

Séance 6 : LISTES, GESTION MÉMOIRE ET EXCEPTIONS

L1 – Université Côte d'Azur

Exercice 1 – Gestion des absences

Dans cet exercice, on considère des listes de notes. On considère qu'une note est soit un flottant entre 0 et 20, soit la chaîne de caractères "ABS".

1. Écrire une procédure `modifier_ABS(liste)` qui modifie la liste donnée en argument en remplaçant les "ABS" par des zéros. Écrire un test avec `assert`.
2. Écrivez une fonction `ABS_vers_zéro_basique(liste)` qui à partir d'une liste de note renvoie une nouvelle liste obtenue en remplaçant les "ABS" par zéro. La liste initiale ne devra pas être modifiée et on ne pourra pas utiliser la méthode `append`. Écrire un test
3. Écrivez une fonction `ABS_vers_zéro(liste)` ayant le même comportement que la fonction précédente, mais cette fois-ci en utilisant `append`.
4. Enfin écrivez une fonction `filtrer_notes(liste)` qui renvoie la liste obtenue à partir de l'argument en supprimant tous les "ABS". La liste renvoyée sera donc plus courte que la précédente.
5. Aurait-on pu écrire cette fonction sans utiliser `append`.

Exercice 2 – Comprendre les compréhensions

Pour chacune des constructions de liste par compréhension, donnez la valeur finale de la liste puis écrivez une boucle `for` construisant une telle liste. On pourra utiliser la méthode `append`.

1. `[x*x for x in range(4)]`
2. `[(i,10-i) for i in range(11)]`
3. `[str(i) for i in range(11) if i%2==0]`

Exercice 3 – Maximisation filtrante

On considère dans cet exercice des listes d'entiers.

1. Écrivez une fonction `max_pair(liste)` renvoyant la plus grande valeur paire de la liste. Si la liste n'en contient aucune, on soulèvera une `ValueError`.

```

1 >>> max_pair([9, 17, 8, 11, 4])
2 8
3 >>> max_pair([9, 7, -12, 11])
4 -12
5 >>> max_pair([9, 7, 11])
6 Traceback (most recent call last):
7   File "<console>", line 1, in <module>
8   File "<console>", line 12, in max_pair
9   ValueError: Pas de valeur paire
10 >>> max_pair([])
11 Traceback (most recent call last):
12   File "<console>", line 1, in <module>
13   File "<console>", line 12, in max_pair
14   ValueError: Pas de valeur paire

```

La difficulté réside dans l'initialisation du maximum. Il y a deux approches. La première consiste à créer une fonction auxiliaire qui autorise d'utiliser le maximum avec None, qui peut ainsi devenir la valeur initiale.

```

1 def mon_max(a,b):
2     if a == None:
3         return b
4     elif b == None:
5         return a
6     else:
7         return max(a,b)
8
9 def max_pair(liste):
10    maximum = None
11    for valeur in liste:
12        if valeur%2==0:
13            maximum = mon_max(maximum, valeur)
14    if maximum == None:
15        raise ValueError("Pas de valeur paire")
16    return maximum

```

L'autre approche consiste à ne pas définir de fonctions auxiliaires mais d'utiliser une variable supplémentaire indiquant si on a déjà trouvé un nombre pair.

```

1 def max_pair(liste):
2     déjà_trouvé = False
3     maximum = None
4     for valeur in liste:
5         if valeur%2==0:
6             if déjà_trouvé:
7                 maximum = max(maximum, valeur)
8             else:
9                 maximum = valeur
10                déjà_trouvé = True
11    if not déjà_trouvé:
12        raise ValueError("Pas de valeur paire")
13
14    return maximum

```

2. Écrivez une fonction `afficher_max_pair(liste)` qui, en utilisant la fonction précédente, affiche un message donnant le plus grand nombre pair s'il existe ou le premier quatrain du poème « Art poétique » si la liste ne contient que des impaires.

```

1 >>> afficher_max_pair([9, 17, 8, 11, 4])
2 Le plus grand pair est 8
3 >>> afficher_max_pair([9, 7, -21, 11]) # Art poétique (Verlaine)
4 De la musique avant toute chose,
5 Et pour cela préfère l'Impair
6 Plus vague et plus soluble dans l'air,
7 Sans rien en lui qui pèse ou qui pose.

```

```

1 def afficher_max_pair(liste):
2     try:
3         valeur = max_pair(liste)
4         print("Le plus grand pair est", valeur)
5     except:
6         print("De la musique avant toute chose,")
7         print("Et pour cela préfère l'Impair")
8         print("Plus vague et plus soluble dans l'air,")
9         print("Sans rien en lui qui pèse ou qui pose.")

```

Exercice 4 – Reconnaître une liste triée

1. Écrivez une fonction `est_triée(L)` qui prend en argument une liste d'entiers `L` et qui renvoie `True` si la liste est triée en ordre croissant. Par exemple, `est_triée([1,2,2])` renvoie `True` et `est_triée([1,5,2])` renvoie `False`.

```

1 def est_triée(L) :
2     for i in range(len(L)-1) :
3         if L[i] > L[i+1] :
4             return False
5     return True

```

2. Écrivez des tests avec `assert`.

```

1 assert est_triée([]) == True
2 assert est_triée([1]) == True
3 assert est_triée([1,3,7]) == True
4 assert est_triée([12, 10]) == False
5 assert est_triée([1,2,6,12,78,70]) == False

```

Exercice 5 – Compactage

Dans cet exercice, vous pouvez utiliser la méthode `append`.

1. Écrivez une fonction `grouper(L)` qui prend en argument une liste `L` et qui renvoie la liste obtenue en remplaçant toute suite d'éléments consécutifs `x, x, x, ..., x` par un seul élément `x`. Par exemple, `grouper([4,4,4,2,2,4])` renvoie `[4,2,4]`.

```

1 def grouper(L) :
2     if len(L) == 0 :
3         return []
4     e = L[0]
5     res = [e]
6     for i in range(1,len(L)) :
7         if L[i] != e :
8             e = L[i]
9             res.append(e)
10    return res

```

2. Écrivez une fonction `compacter(L)` qui prend en argument une liste `L` et qui renvoie la liste obtenue par groupage en indiquant de plus à l'aide d'un couple la taille de chaque groupe. Par exemple, `compacter([4,4,4,2,2,4])` renvoie `[(3,4), (2,2), (1,4)]`.

```

1 def compacter(L) :
2     if len(L)==0 :
3         return []
4     # L contient au moins un élément
5     e = L[0]
6     c = 1
7     res = []
8     for i in range(1,len(L)) :
9         if L[i] != e :
10            res.append((c,e))
11            e = L[i]
12            c = 1
13        else :
14            c = c+1
15    res.append((c,e))
16    return res

```

Tri par insertion

Le tri par insertion est un algorithme de tri qui insère un à un les éléments à trier dans une liste qui contient à la fin le résultat attendu. Par exemple, on aura

```
Étape 0 : triés : []          à trier : [5,8,7,1]
Étape 1 : triés : [5]        à trier : [8,7,1]
Étape 2 : triés : [5,8]     à trier : [7,1]
Étape 3 : triés : [5,7,8]   à trier : [1]
Étape 3 : triés : [1,5,7,8] à trier : []
```

Exercice 6 – Tri par insertion

1. Écrivez une fonction `index_insertion(L,n)` qui prend en arguments une liste triée d'entiers L et un entier n et qui renvoie l'indice de la position à laquelle insérer n dans L afin de garder la liste triée. Par exemple, `index_insertion([1,5,6,10],2)` renvoie 1 car 2 est à l'indice 1 dans la liste `[1,2,5,6,10]`. De plus, si n est déjà dans la liste, on renverra le plus grand indice qui convient. Par exemple, `index_insertion([1,2,6,10],2)` renvoie 2.

```
1 def index_insertion(L,n) :
2     for i in range(len(L)):
3         if L[i]>n :
4             return i
5     return len(L)
```

2. Écrivez une fonction `insertion_triée(L)` qui prend en argument une liste d'entiers L et qui renvoie une nouvelle liste contenant les entiers de L en ordre croissant. Vous pourrez utiliser la méthode `insert` sur les listes.

```
1 def insertion_triée(L) :
2     res = []
3     for i in range(len(L)) :
4         j = index_insertion(res,L[i])
5         res.insert(j , L[i])
6     return res
```