Création et manipulation d'une Pile

Exercice 1 : Construction progressive de la classe `Pile`

Créez un fichier `pile.py` et suivez les étapes **dans l'ordre**. À chaque étape, testez dans le shell (Thonny/REPL) avec de petits exemples.

1. **Boîte vide**
   - Créer une classe `Pile` avec un constructeur `__init__(self)` qui initialise l'état interne (liste Python).
2. **État**
   - Écrire une méthode `est_vide(self) -> bool` qui renvoie `True` si la pile est vide, `False` sinon.
3. **Empilement**
   - Écrire une méthode `empiler(self, x)` qui ajoute l'élément `x` **au sommet** de la pile.
4. **Dépilement**
   - Écrire une méthode `depiler(self)` qui retire et **renvoie** l'élément au sommet. On genère une erreur si la pile est **vide**.
5. **Inspection**
   - Écrire une méthode `sommet(self)` qui renvoie l'élément au sommet **sans** le dépiler.
6. **Affichage**
   - Écrire une méthode `affiche_pile(self)` qui **affiche** la pile du bas vers le haut, avec le **sommet clairement indiqué**.
   - Exemple d'affichage possible :

```
-------
|     |   <- sommet
|  9  |
|     |
-------
|     |
|  5  |
|     |
-------
|     |
|  2  |
|     |
-------   <- bas
```

8. **Représentation texte (optionnel mais recommandé)**
    - Définir `__repr__` ou `__str__` pour un affichage compact (ex. : `Pile([2, 5, 9], sommet=9)`).

Petits tests (dans le shell) :

```
p = Pile()
print(p.est_vide())
p.empiler(2); p.empiler(5); p.empiler(9)
p.affiche_pile()
print("sommet:", p.sommet())
print("taille:", p.taille())
print(p.est_vide())
```

Exercice 2 : Méthodes internes avec contraintes

Dans **toutes** les questions suivantes, vous n'utiliserez **que** les méthodes `empiler`, `depiler` et `est_vide` (et la syntaxe Python de base). Les méthodes sont **attendues dans la classe** `Pile`.

1. `empiler_liste(self, lst)` Empile en **ordre** tous les éléments d'une liste `lst` (définissez clairement si `lst[0]` se retrouve **au bas** ou **près du sommet** après l'opération).
2. `vider_pile(self)` Vide **entièrement** la pile (ne renvoie rien). Quel enchaînement de `depiler` garantit l'invariant ?
3. `taille(self)` Réécrire/adapter `taille` **sans** accéder à la structure interne. *Indice :* il faudra **vider** la pile puis **la reconstruire** à l'identique **sans perdre** ses éléments (utiliser une pile auxiliaire).
4. `inverser_pile(self)` Inverse **en place** l'ordre des éléments de la pile, en utilisant **uniquement** les opérations autorisées. *Indice :* utilisez une ou deux piles auxiliaires.
5. `dupliquer_sommet(self)` (bonus) Duplique l'élément du sommet (après l'appel, le sommet apparaît deux fois de suite). Gérer le cas de la pile vide.

Exercice 3 : Applications algorithmiques avec `Pile`

Ici, vous écrirez des **fonctions** dans un fichier `applications_pile.py` qui **utilisent** la classe `Pile` (importée depuis `pile.py`).

1. **Parenthèses bien formées** Écrire une fonction `bien_parenthese(s: str) -> bool` qui vérifie l'équilibrage des parenthèses `()`, `[]`, `{}`. *Idées :* empiler les ouvrantes ; à chaque fermante, vérifier la correspondance avec le sommet ; à la fin, la pile doit être vide.
2. **Conversion décimal → binaire** Écrire `en_binaire(n: int) -> str` qui renvoie la représentation binaire de `n` en **empilant les restes** de la division par 2 puis en **dépilant** pour construire le résultat. *Exemples attendus :* `en_binaire(13) == "1101"`, `en_binaire(0) == "0"`.

3. **Palindrome** Écrire `est_palindrome(s: str) -> bool` qui utilise une pile pour comparer la chaîne à son renversé.

4. **Évaluation RPN (bonus)** Écrire `eval_rpn(tokens: list[str]) -> int | float` pour évaluer une expression en **notation polonaise inverse**. *Exemple :* `eval_rpn(["2","1","+","3","*"]) == 9`.

Exercice 4 : Vérification de parenthésage d'une expression

On souhaite écrire un programme qui vérifie si une expression mathématique est correctement parenthésée en utilisant une pile.

Principe :

Chaque fois que vous lisez un caractère ouvrant (**(, [, {**), vous l'empilez.

Chaque fois que vous lisez un caractère fermant (**),], }**), vous vérifiez qu'il correspond bien au sommet de la pile.

Si la pile est vide ou si la correspondance est incorrecte, l'expression est mal parenthésée.

À la fin, l'expression est correcte seulement si la pile est vide.

À implémenter :

Écrire une fonction est_bien_parenthesee(expr: str) -> bool qui applique ce principe.

La fonction doit gérer différents types de parenthèses et ignorer les autres caractères (chiffres, opérateurs, lettres).

Exemples attendus :

```
"(a+b) * [c-d]" → True

"(a+b]" → False (mélange de types)

"((a+b)" → False (pile non vide à la fin)

"a+b)" → False (fermeture sans ouverture)
```

Extension possible :

Gérer aussi les guillemets " " ou ' ' pour vérifier l'équilibrage des chaînes de caractères.

Compter la position exacte de l'erreur pour donner un message plus précis.