Informatique Pour Tous

Cours/TD 18 – Révisions et boucles imbriquées

A / Utiliser la fonction *range()*, parcourir une liste et comparer des éléments [Facile]

La fonction *range ()* est un *« built-in »,* native dans *Python* sans appel de bibliothèque. On s'en sert généralement à deux usages :

- Soit pour faire un balayage à l'aide d'une boucle *for* : `for k in range( … ) :`
- Soit pour créer une liste d'entiers, un peu à la manière de *np.linspace*() : `L = list(range(…))`

Étant donné que vous avez moins utilisé la seconde forme, nous allons nous y familiariser. Mais ce n'est pas là la difficulté. Que ce soit dans l'un ou l'autre des deux usages, le problème fréquemment rencontré est l'erreur d'index : d'où part, et surtout où s'arrête mon « range » ?

`range(a=0, b, p=1)` est une fonction à 3 arguments entiers.

- Si on l'appelle avec 2 arguments, `range(a=0, b)` l'argument implicite `p=1`
- Si on l'appelle avec 1 seul argument (le plus courant) `range(b)` les arguments implicites sont `a=0` et `p=1`

Cette fonction **renvoie** un *tuple* (qui peut aisément être converti en liste, cf. le point 2 ci-dessus), qui :

- Part de `a`
- Va de `a` vers `b` avec un pas `p`
- Il s'arrête après $\left[\frac{b-a}{p}\right]_{\text{arrondi sup}}$ itérations !

Exemple : si `p` = 1 (le plus courant), on part de `a` et on fera `(b-a)` itérations, on s'arrêtera donc à `b-1` !

Q1 – a – Que renvoient les commandes suivantes ? Prévoyez à l'avance, puis vérifiez avec votre terminal.

- `range(0, 5, 1)  # équivalent à range(5)`
- `range(-2, 5, 1)  # équivalent à range(-2,5)`
- `range(1, 7, 2)` et `range(1, 6, 2)`
- `range(2, -2, -1)`
- `range(0,0,1)`

b – Comment obtenir les listes suivantes, chacune obtenue en un seul appel de `list(range(…))`

- `[1, 2, 3, 4, … , 9, 10]`
- `[-10, -5, 0, 5, 10]`
- `[5, 4, 3, 2, 1, 0]`

B / Presque de la lecture de fichier texte, mais pas tout à fait ! [Facile]

Supposons que nous ayons une matrice écrite sous forme de fichier texte :

```
Matrice = " 1.1 \t 2.2 \t 3.3\n 4.4 \t 5.5 \t 6.6\n 7.7 \t 8.8 \t 9.9"
```

(on remarque la présence d'espaces, de tabulations pour séparer les colonnes, et de `"\n"` pour séparer les lignes).

Si l'on affiche (commande `print(Matrice)`) cette matrice, l'interpréteur nous affiche :
→ On souhaite ranger cette matrice, actuellement à l'état de texte, en une **liste de listes** (contenant des valeurs de type *float*).

```
print(Matrice)
1.1     2.2     3.3
4.4     5.5     6.6
7.7     8.8     9.9
```

Une fois créée la matrice *M* sous cette forme,

- `M[i]  # correspondra à la ième ligne, donc une liste`
- `M[i][j] # correspondra à l'élément en ième ligne, jème colonne, type float`

Q2 – Compléter le code de la fonction `rangement_table(Mat)` ci-dessous, prenant en argument un fichier texte *Mat* au format défini en haut de page, et renvoyant la liste de listes correspondante.

```
def rangement_table(Mat) :
    Mat = Mat.strip(" ")  # ceci retire les espaces, portée locale (car Mat str !)
    M_a_renvoyer = ...    # initalisation de la variable qu'on renvoie à la fin
    ...                   # utiliser split ...
    for i in range(...    # la fonction doit déterminer le nbre de lignes
        ...
        ...               # utiliser split à nouveau
        for j in range(...       # la matrice n'est pas forcément carrée
            A_ajouter_a_la_ligne = ... # élément qu'on ajoutera en ligne i colonne j
            ...
        ...
    return M_a_renvoyer
```

Tests :

```
Matrice1 = " 1.1 \t 2.2 \t 3.3\n 4.4 \t 5.5 \t 6.6\n 7.7 \t 8.8 \t 9.9 "
print( rangement_table(Matrice1) )
                    # affiche [[1.1, 2.2, 3.3], [4.4, 5.5, 6.6], [7.7, 8.8, 9.9]]

Matrice2 = " 1.11 \t 2.22 \t 3.33 \t 4.44 \n 5.55 \t 6.66 \t 7.77 \t 8.88 "
print( rangement_table(Matrice2) )
                    # affiche [[1.11, 2.22, 3.33, 4.44], [5.55, 6.66, 7.77, 8.88]]
print( type( rangement_table(Matrice2)[0][0] ))    # affiche <class 'float'>
```

C / Recherche du second maximum d'une liste [moyen]

On souhaite ici écrire une fonction `sec_max(L)` qui prenne en argument une liste de valeurs *L* et renvoie la valeur du second élément le plus grand de cette liste.

Pour ce faire, l'algorithme à coder est le suivant :

- Nous allons poser max1 et max2, les 1er et 2nd maxima
- Nous allons parcourir la liste *L*, et à chaque élément rencontré, selon la valeur de cet élément :
  - Soit il sera ignoré (si cet `element < max2`)
  - Soit cet élément prendra la place de max2 (si cet `max1 < element < max2`)
    Attention : Depuis *Python* 3.8, le test `a < b < c` est possible, toutefois, je vous **interdis** de l'utiliser (comme aux concours). Ce test s'écrit :
    `a < b and b < c`
  - Soit cet élément prendra la place de max1 (si cet `element > max1`)
    Dans ce cas, il faudra penser à modifier `max2` en conséquence…
- La fonction renverra uniquement la valeur de `max2` en fin d'algorithme.

Q3 – Écrire la fonction `sec_max(L)` suivant l'algorithme décrit ci-dessus.

Tests :

```
L_tst1 = [1,9,11,50,34,0]
L_tst2 = [5,94,-55,27,-9,73,55,92]
L_tst3 = [3, 5, 2, 1, -4]
print( sec_max(L_tst1) )    # affiche 34
print( sec_max(L_tst2) )    # affiche 92
print( sec_max(L_tst3) )    # affiche 3
```

D / Comptage des occurrences d'un élément dans une liste [Facile]

- Nous avons déjà fait cet exercice dans le TD 9 sur les dictionnaires, exercice C -

La fonction `Counter(L)`, de la bibliothèque `collections` permet de générer un dictionnaire, qui reprend les éléments distincts de la liste, et compte le nombre d'occurrence de chacun de ces éléments. Les développeurs utilisent beaucoup ce genre de fonctions. Deux exemples ci-dessous :

- Nous disposons de la base de données des recettes d'un vendeur de marché, en fonction de la semaine, et nous souhaitons savoir combien de fois il a touché 100€ ou plus.

```
>>> import collections
>>> ventes =[0, 100, 100, 80, 70, 80, 20, 10, 100, 100, 80, 70, 10, 30, 40]
>>> print(collections.Counter(ventes))
Counter({100: 4, 80: 3, 70: 2, 10: 2, 0: 1, 20: 1, 30: 1, 40: 1})
# La réponse est donc 4
```

- Nous disposons d'un texte, et on souhaite connaître le nombre de fois où chaque lettre est utilisée dans ce texte (c'est, entre autres, ce qui permet ensuite de calculer la probabilité de rencontrer une lettre, etc).

```
>>> texte = "ceci est une phrase purement arbitraire"
>>> liste_correspondante = list(texte)  # ça donne ["c","e","c","i"," ","e","s",…]
>>> print(collections.Counter(liste_correspondante))
Counter({'e': 7, ' ': 5, 'r': 5, 'i': 3, 't': 3, 'a': 3, 'c': 2, 's': 2, 'u': 2, 'n': 2, 'p': 2, 'h': 1, 'm': 1, 'b': 1})
# On voit sans surprise que les consonnes, particulièrement "e", sont très représentées.
```

On souhaite ici écrire une fonction `comptage(L)` qui réalise à peu près la même opération. Cette fonction renverra un dictionnaire contenant toutes les occurrences des éléments distincts de `L`. À la différence de `Counter(L)`, nous ne nous intéresserons **pas** ici au classement par ordre décroissant de ces éléments.

Pour le dictionnaire en sortie, chaque **clé** sera un élément distinct de `L`, et la **valeur** associée sera son nombre d'occurrences dans `L`.

Pour ce faire, l'algorithme à coder est le suivant :

- Nous allons déclarer un dictionnaire vide
  Rappel : ceci s'écrit `dico = {}`
- Nous allons parcourir la liste `L`, et à chaque élément rencontré :
  - Si cet élément correspond déjà à une clé, on ajoute 1 au nombre d'occurrences de cet élément
    Rappel : un dictionnaire n'est pas un objet itérable. Pour savoir si une clé existe ou non dans un dictionnaire, on écrira : `if cle in dico :` ou `if cle not in dico :`
  - Sinon, on crée une nouvelle clé dans le dictionnaire, initialisée avec la bonne valeur (je vous laisse réfléchir).
    Rappel : pour ajouter une clé et lui associer une valeur, on écrit : `dico[clé] = valeur`

Q4 – Écrire la fonction `comptage(L)` décrite ci-dessus.

Tests :
```
ventes =[0, 100, 100, 80, 70, 80, 20, 10, 100, 100, 80, 70, 10, 30, 40]
texte = list("ceci est une phrase purement arbitraire")
```

L'affichage (`print`) du dictionnaire renvoyé par le *comptage* devrait vous donner :
```
{0: 1, 100: 4, 80: 3, 70: 2, 20: 1, 10: 2, 30: 1, 40: 1}
{'c': 2, 'e': 7,'i': 3,' ': 5,'s': 2,'t': 3,'u': 2,'n': 2,'p': 2,'h': 1,
     ... 'r': 5, 'a': 3,'m': 1,'b': 1}
```

E / Tri à bulles – exemple récupéré du lycée BUFFON – MPSI PCSI [Moyen et important]

Le tri à bulles est un algorithme de tri « quadratique », peu optimisé, qui se fait **en place**. Nous allons balayer plusieurs fois la liste et comparer chaque élément à l'élément qui le suit directement, et faire si besoin « remonter » (vers la gauche de la liste, pour un tri par ordre croissant) l'élément le plus petit par permutation des deux éléments. Le nom évoquerait des bulles d'air qui remontent à la surface de façon successive.

L'algorithme s'appuie sur deux boucles imbriquées. Comprenons-le au travers d'un exemple : en veut trier par ordre croissant la liste de nombres suivante : `L = [2, 5, 4, 3, 1]`.
Au pire des cas (c'est, volontairement le cas dans l'exemple ici), le plus petit élément est à la fin, il va donc falloir faire 4 passages successifs (c'est-à-dire `len(L)-1`).

À chacun de ces passages `i`, nous allons balayer les éléments, du 1er jusqu'au i^{ème} en partant de la fin (ce que je vous écris ici, c'est en notations mathématiques, il faudra réfléchir aux index en *Python* plus tard).

Lors de ce balayage, on compare chaque élément avec le suivant :

- Si ces éléments sont déjà dans « le bon ordre » (localement), on les laisse tels quel
- Sinon, on permute ces deux éléments
  Rappel : ceci s'écrit `L[j], L[j+1] = L[j+1], L[i]`
  et, important (!) ceci a une portée **globale** si la liste `L` est la liste en argument !

Illustration :

- Premier passage :
  [2, 5, 4, 3, 1], on compare 2 et 5 et on ne fait rien car 5 > 2 ;
  [2, 5, 4, 3, 1], on compare 5 et 4 et on les permute car 5 < 4 ;
  [2, 4, 5, 3, 1], on compare 5 et 3 et on les permute ;
  [2, 4, 3, 5, 1], on compare 5 et 1 et on les permute ;
  ➔ Ainsi on a `L` = [2, 4, 3, 1, 5], fin du premier passage.

- Deuxième passage :
  [2, 4, 3, 1, 5], on compare 2 et 4 et on ne fait rien ;
  [2, 4, 3, 1, 5], on compare 4 et 3 et on les permute ;
  [2, 3, 4, 1, 5], on compare 4 et 1 et on les permute ;
  ➔ Ainsi, on a `L` = [2, 3, 1, 4, 5], fin du deuxième passage.

- Troisième passage :
  [2, 3, 1, 4, 5], on compare 2 et 3 et on ne fait rien ;
  [2, 3, 1, 4, 5], on compare 3 et 1 et on les permute ;
  ➔ [2, 1, 3, 4, 5], fin du troisième passage.

- Quatrième passage :
  [2, 1, 3, 4, 5], on compare 2 et 1 et on les permute ;
  ➔ [1, 2, 3, 4, 5], fin du quatrième passage.

➔ Fin de l'algorithme

Q5 – Écrire la fonction procédurale `tri_bulles(L)` décrite ci-dessus. Pour rappel, un tri **en place** se fait uniquement avec des permutations d'éléments de `L`, et le fait que cette fonction soit **procédurale** signifie qu'elle n'aura pas de `return` : la liste `L` en argument sera modifiée directement à l'extérieur de la fonction.

Comme on l'a déjà vu auparavant, si on écrit :

```
L_tst = [5, 1, 2, 3, 4, 5, -1, 0]
print(tri_bulles(L_tst))      # ceci affiche None, car la fonction n'a pas de return !
```

Q6 – Écrire les lignes de codes permettant d'afficher la liste de test `L_tst = [5, 1, 2, 3, 4, 5, -1, 0]` triée, vous vous en doutez, on est sensé voir affichée la liste : `[-1, 0, 1, 2, 3, 4, 5, 5]`

F / Recherche d'une occurrence d'un mot dans une chaîne de caractères [Difficile]

- Nous avons déjà rencontré un exercice similaire dans le TD 5 sur les listes, exercice A -

On souhaite finalement écrire une fonction `recherche(mot, texte)` qui recherche si elle trouve au moins une occurrence d'un mot dans un texte. On suppose, et c'est inutile de le vérifier, que la longueur du mot est inférieure ou égale à celle du texte mis en argument lors de l'appel.

Contrainte : L'algorithme imposé est le suivant :

- Nous allons balayer, dans l'ordre, toutes les lettres du `texte` (donc, boucle *for*). *On notera* `i` ***l'index de la lettre*** *dans* `texte`.
- Dès que la lettre balayée correspond à la 1ère lettre du `mot` recherché, on rentre dans une seconde boucle.
  - On teste alors, si le 2ème caractère du `mot` correspond au `i+1`ème caractère du `texte`, puis si le 3ème caractère du `mot` correspond au `i+2`ème caractère du texte, etc.
  - **Dès que** on en trouve un qui ne corresponde pas (c'est donc une boucle *while*), on quitte immédiatement cette boucle interne. Mais on continue la boucle externe.
  - En revanche, si on arrive à avoir balayé le `mot` totalement, on peut quitter la fonction et renvoyer `True`.

Si on ne trouve pas le `mot` recherché, on renverra `False`.

Exemple :

```
>>> print(recherche("mot","je suis super motivé"))    # affiche True
>>> print(recherche("mot","la promo ne dit pas mot"))# affiche True
>>> print(recherche("mot","mon modèle est moyen"))    # affiche False
```

Q7 – Écrire la fonction `recherche(mot, texte)` décrite ci-dessus, et la tester sur les exemples donnés.

G / Recherche des 2 valeurs les plus proches dans une liste de nombres [Difficile]

Sur mon site dans la rubrique informatique, vous trouverez deux vidéos qui décrivent deux algorithmes imposés et qui permettent la recherche des deux valeurs les plus proches :

- dans une liste (questions **Q8** à **Q10**) **: difficile**
- puis dans un tableau *Numpy* (question **Q11**) **: très difficile**

À titre d'aide, je vous rappelle les fonctions suivantes de *Numpy* :

```
# import numpy as np
np.shape(T)        #  si  T  est  un  objet  type  numpy.array :  renvoie
                   #(nombre_lignes, nombre_colonnes)
np.zeros([N,M])    #  avec  N  et  M  entiers  (type  int) :  renvoie  une  matrice
                   # (table Numpy) de N lignes, M colonnes, remplie de 0
np.ones([N,M])     # même chose, mais avec une matrice remplie de 1
np.eye(N)          # matrice identité, de taille N
# Entre autres, je vous recommande d'observer la commande suivante :
np.eye(N) - np.ones([N,N]) # ceci renverra une matrice carrée, dont la diagonale
                           # est nulle, et dont tous les autres éléments sont -1
```

Q8 – Écrire et tester la fonction `constr_table(L)` décrite dans la vidéo.

Tests :

```
L1 = [3,9,11,50,34,0]
L2 = [1,9,11,50,34,0]
L3 = [5,94,55,27,9,73,55,92]
```

Voici ce que l'on obtiens avec les commandes … :

```
In [9]: constr_table(L1)
Out[9]:
array([[ 0.,  6.,  8., 47., 31.,  3.],
       [-1.,  0.,  2., 41., 25.,  9.],
       [-1., -1.,  0., 39., 23., 11.],
       [-1., -1., -1.,  0., 16., 50.],
       [-1., -1., -1., -1.,  0., 34.],
       [-1., -1., -1., -1., -1.,  0.]])
```

```
In [10]: constr_table(L2)
Out[10]:
array([[ 0.,  8., 10., 49., 33.,  1.],
       [-1.,  0.,  2., 41., 25.,  9.],
       [-1., -1.,  0., 39., 23., 11.],
       [-1., -1., -1.,  0., 16., 50.],
       [-1., -1., -1., -1.,  0., 34.],
       [-1., -1., -1., -1., -1.,  0.]])
```

```
In [11]: constr_table(L3)
Out[11]:
array([[ 0., 89., 50., 22.,  4., 68., 50., 87.],
       [-1.,  0., 39., 67., 85., 21., 39.,  2.],
       [-1., -1.,  0., 28., 46., 18.,  0., 37.],
       [-1., -1., -1.,  0., 18., 46., 28., 65.],
       [-1., -1., -1., -1.,  0., 64., 46., 83.],
       [-1., -1., -1., -1., -1.,  0., 18., 19.],
       [-1., -1., -1., -1., -1., -1.,  0., 37.],
       [-1., -1., -1., -1., -1., -1., -1.,  0.]])
```

Q9 – Écrire et tester la fonction `index_mini_table(T)` décrite dans la vidéo. Vous pourrez la tester sur les tables générées en **Q8**.

Q10 – Écrire et tester la fonction `deux_val_pp(L)` décrite dans la vidéo.

```
139 L1 = [3,9,11,50,34,0]
140 L2 = [1,9,11,50,34,0]
141 L3 = [5,94,55,27,9,73,55,92]
142
143 print("Pour L1, les 2 val les + proches sont ", deux_val_pp(L1))
144 print("Pour L2, les 2 val les + proches sont ", deux_val_pp(L2))
145 print("Pour L3, les 2 val les + proches sont ", deux_val_pp(L3))
```

⇨

```
Pour L1, les 2 val les + proches sont  (9, 11)
Pour L2, les 2 val les + proches sont  (1, 0)
Pour L3, les 2 val les + proches sont  (55, 55)
```

Q11* – D'autre part, écrire et tester la fonction `ppval(T)` décrite dans la vidéo. Pour le test, on vous donne les tables suivantes :

```
Table1 = np.array([[0, 10, 3, 2],[-1,7,15, 9],[8,-12,4,0]])

Table2 = np.array([[129, 533, 543, 812, 341, 788, 463, 609, 775, 302],
       [667, 929, 278, 774, 236, 495, 965, 515, 96, 137],
       [379,  17, 280, 71, 160, 151, 944, 141, 796, 186],
       [452, 234, 886, 400, 402, 499, 226, 522, 184, 596],
       [981, 472, 433, 384, 645, 752, 818, 404, 627, 913],
       [177, 396, 107, 207, 315, 430, 110, 240, 470, 457]]
```

```
148 print(ppval(Table1))
149 print(ppval(Table2))
```

⇨

```
(0, 0)
(775, 774)
```

Remarque : cette question est « étoilée » car elle est difficile, surtout au niveau des bornes des boucles *for*.