1 TD 2 : Tri minuté

(correction page ??)

Abordé lors de cette séance	
programmation	fonctions, listes
algorithme	tri bulle

L'objectif de ce TD est de programmer un tri bulle. Ce tri sera décomposé en plusieurs petites *fonctions*. La première demi-heure sera utilisée pour définir les fonctions.

Première demi-heure : fonctions .

Une fonction est un moyen d'isoler une partie de programme qu'on répète souvent. Pour calculer une moyenne pondérée sur deux liste de notes, sans fonction :

```
# première moyenne
poids = [1, 2, 1, 2]
notes = [12, 14, 8, 9]
s = 0                              #
w = 0                              #
for p,n in zip (poids, notes) :    # calcule du résultat
    s += p*n                       #
    w += p                         #
resultat = s * 1.0 / w    # * 1.0 évite le problème dus aux divisions entières
                          # 1/2 --> 0,   1 * 1.0 / 2 --> 0.5

# seconde moyenne
poids = [1, 1, 2]
notes = [12, 14, 8]
s = 0
w = 0
for p,n in zip (poids, notes) :
    s += p*n
    w += p
resultat = s * 1.0 / w
```

Le programme est plutôt redondant. On utilise une fonction pour éviter cela :

```
def moyenne_ponderee (poids, notes) :
    s = 0
    w = 0
    for p,n in zip (poids, notes) :
        s += p*n
        w += p
    return s * 1.0 / w        # le mot-clé return désigne le résultat que la fonction calcule

poids = [1, 2, 1, 2]
notes = [12, 14, 8, 9]
resultat = moyenne_ponderee (poids, notes)
```

```
poids = [1, 1, 2]
notes = [12, 14, 8]
resultat = moyenne_ponderee (poids, notes)
```

Le second programme est plus court. Quelqus termes :

- **déclaration** : on déclare une fonction avec le mot-clé `def`, il ne faut oublier de décaler les lignes qui suivent, c'est la seule façon d'indiquer que ces lignes font partie de la fonction et non du reste du programme.
- **paramètres** : `poids`, `notes`, ce sont les entrées de la fonction, ils apparaissent entre parenthèses lors de la déclaration
- **résultat** : c'est l'objectif de la fonction, calculer ce résultat indiqué par le mot-clé `return`.

1) Que se passe-t-il si les tableaux `poids` et `notes` n'ont pas la même longueur ?

2) La fonction suivante prend comme entrée un tableau et le retourne trié.

```
def trie_tableau (tableau) :
    tableau.sort ()
    return tableau

tab = [ 3,5,4]
tri = trie_tableau (tab)
print tab
print tri
```

Qu'affiche les deux dernières lignes ? Cela vous paraît-il étrange ?

3) Le langage *Python* ne sait pas faire grand chose mais il dispose de nombreux modules ou extensions. Chaque ligne `import ...` signifie que le programme va utiliser une extension du langage. Par exemple, on voudrait que la fonction `trie_tableau` retourne un tableau trié mais qui ne modifie pas le tableau envoyé à la fonction. A l'aide de la documentation, utilisez la fonction `copy` du module `copy`.

4) Que fait le programme suivant :

```
def trie_tableau (tableau) :
    tableau.sort ()
    return tableau

tab = ( 3, 5, 4 )
tri = trie_tableau (tab)
print tab
print tri
```

En langage *Python*, les types `None`, `int`, `float`, `str`, `tuple` sont passés par valeur à une fonction : les modifier à l'intérieur de la fonction n'a pas d'impact à l'extérieur. Les types `list` et `dict` sont passés par adresse : les modifier à l'intérieur de la fonction a un impact à l'extérieur.

Seconde demi-heure : la fonction `sort`

5) On considère la liste suivante :

```
l = [ 4, 5, 3, 7, 7, 9, 10, 0 ]
```

L'instruction suivante permet de trier la liste :

```
l.sort ()
```

L'instuction `print l` permet d'afficher le résultat et de vérifier que la liste est bien triée. Rechercher comment trier en sens inverse sur Internet via un moteur de recherche ou en utilisant l'aide *Python*(avec l'instruction `help`).

6) Que donne la même instruction `sort` sur l'exemple suivant :

```
l = [ 4, 5, "ee", 7, "aa", 7, 9, 10, 0, "gg" ]
```

Le résultat est-il intuitif ?

7) Exécuter les deux instructions suivantes :

```
print l [1]
print l [ len (l)-1 ]
```

Le premier résultat correspond au premier ou au second élément. Quel est l'indice du premier élément du tableau ? Le second résultat correspond à un autre élément tableau `l`. Quel est l'indice du dernier élément ?

8) Il existe de nombreux algorithmes de tri. On les distingue selon deux critères :

1. Leur coût ou le nombre de comparaisons nécessaire pour trier un tableau de taille n.
2. Le nombre de copies du tableau initial.

Dans le cas général, on suppose qu'on sait ordonner deux éléments et rien de plus. Pour un tableau de taille n, il est prouvé qu'il n'est pas possible d'effectuer moins de $n \ln n$ comparaisons. Nous allons voir un tri plus lent ($\frac{n^2}{2}$ comparaisons) mais plus simple : le **tri bulle**. On commence par la séquence logique suivante :

1. On commence à $i = 0$
2. On compare les éléments d'indice i et $i + 1$. Si le second est plus petit que le premier, on permute les deux nombres.
3. On se décale en suite d'un cran à $i + 1$. Si $i + 1 < n - 1$, on retourne à l'étape 2.

On commence par écrire une fonction qui permute les deux éléments `i` et `j` d'un tableau :

```
def permutation (l, i, j) :
    ...
```

Troisième demi-heure : tri bulle

On cherche à décomposer le problème en des fonctions aussi petites que possibles puis à les assembler par la suite pour former la séquence décrite ci-dessus.

9) La fonction suivante doit comparer deux éléments consécutifs et les permuter s'ils sont dans le mauvais ordre.

```
def comparaison_permutation (l, i) :
    ...
```

Il faudra utiliser la fonction `permutation` implémentée à la question précédente.

10) Petite pause : exécuter les lignes suivantes. Quel est le dernier entier affiché ?

```
for i in xrange (0, 10) :
    print i
```

11) Etape suivante : on appelle la fonction `comparaison_permutation` pour tous les entiers de i allant de 0 à `len(l)` $-$ `1`.

```
def plusieurs_comparaison_permutation (l) :
    ...
```

Après l'exécution de la fonction `plusieurs_comparaison_permutation`, peut-on considérer le tableau trié ? Pourquoi ?

Quatrième demi-heure : fin du tri bulle .

12) On écrit une troisième fonction pour appeler n fois la fonction `plusieurs_comparaison_permutation` :

```
def n_fois_plusieurs_comparaison_permutation (l) :
    ...
```

Peut-on considérez le tableau trié ? Pourquoi ?

13) Comptez le nombre de comparaisons de la précédente méthode ? Peut-on faire mieux et comment ?

14) Modifiez les fonctions précédentes pour diminuer le nombre de comparaisons ?

Pour aller plus loin ou pour ceux qui ont fini plus tôt .

L'algorithme de tri est maintenant fonctionnel. On veut néanmoins l'altérer.

15) Comment modifier simplement l'algorithme pour trier dans l'autre sens (sens décroissant) ?

16) Comment modifier simplement l'algorithme pour effectuer une permutation aléatoire de la liste

d'éléments ?

17) Comment modifier simplement l'algorithme pour placer les éléments pairs en premier et les éléments impairs en dernier ?

18) Au lieu de parcourir les éléments dans le sens des indices croissants, on les parcourt dans le sens des indices décroissants ? Que cela changerait-il ?

19) Et si on tirait au hasard les deux éléments consécitufs à comparer ? C'est-à-dire qu'au lieu d'utiliser des boucles de 0 à $n-1$, on tire au hasard un élément i entre 0 et $n-2$ pour comparer i et $i+1$ puis les inverser si nécessaire. Est-ce que cela marcherait ?

Remarques .

Un tutoriel vous permettra d'appréhender le tri par fusion qui effectue $n \ln n$ comparaisons.