Chap. 5 — Listes chaînées

1 — Introduction

1.1 — Méthodes de base sur les tableaux

Les tableaux en Python possèdent 2 méthodes essentielles extrêmement efficaces : `pop` et `append`.

`append()`

La méthode `append` ajoute un élément à la fin d'un tableau

[ ]:
```
nb_premiers = ['un', 'trois', 'cinq', 'sept', 'onze']

# ajoute le nombre 13 dans le tableau
nb_premiers.append('treize')
```

Rien ne s'affiche ? C'est normal car `append` est une **méthode** qui agit sur l'objet `nb_premiers`. L'objet a été modifié mais *on ne le voit pas*.

[ ]:
```
# Affiche le contenu de la variable
# nb_premiers

...
```

Comme tu peux le voir, le tableau a bien été modifié !

Dans le bloc suivant, implémenter la fonction `pair(n: int) -> list` qui renvoie un tableau contenant les `n` premiers nombres pairs en commençant par 0

Exemple 1 : `print(pair(5))` devrait afficher `[0, 2, 4, 6, 8]`.

Exemple 2 : `tab_pairs = pair(10)` ne devrait rien afficher mais seulement affecter à `tab_pairs` le tableau des 10 premiers nombres premiers.

[ ]:
```
def pair(n):
    ...


print(pair(5))
```

`pop()`

La méthode `pop` fait deux choses :

- supprimer le dernier élément du tableau
- renvoyer l'élément supprimé

[ ]:
```
nb_premiers.pop()
nb_premiers.pop()
nb_premiers.pop()
```

Normalement, le bloc précédent affiche `'sept'`.

En effet,

- le premier appel à la méthode `pop` supprime le dernier élément (`treize`) et le renvoie. Il n'y a pas d'affectation donc cette information se perd.
- le deuxième appel à la méthode `pop` supprime le dernier élément (`onze`) et le renvoie. Il n'y a pas d'affectation donc cette information se perd.
- le troisième appel à la méthode `pop` supprime le dernier élément (`sept`) et le renvoie. Il n'y a pas d'affectation donc cette information se perd. **Mais** comme on travaille dans un notebook, la dernière ligne de code est affichée si elle renvoie quelque chose (ce qui est le cas ici).

Si on en a besoin, on peut affecter le dernier élément supprimé à une variable pour le réutiliser plus tard.

[ ]:
```
dernier = nb_premiers.pop()

texte = dernier * 5

print(texte)
```

Implémenter la fonction `vide(tab: list) -> None` qui vide un tableau élément par élément en commençant par la fin et en affichant à chaque fois l'élément supprimé.

Exemple 1 : `vide([2, 4, 6])` doit afficher `6`, puis `4` puis `2`.

Exemple 2 : le code suivant

```
annee = [1998, 2003, 2004, 2008, 2021]
vide(annee)
print(annee)
```

doit afficher `1998`, puis `2003`, puis `2004`, puis `2008` puis `2021` puis `[]`. Le dernier affichage s'explique car le tableau `annee` est vide à la fin.

[ ]:
```
def vide(tab):
    ...


annee = [1998, 2003, 2004, 2008, 2021]
vide(annee)
print(annee)
```

1.2 — Ajouter un élément au début d'un tableau

Les tableaux de Python permettent par exemple d'insérer ou de supprimer efficacement des éléments à la fin d'un tableau, avec les opérations `append` et `pop`, mais se prêtent mal à l'insertion ou la suppression d'un élément à une autre position.

En effet, les éléments d'un tableau étant contigus et ordonnés en mémoire, insérer un élément dans une séquence demande de déplacer tous les éléments qui le suivent pour lui laisser une place.

Si par exemple on veut insérer une valeur v à la pemière position d'un tableau

1	1	2	3	5	8	13

il faut d'une façon ou d'une autre construire le nouveau tableau

v	1	1	2	3	5	8	13

Cette opération est cependant très coûteuse, car elle déplace tous les éléments du tableau d'une case vers la droite après avoir agrandi le tableau.

En effet, avec une telle opération :

1. On commence donc par agrandir le tableau, en ajoutant un nouvel élément à la fin avec `append`.

1	1	2	3	5	8	13	None

2. Puis on décale tous les éléments d'une case vers la droite, en prenant soin de commencer par le dernier et de terminer par le premier.

1	1	1	2	3	5	8	13

3. Enfin, on écrit la valeur v dans la première case du tableau.

v	1	1	2	3	5	8	13

Utiliser la description de l'algorithme en 3 étapes ci-dessus pour implémenter la fonction `entete(tab: list, val: int) -> None` **qui insère en début de tableau le nombre entier** `val`**.**

Exemple 1 : Le code suivant

```
tab_impairs = [3, 5, 7, 9]
entete(tab_impairs, 1)
print(tab_impairs)
```

doit afficher `[1, 3, 5, 7, 9]`

Exemple 2 : L'instruction `print(entete( pair(4), 100 ))` *doit afficher* `[100, 0, 2, 4, 6]`.

[ ]:
```
def entete(tab, val):
    ...


tab_impairs = [3, 5, 7, 9]
entete(tab_impairs, 1)
print(tab_impairs)
```

En Python, la méthode `insert(index:int, val)` est équivalente à `entete` si on définit `index` à `0`.

Ainsi, `tab.insert(0, 42)` est équivalent à `entete(tab, 42)`.

Mais comme on l'a dit plus haut, que l'on utilise `entete` ou `insert` on a réalisé au total un nombre d'opérations proportionnel à la taille du tableau. Si par exemple le tableau contient un million d'éléments, on fera un million d'opérations pour ajouter un premier élément. En outre, supprimer le premier élément serait tout aussi coûteux, pour les mêmes raisons.

Dans ce chapitre nous étudions une structure de données, la **liste chaînée**, qui d'une part apporte une meilleure solution au problème de l'insertion et de la suppression au début d'une séquence d'éléments, et d'autre part servira de brique de base à plusieurs autres structures dans les prochains chapitres.

Implémenter à l'aide de la méthode `pop` une fonction `supprime(tab: list)` qui supprime le premier élément du tableau `tab`.

Exemple 1 : Le code suivant

```
tab_impairs = [3, 5, 7, 9]
supprime(tab_impairs)
print(tab_impairs)
```

doit afficher `[5, 7, 9]`

Exemple 2 : L'instruction `print(supprime( pair(4) )` doit afficher `[2, 4, 6]`.

[ ]:
```
def supprime(tab):
    ...
```

```
tab_impairs = [3, 5, 7, 9]
supprime(tab_impairs)
print(tab_impairs)
```