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 procèdure `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 un 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 (ni compréhensions).

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`.

```
>>> max_pair([9, 17, 8, 11, 4])
8
>>> max_pair([9, 7, -12, 11])
-12
>>> max_pair([9, 7, 11])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 12, in max_pair
ValueError:  Pas de valeur paire
>>> max_pair([])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 12, in max_pair
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.

```python
def mon_max(a,b):
    if a == None:
        return b
    elif b == None:
        return a
    else:
        return max(a,b)

def max_pair(liste):
    maximum = None
    for valeur in liste:
        if valeur%2==0:
            maximum = mon_max(maximum, valeur)
    if maximum == None:
        raise ValueError("Pas de valeur paire")
    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.

```python
def max_pair(liste):
    déjà_trouvé = False
    maximum = None
    for valeur in liste:
        if valeur%2==0:
            if déjà_trouvé:
                maximum = max(maximum, valeur)
            else:
                maximum = valeur
                déjà_trouvé = True
    if not déjà_trouvé:
        raise ValueError("Pas de valeur paire")

    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.

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

```python
def afficher_max_pair(liste):
    try:
        valeur = max_pair(liste)
        print("Le plus grand pair est", valeur)
    except:
        print("De la musique avant toute chose,")
        print("Et pour cela préfère l'Impair")
        print("Plus vague et plus soluble dans l'air,")
        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`.

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

2. Écrivez des tests avec `assert`.

```
assert est_triée([]) == True
assert est_triée([1]) == True
assert est_triée([1,3,7]) == True
assert est_triée([12, 10]) == False
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]`.

```
def grouper(L) :
    if len(L) == 0 :
        return []
    e = L[0]
    res = [e]
    for i in range(1,len(L)) :
        if L[i] != e :
            e = L[i]
            res.append(e)
    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)]`.

```
def compacter(L) :
    if len(L)==0 :
        return []
    # L contient au moins un élément
    e = L[0]
    c = 1
    res = []
    for i in range(1,len(L)) :
        if L[i] != e :
            res.append((c,e))
            e = L[i]
            c = 1
        else :
            c = c+1
    res.append((c,e))
    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.

```python
def index_insertion(L,n) :
    for i in range(len(L)):
        if L[i]>n :
            return i
    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.

```python
def insertion_triée(L) :
    res = []
    for i in range(len(L)) :
        j = index_insertion(res,L[i])
        res.insert(j , L[i])
    return res
```