Environnements de programmation Python, et les outils dédiés

Krystof26

Juin 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 nous 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**[1] 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 :

```
user@machine:$ python3 -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 :

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

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

1. https://docs.python.org/fr/3.13/library/venv.html

```
[...]
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** :

```
(.venv) 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 de 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 :

```
1 home = /usr/bin
2 include-system-site-packages = false
3 version = 3.13.2
4 executable = /usr/bin/python3.13
5 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 donc ceux de la bibliothèque standard issue de l'installation de la version 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**[2] 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
```

2. 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 [3].

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 [4].

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
```

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

4. 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
  ↪ 288ms
  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
```

2. https://pypi.org/

```
parso              0.8.4
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
```

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

```
    ↪  kB)
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
    ↪  08dcb019aea6e3e326c39be9741e52adb77a2c19
  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 :

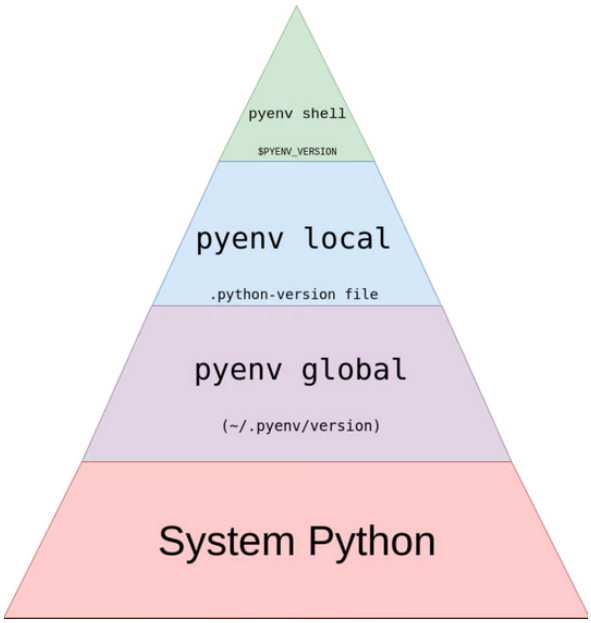

FIGURE 4.1 – Ordre de résolution des commandes **pyenv**

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**

Dans le vaste écosystème du développement Python, la gestion des dépendances et des environnements virtuels peut rapidement devenir un casse-tête. C'est là que **poetry**[1] entre en scène, se présentant comme une solution moderne et élégante pour simplifier ces tâches complexes. **Poetry** (disponible depuis la version **3.9** de Python) est bien plus qu'un simple gestionnaire de paquet, c'est un outil complet qui permet de gérer les dépendances, de créer des environnements virtuels, et de *packager* des projets Python avec une facilité déconcertante.

Poetry rend tout cela possible. En offrant une interface simple et puissante, **poetry** nous permet de nous concentrer sur ce qui compte vraiment : le développement de nos applications.

Dans ce chapitre, nous allons explorer les multiples facettes de **poetry**. Nous commencerons par une introduction à ses fonctionnalités de base, puis nous plongerons dans des aspects plus avancés, tels que la gestion des environnements virtuels, la résolution des dépendances, et le packaging de projets. **poetry** a quelque chose à offrir pour rendre le flux de travail plus efficace et agréable.

1. Site officiel : https://python-poetry.org/

5.1 Installation

Poetry est distribué comme un paquet Python lui-même, il est donc installable dans un environnement virtuel en utilisant **pip**, tout comme n'importe quel autre paquet externe :

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

C'est bien pour l'essayer rapidement. Cependant, la documentation officielle[2] déconseille fortement d'installer **Poetry** dans l'environnement virtuel d'un projet, car **Poetry** dépend de plusieurs paquets externes, et l'on courte le risque d'un conflit de dépendance entre l'une des dépendances du projet et celles requises par **Poetry**. En pratique, il faut donc garder **Poetry** séparé de tout environnement virtuel créé pour le projet Python.

Il est donc plus judicieux d'installer **Poetry** sur le système.

Installation via *pipx*

Poetry sera installé dans un environnement virtuel dédié qui ne sera pas partagé avec d'autres paquets Python. De plus, **pipx** définit un alias à l'exécutable `poetry` afin de pouvoir appeler **Poetry** depuis n'importe quel répertoire sans activer manuellement l'environnement virtuel associé.

```
user@machine: $ pipx install poetry
  installed package poetry 2.1.3, installed using Python
    ↪ 3.13.3
  These apps are now globally available
    - poetry
done!
user@machine: $ poetry --version
Poetry (version 2.1.3)
```

Avec cette méthode d'installation, en tapant `poetry` dans le terminal on se réfère toujours au script exécutable installé dans son environnement virtuel isolé. La commande `poetry` lancée depuis l'environnement virtuel actif d'un projet, sera correctement captée par **Poetry**.

```
user@machine: $ pipx upgrade poetry  # Pour une mise à jour
user@machine: $ pipx uninstall poetry  # Pour la désins
    # tallation
```

2. https://python-poetry.org/docs/#installation

Via l'installateur officiel

Poetry fournit un installateur personnalisé qui installe **Poetry** en l'isolant du reste du système. Exécuter la commande suivante :

```
$ curl -sSL https://install.python-poetry.org | python3 -
Retrieving Poetry metadata

\# Welcome to Poetry!

This will download and install the latest version of Poetry,
a dependency and package manager for Python.

It will add the `poetry` command to Poetry's bin directory, located at:

/home/krystof/.local/bin

You can uninstall at any time by executing this script with the --uninstall
option, and these changes will be reverted.

Installing Poetry (2.1.3): Done

Poetry (2.1.3) is installed now. Great!

You can test that everything is set up by executing:

`poetry --version`
```

Sous l'URL `install.python-poetry.org` se trouve un script Python multi-plateforme qui, plus ou moins, reproduit ce que fait **pipx**, mais d'une manière légèrement différente.

Via un paquet système prédéfini

Certains systèmes d'exploitation peuvent proposer **Poetry** en tant que paquet natif. Par exemple, un système basé sur **Debian GNU/Linux** permet d'installer **Poetry** avec `apt` :

```
user@machine: $ sudo apt install python3-poetry
```

Garder à l'esprit que la version de **Poetry** installée via un paquet système pourrait ne pas être la dernière. De plus, le paquet natif peut apporter plusieurs centaines de mégaoctets de dépendances supplémentaires (environ une cinquantaine de paquets système avec **Debian GNU/Linux** *testing*, comme un autre interpréteur Python, ce qui serait complètement inutile.

Activer l'auto-complétion pour *Zsh*

Copier la configuration d'auto-complétion dans un fichier dédié :

```
user@machine: $ poetry completions zsh > ~/.zfunc/_poetry
```

Puis, ajouter les lignes suivantes dans **/.zshrc**, si elles n'existent pas déjà :

```
fpath+=~/.zfunc
autoload -Uz compinit && compinit
```

5.2 Débuter avec *Poetry*

Création du projet

Création d'un projet **poetry** à l'aide de la commande `poetry new`[3] :

```
user@machine: $ poetry new mon_projet
Created package mon_projet in mon_projet
user@machine: $ cd mon_projet
user@machine: $ tree
```

```
.
|-- pyproject.toml
|-- README.md
|-- src
|   |-- mon_projet
|       |-- __init__.py
|-- tests
    |-- __init__.py

4 directories, 4 files
```

Pour différencier le nom du projet du nom du paquetage Python, nous pouvons utiliser l'argument `-name` :

```
user@machine: $ poetry new mon_projet --name package_project
Created package package_project in mon_projet
user@machine: $ cd mon_projet
user@machine: $ tree
```

3. https://python-poetry.org/docs/cli/#new

```
.
|-- pyproject.toml
|-- README.md
|-- src
|   |-- package_project
|       |-- __init__.py
|-- tests
    |-- __init__.py

4 directories, 4 files
```

Nous obtenons une structure d'un projet de base sans avoir à réfléchir à l'organisation des fichiers Python.

Inspecter la structure du projet

À l'intérieur du sous-répertoire `package_project` se trouve un fichier `__init__.py` vide qui transforme le projet en un paquetage Python importable.

De même, le sous-répertoire `tests/` est un autre paquetage Python destiné à contenir les tests unitaires (éventuellement d'autres types de tests).

Poetry génère également un fichier `README` vide, ainsi qu'un fichier de configuration nommé `pyproject.toml` avec les métadonnées minimales requises pour le projet[4]

pyproject.toml

L'un des fichiers les plus importants pour travailler avec **Poetry** est le fichier `pyproject.toml` :

```
[project]
name = "package-project"
version = "0.1.0"
description = ""
authors = [
    {name = "nom_dev",email = "mon_email@hebergeur.com"}
]
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
]

[tool.poetry]
packages = [{include = "package_project", from = "src"}]
```

4. Cf. le PEP 518 : https://peps.python.org/pep-0518/

```
[build-system]
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"
```

Nous voyons trois sections marquées par des crochets, appelées « *tables* » dans la terminologie **TOML**. Elles contiennent des instructions déclaratives que des outils comme **Poetry** peuvent reconnaître et utiliser pour gérer les dépendances, construire le projet ou effectuer d'autres tâches. Le point (« . ») dans le nom d'une table **TOML** est un délimiteur, qui sépare les différents niveaux de la hiérarchie.

Le fichier `pyproject.toml` commence par la table secondaire `[project]`, dans laquelle sont stockées des informations générales sur le projet. **Poetry** définit quelques clés de table dont certaines doivent toujours être spécifiées :

name : Le nom du paquet de distribution qui apparaîtra sur **PyPI**

version : La version du paquet

description : Une courte description du paquet

authors : Une liste d'auteurs avec le nom et l'email

La table `tool.poetry` contient des informations sur la structure du projet, les dépendances, et d'autres configurations nécessaires pour la gestion du projet avec **Poetry**. La section `packages` spécifie les packages inclus dans le projet. Dans l'exemple ci-dessus, le package `package_project` est inclus et est situé dans le répertoire `src`.

La table `build-system` est utilisée pour définir les exigences et le *backend* de construction du projet. Elle est essentielle pour la construction et l'installation du package. La section `requires` liste les dépendances nécessaires pour construire le projet. La section `build-backend` spécifie le *backend* de construction à utiliser qui est responsable de la construction du package et de la génération des artefacts de distribution (comme les fichiers `.whl` et `.tar.gz`).

Au fur et à mesure que le projet grandit, le fichier `pyproject.toml` grandit avec lui. C'est particulièrement vrai pour la table secondaire `tool.poetry`.

5.3 Travailler avec *Poetry*

Activer un environnement virtuel personnalisé

Poetry est livré avec un support intégré pour les environnements virtuels afin de s'assurer qu'il n'interfère jamais avec l'installation globale de Python. L'outil peut donc prendre en charge la majeure partie de la gestion des environnements virtuels.

Cependant, `Poetry` ne crée pas d'environnement virtuel immédiatement lors de la création d'un nouveau projet. Cela a été conçu pour nous permettre de décider si nous souhaitons gérer nos environnements virtuels nous-même ou laisser **Poetry** les gérer automatiquement.

Poetry détectera un environnement virtuel activé manuellement en exécutant la commande **Poetry** suivante directement dans le répertoire du projet :

```
user@machine: $ python3 -m venv .venv/
user@machine: $ source .venv/bin/activate
(.venv)user@machine: $ poetry env info --path
/home/utilisateur/mon_projet/.venv
```

En affichant les informations sur l'environnement, **Poetry** confirme qu'il utilisera cet environnement pour toutes les commandes ultérieures dans le cadre du projet. En d'autres termes, en ajoutant des dépendances au projet par l'intermédiaire de **Poetry**, elles seront installées dans l'environnement activé comme si nous utilisions la commande `pip install`. **Poetry** mettra également à jour les métadonnées nécessaires dans `pyproject.toml`.

D'autre part, **Poetry** crée automatiquement un environnement virtuel lorsque nous exécuterons certaines commandes sans environnement activé dans le *shell*. Ce sera le cas en ajoutant ou supprimant une dépendance à l'aide de l'interface de ligne de commande de **Poetry**. Cela empêche les projets de perturber l'installation de Python sur l'ensemble du système et garantit que les dépendances du projet restent isolées à tout moment.

Utiliser les environnements virtuels de *Poetry*

Lister tous les environnements virtuels que **Poetry** gère pour le projet en exécutant la commande suivante dans le répertoire du projet :

```
(.venv)user@machine: $ poetry env list
.venv (Activated)
(.venv) user@machine: $ deactivate
user@machine: $  rm -Rf .venv/
user@machine: $  poetry env list
user@machine: $
```

Dorénavant, **Poetry** se chargera de la création et de la gestion des environnements virtuels du projet lors de l'exécution de certaines de ses commandes.

Pour mieux contrôler la création d'un environnement virtuel, il est possible d'indiquer explicitement à **Poetry** la version de Python souhaitée :

```
user@machine: $ poetry env use python3  # ou la version
   ↪ mineure : python3.13
Creating virtualenv package-project-sCgEQKLZ-py3.13 in /
   ↪ home/utilisateur/.cache/pypoetry/virtualenvs
Using virtualenv: /home/utilisateur/.cache/pypoetry/
   ↪ virtualenvs/package-project-sCgEQKLZ-py3.13
```

L'interpréteur Python indiqué à **Poetry** doit satisfaire la contrainte de version du fichier `pyproject.toml`. Dans le cas contraire, **Poetry** le rejettera avec un message d'erreur.

`sCgEQKLZ` est une valeur de hachage encodée en *Base64* du chemin menant au répertoire parent du projet. Elle lie le nom d'un environnement virtuel à l'emplacement du projet sur le disque.

Ainsi, lorsque l'on déplace le projet dans un autre répertoire, **Poetry** le détecte et crée un nouvel environnement virtuel en arrière-plan si nécessaire. Grâce à la chaîne unique au milieu, **Poetry** peut gérer plusieurs projets avec des noms identiques et la même version de Python tout en conservant tous les environnements virtuels dans un seul dossier par défaut.

Sauf indication contraire, **Poetry** crée les environnements virtuels dans le sous-répertoire `virtualenvs/` de son répertoire de cache, qui est spécifique au système d'exploitation. Sous **GNU/Linux** le chemin vers ce répertoire est `/.cache/pypoetry/`.

Pour révéler la configuration **Poetry** actuelle, qui comprend les paramètres `cache-dir` et `virtualenvs.path` :

```
user@machine: $ poetry config --list
cache-dir = "/home/utilisateur/.cache/pypoetry"
data-dir = "/home/utilisateur/.local/share/pypoetry"
installer.max-workers = null
installer.no-binary = null
installer.only-binary = null
installer.parallel = true
installer.re-resolve = true
keyring.enabled = true
python.installation-dir = "{data-dir}/python"  # /home/
   ↪ utilisateur/.local/share/pypoetry/python
requests.max-retries = 0
solver.lazy-wheel = true
system-git-client = false
virtualenvs.create = true
virtualenvs.in-project = null
virtualenvs.options.always-copy = false
virtualenvs.options.no-pip = false
virtualenvs.options.system-site-packages = false
virtualenvs.path = "{cache-dir}/virtualenvs"  # /home/
   ↪ utilisateur/.cache/pypoetry/virtualenvs
virtualenvs.prompt = "{project_name}-py{python_version}"
virtualenvs.use-poetry-python = false
```

Informations sur l'environnement du projet :

```
user@machine: $ poetry env info

Virtualenv
```

```
Python:         3.13.3
Implementation: CPython
Path:           /home/utilisateur/.cache/pypoetry/
   ↪ virtualenvs/package-project-sCgEQKLZ-py3.13
Executable:     /home/utilisateur/.cache/pypoetry/
   ↪ virtualenvs/package-project-sCgEQKLZ-py3.13/bin/
   ↪ python
Valid:          True

Base
Platform:   linux
OS:         posix
Python:     3.13.3
Path:       /usr
Executable: /usr/bin/python3.13
user@machine: $ poetry env info --path
/home/utilisateur/.cache/pypoetry/virtualenvs/package-
   ↪ project-sCgEQKLZ-py3.13
user@machine: $ poetry env info --executable
/home/utilisateur/.cache/pypoetry/virtualenvs/package-
   ↪ project-sCgEQKLZ-py3.13/bin/python
user@machine: $ poetry env list
package-project-sCgEQKLZ-py3.13 (Activated)
user@machine: $ poetry env list --full-path
/home/utilisateur/.cache/pypoetry/virtualenvs/package-
   ↪ project-sCgEQKLZ-py3.13 (Activated)
\end{listing}

Supprimer l environnement virtuel du projet :
\begin{lstlisting}[style=terminal]
user@machine: $ poetry env remove 3.13
Deleted virtualenv: /home/utilisateur/.cache/pypoetry/
   ↪ virtualenvs/package-project-sCgEQKLZ-py3.13
user@machine: $ poetry env info

Virtualenv
Python:         3.13.3
Implementation: CPython
Path:           NA
Executable:     NA

Base
Platform:   linux
OS:         posix
Python:     3.13.3
Path:       /usr
Executable: /usr/bin/python3.13
user@machine: $ poetry env list
```

```
user@machine: $
```

Pour aller plus loin concernant la gestion des environnements : https://python-poetry.org/docs/managing-environments/

Déclarer les dépendances d'exécution

Le fichier `pyproject.toml` est l'endroit où nous pouvons spécifier les paquets Python externes dont notre projet a besoin. L'éditer à la main n'installera rien dans l'environnement virtuel du projet, et c'est là que **Poetry** entre à nouveau en jeu en l'utilisant à l'instar de **pip**.

L'exécution de la commande `poetry add` mettra automatiquement à jour le fichier `pyproject.toml` et installera le paquet en même temps. Il est également possible de spécifier plusieurs paquets en une seule fois :

```
user@machine: $ poetry add paquet1 paquet2
```

Imaginons vouloir installer le paquet **PySide6**[5]. Il nous faudra apporter tout d'abord une modification au fichier `pyproject.toml` :

```
requires-python = ">=3.13, <3.14"
```

Puis :

```
user@machine: $ poetry add pyside6
Using version ^6.9.0 for pyside6

Updating dependencies
Resolving dependencies... (0.3s)

Package operations: 4 installs, 0 updates, 0 removals

  - Installing shiboken6 (6.9.0)
  - Installing pyside6-essentials (6.9.0)
  - Installing pyside6-addons (6.9.0): Installing...
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DAnimation.pyi over existing
   ↪ file
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DCore.pyi over existing file
```

5. https://doc.qt.io/qtforpython-6/

```
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DExtras.pyi over existing file
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DInput.pyi over existing file
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DLogic.pyi over existing file
[...]
  - Installing pyside6-addons (6.9.0)
  - Installing pyside6 (6.9.0): Installing...
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DAnimation.pyi over existing
   ↪ file
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DCore.pyi over existing file
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DExtras.pyi over existing file
Installing /home/krystof/.cache/pypoetry/virtualenvs/
   ↪ package-project-sCgEQKLZ-py3.13/lib/python3.13/site-
   ↪ packages/PySide6/Qt3DInput.pyi over existing file
[...]
- Installing pyside6 (6.9.0)

Writing lock file
```

Cela vient également apporter une modification au fichier `pyproject.toml` :

```
dependencies = [
    "pyside6 (>=6.9.0,<7.0.0)"
]
```

Si l'on souhaite ajouter une version particulière d'un paquetage externe ou définir des contraintes de version personnalisées, **Poetry** le permet :

```
user@machine: $ poetry add pyside6=6.8.3  # Versin
   ↪ compatible avec python3.13

Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 4 updates, 0 removals
```

```
  - Downgrading shiboken6 (6.9.0 -> 6.8.3)
  - Downgrading pyside6-essentials (6.9.0 -> 6.8.3)
  - Downgrading pyside6-addons (6.9.0 -> 6.8.3)
  - Downgrading pyside6 (6.9.0 -> 6.8.3)

Writing lock file
```

A noter que **Poetry** supprime d'abord toutes les versions précédemment installées de ces paquets et rétrograde leurs dépendances indirectes ou transitives si nécessaire. Il déterminera ensuite les versions les plus appropriées de ces paquets, en tenant compte des autres contraintes existantes pour résoudre les conflits potentiels.

Nous aurions pu également demander l'installation de la version de **PySide6** inférieure à la version **6.9.0** (à notre l'utilisation des guillemets pour pas que le signe « < » soit interprété comme opérateur de redirection) :

```
user@machine: $ poetry add "pyside6<6.9.0"

Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 4 updates, 0 removals

  - Downgrading shiboken6 (6.9.0 -> 6.8.3)
  - Downgrading pyside6-essentials (6.9.0 -> 6.8.3)
  - Downgrading pyside6-addons (6.9.0 -> 6.8.3)
  - Downgrading pyside6 (6.9.0 -> 6.8.3)

Writing lock file
```

Pour supprimer le paquet on utilisera la commande `remove` :

```
user@machine: $ poetry remove pyside6
Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 0 updates, 4 removals

  - Removing pyside6 (6.8.3)
  - Removing pyside6-addons (6.8.3)
  - Removing pyside6-essentials (6.8.3)
  - Removing shiboken6 (6.8.3)

Writing lock file
```

Dépendances de groupe et ajout d'extras

Une autre fonctionnalité intéressante de **Poetry** est la possibilité de gérer des groupes de dépendances, permettant ainsi de garder les dépendances logiquement liées séparées des dépendances d'exécution. Par exemple, pendant le développement, nous aurons souvent besoin de paquets supplémentaires, tels que des *linters*, des *checkers* de type ou des *frameworks* de test, qui ne feraient qu'enfler le paquet final, mais qui n'aurons aucun intérêt pour les utilisateurs.

Poetry offre la possibilité de grouper les dépendances sous des noms arbitraires de sorte que nous pouvons installer sélectivement ces groupes plus tard au besoin. Voici comment ajouter quelques dépendances à un groupe appelé dev et quelques dépendances à un autre groupe appelé `test` :

```
user@machine: $ poetry add --group dev black flake8 isort
    mypy pylint
Using version ^25.1.0 for black
Using version ^7.2.0 for flake8
Using version ^6.0.1 for isort
Using version ^1.15.0 for mypy
Using version ^3.3.7 for pylint

Updating dependencies
Resolving dependencies... (1.9s)

Package operations: 17 installs, 0 updates, 0 removals

  - Installing astroid (3.3.10)
  - Installing click (8.1.8)
  - Installing dill (0.4.0)
  - Installing isort (6.0.1)
  - Installing mccabe (0.7.0)
  - Installing mypy-extensions (1.1.0)
  - Installing packaging (25.0)
  - Installing pathspec (0.12.1)
  - Installing platformdirs (4.3.8)
  - Installing pycodestyle (2.13.0)
  - Installing pyflakes (3.3.2)
  - Installing tomlkit (0.13.2)
  - Installing typing-extensions (4.13.2)
  - Installing black (25.1.0)
  - Installing mypy (1.15.0)
  - Installing flake8 (7.2.0)
  - Installing pylint (3.3.7)

Writing lock file
user@machine: $ poetry add --group test pytest faker
Using version ^8.3.5 for pytest
```

```
Using version ^37.1.0 for faker

Updating dependencies
Resolving dependencies... (0.6s)

Package operations: 5 installs, 0 updates, 0 removals

  - Installing iniconfig (2.1.0)
  - Installing pluggy (1.5.0)
  - Installing tzdata (2025.2)
  - Installing faker (37.1.0)
  - Installing pytest (8.3.5)

Writing lock file
```

Seront également ajoutées deux sous-tables supplémentaires dans le fichier `pyproject.toml` :

```
[tool.poetry.group.dev.dependencies]
black = "^25.1.0"
flake8 = "^7.2.0"
isort = "^6.0.1"
mypy = "^1.15.0"
pylint = "^3.3.7"


[tool.poetry.group.test.dependencies]
pytest = "^8.3.5"
faker = "^37.1.0"
```

Il est aussi possible d"ajouter des paquets individuels en option pour laisser l'utilisateur choisir s'il veut les installer :

```
user@machine: $ poetry add --optional mysqlclient psycopg2-
    binary
poetry add --optional mysqlclient psycopg2-binary
Using version ^2.9.10 for psycopg2-binary

Updating dependencies
Resolving dependencies... (0.3s)

No dependencies to install or update

Writing lock file
```

Les dépendances facultatives sont censées être disponibles à l'exécution lorsque explicitement demandées par l'utilisateur lors de l'installation. Il est courant de marquer les pa-

ckages comme optionnels lorsqu'ils sont spécifiques à une plate-forme ou lorsqu'ils fournissent des fonctionnalités, telles qu'un adaptateur de base de données particulier, dont seuls certains utilisateurs auront besoin.

Dans `pyproject.toml`, les dépendances optionnelles sont un peu plus détaillées :

```
[project.optional-dependencies]
mysqlclient = ["psycopg2-binary (>=2.9.10,<3.0.0)"]
```

Cependant, ce n'est pas suffisant pour exposer ces dépendances facultatives à l'utilisateur. On doit également définir des *extras* dans le fichier `pyproject.toml`, qui sont des ensembles de dépendances optionnelles que les utilisateurs peuvent installer ensemble :

```
[tool.poetry.extras]
databases = ["mysqlclient", "psycopg2-binary"]
mysql = ["mysqlclient"]
pgsql = ["psycopg2-binary"]
```

Installer son package avec *poetry*

Imaginons que nous venez de cloner un dépôt **Git** depuis **GitHub** et que nous redémarrions sans environnement virtuel. Pour simuler cela, nous pouvez supprimer certaines des métadonnées de **Poetry** et tout environnement virtuel associé au projet :

```
user@machine: $ rm poetry.lock
user@machine: $ poetry env remove --all
```

Il nous suffira de saisir la commande suivante pour retrouver un environnement virtuel avec toutes les dépendances nécessaires installée :

```
user@machine: $ $ poetry install
```

5.4 Gérer les dépendances

Chaque fois que l'on interagit avec **Poetry** le fichier `pyproject.toml` est mis à jour et les versions résolues sont épinglées dans le fichier **poetry.lock**.

Repartons avec un nouveau projet, un nouvel environnement virtuel et de nouvelles dépendances :

```
user@machine: $ poetry new mon_projet --name code_projet
Created package code_projet in mon_projet
```

```
user@machine: $ cd mon_projet
user@machine: $ tree
```

```
.
\- pyproject.toml
\- README.md
\- src
|   \- code_projet
|       \- __init__.py
\- tests
    \- __init__.py

4 directories, 4 files
```

```
user@machine: $ poetry env use 3.13
Creating virtualenv code-projet-sCgEQKLZ-py3.13 in /home/
   ↪ utilisateur/.cache/pypoetry/virtualenvs
Using virtualenv: /home/utilisateur/.cache/pypoetry/
   ↪ virtualenvs/code-projet-sCgEQKLZ-py3.13
user@machine: $  poetry add requests beautifulsoup4
Using version ^2.32.3 for requests
Using version ^4.13.4 for beautifulsoup4

Updating dependencies
Resolving dependencies... (0.8s)

Package operations: 8 installs, 0 updates, 0 removals

  - Installing certifi (2025.4.26)
  - Installing charset-normalizer (3.4.2)
  - Installing idna (3.10)
  - Installing soupsieve (2.7)
  - Installing typing-extensions (4.13.2)
  - Installing urllib3 (2.4.0)
  - Installing beautifulsoup4 (4.13.4)
  - Installing requests (2.32.3)

Writing lock file
```

Verrouiller manuellement les dépendances

Poetry génère et rafraîchit le fichier `poetry.lock` lorsque nécessaire. Ce fichier de verrouillage n'est pas destiné à être modifié manuellement, mais nous pouvons éditer le fichier

`pyproject.toml` associé. Malheureusement, cela peut parfois entraîner un décalage entre les deux fichiers.

Note : Si nous insistons pour manipuler manuellement le fichier `poetry.lock`, nous allons probablement invalider les hachages sous-jacents, rendant le fichier cassé et inutilisable.

Supposons que nous voulions ramener la bibliothèque **requests** que nous avons retiré du projet.

```
user@machine: $ poetry remove requests
Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 0 updates, 5 removals

  - Removing certifi (2025.4.26)
  - Removing charset-normalizer (3.4.2)
  - Removing idna (3.10)
  - Removing requests (2.32.3)
  - Removing urllib3 (2.4.0)

Writing lock file
```

Nous pouvons éditer le fichier `pyproject.toml` pour y saisir la déclaration nécessaire dans le groupe principal des dépendances. D'abord nous ajoutons dans le fichier à `dependencies`, la ligne : « `requests` »,. Puis lancer :

```
user@machine: $ poetry install
Installing dependencies from lock file

pyproject.toml changed significantly since poetry.lock
   ↪ was last generated. Run ‘poetry lock‘ to fix the
   ↪ lock file.
```

Dans ce cas, **Poetry** refuse d'installer les dépendances le fichier `poetry.lock` ne mentionne pas la bibliothèque **Requests** présente dans le fichier `pyproject.toml` associé.

Pour corriger une telle divergence, nous pouvons supprimer le fichier de verrouillage et exécuter `poetry install` à nouveau pour permettre à **Poetry** de résoudre toutes les dépendances à partir de zéro. Mais cela est potentiellement long. Mais pire encore, seront ignorées les versions spécifiques des dépendances précédemment résolues, supprimant la garantie de compilations reproductibles.

Une approche bien meilleure pour aligner les deux fichiers consiste à verrouiller manuellement les nouvelles dépendances avec la commande `poetry lock` :

```
user@machine: $ poetry lock
Resolving dependencies... (0.5s)

Writing lock file
```

Cela met à jour le fichier `poetry.lock` pour qu'il corresponde au fichier `pyproject.toml` actuel sans installer de dépendances.

Poetry traite toutes les dépendances dans le fichier `pyproject.toml`, trouve les paquets qui satisfont aux contraintes déclarées et épingle leurs versions exactes dans le fichier de verrouillage. Mais **poetry** ne s'arrête pas là car lorsque nous exécutons `poetry lock`, il analyse et verrouille récursivement toutes les dépendances directes.

Synchroniser son environnement

Lorsque le fichier `poetry.lock` correspond à son homologue `pyproject.toml`, nous pouvons enfin installer les dépendances que **Poetry** a verrouillé :

```
user@machine: $ poetry install
Installing dependencies from lock file

Package operations: 5 installs, 0 updates, 0 removals

  - Installing certifi (2025.4.26)
  - Installing charset-normalizer (3.4.2)
  - Installing idna (3.10)
  - Installing urllib3 (2.4.0)
  - Installing requests (2.32.3)

Installing the current project: code-projet (0.1.0)
```

En exécutant `poetry install`, le fichier `poetry.lock` et sont installées toutes les dépendances qui y sont listées. Comme l'environnement virtuel avait déjà la plupart des dépendances requises en place, **Poetry** n'a installé que les dépendances manquantes. Et si on exécute à nouveau la même commande, **Poetry** n'aura plus grand-chose à faire :

```
user@machine: $ poetry install
Installing dependencies from lock file

No dependencies to install or update

Installing the current project: code-projet (0.1.0)
```

Par conséquent, la bibliothèque **requests** sera disponible lorsqu'elle est importée dans une session interactive REPL de Python démarrée via **Poetry** :

```
user@machine: $ poetry run python -q  # -q supprime le
   ↪ message de bienvenue
```

```
>>> import requests
>>> requests.__version__
'2.32.3'
```

Synchroniser l'environnement virtuel avec les paquets verrouillés et épinglés dans le fichier `poetry.lock` :

```
user@machine: $ poetry sync
```

Cela garantit que l'environnement virtuel ne contient que les paquets spécifiés dans les fichiers `pyproject.toml` et `poetry.lock`, évitant ainsi les conflits potentiels causés par des dépendances inutiles ou obsolètes.

Mise à jour et mise à niveau des dépendances

Imaginons un projet avec **Pyside6** version **6.8.3** d'installé. Comparons les dépendances verrouillées avec leurs dernières versions sur **PyPI** :

```
user@machine: $ poetry show --latest
pyside6            6.8.3 6.9.0 Python bindings for the Qt
   ↪  cross-platform application and UI framework
pyside6-addons     6.8.3 6.9.0 Python bindings for the Qt
   ↪  cross-platform application and UI framework (Addons
   ↪ )
pyside6-essentials 6.8.3 6.9.0 Python bindings for the Qt
   ↪  cross-platform application and UI framework (
   ↪ Essentials)
shiboken6          6.8.3 6.9.0 Python/C++ bindings helper
   ↪  module
```

Il semble qu'une mise à jour s'impose :

```
user@machine: $ poetry update --dry-run
    # Pour simuler la mise à jour
Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 0 updates, 0 removals, 4
   ↪ skipped
```

```
  - Installing pyside6 (6.8.3): Skipped for the following
    ↪  reason: Already installed
  - Installing pyside6-addons (6.8.3): Skipped for the
    ↪ following reason: Already installed
  - Installing pyside6-essentials (6.8.3): Skipped for
    ↪ the following reason: Already installed
  - Installing shiboken6 (6.8.3): Skipped for the
    ↪ following reason: Already installed
```

```
user@machine: $ poetry update  # Pour tous les paquets
user@machine: $ poetry update paquet1 paquet2  # Seulement
   ↪ les paquets indiqués
```

Avec cette commande, **Poetry** recherchera une nouvelle version des paquets qui répondent aux contraintes de version listées dans le fichier `pyproject.toml`. Ensuite, il résoudra toutes les dépendances du projet et épinglera leurs versions dans le fichier `poetry.lock`.

Normalement, pour mettre à niveau une dépendance vers une version qui est en dehors des contraintes de version déclarées dans le fichier `pyproject.toml`, nous devons ajuster ce fichier au préalable. Sinon, nous pouvons mettre à niveau une dépendance vers sa dernière version en exécutant la commande `poetry add` avec l'opérateur `at` (`@`) et le mot-clé `latest` :

```
user@machine: $ poetry add pyside6@latest
Using version ^6.9.0 for pyside6

Updating dependencies
Resolving dependencies... (0.1s)

Package operations: 0 installs, 4 updates, 0 removals

  - Updating shiboken6 (6.8.3 -> 6.9.0)
  - Updating pyside6-essentials (6.8.3 -> 6.9.0)
  - Updating pyside6-addons (6.8.3 -> 6.9.0)
  - Updating pyside6 (6.8.3 -> 6.9.0)

Writing lock file
```

Comparer `pyproject.toml` et `poetry.lock`

Les contraintes de version des dépendances déclarées dans le fichier `pyproject.toml` peuvent être assez lâches. Cela permet un certain niveau de flexibilité lors de l'intégration des corrections de bogues ou de la résolution des conflits de version. En ayant plus de versions de paquets à choisir, **Poetry** est davantage susceptible de trouver une combinaison de dépendances compatibles.

D'autre part, **Poetry** suit les versions exactes des dépendances utilisées dans le fichier `poetry.lock`. Cela améliore les performances de **Poetry** en mettant en cache les versions des paquets résolus afin qu'il n'ait pas à les résoudre à nouveau chaque fois que l'on installe ou met à jour des dépendances.

Pour garantir des environnements reproductibles dans une équipe, il est nécessaire d'envisager de transférer le fichier `poetry.lock` au système de contrôle de version. En gardant ce fichier suivi nous nous assurons que tous les développeurs utiliseront des versions identiques des packages requis.

Cependant, il y a une exception notable. Lorsque l'on développe une bibliothèque plutôt qu'une application, il est courant de ne pas *commiter* le fichier `poetry.lock`. Les bibliothèques doivent généralement rester compatibles avec plusieurs versions de leurs dépendances plutôt qu'avec un seul ensemble verrouillé.

5.5 Ajouter *poetry* à un projet existant

Convertir un répertoire en projet poetry

Au lieu d'utiliser la commande `poetry new`, nous utiliserons la commande `poetry init` dans le répertoire du projet.

Voici la structure d'un projet situé dans un répertoire nommé `PyCalendar` :

```
.
|- IMG
|   |- PyCalendar.png
|- main.py
|- README.md
|- requirements.txt
```

Il est composé d'un fichier `PyCalendar.png` appelé depuis le fichier `README.md`, et d'un fichier Python nommé `main.py`. Ce projet fait appel au *framework* **PySide6**, comme le précise le fichier `requirements.txt` :

```
PySide6==6.9.0
PySide6_Addons==6.9.0
PySide6_Essentials==6.9.0
shiboken6==6.9.0
```

Initialisation du projet :

```
user@machine: $ poetry init
```

```
This command will guide you through creating your
   ↪ pyproject.toml config.

Package name [pycalendar]:
Version [0.1.0]:
Description []:
Author [Nom dev <LeDev@email.com>, n to skip]:
License []:
Compatible Python versions [>=3.13]:

Would you like to define your main dependencies
   ↪ interactively? (yes/no) [yes]
        You can specify a package in the following forms:
          - A single name (requests): this will search
             ↪ for matches on PyPI
          - A name and a constraint (requests@^2.23.0)
          - A git url (git+https://github.com/python-
             ↪ poetry/poetry.git)
          - A git url with a revision         (git+https
             ↪ ://github.com/python-poetry/poetry.git#
             ↪ develop)
          - A file path (../my-package/my-package.whl)
          - A directory (../my-package/)
          - A url (https://example.com/packages/my-
             ↪ package-0.1.0.tar.gz)

Package to add or search for (leave blank to skip):

Would you like to define your development dependencies
   ↪ interactively? (yes/no) [yes]
Package to add or search for (leave blank to skip):

Generated file

[project]
name = "pycalendar"
version = "0.1.0"
description = ""
authors = [
    {name = "Nom dev",email = "LeDev@email.com"}
]
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
]


[build-system]
```

```
requires = ["poetry-core>=2.0.0,<3.0.0"]
build-backend = "poetry.core.masonry.api"


Do you confirm generation? (yes/no) [yes]
user@machine: $
```

La commande **poetry init** recueille les informations nécessaires pour générer un fichier `pyproject.toml` en posant des questions de manière interactive. Les valeurs par défaut sont raisonnables pour la plupart des configurations et l'on peut donc systématiquement appuyer sur Entrée pour les accepter, y compris sans déclarer de dépendances.

Nous pouvons ensuite réorganiser notre structure de projet comme cela :

```
|- IMG
|   |- PyCalendar.png
|- pyproject.toml
|- README.md
|- requirements.txt
|- src
    |- pycalendar
        |- main.py
```

Maintenant nous pouvons utiliser toutes les commandes que **Poetry** offre. Exécutons notre script :

```
user@machine: $ poetry run python src/pycalendar/main.py
Creating virtualenv pycalendar-zCgC7Sg9-py3.13 in /home/
   ↪ utilisateur/.cache/pypoetry/virtualenvs
Traceback (most recent call last):
  File "/home/utilisateur/PyCalendar/src/pycalendar/main.
     ↪ py", line 4, in <module>
    from PySide6.QtCore import QSize, Qt
ModuleNotFoundError: No module named 'PySide6'
```

Comme **Poetry** n'a pas trouvé d'environnement virtuel à utiliser, il en a créé un nouveau avant d'exécuter le script. Mais par la suite, le module **PySide6** n'a pas été trouvé. Il va donc nous falloir l'installer.

Importer des dépendances

Si le projet contient déjà un fichier d'exigences (`requirements.txt`), nous allons pouvoir l'importer :

```
user@machine: $ poetry add $(cat requirements.txt)
```

```
Updating dependencies
Resolving dependencies... (0.0s)

The current project s supported Python range (>=3.13) is
   ↪ not compatible with some of the required packages
   ↪ Python requirement:
  - pyside6 requires Python <3.14,>=3.9, so it will not
     ↪ be installable for Python >=3.14

Because pycalendar depends on pyside6 (6.9.0) which
   ↪ requires Python <3.14,>=3.9, version solving failed.

  * Check your dependencies Python requirement: The
     ↪ Python requirement can be specified via the `
     ↪ python` or `markers` properties

    For pyside6, a possible solution would be to set the
       ↪ `python` property to ">=3.13,<3.14"

    https://python-poetry.org/docs/dependency-
       ↪ specification/#python-restricted-dependencies,
    https://python-poetry.org/docs/dependency-
       ↪ specification/#using-environment-markers
```

Il nous faut finalement mettre en adéquation les exigences du paquet et du projet. Modifions simplement les exigences de Python dans `pyproject.toml` :

```
[...]
requires-python = ">=3.13, <3.14"
[...]
```

Et relançons :

```
user@machine: $ poetry add $(cat requirements.txt)
```

L'utilitaire `cat` lit le fichier spécifié et écrit son contenu dans le flux de sortie standard. Ce contenu est passé à la commande `poetry add`, ce qui permet d'installer chaque dépendance répertoriée dans le fichier `requirements.txt` au niveau du projet **Poetry**.

Le fichier `pyproject.toml` s'est vu ajouté les lignes suivantes :

```
dependencies = [
    "pyside6 (==6.9.0)",
    "pyside6-addons (==6.9.0)",
    "pyside6-essentials (==6.9.0)",
```

```
    "shiboken6 (==6.9.0)"
]
```

Mais il est vrai que toutes les dépendances sont lissées, alors que si nous avions créé le projet directement avec **poetry**, puis installé directement nos dépendances, nous aurions simplement :

```
dependencies = [
    "pyside6 (==6.9.0)"
]
```

Mais **Poetry** résoudra cette structure des dépendances en les verrouillant dans le fichier `poetry.lock` correspondant. Visualisation de cette structure des dépendances :

```
user@machine: $ poetry show --tree
```

```
pyside6 6.9.0 Python bindings for the Qt cross-platform application and UI
      framework
|-- pyside6-addons 6.9.0
|   |-- pyside6-essentials 6.9.0
|   |   |-- shiboken6 6.9.0
|   |-- shiboken6 6.9.0 (circular dependency aborted here)
|-- pyside6-essentials 6.9.0
|   |-- shiboken6 6.9.0
|-- shiboken6 6.9.0
pyside6-addons 6.9.0 Python bindings for the Qt cross-platform application
      and UI framework (Addons)
|-- pyside6-essentials 6.9.0
|   |-- shiboken6 6.9.0
|-- shiboken6 6.9.0
pyside6-essentials 6.9.0 Python bindings for the Qt cross-platform
      application and UI framework (Essentials)
|-- shiboken6 6.9.0
shiboken6 6.9.0 Python/C++ bindings helper module
```

Exporter les dépendances depuis Poetry

Poetry permet également de produire un fichier `requirements.txt` via la commande `pip freeze` traditionnelle :

```
user@machine:
    $ poetry run python -m pip freeze > requirements
      .txt
```

Cela est également possible par l'intermédiaire du *plugin* **Export**[6] de **Poetry**, qui permet essentiellement d'exporter les dépendances de `poetry.lock` vers divers formats de fichiers, y compris `requirements.txt`.

Selon la façon dont vous avez installé **Poetry**, voici le choix des commandes pour installer le *plugin* :

```
user@machine: $ poetry self add poetry-plugin-export
user@machine: $  pipx inject poetry poetry-plugin-export
user@machine: $  python -m pip install poetry-plugin-export
```

```
user@machine: $ pipx inject poetry poetry-plugin-export
  injected package poetry-plugin-export into venv poetry
done!
```

Export des dépendances du projet dans un fichier `requirements.txt` :

```
user@machine: $ poetry export --output requirements.txt
```

Et voici le nouveau fichier `requirements.txt` :

```
pyside6-addons==6.9.0 ; python_version == "3.13" \
    --hash=sha256:260a56da59539f476c1635a3ff13591e10f1b04d92155c0617129bc
    53ca8b5f8 \
    --hash=sha256:8cf54065b3d1b4698448fad825378a25c10ef52017d9dff48cead03
    200636d8d \
    --hash=sha256:98f9ad4b65820736e12d49c18db2e570eac63727407fbb59a62ac75
    3e89dc201 \
    --hash=sha256:d8a650644e0b9d1e7a092f6bcd11f25a63706d12f77d442b6ace75d
    346ab5d30 \
    --hash=sha256:fc9dcd63a0ce7565f238cb11c44494435a50eb6cb72b8dbce3b7096
    18989c3dc
pyside6-essentials==6.9.0 ; python_version == "3.13" \
    --hash=sha256:45eaf7f17688d1991f39680dbfd3c41674f3cbb78f278aa10fe0b5f
    2f31c1989 \
    --hash=sha256:69aedfad77119c5bec0005ca31d5620e9bac8ba5ae66c7389160530
    cfd698ed8 \
    --hash=sha256:94a0096d6bb1d3e5cef29ca4a5366d0f229d42480fbb17aa25ad85d
    72b1b7947 \
    --hash=sha256:b18e3e01b507e8a57481fe19792eb373d5f10a23a50702ce540da14
    35e722f39 \
    --hash=sha256:d2dc45536f2269ad111991042e81257124f1cd1c9ed5ea778d7224f
    d65dc9e2b
pyside6==6.9.0 ; python_version == "3.13" \
```

6. https://pypi.org/project/poetry-plugin-export/

```
    --hash=sha256:0103e5d161696db40d75bfbf4e4b7d4f3372903c1b400c4e3379377
    b62c50290 \
    --hash=sha256:09239d1b808f18efccd3803db874d683917efcdebfdf0e8dec449cf
    50e74e7aa \
    --hash=sha256:1a176409dd0dd12b72d2c78b776e5051f569071ec52b7aaadd0a5b3
    333493c24 \
    --hash=sha256:846fbccf0b3501eb31cf0791a46e137615efba6ce540da2b426d79f
    a3e7762c4 \
    --hash=sha256:b8f286a1bd143f3b2bdf08367b9362b13f469d26986c25700af9c4c
    68f79213e
shiboken6==6.9.0 ; python_version == "3.13" \
    --hash=sha256:121ea290ed1afa5ad6abf690b377612693436292b69c61b0f8e10b1
    f0850f935 \
    --hash=sha256:24f53857458881b54798d7e35704611d07f6b6885bcdf80f13a4c8b
    b485b8df2 \
    --hash=sha256:3f585caae5b814a7e23308db0a077355a7dc20c34d58ca4c339ff76
    25e9a1936 \
    --hash=sha256:b61579b90bf9c53ecc174085a69429166dfe57a0b8b894f933d1281
    af9df6568 \
    --hash=sha256:c4d8e3a5907154ac4789e52c77957db95bcf584238c244d7743cb39
    e9b66dd26
```

Le fichier résultant inclut par défaut des hachages et des marqueurs d'environnement, ce qui signifie que nous pouvons travailler avec des exigences très strictes qui ressemblent au contenu du fichier `poetry.lock`.

Si dans un fichier `requirements.txt` on souhaite y ajouter des dépendances de groupes facultatifs (par exemple **dev** et **test**) :

```
user@machine:
    $ poetry export --output requirements.txt --with
     dev,test
```

Pour visualiser les options possibles du *plugin* :

```
user@machine: $ poetry export --help
```

5.6 Les principales commandes de *Poetry*

Pour obtenir la liste de toutes les commandes :

```
user@machine: $ poetry list
```

Commande	Description
`$ poetry -version`	Montre la version **Poetry**.
`$ poetry new`	Créer un nouveau projet **Poetry**.
`$ poetry init`	Ajoute **Poetry** à un projet existant.
`$ poetry run`	Exécuter une commande dans un environnement virtuel géré par **Poetry**.
`$ poetry add`	Ajoute un paquet à `pyproject.toml` et l'installe.
`$ poetry update`	Mise à jour des dépendances du projet.
`$ poetry install`	Installe les dépendances.
`$ poetry show`	Liste les paquets installés.
`$ poetry lock`	Épingle la dernière version des dépendances dans le fichier `poetry.lock`.
`$ poetry lock -no-update`	Actualise le fichier `poetry.lock` sans mettre à jour aucune version de dépendance.
`$ poetry check`	Valide `pyproject.toml`.
`$ poetry config -list`	Montre la configuration de **Poetry**.
`$ poetry env list`	Liste les environnements virtuels du projet.
`$ poetry export`	Exporte `poetry.lock` vers d'autres formats.

Poetry peut également aider à construire et publier des paquets Python[7].

Grâce à **poetry**, nous pouvons désormais gérer nos dépendances avec précision et fluidité, nous permettant de nous concentrer sur l'essentiel.

Cependant, le paysage des outils de développement Python ne cesse d'évoluer, et de nouvelles solutions émergent pour répondre aux divers besoins des développeurs. C'est pourquoi, dans le chapitre suivant, nous nous plongerons dans les fonctionnalités de **uv** et découvrirons comment cet outil peut compléter et enrichir notre boîte à outils de développement Python.

7. Voir « *Comment publier un paquet Python open-source vers PyPI* » : https://realpython.com/pypi-publish-python-package/#poetry

6

uv
Pour une gestion avancée des environnements de développement

> *"Il y a un moyen de faire mieux, trouvez-le."*
> — Thomas Edison

uv est un gestionnaire de dépendances ainsi qu'un outil de *packaging* conçu pour offrir une performance optimale, tout en se révélant être d'une grande simplicité au niveau de son utilisation. **uv** se distingue par sa capacité à résoudre rapidement (écrit en **Rust**[1] une attention fut portée sur la rapidité d'exécution) les dépendances et à créer des environnements virtuels de manière transparente.

Dans ce chapitre, nous allons explorer les grandes lignes de l'outil **uv** et découvrir comment il peut transformer notre approche du développement Python. Nous commencerons par une introduction à ses fonctionnalités de base, puis nous plongerons dans des aspects plus avancés, tels que la gestion des environnements virtuels, la résolution des dépendances, et le *packaging* de projets, optimisant ainsi notre flux de travail.

Cet outil saura certainement devenir un allié incontournable, voire indispensable, pour notre boîte à outils de développement Python.

1. https://www.rust-lang.org/fr

6.1 Présentation d'*uv*

L'idée principale d'**uv** est d'accélérer le flux en accélérant les actions de gestion de projet. Par exemple, pour l'installation de paquets, **uv** est dix à cent fois plus rapide que **pip**.

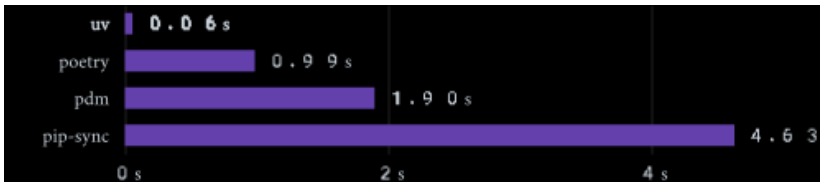

FIGURE 6.1 – Graphique illustrant la rapidité, tiré de la documentation officielle

En outre, **uv** intègre dans un seul outil la plupart des fonctionnalités fournies par des outils tels que **pip**, **pip-tools**, **pipx**, **poetry**, **pyenv**, **twine**[2], **virtualenv**, et bien d'autres encore. **uv** est donc une solution tout-en-un.

Caractéristiques d'**uv** :
— Installation rapide des dépendances
— Gestion des environnements virtuels
— Gestion des versions de Python
— Initialisation du projet : Permet d'échafauder un projet Python complet, y compris le répertoire racine, le dépôt **Git**, l'environnement virtuel, `pyproject.toml`, `README`, etc.
— Gestion des dépendances pour une reproductibilité de l'environnement
— Gestion de la construction et de la publication des paquets
— Prise en charge des outils de développement : Installe et permet d'exécuter des outils de développement, tels que **pytest**[3], **Black**[4] et **Ruff**[5].

Site officiel : https://docs.astral.sh/uv/

Serveur **Discord** : https://discord.gg/JSsj35HH

6.2 Installation

La façon la plus rapide (bien plus rapide) d'installer **uv** est d'utiliser l'installateur autonome fournit par le projet **uv**. Une autre option est d'installer **uv** à partir de **PyPI** en utilisant d'autres outils comme **pipx** ou **pip**[6].

Pour installer la dernière version d'**uv** (sous forme de binaires) :

2. https://twine.readthedocs.io/en/stable/index.html
3. https://docs.pytest.org/en/stable/
4. https://pypi.org/project/black/
5. https://docs.astral.sh/ruff/
6. Pour d'autres modes d'installation voir la documentation officielle : https://docs.astral.sh/uv/getting-started/installation/

```
user@machine:$ curl -LsSf https://astral.sh/uv/install.sh |
   ↪ sh
downloading uv 0.7.8 x86_64-unknown-linux-gnu
no checksums to verify
installing to /home/user/.local/bin
  uv
  uvx
everything's installed!
```

Il est cependant possible d'obtenir une version spécifique d'**uv**[7] en incluant le numéro de version dans l'*URL* :

```
user@machine:$ curl -LsSf https://astral.sh/uv/0.7.7/install
   ↪ .sh | sh
downloading uv 0.7.7 x86_64-unknown-linux-gnu
no checksums to verify
installing to /home/krystof/.local/bin
  uv
  uvx
everything's installed!
```

Vérifier l'installation :

```
user@machine:$ uv --version
uv 0.7.7
```

Activer l'auto-complétion dans **zsh**, à la fois pour **uv** et **uvx** :

```
user@machine:$ echo 'eval "$(uv generate-shell-completion
   ↪ zsh)"' >> ~ /.zshrc
user@machine:$ echo 'eval "$(uvx generate-shell-completion
   ↪ zsh)"' >> ~/.zshrc
```

Mise à jour d'*uv*

Comme le projet uv est en développement actif, de nouvelles versions sont régulièrement publiées. Si **uv** a été installé à l'aide de l'installateur autonome il sera nécessaire d'exécuter la commande suivante :

```
user@machine:$ uv self update
info: Checking for updates...
```

7. Voir la page **GitHub** des différentes versions : https://github.com/astral-sh/uv/releases

```
success: Upgraded uv from v0.7.7 to v0.7.8! https://
   ↪ github.com/astral-sh/uv/releases/tag/0.7.8
```

Désinstallation

Nettoyer les données stockées :

```
user@machine:$ uv cache clean
No cache found at: .cache/uv  # Normal au début
user@machine:$ rm -r "$(uv python dir)"
user@machine:$ rm -r "$(uv tool dir)"
```

Supprimer les binaires **uv** et **uvx** :

```
user@machine:$ rm ~/.local/bin/uv ~/.local/bin/uvx
user@machine:$ uv --version
zsh: command not found: uv
```

Un aperçu rapide

Maintenant qu'**uv** est installé prenons le temps de visualiser très rapidement ce que cet outil a à nous proposer :

```
user@machine:$ uv
An extremely fast Python package manager.

Usage: uv [OPTIONS] <COMMAND>

Commands:
  run      Run a command or script
  init     Create a new pyproject
# Saisir la commande pour voir la suite de la sortie...
```

Pour accéder à l'aide d'une commande d'**uv** :

```
user@machine:$ uv --help <commande>
```

Par exemple, la commande cache :

```
user@machine:$ uv --help cache
```

Cela permet d'ouvrir un visualisateur dans lequel on retrouve les informations sollicitées (ce qui suit n'est qu'un extrait) :

```
Manage uv's cache

Usage: uv cache [OPTIONS] <COMMAND>

Commands:
  clean  Clear the cache, removing all entries or those
    ↪ linked to specific packages
  prune  Prune all unreachable objects from the cache
  dir    Show the cache directory
[...]
```

6.3 La gestion de projet

Pour cette section et la suite du didacticiel, nous utiliserons un exemple d'application qui permet d'afficher un calendrier à l'aide d'une interface graphique créée via le *framework* **Qt for Python**[8]. Le code de cette application se trouve sur un dépôt **GitHub**[9]. Il s'agit d'un petit projet personnel réalisé au cours de mon apprentissage du *framework*, mais rien ne vous empêche de tester toutes les manipulations qui vont suivre à partir de tout autre projet. Le projet **Calendar** ne me sert qu'à illustrer mon propos.

Créer et travailler sur des projets Python avec **uv**, revient à travailler avec le fichier `pyproject.toml`, fichier qui, comme nous l'avons vu dans les chapitres précédents, définit les dépendances.

Création d'un projet

Pour créer et initialiser un projet Python avec **uv**, naviguer jusqu'au répertoire dans lequel on souhaite stocker le projet (par exemple le répertoire personnel de l'utilisateur), puis exécuter la commande :

```
user@machine:$ uv init mon_projet
Initialized project ‘mon_projet‘ at ‘/home/utilisateur/
  ↪ mon_projet‘
```

ou bien :

```
user@machine:$ mkdir mon_projet
```

8. https://doc.qt.io/qtforpython-6/, alias **PySide6** : https://pypi.org/project/PySide6/
9. https://github.com/Krystof2so/PyCalendar - Voir également l'annexe *Code du projet* ***Calendar*** page 87

```
user@machine:$ cd mon_projet
user@machine:$ uv init
Initialized project ‘mon_projet‘
```

Cela crée dans le répertoire du projet (`mon_projet`) la structure suivante :

```
mon_projet/
|
|--.python-version
|--README.md
|--main.py
|--pyproject.toml
```

Le fichier `.python-version` contient la version de Python par défaut pour le projet en cours. Ce fichier indique à **uv** la version de Python à utiliser lors de la création d'un environnement virtuel dédié au projet. Ensuite, il y a un fichier `README.md`. Et l'on trouve un fichier `main.py` qui contient les lignes suivantes :

```
1 def main():
2     print("Hello from mon-projet!")
3
4
5 if __name__ == "__main__":
6     main()
```

Et au final un fichier `pyproject.toml` initié avec une configuration basique :

```
1 [project]
2 name = "mon-projet"
3 version = "0.1.0"
4 description = "Add your description here"
5 readme = "README.md"
6 requires-python = ">=3.13"
7 dependencies = []
```

Nous pouvons changer la description :

```
1 description = "Un simple projet"
```

Initier un projet existant

Nous allons pour cela utiliser le projet **Calendar**[10]. Il suffit tout simplement de se rendre dans le répertoire du projet et de lancer la commande `init` :

```
user@machine:$ uv init
Initialized project ‘calendar‘
user@machine:$ tree
.
|-- IMG
|   |-- PyCalendar.png
|-- main.py
|-- pyproject.toml
|-- README.md
|-- src
|-- pycalendar
        |-- main.py

4 directories, 5 files
```

Cette commande va seulement générer le fichier `pyproject.toml`. Elle ne va ni écraser, ni modifier les fichiers existants, ni modifier la structure du projet. Cependant, cette commande ne fonctionnera pas si un fichier `pyproject.toml` est déjà présent. Si c'est le cas, copier le fichier avant de lancer la commande `uv init`, il pourra servir pour modifier le nouveau fichier généré avec toute configuration pertinente de l'ancien `pyproject.toml`.

Exécution du script d'entrée du projet

```
user@machine:$ uv run main.py
Using CPython 3.13.3 interpreter at: /usr/bin/python3.13
Creating virtual environment at: .venv
Hello from calendar!
user@machine:$ tree -a
.
|-- main.py
|-- pyproject.toml
|-- .python-version
|-- README.md
|-- uv.lock
|-- .venv
    |-- bin
    |   |-- activate
    |   |-- activate.bat
    |   |-- activate.csh
```

10. Cf. annexe *Code du projet* ***Calendar*** page 87

```
    |   |-- activate.fish
    |   |-- activate.nu
    |   |-- activate.ps1
    |   |-- activate_this.py
    |   |-- deactivate.bat
    |   |-- pydoc.bat
    |   |-- python -> /usr/bin/python3.13
    |   |-- python3 -> python
    |   |-- python3.13 -> python
    |-- CACHEDIR.TAG
    |-- .gitignore
    |-- lib
    |   |-- python3.13
    |       |-- site-packages
    |           |-- __pycache__
    |           |   |-- _virtualenv.cpython-313.pyc
    |           |-- _virtualenv.pth
    |           |-- _virtualenv.py
    |-- lib64 -> lib
    |-- pyvenv.cfg

8 directories, 23 files
```

La première fois que l'on exécutez une commande de projet, soit `uv run`, `uv sync` (création de l'environnement du projet) ou `uv lock`, la version de Python utilisée pour le projet est affichée. **uv** crée ensuite un environnement virtuel et un fichier `uv.lock` (cf. infra) à la racine du projet, ainsi qu'un fichier `.gitignore`, installé dans le répertoire de gestion de l'environnement virtuel (`.venv`) dont le contenu est vide.

Si l'on ne souhaite pas qu'**uv** gère l'environnement du projet, désactiver le verrouillage et la synchronisation automatiques du projet dans `pyproject.toml` :

```
[tool.uv]
managed = false
```

Ce qui donnera à la première exécution :

```
user@machine:$ uv run main.py
Hello from mon_projet!
user@machine:$ tree -a
.
|-- main.py
|-- pyproject.toml
|-- .python-version
|-- README.md

1 directory, 4 files
```

6.4 La gestion des dépendances

Ajout et installation des dépendances

Revenons au projet **Calendar** :

```
user@machine:$ cd calendar
user@machine:$ uv init
Initialized project ‘calendar‘
user@machine:$ uv run src/pycalendar/main.py
Using CPython 3.13.3 interpreter at: /usr/bin/python3.13
Creating virtual environment at: .venv
Traceback (most recent call last):
  File "/home/utilisateur/calendar/src/pycalendar/main.py
    ↪ ", line 4, in <module>
    from PySide6.QtCore import QSize, Qt
ModuleNotFoundError: No module named 'PySide6'
```

L'exception levée est sans appel : il nous manque un module nécessaire pour exécuter correctement notre application, le module **PySide6**. Remédions à cela :

```
user@machine:$ uv add pyside6
Resolved 5 packages in 443ms
Prepared 4 packages in 10.63s
Installed 4 packages in 146ms
 + pyside6==6.9.0
 + pyside6-addons==6.9.0
 + pyside6-essentials==6.9.0
 + shiboken6==6.9.0
```

Ou bien en spécifiant une version précise du module :

```
user@machine:$ uv add "pyside6==6.9.0"
Resolved 10 packages in 16ms
Installed 4 packages in 117ms
 + pyside6==6.9.0
 + pyside6-addons==6.9.0
 + pyside6-essentials==6.9.0
 + shiboken6==6.9.0
```

Ces commandes permettent d'ajouter la bibliothèque **PySide6** aux dépendances du projet et de l'installer dans l'environnement virtuel. Le fichier `pyproject.toml` est alors modifié en conséquence :

```
[project]
name = "calendar"
version = "0.1.0"
description = "Simple calendar"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "pyside6>=6.9.0",
]
```

Il est même possible d'ajouter une dépendance depuis un projet **GitHub** (par exemple , le module **requests**) :

```
user@machine:$ uv add git+https://github.com/psf/requests
    Updated https://github.com/psf/requests (
      ↪ c65c780849563c891f35ffc98d3198b71011c012)
Resolved 15 packages in 364ms
      Built requests @ git+https://github.com/psf/
        ↪ requests@c65c780849563c891f35ffc98d3198b71011c
        ↪  012
Prepared 5 packages in 891ms
Installed 5 packages in 7ms
 + certifi==2025.4.26
 + charset-normalizer==3.4.2
 + idna==3.10
 + requests==2.32.3 (from git+https://github.com/psf/
   ↪ requests@c65c780849563c891f35ffc98d3198b71011c012)
 + urllib3==2.4.0
```

Si le projet possédait déjà un fichier `requirements.txt` :

```
PySide6==6.9.0
PySide6_Addons==6.9.0
PySide6_Essentials==6.9.0
shiboken6==6.9.0
```

Il suffisait alors d'en passer par lui :

```
user@machine:$ uv add -r requirements.txt
Using CPython 3.13.3 interpreter at: /usr/bin/python3.13
Creating virtual environment at: .venv
Resolved 5 packages in 1ms
Installed 4 packages in 91ms
 + pyside6==6.9.0
 + pyside6-addons==6.9.0
 + pyside6-essentials==6.9.0
```

```
 + shiboken6==6.9.0
```

Et notre nouveau `pyproject.toml` :

```
[project]
name = "calendar"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "pyside6==6.9.0",
    "pyside6-addons==6.9.0",
    "pyside6-essentials==6.9.0",
    "shiboken6==6.9.0",
]
```

uv add met également à jour le fichier `uv.lock` avec les informations de version pour les éléments suivants :

Dépendances directes : paquets dont le projet dépend directement (**pyside6**).

Dépendances transitives : paquets qui prennent en charge les dépendances directes du projet (les trois autres).

`uv add` fait intégralement le travail de gestion des dépendances en les installant, en éditant le fichier `pyproject.toml` et en maintenant le fichier `uv.lock` à jour. Cela garantit une reproductibilité de l'environnement de travail.

Une fois les dépendances installées, nous pouvons relancer notre script.

Mise à jour et suppression des dépendances

Mise à jour d'un paquet :

```
user@machine:$ uv add --upgrade pyside6  # Ici nous avons dé
   ↪ jà la dernière version
Resolved 5 packages in 158ms
Audited 4 packages in 0.97ms
```

Ou bien :

```
user@machine:$uv lock --upgrade-package pyside6
Resolved 15 packages in 756ms
```

Suppression d'une dépendance :

```
user@machine:$ uv remove pyside6
Resolved 1 package in 11ms
Uninstalled 4 packages in 66ms
 - pyside6==6.9.0
 - pyside6-addons==6.9.0
 - pyside6-essentials==6.9.0
 - shiboken6==6.9.0
```

Il est important de noter que `uv remove` supprime également les dépendances transitives et met à jour les fichiers `pyproject.toml` et `uv.lock` en conséquence.

Les dépendances de développement

Les bibliothèques de test comme **pytest**, les formateurs de code comme **Ruff** et les vérificateurs de types statiques comme **mypy** font partie de ces dépendances de développement. On les installe à l'aide de la commande `uv add --dev paquet` :

```
user@machine:$ uv add --dev pytest
Resolved 10 packages in 335ms
Prepared 4 packages in 137ms
Installed 4 packages in 21ms
 + iniconfig==2.1.0
 + packaging==25.0
 + pluggy==1.5.0
 + pytest==8.3.5
```

Cette commande installe **pytest** dans l'environnement virtuel du projet et ajoute la bibliothèque comme dépendance de développement aux fichiers `pyproject.toml` et `uv.lock`. Lignes ajoutées dans `pyproject.toml` :

```
[dependency-groups]
dev = [
    "pytest>=8.3.5",
]
```

Verrouillage et synchronisation de l'environnement

Le verrouillage consiste à capturer les dépendances spécifiques du projet dans le fichier `uv.lock` qui est un fichier de verrouillage multi-plateforme contenant des informations précises sur les dépendances du projet, c'est-à-dire les versions exactes résolues qui sont installées dans l'environnement du projet. Ce fichier doit être vérifié dans le contrôle de version, ce qui permet des installations cohérentes et reproductibles sur toutes les machines. Ce processus permet donc de reproduire un environnement de travail dans toutes les configurations

possibles, y compris la version et la distribution de Python, le système d'exploitation et l'architecture. Ce fichier est directement géré par **uv**. Par exemple, lorsque l'on exécute `uv run`, le projet est verrouillé et synchronisé avant que la commande ne soit invoquée. Ce comportement garantit que l'environnement du projet reste toujours à jour. Il sera nécessaire d'enregistrer ce fichier dans le système de contrôle de version. Un tel fichier ne doit pas être édité manuellement.

Nous pouvez explicitement verrouiller et synchroniser notre projet avec les commandes suivantes :

```
user@machine:$ uv lock
Resolved 5 packages in 0.93ms
$ uv sync
Resolved 5 packages in 1ms
Audited 4 packages in 0.01ms
```

Ces commandes sont utiles si nous rencontrons des problèmes lors de l'exécution du projet et pour s'assurer que nous utilisons la bonne version de chaque dépendance.

Imaginons un projet cloné depuis un dépôt distant (comme **GitHub**), qui ne contient donc pas le répertoire de l'environnement virtuel. Il nous suffit de lancer depuis le répertoire racine du projet :

```
user@machine:$ uv run src/pycalendar/main.py
Using CPython 3.13.3 interpreter at: /usr/bin/python3.13
Creating virtual environment at: .venv
Installed 13 packages in 96ms
```

Avant d'essayer d'exécuter l'application, **uv** crée un environnement virtuel dans le répertoire `.venv/`. Il installe ensuite les dépendances et exécute enfin l'application.

Pour vérifier quels modules sont présents au niveau de notre environnement virtuel :

```
user@machine:$ uv pip list
Package            Version
------------------ -------
pyside6            6.9.0
pyside6-addons     6.9.0
pyside6-essentials 6.9.0
shiboken6          6.9.0
```

Nous pouvons comparer cette liste avec le contenu du fichier `uv.lock`.

Note : une visualisation sous la forme d'une arborescence (tout en mettant le fichier de verrouillage à jour) :

```
user@machine:$ uv tree
```

```
Resolved 5 packages in 1ms
calendar v0.1.0
|-- pyside6 v6.9.0
    |-- pyside6-addons v6.9.0
    |   |-- pyside6-essentials v6.9.0
    |   |     |-- shiboken6 v6.9.0
    |   |-- shiboken6 v6.9.0
    |-- pyside6-essentials v6.9.0 (*)
    |-- shiboken6 v6.9.0
(*) Package tree already displayed
```

Les options de désactivation et de non-vérification

Pour désactiver le verrouillage automatique, utiliser l'option `-locked` :

```
user@machine:$ uv run src/pycalendar/main.py --locked
user@machine:$
```

Si le fichier de verrouillage n'est pas à jour, **uv** lèvera une erreur au lieu de mettre à jour le fichier de verrouillage.

Utiliser le fichier de verrouillage sans vérifier s'il est à jour :

```
user@machine:$ uv run src/pycalendar/main.py --frozen
user@machine:$
```

De même, pour exécuter une commande sans vérifier si l'environnement est à jour :

```
user@machine:$ uv run --no-sync src/pycalendar/main.py
user@machine:$
```

Le fichier *pylock.toml*

A ce sujet on se réfèrerar au **PEP 751**[11].

`pylock.toml` est un format de sortie de résolution destiné à remplacer `requirements.txt`. `pylock.toml` est standardisé et agnostique, de sorte qu'à l'avenir, les fichiers `pylock.toml` générés par **uv** pourraient être installés par d'autres outils, et vice-versa. Mais ce format demeure encore expérimental et n'est pas pleinement opérationnel.

11. Un format de fichier pour enregistrer les dépendances de Python pour la reproductibilité de l'installation : https://peps.python.org/pep-0751/

Pour exporter un `uv.lock` au format `pylock.toml` (nécessite l'existence du fichier `pyproject.toml`) :

```
user@machine:$ uv export -o pylock.toml
Resolved 5 packages in 2ms
# This file was autogenerated by uv via the following
   ↪ command:
#    uv export -o pylock.toml
lock-version = "1.0"
created-by = "uv"
requires-python = ">=3.13"

[[packages]]
name = "pyside6"
version = "6.9.0"
index = "https://pypi.org/simple"
[...]
```

6.5 Les commandes

Commandes	Description
`uv init`	Créer un nouveau projet Python.
`uv add`	Ajouter une dépendance au projet.
`uv remove`	Supprime une dépendance du projet.
`uv sync`	Synchronise les dépendances du projet avec l'environnement.
`uv lock`	Créer un fichier de verrouillage pour les dépendances du projet.
`uv run`	Exécuter une commande dans l'environnement du projet.
`uv tree`	Affiche l'arbre des dépendances du projet.
`uv build`	Construire le projet dans des archives de distribution.
`uv publish`	Publier le projet dans un index de paquets.

TABLE 6.1 – Commandes et leur description pour créer un projet Python avec **uv**

Nous venons d'explorer en détail les multiples facettes de l'outil **uv**, un outil innovant qui transforme notre approche de la gestion des dépendances et des environnements de développement Python. Que l'on travaille sur des projets de petite ou de grande envergure, **uv** s'adapte à nos besoins et nous permet de nous concentrer sur l'essentiel : le développement.

En intégrant **uv** à notre boîte à outils de développement, nous optimisons notre flux de travail. Les fonctionnalités avancées de **uv**, telles que la résolution rapide des dépendances et la gestion simplifiée des environnements virtuels, en font un allié incontournable pour tout développeur Python soucieux d'efficacité et de qualité.

Annexes

Code du projet *Calendar*

Voici le code Python du projet **Calendar**[12] :

```
import calendar

from datetime import datetime
from PySide6.QtCore import QSize, Qt
from PySide6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout,\
                              QGridLayout, QLabel, QPushButton
from PySide6.QtGui import QPalette, QColor


DAYS_OF_WEEK = ("Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche")


class CalendarApp(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("PyCalendar")
        self.setGeometry(100, 100, 400, 300)

        # Initialiser la date actuelle :
        self.current_date = datetime.now()

        # Layout vertical principal :
        main_layout = QVBoxLayout()
        self.setLayout(main_layout)
```

12. Egalement disponible sur **Github** : https://github.com/Krystof2so/PyCalendar. Il se peut que sur le dépôt **Github** le code différe, le principe reste cependant le même, mais vous pouvez aussi utiliser l'exemple tel que fourni dans cette annexe

```
26
27          # Layout horizontal pour afficher le mois en
              ↪ cours et les boutons de navigation:
28          month_layout = QHBoxLayout()
29          main_layout.addLayout(month_layout)
30
31          # Bouton pour le mois précédent :
32          self.prev_button = QPushButton("<")
33          self.prev_button.setFixedSize(QSize(30, 30))  # D
              ↪ éfinir une taille fixe pour le bouton
34          self.prev_button.clicked.connect(self.
              ↪ show_previous_month)
35          month_layout.addWidget(self.prev_button)
36
37          # QLabel pour afficher le mois en cours :
38          self.month_label = QLabel(self.
              ↪ get_current_month_year())
39          self.month_label.setAlignment(Qt.AlignCenter)
40          month_layout.addWidget(self.month_label)
41
42          # Bouton pour le mois suivant :
43          self.next_button = QPushButton(">")
44          self.next_button.setFixedSize(QSize(30, 30))  # D
              ↪ éfinir une taille fixe pour le bouton
45          self.next_button.clicked.connect(self.
              ↪ show_next_month)
46          month_layout.addWidget(self.next_button)
47
48          # Ajouter la grille du calendrier :
49          self.calendar_layout = QGridLayout()
50          main_layout.addLayout(self.calendar_layout)
51
52          # Remplir la grille avec les jours du mois :
53          self.populate_calendar()
54
55      def get_current_month_year(self):
56          return self.current_date.strftime("%B %Y")
57
58      def is_today(self, day):
59          """Vérifie si le jour donné est aujourd'hui."""
60          today = datetime.now()
61          return day == today.day and self.current_date.
              ↪ month == today.month and self.current_date.
              ↪ year == today.year
62
63      def populate_calendar(self):
64          """Obtenir les jours du mois, et les placer dans
              ↪ la grille."""
```

```
65          year = self.current_date.year
66          month = self.current_date.month
67          cal = calendar.monthcalendar(year, month)
68          # Effacer la grille précédente :
69          self.clear_layout(self.calendar_layout)
70          # Ajouter les jours de la semaine :
71          for col, day_label in enumerate(DAYS_OF_WEEK):
72              label = QLabel(day_label[0:2])
73              label.setAlignment(Qt.AlignCenter)
74              self.calendar_layout.addWidget(label, 0, col)
75          # Ajouter les jours du mois :
76          for row, week in enumerate(cal):
77              for col, day in enumerate(week):
78                  label = QLabel("" if day == 0 else str(
                        ↪ day))
79                  label.setAlignment(Qt.AlignCenter)
80                  if self.is_today(day):
81                      palette = label.palette()
82                      palette.setColor(QPalette.WindowText,
                            ↪  QColor('#A3BE8C'))  # Vert du
                            ↪ thème Nord
83                      label.setPalette(palette)
84                  self.calendar_layout.addWidget(label, row
                        ↪  + 1, col)
85
86      def clear_layout(self, layout):
87          """Effacer tous les widgets d'un layout."""
88          while layout.count():
89              child = layout.takeAt(0)
90              if child.widget():
91                  child.widget().deleteLater()
92
93      def show_previous_month(self):
94          """Afficher le mois précédent."""
95          if self.current_date.month == 1:
96              self.current_date = self.current_date.replace
                    ↪ (year=self.current_date.year - 1, month
                    ↪ =12)
97          else:
98              self.current_date = self.current_date.replace
                    ↪ (month=self.current_date.month - 1)
99          self.month_label.setText(self.
                ↪ get_current_month_year())
100         self.populate_calendar()
101
102     def show_next_month(self):
103         """Afficher le mois suivant."""
104         if self.current_date.month == 12:
```

```
105             self.current_date = self.current_date.replace
                    ↪ (year=self.current_date.year + 1, month
                    ↪ =1)
106         else:
107             self.current_date = self.current_date.replace
                    ↪ (month=self.current_date.month + 1)
108         self.month_label.setText(self.
                ↪ get_current_month_year())
109         self.populate_calendar()
110
111 if __name__ == "__main__":
112     app = QApplication([])
113     window = CalendarApp()
114     window.show()
115     app.exec()
```

Voici une image de ce que ce code propose :

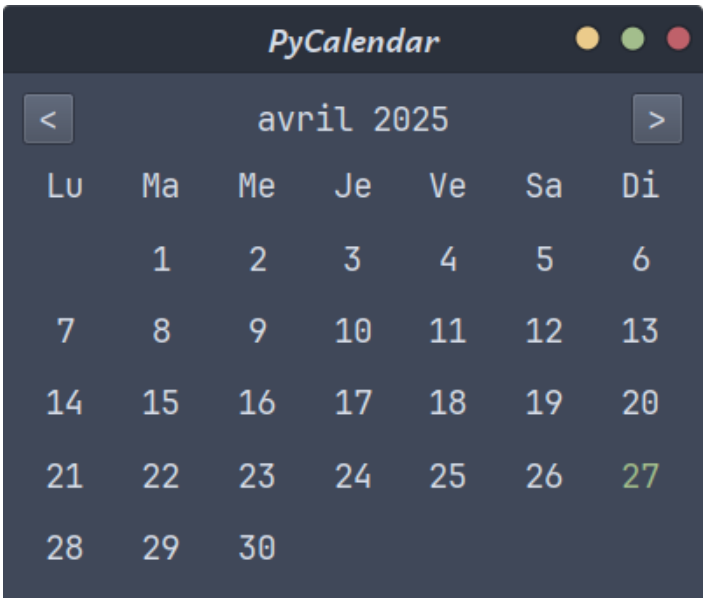

FIGURE 6.2 – Application **Calendar**

L'arborescence du projet est la suivante :

```
Calendar
|-- IMG
|   |-- PyCalendar.png
|-- README.md
|-- src
    |-- pycalendar
        |-- main.py
```