Tableaux

17 octobre 2017

1 Généralités

- Imaginons que l'on veuille stocker en mémoire les cent premiers nombres premiers. Une solution serait de créer 100 variables que l'on pourrait nommer `p1,p2,...,p100`. L'utilisation d'un tableau nous permettra d'utiliser une seule variable et de pouvoir accéder à ses éléments.
- Nous dirons qu'un **tableau** est une suite finie de valeurs d'un même type (entier, flottant, booléen ...), stockées dans des cases mémoires contigües. On parle de **liste** en Python (les éléments d'une liste Python peuvent néanmoins avoir des types différents [1]). Par exemple `t = [4,6,5,8,7]` est un tableau d'entiers. On obtient sa longueur avec la fonction `len` (diminutif de lenght). Ainsi `len(t)` renverra 5.
- Accès aux éléments : ils sont repérés par leur **indice** : attention le premier élément est celui d'indice 0, (penser aux étages dans un ascenceur). Par exemple, `t[0]` vaut 4, et le dernier élément `t[4]` vaut 7.
- La liste vide qui ne contient aucun élément est notée `[]`.
- Une liste est un objet Python **mutable**, (on peut le modifier sans changer son adresse mémoire) contrairement à une chaîne de caractères ou à un tuple.

```
>>> id(t)
59320480
>>> t[0] = 9
>>> t
[9, 6, 5, 8, 7]
>>> id(t)
59320480
```

```
>>> s = 'love'
>>> s[2]
'v'
>>> s[2] ='z'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item
```

1. On verra que Python possède une autre structure de tableaux de nombres optimisés pour les calculs, les tableaux de type `array` du module numpy.

2 Comment construire des listes ?

2.1 La méthode `append`

On peut ajouter facilement un élément à la fin de la liste à l'aide de la **méthode** `append`. L'instruction `t.append(4)` ne renvoie pas une nouvelle liste, elle modifie la liste existante. Par exemple, si t est la liste précédente :

```
>>> t.append(4)
>>> t
[9, 6, 5, 8, 7, 4]
>>> id(t)
59320480
```

On remarque bien que l'identifiant de `t` n'a pas changé.

C'est pratique lorsque l'on construit pas à pas une liste. Souvent la liste de départ est la liste vide. Par exemple si l'on dispose d'une fonction booléenne `estPremier` qui teste la primalité d'un entier, le script suivant construit la liste des nombres premiers inférieurs à 1000

```
ma_liste = [] # liste vide
for n in range(1000):
    if estPremier(n):
        ma_liste.append(n)
```

2.2 Les «list comprehension»

Le script suivant crée la liste des carrés des entiers de 0 à $n-1$:

```
t = []
for k in range(n):
    t.append(k**2)
```

Voici une autre syntaxe permise par Python (on parle de «list comprehension»), proche de l'écriture mathématique $\{k^2 \mid \quad k \in [\![0, n-1]\!]\}$.

```
t = [k**2 for k in range(n)]
```

On peut de plus «filtrer» une «list comprehension». Par exemple, le tableau `t2` va sélectionner les éléments pairs de `t`

```
t2 = [x for x in t if x % 2 ==0]
```

On obtient ainsi un script élégant qui donne la liste des nombres premiers inférieurs à 1000.

```
[ p for p in range(1000) if estPremier(p)]
```

Les listes construites par «compréhension» seront un moyen agréable et efficace (comme nous allons le voir plus bas) de construire des listes Python.

2.3 La concaténation...parfois à éviter

L'opérateur `+` concatène deux listes. L'opérateur `*` est une concaténation itérée.

Par exemple, prenons `t1 = [ 2,5,6]` et `t2 = [3,7]`. Alors `t1 + t2` vaut `[2,5,6,3,7]` et `t2 + t1` vaut `[3,7,2,5,6]`. La concaténation n'est donc pas commutative. Enfin, l'instruction `3*t2` est équivalente à `t2 + t2 + t2` et donc vaut `[3,7,3,7,3,7]`.

Le script suivant fournit donc aussi la liste des nombres premiers inférieurs à 1000.

```
liste_premiers = [] # liste vide
for n in range(1000):
    if estPremier(n):
        liste_premiers = liste_premiers + [n]
```

Pourtant ce script est beaucoup moins performant que celui obtenu ci-dessus avec la méthode `append`. Mais pourquoi donc ? Lorsque l'on écrit `liste_premiers = liste_premiers + [n]`, une nouvelle liste (donc une nouvelle adresse mémoire), est crée à chaque itération. Cela ne se produit pas avec la méthode `append`.

```
>>> t=[2]
>>> id(t)
61896064
>>> t = t + [3]
>>> id(t)
48666504
```

```
>>> t =[2]
>>> id(t)
62754608
>>> t.append(3)
>>> id(t)
62754608
```

2.4 Petit comparatif de performances pour créer une liste nulle

Nous allons comparer le temps de calcul des trois scripts suivants qui créent une liste de zéros de taille $n = 10^5$. Le premier script met une vingtaine de secondes tandis que les deux suivants sont quasi-instantanés.

```
# script 1
t = []
for k in range(n):
    t = t + [0]
```

```
# script 2
t = []
for k in range(n):
    t.append(0)
```

```
# script 3
t = [0 for k in range(n)]
```

On évitera donc le premier script. À noter que l'on peut aussi créer une liste nulle de taille n avec le script suivant `t = [0]* n`. On pourrait croire au premier abord, qu'il est équivalent au script 1, mais non en fait il est tout aussi efficace que les scripts 2 et 3. Nous verrons dans la section 5 qu'il faudra toutefois l'éviter pour la construction de matrices.

3 Quelques Pythonnneries pratiques

3.1 On peut itérer une liste

Le script suivant calcule la somme des éléments du tableau t.

```
t= [4,6,5,8,7]
n = len(t)
s= 0
for i in range(n): # on itère sur les indices
    s = s + s[i]
```

Python permet une syntaxe parfois très pratique, qui évite d'utiliser les indices :

```
s = 0
for element in t:  # on itère sur les éléments
    s = s + element
```

On dit qu'une liste ou une chaîne de caractères sont des **objets itérables**, tout comme les objets de type range, les tuples, les fichiers...

```
>>> mot ='love'
>>> for lettre in mot:
        print (lettre.upper())
L
O
V
E
```

3.2 Le slicing

Ce sont des raccourcis syntaxiques qui permettent le découpage en tranches d'une liste. Par exemple, si `t = [4,5,7,2,3]`,

`t[1:3]` vaut `[5,7]`, `t[2: ]` vaut `[7,2,3]`, `t[ :4]` vaut `[4,5,7,2]`, `t[ : ]` vaut la liste toute entière.

Attention, l'objet t[:] n'est pas le même que t, comme le montre les identifiants.

```
>>> t =[4,5]
>>> id(t)
90100680
>>> t2 = t[:]
>>> id(t2)
90235208
```

Avec un indice négatif, on part de la fin, ainsi `t[-1] = 3` (dernier élément), `t[-2] = 2` (avant dernier-élément).

On peut aussi «slicer» avec un pas négatif. Par exemple `t[4:1:-1]` donnera la liste `[3,2,7]`, c'est-à-dire les éléments `t[4],t[3],t[2]` (-1 est le pas).

Exemples :

1. On peut ainsi obtenir par slicing l'inverse d'une liste `t` avec `t[ : :-1]`. Il y a aussi la méthode `reverse` qui inserve sur place les éléments d'une liste. Par exemple si `t = [3,2,1]` après avoir exécuté `t.reverse()` la liste `t` vaut `[1,2,3]`.

2. Le slicing s'applique aussi aux chaînes de caractères, la fonction suivante teste si une chaîne de caractères est un palindrome, c'est-à-dire se lit aussi bien de la gauche que de la droite («parerap» est un palindrome, mais «para» n'en n'est pas un).

```
def is_palindrome(s):
    m = len(s) // 2
    return s[ : m] == s[ : -(m+1) : -1]
```

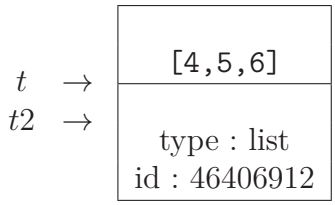
4 Attention aux copies de listes

On pourra avec profit visualiser l'exécution des scripts Python sur cet excellent site `http://www.python`

On considère le script suivant :

```
t = [4,5,6]
t2 = t
t[0] = 7
```

Alors l'élement `t2[0]` vaudra aussi 7 au lieu de 4. Ceci provient de la nature de l'affectation en Python. L'affectation `t2 = t` n'a pas réalisé de copie de la liste [4,5,6]. On a juste rajouté dans l'espace des noms un nouveau nom `t2`. Les noms `t` et `t2` sont deux étiquettes d'un même objet la liste [4,5,6], ce qui est confirmé en demandant `id(t)` et `id(t2)` qui sont les mêmes.



Il faut être sensible à cette problématique notamment dans le cas de fonctions prenant en argument une liste. Par exemple, on désire écrire une fonction `double` prenant en argument une liste et renvoyant une liste dont les éléments sont les doubles de la liste de départ.

```
def double(liste):
    liste2 = liste
    for k in range(len(liste2)):
        liste2[k] = 2 * liste2[k]
    return liste2

t = [2,3,4]
print(double(t))
print(t)
```

La fonction ci-dessus a contaminé la liste t ! Les scripts suivants corrigent ce défaut :

```
def double2(liste):
    n= len(liste)
    liste2 = [0] * n
    for k in range(n):
        liste2[k] = 2 * liste[k]
    return liste2
```

```
def double3(liste):
    liste2 = liste[:]
    for k in range(len(liste2)):
        liste2[k] = 2 * liste2[k]
    return liste2
```

```
def double4(liste):
    return [2*x for x in liste]
```

Ce type de comportement possède bien sûr aussi des avantages. Par exemple une fonction de tri qui ne recopiera pas la liste existante et la triera sur place, ce qui représentera une économie de mémoire.

Par exemple la fonction `sorted` trie mais renvoie une nouvelle liste mais la méthode `sort` modifie la liste de départ.

Pour faire une copie des éléments d'une liste, on peut utiliser du slicing et affecter à une variable `t2` la tranche `t[ : ]`.

```
t = [4,5,6]
t2 = t[ : ]
t[0] = 7
```

Mais attention, si les éléments de t sont eux même des listes, on aura des problèmes.

Par exemple,

```
t =[[2,3],5]
t2 = t[:]
t[0][0] = 7
print(t2)
```

On trouve que `t2` vaut `[ [7,3],5]`.

En revanche modifier `t[1]` ne va pas contaminer `t3`. Utiliser `Pythontutor` pour bien comprendre ce qu'il se passe.

On peut créer des copies «profondes» à l'aide de la fonction `copy.deepcopy`.

Si l'on désire une structure de données «composite» que l'on ne peut pas modifier, on pourra utiliser en Python le tuple.

5 Application à la création de matrices ou tableaux à deux dimensions

On peut modéliser une matrice comme une liste de listes. Par exemple, pour écrire la matrice $A = \begin{pmatrix} 4 & 5 & 8 \\ 2 & 1 & 7 \end{pmatrix}$, on peut écrire la liste de ses lignes : `A = [ [4,5,8], [2,1,7] ]`.

Ainsi `A[0][1]` est l'élément de la ligne d'indice 0 et de la colonne d'indice 1, soit le nombre 4. De même, `A[1][2]` vaut 7.

Attention, pour créer $B = \begin{pmatrix} 4 & 5 & 7 \\ 4 & 5 & 7 \\ 4 & 5 & 7 \end{pmatrix}$, on évitera le script «contaminant»ci-dessous ! !

```
>>> v =[4,5,7]
>>> B = [v,v,v]
>>> B
[[4, 5, 7], [4, 5, 7], [4, 5, 7]]
>>> B[0][2] = 9
>>> B
[[4, 5, 9], [4, 5, 9], [4, 5, 9]]
```

Attention aussi à la création d'une matrice nulle. Par exemple, pour créer la matrice $A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$, on peut exécuter le script `A = [[0]*3]*2`. Mais gros problème,

```
>>> A[0][1] = 10
>>> A
[[0, 10, 0], [0, 10, 0]]
```

.

Là encore, `Pythontutor` nous permet de bien comprendre ce qu'il se passe. Pour la création de la matrice nulle, on utilisera plutôt des «comprehension list» : voici une fonction prenant en argument deux entiers naturels non nuls n et p et renvoyant la matrice nulle avec n lignes et p colonnes :

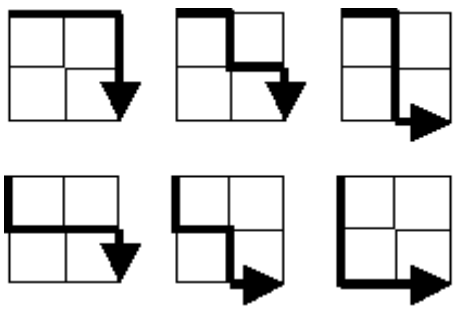
```
def matrice_nulle(m,n):
    return [ [ 0 for j in range(n)] for i in range(m)]

>>> matrice_nulle(2,3)
[[0, 0, 0], [0, 0, 0]]
```

6 Une application : euler project 15

Voici l'énoncé du problème 15 du remarquable site `https://projecteuler.net`.

Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.



How many such routes are there through a 20×20 grid ?

Voici un énoncé qui permet de résoudre ce problème :

Pour m et n dans $\mathbb{N}$, on note $p(m, n)$ le nombre de chemins pour une grille de taille $m \times n$.

1. Que valent $p(m, 0)$ et $p(0, n)$?
2. On prend m et n non nuls. Déterminer une relation de récurrence vérifiée par $p(m, n)$.
3. Compléter le tableau suivant, en déduire $p(4, 3)$.

$m\backslash n$	0	1	2	3
0				
1				
2				
3				
4				

4. En déduire une fonction récursive `path` qui permet de calculer $p(m, n)$.
5. Tracer l'exécution de `path(3,2)` et constater son inefficacité pour une grille 20×20.

On va pallier au trop grand nombre d'appels récursifs en mémorisant les calculs intermédiaires.

On prend $m = 20$ et $n = 20$.

6. Créer une matrice A à $m+1$ lignes et $n+1$ colonnes dont tous les coefficients sont égaux à 1.
7. Écrire un script qui complète cette matrice, de sorte que $A_{m,n}$ soit égal à $p(m, n)$.
8. Prolongement : déterminer $p(m, n)$ sans formule de récurrence par un calcul de dénombrement.
9. Refaire l'exercice, si on rajoute une possibilité de se déplacer en diagonale.

Remarque : on peut compter le nombre d'appels $a(m, n)$ de la fonction `path` lorsque l'on exécute `path(m,n)`. En effet, on a

$$a(m, n) = a(m, n-1) + a(m-1, n) + 1.$$

On obtient ainsi par exemple que $a(4, 3) = 19$ et $a(2, 1) = 5$.