TP – Introduction aux listes et aux chaînes de caractères

La manipulation des listes

Executez les commandes suivantes et vérifiez qu'elles se comportent comme attendu.

```
t1 = [0,3,4,-2,3,-1,0,7] # Définition d'une liste
len(t1)                  # Commande retournant la longueur d'une liste (ici 8)
t1[0]                    # Valeur du premier élément de t1
t1[1]                    # Valeur du second élément de t1
t1[2]                    # Valeur du troisième élément de t1
t1[len(t1)-1]            # Valeur du dernier élément de t1
t1[len(t1)]              # N'existe pas
t1[-1]                   # Valeur du dernier élément de t1
t1[-2]                   # Valeur de l'avant dernier élément de t1
t1[-len(t1)]             # Valeur du premier élément de t1
t1[-len(t1)-1]           # N'existe pas
[2,3,4]+[-1,0]           # Concaténation de listes -> [2,3,4,-1,0]
3*[2,3,4]                # Concatène trois fois la liste [2,3,4] -> [2,3,4,2,3,4,2,3,4]
t1 + 42                  # Mais pourquoi est-il aussi méchant ?
t1 + [42]                # Crée un liste avec les valeurs de t1 suivie de la valeur 42
t1.append(42)            # Ajoute la valeur 42 au bout de la liste t1
t1.extend([-1,0])        # Ajoute en queue de la liste t1 les valeurs -1 et 0
del(t1[6])               # Retire l'élément en position numéro 6
t1.remove(3)             # Retire la première occurence de la valeur 3
t1.remove(123)           # Ne trouve pas la valeur 123 pour la retirer...
t1.remove([12])          # Non remove ne permet pas de retirer une liste de valeurs...
t1.insert(1,3.14)        # Ajoute la valeur 3.14 dans la liste, en case numéro 1
list(range(5))           # Crée la liste [0,1,2,3,4] (nombres compris entre 0 et 5-1)
list(range(3,7))         # Crée la liste [3,4,5,6] (nombres compris entre 3 et 7-1)
list(range(3,7,2))       # Crée la liste [3,5] (idem avec un pas de 2)
t1[0:5]                  # Renvoie la liste des valeurs d'indice compris entre 0 et 5-1
t1[0:5:2]                # idem avec un pas de 2
t1[:5]                   # idem pour les indices compris entre 0 et 5-1
t1[0:]                   # idem pour les indices compris entre 0 et len(t)-1
t1[:]                    # idem pour les indices compris entre 0 et len(t)-1
t1[::2]                  # pareil que t1[0:len(t):2]
t1[::]                   # pareil que t1[0:len(t)]
t1[3]=1                  # Remplace la valeur située à l'indice 3 par la valeur 1
t2=t1                    # met dans t2 la référence vers la liste pointée par t1
                         # ainsi t1 et t2 correspondent en mémoire à la même liste
t3=t1[:]                 # t1[:] crée une nouvelle liste constituée des valeurs de t1.
                         # t1 et t3 sont deux listes différentes en mémoire.
```

Voyons comment créer des listes par *compréhension*, c'est-à-dire en les décrivant :

```
t = [i for i in range(6)]                                  # [0,1,2,3,4,5]
[i**2 for i in range(6)]                                   # [0,1,4,9,16,25]
[i**2 for i in range(6) if i>3]                            # [16,25]
[i**2 for i in range(6) if i%2==0]                         # [0,4,16]
[i for i in range(6) if (i%5) in {0,2,4}]                  # [0,2,4,5]
t2 = [v**2 for v in t]                                     # [0,1,4,9,16,25]
[t2[2*i+1]-1 for i in range(3)]                            # [0,8,24]
[(x,y) for x in [1,2,3] for y in [3,1,4] if x != y ]       # [(1,3),(1,4),(2,3),(2,1),...
                                                           #           ...(2,4),(3,1),(3,4)]
```

Un peu d'entraînement

Ecrire une fonction `make_list` prenant en paramètre un entier n et retournant une liste contenant exactement n zéros.

Ecrire une fonction `premier_element` prenant en paramètre une liste et retournant son premier élément.

Ecrire une fonction `dernier_element` prenant en paramètre une liste et retournant son dernier élément.

Ecrire une fonction `somme` qui prend en paramètre une liste (suposée non vide) et qui retourne la somme des éléments de cette liste. Ainsi `somme([1,10,15])` renverra la valeur `26`. Comment modifier l'algorithme pour que la fonction somme continue de fonctionner avec des entiers mais aussi avec des chaînes de caractères : `somme(["T","S","I"])` retournera alors "TSI".

Ecrire une fonction `entiers_pairs` qui étant donné une liste retourne une liste composée de ses nombres pairs.

Ecrire une fonction `nombre_pairs` qui étant donné une liste d'entiers retourne le nombre d'entiers pairs qui la compose.

Ecrire une fonction `echange` qui étant donné une liste et deux entiers `i` et `j` modifie la liste de façon à échanger les valeurs en case `i` et `j`.

Ecrire une fonction `maximum` qui étant donné une liste (supposée non vide) retourne son plus grand élément. Que retourne `maximum(["T","S","I"])` ; pourquoi ?

Ecrire une fonction `indice_max` qui étant donné une liste retourne l'indice (le numéro de la case) de son plus grand élément.

Ecrire une fonction `appartient` qui étant donné une liste et un élément, retourne `True` si cet élément se trouve dans la liste et `False` sinon.

Ecrire une fonction `are_similar` qui étant donné deux listes retourne `True` si les deux listes contiennent les mêmes éléments (peu importe leur nombre d'apparition) et qui retourne `False` sinon. On pourra commencer par coder une fonction `est_inclus` qui étant donné deux listes dit si les éléments de la premère liste se trouvent être dans la seconde.

Ecrire une fonction `compare` prenant en paramètre le nom d'une fonction `func` (s'appliquant à une liste) ainsi que deux listes. Cette fonction retournera `True` si la fonction `func` retourne la même valeur lorsqu'elle est appelée sur chacune des deux listes. Vérifier que la fonction se comporte correctement en appelant `compare(somme,[1,3,2],[0,0,6])` et `compare(somme,["a","bc","def"],["a","b","cd","ef"])`

Ecrire une fonction `intersection` qui étant donné deux listes d'élements (que l'on suppose tous distincts), retourne une liste représentant leurs éléments communs. On utilisera la fonction `appartient`.

Ecrire une fonction `est_croissant` qui étant donné une liste retourne `True` lorsque les éléments de la liste sont rangés par ordre croissant et `False` sinon. On fera une première version de cette fonction avec une boucle **pour** puis on l'améliorera avec une boucle **tant que** pour éviter de parcourir inutilement toute la liste.

Pour aller plus loin

Ecrire une fonction `plus_grand_plateau` qui étant donné une liste retourne le plus grand nombre de zéros consécutifs présents dans la liste.

Ecrire une fonction `plus_grand_plateau2` qui étant donné une liste retourne le plus grand nombre de zéros consécutifs présents dans la liste en considérant que la liste se répète indéfiniment. On conviendra que ce nombre ne pourra dépasser la longueur de la liste.

Ecrire une fonction `maximum_relatif` qui étant donné une liste et un nombre, retourne la plus grande valeur de la liste parmi les valeurs strictement plus petites que le nombre passé en paramètre.

Ecrire une fonction `tri_decroissant` qui étant donné une liste d'éléments distincts retourne une liste contenant les mêmes éléments rangés par ordre décroissant. On aura recours à la fonction `maximum_relatif`.

Ecrire une fonction `tri_croissant` qui étant donné une liste d'éléments distincts retourne une liste contenant les mêmes éléments rangés par ordre croissant. On aura recours à la fonction `maximum_relatif` ou à la fonction `tri_decroissant`.

Ecrire une fonction `are_equals` prenant deux listes en paramètre et retournant `True` si ces deux listes contiennent les mêmes éléments (avec le même nombre d'occurences) et `False` sinon. Cette fonction utilisera `tri_croissant`. On verifiera que `are_equals([1,2,2,1],[2,2,1,2])` retourne `False` et que `are_equals([1,2,2,1],[1,2,1,2])` retourne `True`.

Ecrire une fonction `make_2d_list` prenant en paramètre 2 entiers p et q et retournant une liste à p éléments ; chacun de ces éléments étant eux même une liste à q éléments valant 0.

Ecrire une fonction `table_multiplication` prenant en paramètre 2 entiers p et q et retournant une liste `l` telle que `l[i][j]` contienne le produit $i \times j$.

Ecrire une fonction `parse_csv_line` prenant en paramètre une chaîne de caractères correspondant à une ligne d'un fichier CSV et retournant la liste correspondante. `parse_csv_line("12,34,56\n")` retournera `[12,34,56])`.

Ecrire une fonction `parse_csv_file` prenant en paramètre un fichier CSV en paramètre et retournant la liste correspondante. Cette fonction utilisera `parse_csv_line`. On créera un fichier `CSV` contenant plusieurs lignes de nombres entiers avec un tableur pour vérifier notre fonction.

Ecrire une fonction `max_2d_list` prenant en paramètre une liste de listes et retournant le plus grand élément trouvé dans ces listes. On pourra utiliser deux boucles **pour** imbriquées ou mieux (en terme de modularité), utiliser la fonction `maximum`. On testera `max_2d_list(parse_csv_file(fichier))`.