2. matrices_cours

October 27, 2020

1 I : Les matrices - partie 1 : définition & parcours de matrice

Une matrice est un tableau de tableaux de même longueur comportant tous des éléments du même type. C'est un objet extrêmement utilisé en informatique : représentation de graphes, représentation d'images, outil de calcul mathématique etc.

Voici une représnetation d'une matrice M comportant 3 lignes et 4 colonnes, qui peut être implémentée par un tableau contenant lui même trois tableaux de 4 éléments :

$$\begin{matrix} 0 & 2 & 4 & 6 \\ 30 & 32 & 37 & 39 \\ 47 & 48 & 49 & 49 \end{matrix}$$

[1]:
```
M = [[0, 2, 4, 6], [30, 32, 37, 39], [47, 48, 49, 49]]
```

On note qu'en Python l'instruction qui définit une matrice peut être écrite sur plusieurs lignes de code ce qui permet d'améliorer la lisibilité (c'est la cas pour toutes les instructions comportant des parenthèses ou des crochets, voir ce post sur Stackoverflow par exemple) :

[2]:
```
M = [[0,  2,  4,  6],
     [30, 32, 37, 39],
     [47, 48, 49, 49]]
```

Ainsi on représente aisément une matrice `M` comme un tableau de lignes. Dans ce cas :

- `M[i]` désigne le i-èmetableau de `M`, c'est à dire la i-ème ligne,
- `M[i][j]` désigne le j-ème élément du i-ème tableau de `M`, c'est à dire la valeur située ligne `i` et colonne `j`.

Remarque 1: C'est une convention quasiment universelle de représenter une matrice comme un tableau de lignes. Il serait très maladroit de représenter une matrice comme un tableau de colonnes.

Remarque 2: Si on dispose d'une matrice non vide on peut facilement obtenir ses dimensions :

[3]:
```
nb_lig = len(M)
nb_col = len(M[0])
nb_lig, nb_col
```

[3]:
```
(3, 4)
```

Remarque 3 : On parcourt une matrice avec une boucle imbriquée (ou double boucle). On distingue les parcours par ligne et par colonne.

[4]:
```
#parcours par ligne

for lig in range(len(M)):
    for col in range(len(M[0])):
        print (M[lig][col])
    print("fin de ligne")
```

```
0
2
4
6
fin de ligne
30
32
37
39
fin de ligne
47
48
49
49
fin de ligne
```

[5]:
```
# parcours par colonne

for col in range(len(M[0])):
    for lig in range(len(M)):
        print (M[lig][col])
    print("fin de colonne")
```

```
0
30
47
fin de colonne
2
32
48
fin de colonne
4
37
49
fin de colonne
6
39
49
fin de colonne
```

Avant de poursuivre sur les matrices, il y a besoin d'une disgression ...

2 II : Python : variables désignant des objets mutables

En python, il y a deux sortes d'objets : les objets mutables et les objets immuables. Les objets immuables ne peuvent pas être modifiés alors que les objets mutables peuvent être modifiés :

- types d'objets immuables : `int, str, float, bool ...`
- types d'objets mutables : `lits, dict ...`

En effet nous avons vu qu'on peut par exemple facilement modifier la valeur des éléments d'un tableau alors que pour les chaînes de caractères il est impossible de modifier un caractère donné :

[6]:
```
mon_tableau = [3, 33, 444, 3333, 33333]
mon_tableau[2] = 333
mon_tableau
```

[6]:
```
[3, 33, 333, 3333, 33333]
```

[7]:
```
ma_chaine = "unf chaîne est immuable"
ma_chaine[2] = "e"
ma_chaine
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-7-13758d2c31b9> in <module>
  1 ma_chaine = "unf chaîne est immuable"
----> 2 ma_chaine[2] = "e"
  3 ma_chaine


TypeError: 'str' object does not support item assignment
```

Remarque :

Attention à ne pas confondre *mutation* d'un objet et *réaffectation* d'une variable :

[8]:
```
mon_tab = ["Léo", "Sacha", "Alice", "Hisham"]
mon_tab[2] = "Béatrice"        #mutation du tableau désigné par la variable␣
↪mon_tab
mon_tab
```

[8]:
```
['Léo', 'Sacha', 'Béatrice', 'Hisham']
```

[9]:
```
mon_tab = ["Enzo", "Léa", "Samuel"]     #réaffectation de la variable mon_tab
mon_tab
```

[9]:
```
['Enzo', 'Léa', 'Samuel']
```

Un différence fondamentale à connaître entre les objets mutables ou immuables est la suivante :

En Python : - les variables désignant des objets immuables “contiennent” la valeur de l’objet en question, - les variables désignant des objets mutables “contiennent” l’adresse en mémoire de l’objet en question.

Voyons les conséquences que cela entraîne.

2.0.1 Lors de l’affectation `tab_b = tab_a`, `tab_b` et `tab_a` désignent le même tableau

Supposons qu’on souhaite effectuer la copie d’un tableau : l’idée de base est d’effectuer une affectation de la forme `tab_b = tab_a` :

[10]:
```
tab_a = [True, False, True, False]
tab_b = tab_a
tab_b
```

[10]:
```
[True, False, True, False]
```

On a ici le sentiment d’avoir copié la valeur de `tab_a` dans `tab_b`. En réalité ce n’est pas le cas : `tab_b` désigne désormais la même adresse que `tab_a`, c’est à dire le même tableau. Par conséquent si on effectue des modifications sur le tableau désigné par `tab_b`, ces modifications affecteront aussi le tableau désigné par `tab_a` (ce qui est normal puisque ces deux variables désignent l’adresse d’un unique tableau !) :

[11]:
```
tab_b.append(True)
tab_b.append(True)
tab_b.append(True)
tab_a
```

[11]:
```
[True, False, True, False, True, True, True]
```

Pour obtenir une copie indépendante de `tab_a`, il faudra utiliser la méthode `deepcopy()` du module `copy` :

[12]:
```
from copy import deepcopy

tab_a = [True, False, True, False]
tab_b = deepcopy(tab_a)
tab_b.append(True)
tab_b.append(True)
tab_b.append(True)
tab_a
```

[12]:
```
[True, False, True, False]
```

[13]:
```
tab_b
```

[13]:
```
[True, False, True, False, True, True, True]
```

2.0.2 Les mutations d'un argument à l'intérieur d'une fonction se répercutent à l'extérieur

Lorsqu'une fonction prend en argument un objet de type mutable, les mutations sur l'objet à l'intérieur du corps de la fonction affectent l'objet à l'extérieur de la fonction (ce qui n'est pas le cas pour les réaffectations de variables). Cela est lié au fait que lors du passage d'arguments de l'extérieur à l'intérieur de la fonction, la "copie" de la variable qui est effectuée pose les mêmes problèmes - dans le cas des objets mutables - que ceux évoqués ci-dessus.

Les trois exemples ci-dessous sont à connaître, en particulier ce qui se passe dans le troisième où l'on mute l'argument passé à la fonction. La mutation affecte l'argument à l'extérieur de la fonction (on parle d'*effet de bord*).

[14]:
```
def reaffecter_sept_fois_v1(x):
    for _ in range(7):
        x = x + 1
    return x

x = 7
y = reaffecter_sept_fois_v1(x)
x, y
```

[14]:
```
(7, 14)
```

[15]:
```
def reaffecter_sept_fois_v2(tab):
    for _ in range(7):
        tab = tab + [1]
    return tab

tab_x = [7]
tab_y = reaffecter_sept_fois_v2(tab_x)
tab_x, tab_y
```

[15]:
```
([7], [7, 1, 1, 1, 1, 1, 1, 1])
```

[16]:
```
def muter_sept_fois(tab):
    for _ in range(7):
        tab.append(1)
    return tab

tab_x = [7]
tab_y = muter_sept_fois(tab_x)
tab_x, tab_y
```

[16]:
```
([7, 1, 1, 1, 1, 1, 1, 1], [7, 1, 1, 1, 1, 1, 1, 1])
```

3 III : Les matrices - partie 2 : initialisation d'une matrice

Pour initialiser une matrice - par exemple avec des zéros - il faut faire attention. La première idée qui vient en tête consiste à utiliser l'opérateur `*`. Nous avons vu que cet opérateur appliqué aux tableaux permet d'obtenir un tableau de taille `N` contenant `N` éléments identiques en effectuant une "copie" de l'élément indiqué :

[17]:
```
N = 15
mon_tab = [0] * N
mon_tab
```

[17]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

[18]:
```
mon_tab[5] = 777
mon_tab[8] = 999
mon_tab
```

[18]:
```
[0, 0, 0, 0, 0, 777, 0, 0, 999, 0, 0, 0, 0, 0, 0]
```

Pour une matrice on peut donc être tenté de faire :

[19]:
```
nb_col = 5
nb_lig = 8
ma_matrice = [[0] * nb_col] * nb_lig
ma_matrice
```

[19]:
```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

Le gros défaut de cette façon de faire est que l'objet `[0] * nb_col` - qui est un tableau - n'a pas été copié `nb_lig` fois : c'est son adresse qui a été copiée `nb_lig` fois. Par conséquent les mêmes problèmes que ceux évoqués dans le II vont survenir :

[20]:
```
ma_matrice[5][3] = 777
ma_matrice
```

[20]:
```
[[0, 0, 0, 777, 0],
 [0, 0, 0, 777, 0],
 [0, 0, 0, 777, 0],
 [0, 0, 0, 777, 0],
 [0, 0, 0, 777, 0],
 [0, 0, 0, 777, 0],
 [0, 0, 0, 777, 0],
```

```
 [0, 0, 0, 777, 0]]
```

Autrement dit toutes les lignes de la matrice désignent une seul eet même adresse mémoire : une modification sur une ligne entraîne donc une modification sur toutes les autres lignes.

Pour initialiser une matrice avec des zéros, on utilisera plutôt les listes définies en compréhension. En effet, avec les listes définies par compréhension, l'expression située avant le `for` est évaluée et créée à chaque itération. Il n'y a pas de problèmes de "copie" :

[21]:
```
ma_matrice = [[ 0 for _ in range(nb_col)] for _ in range(nb_lig)]
ma_matrice
```

[21]:
```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

[22]:
```
ma_matrice[5][3] = 777
ma_matrice
```

[22]:
```
[[0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 777, 0],
 [0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0]]
```

[ ]:
```

```