Chapitre 6 : Structures abstraites 1
Listes, piles, files et dictionnaires

L'an dernier, nous avons étudié des types de données simples comme les booléens ou les entiers. Nous avons également découvert **comment implémenter** de nombreuses structures de données en langage Python : nous avons ainsi utilisé des tableaux et des dictionnaires. Toutefois, à l'inverse des booléens ou des entiers, ces structures ne sont pas natives en Python : ces types sont dits "construits".

En théorie informatique, il convient d'étudier le **modèle abstrait** (aussi appelé **interface**) ayant permis d'élaborer ces types dans divers langages, comme Python. Cela permet de comprendre les limitations des types proposés par Python et nous autorise également à implémenter nos propres structures.

Définition : En informatique, un **type abstrait** est une description mathématique d'un ensemble de données et de l'ensemble des opérations qu'on peut effectuer sur elles. C'est un cahier des charges qu'une structure de données doit respecter lors de sa mise en oeuvre.

Exemple :
Python 2 : x in range(100000) 556 ms
Python 3 : x in range(100000) 1.19 µs

L'interface est la même mais l'implémentation de __contains__ a été changée !

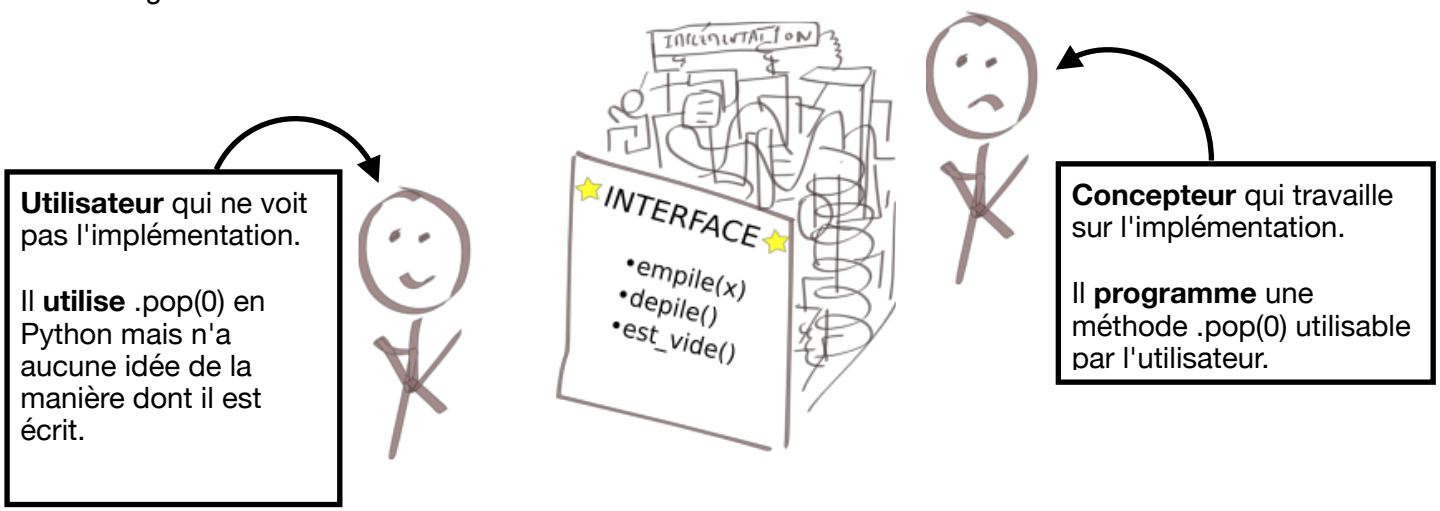
I. Préambule :

1) Différence interface / implémentation

Les structures que nous allons étudier peuvent s'envisager sous deux aspects :
- le côté utilisateur, qui utilisera une **interface** pour manipuler les données.
- le côté concepteur, qui aura choisi une **implémentation** pour construire la structure de données.

L'implémentation va désigner tous les **mécanismes techniques** qui sont mis en œuvre pour que les programmes fonctionnent et (si possible) soient les plus efficaces possibles.

Nous avons déjà abordé ces deux aspects lors de la découverte de la Programmation Orientée Objet. Le principe d'encapsulation fait que l'utilisateur n'a qu'à connaître l'existence des méthodes disponibles, et non pas le contenu technique de celle-ci. Cela permet notamment de modifier le contenu technique (l'implémentation) sans que les habitudes de l'utilisateur (l'interface) ne soient changées.



2) Structure adaptée en fonction des données

En informatique comme dans la vie courante, il est conseillé d'adapter sa manière de stocker et de traiter des données en fonction de la nature de celles-ci :

- le serveur d'un café, chargé de transporter les boissons du comptoir aux tables des clients, n'utilisera pas un sac en plastique pour faire le transport : il préfèrera un plateau.
- pour stocker des chaussettes, on préfèrera les entasser dans un tiroir (après les avoir appairées), plutôt que de les suspendre à des cintres.

De même en informatique, pour chaque type de données et pour chaque utilisation prévue, une structure particulière de données se révélera plus adaptée qu'une autre.

Intéressons nous par exemple aux données linéaires :

- Si ces données qui ne comportent pas de hiérarchie : toutes les données sont de la même nature et ont le même rôle. Par exemple, un relevé mensuel de températures, la liste des élèves d'une classe, un historique d'opérations bancaires…

  Ces données sont «plates» et n'ont pas de sous-domaines : la structure de **liste** paraît parfaitement adaptée.

- Lorsque les données d'une liste sont plutôt des couples (ex : liste de noms/numéros de téléphone), alors la structure la plus adaptée est sans doute celle du dictionnaire.

Les listes et les dictionnaires sont donc des exemples de structures de données linéaires.

D'autres exemples sur des données non-linéaires (abordées Chapitre 8 et 10) :

- *Le disque dur de votre ordinateur est subdivisé en sous-dossiers "bin, src, lib, dev…". Chaque de ces sous-dossiers est aussi subdivisé en sous-dossiers "user, lib, exe…". Une arborescence sera plus adaptée pour représenter cette situation. Les **arbres** seront traitées au chapitre 8.*

- *Enfin, si nos données à étudier sont les relations sur les réseaux sociaux des élèves d'une classe, alors la structure de graphe s'imposera d'elle-même. Les **graphes** seront étudiés au chapitre 10.*

II. Les listes

1) Position du problème

Avant d'aborder les listes, il est important de comprendre quelle est la principale limitation de la structure de tableau en Python. À première vue, cette structure semble efficace : on peut rajouter (resp. supprimer) un élément à la fin du tableau avec `append` (resp. `pop`).

Avec l'instruction `insert`, on peut même rajouter un élément où l'on veut au sein du tableau. Toutefois, cette dernière instruction est terriblement inefficace. En Python, les différents éléments d'un tableau sont contigus et ordonnés en mémoire. Ainsi, la liste

```
lst = [10, 6, 3, 5, 12]
```