Devoir en ligne 2

April 18, 2020

1 Préambule

A des fins de vérification automatique, toutes les réponses, y compris les réponses numériques, sont données en syntaxe *python*.

Dans les zones de saisies *input* de couleur cyan clair, vous entrez les réponses sous la forme attendue. Ce sont des affectations de la forme

```
x = ...# expression éventuellement complexe
```

Lorsqu'il est demandé d'écrire une fonction, vous devez remplir une zone *textarea* de couleur beige. Vous écrivez vos codes dans votre environnement préféré (spyder, jupyter,...). Vous les copiez et les collez ensuite dans les zones réponses prévues. Les fonctions auxiliaires et l'importation de numpy, scipy, matplotlib, sympy sont autorisées mais il faut les copier dans le même textarea que la fonction. Donc chaque zone beige contient : les importations, les fonctions auxiliaires éventuelles, la fonction demandée par le sujet. Pour répondre à une question Q_{i+k}, vous pouvez utiliser la fonction d'une question Q_k précédente sans réécrire le code de cette fonction même si vous n'avez pas répondu à la question Q_k.

Lorsque cela paraît nécessaire, vous êtes autorisés à trier les listes avec la méthode *sort* et à faire usage du paramètre *key*. Cette fonction effectue des effets de bords. Les fonctions habituelles sur les listes sont autorisées.

Aucune commande d'affichage *print* n'est autorisée (mais vous pouvez et même devez faire des tests sur votre machine avant de répondre aux questions).

Ce devoir à la maison est parfois un peu ambitieux car il fait appel à la notion hors programme de dictionnaire. Allez ici pour en apprendre plus sur cette notion indispensable aux développeurs en Python.

Dans tout le devoir les *contenus* des tables sans clé primaire sont des listes de listes de la forme :

```
#Contenu de la table t1:
t1=[
[2, 'Robert', 'PCSI1'] ,
[4, 'Dupont', 'MPSI2'] ,
[6, 'Marie', 'PCSI2'] ,
[7, 'Marie', 'MPSI2'] ,
[10, 'Robert', 'PCSI2'] ,
[3, 'Marie', 'PCSI2'] ,
[9, 'Durant', 'PCSI1'] ,
```

```
[8, 'Durant', 'MPSI2'] ,
[5, 'Dupont', 'PCSI1'] ,
[1, 'Toto', 'PCSI1']
]
```

Il s'agit donc de listes d'enregistrements de même longueur. Les enregistrement sont des listes d'entiers, de flottants ou de chaînes de caractères à l'exclusion de tout autre type. Aucune hypothèse sur le caractère trié ou non n'est faite sur les tables.

Les schémas sont des listes de tuples *(nom, type)*. Par exemple, le schéma de t_1 est:

[23]:
```
s1=[("id",int),("eleve",str),("classe",str)]
```

Les clés sont données sous la forme de liste de numéros d'attributs. Par exemple [0] est visiblement une clé unique pour la table t_1 ci-dessus.

Dans le cas où une table possède une clé primaire, son contenu est stocké sous la forme d'un dictionnaire. Si la clé [0] de t_1 est primaire, alors le contenu de t_1 devient :

```
#stockage de t1 si la clé est [0]
{(2,): ['Robert', 'PCSI1'], (4,): ['Dupont', 'MPSI2'], (6,): ['Marie', 'PCSI2'], (7,): ['Marie
```

Les clés de ce dictionnaire sont des tuples et non des entiers car une clé contient potentiellement plusieurs numéros de colonne.

2 Q1 : contrôle de schéma

Ecrire la fonction

```
def check_type(e,s):
```

qui renvoie un bouléen qui indique si l'enregistrement e est compatible avec le schéma s.

On donne l'indication suivante, très utile ici :

[4]:
```
isinstance(2,str), isinstance(2,int), isinstance("rtt",str), isinstance(3.
↪2,float)
```

[4]:
```
(False, True, True, True)
```

[6]:
```
check_type([22, 'Marie', 'MPSI2'],s1)
```

[6]:
```
True
```

[7]:
```
check_type([2, 'Marie', 3.2],s1)
```

[7]:
```
False
```

3 Q2 : contrôle avant insertion

3.1 Q2a : avec clé unique

Les *clés* (unique) d'une table sont des listes de numéros d'attributs. Par exemple,

```
[0]
```

est une clé unique de la table t_1. Dans cette question on suppose que la clé n'est pas primaire et donc que le contenu est stocké sous la forme d'une liste de listes.

Ecrire la fonction

```
def possible_insertion_uk(e,t,s):
```

qui prend en paramètres une table t, un champs e (qu'on voudrait par exemple insérer dans t) et une clé unique s. La fonction indique par un bouléen s'il est possible d'insérer e dans la table t en respectant la clé. On ne demande pas de vérifier que c est une clé.

[96]:
```
print(t1)
```

```
[[2, 'Robert', 'PCSI1'], [4, 'Dupont', 'MPSI2'], [6, 'Marie', 'PCSI2'], [7,
'Marie', 'MPSI2'], [10, 'Robert', 'PCSI2'], [3, 'Marie', 'PCSI2'], [9, 'Durant',
'PCSI1'], [8, 'Durant', 'MPSI2'], [5, 'Dupont', 'PCSI1'], [1, 'Toto', 'PCSI1']]
```

[97]:
```
possible_insertion_uk([3, 'Marie', 'MPSI3'],t1,[0])
```

[97]:
```
False
```

[98]:
```
possible_insertion_uk([16, 'Marie', 'MPSI3'],t1,[0])
```

[98]:
```
True
```

3.2 Q2b : idem avec clé primaire

On suppose maintenant que la clé est primaire. Son contenu est donc stocké sous forme de dictionnaire comme indiqué en préambule. Ecrire la fonction

```
def possible_insertion_pk(e,t,s):
```

qui indique si on peut insérer l'enregistrement e dans le contenu de la table t lorsque la clé primaire est c. On ne demande pas de vérifier que c est une clé.

[37]:
```
print(t2)
```

```
{(2,): ['Robert', 'PCSI1'], (4,): ['Dupont', 'MPSI2'], (6,): ['Marie', 'PCSI2'],
(7,): ['Marie', 'MPSI2'], (10,): ['Robert', 'PCSI2'], (3,): ['Marie', 'PCSI2'],
(9,): ['Durant', 'PCSI1'], (8,): ['Durant', 'MPSI2'], (5,): ['Dupont', 'PCSI1'],
(1,): ['Toto', 'PCSI1']}
```

[38]:
```
possible_insertion_pk([3, 'Marie', 'MPSI3'],t2,[0])
```

[38]: False

[39]:
```
possible_insertion_pk([16, 'Marie', 'MPSI3'],t2,[0])
```

[39]: True

4 Q3 : Complexité temporelle

Dans cette question on fait l'hypothèse que la base de données ne contient que des entiers, des flottants ou des chaînes de caractères, tous de tailles bornées. On considère donc pour simplifier que les comparaisons entre entiers, flottants ou chaînes de caractères se font en $O(1)$.

Pour récupérer la valeur associée à une clé dans un dictionnaire, on effectue un *hachage* de la clé (voir ici pour les curieux). Cette opération est considérée comme ayant un coût en $O(1)$.

On note N le nombre d'enregistrements de la table t et K celle de son schéma. La longueur de la clé c n'est pas prise en compte car elle est inférieure à celle du schéma.

On note c_1 et c_2 les complexités respective des appels

```
possible_insertion_uk(e,t,c)
possible_insertion_pk(e,t,c)
```

L'ordre de grandeur des complexités (le "grand O" des mathématiciens) se note O en python. On demande d'écrire en syntaxe python les complexités c_1 et c_2.

Saisir c_1 en syntaxe python, par exemple :

```
c1 = O(N**2*K*3*exp(N+log(K)) #formule fausse, bien sûr
```

Saisir c_2 en syntaxe python, par exemple :

```
c2 = O(N**2*log(K)) #formule fausse, bien sûr
```

Pour les curieux, voici les importations et définitions que j'utilise pour contrôler vos réponses :

```
from sympy.series.order import Order
from sympy import log, exp, symbols, Function
N = sym.symbols("n", integer=True)
K = sym.symbols("k", integer=True)
O = Function('O')
```

5 Q4 : nature de la clé

On considère que toutes les tables ont une clé unique ou primaire.

Une table (pas seulement son contenu, avec lequel on a travaillé jusqu'à présent) est stockée sous forme de dictionnaire à trois entrées : contenu, schéma et clé. Une table T est enregistrée sous la forme suivante :

```
T = {"content": t, "schema":s, "key":c}
```

Si la clé est primaire, le contenu t est un dictionnaire comme à la question **Q2b**, sinon c'est une liste de liste comme à la question **Q2a**. Selon la nature du contenu, on est donc en mesure de déterminer s'il s'agit d'une clé primaire ou unique.

Ecrire la fonction

```
def primary_key(T)
```

qui détermine si la clé de T est primaire (True) ou unique (False).

[88]:
```
T1 = {}
T1["content"]=t1
T1["schema"]=s1
T1["key"]=[0]
T1
```

[88]:
```
{'content': [[2, 'Robert', 'PCSI1'],
  [4, 'Dupont', 'MPSI2'],
  [6, 'Marie', 'PCSI2'],
  [7, 'Marie', 'MPSI2'],
  [10, 'Robert', 'PCSI2'],
  [3, 'Marie', 'PCSI2'],
  [9, 'Durant', 'PCSI1'],
  [8, 'Durant', 'MPSI2'],
  [5, 'Dupont', 'PCSI1'],
  [1, 'Toto', 'PCSI1']],
 'schema': [('id', int), ('eleve', str), ('classe', str)],
 'key': [0]}
```

[89]:
```
primary_key(T1)
```

[89]:
```
False
```

[90]:
```
T2 = {}
T2["content"]=t2
T2["schema"]=s1
T2["key"]=[0]
T2
```

[90]:
```
{'content': {(2,): ['Robert', 'PCSI1'],
  (4,): ['Dupont', 'MPSI2'],
  (6,): ['Marie', 'PCSI2'],
  (7,): ['Marie', 'MPSI2'],
  (10,): ['Robert', 'PCSI2'],
```

```
 (3,): ['Marie', 'PCSI2'],
 (9,): ['Durant', 'PCSI1'],
 (8,): ['Durant', 'MPSI2'],
 (5,): ['Dupont', 'PCSI1'],
 (1,): ['Toto', 'PCSI1']},
'schema': [('id', int), ('eleve', str), ('classe', str)],
'key': [0]}
```

[91]:
```
primary_key(T2)
```

[91]:
```
True
```

6 Q5 : Insertion dans tous les cas

Ecrire la *procédure*

```
def insert(e,T)
```

qui insère l'enregistrement e dans la table T. On suppose que cette insertion ne contrarie pas la contrainte de clé, il est inutile de le vérifier. Il faut bien sûr gérer différement cette insertion suivant la nature du *contenu* de la table (donc dictionnaire ou liste de listes).

Cette procédure *insert* ne retourne rien (c'est une procédure !) mais modifie le contenu de la table T.

[102]:
```
insert([11,'Toto','PCSI2'],T1)
```

[104]:
```
T1["content"]
```

[104]:
```
[[2, 'Robert', 'PCSI1'],
 [4, 'Dupont', 'MPSI2'],
 [6, 'Marie', 'PCSI2'],
 [7, 'Marie', 'MPSI2'],
 [10, 'Robert', 'PCSI2'],
 [3, 'Marie', 'PCSI2'],
 [9, 'Durant', 'PCSI1'],
 [8, 'Durant', 'MPSI2'],
 [5, 'Dupont', 'PCSI1'],
 [1, 'Toto', 'PCSI1'],
 [11, 'Toto', 'PCSI2']]
```

[107]:
```
insert([11,'Toto','PCSI2'],T2)
T2["content"]
```

[107]:
```
{(2,): ['Robert', 'PCSI1'],
 (4,): ['Dupont', 'MPSI2'],
 (6,): ['Marie', 'PCSI2'],
```

```
 (7,): ['Marie', 'MPSI2'],
 (10,): ['Robert', 'PCSI2'],
 (3,): ['Marie', 'PCSI2'],
 (9,): ['Durant', 'PCSI1'],
 (8,): ['Durant', 'MPSI2'],
 (5,): ['Dupont', 'PCSI1'],
 (1,): ['Toto', 'PCSI1'],
 (11,): ['Toto', 'PCSI2']}
```

7 Q6 : changement de statut de la clé

L'administrateur de la base de données se rend compte que les utilisateurs accèdent au contenu d'une de ses tables en utilisant sa clé unique préférentiellement. Il serait donc plus rentable de transformer cette clé en clé primaire, l'accès aux enregistrements serait alors plus rapide.

Ecrire la fonction:

```
def unique2primary(T):
```

qui prend en paramètre une table ayant une clé unique et retourne la même table mais stockée en prenant comme index cette clé. La table T n'est pas modifiée, on retourne bien une nouvelle table, en particulier le schéma et la clé de cette nouvelle table ne sont pas enregistrés aux mêmes adresses que leurs homologues de la table initiale.

[116]:
```
T1
```

[116]:
```
{'content': [[2, 'Robert', 'PCSI1'],
  [4, 'Dupont', 'MPSI2'],
  [6, 'Marie', 'PCSI2'],
  [7, 'Marie', 'MPSI2'],
  [10, 'Robert', 'PCSI2'],
  [3, 'Marie', 'PCSI2'],
  [9, 'Durant', 'PCSI1'],
  [8, 'Durant', 'MPSI2'],
  [5, 'Dupont', 'PCSI1'],
  [1, 'Toto', 'PCSI1'],
  [11, 'Toto', 'PCSI2']],
 'schema': [('id', int), ('eleve', str), ('classe', str)],
 'key': [0]}
```

[119]:
```
T_ = unique2primary(T1)
T_
```

[119]:
```
{'schema': [('id', int), ('eleve', str), ('classe', str)],
 'key': [0],
 'content': {(2,): ['Robert', 'PCSI1'],
  (4,): ['Dupont', 'MPSI2'],
```

```
(6,): ['Marie', 'PCSI2'],
(7,): ['Marie', 'MPSI2'],
(10,): ['Robert', 'PCSI2'],
(3,): ['Marie', 'PCSI2'],
(9,): ['Durant', 'PCSI1'],
(8,): ['Durant', 'MPSI2'],
(5,): ['Dupont', 'PCSI1'],
(1,): ['Toto', 'PCSI1'],
(11,): ['Toto', 'PCSI2']}}
```

[125]:
```
id(T_["key"])==id(T1["key"]), id(T_["schema"])==id(T1["schema"]),␣
↪T1["content"][:2]
```

[125]:
```
(False, False, [[2, 'Robert', 'PCSI1'], [4, 'Dupont', 'MPSI2']])
```

8 Q7 : Clé étrangère

Dans cette section, les tables n'ont pas de clé primaire, seulement des clés uniques. Elles peuvent aussi avoir une quatrième association, laquelle désigne une éventuelle contrainte de clé étrangère.

La base de données est une liste de tables.

Considérons par exemple la table *note* suivante :

Certains élèves ont plusieurs notes, d'autre une seule ou aucune. Par exemple, l'élève d'identifiant 1 n'a pas de note.

La base de données elle-même est la liste :

[199]:
```
db = [T1,Note]
print(T1)
```

```
{'content': [[2, 'Robert', 'PCSI1'], [4, 'Dupont', 'MPSI2'], [6, 'Marie',
'PCSI2'], [7, 'Marie', 'MPSI2'], [10, 'Robert', 'PCSI2'], [3, 'Marie', 'PCSI2'],
[9, 'Durant', 'PCSI1'], [8, 'Durant', 'MPSI2'], [5, 'Dupont', 'PCSI1'], [1,
'Toto', 'PCSI1'], [11, 'Toto', 'PCSI2']], 'schema': [('id', <class 'int'>),
('eleve', <class 'str'>), ('classe', <class 'str'>)], 'key': [0]}
```

L'association

```
fk:(0,[[1],[0]])
```

dans la table *Note* se comprend ainsi : la clé étrangère ([2]) de *Note* référence la clé primaire de la table 0, c'est à dire la colonne [0] de la table *T1*. Cela signifie que la colonne *eleve* de *Note* ne peut pas contenir de valeur qui ne soit pas déjà présente dans la colonne *id* de *T1*.

Il y a donc deux choses à vérifier, ce qui nous fait deux questions.

8.1 Q7a : format compatible

Ecrire la fonction

```
def fk_compat(T,db):
```

Elle prend en paramètres une table T et la base de données db dans laquelle elle est contenue (on ne demande pas de vérifier ce point). Si la table T n'a pas de contrainte de clé étrangère, la fonction retourne *True*. Si il y a une clé étrangère, elle est de la forme :

```
"fk":(i,c)
```

où c est une liste de colonnes de T et i le numéro de la table de la base db que référence la clé étrangère. La fonction vérifie que la clé unique de la table numéro i est bien de la même longueur que la clé étrangère. Si ce point est vérifié, la fonction inspecte le schéma des deux tables pour savoir si les attributs des deux clés ont des domaines compatibles. Dans ce cas, elle retourne *True*, sinon *False*.

[171]:
```
fk_compat(Note,db)
```

[171]:
```
True
```

[172]:
```
TT0 = {'content': [], 'key': [0, 3],\
      'schema': [('c1', int), ('c2', float), ('c3', int), ('c4',str)]}

TT1 = {'content': [], 'key': [0],\
      'schema': [('c1', int), ('c2', float), ('c3', int)],\
       'fk': (0, [0,1,2])}

TT2 = {'content': [], 'key': [0, 1,2],\
      'schema': [('c2', float), ('c3', int), ('c4',str)],\
       'fk': (0, [1,2])}

TT3 = {'content': [], 'key': [0, 1,2],\
      'schema': [('c2', float), ('c3', int), ('c4',float)],\
       'fk': (0, [1,2])}

db2 = [TT0,TT1,TT2,TT3]
```

[173]:
```
fk_compat(TT0,db2)
```

[173]:
```
True
```

[174]:
```
fk_compat(TT1,db2)
```

[174]:
```
False
```

[175]:
```
fk_compat(TT2,db2)
```

[175]:
```
True
```

[176]:
```
fk_compat(TT3,db2)
```

[176]:
```
False
```

8.2 Q7b : Insertion possible

On rappelle que les tables dans cette section n'ont pas de clé primaire, seulement une clé unique.

Ecrire la fonction

```
def inserer_fk(e,T,db):
```

Elle vérifie qu'on peut insérer l'enregistrement e dans la table T en respectant la contrainte de clé étrangère de T (on suppose que T en a une) et aussi, bien sûr, la contrainte de clé unique. On dispose pour ces vérifications de la base de données *db*. Il n'y a en revanche pas besoin de tester la compatibilité de la clé unique de T avec la clé primaire de la table vers laquelle elle pointe : on suppose que c'est acquis.

[225]:
```
db=[T1,Note]
T1
```

[225]:
```
{'content': [[2, 'Robert', 'PCSI1'],
  [4, 'Dupont', 'MPSI2'],
  [6, 'Marie', 'PCSI2'],
  [7, 'Marie', 'MPSI2'],
  [10, 'Robert', 'PCSI2'],
  [3, 'Marie', 'PCSI2'],
  [9, 'Durant', 'PCSI1'],
  [8, 'Durant', 'MPSI2'],
  [5, 'Dupont', 'PCSI1'],
  [1, 'Toto', 'PCSI1'],
  [11, 'Toto', 'PCSI2']],
 'schema': [('id', int), ('eleve', str), ('classe', str)],
 'key': [0]}
```

[226]:
```
Note
```

[226]:
```
{'content': [[1, 12.4, 2],
  [2, 4.2, 2],
  [3, 19.2, 2],
  [4, 17.0, 6],
  [5, 10.9, 10],
  [6, 7.8, 10],
  [7, 8.7, 10],
  [8, 1.4, 9],
  [9, 4.5, 9],
  [10, 19.8, 9]],
 'key': [0],
```

```
 'schema': [('id', int), ('resultat', float), ('eleve', int)],
 'fk': (0, [2])}
```

[227]:
```
inserer_fk([9, 13.4, 2],Note,db)
```

[227]: False

[230]:
```
"""
2 est une valeur de T1[0]
et 11 n'est pas une valeur de Note[0]
"""
inserer_fk([11, 13.4, 2],Note,db)
```

[230]: True

[229]:
```
inserer_fk([11, 13.4, 12],Note,db)#12 n'est pas une valeur de T1[0]
```

[229]: False