

Fiche méthode : consécutifs et listes

Éléments consécutifs – Construire une liste

1. Le schéma « éléments consécutifs »

On compare ou combine chaque élément avec son **voisin**. Le parcours se fait obligatoirement **par indice**.

```
for i in range(len(L) - 1): # i va de 0 à len(L) - 2
    ... L[i] ... L[i + 1] ...
```

● Méthode

Pour trouver les bornes, écrire les indices sur un exemple. Avec $L = [a, b, c, d]$ (longueur 4), les couples à traiter sont $(L[0], L[1])$, $(L[1], L[2])$, $(L[2], L[3])$: i prend les valeurs 0, 1, 2, soit `range(3)`, c'est-à-dire `range(len(L) - 1)`. Il y a **un couple de moins que d'éléments**.

Écriture	Boucle
avec le suivant <code>L[i + 1]</code>	<code>for i in range(len(L) - 1):</code>
avec le précédent <code>L[i - 1]</code>	<code>for i in range(1, len(L)):</code>
circulaire : après la dernière case, on revient à la case 0	<code>for i in range(n):</code> avec <code>L[(i + 1) % n]</code>

2. Le schéma « construire une liste »

C'est un accumulateur dont la valeur initiale est la liste vide et la mise à jour un `append`.

```
def nom_fonction(L: list) -> list:
    '''Spécification.'''
    res = [] # avant la boucle
    for x in L:
        if CONDITION: # facultatif : on filtre
            res.append(EXPR) # on ajoute x, ou une valeur calculée
    return res # après la boucle
```

On veut la liste...	Ce que l'on ajoute
des éléments qui vérifient une condition (filtrer)	<code>res.append(x)</code> sous condition
des images des éléments (transformer)	<code>res.append(f(x))</code>
des <i>indices</i> vérifiant une condition	parcours par indice, <code>res.append(i)</code>
des écarts entre voisins	<code>res.append(L[i + 1] - L[i])</code>

3. Tester une propriété : le `return` anticipé

Pour savoir si **tous** les éléments vérifient une propriété, il suffit de trouver **un contre-exemple**. C'est le seul cas où un `return` dans la boucle est correct.

```
def est_croissante(L: list) -> bool:
    '''Renvoie True si L est croissante (au sens large), False sinon.'''
    for i in range(len(L) - 1):
        if L[i + 1] < L[i]:
            return False # contre-exemple trouvé : on arrête
    return True # après la boucle : aucun contre-exemple
```

4. Les pièges

● Attention !

1. `range(len(L))` avec `L[i + 1]` : au dernier tour, `L[len(L)]` n'existe pas, d'où une `IndexError`.
2. `range(len(L))` avec `L[i - 1]` : **aucune erreur n'est signalée**, mais pour `i = 0`, `L[-1]` désigne le *dernier* élément. Le résultat est faux en silence.
3. `res = res.append(x)` : `append` modifie la liste et renvoie `None` ; `res` vaut alors `None`. Écrire seulement `res.append(x)`.
4. Dans un test « pour tout », écrire `else: return True` dans la boucle : la fonction conclut dès le premier couple.

5. Exemple rédigé

Écrire une fonction `variations(T: list) -> list` qui renvoie la liste des variations de température entre deux relevés successifs.

1. **Exemple** : `T = [18, 21, 19, 19]` $\rightarrow$ `[3, -2, 0]` (trois valeurs pour quatre relevés).
2. **Entrée / sortie** : une liste de nombres ; une nouvelle liste de longueur `len(T) - 1` ; `T` n'est pas modifiée.
3. **Schéma** : consécutifs (on a besoin du voisin) et construction de liste (on renvoie une liste).
4. **Code et trace** :

```
def variations(T: list) -> list:
    '''Renvoie la liste des T[i+1] - T[i].'''
    res = []
    for i in range(len(T) - 1):
        res.append(T[i + 1] - T[i])
    return res
```

i	T[i+1] - T[i]	res
-	-	[]
0	21 - 18 = 3	[3]
1	19 - 21 = -2	[3, -2]
2	19 - 19 = 0	[3, -2, 0]