TP N°15 : Le module incontournable du calcul scientifique : Numpy
Les vecteurs

Ce TP a pour objectif de présenter l'essentiel du module Numpy de Python grâce à des exemples à saisir dans la console (sous IDLE) et quelques exercices d'applications.

Présentation du module Numpy :

Le module numpy est la boîte à outils indispensable pour faire du calcul scientifique avec Python. Pour modéliser les vecteurs, les matrices, et plus généralement les tableaux à n dimensions, numpy fournit le type ndarray (*N dimensional array*). Il y a des différences majeures avec les listes (resp. les listes de listes) qui pourraient elles aussi nous servir à représenter des vecteurs (resp. des matrices) :

- Les tableaux numpy sont homogènes, c'est-à-dire constitués d'éléments du même type. On trouvera donc des tableaux d'entiers, des tableaux de flottants, des tableaux de chaînes de caractères, etc.
- La taille des tableaux numpy est fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme le ferait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ce module n'est pas fourni avec la distribution Python de base, il doit être installé à partir du site officel :
`http://docs.scipy.org/doc/numpy-1.10.1/user/install.html`
Le module numpy propose plusieurs sous-modules intéressants :

- `numpy.linalg` : un module d'algèbre linéaire basique,
- `numpy.random` : un module pour les générateurs aléatoires,

On charge traditionnellement l'ensemble du module numpy en le renommant np : `import numpy as np`
Pour ne pas surcharger l'espace des noms et risquer la présence d'homonymes, on ne charge pas un module par l'instruction `from module import *`

Création des tableaux

Création avec np.array

On définit souvent un tableau numpy à partir d'une liste (ou lise de listes), en utilisant la fonction array du module numpy. La fonction array agit comme un mécanisme de conversion de type 'list' vers'numpy.ndarray' qui est le type commun à tous les tableaux numpy.

```
a = np.array( [1, 2, 3] ) # créer un tableau à partie d'une liste
print(type(a))  # <class 'numpy.ndarray'>
print(a.dtype) # int64
b = a.tolist() #créer un objet b de type list, copie de l'objet a de type ndarray
```

Remarques :

— L'attribut dtype (abréviation de *data type*) de a, 'int64' qui indique le type commun à tous les éléments du tableau, ici des entiers codés sur 64 bits, c'est-à-dire quatre octets ou *bytes*.

— La méthode tolist d'un tableau numpy le transforme en la liste (éventuellement une liste de listes) de ses éléments. Attention, ce c'est pas une fonction du module numpy. On n'écrira donc pas `np.tolist(a)` mais `a.tolist()`.

Création avec des fonctions spéciales

La fonction `arange(d, f, h, dtype=...)` crée des vecteurs de *valeurs régulièrement espacées* dans l'intervalle `[d, f[` avec un pas de h.

```
t0 = np.arange(0, 10, 2) # début=0, fin(exclue)=10 , pas=2 -> t0 = [0 2 4 6 8]
t1 = np.arange(3) # tableau de 3 entiers -> t1 = [0 1 2]
t2 = np.arange(3.) # tableau de 3 réels -> t2 = [ 0.  1.  2.]
t3 = np.array([2]*5); #concaténation de 5 exemplaires de la liste [2] -> t3 = [2 2 2 2 2]
```

La fonction `linspace(a, b, n)` retourne un vecteur de n valeurs régulièrement échelonnées du segment `[a, b]` (donc ici les extrémités sont incluses). Par défaut, $n = 50$. Si $b < a$, les valeurs sont obtenues dans l'ordre décroissant. Ces fonctions sont en particulier utilisées pour le tracé des fonctions. Exemple : `b = np.linspace(0, 10, 5)`

Exercice 1

Construire un objet de type numpy.ndarray

1. dont les termes sont les multiples de 3 compris entre 0 et 27 ;
2. de taille égale à 100 dont les extrémités sont 0 et 27 ;

Lecture de valeurs dans un vecteur : « slicing »

La lecture d'éléments d'un tableau `numpy` procède par coupes (« slices » en anglais). La syntaxe est la suivante :

`t[indice_début(inclu) : indice_fin(exclu) : incrément]`

```
v = np.array(range(0,160,10))
v[0] # le 1er élément : compte à partir du début
v[-v.size] # le 1er élément : on compte à partir de la fin
v[-1] # le dernier élément
v[v.size-1] # le dernier élément
```

Quelques coupes de valeurs consécutives dans l'ordre des indices croissants :

```
v[4:11] # de v[4] à v[10], donc 11-4 = 7 éléments consécutifs
v[:] # la totalité du vecteur v
v[::-1] # la totalité du vecteur v mis à l'envers (incrément = -1)
v[:6:2] # (incrément = 2)
```

Accès aux données dans un ordre quelconque :
Si *a* est un tableau, et si *I* est un tableau d'indices (éventuellement une liste ou un tuple), alors *a[I]* renvoie le tableau (de même format que le tableau *I*) formé des éléments *a[i]*, où *i* est dans *I*.

```
v = np.array(range(0,160,10))
a = v[[6,2,1,3]]
indices = np.array([[5,2,1],[3,8,7]])
b = v[indices]
np.take(v,[6, 1, 5, 0]) # renvoie le tableau des valeurs v[6], v[1], v[5], v[0]
```

Écriture de valeurs dans un vecteur

Les tableaux `numpy` sont mutables, on peut donc modifier les valeurs qu'ils contiennent sans redéfinir l'objet lui-même. Si *a* est un vecteur `numpy`, l'instruction `a[n] = x` écrit la valeur `x` en position `n`. La valeur `x` écrite en position `n` du vecteur `a` doit a priori être compatible avec le « `data type` » de `a`.

```
a = np.arange(10); a[7] = 21 # écrit la valeur 21 en position 7
```

On peut écrire plusieurs valeurs simultanément dans un vecteur, en utilisant une syntaxe du genre `a[coupe ]=...`. Il faut simplement veiller à ce que le membre de droite (la source) soit « `array-like` » (une liste, un tuple, un tableau) et que la coupe spécifiée *a* (donc la cible) soient de même taille. Ainsi, les instructions suivantes ont le même effet : `a[4:7] = [111,222,333]` et `a[4:7] = (111,222,333)` ou encore `a[4:7] = np.array([111,222,333])` .

On peut écrire plusieurs valeurs simultanément dans un vecteur dans un ordre quelconque, en utilisant une syntaxe du genre `a[I]=V`, avec `I` un tableau d'indices et `V` un tableau de valeurs à insérer dans le tableau `a`.

```
a[[7, 2, 5, 3]] = (7777, 2222, 5555, 3333) # on modifie a[7], a[2], a[5] et a[3]
a.put([6,1,5,2],[66,11,55,22]) # place 66 en position 6, 11 en position 1, etc.
```

Exercice 2

Définir les objets suivants : `L1 = [1,2,3]` et `L2 = [4,5,6]`

1. Créer les deux vecteurs `v1` et `v2` à partir de `L1` et `L2`.
2. Écrire une fonction `prod_scal(v1,v2)` prenant en entrée deux vecteurs de même taille `v1` et `v2` et renvoyant leur produit scalaire. Donner sa complexité.
3. En déduire une fonction `ortho(v1,v2)` renvoyant un booléen indiquant si les vecteurs sont orthogonaux.
4. Écrire une fonction `prod_vec(v1,v2)` prenant en entrée deux vecteurs de taille 3 et renvoyant leur produit vectoriel. On pourra utiliser la commande `np.zeros(n)` créant un vecteur de taille *n* rempli de 0.