Amérique du nord - mars 2023 - sujet 1

Exercice 1 (Données en table et bases de données - 5 points)
Cet exercice est constitué de deux parties indépendantes.

Un étudiant souhaite développer une application permettant de faciliter le covoiturage pour les déplacements du quotidien. Dans cet objectif, il étudie des données extraites de la Base Nationale des Lieux de Covoiturage (BNLC), disponible sur le site `data.gouv.fr`.
Dans un premier temps (partie A), les données d'une table décrivant des lieux de covoiturage (adresse postale, nombre de places, ...) sont manipulées a l'aide d'un tableau contenant des dictionnaires en langage Python.
Dans un second temps (partie B), une base de données contenant deux tables (les sites de covoiturage et les caractéristiques des communes de France) est exploitée à l'aide du langage SQL.

Partie A : traitement de données en table
Une table est implémentée par un tableau nommé `tab_lieux` contenant des dictionnaires en langage Python. Chaque dictionnaire correspond à un lieu de stationnement pour le covoiturage. Les clés des dictionnaires, communes à tous les dictionnaires, correspondent aux descripteurs utilisés pour cette table :

- `id_lieu` : identifiant du lieu, la donnée est un entier (chaque lieu possède un identifiant unique) ;
- `ad_lieu` : adresse du lieu, la donnée est une chaîne de caractères ;
- `insee` : code INSEE de la commune où se trouve le lieu, la donnée est une chaîne de caractères ;
- `nb_places` : nombre de places du lieu, la donnée est un entier ;
- `type` : nature du parking (supermarché, parking municipal, aire de stationnement située en sortie d'autoroute, ...), la donnée est une chaîne de caractères.

On donne en illustration les trois premiers éléments de ce tableau de dictionnaires :

```
tab_lieux = [{"id_lieu": 1, "ad_lieu": "Place De La Fontaine", "insee": "1024",
              "nb_places": 5, "type": "Supermarché"},
             {"id_lieu": 2, "ad_lieu": "La Boisse", "insee": "1049",
              "nb_places": 100, "type": "Parking municipal"},
             {"id_lieu": 3, "ad_lieu": "Chateau-Gaillard", "insee": "1089",
              "nb_places": 15, "type": "Sortie autoroute"},
             ... ]
```

1. Accès aux informations du tableau :
   (a) Donner, sans justifier, la valeur à laquelle on accède avec l'instruction `tab_lieux[0]["insee"]`.
   (b) Ecrire l'instruction qui permet d'obtenir la valeur `"La Boisse"`.
2. On propose trois blocs d'instructions pour parcourir le tableau et afficher le nombre de places des lieux de covoiturage. Parmi ces trois propositions, deux seulement sont correctes. Les indiquer sans justifier.
   - Proposition 1 :

```
for i in range(len(tab_lieux)):
    print(dico["nb_places"])
```

   - Proposition 2 :

```
for i in range(len(tab_lieux)):
    print(tab lieux[i]["nb_places"])
```

   - Proposition 3 :

```
for dico in tab_lieux:
    print(dico["nb_places"])
```

3. On dispose de la fonction ci-dessous :

```
def fonction_inconnue(table, n, nom_type):
    '''
    Entrée : un tableau de dictionnaires, un entier, une chaîne de caractères
    Sortie : un entier
    '''
    k = 0
    for dico in table:
        if dico["nb_places"] >= n and dico["type"] == nom_type:
            k = k + 1
    return k
```

Décrire, dans le contexte de l'exercice, ce que renvoie cette fonction, lors de l'appel :

```
fonction_inconnue(tab_lieux, 100, "Sortie autoroute")
```

4. La fonction, dont le code est proposé ci-après, permet de renvoyer le code INSEE du lieu de covoiturage possédant le plus grand nombre de places. Recopier et compléter ce code.

```
def insee_max_places(table):
    '''
    Entrée : un tableau de dictionnaires
    Sortie : une chaîne de caractères (le code INSEE du lieu
    possédant le plus grand nombre de places)
    '''
    maxi = .....
    code_insee = .....
    for dico in table:
        if ..... :
            maxi = ......
            code_insee = .....
    return code_insee
```

5. La fonction dont les spécifications sont données ci-après, prend en paramètres une table de lieux de covoiturage au format tableau contenant des dictionnaires et le nom d'un type de lieu de covoiturage. Recopier et compléter le code de cette fonction. Dans le code de la fonction, les pointillées peuvent correspondre à une ou plusieurs lignes de programme.

```
def moyenne_par_type(table, nom_type):
    '''
    Entrée : un tableau de dictionnaires et une chaîne de caractères (type de lieu)
    Cette fonction renvoie, parmi les lieux de covoiturage dont le type est nom_type,
    la moyenne du nombre de places pour le type choisi.
    Sortie : un flottant
    '''
    .....
    return moyenne
```

Exemple avec la table `tab1` ci-après contenant uniquement trois enregistrements :

```
tab1 = [{"id_lieu": 1, "ad_lieu": "Place Du marché", "insee": "1032",
         "nb_place": 10, "type": "Supermarché"},
        {"id_lieu": 2, "ad_lieu": "La Pesse", "insee": "1058",
         "nb_places": 100, "type": "Parking municipal"},
        {"id_lieu": 3, "ad_lieu": "Le Pautet", "insee": "1075",
         "nb_places": 20, "type": "Supermarché"}]
```

L'instruction `moyenne_par_type(tab1, "Supermarché")` renvoie `15`.

Partie B : base de données

On pourra utiliser les mots du langage SQL suivants : `SELECT`, `FROM`, `WHERE`, `JOIN`, `INSERT INTO`, `VALUES`, `COUNT`, `UPDATE`, `SET`.

Afin de pouvoir gérer un site de covoiturage, on utilise une base de données contenant les relations `LIEU` et `COMMUNE`.

Le schéma relationnel, où les clés primaires sont soulignées et les clés étrangères sont précédées du symbole `#`, est le suivant :

`LIEU(id_lieu:entier, ad_lieu:texte, #code_insee:texte, nb_places:entier, type:texte)`

`COMMUNE(code_insee:texte, nom:texte, departement:texte, region:texte, latitude:décimal, longitude:décimal)`

On rappelle qu'en langage SQL la fonction d'agrégation `COUNT` permet de compter un nombre d'enregistrements. Par exemple, pour déterminer le nombre de lieux dans la table `LIEU`, on peut utiliser la requête suivante :

```
SELECT COUNT(id_lieu) FROM LIEU;
```

On donne un extrait des deux premières lignes de ces relations :

`LIEU`

id_lieu	ad_lieu	code_insee	nb_places	type
1	"Place De La Fontaine"	"01001"	5	"Supermarché"
2	"La Boisse"	"01049"	100	"Parking municipal"

`COMMUNE`

code_insee	nom	departement	region	latitude	longitude
"01001"	"L'ABERGEMENT-CLEMENCIAT"	"AIN"	"RHONE-ALPES"	46.1534	4.9261
"01002"	"L'ABERGEMENT-DE-VAREY"	"AIN"	"RHONE-ALPES"	46.0092	5.4280

6. On se place dans la relation `COMMUNE`. Expliquer pourquoi deux communes ne peuvent pas posséder le même code INSEE `code_insee`.
7. Ecrire une requête SQL permettant d'obtenir l'identifiant `id_lieu` et le code INSEE `code_insee` des lieux de covoiturage de type `"Sortie autoroute"`.
8. Ecrire une requête SQL permettant de compter le nombre de lieux de covoiturage se trouvant dans le département `"JURA"`.
9. La commune de code INSEE `"40714"` vient d'être intégrée à la commune voisine `"40146"`. En conséquence, il faut mettre à jour la base de données.
   (a) La requête suivante a été saisie. Une erreur a été renvoyée par le logiciel. Expliquer pourquoi.

```
DELETE FROM COMMUNE WHERE code_insee = "40714"
```

   (b) Les informations liées à la commune de code INSEE `"40714"` devront désormais être rattachées à la commune de code INSEE `"40146"`. Ecrire une requête permettant de mettre jour la table `LIEU`.

Exercice 2 (Processus et logique booléenne - 3 points)
Cet exercice est constitué de trois parties indépendantes.

Partie A : processus
La ligne de commande `ps` tapée dans un terminal permet d'avoir la liste des processus du système.
La commande `ps -eo user, pid, ppid, time, cmd` permet d'afficher pour tous les processus du système les colonnes suivantes :

`USER` : le nom de l'utilisateur qui exécute le processus ;

`PID` : l'identifiant du processus ;

`PPID` : l'identifiant du processus parent ;

`TIME` : le temps d'utilisation du processeur par le processus ;

`CMD` : la commande ou l'application à l'origine de la création du processus.

Voici un extrait de l'écran d'un terminal après exécution de la commande `ps -eo user, pid, ppid, time, cmd`

```
USER       PID  PPID  TIME      CMD
root         1     0  00:00:02  /sbin/init
kernoops  1567     1  00:00:00  /usr/sbin/kerneloops
user01    1611     1  00:00:00  /lib/systemd/systemd --user
user01    1752  1611  00:00:00  /usr/libexec/gnome-session-binary --systemd-
user01    1766  1752  00:00:00  /usr/libexec/at-spi-bus-launcher --launch-im
user01    1770  1611  00:00:51  /usr/bin/gnome-shell
user01    5151  1632  00:00:00  /usr/libexec/gvfsd-network --spawner :1.2 /o
user01    5168  1632  00:00:00  /usr/libexec/gvfsd-dnssd --sp
user01    5348  1770  00:00:01  /snap/vlc/2344/usr/bin/vlc
user01    5468  1770  00:00:01  /snap/gimp/393/usc/bin/gimp
user01    5564  1611  00:00:00  /usr/bin/python3 /usr/bin/gnome-terminal --w
user01    5566  5564  00:00:00  /usr/bin/gnome-terminal .real --wait
user01    5571  1611  00:00:00  /usr/libexec/gnome-terminal-server
user01    5589  5571  00:00:00  bash
user01    5596  5589  00:00:01  /usr/bin/python3 /home/user0l/.local/bin/tho
user01    5603  5596  00:00:00  /usr/bin/python3 -u -B -m thonny.plugins.cpy
user01    5615  1611  00:00:00  /usr/bin/python3 /usr/bin/gnome-terminal --w
user01    5617  5615  00:00:00  /usr/bin/gnome-terminal .real --wait
user01    5622  5571  00:00:00  bash
user01    5629  5622  00:00:00  ps -eo user, pid, ppid, time, cmd
```

En utilisant les données de l'extrait ci-avant, répondre aux questions suivantes :

1. Déterminer le nom de l'application associée au processus dont l'identifiant est `5468`.
2. Déterminer l'identifiant du processus qui a sollicité le processeur le plus longtemps.
3. Déterminer l'identifiant du processus qui a le plus d'enfants.
4. Déterminer le nom du programme dont est issue la commande `ps`.
5. Déterminer la succession des identifiants des processus qui ont permis de générer le processus associé à la ligne de commande : `ps -eo user, pid, ppid, time, cmd` en partant du processus initial dont le PID est `1`.

Partie B : ordonnancement

Dans cet exercice, étant donné un instant initial noté $t_0 = 0$, on dit qu'un processus est caractérisé par :

- sa durée d'exécution, exprimée en unités de temps ;
- son instant d'arrivée, défini à partir de t_0, correspondant à l'instant où le processus est créé par le système d'exploitation.

6. Les trois états principaux d'un processus sont « prêt », « bloqué » et « élu ». Donner la définition de chacun de ces trois états.

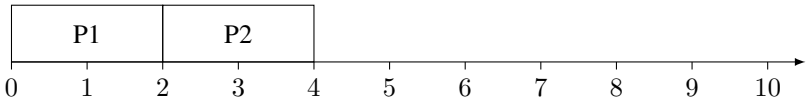
On considère un système d'exploitation qui utilise un ordonnancement par tourniquet pour gérer les processus. Dans cet ordonnancement, le processus élu dispose alors d'un temps donné prédéfini appelé quantum, et s'exécute :

- soit jusqu'à ce qu'il soit terminé (durée d'exécution restante inférieure ou égale au quantum) ;
- soit pendant la durée du quantum (il retourne ensuite à l'état « prêt » et réintègre la file d'attente de l'ordonnanceur en dernière position) ;
- soit jusqu'à ce qu'il se bloque de lui-méme en raison d'une ressource indisponible.

7. On s'intéresse à la situation suivante mettant en jeu trois processus avec un quantum fixé à 2 unités de temps.

Nom du processus	Temps d'exécution	Instant d'arrivée
P1	5	0
P2	3	1
P3	4	5

Recopier et compléter le chronogramme d'exécution des processus.



Partie C : logique booléenne

On donne ci-dessous les tables de vérité des opérateurs `NON`, `ET` et `OU`, dans lesquelles les variables `a` et `b` booléennes prennent les valeurs `0` pour `FAUX` et `1` pour `VRAI`.

NON	
a	NON a
0	1
1	0

ET		
a	b	a ET b
0	0	0
0	1	0
1	0	0
1	1	1

OU		
a	b	a OU b
0	0	0
0	1	1
1	0	1
1	1	1

8. Déterminer la table de vérité de l'expression booléenne `NON(a ET b)`.
9. A l'aide d'une seconde table de vérité, justifier que : `NON(a ET b) = (NON a) OU (NON b)`

Exercice 3 (POO et ABR - 4 points)
Cet exercice contient deux parties indépendantes.

Lors de la création d'un jeu vidéo, un développeur décide d'utiliser la programmation orientée objet. Au cours de ce jeu, différents personnages vont s'affronter tour a tour. Pour cela, le développeur décide de créer une classe `Personnage`.

Un personnage est caractérisé par les données suivantes :

- `clan` : chaîne de caractères qui identifie le clan auquel le personnage appartient ;
- `vie` : nombre entier qui représente le nombre de points de vie du personnage. Un personnage n'est plus actif dés que le nombre de points de vie devient inférieur ou égal à zéro ;
- `force` : nombre entier qui représente le nombre de points de force.

Les personnages peuvent effectuer différentes actions : attaquer un ennemi, se défendre face à un ennemi, etc.

Partie A : programmation orientée objet

1. Recopier et compléter les lignes de code numérotées de 2 à 5 :

```
1  class Personnage:
2      def __init__(..., nom_clan, pts_vie, pts_force):
3          ...clan = nom_clan
4          ...vie = pts_vie
5          ...force = pts_force
6
7      def attaque(...):
8          ...
9
10     def defense(...):
11         ...
```

2. Donner les attributs ainsi que les méthodes de cette classe.
3. Ecrire l'instruction qui permet d'instancier le personnage `luther` du clan `"Umbrella"` avec 100 points de vie et 15 points de force.
4. Ecrire le code Python de la méthode `attaque` correspondant à la description suivante :
   - elle s'appligue sur un personnage (l'attaquant) et prend en paramètre `autre_perso` (le personnage attaqué) ;
   - elle abaisse le nombre de points de vie du personnage attaqué d'une valeur égale au nombre de points de force du personnage qui attaque ;
   - elle renvoie 1 s'il reste des points de vie au personnage attaqué, 0 sinon.

   Exemple : si le personnage `luther` qui a 15 points de force attaque le personnage `rayleigh` qui a 120 points de vie alors l'appel `luther.attaque(rayleigh)` permet de mettre à jour le nombre de points de vie de `rayleigh`, soit $120-15=105$ points de vie. La méthode renvoie 1 dans ce cas puisqu'il reste des points de vie à `rayleigh`.
5. Le développeur souhaite garder en mémoire toutes les actions réalisées par un personnage afin de pouvoir les annuler les unes après les autres et ainsi revenir à un état antérieur du personnage. Indiquer sans la justifier la structure de données la plus pertinente à utiliser pour garder en mémoire les différentes actions du personnage.

Partie B : arbres binaires de recherche

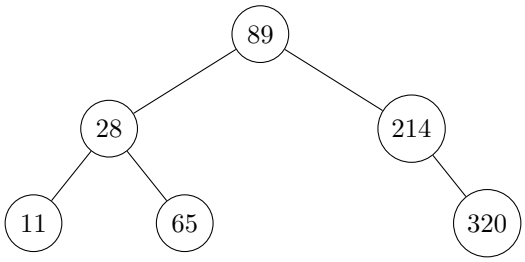
Dans cette partie :

- les arbres binaires de recherche ne peuvent pas contenir plusieurs fois la même clé ;
- un arbre binaire de recherche limité à un nœud a une hauteur égale à 1 ;
- la taille d'un arbre est le nombre de nœuds de cet arbre ;
- la valeur d'un nœud (aussi appelée clé) est :
  - strictement supérieure à celle des nœuds de son sous-arbre gauche ;
  - strictement inférieure à celle des nœuds de son sous-arbre droit.
- la racine d'un arbre binaire est le seul nœud n'ayant pas de parent.

On dispose de l'interface suivante pour manipuler les arbres binaires de recherche :

- `est_vide` : arbre → booléen : renvoie vrai si l'arbre est vide, faux sinon ;
- `gauche` : arbre → arbre : renvoie le sous-arbre gauche ;
- `droit` : arbre → arbre : renvoie le sous-arbre droit ;
- `valeur_racine` : arbre → valeur : renvoie la valeur de la racine de l'arbre.

6. On considére l'arbre binaire de recherche ci-dessous :



   (a) Donner sans justification la hauteur et la taille de l'arbre.

   (b) On insère dans l'arbre binaire de recherche ci-avant la clé dont la valeur est 46. Représenter l'arbre obtenu après cette insertion.

7. . On propose la fonction `parcours` ci-dessous, en langage Python :

```
1 def parcours{arbre):
2     if not est_vide(arbre):
3         parcours(gauche(arbre))
4         print(valeur_racine(arbre))
5         parcours(droit(arbre))
```

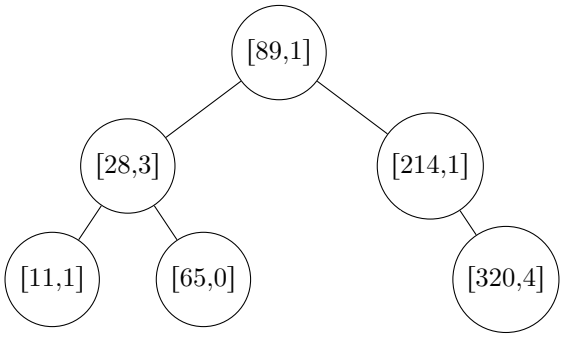
   (a) Donner le nom de ce parcours.

   (b) En considérant l'arbre donné en exemple (sans l'ajout de la valeur 46), indiquer l'ordre dans lequel les valeurs des nœuds seront affichés.

8. Pour le jeu vidéo défini dans la partie A, on souhaite collecter l'équipement d'un personnage (vêtements, armes, nourriture, etc.) en utilisant un arbre binaire de recherche. Chaque type de matériel est identifié par un nombre entier. D'autre part, un personnage peut disposer d'un même matériel en plusieurs exemplaires : il peut, par exemple, posséder douze flèches, toutes de même type (et possédant donc la même référence).
   Dans cette question, les clés sont des tableaux non dynamiques contenant deux informations :

   - un premier entier (la référence du matériel) ;
   - un second entier (la quantité correspondante pour ce matériel).

   Le placement dans l'arbre binaire de recherche se fait en fonction de la référence du matériel (premier élément du tableau).
   L'arbre donné précédemment peut devenir par exemple :



   Si `cle` vaut `[28, 3]`, alors `cle[0]` vaut `28` et `cle[1]` vaut `3`.

   La fonction `recherche_dichotomique` ci-dessous prend en paramètre un arbre binaire de recherche et un entier (référence de matériel) et renvoie la quantité associée au matériel si celui-ci est présent dans l'arbre, `-1` sinon. Recopier et compléter cette fonction :

```
def recherche_dichotomique(arbre, ref_materiel):
    '''
    Entrée : un arbre binaire de recherche dont les noeuds sont des tableaux ;
             un entier (la référence du matériel recherché).
    Sortie : un entier : -1 si la référence du matériel n'est pas présente dans
             l'arbre ou la quantité si la référence est présente.
    '''
    if est_vide(arbre): # référence absente de l'arbre
        return .....
    elif ref_materiel < valeur_racine(arbre)[.....]: # recherche
        return ..... # dans le sous-arbre gauche
    elif ref_materiel > valeur_racine(arbre)[.....]: # recherche
        return ..... # dans le sous-arbre droit
    else: # référence présente, on renvoie la quantité correspondante
        return valeur_racine(arbre)[.....]
```