

TP – Listes Python

Objectifs

- ◊ Savoir créer, parcourir et modifier une liste Python.
- ◊ Savoir construire une nouvelle liste par ajouts successifs.
- ◊ Savoir parcourir une liste par valeurs ou par indices.
- ◊ Choisir un parcours adapté selon le problème posé : parcours complet, recherche avec arrêt, parcours par indices.
- ◊ Comprendre la notion de modification en place d'une liste.

Dans ce TP, on n'utilise pas les fonctions ou méthodes Python qui résolvent directement les questions posées, comme `max`, `min`, la méthode `sort`, `sorted`, ou l'opérateur d'appartenance `elt in lst`.

Travail en classe

Exercice 1 – Premières manipulations de listes

Question 1. On considère deux listes d'entiers de même longueur n :

$$\ell = [\ell_0, \ell_1, \dots, \ell_{n-1}] \quad m = [m_0, m_1, \dots, m_{n-1}]$$

On peut former deux autres listes d'entiers de longueur n , notées s et p , telles que, pour tout indice entier $i \in \{0, \dots, n-1\}$:

$$s_i = \ell_i + m_i \quad p_i = \ell_i \times m_i$$

Écrire la fonction `sum_prod(lst1: list, lst2: list) -> tuple` qui prend en argument deux listes de même longueur et qui renvoie le couple `(s, p)` des deux listes définies ci-dessus. Par exemple :

```
assert sum_prod([1, 2, 3], [4, 5, 6]) == ([5, 7, 9], [4, 10, 18])
assert sum_prod([], []) == ([], [])
```

Question 2. Par analogie avec le produit cartésien de deux ensembles, on souhaite construire la liste de toutes les paires formées d'un élément d'une première liste et d'un élément d'une seconde liste.

Écrire la fonction `cartesian_prod(lst1: list, lst2: list) -> list` qui prend en argument deux listes et qui renvoie la liste des paires formées d'un élément de

`lst1` et d'un élément de `lst2`. Chaque paire sera stockée sous la forme d'une liste à deux éléments. Par exemple, l'appel :

```
| cartesian_prod([1, 2], ["a", "b", "c"])
```

renvoie :

```
| [[1, "a"], [1, "b"], [1, "c"], [2, "a"], [2, "b"], [2, "c"]]
```

Exercice 2 – Autour du maximum d'une liste

Question 1. On veut écrire une fonction `maximum(lst: list) -> int` qui renvoie la valeur du plus grand élément de la liste `lst`. Cette liste est supposée non vide et ne contenir que des entiers. On donne l'algorithme sur lequel repose la fonction.

Algorithme 1 : recherche du maximum d'une liste non vide

Entrée : une liste non vide ℓ

Sortie : le plus grand élément de ℓ

```
1  $m \leftarrow \ell_0$ 
2 foreach  $x \in \ell$  do
3   | si  $x > m$  alors
4   |   |  $m \leftarrow x$ 
5 renvoyer  $m$ 
```

Écrire cette fonction en utilisant un parcours de liste de la forme `for elt in lst`. Tester la fonction sur des listes présentant des situations diverses : maximum en première position, maximum en dernière position, maximum ailleurs que sur les bords, liste à un seul élément, élément maximal apparaissant plusieurs fois.

Question 2. On peut noter que l'emploi de `for elt in lst` entraîne un test inutile, celui du premier élément de la liste. Cependant, la syntaxe associée est très simple et bien adaptée à la situation. Modifier la fonction précédente en utilisant le slicing `lst[1:]` pour éviter ce test inutile.

Le slicing `lst[1:]` construit une nouvelle liste contenant tous les éléments de `lst` sauf le premier.

Question 3. Écrire, sur le même principe, une fonction `minimum(lst: list) -> int` renvoyant la valeur du plus petit élément de la liste `lst`. Tester cette fonction sur plusieurs listes bien choisies.

Question 4. Écrire une fonction `indice_maximum(lst: list) -> int` renvoyant le plus petit indice `k` tel que `lst[k]` soit le plus grand élément de la liste `lst`, en

utilisant une boucle **for**. Par exemple, pour la liste `lst = [2, 4, 1, 5, 3]`, la fonction renvoie 3, puisque le plus grand élément vaut 5 et apparaît à l'indice 3. Quel type de parcours de liste est-il nécessaire d'utiliser ici ?

Question 5. Réécrire la fonction précédente en utilisant une boucle **while**.

Exercice 3 – Présence d'un élément dans une liste

Question 1. On cherche à écrire une fonction `indices_presents(lst: list, elt) -> list` qui renvoie la liste des indices où l'élément `elt` apparaît dans la liste `lst`. Si `elt` n'appartient pas à `lst`, la fonction renvoie 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** ? Justifier.

■ **1.2.** Quel type de parcours est nécessaire si l'on veut accéder aux indices des éléments de la liste ?

■ **1.3.** Écrire la fonction demandée.

Question 2. On cherche désormais à écrire une fonction `est_present(lst: list, elt) -> bool` qui renvoie **True** si l'élément `elt` appartient à `lst`, et **False** sinon. Un élève propose la fonction suivante :

```
def est_present(lst: list, elt) -> bool:
    """Renvoie True si elt appartient a lst, False sinon."""
    trouve = False
    for valeur in lst:
        if valeur == elt:
            trouve = True
    return trouve
```

Expliquer quel est le défaut de cette façon de programmer. Proposer une ou plusieurs améliorations.

Question 3. On donne ci-dessous le code incomplet d'une fonction qui réalise la même action que `est_present`, en utilisant une boucle **while**.

```
def est_present_while(lst: list, elt) -> bool:
    """Renvoie True si elt appartient a lst, False sinon."""
    i = 0
    n = len(lst)
    trouve = False
    while i < n and not trouve:
        if ...:
```

```
        trouve = True
        i = ...
    return trouve
```

■ **3.1.** Compléter le code pour rendre la fonction opérationnelle.

■ **3.2.** Expliquer la nécessité de mettre le test `i < n` dans la condition du **while**.

Question 4. Une autre possibilité, sans utiliser le booléen `trouve`, est de parcourir la liste avec un indice `i` tant que l'on n'est pas arrivé au bout de la liste et tant que l'élément courant `lst[i]` est différent de l'élément `elt` recherché. Proposer le code d'une fonction `est_present_sans_booleen(lst: list, elt) -> bool` qui suit ce principe. Dans la condition du **while**, on prendra soin de tester `i < len(lst)` avant de tester la valeur de `lst[i]`.

Travail à la maison

Exercice 4 – Doublons dans une liste

On cherche à écrire une fonction `sans_doublon(lst: list) -> bool` qui prend comme argument une liste et qui renvoie **True** si les éléments de `lst` sont tous distincts, et **False** sinon.

Pour une liste `lst` de taille n , on peut utiliser l'algorithme suivant.

Algorithme 2 : test d'absence de doublon dans une liste

Entrée : une liste `lst` de taille n

Sortie : **True** si les éléments de `lst` sont tous distincts, **False** sinon

```
1 pour i allant de 0 à n - 2 faire
2     pour j allant de i + 1 à n - 1 faire
3         si lst[i] = lst[j] alors
4             renvoyer False
5 renvoyer True
```

Question 1. Écrire une fonction Python `sans_doublon(lst: list) -> bool` qui implémente cet algorithme à l'aide de deux boucles **for** imbriquées.

Par exemple :

```
assert sans_doublon([2, 4, 1, 5, 3])
assert not sans_doublon([2, 4, 1, 5, 4])
assert sans_doublon([])
assert sans_doublon([7])
```

```
| assert not sans_doublon([3, 3])
```

Question 2. Dans la fonction précédente, les boucles **for** ne sont pas toujours menées à leur terme : dès que deux éléments égaux sont trouvés, la fonction renvoie **False** et le parcours s'interrompt.

Expliquer pourquoi des boucles **while** peuvent alors sembler plus pertinentes que des boucles **for**.

Question 3. Écrire une fonction `sans_doublon_while(lst: list) -> bool` qui réalise le même test, mais en utilisant des boucles **while**.

On pourra introduire une variable booléenne `aucun_doublon`, qui vaut **True** tant qu'aucun doublon n'a été trouvé, et **False** dès qu'un doublon est détecté.

Exercice 5 – Tri à bulles

Les tris étudiés dans cet exercice ont surtout un intérêt pédagogique. Ils permettent de travailler les parcours de listes par indices, les boucles imbriquées, les échanges d'éléments et la notion de tri en place. En revanche, ils sont peu utilisés en pratique, car leur complexité est quadratique dans le pire cas.

Trier une liste consiste à réordonner ses éléments selon une relation d'ordre. Dans cet exercice, on trie des listes de nombres dans l'ordre croissant. Ainsi, trier la liste `[5, 1, 2, 4, 3]` consiste à obtenir la liste `[1, 2, 3, 4, 5]`.

Un tri est dit *en place* lorsqu'il modifie directement la liste donnée en argument, sans construire une nouvelle liste de même taille contenant les mêmes éléments dans le bon ordre. Le tri étudié ici est un tri en place : il réorganise les éléments de `lst` par échanges successifs.

On s'intéresse à une méthode de tri appelée *tri à bulles*. Elle s'applique à une liste `lst` de n éléments. Le principe est le suivant : on parcourt la liste de gauche à droite en comparant deux éléments consécutifs. Si deux éléments voisins ne sont pas dans le bon ordre, on les échange.

Lors d'un premier passage, on compare donc successivement les couples

`(lst[0], lst[1]), (lst[1], lst[2]), ..., (lst[n - 2], lst[n - 1]).`

À la fin de ce passage, le plus grand élément de la liste se trouve en dernière position. On recommence alors le même processus, mais en s'arrêtant une case plus tôt. Les plus grands éléments remontent ainsi progressivement vers la fin de la liste, comme des bulles dans une boisson gazeuse.

Question 1. Appliquer à la main le tri à bulles à la liste

`[5, 1, 2, 4, 3]`.

Indiquer l'état de la liste après chaque passage complet de gauche à droite.

Question 2. Écrire une fonction `tri_bulle(lst: list) -> list` qui trie la liste `lst` selon le principe décrit précédemment.

La fonction modifie la liste reçue en argument et la renvoie.

Question 3. Dans certaines situations, la liste peut être triée avant que tous les passages prévus aient été effectués.

Expliquer comment on peut détecter cette situation.

Question 4. Écrire une fonction `tri_bulle_optimise(lst: list) -> list` qui s'arrête dès qu'un passage complet ne réalise aucun échange.

La fonction modifie la liste reçue en argument et la renvoie.

Pour aller plus loin

Exercice 6 – Tri boustrophédon

Le *tri boustrophédon* est une variante du tri à bulles. Le mot *boustrophédon* désigne un parcours qui alterne les directions, comme le trajet d'un bœuf qui laboure un champ : une ligne dans un sens, la suivante dans l'autre.

Dans le tri à bulles, chaque passage se fait de gauche à droite. Dans le tri boustrophédon, on effectue alternativement :

- ◊ un passage de gauche à droite, qui pousse le plus grand élément restant vers la droite ;
- ◊ un passage de droite à gauche, qui pousse le plus petit élément restant vers la gauche.

Après chaque aller-retour, la zone restant à trier se réduit donc à la fois à gauche et à droite.

Question 1. Appliquer à la main le principe du tri boustrophédon à la liste

`[5, 1, 2, 4, 3]`.

Indiquer l'état de la liste après chaque passage de gauche à droite, puis après chaque passage de droite à gauche.

Question 2. Écrire une fonction `tri_boustrophedon(lst: list) -> list` qui trie la liste `lst` selon ce principe.

La fonction modifie la liste reçue en argument et la renvoie.

On pourra utiliser deux indices `gauche` et `droite` pour délimiter la partie de la liste qui reste à trier.

Question 3. Dans le tri à bulles optimisé, on peut s'arrêter lorsqu'un passage complet ne réalise aucun échange.
Adapter cette idée au tri boustrophédon.

Exercice 7 – Second maximum

Cet exercice est plus délicat que les précédents.

On veut écrire une fonction `second_maximum(lst: list) -> int | None` qui prend comme argument une liste non vide d'entiers `lst` et qui renvoie son second maximum strict.

Le second maximum strict est la plus grande valeur de la liste qui est strictement inférieure au maximum. Par exemple, le second maximum strict de la liste

[2, 4, 1, 5, 3]

est 4.

Pour que cette notion soit définie, il faut que la liste contienne au moins deux valeurs distinctes. Par exemple, la liste [3, 3, 3] n'a pas de second maximum strict. Dans ce cas, la fonction renvoie `None`.

Le principe proposé est le suivant :

- ◊ on commence par chercher le premier élément de `lst` qui est différent de `lst[0]` ;
- ◊ si un tel élément n'existe pas, tous les éléments de la liste sont égaux et la fonction renvoie `None` ;
- ◊ sinon, on dispose de deux valeurs distinctes : `lst[0]` et cet autre élément ;
- ◊ on initialise alors correctement deux variables `maximum_actuel` et `second_maximum_actuel` avec la plus grande et la plus petite de ces deux valeurs ;
- ◊ on parcourt ensuite le reste de la liste en mettant à jour ces deux variables chaque fois que c'est nécessaire ;
- ◊ à la fin du parcours, la fonction renvoie `second_maximum_actuel`.

Question 1. Expliquer pourquoi la première recherche se prête naturellement à une boucle `while`.

Question 2. Écrire la fonction `second_maximum`. On prendra soin de choisir, pour chacune des deux phases de l'algorithme, le type de boucle le plus adapté.

Question 3. Tester la fonction sur plusieurs listes bien choisies, par exemple :

[2, 4, 1, 5, 3], [5, 5, 4], [3, 3, 3], [1, 2], [2, 1].