1. Introduction

Le but du TP est d'apprendre à programmer avec les fonctions de type map et reduce avec Python. Ce langage est particulièrement facile à apprendre avec une syntaxe et des fonctions simples. Et comme il est interprété, le cycle de développement est très court. Dans le TP3, ce seront les mêmes concepts, mais en Java et sur Hadoop.

Téléchargez le script Python3 tp2.py. Dedans, chaque question/réponse sera précédée d'un commentaire et d'un `print("\n### Question N.N.N")` indiquant son numéro. Lorsqu'on lance l'exécution, on voit chaque numéro de question et ses résultats. Voici le début de ce script ; attention, les réponses sont partiellement fausses :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# pour faciliter l'emploi de map, filter et reduce
maporig = map
map = lambda f, l: list(maporig(f, l))
filterorig = filter
filter = lambda f, l: list(filterorig(f, l))
from functools import reduce


# Question 3.1.a
print("\n### Question 3.1.a")
def carre(n):
    return n*3
print( map(carre, range(1, 7)) )

# Question 3.1.b
print("\n### Question 3.1.b")
def mkimpair(i):
    return "euh?"
print( map(mkimpair, range(1, 7)) )

# Question 3.2.a
print("\n### Question 3.2.a")
phrase = "BARATATINER ecraser son interlocuteur sous une masse de propos pourtant bien le
mots = None
```

Quelques points seront enlevés si vous ne respectez pas ces consignes.

Ce fichier `tp2.py` sera à déposer sur Moodle (remise du TP2) en fin de séance.

Pour travailler, on vous conseille l'éditeur `idle-python3`. C'est un éditeur conçu pour Python3. En appuyant sur `F5`, ça lance l'exécution du programme.

Le programme est écrit en Python3. Ce langage optimise la gestion des collections produites par `map`, `filter`, etc. Par exemple, `map` ne retourne pas une liste mais un itérateur. Ça permet d'économiser la mémoire et le temps de calcul dans le cas où la liste serait énorme. Mais pour vous, c'est une complication de plus. C'est pour cela que les fonctions map et filter ont été redirigées vers list(map...) et list(filter...).

2. Rappels sur les listes

Ouvrez un terminal shell et tapez `python3`. Ça lance l'interpréteur Python en ligne de commande. Vous pouvez y faire des essais simples, mais pas enregistrer vos actions.

Une liste se définit très simplement en Python, essayez ceci dans le terminal :

```
liste = [1, 2, 3, 5, 8, 11]
print(liste)
divers = [4, 'ok', 2.71, "ça va"]
print(divers)
vecteurs = [ (1,2), (5,3), (-2,4) ]
print(vecteurs)
entiers = range(1, 10, 1)
print(entiers)
print(list(entiers))
```

Quelques remarques :

- Les chaînes ou les caractères isolés doivent être mis entre `'quotes'` ou `"guillemets"`.
- La liste `vecteurs` est une liste de *tuples* (n-uplet). Un *tuple* Python est une sorte de liste, mais constante (nombre d'éléments et éléments fixes).
- La fonction `range(premier, limite, écart)` crée un itérateur sur les entiers allant de premier à limite exclue. L'écart est optionnel, par défaut c'est 1. Et on convertit ce `range` (un itérateur appelé *générateur* en python) en liste pour l'affichage.

On peut aussi générer une liste à partir d'une chaîne, avec la méthode `split(séparateur)`. Comme en Java, elle découpe une chaîne en mots selon le séparateur indiqué. Essayez ceci, les espaces du print sont pour la lisibilité :

```
print( "bonjour tout le monde".split(' ') )
print( "bonjour tout le monde".split('o') )
```

La fonction inverse de `split` s'appelle `join` :

```
print( 'OO'.join("bonjour tout le monde".split('o')) )
```

Une liste de couples peut être produite par la fonction `enumerate(liste)`. Ces couples sont formés de l'indice de l'élément et de l'élément de la liste. NB: c'est un itérateur, alors pour afficher la liste résultante, il faut faire `print( list(enumerate(liste)) )`.

Une liste possède différentes fonctions parmi lesquelles il faut connaître :

- `len(liste)` retourne le nombre d'éléments de la liste,
- `liste[indice]` retourne l'élément indiqué par son indice. Comme en C et Java, les indices commencent à zéro. En Python, les indices peuvent être négatifs ; ils signifient alors de se repérer par rapport à la fin de la liste. Par exemple `mots[-1]` est le dernier élément de `mots`. Ces indices marchent aussi sur des tuples.

- `sum(liste)` retourne la somme des éléments de la liste si elle contient des entiers. Il y a aussi `min` et `max`. Nous verrons que ce sont des fonctions de réduction.

3. Application de fonctions sur une liste

La programmation fonctionnelle consiste, entre autres, à écrire des traitements à appliquer sur des listes pour produire d'autres listes ou des valeurs récapitulatives (agrégations). Ces traitements sont définis à l'aide de fonctions « transformatrices de données ».

Par exemple :

```
def negatif(n):
    return -n

l1 = [1, 3, -2, 8, -4]
print(l1)
l2 = map(negatif, l1)
print(list(l2))
l3 = map(negatif, map(negatif, l1))
print(list(l3))
```

Dans cet exemple, la fonction `negatif` est passée en paramètre à la fonction `map`, comme si c'était un objet : on ne met que son nom, sans aucun paramètre.

La fonction `map(f, l)` est l'une des fonctions qui nous intéresse beaucoup dans ce cours. Elle applique f sur chaque élément de la liste l et retourne un itérateur sur les résultats. Pour l'affichage, on convertit cet itérateur en liste.

$$map(f, [a, b, c, d]) = [f(a), f(b), f(c), f(d)]$$

On peut imbriquer les appels :

```
l1 = [1, 3, -2, 8, -4]
l3 = map(negatif, map(negatif, l1))
print(list(l3))
```

On revient maintenant sur le fichier `tp2.py` à compléter.

3.1. Exercices avec `map` sur des entiers

Chacun de ces exercices demande d'écrire une fonction qui sera appelée sur les éléments d'une séquence pour les transformer dans le résultat attendu.

a. Écrire la fonction Python `carre(n)` qui permet d'obtenir la liste des carrés des 10 premiers entiers naturels en faisant `map(carre, range(1, 11))`.
b. Écrire la fonction Python `mkimpair(i)` qui permet d'obtenir la liste des 10 premiers entiers impairs en faisant `map(mkimpair, range(0, 10))`. Piège : il ne faut pas voir cette fonction comme un filtre qu'on appliquerait sur la liste des 20 premiers entiers et qui ne garderait que les impairs. Il faut penser à une fonction qui calcule $Impair_i$ en tant que i-ème élément de la suite des entiers impairs.

3.2. Exercices avec `map` sur des mots

a. Écrire une instruction Python qui affecte la variable `mots` avec une liste construite en découpant la phrase « `BARATATINER écraser son interlocuteur sous une masse de propos pourtant bien légers` » (Alain Créhange, voir ce lien). Remarque : pour que `split` fonctionne bien, il faut aussi retirer la ponctuation.
b. Afficher le nombre de mots de la liste `mots` : tout simplement `len(mots)`. Puis afficher la longueur de chaque mot de cette liste. Il suffit d'appliquer par `map` la fonction `len` aux éléments de la liste `mots`.
c. Écrire la fonction `est_voyelle(lettre)` qui retourne `True` si la lettre est une voyelle, `False` sinon. Il suffit qu'elle retourne `lettre.lower() in 'aeiouy'`. Appliquer cette fonction à la chaîne `'bonjour'`. Vous remarquerez que les chaînes sont considérées comme des listes de lettres en Python.
d. Écrire la fonction `bool_to_int(b)` qui retourne l'entier 1 si le paramètre est `True`, 0 sinon. Appliquer cette fonction au résultat de la question précédente en imbriquant les deux appels à `map`.
e. Écrire la fonction `nb_voyelles(mot)` qui retourne le nombre de voyelles du *mot*. Elle se contente d'utiliser `sum` sur le résultat de la question précédente appliquée au mot. Appliquer cette fonction sur chaque élément de la liste `mots` – cela affiche la liste des nombres de voyelles des mots de la phrase.

3.3. Exercices avec `map` traitant des tuples

On va compliquer un peu. Maintenant, au lieu de retourner une simple valeur, la fonction fournie à `map` retourne deux valeurs sous forme d'un couple. Il suffit d'écrire `return (valeur1, valeur2)`. Pour l'instant, ça va paraître un peu artificiel, mais ça trouvera son utilité plus loin.

a. Écrire la fonction `lng_nb_voyelles(mot)` qui retourne des couples formés de la longueur du mot et du nombre de voyelles – ce nombre est obtenu par la fonction `nb_voyelles` de la question précédente. Appliquer cette fonction à la liste `mots`.

On complique encore un peu. Cette fois, c'est la liste qui est traitée par `map` qui est composée de couples. Par exemple, `map(fonction, [ (-2, 3), (5, -1), (3, 4) ])`. Dans ce cas, il faut que le paramètre de la fonction représente un couple. Par exemple :

```
def fonction( a_b ):
    a,b = a_b
    return a + b
print( map(fonction, [ (-2, 3), (5, -1), (3, 4) ]) )
```

b. Écrire la fonction `invquotient( a_b )` qui retourne `float(b)/float(a)` dans cet ordre. Appliquer cette fonction au résultat de l'exercice précédent pour obtenir le taux moyen de voyelles dans chaque mot de la phrase.
c. Écrire la fonction `mult( a_b )` qui retourne le produit de a par b. Soit un code ISBN10 écrit sous forme d'une liste, par exemple `isbn10 = [0,3,8,7,7,1,6,7,5,0]`. Faites afficher `enumerate(isbn10)` (il faut écrire `print( list(enumerate(isbn10)) )`. Appeler la fonction `map` avec la fonction `mult` sur `enumerate(isbn10)`. Vous obtiendrez une liste de nombres. Affichez leur somme : `print( sum(map(mult, enumerate(isbn10))) )`. Normalement, c'est un nombre divisible par 11, sinon le code ISBN est mauvais.

d. On s'intéresse maintenant aux codes ISBN13 : `isbn13=[9,7,8,2,8,2,2,7,0,1,6,2,4]`. Le dernier chiffre est un code de vérification selon une méthode à peine plus complexe. Écrire la fonction `poids13` qui reçoit un couple (i, v) en paramètre et qui retourne v si i est pair et $v * 3$ si i est impair. Appliquez cette fonction à `enumerate(isbn13)`, la somme des nombres doit être un multiple de 10.

e. Parfois, au lieu de `enumerate` pour créer des couples (indice, valeur), on utilise `zip(liste1, liste2)`. Cette dernière fabrique des couples (i^{e} élément de liste1, i^{e} élément de liste2). Par exemple, `zip(isbn13, [1,3,1,3,1,3,1,3,1,3,1,3,1])` permet d'écrire la vérification du code ISBN13 autrement. Montrez comment.

4. Filtrage

Voici maintenant une nouvelle action distribuée sur les listes : `filter(fonction, liste)`. Elle retourne les éléments de la liste pour lesquels la fonction retourne `True`. Il y a une différence entre `filter` et `map`. La seconde renvoie les résultats de la fonction appliquée aux éléments de la liste, tandis que la première retourne certains éléments de la liste traitée (pas les résultats des tests).

a. Écrire la fonction `est_pair(n)` qui retourne `True` si n est pair, `False` sinon (l'opérateur modulo s'écrit `%`). Utilisez cette fonction ainsi que `carre` du 3.1.a pour afficher uniquement les carrés pairs des 20 premiers entiers positifs :`[4, 16, 36, 64,...]`. C'est à dire qu'on traite tous les entiers entre 1 et 20, on calcule leur carré et on élimine ceux qui sont impairs comme 1, 9, 25, 81…
NB: la réponse peut aussi être trouvée par un simple raisonnement mathématique. Comment pourrait-on faire pour obtenir les entiers dont le carré est pair ?

Il y a un cas particulier, quand la fonction est `None`, comme dans `filter(None, [1, 2, None, 4])`. Elle retourne la liste des éléments autres que `None`.

5. Fonction ou *lambda* ?

La notation *lambda* permet d'économiser une définition de fonction lorsque le corps de la fonction est réduit à une expression (un seul calcul). La syntaxe est alors `lambda paramètres: expression`, au lieu de `def fonction(paramètres): return expression`.

- Map avec une fonction :

```
def inverse(x):
    return 1.0 / float(x)
print( map(inverse, range(1,10)) )
```

- Map avec une *lambda* :

```
print( map(lambda x: 1.0 / float(x), range(1,10)) )
```

Il est possible de ré-écrire certains des exercices de ce TP à l'aide d'une *lambda*, ceux où la fonction ne fait que retourner une expression. Par contre, c'est impossible dès qu'il y a une boucle ou une conditionnelle[1]. L'autre problème des *lambas*, c'est qu'elles sont peu lisibles. On les réservera donc à des instructions très simples.

6. Lecture d'une liste à partir de fichiers

On va se tourner vers des données un peu plus utiles que des listes de nombres ou des phrases humoristiques. On s'intéresse aux fichiers CSV. Ils représentent une table de base de données ; chaque ligne représente un n-uplet de la table ; les champs sont séparés par un caractère spécial absent des données, en général une tabulation, un « ; » ou une « , ».

On va travailler sur un fichier qui décrit les arbres remarquables de Paris, arbres.csv du TP précédent. Chaque ligne décrit un arbre : position GPS, arrondissement, genre, espèce, famille, année de plantation, hauteur, circonférence, etc. Les champs sont séparés par des `';'`. Voici ses premières lignes.

```
Geo point;ARRONDISSEMENT;GENRE;ESPECE;FAMILLE;ANNEE PLANTATION;HAUTEUR;CIRCONF...
(48.857140829, 2.295334554);7;Maclura;pomifera;Moraceae;1935;13.0;;Quai Branly...
(48.8685686134, 2.31331809304);8;Calocedrus;decurrens;Cupressaceae;1854;20.0;...
```

Voici comment le lire en Python et en faire une liste de n-uplets (chacun étant un dictionnaire). Ajoutez cet extrait à `tp2.py` :

```
with open('arbres.csv') as entree:
    noms = entree.readline().strip().split(';')
    arbres = map(lambda ligne: dict(zip(noms, ligne.strip().split(';'))), entree)
```

Voici quelques explications.

- `with open... as` est une structure de contrôle Python qui permet d'ouvrir un fichier sous le nom `entree` en veillant à le refermer hors du bloc `with` même s'il y a une exception dans le bloc.
- `entree.readline().strip().split(';')` enchaîne plusieurs méthodes. `entree.readline()` retourne la prochaine ligne du fichier ; comme on vient de l'ouvrir, c'est la première. `strip()` supprime le `\n` de la ligne. `split()` sépare en mots. On obtient donc la liste des noms des champs.
- La troisième ligne est la plus complexe. Elle est du genre `arbres = map(fonction, entree)`. Lorsqu'on applique `map` à un fichier, ça applique la fonction sur chacune de ses lignes. C'est pour ça que la *lambda* a un paramètre appelé `ligne`. Son corps consiste en `dict(zip(noms, liste des mots))`. La fonction `dict` demande une liste de couples et construit un dictionnaire. Donc, avec chaque ligne, ça crée un dictionnaire {nom du champ: valeur du champ, ...}.

Juste pour vérifier, faites `print(arbres)` : chaque élément est un dictionnaire tel que `{'ANNEE PLANTATION': '1935', 'OBJECTID': '6', 'ARRONDISSEMENT': '7', ...}`. On peut par exemple avoir la hauteur du premier arbre par `arbres[0]['HAUTEUR']`.

En ce qui concerne les champs de type nombre, voici comment faire, par exemple pour récupérer la circonférence d'un arbre :

[1] Il y a une syntaxe Python qui permet d'écrire des conditionnelles simples : `valeurV if condition else valeurF`.

```
def get_circonference(arbre):
    try:
        return float(arbre['CIRCONFERENCE'])
    except:
        return None
```

Ces accesseurs vont être utilisés dans des fonctions `map` pour extraire de l'information de ce fichier.

6.1. Exercices sur le fichier CSV

a. Écrire la fonction `get_annee(arbre)` qui reçoit un n-uplet et qui retourne l'année de plantation de l'arbre. Elle retourne le champ `ANNEE PLANTATION` sous forme d'un entier. Appliquer cette fonction sur la liste des arbres.
b. Écrire la fonction `est_vieil_arbre(arbre)` qui retourne `True` si l'année de l'arbre est connue (pas `None`) et strictement inférieure à 1800 (utiliser `get_annee`). Avec un `filter` et un `map`, afficher uniquement les années des arbres concernés.
c. Écrire la fonction `get_hauteur(arbre)` qui retourne la hauteur sous forme d'un réel. Affecter la variable `hauteurs` avec le résultat de `get_hauteur` sur les arbres.
d. On voudrait calculer la somme des hauteurs avec la fonction `sum`, mais le problème, c'est qu'il y a des `None` dans la liste. Utiliser `filter` pour éliminer les `None`. Puis calculer la somme des hauteurs des arbres. Calculer ensuite la hauteur moyenne des arbres en utilisant cette somme et `len`.

7. Réduction

Jusqu'à présent, on a utilisé les fonctions d'agrégation prédéfinies `len`, `sum`, `min` et `max`. Ces fonctions calculent une valeur récapitulative à partir d'une collection. Il est possible de définir notre propre fonction. Elle doit recevoir deux paramètres et retourner leur agrégation. Il faut seulement qu'elle soit associative afin que le calcul soit correct. C'est une contrainte importante, il y a peu de fonctions qui le sont.

Ensuite, on fait `reduce(fonction, liste)` pour calculer l'agrégation.

Par exemple, voici comment calculer 10 ! :

```
def produit(a, b):
    return a * b

print("10! = %d" % reduce(produit, range(1,11)))
```

NB: la fonction `mult` de la page 4 ne peut pas être utilisée car elle ne reçoit qu'un seul paramètre, un tuple.

7.1. Exercices de réductions

a. Écrire la fonction d'agrégation `max_arbres` qui reçoit deux arbres venant de la liste `arbres` et qui retourne uniquement celui des deux qui a la plus grande hauteur. Utiliser cette fonction pour afficher le plus grand arbre de la liste.

b. La fonction $moyenne(a, b) = \frac{1}{2}a + \frac{1}{2}b$ n'est pas associative car $moyenne(moyenne(a, b), c) \neq moyenne(a, moyenne(b, c))$. Donc il n'est pas possible d'écrire une fonction `moyenne(a,b)` qui puisse être employée dans un `reduce`. Faites quand même l'essai sur la liste `nombres = [1,3,5,4,6,8]`.

c. Le problème de la réduction d'une moyenne vient du fait que $moyenne(moyenne(a, b), c) = \frac{1}{4}a+\frac{1}{4}b+\frac{1}{2}c$. Les poids des valeurs sont incorrects. Alors pour calculer quand même la moyenne d'une liste de valeurs par un *reduce*, on va écrire une fonction `moyenne_poids( (v1,p1), (v2,p2) )` qui travaille avec des couples (valeur, poids). Initialement le poids de chaque valeur est 1. Quel couple faut-il retourner dans `moyenne_poids` pour que `reduce(moyenne_poids, zip(nombres, [1.0]*len(nombres)))` soit correct ? (pour comprendre, faites afficher `[1.0]*len(nombres)` puis `zip(nombres, [1.0]*len(nombres))`).

8. Travail à rendre

Déposez le fichier `tp2.py` sur la page Moodle dédiée.