avec d'autres collaborateurs du projet.
— Permet d'installer des paquets sans avoir besoin de droits administrateurs sur la machine, puisque ces paquets ne seront pas accessibles en dehors de l'environnement virtuel.

1.3 Structure d'un environnement virtuel

Une structure de répertoires

Un environnement virtuel Python est une structure de répertoires qui offre tout ce dont nous avons besoin pour exécuter un environnement Python léger mais isolé.

En créant un nouvel environnement virtuel à l'aide du module **venv**, Python crée une structure de répertoires autonome et copie ou établit des liens symboliques avec les fichiers exécutables de Python dans cette structure. Nous pouvons visualiser cette structure à l'aide de la commande `tree` (la sortie de cette commande est alors très longue).

Nous retrouvons les répertoires suivants :

bin/ : contient les fichiers exécutables de l'environnement virtuel. Les plus importants

sont l'interpréteur Python et l'exécutable **pip**. Le répertoire contient également des scripts d'activation pour l'environnement virtuel.

include/ : est un répertoire initialement vide que Python utilise pour inclure des fichiers d'en-tête en langage C pour les paquets installés et qui dépendent d'extensions en C.

lib/ : contient le répertoire `site-packages/`, répertoire qui est l'une des principales raisons de la création de l'environnement virtuel. C'est dans ce dossier que seront installés les paquets externes utilisés par l'environnement virtuel.

lib64/ : sur de nombreux systèmes Linux est un lien symbolique vers `lib/` pour des raisons de compatibilité.

Un répertoire `{nom}-{version}.dist-info/`, obtenu par défaut pour **pip**, contient des informations sur la distribution des paquets.

pyvenv.cfg : est un fichier crucial pour votre environnement virtuel (cf. infra).

Il y a trois parties essentielles pour un environnement virtuel :

— Une copie ou un lien symbolique du binaire Python
— Un fichier `pyenv.cfg`
— Un répertoire `site-packages`

Par défaut, **venv** n'installe que **pip**, qui est l'outil recommandé pour installer des paquets Python :

```
(venv)user@machine: $ python3 -m pip list
Package    Version
-------    -------
pip        25.0.1
```

Un environnement virtuel n'est finalement qu'une structure de répertoires qu'il est possible de supprimer et de recréer à tout moment.

Une installation Python isolée

Pour isoler l'environnement virtuel du système, **venv** reproduit la structure des répertoires qu'une installation standard de Python crée.

Le fichier `pyvenv.cfg` est un petit fichier qui contient quelques paires clé-valeur, paramètres essentiels pour faire fonctionner l'environnement virtuel :

```
home = /usr/bin
include-system-site-packages = false
version = 3.13.2
executable = /usr/bin/python3.13
command = /usr/bin/python3 -m venv /chemin/projet/venv
```

Même si l'environnement virtuel est isolé, nous pouvons toutefois accéder aux modules de la bibliothèque standard de Python car notre environnement virtuel réutilise les modules

intégrés de Python et de la bibliothèque standard de l'installation de Python utilisée pour créer l'environnement virtuel. Comme nous avons toujours besoin d'une installation Python existante pour créer notre environnement virtuel, **venv** opte pour réutiliser les modules de bibliothèque standard existants afin d'éviter les coûts liés à leur copie dans un nouvel environnement virtuel.

En plus des modules de la bibliothèque standard nous pouvons donner à notre environnement virtuel l'accès au `site-packages` de l'installation de base par un argument lorsque l'on crée l'environnement :

```
user@machine: $ python3 -m venv venv/ --system-site-packages
```

En ajoutant `–system-site-packages`, Python définit la valeur `include-system-site-packages` dans `pyvenv.cfg` à `true`. Ce paramètre signifie que nous pouvons utiliser n'importe quel paquet externe que nous avons installé sur le système de base Python comme si nous les avions installés dans notre environnement virtuel.

Cette connexion ne fonctionne que dans une seule direction, car les nouveaux paquets installés dans l'environnement virtuel ne se mélangeront pas avec les paquets présents. Python respectera la nature isolée des installations de l'environnement virtuel et les placera dans le répertoire `site-packages` propre à l'environnement virtuel.

1.4 Fonctionnement d'un environnement virtuel

Lorsque l'on crée un environnement virtuel à l'aide de **venv**, le module recrée la structure des fichiers et répertoires d'une installation standard de Python présente sur le système. Python copie également dans le répertoire l'exécutable Python avec lequel nous avons appelé **venv** :

```
venv/
|
|- bin/
|    |- Activate.ps1
|    |- activate
|    |- activate.csh
|    |- activate.fish
|    |- pip
|    |- pip3
|    |- pip3.13
|    |- python
|    |- python3
|    |- python3.13
|
|- include/
|
```

```
|- lib/
|   |
|   |- python3.13/
|       |
|       |- site-packages/
|
|- lib64/
|   |
|   |- python3.13/
|       |
|       |- site-packages/
|
|- pyvenv.cfg
```

Cette structure ressemble à celle que l'on retrouve au niveau du système d'exploitation. **venv** crée cette structure de répertoires pour s'assurer que Python fonctionnera comme prévu en isolation, sans avoir besoin d'appliquer des modifications supplémentaires.

L'interpréteur Python dans un environnement virtuel créé avec **venv** cherche d'abord un fichier `pyvenv.cfg`. Si ce fichier est trouvé et contient une clé `home`, il utilise cette clé pour définir deux variables :

sys.base_prefix : le chemin vers l'exécutable Python utilisé pour créer l'environnement virtuel.

sys.prefix : le répertoire contenant `pyvenv.cfg`

Si `pyvenv.cfg` n'est pas trouvé, l'interpréteur détermine qu'il n'est pas dans un environnement virtuel, et `sys.base_prefix` et `sys.prefix` pointent vers le même chemin.

Dans l'environnement virtuel :

```
>>> import sys
>>> sys.base_prefix
'/usr'
>>> sys.prefix
'/home/chemin/vers/venv'
```

En dehors de l'environnement virtuel :

```
>>> import sys
>>> sys.base_prefix
'/usr'
>>> sys.prefix
'/usr'
```

Ainsi, un environnement virtuel Python dans sa forme la plus simple n'est rien de plus qu'une copie ou un lien symbolique du binaire Python accompagné d'un fichier `pyvenv.cfg` et d'un répertoire `site-packages`.

Si ces deux variables ont des valeurs différentes, alors Python adapte où il va rechercher les modules : l'interpréteur Python de l'environnement virtuel utilise les modules de la bibliothèque standard de l'installation Python de base tout en pointant vers son propre répertoire interne `site-packages` pour installer et accéder aux paquets externes.

Le renvoi vers la bibliothèque standard permet d'obtenir un environnement virtuel Python léger, que nous pouvons rapidement créer puis supprimer lorsque nous n'en n'avons plus besoin. Pour ce faire, **venv** ne copie que les fichiers nécessaires.

Pour s'assurer que les scripts que nous voulons exécuter utilisent l'interpréteur Python dans notre environnement virtuel, **venv** modifie la variable d'environnement `PYTHONPATH` (voir les différences de résultat de l'instruction `sys.path` dans l'environnement virtuel et en dehors de cet environnement). Ce changement dans les paramètres de chemin de Python crée effectivement l'isolement des paquets externes dans l'environnement virtuel.

Pour lancer un interpréteur Python dans l'environnement virtuel, de la même manière que si nous l'avions activé au préalable :

```
user@machine: $ /home/chemin/vers/venv/bin/python
```

Pour vérifier que l'interpréteur pointe bien vers l'environnement virtuel et vers l'exécutable Python idoine :

```
>>> from sys import prefix, executable
>>> prefix
/home/chemin/vers/venv
>>> executable
/home/chemin/vers/venv/bin/python
```

Tant que nous fournissons le chemin d'accès à notre exécutable Python, nous n'avons pas besoin d'activer notre environnement virtuel pour profiter des avantages qu'il offre.

1.5 Personnaliser un environnement virtuel

Écraser un environnement virtuel et le remplacer par un autre

```
user@machine: $ python3 -m venv venv/
user@machine: $ python3 venv/bin/pip install requests
user@machine: $ python3 venv/bin/pip list
Package            Version
-------            -------
certifi            2025.1.31
charset-normalizer 3.4.1
idna               3.10
```

```
pip                 25.0.1
requests            2.32.3
urllib3             2.4.0
user@machine: $ python3 -m venv venv/ --clear
user@machine: $ python3 venv/bin/pip list
Package             Version
-------             -------
pip                 25.0.1
```

Si l'option `-clear` n'est pas précisée, et qu'un environnement virtuel du même nom existe, rien ne sera alors exécuté, et le premier environnement virtuel sera conservé.

Créer plusieurs environnements virtuels en une seule fois

Si un seul environnement virtuel ne suffit pas, il est possible d'en créer plusieurs distincts en une seule fois en indiquant plusieurs chemins d'accès à la commande :

```
user@machine: $ python3 -m venv venv/ /chemin/venv-copy
```

Il est tout à fait possible de créer autant d'environnements virtuels que de chemins indiqués (chemins séparés par un espace).

Mise à jour des dépendances de base

Lorsque l'on crée un environnement virtuel à l'aide de **venv** avec ses paramètres par défaut et que l'on installe ensuite un paquetage externe à l'aide de **pip**, nous pouvons rencontrer un message indiquant que l'installation de **pip** est obsolète. En fait **pip** se connecte à **PyPI**[1] et identifie également si **pip** lui-même est obsolète, et si tel est le cas la commande affiche l'avertissement d'obsolescence. Il est alors préférable de disposer de la dernière version de **pip** pour éviter les problèmes de sécurité ou les bogues qui pourraient subsister dans une version plus ancienne. Pour un environnement virtuel existant, nous pouvons suivre les instructions que **pip** imprime dans le terminal pour une mise à jour.

Pour créer un environnement virtuel en demandant une mise à jour automatique de **pip** :

```
user@machine: $ python3 -m venv venv/ --upgrade-deps
```

1. https://pypi.org/

Éviter d'installer *pip*

Dans la plupart des cas, nous voudrons que **pip** soit installé dans notre environnement virtuel car il nous servira pour installer des paquets externes provenant de **PyPI**. Cependant, si nous n'avons pas besoin de **pip**, nous pouvons utiliser `--without-pip` pour créer un environnement virtuel sans **pip**, ce qui ne créera que la structure des répertoires (structure plus légère qu'avec **pip** installé : quelques ko au lieu de plusieurs Mo).

```
user@machine: $ python3 -m venv venv/ --without pip
```

Inclure le `site-packages` du système

Pour cela on ajoutant l'option `--system-site-packages` lors de la création de l'environnement virtuel. Et tout paquet externe supplémentaire, sera ensuite placé dans le répertoire `site-packages` de l'environnement virtuel. Il faut toutefois garder à l'esprit que nous n'aurons qu'un accès en lecture au répertoire `site-packages` du système.

```
user@machine: $ python3 -m venv venv/ --system-site-packages
```

Dans le fichier `pyenv.cgf` nous trouvons alors la ligne suivante :

```
include-system-site-packages = true
```

Mettre à jour votre Python pour qu'il corresponde à celui du système

En construisant son environnement virtuel en utilisant des copies plutôt que des liens symboliques et que l'on souhaite ensuite mettre à jour la version de base de Python sur le système, nous pourrions rencontrer un problème de décalage de version avec les modules de la bibliothèque standard.

Le module **venv** offre une solution à ce problème. L'argument optionnel `--upgrade` permet de conserver le répertoire `site-packages` intact tout en mettant à jour les fichiers binaires avec les nouvelles versions du système :

```
user@machine: $ python3 -m venv venv/ --upgrade
```

1.6 Gestion des environnements virtuels

Où créer les répertoires d'environnement ?

Puisqu'un environnement virtuel Python n'est qu'une structure de répertoires, nous pouvons le placer n'importe où sur le système. Et il existe deux solutions, qui présentent chacune des avantages et des inconvénients, quant à l'endroit où créer les répertoires d'environnement virtuel :

— Dans chaque répertoire de projet individuel
— Dans un emplacement unique, par exemple dans un sous-répertoire du répertoire personnel

Avec la première solution, nous créons un nouvel environnement virtuel dans le répertoire racine du projet :

```
project_name/
|
|- venv/
|
|- src/
```

Le répertoire de l'environnement virtuel cohabite alors avec tout le code du projet. L'avantage de cette structure est que l'on sait d'emblée quel environnement virtuel appartient à quel projet et l'activation se réalise à l'aide d'un chemin relatif court.

Avec la seconde solution tous les environnements virtuels sont placés dans un seul répertoire :

```
name/
|
|- Desktop/
|
|- Documents/
|   |
|   |- projects/
|       |
|       |- django-project/
|       |
|       |- flask-project/
|       |
|       |- pandas-project/
|
|- venvs
|   |
|   |- django-venv/
```

```
|   |
|   |- flask-venv/
|   |
|   |- pandas-venv/
|
|- Autres_répertoires/
```

Si tous les environnements virtuels sont centralisés dans un même répertoire, il est nécessaire d'user d'un chemin absolu pour les activer.

A noter que la plupart des IDE offrent cette possibilité de choix [2].

Des objets jetables

Les environnements virtuels sont des structures de répertoires jetables. Il ne faut donc pas y ajouter manuellement du code ou des informations supplémentaires.Tout ce qui s'y trouve doit être géré par le gestionnaire de paquets.

Un environnement virtuel ne doit pas non plus être livré au contrôle de version. Les environnements virtuels ne sont pas des installations Python entièrement autonomes, il ne s'agit donc pas d'une application portable. De même, les environnements virtuels en production sont déconseillés [3].

Épingler les dépendances

Pour reproduire un environnement virtuel il nous faut décrire son contenu via un fichier spécifique : `requirements.txt`.

```
(.venv)user@machine: $ python3 -m pip freeze > requirements.txt
```

Ce fichier contient une liste des dépendances externes installées dans l'environnement virtuel. C'est à l'aide de ce fichier que nous saurons le récréer.

```
(.venv) user@machine: $ deactivate
user@machine: $ rm -Rf .venv
user@machine: $ python3 -m venv .new-venv/
user@machine: $ source .new-venv/bin/activate
(.new-venv)user@machine: $ python3 -m pip install -r requiremen
    ts.txt
```

2. Pour **Visual Studio Code** : `https://code.visualstudio.com/docs/python/environments`. Pour **PyCharm** : `https://www.jetbrains.com/help/pycharm/creating-virtual-environment.html`

3. `https://realpython.com/python-virtual-environments-a-primer/#avoid-virtual-environments-in-production`

En soumettant le fichier `requirements.txt` au contrôle de version, nous livrons le code du projet ainsi que la recette qui permettra de recréer le même environnement virtuel.

Cependant ce fichier contient des limites :

— Il ne contient pas d'informations sur la version de Python utilisée lors de la création de l'environnement virtuel.
— Peuvent ne pas être inclues les informations sur la version des sous-dépendances des dépendances.

C'est alors que d'autres outils de gestion des dépendances seront nécessaires. Nous aborderons l'outil **Poetry** dans un prochain chapitre et qui sait répondre à ces limites.

Nous venons d'explorer les fondamentaux des environnements virtuels en Python et l'utilisation du module intégré **venv**. Nous avons découvert comment créer et gérer des environnements isolés pour nos projets, assurant ainsi une gestion propre et organisée des dépendances. Si **venv** offre une solution simple et efficace, dans le chapitre qui suit nous allons voir un outil encore plus puissant et flexible : **virtualenv**.

2

virtualenv
Doper ses environnements virtuels

> *“Le virtuel ne s’oppose pas au réel, mais seulement à l’actuel. Le virtuel possède une pleine réalité, en tant que virtuel.”*
> — Gilles Deleuze

virtualenv[1] est un outil spécialement conçu pour créer des environnements Python isolés. Il s’agit d’un sur-ensemble de **venv**, offrant des fonctionnalités supplémentaires.

Installation sur **Debian GNU/Linux** :

```
user@machine: # aptitude install python3-virtualenv
```

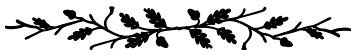
Création et activation de l’environnement virtuel (depuis le répertoire du projet) :

```
user@machine: $ virtualenv .venv/
created virtual environment CPython3.13.3.final.0-64 in 288
  ↪ ms
  creator CPython3Posix(dest=/home/user/mon_projet/.venv,
    ↪ clear=False, no_vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, via=copy,
    ↪ app_data_dir=/home/user/.local/share/virtualenv)
    added seed packages: pip==25.1.1
  activators BashActivator,CShellActivator,FishActivator,
    ↪ NushellActivator,PowerShellActivator,PythonActivator
user@machine: $ source .venv/bin/activate
(venv) user@machine: $
```

1. https://virtualenv.pypa.io/en/latest/user_guide.html

virtualenv crée l'environnement isolé beaucoup plus rapidement que le module **venv** intégré, ce qui est possible parce que l'outil met en cache les données d'application spécifiques à la plate-forme, qu'il peut lire rapidement.

...A poursuivre...



3

pip
Maîtriser l'art de la gestion des paquets Python

> *"La première règle de l'écologie, c'est que les éléments sont tous liés les uns aux autres."*
> — Barry Commoner

Dans le vaste écosystème de Python, **pip**[1] se révèle comme étant un outil indispensable. En tant que gestionnaire de paquets par défaut, pip permet d'installer, de mettre à jour et de supprimer des bibliothèques Python de manière simple et efficace. Que l'on travaille sur un petit projet personnel ou sur une application complexe, **pip** offre la flexibilité et la puissance nécessaires pour gérer les dépendances des projets avec précision.

L'un des principaux avantages de **pip** est sa simplicité d'utilisation. Avec des commandes intuitives, nous pouvons installer des paquets en quelques secondes, explorer de nouvelles bibliothèques. De plus, **pip** est compatible avec une multitude de dépôts de paquets, offrant ainsi un accès à une vaste gamme de bibliothèques et d'outils développés par la communauté Python.

Dans ce chapitre, j'aborderai les fonctionnalités essentielles de **pip** et comment tirer le meilleur parti de cet outil pour optimiser le flux de travail de développement.

3.1 Utilisation dans un environnement virtuel

Pour éviter d'installer des paquets directement dans l'installation Python du système, il est recommandé d'utiliser un environnement virtuel. Tous les paquets utilisés dans cet

1. Documentation officielle : `https://pip.pypa.io/en/stable/`

environnement seront alors indépendants de l'interpréteur du système.

Cela présente trois avantages principaux :
— L'assurance d'utiliser la bonne version de Python pour le projet en cours.
— L'assurance de se référer à la bonne instance de **pip**.
— L'utilisation d'une version de paquetage spécifique pour un projet sans affecter les autres projets.

3.2 Installer des paquets avec *pip*

Le langage Python dispose d'une bibliothèque standard, mais également de paquets publiés sur le **Python Package Index**[2], également connu sous le nom de **PyPI**, qui héberge une vaste collection de paquets, y compris des cadres (*framework*) de développement, des outils et des bibliothèques.

Utilisation de *Python Package Index*

Une fois l'environnement virtuel créé et activé, toutes les commandes **pip** alors exécutées le seront dans l'environnement virtuel.

Pour installer des paquets, **pip** fournit la commande `install` :

```
(.venv)user@machine: $ python3 -m pip install nom_paquet
```

La commande **pip** recherche le paquet dans **PyPI**, résout ses dépendances (dépendances répertoriées dans les métadonnées du paquet) et installe tout ce qui est nécessaire dans l'environnement virtuel actuel. C'est toujours la dernière version du paquetage qui est installée.

Il est également possible d'installer plusieurs paquets en une seule commande :

```
(.venv)user@machine: $ python3 -m pip install nom_paquet1 nom_p
    aquet2
```

La commande `list` permet d'afficher les paquets installés dans l'environnement, ainsi que leurs numéros de version. Exemple :

```
(.venv)user@machine: $ python3 -m pip list
Package            Version
------------------ --------
jedi               0.19.2
packaging          24.2
parso              0.8.4
```

2. https://pypi.org/

```
pip                25.0
pudb               2024.1.3
Pygments           2.19.1
typing_extensions  4.13.1
urwid              2.6.16
urwid_readline     0.15.1
wcwidth            0.2.13
```

Consulter les informations (métadonnées d'un paquet) à l'aide de la commande *show* :

```
(.venv)user@machine: $ python3 -m pip show jedi
Name: jedi
Version: 0.19.2
Summary: An autocompletion tool for Python that can be used
   ↪ for text editors.
Home-page: https://github.com/davidhalter/jedi
Author: David Halter
Author-email: davidhalter88@gmail.com
License: MIT
Location: /home/user/projet/.virtual_py/lib/python3.13/site-
   ↪ packages
Requires: parso
Required-by: pudb
```

Utilisation d'un *package index* personnalisé

Par défaut, **pip** utilise **PyPI** pour rechercher des paquets, mais il est également possible de définir un index personnalisé (disponible par exemple sur un serveur privé). Un *package index* personnalisé doit être conforme avec la norme définie dans la `PEP 503`[3] pour fonctionner avec **pip**. Tout index personnalisé qui suit la même API peut être ciblé avec l'option `--index-url` ou `-i`.

Par exemple, pour installer l'outil **rptree** à partir de l'index des paquets `TestPyPI`[4] :

```
(.venv)user@machine: $ python3 -m pip install -i https://test.
   ↪ pypi.org/simple/ rptree
Looking in indexes: https://test.pypi.org/simple/
Collecting rptree
  Using cached https://test-files.pythonhosted.org/packages
     ↪ /0d/bd/83
     ↪ a44ae145bd05b4b667aca5be7b5a7cf61d0d26577c0c93f2a2d0d
     ↪  c2c59/rptree-0.1.0-py3-none-any.whl.metadata (1.5 kB
     ↪ )
```

3. API de dépôt simple : `https://peps.python.org/pep-0503/`
4. `https://test.pypi.org/`

```
Using cached https://test-files.pythonhosted.org/packages/0d
  ↪ /bd/83
  ↪ a44ae145bd05b4b667aca5be7b5a7cf61d0d26577c0c93f2a2d0dc2
  ↪  c59/rptree-0.1.0-py3-none-any.whl (4.4 kB)
Installing collected packages: rptree
Successfully installed rptree-0.1.0
```

Installation de paquets depuis des dépôts *Git*

pip fournit également l'option d'installer des paquets à partir d'un dépôt **Git**.

```
(.venv)user@machine: $ python3 -m pip install git+https://
  ↪ github.com/realpython/rptree
Collecting git+https://github.com/realpython/rptree
  Cloning https://github.com/realpython/rptree to /tmp/pip-
    ↪ req-build-ugz_ee9b
  Running command git clone --filter=blob:none --quiet https
    ↪ ://github.com/realpython/rptree /tmp/pip-req-build-
    ↪ ugz_ee9b
  Resolved https://github.com/realpython/rptree to commit 08
    ↪ dcb019aea6e3e326c39be9741e52adb77a2c19
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: rptree
  Building wheel for rptree (pyproject.toml) ... done
  Created wheel for rptree: filename=rptree-0.1.1-py3-none-
    ↪ any.whl size=5098 sha256=
    ↪ ea7c230770b418e6c83cc3ab48525c934051303713ea1fd195110
    ↪  9e1c41a4451
  Stored in directory: /tmp/pip-ephem-wheel-cache-td03c1n1/
    ↪ wheels/bd/b3/32/3714
    ↪ ad3461eca52f1805ec11248c17c60e77c1ae8746f79179
Successfully built rptree
Installing collected packages: rptree
Successfully installed rptree-0.1.1
```

Le schéma `git+https` pointe vers le dépôt `Git` qui contient un paquet installable.

Pour faire appel à d'autres dépôts **VCS** voir la page de documentation dédiée[5].

5. https://pip.pypa.io/en/stable/topics/vcs-support/

3.3 Le fichier d'exigence

Les dépendances externes sont aussi appelées exigences. Souvent, les projets Python épinglent leurs exigences dans un fichier appelé `requirements.txt` ou similaire. Le format de fichier *requirements* permet de spécifier précisément quels paquets et versions doivent être installés.

pip dispose d'une commande de gel (`freeze`) qui affiche les paquets installés au format requis, et qui permet également de rediriger la sortie vers un fichier afin de générer un fichier d'exigences :

```
(.venv)user@machine: $ python3 -m pip freeze > requirements.txt
```

Exemple de fichier `requirements.txt` que l'on peut obtenir après avoir installé les paquets **pyside6** et **requests** :

```
certifi==2025.4.26
charset-normalizer==3.4.2
idna==3.10
PySide6==6.9.0
PySide6_Addons==6.9.0
PySide6_Essentials==6.9.0
requests==2.32.3
shiboken6==6.9.0
urllib3==2.4.0
```

Ce fichier peut éventuellement porter un autre nom, cependant une convention largement adoptée consiste à lui donner le nom `requirements.txt`.

Nous pouvons demander à **pip** d'installer les paquets listés dans `requirements.txt` dans un nouvel environnement :

```
user@machine: $ python3 -m venv .venv/
user@machine: $ source .venv/bin/activate
(.venv)user@machine: $ python3 -m pip install -r requirements
    .txt
user@machine: $ python3 -m pip list
Collecting certifi==2025.4.26 (from -r requirements.txt (
  ↪ line 1))
  Using cached certifi-2025.4.26-py3-none-any.whl.metadata
    ↪ (2.5 kB)
Collecting charset-normalizer==3.4.2 (from -r requirements.
  ↪ txt (line 2))
  Using cached charset_normalizer-3.4.2-cp313-cp313-
    ↪ manylinux_2_17_x86_64.manylinux2014_x86_64.whl.
    ↪ metadata (35 kB)
```

```
Collecting idna==3.10 (from -r requirements.txt (line 3))
  Using cached idna-3.10-py3-none-any.whl.metadata (10 kB)
Collecting PySide6==6.9.0 (from -r requirements.txt (line 4)
   ↪ )
  Using cached PySide6-6.9.0-cp39-abi3-manylinux_2_28_x86_64
     ↪ .whl.metadata (5.5 kB)
Collecting PySide6_Addons==6.9.0 (from -r requirements.txt (
   ↪ line 5))
  Using cached PySide6_Addons-6.9.0-cp39-abi3-
     ↪ manylinux_2_28_x86_64.whl.metadata (4.2 kB)
Collecting PySide6_Essentials==6.9.0 (from -r requirements.
   ↪ txt (line 6))
  Using cached PySide6_Essentials-6.9.0-cp39-abi3-
     ↪ manylinux_2_28_x86_64.whl.metadata (3.9 kB)
Collecting requests==2.32.3 (from -r requirements.txt (line
   ↪ 7))
  Using cached requests-2.32.3-py3-none-any.whl.metadata
     ↪ (4.6 kB)
Collecting shiboken6==6.9.0 (from -r requirements.txt (line
   ↪ 8))
  Using cached shiboken6-6.9.0-cp39-abi3-
     ↪ manylinux_2_28_x86_64.whl.metadata (2.7 kB)
Collecting urllib3==2.4.0 (from -r requirements.txt (line 9)
   ↪ )
  Using cached urllib3-2.4.0-py3-none-any.whl.metadata (6.5
     ↪ kB)
Using cached certifi-2025.4.26-py3-none-any.whl (159 kB)
Using cached charset_normalizer-3.4.2-cp313-cp313-
   ↪ manylinux_2_17_x86_64.manylinux2014_x86_64.whl (148 kB)
Using cached idna-3.10-py3-none-any.whl (70 kB)
Using cached PySide6-6.9.0-cp39-abi3-manylinux_2_28_x86_64.
   ↪ whl (558 kB)
Using cached PySide6_Addons-6.9.0-cp39-abi3-
   ↪ manylinux_2_28_x86_64.whl (166.3 MB)
Using cached PySide6_Essentials-6.9.0-cp39-abi3-
   ↪ manylinux_2_28_x86_64.whl (94.2 MB)
Using cached requests-2.32.3-py3-none-any.whl (64 kB)
Using cached urllib3-2.4.0-py3-none-any.whl (128 kB)
Using cached shiboken6-6.9.0-cp39-abi3-manylinux_2_28_x86_64
   ↪ .whl (206 kB)
Installing collected packages: urllib3, shiboken6, idna,
   ↪ charset-normalizer, certifi, requests,
   ↪ PySide6_Essentials, PySide6_Addons, PySide6
Successfully installed PySide6-6.9.0 PySide6_Addons-6.9.0
   ↪ PySide6_Essentials-6.9.0 certifi-2025.4.26 charset-
   ↪ normalizer-3.4.2 idna-3.10 requests-2.32.3 shiboken6
   ↪ -6.9.0 urllib3-2.4.0
```

Nous pouvons désormais soumettre `requirements.txt` dans un système de contrôle de version comme **Git** et l'utiliser pour répliquer le même environnement sur d'autres machines.

Formats du fichier d'exigence

Chaque ligne du fichier d'exigence indique quelque chose à installer, ou des arguments pour `pip install`.

Il est possible de spécifier des exigences sous forme de noms simples :

```
certifi
charset-normalizer
```

Il est toutefois possible de spécifier les versions de dépendances à l'aide d'opérateurs de comparaison qui donnent une certaine flexibilité pour assurer la mise à jour des paquets tout en définissant la version de base d'un paquet.

Pour cela, éditer `requirements.txt` afin de transformer les opérateurs d'égalité (==) en plus grand ou égal (>=) :

```
certifi==2025.4.26
charset-normalizer==3.4.2
idna==3.10
PySide6>=6.9.0
PySide6_Addons>=6.9.0
# ...etc...
```

A noter le symbole # qui permet d'insérer des commentaires.

Ensuite, nous pouvons mettre à niveau les packages en exécutant la commande `install` avec le commutateur `-upgrade` ou le raccourci `-U` :

```
(.venv)user@machine: $ python3 -m pip install -U -r requiremen
    ts.txt
```

Si une nouvelle version est disponible pour un paquet répertorié, le paquet sera mis à niveau.

Nous pouvons également modifier le fichier des exigences pour empêcher l'installation d'une version donnée ou ultérieure :

```
PySide6>=6.9.0, <6.9.5
PySide6_Addons>=6.9.0, <6.9.5
```

Page de la documentation officielle permettant de visualiser divers exemples de formats de fichier d'exigence : `https://pip.pypa.io/en/stable/cli/pip_install/#pip-install-examples`.

Séparation des dépendances de production et de développement

Pour cela on va créer un second fichier d'exigences, `requirementsi_dev.txt`, pour lister les outils supplémentaires (par exemple **pytest** permettant de mettre en place un environnement de développement :

```
pytest>=x.y.z
```

Avoir deux fichiers de configuration exigera d'utiliser **pip** pour installer les deux : `requirements.txt` et `requirements_dev.txt`. Cependant, **pip** permet de spécifier des paramètres supplémentaires dans un fichier d'exigences, ainsi, il est possible de modifier `requirements_dev.txt` pour installer également les exigences à partir du fichier production `requirements.txt` :

```
-r requirements.txt
pytest>=x.y.z
```

Maintenant, dans l'environnement de développement, il suffit d'exécuter cette seule commande pour installer toutes les exigences :

```
(.venv)user@machine: $ python3 -m pip install -r requirements
    _dev.txt
```

Dans un environnement de production, il suffit uniquement d'installer les exigences de production, soit d'utiliser le fichier `requirements.txt`.

Geler les exigences pour la production

Une fois le projet prêt pour la publication, il nous suffira de créer le fichier `requirement s_lock.txt`.

3.4 Désinstaller des paquets

Lorsque que l'on installe un paquet il arrive que des dépendances (d'autres paquets) soient installées. Plus on installe de paquets, plus il y a de chances que plusieurs paquets aient des dépendances communes. C'est là que la commande `show` s'avère utile. Ainsi, avant de désinstaller un paquet, on s'assure d'exécuter la commande `show` pour ce paquet.

Si le champ `Required-by` est vide, cela signifie qu'il n'y a aucune interdépendance avec un autre paquet. Par contre il faut également lancer la commande `show` sur les autres dépendances pour procéder à la même vérification. Une fois vérifié l'ordre des dépendances des paquets que nous souhaitons désinstaller, nous pouvez les supprimer en utilisant la commande `uninstall` :

```
(.venv)user@machine: $ python3 -m pip uninstall paquet
```

La commande de désinstallation montre les fichiers qui seront supprimés et demande confirmation. Il est possible de passer outre cette demande de confirmation :

```
(.venv)user@machine: $ python3 -m pip uninstall paquet -y
```

Supprimer plusieurs paquets en même temps en passant toujours outre la demande de confirmation :

```
(.venv)user@machine: $ python3 -m pip uninstall -y paquet1 paq
    uet2 paquet3
```

Il est également possible de désinstaller tous les packages répertoriés dans le fichier des exigences en fournissant l'option `-r requirements.txt` :

```
(.venv)user@machine: $ python3 -m pip uninstall -r requirements.
    txt -y
```

Remarque : en travaillant dans un environnement virtuel, il peut être moins compliqué de supprimer l'environnement virtuel et d'en créer un nouveau. Ensuite, nous installerons les paquets dont nous avons besoin au lieu d'essayer de désinstaller les paquets dont nous n'avons pas besoin.

Utiliser `pip uninstall` est un bon moyen de désinstaller un paquet du système si celui-ci a été installé accidentellement.

Nous venons d'explorer les multiples facettes de **pip**, en découvrant comment installer, mettre à jour et gérer les dépendances de nos projets avec facilité et précision. Cependant, la gestion des environnements de développement ne s'arrête pas là. Pour aller plus loin et optimiser encore davantage le flux de travail, il est essentiel de maîtriser les outils qui permettent de gérer plusieurs versions de Python sur un même système. C'est là que **pyenv** entre en jeu.

4

pyenv
Simplifier la gestion des versions de Python

> *“Sois comme l’eau : elle s’adapte à toute forme sans jamais perdre sa nature.”*
> — Citation taoïste

À mesure que l’on progresse dans la pratique du langage Python, une réalité s’impose : tous les projets ne parlent pas le même dialecte. Certains réclament une version ancienne, d’autres tirent parti des nouveautés les plus récentes. Installer plusieurs versions de Python sur une même machine peut alors devenir source de confusion, voire de conflit.

C’est ici qu’intervient **pyenv**, un outil discret mais redoutablement efficace, qui permet de jongler aisément entre les versions de Python. À la manière de l’eau qui épouse la forme du vase sans jamais perdre sa nature, **pyenv** s’adapte à chaque projet, chaque environnement, sans rien imposer au système global.

Ce chapitre nous guidera pas à pas dans la découverte et l’utilisation de **pyenv** : de son installation à sa maîtrise au quotidien. L’objectif n’est pas seulement de fournir un outil de plus, mais de proposer une nouvelle manière d’interagir avec notre environnement de développement — plus souple, plus propre, et infiniment plus adaptée.

Page **GitHub** du projet **pyenv** : `https://github.com/pyenv/pyenv`

4.1 Installer *pyenv*

Avant d’installer **pyenv** lui-même, nous aurons besoin de quelques dépendances spécifiques à notre système d’exploitation. Ces dépendances sont principalement des utilitaires de développement écrits en **C** et sont nécessaires parce que **pyenv** installe Python en le

construisant à partir des sources.

Les dépendances de construction

Les dépendances de construction varient selon la plate-forme. Sur **Debian GNU/Linux** il nous faudra installer les dépendances suivantes :

```
user@machine: # apt install -y make build-essential libssl-dev
  ↪ zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev
  ↪ wget curl llvm libncurses5-dev libncursesw5-dev xz-
  ↪ utils tk-dev libffi-dev liblzma-dev python3-openssl
```

Utiliser le programme d'installation de *pyenv*

Après avoir installé les dépendances, nous sommes prêts à installer **pyenv** lui-même. Il est recommandé d'utiliser le projet **pyenv-installer**[1].

```
$ curl https://pyenv.run | bash
```

Ceci installera **pyenv** ainsi que quelques *plugins* utiles :

pyenv : L'application **pyenv** actuelle
pyenv-virtualenv : *Plugin* pour **pyenv** et les environnements virtuels
pyenv-update : *Plugin* pour mettre à jour **pyenv**
pyenv-doctor : *Plugin* pour vérifier que **pyenv** et les dépendances sont installés
pyenv-which-ext : *Plugin* pour rechercher automatiquement les commandes système

Pour utiliser **pyenv** avec **zsh** :

```
$ echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc
$ echo '[[ -d $PYENV_ROOT/bin ]] && export PATH="$PYENV_ROOT/bin:$PATH"
    ' >> ~/.zshrc
$ echo 'eval "$(pyenv init - zsh)"' >> ~/.zshrc
```

Puis relancer le *shell* :

```
$ exec "\$SHELL"
```

Note de désinstallation : saisir la commande `\$ rm -Rf ~/.pyenv`, puis supprimer les lignes ajoutées au fichier `.zshrc`. Et enfin, relancer le *shell*.

1. https://github.com/pyenv/pyenv-installer

4.2 Utiliser *pyenv* pour installer *Python*

Pour visualiser toutes les versions disponibles de **CPython** :

```
$ pyenv install --list | grep " 3\.[678]"  # de python 3.6 à 3.8
$ pyenv install --list | grep " 3\.1[23]"  # de python 3.12 à 3.13
```

Sortie pour 3.12 à 3.13 :

```
3.12.0
3.12-dev
3.12.1
3.12.2
3.12.3
3.12.4
3.12.5
3.12.6
3.12.7
3.12.8
3.12.9
3.12.10
3.13.0
3.13.0t
3.13-dev
3.13t-dev
3.13.1
3.13.1t
3.13.2
3.13.2t
3.13.3
3.13.3t
```

De même, pour voir toutes les versions de **Jython** :

```
$ pyenv install --list | grep "jython"
```

Pour visualiser tout ce qu'il est possible d'installer :

```
$ pyenv install --list
```

Installer une version de Python :

```
user@machine: $ pyenv install -v 3.14.0a7
[...]
```

```
Installing collected packages: pip
  WARNING: The scripts pip3 and pip3.14 are installed in '/
    ↪ home/utilisateur/.pyenv/versions/3.14.0a7/bin' which
    ↪ is not on PATH.
  Consider adding this directory to PATH or, if you prefer
    ↪ to suppress this warning, use --no-warn-script-
    ↪ location.
Successfully installed pip-25.0.1
/tmp/python-build.20250507123332.8659 ~
~
Installed Python-3.14.0a7 to /home/utilisateur/.pyenv/
  ↪ versions/3.14.0a7
```

`-v` propose le mode `verbose`. A noter que l'installation prendra environ 340 Mo d'espace disque.

Désinstallation :

```
user@machine: $ rm -rf ~/.pyenv/versions/3.14.0a7
```

ou bien :

```
user@machine: $ pyenv uninstall 3.14.0a7
```

Emplacement de l'installation

pyenv fonctionne en construisant Python à partir des sources. Chaque version installée se trouve dans le répertoire racine de **pyenv**. Pour visualiser les versions installées :

```
user@machine: $ ls ~/.pyenv/versions/
3.12.10  3.14.0a7
```

Utilisation d'une de ces versions

Visualiser les versions dont on dispose :

```
user@machine: $ pyenv versions
* system (set by /home/utilisateur/.pyenv/version)
  3.12.10
  3.14.0a7
```

L'astérisque (*) indique que la version système de Python est actuellement active. Ce paramètre est défini par un fichier situé dans le répertoire racine de **pyenv**. Cela signifie que, par défaut, c'est toujours le système Python qui est utilisé.

```
user@machine: $ python --version  # ou -V
Python 3.13.3
user@machine: $ which python  # Confirmation avec 'which'
/home/utilisateur/.pyenv/shims/python
```

C'est ainsi que **pyenv** fonctionne. **pyenv** s'insère dans le **PATH** et, du point de vue du système d'exploitation, c'est l'exécutable qui est appelé. Pour voir le chemin réel :

```
user@machine: $ pyenv which python
pyenv: python: command not found

The 'python' command exists in these Python versions:
  3.12.10
  3.14.0a7

Note: See 'pyenv help global' for tips on allowing both
      python2 and python3 to be found.
user@machine: $ pyenv which python3
/usr/bin/python3
```

Si on souhaite utiliser la version **3.14.0a7** :

```
user@machine: $ pyenv global 3.14.0a7
user@machine: $ python -V
Python 3.14.0a7
user@machine: $ pyenv versions
  system
  3.12.10
* 3.14.0a7 (set by /home/utilisateur/.pyenv/version)
```

Un bon moyen de s'assurer que la cette version choisie de Python fonctionne correctement est d'exécuter la suite de tests intégrée (Cela prend beaucoup de temps) :

```
user@machine: $ python -m test
```

Pour revenir à la version système :

```
user@machine: $ pyenv global system
```

4.3 Les commandes de *pyenv*

Pour une liste complète des commandes :

```
user@machine: $ pyenv commands
activate
commands
completions
deactivate
doctor
exec
global
help
hooks
init
install
latest
local
prefix
rehash
root
shell
shims
uninstall
update
--version
version
version-file
version-file-read
version-file-write
version-name
version-origin
versions
virtualenv
virtualenv-delete
virtualenv-init
virtualenv-prefix
virtualenvs
whence
which
```

Chaque commande dispose d'un drapeau `-help` qui donnera des informations plus détaillées.

```
user@machine: $ pyenv global --help
Usage: pyenv global <version> <version2> <..>
```

```
Sets the global Python version(s). You can override the
   ↪ global version at
any time by setting a directory-specific version with 'pyenv
   ↪  local'
or by setting the 'PYENV_VERSION' environment variable.

<version> can be specified multiple times and should be a
   ↪ version
tag known to pyenv.  The special version string 'system'
   ↪ will use
your default system Python.  Run 'pyenv versions' for a list
   ↪  of
available Python versions.

Example: To enable the python2.7 and python3.7 shims to find
   ↪  their
         respective executables you could set both versions
            ↪ with:

'pyenv global 3.7.0 2.7.15'
```

Voyons quelques commandes parmi les plus utilisées.

install

Pour télécharger et installer Python.

Drapeau	Description
`-l` ou `-list`	Liste toutes les versions de Python disponible pour l'installation
`-g` ou `-debug`	Construit une version de débogage de Python
`-v` ou `-verbose`	Mode verbeux : impression de l'état de la compilation sur **stdout**

TABLE 4.1 – Avec la commande `install`

versions

Affiche toutes les versions de Python actuellement installées (comme vu supra).

Pour ne visualiser que la version active :

```
user@machine: $ pyenv version
system (set by /home/utilisateur/.pyenv/version)
```

which

Cette commande permet de voir le chemin complet de l'exécutable que **pyenv** utilisera. Par exemple, si l'on souhaite voir où **pip** est installé :

```
user@machine: $ pyenv which pip
/home/utilisateur/.pyenv/versions/3.14.0a7/bin/pip
```

global

Définir la version globale de Python :

```
user@machine: $ pyenv global 3.12.10
user@machine: $ python3
Python 3.12.10 (main, May  7 2025, 16:29:42) [GCC 14.2.0] on
   ↪  linux
Type "help", "copyright", "credits" or "license" for more
   ↪ information.
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
user@machine: $ pyenv global system
user@machine: $ python3
Python 3.13.3 (main, Apr 10 2025, 21:38:51) [GCC 14.2.0] on
   ↪ linux
Type "help", "copyright", "credits" or "license" for more
   ↪ information.
>>> exit
$
```

local

La commande `local` est souvent utilisée pour définir une version de Python spécifique à une application. Nous pouvons l'utiliser pour définir la version **2.7.15** :

```
user@machine: $ pyenv local 2.7.15
```

Cette commande crée un fichier `.python-version` dans le répertoire courant. Si **pyenv** est actif dans l'environnement, ce fichier activera automatiquement cette version.

shell

La commande `shell` est utilisée pour définir une version de Python spécifique au *shell*. Par exemple, pour tester la version **3.8-dev** de Python :

```
user@machine: $ pyenv shell 3.8-dev
```

Cette commande active la version spécifiée par la variable d'environnement **PYENV_VERSION**. Cette commande écrase toutes les applications ou tous les paramètres globaux. Pour désactiver la version, nous pouvons utiliser le drapeau `-unset`.

4.4 Spécifier la version de Python

L'une des parties les plus déroutantes de **pyenv** est la façon dont la commande python est résolue et quelles commandes peuvent être utilisées pour la modifier. Comme mentionné dans les commandes, il y a trois façons de modifier la version de Python utilisée. L'ordre de résolution des commandes ressemble un peu à ceci :

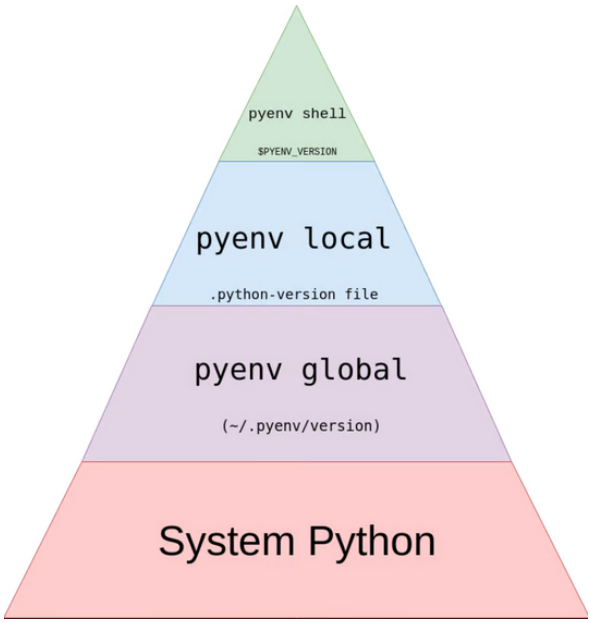


FIGURE 4.1 – Ordre de résolution des commandes **pyenv**

Voyons un exemple rapide :

```
user@machine: $ pyenv versions
* system (set by /home/utilisateur/.pyenv/version)
  3.12.10
  3.14.0a7
```

Ici, c'est le système Python qui est utilisé.

```
user@machine: $ pyenv global 3.14.0a7
$ pyenv versions
  system
  3.12.10
* 3.14.0a7 (set by /home/utilisateur/.pyenv/version)
```

pyenv utilise maintenant **3.6.8** comme version de **Python**. Il indique même l'emplacement du fichier qu'il a trouvé. Ce fichier existe effectivement :

```
user@machine: $ cat ~/.pyenv/version
3.14.0a7
```

Maintenant, créons un fichier **.python-version** avec `local` :

```
user@machine: $ pyenv local 3.12.10
user@machine: $ pyenv versions
  system
* 3.12.10 (set by /home/utilisateur/.python-version)
  3.14.0a7
user@machine: $ cat .python-version
3.12.10
```

pyenv indique comment il doit résoudre la commande python. Cette fois, elle provient de **/.python-version**. A Noter que la recherche de **.python-version** est récursive.

```
user@machine: $ pyenv shell 3.14.0a7
user@machine: $ pyenv versions
  system
  3.12.10
* 3.14.0a7 (set by PYENV_VERSION environment variable)
```

Tout ce que cela a fait, c'est définir la variable d'environnement `$PYENV_VERSION` :

```
user@machine: $ echo $PYENV_VERSION
3.14.0a7
```

4.5 Environnement virtuel et *pyenv*

pyenv allié à un environnement virtuel est un mariage parfait. **pyenv** dispose d'un *plugin* appelé **pyenv-virtualenv** qui permet de travailler avec plusieurs versions de Python et plusieurs environnements virtuels en un clin d'œil.

pyenv gère plusieurs versions de Python.

virtualenv/venv gère les environnements virtuels pour une version spécifique de Python.

pyenv-virtualenv gère les environnements virtuels pour différentes versions de Python.

Création d'un environnement virtuel

```
user@machine: $ pyenv virtualenv <version_python> <nom_envir
    onnement>
```

Une bonne pratique consiste à nommer les environnements du même nom que le projet. Par exemple, en travaillant sur `moni_projet` développé avec **Python 3.6.8** :

```
user@machine: $ pyenv virtualenv 3.6.8 mon_projet
```

Activation

```
user@machine: $ pyenv local mon_projet
```

Cela crée un fichier **.python-version** dans le répertoire de travail actuel et l'environnement sera automatiquement activé.

Vérification :

```
user@machine: $ pyenv which python
/home/utilisateur/.pyenv/versions/my_project/bin/python
```

Une nouvelle version a été créée sous le nom de `my_project` et l'exécutable python pointe vers cette version. En regardant n'importe quel exécutable fourni par cet environnement, nous verrons la même chose. Prenons, par exemple, **pip** :

```
user@machine: $ pyenv which pip
/home/utilisateur/.pyenv/versions/my_project/bin/pip
```

Activer / Désactiver :

```
user@machine: $ pyenv activate <nom_environnement>
user@machine: $ pyenv deactivate
```

4.6 Travailler avec plusieurs environnements

Supposons ces diverses versions de Python installées :

```
user@machine: $ pyenv versions
* system (set by /home/krystof/.pyenv/version)
  3.11.12
  3.14.0a7
```

Par défaut, c'est le système Python qui est utilisé.

Nous souhaitons maintenant travailler sur deux projets différents :
— **projet_1**, qui supporte **Python 3.11.12**.
— **projet_2**, qui supporte **Python 3.14.0a7**.
Créons un environnement virtuel pour chaque projet :

```
user@machine: $ mkdir projet_1 projet_2
user@machine: $ cd projet_1
user@machine: $ pyenv which python3
/usr/bin/python3
user@machine: $ pyenv virtualenv 3.11.12 projet_01
user@machine: $ pyenv local projet_01
user@machine: $ python3 -V
Python 3.11.12
user@machine: $ cd ../projet_2
user@machine: $ pyenv which python3  # En changeant de ré
   ↪ pertoire on revient au système Python
/usr/bin/python3
user@machine: $ pyenv virtualenv 3.14.0a7 projet_02
user@machine: $ pyenv local projet_02
user@machine: $ python3 -V
Python 3.14.0a7
```

Plus besoin de se rappeler d'activer les environnements : en passant d'un projet à l'autre, **pyenv** se charge d'activer automatiquement les bonnes versions de Python et les bons environnements virtuels

Nous venons d'explorer les multiples facettes de **pyenv**, un outil puissant qui permet de gérer efficacement différentes versions de Python. Grâce à **pyenv**, nous pouvons désormais basculer entre les versions de Python avec une facilité déconcertante, optimisant ainsi nos environnements de développement pour répondre aux besoins spécifiques de chaque projet.

Cependant, la gestion des dépendances et des paquets reste un aspect crucial du développement Python. C'est là que **poetry** entre en jeu. Dans le chapitre suivant, nous plongerons dans les potentialités de **poetry**, un outil moderne qui simplifie la gestion des dépendances et des environnements virtuels, nous permettant de gérer nos projets Python avec une précision et une efficacité inégalées.

5

poetry
Un allié précieux pour le développement Python

> *“La poésie est ce qu’il y a de plus réel, c’est ce qui n’est complètement vrai que dans un autre monde.”*
> — Charles Baudelaire