

ENG-209

Data science pour ingénieurs avec Python

*Cours 3:
Modules, numpy*

Jean-Philippe Pellet

30 septembre 2024

Organisation

- 14.10.2024: **Midterm**, 10h15, INF3, 120 minutes (160 si temps suppl.)
 - Partie QCM (questions plutôt théoriques)
 - Partie programmation (compléter/corriger/écrire du code dans un Jupyter notebook, à soumettre par Moodle)
 - Travail sur machine virtuelle uniquement depuis poste INF3
 - Matériel autorisé: résumé **personnel** au format d'une feuille A4 recto-verso
 - Examen blanc en ligne

Cours 3

Modules
numpy

Cours 3

Modules
numpy

Modules et imports

```
import math
print(math.cos(math.pi))
```

```
import math as m
print(m.cos(m.pi))
```

```
from math import *
print(cos(pi))
```

```
from math import pi, cos as cosine
print(cosine(pi))
```

```
from typing import List
```

Les bibliothèques Python sont organisées par module. Seul un petit nombre d'éléments sont préimportés. Le reste doit l'être explicitement avec import.

*«J'utilise le module existant qui s'appelle math»
Tout ce qui est dans math est utilisable avec préfixe*

Même chose, mais en renommant le module pour le fichier en cours avec un import ... as

*Pour un usage sans préfixe, il faut un from ... import.
On peut tout importer avec **

... mais on a meilleur temps de n'importer que ce qu'on utilise. On peut aussi renommer des membres importés

C'est aussi valable pour les types (qui aussi sont des valeurs presque «normales» en Python)

Déclarer un module

Dans `mytools.py`:

```
from typing import List
```

```
def double(values: List[int]) -> List[int]:  
    return [2 * x for x in values]
```

```
def make_string(values: List[int], separator: str = ", ") -> str:  
    return separator.join([str(x) for x in values])
```

Dans un autre fichier du même dossier:

```
from mytools import *  
print(double([1, 2, 3]))  
print(make_string([1, 2, 3], separator=" -> "))
```

ou:

```
from mytools import double as dbl  
print(dbl([1, 2, 3]))
```

Chaque fichier .py est importable comme module.

Dans un fichier `mytools.py`, des fonctions, classes, variables et autres déclarations peuvent résider «normalement»

(Attention, le code au niveau zéro est directement exécuté)

Dans un autre fichier, on peut réutiliser ce que `mytools.py` fournit avec des imports normaux

Renommer reste possible

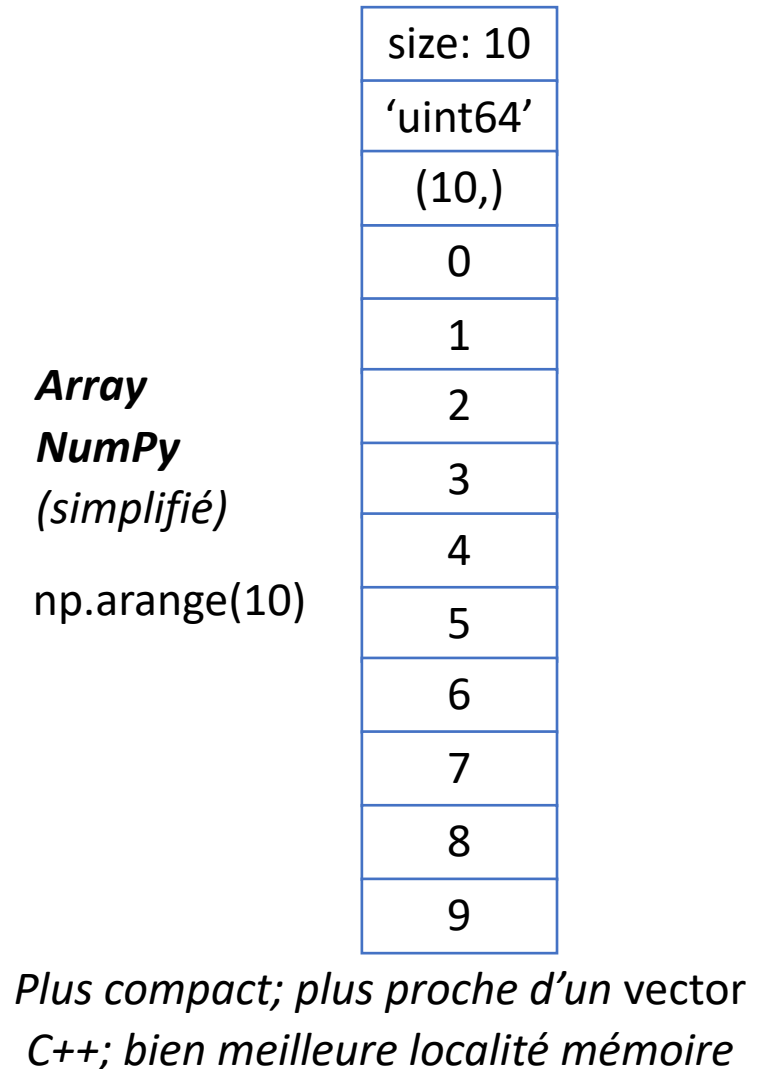
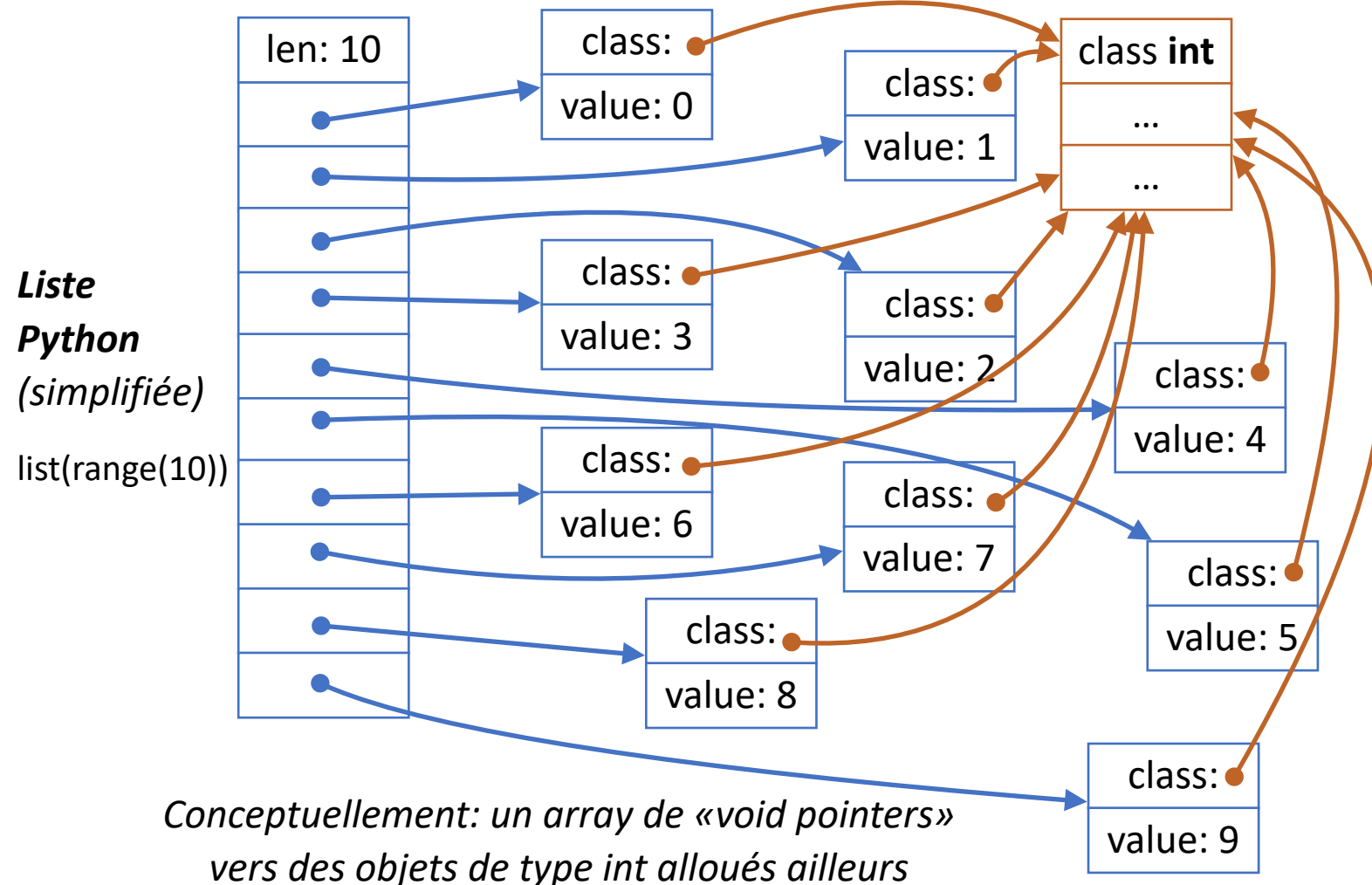
Cours 3

Modules
numpy

numpy — Numerical Python

- **Prix des langages très flexibles et dynamiques: la performance**
 - ✦ Exemple des listes hétérogènes
 - ✦ Moins bien pour: listes homogènes auxquelles on applique la même opération
- **NumPy: implémentation native (en C) d'arrays homogènes multidimensionnels**
 - ✦ Types possibles: uint8, int8, ..., [u]int64; float16, float32, float64; complex64, complex128
 - ✦ Dim. 0: scalaire; dim. 1: vecteur; dim. 2: matrice; 3+: tenseur...
 - ✦ Les opérations numériques à grande échelle sont rapides
 - ✦ Explique une partie du succès de Python dans ce qui tourne autour de Data Science!
 - * Utilisé par nombre d'autres bibliothèques de traitement numérique
- **Documentation exhaustive:** <https://numpy.org/doc/stable/>

Layout en mémoire: *list* vs. *np.array*



Plusieurs dimensions: *row-major*, *column-major*

1	2	3
4	5	6
7	8	9

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

row-major layout
(à la C)

1	2	3
4	5	6
7	8	9

1	4	7	2	5	8	3	6	9
---	---	---	---	---	---	---	---	---

column-major layout
(à la Fortran)

Les bases des *np.array*s

```
import numpy as np

v = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

print(v) # [ 1  2  3  4  5  6  7  8  9 10 11 12]
print(f"type of each element: {v.dtype}")
print(f"memory size: {v.size * v.itemsize} bytes")
print(f"dimensions: {v.ndim}; shape: {v.shape}")
                    # 1                # (12,)

v = v.reshape((4, 3))

v = v.T
print(f"new shape: {v.shape}") # (3, 4)

v = np.arange(1, 13)
v = np.arange(1, 13, 0.5)

x = np.linspace(0, 2*np.pi, 100)
```

On importe d'habitude numpy en tant que np

Un array numpy peut se créer à partir d'une liste normale. La représentation sera compactée

*dtype donne ici 'int64', le type de stockage utilisé
size est le nombre d'éléments; itemsize la taille en
bytes de chaque élément*

ndim et shape permettent de connaître la forme de

On peut changer la forme d'un array

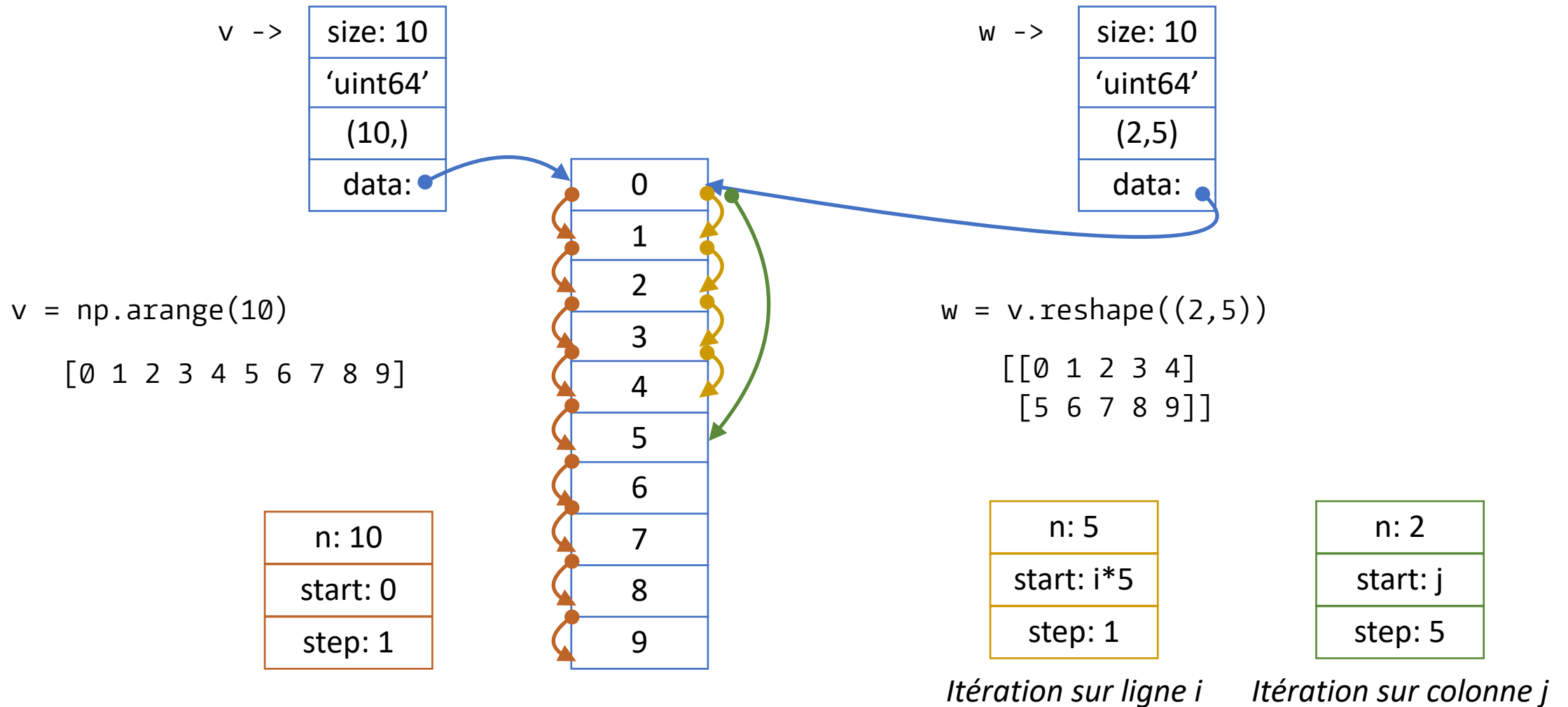
On peut le transposer facilement (matrices)

arange est le pendant de range pour les arrays...

... mais en plus flexible: le step peut ne pas être entier

*Pour avoir une borne supérieure précise et incluse,
on a aussi linspace, avec lequel on choisit le nombre
de points intermédiaires*

Layout en mémoire et *reshape*



Opérations sur des *np.array*s

```
v = np.arange(10)
v = v + 2
v = v ** 2
```

```
v = v / 2
print(f"new type: {v.dtype}") # float64
```

```
def invsqrt(x: float) -> float:
    return 1 / math.sqrt(x)
```

```
invsqrt = np.vectorize(invsqrt)
```

```
v = invsqrt(v)
```

```
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
```

```
import matplotlib.pyplot as plt
plt.plot(x, y)
```

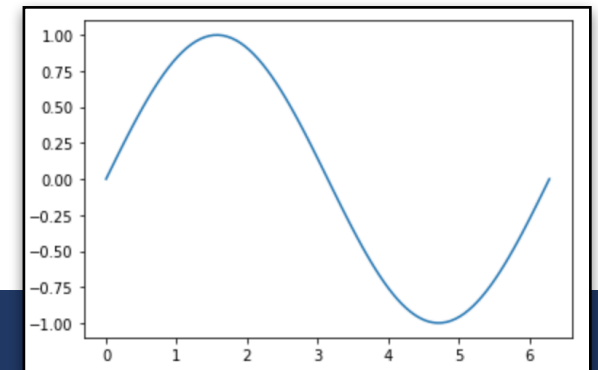
On applique directement les opérations arithmétiques de base: elles se font élément par élément sur tout l'array

Chaque opération retourne un nouvel array, qui peut avoir un autre type selon l'opération (beware linters...)

Pour rester efficace, on évite les boucles qui modifient valeur par valeur. On préfère «vectoriser» une fonction et l'appliquer ensuite. Exemple avec invsqrt ici, qui sera appliqué à chaque élément.

Il y a aussi toute une série de fonctions prédéfinie par numpy, par exemple sin

*C'est facile de grapher des fonctions ainsi!
(Davantage en 2^e partie de cours)*



Fonctions de base en algèbre linéaire

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

```
x = np.linalg.solve(A, b)
```

```
z = A @ x - b
```

```
if not np.allclose(z, 0):  
    print("error")
```

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

```
v1 = np.random.rand(3)  
v2 = np.random.rand(3)
```

```
print(v1 * v2)  
print(np.dot(v1, v2))  
print(np.cross(v1, v2))  
print(np.outer(v1, v2))
```

La fonction solve(A, b) calcule x tel que $Ax = b$

On peut vérifier la solution trouvée. L'opérateur @ dénote la multiplication de matrices

C'est délicat de tester si $z == [0,0,0]$; on utilisera plutôt la fonction np.allclose pour s'abstraire des erreurs d'arrondis en virgule flottante

On peut calculer le déterminant, inverser une matrice...

Plus: <https://numpy.org/doc/stable/reference/routines.linalg.html>

Attentions aux différentes manières de multiplier des non-scalaires: élément par élément, produit scalaire, produit vectoriel, produit extérieur, etc.

Résumé du cours d'aujourd'hui

- Le code de fichiers externes est réutilisable après avoir fait des imports
- **numpy**: Représentation compacte en mémoire
- **Création des arrays à partir de listes ou de fonctions-constructeurs**
 - ✦ `np.arange`, `np.linspace`, `np.ones`, `np.zeros`, `np.random.rand`, etc.
- **Manipulations automatiquement parallèles avec les opérateurs arithmétiques**
 - ✦ Plus toute une série de fonctions prédéfinies
 - ✦ `np.vectorize` pour appliquer sa propre fonction
 - ✦ On essaie d'éviter les boucles Python traditionnelles
- **Indexation et slicing toujours possibles**
 - ✦ On utilisera `M[i, j]` plutôt que `M[i][j]`
 - ✦ Des notations plus avancées pour sélectionner certaines dimensions
- **Au début: difficile de prédire ce qui va être rapide et ce qui va être «lent»**
 - ✦ Est «lente» toute opération qui doit produire des structures intermédiaires qui ne sont pas du numpy

Exercices: Troubleshooting

- **numpy n'est pas installé**
 - ✦ Dans un terminal: `pip3 install numpy`
 - ✦ Puis redémarrez le Jupyter notebook avec le bouton en haut
- **Les erreurs de types n'apparaissent pas dans les notebook**
 - ✦ Installez l'extension VS Code Mypy de Matan Grover
 - ✦ Dans un terminal: `pip3 install mypy`
 - ✦ Ajoutez-y ces lignes au fichier **`.vscode/settings.json`**:

```
"mypy.checkNotebooks": true,  
"mypy.mypyExecutable": "${workspaceFolder}/venv/bin/mypy",  
"mypy.dmypyExecutable": "${workspaceFolder}/venv/bin/dmypy",
```

Autoévaluation — objectifs



- **Je suis capable de/d'...**

- ✦ importer des déclarations d'autres fichiers et modules
- ✦ décrire pourquoi numpy est plus performant qu'une liste Python
- ✦ créer des arrays numpy, en changer la forme
- ✦ appliquer des opérations sur les arrays en évitant les boucles
- ✦ résoudre des problèmes d'algèbre linéaire simples avec numpy
- ✦ faire la différence entre différents types de multiplication
- ✦ consulter la documentation de numpy pour résoudre des tâches plus complexes