université
PARIS-SACLAY

Programmation en Python
Master 2 Réseaux Télécoms
TP 9

1 Environnements virtuels

Lorsque vous écrivez une application en Python, il est nécessaire de pouvoir la transmettre à un tiers. Le problème survient lorsque votre application utilise des bibliothèques Python externes (qui ne sont pas incluses dans la bibliothèque standard de Python). Il est alors nécessaire de transmettre avec votre code des informations qui permettraient au client de reproduire votre environnement de travail.

En plus, dans la plupart des environnements serveur, le système opérationnel sera du type *Linux* qui contient sa propre version de Python nécessaire pour le bon fonctionnement de plusieurs outils. Si vous essayez d'installer des bibliothèques externes, le serveur risque de devenir instable (Fig. 1) Comment donc faire pour fabriquer un environnement de travail reproduisible qui n'interfère ni avec les bibliothèques Python installées ailleurs dans le système ni avec d'autres applications Python ?

Au cours de l'histoire de Python plusieurs solutions ont été proposées, toutes un peu différentes :

— `pipenv`
— `pipx`
— `virtualenv` : Solution non-officielle recommandée avant Python 3.3
— `conda`
— `venv` : Solution officielle introduite dans Python 3.3

Dans toutes ces solutions, un environnement virtuel est un dossier dans lequel sont contenus des fichiers qui indiquent à Python les bibliothèques qui devraient être installées. Dans cet environnement, les bibliothèques installées ailleurs n'existent pas. En dehors de cet environnement, ses bibliothèques n'existent pas. Autrement dit, un environnement virtuel est un petit monde à part. Lorsque l'on

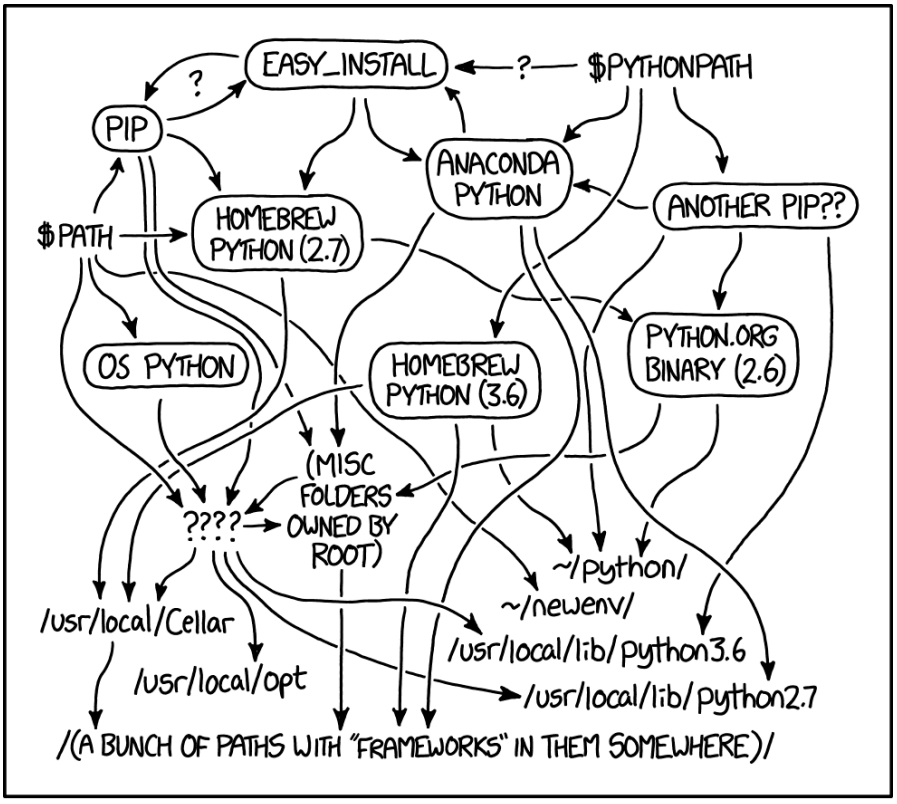

FIGURE 1 – `https://xkcd.com/1987/`

active cet environnement, on oublie l'existence du monde externe pour tout ce qui concerne les bibliothèques installées.

Dans cet exercice, nous allons créer un environnement virtuel de Python en utilisant la solution officielle `venv`.

1. Modifiez les droits d'exécution des scripts dans le Power Shell de Windows. Pour cela, lancez un Power Shell et exécuter la commande suivante :

   ```
   Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
   ```

   Cela est nécessaire, car l'execution du script qui active l'environnement virtuel peut être bloquée par Windows.

2. Créez un répertoire `C:/Temp/envtp9` dans lequel va être contenu notre environnement virtuel.

3. Créez un environnement virtuel dans le répertoire `C:/Temp/envtp9` avec la commande suivante :

```
<path_to_python>/python.exe -m venv C:/Temp/envtp9
```

où `<path_to_python>` est le chemin vers l'interpréteur Python du système. Cela crée les fichiers nécessaires pour que Python considère le dossier comme un environnement virtuel.

4. Dans VSCode, sélectionnez cet environnement comme l'environnement Python avec la commande `Python: Select Interpreter` de la palette de commandes (`Ctrl+Shift+P`). L'interpréteur correspondant se trouve dans le répertoire `C:/Temp/envtp9/Scripts`. Notez que le nom de l'interpréteur doit s'afficher à gauche dans la barre de statut de VSCode.
5. Lancez un terminal dans VSCode. Notez que l'invite de commande est maintenant préfixée par `(envtp9)`.
6. On peut maintenant installer des packages Python quelconques dans cet environnement, par exemple :

   ```
   pip install Faker
   ```

   (pour installer un package qui produit des données falsifiées.)
7. Vous pouvez essayer d'exécuter `pip list` dans cet environnement et dans un invite de commande normal. Comparez le résultat.

2 Un package rudimentaire

Un projet réaliste utilise souvent plusieurs fichiers. Prenons par exemple une application web qui afficherait les positions des avions comme le fait *Flighradar*. Cette application aurait besoin d'effectuer plusieurs tâches dont les codes sont très distincts :

— Communiquer avec les récepteurs radio pour obtenir les positions GPS des avions
— Gérer une base de donnée de tous les avions (avec des photos, modèles d'avion etc...)
— Communiquer avec un client (serveur-client)
— Produire une carte interactive à envoyer à chaque client

On pourrait alors s'imaginer une structure comme suit :

```
Flighttracker
├── RadioCommunication.py
├── Database.py
├── Server.py
└── MapRender.py
```

Chaque fichier Python sera automatiquement aussi un *module*. Le nom du module sera juste le nom du fichier sans l'extension `.py`. Ainsi le dossier `Flighttracker` contient 4 modules : `RadioCommunication`, `Database`, `Server`, `MapRender`. Un *package* Python est juste un dossier contenant plusieurs modules avec quelques fichiers supplémentaires qui indiquent quels modules (et eventuellement quelles fonctions de chaque module) sont rendues visibles à l'utilisateur.

Dans cet exercice, nous allons installer un package rudimentaire de Python à partir de son répertoire source dans l'environnement `envtp9`.

1. Décompressez le fichier `packdir.zip` dans le repertoire courant du Shell de VSCode et lancez la commande suivante (prenez soin que l'environnement `envtp9` soit bien activé) :

   ```
   pip install packdir/tp9package
   ```

2. Vérifiez que le package est apparu dans le sous-répertoire `lib/python3.9/site-packages` de l'environnement `envtp9`. Vérifiez que `pip list` l'affiche bien.
3. Lancez un REPL de Python et vérifiez que l'on peut importer le package `tp9package`. Testez que la fonction `hello()` est accessible. Qu'en est-il de la fonction `boo()` ?
4. Regardez le contenu de `__init__.py` du dossier `tp9package`. En vous souvenant que les noms des modules sont les noms des fichiers sans l'extension, proposez une explication à ce que vous avez observé dans la question précédente.

3 Base de données analytique

Comme mentionné dans l'exercice précédent, il est souvent nécessaire pour une application web de communiquer avec une base de données. Dans certains cas il est possible d'utiliser un *ORM* (Object–relational mapping) : une traduction automatique des tables de la base de données en des objets Python. Ce n'est pas toujours la solution préférée et aujourd'hui on va écrire une application qui va communiquer avec une base de données SQLite en y envoyant directement des commandes SQL.

Nous allons donc créer une interface Python à une base de donnée analytique représentant le Code Officiel Géographique français. On peut trouver l'information sur le modèle de données sur le site de l'INSEE :
`https://www.insee.fr/fr/information/2560705`
Le fichier `cog.db` contient une base de données SQLite avec deux tables, `regions` et `departements`, avec la structure suivante :

```
CREATE TABLE "regions" (
"REGION" INTEGER NOT NULL,
"CHEFLIEU" TEXT NOT NULL,
"TNCC" INTEGER NOT NULL,
"NCC" TEXT NOT NULL,
"NCCENR" BLOB NOT NULL,
PRIMARY KEY("REGION")
);

CREATE TABLE "departements" (
"REGION" INTEGER NOT NULL,
"DEP" TEXT NOT NULL,
"CHEFLIEU" TEXT NOT NULL,
"TNCC" INTEGER NOT NULL,
"NCC" TEXT NOT NULL,
"NCCENR" BLOB NOT NULL,
PRIMARY KEY("DEP"),
FOREIGN KEY(REGION) REFERENCES regions(REGION)
);
```

Complétez le code des méthodes `department_name` et `department_set` de la classe `cog` :

1. La méthode `department_name` prend comme argument le code du département (attention, ce code n'est pas forcement un nombre !) et retourne le nom du département en typographie riche (majuscules, minuscules, accentuation). Notez bien que la commande `self.conn.text_factory = bytes` a comme effet que la connexion `conn` retourne des chaines de caractères en format binaire. Il convient donc à les décoder en utilisant l'encodage `'latin-1'` (vous pouvez vous inspirer par le code de la méthode `region_name`). Si le code ne correspond à aucun département, la méthode doit lever une `ValueError`.
2. La méthode `department_set` prend comme argument le nom d'une région en lettres majuscules, et retourne un ensemble de noms de départements de cette région en typographie riche. Si l'argument ne correspond à aucune région, la méthode doit retourner un ensemble vide.

On peut tester le code avec le test unitaire `test_cog.py`.