L'intégramme de Lewis Caroll – Corrigé

L'énigme des 5 maisons.

Les intégrammes sont des petits jeux de logique semblables à des casse-têtes abstraits. Partant d'un certain nombres d'indices sur des éléments, le jeu consiste à déduire des relations non triviales entre ceux-ci, de manière à résoudre, au final, une énigme qui ne semblait pas corrélée aux indices initiaux.

L'un des intégrammes le plus célèbre est celui des 5 maisons, que l'on attribue tour à tour à Lewis Caroll ou à Einstein, bien qu'aucune preuve de la paternité de l'un ou de l'autre n'ait jamais été apportée. Lewis Caroll jouant aussi bien avec les mots [1] qu'avec les symboles mathématiques [2], on l'imagine sans peine poser l'énigme suivante :

Cinq voisins de nationalités et de professions différentes habitent les cinq premières maisons d'une même rue dont les façades sont singulières. Adepte d'une boisson particulière, chacun d'eux partage sa vie avec son animal de la façon suivante :

1. L'Anglais habite la maison rouge.
2. L'Espagnol adore son chien.
3. L'Islandais est ingénieur.
4. La maison verte sent bon le café.
5. La maison verte est située immédiatement à gauche de la blanche.
6. Le sculpteur possède un âne.
7. Le diplomate habite la maison jaune.
8. Le Norvégien habite la première maison à gauche.
9. Le médecin habite la maison voisine de celle où demeure le propriétaire du renard.
10. Le diplomate voit un cheval dans le jardin voisin tous les matins en ouvrant ses volets.
11. La maison du milieu reçoit une livraison de lait tous les mardis.
12. Le Slovène boit du thé.
13. Le violoniste presse ses oranges à la main pour son jus quotidien.
14. Le Norvégien jalouse le joli bleu de la maison voisine.

Qui élève un zèbre ?

1. Vous connaissez bien sûr *Les Aventures d'Alice aux pays des Merveilles...*
2. ...mais avez-vous déjà entendu parler de *l'algorithme de condensation de Dodgson* ? Selon la nature de ses activités, Lewis Caroll jonglait entre deux identités qu'il dissociait entièrement. Mathématicien, il signait alors de son nom de naissance : Charles Lutwidge Dodgson.

Avec un peu d'astuce, l'énigme se laisse résoudre à la main. Pourtant, si la patience vous manque, il est intéressant de savoir que l'on peut reléguer ce type de tâche à son ordinateur. Avant de faire travailler Python, nous allons "relaxer" les hypothèses pour nous permettre malgré tout d'obtenir un résultat sans attendre plusieurs jours. Pour les curieux, l'explication viendra plus loin : retenez qu'autrement, Python serait vaillant mais lent. Le problème simplifié sur lequel je vous propose de vous pencher est le suivant :

Enigme des 5 maisons pour Python. Cinq voisins de nationalités et de professions différentes habitent les cinq premières maisons d'une même rue. Chacun d'eux partage sa vie avec son animal de la façon suivante :

1. L'Anglais est sculpteur ou médecin, nul ne sait.
2. L'Espagnol adore son chien.
3. L'Islandais est ingénieur.
4. Le sculpteur possède un âne.
5. Le Norvégien habite la première maison à gauche.
6. Le médecin habite la maison voisine de celle où demeure le propriétaire du renard.
7. Le diplomate voit un cheval dans le jardin voisin tous les matins en ouvrant ses volets.
8. Le Slovène ne comprend pas son voisin Islandais.
9. Le violoniste ne parle pas norvégien. Tant mieux, son cheval non plus.
10. Le chien a peur du renard et a obligé son maître à déménager pour ne plus être côte à côte.
11. L'Islandais est rassuré d'avoir un voisin médecin depuis son cancer de l'orteil, et le Norvégien partage son avis pour se faire prescrire des ampoules de vitamine D plus facilement.

Qui élève un zèbre ?

Résolution de l'énigme en Python.

Les questions qui suivent sont volontairement moins guidées qu'à l'habitude. Pour certaines, des pistes de résolution se trouvent en fin de document. Je vous conseille pourtant de réfléchir au mieux avant d'aller lire ces indices – et si vous pouvez vous en passez, c'est encore mieux !

Question 1 : Structures de données. Votre algorithme va manipuler plusieurs ensembles de données : les nationalités, les professions, les animaux, – et, éventuellement, les emplacements, bien que l'on puisse faire sans. Il est donc naturel de commencer par déclarer ces 3 groupes de mots, chacun ayant 5 éléments. Parmi les listes, les n-uplets, les ensembles et les dictionnaires, quelle est la structure de données qui vous semble la plus appropriée ? Pourquoi ? Déclarez vos 3 premiers ensembles **nati, prof** et **anim** en

conséquence.

Correction : Un n-uplet sert à lier entre-eux plusieurs éléments éventuellement de natures différentes, ce qui n'est pas le cas pour le moment. De la même manière, les dictionnaires qui associent à une clef une valeur ne nous sont pas de grande utilité. Ici, il est nécessaire de pouvoir parcourir l'ensemble des nationalités (par exemple), il faut donc choisir parmi les ensembles ou les listes. L'avantage des ensembles sur les listes réside dans l'efficacité de l'ajout et du retrait des éléments, mais les groupes que nous devons déclarer ne changeront pas, a priori, au cours de l'algorithme. Pour contourner l'inconvénient de ne pas pouvoir ordonner les éléments, nous utiliserons donc des listes. Les 3 listes sont finalement déclarées et initialisées comme suit :

```
#list[str]
nati=["anglais","espagnol","islandais","norvegien","slovene"]
#list[str]
prof=["ingenieur","sculpteur","diplomate","medecin","violoniste"]
#list[str]
anim=["chien","ane","renard","cheval","zebre"]
```

Avant de répondre à la question de savoir qui élève un zèbre, vous allez devoir créér et tester différentes configurations, c'est-à-dire différentes combinaisons d'éléments à l'intérieur d'une même maison. Par exemple, on souhaite pouvoir faire comprendre à Python que le sculpteur anglais a un âne (c'est un exemple, pour le moment je suis comme vous : je n'en sais rien). Quel type de données vous semble approprié ? Déclarez un alias **maison** pour alléger les notations.

Correction : Un n-uplet sert à lier entre-eux plusieurs éléments éventuellement de natures différentes, on déclare donc le type maison :

```
#type maison = tuple[str,str,str]
```

Question 2 : Squelette de l'algorithme. Sans chercher une manière intelligente de procéder, proposez une méthode pour résoudre le problème, en décomposant celui-ci en plusieurs sous-étapes. En particulier, je vous déconseille d'essayer de calquer la méthode astucieuse que vous pourriez avoir envie d'appliquer si vous cherchiez à résoudre l'énigme "à la main". Un algorithme un peu brutal nous suffira *tant que cela fonctionne.* Donnez les signatures des fonctions que vous souhaitez créer.

Correction : Comme suggéré dans l'indice, nous nous contenterons d'un algorithme naïf de type "recherche exhaustive". La recherche exhaustive consiste à parcourir toutes les solutions d'un problème et à tester successivement si elles conviennent ou non. Il s'agit toujours de la méthode la plus simple, et donc la plus coûteuse (en terme de nombre de calculs effectués par votre ordinateur), mais, lorsque les tailles des données manipulées le permettent, c'est déjà une solution.

Une petite parenthèse à ce propos : il y a ici $(5!)^3 = 1728000$ configurations possibles qui correspondent chacune à la répartition des éléments dans les maisons. Il y a en effet 5! façons de répartir les nationalités (idem pour les professions et les animaux) parmi les habitants. Soit, au total, un peu moins de 2^{14} possibilités. Est-ce beaucoup ? Ou plutôt : est-ce que votre ordinateur va pouvoir parcourir toutes les possibilités rapidement ? A priori, oui, et assez rapidement. En revanche, pour le cas de l'énigme initiale, les considérations de boissons et de couleurs auraient amené $(5!)^5 = 24883200000$ configurations possibles. Avec le même algorithme que celui que je vous propose de faire, j'ai estimé le temps de calcul sur mon ordinateur à... 2 jours en moyenne, et $3,75$ jours dans le pire cas ! Vous comprenez mieux pourquoi j'ai simplifié les hypothèses.

Le squelette de l'algorithme sera donc le suivant. On définira la fonction **recherche** sans paramètre qui renvoie la configuration trouvée, ou, éventuellement, un message indiquant qu'il n'existe pas de solution. Je vous propose à partir de maintenant de voir une configuration comme une liste de maisons à 5 éléments. Ainsi, la signature de la fonction précédente est :

```
def recherche():
    """->list[maison] + str"""
```

Ensuite, une fonction **solution**, sans paramètre elle aussi, se servira de l'exécution de la fonction précédente pour donner la réponse attendue, à savoir, qui est le propriétaire du zèbre. Elle est donc de signature :

```
def solution():
    """->str"""
```

Il est clair que la difficulté réside dans l'écriture de la première fonction **recherche**. Décomposons cette étape : en premier lieu, il s'agira de parcourir l'ensemble des configurations possibles, puis, pour chacune d'elle, de tester si elle satisfait les contraintes de l'énigme ou non. Ceci présuppose donc la création de deux nouvelles fonctions. La fonction **permutation** prendra en argument une liste L pour renvoyer la liste de toutes les listes possibles composées des éléments de L permutés. L'appel de cette fonction sera effectué sur chacune des 3 listes de critères définies à la question 1. A chaque tour de boucle (= à chaque création d'une nouvelle configuration), il s'agira d'exploiter une nouvelle combinaison de ces listes, et de tester si elle convient grâce à la fonction **verification**. Les signatures de ces deux dernières fonctions sont donc :

```
def permutation(L):
    """list[alpha]->list[list[alpha]] """

def verification(config):
    """list[maison]->bool"""
```

Question 3 : Liste des permutations d'une liste. Ecrire la fonction **permutation** dont la spécification est la suivante :

```
def permutation(L):
    """list[alpha]->list[list[alpha]]
    Hypothese : len(L)!=0
    retourne la liste composée de toutes les listes possibles crées
    à partir des éléments de L et dont les éléments ont été permutés.
    La liste de retour a donc pour longueur len(L)!"""
```

Par exemple :

```
>>>permutation([1,2,3])
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Correction : L'idée est de passer par une fonction intermédiaire qui retourne la liste de toutes les permutations possibles sur un ensemble d'entiers. Celle-ci s'exécute de manière récursive, c'est-à-dire en s'appelant elle-même un certain nombre de fois. Voici le code :

```
def modifierliste(L,i,n):
    """list[int]*int*int->list[int]
    hypo : i apparait exactement une fois dans la liste L, qui est au moins
    de longueur 1
    retourne la liste L ou l'on a remplace l'entier i (qui apparait)
    par l'entier n"""
    #LR:list[int]
    LR=[]
    #e : int
    for e in L:
        if e==i:
            LR.append(n)
        else:
            LR.append(e)
    return LR

#jeu de tests
assert modifierliste([1,2,3],3,5)==[1,2,5]


def permutation_entiers(n):
    """int->list[list[int]]
    hypo : n >=1
    retourne la liste des n! permutations possibles des entiers compris
    entre 0 et n-1. Une permutation est representee elle-meme par
    une liste d'entiers, celle des images."""
    if n==1:
        return [[0]]
```

```
    else:
        #L : list[list[int]]
        LR=[] # c'est la liste de retour que l'on complete pas à pas
        # i : int
        for i in range(0,n):
            #LLauxgrande : list[list[int]]
            LLauxgrande=permutation_entiers(n-1) #On travaille de manière
                                                 #récursive
            #j : int
            for j in range(0,len(LLauxgrande)):
                #Lauxpetite:list[int]
                Lauxpetite=modifierliste(LLauxgrande[j],i,n-1)
                       #Lauxgrande[j] est une petite liste, ie une seule
                       #permutation de n-1 eltts. On la modifie pour
                       #remplacer l'elt i par l'elt manquant, c'est à dire n
                LR.append([i]+Lauxpetite)
    return LR

#jeu de tests
assert permutation_entiers(1)==[[0]]
assert permutation_entiers(3)==[[0, 2, 1], [0, 1, 2], [1, 0, 2], \
                                        [1, 2, 0], [2, 0, 1], [2, 1, 0]]



def permutation(L):
    """list[alpha]->list[list[alpha]]
    hypothese : len(L)!=0
    retourne la liste composee de toutes les listes possibles crees
    a partir des elements de L et dont les elements ontete permutes.
    La liste de retour a donc pour longueur len(L)!"""
    #LL:list[list[int]]
    LL=permutation_entiers(len(L))
    #LLR : list[list[alpha]]
    LLR=[] # La liste resultat. On traduit simplement les entiers en str
    #liste:list[int]
    for liste in LL:
        #listealpha:list[alpha]
        listealpha=[]
        #entier:int
        for entier in liste:
            listealpha.append(L[entier])
        LLR.append(listealpha)
```

```
    return LLR

#jeu de tests
assert permutation(['lewis'])==[['lewis']]
assert permutation(["charles","lutwidge","dodgson"])== \
       [['charles', 'dodgson', 'lutwidge'], ['charles', 'lutwidge', 'dodgson'],\
        ['lutwidge', 'charles', 'dodgson'], ['lutwidge', 'dodgson', 'charles'], \
        ['dodgson', 'charles', 'lutwidge'], ['dodgson', 'lutwidge', 'charles']]
```

Question 4 : Traduction des contraintes. Ecrivez la fonction **verification** dont la spécification est la suivante :

```
def verification(config):
    """list[maison]->bool
    Hypothese : config est une liste de maison utilisant une seule et unique
    fois chaque critère (par exemple, il ne peut y avoir 2 ânes dans la rue).
    Retourne True si et seulement si config vérifie toutes les contraintes
    de l'enigme.
    """
```

Correction :

```
def verification(config):
    """list[maison]->bool
    Hypothese : config est une liste de maison utilisant une seule et unique
    fois chaque critère
    Retourne True si et seulement si config vérifie toutes les contraintes
    """
    #conditions classiques : 1,2,3,4,9 On deconstruit chaque maison
    #et on vérifie qu'il n'y a pas d'incohérence.
    #Remarque, 8 n'est pas une contrainte.
    #place:int
    place=-1 #pour retenir certains elts necessaires aux conditions spatiales
    for (n,p,a) in config:
        place=place+1
        if n=="anglais":
            if not (p=="sculpteur" or p=="medecin"):
                return False
        elif n=="espagnol":
            if not a=="chien":
                return False
        elif n=="islandais":
            if not p=="ingenieur":
```

```
                return False
        elif n=="norvegien":
            if p=="violoniste":
                return False
        else:
            #placeslovene:int
            placeslovene=place
                return False
        elif p=="sculpteur":
            if not a=="ane":
                return False
        elif p=="medecin":
            #placemedecin:int
            placemedecin=place
        elif p=="diplomate":
            #placediplomate:int
            placediplomate=place
        if a=="renard":
            #placerenard :int
            placerenard=place
        elif a=="cheval":
            #placecheval : int
            placecheval=place
    #conditions spatiales : 5,6,7
    (n,p,a)=config[0]
    if n!="norvegien": #5
        return False
    # Pour les 6 et 7 on retient les parametres
    #au moment du parcours des conditions non spatiales
    if placeislandais+1!=placeslovene and placeislandais-1!=placeslovene: #8
        return False
    if placemedecin+1!=placerenard and placemedecin-1!=placerenard: #6
        return False
    if placediplomate+1!=placecheval and placediplomate-1!=placecheval: #7
        return False

    ##Et si on est arrive jusque la... c'est que l'on a la bonne configuration !
    return True
```

Question 5 : Recherche exhaustive. Ecrivez la fonction **recherche** dont la spécification est :

```
def recherche():
```

```
    """->list[maison] + str
    Retourne une configuration qui satisfait toutes les conditions, si elle
    existe, et un message d'erreur sinon.
    """
```

Correction :

```
def recherche():
    """->list[maison] + str
    Retourne une configuration qui satisfait toutes les conditions, si elle
    existe, et un message d'erreur sinon.
    """
    #n::list[str]
    for n in permutation(nati):
        #p:list[str]
        for p in permutation(prof):
            #a:list[str]
            for a in permutation(anim):
                #config=list[maison]
                config=[]#la liste resultat possible
                #i : int. i indique la maison que l'on considere
                for i in range(0,5): #les maisons vont de 0 à 4
                    config.append((n[i],p[i],a[i]))
                    # on construit une configuration possible
                if verification(config):
                    #on teste si toutes les conditions sont realisees
                    return config
    return "Ton probleme est mal donne, il n'a pas de solution !"
```

Question 6 : Solution finale. Ecrivez la fonction **solution** sans paramètre qui répond à la question posée.... et lancez votre fonction ! Un mot à dire sur le temps d'exécution ?

Correction :

```
def solution():
    """->str
    Retourne la nationalité du propriétaire du zèbre.
    """
    #configuration : list[maison]
    configuration=recherche()
    for (n,p,a) in configuration:
        if a=="zebre":
            return n
```

Python nous dit que le propriétaire du zèbre est anglais ! En important la bibliothèque time :

```
import time
```

et en tapant

```
t0=time.clock()
solution()
print (time.clock() - t0, "seconds process time")
```

on obtient un temps d'une quinzaine de secondes environ.

Indices

- **Question 2** : Partez du principe que vous pouvez tester toutes les configurations possibles. A partir de là, il s'agit :
  1. de les créer.
  2. de les tester une par une.

  Lire les intitulés des questions suivantes peut aussi vous aider...
- **Question 3** : Passez par une fonction intermédiaire de spécification :

```
def permutation_entiers(n):
    """int->list[list[int]]
    Hypothese : n >=1
    Retourne la liste des n! permutations possibles des entiers
    compris entre 0 et n-1. Une permutation est representee
    elle-meme par une liste d'entiers, celle des images."""
```

  Que vous pourrez définir par récurrence, c'est-à-dire qu'elle peut "s'appeler elle-même". Pour construire cette dernière vous pouvez vous aider de la fonction (à implémenter) :

```
def modifierliste(L,i,n):
    """list[int]*int*int->list[int]
    Hypothese : i apparait exactement une fois dans la liste L,
    qui est au moins de longueur 1
    Retourne la liste L ou l'on a remplace l'entier i (qui apparait)
    par l'entier n"""
```

  Par exemple,

```
>>>modifierliste([1,2,3],3,5)
[1,2,5]
>>>permutation_entiers(3)
[[0, 2, 1], [0, 1, 2], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

- **Question 4** : Séparez les conditions spatiales des autres et traitez les une fois les conditions plus faciles étudiées.
- **Question 5** : Cette question est plutôt simple si vous vous servez correctement des (corrigés des) fonctions précédentes.