Cours 12 novembre — Résumé de la séance

1) Rappels méthodologie

Nous avons repris la méthode pour écrire une fonction. Si vous n'êtes pas très à l'aise en programmation, il faut passer par 3 étapes essentielles :

1) **sur une feuille de papier et en utilisant un crayon**, je travaille sur un exemple. Je me force à réaliser les mêmes étapes que celles que ferait une ordinateur. Pas de raccourcis ici.
2) j'écris un programme qui fonctionne sur l'exemple sur lequel j'ai travaillé. Je traduis toutes les étapes que j'ai identifié au 1) en langage Python.
3) je généralise mon programme en le transformant en fonction qui fonctionne dans n'importe quel cas. **Attention, je n'oublie pas le return, essentiel pour une fonction.**

2) Séance d'application

Nous avons donc repris les exercices 1 à 4 en passant beaucoup de temps sur la méthodologie.

Refaites les exercices en passant par l'utilisation de la méthode. Ne sautez pas l'étape papier/ crayon, essentielle pour passer à la programmation.

Exercice 1

1) Comment réaliser la somme d'une liste de nombre ?

Par exemple, si on veut ajouter les nombres : 7 , 12, 8, 13

Un humain va avoir tendance à trouver des groupements facilitant le calcul mental : ici, (7+ 13) = 20 puis 12+8 = 20 donc 20+20=40 CQFD.

Un ordinateur ne peut pas réaliser une telle opération : elle n'aurait d'ailleurs aucun intérêt car pour un ordinateur, c'est aussi facile de faire 7+12 que 12+8… **Un ordinateur a une limite très importante : il ne peut faire qu'une seule et unique chose à la fois.**

Ainsi, pour auditionner ces nombres, il va faire :

- On prend les 2 premiers nombres : 7, 12 et on les ajoute :
  - 7+12 = 19 à enregistrer dans une case mémoire appelée total.
- On prend le résultat précédent 19 (stocké dans total) et le nombre suivant de ma liste de nombre (8) et on les ajoute
  - 8+19 = 27 à enregistrer dans une case mémoire appelée total2.
- On prend le résultat précédent 27 (stocké dans total2) et le nombre suivant de ma liste de nombre (13) et on les ajoute
  - 27+13 = 40 à enregistrer dans une case mémoire appelée total3.

On pourrait écrire cela :

```
total = 7+12
total2 = total +8
total3 = total2 +13
```

La question que l'on peut se poser est : pourquoi toujours stocker dans une case mémoire différente ? Finalement, nous ne sommes pas intéressés par les résultats intermédiaires qui peuvent être remplacés. Donc, total3 et total2 sont inutiles. Le programme devient :

```
total = 7 + 12
total = total + 8
total = total + 13
```

Maintenant, on souhaite que ce programme soit adapté pour n'importe quel nombre. On va donc devoir remplacer les valeurs numériques 7, 12, 8, 13 par des valeurs issues du tableau tab : tab= [7, 12, 8, 13].
7 est la valeur n°0 du tableau tab
12 est la valeur n°1 du tableau tab
8 est la valeur n°2 du tableau tab
13 est la valeur n°3 du tableau tab

On réécrit :

```
total = tab[0] + tab[1]
total = total + tab[2]
total = total + tab[3]
```

Il faut maintenant réfléchir aux cas particuliers. Que se passe-t-il si j'ai un tableau avec un seul nombre tab = [2]. La première ligne va planter… On va donc faire commencer total à 0 et généraliser notre principe d'addition :

```
total = 0
total = total + tab[0]
total = total + tab[1]
total = total + tab[2]
total = total + tab[3]
```

Finalement, je m'aperçois que je suis en train de faire des copier/coller de la même ligne avec un indice (0,1,2,3) qui change… Cela est le signe d'une boucle for :

```
total = 0
for i in range(4):
    total = total + tab[i]
print(total)
```

Nous sommes parvenus à un programme Python qui fonctionne correctement. Transformons-le maintenant en fonction :

```
def somme(tab):
    total = 0
    nombreElements = len(tab)
    for i in range(nombreElements):
        total = total + tab[i]
    return total

print(somme([4,5,6,3,1]))
```

On a remplacé 4 par len(tab) qui me donne le nombre de nombres total présent dans mon tableau
On a aussi remplacé le print final par un return qui me renvoie une valeur hors de ma fonction.

Exercice 2 : la fonction echangeAttendu est la fonction attendue comme réponse. Toutefois, je vous propose une autre méthode qui est plus rapide et lisible. Attention, la sauvegarde de tab[i] est toujours effectuée même si elle n'apparait pas.

Exercice 3 :

1) La fonction verifieTaille demande de vérifier que les tailles de mes tableaux sont identiques. Cela signifie que s'ils ont même nombre d'éléments, je réponds True. Sinon, je réponds False.

C'est donc un if simple.

2) Le fonction hamming demande de regarder combien de différences sont présentes entre les valeurs de deux tableaux.

Exemple :
T1 = [2,3,4,5,6]
T2 = [2,5,4,5,8]

Algorithme 0:
Si T1 et T2 n'ont pas le même nombre d'éléments, on renvoie -1
Sinon,
    on va comparer un à un les éléments de T1 et de T2:
        si le n°0 de T1 est différent du n°0 de T2, on compte +1 différence
        si le n°1 de T1 est différent du n°1 de T2, on compte +1 différence
        si le n°2 de T1 est différent du n°2 de T2, on compte +1 différence
        etc.

Pour comparer un à un les éléments, il faut donc utiliser une boucle for. Notre constatation est renforcée par le fait que les indices font 0, 1, 2, …

Pour compter mon nombre de différence, je vais avoir besoin d'un compteur. Mon programme va donc être modifiée.

Algorithme 1:
Si T1 et T2 n'ont pas le même nombre d'éléments, on renvoie -1
Sinon,
    nbDifference = 0
    on va comparer un à un les éléments de T1 et de T2:
        si le n°0 de T1 est différent du n°0 de T2 :
            nombreDifference = nombreDifference + 1
        si le n°1 de T1 est différent du n°1 de T2 :
            nombreDifference = nombreDifference + 1
        si le n°2 de T1 est différent du n°2 de T2 :
            nombreDifference = nombreDifference + 1

Cela nous permet d'expliquer le programme écrit en PJ.