

## TP n° 6 : Structures de données

### 1 Applications directes du cours

QUESTION 1 (*Produit des éléments d'une liste*) Écrire une fonction calculant le produit des éléments d'une liste d'entiers.

QUESTION 2 (*Somme des éléments vérifiant un prédicat*) Écrire une fonction calculant la somme des éléments divisibles par 5 d'une liste d'entiers positifs.

QUESTION 3 (*Nombre d'éléments vérifiant un prédicat*) Écrire une fonction calculant le nombre de caractères en majuscules d'une chaîne de caractères. On pourra utiliser la méthode `isupper`.

QUESTION 4 (*Calculs de sommes*) Écrire des fonctions calculant explicitement :

$$(S_n)^2 = \sum_{k=0}^n k^3 \qquad H_n = \sum_{k=1}^n \frac{1}{k} \qquad T_n = \sum_{k=1}^{n-1} \frac{1}{k(k+1)}$$

### 2 Méthodes à connaître

#### Méthode 1

Lorsque l'on souhaite parcourir une collection indexable, on se pose la question : « *est-il utile d'accéder aux indices ?* »

- Oui : on itère *par indices* : `for i in range(len(iterable))`
- Non : on itère *par éléments* : `for element in iterable`

#### Remarque 1

On peut toujours itérer par indices, même lorsque l'on a besoin uniquement des éléments, mais c'est quand même moins élégant.

QUESTION 5 Modifier la fonction suivante pour utiliser un parcours par éléments au lieu d'un parcours par indices :

PYTHON

```
def maximum(liste):
    """À compléter."""
    maxi = 0
    for i in range(len(liste)):
        if liste[i] > maxi:
            maxi = liste[i]
    return maxi
```

Que fait cette fonction ? Adapter la fonction pour calculer le maximum d'une liste non vide d'entiers quelconques.

#### Méthode 2

Si l'on a besoin d'accéder à la fois aux indices et aux éléments, on utilise une itération par indices (les plus avancés pourront utiliser `enumerate`).

QUESTION 6 Écrire une fonction qui parcourt une liste et qui affiche chaque élément en indiquant sa position dans la liste. On utilisera la méthode `format` des chaînes de caractères. Par exemple l'appel `affiche_elements([4, 2, 2])` affiche :

```
L'élément d'indice 0 est 4
L'élément d'indice 1 est 2
L'élément d'indice 2 est 2
```

#### Méthode 3

Lorsque l'on souhaite utiliser simultanément deux éléments successifs d'une liste, le parcours par indices est indiqué.

QUESTION 7 Écrire une fonction `est_triee(liste)` qui vérifie si une liste d'entiers est triée par ordre croissant. Attention à ne pas accéder à des éléments de la liste qui n'existeraient pas !

**Méthode 4**

Pour compter le nombre d'éléments dans une collection qui vérifient un prédicat donné :

1. on crée un compteur qui joue le rôle d'*accumulateur* (le plus souvent on l'initialise à 0). On choisit un nom de variable adapté à ce que l'on compte ;
2. on parcourt la collection par éléments ou par indices ;
3. pour chaque élément on incrémente le compteur s'il le faut ;
4. on renvoie la valeur du compteur.

On peut avoir besoin de plus d'un compteur ou de renvoyer plutôt le résultat d'un calcul faisant intervenir le compteur.

QUESTION 8 Écrire une fonction qui vérifie s'il y a strictement plus d'occurrences de la lettre "a" que de la lettre "e" dans une chaîne de caractères, c'est-à-dire qui renvoie **True** si c'est le cas et **False** sinon.

**Méthode 5**

Pour calculer une somme d'éléments dans une collection qui vérifient un prédicat donné, on procède comme dans la méthode précédente mais l'accumulateur joue le rôle de somme partielle.

QUESTION 9 (♥) Écrire une fonction qui calcule la moyenne d'une liste non vide de nombres.

**Méthode 6**

Pour savoir si une propriété sur une liste est vraie ou pour rechercher si au moins un élément satisfait un prédicat, on itère sur les éléments et on utilise le fait que **return** arrête la fonction pour interrompre la recherche au plus tôt.

QUESTION 10 (♥) Écrire une fonction `appartient(element, liste)` qui vérifie si un élément appartient à une liste, sans utiliser la construction `element in liste`.

**Méthode 7**

Pour créer une liste dont on connaît une expression du terme général :

1. on initialise une liste vide qui joue le rôle d'*accumulateur* ;
2. on itère sur un **range** ;
3. on ajoute le terme général à la liste avec la méthode **append** ;
4. on renvoie la liste ainsi construite.

QUESTION 11 Modifier la première fonction de la question 4 pour obtenir une fonction qui renvoie la liste des cubes des  $n$  premiers entiers naturels.

**Méthode 8**

Pour créer une ou plusieurs listes à partir d'une autre liste, on fait comme pour la méthode précédente, mais en itérant sur la liste, éventuellement avec plusieurs accumulateurs.

QUESTION 12 Écrire une fonction `separe_pairs_impairs(liste)` qui renvoie un couple de listes (**pairs**, **impairs**) contenant respectivement, dans le même ordre relatif, les éléments pairs et impairs d'une listes d'entiers.

**Méthode 9**

Pour créer une liste à partir de plusieurs listes, on procède comme précédemment avec une liste comme accumulateur, et pour mettre à jour l'accumulateur avec une sous-liste, on utilise au choix :

- `liste += sous_liste`
- `liste.extend(sous_liste)`

**Remarque 2**

*On pourrait aussi utiliser `liste = liste + sous_liste`, mais nous verrons prochainement pourquoi ce serait une mauvaise idée.*

QUESTION 13 Écrire une fonction `aplatir` qui, à partir d'une liste de listes, renvoie une liste formée de tous les éléments, dans l'ordre, de ces listes. Par exemple `aplatir([[0, 1], [2], [3, 4, 5]])` renvoie la liste `[0, 1, 2, 3, 4, 5]`.

## Méthode 10

Pour manipuler des chaînes de caractères, on use et on abuse des méthodes `format`, `split` et `join`.

QUESTION 14 Que fait le programme suivant ?

PYTHON

```
def mystere(x, y, z):
    """Docstring et noms de variables à modifier."""
    return z.join(x.split(y))
```

### 3 Des exercices pour s'entraîner

EXERCICE 1 (*Factorielle*) Écrire une fonction `factorielle(n)` qui calcule  $n!$ . *Indication* : réutiliser une fonction précédemment écrite.

EXERCICE 2 (*Quelques listes à produire*) Créer les listes (il s'agit de remplir les `...`, et de préférence pas à la main...) :

- L1 constituée de 12 zéros
- `L2 = [42, 41, 40, 39, ..., 26, 25, 24]`. On remarquera que `range` accepte un pas négatif en troisième argument
- `L3 = [0, 0, 0, 2, 2, 2, 4, 4, 4, ..., 10, 10, 10]`
- `L4 = [1, 2, 2, 3, 3, 3, ..., 7, 7, 7, 7, 7, 7, ..., 10, 10]`
- `L5 = [1, 2, 1, 3, 2, 1, 4, 3, 2, 1, ..., 9, 8, 7, 6, 5, 4, 3, 2, 1]`

EXERCICE 3 Écrire une fonction `table_multiplication(m, n)` qui prend en arguments deux entiers  $n$  et  $m$  et qui imprime à l'écran la table de multiplication de  $m$  pour les  $n$  premiers entiers.

Par exemple, `table_multiplication(9, 4)` affiche :

```
0 fois 9 = 0
1 fois 9 = 9
2 fois 9 = 18
3 fois 9 = 27
```

Modifier la fonction pour renvoyer en sortie une chaîne de caractères contenant cette table *au lieu* de l'afficher. On rappelle que le caractère spécial `'\n'` correspond au caractère indiquant un passage à la ligne dans une chaîne de caractère. L'appel `table_multiplication(9, 4)` n'affichera donc rien, en revanche l'appel `print(table_mul4)` doit afficher exactement la même chose que ci-dessus.

EXERCICE 4

- Écrire une fonction `somme_chiffres(n)` qui renvoie la somme des chiffres de l'écriture décimale de l'entier  $n$ . Par exemple `somme_chiffre(139)` doit renvoyer 13 car  $1 + 3 + 9 = 13$ . *Indication* : passer par les chaînes de caractères.
- Résoudre les problèmes du PROJECT EULER :
  - <http://projecteuler.net/problem=16>
  - <http://projecteuler.net/problem=20>

EXERCICE 5

Écrire des fonctions qui permettent d'afficher des triangles à  $n$  lignes de la forme :

```
*           *           *           *           *
**          --          **          ***          * *
***         ***         * *         *****         * *
****        ----        * *         *          *
*****       *****       * *         *          *
*****      - - - -      *          *          *
*****      *          *          *          *          *
```

(AN UNMATCHED LEFT PARENTHESIS  
CREATES AN UNRESOLVED TENSION  
THAT WILL STAY WITH YOU ALL DAY.

<https://xkcd.com/859/>