Gonçalves
Taïga
Groupe D1b
Encadrant : Vuillemot Romain

BE #3 : Base de données SQL

Ce BE consiste à manipuler des bases de données SQL à l'aide de python. Nous commençons d'abord par utiliser le logiciel DB browser for SQLite, en y important la base de données hôtellerie.db.
Essayons d'exécuter la requête suivante :

```
SELECT nom, ville
FROM hotel;
```

Le résultat est le suivant :

	nom	ville
1	Chez Philippe	Bordeaux
2	Grand hotel	Grenoble
3	Hotel chez soi	Lyon
4	Hotel de la gare	Bordeaux
5	Hotel des ambassadeurs	Grenoble
6	Hotel des voyageurs	Nice
7	Hotel terminus	Bordeaux
8	Hotel terminus	Grenoble
9	Hotel terminus	Nice
10	L excelsior	Nice
11	La nuit noire	Bordeaux
12	Le grand hotel	Nice

On obtient un tableau à 12 lignes. La requête a en effet affiché les 12 hôtels de clé primaire différente qui étaient enregistré dans la base de données.

Nous allons donc utiliser les différentes fonctionnalités de ce logiciel sur python, en important le package sqlite3.
Maintenant, nous allons créer une classe HotelDB permettant de réaliser 4 différentes requêtes.

Première requête :

Commençons par la requête get_name_hotel_etoile(self,n). L'argument n est une chaîne de caractères désignant le nombre d'étoiles. Cette requête a pour but d'afficher le nom de tous les hôtels à n étoiles. Elle est codée de la manière suivante :

```
def get_name_hotel_etoile(self,n):
    conn = sqlite3.connect(self.__BDD)
    curseur = conn.cursor()
    liste=' Hôtels ayant {} étoile(s):'.format(n)+'\n      '
    try:
        hotels=curseur.execute("SELECT nom FROM hotel WHERE etoiles={}".format(n)).fetchall()
        if len(hotels)!=0:
            for ligne in hotels:
                liste+=str(ligne[0])+'\n      '
        else:
            liste+="Aucun\n"
    except:
        liste=" Aucun résultat ne correspond à votre recherche. Modifiez l'argument '{}'.\n".format(n)
    conn.close()
    return liste
```

Pour n= '2', la requête affiche en sortie :

```
Hôtels ayant 2 étoile(s):
    La nuit noire
    Hotel chez soi
    Chez Philippe
```

Les lignes « try » et « except » permettent de ne pas afficher d'erreur même si l'argument n entré n'est pas conforme à ce qui est attendu. Par exemple, en entrant n='-1' et n='Hello', on obtient :

```
Hôtels ayant -1 étoile(s):
    Aucun
Aucun résultat ne correspond à votre recherche. Modifiez l'argument 'Hello'.
```

Deuxième requête :

Ecrivons maintenant une requête permettant d'enregistrer un nouveau client dans la base de données. Dans le cas où le client est déjà enregistré, la requête doit renvoyer le numéro de ce client. Ce critère est vérifié au début du script :

```
numclient = curseur.execute("SELECT * FROM client WHERE nom='{}' AND prenom='{}'".format(nom,prenom)).fetchall()
if len(numclient)!=0:
    return "le client est déjà enregistré, son numéro est: "+str(numclient[0][0])
```

Si le client n'est effectivement pas encore enregistré, la requête doit d'abord lister tous les numéros de clients déjà utilisé, puisque ce champ représente la clé primaire de la table 'client'.

```
liste_num = curseur.execute("SELECT numclient FROM client").fetchall()
```

Ensuite, le programme crée un nouveau numéro encore disponible et l'affecte au nouveau client à l'aide de la commande INSERT INTO :

```
new_num = 1
while (new_num,) in liste_num:
    new_num+=1
curseur.execute("INSERT INTO client VALUES({},'{}','{}')".format(new_num,nom,prenom))
conn.commit()
return "Ce client est désormais enregistré. Son numéro est: "+str(new_num)
```

Nous allons tester ce programme en essayant d'enregistrer deux fois le client 'Taïga GONCALVES' :

```
Ce client est désormais enregistré. Son numéro est: 180
```

```
le client est déjà enregistré, son numéro est: 180
```

On constate bien que la vérification du critère mentionné au début fonctionne correctement.

Troisième requête (requête libre) :

Nous allons essayer d'écrire un programme permettant d'étudier l'affluence d'un hôtel entre deux dates.

hotel(numhotel, nom, ville, etoile)
chambre(numchambre, numhotel, etage, type, prixnuitht)
client(numclient, nom, prenom)
occupation(numoccup, numclient, numhotel, datearrivee, datedepart)
reservation(numresa, numclient, numchambre, numhotel, datearrivee, datedepart)

Les arguments à entrer sont les suivants :

- Le nom de l'hôtel. On peut rentrer * pour tous les sélectionner.
- La période sur laquelle on veut étudier l'affluence de l'hôtel. On entrera les dates de début et de fin sous la forme année-mois-jour afin de rester fidèle aux notations dans la base de données.
- Un entier n, permettant de modifier l'histogramme obtenu en sorti. Son utilité sera expliquée par la suite.

Remarque : On pourrait prendre en compte le type de la chambre pour savoir combien de personnes logent par chambre. Mais cette information n'est pas sûre, car on pourrait avoir une seule personne dans une chambre double, ou trois personnes (dont un enfant) dans une chambre double. On ne connait pas non plus le nombre de lits dans les suites. On ne prend donc pas en compte cette information.

La première étape consiste à récupérer les dates pour lesquelles les chambres étaient occupées ou réservées. Seuls les chambres des hôtels sélectionnés dans l'argument sont prises en compte :

```
if hotel == "*":
    num_hotel = curseur.execute("SELECT numhotel FROM hotel").fetchall()
    liste_num = tuple([num_hotel[i][0] for i in range(len(num_hotel))])
    do = curseur.execute("SELECT datedepart, datearrivee FROM occupation WHERE numhotel IN {}".format(liste_num)).fetchall()
    dr = curseur.execute("SELECT datedepart, datearrivee FROM reservation WHERE numhotel IN {}".format(liste_num)).fetchall()
else:
    num_hotel = curseur.execute("SELECT numhotel FROM hotel WHERE nom='{}'".format(hotel)).fetchall()
    liste_num = num_hotel[0][0]
    do = curseur.execute("SELECT datedepart, datearrivee FROM occupation WHERE numhotel == {}".format(liste_num)).fetchall()
    dr = curseur.execute("SELECT datedepart, datearrivee FROM reservation WHERE numhotel == {}".format(liste_num)).fetchall()

conn.close() #On n'a plus besoin de la BDD, donc on ferme l'accès dès la fin de cette étape.
```

Il faut bien noter que les listes do et dr définies dans la première étape ne contiennent pas toutes les dates durant lesquelles les chambres étaient occupées. En effet, ces listes ne prennent en compte que la date d'arrivée et de départ de chaque client. C'est pour cela que dans cette seconde étape, nous créons une liste dates dans lesquelles toutes les dates (do et dr confondues) sont prises en considération :

```
dates = []
for D in do:
    if type(D[0])==str and type(D[1])==str:
        dates+=trans_periode(D[1],D[0])  #tra
for D in dr:
    if type(D[0])==str and type(D[1])==str:
        dates+=trans_periode(D[1],D[0])
```

Dans ce code, nous utilisons la fonction trans_periode, définie en dehors de la classe HotelDB car elle est utilisée plusieurs fois, y compris dans la quatrième requête. Son code est le suivant :

```
def trans_periode(dd,df):
    """Créee une liste comprenant toutes les dates entre dd et df (dd et df sont de la forme année-mois-jour)"""
    dd2,df2 = trans_date(dd),trans_date(df)
    periode = [dd2]
    while periode[-1] < df2:
        D = periode[-1] + datetime.timedelta(days=1)
        periode+=[D]
    return periode
```

La fonction trans_date est également implémentée en dehors de la classe HotelDB :

```
def trans_date(D):
    """Transforme la date écrite avec le format aaaa-mm-dd en objet de type datetime.date"""
    return datetime.date(int(D[0:4]),int(D[5:7]),int(D[8:10]))
```

Dans une troisième étape, nous préparons l'axe des abscisses. Nous créons deux listes :

- Période est la liste contenant toutes les dates (dans l'ordre) de la période sur laquelle on veut évaluer l'affluence des clients.
- Période_pas_n ne prend en compte qu'une seule date toutes les n dates.

```
date_debut = trans_date(dd) #trans_date() est une fonction p
date_fin = trans_date(df)
periode = trans_periode(dd,df)
periode_pas_n = [date_debut]

while periode_pas_n[-1] < date_fin:
    D_pas_n = periode_pas_n[-1] + datetime.timedelta(days=n)
    periode_pas_n+=[D_pas_n]
```

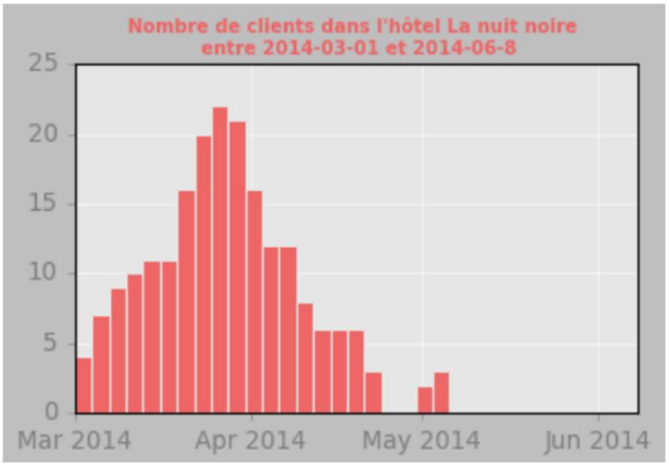
La dernière étape consiste simplement à tracer l'histogramme caractérisant le nombre de clients en fonction de la date.

```
plt.style.use('classic')
plt.figure(figsize=(5,3))
ax = plt.axes(facecolor='#E6E6E6')
ax.set_axisbelow(True)
plt.grid(color='w', linestyle='solid')
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()
if hotel == '*':
    ax.set_title("Nombre de clients dans tous les hôtels \n entre {}
else:
    ax.set_title("Nombre de clients dans l'hôtel {} \n entre {} et {
marqueurs = []
labels = []
for date in periode:
    if date.day == 1:
        marqueurs+=[date]
        date_str = date.strftime("%b %Y")
        labels+=[date_str]
plt.xticks(marqueurs,labels)
ax.tick_params(colors='gray', direction='out')
for tick in ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')
ax.hist(dates, periode_pas_n, edgecolor='#E6E6E6', color='#EE6666')
plt.savefig('affluence.png')
plt.show()
plt.close()
```

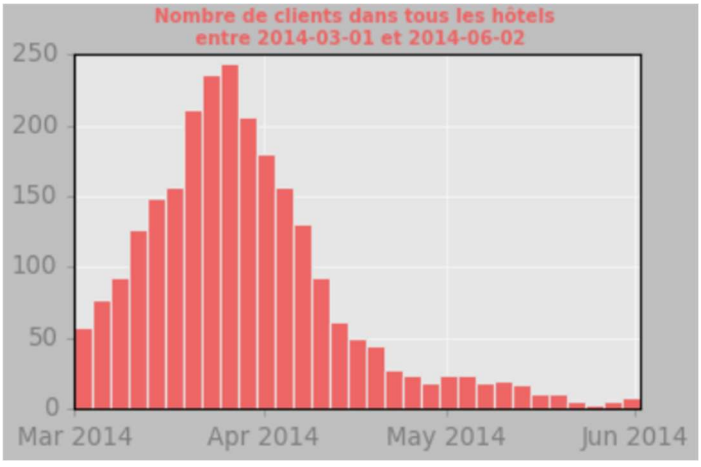
La liste période définie dans l'étape 3 permet de créer des graduations sur l'axe des abscisses (aves ajout de labels). Quant à période_pas_n, elle est utilisée de sorte à regrouper le nombre de clients dans l'hôtel par paquet de n jours. Ainsi, si n=1, on obtient un histogramme comptant le nombre de client chaque jour. Si n=2, on obtient un histogramme avec le nombre de clients tous les deux jours, et ainsi de suite.

Nous avons testé cette requête avec deux cas :

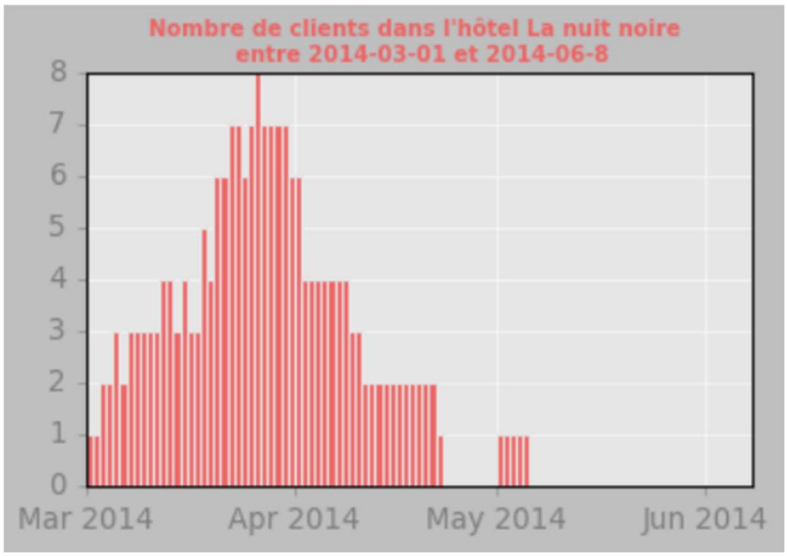
```
print(aHotelDB.affluence("La nuit noire","2014-03-01","2014-06-8",3))
```




```
print(aHotelDB.affluence("*","2014-03-01","2014-06-02",3))
```

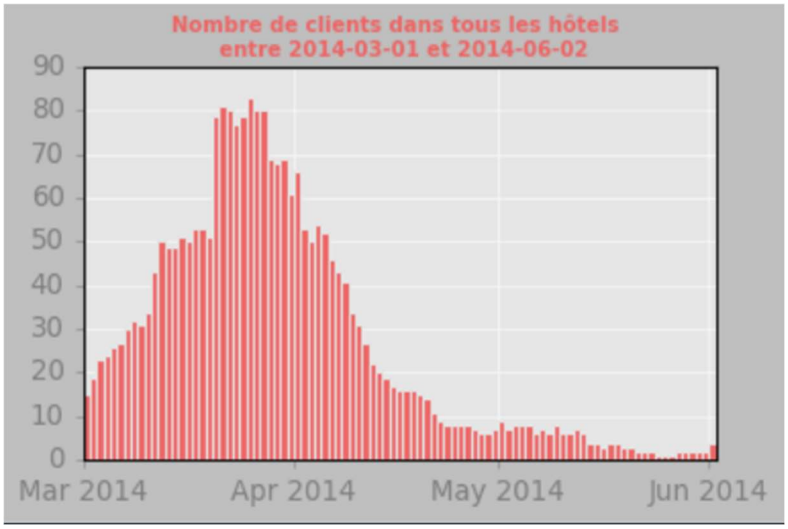


Ces deux histogrammes ont été tracés pour n=3. Si on prenait n=1 pour le cas de gauche, on obtiendrait le résultat suivant :



On observe certains pics trop accentués. Cela est dû au fait qu'il n'y a pas beaucoup de clients par nuit dans cet hôtel. Ainsi, il est plus pertinent de visualiser un histogramme avec n=3 plutôt que n=1, d'où l'intérêt de cet argument.

Si on prenait n=1 pour le cas de droite, on obtiendrait le résultat suivant :



Dans cet histogramme, on n'a pas de pics aberrants, car le nombre de clients par nuit et bien plus élevé. Dans cette situation, il est donc plus intéressant de visualiser l'histogramme pour n=1 plutôt que pour n=3. Ainsi, il est bien pratique de pouvoir modifier cet argument lors de l'appel de cette requête.

Quatrième requête (requête libre) :

Dans cette dernière requête, il sera possible de visualiser le profit réalisé par chaque hôtel entre deux périodes. Par soucis de manque de données, nous ne calculons pas réellement de 'profit', mais nous comptons tout simplement la somme d'argent collectée par l'hôtel grâce à l'occupation/la réservation des chambres.

hotel(numhotel, nom, ville, etoile)
chambre(numchambre, numhotel, etage, type, prixnuitht)
client(numclient, nom, prenom)
occupation(numoccup, numclient, numhotel, datearrivee, datedepart)
reservation(numresa, numclient, numchambre, numhotel, datearrivee, datedepart)

Les arguments à entrer sont les suivants :

- Les dates dd et df de début et de fin de la période où l'on compte la somme d'argent récoltée par les hôtels.
- Le nombre d'étoiles i (en chaîne de caractères) des hôtels pour lesquels on réalise la requête. On entre i='all' pour prendre en compte tous les hôtels.

Tout d'abord, nous allons créer une liste appelée 'liste_hotel' contenant des sous-listes. Ces dernières sont pour l'instant composées du numéro d'identification et du nom de l'hôtel qu'on veut étudier.

```
if n == "all":
    hotel = curseur.execute("SELECT numhotel,nom FROM hotel").fetchall()
    # liste_num = tuple([hotel[i][0] for i in range(len(hotel))])
    liste_hotel = [[hotel[i][0],hotel[i][1]] for i in range(len(hotel))]
else:
    hotel = curseur.execute("SELECT numhotel,nom FROM hotel WHERE etoiles='{}'".format(n)).fetchall()
    # liste_num = tuple([hotel[i][0] for i in range(len(hotel))])
    liste_hotel = [[hotel[i][0],hotel[i][1]] for i in range(len(hotel))]
```

Nous allons maintenant compléter cette liste, en y ajoutant dans chaque sous-liste la somme d'argent récoltée par l'hôtel entre dd et df. On utilise donc une boucle for.

```
for i in range(len(liste_hotel)):

    do = curseur.execute("SELECT datedepart, datearrivee, c.numhotel, prixnuitht FROM occupation as o JOIN chambre as c ON"
                         " o.numchambre=c.numchambre AND o.numhotel=c.numhotel WHERE c.numhotel = {}".format(liste_hotel[i][0])).fetchall()

    dr = curseur.execute("SELECT datedepart, datearrivee, c.numhotel, prixnuitht FROM reservation as r JOIN chambre as c ON"
                         " r.numchambre=c.numchambre AND r.numhotel=c.numhotel WHERE c.numhotel = {}".format(liste_hotel[i][0])).fetchall()
    ajout_profit(do,i)
    ajout_profit(dr,i)


conn.close()
```

Pour calculer la somme d'argent gagnée par chaque hôtel de la sous-liste 'i', nous créons d'abord deux liste do et dr, contenant les informations sur les dates d'occupation de toutes les chambres de l'hôtel concerné ainsi que le prix de ces chambres.

On utilise ensuite la fonction ajout_profit(D,i) avec D=do ou dr et i une liste. Le code de cette fonction est détaillé ci-dessous :

```
def ajout_profit(D,i):
    """ajoute la somme d'argent récolté par l'hôtel i (D=do ou D=dr)"""
    profit = 0
    for hotel in D:
        datedepart,datearrivee=hotel[0],hotel[1]
        if type(datedepart)==str and type(datearrivee)==str and trans_date(datearrivee) in periode:
            if trans_date(datedepart) in periode:
                nb_jours = trans_date(datedepart)-trans_date(datearrivee)
            else:
                nb_jours = periode[-1]-trans_date(datearrivee)
            profit+= nb_jours.days * hotel[3]
    liste_hotel[i].append(profit)
```

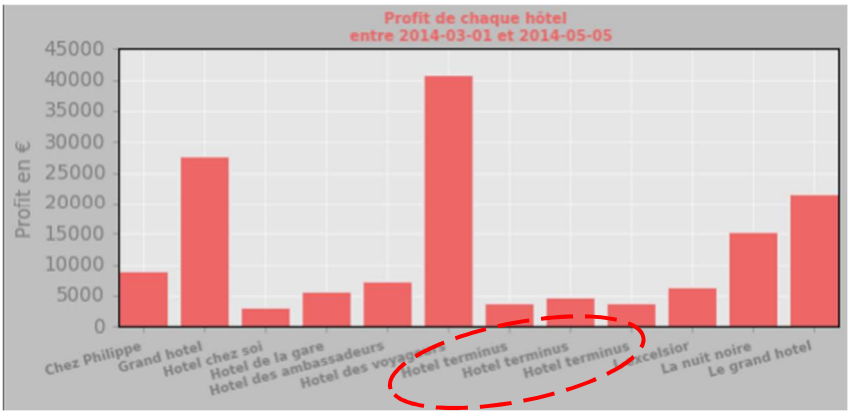
La liste 'liste_hotel' contient désormais toutes les informations nécessaires pour tracer un diagramme à bandes. Le code de ce tracé est le suivant :

```
plt.style.use('classic')
plt.figure(figsize=(8,3))
ax = plt.axes(facecolor='#E6E6E6')
ax.set_axisbelow(True)
plt.grid(color='w', linestyle='solid')
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()
x=range(1,1+len(nom_hotel))
if n == 'all':
    ax.set_title("Profit de chaque hôtel \n entre {} et {}".format(dd,df),fontsize=9, fontweight='bold', color='#EE6666')
else:
    ax.set_title("Profit des hôtels {} étoile(s) \n entre {} et {}".format(n,dd,df),fontsize=9, fontweight='bold', color='#EE6666')
plt.xticks(x,nom_hotel,ha='right',rotation=15,fontsize=8, fontweight='bold')
ax.tick_params(colors='gray', direction='out')
for tick in ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')
ax.set_ylabel('Profit en €', color ='gray')
ax.bar(x,profits, edgecolor='#E6E6E6', color='#EE6666')
plt.savefig('profits.png')
plt.show()
plt.close()
```

Testons ce programme, en entrant :

```
print(aHotelDB.profits("2014-03-01","2014-05-05","all"))
```

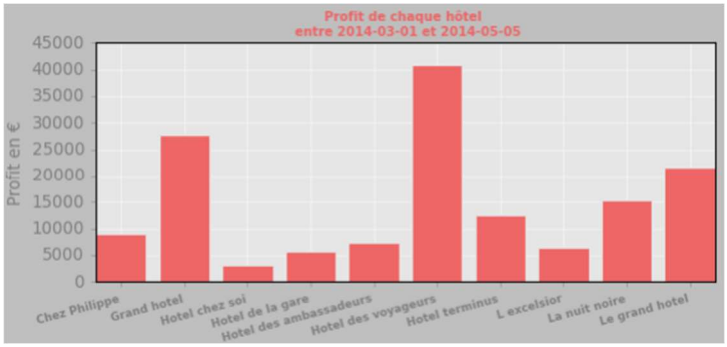
Le diagramme obtenu est :



Un problème survient : l'hôtel « Hôtel terminus » apparaît trois fois. Cela est dû au fait qu'il existe dans plusieurs villes. On voudrait alors fusionner ces trois hôtels, puisqu'ils sont gérés par la même entreprise. On rajoute alors une partie de code avant de réaliser le tracé du diagramme :

```
nom_hotel = []
profits = []
#Cette section est faite car certains hôtels ont le même nom mais
#L'argent récolté par les hôtels de même nom sont alors fusionné
for i in range(len(liste_hotel)):
    if liste_hotel[i][1] not in nom_hotel:
        nom_hotel+=[liste_hotel[i][1]]
        profits+=[liste_hotel[i][2]]
    else:
        #Ici, on fusionne le profit des deux hôtels de même nom.
        for j in range(len(nom_hotel)):
            if liste_hotel[i][1]==nom_hotel[j]:
                profits[j]+=liste_hotel[i][2]
```

Le tracé qu'on obtient est désormais le suivant :



L'hôtel terminus n'apparaît plus qu'une fois. Le programme semble bien fonctionnel.

Pour terminer, retestons ce programme en entrant : `print(aHotelDB.profits("2014-03-01","2014-05-05","2"))`

Le diagramme obtenu est :