Lab0 - Introduction à Python
Automne 2020

Professeur : Benjamin Leener
Chargés de laboratoire : Vicente Enguix, Gabriel Lepetit-Aimon

Installation de l'environnement de développement

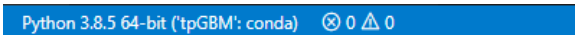
Les laboratoires de ce cours seront codés en **Python 3**. Pour chaque laboratoire (y compris celui-ci) un notebook IPython (au format .ipynb) vous sera fourni. Ce document est équivalent au Matlab Live Editor, il contiendra la structure des exercices et le squelette du code organisé en cellule. Le code et les réponses aux questions seront à compléter dans le Notebook que vous remettrez sous Moodle.
Vous aurez besoin d'une version de Python >3.5, des modules **`numpy`**, **`matplotlib`**, **`pandas`**, **`scipy`** et **`cv2`** et d'un éditeur de Notebook IPython (jupyter, vscode...).

Voici une des procédures possibles pour installer ces outils :
1. Installer miniconda un version allégée du gestionnaire d'environnement Anaconda. Miniconda vient avec Python 3.8 et simplifiera l'installation de modules supplémentaires.
2. Lancer la ligne de commande **`Anaconda Prompt`** sous Windows, ou votre terminal habituel sous linux et macOS et exécuter les lignes de commandes suivantes qui créeront un environnement python **`tpGBM`** et y installeront les modules nécessaires :

```
conda create -n tpGBM python=3.8 numpy matplotlib pandas scipy ipykernel notebook
conda activate tpGBM
pip install opencv-python
```

3. Installer l'éditeur vscode puis son extension ms-python.
4. Ouvrir le notebook **`tp0.ipynb`** dans vscode (**`File > Open folder`** puis sélectionner le chemin du dossier qui contient le notebook et ouvrir le fichier depuis le panneau **`EXPLORER`**)

5. Assurer vous que la version de python sélectionnée dans la barre de statut bleu en bas de l'écran est bien **`tpGBM`** :



Vérifier aussi que le notebook est indiqué comme **`Trusted`** en haut à droite de l'éditeur dans les informations de connexions au serveur python. Un popup apparaîtra probablement si ça n'est pas le cas.

6. Tenter d'exécuter la première cellule de code en cliquant sur l'icone ▷ dans le bandeau de la cellule. Le serveur jupyter devrait démarrer automatiquement en arrière plan et l'indicateur d'ordre d'exécution de la cellule devrait indiquer **`[1]`**. Il est possible qu'il faille redémarrer vscode pour que les précédents changements soient pris en compte.

Tous ces outils de développement sont libres et ne nécessitent aucune licence, si vous avez des difficultés à les installer, contactez votre chargé de laboratoire.

La suite de ce laboratoire est optionnelle si vous maîtrisez déjà Python et les modules **`numpy`** et **`matplotlib`**. Vous trouverez à la fin de ce document et sur moodle une *cheatsheet* qui énumère les fonctions usuelles de ces modules que nous utiliserons durant cette session.

Ce laboratoire présente les deux modules suivant :

— `numpy` fournit des outils pour manipuler des matrices de dimension arbitraire. Il est importé dans le notebook sous le nom **`np`**.

— `matplotlib.pyplot` reproduit les méthodes de matlab pour le tracé de graphs. Il est importé sous le nom de **`plt`**.

? Bien qu'ils soient installés, les modules pythons ne sont pas directement utilisables dans l'interpréteur natif : il est nécessaire de les importer avec la commande **`import`** ***`nom_du_module`***. Les fonctions de ce module seront alors disponible dans le reste du document par ***`nom_du_module.nom_de_fonction()`***.

i Lors de son import, on peut attribuer un acronyme à un module pour y accéder plus aisément dans le reste du code. Ainsi **`import numpy as np`** importe le module numpy sous le nom **`np`**. La variable **`pi`** du module sera alors accessible via **`np.pi`**.

i Il est aussi possible de n'importer que certains attributs d'un module pour pouvoir y accéder directement dans le code. Ainsi **`from numpy import pi`** permet d'accéder à la valeur de π directement sous le nom **`pi`** (plutôt que **`np.pi`**).

Exercice I : **Vecteurs et Courbes**

1. Calculer et afficher le résultat de $\sin(0)$ et $\sin(\frac{\pi}{2})$.

> ? Le module **`numpy`** fourni de nombreuses fonctions mathématiques (notamment **`np.sin()`**) et de constantes (notamment **`np.pi`**).

2. Définir le vecteur t composé de 401 valeurs réparties uniformément entre 0 et 4 avec `np.linspace`, puis afficher la longueur du vecteur.

> ? La longueur de tout objet similaire à une liste (liste, tuple, vecteur, chaine de caractères...) peut être lu avec **`len(t)`**.

> i Dans vscode vous pouvez, depuis la barre d'outils propre au Notebook, afficher la liste des variables définie dans le document.

3. Calculer les signaux $s_0 = \sin(\pi t)$, $s_1 = \frac{\sin(3\pi t)}{3}$ et $s_2 = \frac{\sin(5\pi t)}{5}$ où t est la valeur définie à la question précédente.

> ? Les fonctions mathématiques comme les opérations arithmétiques, lorsqu'elles sont appliquées à une matrice numpy, s'appliquent à chacun des éléments de la matrice.

4. Affichez sur le même graphique : s_0, s_1 et s_2 et leur somme. Pour rendre le diagramme plus lisible tracer les 3 signaux en pointillés. N'oubliez pas le titre et la légende !

> ? Les méthodes à utiliser et un lien vers leur documentation sont disponibles dans la liste de fonction présente à la fin de ce document.

On peut observer que la somme des signaux reste périodique mais ressemble de moins en moins aux sinusoïdes dont elle est composée. Essayons d'augmenter encore le nombre de signaux sommés.

5. Calculer et afficher le signal : $\sum_{i=0}^{50} \frac{\sin((2i+1)\pi t)}{2i+1}$.

Vous pourriez définir un vecteur nul de même taille que t avec **`s = np.zeros_like(t)`** puis utiliser une boucle **`for i in range(50):`** pour ajouter successivement à **`s`** chacun des signaux de la somme.

En Python, les blocs de codes lors de déclaration des boucles, des conditions, des fonctions, etc, ne sont pas délimité par des **`{ }`** (comme en C) ou par **`end`** (comme en Matlab), mais par leur niveau d'indentation (généralement 4 espaces ou une tabulation par niveau). Par exemple :

```
for i in range(10):
→   if i % 2:
→   →   print(i, " est impair")
→   else:
→   →   print(i, " est pair")
print("Fin de boucle")
```

6. Réessayer pour i allant de 0 à 500 et afficher le signal. Vous venez d'approximer un signal carré par une somme de sinusoïdales !

En traitement du signal, on nomme *décomposition en série de Fourier* le procédé qui permet de décomposer un signal périodique en une somme infinie de sinusoïdes de fréquences supérieures. Sa généralisation à tout signal complexe continu par morceaux s'appelle *la transformé de Fourier* que vous serez amené à utiliser dans ce cours.

Exercice II : Images et fonctions

Cet exercice présente le stockage et la manipulation d'images sous forme de matrices numpy.

1. Charger et afficher l'image **`chat.png`**. Dans la suite de l'énoncé on considèrera qu'elle est stockée dans la variable **`img`**.

> ? Vous pourrez utiser les méthodes **`imread`** et **`imshow`** du module **`matplotlib.pyplot`**.

2. Vérifier que l'image est stockée sous la forme d'une matrice numpy : **`numpy.ndarray`** en affichant sont type (accessible par **`type(img)`**). Afficher le format de stockage des données et les dimensions de la matrice. Identifier quel canal correspond à la hauteur, la largeur et au nombre de canaux de l'image.

> ? Les matrices numpy possèdent des attributs qui décrivent les propriétés de la matrice. Ainsi **`img.dtype`** contient le format des données, **`img.ndim`** contient le nombre de dimensions de la matrice et **`img.shape`** sa taille selon chaque dimensions.

> i Le format de données détermine les types de nombres stockable par la matrice : flotant ou entier, signé ou non-signée (valeur uniquement positive), nombre d'octet de stockage... Pour limiter la taille de stockage des images, les pixels sont généralement codés en **`uint8`** : entier non-signé sur un octet (valeurs comprises entre 0 et 255) mais pour le traitement elles sont parfois converties en **`float32`** (flotant signé sur 4 octets, valeur comprises entre 0 et 1). Lorsque vous manipulerez des images, portez attention au format attendu par les différents modules de traitement.

3. Affichez le minimum, le maximum et la moyenne des valeurs de l'image.

> ? Les matrices numpy disposent de nombreuses méthodes qui dérivent des résultats de leurs valeurs. Minimum : **`img.min()`** ; maximum : **`img.max()`** ;
> somme : **`img.sum()`** ; moyenne : **`img.mean()`** ; écart-type : **`img.std()`** ;
> matrice transposée : **`img.transpose()`**...

4. Afficher la moyenne des valeurs pour chaque canal de l'image.

> **?** Le comportements des méthodes présentées à la question précédente peut être modifié par des arguments optionnels : en l'occurrence pour mean le paramètre **`axis`** détermine quelles dimensions de la matrice seront moyennées.

> **i** En Python, il existe deux types d'arguments : les arguments obligatoire et les arguments optionnels (qui prennent une valeur par défaut lorsqu'il ne sont pas spécifié). Pour spécifier la valeur d'un argument on peut :
>
> — fournir sa valeur : dans ce cas c'est sa position qui déterminera à quel argument elle correspond (la première valeur correspond au premier argument, la seconde au second...). Ici **`img.mean( (0,1) )`**.
>
> — spécifier le nom de l'argument. Ici **`img.mean( axis=(0,1) )`**. Cette syntaxe est souvent plus lisible et permet de modifier le 3ème paramètres optionnels sans spécifier la valeur des 2 premiers.
>
> Lorsqu'on spécifie plusieurs arguments, les deux méthodes peuvent être combinées en plaçant d'abord les arguments de position puis les arguments nommés : **`plt.plot([0,1,2], [0,1,2], color='red', linestyle='-')`**.

5. Afficher uniquement la tête du chat : tronquer l'image pour afficher la fenêtre comprises entre les lignes 93 à 230 et les colonnes 240 à 364.

> **?** Pour accéder à des éléments spécifiques d'une matrice numpy (mais ça vaut aussi pour tous les objets similaire à des listes) ont utilise la syntaxe : **`img[`*`dim1`*`, `*`dim2`*`]`** i.e. pour lire la case placée sur la première ligne, dernière colonne : **`img[0, -1]`**.
> Il est possible de lire un intervalle de cases consécutives avec la syntaxe :
> **`img[`*`ymin`*`:`*`ymax`*`, `*`xmin`*`:`*`xmax`*`]`**
> Si l'index min est omit l'intervalle commencera au premier élément, si l'index max est omit l'intervalle s'étendra jusqu'au dernier élément. Ainsi **`img[:, -3:]`** sélectionnera les 3 dernières colonnes (de la première à la dernière ligne).
> L'ensemble des fonctions d'indexages des matrices numpy sont décrites ici.

> **!** Attention ! En Python (et contrairement à Matlab) les indexes commence à 0 et les intervalles vont de l'index min à l'index max *exclu*. Ainsi l'intervalle correspondant aux 3 premiers éléments est **`0:3`** soit **`[0,1,2]`**.

6. Affichez le canal bleu de l'image. (Soit toutes les lignes et colonnes du 3e canal.)

Les smileys doivent être noirs puisqu'ils ne contiennent pas de bleu.

> **?** Lorsque **`imshow`** doit afficher un matrice avec 3 ou 4 canaux, il considère automatiquement que c'est une image RGB (ou RGBA). Cependant lorsqu'il doit afficher une matrice avec 1 canal, il considère que c'est une carte d'intensité et l'affiche tel quel (Intensités min en violet et intensités max en jaune). Dans notre cas, on souhaite afficher le canal bleu en noir et blanc, il faut donc spécifier à **`imshow`** la *color map* à utiliser avec le paramètre : **`cmap='gray'`**.

Pour stocker et manipuler les images couleurs, on encode la couleur de chaque pixel dans 3 canaux. Quand cette image est destinée à être affichée sur un écran, on utilise le plus souvent le format **`rgb`** où chaque canal encode l'intensité du rouge (1er canal), vert (2e canal), bleu (3e canal). Parfois un 4ème canal est ajouté pour stocker la transparence du pixel (le canal alpha).

Pour améliorer la lisibilité de notre image on va afficher chaque canal dans sa couleur originale. Pour se faire, l'image affichée doit avoir ses 3 canaux de couleurs dont 2 nuls.

7. Créer une fonction qui retourne une image dont tous les canaux de couleurs ont été mis à 0 sauf un, spécifié en paramètre.

La signature de la fonction est donnée dans le notebook.

Comme souvent en informatique, il y a plusieurs solutions pour coder cette fonction. Une solution consiste à duppliquer l'image puis à faire une série de condition qui annule le 1er, 2eet 3ecanaux si **`channel`** est différent de **`0`**, **`1`**, **`2`** respectivement.

Attention en Python tous les objets (à l'exception des types primaires : **`int`**, **`float`**, **`bool`** et **`tuple`**) sont passés en référence. C'est à dire que la copie d'une variable ne la duplique pas en mémoire : une modification de l'une modifiera l'autre. Par exemple :

```
» a = [0,1,2]
» b = a
» b[0] = 3
» print(a)
[3, 1, 2]
```

Lorsqu'on modifie un argument d'une fonction dans sa définition il faut donc généralement en forcer la copie : **`img = img.copy()`**. Ainsi la variable passer en argument lors de l'appel de la fonction ne sera pas modifiée en dehors.

8. Affichez côte à côte dans la même fenêtre tous les canaux de l'image.

`fig, (ax0, ax1, ax2) = plt.subplots(1, 3)` permet de subdiviser l'aire de tracé pour en afficher trois graphiques côtes à côtes.

Vous devriez obtenir un résultat semblable à :

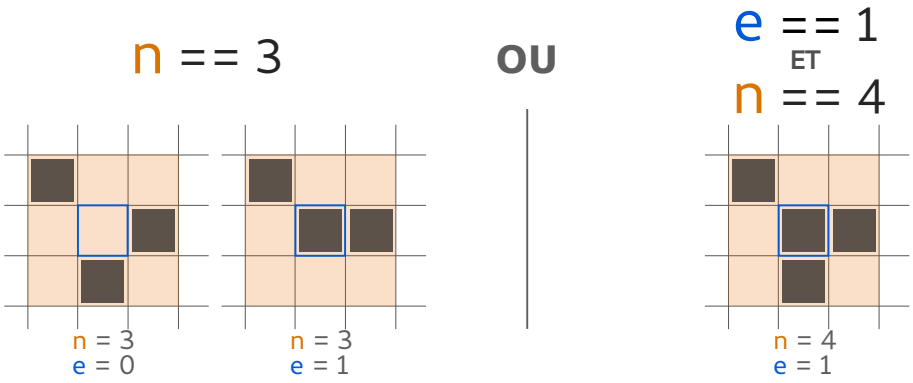
Exercice III : Images binaires, calcul matriciel, convolution et jeu de la vie

Le jeu de la vie est un automate cellulaire qui modélise l'activité cellulaire. Ses règles, imaginées en 1970 par John Horton Conway, sont très simples :

- Le «plateau du jeu» est une grille à deux dimensions dont chaque case (ou cellule) peut-être soit vivante soit mortes.
- Une cellule morte ayant 3 voisines vivantes devient vivante ou «nait».
- Une cellule vivante ayant 2 ou 3 voisines vivantes reste vivante.
- Une cellule vivante ayant 4 voisines ou plus meurt de surpopulation.
- Le jeu est itératif : une itération consiste à recalculer l'état de chaque cellule.

Reformulons la règle de décision permettant de choisir si une cellule sera vivante ou non à la prochaine itération. Si on nomme `e` l'état d'une cellule dont on veut déterminer l'état à l'itération suivante (vivante : `1`, morte : `0`) et `n` le nombre de cellules actuellement vivantes dans un carré de 3 cases de cotés centré sur cette cellule, alors elle sera vivante que si :

```
n == 3 ou (e==1 et n==4).
```



1. Chargez la matrice `map.npy` avec `np.load()` et affichez la. Cette matrice est l'image binaire que nous allons faire évoluer avec l'algorithme du jeu de la vie.

2. Pour calculer l'expression récursive on va voir besoin de quantifier le nombre de cellule vivante dans le voisinage de chaque pixel. La fonction **`N(map)`** définie plus bas, renvoie une carte où tous les pixels ont été remplacés par leur nombre de cellule vivante dans le voisinage tel que défini en introduction.

```
def N(carte):
    from scipy.ndimage import convolve
    return convolve(carte*1, np.ones((3,3)), mode='constant')
```

Vérifiez que la fonction **`N(carte)`** renvoie bien la matrice **`n`** telle que définie en introduction, en affichant **`N(map)`**.

> **`convolve`** implémente d'une opération de traitement d'image appelée *la convolution*. Cette opération somme, pour chaque pixel, son intensité et celles des ces voisins pondérées par un masque (ici un masque unitaire 3x3). Vous la découvrirez plus en détail dans le chapitre 3 et 4.

3. On nomme **`e`** l'image binaire (où les pixels vivants valent **`1`** et les pixels morts valent **`0`**) et **`N(e)`** la carte contenant pour chaque pixel la valeur **`n`** telle que définie précédemment (nombre de cellules vivantes dans un carré de 3x3 autour du pixel).

Définir la fonction **`compute_next_map(e)`** qui prend en argument la carte actuelle **`e`** et renvoie la carte suivante selon la relation de récursion :

$$e_{i+1} \;=\; (N(e_i) == 3) \;\text{ OU }\; (e_i == 1 \text{ ET } N(e_i) == 4)$$

> En algèbre booléenne où **`FAUX`** est représenté par **`0`** et **`VRAI`** par tout entier positif, on peut remplacer l'opérateur **`OU`** par **`+`** et l'opérateur **`ET`** par **`*`**.

> En Python, les boucles **`for`** sont particulièrement lentes (comparée au C/C++). On n'effectue donc jamais de boucle sur tout les pixels d'une image.

> On décrira plutôt l'opération à exécuter élément par élément, et on laissera Numpy l'effectuer pour tous les éléments du tableau.
> Ainsi pour une matrice **`M`** à n dimensions, **`M==2`** renverra une matrice binaire de même taille que **`M`** qui vaut 1 pour les cases de **`M`** égale à **`2`** et 0 sinon.
> De même, **`r = (M==1) + (M==4)`** stockera dans **`r`** la somme élément par élément de **`M==1`** et **`M==4`**, et est équivalent (en beaucoup plus rapide) à :
>
> ```
> r = np.zeros_like(M)
> for y in range(M.shape[0]):
> →   for x in range(M.shape[1]):
> →   →   r[y,x] = (M[y,x]==1) + (M[y,x]==4)
> ```

4. Calculer et afficher côte-à-côte les cartes e_0 (la carte initiale égale à la matrice **`map`**), e_1, e_2 et e_3 calculées par appels successifs de la méthode **`compute_next_map`**.

5. Afficher à l'aide d'une boucle **`while`** l'évolution de la carte binaire soumise à l'algorithme du jeu de la vie, itération après itération.

Au bout d'un certain temps, l'algorithme attendra un état stable : un cycle de 2 images qui se répètent à l'infini. Vous devrez détecter ce cycle pour arrêter la boucle.

> **?** Pour mettre à jour la figure affichée sous la cellule de code, vous pouvez utiliser **`clear_output(wait=True)`** avant **`plt.show()`** pour remplacer la figure précédente et non en ajouter une nouvelle.
> Le tracé des figure dans vscode ou jupiter prend beaucoup de temps : environ un quart de seconde. Pour atteindre plus vite le cycle stable (après plus de 1700 itérations) vous pouvez ne tracer qu'une itération sur 20.

> **?** Pour vérifier si le cycle stable est atteint, il vous faudra en permanence stocker dans une liste les deux dernières itérations de la carte et, à chaque itération, les comparer avec la nouvelle.
> Pour comparer si deux matrices Numpy sont égales vous pouvez utiliser :
> **`(previous_map == current_map).all()`**

Python CHEATSHEET

Python Standard

POLYTECHNIQUE MONTRÉAL

Raccourcis Clavier (Notebook)

Exécuter la cellule		**Ctrl + Enter**
Exécuter et passer à la cellule suivante		**Shift + Enter**
Auto-complétion	**(vscode)**	**Ctrl + Espace**
Auto-complétion	**(jupyter)**	**TAB**

Types Primaires

Chaine de caractères	**str**	s=**'3.0'** ou s=**"False"**
Nombre Entier	**int**	i=**3**
Nombre Flottant	**float**	f=**3.0**
Boolean	**bool**	b=**True** ou b=**False**
Liste	**list**	l=**['3.0', False, 3]**
Liste immuable	**tuple**	t=**('3.0', False, 3)**
Dictionnaire	**dict**	d=**{'pi': 3.14,'faux': False}**
Ensemble	**set**	e=**{1, 4, 5, 10}**
Objet nul		**None**
Lire le type d'une variable		**type(** *variable* **)**

Liste

Instancier une liste	**l = [** *1*, *'2'*, *False* **]**
Ajouter un élément	**l.append(** *value* **)**
Insérer un élément	**l.insert(** *i* **,** *value* **)**
Supprimer un élément	**l.remove(** *value* **)**
Trier la liste	**l.sort( )**
Concatener deux listes **l1** et **l2**	**l1 + l2**

Chaine de caractères

Instancier une chaine	**s = "texte"**
Concaténer deux chaines	*s* **+** *'chaine2'*
Test si *s* commence par *'te'*	*s*.**startswith(***'te'***)**
Test si *s* fini par *'te'*	*s*.**endswith(***'te'***)**
Remplacer 'tex' par 'chan'	*s*.**replace(***'tex'*, *'chan'***)**
Formater	**"%i / %s / %.2f" % (i, s, f)**

Dictionnaire

Instancier une dictionnaire	**d = {'***key1***':** *v1*, **'***key1***':** *v2***}**
Modifier un élément	**d['***key***'] =** *value*
Itérer sur un dictionnaire	**for** *key*, *value* **in** *d*.**items():**

Opérations Générique

Déclarer une fonction	**def** *nom* **(** *arg1*, *arg2* **=** *defaut* **):** → ... → **return** *valeur*
Boucle: For	**for** i **in range(***n* **):**
Boucle: While	**while** *condition* **:**
Condition: Si	**if** *condition* **:**
Condition: Sinon si	**elif** *condition* **:**
Condition: Sinon	**else:**
Afficher dans la console.	**print(** *"description:"* **,** *valeur* **)**
Commentaire	**#** *Commentaire*

Opérateurs

Égal / Différent	**==** / **!=**
Supérieur / Supérieur ou Égal	**>** / **>=**
Inférieur / Inférieur ou Égal	**<** / **<=**
Inversion booléenne	**not**
ET booléen / OU booléen	**and** / **or**
Test si nul	**is None**
a^b	**a ** b**

Commun aux list, tuple, str...

Longueur	**len(l)**
Supprimer le i^{ème} élément	**del l[***i***]**
Tester si l'objet contient une valeur	***value*** **in l**
Itérer sur la liste	**for** *element* **in** *l* **:**
Appliquer la fonction **f** à toute la liste	**[** f(v) **for** v **in** l **]**
Filtrer la liste des objets nuls	**[** v **for** v **in** l **if** v **is not None]**

Indexes

Les indexes commence à 0 !

Lecture / Écriture du i^{ème} élément	**v = l [i] / l [i] = v**
Accéder au i^{ème} dernier élément	**l [-i]**
Accéder du 2ème au 4ème éléments	**l [1:4]**
Accéder aux 3 premiers éléments	**l [:3]**
Accéder aux 3 derniers éléments	**l [-3:]**
Accéder à un élément sur deux	**l [::2]**
Inverser l'ordre des éléments	**l [::-1]**

Python CHEATSHEET

Manipulation de Signaux

POLYTECHNIQUE MONTRÉAL

Tableau (Matrices)

```
import numpy as np
```

Création de tableaux

Convertir une liste en tableau	arr = **np.array(** [0,1,2] **)**
Créer un tableau **N**x**M** de 0	arr = **np.zeros((N,M))**
Créer un tableau **N**x**M** de 1	arr = **np.ones((N,M))**
Créer une matrice identité **N**x**N**	arr = **np.eye(N)**
Vecteur: [**j**, **j**+**i**, ..., **k**]	arr = **np.arange(j, k, i)**
Intervalle de **j** à **k** avec **N** points	arr =**np.linspace(j, k, N)**
Charger depuis un fichier	arr = **np.load('path')**
Sauvegarder vers un fichier	**np.save('path',** arr **)**

Propriétés des tableaux

Type des données	arr.**dtype**
Nombre de dimensions	arr.**ndim**
Taille sur chaque dimension	arr.**shape**

Calculs sur les tableaux

Valeur minimale / maximale	arr.**min()** / arr.**max()**
Index de la valeur minimale	arr.**argmin()**
Index de la valeur maximale	arr.**argmax()**
Valeur moyenne	arr.**mean()**
Écart-type	arr.**std()**
Somme de tous les éléments	arr.**sum()**
Produit de tous les éléments	arr.**prod()**
Test si tout le tableau est vrai	arr.**all()**

Conversion de tableaux

Modifier le type de données	arr.**astype (***np.dtype***)**
Modifier les dimensions	arr.**reshape(** (*d1, d2, ...*) **)**
Permuter les axes	arr.**transpose(** *a1, a2, ...* **)**
Transformer en vecteur	arr.**flatten()**
Dupliquer le tableau	arr.**copy()**

Graphique

```
import matplotlib.pyplot as plt
```

Créer la zone du tracé

Créer un graph	fig, ax = **plt.subplots()**
Créer plusieurs sous-graphs	fig, axs = **plt.subplots(** *ny*, *nx* **)**
Afficher le graph	fig**.show()**
Choisir la taille du graph	fig**.set_size_inches(***width*, *height* **)**

Tracé le graphique

Tracer une courbe	ax.**plot(** x, y,*label="nom", color='red',...***)**
Tracer un nuage de point	ax.**scatter(**x, y,*label="nom", color='r',...***)**
Tracer un histogramme	ax.**bar(** x, height, *label="nom", color='r',...***)**
Trace un polygone	ax.**fill(** x, y, *color='r',...***)**
Tracer une ligne horizontal	ax**.hlines(** y, xmin, xmax **)**
Tracer une ligne vertical	ax**.vlines(** x, ymin, ymax **)**
Afficher une annotation	ax.**annotate(** "texte", (xpos, ypos) **)**
Affiche une image	ax**.imshow(** array, *cmap="gray"* **)**

Configurer l'apparence du graphique

Spécifier un titre au graph	ax.**set_title(** "Titre" **)**
Spécifier un titre global	fig.**suptitle(** "Titre de la figure" **)**
Spécifier un titre à l'axe X	ax.**set_xscale(** "Axe X" **)**
Spécifier un titre à l'axe Y	ax.**set_yscale(** "Axe X" **)**
Désactiver les axes	ax.**axis("off")**
Afficher une grille	ax.**grid(** *color='r', linewidth=2* **)**
Affiche la légende	ax.**legend()**

Manipulation d'images

Charger une image	img = **plt.imread("***fichier.jpg* **")**
Accès à une région	**img[** y_{min}: y_{max} , x_{min}: x_{max}**]**
Accès à un canal	**img[** : , : , canal **]**

Pour plus de détails sur les commandes de traitement de signaux, consulter les documentations respectives de **Numpy** et **Matplotlib**.
Tous les icônes et les noms de méthodes présentés dans ce document sont des liens vers ces documentations.