Feuille exercices de bac – Bdd

Exercice 1 : *Cet exercice porte sur la programmation Python, la programmation orientée objet, les bases de données relationnelles et les requêtes SQL.*

L'objectif est de faciliter la gestion du système d'information d'un camping municipal. Les informations nécessaires sont stockées dans une base de données relationnelle composée de trois relations. On pourra utiliser les mots-clés SQL suivants : `AND`, `FROM`, `INSERT INTO`, `JOIN`, `ON`, `SELECT`, `SET`, `UPDATE`, `VALUES`, `WHERE`.

Voici le schéma des deux premières relations :

```
Client ( id_client, nom, prenom, adresse, ville, pays, telephone)
Reservation ( id_reservation , #id_client, #id_emplacement, nombre_personne,
            date_arrivee, date_depart)
```

Dans ce schéma :

- la clé primaire de chaque relation est définie par son attribut souligné ;
- les attributs précédés de # sont les clés étrangères.

La troisième relation est appelée `Emplacement` et elle contient tous les emplacements du camping. Le tableau ci-dessous en donne un extrait.

Emplacement			
id_emplacement	nom	localisation	tarif_journalier
1	myrtille	A4	25
2	mirabelle	D1	35
3	mangue	B2	29.90
4	mandarine	B1	25
5	mûre	C3	29.90
6	melon	A2	25

Partie A

1) Citer deux avantages à utiliser une base de données relationnelle plutôt qu'un fichier texte ou un fichier tableur.
2) Quelle doit être la caractéristique d'un attribut pour pouvoir être utilisé en tant que clé primaire ?
3) Dans la relation `Reservation`, quel est le rôle des clés étrangères `id_client` et `id_emplacement` ?
4) Donner le schéma relationnel de la relation `Emplacement` en précisant la clé primaire et le type de chacun des attributs.
5) À partir de l'extrait du contenu de la relation `Emplacement` donner le résultat de la requête ci-contre :

```
SELECT id_emplacement, nom, localisation
FROM Emplacement
WHERE tarif_journalier = 25;
```

6) Écrire une requête permettant de donner le `nom` et le `prenom` de tous les clients habitant à `'Strasbourg'`.
7) Écrire une requête permettant d'ajouter un nouveau client :
   - id_client `42` ;
   - nom `'CODD'` ;
   - prénom `'Edgar'` ;
   - numéro de téléphone `'0555555555'`.
   - adresse `'28 rue des Capucines'` ;
   - ville `'Lyon'` ;
   - pays `'France'` ;
8) Écrire une requête SQL permettant de récupérer les informations ci-dessous concernant la réservation dont l'identifiant `id_reservation` est 18 :

- `Client.nom`
- `Client.prenom`
- `Reservation.nombre_personne`
- `Reservation.date_arrivee`
- `Reservation.date_depart`
- `Emplacement.tarif_journalier`

Partie B

Dans cette partie, on souhaite éditer une facture correspondant au séjour d'un client. Pour cela, on dispose d'une fonction Python qui récupère auprès de la base de données, à la manière de la question 8, les informations concernant la réservation voulue et renvoie le résultat sous forme d'un tuple contenant trois objets respectivement des classes `Client`, `Reservation` et `Emplacement`.

```
from datetime import datetime

class Client:
    def __init__(self, nom, prenom, adresse, ville, pays, telephone):
        self.nom = nom
        self.prenom = prenom
        self.adresse = adresse
        self.ville = ville
        self.pays = pays
        self.telephone = telephone

class Reservation:
    def __init__(self, id_reservation, nombre_personne,
                 date_arrivee, date_depart):
        self.id_reservation = id_reservation
        self.nombre_personne = nombre_personne
        self.date_arrivee = date_arrivee
        self.date_depart = date_depart

    def nb_jours(self):
        """ renvoie, à l'aide de l'attribut days de la classe
            timedelta, un entier correspondant
            au nombre de jours passés au camping."""
        return (self.date_depart - self.date_arrivee).days

class Emplacement:
    def __init__(self, nom, tarif_journalier):
        self.nom = nom
        self.tarif_journalier = tarif_journalier
```

9) Expliquer pourquoi le terme **self** est utilisé comme paramètre pour les méthodes des classes `Client`, `Reservation` et `Emplacement`.

10) Instancier une variable `client01` de la classe `Client` représentant un client se nommant CODD Edgar habitant au 28 rue des Capucines à Lyon, France, ayant pour numéro de téléphone le 0555555555.

On considère un tuple constitué de trois objets, respectivement dans cet ordre, des classes `Client`, `Reservation` et `Emplacement`. On souhaite écrire une fonction qui renvoie le montant dû par ce client pour cet emplacement et pour cette durée de séjour. Sachant qu'au tarif journalier de location de l'emplacement il faut ajouter une taxe de séjour de 2,20 € par jour et par personne.

Exemple de calcul du montant à régler pour un client ayant réservé pour 4 personnes pendant 12 jours un emplacement à 30 € la journée :

```
>>> 30 * 12 + 4 * 2.20 * 12
465.6
```

11) Compléter la dernière ligne de la fonction `montant_a_regler`.

```
def montant_a_regler(triplet):
    """ renvoie le montant en euros à régler pour cette réservation """
    client, reservation, emplacement = triplet
    return ...
```

Chaque facture doit posséder ce que l'on appelle communément un numéro de facture unique. En réalité il s'agit d'une chaîne de caractères. Pour ses factures, depuis 2018, le camping a adopté le format `'AAAA-MMM-xxx'` composé des trois chaînes de caractères ci-dessous :

- `'AAAA'` une année comprise entre 2018 et 2024 ;
- `'MMM'` les trois premières lettres du mois en anglais ;
- `'xxx'` désigne trois chiffres.

On décide d'écrire une fonction `facture_est_valide` pour tester si une chaîne de caractères représente un numéro de facture valide ou non. Voici quelques exemples du comportement attendu de la fonction `facture_est_valide`.

```
>>> facture_est_valide('2024-MAY-230')
True
>>> facture_est_valide('2012-MAY-230')
False
>>> facture_est_valide('2024-MAI-230')
False
>>> facture_est_valide('2024-JUN-23')
False
```

On considère le programme suivant :

```
1  calendrier = ['JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN',
2                'JUL', 'AUG', 'SEP', 'OCT', 'NOV', 'DEC']
3
4  def separe(chaine):
5      # renvoie une liste constituée de chaînes qui étaient
6      # séparées par le caractère -
7      return chaine.split('-')
8
9  def que_des_chiffres(chaine):
10     """ renvoie vrai si chaine n'est constituée que
11         des caractères de 0 à 9 faux sinon """
12     for car in chaine:
13         if not(car in "0123456789"):
14             return False
15     return True
```

```
17 def facture_est_valide(chaine):
18     """ renvoie vrai si chaine est une chaîne de
19         caractères conforme au modèle de facture """
20     partie = separe(chaine)
21     if not(len(partie) == 3):
22         return False
23     annee, mois, numero = partie[0], partie[1], partie[2]
24     if not(que_des_chiffres(annee)):
25         return False
26     if not(len(annee) == 4) or not(2018 <= annee <= 2024):
27         return False
28     # Reste à faire vérifier les mois MMM
29     ...
30     # Reste à faire vérifier le numéro xxx
31     ...
32     return True
```

On rappelle que la fonction `split` en Python divise une chaîne de caractères en une liste de sous-chaînes en fonction d'un séparateur spécifié.
Par exemple :

```
texte = 'Bonjour-le-monde'
separateur = '-'
resultat = texte.split(separateur)
```

donne comme résultat `['Bonjour', 'le', 'monde']`

12) Expliquer pourquoi une erreur se produit à l'exécution de la fonction `facture_est_valide` donnée ci-dessus.
13) Proposer une correction du code pour que cette erreur ne se produise plus.
14) Compléter le code afin de vérifier les mois (ligne 29) et le numéro (ligne 31) dans la fonction `facture_est_valide`. Pour chaque vérification, il est possible d'insérer une ou plusieurs lignes.

Exercice 2 : *Cet exercice porte sur les bases de données relationnelles, les requêtes SQL et la programmation en Python.*
L'énoncé de cet exercice utilise des mots-clés du langage SQL suivants : **SELECT**, **FROM**, **WHERE**, **JOIN ... ON**, **UPDATE ... SET**, **INSERT INTO ... VALUES ...**, **COUNT**, **ORDER BY**.
La clause **ORDER BY** suivie d'un attribut permet de trier les résultats par ordre croissant de l'attribut précisé. **SELECT COUNT**(*) renvoie le nombre de lignes d'une requête.
Amélie souhaite organiser sa collection de CD. Elle a commencé par enregistrer toutes les informations sur un fichier CSV mais elle trouve que la recherche d'informations est longue et fastidieuse. Elle repense à son cours sur les bases de données et elle se dit qu'elle doit pouvoir utiliser une base de données relationnelle pour organiser sa collection.
Partie A
Dans cette partie on utilise une seule table. Voici un extrait de la table `Chanson`.

Chanson			
id	titre	album	groupe
1	Sunburn	Showbiz	Muse
2	Muscle Museum	Showbiz	Muse
3	Showbiz	Showbiz	Muse
4	New Born	Origin of Symmetry	Muse
5	Sing for Absolution	Absolution	Muse
6	Hysteria	Absolution	Muse
7	Welcome too the Jungle	Appetite for Destruction	Guns N' Roses
8	Muscle Museum	Hullabaloo	Muse
9	Showbiz	Hullabaloo	Muse

1) L'attribut `titre` peut-il être une clé primaire pour la table `Chanson`? Justifier.
2) Donner le résultat de la requête suivante :
```
SELECT titre, album FROM Chanson WHERE groupe = "Guns N' Roses";
```
3) Écrire une requête SQL permettant d'obtenir tous les titres des chansons de l'album Showbiz dans l'ordre croissant.
4) Écrire une requête SQL permettant d'ajouter la chanson dont le titre est Megalomania de l'album Hullabaloo du groupe Muse.

Amélie a remarqué une faute de frappe dans la chanson Welcome too the Jungle qui s'écrit normalement Welcome to the Jungle.

5) Écrire une requête SQL permettant de corriger cette erreur.

Partie B

Dans cette partie on utilise trois tables. Voici des extraits des trois tables `Chanson`, `Album` et `Groupe`.

Chanson		
id	titre	id_album
1	Sunburn	1
2	Muscle Museum	1
3	Showbiz	1
4	New Born	2
5	Sing for Absolution	4
6	Hysteria	4
7	Welcome to the Jungle	5
8	Muscle Museum	3
9	Showbiz	3

Album			
id	titre	année	id_groupe
1	Showbiz	1999	1
2	Origin of Symmetry	2001	1
3	Hullabaloo	2002	1
4	Absolution	2003	1
5	Appetite for Destruction	1987	2

Groupe	
id	nom
1	Muse
2	Gun N' Roses

6) Expliquer l'intérêt d'utiliser trois tables *Chanson*, *Album* et *Groupe* au lieu de regrouper toutes les informations dans une seule table.
7) Expliquer le rôle de l'attribut *id_album* de la table *Chanson*.
8) Proposer alors un schéma relationnel pour cette version de la base de données. On pensera à bien spécifier les clés primaires en les soulignant et les clés étrangères en les faisant précéder par le symbole #.
9) Écrire une requête SQL permettant d'obtenir tous les noms des albums contenant la chanson Showbiz.
10) Écrire une requête SQL permettant d'obtenir tous les titres avec le nom de l'album des chansons du groupe Muse.
11) Décrire par une phrase ce qu'effectue la requête SQL ci-contre :

```
SELECT COUNT(*) AS tot FROM Album AS a
JOIN Groupe AS g ON a.id_groupe = g.id
WHERE g.nom = 'Muse';
```

Partie C

Dans cette partie, on utilise Python.

Amélie a remarqué que son professeur ne parle jamais d'ordre alphabétique mais d'ordre lexicographique lorsqu'il fait une requête avec **ORDER BY**.

Elle a compris qu'il s'agissait de l'ordre du dictionnaire mais elle se demande comment elle pourrait elle-même écrire une fonction `ordre_lex(mot1, mot2)` de comparaison entre deux chaînes de caractères en utilisant l'ordre lexicographique. La fonction `ordre_lex(mot1, mot2)` prend en arguments deux chaines de caractères et renvoie un booléen. Une rapide recherche lui permet de trouver le résultat suivant :

Lorsque l'on compare deux chaînes de caractères suivant l'ordre lexicographique, on commence par comparer les deux premiers caractères de chacune des deux chaînes, puis en cas d'égalité on s'intéresse au second, et ainsi de suite. Le classement est donc le même que celui d'un dictionnaire. Si lors de ce procédé on dépasse la longueur d'une seule des deux chaînes, elle est considérée plus petite que l'autre. Lorsqu'on dépasse la longueur des deux chaînes au même moment, elles sont nécessairement égales.

Amélie commence par écrire quelques assertions que sa fonction devra vérifier.

```
assert ordre_lex("", "a") == True
assert ordre_lex("b", "a") == ...
assert ordre_lex("aaa", "aaba") == ...
```

12) Compléter les assertions ci-dessus.

On suppose que les chaînes de caractères `mot1` et `mot2` ne sont composées que des lettres de l'alphabet, en minuscule, et la comparaison entre deux lettres peut se faire avec les opérateurs classiques == et <.

Par exemple :

```
>>> "" < "a"
True
>>> "b" == "a"
False
```

Enfin, le slice `mot1[1:]` renvoie la chaîne de caractère de `mot1` privée de son premier caractère.

Par exemple :

```
>>> mot1 = "abcde"
>>> mot2 = mot1[1:]
>>> mot2
"bcde"
```

13) Recopier et compléter la fonction récursive `ordre_lex` ci-dessous qui prend pour paramètre deux chaînes de caractères `mot1` et `mot2` et qui renvoie **True** si `mot1` précède `mot2` dans l'ordre lexicographique.

```
def ordre_lex(mot1, mot2):
    if mot1 == "":
        return True
    elif mot2 == "":
        return False
    else:
        c1 = mot1[0]
        c2 = mot2[0]
        if c1 < c2:
            return ...
        elif c1 > c2:
            return ...
        else:
            return ...
```

14) Proposer une version itérative de la fonction `ordre_lex`.