

Chercher le meilleur, copier une liste

● Méthode

Exercice flash (10 minutes, sur papier). Écrire une fonction `hausses(L: list) -> list` qui renvoie la liste des indices `i` tels que `L[i+1] > L[i]`. Exemple : `hausses([3, 5, 2, 4, 4, 6])` renvoie `[0, 2, 4]`.

Corrigé.

```
def hausses(L: list) -> list:
    '''Renvoie la liste des indices i tels que L[i+1] > L[i].'''
    res = []
    for i in range(len(L) - 1):
        if L[i + 1] > L[i]:
            res.append(i)
    return res
```

● Méthode

Pour chaque fonction : **exemple à la main, entrée/sortie, schéma, code puis trace**. Se demander systématiquement : renvoie-t-on une *valeur* ou un *indice*? Les fonctions `max`, `min`, `sort`, `sorted`, `index` et `deepcopy` sont interdites.

● Exercice 1

Échauffement. Les listes sont supposées non vides. Pour chaque fonction, écrire d'abord un exemple d'appel et le résultat attendu.

1. `maximum(L: list) -> float` renvoie le plus grand élément de `L`. Tester mentalement votre fonction sur `[-3, -1, -7]`.
2. `indice_min(L: list) -> int` renvoie l'indice du premier plus petit élément de `L`.
3. `dernier_indice_max(L: list) -> int` renvoie l'indice du *dernier* plus grand élément. Exemple : `dernier_indice_max([4, 9, 2, 9])` renvoie 3.
4. `amplitude(L: list) -> float` renvoie la différence entre le plus grand et le plus petit élément, en un seul parcours de la liste.
5. `plus_long(mots: list) -> str` renvoie le premier mot de longueur maximale de la liste de chaînes `mots`.

Corrigé.

```
def maximum(L: list) -> float:
    '''L non vide. Renvoie le plus grand élément de L.'''
    m = L[0] # et non 0 : faux sur [-3, -1, -7]
    for x in L:
        if x > m:
            m = x
    return m

def indice_min(L: list) -> int:
    '''L non vide. Renvoie l'indice du premier minimum de L.'''
    i_min = 0
    for i in range(1, len(L)):
        if L[i] < L[i_min]:
            i_min = i
    return i_min

def dernier_indice_max(L: list) -> int:
```

```

'''L non vide. Renvoie l'indice du dernier maximum de L.'''
i_max = 0
for i in range(1, len(L)):
    if L[i] >= L[i_max]: # >= : on remplace aussi en cas d'égalité
        i_max = i
return i_max

def amplitude(L: list) -> float:
    '''L non vide. Renvoie max(L) - min(L), en un seul parcours.'''
    m = L[0]
    M = L[0]
    for x in L:
        if x < m:
            m = x
        if x > M:
            M = x
    return M - m

def plus_long(mots: list) -> str:
    '''mots : liste non vide de chaînes.
    Renvoie le premier mot de longueur maximale.'''
    meilleur = mots[0]
    for mot in mots:
        if len(mot) > len(meilleur):
            meilleur = mot
    return meilleur

```

● Exercice 2

Qu'affiche le programme? Pour chaque programme, prévoir l'affichage *sans ordinateur*, puis justifier en une phrase. Vérifier ensuite sur machine.

```

# (a)
L = [1, 2, 3]
M = L
M[0] = 9
print(L)

# (b)
L = [1, 2, 3]
M = L.copy()
M[0] = 9
print(L)

# (c)
M = [[1, 2], [3, 4]]
N = M.copy()
N[0][0] = 9
N.append([5])
print(M)

```

```

# (d)
def f(L):
    L.append(0)

A = [1]
f(A)
print(A)

# (e)
def g(L):
    L = L + [0]

B = [1]
g(B)
print(B)

# (f)
T = [[0] * 3] * 2
T[0][0] = 1
print(T)

```

Corrigé.

- (a) [9, 2, 3] : **M** et **L** désignent la même liste.
- (b) [1, 2, 3] : **M** est une nouvelle liste.
- (c) [[9, 2], [3, 4]] : la copie ne porte que sur le premier niveau. **N** est une nouvelle liste (l'ajout de [5] ne touche pas **M**), mais ses éléments sont les *mêmes* lignes que celles de **M**.
- (d) [1, 0] : la fonction modifie en place la liste reçue.

- (e) `[1] : L + [0]` crée une nouvelle liste, et `L = ...` change seulement ce que désigne le nom local `L`; `B` n'est pas modifiée.
- (f) `[[1, 0, 0], [1, 0, 0]]` : les deux lignes sont la même liste. Écrire `T = [[0] * 3 for i in range(2)]`.

● Exercice 3

Point culminant d'une randonnée (d'après CCINP PC 2025).

Un itinéraire est une liste non vide de triplets (`latitude`, `longitude`, `altitude`) ; le premier point est le départ. Le *dénivelé global* est la différence entre l'altitude maximale et l'altitude du point de départ.

1. Écrire une fonction `altitude_maximale(iti: list) -> float`.
2. En utilisant la fonction précédente, écrire `denivele_global(iti: list) -> float`. Tester sur l'itinéraire dont les altitudes sont 812, 1045, 1320 et 1280.
3. Écrire une fonction `point_culminant(iti: list) -> tuple` qui renvoie le couple (`latitude`, `longitude`) du point le plus haut.
4. Écrire une fonction `altitude_minimale_apres(iti: list, k: int) -> float` qui renvoie l'altitude minimale des points d'indices `k` à la fin, avec $0 \leq k < \text{len}(\text{iti})$.
5. Quelle est la complexité de `altitude_maximale`? Pourquoi serait-il maladroit de trier la liste des altitudes pour trouver le maximum?

Corrigé.

```
def altitude_maximale(iti: list) -> float:
    '''iti : liste non vide de triplets (lat, long, alt).
    Renvoie l'altitude maximale de l'itinéraire.'''
    m = iti[0][2]
    for (lat, long, alt) in iti:
        if alt > m:
            m = alt
    return m

def denivele_global(iti: list) -> float:
    '''Renvoie l'altitude maximale moins l'altitude du départ.'''
    return altitude_maximale(iti) - iti[0][2]

def point_culminant(iti: list) -> tuple:
    '''Renvoie (latitude, longitude) du premier point le plus haut.'''
    i_max = 0
    for i in range(1, len(iti)):
        if iti[i][2] > iti[i_max][2]:
            i_max = i
    return (iti[i_max][0], iti[i_max][1])

def altitude_minimale_apres(iti: list, k: int) -> float:
    '''0 <= k < len(iti). Renvoie l'altitude minimale des points
    d'indices k à len(iti) - 1.'''
    m = iti[k][2]
    for i in range(k + 1, len(iti)):
        if iti[i][2] < m:
            m = iti[i][2]
    return m
```

Question 2 : $1320 - 812 = 508$ m. Question 5 : un seul parcours, donc $O(n)$; un tri coûte $O(n \log n)$ et ne sert à rien ici.

● Exercice 4

Classer des spectres (d'après CCINP PC 2026).

Un spectre est une liste de flottants. On dispose de la fonction `distance(S1, S2)` écrite dans le TD sur l'accumulateur. On dispose aussi d'une liste `centres` de spectres de référence, tous de même longueur.

1. Écrire une fonction `centre_le_plus_proche(S: list, centres: list) -> int` qui renvoie l'indice du spectre de `centres` le plus proche de `S`.
2. Pourquoi est-il préférable de ne calculer `distance(S, centres[i])` qu'une fois par tour de boucle? Modifier la fonction en conséquence.
3. Écrire une fonction `partition(spectres: list, centres: list) -> list` qui renvoie la liste des indices des centres les plus proches de chaque spectre de `spectres`.
4. Avec `centres = [[0, 0], [1, 2], [5, 5]]` et `spectres = [[1, 1], [4, 6], [0, 1]]`, calculer à la main le résultat de `partition`. On donne les distances arrondies :

	[0, 0]	[1, 2]	[5, 5]
[1, 1]	1,00	0,71	4,00
[4, 6]	5,10	3,54	1,00
[0, 1]	0,71	1,00	4,53

Corrigé.

```
def centre_le_plus_proche(S: list, centres: list) -> int:
    '''centres : liste non vide de spectres de même longueur que S.
    Renvoie l'indice du centre le plus proche de S.'''
    i_min = 0
    d_min = distance(S, centres[0]) # distance du meilleur candidat
    for i in range(1, len(centres)):
        d = distance(S, centres[i]) # calculée une seule fois
        if d < d_min:
            i_min = i
            d_min = d
    return i_min

def partition(spectres: list, centres: list) -> list:
    '''Renvoie la liste des indices des centres les plus proches
    de chaque spectre de spectres.'''
    res = []
    for S in spectres:
        res.append(centre_le_plus_proche(S, centres))
    return res
```

Question 2 : chaque calcul de distance coûte $O(N)$; la comparaison `distance(S, centres[i]) < distance(S, centres[i_min])` en ferait deux par tour. On mémorise donc aussi `d_min` : c'est un cas où garder deux variables est justifié.

Question 4 : [1, 2, 0].

● Exercice 5

Pour aller plus loin : copier une partie d'awalé (d'après CCINP PC 2024).

L'état d'une partie est un dictionnaire `jeu` construit ainsi :

```
jeu = {}
jeu['joueur1'] = 'Alice' # nom du premier joueur
jeu['joueur2'] = 'Bob'   # nom du second joueur
jeu['score'] = [0, 0]    # graines récoltées par chaque joueur
jeu['n'] = 0             # nombre de tours déjà joués
jeu['plateau'] = [4] * 12 # graines de chaque case
```

1. Pour tester un coup sans jouer vraiment, on écrit `essai = jeu.copy()`, puis on modifie `essai['plateau'][3]`. Le plateau de `jeu` est-il modifié? Justifier.
2. Écrire une fonction `copie(jeu: dict) -> dict` qui renvoie une copie complète de `jeu` : aucune modification de la copie ne doit modifier `jeu`.

3. On dispose de la fonction `deplacer_graines(plateau, case)` du TD sur les éléments consécutifs, qui modifie le plateau en place. Écrire une fonction `affame(plateau: list, case: int) -> bool` qui renvoie **True** si, après avoir semé `case`, les cases 6 à 11 sont toutes vides, et **False** sinon. L'argument `plateau` ne doit pas être modifié.

Corrigé.

1. Oui : `jeu.copy()` crée un nouveau dictionnaire, mais la valeur associée à '`plateau`' est la *même* liste dans les deux dictionnaires.
2. et 3.

```
def copie(jeu: dict) -> dict:
    '''Renvoie une copie de jeu dont les listes sont elles aussi copiées.'''
    res = {}
    for cle in jeu:
        if type(jeu[cle]) == list:
            res[cle] = jeu[cle].copy()    # nouvelle liste
        else:
            res[cle] = jeu[cle]           # str et int ne sont pas modifiables
    return res

def affame(plateau: list, case: int) -> bool:
    '''Renvoie True si semer case vide toutes les cases 6 à 11.
    plateau n'est pas modifié.'''
    p = plateau.copy()                  # on travaille sur une copie
    deplacer_graines(p, case)
    for i in range(6, 12):
        if p[i] > 0:
            return False                # une case non vide suffit
    return True
```

● Méthode

Travail personnel : deux exercices flash (15 minutes chacun).

1. `plus_petit_positif(L: list) -> float` renvoie le plus petit élément strictement positif de `L`, ou **None** s'il n'y en a aucun. Exemple : `plus_petit_positif([-2, 5, 0, 3, 8])` renvoie 3.
2. `copie_matrice(M: list) -> list` renvoie une copie de la liste de listes `M` dont la modification ne change pas `M`. On n'utilisera ni compréhension de liste ni `deepcopy`.

Corrigé.

```
def plus_petit_positif(L: list) -> float:
    '''Renvoie le plus petit élément strictement positif de L,
    ou None s'il n'y en a pas.'''
    meilleur = None                    # pas encore de candidat
    for x in L:
        if x > 0 and (meilleur is None or x < meilleur):
            meilleur = x
    return meilleur

def copie_matrice(M: list) -> list:
    '''Renvoie une copie de M, ligne par ligne.'''
    res = []
    for ligne in M:
        res.append(ligne.copy())
    return res
```