Chapitre III

Programmation d'algorithmes de base

Table des matières

Partie A

Recherche séquentielle dans un tableau unidimensionnel. Dictionnaire

Dans cette partie, nous nous intéresserons à la programmation d'algorithmes de recherches dans un tableau, par exemple, la recherche d'un élément donné ; d'un maximum lorsque les éléments font partie d'ensemble ordonné. Nous utiliserons également la structure de dictionnaire pour compter les occurrences des éléments d'un tableau.

1. Tableau unidimensionnel

En algorithmique, les tableaux unidimensionnel s'apparentent aux suites mathématiques : ils permettent de manipuler sous un même nom plusieurs variables (a priori) de même type chacune repérée par un numéro : son indice dans le tableau.

a. Définitions

Définition 1. *Tableau unidimensionnel*

Un **tableau unidimensionnel** est une collection finie d'éléments dont chacun est désigné de manière unique par un nombre entier naturel appelé **indice** de l'élément dans le tableau. *Le temps d'accès aux éléments d'un tableau par leurs indices est constant.*
On appelle **taille** d'un tableau la nombre d'éléments qui le compose.

Exemple 1. *Tableaux en Python*

Dans la suite, nous programmerons nos tableaux en Python grâce au type `list` ou encore `tuple`. Nous avons vu rapidement le type `array` fourni par le module `numpy` ; ce type est un "meilleur" représentant de ce qu'est un tableau en informatique : données de même type et ajout/suppression impossibles pour garantir un accès aux éléments en temps constant.

Tableau avec le type `list`

```
>>>t = [1,2,7,5,3]
```

Tableau avec le type `tuple`

```
>>>t = (1,2,7,5,3)
```

Dans la suite de cette partie et dans les T.P. qui en dépendent, nous pourrons appliquer nos programmes de recherche indépedemment aux types `list` et `tuple` car la syntaxe d'accès aux élements est la même pour ces deux types.

Remarque 1.

Dans certains langages de programmation (les langages à typage statique comme C ou OCaml), les éléments d'un tableau doivent tous avoir le même type, on définit dans ce cas le **type d'un tableau** comme étant le type commun de ses élements.
En Python, qui est un langage à typage dynamique, les listes (type `list`) et les tuples (type `tuple`) permettent des collections hétérogènes d'élements, par exemple, `[1,'coucou',[2,3]]` est valide.

2. TP : Recherche séquentielle dans un tableau unidimensionnel

Dans le T.P. suivant, nous allons étudier le principe de recherches séquentielles dans un tableau :

TP n°2 - Recherches séquentielles

Partie B

Boucles imbriquées

Dans les parties précédentes, nous avons souvent utilisé les boucles `for` et `while` pour parcourir des listes d'entiers ou des chaînes de caractères : on avait affaire à des tableaux unidimensionnels. Dans ce cas, une simple boucle allant de 0 jusqu'à la taille du tableau moins 1 permet de récupérer la valeur de chaque élément du tableau.
Mais lorsqu'on a affaire à un tableau bidimensionnel ou même multidimensionnel, par exemples, une liste de listes ou une image numérique, nous devrons recourir à plusieurs boucles imbriquées pour accéder à ses éléments.

1. Exemples de boucles imbriquées

Voici un premier exemple, le parcourt d'une liste de listes :

Exemple 2.

```
1 M=[[1,2,3],[4,5],[7,8,9,10]] #liste de listes
2
3 for i in range(len(M)):
4     for j in range(len(M[i])):
5         print(M[i][j],end=' ')
6     print('fin pour i =',i)
```

Dans cet exemple, la liste `M` contient 3 éléments : les listes `[1,2,3]`, `[4,5]` et `[7,8,9,10]`. Ainsi, la variable `i` va de 0 à 2 dans la 1ère boucle `for` et pour chacune de ces valeurs de `i`, la deuxième boucle est appelée :

- Pour `i` = 0, `M[i]` vaut `[1,2,3]` donc `len(M[i])` vaut 3. Ainsi, le variable `j` prend les valeurs de 0 à 2, ce qui affichera alors, dans l'ordre, les valeurs de `M[0][0]`, `M[0][1]` et `M[0][2]` i.e. `1`, `2` et `3`.
- Pour `i` = 1, `M[i]` vaut `[4,5]` donc `len(M[i])` vaut 3. Ainsi, le variable `j` prend les valeurs de 0 puis 1, ce qui affichera alors, dans l'ordre, les valeurs de `M[1][0]` et `M[1][1]` i.e. `4` et `5`.
- Pour `i` = 2, `M[i]` vaut `[7,8,9,10]` donc `len(M[i])` vaut 4. Ainsi, le variable `j` prend les valeurs de 0 à 3, ce qui affichera alors, dans l'ordre, les valeurs de `M[2][0]`, `M[2][1]`, `M[2][2]` et `M[2][3]` i.e. `7`, `8`, `9` et `10`.

oui, oui, il manque le 6 !

Mathématiquement également, lorsqu'on a affaire à des sommes avec plusieurs indices de sommation, on a recours aux boucles imbriquées pour effectuer leur calcul numérique. Deuxième exemple, le calcul d'une somme double :

Exemple 3.

On veut calculer numériquement, pour une valeur donnée d'un entier naturel n, la somme suivante :

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{1}{2^{i+j}}.$$

On procède exactement de la même façon que pour un calcul de tête "brutal" : pour chaque valeur de i entre 1 et n, on cumule les sommes cumulées des $\frac{1}{2^{i+j}}$ avec j allant de 1 à n.
On définit donc la fonction suivante d'argument `n` entier naturel :

```
1 def somme_double(n):
2     S = 0 # contenant du cumul
3     for i in range(1,n+1):
4         for j in range(1,n+1):
5             S += 1/2**(i+j)
6     return S
```

Remarque 2.

Lors de l'appel de `somme_double(n)` avec n un enter naturel, l'instruction `S += 1/2**(i+j)` est exécuté n^2 fois, on dit alors cet algorithme est de *complexité temporelle* **quadratique**.
On rappelle que lorsque nous avons vu l'algorithme de recherche séquentielle dans un tableau de taille n, dans le pire des cas, il y avait n itérations : la complexité temporelle était alors **linéaire**.

Nous préciserons la notion de complexité temporelle plus tard dans le cours.

2. TP : Boucles imbriquées

Dans le T.P. suivant, nous allons nous familiariser avec les boucles imbriquées :

TP n° ?? A FINIR !- Boucles imbriquées

Partie C

Fonctions récursives

Dans cette partie, nous allons étudier une autre façon de penser nos algorithmes. Jusqu'à présent, nous avons utilisé des boucles (for ou while) pour les programmer - on dit que ce sont des algorithmes *itératifs* ; le principe de *récursivité* va nous permettre d'avoir une nouvelle approche, dans certains cas plus naturelle, pour le faire.

1. Définition et premier exemple

Définition 2. *Fonction récursive*

On appelle **fonction récursive** est une fonction qui s'appelle elle-même.

Pernons l'exemple du calcul de $n!$ pour un entier naturel n :

Remarque 3. *Version itérative*

Nous connaissons déjà l'algorithme itératif permettant ce calcul, celui-ci étant basé sur l'égalité : pour $n \in \mathbb{N}$,

$$n! = \prod_{k=1}^{n} k$$

```
1 def facto_it(n):
2     p=1
3     for k in range(1,n+1):
4         p=p*k
5     return k
```

Exemple 4. *Version récursive*

L'algorithme récursive est basé sur la définition *récurrente* de $n!$:

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \quad \text{pour } n \in \mathbb{N}^* \end{cases}$$

On remarque alors que si on connait la valeur de $(n-1)!$, on obtient la valeur de $n!$ en la multipliant simplement par n. Et si on ne connait pas $(n-1)!$, on l'obtient en multipliant $(n-2)!$ par $n-1$; et si on ne connait pas $(n-2)!$ ... etc...
A force de "faire -1", on arrivera sur $0!$ dont on connait la valeur. Il ne reste plus qu'à remonter toute cette chaîne pour obtenir $n!$.

Voyons comment on peut traduire algorithmiquement ce principe :

```python
def facto_rec(n):
    if n==0:
        return 1
    else:
        return n*facto_rec(n-1)
```

Pour mieux comprendre le déroulement d'un appel de la fonction `facto_rec`, on ajoute y les instructions suivantes qui nous permettront de suivre ce qui s'y passe :

Regardons sous le capot !

```python
def facto_rec(n):
    if n==0:
        print('\nappel de fact_rec('+str(n)+') : renvoie 1\n')
        return 1
    else:
        print(n*'--'+' appel de fact_rec('+str(n)+')')
        f=facto_rec(n-1)
        p = n*f
        print(n*'--'+' fin de fact_rec('+str(n)+') :',end=' ')
        print('calcule et renvoie p='+str(n)+'x'+str(f)+'='+str(p))
        return p
```

Ainsi on obtient le résultat suivant lors de l'appel de `facto_rec(5)` :

```
>>> facto_rec(5)
---------- appel de fact_rec(5)
-------- appel de fact_rec(4)
------ appel de fact_rec(3)
---- appel de fact_rec(2)
-- appel de fact_rec(1)

appel de fact_rec(0) : renvoie 1

-- fin de fact_rec(1) : calcule et renvoie p=1x1=1
---- fin de fact_rec(2) : calcule et renvoie p=2x1=2
------ fin de fact_rec(3) : calcule et renvoie p=3x2=6
-------- fin de fact_rec(4) : calcule et renvoie p=4x6=24
---------- fin de fact_rec(5) : calcule et renvoie p=5x24=120
120
```

On se rend alors compte que tant que `facto_rec(0)` n'est pas appelé, aucun calcul n'est effectué. Et ensuite, une fois l'appel de `facto_rec(0)` effectué, les résolutions de chaque appel de `facto_rec(k)` se font dans l'ordre inverse des appels initiaux.
Métaphoriquement, le "fonctionnement" est celui d'une pile d'assiettes que l'on doit laver : on empile les assiettes sales (la première arrivée est en dessous, la dernière au dessus) puis lorsqu'on commence le nettoyage, on lave donc en premier la dernière arrivée, etc... et on lave en dernier la première arrivée !

Ainsi, pour qu'un langage puisse réaliser l'implémentation de la récursivité, on a besoin d'utiliser une structure de données particulière qui puisse "empiler" et "dépiler" les appels successifs d'une fonction récursive : il s'agit de la structure de ... *pile* ! Comme la pile d'assiettes donc !
Nous ne nous attarderons pas sur cette les détails de cette structure mais il s'agit ici d'un des principaux désavantage de la programmation récursive par rapport à la programmation itérative : la pile garde en mémoire les appels, il faut donc stocker cette pile... qui peut prendre beaucoup de place !
Par exemple, Python limite par défaut la pile à 1000 appels récursifs :

```
>>>facto_rec(1000)
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Et ceci afin d'éviter une saturation de la mémoire vive.
On peut bien-sûr changer la limite imposée par défaut en utilisant le module `sys` :

```
>>>import sys
>>>sys.setrecursionlimit(5000) # pile limitée à 5000 appels
>>>facto_rec(1026)
569839580837529726174784496312677583521933422575475183889041052016399844018696650531595
    577159668500701231196285408984960987258166443221754583110658932040867726331471083 4
    624305697976824825437733244466961565735380528793313280499074574343649054690397666 9
    388637057510228419694111194928756246646999762285038999206549882862153198393276999 5
    720694360472005634855525111499712718404462615279895084631053970724993037911141630 7
    679360933445715402975220286469018758502811367036733033522409128102373266320675841 6
    550764619012830494798090616064517584767164898979791895136003949716341773341759155 9
    382679738324973912777643131878391905194575264092769415388780953918613570243143711 6
    642677457860352459469311671652197042308234195076670757539996057261842880527214620 7
    693545218436159789623096408868848150564660232143405367854073009557675400213033644 7
    661078006809509858564494512524716943840076895974478933745829423139190503710121348 5
    225075174463814730235979483830160011666169603102053261100846513696272700424124173 7
    498286501071800017517493450478665267908891313779690456831057394531709655967569434 7
    849186188594503570230357756606584374742204997222829979666830425562187944396677685 0
    970349597068905843053611308103640560297604230186981295225915341371572793193975185 6
    565058309439494983412658925071328836595087912649467119832515775851340044962538386 7
    285198780334774763224567440332490888710717338603780562394309456378484808717411959 8
    251917413529196349257750925381131358803122006865476907927724278888483043230457227 6
    119961585179132339822315132009290421068610130093664850870627908695015923552455127 0
    955782716714820428932635209669583127379431410048874820646678000861740916553651899 4
    934501572404604551439360736268830241406757421001798303656590812733185107018730556 7
    415164329561459974767497904936171581162505744478211317376109588427554322849228311 5
    964765782322977406642758508028040499481622124924657230472628579836989452302285776 5
    924173586792576520765504199142985043677376387042758307818121101899102109851920226 3
    835046953787865734617373735730511113688676709580675845789500329736771781470619195 5
    685516380659585541673291216234505734968338954373890603493659219679444105339371807 1
    320192933574838850569150790213791133778072914137417055096831228996334583586739707 4
    374012646802642582506659724766172522613233374970119100460809716155223456179239716 1
    436242561519173110333257075198509964941412944516898005023622461864281540887990448 5
    244141568000000000000000000000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    000000000000000000
```

2. TP : Récursivité

Dans le T.P. suivant, nous allons nous exercer à programmer récursivement :

TP n°5 - Récursivité

Partie D

Algorithmes dichotomiques

Dans cette partie, nous allons étudier des algorithmes **dichothomique** : le principe de ces algorithmes repose sur la "devise" : *diviser pour régner* qui est un principe de base en algorithmie. L'idée de résolution d'un problème par un algorithme dichotomique est de couper le problème en deux sous-problèmes que l'on résout de nouveau par ce même algorithme ! Voici quelques exemples de ce principe qui semble de prime abord déroutant :

1. Recherche dichotomique

Dans la première partie de ce chapitre et les T.P. associés, nous avons étudié et programmé un algotithme de recherche d'un élément dans un tableau dont les données n'avaient pas de propriété particulière. Nous avons vu que dans le pire des cas - l'élément recherché n'est pas dans le tableau - l'algorithme faisait autant de comparaison que la taille du tableau ce qu'on a traduit par une complexité temporelle linéaire de l'algorithme.
En supposant une propriété supplémentaire sur les données du tableau, nous allons voir qu'on peut diminuer drastiquement le nombre de comparaison et donc la complexité dans un nouvel algorithme de recherche : la recherche dichotomique.

Principe de la recherche dichotomique :
On considère un tableau `L` que l'on suppose **triée** on suppose donc que ses éléments appartiennent à un ensemble muni d'un (pré-)ordre total et on considère un élément `x` que l'on se propose de chercher dans `L`.

- Si `L` est vide, `x` n'est pas dans `L`.
- Si `L` n'est pas vide, on "coupe" le tableau `L` en deux en son milieu (ou presque) et on considère l'élément `l` à cet endroit :
  - si `l` vaut `x`, alors `x` est dans `L` ;
  - si `l` est strictement plus petit que `x`, alors, `L` étant triée, `x` est potentiellement dans la moitié droite de `L` ;
  - si `l` est strictement plus grand que `x`, alors, `L` étant triée, `x` est potentiellement dans la moitié gauche de `L` ;
- Dans les deux derniers cas, on répète l'algorithme sur la moitié qui contient potentiellement `x`.

Décrite de la sorte, on "sent" que la recherche dichotomique se prête parfaitement au principe récursif. Dans le T.P. qui suivra, nous programmerons l'algorithme tout de même de manière itérative et récurive pour comparer ces deux philosophies.

2. Exponentiation rapide

Voici un deuxième exemple d'algorithme dichotomique : l'algorithme d'**exponentiation rapide** qui permet de calculer un nombre à une puissance entière. Son principe est le suivant :

Principe de l'exponentiation rapide :
On considère un nombre x et un entier naturel n.

- Si n vaut 0, on renvoie 1.
- Si $n \geq 1$:
  - si n est pair, on a $x^n = \left(x^{\frac{n}{2}}\right)^2$;
  - si n est impair, on a $x^n = x\left(x^{\frac{n-1}{2}}\right)^2$;
- puis on répète l'algorithme pour le calcul de $x^{\frac{n}{2}}$.

Comme pour la recherche dichotomique, la description précédente de l'exponentiation rapide est récursive ; mais nous la programmation selon les principes itératif et récursif dans le T.P. suivant.

3. TP : Algorithmes dichotomiques

Dans le T.P. suivant, nous allons nous programmer la recherche dichotomique et l'exponentiation rapide :

TP n°6 - Algorithmes dichotomiques

Partie E

Matrices de pixels et images

Dans cette partie nous allons manipuler les tableaux bi-dimentionnel, plus communément appelés *matrices* en Mathématiques. En informatique, notamment dans sa partie dédiée au traitement d'images, les matrices sont très utiles pour modéliser une image numérique : chaque coefficient de la matrice encode un pixel de l'image.

Dans un premier temps, nous allons, grâce à la classe `list` de Python, créer notre propres matrices sous forme de listes de listes et quelques fonctions qui leur sont dédiées.
Nous verrons ensuite la modélisation des matrices de la classe `array` de Numpy.

1. Les matrices ou tableaux multi-dimensionnels

a. Création et manipulation d'une matrice

Pour la modélisation matricielle en Python, nous allons considérer une matrice `M` comme une liste de listes toutes de même taille :

```
#création d'une matrice de taille 2x3
M = [
[1,2,3],
[3,4,5]
]
```

Il faut bien garder à l'esprit que nous modélisons les matrices : les sous-listes de `M` correspondent donc à ses lignes ; elles **doivent** donc toutes avoir **la même taille** (qui correspond donc au nombre de colonnes).

Grâce à ce que nous connaissons déjà sur le comportement et la syntaxe des listes, on peut accéder aux éléments de la matrice `M`, connaître son nombre de lignes ou de colonnes. **Attention !** il ne faut pas oublier que le premier élément d'un liste en Python est d'**indice 0** !

```
>>> M[0][1] # valeur du coefficient de la 1ère ligne et 2ème colonne de M
2

>>> len(M) # nombre de lignes
2

>>> len(M[0]) # nombre d'éléments de la 1ère ligne et donc nombre de colonnes de M
3

>>> M[1] #retourne la deuxième ligne de la matrice
[3,4,5]
```

Attention au comportement des listes !

On rappelle que le type `list` en Python a un comportement spécifique concernant les copies : étant donnée une liste `L`, il faut utiliser la fonction `copy` ou même `deepcopy` du module `copy` si `L` est une liste de listes pour créer une copie indépendante de `L`.

```
L=[[1,2],[3,4]]
K=L #crée non pas une copie mais un alias de L
K[0][1]=6 #modifie K et L !!!
```

Impact sur L

```
>>> L[0][1]
6
```

```
from copy import deepcopy
L=[[1,2],[3,4]]
J=deepcopy(L) #crée un copie de L indépendante de L
J[0][1]=6 #modifie J mais pas L
```

Pas d'impact sur L

```
>>> L[0][1]
2
```

Attention également à l'utilisation de la syntaxe `[[???]]*n` pour créer une liste de listes toutes identiques. On aurait par exemple envie, pour créer une matrice `M` de taille 4×5 dont toutes les lignes sont remplies de zéros, d'écrire :

```
>>> M=[[0]*5]*4
```

Au premier abord, cela semble créer ce qu'on attend :

```
>>> M
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

Mais NON ! la syntaxe précédente présente le défaut de créer des alias de listes (comme dans l'exemple précédent de `K` par rapport à `L`) et pas des copies indépendantes. Voyons cela :

```
>>> M[0][4]=3
>>> M
[[0, 0, 0, 0, 3], [0, 0, 0, 0, 3], [0, 0, 0, 0, 3], [0, 0, 0, 0, 3]]
```

Aïe, comme chacune des sous-listes sont des alias, elles sont toutes modifiées lorsqu'on en modifie une !

Pour pallier ce problème, on utilisera la syntaxe `[[0]*5 for k in range(5)]` pour créer une liste de `5` copies indépendantes de `[0]*4` (car elle est recalculée `5` fois et pas simplement dupliquée `5` fois comme avec la syntaxe précédente).

2. Les array Numpy

Nous avons vu comment modéliser les matrices mathématiques grâce aux listes (de type `list`) natives de Python.
Les "`list`" de Python sont très pratiques : elle peuvent contenir n'importe quel type d'objets et on peut leur ajouter/supprimer des éléments très facilement.
En contrepartie, le temps d'accès aux éléments n'est pas le plus efficace possible.
Or, on remarque que nos modélisations de matrices n'ont pas forcément besoin de toutes les propriétés offertes par le type `list`, en effet :

- tous les éléments d'une matrice ont la même nature ;
- une matrice a une taille "constante" ;

Ainsi, afin de gagner en rapidité lors de l'accès aux éléments, il est plus pertinent de modéliser une matrice par une structure en tableau (multidimensionnel) qui a l'avantage de permettre l'accès à ses éléments à coût constant, mais qui ne peut contenir que des éléments de même nature et qui est de taille fixée à la création (ce qui ne nous gênera pas ici !).

Pour modéliser nos matrices, nous allons utiliser le module `numpy` qui propose un nouveau type d'objet, le type `ndarray`, que nous appellerons plus simplement "*array*" (tableau en anglais).

Tout d'abord, importons le module `numpy` avec son surnom "habituel" !

```
1 import numpy as np
```

a. Création d'un array

Pour créer un array, le module `numpy` fournit la fonction `array(M)` qui prend un argument obligatoire `M` qui est une liste de listes où toutes les sous-listes de `M` ont la même taille et contiennent des éléments de même type (en tenant compte des conversions implicites de Python, par ex : `int` vers `float` vers `complex`).

Voici quelques exemples de création d'array :

```
1 M=[
2 [1,2,3],
3 [4,5,6]
4 ] #on crée une "matrice" M de taille 2x3
```

```
A=np.array(M) #on la convertit en array numpy

B=np.array([[2,1],[1.2,np.pi],[0,0]]) #on crée directement un array en lui donnant une
     matrice explicite
#on note que puisque certains coeeficients sont de type float, les coefficients de
     type int sont convertis en float lors de la création de l'array
```

La fonction `array(M,dtype)` prend également un argument facultatif `dtype` qui permet de préciser le type des données que l'on souhaite mettre dans l'array. Par exemple :

```
M=[
[1,2,3],
[4,5,6]
] #on crée une "matrice" M de taille 2x3
A=np.array(M,dtype=float) #les élements de A seront convertis en float même s'ils
     étaient tous de type int au départ.
```

Valeurs possibles de `dtype`

- **`bool`** (True/False ou 1/0) ;
- **`int`** ;
- **`float`** ;
- **`complex`** ou plus intéressant encore :
- **`(u)int8`**/**`(u)int16`**/**`(u)int32`**/**`(u)int64`**=**`int`** qui permettent de préciser sur combien de bits (8, 16 ,32 ou 64 seront codés les entiers relatifs (naturels en rajoutant le u) de l'array. Cela nous permettra notamment, lorsque nous manipulerons des images, en niveau de gris par exemple, de les rendre plus "légères" en mémoire en utilisant `dtype='uint8'` (entiers de 0 à 255).
  Attention, ces `dtype` sont à préciser sous forme de chaînes de caractères (puisque que ce ne sont pas des **`type`** Python comme les précédents) ; on indiquera alors, par exemple : `dtype='uint8'` ou encore `dtype='int64'`.

Quelques fonctions permettant de créer des array particuliers

- `zeros((n,m),dtype=type)` renvoie un array de `n` lignes, `m` colonnes rempli de 0 (convertis dans le type indiqué).
  Remarque : l'argument obligatoire est un **`tuple`** donc les parenthèses sont nécessaires ; on peut également créer des array remplis de zéros uni/tri/quadri/etc-dimensionnels avec un **`tuple`** adéquat. Par exemple, on pourra tester les instrictions suivantes : `>>> np.zeros((2,2))`, `>>> np.zeros((9,),dtype=int)`, `>>> np.zeros((2,5,3))`.
- `ones(tuple,dtype=type)` se comporte comme `zeros` mais renvoie un array remplis de 1.
- `eye(n)` renvoie la matrice identité de taille `nxn`
- `diag(L)` renvoie la matrice diagonale de taille **`len(L)`** `x` **`len(L)`** dont les coefficients diagonaux sont les éléments de `L` (si ses éléments sont de même type bien-sûr).

b. Manipulation d'un array

Pour récupérer des éléments d'un array, tout ce que l'on connaît sur les listes s'applique, par exemple :

```
>>> A=np.array([
[1,2,5],
[3,4,0],
],dtype=float) #on crée un array A de flottants

>>> A[0] #renvoyer la première ligne
[1.,2.,5.]
>>> A[1][2] #coefficients de 2ème ligne, 3ème colonne
0.
```

Mais on a également quelques raccourcis syntaxiques bien pratiques par rapport au listes simples :
Si `A` est un array à 2 dimensions :

— `A[i,j]` renvoie l'élément de $i+1$ème ligne et $j+1$ème colonne (équivalent `A[i][j]`)

```
>>> A[1,2] #équivalent à A[1][2]
0.
```

— Le slicing des listes est généralisé aux array, notamment pour les colonnes :
`A[:,j]` renvoie le $j+1$ème colonne de `A` (ce qui était impossible à faire aussi simplement avec une matrice modélisée par une liste de listes)

```
>>> A[:,1]
array([2.,4.])
```

c. Opérations sur les array

Dans la partie précédente, nous avons codé plusieurs opérations matriciels pour notre modélisation en liste de listes. Grâce aux array, la syntaxe de ces opérations est naturelle et simple :
On considère `A,B` des array de mêmes dimensions et `c` un nombre (int, float ou complex).

— `A+B` renvoie l'array somme coefficient par coefficient de `A` et `B`

— `A*B` renvoie l'array produit coefficient par coefficient de `A` et `B`
Attention ! Ce N'est PAS le produit matriciel et l'instruction `A**n` renvoie `A*...*A` (`n` termes)

— `c*A` renvoie l'array des coefficients de `A` tous multipliés par `c`

Si les array `A,B` sont **bi-dimensionnels** et donc modélisent des matrices, on a les outils suivants :

— `dot(A,B)` renvoie l'array correspondant au **produit matriciel** de `A` par `B` s'il est possible ; de plus `matrix_power(A,n)` renvoie A^n.

— `A.T` renvoie l'array correspondant à **la transposée** de `A`

Toujours pour des matrices modélisées par des array, on a accès au déterminant et à l'inverse (si elle existe) à partir d'un sous-module de `numpy` : la bibliothèque `numpy.linalg`
Je vous invite à aller faire un tour sur la page de la sous-module numpy.linalg pour connaître toutes ses spécificités.

```
1 import numpy.linalg as lin
```

— `det(A)` renvoie le **déterminant** de `A`

— `inv(A)` renvoie l'array correspondant à **l'inverse** de `A` si elle existe

3. Matrices de pixels

Dans cette partie, nous allons utiliser des array pour modéliser des images ! Plus précisément, des images numériques... Nous avons l'habitude de regarder des images sur nos écrans d'ordinateur, télévision, téléphone, etc... Tout ces écrans sont décomposés en pixel (mot valise anglais pour "picture element") qui peuvent chacun prendre une couleur différente des autres. Ainsi, pour afficher quelque chose sur un tel écran, il suffit de donner la liste des couleurs de chaque pixel. Un écran étant rectangulaire, on a donc besoin d'un tableau bi-dimensionnel dont chaque élément codera la couleur d'un pixel de l'écran.
Ainsi, les matrices (et donc numériquement, les array) sont de parfaits outils pour modéliser une image numérique !

a. Charger et Enregistrer une image

Pour Python, plusieurs modules permettent de traiter les images. Ici, nous allons utiliser les modules `matplotlib.pyplot` et `imageio`. Ce dernier module n'est pas installé de base sur EduPython. Voici comment l'installer dans cet environnement :

> *Outils > Outils > Installation d'un nouveau module*
> Puis, après avoir sélectionné l'installateur *pip* ou *conda*, on indique le nom du module à installer (ici *imageio*).

Remarque : nous aurions également pu utiliser le module `PIL` qui permet essentiellement de faire la même chose qu'`imageio` et qui est déjà préinstallé sur EduPython mais comme ça, nous avons vu comment installer un module !
Et aussi, le module `matplotlib.pyplot` possède également des fonctions de lecture (`imread`) et écriture (`imsave`) d'images, mais contrairement à `imageio` et `PIL`, l'écriture ne gère pas les images en niveau de gris, seulement les images en couleur : si on ne souhaite manipuler que des images en couleur, on peut donc se contenter du module `matplotlib.pyplot` seul.

Pour la suite du cours et pour les TP, nous allons écrire des fonctions pour nous simplifier la vie dans la suite.

Le module `imageio` permet de faire de nombreuses opérations sur les images, mais nous allons principalement l'utiliser pour transformer une image numérique en array et réciproquement convertir un array en image. Pour cela, nous allons utiliser les fonctions `imread(chemin)` et `imsave(chemin, array)`.
Le paramètre `chemin` de ces deux fonctions est une chaîne de caractères qui doit contenir le chemin absolu de l'image à charger/enregistrer.
Pour nous éviter de faire des copier/coller permanents du chemin du dossier vers lequel charger/enregistrer nos images, nous allons créer des fonctions "raccourcis" pour n'avoir à indiquer que le nom de l'image et son extension (le désavantage sera qu'on n'utilisera qu'un seul et même dossier pour nos images, ce qui ne nous dérangera pas outre mesure ce que nous ferons dans ce cours) :

```python
import imageio as io

#Chemin absolu du dossier
dossier="C:/User/Images/" #Ceci est un exemple, à vous de remplacer par le chemin d'un
      dossier sur votre ordinateur !

def charger(nom_point_extension):
    """ renvoie l'array correspondant à l'image nommé nom_point_extension dans le
        dossier "dossier" """
    return io.imread(dossier+nom_point_extension)

def sauver(nom_point_extension, A):
    """ Convertit l'array A en image sous le nom nom_point_extension dans le dossier "
        dossier" """
    io.imsave(dossier+nom_point_extension, A)
```

Pour visualiser rapidement les array que nous allons manipuler et créer sous forme d'image sans les enregistrer en "dur", nous allons utiliser `plt.imshow(A)` du module `matplotlib.pyplot` (avec une petite personnalisation pour gérer les images en niveau de gris) :

```python
import matplotlib.pyplot as plt

def voir(A):
    """ Affiche l'array A sous forme d'image """
    if len(A.shape) <= 2: #image ou ligne en niveau de gris
        plt.imshow(A, cmap='gray', vmin=0, vmax=255)
    else: #image en couleur
        plt.imshow(A)
    plt.show()
```

b. Représentation d'une image numérique en niveau de gris

Pour coder une image en niveau de gris, chaque pixel est encodé par une nombre compris entre 0 et 255 qui désigne le niveau de gris du pixel : 0 correspond au noir et 255 correspond au blanc et entre les deux... des niveaux de gris. Comme on a 256 valeurs, on pourra utiliser le `dtype='uint8'` (voir la partie sur les array) pour les array représentants nos images en niveaux de gris.

Créons une première image en niveau de gris : un dégradé de noir vers du blanc.
On commence par définir un array de la taille correspondant au nombre de pixels souhaités en hauteur par longueur : disons 300 pixels par 400.

```python
img = np.zeros((300,400),dtype='uint8')
```

Puis on passe de 0 à 255 de manière "continue" de la première ligne à la dernière : on peut utiliser la fonction linéaire $f : i \mapsto 255\frac{i}{299}$ - où i désigne un indice de ligne qui va de 0 à 299. Mais comme un niveau de gris est un entier, on doit prendre la partie entière des nombres obtenus : la fonction `int` permet de le faire.

```
for i in range(300):
    img[i,:]=[int(255*i/299) for j in range(400)]
```

Ensuite, on peut observer notre image ou la sauvegarder sur le disque dur

```
>>> voir(img)
>>> sauver('degrade.jpg', img)
```

Voilà le résultat :

c. Représentation d'une image numérique en couleur (RVB)

Pour une image en couleurs, il y a différentes façons de coder chaque pixels. Nous verrons ici la méthode RVB=Rouge, Vert, Bleu (RGB en anglais) où chacune de ces trois couleurs possède 256 niveaux de profondeur entre 0 et 255 : 0 désignant l'absence de la couleur et 255 le niveau maximum de la couleur. Ainsi, un pixel sera encodé par un array `[R,V,B]` avec `R,V,B` chacun entre 0 et 255.
Sans plus attendre, créons une première image en couleurs : le drapeau français (sûrement avec des nuances approximatives !) de taille 200 pixels par 300.

Pour ce faire, on crée un array bi-dimensionnel 200 par 300 de triplets `R,V,B`... ce qui revient à faire un array tri-dimensionnel 200 par 300 par 3 !

```
img_couleur = np.zeros((200,300,3),dtype='uint8')
```

On colore donc trois bandes de 100 pixels chacune : la première en bleu : `[0,0,255]` ; le seconde en blanc : `[255,255,255]` ; puis la dernière en rouge : `[255,0,0]`.

```
for j in range(100):
    img_couleur[:,j]=[[0,0,255] for i in range(200)]
for j in range(100,200):
    img_couleur[:,j]=[[255,255,255] for i in range(200)]
for j in range(200,300):
```

```
6     img_couleur[:,j]=[[255,0,0] for i in range(200)]
```

Résultat (avec les instructions `voir(img_couleur)` ou `sauver('drapeau.jpg', img_couleur)`) :