> Version pdf

# Type abstrait (Bac 🎯)

## Type abstrait

**"Point de cours 1"**

Vous avez manipulé en Première des types de **données structurées** en Python, comme les *tableaux/listes* ou les *dictionnaires*. Chacun de ces types de données propose un ensemble d'*opérations* permettant de manipuler les données qui constitue son **interface**. L'utilisateur n'a pas besoin de connaître **l'implémentation** des données et des opérations pour les manipuler.

**"Exemple 1"**

Le type `list` en Python permet d'organiser les données dans une séquence de cases mémoires contigues, appelée *tableau*. Chaque élément est accessible directement par son indice, ce qui n'est pas le cas pour le type abstrait *liste* où il faut d'abord parcourir tous les éléments précédents. Il est donc plus correct de qualifier cette structure de *tableau* que de *liste*. Néanmoins, la taille de ce tableau peut être redimensionnée, alors que dans le type abstrait *tableau* la taille est fixée, il est donc plus correct de qualifier cette structure de *tableau dynamique* que de *tableau*.

```
>>> t1 = []  # tableau vide
>>> t1.append(4)  # méthode append de tableau dynamique
>>> t1[0] = 5 # opérateur crochet équivalent à t1.__setitem__(0, 5)
>>> t1[0] # opérateur crochet équivalent à t1.__getitem__(0)
5
```

Plus généralement, un **type abstrait** de données est une structure qui offre une une **interface** publique de manipulation des données qui est constituée d'**opérations**. **L'implémentation** des données et des opérations peut être dissimulée à l'utilisateur. C'est le principe d'*encapsulation*.

### "Type abstrait tableau"

Le *type abstrait tableau statique* permet de stocker un ensemble de données dans un nombre fixée de cases mémoires contigues en mémoire. Contrairement aux tableaux dynamiques du type `list` de Python, les tableaux statiques ne sont pas redimensionnables et ne peuvent donc stocker qu'un nombre maximal de données.

Chaque case peut être accessible directement en lecture ou en écriture, comme les cases de la mémoire RAM dans l'architecture de Von Neumann.

Voici une interface minimale :

| Opération | Signature | Description |
|---|---|---|
| creer_tableau | creer_tableau(taille) | Renvoie un tableau de taille fixée |
| lire_case | lire_case(tableau, index) | Accès direct en lecture à la valeur de la case du tableau d'index fixé |
| modifier_case | modifier_case(tableau, index, valeur) | Accès direct en écriture à la valeur de la case du tableau d'index fixé |

### "Type abstrait dictionnaire"

Un tableau statique ou dynamique permet un *accès direct* aux données mais par le biais d'un index entier. Le *type abstrait dictionnaire* permet également un accès direct mais par le biais d'une *clef* qui n'est pas forcément un entier.

Voici une interface minimale :

| Opération | Signature | Description |
|---|---|---|
| creer_dico | creer_dico() | Renvoie un dictionnaire |
| valeur | valeur(dico, clef) | Accès direct en lecture à la valeur associée à la clef fixée |
| ajouter | ajouter(dico, clef, valeur) | Accès direct en écriture à la valeur associée à la clef fixée |

### "Point de cours 2"

On a vu dans les exercices 1 et 2 qu'il peut exister différentes implémentations d'une même interface de **type abstrait de données**.
Les performances liées aux *complexités temporelle et spatiale* peuvent différer selon les implémentations.

#### "Exemple 2"

Par exemple l'accès à la valeur associée à une clef est de coût constant pour l'implémentation du *type abstrait dictionnaire* avec le type `dict` de Python mais dans le pire des cas, le coût peut être égal au nombre de clefs dans le dictionnaire si on stocke les couples `(clef, valeur)` dans un tableau comme dans la question 2 de l'exercice 2.