- Méthodes pratiques sur NUMPY -

Méthodes pratiques sur Numpy

1. Quelques notions intéressantes sur NUMPY

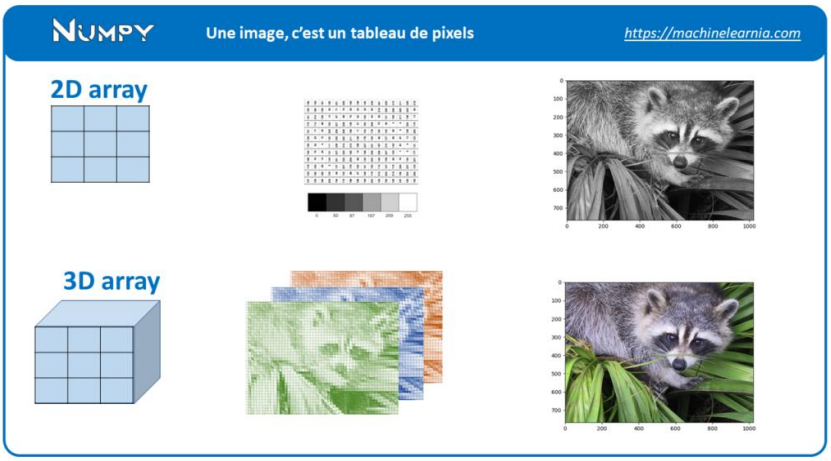
Si vous utilisez Python pour faire de l'ingénierie ou tout autre chose qui touche aux **mathématiques**, alors vous devez avoir des notions sur Numpy.

Pourquoi ? Parce que **Numpy** est **LE package** qui permet d'utiliser et d'appeler de multiples fonctions mathématiques prédéfinies, de créer des **matrices** et de faire des mathématiques de manière **ultra-performante** (Numpy étant développé en C) ! En tant que futur ingénieur, scientifique ou mathématicien, vous le savez sans doute: les matrices sont la base de tout.

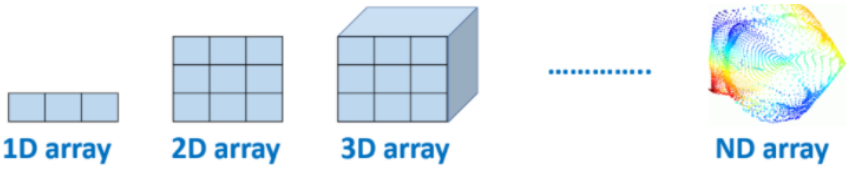
Mais Numpy permet aussi de faire de l'algèbre, du traitement d'images….etc…

1.1. Le tableau N dimensions

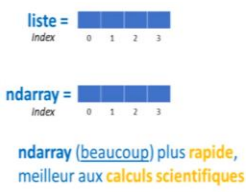
Au cœur de Numpy se trouve un objet très puissant: le tableau à N-Dimensions (**ndarray**).
Il permet d'effectuer beaucoup d'actions mathématiques **avancées**, il permet de contenir une **infinité de données**, et est **très rapide d'exécution**.



En ingénierie, on travaille le plus souvent avec des tableaux à 2 dimensions (dataset, image, matrice). Parfois à 3 dimensions (pour une image en couleur, qui contient les couches Rouge, Vert, Bleu)



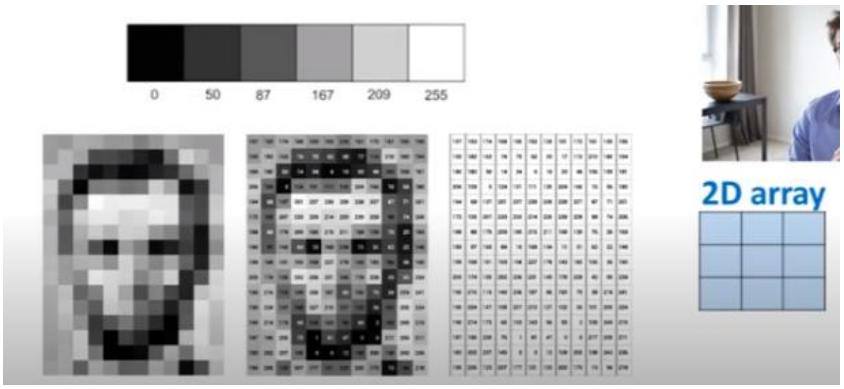
Les tableaux Numpy ressemblent aux listes (si 1 dimensions) ou aux tableaux de listes de listes, mais sont beaucoup plus puissants que les listes de listes et permet d'accéder à beaucoup de méthodes de calculs scientifiques.



Un tableau **ndarray à 2 dimensions** ressemble à un tableau excel avec des lignes et colonnes.

Méthodes pratiques sur Numpy

Voici un exemple de tableau Numpy permettant de stocker les pixels d'une image en niveau de gris :



ndarray 800lignes, 1000 colonnes

Un tableau **ndarray à 3 dimensions** pour la même image mais en couleurs (3 niveaux : vert, bleu, orange) donc 3 dimensions.

Pour info, l'assemblage des 3 couches donnera la couleur de chaque pixel par composition.

1.2. Les fonctions usuelles dans Numpy

Une fois importée la bibliothèque Numpy, vous pouvez appeler n'importe quelles fonctions usuelles.
Pour exécuter un sinus, exp, log…il faut **charger à chaque fois l'outil voulu** avec **np.*outil*()** comme dans l'exemple :

```
import numpy as np
np.cos(np.pi / 3)
```

```
0.5000000000000001
```

Autre exemple :

```
[[2 3 7]
 [8 9 1]]
```

Notre tableau A sur lequel on veut calculer l'exp de chaque élément

```
print(np.exp(A))
```

```
[[2.71828183e+00 7.38905610e+00 2.00855369e+01]
 [1.09663316e+03 2.98095799e+03 8.10308393e+03]]
```

Méthodes pratiques sur Numpy

1.3. Générateurs de tableaux Numpy (exemples)

Tester dans Jupiter ou votre idle ces exemples de tableaux Numpy :

```
import numpy as np

A=np.array([1,5,8,7.3,5,9])          # tableau avec 6 valeurs choisies
B = np.zeros((2, 3))                 # tableau de 0 aux dimensions 2x3
C = np.ones((2, 3))                  # tableau de 1 aux dimensions 2x3
D = np.full((2, 3),9)                # tableau de 9 aux dimensions 2x3
E = np.random.randn(2, 3)            # tableau aléatoire (distribution normale) aux dimensions 2x3

F = np.random.rand(2, 3)             # tableau aléatoire (distribution normale)
G = np.random.randint(0, 10, [2, 3]) # tableau d'entiers aléatoires de 0 à 10 et de dimension 2x3
```

Voici quelques exemples autour des matrices à 2 dimensions de n*m éléments:

```
M = np.array([[1, 2, 3],[4, 5, 6]]) # 2 lignes, 3 colonnes
print(M)
print(type(M)) # class nadarray
```

```
[[1 2 3]
 [4 5 6]]
<class 'numpy.ndarray'>
```

```
M.shape # longueur des dimensions
```

```
(2, 3)
```

```
#accès aux éléments
L = [] # liste auxiliaire pour stocker les éléments linéaires
for i in range(2):
    for j in range(3):
        L.append((M[i, j])) #i: indice de ligne, j : indice de colonne
print(L)
```

```
[1, 2, 3, 4, 5, 6]
```

Création d'une matrice $M \in \mathcal{M}_{n,p}(\mathbb{R})$, dont le terme général $M_{i,j}$ donné.

```
M =  np.array([[j**2 + i  for j in range(4) ] for i in range(3) ])
M
```

```
array([[ 0,  1,  4,  9],
       [ 1,  2,  5, 10],
       [ 2,  3,  6, 11]])
```

Attention : + à l'ordre des boucles `for` dans les listes en compréhension, + aux décalages d'indices l'indice de ligne commence à $i = 1$ pour la matrice, mais à `i=0` pour Python.

Exemple : stocker les éléments d'un matrice `M[i,j]` dans une liste de listes `L[i][j]`

```
n,m=M.shape # détermination de la taille de matrice
L=[[0 for j in range(m) ] for i in range(n)] # attention à l'ordre des indices!
for i in range(n):
    for j in range(m):
        L[i][j]=M[i,j] # noter la différence de syntaxe
print(L)
```

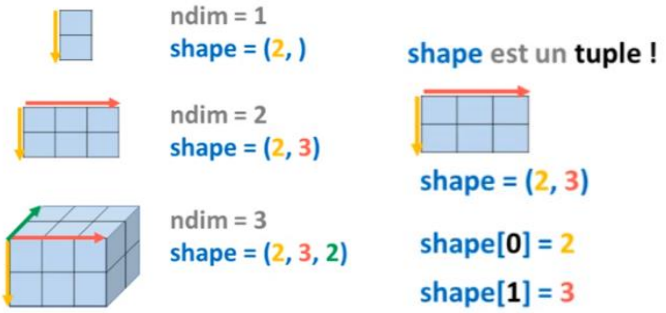
```
[[0.40446488212588894, 0.011148282221702832, 0.9149642842562511],
[0.29639530974231065, 0.3183608974537544, 0.40483570584575557]]
```

Remarque : L est une liste de listes.

1.4. Attributs et méthodes Numpy

La classe du tableau à n-dimensions (ndarray) propose plusieurs **attributs** et **méthodes**. Voici les plus utiles, qu'il faut absolument connaitre !

L'attribut ***shape*** : permet de voir la forme du tableau (nb lignes, colonnes, profondeur). On peut accéder au nb de lignes par shape[0] et nb colonnes par shape[1].



Tester sur l'exemple suivant :

```
C = np.ones((3, 2))  # tableau de 1 aux dimensions 3x2
print(C)
print(C.shape)       # tableau de taille (3,2)
print(C.shape[1])    # tableau avec 2 colonnes
```

L'attribut ***size*** : permet de voir le nombre d'élément dans le tableau

Tester sur l'exemple suivant :

```
C = np.ones((3, 4)) # tableau de taille (3,4) remplit de 1
print(C)
print(C.size)       # tableau de taille 12(éléments)
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Méthodes pratiques sur Numpy

1.5. Les constructeurs dans Numpy

Tout d'abord ***np.linspace()*** :

Permet de créer une liste de n valeurs comprises entre 2 bornes équi-réparties (très pratique pour discrétiser en méthode numérique l'axe des abscisses).

```
np.linspace(0, 10, 20)
```

```
array([ 0.        ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
        2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
        5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
        7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])
```

Voici aussi ***np.arange()*** :

Outil très pratique pour un axe d'abscisses à valeurs discrètes (histogrammes …)

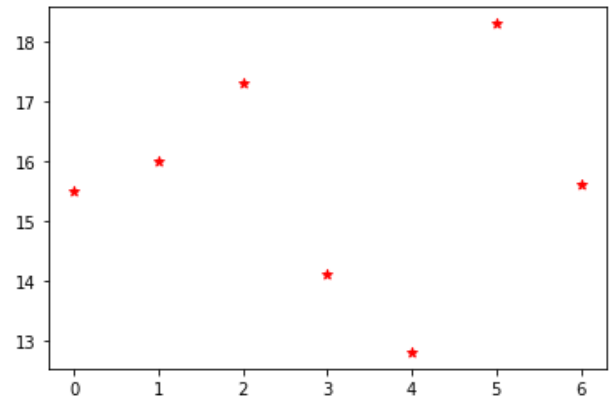

```
np.arange(0, 10, 1)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Tester vous en tracant un graphique avec « scatter » dans Matplotlib *(on détaillera ce module de tracé plus en détauil lors des prochaines séances)* qui affiche ceci :

```
import numpy as np
from matplotlib import pyplot
X=np.arange(0,7,1)
Y=np.array([15.5,16,17.3,14.1,12.8,18.3,15.6])
pyplot.scatter(X,Y,c='red',marker='*')
```

```
<matplotlib.collections.PathCollection at 0xc79d34400>
```



Méthodes pratiques sur Numpy

1.6. Manipulation de tableau dans Numpy

Voici quelques méthodes intéressantes pour **manipuler des tableaux (redimensionner, assembler, diviser…)**.
On va partir de 2 tableaux créés, un remplit de 0 et l'autre de 1 :

```
A=np.zeros((3,2))
B=np.ones((3,2))
print(A)
```

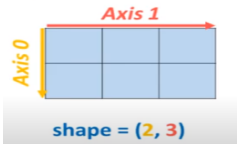
```
[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

On va regarder comment les assembler de façon horizontale ou verticale…

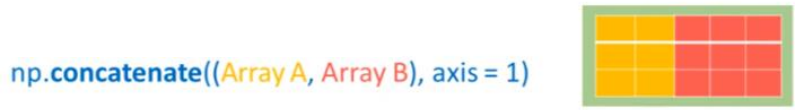
```
print(B)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

La méthode appelée **concatenate()** permet de faire des assemblages en désignant l'axe d'assemblage.



Attention : axis=0 est pour un assemblage vertical, axis=1 pour horizontal.



Tester vous avec nos 2 tableaux :

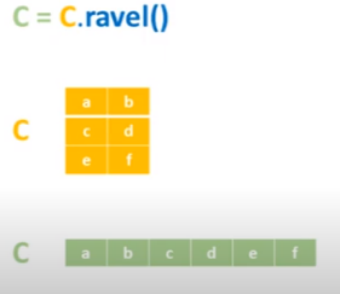
```
C=np.concatenate((A,B),axis=1)
print(C)
```

```
[[0. 0. 1. 1.]
 [0. 0. 1. 1.]
 [0. 0. 1. 1.]]
```

```
C=np.concatenate((A,B),axis=0)
print(C)
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
```

Enfin, la méthode appelée ***ravel()*** permet d'applatir un tableau sur une ligne (liste) .



Tester vous toujours sur votre dernier tableau C :

```
[[0. 0. 0. 0.]
 [0. 0. 1. 1.]
 [1. 1. 1. 1.]]
```

```
C=C.ravel()
print(C)
```

```
[0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
```

Méthodes pratiques sur Numpy

2. Navigation dans les tableaux Numpy…

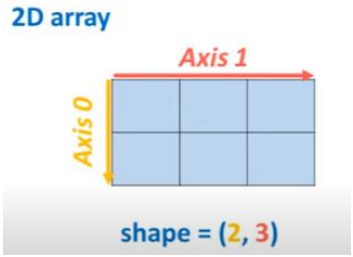
Nous allons voir comment naviguer à l'intérieur d'un tableau Numpy avec des méthodes similaires aux listes, à savoir :

- **Indexing,**
- **Slicing**
- **Subsetting**
- **Boolean indexing**

Sauf que naviguer dans un espace à n dimensions, c'est plus compliqué que dans une liste à 1 dimension !!

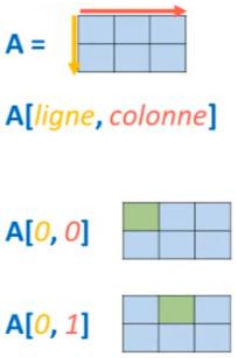
Nuage de valeurs d'un tableau numpy n dimensions

Rappel : sur un tableau 2D, il y a 2 axes.



2.1. Indexing

Il s'agit d'accéder aux éléments via leurs index de ligne et colonne.



Tester vous avec cet exemple :

```
A=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(A)
```

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

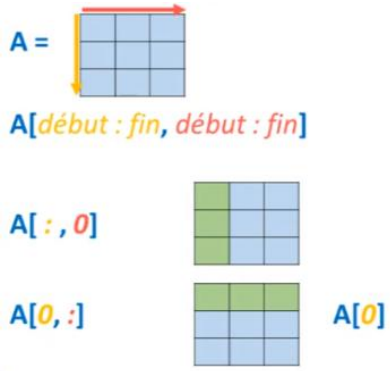
```
print(A[1,1])
```

```
5
```

Méthodes pratiques sur Numpy

2.2. Slicing

Il s'agit d'accéder aux éléments par slicing avec un index de début, un index de fin et optionnellement un pas.



Tester vous avec notre exemple de tableau précédent:

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

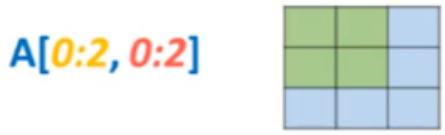
```
print(A[:,0])
```
→ *Retourne tous les éléments colonne 0*

```
[1 4 7]
```

```
print(A[0,:])
```
→ *Retourne tous les éléments ligne 0 mais A[0] marche aussi…*

```
[1 2 3]
```

Sélection d'un bloc dans le tableau :



Tester vous avec notre exemple de tableau précédent:

```
print(A[0:2,0:2])
```

```
[[1 2]
 [4 5]]
```

2.3. Subsetting

Il s'agit maintenant de **modifier des parties d'un tableau, extraire une partie** …etc…

Par exemple, toujours avec notre tableau précédent, on a construit un tableau B de dimension 2*2 extrait du tableau A :

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

```
B=A[0:2,0:2]
print(B)
```
→ *Extraction voulue via indices*

```
[[1 2]
 [4 5]]
```

On peut **aussi remplacer une partie du tableau par d'autres valeurs** :

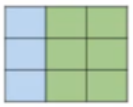
```
A[0:2,0:2]=10
print(A)
```

```
[[10 10  3]
 [10 10  6]
 [ 7  8  9]]
```

Méthodes pratiques sur Numpy

Exercice 1: On veut extraire la zone grisée, comment faire ?

A[?, ?]



On reprend notre dernier tableau et on récupère les 2 dernières colonnes :

```
[[10 10  3]
 [10 10  6]
 [ 7  8  9]]
```

```
B=A[:,1:]
print(B)
```

→ *Extraction voulue via indices (2 solutions)* ←

```
B=A[:,-2:]
print(B)
```

```
[[10  3]
 [10  6]
 [ 8  9]]
```

Exercice 2: On veut mettre des 1 dans la zone grisée, comment faire ?

A[*début : fin*, *début : fin*]

A[?, ?]

On part d'un tableau 4*4 remplit de 0 :

```
A=np.zeros((4,4))
print(A)
```

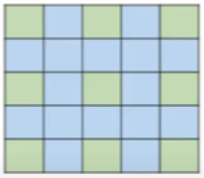
```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
A[1:3,1:3]=1
print(A)
```

```
[[0. 0. 0. 0.]
 [0. 1. 1. 0.]
 [0. 1. 1. 0.]
 [0. 0. 0. 0.]]
```

Exercice 3: On veut mettre des 1 dans les zones grisées, comment faire ?

A[::2, ::2]



On part d'un tableau 5*5 remplit de 0 :

```
A=np.zeros((5,5))
print(A)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

```
A[::2,::2]=1
print(A)
```

```
[[1. 0. 1. 0. 1.]
 [0. 0. 0. 0. 0.]
 [1. 0. 1. 0. 1.]
 [0. 0. 0. 0. 0.]
 [1. 0. 1. 0. 1.]]
```

SUPER pratique en traitement des images (cf info de PT)

Méthodes pratiques sur Numpy

2.4. Boolean indexing

Prenons un tableau A que l'on va rentrer de nombres aléatoires entre 0 et 10 dans un tableau de 5*5.

```
A=np.random.randint(0,10,[5,5])
print(A)
```

```
[[4 9 2 9 7]
 [8 5 8 8 0]
 [4 0 1 9 1]
 [1 1 4 8 0]
 [5 2 1 6 2]]
```

Avec ce tableau on peut faire des opérations de comparaisons. Exemple qui nous dit dans notre tableau qui est inférieur à 5 :

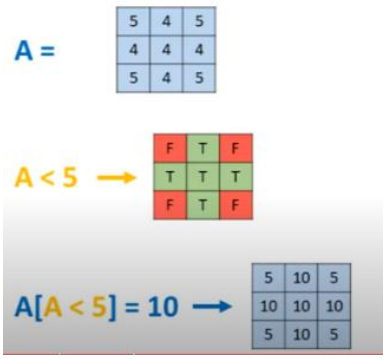
```
print(A<5)
```

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

Ce tableau avec true et False s'appelle un **masque**.

Vous pouvez maintenant sélectionner tous les éléments < à 5 et dire qu'ils valent maintenant 10 :

Principe :



Tester avec notre tableau ci-dessus :

```
A[A<5]=10
print(A)
```

```
[[10  9 10  9  7]
 [ 8  5  8  8 10]
 [10 10 10  9 10]
 [10 10 10  8 10]
 [ 5 10 10  6 10]]
```

On vient de faire du **Boolean Indexing**

Très intéressant en **analyse de données par rapport à des seuils** définis …

Tester avec une double condition :

```
A[(A<5) & (A>2)]=10
print(A)
```

```
[[10  9 10  9  7]
 [ 8  5  8  8 10]
 [10 10 10  9 10]
 [10 10 10  8 10]
 [ 5 10 10  6 10]]
```

3. Algèbre linéaire avec Numpy

Nous allons voir comment travailler avec les tableaux en algèbre linéaire, car ce sont des matrices …

Méthodes pratiques sur Numpy

Je commence par créer 2 matrices remplies de 0 A et B :

```
A=np.ones((2,3))
print(A)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
B=np.ones((3,2))
print(B)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

3.1. Transposée de matrice :

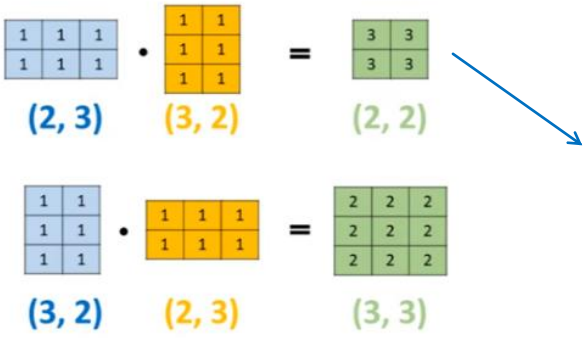
On utilise la méthode T qui veut dire fait la transposée :
Exemple avec la transposée de A :

```
print(A.T)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

3.2. Somme et produit matriciel :

On va maintenant faire un produit matriciel entre A et B.
On utilise la méthode dot ou l'opérateur *.



Idem pour une somme de matrice…

```
A=np.array(([1,2],[3,4]))
B=np.array(([5,6],[7,8]))
C=A+B
print(C)
```

```
[[ 6  8]
 [10 12]]
```

Vous pouvez tester par exemple :

```
print(A.dot(B))
```

```
[[3. 3.]
 [3. 3.]]
```

Ou encore :

```
A=np.array(([1,2],[3,4]))
B=np.array(([5,6],[7,8]))
C=A*B
print(C)
```

```
[[ 5 12]
 [21 32]]
```

Trop fort !!

Méthodes pratiques sur Numpy

L'algèbre linéaire avec Numpy se fait via le package **numpy.linalg** qui présente plein de méthodes…(déterminant, inversion matrice, valeurs propres…)

Exemple du calcul du déterminant :

Voici un exemple avec une matrice carrée aléatoire :

```
A=np.random.randint(0,10,[3,3])
print(A)
```

```
[[8 6 2]
 [4 2 3]
 [7 3 2]]
```

```
print(np.linalg.det(A))
```

```
33.99999999999999
```

→ *Valeur déterminant*

Exemple inversion matrice (car déterminant non nul) :

Avec ma matrice ci-dessus

```
print(np.linalg.inv(A))
```

```
[[-0.14705882 -0.17647059  0.41176471]
 [ 0.38235294  0.05882353 -0.47058824]
 [-0.05882353  0.52941176 -0.23529412]]
```

Il y a plein de méthodes sous linalg…utiliser le guide Numpy si besoin de méthodes particupières…

3.3. Exercice d'application à la résolution d'un système de n équations à n inconnues :

Pour résoudre un système de Cramer AX=B (n équations à n inconnues) avec Numpy, il suffit d'importer *linalg.solve*
A et B sont les 2 matrices du système à résoudre.

Exemple :

$$\begin{cases} 2x+2y-3z=2 \\ -2x-y-3z=-5 \\ 6x+4y+4z=16 \end{cases}$$

On note $A=\begin{pmatrix} 2 & 2 & -3 \\ -2 & -1 & -3 \\ 6 & 4 & 4 \end{pmatrix}$ et $B=\begin{pmatrix} 2 \\ -5 \\ 16 \end{pmatrix}$

Préparer votre début de code ainsi :

```
from copy import *
from math import *
import numpy as np
from numpy import linalg
```

Méthodes pratiques sur Numpy

Résoudre les systèmes linéaires suivant à l'aide de *linalg* et de sa fonction *linsolve* : np.linalg.solve(A,B)

$$\begin{cases} 2x + 3y = 5 \\ 5x - 2y = -16 \end{cases}, \quad \begin{cases} 2x + 2y - 3z = 2 \\ y - 6z = -3 \\ z = 4 \end{cases} \quad \text{et} \quad \begin{cases} 2x + 2y - 3z = 2 \\ -2x - y - 3z = -5 \\ 6x + 4y + 4z = 16 \end{cases}.$$

Voilà, on a terminé de parcourir le package NUMPY minimum, qui vous sera bien utile pour votre TIPE, pour l'épreuve de modélisation et vos futurs calculs scientifiques d'ingénieurs.

Synthèse : Tableau vs Liste de listes

Tableau	Liste de listes
éléments du même type	éléments de type quelconque
nombre d'éléments fixé	ajout possible (concaténation)
`+` : ajouter des éléments	`+` : concaténer
`*` : multiplier	`*` : répéter
`==` : comparaison terme à terme	`==` : comparaison globale
`tab[i,j]`	`L[i][j]`
`n,m = tab.shape`	`n = len(L) ; m = len(L[0])`