chapitre 5

Les Listes

Table des matières

I Définition et création

Définition : Un **tableau** est une structure de données composite constituée d'une séquence ordonnée d'éléments qui peuvent être des entiers, des flottants, des chaînes de caractères, des booléens ....

En Python, il n'existe pas de type tableau mais un type `list`. Comme un tableau, une **liste** Python est une **séquence ordonnée** d'objets qui peuvent être de types différents. De plus les listes sont des objets Python modifiables (on dit aussi mutable), en particulier leur taille peut évoluer.

Remarque : Dans certains langages (Java ...), tous les éléments du tableau doivent être de même type et il faut déclarer la taille du tableau avant de l'utiliser pour réserver la place en mémoire.

Définition d'une liste en extension.

En Python pour définir une liste on place les éléments entre des crochets en les séparant par de virgules :

Exemple 5.1.

```
>>> [42, "BCPST", 2<3, 3.14, [5,4]]
[42, 'BCPST', True, 3.14, [5, 4]]
>>> [4, 5, -1, 12]
[4, 5, -1, 12]
```

Définition d'une liste en compréhension.

En mathématique il est facile d'écrire l'ensemble des carrés des entiers de 1 à 50 sans tous les énumérer :

$$E = \{k^2,\ k \in [\![1, 50]\!]\}.$$

Cette écriture a une traduction immédiate en python : `L=[k**2 for k in range(1,51)]`

On peut donc aussi définir des listes d'une autre manière dite en compréhension.

Ces listes s'écrivent de la manière suivante :

[*expression* `for` *var* `in` *itérateur*]
[*expression* `for` *var* `in` *itérateur* `if` *condition*]

Exemple 5.2.

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [k**2 for k in range(1,12)] # les carrés de 1 à 11
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> [i for i in range(100) if i%13==0] # les multiples de 13 de 0 à 99
[0, 13, 26, 39, 52, 65, 78, 91]
>>> from random import randint
# randint(a,b) donne un entier au hasard dans |[a,b]|
>>> alea = [randint(0, 20) for _ in range(22)] # la variable n'est pas obligé de
    porter un nom si l'expression ne dépend pas de la variable
>>> alea
[17, 3, 13, 14, 11, 11, 6, 2, 4, 17, 17, 7, 13, 20, 14, 16, 9, 13, 6, 13, 5, 13]
```

Remarque : on peut aussi se servir directement de `range` et de la fonction `list()` si on veut une liste d'entiers consécutifs ou espacés d'un pas constant.

Exemple 5.3.

```
>>> liste=list(range(5,15))
>>> liste
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> listebis=list(range(5,15,2))
>>> listebis
[5, 7, 9, 11, 13]
```

Remarque : La fonction `list` permet de transformer un itérable en une liste

```
>>> list(range(2,8))      # En python 2 list est inutile
[2, 3, 4, 5, 6, 7]
>>> list("BCPST")
['B', 'C', 'P', 'S', 'T']
```

Ce qu'on peut retenir pour l'instant :

- ⋆ Dans une liste l'ordre des éléments compte `[1, 2, 3, 4]` n'est pas la même liste que `[1, 3, 2 ,4]`.
- ⋆ On peut créer des listes vides de la manière suivante `liste=[]` ou `liste=list()`. On s'en sert en particulier pour initialiser des variables qui peuvent évoluer lors de l'exécution de programmes.
- ⋆ Les éléments d'une liste ne sont pas forcément de même type.
- ⋆ En particulier on peut faire une liste de listes ce qui permet d'obtenir un tableau à deux dimensions. (*On traitera des exercices sur cela*)
- ⋆ Les listes sont des objets mutables, c'est-à-dire qu'on peut modifier une liste et que cette modification ne change pas la référence mémoire (cf. suite du cours).

Remarque : on s'interdira de nommer une liste simplement `l` qu'on peut confondre avec `1`. À ce propos, voici ce qu'on peut lire dans la documentation Python :

Names to Avoid : « Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead. »

Exercice 5.1 *Pour commencer*

Créer dans un fichier, les listes suivantes, qui pourront servir dans la suite du TP pour vérifier les fonctions qui auront été écrites :

```
t1 = [10, 30, 42, 2, 17, 5, 30, -20]
t2 = [i**2 for i in range(-3,6)]
t3 = [i**3 for i in range(1000) if (i % 5) in [0,2,4]]

t4 = [841.0]
for i in range(20):
    t4.append(t4[i]/3+28)
```

On verra un peu plus loin ce que signifie la commande `t4.append(t4[i]/3+28)`

II Manipulation des listes

2.1 Accès aux éléments et slicing

2.1.1 Accès aux éléments

Tout d'abord on peut connaitre la longueur d'une liste avec la fonction `len` .

Si `li` désigne une liste donnée de longueur `n`, Python numérote les éléments de la liste de `0 à n-1` . Dès lors, pour accéder à l'élément numéroté `k`, on utilise tout simplement la commande

`li[k]`

L'utilisation d'un indice négatif permet d'accéder aux éléments à partir de la fin de la liste. Ainsi `li[-1]` désigne le dernier élément, `li[-2]` l'avant dernier, *etc.*

```
>>> li=[3,4,-5,-6,7,8,9,10,21,22,23]
>>> len(li)
11
>>> li[5]
8
>>> li[12]
Traceback ...
IndexError: list index out of range
>>> li[-1]
23
```

Exercice 5.2 Quelles sont les longueurs des listes créées dans l'exercice précédent ?

Exercice 5.3 Écrire une fonction réalisant l'échange de deux valeurs dans un tableau. Cette fonction recevra comme arguments un tableau et deux indices (distincts) correspondant à des positions réelles dans ce tableau, et effectuera l'échange.

```
>>> t = [10, 42, 1, 3, 5, 7, 1515, -10]
>>> echange(t, 1, 6)
>>> t
[10, 1515, 1, 3, 5, 7, 42, -10]
```

2.1.2 Le cas des listes de listes

La liste `[[1, 2, 3],[4, 5, 6]]` représente la matrice $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$. Comme précédemment pour accéder aux éléments de cette liste, la numérotation commence à 0.

```
>>> M=[[1, 2, 3],[4, 5, 6]]
>>> len(M)
2
>>> M[1]
[4, 5, 6]
>>> M[0][2]
3
```

Exercice 5.4 Écrire une fonction `dimensions` qui prend en entrée une matrice (un tableau à deux dimensions) et retourne le nombre de lignes et le nombre de colonnes de cette matrice.

```
>>> m1
[[0, 0], [0, 0], [0, 0]]
>>> dimensions(m1)
(3, 2)
```

2.1.3 Slicing

Pour faire du slicing (qui signifie plus ou moins découpage en tranche) on opère comme lors de l'utilisation de `range`. La commande `li[i:j]` donne la liste des éléments de `li` dont les indices sont entre `i` et `j-1`. Plus généralement `li[i:j:p]` va construire la liste des éléments `li[k]` pour `k` partant de `i` avec un pas de `p` et ne prenant que des valeurs strictement inférieures à `j`. On peut aussi utiliser des indices négatifs.

```
>>> li=[3,4,-5,-6,7,8,9,10,21,22,23]
>>> li[2:6]
[-5, -6, 7, 8]
>>> li[2:9:2]     # on découpe li de 2 à 9-1 de 2 en 2
[-5, 7, 9, 21]
>>> li[:5]        # les 5 premiers éléments
[3, 4, -5, -6, 7]
>>> li[-4:]       # les 4 derniers
[10, 21, 22, 23]
>>> li[1:-2]
[4,-5,-6,7,8,9,10,21]
```

Exercice 5.5 *Un peu de slicing*

1. Créer la liste des dix derniers éléments de `t3`.
2. Créer la liste constituée des éléments de `t3` sauf les 250 premiers et les 250 derniers.
3. Créer la liste constituée des cinq premiers éléments de `t4` suivie des cinq derniers de cette même liste.

2.2 Modification, ajout et suppression d'éléments

2.2.1 Modification

Une liste étant un objet mutable, on peut modifier des éléments d'une liste. On procède comme dans l'exemple.

```
>>> L=[42, "BCPST", 2<3, 3.14, [5,4]]
>>> L[2]=852
>>> L
[42, "BCPST", 852, 3.14, [5,4]]
```

2.2.2 Ajout

Pour ajouter un ou des éléments à une liste. Il existe pour cela ce qu'on appelle des méthodes, qui s'appliquent aux listes. Elles s'utilisent toutes de la même manière : `nom_liste.methode`

```
>>> li=[5,6,9,12]
>>> li.append(-14)       # La méthode append ajoute un élément en fin de liste
>>> li
[5, 6, 9, 12, -14]
>>> li.extend([12,0,-3]) # ajoute en fin de liste les éléments 12,0 et -3
>>> li
[5, 6, 9, 12, -14, 12, 0, -3]
>>> li.insert(1,"Ici")   # insert ajoute un élément à n'importe quelle position
>>> li
[5, 'Ici', 6, 9, 12, -14, 12, 0, -3]
```

En pratique on se servira essentiellement de `append`, notamment pour construire petit à petit une liste de la manière suivante :

```
>>> li=[]      # on crée une liste vide
>>> for i in range(4,15):
        li.append(i**2)
>>> li
[16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

2.2.3 Suppression

On peut supprimer un ou des éléments d'une liste avec `del`

```
>>> L=[42, "BCPST", 2<3, 3.14, [5,4]]
>>> del L[2]      # ou del(L[2])
>>> L
[42, "BCPST", 3.14, [5,4]]
>>> li=[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> del li[2:6]
>>> li
[0, 1, 36, 49, 64, 81, 100]
```

Une autre façon est d'utiliser la méthode `pop`. (`li.pop(indice)` *Enlève et renvoie l'objet situé à la place indice*),

```
>>> L=[42, "BCPST", 2<3, 3.14, [5,4]]
>>> L.pop(1)
"BCPST"
>>> L
[42, 2<3, 3.14, [5,4]]
```

2.3 Les listes, des objets itérables

On peut parcourir une liste avec une boucle `for` (on dit qu'une liste est itérable) :

Parcours par valeurs

```
>>> L=[42, "BCPST", 2<3, 3.14, [5,4]]
>>> for x in L:
        print(x)
42
BCPST
True
3.14
[5, 4]
```

Parcours par indices

```
>>> L=[42, "BCPST", 2<3, 3.14, [5,4]]
>>> for i in range(len(L)):
        print(L[i])
42
BCPST
True
3.14
[5, 4]
```

On aura donc toujours deux manières de parcourir une liste suivant que l'on s'intéresse aux éléments de la liste (avec `for x in L`) ou que l'on s'intéresse aux éléments de la liste mais aussi à leur indice.

Exercice 5.6 *Devinette 1*

« Deviner » les valeurs des listes `t5`, `t6`, et `t7` après avoir exécuté les commandes suivantes :

```
1 t5 = [t2[2*i + 1] for i in range(3)]
2 t6 = [x**2 for x in t2]
3 t7 = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
```

Taper ces commandes et vérifier.

Exercice 5.7 *Création de liste en compréhension*

1. Créer la liste constituée des termes d'indice pair de `t1`
2. Créer la liste constituée des termes pairs de `t1`

Exercice 5.8

1. Écrire une fonction `somme` calculant la somme des éléments d'un tableau. Bien entendu, une telle fonction existe déjà en Python, elle se nomme `sum`, l'idée est de la reprogrammer.

```
def somme(tab):
....
>>> somme(t1)
116
```

2. Écrire une fonction `moyenne` calculant la moyenne des éléments d'un tableau, puis une autre `ecart_type` calculant l'écart-type.
   Si T est la liste, les formules sont les suivantes, où $|T|$ désigne la longueur de T.

$$\mu = \frac{1}{|T|}\sum_{x\in T} x, \qquad \sigma = \left(\frac{1}{|T|}\sum_{x\in T}(x-\mu)^2\right)^{1/2} = \left(\left(\frac{1}{|T|}\sum_{x\in T} x^2\right) - \mu^2\right)^{1/2}$$

```
>>> moyenne(t2)
7.666666666666667
>>> ecart_type(t2)
7.803133273813083
```

Exercice 5.9

1. Écrire une fonction `maximum` calculant le maximum des éléments d'un tableau.
2. Écrire une fonction renvoyant une position de ce maximum. (*Dans un premier temps on utilisera des inégalités strictes, puis on modifiera en utilisant ensuite des inégalités au sens large. On testera ensuite la différence sur la liste* `t1`)

```
>>> maximum(t1)
42
>>> position_maximum(t1) # inégalités strictes
2
```

3. Écrire une fonction renvoyant le maximum et la liste des positions où ce maximum est atteint.

Exercice 5.10 Écrire une fonction `croissante` qui détermine si les éléments d'une liste sont rangés par ordre croissant

Exercice 5.11 Soit (u_n) la suite définie par $u_0 = 1$ et $u_{n+1} = \sum_{k=0}^{n} (n+k)u_k$.

Écrire une fonction donnant le terme de rang `n` de cette suite.

indic. Pour calculer un terme de la suite, il faut avoir mémorisé tous les termes précédents. Pour stocker les valeurs de la suite déjà calculées, on utilisera une liste.

2.4 Pour compléter

⋆ On peut tester l'égalité de deux listes avec `==`.

⋆ On peut tester si un élément est présent ou non dans une liste avec la commande `in`.

⋆ Si `L` et `M` sont deux listes alors `L+M` permet de concaténer les deux listes.
Attention comme on le verra plus loin, ce n'est pas exactement la même chose que de rajouter la liste `M` à la liste `L` avec la méthode `extend`.

```
>>> M=[1,2,3,4]
>>> M==[4,1,2,3]
False
>>> 5 in M
False
>>> 3 in M
True
>>> L=[5,6,7,8]
>>> M+L
[1, 2, 3, 4, 5, 6, 7, 8]
```

⋆ Il existe d'autres méthodes sur les listes. On peut accéder à toute celles-ci en tapant `dir(list)`. Par exemple :
- `index` (`li.index('a')` *index de la première occurrence de* `'a'` *dans* `li`),
- `count` (`li.count('a')` *compte les occurrences de* `'a'` *dans* `li`),
- `reverse` (`li.reverse` *inverse la liste*),
- `sort` (`li.sort()` *tri la liste* `li`). On apprendra un peu plus tard à trier les listes tout seul.

Elles s'utilisent comme dans l'exemple : `nom_liste.methode`

```
>>> L=[5,6,7,8]
>>> L.reverse()
>>> L
[8, 7, 6, 5]
```

⋆ Comme on l'a évoqué dans le chapitre sur les fonctions, `map(f,L)` calcule la liste obtenue en appliquant la fonction `f` à tous les éléments de `L` (attention, en Python 3, on obtient en fait un itérateur qu'il faut ensuite convertir en liste avec la fonction `list`)

```
>>> def f(x):
        return x**2+2

>>> L=[1, 2, 3]
>>> map(f,L)
<map at 0x4eba7f0>
>>> list(map(f, L))
[3, 6, 11]
```

Exercice 5.12 L'idée de cet exercice est de reprogrammé certaines des méthodes qui ont été données plus haut. On s'interdira donc de les utiliser pour répondre aux questions.

1. Écrire une fonction `compartab` qui teste si deux tableaux sont égaux.

```
>>> compartab(t1,t2)
False
```

2. Écrire une fonction `inversion` qui inverse l'ordre des éléments d'un tableau.

```
>>> inversion(t1)
[-20, 30, 5, 17, 2, 42, 30, 10]
```

3. Écrire une fonction `remplace` qui remplace dans un tableau toutes les occurrences de `x` par `y`. (*par exemple qui remplace tous les 4 par des 6*). Tester sur le tableau `alea` dont la construction est donnée ci-dessous.

```
from random import randint
alea = [randint(0, 20) for _ in range(100)]
```

III Référencement et copie de listes

3.1 Référencement/aliasing

Avant de continuer l'étude des listes, revenons un instant aux variables et revenons à quelque chose que nous avons commencé à voir dans les premiers chapitres. En Python, une variable est un nom et une adresse. Il s'agit de l'adresse mémoire où se trouve la valeur de la variable.

```
>>> a = 1
>>> id(a)
1382195600
>>> a = 2
>>> id(a)
1382195632
>>> a = a + 3
>>> a
5
```

```
>>> id(a)
1382195728
>>> a += 8
>>> a
13
>>> id(a)
1382195984
>>> b = a
>>> id(a), id(b)
(1382195984,
     1382195984)
>>> a = a + 2
>>> (a, b)
(15, 13)
>>> id(a), id(b)
(1382196048,
     1382195984)
```

Regardons maintenant ce qu'il se passe pour les listes

```
>>> L = [8, 5, 2]
>>> id(L)
85948296
>>> L[0] = 9          # il y a modification de la liste elle-même
>>> L
[9, 5, 2]
>>> id(L)
85948296              # Le référencement est le même
>>> L.append(18)      # L.append(a) opère directement sur la liste
>>> L
[9, 5, 2, 18]
>>> id(L)
85948296              # Le référencement est le même
>>> L = L + [12]      # L = L + [a] créé une nouvelle liste
>>> L
[9, 5, 2, 18, 12]
>>> id(L)
86468104              # Le référencement change
>>> L += [2014]       # L += [a] opère directement sur la liste
>>> L
[9, 5, 2, 18, 12, 2014]
>>> id(L)
86468104              # Le référencement est le même
```

A l'inverse des objets de type `int, float, bool` ou `str` les objets de type `list` sont mutables. En particulier l'application d'une méthode à une liste change l'objet sur place (le référencement ne change pas) sans créer un nouvel objet.

La raison pour laquelle Python agit ainsi est qu'il évidemment moins couteux de travailler directement sur une liste en lui rajoutant un élément plutôt que de recopier toute la liste pour en créer une nouvelle.

3.2 Liste et fonctions

Nous avons déjà parlé de ce qu'il se passe lorsqu'on applique une fonction à une liste dans un chapitre précédent, mais il est le bon de le rappeler ici.

Exemple 5.4.

```
1 def ajoute(L, x):
2     L.append(x)
3     return L
```

```
>>> L = [8, 5, 2]
>>> ajoute(L, 42)
[8, 5, 2, 42]
>>> L
[8, 5, 2, 42]
```

Comme on le voit, cette fonction est à effet de bord : elle modifie la liste passée en paramètre. Dans le cas des objets mutables, ils ne sont pas passés par valeur mais par adresse et peuvent donc être modifiés par l'exécution d'une fonction (contrairement à ce que l'on a vu pour les arguments de type simple).

3.3 Copie de listes

Commençons par un exemple

```
>>> li1=[5,2,9]
>>> li2=li1
>>> id(li1),id(li2)
(85949576, 85949576)
>>> li1[2]=10
```

```
>>> li1
[5, 2, 10]
>>> li2
[5, 2, 10]
>>> id(li1),id(li2)
(85949576, 85949576)
```

On constate que pour des listes (et des objets mutables en général), l'affectation `li2 = li1` ne s'exécute pas comme pour les types simples : il n'y a pas évaluation puis affectation mais création d'une nouvelle variable qui pointe vers la même adresse. C'est l'adresse de `li1` qui est utilisée et pas sa valeur.

Si l'on décide alors de muter le dernier élément de la liste `li1` en écrivant `li1[2] = 10`, la liste devient `[5,2,10]` mais son adresse ne change pas en mémoire. Les variables `li1` et `li2` pointant vers la même adresse, si on demande `li2`, on obtient `[5,2,10]`. Tout se passe donc comme si la mutation de `li1` s'était répercutée sur `li2`.

Il existe plusieurs façons pour « bien » copier une liste.

À la main

```
>>> li1=[8, 5, 2]
>>> li2=[]
>>> for i in li1:
        li2.append(i)
>>> li2
[8, 5, 2]
>>> li1[0]=10
>>> li1, li2
([10, 5, 2], [8, 5, 2])
```

Par slicing

```
>>> li1=[8, 5, 2]
>>> li2=li1[:]
>>> li2
[8, 5, 2]
>>> id(li1),id(li2)
(85808072, 85948168)
>>> li1[0]=10
>>> li1, li2
([10, 5, 2], [8, 5, 2])
```

Avec la fonction `list`

```
>>> li1=[8, 5, 2]
>>> li2=list(li1)
>>> li2
[8, 5, 2]
>>> id(li1),id(li2)
(85912328, 86458760)
>>> li1[0]=10
>>> li1, li2
([10, 5, 2], [8, 5, 2])
```

Avec la fonction `copy` du module `copy`

```
>>> li1=[8, 5, 2]
>>> from copy import *
>>> li2=copy(li1)
>>> li2
[8, 5, 2]
>>> id(li1),id(li2)
(86366920, 86461576)
>>> li1[0]=10
>>> li1, li2
([10, 5, 2], [8, 5, 2])
```

On pourra retenir la syntaxe `li2 = li1[:]` qui semble la plus simple.

Pour voir la différence lors de l'exécution, je vous invite à taper les quelques lignes de commandes qui suivent sur le site suivant : http://pythontutor.com/visualize.html

Méthode 1

```
>>> li1=[5,2,9]
>>> li2=li1
>>> li1[2]=10
```

Méthode 2

```
>>> li1=[5,2,9]
>>> li2=li1[:]
>>> li1[2]=10
```

Exercice 5.13 *Devinette 2*

« Deviner » les valeurs des listes `k,t,m,n` après l'exécution de ces lignes :

```
1 k = [10, 15, 12]
2 t = k
3 m = t
4 n = m[:]
5 m[1] = 17
6 n[0] = 19
```

Taper ces commandes et vérifier.
Aller sur http://pythontutor.com/visualize.html entrer le code précédent, et visualiser pas à pas l'exécution

Pour être un peu plus complet, la commande `li2=li1[:]` ne suffira pas totalement si la liste `li1` contient elle même une (des) liste(s). Encore une fois on comprend très bien ce qui se passe si vous allez sur le site évoqué plus haut, en tapant ces lignes de commande :

```
li1=[5,[3,4],-12]
li2=li1[:]
li1[1][1]=10
```

Essayons malgré tout d'expliquer ici le phénomène.

```
>>> li1=[5,[3,4],-12]
>>> li2=li1[:]
>>> id(li1[1]),id(li2[1])
(49744200, 49744200)
>>> # les deux sous listes [3,4] de li1 et li2 pointent vers le même endroit
>>> li1[1][1]=10      # on change la valeur dans la deuxième colonne deuxième
    ligne (le 4 est remplacé par 10)
>>> li1,li2           # les deux listes sont changées
([5, [3, 10], -12], [5, [3, 10], -12])
```

Dans le cas où on a des listes dans les listes, pour réaliser une vraie copie, on utilisera la fonction `deepcopy` du module `copy`.

```
>>> from copy import deepcopy
>>> li1=[5,[3,4],-12]
>>> li2=deepcopy(li1)
>>> li1[1][1]=10
>>> li1,li2
([5, [3, 10], -12], [5, [3, 4], -12])
```

3.4 Le cas des listes de listes

Comme on vient de la voir, il faut faire un peu attention aux problèmes d'aliasing dès que l'on travaille avec des listes de listes. Si l'on veut par exemple définir une matrice $J = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$, il y a *a priori* plusieurs manières de procéder :

```
>>> t = [1, 1, 1]
>>> J1 = [t, t, t]
>>> J2 = [t for i in range(3)]
>>> J3 = [[1 for i in range(3)] for j in range(3)]
>>> J4 = [[1]*3]*3
>>> J5 = [[1]*3 for j in range(3)]
>>> J1
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
# On obtiendrait exactement la même chose pour J2, J3, J4 et J5
```

Tant que l'on n'utilise ces matrices qu'en lecture (c'est-à-dire tant qu'on ne les modifie pas), il n'y a aucun problème. Si par contre on veut modifier l'un des coefficients, tout se complique :

```
>>> J1[0][0] = 3
>>> J1
[[3, 1, 1], [3, 1, 1], [3, 1, 1]]
>>> J2
[[3, 1, 1], [3, 1, 1], [3, 1, 1]]
# En faisant J1[0][0] = 3, on a modifié J1[0], c'est-à -dire t.
# Mais les deuxième et troisième lignes de J1 et J2
# sont aussi égales à t, elles ont donc également été modifiées.
>>> J3
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
# J3 (comme J4 et J5) n'a rien à voir avec t, elle est donc restée inchangée.
>>> J3[0][0] = 4
>>> J3
[[4, 1, 1], [1, 1, 1], [1, 1, 1]]
# C'est plus raisonnable.
>>> J4[0][0] = 8
>>> J4
[[8, 1, 1], [8, 1, 1], [8, 1, 1]]
# Les trois lignes de J4 "pointent" vers la même liste : si l'on en modifie une,
# on modifie les trois.
>>> J5[0][0] = 9
>>> J5
[[9, 1, 1], [1, 1, 1], [1, 1, 1]]
# Pour J5, le problème ne se pose pas.
```

Le meilleur moyen de ne jamais se tromper, c'est de systématiquement utiliser la même technique que pour définir `J3`.

Exercice 5.14 Sur le principe de la construction de `J3` écrire une fonction `zeros(n,p)` qui retourne la matrice nulle de $\mathcal{M}_{n,p}(\mathbb{R})$.

Exercice 5.15 Écrire une fonction `copie` qui réalise la copie d'une matrice. (*On s'interdira d'utiliser la fonction* `deepcopy` *du module* `copy`)

```
1 >>> m1=zeros(3,2)
2 >>> m2 = copie(m1)
3 >>> m1[1][1] = 3
4 >>> m1, m2
5 ([[0, 0], [0, 3], [0, 0]], [[0, 0], [0, 0], [0, 0]])
```

IV D'autres exercices

Exercice 5.16 *Opérations sur les matrices.*

1. Écrire une fonction réalisant l'addition de deux matrices (de mêmes dimensions).
2. Écrire une fonction réalisant la multiplication d'une matrice par un scalaire.
3. Écrire une fonction renvoyant la transposée d'une matrice donnée en paramètre.
4. Écrire une fonction réalisant la multiplication de deux matrices de dimensions compatibles.
5. Écrire une fonction réalisant le calcul de A^q, avec $A \in \mathcal{M}_n(\mathbb{R})$ et $q \in \mathbb{N}$ donnés.
6. Écrire une fonction testant le caractère symétrique d'une matrice (et renvoyant donc un booléen).

Exercice 5.17 Écrire une fonction `somme_vect(t1,t2)` qui renvoie la somme de `t1` et `t2` (somme élément par élément). La fonction commencera par vérifier que les deux listes ont la même longueur, et renverra une erreur si ce n'est pas le cas.

Exercice 5.18

1. Écrire une fonction `sommes_cumulees(t)` qui renvoie le tableau des sommes cumulées de `t`. On doit avoir :

```
>>> sommes_cumulees(t1)
[10, 40, 82, 84, 101, 106, 136, 116]
```

2. Si `t` est de longueur n, quel sera l'ordre de grandeur du nombre d'additions effectuées par votre fonction (n, n^2, $n!$...) ? Si ce n'est pas déjà le cas, essayez de trouver une version faisant environ n additions.

Exercice 5.19

1. Écrire une fonction `deux_plus_gros(t)` qui renvoie les deux plus grands éléments du tableau `t` (on supposera que `t` a au moins deux éléments).
2. La question précédente est imprécise : qu'est censée faire la fonction si on lui donne un tableau dans lequel le maximum apparaît plusieurs fois ? Pour `[12, 4, 12, 1, 7, 3, 12, 5]`, on peut raisonnablement renvoyer `(12, 12)` ou `(12, 7)`. Déterminer ce que ferait votre fonction sur cet exemple, et écrire une nouvelle version qui ait l'autre comportement.
3. Que ce passe-t-il si tous les éléments du tableau sont les mêmes.

Exercice 5.20

1. Écrire une fonction `permuteligne` qui permute deux lignes d'une matrice. (*paramètres d'entrée : la matrice et les deux numéros des lignes*)
2. Écrire une fonction `rempl` qui remplace dans une matrice `A` toutes les occurrences de `x` par `y`. (*par exemple qui remplace tous les 4 par des 6*)

Exercice 5.21 Écrire une fonction `pascal` qui récite les `n` premières lignes du triangle de Pascal.

Remarque : : On évitera d'utiliser la formule générale de $\binom{n}{k}$, on utilisera plutôt $\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$ pour calculer les termes d'une ligne à l'autre.