# TP 14
# Numpy et Matplotlib

Nous introduisons deux bibliothèques qui sont d'utilité fondamentale dans toutes les sciences des données (traiter des grands tableaux, matrices, avec des millions de données, faire des calculs et des statistiques dessus) et qui contribuent au succès croissant de Python dans ces domaines. De plus, ce TP fait le lien avec les cours de mathématiques à la fois pour les matrices et pour les fonctions.

Pour tout le TP, on peut écrire et exécuter une fois pour toute au début

```
import numpy as np
import matplotlib.pyplot as plt
```

On pourra se servir des documents suivants :
- https://www.concours-agro-veto.fr/sites/default/files/media/2025-10/polypython.pdf : Aide-mémoire Python distribué au concours Agro-Véto.
- https://matplotlib.org/cheatsheets/cheatsheets.pdf : Aide-mémoire de la bibliothèque Matplotlib (un peu compliqué, mais illustre bien toutes les possibilités).

## I Tableaux numpy

La bibliothèque Numpy introduit un nouveau type d'objet qu'on appellera **tableau**. Ceux-ci ressemblent en apparence beaucoup aux listes, mais leur fonctionnement interne est bien différent. Ils sont notamment très efficaces dans le cas où ils contiennent des millions de données, et cela nécessite de se pencher un peu plus sur le fonctionnement interne de l'ordinateur pour bien comprendre.

### I.1 Aperçu sur les tableaux à une dimension

Les tableaux sont des objets du type `ndarray`. On peut les créer avec les fonctions suivantes :

- Conversion depuis une liste : `np.array(L)`

```
>>> X = np.array([1, 3, 5, 7])
>>> print(X)
[1 3 5 7]
```

- Tableau de $n$ zéros : `np.zeros(n)`

```
>>> X = np.zeros(10)
>>> print(X)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

- Tableau d'entiers consécutifs : `np.arange(a, b)` ou `np.arange(n)`, avec la même syntaxe que `range`

```
>>> X = np.arange(10)
>>> print(X)
[0 1 2 3 4 5 6 7 8 9]
```

- Tableau de $n$ valeurs « linéairement espacées » entre deux bornes $a$ et $b$ : `np.linspace(a, b, n)`

```
>>> X = np.linspace(2, 3, 11)
>>> print(X)
[2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0]
```

Comme avec les listes, on peut :
- Demander la longueur du tableau : `len(X)`,
- Accéder directement au $i$-ème élément : `X[i]`, numérotés de 0 à $n-1$ comme d'habitude,
- Trancher le tableau : `X[a:b]` est le tableau constitué des éléments d'indice $i$ tel que $a \leqslant i < b$.

Il existe aussi `X[a:]` (à partir de l'indice $a$) et `X[:b]` (jusqu'à l'indice $b$). Les indices négatifs reviennent à la fin : `X[-1]` est le dernier élément du tableau. Les tranches sont compatibles avec les indices négatifs, ainsi `X[:-1]` est la tableau sans son dernier élément, et `X[1:]` sans le premier.

Quelles sont alors les différences avec les listes Python ?

- Les tableaux Numpy sont représentés en mémoire comme un unique bloc, réservé dès le départ, dans lequel les valeurs sont posées exactement les unes à côté des autres dans des cases consécutives et de même taille. Cela permet à l'ordinateur de calculer directement l'adresse mémoire de chaque élément (chaque case de la mémoire possède une adresse, comme des maisons dans une très très longue rue) et d'y accéder rapidement. Les listes Python ne sont pas aussi efficaces et les éléments sont parfois rangés « en vrac ».
- Les éléments du tableau ont un **type** et doivent tous avoir le même type. Ce type détermine à la fois la place qu'occupe chaque élément en mémoire et donc la taille des cases (par exemple un entier 8 bits occupe 1 octet et peut contenir 256 valeurs ; mais un entier 64 bits occupe 8 octets et peut contenir $2^{64} \approx 1{,}8 \cdot 10^{19}$ valeurs), et comment le nombre est représenté en mémoire (avec 8 bits sans signe on a tous les entiers de 0 à 255, mais avec signe on peut aller de $-128$ à $+127$ ; les nombres à virgule flottante ont une représentation encore bien différente dans un espace de 64 bits).
- À cause de cette structure, les tableaux ont une **taille fixe**, déterminée à leur création. On ne peut pas si facilement faire un `append` ou insérer des éléments en plein milieu. En contrepartie, ils sont **compacts** et **efficaces** : si on choisit le bon type adapté aux données à traiter alors aucune place n'est perdue et les opérations sont effectuées le plus rapidement possible.

On accède au type avec la variable `X.dtype`, et à la place occupée en mémoire (en octets) avec la variable `X.nbytes`, observez :

```
>>> X = np.array([1, 3, 5])
>>> type(X)
<class 'numpy.ndarray'>
# X est un tableau numpy
>>> X.dtype
dtype('int64')
# entiers codés sur 64 bits, soit 8 octets
>>> len(X)
3
# 3 éléments
>>> X.nbytes
24
# total : 3 * 8 = 24 octets octets occupés en mémoire
```

Ou bien :

```
>>> X = np.linspace(0, 1, 20)
>>> print(X)
[0.         0.05263158 0.10526316 0.15789474 0.21052632 0.26315789
 0.31578947 0.36842105 0.42105263 0.47368421 0.52631579 0.57894737
 0.63157895 0.68421053 0.73684211 0.78947368 0.84210526 0.89473684
 0.94736842 1.        ]
>>> X.dtype
dtype('float64')
# nombres à virgule flottante sur 64 bits soit 8 octets
>>> len(X)
20
# 20 valeurs
>>> X.nbytes
160
# total : 8 * 20 octets
```

Ou encore :

```
>>> Z = np.arange(10, dtype="uint8")
>>> print(Z)
[0 1 2 3 4 5 6 7 8 9]
>>> Z.dtype
dtype('uint8')
# entiers non-signés sur 8 bits, soit 1 octet
>>> Z.nbytes
10
# exactement 10 octets au total
```

Ce dernier tableau pose problème si on veut y stocker des valeurs au-delà de 255…

**Exercice 1**
Tester les deux lignes suivantes :

```
>>> X = np.arange(300, dtype="uint8")
>>> print(X)
```

Que se passe-t-il ? Ré-essayer en remplaçant `"uint8"` par `"int8"` puis par `"int64"`.

## I.2 Les opérations vectorielles

Les opérations mathématiques habituelles `+`, `*`, etc ont été reprogrammées pour agir directement sur les tableaux Numpy, en effectuant toutes leurs opérations « case par case ». Observez :

```
>>> X = np.array([1, 3, 5])
>>> Y = np.array([6, -3, 8])
>>> X + Y
array([ 7,  0, 13])
>>> X * Y
array([ 6, -9, 40])
>>> -X
array([-1, -3, -5])
>>> Y**2
array([36,  9, 64])
```

L'intérêt de ces opérations — que l'on sait faire sur les listes avec une banale boucle `for` — est qu'elles s'exécutent beaucoup plus rapidement pour l'ordinateur. De façon très simplifiée, l'instruction est comprise « d'un seul coup » par le processeur (au lieu d'exécuter une boucle `for` et de devoir décoder les instructions à chaque étape) et tire parti au mieux de toutes les optimisations possibles pour calculer rapidement.

On les appelle ici des **opérations vectorielles**, où le mot « vecteur » est synonyme de tableau de nombres (ou en mathématiques : élément de $\mathbb{R}^n$). Ce sont des opérations qui agissent sur des vecteurs et non pas simplement sur des nombres.

La bibliothèque Numpy contient aussi de nombreuses fonctions mathématiques usuelles qui s'appliquent directement à chaque case d'un tableau : `np.exp()`, `np.sin()`, `np.cos()`, `np.arctan()`, `np.log()`, `np.sqrt()` (racine carrée), ainsi que des constantes comme `np.pi` (nombre $\pi$)… Ces opérations vectorielles s'exécutent d'un ordre de grandeur du millier de fois plus rapide que de faire une boucle Python pour les appliquer sur chaque élément.

En pratique, elles seront beaucoup utilisées combinées avec `np.linspace(a, b, n)` pour avoir une représentation d'une fonction sur un intervalle $[a, b]$ « échantillonnée » sur $n$ points. Par exemple pour travailler avec la fonction exponentielle sur $[0, 1]$ en divisant cet intervalle en 100 points :

```
>>> X = np.linspace(0, 1, 100)
>>> print(X)
[0.         0.01010101 0.02020202 0.03030303 0.04040404 0.05050505
 0.06060606 0.07070707 0.08080808 0.09090909 0.1010101  0.11111111
 ...
 0.96969697 0.97979798 0.98989899 1.        ]
>>> Y = np.exp(X)
>>> print(Y)
[1.         1.0101522  1.02040746 1.03076684 1.04123139 1.05180218
 1.06248028 1.07326679 1.0841628  1.09516944 1.10628782 1.11751907
 ...
 2.6371452  2.66391802 2.69096264 2.71828183]
```

**À retenir**

La combinaison de `X = np.linspace(a, b, n)` puis de fonctions vectorielles appliquées à `X` permet d'obtenir une image échantillonnée sur $n$ points d'un intervalle $[a, b]$ et d'une fonction sur cet intervalle.

# II Représentations graphiques

La bibliothèque Matplotlib premet de tracer de très nombreux types de graphiques. La fonction principale que nous utiliserons est `plt.plot(X, Y)` qui prend au moins deux arguments : `X` et `Y` sont tous les deux des listes ou bien des tableaux `numpy`, de même taille ; `X` une liste d'abscisses et `Y` une liste d'ordonnées, pour un ensemble de points qui vont être automatiquement reliés. Ensuite, la fonction `plt.show()` permet d'afficher le graphique.

## II.1 Graphes de fonctions

Pour représenter graphiquement une fonction, on a donc besoin de créer un tableau d'abscisses `X` puis un tableau des ordonnées `Y`, en utilisant toute la méthode de la section précédente. Il faut choisir manuellement le nombre de points d'échantillonnage, par exemple $n = 100$. Voici le modèle de base, pour par exemple $x \mapsto x^2 - 3x + 2$ sur $[-4, 4]$ :

```
# abscisses
X = np.linspace(-4, 4, 100)
# ordonnées
Y = X**2 - 3*X + 2
# tracer et afficher
plt.plot(X, Y)
plt.show()
```

**Exercice 2**

Tracer les graphes des fonctions suivantes.

- $f_1 : x \mapsto x^3 - 5x$ sur $[-4, 4]$
- $f_2 : x \mapsto \sin(x)$ sur $[0, 2\pi]$,
- $f_3 : x \mapsto e^x - 3x + 1$ sur $[-3, 3]$,

Le nombre de points d'échantillonnage doit être choisi pour être suffisamment fin, sinon la courbe n'est pas assez lisse. Mais si on en met trop, le tableau est inutilement trop gros et le programme peut être lourd à charger. Comme ci-dessus, $n = 100$ est un bon compromis pour l'instant.

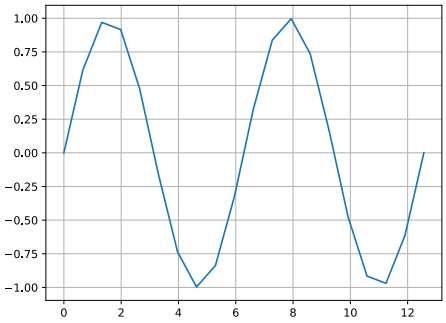


Figure 1. – La fonction sinus sur $[0, 4\pi]$ avec $n = 20$ points d'échantillonnage : c'est trop peu.

La bibliothèque Matplotlib contient de nombreuses options pour configurer le tracé et l'apparence de la fenêtre. Citons seulement :

- `plt.title("titre")` : donne un titre à la fenêtre.
- `plt.xlabel("titre")` : donne un titre à l'axe des $x$.
- `plt.ylabel("titre")` : de même pour l'axe des $y$.
- `plt.xlim(a, b)` : fixe les bornes sur l'axes des $x$ entre $a$ et $b$. Si on ne les fixe pas manuellement, elles sont ajustées automatiquement pour faire rentrer tout le graphe.
- `plt.ylim(a, b)` : de même pour l'axe des $y$.
- `plt.axis("equal")` : rend le repère orthonormé.
- `plt.grid()` : affiche une grille.
- Un troisième argument passé à `plt.plot()` sous la forme d'une chaine de caractères permet à la fois de changer le type de point, le style de trait et la couleur. Par exemple `"+-r"` signifie « points tracés par des symboles plus, reliés par des lignes simples, couleur rouge ». Voir l'aide-mémoire ou la documentation. Ces options, et bien d'autres encore, peuvent être passées à `plt.plot()` sous forme d'arguments optionnels, par exemple `marker="+"`, `linestyle="-"`, `color="red"`. Consulter l'aide-mémoire Matplotlib pour la liste complète.
- Les appels successifs à `plt.plot()` enregistrent les graphiques au fur et à mesure, jusqu'à ce que `plt.show()` les affiche en les superposant. Lorsqu'on trace plusieurs graphiques sur une même figure, il est fort utile de régler manuellement les couleurs et les limites de la fenêtre.

**Exercice 3**
Améliorer le tracé des fonctions précédentes (couleur, style de ligne, titres des fenêtres et des axes).

## II.2 Courbes paramétrées

Dans une **courbe paramétrée**, on trace un point de coordonnées $(x(t), y(t))$ avec un paramètre $t$ qui varie dans un certain intervalle, et donc $x$, $y$ sont tous les deux des fonctions du même $t$. Pour tracer une telle courbe, il faut donc échantillonner un intervalle pour $t$ dans un tableau `T`, puis en déduire deux tableaux `X` et `Y`. Le modèle de base est le suivant qui trace la courbe paramétrée (cercle)

$$\begin{cases} x(t) = \cos(t) \\ y(t) = \sin(t) \end{cases}, \quad t \in [0, 2\pi]$$

```
T = np.linspace(0, 2*np.pi, 100)
X = np.cos(T)
Y = np.sin(T)
plt.plot(X, Y)
plt.show()
```

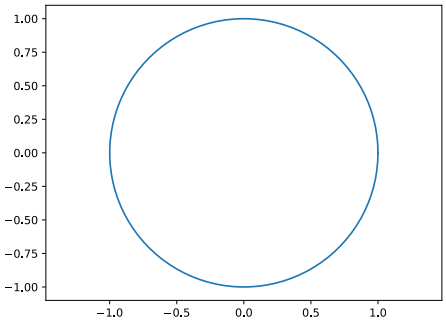


Figure 2. – Un cercle.

**Exercice 4**
Tracer les courbes suivantes. On inclut un lien vers le très beau site mathcurve.com recensant des centaines de courbes mathématiques.

1. Les *courbes de Lissajous* (https://mathcurve.com/courbes2d/lissajous/lissajous.shtml)

$$\begin{cases} x(t) = \cos(pt) \\ y(t) = \sin(qt) \end{cases}, \quad t \in [0, 2\pi]$$

   pour différentes valeurs du couple $(p, q)$, par exemple $(2, 3)$, $(2, 5)$, $(3, 5)$. Attention au nombre de points d'échantillonnages pour que la courbe ait l'air suffisamment lisse !

2. La *cardioïde* (https://www.mathcurve.com/courbes2d/cardioid/cardioid.shtml)

$$\begin{cases} x(t) = (1 + \cos(t))\cos(t) \\ y(t) = (1 + \cos(t))\sin(t) \end{cases}, \quad t \in [-\pi, \pi]$$

3. L'*astroïde* (https://www.mathcurve.com/courbes2d/astroid/astroid.shtml)

$$\begin{cases} x(t) = (\cos(t))^3 \\ y(t) = (\sin(t))^3 \end{cases}, \quad t \in [-\pi, \pi]$$

4. La *strophoïde droite* (https://www.mathcurve.com/courbes2d/strophoid/strophoid.shtml)

$$\begin{cases} x(t) = \frac{1-t^2}{1+t^2} \\ y(t) = t\frac{1-t^2}{1+t^2} \end{cases}, \quad t \in \mathbb{R}$$

   en centrant correctement la figure (axes orthonormées, limites de la fenêtre) sur la partie intéressante.

## II.3 Suites

On souhaite maintenant représenter graphiquement une suite.

**Exercice 5**

Soit la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = 0$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \dfrac{2}{1+2u_n}$. On pose la fonction $f : x \mapsto \dfrac{2}{1+2x}$.

1. Représenter sur un même graphique, et en deux couleurs différentes, la courbe représentative de $f$ et la droite d'équation $y = x$. Que conjecture-t-on quant au comportement de la suite ?
2. Écrire une fonction `suite(n)` qui renvoie la liste des $n$ premiers termes de la suite.
3. Représenter graphiquement la suite, avec en abscisse un tableau de valeurs de $n$ (obtenue avec `np.arange(n)`) et en ordonnée les valeurs de la suite. On pourra configurer la couleur et le type de point, qu'on ne veut certainement pas relier :

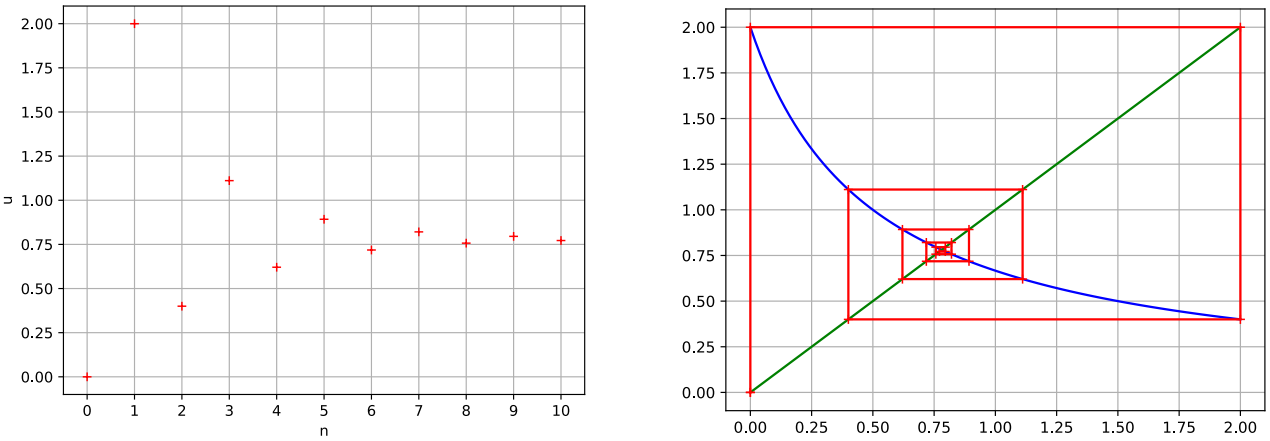


Figure 3. – La suite $(u_n)_{n\in\mathbb{N}}$.

4. Bonus : pouvez-vous tracer le diagramme en toile d'araignée à la question 1 ? Les points à relier ont pour coordonnées successives $(u_0, 0)$, $(u_0, u_1)$, $(u_1, u_1)$, $(u_1, u_2)$, $(u_2, u_2)$, $(u_2, u_3)$, etc.

# III Dériver et intégrer

## III.1 Dériver

Rappelons la formule suivante : pour une fonction $f$ et un point $a \in \mathbb{R}$

$$f'(a) = \lim_{x\to a} \frac{f(x) - f(a)}{x - a}$$

Supposons maintenant que la fonction $f$ est échantillonnée avec un tableau Numpy, c'est-à-dire qu'on dispose d'un tableau `X` d'abscisses et d'un tableau `Y` d'ordonnées. On veut échantillonner de même sa dérivée. Alors on approchera les quantités $f(x) - f(a)$, pour $x \to a$, par l'écart entre les valeurs les plus proches possibles `Y[i+1] - Y[i]`, qu'on divisera par l'écart `X[i+1] - X[i]`. On obtient un nouveau tableau `Z`, qu'on peut tracer en ordonnées par rapport à `X` pour visualiser la dérivée de $f$. Attention, ce `Z` est nécessairement de taille un de moins que `Y`…

**Exercice 6**

1. Écrire une fonction `derive(X, Y)` qui prend en argument deux tableaux supposés de même taille, représentant une fonction échantillonnée, et renvoyant un tableau `Z` de taille un de moins représentant la dérivée.
2. Bonus : pouvez-vous l'écrire *sans* boucle, mais uniquement avec les opérations vectorielles de Numpy ?

**Exercice 7**
Pour les fonctions suivantes, tracer sur un même graphe la fonction $f$ et sa dérivée (obtenue à l'aide de la fonction `derive` précédente, et non pas en calculant la fonction dérivée à la main), de deux couleurs différentes, éventuellement en testant diverses valeurs pour le nombre de points d'échantillonage :

1. $x \mapsto \arctan(x)$ pour $x \in [-6, 6]$,
2. $x \mapsto \sin(x)$ pour $x \in [-2\pi, 2\pi]$.
3. $x \mapsto x^2 e^{-x}$ pour $x \in [-1, 4]$.

### III.2 Intégrer

Pour intégrer une fonction (« calculer l'aire sous la courbe ») $f$ sur un intervalle $[a, b]$, on utilise la **méthode des rectangles à gauche** qui consiste à approximer l'aire sous $f$ entre les points d'abscisse $a$ et $a + h$, pour $h$ très petit, par l'aire d'un rectangle de base $h$ et de hauteur (à peu près constante) $f(a)$, c'est-à-dire le produit $h \times f(a)$. Si on suppose que $f$ est échantillonnée par un tableau Numpy, d'abscisse `X` et d'ordonnée `Y`, alors il faut multiplier les écarts `X[i+1] - X[i]` par `Y[i]` et sommer tout cela.

**Exercice 8**
1. Écrire une fonction `integre(X, Y)`
2. Bonus : pouvez-vous l'écrire *sans* boucle, mais uniquement avec les opérations Numpy ? La fonction `np.sum(X)` calcule la somme de tous les éléments d'un tableau `X`.

Pour tester la fonction :

**Exercice 9**
1. (Mathématiques) Calculer l'intégrale suivante :
$$I = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$$
2. (Python) Donner une approximation de cette intégrale, avec la fonction `integre`, pour de plus en plus de points d'échantillonnage (on pourra écrire une fonction `integrale(n)` qui utilise $n$ points).

## IV Annexe : tableaux à plusieurs dimensions

Le module `numpy` est aussi particulièrement efficace pour gérer des tableaux à plusieurs dimensions.

À deux dimensions, un tableau `X` est composé de lignes et de colonnes. On accède à l'élément de la ligne $i$ et de la colonne $j$ (tous les deux numérotés à partir de 0, comme d'habitude) avec la syntaxe `X[i, j]`. On peut par exemple en créer avec les syntaxes suivantes :

- `np.zeros((n, p))` en lui donnant en argument un tuple $(n, p)$ pour un tableau à $n$ lignes et $p$ colonnes :

```
>>> X = np.zeros((3, 5))
>>> print(X)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

- Utiliser la méthode `reshape`, directement après la création du tableau, pour remodeler la forme selon les besoins :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

Pour un tel tableau, la variable `X.shape` est le tuple $(n, p)$. Le nombre total de cases est exactement $n \times p$, qu'on obtient aussi avec la variable `X.size` (pour un tableau à une dimension, c'est la même chose que `len(X)`).

Le nombre de dimensions du tableau est appelé dans le vocabulaire Numpy le nombre d'**axes**. Un tableau à trois axes ressemble à un empilement de tableaux de dimension 2 et ses cases sont indicées par la syntaxe `X[i, j, k]`.

Exemple avec un tableau de forme $(3, 2, 4)$, représenté comme un empilement de 3 tableaux à 2 lignes et 3 colonnes :

```
>>> X = np.arange(24).reshape(3, 2, 4)
>>> print(X)
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]
>>> X.shape
(3, 2, 4)
>>> X.size
24
```

Enfin, on peut trancher un tableau à plusieurs axes, indépendamment sur chaque axe, en séparant par des virgules les différents syntaxes de tranche :

```
>>> X = np.arange(24).reshape(4, 6)
>>> print(X)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> print(X[:3, 1:4]) # lignes 0, 1, 2, colonnes 1, 2, 3
[[ 1  2  3]
 [ 7  8  9]
 [13 14 15]]
>>> print(X[:, :1]) # toutes les lignes, seulement la première colonne
[[ 0]
 [ 6]
 [12]
 [18]]
>>> print(X[:, 0]) # toutes les lignes, colonne 0
[ 0  6 12 18]
# c'est presque pareil mais c'est un tableau à un seul axe
>>> print(X[:, ::-1]) # mêmes lignes, colonnes renversées
[[ 5  4  3  2  1  0]
 [11 10  9  8  7  6]
 [17 16 15 14 13 12]
 [23 22 21 20 19 18]]
```

*Remarque.* Il est intéressant de réfléchir au fait que toutes ces opérations (`reshape`, tranches) ne « bougent » rien dans la mémoire : les éléments d'un tableau à plusieurs dimensions restent toujours rangés les uns à la suite des autres, alignés sur des adresses mémoires consécutives. Ce qui change avec la forme du tableau, c'est seulement la façon de numéroter ces mêmes éléments. Observons par exemples les deux tableaux suivants :

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
| 9 | 10 | 11 |
| 12 | 13 | 14 |

Dans les deux cas, il s'agit bien de cases de mémoire rangées consécutivement et numérotées de 0 à 14. Prenons par exemple l'élément numéroté 8 : c'est *parce que* le premier est de forme $(3, 5)$ que cet élément est en ligne 1 colonne 3, alors que dans le deuxième de forme $(5, 3)$ il est en ligne 2 colonne 2. De même, prenons par exemple la case $(2, 1)$ : elle correspond à l'élément 11 dans le premier tableau mais à 7 dans le deuxième, ce que l'on peut savoir uniquement en connaissant leur forme. C'est toujours un petit jeu d'arithmétique qui permet de calculer, à partir de la connaissance de la taille et de la forme d'un tableau, à quelle adresse se situera l'élément en ligne $i$ et colonne $j$ ; ou réciproquement, étant donnés des éléments numérotés consécutivement en mémoire, de décider à quelle ligne et à quelle colonne correspond le $n$-ième élément.

Les opérations de tranches, elles aussi, ne « tranchent » rien du tout dans la mémoire. Par exemple l'élément d'indice $i$ de `X[1:]` est l'élément d'indice $i + 1$ de `X`, là encore il s'agit seulement de quelques manipulations arithmétiques cachées à l'utilisateur.