Jusque maintenant, on a utilisé des variables pour stocker les données dans nos programmes python. Cette méthode, suffisante dans certains problèmes n'est cependant, et à minima, mal adaptée lorsque l'on doit traiter un grand nombre de données ou bien que le nombre de ces données n'est pas connu à l'avance.

Pour pouvoir traiter un grand nombre de données dans un seul programme, les tableaux sont une solution efficace.

1 Problème : la pyramide des âges

Considérons un programme qui stocke la pyramide des âges d'une population et permet à un utilisateur de choisir une tranche d'âge puis qui l'affiche. Avec les outils que l'on a vu jusque maintenant, ce prgramme ressemblerait à ceci :

```
age0 = 691165 #nombre de personnes entre 0 et 1 an
age1 = 710534 #nombre de personnes entre 0 et 1 an
age2 = 728579 #nombre de personnes entre 0 et 1 an
...

a = int(input("Age ? "))
if a == 0:
    print(age0)
elif a==1:
    print(age1)
elif a==2:
    print(age2)
...
else:
    print(0)
```

Un tel programme n'est pas vraiment raisonnable à écrire car pour une tâche relativement simple, le nombre de lignes de code est bien trop important. De plus, si on voulait écrire un programme similaire affichant une donnée parmi des milliers ou plus, la tâche deviendrait humainement impossible.

2 Notion de tableau

Les tableaux permettent de regrouper plusieurs valeurs dans un seul objet informatique. Une première manière en Python de créer un tableau est la suivante :

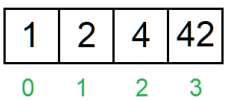
```
t = [1,2,4,42]
```

Dans cet exemple, `t` est un tableau contenant les valeurs 1, 2, 4 et 42.

Création de tableau (méthode 1) : Pour créer un tableau, on écrit entre crochet les valeurs qu'il doit contenir, séparées par des virgules.

Attention : Dans un tableau, les données sont rangées dans un certain ordre. Ainsi le tableau `t1 = [1,2,3]` n'est pas le même tableau que `t2 = [1,3,2]`.

Quand un tableau a été créé, il est alors possible d'accéder ou de modifier ses différentes valeurs à partir de leurs positions dans celui-ci, les positions commençant toujours à 0. Par exemple, pour accéder à la valeur 42 de `t`, il suffit d'utiliser la syntaxe `t[3]`.



On peut dès lors, par exemple :

— afficher 42 avec `print(t[3])` ;

— Modifier 42 en 12 avec `t[3] = 12`.

Accès aux valeurs d'un tableau : Pour accéder à une valeur d'un tableau `t` on utilise la syntaxe `t[i]` où `i` est la position (on parle plutôt d'**indice**) dans le tableau de la valeur considérée.

Modification des valeurs d'un tableau : Pour modifier une valeur d'un tableau `t` on utilise la syntaxe `t[i] = v` où `i` est l'indice dans le tableau de la valeur considérée et `v` la nouvelle valeur.

Remarques : Certaines autres structures stockant des collections d'objets utilisent la même syntaxe pour l'accès et la modification des éléments par exemple les chaînes de caractères, les tuples etc. Cependant, selon la structure, il n'est pas toujours possible de modifier les valeurs comme dans les tableaux. On dit que les tableaux sont des structures **mutables**.

> **Taille d'un tableau et indices :** La taille d'un tableau est le nombre de valeurs qu'il contient. Ainsi dans notre exemple le tableau `t` est de taille 4. On peut y accéder grâce à la fonction `len(t)` de Python.
>
> En particulier, un indice de tableau de taille `n` est toujours compris entre `0` et `n-1`. Lorsque ce n'est pas le cas, l'interpréteur Python renvoie une erreur.

Tester les commandes suivantes :

- `t=[1,2,4,42]`
- `print(len(t))`
- `print(t[0])`
- `t[2] = t[2]+1`
- `print(t[2])`
- `print(t[4])`
- `t[4] = 78`
- `print(t[-1])`
- `print(t[-2])`

Que ce passe-t-il si on utilise des indices négatifs ? Attention : ce fonctionnement est une spécificité de Pyhton et ne fonctionne généralement pas dans d'autres langages.

Retour sur la pyramide des âges : On peut maintenant écrire de manière plus concise notre programme sur la pyramide des âges :

```
pda = [691166, 710534, 728579, ...]

a = int(input("Age ? "))
print(pda[a])
```

3 Parcours d'un tableau

Pour la suite du TP et afin de tester nos programme, nous allons générer un tableau aléatoirement. Entrer le code suivant (la syntaxe n'est pour le moment pas à retenir).

```
from random import *
pda = [50000 + i*(100-i)*10 + randint(0,10000) for i in range(100)]
```

Toujours dans notre exemple de pyramide des âges, supposons que l'on veuille calculer et afficher le nombre de personnes ayant entre 10 et 20 ans. Une possibilité est d'écrire :

```
print(pda[10]+pda[11]+...+pda[19]})
```

Encore une fois, cette solution est longue à écrire et est difficilement modifiable par exemple pour afficher le nombre de personnes entre 7 et 77 ans.

Pour palier à ce problème on parcourt le tableau à l'aide d'une boucle `for` :

```
age_min = 10
age_max = 20
somme = 0
for i in range(age_min, age_max):
    somme = somme+pda[i]
print("Il y a", somme, "personnes entre", age_min, "et", age_max, "ans.")
```

Avec cette solution, on peut facilement modifier les valeurs de `age_min` et `age_max`, voire même les demander à l'utilisateur :

```
age_min = int(input("Age minimum ? "))
age_max = int(input("Age maximum ? "))
somme = 0
for i in range(age_min, age_max):
    somme = somme+pda[i]
print("Il y a", somme, "personnes entre", age_min, "et", age_max, "ans.")
```

Parcours d'un tableau : Pour effectuer une tâche sur chaque élément d'un tableau `t` entre les indices `i_min` (inclus) et `i_max` (exclus) on utilise une boucle `for` :

```
for i in range(indice_min, indice_max):
    #instructions a repeter dans lesquelles t[i] represente la valeur courante du tableau
    ...
```

En particulier, si on veut traiter toutes les données du tableau :

```
n = len(t)
for i in range(n):
    #instructions a repeter dans lesquelles t[i] represente la valeur courante du tableau
    ...
```

4 Construire de grands tableaux

Quand on veut travailler avec des tableaux de grande taille la première méthode `t = [1,2,4,42]` ne fonctionne plus car cela implique de rentrer toutes les valeurs à la main. Une autre méthode (on en verra encore d'autre plus tard) consiste à créer un tableau de la taille voulue puis de le remplir ensuite via un parcours (les données pouvant être calculées ou récupérées dans un fichier externe).

Création de tableau (méthode 2) : Pour créer un tableau `t` de taille `n` ne contenant que des `0` (`0` pouvant être remplacé par toute autre valeur), on utilise la syntaxe suivante :

```
t = [0]*n
```

On peut ensuite modifier ces 0 en accédant à `t[i]` où `i` est compris entre `0` et `n-1`.

Tester le programme suivant :

```
t = [0]*100
for i in range(100):
    t[i] = i**2
print("taille du tableau :", len(t))
for i in range(100):
    print("Valeur d'indice", i, ":", t[i])
```

Remarques :

- Dans `t = [0]*100` l'utilisation de l'opérateur `*` ne fait pas référence à une multiplication mais à la la mise bout à bout (concaténation) 100 fois d'un tableau `[0]`.
- On peut d'ailleurs si on a deux tableaux `t1` et `t2` les concaténer avec l'opérateur `+` en écrivant par exemple `t = t1+t2`.

Tester le programme suivant et expliquer les affichages :

```
t1 = [1,2,3]
t2 = [10,11]
print(t1 + 3*t2)
print(3*t2 + t1)
```

5 Tableaux et variables

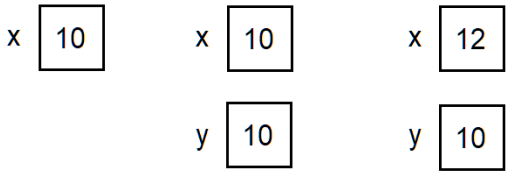
5.1 Tableaux et adresses mémoire

Étant donné la présentation faite des tableaux ici, on pourrait avoir l'impression que ceux-ci se comportent comme un ensemble de variables numérotées. C'est une bonne première idée mais cela ne suffit pas à expliquer certains comportements. Rentrons un peu dans les détails.

Tester le programme suivant :

```
x = 10
y = x
x = 12
print(x, y)
```

On peut représenter les états mémoire successifs de l'interpréteur par le schéma suivant :



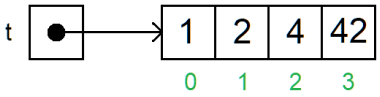
Tester le programme suivant :

```
t1 = [10]
t2 = t1
t1[0] = 12
print(t1, t2)
```

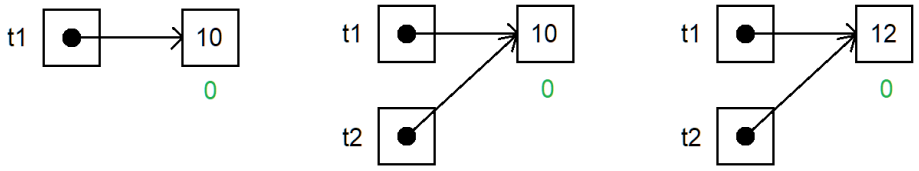
Cette fois, la modification de `t1` a un effet sur `t2`. Cela provient de la manière dont sont implémentés les tableaux :

> Un tableau `t` est **une variable contenant une adresse mémoire** à partir de laquelle se trouvent des **cases mémoires consécutives** stockant ses valeurs.

Ainsi le tableau `t = [1,2,4,42]` peut être représenté de la sorte :



On peut alors représenter les états mémoire successifs de l'interpréteur dans le programme précédent par le schéma suivant :



5.2 Impact sur les fonctions

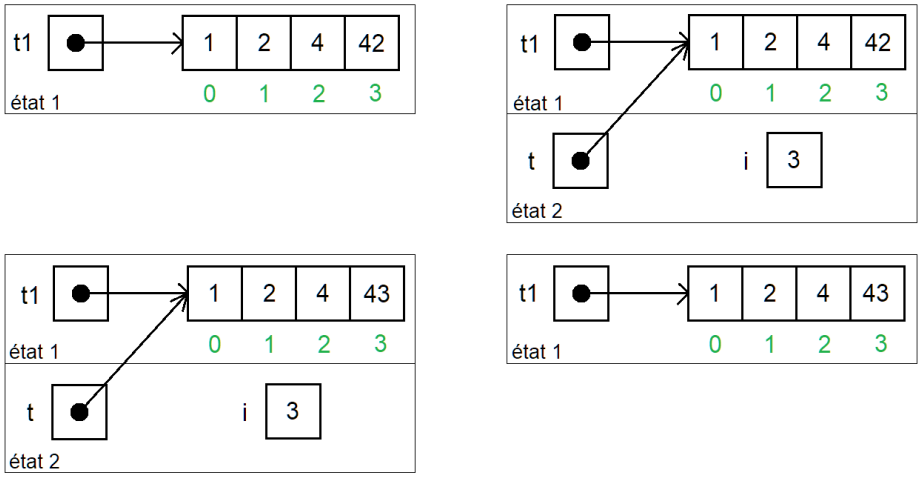
Cette implémentation a également un effet sur le comportement des fonctions. En effet on a vu dans dans le chapitre P4 (partie 4.2) qu'il est impossible de modifier une variable passée en argument d'une fonction par effet de bord. Ce n'est pas tout à fait vrai pour les tableaux. Prenons un exemple :

```
def incrementer(t,i):
    t[i] = t[i]+1

t1 = [1,2,4,42]
incrementer(t1,3)
print(t)
```

Contrairement à la fonction prise en exemple dans le chapitre P4, cette fonction modifie `t` pourtant passé en argument.

Représentons les états mémoires successifs (en se rappelant que l'appel de `incrementer(t1,3)` initialise des variables `t` et `i` aux valeurs `valeur de t` et `3` dans un nouvel état mémoire :



Puisque le tableau `t` de l'état 2 contient l'adresse mémoire des cases de `t1`, la modification des cases de `t` modifie également les cases de `t1` pourtant absent de cet état.