TD 3 : LISTES

Exercice 1. Tri d'une liste

(1) Ecrire une fonction `argmin(L)` qui prend en argument une liste `L` de nombres décimaux et renvoie l'**indice** du plus petit élément de `L`.

(2) Ecrire une fonction `tri(L)` qui prend en argument une liste `L` de nombres décimaux et renvoie une nouvelle liste comportant les éléments de `L` triés dans l'ordre croissant. La fonction `tri` fera appel à la fonction `argmin` de l'exercice précédent et ne fera pas appel à une fonction de tri de Python.

Exercice 2. Chercher une valeur dans une liste, version itérative

N.B. Pour faire cet exercice vous ferez semblant d'ignorer l'existence de l'opérateur « élément in liste » (mais vous pouvez utiliser la boucle « for var in liste »).

(1) Écrivez une fonction itérative[1] `present(L, X)` qui prend pour arguments une liste `L` et une valeur `X` et répond `True` ou `False` en réponse à la question « X est-il un élément de la liste L ? »

Essai :

```
liste = [ 10, 20, 30, 40, 50, 60, 70, 80, 90 ]
print 50, '-->', present(liste, 50)
print 55, '-->', present(liste, 55)
Affichage obtenu :
50 --> True
55 --> False
```

(2) Modifiez la fonction précédente pour qu'elle renvoie l'indice de la première occurrence de `X` dans `L` ou `-1` lorsque `X` n'est pas membre de `L`.

Avec le même code d'essai que ci-dessus, l'affichage obtenu est à présent :

```
50 --> 4
55 --> -1
```

Exercice 3. Recherche du minimum et du maximum d'une liste

(1) Écrivez une fonction `minEtMax(uneListe)` qui prend pour argument une liste dont les éléments appartiennent à un ensemble ordonné (nombres, chaînes de caractères, etc.) et qui détermine et renvoie la plus petite et la plus grande valeur de la liste.

(2) Exemple d'utilisation :

```
liste = [20, 25, 40, 15, 35, 10, 40, 28, 17 ]
print minEtMax(liste)
affichage obtenu : (10, 40).
```

Exercice 4. Recherche du minimum et du maximum d'une liste, version 2

(1) Écrivez une fonction `minEtMax(uneListe, compar)` qui prend pour argument une liste et détermine son minimum et son maximum selon le critère `compar`.

C'est l'utilisateur qui fournit la fonction `compar` au moment d'appeler `minEtMax`. La convention est : `compar(a, b)` renvoie une valeur négative, nulle ou positive selon que `a` est inférieur, égal ou supérieur à `b`.

Application. Utilisez cette fonction `minEtMax` pour déterminer le minimum et le maximum d'une liste de chaînes de caractères, en utilisant le critère suivant : `a` sera tenue pour

1. Itératif (c'est-à-dire basé sur une boucle) s'oppose généralement à récursif (c'est-à-dire écrit sous la forme d'une fonction qui s'appelle elle même).

inférieure à `b` si `a` est plus courte que `b` ou si `a` et `b` sont de même longueur et `a < b` pour l'ordre habituel des chaînes.

Exemple d'utilisation :

```
def comp(a, b):
c'est à vous d'écrire cette fonction
liste = [ 'aabb', 'cc', 'bbb', 'd', 'aaaa' ]
print minEtMax(liste, comp)
Affichage obtenu : ('d', 'aabb').
```

Exercice 5. Fusion de deux listes ordonnées

(1) Écrivez une fonction `fusion(l1, l2)` qui prend deux listes `l1` et `l2`, sans doublons et triées (c'est-à-dire arrangées en ordre croissant) et les réunit en une unique liste, également triée. Les éventuels doublons doivent être réduits (i.e. on ne conservera dans la liste produite qu'un des deux éléments du doublon).

Exemple d'utilisation :

```
L1 = [ 4, 6, 12, 14, 16, 22, 24, 26, 28, 30 ]
L2 = [ 2, 8, 10, 14, 18, 20 ]
print fusion(L1, L2)
Affichage obtenu :
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

Indication. Une des manières de programmer l'algorithme de la fusion de deux séquences ordonnées (listes, tableaux, etc.) `A` et `B` consiste à maintenir deux indices `i` et `j`, initialisés à `0`, tels que `i` (resp. `j`) désigne le premier élément de `A` (resp. `B`) qui n'a pas encore été copié dans `C` (`C` est la séquence, initialement vide, résultat de la fusion). Et on répète l'opération suivante :

comparer `A[i]` et `B[j]` :

— si `A[i]` $<$ `B[j]` , copier `A[i]` dans `C` et incrementer `i`
— sinon, si `A[i]` $>$ `B[j]` , copier `B[j]` dans C et incrementer `j`
— sinon, copier `A[i]` dans `C` puis incrementer `i` et `j`

Ces itérations s'arrête lorsqu'une des deux séquences a été épuisée. Ne pas oublier de recopier alors les éléments restants de l'autre séquence dans le résultat.

Exercice 6. Trois transformations à la manière de Lisp

(1) Écrivez une fonction `trans(L, F)` qui prend une liste `L` et une transformation `F` et qui renvoie la liste formée des transformés par `F` des éléments de `L` : si `L` est la liste $[\alpha_0, \alpha_1 \dots \alpha_{k-1}]$ cette fonction renvoie la liste $[F(\alpha_0), F(\alpha_1) ... F(\alpha_{k-1})]$

Remarque 1. Pour cet exercice, faites semblant de ne pas connaître l'existence en Python d'une construction appelée list comprehension qui fait exactement le travail demandé à la fonction trans.

Exemple d'utilisation. Proposons-nous d'afficher la liste des carrés des nombres impairs positifs inférieurs à 1000 :

```
liste = range(1, 100, 2)
def carre(x):
return x * x
print trans(liste, carre)
Affichage obtenu :
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, ... 9409, 9801]
3.
```

Remarque 2. La fonction `carre` est très simple et à usage unique : elle apparaît dans l'appel de trans et probablement nulle part ailleurs. Dans ce genre de situation il est plus commode d'employer la construction `lambda`, qui permet de définir une fonction sans lui associer un nom. L'exemple précédent devient :

```
liste = range(1, 100, 2)
print trans(liste, lambda x : x * x)
Le résultat obtenu est le même.
```

(2) Écrivez une fonction `cumul(L, ope, init)` qui prend une liste `L`, une opération binaire `ope` et l'élément neutre `init` de cette opération et qui calcule et renvoie le cumul des éléments de `L` pour l'opération `ope`.

Exemple d'utilisation. Calculons la some et le produit des dix premiers nombres entiers strictement positifs :

```
liste = range(1, 10 + 1)
print 'liste :', liste
print 'somme :', cumul(liste, lambda x,y : x + y, 0)
print 'produit :', cumul(liste, lambda x,y : x * y, 1)

Affichage obtenu :
liste : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
somme : 55
produit : 3628800
```

Testez encore votre fonction en calculant la concaténation des chaînes d'une liste de chaînes.

(3) Écrivez une fonction `filtre(L, cond)` qui prend une liste `L` et une fonction booléenne à un argument `cond` et construit la sous-liste de `L` formée des éléments qui satisfont la condition `cond`.

Exemple d'utilisation. Affichons la liste des entiers positifs inférieurs à 1000 qui ne sont pas multiples de 2, 3 ou 5 :

```
liste = range(1000)
def acceptable(x):
return x % 2 != 0 and x % 3 != 0 and x % 5 != 0
print filtre(liste, acceptable)
Affichage obtenu :
[1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, ... 989, 991, 997]
```

Exercice 7. Permutations aléatoires

(1) Écrivez une fonction permut(n) qui construit et renvoie une permutation aléatoire de la liste `[0, 1, ...n - 1]`.

Exemple d'utilisation :

```
>>> permut(10)
[8, 7, 0, 3, 5, 4, 1, 2, 9, 6]
>>>
```

Conseil. Utilisez la fonction `randint(a, b)` du module `random`.

(2) En utilisant la fonction précédente, écrivez une fonction permliste(uneListe) qui construit et renvoie une permutation aléatoire de la liste donnée comme argument.

Exemple d'utilisation :

```
>>> print permliste(['André', 'Béa', 'Charles', 'Diane', 'Emile', 'Fabrice'])
['Béa', 'Emile', 'Diane', 'André', 'Fabrice', 'Charles']
>>>
```