Informatique PT. 2024 - 2025

Mathématiques & Informatique - F. Gaunard
http://frederic.gaunard.com
Lycée Voltaire, Paris 11e.

1 Rappels d'algorithmique

On commence ce chapitre par la reprise de quelques exercices du **Cahier de vacances**.

Exercice 1. **Extrait du cahier de vacances**

Écrire une fonction `Python` d'en-tête `def symetrie(P,Q) :` qui prend en argument deux listes de même longueur `P`=$[p_0,\dots,p_n]$ et `Q`=$[q_0,\dots,q_n]$ et qui calcule et renvoie la valeur de la somme `s` où

$$\texttt{s} = \sum_{i=0}^{n} p_i q_{n-i}.$$

Exercice 2. **Extrait du cahier de vacances**

Écrire une fonction `Python` d'en-tête `def ordre_inverse(x) :` qui prend en argument un nombre entier `x` et renvoie un nombre qui composé des mêmes chiffres que `x` mais dans l'ordre opposé. Par exemple, on doit avoir le résultat d'exécution ci-dessous :

```
>>> ordre_inverse(8973)
3798
```

Exercice 3. **Extrait du cahier de vacances**

Écrire une fonction `Python` d'en-tête `def nb_sommets_isoles(L):` qui prend en argument une liste L d'adjacence d'un graphe $\mathcal{G}$ et qui renvoie le nombre de sommets isolés du graphe.

1 Listes

1.1 Parcours des termes d'une liste *via* leur indice

De nombreux algorithmes nécessitent de parcourir une **liste** (il n'y a qu'à regarder les trois exercices ci-avant déjà...). L'exemple suivant décrit l'un d'entre eux.

Exemple 1.

La fonction suivante prend en entrée une liste `L`, et renvoie en sortie le nombre de termes nuls présents dans cette liste.

```
def nb_zero(L):
    n=len(L)
    S=0
    for k in range(n):
        if L[k]==0:
            S=S+1
    return S
```

```
>>> nb_zero([1,0,0,2,3,0])
3
```

Le code `for k in range(n)` implique que la variable `k` prendra successivement les valeurs `0,1,2,...,n-1` et donc `L[k]` prendra successivement les valeurs `L[0],L[1],L[2],...,L[n-1]`.

Exercice 4.
Écrire une fonction qui prend en entrée une liste de nombres et renvoie la somme de ses éléments.

Exercice 5.
Écrire une fonction qui prend en entrée une liste de nombres et renvoie `True` si l'un de ses éléments est nul, et `False` sinon.

1.2 Parcours par valeurs des termes d'une liste

Dans le code de la fonction `nb_zero` ci-dessus, on parcourt la liste *par indices* (la variable `k` de la boucle `for` prend successivement toutes les valeurs des indices possibles des éléments de la liste `L`). Mais il est possible de parcourir la liste directement *par valeurs* de la façon suivante (la variable `x` de la boucle for suivante prend successivement toutes les valeurs des éléments de la liste `L`).

```
def nb_zero2(L):
    S=0
    for x in L:
        if x==0:
            S=S+1
    return S
```

```
>>> nb_zero2([1,0,0,2,3,0])
3
```

Exercice 6.
Reprendre les **Exercices 4** et **5** en effectuant cette fois un parcours par valeurs des listes.

☞ Si le parcours par valeurs d'une liste est en général plus rapide à implémenter que le parcours par indices, ce dernier est préférable si on s'intéresse à l'indice des termes.

1.3 Ajout et suppression d'un élément dans une liste

En `Python`, l'instruction `L.append(x)` permet d'ajouter l'élément `x` à la fin de la liste `L`. Cette instruction modifie la liste `L`. En particulier, sa longueur est augmentée de 1. De manière analogue, l'instruction `L.pop()` permet de supprimer le dernier élément de la liste `L`, si la liste `L` n'est pas vide.

Exemple 2.

```
>>> L=[1,3,4]
>>> L
[1,3,4]
>>> L.append(2)
>>> L
[1,3,4,2]
```

```
>>> L.pop()
2
>>> L
[1,3,4]
```

On constate sur cet exemple que l'instruction `L.pop()` renvoie un résultat : l'élément qui vient d'être supprimé dans la liste. On peut donc affecter ce résultat à une variable `a`. L'instruction `a=L.pop()` effectue donc deux choses en même temps : elle modifie la liste `L` en lui enlevant son dernier élément, et elle affecte à la variable `a` la valeur de cet élément. Attention, l'instruction `L.append(x)` ne renvoie aucun résultat, elle se contente de modifier la liste!

Exemple 3.

```
>>> L=[1,3,4]
>>> a=L.pop()
>>> a
4
>>> L
[1,3]
```

```
>>> b=L.append(2)
>>> b
>>> L
[1,3,2]
>>> L=L.append(3)
>>> L
```

En fait il est possible de supprimer n'importe quel élément d'une liste, en spécifiant son indice dans la fonction `pop`. Le résultat est alors la valeur de cet élément.

De la même manière, il est possible d'insérer un élément dans une liste à un certain rang `i`. L'instruction `L.insert(i,x)` insère l'élément `x` à l'indice `i`. Notez bien qu'après l'exécution de ces deux instructions, le rang des éléments situés après le `i`-ème est modifié.

Exemple 4.

```
>>> L=[1,3,4,8,0]
>>> a=L.pop(2)
>>> L
[1,3,8,0]
>>> a
4
```

```
>>> L.insert(1,7)
>>> L
[1,7,3,8,0]
>>> L[3]
8
```

1.4 Construction de listes à l'aide d'une boucle for

Supposons maintenant que l'on veuille construire une liste de la forme `[f(0),...,f(n-1)]` où f est une fonction déclarée au préalable, et n un entier. Bien entendu, on peut toujours entrer chacun des éléments de la liste un à un au clavier, mais si l'entier n est grand cela risque d'être long! On peut accélérer la définition d'une telle liste en utilisant une boucle `for`.

Une solution est de construire la liste terme par terme: on part d'une liste vide, et lors du k-ième passage dans la boucle on construit le k-ième terme de la liste (on dit que l'on effectue une construction par *couches* ou par *strates*).

Par exemple, la fonction déclarée ci-dessous prend en entrée un entier n et une fonction f, puis renvoie la liste `[f(0),...,f(n-1)]`.

```
def construction_liste(f,n):
    L=[ ]
    for k in range(n):
        L.append(f(k))
    return L
```

Exemple 5. Liste des carrés des cinq premiers entiers avec boucle `for`

```
def g(x):
    return x**2
```

```
>>> construction_liste(g,5)
[0, 1, 4, 9, 16]
```

1.5 Construction de listes par compréhension

Il existe une syntaxe dédiée à la construction de listes de la forme `[f(0),...,f(n-1)]`, qui est la suivante:

```
L=[f(i) for i in range(n)]
```

Lorsque l'on utilise cette syntaxe, on dit que l'on a défini la liste *par compréhension*.

Exemple 6. Liste des carrés des cinq premiers entiers par compréhension

```
def g(x):
    return x**2
```

```
>>> [g(i) for i in range(5)]
[0, 1, 4, 9, 16]
```

Sur cet exemple, la définition par compréhension est donc nettement plus rapide à implémenter. Néanmoins, dans des cas plus compliqués on ne peut pas utiliser la compréhension, et il est finalement nécessaire d'appliquer la première méthode.

Remarque 1.

Il n'est jamais *nécessaire* d'utiliser un parcours par valeurs des termes d'une liste, ou une définition par compréhension: on peut toujours s'en sortir en utilisant un parcours par indice. Le parcours par valeurs et la compréhension permettent toutefois de raccourcir la longueur du code, et sont donc recommandés par le jury de la banque PT, mais vous pouvez choisir de ne jamais les utiliser.

1.6 Extraction d'une sous-liste

Il est possible d'extraire une sous-liste d'une liste (on parle de *slicing* en anglais), en précisant les indices des éléments que l'on souhaite extraire: la commande `L[i:j]` renvoie la liste `[L[i],...,L[j-1]]` (observez bien que le dernier terme est `L[j-1]` et non pas `L[j]`: on obtient donc une liste de longueur $j-i$).

Exemple 7.

```
>>> L=[1,2,3,4,5]
>>> L[1:4]
[2, 3, 4]
>>> L[1:5]
[2, 3, 4, 5]
```

1.7 Concaténation & duplication de listes

Tout comme les chaînes de caractères, les listes supportent l'opérateur `+` de concaténation, ainsi que l'opérateur `*` pour la duplication.

La commande `L*n`, où n est un entier naturel renvoie une liste dont les éléments de L sont répétés n fois, bout à bout. Si $n=0$, cela renvoie donc une liste vide.
Il ne s'agit donc pas d'une multiplication par n terme à terme...

Exemple 8.

```
>>> [0,1]*4
[0, 1, 0, 1, 0, 1, 0, 1]
```

Attention, l'opérateur `+` n'est pas la somme terme à terme de deux listes. Le résultats est une liste dont le nombre de termes est égal à la somme des longueurs des deux listes et dont les termes sont successivement ceux de la première liste, puis ceux de la seconde.

Exemple 9.

```
>>> [k for k in range(5)]+[1]*4
[0, 1, 2, 3, 4, 1, 1, 1, 1]
```

1.8 Copie de listes

Il peut être utile de copier une liste, notamment quand on ne veut pas faire de tri *en place*.
On attire l'attention sur un comportement de `Python` qui peut paraitre étrange lorsqu'on souhaite copier une liste :

```
>>> liste1=[1,2,3,4]
>>> liste2=liste1
>>> liste1[3]=0
>>> liste1
[1, 2, 3, 0]
>>> liste2
[1, 2, 3, 0]
```

La modification de `liste1` a modifié également `liste2`. Cela vient du fait que `Python` a effectué la copie de liste *par référence*. Ainsi, les deux listes pointent vers le même objet dans la mémoire.
Pour contrer ce problème, on utilise l'instruction `L.copy()` :

```
>>> liste1=[1,2,3,4]
>>> liste2=liste1.copy()
>>> liste1[3]=0
>>> liste1
[1, 2, 3, 0]
>>> liste2
[1, 2, 3, 4]
```

2 Récursivité

Définition 1.
On appelle *fonctions récursives* les fonctions qui s'appellent elles-même. Avec ce paradigme de programmation on peut se dispenser d'utiliser des boucles, et aussi fréquemment d'utiliser des affectations.

Exemple 10.
Par exemple, la fonction récursive suivante calcule la factorielle d'un entier:

```
def factorielle(n):
    if n==0:                          <-- Initialisation
        return 1
    else:                             <-- Hérédité
        return n*factorielle(n-1)
```

L'exécution de cette fonction engendre des appels récursifs successifs qu'il faut *empiler*, et ensuite exécuter dans le sens inverse, puisque chaque appel doit attendre le résultat de la requête qu'il a engendré pour pouvoir être exécuté.
Dès qu'une réponse est possible, elle permet ainsi de remonter en dépilant les appels en attentes. Ce processus est illustré par le diagramme suivant qui donne l'exemple de l'exécution de `factorielle(4)`.

```
factorielle(4)
                ↪ factorielle(3)
                                ↪ factorielle(2)
                                                ↪ factorielle(0)
                                                                  sortie : 1
                                                sortie : 1 ↩
                                ↪ sortie : 2 ↩
                ↪ sortie : 6 ↩
sortie : 24 ↩
```

Par opposition, une fonction qui n'est pas récursive (utilisant seulement des boucles) est dite *itérative*.

Pour calculer la factorielle d'un entier, on aurait également pu utiliser la fonction itérative suivante:

```
def factorielle2(n):
    P=1
    for k in range(n):
        P=P*(k+1)
    return P
```

Sur cet exemple, on ne peut pas dire que l'une des deux fonctions est plus simple à coder que l'autre. On a simplement deux possibilités correspondant à deux paradigmes de programmation différents. On peut toutefois noter que, si les complexités temporelles de ces fonctions sont les mêmes (il s'agit d'une complexité linéaire, en $O(n)$), la version récursive a une complexité spatiale plus importante, puisque la pile des appels récursifs doit être enregistrée.

Notons enfin que dans le langage `Python`, le nombre d'appels récursifs est limité à environ 1000 (précisément 987 en `Python 3.5.2`), afin d'éviter de saturer la mémoire. On peut évidemment modifier cette limite, mais elle est suffisante pour traiter un grand nombre de problèmes (notamment ceux qui nécessitent un nombre d'appels logarithmique en la taille

des données, lorsque cette taille reste *raisonnable*.)

Exercice 7.

Écrire une fonction récursive `puissance` qui prend en entrée un flottant x et un entier n puis renvoie x^n.

Exercice 8.

Écrire une fonction récursive `nb_zerosR` qui prend en entrée une liste `L`, et renvoie en sortie le nombre de termes nuls présents dans cette liste.

Exercice 9.

Que fait la fonction `mystere` ci-dessous ?

```
def mystere(L):
    if len(L)==0 :
        return [ ]
    return [L.pop(-1)]+mystere(L)
```

3 Algorithmes de tri

Trier un ensemble d'objets consiste à ordonner ces objets en fonction de clés et d'une relation d'ordre définie sur celles-ci. Par exemple, trier une liste d'entiers revient à ranger ceux-ci dans l'ordre croissant . Il s'agit d'une opération très utile (il est par exemple beaucoup plus rapide de rechercher des éléments dans une liste si celle-ci a été triée, par exemple pour une *recherche dichotomique*) et donc fréquemment utilisée. De nombreux algorithmes utilisent des tris.

Lorsque l'on veut trier une liste très longue, outre la question de la rapidité du tri, se pose la question de l'utilisation de l'espace mémoire. La mémoire est fortement sollicitée lorsque les données à trier sont recopiées, globalement ou en partie, dans des variables différentes, et ce de nombreuses fois lors du tri. Au contraire, l'espace mémoire est économisé lorsque l'on effectue le tri « en place ».

Définition 2. **Tri en place**

Un tri est dit *en place* (ou *sur place*) s'il modifie directement la structure qu'il est en train de trier et ne nécessite pas l'allocation d'une nouvelle structure.

3.1 Le tri par insertion (*insertion sort*)

C'est le tri du joueur de cartes. On fait comme si les éléments à trier étaient donnés un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1. On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite.
Le principe du tri par insertion est donc d'insérer à la $n-$ième itération le n-ième élément à la bonne place. C'est un tri en place.

```
def tri_par_insertion(L):
    N = len(L)
    for n in range(1,N):
        a_trier = L[n]
        j = n-1
        while j>=0 and L[j] > a_trier:
            L[j+1] = L[j] # decalage
            j = j-1
        L[j+1] = a_trier
```

La complexité du tri par insertion est $O(n^2)$ dans le pire cas (et en moyenne), et linéaire dans le meilleur cas.

3.2 Le tri par sélection (*selection sort*)

Le *tri par sélection* consiste à sélectionner le plus petit élément de la liste à trier, puis à le permuter avec l'élément situé en tête de liste. On répète ensuite le processus avec la liste des éléments restants (tous sauf le premier) jusqu'à ce que la liste soit triée.

On commence par implémente le tri par sélection en place. Pour cela, on écrit d'abord une fonction `minimum(L,m)` qui renvoie l'indice du plus petit terme de la sous-liste `L[m:len(L)]`.

```
def minimum(L,m):
    candidat=m
    for k in range(m+1,len(L)):
        if L[k]<L[candidat]:
            candidat=k
    return candidat

def tri_par_selection_en_place(L):
    for i in range(len(L)):
        mini=minimum(L,i)
        L[i],L[mini]=L[mini],L[i]
```

On peut démontrer que le tri par sélection a une complexité quadratique, c'est-à-dire en $O(n^2)$, puisque le code correspondant contient deux boucles imbriquées l'une dans l'autre.
En effet, l'algorithme effectuera $(n-1)+(n-2)+...+1 = n(n-1)/2$ comparaisons.

3.3 Le tri par fusion (*merge sort*)

Le principe du tri fusion est de partager la liste à trier en deux sous-listes de même longueur. Une fois celles-ci triées récursivement, il suffit de les fusionner. Il est très difficile de fusionner deux sous-listes d'une même liste en place, donc on se contentera d'une implémentation de la version fonctionnelle (qui a pour inconvénient une mauvaise complexité spatiale).

On commence par écrire une fonction récursive `Fusion` qui prend en entrée deux listes triées et qui renvoie la liste triée contenant les éléments des deux listes d'entrée.

```
def Fusion(L1,L2):
    if L1==[ ]:
        return L2
    if L2==[ ]:
        return L1
    if L1[0]<L2[0]:
        return [L1[0]]+Fusion(L1[1:len(L1)],L2)
    if L1[0]>=L2[0]:
        return [L2[0]]+Fusion(L1,L2[1:len(L2)])
```

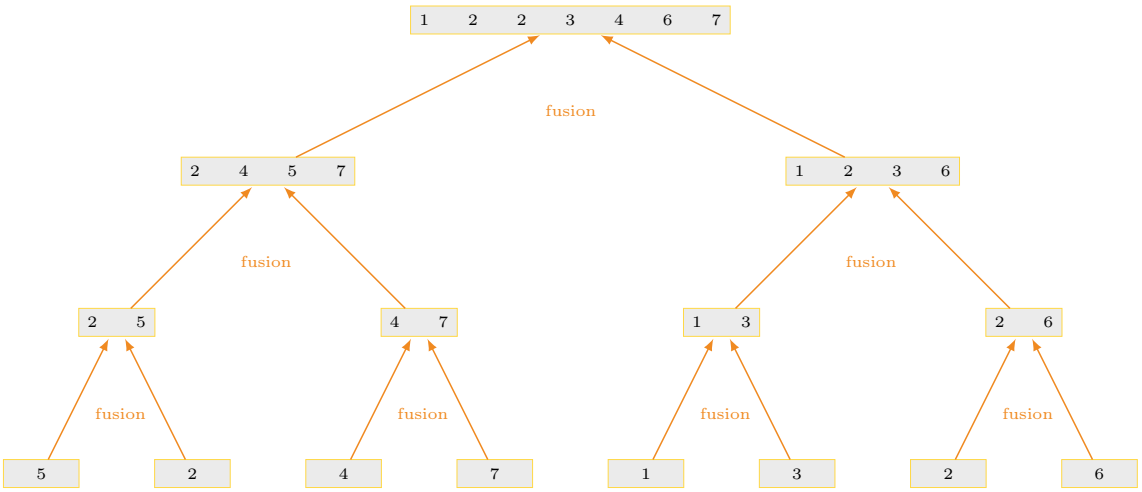
Fusion dans *DragonBall Z*
© Studio Bird / Shueisha, Toei Animation.

☞ On en déduit une implémentation du tri fusion :

```
def TriFusion(L):
    if len(L)<=1:
        return L
    else:
        n=len(L)
        p=n//2
        return Fusion(TriFusion(L[0:p]),TriFusion(L[p:n]))
```

Exemple 11.

La figure ci-dessous illustre la remontée de l'algorithme du tri par Fusion sur la liste $L = [5, 2, 4, 7, 1, 3, 2, 6]$.



Si `len`$(L) = n = 2^p$, l'appel de `TriFusion(L)` induit $p = \log_2(n)$ appels récursifs. Et à chaque appel, on effectue une fusion, qui a pour complexité dans le pire des cas O (`len`$(L1)$ + `len`$(L2)$), soit au maximum $O\left(\frac{n}{2} + \frac{n}{2}\right) = O(n)$. Finalement, la complexité dans le pire des cas est $\log_2(n) \times O(n)$.
On peut même montrer que, dans tous les cas, la complexité du tri fusion est en $O(n \log_2 n)$.
☞ Ce résultat est à relativiser par le fait que ce tri n'est pas en place et possède une complexité spatiale non négligeable dans l'implémentation proposée ci-dessus.

3.4 Le tri rapide (*quicksort*)

Comme le tri par fusion, l'algorithme de tri rapide consiste à partager la liste L à trier en deux sous-listes $[L[0], ..., L[q-1]]$ et $[L[q+1], ..., L[n-1]]$ autour d'un *pivot* $L[q]$ de telle sorte que tous les éléments de la sous-liste $[L[0], ..., L[q-1]]$ soient inférieurs ou égaux à $L[q]$ lui-même inférieur ou égal à tous les éléments de la sous-liste $[L[q+1], ..., L[n-1]]$.
Les deux sous-listes sont alors triées par des appels récursifs au tri rapide.

Dès lors qu'est connue la fonction qui partitionne en deux, la fonction récursive de tri rapide n'est pas difficile à écrire

```
def tri_rapide(L, indice_min, indice_max) :
    if indice_min < indice_max :
        pivot=partition(L, indice_min, indice_max)
        tri_rapide(L, indice_min, pivot-1)
        tri_rapide(L, pivot+1, indice_max)
```

La difficulté est donc l'écriture de la fonction `partition` qui réarrange la liste sur place et permettra un *découpage* autour du pivot.

- le pivot est placé à la fin (arbitrairement), en l'échangeant avec le dernier élément du sous-tableau ;
- tous les éléments inférieurs au pivot sont placés en début du sous-tableau ;
- le pivot est déplacé à la fin des éléments déplacés.

```
def partition(L, indice_min, indice_max) :
    x=L[indice_max]
    i=indice_min - 1 # indice du plus grand élément inférieur ou égal au pivot
    for j in range(indice_min, indice_max) : # on parcourt tous les éléments
        if L[j] <= x : # si on trouve plus petit que le pivot
            i=i+1
            L[i], L[j] = L[j], L[i] # on  place ce terme plus petit à gauche du pivot
    L[i+1], L[indice_max] =  L[indice_max], L[i+1] # on place le pivot
    return i+1 # indice du pivot dans la liste ré-arrangée
```

☞ La complexité moyenne du tri rapide est $O(n \log n)$, ce qui est optimal pour un tri par comparaison, mais la complexité dans le pire des cas est quadratique. Malgré ce désavantage théorique, c'est en pratique un des tris les plus rapides, et donc un des plus utilisés.

4 Tableaux

4.1 Listes de listes

Pour modéliser des tableaux en `Python`, on peut utiliser des listes de listes.

```
>>> T=[[1,2],[3,4]]
>>> T[0][1]
2
```

Mais la syntaxe permettant d'accéder aux termes n'est pas très commode, et il n'existe aucune opération spécifiquement définie sur ces objets. On préfère donc souvent utiliser des tableaux `numpy`.

4.2 Le type ndarray de numpy

Pour utiliser les tableaux sous Python, il faut importer le module `numpy`, en utilisant la ligne de code :

```
import numpy as np
```

☞ La commande `as` permet de renommer le module `numpy`. Or pour appeler une fonction issue d'un module, on doit préfixer le nom de celle-ci par le nom du module: ici on pourra donc utiliser les fonctions de `numpy` en faisant précéder leur nom de `np`.
Le type associé aux tableaux définis sous `numpy` est `ndarray`. Ces tableaux sont définis grâce à la commande `array` de la bibliothèque `numpy`, qui prend en entrée une liste dont les termes sont des listes (chacune d'entre elles codera une ligne du tableau), et renvoie un objet de type `ndarray`.

```
>>> T=np.array([[1,2],[3,4]])
>>> T
array([[1, 2],
       [3, 4]])
>>> type(T)
<class 'np.ndarray'>
```

Remarque 2.
Contrairement au cas des listes, tous les éléments d'un tableau de type `ndarray` doivent être du même type, et ce type ne peut être que le type entier ou le type flottant.

La commande `shape` permet de calculer la *taille* d'un tableau (renvoyée sous forme de tuple *(nombre de lignes, nombre de colonnes)*), et la commande `zeros` permet de définir un tableau de taille quelconque ne contenant que des zéros (la taille sera spécifiée sous la forme d'un tuple ou d'une liste).

```
>>> np.shape(T)
(2, 2)
>>> print(np.zeros((2,3)))
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Pour accéder à un élément d'un tableau (en lecture ou en écriture), il faut indiquer entre crochets les indices de ligne et de colonne de cet élément, comme pour une liste (et attention les indices de lignes et de colonnes commencent toujours à 0).

```
>>> T[1,1]
4
```

4.3 Opérations élémentaires sur les tableaux

On peut extraire la ligne numéro i (resp. la colonne j) d'un tableau T avec la commande `T[i,:]` (resp. `T[:,j]`).

```
>>> T[0,:]
array([1, 2])
>>> T[:,1]
array([2, 4])
```

Enfin on peut extraire un sous-tableau d'un tableau T avec la commande `T[i1:i2,j1:j2]`: celle-ci renvoie le tableau contenant l'intersection des lignes $i_1, \dots, i_2 - 1$ et des colonnes $j_1, \dots, j_2 - 1$ de T.

```
>>> T2=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> print(T2)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> T2[0:2,0:2]
array([[1, 2],
       [4, 5]])
```

Il faut enfin noter que les tableaux de type `ndarray` sont de taille fixe: il n'existe donc pas d'équivalent de la primitive `append` s'appliquant à ceux-ci.

Exercice 10.

Écrire une fonction permettant de calculer la moyenne des termes d'un tableau 2D.

Exercice 11.

Écrire une fonction qui prend en entrée un tableau 2D de nombres et renvoie `True` si l'un de ses éléments est nul, et `False` sinon.

☞ On peut parcourir par valeurs un tableau `A`. On prendra garde au type de `x` dans l'instruction `for x in A`, où `x` vaudra alternativement chacune des lignes de `A`.

L'un des intérêts des tableaux de type `ndarray` par rapport aux listes de listes est que les opérations élémentaires s'appliquent à ceux-ci comme si c'étaient des vecteurs. De plus, les fonctions de la bibliothèque `numpy` s'appliquent **termes à termes** à ceux-ci.

```
>>> T
array([[ 0.        , 3.14159265],
       [ 1.57079633, 0.        ]])
>>> T-T
array([[ 0., 0.],
       [ 0., 0.]])
```

```
>>> T+2*T
array([[ 0.        , 9.42477797],
       [ 4.7123890, 0.        ]])
>>> np.cos(T)
array([[  1.0000000e+00, -1.000000e+00],
       [  6.1232340e-17,  1.000000e+00]])
```

4.4 Les tableaux 1D

Pour définir des tableaux de type `ndarray` ne contenant qu'une seule ligne, on peut utiliser une syntaxe spécifique (on met des `[ ]` en moins):

```
>>> T=np.array([1,2,3])
>>> T
array([1, 2, 3])
>>> T[0]
1
```

On parle alors de tableaux 1D (à une dimension) qui contiennent une seule ligne, par opposition aux tableaux 2D (à deux dimensions) qui contiennent une ou plusieurs lignes, car leurs termes sont repérés par un seul indice (au lieu de deux). En termes de contenus, ces tableaux sont équivalents à des listes, et on accède par exemple au terme d'indice `k` avec la commande `T[k]`. Mais attention, leur type est toujours `ndarray`, et toutes les fonctions s'appliquant aux listes ne peuvent pas être utilisées sur ceux-ci, comme par exemple `append`.

Les tableaux 1D correspondent à des vecteurs sur lesquels `numpy` permet d'effectuer du calcul vectoriel usuel (combinaison linéaire, produit scalaire, produit vectoriel, ...).

☞ Lorsque l'on extrait une ligne ou une colonne d'un tableau 2D, on obtient un tableau 1D.

Exercice 12.

Tester l'entrée `T=np.array([1,2,3])` dans la console. Quelles différences observe-t-on avec l'entrée `U=np.array([[1,2,3]])`?