INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

NumPy

Introduction et référence pratique

UNIVERSITÉ DE LYON

Qu'est-ce que NumPy ?

- NumPy (en anglais Numerical Python) est une bibliothèque Python open source utilisée dans tous les domaines de la science et de l'ingénierie. C'est la norme universelle pour travailler avec des données numériques en Python. NumPy est largement utilisée dans SciPy, Matplotlib, Pandas, scikit-learn, scikit-image et la plupart des "packages" de calcul scientifique en Python.

- Pour accéder à NumPy, on l'importe souvent avec l'alias np:

```
1 import numpy as np
2 # désormais toutes les définitions dans numpy sont accessibles avec la qualification np
3 # ex. np.pi pour la valeur approchée de la constante pi
```

Principe de base de Numpy : type nd-array

- Objet central de Numpy : tableau multidimensionnel **`nd-array`** (n-dimensional array en anglais)
  - Similaire à la `list` python
  - Optimisé pour les calculs
  - Contient **uniquement des éléments du même type**
- NumPy offre une énorme bibliothèque de fonctions mathématiques et d'algorithmes pour utiliser des ndarray de manière efficace.

Principe de base de Numpy : Vectorisation

- **Pas de boucle explicite** pour les calculs terme à terme
- **Exemple** :
  somme terme à terme de *u = <1, 2, 3>*, et *v = <10, 20, 30>* dans *w = u + v = <11, 22, 33>*

Avec Numpy (code vectorisé)

```
1 u = np.array([1, 2, 3])
2 v = np.array([10, 20, 30])
3 w = u + v # somme directe,
4 print(w)
```

```
[11 22 33]
```

Avec les listes python

```
1 u = [1, 2, 3]
2 v = [10, 20, 30]
3 w = []
4 for i in range(len(u)):
5     w.append(u[i] + v[i])
6 print(w)
```

```
[11, 22, 33]
```

- Le code vectorisé présente de nombreux avantages :
  - plus concis, plus facile à lire et ressemble plus à une notation mathématique
  - un code plus efficace
- **Important : éviter les boucles explicites avec des tableaux NumPy chaque fois que cela est possible** (utiliser les opérateurs et fonctions vectorisés)**.**

Sur l'usage de la documentation

- Ces diapos comme les supports de TD n'ont pas de vocation d'être exhaustifs, **d'où l'intérêt de vous servir de la documentation quand le besoin se présente !**
- Les bibliothèques de Python s'auto-documentent avec les docstrings. Pour y accéder:

1. Ajouter un point d'interrogation à la fin de l'identifiant ou utiliser la fonction help.

```
np.array?
```

```
help(np.array)
```

2. **Dans Jupyter Notebook seulement** : mettre votre curseur dans l'identifiant et saisir la combinaison de touches Maj+Tab. Une fenêtre contextuelle apparaît.
3. NumPy offre une fonction spécifique pour chercher par mot clé dans sa documentation:

```
1 np.lookfor("create array")
```

```
Search results for 'create array'
--------------------------------
numpy.array
    Create an array.
numpy.memmap
    Create a memory-map to an array stored in a *binary* file on disk.
numpy.diagflat
    Create a two-dimensional array with the flattened input as a diagonal.
```

Création des tableaux : à partir d'une liste python

- On peut créer un nd-array à partir d'une liste python avec la fonction np.array()
  - À partir d'une liste à 1 dimension

```
1 v = np.array([42, 182, 666])
2 print(v)
```

```
[ 42 182 666]
```

  - À partir d'une liste à 2 dimensions (ici une matrice de dimensions 2x3)

```
1 mat = np.array([[78, 69, 36], [39, 25, 9]])
2 print(mat)
```

```
[[78 69 36]
 [39 25  9]]
```

Création des tableaux : à partir d'un intervalle

Uniquement pour des tableaux 1D !

1. np.arange() : l'équivalent de la fonction range en Python. Prend en paramètre les bornes (borne supérieure non incluse) et le pas (facultatif).

```
1  u = np.arange(10) # par défaut début = 0, pas = 1
2  print(f"u = {u}")
3
4  v = np.arange(2, 10, 2) # pas de 2
5  print(f"v = {v}")
```

```
u = [0 1 2 3 4 5 6 7 8 9]
v = [2 4 6 8]
```

```
1  v = np.arange(11, 0, -2)
2  print(v)
```

```
[11  9  7  5  3  1]
```

2. np.linspace() : permet de spécifier le nombre d'éléments au lieu du pas (3ème paramètre). Attention, les bornes sont incluses.

```
1  w = np.linspace(0, 1, 6)   # début, fin comprise, nombre de valeurs
2  print(w)
```

```
[0.  0.2 0.4 0.6 0.8 1. ]
```

Création des tableaux : tableaux courants

- np.zeros() pour créer un tableau rempli de 0
- np.ones() pour créer un tableau rempli de 1
- np.eye() pour créer une matrice identité

Voir la liste complète dans la doc

```python
a = np.ones(4) # Une seule valeur donnée => tableau 1D
print(f"a = {a}")

b = np.zeros((2, 2)) # les dimensions sont données entre parenthèses
print(f"b = \n {b}") # \n signifie retour à la ligne

identite = np.eye(3) # il s'agit d'une matrice carrée, spécifier une dimension suffit
print(f"identite = \b {identite}")
```

```
a = [1. 1. 1. 1.]
b =
 [[0. 0.]
 [0. 0.]]
identite = [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Les propriétés des tableaux

- le nombre de dimensions a.ndim,
- la taille de chaque dimension a.shape (un tuple),
- le nombre total d'éléments a.size
- le type d'élément a.dtype.

```python
a = np.arange(10)
print(a)
print(a.ndim) # nombre de dimensions
print(a.shape) # dimensions
print(a.size) # nombre d'éléments
print(a.dtype) # type d'élément
```

```
[0 1 2 3 4 5 6 7 8 9]
1
(10,)
10
int64
```

```python
mat = np.random.random((2,5))
print(mat)
print(mat.ndim)
print(mat.shape)
print(mat.size)
print(mat.dtype)
```

```
[[0.81292496 0.01913796 0.41595166 0.46051881 0.09465438]
 [0.86220585 0.42762071 0.10714434 0.57265686 0.64753109]]
2
(2, 5)
10
float64
```

Remarques sur le type d'éléments

- NumPy déduit le type d'élément le mieux adapté lors de la création d'un tableau ou selon le résultat d'un calcul

```
1 u = np.array([True, False])
2 print(u.dtype)
3 v = np.array([1, 2])
4 w = np.array([3.4, 5.6])
5 x = v + w
6 print(x.dtype)
```

```
bool
float64
```

Opérations sur les tableaux

1. Opérations arithmétiques

- Les opérations arithmétiques habituelles +, -, *, **, /, % etc. s'appliquent terme à terme

```
1 u = np.array([20, 30, 40, 50])
2 v = np.arange(4)
3 print(f"u - v = {u - v}")
4 print(f"u * v = {u * v}")
5 print(f"u ** v = {u ** v}")
6 print(f"v / u = {v / u}")
```

```
u - v = [20 29 38 47]
u * v = [  0  30  80 150]
u ** v = [     1     30   1600 125000]
v / u = [0.    0.033 0.05  0.06 ]
```

```
1 a = np.array([[1, 1], [0, 1]])
2 b = np.array([[2, 0], [3, 4]])
3
4 print(f"a + b =\n {a + b}")
5 print(f"a - b =\n {a - b}")
```

```
a + b =
 [[3 1]
 [3 5]]
a - b =
 [[-1  1]
 [-3 -3]]
```

- Attention : l'opérateur de multiplication * désigne un produit terme à terme.

  Utilisez l'opérateur @ pour un produit matriciel.

```
1 a = np.array([[1, 1], [0, 1]])
2 b = np.array([[2, 0], [3, 4]])
3
4 print(f"a * b =\n {a * b}")
5 print(f"a @ b =\n {a @ b}")
```

```
a * b =
 [[2 0]
 [0 4]]
a @ b =
 [[5 4]
 [3 4]]
```

Opérations sur les tableaux

2. La diffusion

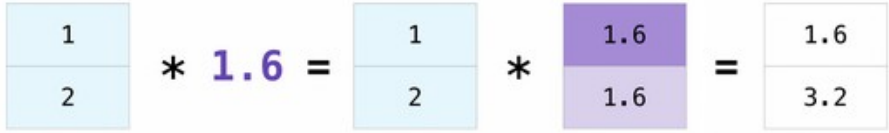
Exemple : multiplication d'un vecteur par un scalaire

```
1 v = np.array([1, 2])
2 u = 1.6 * v
3 print(u)
```

```
[1.6 3.2]
```

Dans les opérations impliquant des nd-array de **dimensions différentes** (ou des scalaires), NumPy **modifie les dimensions** des opérandes sous certaines conditions afin que l'opération puisse s'appliquer terme à terme.

Dans l'exemple précédent, on dit que NumPy **diffuse** la valeur scalaire 1.6 dans le vecteur v.



La diffusion permet d'écrire des expressions telles que :

```
1 v = np.array([20, 30, 40, 50])
2 print(v < 35) # comparer chacun des éléments de a avec la valeur 35
3 print(v ** 3) # prendre le cube de chacun des éléments de a
```

```
[ True  True False False]
[  8000  27000  64000 125000]
```

Opérations sur les tableaux

3. Les fonctions universelles

- NumPy fournit les fonctions mathématiques courantes telles que *sin()*, *tanh()*, *exp()*, *log()* etc.
  Ces fonctions opèrent élément par élément sur un tableau, produisant un nouveau tableau.

```
1 x = np.arange(4)
2 print(np.exp(x))
3 print(np.sqrt(x))
4 print(10 * np.sin(0.25 * np.pi * x))
```

```
[ 1.     2.718  7.389 20.086]
[0.    1.    1.414 1.732]
[ 0.     7.071 10.     7.071]
```

Opérations sur les tableaux
4. L'indexation

- L'indexation des tableaux NumPy se fait **de la même façon que les séquences Python.**

```python
v = np.array([1, 2, 3])
print(f"v[0] = {v[0]}")
# print(v[3]) génère une erreur IndexError
print(f"v[-1] = {v[-1]}") # indices négatifs rebouclent
print(f"v[-2] = {v[-2]}")
v[1] = 200
print(f"v = {v}")
```

```
v[0] = 1
v[-1] = 3
v[-2] = 2
v = [  1 200   3]
```

- Pour un tableau 2D, il faut spécifier les deux indices **séparés par une virgule** : `mat[i,j]`

```python
mat = np.arange(1, 10).reshape((3, 3))
print(f"mat =\n {mat}")
print(f"mat[1, 2] = {mat[1, 2]}")
mat[2, 2] = 666
print(f"mat =\n {mat}")
```

```
mat =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
mat[1, 2] = 6
mat =
 [[  1   2   3]
 [  4   5   6]
 [  7   8 666]]
```

Opérations sur les tableaux

5. L'indexation à l'aide des tableaux

- On utilise peu l'indexation avec un indice en Numpy (opérations vectorisées)
- Plus utile : **indexer les tableaux à l'aide d'autres tableaux**
- **Exemple** : on a un tableau v 1D de 10 entiers aléatoires dans l'intervalle [-5, 5[.

```
1 v = np.random.randint(-5, 5, 10)
2 print(v)
```

```
[ 1 -1 -2 -1 -4  3  1  3  2 -4]
```

On peut mettre tous les éléments négatifs à 0 **sans faire de boucle.**

Opérations sur les tableaux

5. L'indexation à l'aide des tableaux (suite)

1) Créer un tableau booléen `negatifs` dont les éléments indiquent si les éléments de v respectent la condition (< 0)

```
1 negatifs = v < 0
2 print(negatifs)
```

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

On peut i**ndexer le tableau `v` avec le tableau `negatifs`** pour ne récupérer que les éléments correspondant aux valeurs True :

```
1 print(v[negatifs])
```

```
[-1 -2 -1 -4 -4]
```

2) Mettre à 0 ces éléments

```
1 v[negatifs] = 0
2 print(v)
```

```
[1 0 0 0 0 3 1 3 2 0]
```

Note : On peut utiliser le tableau de booléen directement, sans passer par une variable intermédiaire

```
1 v = np.random.randint(-5, 5, 10)
2 print(v)
3 v[v < 0] = 0
4 print(v)
```

```
[ 0  2  4 -4  0  1  0 -4  1 -5]
[0 2 4 0 0 1 0 0 1 0]
```

Opérations sur les tableaux

6. Le tranchage (slicing) : en 1D

- Permet de récupérer **une partie** d'un tableau v :
  - Donner les indices de début et de fin séparés par deux-points : **`v[deb:fin]`**
  - La borne de fin est exclue
  - On peut donner un pas facultatif : **`v[deb:fin:pas]`**

```
1 v = np.arange(10, 100, 10)
2 print(f"v = {v}")
3 print(f"v[1:3] = {v[1:3]}")
4 print(f"v[2:] = {v[2:]}") # absence d'indice de fin => la tranche s'étend jusqu'à la fin de la dimension
5 print(f"v[:4] = {v[:4]}") # absence d'indice de début => la tranche commence au début de la dimension
6 print(f"v[-4:] = {v[-4:]}") # indiced négatifs possibles ! ça reboucle comme pour les indices normales
7 print(f"v[1:9:2] = {v[1:9:2]}") # avec un pas de 2
```

```
v = [10 20 30 40 50 60 70 80 90]
v[1:3] = [20 30]
v[2:] = [30 40 50 60 70 80 90]
v[:4] = [10 20 30 40]
v[-4:] = [60 70 80 90]
v[1:9:2] = [20 40 60 80]
```

Opérations sur les tableaux

6. Le tranchage : en 1D (suite)

- On peut modifier tous les éléments choisis

```
1 v = np.array([10, 20, 30, 40, 50])
2 print(v)
3 v[1:3] =  6
4 print(v)
```

```
[10 20 30 40 50]
[10  6  6 40 50]
```

- Attention, **l'indexation avec une tranche ne crée pas une copie mais une référence aux éléments du tableau initial**, les modifications impactent le tableau initial :

```
1 v = np.array([10, 20, 30, 40, 50])
2 print(v)
3 u = v[1:3]
4 u[0] = 200
5 print(v)
6 print(u)
```

```
[10 20 30 40 50]
[ 10 200  30  40  50]
[200  30]
```

Petit exercice : Que fait le code suivant ?

```
1 v = np.array([1, 2, 3, 4, 5])
2 print(v[::-1])
```

Opérations sur les tableaux

6. Le tranchage : en 2D

- Pour un tableau 2D, on donne les indices de tranche pour chaque dimension **`mat[d1:f1, d2:f2]`**.

```
mat = np.arange(1, 10).reshape((3, 3))
print(mat)
```

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

```
print(mat[0:2, 0:2]) # récupérer la matrice 2x2 du coin haut gauche
print(mat[:2, :2])   # équivalent sans spécifier le début
print(mat[:2, 1:3])  # récupérer la matrice 2x2 du coin haut droite
print(mat[:, 1:3])   # récupérer les 2 dernières colonnes
print(mat[:2, :])    # récupérer les 2 premières lignes
```

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

- On peut spécifier une tranche pour une dimension et un indice pour l'autre :

```
print(mat[:, 1]) # récupère la colonne du milieu
print(mat[2, :]) # récupère la dernière ligne
# récupère les 2 derniers éléments de la dernière colonne
print(mat[1:3, 2])
```

```
[2 5 8]
[7 8 9]
[6 9]
```

Opérations sur les tableaux

7. Les réductions

- Ce sont des **opérations vectorisées qui réduisent le nombre de dimensions** du tableau. On trouve toutes les opérations renvoyant une valeur à partir d'une séquence : somme, minimum/maximum, moyenne…

- En 1D

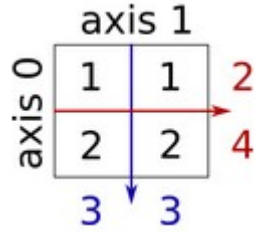
```
x = np.random.randn(200) # randn = normal
print(f"minimum = {np.min(x)}")
print(f"maximum = {np.max(x)}")
print(f"somme = {np.sum(x)}")
print(f"moyenne = {np.mean(x)}")
print(f"écart-type = {np.std(x)}")
print(f"médian = {np.median(x)}")
```

```
minimum = -2.744392267357242
maximum = 2.4853836757180243
somme = 37.26292015326598
moyenne = 0. 1863146007663299
écart-type = 0.9087914634560398
médian = 0.1435588114214763
```

- En 2D

```
x = np.array([[1, 1], [2, 2]])
print(np.sum(x))
```

```
6
```



On peut réduire à une dimension pour calculer la somme **par colonne ou par ligne** : on donne la dimension dans laquelle appliquer l'opération

```
print(np.sum(x,axis=0)) # somme par colonnes
print(np.sum(x,axis=1)) # somme par lignes
```

```
[3 3]
[2 4]
```

Opérations sur les tableaux

*7. Les réductions : np.arg**

Il existe des versions de `np.min()` et `np.max()` qui **renvoient l'indice** de l'élément minimum ou maximum au lieu de renvoyer la valeur : `argmin()` et `argmax()`.

```
x = np.random.randint(-5, 5, 10)
print(x)
print(f"indice du minimum = {x.argmin()}")
print(f"indice du maximum = {x.argmax()}")
```

```
[ 0 4 4 2 -4 -5 1 -2 -2 -1]
indice du minimum = 5
indice du maximum = 1
```

De même, il existe une version de np.sort() qui renvoie les indices du tableau trié (`argsort()`) :

```
x = np.random.randint(-5, 5, 10)
print(x)
idx = np.argsort(x)
print(idx)
print(x[idx])
```

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

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Pour aller plus loin

UNIVERSITÉ DE LYON

Références

- NumPy quickstart guide
- NumPy: the absolute basics for beginners
- Matplotlib tutorials
- Matplotlib gallery
- Scipy Lecture Notes