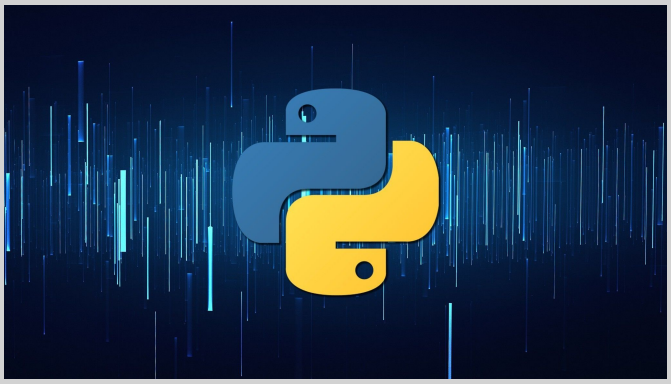
Environnements de programmation Python, et les outils dédiés



Krystof26

19 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

Préface 2

1 Les environnements virtuels **5**

1.1 Le module venv 6

1.2 Intérêt des environnements virtuels 8

1.3 Structure d'un environnement virtuel 9

1.4 Fonctionnement d'un environnement virtuel 11

1.5 Personnaliser un environnement virtuel 13

1.6 Gestion des environnements virtuels 16

2 *virtualenv* **19**

3 *pip* **21**

3.1 Utilisation dans un environnement virtuel 21

3.2 Installer des paquets avec *pip* 22

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
```

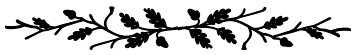
Création et activation de l’environnement virtuel (depuis le répertoire du projet) :

```
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.

2. `https://pypi.org/`