TP : Listes et programmation fonctionnelle

Instruction générale : hormis pour les exercices corrigés collectivement, vous ferez valider votre travail par l'enseignant.

1 Premières fonctions

Exercice 1. Récupérer le fichier listes.py ici, il contient les fonctions de bases nécessaires à la construction des listes dans un paradigme fonctionnel. Ouvrir ce fichier puis identifier les fonctions qu'il contient. Celles-ci pourront être appelées ultérieurement grâce à la commande `from listes import *` dans d'autres programmes à condition que les fichiers soient dans le même dossier.

Exercice 2. Écrire une fonction `entiers_entre_bornes` prenant en entrée deux entiers et permettant de créer la liste des entiers entre ceux-ci dans l'ordre croissant (attention la tête de la liste sera en première position).

Exercice 3. [Mapping] Nous allons maintenant implémenter une fonctionnalité que l'on rencontre dans la majorité des langages qui manipulent les listes : le mapping. Le mapping consiste à appliquer le même traitement à tous les éléments de la liste et à construire une nouvelle liste avec les résultats obtenus.

1. Soit `fonction` une fonction et `liste` une liste d'éléments auxquels peut s'appliquer cette fonction. Écrire une fonction récursive `mapping(liste,fonction)` qui renvoie une liste construite en appliquant `fonction` à tous les éléments de `liste`.
2. Programmer les fonctions `carre` et `double` qui renvoient respectivement le carré et le double du nombre passé en argument.
3. Utiliser les fonctions des questions précédentes pour construire la liste des carrés et la liste des doubles des éléments de la première liste.

Exercice 4. [Sélection] Dans cet exercice nous allons implémenter une deuxième fonctionnalité sur les listes : la **sélection**. La sélection consiste à créer la liste des éléments de la liste de départ qui vérifient une certaine propriété. Une fonction `predicat` renvoyant `True` ou `False` indique si l' élément vérifie la propriété ou non.

1. Écrire le code de la fonction récursive `selection(liste, predicat)` qui renvoie une liste construite en sélectionnant tous les éléments `x` de `liste` tels que `predicat(x) == True`.
2. Programmer les fonctions prédicats `est_pair` et `est_inferieur_a_5` qui renvoient `True` ou `False` selon que le nombre passé en argument est respectivement pair ou inférieur à 5.

Exercice 5. [Agrégation] Nous allons maintenant implémenter la fonctionnalité **d'agrégation** (ou **réduction**) des éléments d'une liste. C'est une fonctionnalité très puissante, en effet elle permet aussi bien d'effectuer la somme des éléments de la liste, que leur produit, ou encore que la recherche du maximum ou du minimum. Toutes ces fonctionnalités consistent, en partant d'une **graine** a, à parcourir la liste et à accumuler un certain résultat pendant le parcours.

Regardons en détail, soit $[a_0, a_1, a_2]$ une liste.

— La somme des éléments de la liste est $S = a_0 + (a_1 + (a_2 + a))$ avec $a = 0$.

— Le produit des éléments est $P = a_0 \times (a_1 \times (a_2 \times a))$ avec $a = 1$.

— Le maximum de la liste est $M = \max(a_0, \max(a_1, \max(a_2, a)))$ avec $a = -\infty$.

Si la liste est $[a_0, a_1, ..., a_n]$ alors l'agrégation par la fonction f de cette liste consiste à calculer $f(a_0, f(a_1, f(...f(a_n, a))...)$.

1. Programmer la fonction `agregation` dont on donne le début du code ci-dessous.

```
def agregation(liste, fonction, graine) :
    if est_vide(liste) :
        return graine
    else :
        ...
```

2. Programmer des fonctions `somme(x,y)` et `produit(x,y)` et tester la fonction d'agrégation en calculant la somme et le produit des éléments de la première liste.
3. Tester également la recherche du maximum et du minimum.

2 Fonction `lambda`

Imaginons maintenant que nous devions calculer régulièrement la somme des éléments d'une liste. Il n'est pas commode d'écrire à chaque fois `agregation(liste, somme, 0)`. La programmation fonctionnelle permet de créer une version de cette fonction pour laquelle certains arguments (ici `fonction` et `graine`) sont déjà renseignés. C'est ce qu'on appelle une fonction partielle.

Exemple : considérons la fonction définie par $f(x\,;y) = x \times y$. Si $g(x) = 2x$, on peut dire que g est une fonction partielle de f avec $g(x) = f(x\,;2)$. On pourrait alors écrire cela en Python :

```
def f(x,y) :
    return x*y

def g(x):
    return f(x,2)
```

Toutefois, cela est un peu lourd et la fonction `g` ne sera pas peut-être pas réutilisée. Il est possible de condenser cela et de déclarer des fonctions sans utiliser `def` ; on utilise pour cela `lambda` comme suit :

```
def f(x,y) :
     return x*y

g = lambda x : f(x,2)
```

Ainsi, on peut écrire la fonction `somme_liste` de la façon suivante :

```
somme = lambda x,y : x+y
somme_liste = lambda liste : agregation(liste, somme, 0)
```

Ou encore, condensé en une seule ligne :

```
somme_liste = lambda liste : agregation(liste,lambda x,y : x+y,0)
```

On remarquera que dans cette dernière version, il n'y a pas de fonction `somme` de définie et que celle-ci n'est donc pas utilisable ailleurs que dans la fonction `agregation`. Si l'on souhaite utiliser une fonction plusieurs fois, il vaut donc mieux la définir à part.

Exercice 6. Créer et tester les fonctions suivantes à l'aide de fonctions partielles.

1. `somme_liste(liste)`.
2. `produit_liste(liste)`.
3. `max_liste(liste)`.
4. `min_liste(liste)`.

3 Pour aller plus loin

Exercice 7. [Somme des carrées] Écrire une fonction qui calcule la somme des carrés des éléments d'une liste en utilisant la fonction `agregation`.

Exercice 8. [Longueur d'une liste] Écrire une fonction déterminant la longueur d'une liste.

Exercice 9. [Rangs pairs et impairs] Écrire des fonctions renvoyant les termes de rangs pairs et impairs d'une liste.

Exercice 10. [Miroir d'une liste] Écrire une fonction prenant une liste en entrée et la retournant renversée.

Exercice 11. [Vérification d'un prédicat]

1. Écrire une fonction `tous_pour_un(liste, predicat)` qui renvoie `True` si tous les éléments de la liste vérifient le prédicat et `False` sinon.
2. Écrire une fonction `un_pour_tous(liste, predicat)` qui renvoie `True` si un des éléments de la liste vérifie le prédicat et `False` sinon.