Création d'un environnement conda

Conda correspond à un gestionnaire de paquets et d'environnements de programmation, développé en particulier pour **Python**. Il fonctionne également pour d'autres langages, R notamment.

Il est avantageux d'utiliser **conda** plutôt qu'une distribution classique de **Python**, et ce pour différentes raisons :

- **Versionnage simplifié**, avec une gestion automatique des versions des paquets, des conflits, mises à jour, etc.
- Possibilité de récupérer des **paquets distribués sous différents formats** (wheel, compilé, pré-compilé, etc.) et de les installer sans grande difficulté. En effet, certaines bibliothèques, notamment celles utilisées pour le traitement d'images ou l'intelligence artificielle, requièrent de les compiler avant de pouvoir les utiliser. Cette étape, pas nécessairement simple à réaliser selon les cas, est ici complètement prise en charge par **conda**.
- **Création d'environnements de programmation** hermétiques. Ces environnements correspondent à des écosystèmes Python indépendants les uns des autres. Il est donc possible d'en créer une multitude, avec différentes versions de Python (e.g. l'un en 3.7 et l'autre en 2.7), différents paquets, les mêmes paquets mais avec des versions différentes, etc. Il est ainsi d'usage de créer un environnement spécifique à un projet par exemple, contenant le strict minimum pour faire tourner les programmes développés.
- **Distribution des environnements** auprès d'autres utilisateurs ou collaborateurs. **Conda** permet de créer un fichier dans lequel sont recensées l'ensemble des caractéristiques d'un environnement. Une autre personne peut donc l'installer pour avoir exactement le même environnement de travail, et lancer les programmes sans risque de rencontrer des erreurs de compatibilité ou de dépendances manquantes.

Conda est disponible dans plusieurs **distributions**, les deux principales étant :

- **Anaconda** [https://www.anaconda.com/] – Une distribution qui fournit un environnement de base très complet. Elle est en revanche assez lourde lors du téléchargement et des mises à jour.
- **Miniconda** [https://docs.conda.io/en/latest/miniconda.html] – Une distribution minimaliste qui donne juste ce qu'il faut pour démarrer avec **conda**.

Téléchargez et installez la distribution de votre choix. Lors de l'installation, veillez bien à accepter l'ajout de l'utilitaire **conda** au *path* du système. Cela permettra de pouvoir utiliser l'outil depuis un terminal. Par précaution, redémarrez la machine afin de sauvegarder les changements apportés à la machine. Une fois cette étape terminée, ouvrez une invite de commande (Windows) ou un terminal (Linux/macOS) et tapez la ligne suivante pour vérifier l'intégrité de l'installation :

```
conda
```

Une liste des arguments disponibles pour utiliser **conda** devrait s'afficher. Le logiciel est correctement installé ! Si cela ne fonctionne pas, vous avez potentiellement répondu négativement à l'ajout de **conda** au *path* du système. Retentez alors l'installation. Si vous êtes sur **Windows**, essayez sinon de passer par **Anaconda Prompt** plutôt que par l'invite de commande.

Enfin, terminez par une mise à jour de l'outil :

```
conda update -n base conda
```

Installation de l'environnement préparé

Dans le cadre de cette formation, un environnement de programmation `py_prog_niv2.txt` a été préparé. Celui-ci contient l'ensemble des paquets nécessaires à la réalisation des exercices. Vous pouvez par ailleurs l'utiliser dans le cadre de vos travaux personnels.

Pour installer cet environnement, il suffit d'utiliser la commande suivante :

```
conda env create --name py_prog_niv2 --file /chemin/vers/mon/py_prog_niv2.
    txt
```

Une fois l'environnement installé, vous pouvez commencer à l'utiliser. Pour cela, il faut d'abord l'activer pour ensuite accéder à l'écosystème Python :

```
conda activate py_prog_niv2
```

Vous êtes maintenant dans l'environnement. Il est alors possible de lancer `python`, ou d'utiliser **Jupyter Notebook**, plus intuitif pour les débutants et particulièrement intéressant pour structurer son code de façon plus compréhensible. Pour utiliser **Jupyter**, il faut d'abord lui présenter l'environnement afin qu'il l'enregistre. Pour cela, utilisez la commande suivante :

```
python -m ipykernel install --name py_prog_niv2
```

Si vous ne souhaitez pas travailler directement dans l'invite de commandes Python et préférez **Jupyter**, vous pouvez alors lancer votre session avec la commande suivante, pour ensuite créer vos *notebooks* et commencer à programmer :

```
jupyter lab
```

Afin de quitter une session Jupyter, retournez dans le terminal et utilisez la combinaison de touches `CTRL + C`. Enfin, pour sortir de votre environnement de programmation, utilisez la commande suivante :

```
conda deactivate
```

Création d'un environnement personnalisé

A l'issue du cours, vous pourrez continuer à utiliser l'environnement fourni et l'enrichir, ou en développer un répondant spécifiquement à vos besoins.

Lorsque vous téléchargez des paquets pour votre environnement, il est recommandé de les récupérer du même dépôt (ou *channel*). L'un des plus connus, **Conda Forge**, est ici préconisé. Pour en faire le dépôt standard lors du téléchargement des bibliothèques, utilisez les lignes suivantes :

```
conda update -n base conda
conda config --add channels conda-forge
conda config --set channel_priority strict
conda update --all
```

Pour créer un nouvel environnement, utilisez la commande suivante (la version de Python étant à adapter à vos besoin) :

```
conda create -n nom_environnement python=3.7
```

Vous pouvez ensuite télécharger des paquets avec la commande suivante, ici un exemple pour Pandas, JupyterLab et IPyKernel :

```
conda install -n nom_environnement pandas jupyterlab ipykernel
```

Enfin, pour lister l'ensemble des paquets installés dans votre environnement, vous pouvez utiliser la commande :

```
conda list -n nom_environnement
```

Il arrive qu'un paquet n'est disponible sur aucun dépôt de **Conda**. Vous pourrez alors utiliser la commande native *pip* pour installer les librairies inconnues :

```
pip install un_paquet
```

Ou alors :

```
python -m pip install un_paquet
```

Les tableaux en Python - Numpy

La maîtrise des tableaux en Python est une technique essentielle à acquérir pour le traitement d'images. **Numpy** (Numerical Python) est la librairie la plus utilisée pour la gestion des tableaux, notamment en *datascience* et en traitement d'image. En effet, une image peut être assimilée à un tableau de deux dimensions où chaque case correspond à un pixel et exprime l'intensité lumineuse (ou la réfléctance dans le cas des images satellites). Un tableau **Numpy** n'est rien d'autre qu'une liste en Python mais de multiples méthodes permettent de faciliter et d'accélérer les traitements.

Lancez tous les traitements Python dans un Notebook Jupyter sur votre machine pour visualiser toutes les exécutions. N'hésitez pas à regarder la documentation sur le site officiel de Numpy [https://numpy.org/].

Avant d'utiliser cette librairie, il est nécessaire de l'installer à votre environnement de programmation et de l'importer :

```
import numpy as np
```

Vous pouvez ensuite créer un tableau d'une dimension :

```
np.array([1, 4, 2, 5, 3])
```

Contrairement aux listes classiques en Python, un tableau **Numpy** ne peut posséder qu'un seul type de données. Lors de la création d'une liste, il est possible d'indiquer en paramètre le type des données souhaité :

```
np.array([1, 4, 2, 5, 3], dtype=int)
```

Il est souvent nécessaire de créer des tableaux avec des valeurs prédéfinies pour chaque cellule, comme un tableau avec uniquement des zéros :

```
np.zeros(5, dtype=int)
```

Avec uniquement des uns :

```
np.ones((4, 4), dtype='float32')
```

Ou avec une valeur particulière comme ici 2022 :

```
np.full((4, 4), 2022)
```

Il est possible d'extraire différentes informations des tableaux qui vous seront très utiles (surtout *shape* !!) :

```
np.random.seed(0)
x1 = np.random.randint(15, size=6)  # Tableau de dimension 1
print("nombre de dimensions de x1: ", x1.ndim)
print("forme de x1: ", x1.shape)
print("taille de x1: ", x1.size)
print("type de x1: ", x1.dtype)
```

Calcul matriciel

Un tableau n'est autre qu'une matrice (algèbre linéaire) contenant un certain nombre de colonnes (c) et un certain nombre de lignes (l).

Soit a et b deux matrices de taille (2, 3) (donc (l, c)) :

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad b = \begin{bmatrix} 2 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

Le produit **terme à terme** de a par b donnera :

$$a \times b = \begin{bmatrix} 1 \times 2 & 2 \times 1 & 3 \times 3 \\ 4 \times 3 & 5 \times 2 & 6 \times 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 9 \\ 12 & 10 & 6 \end{bmatrix}$$

Note : pour un produit terme à terme, les deux matrices doivent être de même taille. La division se fera également terme à terme. Il est commutatif.

et en Python :

```
a = np.array([[1, 2, 3],
              [4, 5, 6]])
b = np.array([[2, 1, 3],
              [3, 2, 1]])
a*b
a/b
a+5
```

Le produit **matriciel** est défini si le nombre de colonnes de la première matrice est égale au nombre de lignes de la secondes matrices :

$$(m \times n).(n \times k)$$

Nous différencions le produit matriciel du produit terme à terme par l'utilisation d'un point (.). Attention, le produit matriciel classique n'est pas commutatif. La taille de la matrice issue du produit matriciel sera :

$$(m \times k)$$

Soit c, une matrice de taille (2,3) et d une matrice de taille (3,1) :

$$c = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad d = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix}$$

Le produit matriciel de c par d donnera :

$$c.d = \begin{bmatrix} 1 \times 4 + 2 \times 2 + 3 \times 1 \\ 4 \times 4 + 5 \times 2 + 6 \times 1 \end{bmatrix} = \begin{bmatrix} 11 \\ 32 \end{bmatrix}$$

Et en Python :

```
c = np.array([[1, 2, 3],
                    [4, 5, 6]])
d = np.array([[4],
                   [2],
                   [1]])
np.dot(c,d)
```

La somme et la soustraction ne peuvent se faire que sur des matrices de tailles égales et se font terme à terme.

Dans vos travaux de télédétection, vous serez la plupart du temps amenés à utiliser le produit et la division terme à terme. Le produit matriciel classique ne s'utilise que dans très peu de cas dont certains très spécifiques (mathématiques, intelligence artificielle, physique ...).

Le slicing

Lorsque vous manipulez des tableaux, vous aurez souvent besoin d'extraire des sous ensembles (ou sous tableau) de ces tableaux. Pour cela, une technique de manipulation de tableaux appelé sclicing (tranchage) nous est proposé par Python ... et elle est bien pratique :-)

a est un tableau de taille 5 contenant des entiers :

```
a = np.array([10, 20, 30, 40, 50])
```

Note : un tableau en programmation commence à 0 et finit à [taille du tableau - 1].

Nous souhaitons extraire le sous tableau allant de 20 à 40 inclus :

```
a[1:4]
```

On peut aussi facilement extraire le dernier élément du tableau :

```
a[-1]
```

Et donc inverser le tableau :

```
a[::-1]
```

Le slice [n:m] extrait tous les éléments de n inclus à m exclu. Cette technique fonctionne sur les tableau 1D mais aussi sur les tableau 2D, 3D, 4D, 5D ... nD. Oui, un tableau peut avoir bien plus que 3 dimensions.

Soit b un tableau 2D de dimension (2, 3) contenant des entiers aléatoires de 0 à 10 ($[0; 10[$) :

```
b = np.random.randint(10, size =(2,3))
```

Je souhaite extraire le chiffre de la première ligne et de la deuxième colonne :

```
b[0, 1]
```

Pour rappel, cela signifie : b[l, c] où l correspond à l'indice de la ligne et c à l'indice de la colonne.

On extrait le sous tableau allant de la 1ere colonne à la 2eme colonne pour uniquement la première ligne :

```
b[0, 1:3]
```

Maintenant, je souhaite extraire le sous tableau en enlevant la première colonne :

```
b[:, 1:]
```

Pour traduire cette instruction : "Je prends toutes les lignes et je prends les colonnes allant de la deuxième (1 car on commence à 0) jusqu'à la dernière."

Changement de taille des tableaux

Il est également possible de changer facilement la taille d'un tableau avec l'attribut shape d'un tableau ou la fonction reshape de Numpy :

```
u = np.arange(1, 16)
u.shape = (3, 5)
np.reshape(u, (3,5))
```

et passer d'un tableau d'une ligne à un tableau d'une colonne :

```
v = np.arange(1, 16)
v.shape = (np.size(v), 1)
```

Exercices

En utilisant l'invite de commandes **conda**, créez un environnement de programmation appelé *mon_super_env*, activez le et installez **Numpy** et **Jupyter Notebook**. Faites différents *markdown* par question pour avoir un TP structuré. N'oubliez pas, en programmation, Google et StackOverflow sont vos amis ... toujours en anglais, la communauté prog travaille uniquement en anglais ... d'ailleurs une astuce pour faciliter vos recherches ... [https://www.codegrepper.com/].

Petit tip pour avoir la documentation d'une fonction Numpy en ligne de commande :

```
python -c "import numpy; numpy.info(numpy.add)"
```

Ou directement en Python :

```
import numpy as np
np.info(np.add)
```

- **Question 1** – Créer un vecteur de valeurs nulles de taille 10.
- **Question 2** – Créer un vecteur de valeurs nulles de taille (10, 10).
- **Question 3** – Créer un vecteur de valeurs nulles de taille 10 et remplacez la 5ème valeur par un 1.
- **Question 4** – Créer un tableau de taille 50 et l'inverser (le premier élément deviendra le dernier).

- **Question 5** – Créer une matrice 3×3 avec des valeurs allant de 0 à 8 (Indices : reshape, shape ...).
- **Question 6** – Créer une matrice identité de taille 3×3 (Indices : [https://fr.wikipedia.org/wiki/Matrice_identit%C3%A9] et documentation de *np.eye*).
- **Question 7** – Créer une matrice de dimension $3 \times 3 \times 3$ avec des valeurs aléatoires.
- **Question 8** – Créer une matrice de dimension 10×10 avec des valeurs aléatoires et extraire le min, le max, la moyenne (*mean*), l'écart-type (*std* pour standard deviation) et le shape.
- **Question 9** – Créer une matrice 2D avec des 1 sur les bords et des 0 à l'intérieur. Exemple :

$$m = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- **Question 10** – Créer une matrice aléatoire m de taille 10×10 et normalisez là avec la formule suivante :

$$n = \frac{m - \bar{m}}{\sigma(m)}$$

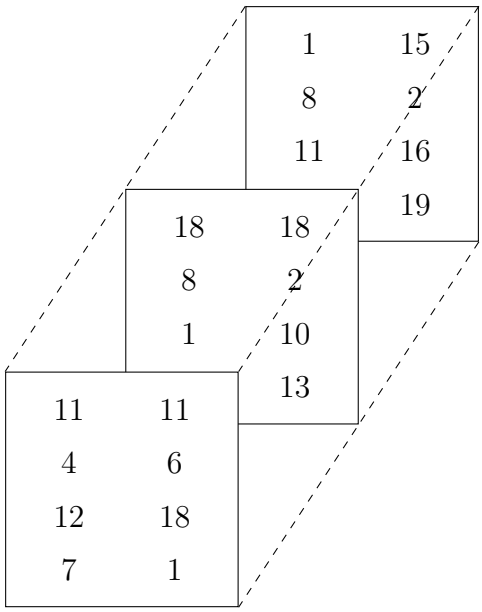
où $\bar{m}$ est la moyenne de m et $\sigma(m)$ l'écart-type de m.

- **Question 11** – Faire la multiplication d'une matrice 5×3 par une matrice 3×2.
- **Question 12** – Générer une matrice 1D avec des valeurs allant de 0 à 15 et rendre négatifs ($\times -1$) tous les nombres entre 5 et 12.
- **Question 13** – Générer une matrice 2D avec des valeurs allant de 0 à 30 de taille 5×6 et rendre négatifs ($\times -1$) tous les nombres de l'avant dernière ligne (et uniquement de cette ligne là !).
- **Question 14** – Générer deux matrices 2D de taille 1000×1000 et faire la somme en parcourant chaque élément des deux matrices avec une boucle (boucle sur les lignes et colonnes) et en utilisant Numpy. Comparer les temps d'exécution des deux méthodes.

```
import time
start_time = time.time()
...
end_time = time.time()
```

- **Question 15** – Générer un tableau d'entiers de 845×893 avec des valeurs allant de 0 à 20 et compter le nombre d'occurrence de chaque élément du tableau. (Indice : Google pour l'occurrence :-))

- **Question 16** – Générer trois tableau d'entiers de 5×5 avec des valeurs entières allant de 0 à 20. Faire un stack de ces tableaux dans un seul est unique tableau de dimension 3. Ci-dessous un exemple de tableau de dimension 3 :



- **Question 17** – Générer une matrice de 1000×1000 avec des entiers aléatoires entre 0 et 50, la normaliser (cf. Question 10) et la binariser (tous les éléments supérieurs à 0.5 prennent la valeur 1, sinon ils prennent la valeur 0). Ne pas utiliser de boucles ! (Indices pour la normalisation : np.where).