Langages de script (Python)
CMTP n° 9 : Préparation à l'examen

Dans ce dernier CMTP nous allons voir des exemples d'exercices d'examen.
Pour info : l'examen aura lieu *en présentiel* jeudi 12 mai de 9h30 à 12h.
L'examen sera sur papier et le sujet sera composé de 5 exercices (à différence de ce dernier CMTP, qui en contient moins, en considération de la durée du créneau). Aucun document ne sera autorisé sauf *au plus* deux feuilles recto/verso (4 pages en total, l'équivalent d'une copie double) qui sont strictement personnels et doivent apporter votre nom et prénom sur chaque page.

Exercice 1 :

On considère la définition de `l` suivante :

```
a = [['a', [2, 3, 'b'], [4]], [['c', 6], [7]], 10, (11, 9), (1, 2)]
```

Indiquer ce qui est affiché par les commandes suivantes : (Indiquer "Erreur" si un erreur se produit).

`print(2*a[0])`	
`print(a[2]-1)`	
`print(a[0][0:2])`	
`print(a[0][1][-1])`	
`print(a[0][1]+[a[3][0]])`	
`print(a-a)`	
`print(a[0:1:2])`	
`print(a[0][1:-1])`	

Exercice 2 :

Le but de cet exercice est d'implémenter des fonctions de manipulation des *listes associatives*. Ces dernières permettent d'obtenir les mêmes fonctionnalités qu'un dictionnaire mais en utilisant uniquement des listes. Une façon d'implémenter une liste associatives est d'utiliser une liste de couples (clef, valeur).

1. Écrivez une fonction `list_assoc_append(l, key, value)`. Cette fonction permet de lier `value` à la clef `key` dans la liste `l`.
2. Écrivez une fonction `list_assoc(l, key)` qui pour une clef `key` renvoie la dernière valeur liée dans liste `l`. Si la clef n'est pas dans la liste, retournez `None`. Elle doit se comporter ainsi :

```
l = list([])
list_assoc_append(l, "answer", 10)
list_assoc_append(l, "answer", 42)
list_assoc(l,"answer")
>>> 42
list_assoc(l, "toto")
>>> None
```

3. Écrivez une fonction `list_assoc_split(l)` qui renvoie deux listes, la première contenant toutes les clefs et la deuxième contenant toutes les valeurs.

```
l = list([])
list_assoc_append(l, "answer", 10)
list_assoc_append(l, "answer", 42)
list_assoc_split(l)
>>> (["answer", "answer"], [10, 42])
```

4. Écrivez une fonction `list_assoc_map(l, f)` qui renvoie une liste dont chaque élément est le résultat de f(k, v), où (k, v) est la couple correspondante dans l. Par exemple :

```
def plus1(k, v):
    return v + 1

l = list([])
list_assoc_append(l, "foo", 10)
list_assoc_append(l, "bar", 41)
list_assoc_map(l, plus1)
>>> ([11, 42])
```

Exercice 3 :

Le but de cet exercice est d'implémenter une version simplifiée de `make`. Cette outil permet d'automatiser de commandes de compilation à lancer sur des fichiers, surtout `make` est capable de déterminer l'ordre dans lequel les commandes doivent être lancées.
Ces fichiers `makefiles` sont composés de règles. Chaque règles s'écrit dans le format suivant.

```
goal: dependencies
    command
```

- Chaque règle contient trois éléments : un but, une liste de dépendances et une commande.
- Le but est le nom du fichier produit par la commande de compilation.
- Les dépendances sont les fichiers nécessaires pour pouvoir exécuter la commande de compilation.
- La commande est une ligne de commande qu'on lance habituellement dans le shell, dont le résultat est la création du but. Par exemple `gcc -o main.exe main.c`.
- La syntaxe est définie ainsi. La première ligne est séparée en deux par deux point et un espace comme cela `':␣'`. La partie de gauche désigne le but et la partie droite les dépendances séparées par des espaces. La ligne suivante commence par un caractère de tabulation suivie d'une chaîne de caractères qui représente la ligne de commande à exécuter.
- En voici un exemple :

```
lslogo: main.o parser.o interpreter.o printer.o utils.o
	gcc -o lslogo main.o parser.o interpreter.o printer.o utils.o

main.o: main.c types.h
	gcc -c main.c

parser.o: parser.c types.h utils.o
```

```
	gcc -c parser.c

interpreter.o: interpreter.c types.h utils.o
	gcc -c interpreter.c

printer.o: utils.o types.h
	gcc -c printer.c

test.result: lslogo test.c parser.o interpreter.o utils.o types.h
	sh runtest.sh

utils.o: utils.c types.h
	gcc -c utils.c
```

L'exécution d'un Makefile consiste en trois étapes. La première est de lire le fichier, puis de déterminer l'ordre d'exécution des commandes, et finalement de les exécuter dans un ordre valable.

- La première étape consiste à parser un Makefile à partir d'un fichier textuelle. Il y a trois informations par règle à retenir (le but, les dépendance et la commande). Ainsi vous pourrez, p.ex., construire un dictionnaire qui à chaque but associe ses dépendances et sa commande compilation.
- La deuxième étape consiste a écrire un petit algorithme qui va calculer une liste des buts dans l'ordre dans lequel il faut exécuter chaque commande. Par exemple, le but `lslogo` dépend `main.o`. Il faut donc obtenir `main.o` avant de lancer la commande pour obtenir `lslogo`. Le but `test.result` est donc le dernier car il dépend de tous les autres buts alors que le but `utils.o` n'a aucune dépendance sur les buts il peut être le premier, le but `main.o` aussi pourrait être le premier.
  Pour cela vous pouvez utiliser sur l'algorithme de tri topologique, qui vous permet de construire une liste tel que pour deux élément a et b qui se suivent dans la liste, b peux dépendre de a et a ne peux pas dépendre de b. Le voici en pseudo-code :

```
fonction tri_topo_recursif(marques, solution, graph, s)
    Si marques[s] = NON_VISITE faire:
      marques[s] := VISITE_EN_COURS;
      Pour chaque voisin v dans voisins(s, graph) faire:
        Si marques[v] = VISITE_EN_COURS faire:
          lever une erreur pour DefinitionCyclique;
        Si marques[v] = NON_VISITE faire:
          tri_topo_recursif(marques, solution, graph, v);
      marques[s] := VISITE_FINI;
      ajouter(solution,s);

fonction tri_topo(graph)
    marques := {};
    solution := [];
    Pour chaque sommet s du graph faire:
      marques[s] := NON_VISITE;
    Pour chaque sommet s du graph faire:
      topo_sort_recursive(marques, solution, s);
    Returner solution.
```

  Pour l'exemple précédent une solution valable serait donc :

```
["main.o", "utils.o", "interpreter.o", "parser.o", "printer.o", "lslogo.o",
 "test.result"]
```

  Attention, certaines dépendances ne sont pas des buts, par exemple `types.h`, veillez à prendre cela en compte.
- Finalement pour exécuter une commande Unix depuis python vous pouvez utiliser la fonction suivante où la commande à exécuter est `ls -al`.

```
from subprocess import call
```

```
call("ls -al", shell=True)
```

Chaque question est indépendante et vous pouvez répondre aux questions même si vous n'avez pas réussi les questions précédentes.

1. Écrire une fonction `parse_file(filename)` qui transforme un fichier `makefile` en un dictionnaire qui à chaque but associe sa commande et ses dépendances ;
2. Écrire une fonction `rec_topo_sort(marks, schedule, rules, root)` qui implémente la partie récursive de l'algorithme du tri topologique ;
3. Écrire une fonction `get_schedule(rules)` qui implémente la deuxième partie de l'algorithme et retourne la liste de buts dans le bon ordre ;
4. Écrire une fonction `run_schedule(schedule, rules)` qui exécute la commande de chaque but dans l'ordre obtenu par `get_schedule`