IFOSUP

Institut de Formation Supérieure
Ville de Wavre

MATHEMATIQUES APPLIQUEES A L'INFORMATIQUE
Bibliothèque NumPy pour Python

Gérard Barmarin

2023-2024

Importation de bibliothèques

Python intègre un certain nombre de fonctions mais il n'est pas possible d'y intégrer toutes les fonctions imaginables
Il existe donc des bibliothèques regroupant les fonctions utiles dans un domaine particulier.

Le module math, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que sinus, cosinus, tangente, racine carrée, etc. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script python :

```
from math import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant toutes les fonctions (c'est la signification du symbole *) du module math qui contient une bibliothèque de fonctions mathématiques pré-programmées. Vous pouvez ensuite les utiliser… si vous connaissez leur nom et la structure des paramètres à passer à la fonction dans les parenthèses!

```
x = sin(angle) # pour assigner à la variable x le sinus de angle (en radians!)
```

Bibliothèque NumPy

NumPy (http://www.numpy.org/) pour «Numerical Python extensions» est une bibliothèque pour le langage Python, destinée à manipuler des matrices ou des tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux et qui permet d'effectuer des calculs numériques.

Cette bibliothèque open source fournit de multiples fonctions permettant notamment de créer directement un tableau depuis un fichier ou au contraire de sauvegarder un tableau dans un fichier, de manipuler des vecteurs, des matrices et des polynômes.

NumPy est la base de SciPy, regroupement de bibliothèques Python autour du calcul scientifique qui contient des modules pour l'optimisation, l'algèbre linéaire, les statistiques, le traitement du signal ou encore le traitement d'images..

Les objets Python de la bibliothèque de vision par ordinateur OpenCV utilisent des tableaux NumPy pour stocker et traiter les données. Comme les images sont simplement représentées sous forme de tableaux tridimensionnels, l'indexation, le découpage ou le masquage avec d'autres tableaux sont des moyens très efficaces d'accéder à des pixels spécifiques d'une image.

Pour pouvoir utiliser la bibliothèque Numpy, il faut bien sûr qu'elle soit installée au préalable!

Importation de la bibliothèque NumPy

On supposera pour la suite du cours que l'importation de NumPy a été faite avec l'alias np en utilisant donc la syntaxe suivante:

```
import numpy as np
```

Il faudra donc faire précéder les fonctions de numpy de 'np.' pour obtenir quelque-chose de la forme « np.xxx() »

Un nouveau type de variable avec Numpy: le type array

Numpy ajoute le type array qui est similaire à une liste (list) avec la condition supplémentaire que tous les éléments sont du même type.

Pour nous, ce sera donc un tableau d'entiers, de flottants voire de booléens.

La plupart des tableaux sont uni ou bi-dimensionnels mais il peut exister des tableaux multidimensionnels, par exemple, de dimension 3. C'est un peu comme si on avait une matrice dont les éléments sont eux-mêmes des matrices bi-dimensionnelles!

Créer un tableau (array) avec Numpy

Une première méthode pour créer un tableau consiste à convertir une liste en un tableau via la commande array. Le deuxième argument est optionnel et spécifie le type des éléments du tableau.

```
>>> a = np.array([1, 4, 5, 8], float )
>>> a                                   # renvoie le type de la variable et son contenu
array([ 1., 4., 5., 8.])
>>>type (a)                            # renvoie le type de la variable
<type 'numpy.ndarray'>
```

Un autre type de variable avec Numpy: le type mat (comme matrix ou matrice)

Numpy propose aussi le type mat comme matrice, qui est exclusivement un tableau bi-dimensionnel.
Il permet de saisir les matrices et de faire le produit matriciel avec le symbole * ce qui est + facile
Attention cependant, les fonctions telles que **ones**, **eye** retournent un objet de type array et pas mat.

```
>>> a=np.mat('[1 2 4 ; 3 4 5.]')      # les ; séparent les lignes, un espace sépare les éléments
>>> b=np.mat('[2. ; 4 ; 6]')
>>> print a
[[ 1. 2. 4.]
 [ 3. 4. 5.]]                        # on retrouve bien une présentation de type matrice
>>> print b
[[ 2.]
 [ 4.]
 [ 6.]]
>>> print a*b
[[ 34.]
 [ 52.]]                             # on retrouve bien le produit matriciel ligne/colonne
```

Tout objet array est convertible en type mat et réciproquement (sous la condition que le tableau (array) soit uni ou bi-dimensionnel)

```
>>> a=np.array([1, 2, 4])
>>> np.mat(a)
matrix([[1, 2, 4]])
```

Numpy propose le redimensionnement d'un tableau avec la fonction **reshape()**
Il faut tout de même respecter une condition : le nombre d'éléments doit être le même !

```
>>> a=np.arange(16)                              # arange(16) crée une liste de 0 à 15
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> a.reshape(4,4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])                        # on a regroupé les 16 él. En 4 lignes de 4 él.
>>> a.reshape(2,8)
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])     # ici c'est en 2 lignes de 8 éléments
```

La fonction **numpy.size()** renvoie le nombre d'éléments du tableau.

```
a = np.array([2,5,6,8])
np.size(a)
4
b = np.array([[1, 2, 3],[4, 5, 6]])
np.size(b)
6
```

La fonction **numpy.shape()** (forme, en anglais) renvoie la taille du tableau.

```
a = np.array([2,5,6,8])
np.shape(a)
(4,)                                    # écriture complète: (4,1), le 1 est sous-entendu
b = np.array([[1, 2, 3],[4, 5, 6]])
np.shape(b)
(2, 3)                                  # ici on a bien deux lignes de 3 éléments (3 colonnes)
```

Création d'un tableau de type matrice unité (matrice identité) : numpy.eye()

eye(n) renvoie un **tableau** 2D carré de taille n x n, avec des uns sur la diagonale et des zéros partout ailleurs.

```
np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Création d'un tableaux de 0 et d'une matrice nulle : numpy.zeros()

zeros(n,m) renvoie un **tableau** de n x m zéros que l'on peut ensuite transformer en matrice.

```
np.zeros(3)
array([ 0.,  0.,  0.])
# zeros((m,n)) renvoie tableau 2D de taille m x n, c'est-à-dire de shape (m,n).
np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Création d'un tableaux de 1 : numpy.ones()

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

Création d'une matrice diagonale : numpy.diag(v)

np.diag(v) renvoie une matrice diagonale dont la diagonale est le vecteur v et tous les autres éléments sont à 0

Création d'une matrice à diagonale décalée : numpy.diag(v,k)

np.diag(v,k) renvoie une matrice dont la 'diagonale' décalée de k est le vecteur v (k est un entier relatif)

Création d'une matrice contenant des valeurs aléatoires : numpy.random.rand()

np.random.rand(n) renvoie un vecteur de taille n ,
np.random.rand(n,p) renvoie une matrice de taille n x p à coefficients aléatoires uniformes sur [0,1]

Opérations sur les tableaux (matrices)

Addition de 2 tableaux/matrices

L'opérateur + additionne terme à terme deux tableaux/matrices de même dimension.

Addition d'un nombre identique à chaque élément

La commande 2.+a renvoie le tableau/matrice dont tous les éléments sont ceux de a plus 2.

Multiplication d'un tableau/matrice par un scalaire

Multplier une tableau/matrice par un scalaire se fait selon le même principe : la commande 2*a renvoie le tableau/matrice de même dimension dont tous les éléments ont été multipliés par 2.

Opérations sur les tableaux (matrices)

Multiplication terme à terme de 2 tableaux (produit d'Hadamard ou de Schur)

L'opérateur * ne fait que multiplier terme à terme deux tableaux de même dimension.
Pour obtenir le produit matriciel (le vrai) la commande à utiliser est np.dot(array1, array2)

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

b = np.array([[2, 1, 3],
              [3, 2, 1]])
a*b

array([[ 2,  2,  9],
       [12, 10,  6]])
```

Mais attention, le même opérateur * appliqué à 2 matrices compatibles donnera le veritable produit matriciel ligne/colonne! (voir diapo suivante)

Opérations sur les tableaux (matrices)

Multiplication matricielle Ligne/colonne ou produit matriciel

Pour multiplier **2 tableaux** selon le véritable produit matriciel ligne/colonne, il faut utiliser la méthode .dot ou a @ b qui est équivalent à np.dot(a, b) depuis Python 3.5 et NumPy 1.10 :

```
a = np.array([[1, 2, 3],[4, 5, 6]])
b = np.array([[4],[2],[1]])
np.dot(a,b)                              # ou a @ b
array([[11],
       [32]])                            # on retrouve bien le résultat du produit matriciel
```

Par contre, l'opérateur * appliqué à deux matrices pour lesquelles la multiplication est possible donne bien directement le produit matriciel correct.

```
a=np.mat('[1 2 3 ; 4 5 6]')      # les ; séparent les lignes, un espace sépare les éléments
b=np.mat('[4 ; 2 ; 1]')
print a
[[ 1 2 3]
 [ 4 5 6]]                       # on retrouve bien une présentation de type matrice
print b
[[4]
 [2]
 [1]]
print a*b
[[11]
 [32]]                           # on retrouve bien le produit matriciel ligne/colonne
```

Transposée d'une matrice: .T

```
a = np.array([[1, 2, 3],
              [4, 5, 6]])
a.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Complexe conjugué : numpy.conj()

```
u = np.array([[  2j, 4+3j],
              [2+5j, 5   ],
              [   3, 6+2j]])
np.conj(u)
array([[ 0.-2.j,  4.-3.j],
       [ 2.-5.j,  5.+0.j],
       [ 3.+0.j,  6.-2.j]])
```

Transposé complexe conjugué : numpy.conj().T

```
np.conj(u).T
array([[ 0.-2.j,  2.-5.j,  3.+0.j],
       [ 4.-3.j,  5.+0.j,  6.-2.j]])
```

Calcul du déterminant d'une matrice : numpy.linalg.det()

La méthode numpy.linalg.det() permet de calculer le déterminant d'un tableau au sens du déterminant d'une matrice.

```
from numpy.linalg import det
a = np.array([[1, 2],
              [3, 4]])
det(a)
-2.0
```

Inverse d'une matrice carrée : numpy.linalg.inv()

numpy.linalg.inv() permet d'inverser un tableau au sens de l'inversion de matrice

```
from numpy.linalg import inv
a = np.array([[1, 3, 3],
              [1, 4, 3],
              [1, 3, 4]])
inv(a)
array([[ 7., -3., -3.],
       [-1.,  1.,  0.],
       [-1.,  0.,  1.]])
```

La méthode getI permet d'inverser une matrice avec comme résultat une matrice.

```
m = np.matrix('[1, 2; 3, 4]'); m
matrix([[1, 2],
        [3, 4]])
>>> m.getI()
matrix([[-2. , 1.  ],
        [ 1.5, -0.5]])
>>> m.getI() * m
matrix([[ 1., 0.], # vérification
        [ 0., 1.]])
```

Résolution d'un système d'équations linéaires : numpy.linalg.solve()

Pour résoudre le système d'équations linéaires du type 3 * x + y = 9 et x + 2 * y = 8 , on peut utiliser la méthode `np.linalg.solve(a,b)` où a est une matrice carrée et b un vecteur ou une matrice (avec condition de compatibilité)

```
a = np.array([[3,1], [1,2]])                    # matrice des coefficients en x et en y des 2
équations
b = np.array([9,8])                             # matrice des termes indépendants
x = np.linalg.solve(a, b)                       # x est la matrice qui contient les valeurs de x
                                                # x et y qui sont les solutions
array([ 2.,  3.])
```

Pour vérifier que la solution est correcte :

```
np.allclose(np.dot(a, x), b)
True                                            # la fonction allclose renvoie vrai si la
                                        # solution est correcte
```

valeurs propres et vecteurs propres d’une matrice:

calcul des valeurs propres et vecteurs propres : `np.linalg.eig(a)`

Sources:

http://math.mad.free.fr/depot/numpy/base.html

https://numpy.org/doc/stable/reference/generated/numpy.matrix.html

http://math.mad.free.fr/depot/numpy/calmat.html

Exercices sur les matrices en Python

Enoncé exercice 1.3 :

Premiers avec NumPy: utilisation des arrays pour manipuler des matrices
Ecrire un pgm en Python utilisant Numpy qui :

- Affiche les Arrays

A= ([4.1, 2.0, 0],
[4.6, 1, 6],
[2, 8, 3])

B= ([1, 1, 0],
[1.0, 1, 1],
[2, 2, 2])

- Calcule A+B et l'affiche (A quoi correspond cette addition?)

Solution possible Exercice 1.3 : `math-ex1-3.py` (en téléchargement sur Teams)

Enoncé exercice 1.4 :

- Création et affichage de la 'matrice' 3x3 suivante avec des éléments de type float:
  M=([4.1, 2.0, 0],
  [4.6, 1, 6],
  [2, 8, 3])

- Affichage de l'élément m_{23} de cette ‘matrice’
- Affichage de la 3ème ligne (quelle que soit la matrice introduite)
- Affichage de la première colonne (quelle que soit la matrice introduite)

- Création et Affichage d'une 'matrice' 3x3 de type entier ne contenant que des 1

- Création et Affichage d'une 'matrice' unité diagonale 5x5 éléments type float

- Réarrangement et affichage de la liste linéaire suivante en ‘matrice’ 3x2:

- A = ([2, 4, 6, 12, 24, 36])

Solution possible Exercice 1.4 : `math-ex1-4.py` (en téléchargement sur Teams)

Enoncé exercice 1.5 :

Ecrire un pgm en Python utilisant Numpy qui :

- Affiche les Arrays suivantes:

```
A= ([4.1, 2.0, 0],
    [4.6, 1,   6],
    [2,   8,   3])
B= ([1,   1,   0],
    [1.0, 1,   1],
    [2,   2,   2])
C= ([1,   2],
    [0,   1],
    [3,   1],)
```

- Calcule A x B et l'affiche (produit d'Hadamard)
- Calcule A@B et l'affiche (produit matriciel)
- Calcule le produit matriciel de A par B (mais sans utiliser l'opérateur @, en créant votre propre algorithme), de A par C et de C par A en indiquant si ce n'est pas possible pourquoi.

Solution possible Exercice 1.5 : `math-ex1-5.py` (en téléchargement sur Teams)

Enoncé exercice 1.6 :

Ecrire un pgm en Python utilisant Numpy qui :

- Propose l'introduction des éléments de 2 matrices quelconques (en demandant avant l'introduction leur dimensions)
- Calcule et affiche le produit matriciel de A par B et de B par A en indiquant si ce n'est pas possible pourquoi.
- Calcule la somme de A+B en indiquant si ce n'est pas possible pourquoi.
- Calcule et affiche la transposée de A en indiquant son nombre de lignes et de colonnes

Solution possible Exercice 1.6 : `math-ex1-6.py` (en téléchargement sur Teams)