LE MODULE NUMPY

Il n'est pas question de connaître par cœur toutes les commandes présentées ici. Vous devez surtout retenir le fonctionnement de `numpy` et connaître quelques commandes essentielles. Pour les autres, soit vous les retrouverez dans ce cours ou dans toute autre documentation (utilisez l'autocomplétion de Pyzo), soit vous les programmerez à nouveau.

La documentation officielle est disponible sur le site https://docs.scipy.org/doc/numpy/ et elle est extrêmement claire !

1 DE LA LISTE AU TABLEAU

A Chargement du module

Nous chargerons l'ensemble du module numpy avec la commande

```
import numpy as np
```

Python 1: Numpy : Import du module

Tous les noms de fonction de ce module seront préfixées par "`np.`". Cela permet de distinguer les fonctions qui viennent de ce module par rapport aux autres et d'éviter les risques liés aux homonymies[1].

Pas de préfixe pour les méthodes : On remarquera que le préfixe "`np.`" n'est pas utile lorsqu'il s'agit d'une méthode que l'on applique à un objet numpy. En effet, une méthode est une fonction qui se trouve programmée *dans* l'objet lui-même et Python ne la cherche donc pas dans une bibliothèque, mais directement dans l'objet qui est de type tableau.

B Spécificité des tableaux

Un tableau ressemble beaucoup à une liste, mais en plus *rigide* :

- Un tableau a une taille fixée lors de sa création.
  On ne peut donc pas lui rajouter ou enlever des éléments comme avec une liste. Ici, pas de méthode `append`, `pop`...
- Tous les éléments d'un tableau doivent être du même type[2] (lui aussi fixé à la création).

Ces contraintes permettent d'optimiser la gestion mémoire et les opérations sur les tableaux. Ainsi, à la création, la taille et le type des données du tableau permettent à Python de savoir exactement quelle espace mémoire prendra le tableau et cela ne variera plus.

Les accès aux éléments du tableau, les opérations pourront être optimisés et plus rapides qu'avec des listes.

C Création de tableaux à partir de listes

On peut créer un tableau à partir d'une liste. Ce tableau peut avoir une ou plusieurs dimensions (par exemple pour faire des matrices de taille $n \times p$).

```
liste = [1,2,3,4,5]
tableau = np.array(liste)                    # crée un tableau à partir de la liste
type(tableau)                                # numpy.ndarray
tableau.dtype                                # int64 : type des éléments du tableau
matrice = np.array([[ 1,0,0],[0,1,0]])       # matrice de taille 2x3
tab = np.array([1,'a',2])                    # erreur : des types différents
```

Python 2: Numpy : Création d'un tableau

La commande `arange` fonctionne comme la commande `range` mais pour créer un tableau. On en profite pour retrouver une autre commande que l'on connaît déjà `linspace`.

[1] Deux fonctions issues de deux bibliothèques différentes peuvent porter le même nom et avoir un comportement différent. C'est le cas de `randint(a,b)` selon qu'il est issu de `random` ou de `math`. L'utilisation d'un alias lors du chargement des bibliothèques permet alors de distinguer ces deux fonctions homonymes.

[2] Il existe une méthode pour contourner cette contrainte, mais ce n'est pas l'*objet* ici.

```
tableau = np.array(range(100))
tableau = np.arange(100)              # même résultat
tableau = np.arange(-5,6)
tableau = np.arange(-5,6.4,.2)        # la commande accepte des arguments non entier
tableau = np.linspace(-5,6,100)       # le troisième argument est le nombre de points.
```

Python 3: Numpy : Commandes arange et linspace

Réciproquement, on peut transformer un tableau en liste avec la méthode `tolist()` (cela crée un nouvel objet).

```
matrice.tolist()              # crée une liste à partir du tableau
type(matrice)                 # renvoie numpy.ndarray car matrice n'est pas modifiée
matrice = matrice.tolist()    # remplace le tableau par une liste.
matrice = array(matrice)      # revient au tableau
```

Python 4: Numpy : passage des tableaux aux listes

Exercice

Quelle est la différence entre la fonction `list` et la méthode `tolist()` ?

D Attributs d'un tableau

```
matrice.dtype                  # type des éléments du tableau
matrice.shape                  # dimension sous la forme d'un tuple
matrice.shape[0]               # nombre de lignes d'une matrice
matrice.shape[1]               # nombre de colonnes d'une matrice
matrice.size                   # nombre total de coefficients
matrice.ndim                   # nombre de dimensions du tableau (2 pour matrice)
```

Python 5: Numpy : Attributs d'un tableau

E Lecture-écriture

Le parcours des tableaux est très proche de celui des listes avec le slicing.

Par contre, lorsqu'il y a plusieurs coordonnées, on accède à l'élément en mettant un `tuple` entre les crochets : on sépare les coordonnées par des virgules au lieu de les mettre dans des crochets successifs comme nous le faisions avec les listes de listes.

```
tableau = np.arange(10)
matrice = np.array([[10*j+i for i in range(10)] for j in range(8)])

tableau[4]                # élément d'indice 4 dans le tableau
matrice[4]                # tableau de la ligne d'indice 4
matrice[4,:]              # idem
matrice[4,3]              # élément d'indices 4,3
matrice[5:,5:]            # bloc inférieur droit
matrice[::-1]             # inverse l'ordre des lignes
```

Python 6: Numpy : slicing et accès aux coefficients

```
# cas des listes de listes
lst =  [[10*j+i for i in range(10)] for j in range(8)]
lst[5][2]                 # dans la liste lst[5], je demande l'élément 2

# cas des tableaux à deux dimensions
matrice = np.array(lst)
matrice[5,2]              # je demande le coefficient de coordonnées 5,2 du tableau.
matrice[5][2]             # Numpy est gentil : fonctionne aussi.
```

Python 7: Numpy : différence d'accès listes - tableaux

```
tableau[4] = 20          # modifie l'élément d'indice 4
tableau[4] = 'a'         # erreur, pas le bon type
tableau[4] = 19.5        # remplace par int(19.5)=19
matrice[5,7] = 12
matrice[:5,:5] = 0       # remplace le bloc 5x5 en haut à gauche par des 0
matrice[:2,1] = [-1,-2] # remplace les deux premières coordonnées de la colonne 1
matrice[:2,1] = [1,2,3] # erreur, mauvaise dimension
```

Python 8: Numpy : écriture dans un tableau

F Difficultés avec les pointeurs mémoire

Il faut faire attention aux effets de gestion de la mémoire dans Python.

Contrairement aux listes, lorsque vous utilisez le *slicing*, Python ne recopie pas la partie du tableau concernée, mais fait pointer une nouvelle variable vers un extrait du même tableau.

Si vous modifiez le contenu de cette nouvelle variable, cela modifiera aussi le tableau initial.

La méthode `copy()` permet de faire une copie complète du tableau.

```
# ****  avec les listes  ****
liste = [0,1,2,3,4]
liste2 = liste
liste2 is liste               # True : même objet mémoire

liste3 = liste[:]
liste3 is liste               # False : nouvel objet mémoire

liste4 = liste.copy()
liste4 is liste               # False : nouvel objet mémoire

# **** avec les tableaux ****
tab = np.array([0,1,2,3,4])
tab2 = tab
tab2 is tab                   # True : même objet mémoire

tab3 = tab[:]
tab3 is tab                   # /!\ True : même objet mémoire

tab4 = tab.copy()
tab4 is tab                   # False : nouvel objet mémoire

# ****      exemples    ****
matrice = np.array([[10*j+i for i in range 10] for j in range 7])

extrait = matrice[:5,:5]     # bloc 5x5 en haut à gauche de matrice
extrait[:,:] = 0             # remplit extrait avec des zéros
print(matrice)               # le bloc est aussi modifié dans matrice
```

Python 9: Numpy : Difficultés avec les pointeurs mémoire

G Extraction d'un sous-tableau

Lorsque les cellules que l'on veut extraire sont contiguës (ou de 2 en 2...) le slicing permet d'extraire facilement un sous-tableau de celui existant.

On peut vouloir extraire de façon plus manuelle et créer un tableaux à partir des coefficients d'un autre.

Pour cela on utilise le *fancy indexing* : si on veut un tableau qui contienne certains coefficients extraits d'un autre tableau, on crée le tableau avec les coordonnées des dits coefficients. C'est plus simple à comprendre sur un exemple :

```
tableau = np.arange(0,40,4))        # tableau initial
indices = np.array([2,4,6],[1,3,5]) # indices à extraire
```

```
tableau[indices]                    # tableau de taille 2x3 avec les coefficients
    indiqués par indices.
```

Python 10: Numpy : fancy indexing

H Parcours itératif d'un tableau

Un tableau est un objet *itérable*, il peut être parcouru avec une boucle **for** par exemple.

Par contre, seule sa première dimension est parcourue. Ainsi, pour une matrice à deux dimensions, la variable parcourra les lignes (qui sont des tableaux).

```
tableau = np.arange(0,40,4)         # tableau initial
for i in tableau:
  print(i)

matrice = [[5*j+i for i in range(5)] for j in range(3)]
for i in matrice:
  print(i)                          # affiche les lignes de matrice

# que fait cette procédure ?
for i in matrice:
  for j in i:
    print(j)
```

Python 11: Numpy : Parcours d'un tableau

Pour parcourir l'intégralité d'un tableau, on peut *l'aplatir* avec la méthode `.flat`. Cela revient à mettre les lignes bout-à-bout.

Petites subtilités : la méthode `.flat` se contente de créer *un parcours*[3] de tableau (elle ne créé donc pas un nouveau tableau, mais indique seulement comment parcourir celui existant).

Il existe deux autres méthodes qui ressemblent. La méthode `.flatten()` crée qui crée un nouveau tableau applati et la méthode `.ravel()` crée *une vue* applatie du tableau.

Ainsi, avec la méthode `.flatten()`, on créé un nouvel objet mémoire qui existe indépendemment du tableau initial.

Avec la méthode `.ravel()`, on pointe vers le même objet mémoire, mais présenté sous forme applatie. Cette méthode va donc plus vite, mais si vous modifiez le tableau obtenu avec cette méthode, vous modifierez aussi le tableau *source*.

La méthode `.flat` ne sert que pour un parcours (avec une boucle `for` par exemple. L'objet renvoyé n'est pas un tableau, mais un itérateur.

Pour changer la forme d'un tableau, voir les méthodes `reshape` (nouvelle vue) et `resize` (nouveau tableau).

```
matrice = np.array([[5*j+i for i in range(5)] for j in range(3)])
for i in matrice.flat:
  print(i)

sum(x for x in matrice)             # renvoie un tableau avec les sommes des colonnes
sum(x for x in matrice.flat)        # somme de toute la matrice.
```

Python 12: Numpy : aplatissement d'un tableau

2 CRÉATION DES TABLEAUX USUELS

En général, il est suffisant de construire des tableaux à partir de listes en utilisant les listes par compréhension autant que possible. Il existe néanmoins des commandes pour créer les tableaux usuels. En voici quelques unes.

Pour chacune des commandes qui suivent, vous devez être capable de générer le même tableau avec une liste.

On commencera pas se rappeler les commandes `arange` et `linspace`, déjà vues plus haut dans le programme 3.

[3] On appelle cela un itérateur en Python.

A Les tableaux nuls

Ils sont construits par la fonction `zeros(`*`taille`*`)` où *taille* est un tuple qui contient les dimensions.

Par défaut, les éléments du tableau sont de type `float` (modifiable avec un argument optionnel).

```
np.zeros(10)                    # tableau à une dimension de taille 10 rempli de 0
np.zeros((3,5))                 # taille 3x5 ne pas oublier les doubles parenthèses.
np.zeros((3,5),int)             # de type entier.
```

Python 13: Numpy : command zeros

B Les tableaux unitaires

Ils ne comportent que des 1 et sont construits par la fonction `ones(`*`taille`*`)`. La syntaxe est la même que la fonction `zeros`.

```
np.ones(10)                     # tableau à une dimension de taille 10 rempli de 1
np.ones((3,5))                  # taille 3x5 rempli de 1
5*np.ones((3,5))                # taille 3x5 rempli de 5
np.ones((3,5),int)              # de type entier.
```

Python 14: Numpy : command ones

C La matrice identité

Elle se construit avec la commande `eye(`*`taille`*`)`.

Avec des arguments supplémentaires, elle permet de décaler la diagonale de 1 ou de réaliser des matrices rectangulaires.

```
np.eye(4)                       # identité de taille 4x4
np.eye(4,dtype=int)             # identité de taille 4x4, de type int
np.eye(4,k=1)                   # surdiagonale avec des 1
np.eye(4,k=-1)                  # sousdiagonale avec des 1
```

Python 15: Numpy : command eye

D Les matrices diagonales

Elles sont définies par leur diagonale.

```
np.diag([1,1,1])                # identité de taille 3x3
np.diag([1,2,3,4])              # taille 4x4 avec 1,2,3,4 en diagonale
np.diag([1,2,3],k=1)            # taille 4x4, surdiagonale avec 1, 2, 3
np.diag([1,2,3],k=-1)           # sousdiagonale
```

Python 16: Numpy : command diag

E Les matrices triangulaires inférieures

Elles sont remplies de 1 sous la diagonale (au sens large).

```
np.tri(3)                       # triangulaire inférieure de taille 3x3
```

Python 17: Numpy : command tri

F Éviter les listes par compréhension

Pour créer un tableau dont les valeurs dépendent des indices dans le tableau, on utilisait jusqu'à présent les listes par compréhension. On peut le faire directement avec la fonction `fromfunction`.

Cette méthode est meilleure car elle évite de créer une liste intermédiaire.

```
def f (i,j): return 5*i+j
matrice = np.fromfunction(f,(3,5))

# ou plus simplement
matrice = np.fromfunction(lambda i,j:5*i+j,(3,5))

#tableau à une dimension
matrice = np.fromfunction(lambda x:2*x-1,(10,))
```

Remarque : il ne faut pas de parenthèses autour du couple `i,j` dans la définition avec **lambda**.
Lorsque le tableau n'a qu'une seule dimension, il faut indiquer $(n,)$ pour l'argument shape.

G Tableaux aléatoires

De nombreuses fonctions de la bibliothèque `random` ont leur équivalent dans le *sous-module* `random` de `numpy`.

Malheureusement, il n'existe pas de `randrange` dans `numpy`.

```
matrice = np.random.rand(3,5)
matrice = np.random.randint(100,size=(3,5))
matrice = np.random.randint(100,size=5)
```

3 Opérations matricielles

A Opérations terme à terme

Par défaut, toutes les opérations entre tableau s'effectuent terme à terme. C'est le cas des opérations $+, \times \ldots$, mais aussi des comparaisons...

Ainsi, lorsque l'on effectue une opération usuelle entre une matrice et un scalaire, cette opération est appliquée à chacun des coefficients.

On peut également appliquer les fonctions usuelles sur les tableaux. Ces fonctions sont alors appliquées coefficient par coefficient (il faut ajouter le préfixe `np.` au nom de la fonction).

Attention : Le produit `*` entre matrice est un produit coefficient par coefficient, ce n'est pas le produit matriciel.

```
J2 = 2*np.ones((3,3))          # matrice 3x3 remplie de 2
id = np.eye(3)
tableau = np.arange(10)

2*J2                           # matrice 3x3 remplie de 4
id+J2                          # addition matricielle : terme à terme
id*J2                          # différent de J2, c'est la matrice 2*id

np.sqrt(tableau)               # racine carrée des coefficients
np.log(tableau+1)              #log népérien
np.sin(tableau)

tableau < 3                    # [True,True,True,False,False,...]
```

Python 18: Numpy : Opérations usuelles

Pour modifier la matrice, on peut utiliser les commandes `+=`, `*=`... auxquelles nous sommes habitués. Par contre, le comportement est un peu différent de celui des listes. En particulier, la matrice n'est pas réaffectée, mais modifiée en place.

```
J2 = np.ones((3,3))          # matrice 3x3 remplie de 1
J2 *= 2                      # mutliplie les coefficients par 2 -> 2
J2 += 1                      # et ajoute un à chaque coefficients -> 3
```

Python 19: Numpy : Opérations en place

B Vectorisation d'une fonction

Lorsque l'on définit une fonction, en général, on ne peut pas l'appliquer à un tableau numpy (terme à terme).

Pour pouvoir le faire, on *vectorise* cette fonction avec la commande `vectorize`.

```
from math import sqrt

tableau = np.arange(10)
def f(x):
  return (x-1)/(sqrt(x+1))

f(tableau)                    #erreur

vf = np.vectorize(f)
vf(tableau)
```

Python 20: Numpy : Vectorisation

C Transposition

```
matrice = np.fromfunction(lambda i,j:5*i+j,(3,5))
matrice.transpose()
matrice.T
```

Python 21: Numpy : Transposition d'une matrice

D Produit matriciel, puissances, inverse

```
matrice = np.fromfunction(lambda i,j:5*i+j,(3,5))
J = np.ones((3,3))
id = np.eye(3)

#Produit matriciel
np.dot(J,matrice)        # produit matriciel J*matrice
np.dot(id,J)             # =J
np.dot(J,id)             # =J
np.dot(matrice,J)        # erreur dû au dimensions
np.dot(matrice.T,J)

#Puissances
np.linalg.matrix_power(J,5)

#Inverse
np.linalg.inv(J)         # erreur, pas inversible
np.linalg.inv(id)        # elle-même
np.linalg.inv(id+J)      # inverse (valeurs approchées)

#Système matriciel
b = np.array([0,0,4,5,-1])
a = np.ones(5)-np.tri(5,k=-1)
np.linalg.solve(a,b)     # résout ax=b
```

Python 22: Numpy : Operations matricielles

La résolution des systèmes avec `linalg.solve` ne fonctionne que lorsqu'il s'agit d'un système de Cramer.

4 TESTS AVEC LES TABLEAUX

Pour tester si deux tableaux sont égaux (terme à terme), on utilise la commande `array_equal`.

Nota : la commande "`a==b`" renvoie une matrice de même taille que `a` et `b` contenant `True` ou `False` pour chaque position dans le tableau. C'est le tableau des $(a_k == b_k)_{k\in I}$.

Il en est de même avec les autres opérateurs logiques.

```
a = np.ones((3,5))
b = a.copy()
c = eye(3,5)
np.array_equal(a,b)          # True
np.array_equal(a,c)          # False
a == b                       # Tableau de taille 3x5 rempli de True
```

Python 23: Numpy : Égalité de tableaux

On peut aussi utiliser les méthodes "`.any()`" et "`.all()`" qui testent respectivement si l'un ou la totalité des éléments du tableau valent « `True` ».

```
a = np.ones((3,5))
b = a.copy()
c = np.eye(3,5)
d = np.zeros((3,5))
(a == b).any()        # True
(a == b).all()        # True
(a == c).any()        # True
(a == c).all()        # False
(a == d).any()        # False
(a < 1).any()         # False
(c < 1).any()         # True
```

Python 24: Numpy : méthodes any et all

Remarque importante : La fonction `allclose` teste si deux tableaux sont égaux à une incertitude près (absolue et relative). Souvent, cette fonction sera plus utile que le test de l'égalité exacte. En effet, Python réalise des erreurs d'arrondis dans ses calculs, et souvent, deux objets qui devraient être rigoureusement égaux mathématiquement, diffèrent très légèrement dans Python pour ces raisons d'arrondis.

On pourra aussi consulter la fonction `isclose`.

5 DES STATISTIQUES ET PROBABILITÉS AVEC NUMPY

A Fonctions statistiques

```
serie = np.random.randint(10,100)
serie.max()                              # max
serie.min()                              # min
serie.sum()                              # somme
serie.mean()                             # moyenne
serie.mean()==serie.sum()/serie.size     # True
serie.var()                              # variance
serie.std()                              # écart-type
serie.median()                           # médiane
```

Python 25: Numpy : Statistiques univariées

Python donne aussi la matrice de covariance de deux séries statistiques. La matrice est ainsi définie

$$\text{cov(x,y)} = \begin{pmatrix} \sigma_x^2 & \sigma_{x,y} \\ \sigma_{x,y} & \sigma_y^2 \end{pmatrix}$$

La coefficient de corrélation est aussi donné dans la matrice de corrélation

$$\text{R(x,y)} = \begin{pmatrix} 1 & \rho(x,y) \\ \rho(x,y) & 1 \end{pmatrix}$$

```
seriea = np.random.randint(10,100)
serieb = np.random.randint(10,100)
np.cov(seriea,serieb)
np.corrcoef(seriea,serieb)
```

Python 26: Numpy : Statistiques bivariées

Dans la suite de cette section, on utilisera le sous-module `random` de numpy. Nous l'avons déjà utilisé pour créer des matrices avec des coefficients aléatoires dans le paragraphe G.

B Mélange aléatoire d'un tableau

La commande `random.shuffle` permet de mélanger un tableau en place.

```
urne = np.arange(10)
np.random.shuffle(urne)                    # urne est modifiée
```

Python 27: Numpy : Mélange d'un tableau

C Tirage dans une urne

La commande `random.choice` permet de tirer un élément (ou un échantillon) de façon aléatoire dans un tableau à une dimension.

Lorsque l'on tire plusieurs éléments successifs, l'argument optionnel `replace=False` permet de faire un tirage sans remise.

```
urne = np.arange(10)
elt = np.random.choice(urne)               # tirage d'un élément dans urne
np.random.choice(urne,5)                   # tirage de 5 éléments avec remise
np.random.choice(urne,5,replace=False)     # tirage de 5 éléments sans remise
np.random.choice(urne,11,replace=False)    # erreur
np.random.choice(6,1,replace=False)        # tirage de 1 élément dans [0,6[
```

Python 28: Numpy : Tirage dans une urne

On peut aussi rajouter un argument optionnel pour préciser les probabilités de tirage de chaque élément (par défaut, les tirages sont équiprobables).

D Loi binomiale

La loi binomiale $\mathscr{B}(n,p)$ est modélisée par la fonction `random.binomial(n,p)`. Celle-ci renvoie le nombre de succès après n obtenus lors de la réalistion de n épreuves de Bernoulli indépendantes. La probabilité de succès à chaque épreuve est donnée par $p \in [0,1]$.

La probabilité d'obtenir k succès est alors

$$P(X=k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Un argument optionnel `size=nb` permet de réaliser `nb` fois l'expérience.

```
np.random.binomial(50,0.6)          # nombre de succès selon la loi binomiale B(n,p)
np.random.binomial(50,0.6,size=10)  # réaliser 10 fois l'expérience
np.random.binomial(50,0.6,10)       # idem
```

Python 29: Numpy : Loi binomiale

E Loi géométrique

La loi géométrique est modélisée par la fonction `random.geometric(n,p)`. La fonction renvoie le numéro du tirage correspondant au premier succès lors de la répétition d'épreuves de Bernoulli indépendantes et de paramètre $p \in [0,1]$.

La probabilité d'obtenir le premier succès au $k^{\text{ième}}$ tirage est alors

$$P(X=k) = (1-p)^{k-1}p$$

```
np.random.geometric(0.001)          # premier succès pour p=0.001
np.random.geometric(0.001,size=10)  # réaliser 10 fois l'expérience
np.random.geometric(0.001,10)       # idem
```

Python 30: Numpy : Loi géométrique

F Loi hypergéometrique

La loi hypergéométrique est modélisée par la fonction `random.hypergeometric(ngood,nbad,nsample)`. La fonction renvoie le nombre de succès lors du tirage (sans remise) d'un échantillon de taille `nsample` dans une urne contenant `ngood` cas favorables et `nbad` cas défavorables.

La probabilité d'obtenir k succès est alors

$$P(X=k) = \frac{\binom{\texttt{ngood}}{k}\binom{\texttt{nbad}}{\texttt{nsample}-k}}{\binom{\texttt{ngood+nbad}}{\texttt{nsample}}}$$

```
np.random.hypergeometric(80,20,5)         # nombre de boules blanches lorsque l'on en
    tire 5 dans une urne qui en contient 80 et 20 noires
np.random.hypergeometric(80,20,5,size=10)# réaliser 10 fois l'expérience
np.random.hypergeometric(80,20,5,10)      # idem
```

Python 31: Numpy : Loi hypergéométrique

6 QUELQUES COMMANDES POUR ALLER PLUS LOIN

La connaissance des commandes suivantes est complètement hors des exigences du programme. Elles sont présentées ici pour leur intérêt intrinsèque, mais aussi parce que nous les utiliserons dans le TD sur les images en vue d'optimiser certaines manipulations.

A Notation d'Einstein pour les sommes

Cette écriture des sommes n'est pas due à Einstein lui-même, mais il l'a popularisé. Elle est très largement utilisée par les physiciens. Ce n'est qu'une notation. Elle n'apporte donc rien de nouveau conceptuellement, mais elle permet d'alléger les écritures. L'idée de base et de ne pas écrire le signe somme $\sum$, mais de le laisser sous-entendu : lorsqu'un indice se répète entre deux coefficients, alors, il s'agit d'une somme sur cet indice (les bornes de sommation sont sous-entendues).

Par exemple, le produit matriciel peut s'écrire très simplement :

$$a_{i,k}b_{k,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j}$$

Dans la première écriture, l'indice k se répète et indique donc que l'on fait la somme de ces produits pour k variant dans son ensemble d'indices naturel.

L'indice c_k du produit entre deux polynômes s'écrit :

$$a_i b_{k-i} = \sum_i a_i b_{k-i}$$

Ici, c'est l'indice i qui est répété et qui sert donc d'indice de sommation.

On peut aussi avoir des sommes doubles, triples... si on repète plusieurs indices.

Numpy propose une méthode de calcul très puissante qui se base sur cette notation.

Pour simplifier, nous ne l'utiliserons qu'avec deux tableaux.

Exemple avec le produit matriciel :

On part de deux matrices A et B avec deux dimensions chacune. Chaque coefficient est donc défini par deux coordonnées. On les a appelé ik pour A et kj pour B.

À partir de ces coefficients, on construit une nouvelle matrice aussi à deux dimensions ij, dont les coefficients valent avec la notation d'Einstein

$$C[i,j]] = A[i,k]B[k,j] = \sum_k A[i,k]B[k,j]$$

On écrit donc le code :

```
A = np.array([[1,2,1],[2,2,1],[-1,0,3]])
B = np.array([[1,0,0],[1,2,1],[-1,0,1]])

C = np.einsum('ik,kj->ij',A,B)
(C == A.dot(B)).all()
```

La dernière commande sert à vérifier que l'on obtient la même matrice qu'avec le produit matriciel obtenu par la commande `dot`.

Somme des coefficients diagonaux d'une matrice :

```
A = np.array([[1,2,1],[2,2,1],[-1,0,3]])

tr = np.einsum('ii->',A)
```

Extraction de la diagonale :

On construit le tableau dont les coefficients sont ceux de la diagonale de A : on extraie les $A[i,i]$. Pour cela, il ne faut pas faire une somme d'indice i. Cet indice i ne disparait donc pas et se retrouve dans le résultat (après le flèche).

```
A = np.array([[1,2,1],[2,2,1],[-1,0,3]])

D = np.einsum('ii->i',A)
```

B Modifier la forme du tableau, les concaténer...

Ces fonctions sont particulièrement utiles lors d'un travail sur des images, la figure 1 donne des exemples.

Les fonctions de concaténation

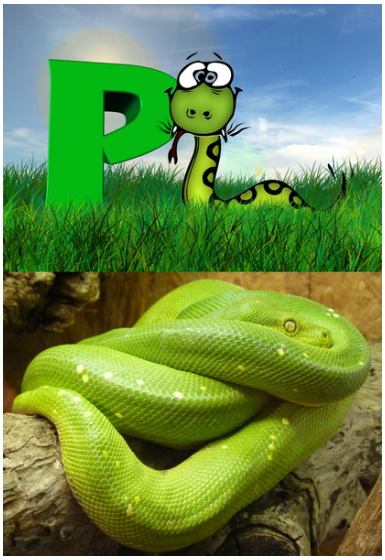
- **vtack** concatène deux tableaux *verticalement* : les lignes sont mises à la suite les unes des autres (voir la figure 1a).
- **hstack** réalise la même chose avec les colonnes (voir la figure 1b).
- **concatenate** généralise les fonctions précédentes en donnant le choix de l'axe suivant lequel faire la concaténation.

  Par exemple `concatenate((A,B),axis = 0` revient à faire `vstack(A,B)` et `concatenate((A,B),axis = 1` revient à faire `hstack(A,B)`.

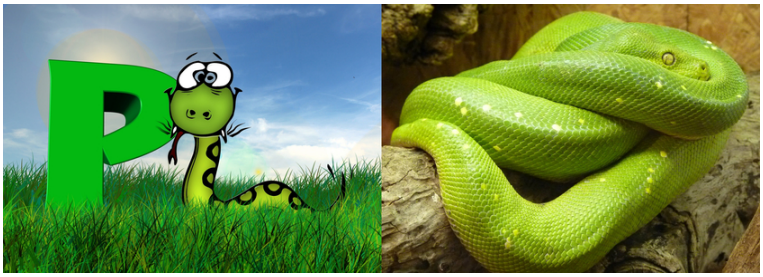
La fonction de réplication **tile(A,reps)** crée un nouveau tableau en répétant A, un certain nombre de fois.

`reps` indique les répétitions à effectuer. Par exemple, si A est une matrice de taille 2×2, alors `reps` doit être un couple. Chaque coefficient correspond au nombre de répétitions de l'axe considéré.

Par exemple, si A est de dimensions $M\times N$, alors `tile(A,[2,3])` est de dimension $2M\times 3N$. Chaque ligne de A est "doublée" et chaque colonne est triplée.



(a) Fonction vstack



(b) Fonction hstack

(c) Fonction tile

Figure 1: Manipulation d'images

```
A = np.array([[1, 2], [3, 4]])
  array([[1, 2],
         [3, 4]])
np.tile(A, 2)
  array([[1, 2, 1, 2],
         [3, 4, 3, 4]])

np.tile(A, (2, 1))
  array([[1, 2],
         [3, 4],
         [1, 2],
         [3, 4]])
B = np.array([1,2,3,4])
np.tile(B,(4,1))
  array([[1, 2, 3, 4],
         [1, 2, 3, 4],
         [1, 2, 3, 4],
         [1, 2, 3, 4]])
```

On a utilisé cette fonction pour l'image 1c. Cela revient à utiliser la fonction `concatenate` avec plusieurs fois le même tableau.

La commande newaxis permet de créer un nouvel axe (une nouvelle dimension) dans un tableau lors d'un slicing.

```
A = np.array([[1, 2], [3, 4]])
A.shape
    (2,2)
A[:,newaxis,:].shape
```

```
    (2,1,2)
```

C Comparaisons avancées entre tableaux :

La fonction where renvoie un tableau issu de la comparaison de deux autres.

Par exemple, si x et y sont deux tableaux de même taille, `np.where(x<y,x,y)` renvoie un tableau de même taille où chaque coefficient est le plus petit entre celui de x et celui de y.

`np.where(x<y,100,y)` renvoie le tableau où chaque coefficient vaut 100 si le coefficient de x est plus grand que celui de y, et celui de y sinon.

```
x = np.arange(9.).reshape(3, 3)
  array([[ 0.,  1.,  2.],
         [ 3.,  4.,  5.],
         [ 6.,  7.,  8.]])
np.where(x < 5, x, -1)
  array([[ 0.,  1.,  2.],
          [ 3.,  4., -1.],
          [-1., -1., -1.]])
```

Les fonctions maximum, minimum sont des cas particuliers de `where`.

`maximum(x,y)` renvoie le tableau dont chaque coefficient est le plus grand entre celui de x et celui de y. C'est la même chose que `np.where(x1 >= x2, x1, x2)`.

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

D À vous de chercher

Le module numpy possède de nombreuses autres possibilités : recherche dans un tableau, calculs statistiques, calculs avec les polynômes (Lagrange...)...À vous d'aller trouver la bonne fonction si vous ne voulez pas la programmer à nouveau.

N'hésitez pas à consulter l'aide en ligne : https://docs.scipy.org/doc/numpy/.