La simulation numérique
Numpy et Matplotlib
Les modules incontournables du calcul scientifique

Anis SAIED
Institut Préparatoire aux Etudes d'Ingénieurs de Nabeul (IPEIN),
Université de Carthage,
Tunisie
anis_saied@hotmail.com

2025/26

Table des matières

Introduction

Ce document représente un rappel au module `Numpy` du Python. Il a pour but de présenter l'essentiel grâce à des exemples à saisir dans la console (sous Idle), et dont analysera les résultats.

1 Le module `Numpy` : Présentation

Le calcul scientifique nécessite la manipulation de données en quantité souvent importante.
`NumPy` (**Num**erical **Py**thon) est une bibliothèque Python pour travailler avec les grands ensembles de données de manière efficace.
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, 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` [1] propose plusieurs sous-modules intéressants :

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

Comment charger le module `numpy` ?
On charge traditionnellement l'ensemble du module numpy en le renommant `np` :

1. http ://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

```
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 *`

2 Les tableaux numpy

2.1 Création avec np.array

On définit souvent un tableau numpy à partir d'une liste (ou liste 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],
                [2, 3, 4],
                [0, 1, 0]])

#[[1 2 3]
# [2 3 4]
# [0 1 0]]
```

Mais ce ne sont pas des listes :

```
print(type(a))

# <class 'numpy.ndarray'>
```

L'attribut dtype :

```
print(a.dtype)

# int64
```

⚠ Il ne faut pas confondre le type des tableaux numpy : 'numpy.ndarray' avec le résultat renvoyé par 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*.

Remarque : 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(). Exemple :

```
a = np.array( [1, 2, 3] ) # créer un tableau à partir d'une liste
b =a.tolist() #créer un objet b de type list, copie de l'objet a de type ndarray

# [1 2 3]
```

2.2 Création avec des fonctions spéciales

2.2.1 Vecteurs spéciaux (tableaux uni-dimensionnels)

La fonction **arange** crée des vecteurs de *valeurs régulièrement espacées*.
La syntaxe est **arange(d,f,h,dtype=...)**, et génère les valeurs de l'intervalle [d, f[avec un pas de h. Par défaut, **d = 0**, **h = 1** et **dtype = float**.

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

# [0 2 4 6 8] [0 1 2] [ 0. 1. 2.] [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.

```
b = np.linspace(0, 10, 5); b # début=0, fin(inclue)=10 , n=5

#[ 0. 2.5 5. 7.5 10.]
```

2.2.2 Matrices spéciales (tableaux bi-dimensionnels)

Découvrons quelques fonctions spéciales : np.ones, np.zeros, np.eye, np.diag,...

Rappelons que chez les booléens, false = 0 = « nul » = « vide », et True = 1 = « non nul » = « non vide » :

```
a = np.ones((3, 5)); a # 3 lignes x 5 colonnes, dtype = float (par défaut)
b = np.ones((3,5),np.int); b
c = np.zeros((3, 5)); c
d = np.zeros((3, 5),dtype=np.bool); d
e = np.eye(3); e #Créer une matrice identité d'ordre n
g = np.eye(4,4); g
h = np.diag([1,2,3]); h #Créer une matrice diagonale
i = np.diag([1,2,3],1); i # essayer : -1, 2 (décalage de la diagonale)
# np.concatenate : permet de créer des matrices par blocs
A=np.ones((2,3)); A
B=np.zeros((2,3)); B
AB0 = np.concatenate((A,B),axis=0); AB0 #B sous A
AB1 = np.concatenate((A,B),axis=1); AB1 #B à droite de A
# np.column_stack : construire la matrice dont les colonnes
# sont les vecteurs passés en arguments.
v1 =np.array([11,21,31]); v1
v2 =np.array([12,22,32]); v2
v3 =np.array([13,23,33]); v3
v = np.column_stack((v1,v2,v3)); v
#Autres méthodes utiles
v.fill(1); v #remplir le tableau v par une valeur constante.
a.fill(3.14); a #remplir le tableau a par la valeur flottante 3.14

#La valeur flottante 3.14 a été convertie en valeur
# entière 3 pour le remplissage
v.fill(3.14); v

z = np.array([[1,2],[3,4]]*2); z #tableau avec des répétitions
```

2.2.3 Tableaux répondant à une formule donnée

La fonction **fromfunction** permet de construire un tableau dont le terme général obéit à une formule donnée.

```
def f(i,j): return 10*i+j

t1 = np.fromfunction(f,(4,5)); t1 # chaque élément t1[i,j] = f(i,j)

t2 = np.fromfunction(f,(4,5),dtype=int); t2 #On exige un tableau de type int

#Le même tableau peut être obtenu autrement
t3 = np.array([[f(i,j) for j in range(5)] for i in range(4)]); t3
```

Exemple :
Matrice d'Hilbert, de terme général : $H[i,j] = \frac{1}{(i+j+1)}$ avec $i, j \geq 0$

```
t4 = np.fromfunction(lambda i,j: 1/(i+j+1),(4,4)); t4
```

2.2.4 Tableaux aléatoires

Les fonctions qui forment des tableaux aléatoires sont présentes dans le sous-module **random** du module numpy.

- La fonction **random.rand** renvoie un tableau de valeurs aléatoires dans l'intervalle [0, 1]. Elle prend comme argument un entier n_1 (respectivement une séquence n_1, n_2,...) et renvoie un vecteur de longueur n_1 (respectivement un tableau de format $n_1 \times n_2 \times ...$).
- La fonction **random.randint** renvoie une valeur ou un tableau de valeurs entières aléatoires au sens de la distribution uniforme dans un intervalle semi-ouvert [a, b[. La syntaxe est :
  — **random.randint(a,b)** pour une seule valeur,
  — **random.randint(a,b,n)** pour un vecteur de longueur n,
  — **random.randint(a,b,(n,p))** pour une matrice de format $n \times p$.
- La fonction **random.choice** renvoie un échantillon aléatoire de valeurs extraites d'un tableau a.

```
# tirage de 6 entiers aléatoires dans [0, 1[
a = np.random.rand(6); a

# tirage de 10 entiers aléatoires dans [0, 10[
b = np.random.randint(0, 10, size=(2, 5)); b

# a1 = une valeur choisie aléatoirement à partir du tableau a
a1 = np.random.choice(a); a1

# a2 = une succession de 10 valeurs choisies aléatoirement à partir du tableau a
a2 = np.random.choice(a,10); a2

# a2 = un tableau 2 x 10 de valeurs choisies à partir du tableau a
a3 = np.random.choice(a,(2,10)); a3
```

2.3 Lecture des données dans un fichier

Si des données sont stockées dans un fichier texte avec délimiteur, par exemple le format ***csv***[2] (comma-separated values), on peut construire la matrice de ces données avec la fonction `np.genfromtxt`. Créer, dans le répertoire courant, le fichier *data.csv* contient les données suivantes :

```
1, 2, 3, 4, 5
6, 0, 0, 7, 8
0, 0, 9, 10 ,11
```

On peut alors définir une matrice de la façon suivante :

```
m = np.genfromtxt("./data.csv", delimiter = ',')
print (m)

# [[ 1. 2. 3. 4. 5.]
# [ 6. 0. 0. 7. 8.]
# [ 0. 0. 9. 10. 11.]]
```

2.4 Dimensions d'un tableau

2.4.1 Attributs de dimension d'un tableau

Les tableaux `numpy` sont des objets, qui possèdent certains attributs (des propriétés de ces objets). Modifier ces attributs est une action « en place », sans copie du tableau.

- **`size`** indique le nombre d'éléments du tableau.
  Variante : `np.size(a,0)` et `np.size(a,1)` donnent respectivement le nombre de lignes (ou `np.alen(a)`) et le nombre de colonnes d'une matrice.
- **`shape`** donne le tuple de son format (nombre le lignes, de colonnes...)
- **`ndim`** le nombre d'indices nécessaires au parcours du tableau.

```
a = np.array( [1, 2, 3] )
print(a.size)

# 3

print(a.shape) # ou np.shape(a)

# (3,)

print(a.ndim)

# 1

b = np.array([  [1, 2, 3],
                [4, 5, 6]])
print(b.size)

# 6

print(b.shape)
```

2. Un fichier CSV est un fichier texte, par opposition aux formats dits « binaires ». Chaque ligne du texte correspond à une ligne du tableau et les virgules correspondent aux séparations entre les colonnes. Les portions de texte séparées par une virgule correspondent ainsi aux contenus des cellules du tableau. Une ligne est une suite ordonnée de caractères terminée par un caractère de fin de ligne

```
# (2, 3)

print(b.ndim)

# 2
```

L'attribut **dtype** n'est pas modifiable, puisque le stockage d'un entier n'utilise pas le même nombre de bits que le stockage d'un flottant ou d'un complexe. Il peut cependant être imposé à la création du tableau numpy, pour éviter le casting automatique.

```
c = np.array( [[1, 0],[0, 2] ], dtype = complex)
print(c)

# [[ 1.+0.j 0.+0.j]
# [ 0.+0.j 2.+0.j]]
```

Si l'on souhaite modifier le dtype d'un tableau, il faut passer par une copie avec conversion, en utilisant la méthode astype :

```
d = c.astype(int)
print(d)

# [[1 0]
# [0 2]]
```

2.4.2 Redimensionnement par « reshape » ou « resize »

La principale méthode pour modifier la taille d'un tableau est **reshape**.
Le redimensionnement d'un tableau est, quand même, soumis à la contrainte que le nombre total d'éléments doit rester constant. On pourra ainsi passer (dans les deux sens) d'un vecteur de taille n à une matrice de taille (p, q) ou à une matrice de taille (r, s) à condition que $n = pq = rs$.

Si a est un tableau numpy, l'expression **a.reshape(n,p)** renvoie une copie redimensionnée (le contenu de la variable initiale n'est donc pas affecté, comme on le constate ci-dessous)

```
a = np.array(range(6)); a

#array([0, 1, 2, 3, 4, 5])

a.reshape(1,6)

#array([[0, 1, 2, 3, 4, 5]])

a

#array([0, 1, 2, 3, 4, 5])

a.reshape(2,3)

#array([[0, 1, 2],
# [3, 4, 5]])

a.reshape(3,2)

# array([[0, 1],
# [2, 3],
# [4, 5]])
```

```
a.reshape(6,1)

# array([[0],
# [1],
# [2],
# [3],
# [4],
# [5]])
```

Il y a une autre possibilité qui consiste à réinitialiser l'attribut **shape** d'un tableau numpy. La différence avec ce qui précède est que le redimensionnement est fait « sur place », et donc que le contenu de la variable s'en trouve affecté.

L'attribut **shape** (format) d'un tableau numpy s'écrit sous la forme

- un entier n pour un vecteur de longueur n,
- couple (n, p) pour une matrice de type n × p

```
a = np.array(range(6)); a

#array([0, 1, 2, 3, 4, 5])

a.shape = (2,3); a # devient matrice 2 × 3

#array([[0, 1, 2],
# [3, 4, 5]])

a.shape = (3,2); a # devient matrice 3 × 2

#array([[0, 1],
# [2, 3],
# [4, 5]])

a.shape = 6; a # redevient un vecteur

# array([0, 1, 2, 3, 4, 5])
```

2.4.3 Aplatissement d'un tableau

La méthode **flatten** d'un tableau numpy renvoie une copie « aplatie » de ce tableau.

C'est l'argument `order=...` dont la valeur par défaut est `order='C'` et qui peut être modifié en `order='F'`.

- `order='C'` : le parcours s'effectue d'abord de gauche à droite sur une ligne avant de passer à la ligne suivante (donc l'indice de colonne varie prioritairement).
- `order='F'` : le parcours s'effectue d'abord de haut en bas sur une colonne avant de passer à la colonne suivante (donc l'indice de ligne varie prioritairement).

```
a = np.array([[1,5,3],[2,7,4]]); a
a.flatten() # parcours à droite d'abord  : array([1, 5, 3, 2, 7, 4])
a.flatten('F') # parcours Fortran (en bas d'abord) : array([1, 2, 5, 7, 3, 4])
```

2.4.4 Suppressions/insertions de lignes, de colonnes

delete(a,k,axis=0) et **delete(a,k,axis=1)** suppriment respectivement la ligne et la colonne d'indice k du tableau a.
Le troisième argument indique donc le numéro d'indice (l'axe) selon lequel on pratique cette suppression.

```
a = np.array([[ 0, 1, 2, 3, 4],[10, 11, 12, 13, 14],[20, 21, 22, 23, 24],[30, 31, 32, 33
b = np.delete(a,2,0); b # supprimer la ligne d'indice 2 de a
c = np.delete(b,3,1); c # supprimer la colonne d'indice 3 de b
```

On peut également supprimer plusieurs colonnes (ou lignes) à la fois. Voici quelques exemples :

```
np.delete(a,np.s_[0::2],1) #supprime les colonnes d'indice pair
np.delete(a,np.s_[1::2],1) #supprime les colonnes d'indice impair
np.delete(a,[0,2,3],1) #supprime les colonnes d'indice 0, 2, 3
```

Les expressions **insert(a,k,v,axis=0)** et **insert(a,k,v,axis=1)** permettent d'insérer la valeur v respectivement avant la k-ième ligne ou avant la k-ième colonne de a. Voici deux exemples :

```
a = np.array([[ 0, 1, 2, 3],[10, 11, 12, 13],[20, 21, 22, 23]]); a #une matrice a de ty
np.insert(a,2,-1,axis=1) #insère des -1 juste avant la colonne d'indice 2
np.insert(a,1,0,axis=0) #insère des 0 juste avant la ligne d'indice 1
```

Les expressions **append(a,v,axis=0)** et **append(a,v,axis=1)** permettent d'ajouter une matrice v après la dernière ligne (axis=0), respectivement après la dernière colonne(axis=1) de a. Il faut veiller à ce que la matrice ajoutée ait un format compatible avec celui de a. Voici deux exemples :

```
a = np.array([[ 0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]]); a #une matrice a de
np.append(a,[[7],[8],[9]],1) # ajoute une colonne [[7,8,9]] après la dernière colonne
np.append(a,[[3,5,7,9]],0) # ajoute une ligne [3,5,7,9] après la dernière ligne
np.append(a,np.identity(3),1) #ajouter la matrice identité à droite de la matrice a
np.append(a,np.identity(3,int),1) #l'argument int permet d'avoir un résultat à coeffic
```

2.4.5 Permutations/rotations de lignes, de colonnes

fliplr(m) inverse l'ordre des colonnes de m : ici « lr » est mis pour « left right ».
flipud(m) inverse l'ordre des lignes de m : ici « ud » est mis pour « up down ».
Remarque : ça ne marche pas pour les vecteurs, car il faut qu'il y ait au moins deux dimensions.

```
a = np.array([[ 0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]]); a #une matrice a de
np.fliplr(a) #inverse l'ordre des colonnes de a
a # le contenu initial de a est inchangé
np.flipud(a) #inverse l'ordre des lignes de a
a # le contenu initial de a est inchangé
```

2.5 Lecture dans un tableau

2.5.1 Lecture de valeurs dans un vecteur, « slicing »

La lecture d'éléments d'un tableau numpy procède par coupes (« slices » en anglais).

```
M[indice_début(inclu) : indice_fin(exclu):incrément].
```

Exemples :

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

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

Remarque : les tableaux a, b et c formés ci-dessus (et qui ne sont pas obtenus pas « slicing ») ne sont pas de simples « vues » sur une partie de v. Ils sont donc indépendants de v (modifier a ou b ou c n'affecte pas v et réciproquement).

2.5.2 Lecture de valeurs dans une matrice

Il est possible de sélectionner une sous matrice, en ne gardant que quelques lignes consécutives et/ou certaines colonnes consécutives.
La spécification la plus générale d'une coupe d'un tableau M est :

```
M[numero_ligne : numero_colonne].
M[ligne_début(inclue) : ligne_fin(exclue),colonne_début(inclue) :
colonne_fin(exclue)].
```

Si M est une matrice d'ordre $a * b$:

- `M[a,b]` : élément en position (a,b)
- `M[a] ou M[a,:]` : Vecteur ligne en pistion a
- `M[:,b]` : Vecteur colonne en position b
- `M[a:b,c:d]` : lignes de a à b-1, colonnes de c à d-1
- `M[:,:]` : une copie intégrale de M avec nouvelle référence
- `M[:a,:b]` : sous matrice les lignes du début jusqu'à a exclu ; les colonnes de début jusqu'à b exclu (Les a preimères lignes, les b premières colonnes).
- `M[::-1]` : On inverse l'ordre des lignes
- `M[:,::-1]` : On inverse l'ordre des colonnes
- `M[::2,::2]` : lignes et colonnes paires
- `M[1::2,1::2]` : lignes et colonnes impaires

```
c = np.array( range(24) )
c.shape = (3, 4, 2)
print (c)
print(c[1,:,:])
#Accès au troisième élément de la première colonne de la deuxième ligne du tableau c
print(c[1][0][2])
```

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

```
m = np.arange(24).reshape(4,6); m
a = m[[3,1,0,1]]; a #on demande le tableau des lignes m[3] , m[1] , m[0] et m[1]
i = np.array([3,0,2]) # trois indices
m.take(i,axis=0) # lectures de lignes
m.take(i,axis=1) # lectures de colonnes
m.take(i) # lecture de m "aplati"
```

2.6 Écriture dans un tableau

Les tableaux numpy sont mutables, on peut donc modifier les valeurs qu'ils contiennent sans redéfinir l'objet lui-même.

2.6.1 Écriture de valeurs dans un vecteur

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 # le vecteur des entiers de 0 à 9
a[7] = 21 # écrit la valeur 21 en position 7
a # affiche le nouveau contenu de a
#array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
#array([ 0, 1, 2, 3, 4, 5, 6, 21, 8, 9])
```

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 :
`a[4:7] = [111,222,333]`,
`a[4:7] = (111,222,333)` ou encore
`a[4:7] = np.array([111,222,333])` ont le même effet.

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

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

2.6.2 Écriture de valeurs dans une matrice

```
a = np.array(range(10))
a.shape = (2, 5)
print(a)

#[[0 1 2 3 4]
#[5 6 7 8 9]]

a[0,0] = 7
print(a)

#[[7 1 2 3 4]
#[5 6 7 8 9]]
```

```
a[0,:] = [8, 7, 6, 5, 4]

#[[8 7 6 5 4]
#[5 6 7 8 9]]

#les éléments pairs de a sont divisés par 2
a[ a % 2 == 0 ] /= 2
print(a)

#[[4 7 3 5 2]
#[5 3 7 4 9]]
```

Autres méthodes utiles :

```
np.place(a, np.mod(a,3) == 0, 0); a # remplace toutes les valeurs multiples de 3 par de
np.copyto(a,-1, where=np.mod(a,3) == 0); a # remplace les valeurs multiples de 3 par de
```

2.7 Copies de tableaux, « vues » sur des tableaux

Le module numpy permet de travailler efficacement sur des tableaux possédant un nombre fixé d'éléments, ceux-ci ayant un type donné. Tout cela se passe à l'initialisation du tableau.
Avec ces tableaux, on applique donc un principe d'économie qui consiste à effectuer aussi peu que possible de recopies de tableaux en mémoire, sauf demande explicite. Tout dépend de la fonction utilisée sur le tableau.
L'exemple suivant est très important (parce que la situation évoquée ici revient souvent).
Le vecteur a contenant un tableau numpy, la variable *a* pointe en fait sur la position de ce tableau en mémoire. L'instruction `b = a` ne crée pas une nouvelle copie du tableau *a*, mais elle demande à la variable *b* de pointer sur le même tableau physique que *a*.

```
a = np.array([[ 0, 1, 2, 3],[10,11,12,13],[20,21,22,23]])
b = a
```

On exprime cette situation en disant que *a* et *b* constituent une même ***vue*** (« view » in english) d'un tableau *numpy*.
La conséquence immédiate est que toute modification apportée à cette vue (que ce soit par l'intermédiaire de la variable *a* ou de la variable *b*) affecte ce que « voient » et *a* et *b*.

```
b[2,1] = 9 #on modifie b[2,1] dans le tableau b
b

#array([[ 0, 1, 2, 3],
#       [10,11,12,13],
#       [20, 9,22,23]])
```

Ici, par exemple, après avoir posé `b = a`, l'instruction `b[2,1]=9` modifie le tableau vu par *b*, c'est-à-dire le tableau vu par *a*. C'est confirmé quand on évalue les expressions `id(a)` et `id(b)` qui donnent un résultat identique (l'adresse de ce que voient *a* et *b*).

```
a # la modification s'est répercutée sur a
id(a)
id(b)
```

```
#array([[ 0, 1, 2, 3],
#       [10,11,12,13],
#       [20, 9,22,23]])
#19127280
#19127280
```

c'est normal car a et b partagent une même vue.
Il est bien sûr possible de « déconnecter » complètement les tableaux vus respectivement par deux identificateurs.
L'expression `a.copy()` renvoie une copie « neuve » du tableau a, indépendante de l'original.
Application : Reécrire le même code en changeant `b = a` par `b = a.copy()`

3 Les fonctions universelles

Une *fonction universelle* (le terme exact est « ufunc », abréviation de *universal function*) est une fonction qui peut s'appliquer terme à terme aux éléments d'un tableau.
Si f est une *ufunc* et si $a = [a_0, a_1, ..., a_{n-1}]$ est un tableau, alors *f(a)* renvoie le tableau $[f(a_0), f(a_1), ..., f(a_{n-1})]$.
Les a_k peuvent être des tableaux, par exemple les lignes d'une matrice m. La fonction f s'applique alors récursivement aux éléments de m.
Un grand nombre de fonctions usuelles sont directement « universalisées » dans `numpy`.
Les fonctions mathématiques gardent le même nom (préfixé par np si on a importé `numpy` par `import numpy as np`).
Si on effectue une opération arithmétique entre un tableau a et un scalaire x, tout se passe comme si x était élevé au rang de tableau constant de même format que a.
Pour la documentation officielle en anglais, voir : http ://docs.scipy.org/doc/numpy/reference/ufuncs.html
Le mieux est de commencer par quelques opérations arithmétiques simples sur un vecteur ou une matrice.

3.1 Les opérations arithmétiques

Les opérations +, *, /, ==, etc s'appliquent aux tableaux `numpy`, mais TERME à TERME.
C'est cohérent avec la définition mathématique pour l'addition, mais ça ne l'est plus du tout pour la multiplication, les puissances etc.
L'expression `a*b` renvoie le tableau des produits terme à terme des éléments de a et b : elle ne désigne donc pas un produit matriciel au sens où on l'entend habituellement (à suivre).

```
a = np.random.randint(0, 10, (2, 5)); a

# [[0 9 0 6 6]
# [8 9 2 0 0]]

negative(a) #tableaux des opposés : autre syntaxe possible -a

#[[ 0, -9, 0, -6, -6],
# [-8, -9, -2, 0, 0]]

b = np.random.randint(1, 10, (2, 5)); print(b)

# [[3 1 3 5 2]
# [8 7 7 1 6]]

print(a + b) # add(a,b) additionne terme à terme les éléments de a et b
print(a - b) # subtract(a,b) soustrait terme à terme les éléments de b à ceux de a
```

```
# [[ 3 10 3 11 8]
# [16 16 9 1 6]]

print(a * b) # multiply(a,b) multiplie terme à terme les éléments de a et b

# [[ 0 9 0 30 12]
# [64 63 14 0 0]]

print(a // b) # floor_divide(a,b) quotients (entiers) des divisions des éléments de a
mod(a,b) # donne les restes dans les divisions des éléments de a par ceux de b

# [[0 9 0 1 3]
# [1 1 0 0 0]]

print(a / b) # divide(a,b) quotients (flottants) terme à terme des éléments de a par c

# [[ 0. 9. 0. 1.2 3.]
# [ 1. 1.28571429 0.28571429 0. 0.]

print(a ** b) # power(a,b) élève les éléments de a à la puissance les éléments de b

# [[ 0 9 0 7776 36]
# [16777216 4782969 128 0 0]]

print(a == b)

# [[False False False False False]
# [True False False False False]]
```

3.2 Fonctions mathématiques usuelles

Toutes les fonctions mathématiques usuelles sont présentes sous forme universelle dans le module `numpy` (il faut bien penser à les préfixer par `np`).

- `log log10 log2` : logarithme népérien (resp. de base 10, de base 2)
- `degrees(a) (ou encore rad2deg)` : convertit les angles x de radians en degrés
- `radians(a) (ou encore deg2rad)` : convertit les angles x de degrés en radians
- fonctions trigonométriques : `sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, arctanh`

Exemple :

```
rad = np.array([pi/6, pi/4, pi/3, pi/2, pi]); rad
deg = np.degrees(rad); deg

# [ 30., 45., 60., 90., 180.]
```

3.3 Vectorisation d'une fonction

Pour qu'une fonction f puisse s'appliquer, terme à terme, à tous les éléments d'un ou de plusieurs tableaux (selon que f accepte un ou plusieurs arguments), il faut la *« vectoriser »*.

Considérons par exemple la fonction $f : x \longmapsto \dfrac{x + sin(x)}{x + cos(x)}$

```
def f(x): return (x+sin(x))/(x+cos(x))
a = np.arange(1,7); a
f(a) # Tenter d'appliquer la fonction f à un tableau numpy conduit à une erreur
```

Il faut donc utiliser la fonction `vectorize` pour rendre f universelle.

```
vf = np.vectorize(f) # la version vectorisée de f
vf(a) # on peut applique f terme à terme aux éléments de a
(a+np.sin(a))/(a+np.cos(a))#même résultat avec des opérations usuelles (toutes déjà ve
```

Exemple 2 :

```
def f(x,y):
        return 0 if x < y else y
vf = np.vectorize(f)
a = np.arange(1,7); a # array([1, 2, 3, 4, 5, 6])
b = np.array([4,1,8,7,2,9]); b # array([4, 1, 8, 7, 2, 9])
vf(a,b) # array([0, 1, 0, 0, 2, 0])
vf(b,a) # array([1, 0, 3, 4, 0, 6])
```

En continuant sur cet exemple, on voit bien que la vectorisée de f agit comme une « *ufunc* » (fonction universelle).

```
vf(a,3) #array([0, 0, 3, 3, 3, 3])
vf(3,b) # array([0, 1, 0, 0, 2, 0])
vf(5,[[1,6,2],[3,4,8],[6,3,2]]) # array([[1, 0, 2],[3, 4, 0],[0, 3, 2]])
```

3.4 Quelques méthodes sur les tableaux

On peut utiliser les méthodes des objets (ou les fonctions associées du module `numpy`) : `a.max()`, `a.min()`, `a.sum()`, `a.prod()`, `a.mean()` (moyenne arithmétique), `np.apply_along_axis`.

```
a = np.random.randint(0, 10, (2,5))
print(a)

# [[8 5 0 1 3]
# [2 2 1 5 3]]

print(a.sum()) # ou encore np.sum(a)

# 30

a.sum(axis=0) # somme des lignes

# [10 7 1 6 6]

a.sum(axis=1) # somme des colonnes

# [17 13]

print(a.mean()) #moyenne arithmétique

# 3.0

print(a.mean(axis=0))

# [ 5. 3.5 0.5 3. 3. ]

print(a.mean(axis=1))

# [ 3.4 2.6]

m = np.arange(15).reshape(3,5); m
np.apply_along_axis(np.mean,0,m) # moyenne ligne
np.apply_along_axis(np.mean,1,m) # moyenne colo

print(a.max())

# 8

print(a.max(axis=0))
np.amax(a) # le minimum de a aplati, de même pour np.amin(a)
np.amax(a,0) # suivant le premier indice (la moyenne de lignes pour chaque colonne)
np.amax(a,1) # suivant le deuxième indice (la moyenne de colonnes pour chaque ligne)

# [8 5 1 5 3]

print(a.max(axis=1))
```

```
# [8 5]
```

Les fonctions **argmax(a[,axis])** et **argmin(a[,axis])** renvoie les indices où se trouvent le ou les éléments maximum(s) (respectivement minimum(s)) dans un tableau).

```
np.argmin(a), np.argmax(a) # la position du min (resp du max) dans le tableau à plat
np.argmin(a,0) # n° de ligne des éléments minimuns, colonne par colonne
np.argmax(a,0) # n° de ligne des éléments maximums, colonne par colonne
np.argmin(a,1) # n° de colonne des éléments minimuns, ligne par ligne
```

3.5 Opérations logiques sur tableaux booléens

Si a est un tableau de booléens, l'expression **logical_not(a)** renvoie le tableau des valeurs booléennes contraires.

```
a = np.array([False, True, False,True, False, True, True]);
print(np.logical_not(a))
```

Les fonctions **logical_and**, **logical_or**, **logical_xor** prennent en argument deux tableaux a et b de booléens.
Elles appliquent une certaine fonction logique aux éléments de a et b, terme à terme, et renvoient le tableau des résultats.

```
a = np.array([False, True, False,True, False, True, True])
b = np.array([True, True, False,True, False, False, True])
np.logical_and(a,b)
np.logical_or(a,b)
```

4 Tests et comparaisons sur des tableaux

4.1 Comparaisons entre tableaux

La fonction **array_equal(a,b)** répond True si les tableaux a et b ont même format et même contenu, et False sinon.

```
a = np.arange(0,12).reshape(3,4); a
b = np.copy(a); b
np.array_equal(a,b) # mêmes tableaux : True
b = b.reshape((4,3)); b # mêmes éléments, mais forme différente, donc tableaux différe
np.array_equal(a,b) #  False
```

Autres fonctions :

- **greater(a,b)** : test et renvoie le tableau des booléens $a_k > b_k$
- **greater_equal(a,b)** : test si $a_k \geq b_k$
- **less(a,b)** : test si $a_k < b_k$
- **less_equal(a,b)** : test si $a_k \leq b_k$
- **equal(a,b)** : test si $a_k = b_k$
- **not_equal(a,b)** : test si $a_k \neq b_k$

4.2 Tris de tableau

np.sort(a) renvoie une copie triée du tableau a (l'original n'est donc pas modifié).
Mais attention : l'expression **a.sort()** trie le tableau a sur place. Il y a donc une différence importante entre la fonction `sort` du module `numpy` et la méthode `sort` d'un objet `numpy` particulier.

```
a = np.random.randint(100,1000,size=13); a
np.sort(a) # on utilise la fonction sort du module numpy
a # le tableau a original n'est donc pas été modifié
a.sort() # on utilise la méthode sort de l'objet
a # cette fois-ci le tri a été effecuté sur place
```

4.3 Recherches dans un tableau

any(a) teste si le tableau a contient au moins un élément « vrai »

```
a = np.random.random_integers(10,99,size=20); a
np.any(a>90)
a = np.random.random_integers(10,99,size=(5,6)); a
np.any(a>90,axis=0) # colonne par colonne, teste s'il y a au moins un élément > 90
np.any(a>90,axis=1) # ligne par ligne, teste s'il y a au moins un élément > 90
```

all(a) teste si tous les éléments du tableau a sont « vrais »

```
a = np.random.random_integers(10,99,size=20); a
np.all(a>10) # est-ce que tous les éléments de a sont > 10 ?
a = np.random.random_integers(10,99,size=(3,10)); a
np.all(a \% 2 == 0, axis=0) # colonne par colonne, est-ce que tous les éléments sont p
np.all(a \% 17, axis=1) # ligne par ligne, tous les éléments sont-ils non divisibles p
```

nonzero(a) renvoie le tableau des positions des éléments non nuls de a.

```
a = np.random.random_integers(-1,1,15); a
i = np.nonzero(a); i # tableaux des positions des éléments non nuls de a
a[i] # récupère le tableau des éléments non nuls
np.count_nonzero(a) # compte le nombre d'éléments non nuls du tableau a
```

where(condition) renvoie le tableau des positions des éléments d'un vecteur qui possèdent une propriété particulière.

```
a = np.random.random_integers(100,1000,15); a
i = np.where(a \% 2); i # tableau des positions des éléments impairs
a[i] # récupère le tableau de ces éléments impairs
```

La fonction `where` admet deux arguments facultatifs sous la forme d'un tableau a et d'un tableau b. Achaque fois que la condition qui est le premier argument de where est vraie, c'est l'élément correspondant de a qui est renvoyé, sinon c'est l'élément de b.

```
rdi = np.random.random_integers # un raccourci commode
a = rdi(1,9, size=(5, 6)); a
b = rdi(1,9, size=(5, 6)); b
np.where(a<5,a,b)
np.where(a<5,0,b)
np.where(a<5,0,1)
```

4.4 Tableaux d'un point de vue ensembliste

- **unique(a)** : renvoie les éléments de a triés, avec suppression des doublons.
- **in1d(a,b)** : renvoie le vecteur des tests de l'appartenance des éléments du vecteur a dans le vecteur b.
- **intersect1d(a,b)** : renvoie l'intersection des deux vecteurs a et b.
- **union1d(a,b)** : renvoie l'union ensembliste des deux vecteurs a et b.
- **setdiff1d(a,b)** : renvoie les éléments de a qui ne sont pas dans b.
- **setxor1d(a,b)** : renvoie la différence symétrique des vecteurs a et b.

```
a = np.array([5,8,3,2,4,2,3,5,7])
np.unique(a)
a = np.arange(0,25,3); a
b = np.arange(0,25,6); b
np.in1d(a,b) # les éléments de a sont-ils dans b ?
np.in1d(b,a) # les éléments de b sont-ils dans a ?
a = np.arange(0,25,2); a # multiples de 2 dans l'intervalle [0,25[
b = np.arange(24,-1,-3); b # multiples de 3 dans l'intervalle [0,25[
np.intersect1d(a,b) # multiples à la fois de 2 et de 3
np.union1d(a,b) # multiples de 2 ou de 3
np.setdiff1d(a,b) # multiples de 2, mais pas de 3
np.setxor1d(a,b) # multiples de 2 ou de 3, mais pas de 6
```

5 Traçage des courbes : Module Matplotlib

Le module `Matplotlib` est chargé de tracer les courbes :

```
>>> import matplotlib.pyplot as plt
```

Dans cette section on donne un bref aperçu des possibilités graphiques de `Matplotlib`. Il est possible de générer des graphiques à deux et à trois dimensions, et d'agir facilement sur les attributs du graphe (couleurs, type de ligne, axes, annotations...).
Les commandes présentées ci-dessous sont utiles notamment pour être introduites dans des scripts et automatiser la production de figure.

la fonction `plot()` :
D'une manière générale les fonctions **`plt.plot`** attendent des tableaux de points du plan. Selon les options, ces points du plan sont reliés entre eux de façon ordonnée par des segments : le résultat est une courbe.
La syntaxe de base de la fonction **`plot()`** est la suivante :
`plot(x,y)` permet de tracer une courbe reliant les points dont les **abscisses** sont données dans le vecteur **x** et les **ordonnées** dans le vecteur **y**.
Commençons par la fonction sinus :

```
import matplotlib.pyplot as plt
import numpy as np
from math import pi
x= np.arange(0,2*pi,pi/16) # ou x= np.linspace(0,2*pi,100) :  échantillonner la variab
plt.plot(x,np.sin(x),label="sin") # on utilise la fonction sinus de Numpy
plt.axis("equal") # Imposer une échelle identique sur les deux axes de coordonnées.
plt.xlim(-2,10) # Fixer l'intervalle de visualisation sur l'axe des abscisses.
```

```
plt.ylabel('fonction sinus')
plt.xlabel("l'axe des abcisses")
plt.title('Fonction sinus')
plt.savefig('ma_figure.png') # sauvegarder la figure (en .png ou .pdf), pour la voir o
plt.show() #Afficher l'image
plt.clf()
```

Si tout se passe bien, une fenêtre doit s'ouvrir avec la figure ci-dessus.

Exercice :
Tracer la courbe représentative de $x \longmapsto cos(x)$ sur l'intervalle [-5, 5] à l'aide de la fonction `plot()`.
Pour utiliser la fonction `plot()` de `matplotlib.pyplot`, on pourra exécuter les instructions :

```
{ X = arange(-5,5,0.1)
{ Y = cos(X)
```

5.1 Manipuler plusieurs graphiques

On peut utiliser plusieurs fenêtres graphiques en même temps, à condition de stoker les figures dans des variables :

```
t = np.linspace(0,2*pi,50)
fig1 = plt.figure()
plt.plot(t,np.cos(t),label="cos")
fig2 = plt.figure()
plt.plot(t,np.sin(t),label="sin")
plt.show() #Afficher toutes les figures
```

Une même fenêtre peut intégrer plusieurs graphes non superposés grâce à la commande `subplot` :
La commande `subplot(n,m,k)` permet de subdiviser la fenêtre en $n * m$ cases, n lignes et m colonnes (comme pour les matrices).
Ces cases sont numérotées de gauche à droite et de haut en bas. La valeur de k permet de spécifier dans quelle case on désire faire un graphique.

```
plt.subplot(2,1,1)
plt.plot(t,np.cos(t))
plt.subplot(2,1,2)
plt.plot(t,np.sin(t))
plt.show() #Afficher toutes les figures
```

Il est possible de superposer deux courbes sur le même graphique :

```
plt.plot(t,np.cos(t),t,np.sin(t))
plt.show() #Afficher toutes les figures
```

5.2 Graphiques à 3 dimensions : Les courbes

Il est possible de tracer des courbes dans l'espace avec la librairie Axes3D. Par exemple une hélice :

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.plot(np.cos(t),np.sin(t),t,label="helice")
plt.show()
```

Exemple 1 :

```
import matplotlib.pyplot as plt
import numpy as np
x=np.linspace(-5,5,100)
p1=plt.plot(x,np.sin(x),marker='o')
p2=plt.plot(x,np.cos(x),marker='v')
plt.title("Fonctions trigonometriques") # Problemes avec accents (plot_directive) !
plt.legend([p1, p2], ["Sinus", "Cosinus"])
plt.show()
```

Exemple 2 :

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```