TP (S1) 2 Recherches séquentielles - Listes et dictionnaires

Objectifs

- S'approprier l'environnement de travail.
- Revoir les éléments de base du langage : variables, tests, boucles, fonctions.

1. RECHERCHES SIMPLES

Exercice 1 **Autour du maximum d'une liste** [Sol 1]

1) On veut écrire la fonction `maximum(L:list)->int` renvoyant la valeur du plus grand élément de la liste L (celle-ci étant supposée non vide, et ne contenir que des entiers). On donne l'algorithme sur lequel repose la fonction :

- affecter le premier élément de la liste `L` à une variable `M`,
- pour chacun des éléments `e` de `L`,
  - si `e` est plus grand que `M`, affecter la valeur de `e` à `M`.

Écrivez cette fonction en utilisant un parcours de la liste à l'aide de l'instruction `for e in L`. Testez la fonction sur des listes présentant des situations diverses (maximum en première position, en dernière position, ailleurs que sur les bords, liste à un seul élément, élément maximal apparaissant plusieurs fois dans la liste).

2) On peut noter que l'emploi de `for e in L` entraîne un test inutile (celui du premier élément de `L`). Cependant la syntaxe associée est très simple et bien adaptée à la situation. Modifier la fonction précédente en utilisant un slicing sur `L` pour remédier à ce problème.

3) Écrivez sur le même principe une fonction `minimum(L:list)->int` renvoyant cette fois la valeur du plus petit élément de la liste L, et testez-la sur plusieurs listes bien choisies.

4) Écrivez une fonction `ind_maximum(L:list)->int` renvoyant le plus petit indice `k` tel que `L[k]` soit le plus grand élément de la liste `L`, et utilisant une boucle for. Par exemple, pour la liste `L=[2,4,1,5,3]`, la fonction renverra 3 puisque le plus grand élément qui vaut 5 apparaît à l'indice 3. Quel type de parcours de liste est-il nécessaire d'utiliser ici ?

5) Réécrire la fonction précédente en utilisant une boucle `while`.

Exercice 2 **Manipulations de dictionnaire** [Sol 2] On considère un dictionnaire `D` tel que chaque clé de `D` soit une chaine de caractère (de type `str`), et tel que la valeur associée à chaque clé soit un nombre entier. Par exemple, on pourra avoir `D={'a':5,'b':3,'c':8 }`.

1) Écrire le script d'une fonction `trouveCle(D:dict,v:int)->list` qui renvoie la liste des clés ayant pour valeur `v` (si aucune clé n'a pour valeur `v`, la fonction renverra la liste vide).

2) Écrire le script d'une fonction `cleMax(D:dict)->str` renvoyant la clé pour laquelle la valeur associée est maximale (dans le cas où plusieurs clés possèdent cette valeur maximale, on renverra indifféremment une de ces clés). On pourra s'inspirer de la méthode vue dans l'exercice 1, et utiliser l'instruction `M = -float('inf')` qui permet de créer une variable `M` plus petite que n'importe quelle nombre flottant ou entier.

3) Reprendre la question précédente en écrivant le script d'une fonction `cleMin(D:dict)->str` renvoyant la clé pour laquelle la valeur associée est minimale.

Exercice 3 **Présence d'un élément dans une liste** [Sol 3]
On s'intéresse ici à deux recherches qui ont pour but de savoir si un élément donné `e` est présent dans une liste `L`.

1) Pour la première recherche, on cherche à écrire une fonction `ind_present(L:list,e)->list` qui renvoie la liste des indices où l'élément `e`

apparait dans la liste `L`. Si `e` n'appartient pas à `L`, la fonction devra renvoyer la liste vide.

1.1) Le parcours de liste doit-il ici se faire à l'aide d'une boucle `for` ou d'une boucle `while` ?

1.2) Écrire la fonction demandée.

2) Pour la deuxième recherche, on cherche à écrire une fonction `present(L:list,e)->bool` qui renvoie `True` si l'élément `e` appartient à `L`, et `False` sinon.

2.1) Un élève propose la fonction la solution suivante :

```
def present(L:list,e)->bool:
    """ Renvoie True si e est un élément de L et False \
    ↪ sinon """
    pres = False
    for v in L:
        if v == e:
            pres = True
    return pres
```

Expliquer quel est le défaut de cette façon de programmer. Pour répondre au problème posé, faut-il utiliser une boucle `for` ou une boucle `while` ?

2.2) Écrire une version de la fonction demandée qui tienne compte de la remarque précédente.

Exercice 4 **Second maximum** [Sol 4]

Cet exercice est plus délicat que les précédents. On veut écrire une fonction `second_maximum(L:list)->int or None` renvoyant le second maximum de la liste non vide d'entiers L. Il s'agit de l'élément de la liste ayant la valeur « juste en-dessous » du maximum. Par exemple le second maximum de la liste `L=[2,4,1,5,3]` est 4. Pour que cette notion soit définie, il faut que la liste possède au moins deux éléments distincts, sans quoi la fonction renverra la valeur `None`.

Le principe proposé (un autre plus performant le sera dans la prochaine partie) est le suivant :

- on commence par parcourir la liste de manière à trouver le premier élément e qui n'est pas égal à L[0] (si on n'en trouve pas, la fonction renverra `None`) ;
- on affecte aux variables M et sM respectivement la plus grande et la plus petite des valeurs L[0] et e ;
- on continue ensuite le parcours de la liste, à partir de l'élément suivant e, sur un principe similaire à celui utilisé pour la fonction `maximum`, en actualisant les valeurs de M et sM suivant différents cas pour la valeur de l'élément courant de la liste (par exemple, si l'élément courant est strictement supérieur à M, alors sM prend la valeur de M et M prend la valeur de l'élément courant) ;
- quand tous les éléments de la liste ont été parcourus, la fonction renvoie `sM`.

Écrivez la fonction `second_maximum`, en analysant bien pour chacune des deux boucles successives si elle doit se faire avec `for` ou `while`, et testez-la sur plusieurs listes bien choisies.

2. RECHERCHES PLUS ÉLABORÉES

Exercice 5 **Tri à bulles** [Sol 5] On s'intéresse ici à une méthode de tri appelée *tri à bulles*, qui s'applique à un ensemble de données contenues dans une liste de n éléments. Le principe de base en est le suivant : on commence par parcourir les éléments de T pour un indice i variant de 0 à $n-2$ (soit $n-1$ valeurs de i), et à chaque itération, si $T[i] > T[i+1]$, alors on permute $T[i]$ et $T[i+1]$. L'élément le plus grand est alors en dernière position de T. On recommence ensuite le processus en diminuant à chaque fois d'une unité le nombre d'éléments parcourus depuis le début de T. Cette méthode est nommée tri à bulles car les éléments les plus grands remontent petit à petit vers la fin de la liste comme des bulles dans une boisson gazeuse.

1) Écrire une fonction `triBulle(T: list) -> list` qui effectue le tri de `T` selon le principe décrit précédemment.

2) Appliquer à la main le principe de ce tri sur la liste `[5, 1, 2, 4, 3]`. Comment l'algorithme pourrait-il être amélioré ?

3) Proposer une nouvelle version de `triBulle(T: list) -> list` qui tienne compte de cette amélioration.

Exercice 6 **Comptage des éléments** [Sol 6]

On souhaite écrire une fonction `compte(L : list) -> dict` qui pour une liste L donnée renvoie un dictionnaire (notez que `dict` est le nom du type associé la structure de dictionnaire) ayant pour clés les différents éléments de la liste et pour valeur associée le nombre de fois où cet élément est présent dans L.

Par exemple, pour la liste L = [1,5,1,2,5,5,3,2] où l'élément 1 apparaît 2 fois, l'élément 5 apparaît 3 fois, …, l'appel `compte(L)` doit renvoyer le dictionnaire

`{1:2,5:3,2:2,3:1}`. (Notez que sur cet exemple les clés sont des entiers et pas des chaînes de caractères.)

Par concevoir cette fonction, on propose la démarche suivante :

- on initialise une variable `D` à un dictionnaire vide (qui se note {}, de la même manière que [] représente la liste vide) ;
- puis on parcourt chaque élément e de la liste L : si e n'est pas déjà présent comme clé dans le dictionnaire D (ce qui signifie que c'est la première fois que l'on trouve la valeur e dans la liste L) on ajoute cette clé au dictionnaire en lui associant la valeur 1 ; et si au contraire e apparaissait déjà en tant que clé dans le dictionnaire on modifie la valeur associée en lui ajoutant 1 (pour tenir compte de cette nouvelle occurrence de la valeur).

1) Écrire cette fonction et testez-la sur plusieurs listes bien choisies.

2) Modifiez la fonction précédente pour écrire une fonction `indices(L : list) -> dict` renvoyant toujours un dictionnaire dont chaque clé est un élément présent dans L, mais la valeur associée est la liste de tous les indices où cet éléments apparaît.

Par exemple, pour la liste L = [1,5,1,2,5,5,3,2] où l'élément 1 apparaît en indices 0,2 et l'élément 5 apparaît en indices 1,4,5 …, l'appel `indices(L)` doit renvoyer le dictionnaire `{1:[0,2],5:[1,4,5],2:[3,7],3:[6]}`.

3) On considère maintenant le problème inverse de la question précédente. Soit D un dictionnaire dont les clés sont les éléments présents dans une liste L et dans lequel la valeur associée à chaque clé est la liste des indices où cet élément est présent dans L. Écrire la fonction `dict2list(D: dict) -> list` qui reconstruit et renvoie la liste L connaissant D.

Exercice 7 Second maximum : le retour [Sol 7]

Revenons sur la fonction `second_maximum(L : list)-> int or None` écrite dans la partie précédente. L'algorithme présenté n'est pas optimal car il conduit à faire plus de comparaisons entre les éléments de la liste que nécessaire. Il existe un algorithme réduisant ce nombre de comparaisons, appelé algorithme du **tournoi**.

Il s'agit d'organiser une succession de « matchs » entre les éléments de la liste (le plus grand élément remportant la rencontre) selon le principe d'un tournoi à élimination directe. Le vainqueur du tournoi est bien sûr le maximum de la liste. En observant la liste des éléments que ce maximum a rencontrés et vaincus, on constate que son plus grand élément est le second maximum de la liste initiale (en effet le second maximum ne peut avoir été vaincu que par le maximum). L'avantage de cette méthode est que la deuxième recherche de maximum se fait sur une liste beaucoup plus petite (uniquement les adversaires du maximum). On peut démontrer que si on note n la longueur de la liste, alors la méthode « naïve » de la partie précédentes conduit à effectuer environ $2n$ comparaisons, alors que l'algorithme du tournoi n'en effectue qu'environ $n + \log_2(n)$ (soit un gain pratiquement d'un facteur 2 quand n est grand puisque $\log_2(n)$ est très significativement plus petit que n).

Voici le déroulement de l'algorithme du tournoi sur la liste

$$L = [3,1,4,5,2,2,1,3,2,4].$$

- Dans un premier temps, on organise les rencontres des joueurs pris 2 à 2 : 3 rencontre 1 ; 4 rencontre 5 ; 2 rencontre 2 ; 1 rencontre 3 et 2 rencontre 4. Chaque vainqueur ou ex-aequo est stocké dans une nouvelle liste V (en cas d'ex-aequo, on stocke une seule fois la valeur), on obtient ici

  $$V = [3,5,2,3,4].$$

  On construit ensuite un dictionnaire D dont les clés sont les vainqueurs de chaque rencontre et la valeur une liste contenant l'élément qu'il a vaincu (ou les éléments si la même valeur a vaincu dans plusieurs matchs). Attention ! en cas d'ex-aequo il n'y a ni vainqueur ni vaincu à stocker dans le dictionnaire. On obtient ici :

  $$D = \{3 : [1,1], 5 : [4], 4 : [2]\}.$$

- On recommence alors le même traitement sur la liste V qui remplace L. Si comme ici la liste a une taille impaire, le dernier élément est automatiquement rajouté à la nouvelle liste des vainqueurs sans faire de match. En outre, on ajoute les nouveaux résultats aux anciens dans le dictionnaire D. On obtient donc :

  $$V = [5,3,4] \text{ et } D = \{3 : [1,1,2], 5 : [4,3], 4 : [2]\}.$$

- On poursuit de même :

  $$V = [5,4] \text{ et } D = \{3 : [1,1,2], 5 : [4,3,3], 4 : [2]\}.$$

- Et enfin :

  $$V = [5] \text{ et } D = \{3 : [1,1,2], 5 : [4,3,3,4], 4 : [2]\}.$$

  La liste V n'ayant plus qu'un élément, 5 est le maximum de la liste L. Pour trouver le second-maximum de L, il suffit de récupérer dans le dictionnaire D la liste [4,3,3,4] des perdants devant 5, et d'en chercher le maximum qui est bien 4 (on pourra bien sûr utiliser la fonction `maximum` écrite dans l'exercice 1.

Notons que si l'algorithme est lancé sur une liste L ne possédant qu'une seule valeur, celle-ci est bien sûr vainqueur du tournoi mais le dictionnaire est vide (toutes les rencontres ont données des ex-aequo), et dans ce cas la fonction doit renvoyer la valeur `None`.

Écrivez le nouveau code de la fonction `second_maximum` utilisant l'algorithme du tournoi.

SOLUTIONS DES EXERCICES

Solution 1

1)
```
def maximum(L:list)->int :
    """ Renvoie le plus grand élément de la liste non vide
    d'entiers L """
    M = L[0]
    for e in L :
        if e > M:
            M = e
    return M
```

2) Avec slicing, les éléments de `L` à tester sont ceux contenus dans `L[1:]`.

```
def maximum(L:list)->int :
    """ Renvoie le plus grand élément de la liste non vide
    d'entiers L """
    M = L[0]
    for e in L[1:]:
        if e > M:
            M = e
    return M
```

3) On remplace > par < (le changement du nom de la variable M en *m* est purement esthétique).

```
def minimum(L:list)->int :
    """ Renvoie le plus petit élément de la liste non vide
    d'entiers L """
    m = L[0]
    for e in L[1:]:
        if e < m:
            m = e
    return m
```

4) Évidemment on boucle ici sur les **indices** :

```
def ind_maximum(L:list)->int :
    """ Renvoie un indice du plus grand élément de la liste \
    ↪ non vide
    d'entiers L """
    ind_M = 0
    for i in range(1,len(L)):
        if L[i] > L[ind_M]:
            ind_M = i
    return ind_M
```

Notez que si le maximum est présent plusieurs fois dans la liste, ce programme renvoie l'indice de sa première occurrence.

5) On peut aussi utiliser une boucle `while` :

```
def ind_maximum(L:list)->int :
    """ Renvoie un indice du plus grand élément de la liste \
    ↪ non vide d'entiers L """
    ind_M = 0
    i = 1
    while i < len(L):
        if L[i] > L[ind_M]:
            ind_M = i
        i = i+1
    return ind_M
```

Solution 2

1)
```
def trouveCle(D:dict,v:int)->list:
    L = []
    for c in D:
        if D[c] == v:
            L.append(c)
    return L
```

2)
```
def cleMax(D:dict)->str:
    M = -float('inf')
    for c in D: # on parcourt les clés
        if D[c] > M:  # cas où la valeur associée à c est plus \
        ↪ grande que le maximum local
            M = D[c]
```

```
            cMax = c
    return cMax
```

3) On remplace > par < (le changement du nom de la variable M en *m* est purement esthétique).

```
def cleMin(D:dict)->str:
    m = float('inf')
    for c in D: # on parcourt les clés
        if D[c] < m: # cas où la valeur associée à c est plus \
        ↪ petite que le minimum local
            m = D[c]
            cMin = c
    return cMin
```

Solution 3

1) 1.1) On doit tester les *n* éléments de la liste `L`, quel que soit son contenu. Il est préférable d'utiliser une boucle **`for`**.

1.2) Fonction `ind_present`

```
def ind_present(L:list,e)->list :
    """ renvoie la liste des indices où e est présent dans \
    ↪ la liste L """
    L1 = []
    for i in range(len(L):
        if L[i] == e:
            L1.append(i)
    return L1
```

2) 2.1) La boucle **`for`** parcourt l'intégralité de la liste, même si l'élément recherché est présent au début de la liste. Il vaut donc mieux utiliser une boucle **`while`** qui parcours la liste *tant qu'on a pas trouvé l'élément et qu'on n'est pas au bout de la liste.*

2.2) Il existe plusieurs manières de faire, une possibilité est :

```
def present(L:list,e) -> bool:
    """ renvoie True si x est un élément de L et False \
    ↪ sinon """
    i = 0
    n = len(L)
    while i < n and L[i] != e :
        i = i+1
    return i < len(L)
```

On peut faire plusieurs commentaires :

- On continue à boucler tant qu'**aucune des deux** conditions de sortie n'est vérifiée (la négation d'un « ou » est un « et ») : `i < n` signifie que l'on n'a pas encore examiné la liste en entier, et `L[i] != e` signifie que l'élément courant n'est pas *e*.
- Le test écrit `i < n and L[i] != e` permet, grâce à l'évaluation paresseuse de python, d'assurer que `L[i]` soit bien défini à chaque itération, et que donc le test `L[i] != e` ne provoque pas d'erreur. Si l'on avait écrit le test sous la forme `L[i] != e and i < n`, alors il y aurait une erreur dans tous les cas où `e` n'appartient pas à `L`.
- On renvoie le résultat d'un test, donc on renvoie bien **`True`** ou **`False`** : si la condition `i < n` est évaluée à **`True`**, c'est que la sortie de boucle s'est faite sur l'autre condition, et donc que l'on a trouvé *e* dans L, et la fonction renvoie donc bien **`True`** dans ce cas. A l'inverse, si la condition `i < n` est évaluée à **`False`**, c'est que la sortie de boucle s'est faite après avoir examiné sans succès chaque élément de L, donc *e* n'appartient pas à la liste, et la fonction renvoie bien **False** dans ce cas.

Solution 4

```
def second_maximum(L:list)->int or None:
    """ renvoie le second maximum de la liste non vide
    d'entiers L s'il existe, et None sinon """
    M = L[0]
    i = 0
    n = len(L
    while i< n and L[i] == M :
        i += 1
    if i == n :
        return None
    else :
        if L[i] < M :
            sM = L[i]
        else :
            sM,M = M,L[i]
```

```
        for e in L[i+1:]:
            if e > M:
                sM,M = M,e
            elif sM < e < M:
                sM = e
    return sM
```

Solution 5

1) fonction `triBulle(T:list)->list`

```
def triBulle(T: list) -> list:
    """
    entrée : T liste d'éléments comparables
    sortie : T triée
    """
    n = len(T)
    for t in range(n-1,0,-1):  # t est la taille du tableau \
    ↪ que la variable i parcourt
        for i in range(t):
            if T[i] > T[i+1]:
                T[i], T[i+1] = T[i+1], T[i]
    return T
```

2) Pour la liste donnée, à la fin de la deuxième itération, le tableau est déjà trié, et la troisième itération ne conduit à aucune permutation de valeurs (ni la quatrième bien sûr). On peut donc interrompre l'algorithme dès qu'un parcours ne conduit à aucun échange.

3) Nouvelle version de `triBulle(T:list)->list`

```
def triBulle(T:list)->list:
    """
    entrée : T liste d'éléments comparables
    sortie : T triée
    """
    n = len(T)
    t = n-1  # indice du dernier
    echange = True
    while t > 0 and echange:
        echange = False
        for j in range(t):
            if T[j] > T[j+1]:
                T[j], T[j+1] = T[j+1], T[j]
                echange = True
        t -= 1
    return T
```

Solution 6

1) La fonction :

```
def compte(L : list) -> dict :
    """ renvoie un dictionnaire dont les clés sont les éléments
    présents dans la liste L et les valeurs associées sont
    le nombre de fois que cet élément est présent."""
    D = {}
    for e in L :
        if e not in D :
            D[e]=1
        else :
            D[e] += 1
    return(D)
```

Remarque : l'instruction `if` pourrait être remplacée par `D[e] = 1 + D.get(e,0)`.

2) La fonction :

```
def indices(L : list) -> dict :
    """ renvoie un dictionnaire dont les clés sont les éléments
    présents dans la liste L et les valeurs associées sont la \
    ↪ liste
    des indices où cet élément est présent."""
    D = {}
    for i in range(len(L)) :
        if L[i] not in D :
            D[L[i]] = [i]
        else :
            D[L[i]].append(i)
    return(D)
```

3) La fonction :

```python
def dict2list(D:dict)->list:
    """ renvoie la liste L correspondant au dictionnaire D \
    ↪ obtenu par
    indices(L) """
    # détermination de la taille de L
    n = 0
    for e in D:
        n += len(D[e])
    # initialisation de L
    L = [0]*n
    # modification des éléments de L
    for e in D:
        for i in D[e]:
            L[i] = e
    return L
```

Solution 7

```python
def second_maximum(L : list) -> int or None :
    """ renvoie le second maximum de la liste non vide
    d'entiers L s'il existe, et None sinon """
    D = {}
    while len(L) > 1 :
        V = []
        for i in range(0,len(L)-1,2) :  # on parcourt les
            if L[i]> L[i+1] :  # indices de 2 en 2
                g, p = L[i], L[i+1]  # g est le gagnant
            else :  # et p le perdant
                g, p = L[i+1], L[i]  # g=p si ex-eaquo
            V.append(g)
            if g!=p :  # ne pas considérer les ex-eaquo
                if g not in D :  # comme perdants
                    D[g] = [p]  # dans le dictionnaire
                else :
                    D[g].append(p)
        if len(L)%2 != 0 :    # si la taille est impaire
            V.append(L[len(L)-1])  # le dernier élément est \
            ↪ gagnant
        L = V  # actualisation de la liste des adversaires
    if D == {} :
        return(None)  # si aucun vainqueur pas de second \
        ↪ meilleur
    else :
        return(maximum(D[L[0]]))  # sinon, c'est le meilleur
                                  # perdant devant le vainqueur
```