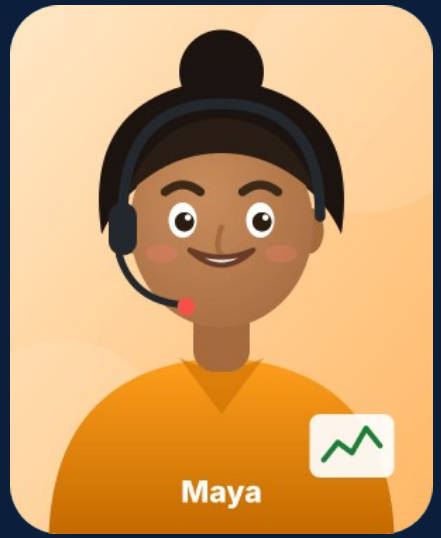
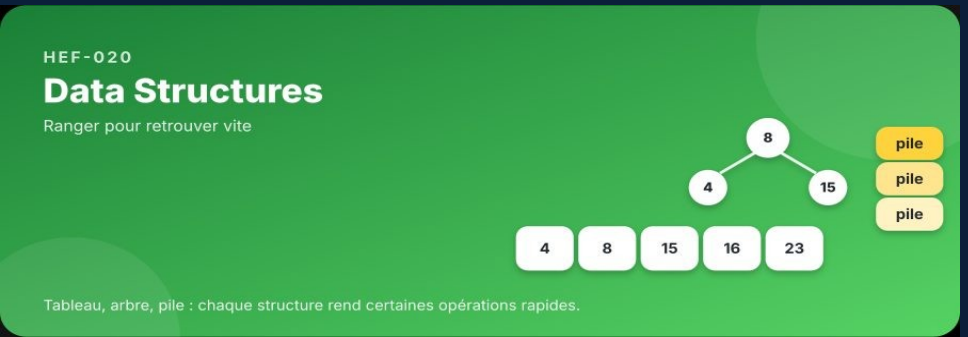
HEF-020 · MISSION 02 / 10

Tableaux et listes chaînées

Deux façons de ranger des éléments en ligne : une rangée de cases, ou une chaîne de maillons.





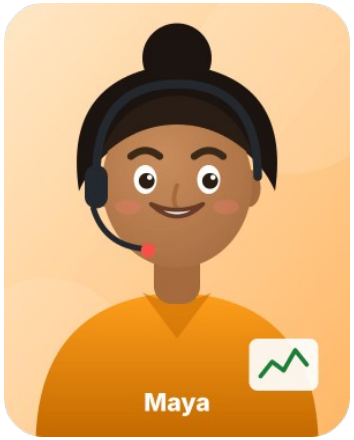
Badge : Maillon fort

75 min · débutant

L'histoire

Kodi doit gérer la **liste d'attente** d'une formation : on y ajoute des membres en fin de liste, parfois au milieu pour les prioritaires, et on retire ceux qui abandonnent.

Il utilise une liste Python. Tout marche, jusqu'à 200 000 inscrits : insérer au début devient très lent.



> *« Il faut savoir ce qu'il y a derrière cette liste », dit Sam.*

Tu découvres les deux grandes familles de structures linéaires.

Ta mission

À la fin, je serai capable de…

1. expliquer comment un tableau stocke ses éléments
2. expliquer le principe d'une liste chaînée
3. comparer accès, insertion et suppression des deux
4. implémenter une liste chaînée simple
5. choisir l'une ou l'autre selon l'usage

Défi d'échauffement

Imagine deux façons d'organiser 10 personnes :

- **A** : elles sont assises sur un banc numéroté de 0 à 9 ;
- **B** : elles forment une chaîne, chacune tient la main de la suivante, et tu ne connais que la première.

Pour trouver la personne n°7, laquelle des deux est la plus rapide ? Et pour insérer quelqu'un au milieu ?

Le concept

Le tableau

Un **tableau** range ses éléments dans des cases **contiguës** en mémoire, numérotées à partir de 0.

```
indice :   0    1    2    3    4
valeur : [ 12 ][ 7 ][ 25 ][ 3 ][ 18 ]
```

- **Accès par indice** : immédiat. L'ordinateur calcule l'adresse directement.
- **Insertion/suppression au milieu** : il faut décaler tous les éléments suivants.
- **Ajout à la fin** : rapide en général.

La `list` de Python est un tableau dynamique (elle s'agrandit quand il le faut).

Le concept

La liste chaînée

Une **liste chaînée** est une suite de **maillons**. Chaque maillon contient une valeur et une **référence** vers le suivant.

```
tête → [12 | •]→ [7 | •]→ [25 | •]→ [3 | None]
```

- **Accès au n-ième** : il faut suivre les maillons un par un.
- **Insertion/suppression** (si on est déjà au bon endroit) : on modifie juste des références, sans décaler.
- **Mémoire** : chaque maillon coûte un peu plus (valeur + référence).

Le concept

La liste chaînée (suite)

```
class Maillon:
    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

class ListeChainee:
    def __init__(self):
        self.tete = None

    def ajouter_debut(self, valeur):
        self.tete = Maillon(valeur, self.tete)

    def vers_liste(self):
        resultat, courant = [], self.tete
        while courant is not None:
            resultat.append(courant.valeur)
            courant = courant.suivant
        return resultat
```

Le concept

Comparaison

Opération	Tableau	Liste chaînée
Lire le n-ième	rapide	lent (il faut parcourir)
Ajouter au début	lent (décalage)	rapide
Ajouter à la fin	rapide	rapide si on garde la fin, sinon lent
Insérer au milieu	lent (décalage)	rapide une fois la position atteinte
Mémoire	compacte	plus gourmande

🧠 ***Astuce*** *: en pratique, un tableau est le bon choix par défaut. La liste chaînée est utile pour ses insertions/suppressions locales rapides.*

Exemple pas à pas

Problème : insérer un membre prioritaire en 2e position dans `["Ada", "Bob", "Cléo"]`.

Avec un tableau (liste Python) : `liste.insert(1, "Prio")`. Les éléments suivants sont décalés d'une case.
Avec une liste chaînée : on se place sur le maillon de "Ada", puis :

```
class Maillon:
    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

def inserer_apres(maillon, valeur):
    maillon.suivant = Maillon(valeur, maillon.suivant)

def vers_liste(tete):
    r, c = [], tete
    while c is not None:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        r.append(c.valeur)
        c = c.suivant
    return r

tete = Maillon("Ada", Maillon("Bob", Maillon("Cléo")))
inserer_apres(tete, "Prio")
print(vers_liste(tete))   # ['Ada', 'Prio', 'Bob', 'Cléo']
```

Étapes : (1) créer le nouveau maillon qui pointe vers "Bob", (2) faire pointer "Ada" vers le nouveau maillon. **Aucun décalage.** L'ordre des deux étapes compte : si on change d'abord "Ada", on perd "Bob".

À toi, avec aide

Complète la méthode `ajouter_fin` d'une liste chaînée (avec `tete`). Si la liste est vide, le nouveau maillon devient la tête. Sinon, on parcourt jusqu'au dernier maillon (celui dont `suivant` est `None`).
Teste : ajouter 1, 2, 3 donne `[1, 2, 3]`.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Cas particulier : `self.tete is None`.

Indice 2

Parcours avec `courant = self.tete` tant que `courant.suivant is not None`.

Indice 3

À la sortie de la boucle, `courant` est le dernier maillon : `courant.suivant = Maillon(valeur)`.

À toi, avec aide · réponse

Réponse

```
class Maillon:
    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

class ListeChainee:
    def __init__(self):
        self.tete = None

    def ajouter_fin(self, valeur):
        nouveau = Maillon(valeur)
        if self.tete is None:
            self.tete = nouveau
            return
        courant = self.tete
        while courant.suivant is not None:
            courant = courant.suivant
        courant.suivant = nouveau
```

À toi, avec aide · réponse

Réponse (suite)

```
    def vers_liste(self):
        r, c = [], self.tete
        while c is not None:
            r.append(c.valeur)
            c = c.suivant
        return r

l = ListeChainee()
for x in (1, 2, 3):
    l.ajouter_fin(x)
print(l.vers_liste())   # [1, 2, 3]
```

À toi, en solo

Ajoute à `ListeChainee` :

1. `supprimer(valeur)` : retire le premier maillon contenant la valeur (ne fait rien si absente) ;
2. `longueur()` ;
3. `inverser()` : retourne la liste sur place, en changeant les références.

Teste sur des listes vides, à un élément et à plusieurs éléments. Cas piège : supprimer la tête.

À toi, en solo · réponse

Pistes de correction

```
class Maillon:
    def __init__(self, valeur, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

class ListeChainee:
    def __init__(self):
        self.tete = None

    def ajouter_debut(self, v):
        self.tete = Maillon(v, self.tete)

    def vers_liste(self):
        r, c = [], self.tete
        while c is not None:
            r.append(c.valeur)
            c = c.suivant
        return r
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def supprimer(self, valeur):
        if self.tete is None:
            return
        if self.tete.valeur == valeur:
            self.tete = self.tete.suivant
            return
        c = self.tete
        while c.suivant is not None and c.suivant.valeur != valeur:
            c = c.suivant
        if c.suivant is not None:
            c.suivant = c.suivant.suivant

    def longueur(self):
        n, c = 0, self.tete
        while c is not None:
            n += 1
            c = c.suivant
        return n
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def inverser(self):
        precedent, courant = None, self.tete
        while courant is not None:
            suivant = courant.suivant
            courant.suivant = precedent
            precedent, courant = courant, suivant
        self.tete = precedent

l = ListeChainee()
for x in (3, 2, 1):
    l.ajouter_debut(x)
l.supprimer(1)
l.inverser()
print(l.vers_liste(), l.longueur())   # [3, 2] 2
```

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Perdre le reste de la chaîne	On écrase une référence avant de la sauvegarder	Sauvegarder `suivant` avant de modifier
Oublier la tête vide	`AttributeError` sur `None`	Traiter le cas liste vide en premier
Utiliser une liste chaînée pour lire par indice	Chaque lecture parcourt la chaîne	Choisir un tableau si on lit par position
Confondre indice et valeur	Mauvais élément modifié	Nommer clairement `i` et `valeur`

Mini-boss

Mini-boss

Compare **concrètement** tableau et liste chaînée.

1. Implémente `ListeChainee` avec `ajouter_debut`, `ajouter_fin`, `lire(i)`, `supprimer_debut`.
2. Mesure avec `time.perf_counter()` : (a) 20 000 insertions au début, (b) 20 000 lectures de la position 10 000, pour une `list` Python et pour ta liste chaînée.
3. Remplis un tableau des temps.
4. Conclus : pour quel usage chaque structure gagne-t-elle ?

Mini-boss · réponse

Pistes de correction

Les insertions au début : `list.insert(0, x)` est lente (décalage de tout), la liste chaînée est rapide. Les lectures par indice : la `list` est immédiate, la liste chaînée est lente (parcours). Conclusion attendue : tableau pour lire par position et parcourir ; liste chaînée pour insertions/suppressions fréquentes en tête.

Quiz éclair

Quiz éclair

1. Dans un tableau, lire l'élément d'indice 5 est :

A. proportionnel à la taille
B. impossible
C. immédiat

2. Dans une liste chaînée, atteindre le 5e élément :

A. demande de suivre les maillons
B. est immédiat
C. est impossible

3. Insérer au début d'un grand tableau est :

A. gratuit

Quiz éclair

Quiz éclair (suite)

B. coûteux (décalage)
C. impossible

4. 🔁 *Révision mission 01* : Un ADT décrit :

A. son code interne
B. sa couleur
C. ses opérations

5. 🔁 *Révision HEF-010, mission 03* : Une précondition :

A. ce que l'appelant doit fournir
B. la valeur de retour
C. un test

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Dans quelles situations de ta vie préfères-tu un accès direct, et dans lesquelles une chaîne ?
2. Quel est le piège le plus facile à oublier dans les listes chaînées ?

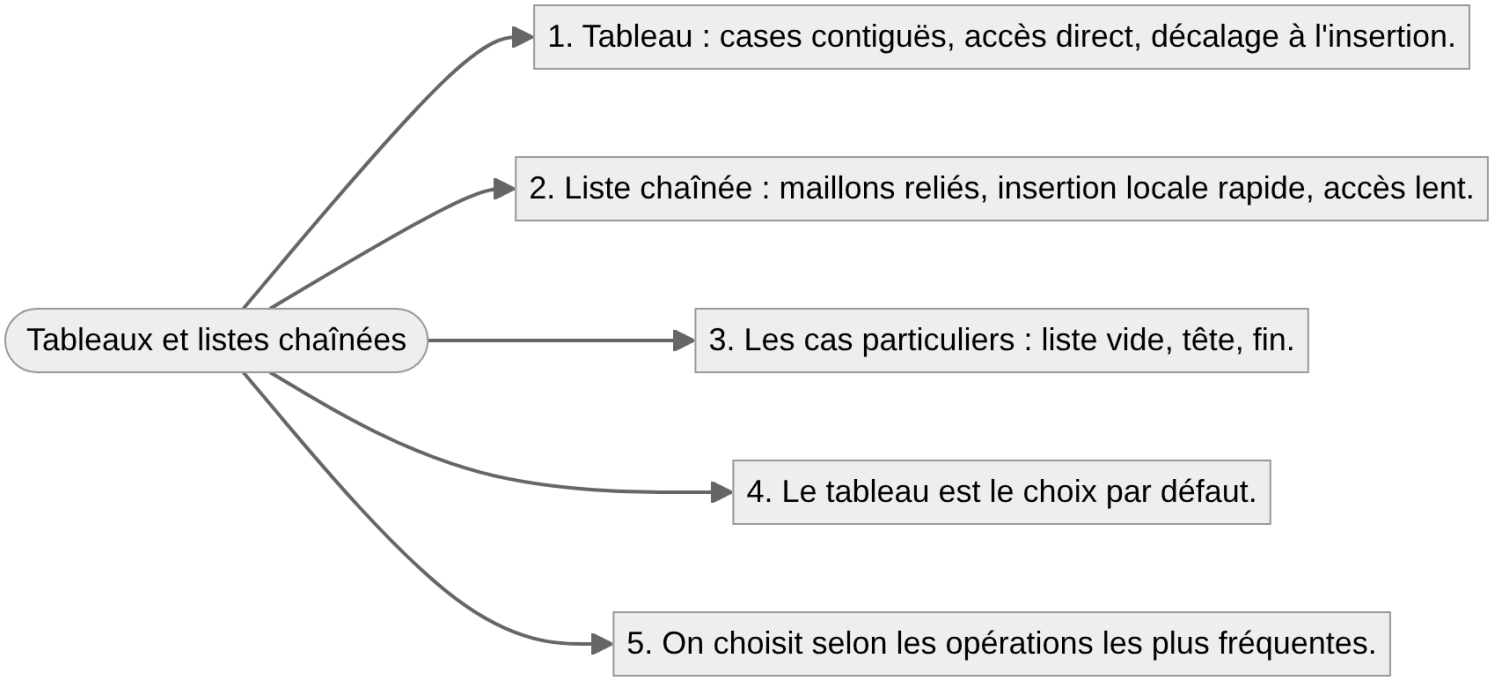
À retenir

À retenir

1. Tableau : cases contiguës, accès direct, décalage à l'insertion.
2. Liste chaînée : maillons reliés, insertion locale rapide, accès lent.
3. Les cas particuliers : liste vide, tête, fin.
4. Le tableau est le choix par défaut.
5. On choisit selon les opérations les plus fréquentes.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maillon fort

Prochaine mission : Piles et files

