

Fiche méthode : accumulateur

De l'énoncé au code – Le schéma « accumulateur »

1. La routine en 4 étapes

À appliquer à **chaque question** où l'on demande d'écrire une fonction, même si elle paraît facile.

● Méthode

1. **Un exemple à la main.** Reprendre l'exemple de l'énoncé, ou en inventer un tout petit (3 ou 4 éléments). Écrire le résultat attendu.
2. **Entrée / sortie.** Quels sont les types des arguments? Que renvoie-t-on (un nombre, une liste, un booléen, rien)? A-t-on le droit de modifier les arguments?
3. **Reconnaître le schéma.** Se demander ce que l'on fait à *la main* sur l'exemple : « je parcours et j'additionne », « je cherche le plus grand », « je fabrique une nouvelle liste » ...
4. **Squelette, remplissage, trace.** Écrire le squelette du schéma, l'adapter à la question, puis exécuter le code à *la main* sur l'exemple de l'étape 1 en suivant les variables.

Les étapes 1 et 2 prennent une minute et évitent la plupart des blocages : on ne sait pas écrire un programme qui fait quelque chose que l'on ne sait pas faire soi-même sur un exemple.

2. Le schéma accumulateur

On parcourt une liste et on met à jour une variable à chaque élément. C'est le schéma le plus fréquent aux concours.

```
def nom_fonction(L: list) -> type_resultat:
    '''Spécification : ce que renvoie la fonction.'''
    acc = VALEUR_INITIALE          # 1. avant la boucle
    for x in L:                    # 2. parcours
        acc = MISE_A_JOUR(acc, x) # (éventuellement sous condition)
    return acc                     # 3. après la boucle
```

On veut...	Valeur initiale	Mise à jour
une somme	0	<code>acc = acc + x</code>
un nombre d'éléments vérifiant une condition	0	<code>if condition: acc = acc + 1</code>
un produit	1	<code>acc = acc * x</code>
une somme de carrés d'écart	0	<code>acc = acc + (x - m)**2</code>

3. Parcours par valeur ou par indice?

```
for x in L:          # par valeur
    ... x ...
```

```
for i in range(len(L)): # par indice
    ... L[i] ...
```

● Méthode

- Par défaut, parcourir **par valeur** : c'est plus court et plus sûr.
- Parcourir **par indice** quand on a besoin de la position : renvoyer un indice, comparer `L[i]` à `L[i+1]`, parcourir **deux listes en parallèle** (`S1[i]` et `S2[i]`).
- Quand les éléments sont des tuples, on peut les dépaqueter : `for (lat, long, alt) in iti:`

4. Les quatre pièges classiques

● Attention !

1. **Initialisation dans la boucle** : l'accumulateur est remis à zéro à chaque tour.
2. **return dans la boucle** : la fonction s'arrête au premier élément.
3. **print au lieu de return** : le résultat s'affiche, mais la fonction renvoie **None** et on ne peut pas l'utiliser dans un calcul.
4. **Mélanger valeur et indice** : `for x in L`: puis `L[x]`.

Le repère visuel : l'initialisation et le **return** sont **au même niveau d'indentation que le for**.

5. Exemple rédigé

Écrire une fonction `nb_depassements(T: list, seuil: float) -> int` qui renvoie le nombre de relevés de température de `T` strictement supérieurs à `seuil`.

1. **Exemple** : `T = [18.5, 21.0, 19.2, 23.4]`, `seuil = 20` → deux dépassements (21.0 et 23.4), résultat 2.
2. **Entrée / sortie** : une liste de flottants et un flottant ; on renvoie un entier ; `T` n'est pas modifiée.
3. **Schéma** : « je parcours les relevés et je compte ceux qui dépassent » → accumulateur de comptage, parcours par valeur.
4. **Code et trace** :

```
def nb_depassements(T: list, seuil: float) -> int:
    '''Renvoie le nombre d'éléments de T
    strictement supérieurs à seuil.'''
    n = 0                # aucun dépassement vu
    for t in T:
        if t > seuil:
            n = n + 1    # un de plus
    return n
```

t	t > 20	n
–	–	0
18.5	non	0
21.0	oui	1
19.2	non	1
23.4	oui	2

● Bilan

Au concours, le code doit être « expliqué et commenté » : une ligne de spécification et un commentaire sur le rôle de chaque variable suffisent.