Rappels Python - Intro Matrices

October 7, 2021

1 "Rappels" Python : listes, boucle for, parcours d'objets itérables, représentations de matrices par des listes de listes...

1.1 PYTHON

Langage de programmation objet interprété, placé sous licence libre et fonctionnant sur la plupart des plates-formes. Il s'est beaucoup développé dans le monde scientifique, grâce à ses nombreuses bibliothèques destinées au calcul numérique (*numpy,Scipy, Matplotlib*). Vous pouvez télécharger gratuitement Python sur Python.org ou choisir d'installer une distribution packagée comme *anaconda* contenant les outils essentiels au calcul scientifique (https://www.anaconda.com/download/). Un peu plus tard, nous utiliserons l'environnement de développement *Spyder*, présent dans la suite Anaconda.

1.2 Jupyter

Jupyter NoteBook est une application web qui permet de créer et de partager des documents qui contiennent du code Python et du texte. Vous pouvez, dans chaque cellule de code, taper le nombre de lignes que vous souhaitez. Pour valider l'ensemble des instructions contenues dans une cellule, **SHIFT + ENTER** ou le bouton **Exécuter** !
Par moments, dans ce Notebook, vous trouverez des cellules **je teste** pour laisser libre cours à votre imagination, mais il est aussi possible d'ajouter des **cellules** de codes ou de textes à tout moment dans votre notebook. Ou de modifier les cellules existantes.

2 Listes et tuples sous Python

Sous Python, les listes et les tuples sont des suites d'objets. Ces objets peuvent être de n'importe quel types et il peut y avoir des types différents dans une même liste ou dans un même tuple.

[1]:
```
liste1 = [3,6,8,1,-4,7]
```

[2]:
```
print(liste1)
```

```
[3, 6, 8, 1, -4, 7]
```

Remarque : mode interactif Dans cet environnement, l'interpréteur Python est en mode interactif. Il fonctionne un peu comme une calculette... Les expressions sont évaluées **et affichées**. L'usage du **print** dans ce mode n'est donc pas nécessaire.

[3]:
```
liste1
```

[3]:
```
[3, 6, 8, 1, -4, 7]
```

[4]:
```
tuple1=(4,5,'a')
```

[5]:
```
tuple1
```

[5]:
```
(4, 5, 'a')
```

[6]:
```
liste2 = [1,'a',3,[4,7]]
liste2
```

[6]:
```
[1, 'a', 3, [4, 7]]
```

2.1 Séquence

Sous Python, on utilise le terme de **séquence** lorsque les éléments d'une suite sont accessibles par un indice : * indice 0 pour le premier élément, 1 pour le deuxième,..., n-1 pour le n-ième * indice -1 pour le dernier élément, -2 pour l'avant dernier....

Il est possible de définire des **tranches** (slice) d'indices et précisant éventuellement un **pas** : * [0:4], de l'élément d'indice 0 à celui d'indice 3 (le dernier élément n'est pas inclus) : [0:4] est équivalent à [0,1,2,3] * [10:18:2], de l'élément d'indice 10 à l'élément d'indice 17 avec un pas de 2 : [10:18:2] est équivalent à [10,12,14,16]

Les listes, les tuples et même les chaînes de caractères sont des séquences.
Les dictionnaires, que nous verrons un peu plus tard, ne sont pas des séquences, leurs éléments ne sont pas accessibles par un indice.

[8]:
```
liste1[2]
```

[8]:
```
8
```

[9]:
```
liste1[3:8:2]
```

[9]:
```
[1, 7]
```

2.1.1 Fonction len

Très importante pour parcourir une séquence, la fonction **len** renvoie son nombre d'éléments.

[10]:
```
len(liste1)
```

[10]:
```
6
```

Remarque : chaînes de caractères Sous Python, les chaînes de caractères sont aussi des séquences puisqu'on peut accéder à chaque caractère d'une chaîne en utlisant des indices

[11]:
```
ch='il fait beau'
```

[12]:
```
ch[3]
```

[12]:
```
'f'
```

[13]:
```
ch[0:11:2]
```

[13]:
```
'i atba'
```

2.2 Mutables ou non mutables ?

Les listes, à la différence des tuples sont **mutables**.

Dans un objet **mutable** : * on peut ajouter ou enlever des éléments * les éléments peuvent être réaffectés

Ajout d'un élément : append()

[14]:
```
liste1
```

[14]:
```
[3, 6, 8, 1, -4, 7]
```

[15]:
```
liste1.append(5)
liste1
```

[15]:
```
[3, 6, 8, 1, -4, 7, 5]
```

Modification d'un élément

[16]:
```
liste1[2]=0
```

[17]:
```
liste1
```

[17]:
```
[3, 6, 0, 1, -4, 7, 5]
```

[18]:
```
tuple1[0]
```

[18]:
```
4
```

[19]:
```
tuple1[0]=2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-c4be70ce3261> in <module>
----> 1 tuple1[0]=2

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

2.3 Manipulation de listes : définition et méthodes

Nous avons vu dans la première partie du cours que les ensembles pouvaient être définis **en extension** ou **en compréhension** * En extension, c'est-à-dire en énumérant ses éléments: $A = \{2, 4, 6, 8, 10\}$ * En compréhension, en décrivant une propriété commune aux éléments de l'ensemble : $A = \{2i \ / \ i \in \{1, ..., 5\}\}$

On peut faire de même pour définir une liste en Python.

Définition d'une liste en extension

[20]:
```
liste2=[2,4,6,8,10]
liste2
```

[20]:
```
[2, 4, 6, 8, 10]
```

Définition d'une liste en compréhension Pour ceux qui n'ont pas experts en Python, vous trouverez des précisions sur la **boucle for** à la suite du paragraphe sur les méthodes.

[21]:
```
liste3=[2*i for i in range(1,6)]
liste3
```

[21]:
```
[2, 4, 6, 8, 10]
```

[22]:
```
liste4=[2*i for i in range(6)]
liste4
```

[22]:
```
[0, 2, 4, 6, 8, 10]
```

[23]:
```
liste5=[i for i in range(2,11,2)]
liste5
```

[23]:
```
[2, 4, 6, 8, 10]
```

Remarque : initialisation d'une liste Sous Python, il n'est pas nécessaire de déclarer les varaibles que l'on manipule. La déclaration et l'initialisation se font en même temps.
Quand nous saisissons l'instruction **x=3**, Python "devine" que la variable **x** est de type entier, lui alloue l'espace mémoire correspondant et assigne la valeur 3 à la variable x (c'est une des caractéristiques des langages de haut niveau).
Python ne peut cependant pas tout deviner...Si vous souhaitez manipuler une liste **liste**, vous ne pouvez pas directement lui affecter des valeurs sans l'avoir au préalablement initialisée :

[24]:
```
liste[0]=1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-479436e3f131> in <module>
----> 1 liste[0]=1

NameError: name 'liste' is not defined
```

Ci-dessous, quelques exemples d'initialisations de listes. Nous avons choisi de leur affecter des valeurs "vides" (**None**), mais nous aurions pu, bien sûr, utiliser n'importe quelle autre valeur (0, -1,"a"...)

[25]:
```
liste6=[None]*10
liste6
```

[25]:
```
[None, None, None, None, None, None, None, None, None, None]
```

[26]:
```
liste7=[None for i in range(10)]
liste7
```

[26]:
```
[None, None, None, None, None, None, None, None, None, None]
```

[27]:
```
liste8=[]
for i in range(10):
    liste8.append(None)
liste8
```

[27]:
```
[None, None, None, None, None, None, None, None, None, None]
```

2.3.1 Méthodes sur les listes (ces méthodes sont rappelées dans le "Memento Python")

- **list1.append(x)** : ajoute l'élément x à la fin de la liste *list1*
- **list1.insert(i, x)** : insère, dans *list1*, un élément à la position indiquée (i est la position de l'élément avant lequel l'insertion doit s'effectuer)
- **list.remove(x)** : supprime de la liste le premier élément dont la valeur est égale à x
- **list.pop(i)** : enlève de la liste l'élément situé à la position indiquée et **le renvoie** en valeur de retour
- **list.clear()** : supprime tous les éléments de la liste
- **list.index(x[, start[, end]])** : renvoie la position du premier élément de la liste dont la valeur égale x (en commençant par zéro)
- **list.count(x)** : renvoie le nombre d'éléments ayant la valeur x dans la liste
- **list.sort(key=None, reverse=False)** : classe les éléments sur place (les arguments peuvent personnaliser le classement, voir sorted() pour leur explication)
- **list.reverse()** : inverse l'ordre des éléments de la liste, sur place
- **list1.copy()** : renvoie une copie superficielle de la liste *list*
- **list(list1)** : renvoie une copie profonde de la liste *list1*

[28]:
```
liste = ["a", "b", "c","a"]
liste.remove("a")
liste
```

[28]:
```
['b', 'c', 'a']
```

[29]:
```
liste.reverse()
liste
```

[29]:
```
['a', 'c', 'b']
```

[30]:
```
x=liste.pop(0)
```

[31]:
```
x
```

[31]:
```
'a'
```

[32]:
```
liste
```

[32]:
```
['c', 'b']
```

[33]:
```
y=liste.pop(-1)
```

[34]:
```
liste
```

[34]:
```
['c']
```

[35]:
```
liste = ["a","a","a","b","c","c"]
liste.count("a")
```

[35]: 3

[36]:
```
liste.insert(2, "b")
liste
```

[36]: ['a', 'a', 'b', 'a', 'b', 'c', 'c']

2.3.2 Opérations sur les listes

[37]:
```
[1, 4] + [4]
```

[37]: [1, 4, 4]

[38]:
```
[5, 2] * 4
```

[38]: [5, 2, 5, 2, 5, 2, 5, 2]

[39]:
```
[1, 4] == [2,4]
```

[39]: False

3 Boucle bornée (boucle for) - Parcours de listes

La boucle for est utilisée lorsque le nombre de répétitons est déterminé à l'avance.
Syntaxe basique
$ for i in range(n):
1cminstruction1
1cminstruction2$

for i in range(n) : la variable i prend successivement les valeurs 0,1,2...jusqu'à n-1. Et pour chacune des valeurs de i. le bloc d'instructions de la boucle est exécuté.
Il n'y a pas de marque de début et de fin pour définir ce bloc d'instruction, celui-ci est déterminé par **l'indentation** (décalage d'une tabulation ou de 4 espaces par rapport à la position du for).
Il est bien sûr possible faire parcourir à i des listes qui ne commencent pas par zéros ou des entiers non consécutifs : * **for i in range(2,10)** : i prend toutes les valeurs de 2 à 9 * **for i in range(2,10,3)** : i prend les valeurs 2,5,8

[40]:
```
for i in range(10,30,4):
    print(i)
```

```
10
14
18
22
26
```

3.1 Objets itérables

Les objets **iterables** sont des objets qui peuvent être parcourus avec une boucle for. Les objets qui sont des **séquences**, c'est-à-dire constitués d'éléments indicés, comme les listes, les tuples ou les chaînes de caractères sont **itérables**. Nous verrons plus tard que les dictionnaires, qui ne sont pas des séquences, sont quand même itérables.
Nous allons ici donner quelques exemples de parcours de listes.

Parcours des éléments d'une liste

[41]:
```
liste=[9,6,4,-1,,2]
```

[42]:
```
for val in liste:
    print(val)
```

```
9
6
4
-1
2
```

Parcours des éléments d'une liste en utilisant leurs indices

[43]:
```
len(liste)
```

[43]: 5

[44]:
```
for i in range(len(liste)):
    print(liste[i])
```

```
9
6
4
-1
2
```

Une erreur et un message d'erreur classique ("un entier n'est pas itérable"):

[45]:
```
for i in len(liste):
    print(liste)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-027ab6710d3f> in <module>
----> 1 for i in len(liste):
      2     print(liste)

TypeError: 'int' object is not iterable
```

4 Applications : représentations de matrices par des listes de listes

Dans le première partie de la ressource **R1-07 : Outils mathématiques fondamentaux**, nous abordons la notion de matrices réelles, très utiles en informatique, qui sont tout simplement des tableaux de nombres.

Ci-dessous quelques exemples de matrices :

$$M_1 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix}, M_2 = \begin{pmatrix} 5 & 3 \\ 2 & 4 \\ -5 & 1 \end{pmatrix}, M_3 = \begin{pmatrix} 1 & 0 & 1 \\ 2 & -5 & 3 \\ 4 & 7 & 2 \end{pmatrix} \text{ et } M_4 = \begin{pmatrix} 2 & 1 & -2 \\ 0 & 2 & 4 \end{pmatrix}$$

M_1 a 2 lignes et 3 colonnes, M_2 a 3 lignes et 2 colonnes, M_3 a 3 lignes et 3 colonnes, M_4 a 2 lignes et 3 colonnes.

Pour représenter une matrice sous Python, nous pouvons utiliser une liste de listes, chaque élément de la liste correspondant à une ligne.
La représentation la plus usuelles d'une matrice n'est pas la liste de liste mais le tableau (**array**) que nous trouverons un peu plus tard dans la bibliothèque **numpy**. Pour l'instant, la liste de listes nous permet de bien comprendre les opérations sur les matrices et la manipulation des indices.

[46]:
```
M1=[[1,3,-1],[2,0,4]]
M1
```

[46]:
```
[[1, 3, -1], [2, 0, 4]]
```

[47]:
```
M2=[[5,3],[2,4],[-5,1]]
M2
```

[47]:
```
[[5, 3], [2, 4], [-5, 1]]
```

[48]:
```
M3=[[1,0,1],[2,-5,3],[4,7,2]]
M3
```

[48]:
```
[[1, 0, 1], [2, -5, 3], [4, 7, 2]]
```

[49]:
```
M4=[[2,1,-2],[0,2,4]]
M4
```

[49]:
```
[[2, 1, -2], [0, 2, 4]]
```

M étant une liste de liste représentant une matrice, comment afficher le contenu de sa 2ème ligne, sous forme de liste ? Par exemple, pour $M_1 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix}$, il faudrait afficher la liste [2,0,4].

[50]:
```
M1[1]
```

[50]:
```
[2, 0, 4]
```

M étant une liste de liste représentant une matrice, comment afficher le terme de la 2ème ligne et 1ère colonne de cette matrice ? Par exemple, pour $M_1 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix}$, il faudrait afficher la valeur 2.

[51]:
```
M1[1][0]
```

[51]: 2

M étant une liste de liste représentant une matrice, comment afficher le contenu de sa 2ème colonne sous forme de liste ? Par exemple, pour $M_1 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix}$, il faudrait afficher la liste [3,0].
Si vous bloquez, passez à la suite...

[52]:
```
col=[]
for liste in M1:
    col.append(liste[1])
```

[53]:
```
col
```

[53]: [3, 0]

4.0.1 Petite parenthèse sur les fonctions

Pour répéter le même traitement sur différents objets, on a intérêt à créer une fonction.
Par exemple, on souhaite extraire la colonne d'une matrice dans plusieurs circonstances, l'indice de la colonne pouvant varier ainsi que la matrice. Nous allons créer pour cela une fonction, appelée par exemple $colonne(M, j)$, qui prend en entrée (paramètres) une matrice M et un indice de colonne j et qui renvoie la colonne sous forme de liste.
La définition d'une fonction commence par l'instruction *def*. Comme pour l'instruction for, le bloc qui définit le corps de la fonction est déterminé par **l'indentation**. Le mot-clé *return* est suivi de la valeur que la fonction doit renvoyer.

Les programmes suivants permettent de créer cette fonction $colonne(M, j)$**.**

Définition de la fonction colonne (colonne1) en utilisant append :

[55]:
```
def colonne1(M,j):
    col=[]
    for i in range(len(M)):
        col.append(M[i][j])
    return col
```

Appel de la fonction colonne1 : 3ème colonne de $M_1 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix}$: $j = 2$ et $M = M_1$

[56]:
```
colonne1(M1,2)
```

[56]: `[-1, 4]`

Définition de la fonction colonne (colonne2) en utilisant une définition de liste en compréhension.

[57]:
```
def colonne2(M,j):
    col=[M[i][j] for i in range(len(M))]
    return col
```

Appel de la fonction colonne2 : 1ère colonne de $M_1 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix}$: $j = 0$ et $M = M1$

[58]:
```
colonne2(M1,0)
```

[58]: `[1, 2]`

5 Opérations sur les matrices

5.1 Multiplication d'une matrice par un scalaire

$3M_1 = 3\begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 9 & -3 \\ 6 & 0 & 12 \end{pmatrix}$: tous les termes de la matrice sont multipliés par 3 ## Somme de deux matrices de mêmes dimensions (mêmes nombres de lignes et de colonnes) $M_1 + M_4 = \begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix} + \begin{pmatrix} 2 & 1 & -2 \\ 0 & 2 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 4 & -3 \\ 2 & 2 & 8 \end{pmatrix}$: la somme est obtenue en additionant les termes de même ligne et même colonne. ## Produit de deux matrices Le produit ne s'obtient pas en faisant le produit des termes de même ligne et même colonne. Nous verrons la formule ultérieurement. ## Transposée ${}^tM_1 = {}^t\begin{pmatrix} 1 & 3 & -1 \\ 2 & 0 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 0 \\ -1 & 4 \end{pmatrix}$: on échange les lignes et les colonnes.

6 Exercices

Ecrire les codes des fonctions suivantes : * une fonction $multscal(M, x)$ qui prend en paramètre une liste de liste M représentant une matrice et un réel x et qui renvoie la liste de listes associée au produit xM * une fonction $somme(M, N)$ qui prend en paramètre deux listes de listes M et N représentant deux matrice de mêmes dimensions et qui renvoie la liste de listes associée à la somme $M + N$ * une fonction $transpose(M)$ qui prend en paramètre une liste de listes M représentant une matrice et qui renvoie la liste de listes associée à la transposée ${}^tM$