1 Représenter un tableau

1.1 À propos des tableaux

En informatique, un tableau désigne un ensemble ordonné d'informations, le plus souvent de même type. Nous avons déjà eu l'occasion de manipuler des tableau à une dimension, en utilisant les listes de Python pour les représenter.

Par exemple, le tableau :

14	42	17	54	37	11

était représentable par la liste

```
In [1]: T = [ 14, 42, 17, 54, 37, 11 ]
```

Python permet un accès facile et direct à n'importe laquelle de ces données, ce qui convient pour représenter un tableau.

Cette représentation reste toutefois un peu inefficace en terme de performances (le fait que les listes puissent contenir n'importe quoi et voire leur taille varier empêche certaines optimisations), mais surtout limitée lorsque l'on essaie de travailler avec des tableaux plus complexes, par exemple à plusieurs dimensions.

1.2 Comment représenter un tableau à deux dimensions ?

Supposons que l'on souhaite manipuler des données contenues dans un tableau à deux dimensions, tel que celui-ci :

14	42	17
54	37	11

Python (comme la plupart des langages de programmation) ne dispose pas de structures de données destinées spécifiquement à stocker un tableau à plusieurs dimensions. Fort heureusement, il reste néanmoins possible de manipuler un tel tableau en Python.

La solution la plus simple consiste à le définir comme une *liste de lignes*, chaque ligne étant elle-même une liste. Ainsi, on écrira

```
In [2]: T = [ [ 14, 42, 17 ], [ 54, 37, 11 ] ]

In [3]: T
Out[3]: [[14, 42, 17], [54, 37, 11]]
```

Pour obtenir une ligne donnée, par exemple la seconde ligne (la ligne d'index 1), on écrira :

```
In [4]: T[1]
Out[4]: [54, 37, 11]
```

Et donc, pour accéder à l'élément situé sur la seconde ligne, à la première colonne, on utilise

```
In [5]: T[1][0]
Out[5]: 54
```

Ce n'est pas l'unique façon de représenter un tableau à deux dimensions en utilisant des listes (on pourrait par exemple le mémoriser sous la forme d'une liste de colonnes), mais c'est la plus couramment utilisée.

Ces solutions ne sont cependant pas sans inconvénients. Par exemple, rien ne dit que la liste représentant le tableau ne contient que des listes (représentant chacune des lignes). Ni que les lignes ont toutes une même longueur.

Un module Python, appelé `numpy`, propose un type particulier, appelé `numpy.array`, destiné à manipuler des tableaux.

L'importation du module se fait de la façon habituelle :

```
In [6]: import numpy
```

Il est très simple de convertir une liste de listes, telle que définie précédemment, en un `numpy.array` :

```
In [7]: A = numpy.array(T)

In [8]: type(A)
Out[8]: numpy.ndarray

In [9]: A
Out[9]:
array([[14, 42, 17],
       [54, 37, 11]])
```

Premier petit avantage, l'affichage fait apparaître clairement la structure bi-dimensionnelle du tableau, les colonnes étant ici convenablement alignées.

Nous avons ainsi créé un `numpy.array` à deux dimensions, mais il est tout aussi possible de créer un tableau à une seule dimension, ou bien trois dimensions et plus !

Nous verrons que ces `numpy.array` ont beaucoup de choses en commun avec les listes. Il y a toutefois trois différences majeures entre les `numpy.array` et des représentations sous forme de listes :

- tous les éléments dans un `numpy.array` sont de même type (entier, flottant, booléen...) ;
- la taille du tableau est constante (il n'est pas possible d'insérer ou de supprimer un élément comme pour une liste [1]) ;
- dans un `numpy.array` à plusieurs dimensions, toutes les lignes (ou colonnes) ont nécessairement la même longueur.

Ces contraintes supplémentaires sur le contenu permettent à Python de travailler plus efficacement (et donc plus rapidement) avec de tels tableaux, ce qui peut être très appréciable dans le cadre du calcul numérique.

1.3 Accéder aux éléments d'un `numpy.array`

Pour accéder à un élément, on peut utiliser exactement les mêmes outils que pour une liste de listes. Par exemple, pour notre `numpy.array` `A` à deux dimensions, utiliser un indice entre crochets permet d'obtenir la ligne correspondante (sous la forme d'un `numpy.array` à une seule dimension) :

```
In [10]: A[1]
Out[10]: array([54, 37, 11])
```

Et deux indices de suite d'accéder directement à un élément précis :

```
In [11]: A[1][0]
Out[11]: 54
```

Mais on peut aussi utiliser une notation un peu différente, indiquant directement les coordonnées de la case qui nous intéresse [2] :

```
In [12]: A[1,0]
Out[12]: 54
```

De même que l'on pouvait effectuer des « slices » sur des listes, on peut se servir de cette notation pour obtenir des morceaux de tableaux.

Par exemple, on peut obtenir les deux cases en haut à gauche du tableau (soit la ligne d'indice 0 et les colonnes d'indices 0 à 1) avec un découpage :

```
In [13]: A[0,0:2]
Out[13]: array([14, 42])
```

1. Il existe des solutions pour contourner ce problème, mais les algorithmes travaillant avec des tableaux devraient éviter d'avoir à changer les dimensions du tableau fréquemment.
2. En fait, en plaçant un « tuple » entre les crochets.

Ou bien encore l'intégralité de la deuxième colonne :

```
In [14]: A[0:2,1]
Out[14]: array([42, 37])
```

Il est possible, comme pour une liste, d'omettre les bornes du « slice » lorsqu'il s'agit des extrémités, aussi est-il possible d'obtenir la deuxième colonne simplement en écrivant :

```
In [15]: A[:,1]
Out[15]: array([42, 37])
```

On peut également utiliser un pas, et donc obtenir le sous-tableau constitué des seules colonnes impaires par exemple :

```
In [16]: A[:,::2]
Out[16]:
array([[14, 17],
       [54, 11]])
```

Attention, cela ne fonctionne *qu'avec cette notation*, comme on peut le voir ici :

```
In [17]: A[:][1]
Out[17]: array([54, 37, 11])
```

On n'obtient pas, en effet, la seconde colonne mais la *seconde ligne* ici, car `A[:]` désigne le tableau constitué de l'ensemble des lignes de `A`, soit exactement `A`! `A[:][1]` est donc équivalent à `A[1]`, soit la seconde ligne de `A`.

La prudence est de mise, les `numpy.array` acceptent beaucoup de choses comme coordonnées (listes, ranges, etc.), pour laisser un maximum de souplesse quand il s'agit de d'extraire des morceaux d'un `numpy.array`, mais le résultat ne sera pas forcément toujours celui auquel on peut s'attendre. Nous n'énumérerons pas ici toutes possibilités offertes.

1.4 Dimensions d'un `numpy.array`

Il peut être utile de connaître les dimensions d'un tableau. On peut, pour ce faire, utiliser la fonction `len` qui, pour un `numpy.array`, retourne, comme pour une liste de listes, le nombre de lignes du tableaux. Pour connaître le nombre de colonnes, on peut demander la longueur d'une ligne quelconque, par exemple la première :

```
In [18]: len(A)
Out[18]: 2

In [19]: len(A[0])
Out[19]: 3
```

On peut aussi faire suivre le nom désignant le tableau de « `.shape` », sans parenthèses, ce qui fournit l'ensemble des dimensions du tableau[3] (deux dans le cas d'un tableau bidimentionnel) :

```
In [20]: A.shape
Out[20]: (2, 3)
```

Il peut être utile dans un algorithme de mémoriser le nombre de lignes et de colonnes d'un tableau à deux dimensions, par exemple de la sorte :

```
In [21]: nblignes, nbcolonnes = A.shape

In [22]: nblignes
Out[22]: 2

In [23]: nbcolonnes
Out[23]: 3
```

Naturellement, `len(A.shape)` donnerait le nombre de dimensions du `numpy.array`.

1.5 Construire des `numpy.array`

Une façon de construire un `numpy.array`, on l'a vu, consiste à convertir une liste de listes. Comme tous les éléments doivent être de même type, Python choisira le type qui lui semble le plus adapté[4], généralement des entiers ou des flottants.

Dans notre exemple, puisque tous les éléments dans les listes de listes étaient des entiers, il a construit un tableau d'entiers. Si ne serait-ce qu'un seul élément avait été un flottant, tout le `numpy.array` construit aurait contenu des flottants.

Une autre façon de construire un `numpy.array` est de construire un tableau ne contenant que des zéros, grâce à la commande `numpy.zeros(taille)` à laquelle on fournit un paramètre décrivant la forme du tableau[5]. La taille du tableau doit être précisée, pour un tableau bidimensionnel, sous la forme d'un couple (hauteur, largeur), de cette façon[6] :

```
In [24]: numpy.zeros( (2, 3) )
Out[24]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

La fonction `numpy.ones` permet d'obtenir la même chose avec des 1 à la place des 0.

La fonction `numpy.identity(taille)` crée quant à elle spécifiquement des tableaux à deux dimensions, toujours « carrés », dont la taille est spécifiée par l'entier passé en argument, avec des 1 sur la diagonale et des 0 partout ailleurs.

```
In [25]: numpy.identity(3)
Out[25]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Ce ne sont là que quelques exemples d'usage courant. De nombreuses fonctions permettent de créer des `numpy.array` particuliers, on se reportera à la documentation en cas de besoin.

Citons tout de même deux fonctions bien utiles, créant des `numpy.array` à une seule dimension.

Tout d'abord, `numpy.arange` crée un `numpy.array` de la même façon que `range` génère un ensemble de valeurs, à ceci près que l'on peut utiliser des flottants :

```
In [26]: numpy.arange(0.5, 4.5, 1.6)
Out[26]: array([ 0.5,  2.1,  3.7])
```

Ce peut être un bon remplacement de `range`, justement, lorsque l'on veut itérer sur des valeurs non-entières :

```
In [27]: for x in numpy.arange(0.5, 4.5, 1.6) :
    ...:     print(x**2)

0.25
4.41
13.69
```

On se méfiera cependant des problèmes éventuels d'arrondis qui peuvent survenir lorsque l'on utilise des flottants, il est possible parfois de se retrouver avec un élément de plus ou de moins que prévu si l'on n'est pas très prudent.

La fonction `numpy.linspace`, enfin, construit un tableau de façon très similaire, mais au lieu d'un pas, on définit le *nombre* de valeurs que l'on souhaite entre les deux bornes[7] (la première et la dernière valeur correspondant exactement aux bornes en question) :

```
In [28]: numpy.linspace(0.5, 4.5, 6)
Out[28]: array([ 0.5,  1.3,  2.1,  2.9,  3.7,  4.5])
```

3. Il n'y a pas de parenthèses après `A.shape` car cela désigne en quelque sorte une « propriété » du tableau. En fait, il est même possible de changer ces dimensions en affectant de nouvelles valeurs à `A.shape`, sous réserve que le nombre total de cases dans le tableau ne change pas.
4. Om peut imposer ce type via l'argument facultatif `dtype`, en indiquant par exemple `dtype='float'`.
5. Et éventuellement un paramètre `dtype` précisant le type des données, `numpy` utilisant des flottants par défaut.
6. Attention aux doubles parenthèses, une première paire car c'est un appel de fonction, une seconde pour signifier qu'il y a un seul argument qui est un couple de valeur, et non deux arguments.
7. Si le troisième argument est omis, le `numpy.array` contiendra 50 valeurs.

2 Travailler avec des numpy.array

2.1 Opérations sur les numpy.array

Supposons que l'on ait deux tableaux de même taille, par exemple :

```
In [29]: T1 = np.array([ [ 1, 2, 3 ], [ 4, 5, 6 ] ])

In [30]: T2 = np.array([ [ 1, 1, 1 ], [ 2, 2, 2 ] ])

In [31]: T1
Out[31]:
array([[1, 2, 3],
       [4, 5, 6]])

In [32]: T2
Out[32]:
array([[1, 1, 1],
       [2, 2, 2]])
```

Il est possible d'utiliser les opérateurs usuels (+, -, *, /, //, %, **, etc.) sur ces tableaux T1 et T2. Les opérations sont effectuées élément par élément :

```
In [33]: T1+T2
Out[33]:
array([[2, 3, 4],
       [6, 7, 8]])

In [34]: T1*T2
Out[34]:
array([[ 1,  2,  3],
       [ 8, 10, 12]])
```

Naturellement, le type du résultat peut différer de celui des opérandes, en fonction des besoins du calcul, comme lorsque l'on divise deux entiers (ce qui donne un résultat flottant) :

```
In [35]: T1/T2
Out[35]:
array([[ 1.,  2., 3.],
       [ 2., 2.5, 3.]])
```

Il n'est en fait pas nécessaire que les deux tableaux soient de même taille. Si ce n'est pas le cas, Python essaiera de faire du *broadcasting*, c'est-à-dire de dupliquer les données « manquantes » de chaque opérande pour obtenir deux tableaux de même taille. Les détails de ce mécanisme sont complexes [8], mais il est deux situations où cela peut être utile.

Tout d'abord, il est possible d'effectuer une opération entre un nombre (entier ou flottant) et un tableau. Ainsi, on peut écrire

```
In [36]: T1 + 3
Out[36]:
array([[4, 5, 6],
       [7, 8, 9]])

In [37]: T1 ** 2
Out[37]:
array([[ 1,  4,  9],
       [16, 25, 36]])
```

Les éléments de T1 dans les deux exemples précédents ont été respectivement incrémentés de 3, et mis au carré. En fait, le broadcasting de numpy transforme la somme T1+3, soit $\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5&6\\\hline\end{array}+3$, en $\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5&6\\\hline\end{array}+\begin{array}{|c|c|c|}\hline 3&3&3\\\hline 3&3&3\\\hline\end{array}$.

Ce mécanisme fonctionne dans les deux sens, et l'on peut donc également écrire

```
In [38]: 2 ** T1
Out[38]:
array([[ 2,  4,  8],
       [16, 32, 64]], dtype=int32)
```

2.2 Fonctions et numpy.array

La seule fonction ne nécessitant pas le chargement d'un module que l'on peut envisager d'utiliser sur des numpy.array, abs, fonctionne comme on s'y attend (cela donne un numpy.array contenant les valeurs absolues ou les modules des éléments du numpy.array passé en argument).

Les fonctions du module math (ou cmath) ne peuvent en revanche pas être utilisées sur des numpy.array [9] :

```
In [39]: math.sin(T1)

TypeError: only length-1 arrays can be converted to Python scalars
```

Pour cette raison, le module numpy propose des variantes des fonctions du module math

8. Il n'est pas indispensable de les connaître, mais ils sont présentés brièvement en fin du présent chapitre.
9. Le message d'erreur, pas nécessairement très clair, laisse toutefois supposer qu'un tableau à un seul élément peut être argument de ces fonctions, et c'est effectivement le cas, le résultat n'étant en revanche pas un numpy.array mais un flottant ou un complexe, généralement.

qui peuvent, elles, fonctionner sur des tableaux :

```
In [40]: numpy.sin(T1)
Out[40]:
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ]])
```

La plupart des fonctions des modules `math` et `cmath` sont disponibles dans le module `numpy`, adaptées à l'utilisation des tableaux (mais elles peuvent tout aussi bien être utilisées « normalement »), et sont compatibles avec les complexes.

Le module `numpy` fournit par ailleurs également des remplacements pour les fonctions `sum`, `min`, `max`, `any` et `all`, car les fonctions de base de Python ne conviennent pas, en général, pour des `numpy.array` :

```
In [41]: sum(T1)
Out[41]: array([5, 7, 9])

In [42]: numpy.sum(T1)
Out[42]: 21

In [43]: max(T1)

ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()

In [44]: numpy.max(T1)
Out[44]: 6
```

Mais attention, il convient d'être très prudent et de ne pas *remplacer* les fonctions de Python en question par les versions fournies par `numpy` sans être sûr de son fait, au risque d'avoir de très mauvaises surprises. En effet, on peut écrire, avec la fonction `max` de Python :

```
In [45]: max( i**2 for i in range(5) )
Out[45]: 16
```

En revanche, avec la fonction `max` de remplacement fournie par `numpy`, on obtient :

```
In [46]: numpy.max( i**2 for i in range(5) )
Out[46]: <generator object <genexpr> at 0x16FFD8A0>
```

Aussi est-il potentiellement dangereux d'utiliser « `from numpy import max` » et, infiniment plus risqué encore d'utiliser « `from numpy import *` » !

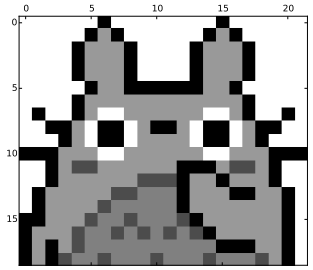
3 Représenter des images

3.1 Images noir et blanc

Dans le domaine du numérique, les photographies et images sont en fait des mosaiques, car il n'est possible de mémoriser qu'un nombre fini d'informations [10]. Elles sont en général constituées d'un agencement régulier de zones de couleur et d'intensité uniforme, toutes rectangulaires et de même taille. On donne à ces éléments le nom de *pixels* (pour *picture element*). Il est donc naturel de représenter une image par un tableau.

Une image en noir et blanc sera donc un simple tableau à deux dimensions, contenant dans chaque case une intensité. Il est courant de représenter cette intensité soit par un nombre flottant (0.0 représentant généralement du noir, 1.0 du blanc), soit par un entier non signé sur p bits (auquel cas 0 représente le noir, et $2^p - 1$ le blanc).

Un exemple d'image noire et blanc et le tableau (contenant des flottants) la représentant :




```
1.0 1.0 1.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 0.0 0.6 0.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0 0.6 0.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.0 1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.0 1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.0 1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.6 0.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 0.0 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.0 1.0 1.0 1.0 1.0
1.0 0.0 1.0 1.0 0.0 0.6 1.0 1.0 0.6 0.6 0.6 0.6 0.6 1.0 1.0 0.6 0.0 1.0 1.0 0.0 1.0
1.0 1.0 0.0 0.0 0.6 1.0 0.0 0.0 1.0 0.6 0.0 0.0 0.6 1.0 0.0 0.0 1.0 0.6 0.0 0.0 1.0
1.0 1.0 1.0 0.0 0.6 1.0 0.0 0.0 1.0 0.6 0.6 0.6 0.6 1.0 0.0 0.0 1.0 0.6 0.0 1.0 1.0
0.0 0.0 0.0 0.6 0.6 0.6 1.0 1.0 0.6 0.6 0.6 0.6 0.6 1.0 1.0 0.6 0.6 0.6 0.0 0.0 0.0
1.0 1.0 0.0 0.6 0.3 0.3 0.6 0.6 0.6 0.6 0.6 0.6 0.0 0.0 0.0 0.6 0.3 0.6 0.0 1.0 1.0
1.0 1.0 0.0 0.6 0.6 0.6 0.6 0.6 0.6 0.3 0.3 0.0 0.6 0.6 0.0 0.6 0.6 0.6 0.0 1.0 1.0
1.0 0.0 0.6 0.6 0.6 0.6 0.6 0.3 0.3 0.5 0.5 0.5 0.0 0.6 0.6 0.6 0.0 0.0 0.6 0.6 0.0 1.0
1.0 0.0 0.6 0.6 0.6 0.6 0.3 0.5 0.5 0.5 0.5 0.5 0.0 0.6 0.6 0.6 0.6 0.6 0.6 0.0 1.0
0.0 0.0 0.6 0.6 0.6 0.3 0.5 0.5 0.3 0.5 0.5 0.5 0.3 0.0 0.6 0.6 0.6 0.6 0.6 0.0 1.0
0.0 0.6 0.6 0.6 0.3 0.5 0.5 0.3 0.5 0.3 0.5 0.3 0.5 0.3 0.0 0.6 0.6 0.6 0.6 0.0 1.0
0.0 0.6 0.0 0.6 0.3 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.0 0.0 0.0 0.6 0.6 0.0 1.0
0.0 0.6 0.0 0.3 0.5 0.5 0.3 0.5 0.5 0.5 0.3 0.5 0.5 0.5 0.3 0.5 0.5 0.5 0.3 0.6 0.0 1.0
```

La plupart des formats de fichiers enregistrent une image sous la forme de tableaux d'entiers (sur 8 bits, généralement). Cependant, lorsque l'on modifie une image, il n'est pas rare d'utiliser des flottants, car ils permettent plus de précision dans les calculs intermédiaires. Lors de l'enregistrement du résultat, une fois les modifications terminées, chaque intensité est simplement « arrondie » à l'intensité la plus proche représentable par le format de fichier choisi.

Le morceau de code suivant construit une image de taille 200 × 200, représentant un disque blanc (de diamètre 160 pixels) sur un fond noir, sous la forme d'un tableau de flottants :

```
In [47]: img = np.zeros( (200, 200) )

In [48]: for l in range(200) :
    ...:     for c in range(200) :
    ...:         if math.sqrt((100-l)**2 + (100-c)**2) < 80.0 :
    ...:             img[l,c] = 1.0
```

10. Les photographies argentiques sont en fait aussi des mosaiques, constituées de grains de chlorure d'argent plus ou moins noircis par la lumière reçue, même si leur répartition n'est pas régulière.

3.2 Images couleur

Pour les images couleurs, ce n'est pas beaucoup plus compliqué. En théorie, il faudrait, pour représenter parfaitement une image en couleur, indiquer, pour chaque pixel, l'intensité de chacune des longueurs d'ondes visibles. Le problème, c'est qu'il y en a une infinité...

Cependant, les cellules photosensibles situés dans la rétine de l'œil humain sont de deux types, appelés batonnets et cônes. Les bâtonnets sont essentiellement sensibles à l'intensité et non à la couleur, et sont surtout présents en nombre au niveau de la vision périphérique. Au centre de la rétine, ce sont les cônes, qui eux sont sensibles à la couleur, qui sont les plus nombreux.

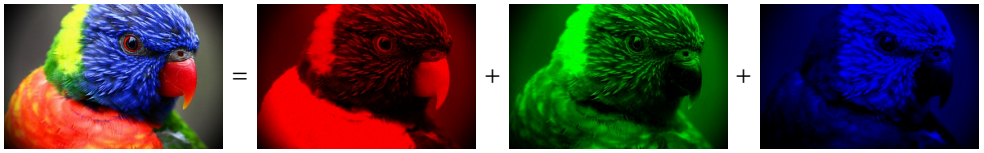
Or on ne trouve sur la rétine que trois variétés de cônes (erytholabes, chlorolabes et cyanolabes), chacun ayant sa propre courbe de sensibilité aux différentes longueurs d'ondes, centrées respectivement sur le rouge, le vert et le bleu [11].

Lorsque la rétine reçoit un rayonnement dont la longueur d'onde correspond au « jaune » du spectre, les cônes erytholabes et chlorolabes réagissent, et le cerveau a pour habitude d'associer cela à la couleur jaune. Cependant, il est incapable de faire la différence entre une longueur d'onde jaune, et un rayonnement mélangeant des longueurs d'onde verte et rouge, puisque cela fait réagir les mêmes cônes.

Cela a une conséquence intéressante : pour « simuler » n'importe quelle couleur que le cerveau peut identifier, il suffit de pouvoir générer un mélange de trois longeurs d'onde, rouge, verte et bleue, associées aux différentes variétés de cônes. On parle de synthèse additive.

C'est le principe utilisé par la quasi-totalité des dispositifs d'affichage, où chaque pixel coloré est en fait une association d'une source émettant de la lumière rouge, d'une source émettant de la lumière verte, et d'une source émettant de la lumière bleue, dans des proportions bien choisies pour produire n'importe quelle couleur.

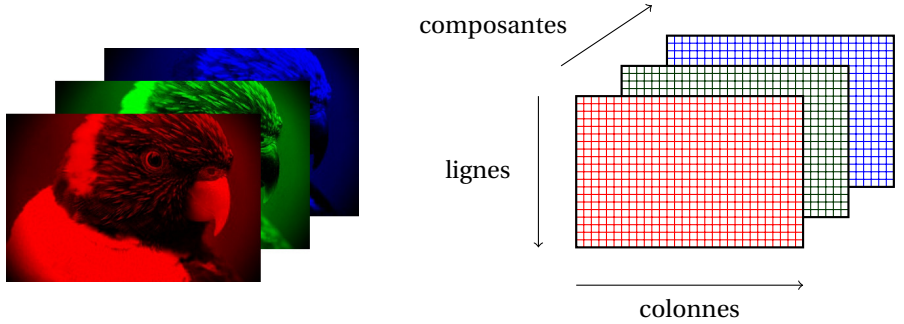
Une image en couleur est donc la combinaison d'une image rouge, d'une image verte, et d'une image bleue. L'exemple ci-dessous illustre la décomposition d'une image en couleur en trois images rouge, verte et bleue.



Le bec rouge est essentiellement visible sur la première des trois images, tandis que le plumage à l'avant du crâne, d'un bleu tirant légèrement sur le vert, apparaît tout naturellement principalement sur la troisième. Le plumage à l'arrière du crâne, de couleur jaune, est obtenue en faisant parvenir à l'œil le mélange de rouge et de vert, et se trouve donc visible sur les deux premières images.

11. Les « défauts » de vision des couleurs, très fréquents, consiste en une sensibilité moindre, ou plus rarement nulle, de l'une des trois variétés de cônes, ou en un déplacement du maximum de sensibilité de l'une d'elles.

Les images en couleurs sont donc enregistrées sur le même principe : pour chaque pixel (chaque « case »), on mémorise *trois* valeurs d'intensité, pour chacune des trois couleurs (ou *composantes*). On travaillera donc généralement avec des tableaux à trois dimensions, de taille `nb_lignes` × `nb_colonnes` × 3.



Ainsi, `img[17, 42, 0]` indiquera par exemple la quantité de rouge dans le pixel à la 18e ligne et la 43e colonne, tandis que `img[17, 42, 1]` correspondra à la quantité de vert et `img[17, 42, 2]` à celle de bleu.

Un tel tableau à trois dimensions peut directement être affiché [12] par `matplotlib.pyplot` grâce à l'instruction `imshow` :

```
In [49]: img.shape
Out[49]: (2304, 3456, 3)

In [50]: matplotlib.pyplot.imshow(img)
Out[50]: <matplotlib.image.AxesImage at 0x332adcf0>

In [51]: matplotlib.pyplot.show()
```

Il existe d'autres systèmes de représentation des couleurs, essentiellement équivalents, qui peuvent être utilisés en vidéo par exemple. Des applications dont l'observation à l'œil n'est pas le but peuvent utiliser plus de trois composantes de couleurs (par exemple en astrophotographie, où l'on peut considérer un grand nombre de longueurs d'ondes, y compris en dehors du visible).

En imprimerie, on procède de la même façon, mais en synthèse soustractive, avec des encres qui absorbent chacune des couleurs élémentaires : le rouge (avec de l'encre cyan), le vert (avec de l'encre magenta) et le bleu (avec de l'encre jaune).

Il peut d'ailleurs arriver que des images destinées à l'imprimerie soient stockées sous la forme de tableau « CMY » où sont mémorisées les quantité de chacune des encres primaires pour chaque pixel (ou souvent « CMYK » où l'on précise aussi la quantité de noir, le mélange des trois autres encres ne donnant généralement pas un noir assez sombre).

12. La fonction `imshow` tentera de « deviner » la façon dont est représentée l'image (flottants entre 0.0 et 1.0, entiers entre 0 et 255...), aussi peut-il parfois y avoir quelques difficultés si l'on se retrouve par exemple avec des flottants entre 0.0 et 255.0 suite à un calcul qui a changé le type des arguments.

4 Représenter des vecteurs, matrices et tenseurs

4.1 Création

Matrices, vecteurs et autres tenseurs sont d'autres objets naturellement candidats à la représentation par des `numpy.array`. Ainsi, on peut représenter les éléments

$$M_1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad M_2 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad V_L = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \qquad V_C = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

par les `numpy.array` suivants :

```
In [52]: M1 = numpy.array([ [1, 2], [3, 4], [5, 6] ])

In [53]: M2 = numpy.array([ [1, 2, 3], [4, 5, 6] ])

In [54]: VL = numpy.array([ [1, 2, 3] ])

In [55]: VC = numpy.array([ [1], [2], [3]])
```

On peut utiliser toutes les opérations déjà vues sur les `numpy.array`. Ainsi, la norme d'un vecteur peut être simplement calculée par

```
In [56]: sqrt( numpy.sum( VC**2 ) )
Out[56]: 3.7416573867739413
```

On rappelle que le `**2` a pour effet, dans l'expression précédente, d'élever au carré *un à un chacun des termes* du vecteur. Peu importe ici que l'on utilise la fonction `sqrt` du module `math` ou du module `numpy`, car on travaille sur un flottant.

4.2 Produit matriciel

Il manque toutefois un outil important pour travailler avec vecteurs et matrices : le produit ! En effet, l'opérateur `*` réalise un produit terme à terme, différent du produit matriciel. On rappelle que le produit d'une matrice A de taille $n \times p$ et d'une matrice B de taille $p \times q$ est la matrice C de taille $n \times q$ dont les coefficients s'écrivent [13]

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} \times B_{k,j}$$

On pourrait bien évidemment envisager de programmer ce produit matriciel en Python. Cela ne présente pas de difficulté particulière, on peut créer une matrice C de la bonne taille et calculer ses coefficiants, la fonction ressemblant à celle-ci [14] [15] :

```
def Multiplie(A, B) :
    """Retourne le produit matriciel des matrices A et B"""

    nb_lgn_A, nb_col_A = A.shape
    nb_lgn_B, nb_col_B = B.shape

    if nb_col_A != nb_lgn_B :
        raise ValueError('Les dimensions ne sont pas compatibles')

    C = numpy.zeros( (nb_lgn_A, nb_col_B) )

    for i in range(0, nb_lgn_A) :
        for j in range(0, nb_col_B) :
            C[i, j] = sum(A[i, k]*B[k, j] for k in range(0, nb_col_A))

    return C
```

Fort heureusement, `numpy` nous fournit déjà une fonction, appelée `numpy.dot`, effectuant ce produit matriciel :

```
In [57]: numpy.dot(M1, M2)
Out[57]:
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])

In [58]: numpy.dot(M2, M1)
Out[58]:
array([[22, 28],
       [49, 64]])
```

Évidemment, les dimensions des deux opérandes doivent être compatibles pour que le produit soit possible :

```
In [59]: numpy.dot(M1, M1)

ValueError: shapes (3,2) and (3,2) not aligned: 2 (dim 1) != 3 (dim 0)
```

13. Expression dans laquelle les indices commencent à 1, comme il est d'usage en mathématiques.

14. Le mot-clé **`raise`** permet de déclencher une erreur (ou plus précisément, dans le language informatique, « *lever une exception* », ici de type arithmétique. Il est recommandé, dans une fonction, de gérer les cas qui pourraient poser problème d'une façon ou d'une autre, mais le mécanisme des exceptions n'est pas à connaître, même si on aura l'occasion d'en aperçevoir de temps à autre.

15. Précisons aussi que la fonction proposée retournera une matrice de flottants, même si les arguments sont des matrices d'entiers. Ce n'est pas idéal, et on peut procéder différemment, mais comme nous n'avons pas vraiment besoin de cette fonction, on ne cherchera pas à l'améliorer.

On peut aussi multiplier matrices et vecteurs, sous réserve que les dimensions soit correctes (donc que les vecteurs soient bien orientés, en ligne ou en colonne) :

```
In [60]: numpy.dot(M2, VC)
Out[60]:
array([[14],
       [32]])

In [61]: numpy.dot(VL, M1)
Out[61]: array([[22, 28]])
```

On peut écrire, de façon équivalente, ces produits matriciels un peu différemment :

```
In [62]: M1.dot(M2)
Out[62]:
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])

In [63]: M2.dot(VC)
Out[63]:
array([[14],
       [32]])
```

Il alors est possible de chaîner ces produits (avec une associativité à gauche) de sorte que l'expression

```
In [64]: M1.dot(M2).dot(VC)
Out[64]:
array([[ 78],
       [170],
       [262]])
```

est une autre manière d'écrire

```
In [65]: numpy.dot(numpy.dot(M1, M2), VC)
Out[65]:
array([[ 78],
       [170],
       [262]])
```

Signalons qu'il est possible de définir un vecteur comme un tableau à une seule dimension, et non un tableau à deux dimension comme pour `VC` et `VL` :

```
V = numpy.array( [1, 2, 3] )
```

Les produits avec une matrice (`numpy.array` à deux dimensions) fonctionnent, car `numpy` « devinera » s'il faut utiliser un vecteur ligne ou un vecteur colonne. Le résultat sera alors également un tableau à une seule dimension, dans ce cas.

```
In [66]: numpy.dot(V, M1)
Out[66]: array([22, 28])

In [67]: numpy.dot(M2, V)
Out[67]: array([14, 32])
```

Ce n'est pas la manière la plus sûre de procéder, car laisser à `numpy` le soin de choisir comment interpréter vos données et le plus sûr moyen d'avoir des problèmes. Par exemple, comment interpréter l'expression suivante ?

```
In [68]: numpy.dot(V, V)
```

En effet, est-ce[16] $\begin{bmatrix}1 & 2 & 3\end{bmatrix} \times \begin{bmatrix}1\\2\\3\end{bmatrix}$ ou $\begin{bmatrix}1\\2\\3\end{bmatrix} \times \begin{bmatrix}1 & 2 & 3\end{bmatrix}$?

4.3 Trace, transposées, sous-matrices

Il existe par ailleurs de nombreuses fonctions destinées au calcul matriciel dans le module `numpy`.

Par exemple, on peut obtenir la trace d'une matrice carrée très simplement :

```
In [69]: M = numpy.array( [ [1, 2], [3, 4] ] )

In [70]: M
Out[70]:
array([[1, 2],
       [3, 4]])

In [71]: M.trace()
Out[71]: 5
```

On peut également extraire la diagonale d'une matrice carrée avec la fonction `numpy.diag` :

```
In [72]: numpy.diag(M)
Out[72]: array([1, 4])
```

16. C'est la première solution, autrement dit la norme au carré du vecteur que l'on obtient, mais ce genre de difficulté rend préférable l'utilisation de tableaux à deux dimensions pour la manipulation de vecteurs.

La transposée d'une matrice ou d'un vecteur s'obtient tout aussi simplement :

```
In [73]: M1
Out[73]:
array([[1, 2],
       [3, 4],
       [5, 6]])

In [74]: M1.transpose()
Out[74]:
array([[1, 3, 5],
       [2, 4, 6]])

In [75]: VC
Out[75]:
array([[1],
       [2],
       [3]])

In [76]: VC.transpose()
Out[76]: array([[1, 2, 3]])
```

On peut également écrire [17]

```
In [77]: M1.T
Out[77]:
array([[1, 3, 5],
       [2, 4, 6]])
```

Attention, la transposée ainsi obtenue est un *nouveau* `numpy.array`, mais il n'est pas indépendant : il partage ses données avec la matrice originale (ce qui permet de ne pas recopier toutes les données en mémoire tant que ce n'est pas indispensable) !

```
In [78]: M3 = M2.transpose()

In [79]: M3[1,0] = 42

In [80]: M2
Out[80]:
array([[ 1,  42,  3],
       [ 4,  5,  6]])
```

La modification de `M3[1,0]` dans l'exemple précédent a modifié également le terme `M2[0,1]` de `M2`. Il faudra une *copie* pour les dissocier complètement.

Ce n'est pas la seule source potentielle de problèmes. Ainsi, demander une sous-matrice, une diagonale ou une simple ligne d'un `numpy.array` fournit bien un *nouveau* `numpy.array` mais dont les données sont également liées :

```
In [81]: M2
Out[81]:
array([[ 1, 42,  3],
       [ 4,  5,  6]])

In [82]: premiere_ligne = M2[0]

In [83]: premiere_ligne
Out[83]: array([ 1, 42,  3])

In [84]: premiere_ligne[1] = 2

In [85]: premiere_ligne
Out[85]: array([1, 2, 3])

In [86]: M2
Out[86]:
array([[1, 2, 3],
       [4, 5, 6]])
```

4.4 Autres fonctions d'algèbre linéaire

Différentes fonctions utiles en algèbre linéaire on été regroupées dans un sous-module de numpy appelé `numpy.linalg`.

Par exemple, on peut obtenir le déterminant d'une matrice carrée, grâce à la fonction `numpy.linalg.det` :

```
In [87]: M = numpy.array( [ [1, 1, 1], [0, 2, 2], [0, 0, 4] ] )

In [88]: M
Out[88]:
array([[1, 1, 1],
       [0, 2, 2],
       [0, 0, 4]])

In [89]: numpy.linalg.det(M)
Out[89]: 7.9999999999999982
```

17. La documentation `numpy` spécifie une différence entre les deux écritures, mais ladite documentation est manifestement erronée pour la version actuelle de `numpy` !

Aux erreurs d'arrondi près, évidemment, comme toujours lorsque l'on travaille avec des flottants (et ce, même si dans ce cas, l'utilisation de la formule la plus générale du déterminant ne causerait pas une telle erreur) !

On peut également inverser la matrice si elle est inversible :

```
In [90]: numpy.linalg.inv(M)
Out[90]:
array([[ 1.  , -0.5 ,  0.  ],
       [ 0.  ,  0.5 , -0.25],
       [ 0.  ,  0.  ,  0.25]])
```

Précisons que la matrice inverse obtenue avec `numpy.linalg.inv` est totalement indépendante de la matrice fournie en argument, contrairement à ce qui se passait pour la transposée.

On peut même obtenir les valeurs propres ainsi qu'une base propre (on se reportera au cours de mathématique pour les détails !) :

```
In [91]: numpy.linalg.eig(M)
Out[91]:
(array([ 1.,  2.,  4.]),
 array([[ 1.        ,  0.70710678,  0.42640143],
        [ 0.        ,  0.70710678,  0.63960215],
        [ 0.        ,  0.        ,  0.63960215]]))
```

Le premier élement retourné par la fonction est un vecteur contenant les différentes valeurs propres (qui peuvent être complexes), le second un `numpy.array` bidimensionnel où chaque colonne est un vecteur propre associé à chacune des valeurs propres.

5 Quelques précisions supplémentaires sur les `numpy.array`

5.1 Introduction

Il est loin d'être indispensable de savoir ce qui suit, mais pour les plus curieux, et ceux qui seraient amenés à travailler, par exemple dans le cadre des TIPE, avec des `numpy.array`, quelques remarques utiles ont été regroupées dans cette dernière section.

5.2 À propos des types

Comme on l'a dit, les `numpy.array` contiennent des éléments qui sont d'un unique type. `numpy` tâche de choisir le type le plus adapté en fonction des données qu'on lui fournit, affichant une préférence pour les flottants sur 64 bits s'il a le choix (ainsi, un `numpy.array` de zéros sera, sauf mention contraire, rempli de flottants).

```
In [92]: M = numpy.zeros( (2,2) )

In [93]: M
Out[93]:
array([[ 0.,  0.],
       [ 0.,  0.]])
```

Il est possible de « forcer » l'utilisation d'un type en précisant un argument supplémentaire, toute donnée introduite dans le tableau étant automatiquement convertie :

```
In [94]: M = numpy.zeros( (2,2), dtype='int' )

In [95]: M
Out[95]:
array([[0, 0],
       [0, 0]])

In [96]: M[0,0] = 3.14

In [97]: M
Out[97]:
array([[3, 0],
       [0, 0]])
```

Les types les plus utiles sont `'float'` (flottants), `'int'` (entiers), `'bool'` (pour des valeurs booléennes) et `'complex'` (nombres complexes, représentés en mémoire par deux flottants, pour les parties réelles et imaginaires)

Pour convertir une matrice vers un type différent, une solution [18] parmi d'autres est d'utiliser la fonction `.astype()` :

```
In [98]: M.astype('float')
Out[98]:
array([[ 3.,  0.],
       [ 0.,  0.]])
```

Si l'on y regarde de plus près, cependant, le type des éléments `'int'` n'est pas le type entier dont on a l'habitude :

```
In [99]: type(M[0,0])
Out[99]: numpy.int32
```

Pour les tableaux, en effet, `numpy` ne peut utiliser les entiers habituels de Python, dont la taille en mémoire est variable, car ils ne permettent pas certaines optimisations. Il a

18. La matrice `M` n'est pas modifiée, on obtient une *nouvelle* matrice avec des éléments du type demandé. Pour modifier le type de `M`, on peut écrire `M[:] = M.astype('float')`.

donc choisi ici des *entiers signés sur 32 bits*, ce qui signifie qu'on ne peut utiliser des entiers qu'entre -2^{31} et $2^{31}-1$.

Ainsi, une erreur apparaît si on essaie de spécifier une valeur trop grande lors d'une mutation d'un des éléments de la matrice :

```
In [100]: M[0,0] = 2**31

OverflowError: Python int too large to convert to C long
```

Et il est possible d'observer des erreurs de débordement dans les calculs [19] :

```
In [101]: M[0,0] = 2**30

In [102]: M
Out[102]:
array([[1073741824,          0],
       [         0,          0]])

In [103]: M = M*2

In [104]: M
Out[104]:
array([[-2147483648,           0],
       [          0,           0]])
```

Lorsque l'on effectue une opération dont le résultat est un nouveau `numpy.array`, les types des éléments du nouveau `numpy.array` sont adaptés aux besoins. Par exemple,

```
In [105]: M = numpy.array( [ [1, 2], [3, 4] ], dtype='int' )

In [106]: M
Out[106]:
array([[1, 2],
       [3, 4]])

In [107]: numpy.sqrt(M)
Out[107]:
array([[ 1.        ,  1.41421356],
       [ 1.73205081,  2.        ]])
```

Dans la très grande majorité des cas, les changements de type se font de façon transparente et naturelle [20].

19. Parfois, `numpy` les signalera par un avertissement, mais ce n'est pas systématique.
20. Comme toute règle, il y a des exceptions. On se méfiera par exemple des `numpy.array` contenant des entiers non signés sur 64 bits (`'uint64'`), car si on leur ajoute par exemple un entier signé, dans une tentative d'éviter les débordements, `numpy` effectue les calculs sur des flottants avant de reconvertir le résultat en entier, et il peut y survenir des phénomènes d'absorption, inattendus avec des entiers.

5.3 Rechercher un élément dans un `numpy.array`

Nous avons eu l'occasion de voir qu'il était fréquent que l'on ait besoin de rechercher la présence d'un élément dans une liste. Il peut être utile de faire la même chose dans un `numpy.array`.

Le mot-clé « **in** » se trouve fonctionner tout aussi bien [21] sur des `numpy.array`, qu'ils aient une ou plusieurs dimensions [22] :

```
In [108]: M = numpy.array([[ 1, 2, 3 ], [ 4, 5, 6 ]])

In [109]: 5 in M
Out[109]: True
```

Si on écrit la fonction nous-même, dans l'implémentation la plus simple, il faudra autant de boucles qu'il y a de dimensions. Par exemple, pour un tableau à deux dimensions, on peut écrire

```
def Contient(M, x)
    """Return True si x est présent dans le numpy.array
       à deux dimensions M, et False sinon"""

    nb_lgn, nb_col = M.shape

    for i in range(nb_lgn) :
        for j in range(nb_col) :
            if M[i, j] == x :
                return True

    return False
```

Cela n'est gère pratique, car il faut connaître par avance le nombre de dimensions du `numpy.array`. Or, nous voulons simplement prendre tous les éléments du `numpy.array` un par un. Pour ce faire, on peut utiliser la fonction `.ravel()` qui retourne un `numpy.array` à une seule dimension contenant tous les éléments [23] :

```
In [110]: M.ravel()
Out[110]: array([1, 2, 3, 4, 5, 6])
```

Ainsi, on peut effectuer des boucles sur les éléments d'un `numpy.array`, indépendam-

21. Dans le cas où l'on met un *élément* (entier, flottant...) à gauche du « in » : `numpy` permet de placer à gauche du « **in** » des `numpy.array`, mais le comportement, pas totalement documenté, est parfois déroutant et surprenant.
22. Alors qu'on ne pourrait pas trouver un élément dans une liste de liste avec un **in**, comme on l'a vu dans le cours sur les listes.
23. Il existe aussi une fonction `.flatten()` qui fait (presque) la même chose.

ment de son nombre de dimensions :

```
In [111]: for elem in M.ravel() :
     ...:     print(elem, end=" ")

1 2 3 4 5 6
```

Cela permet donc de simplifier[24] notre fonction `Contient`, tout en la généralisant à des `numpy.array` avec un nombre quelconque de dimensions :

```
def Contient(M, x)
    """Return True si x est présent dans le numpy.array M
       et False sinon"""

    for elem in M.ravel() :
        if elem == x :
            return True

    return False
```

5.4 Appliquer une fonction aux éléments d'un `numpy.array`

Nous avons vu un peu plus tôt que l'on pouvait utiliser un `numpy.array` comme argument de nombreuses fonctions telles que `numpy.sin`, le résultat étant un `numpy.array` de la même forme que l'argument, contenant les résultats de la fonction appliquée à chacun des éléments de l'argument.

Une fonction `f` qui n'utiliserait que des opérateurs de Python et des fonctions mathématiques provenant du module `numpy` fonctionnera de la même façon, sur les éléments d'un `numpy.array` :

```
In [112]: M = numpy.array( [[1, 2, 3], [4, 5, 6]] )

In [113]: def f(x) :
     ...:     return 2 * numpy.sqrt(x)

In [114]: f(T1)
Out[114]:
array([[ 2.        ,  2.82842712,  3.46410162],
       [ 4.        ,  4.47213595,  4.89897949]])
```

Mais bien souvent, il n'est en principe pas possible d'utiliser une fonction sur un `numpy.array` si elle est un peu plus compliquée, plus particulièrement si elle contient des tests, des boucles, ou des fonctions ne provenant pas du module `numpy` :

```
In [115]: M = numpy.array( [[1, 2, 3], [4, 5, 6]] )

In [116]: def f(x) :
     ...:     if x % 2 == 0 :
     ...:         return x // 2
     ...:     else :
     ...:         return 3*x + 1

In [117]: f(M)
ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()
```

L'erreur n'est pas claire, mais il n'est pas content...

La fonction `numpy.vectorize` peut cependant construire une nouvelle fonction qui, elle, acceptera un `numpy.array` comme argument, à partir de note fonction `f` :

```
In [118]: fvec = numpy.vectorize(f)

In [119]: fvec(M)
Out[119]:
array([[ 4,  1, 10],
       [ 2, 16,  3]])
```

5.5 Démystifier le broadcasting

Les règles qui permettent à `numpy` d'effectuer des opérations sur des `numpy.array` de taille différentes ne sont en fait pas *si* compliquées.

Supposons que l'on cherche à effectuer une opération du type `T1` $\oplus$ `T2` où `T1` et `T2` n'ont pas la même dimension (et où $\oplus$ est un opérateur mathématique quelconque). Par exemple, `T1` a une taille $3 \times 1 \times 2 \times 4$ et `T2` une taille $5 \times 1 \times 4$.

Tout d'abord, `numpy` égalise le nombre de dimensions, en considérant par exemple ici que `T2` une taille $1 \times 5 \times 1 \times 4$ (les dimensions existantes sont calées à droite).

L'opération est possible si, pour chacune des dimensions :

- la taille est la même (aucun problème particulier) ;
- la taille de l'un des deux `numpy.array`, pour la dimension considérée, est égale à 1 (auquel cas les données de ce `numpy.array` seront répétées autant de fois qu'il y a d'éléments dans cette direction dans le second `numpy.array`).

Dans le cas de `T1` et `T2`, ces conditions sont donc bien respectées, aussi les opérations seront possibles.

24. Ici encore, si, dans un concours, un problème d'algorithmique demande d'effectuer une recherche dans un tableau à plusieurs dimensions, il est probablement plus prudent d'écrire explicitement les boucles, même si vous pouvez toujours signaler cette alternative.

```
In [120]: T1 = numpy.array([
  [ [ [ 11, 12, 13, 14], [ 15, 16, 17, 18 ] ] ],
  [ [ [ 21, 22, 23, 24], [ 25, 26, 27, 28 ] ] ],
  [ [ [ 31, 32, 33, 34], [ 35, 36, 37, 38 ] ] ] ])

In [121]: T2 = numpy.array([
  [ [1100, 1200, 1300, 1400] ],
  [ [2100, 2200, 2300, 2400] ],
  [ [3100, 3200, 3300, 3400] ],
  [ [4100, 4200, 4300, 4400] ],
  [ [5100, 5200, 5300, 5400] ] ])

In [122]: T1.shape
Out[122]: (3, 1, 2, 4)

In [123]: T2.shape
Out[123]: (5, 1, 4)

In [124]: T = T1+T2

In [125]: T.shape
Out[125]: (3, 5, 2, 4)
```

De la même façon, des `numpy.array` de taille $(1,3)$ et $(2,1)$ sont compatibles, et l'opération $\begin{array}{|c|c|c|}\hline 1&2&3\\\hline\end{array} + \begin{array}{|c|}\hline 4\\\hline 5\\\hline\end{array}$ est traduite en $\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 1&2&3\\\hline\end{array} + \begin{array}{|c|c|c|}\hline 4&4&4\\\hline 5&5&5\\\hline\end{array}$.

```
In [126]: numpy.array( [[ 1, 2, 3 ]] ) + numpy.array( [[ 4 ], [ 5 ]] )
Out[126]:
array([[5, 6, 7],
       [6, 7, 8]])
```

Ce mécanisme est souvent très pratique. Par exemple, pour une image couleur stockée dans un `numpy.array` nommé `img`, on peut réduire de moitié la composante rouge de chacun des pixels très simplement :

```
In [127]: img * numpy.array( [ 0.5, 1, 1 ] )
```

Ou bien, si l'on travaille avec des entiers pour les composantes de couleur, et que l'on souhaite rester sur des entiers, on peut aussi écrire

```
In [128]: img // numpy.array( [ 2, 1, 1 ] )
```