

Rapport d'activité - SAE 15 - Nathan Manton

Dans le cadre de la SAE 1.5 - Traiter des données, j'ai été amené à mener un projet de collecte, stockage et traitement de données.

Lecture du sujet

A la lecture du sujet, on remarque qu'il faudra produire un outil qui permettra les choses suivantes

- Collecte des données
 - Données des parkings
 - Données des vélos
 - Données du tramway
- Stockage des données
 - Traduire les fichiers xml des parkings en données stockables et réutilisables
 - De même pour les fichiers Json des vélos
 - De même pour le tramway
- Mise en forme des données
- Traitement des données
 - Permettre un affichage des données en fonction de plusieurs paramètres temporels
 - date de début
 - date de fin
 - laps de temps
 - Permettre un affichage sur un plan géographique de la ville (à faire si le temps le permet)

Pour l'instant, nous garderons l'interprétation des données pour l'humain, il n'est pas exclu qu'un programme pouvant interpréter les données soit codé

Acquisition des données

En regardant les données disponibles sur le site Open Data Montpellier, on voit que les données en rapport avec le tramway ne peuvent pas nous donner d'informations sur leur utilisation, nous ne pouvons avoir que des informations géographiques qui nous serviront pour le traitement et l'interprétation. Pour rendre mon code utilisable uniquement avec les programmes, je choisis tout de même d'ajouter une fonction pour récupérer le fichier contenant les informations sur les stations de tramway de la ville. Je profite de cette fonction pour prendre les mêmes informations sur les stations veloMag. Pour les parkings automobiles, les informations n'existent pas sur le site de l'agglomération, je vais alors créer un fichier .csv manuellement.

Parkings automobiles

```
def getPark(idPark:str,path="."):  
  
response=requests.get(f"https://data.montpellier3m.fr/sites/default/files/ressources/  
{idPark}.xml") #Acquisition du fichier xml du parking grâce à la variable idPark qui
```

```

renseigne l'identifiant du parking
    file=open(f"{path}/{idPark}_{int(time.time())}.xml","w+", encoding="UTF-8")
#Création d'un fichier pour stocker le contenu du fichier .xml téléchargé, si
l'utilisateur veut enregistrer le fichier dans un répertoire particulier, il peut
renseigner la variable ``path`` par défaut, la fonction sauvergarde le fichier dans
le même répertoire que le programme
    file.write(response.text) #Ecriture du fichier
    file.close() #Fermeture de l'instance de fichier
    return file.name #On retourne le chemin du fichier.

```

Parkings veloMag

```

def getVelo(path="."):
    response=requests.get("https://montpellier-fr-
smooove.klervi.net/gbfs/en/station_status.json") #Acquisition du fichier json
représentant l'état de toutes les stations veloMag
    file=open(f"{path}/veloMag_{int(time.time())}.json","w+", encoding="UTF-
8")#Création d'un fichier pour stocker le contenu du fichier .json téléchargé, si
l'utilisateur veut enregistrer le fichier dans un répertoire particulier, il peut
renseigner la variable ``path`` par défaut, la fonction sauvergarde le fichier dans
le même répertoire que le programme
    file.write(response.text) #Ecriture du fichier
    file.close() #Fermeture de l'instance du fichier
    return file.name #On retourne le chemin du fichier

```

Emplacement des parkings veloMag et des stations de Tramway

```

def getInfos(path="."):
    #Dictionnaire contenant les URLs des stations liés au moyen de transport
    urls =
{"tram":"https://data.montpellier3m.fr/sites/default/files/ressources/MMM_MMM_ArretsT
ram.json","veloMag":"https://montpellier-fr-
smooove.klervi.net/gbfs/en/station_information.json"}
    files=[] #Liste qui contiendra les deux fichiers d'informations récupérés
    for key in urls.keys: #Boucle pour les deux urls
        response=requests.get(urls[key]) #Récupération du fichier
        file=open(f"{path}/{key}.json","w+", encoding="UTF-8") #Création des fichiers
.json avec les informations
        file.write(response.text) #Ecriture du fichier
        file.close() #Fermeture de l'instance du fichier
        files.append(file.name)#On ajoute le chemin d'accès au fichier dans la liste
    return files #On retourne la liste contenant les deux fichiers

```

Emplacement des parkings automobile

A l'aide de Google Maps et de la liste des parkings sur le site d'Open Data Montpellier, j'ai récupéré manuellement les coordonnées GPS des parkings dont nous pouvons traiter les données afin de les utiliser lors du traitement et donc de l'interprétation des données.

Mise en forme des données

Pour faciliter les passages entre fichiers, base de données et programmes de traitement des données, j'ai choisi de créer de nouvelles classes, une par type de station

- Une classe `parking` contenant :
 - Un attribut `time` de type `int` donnant l'heure de la prise d'information en secondes depuis epoch UNIX
 - Un attribut `parkID` de type `str` donnant l'identifiant du parking
 - Un attribut `open` de type booléen donnant l'état d'ouverture du parking (True pour ouvert, False pour fermé)
 - Un attribut `free` de type `int` donnant le nombre de places libres dans le parking
 - Un attribut `total` de type `int` donnant le nombre de places totales dans le parking
- Une classe `velo` contenant :
 - Un attribut `time` de type `int` donnant l'heure de la prise d'information en secondes depuis epoch UNIX
 - Un attribut `id` de type `int` donnant l'identifiant de la station
 - Un attribut `bikes` de type `int` donnant le nombre de vélos disponibles
 - Un attribut `dis` de type `int` donnant le nombre de vélos indisponilbes mais garés à la station
 - Un attribut `freede` de type `int` donnant le nombre de places disponibles à la station

Elles sont définies avec les codes suivants

Classe `parking`

```
class parking:
    def __init__(self, parkID, open, free, total):
        self._time = int(time.time())
        self._parkID = parkID
        self._open = open
        self._free = free
        self._total = total

    #définition des getter et setters des attributs
    @property
    def time(self):
        return self._time
    @time.setter
    def time(self, time):
        #check si time est bien un entier
        if type(time) == int:
            self._time = time
        else:
            raise TypeError("time type must be an int !")

    @property
    def parkID(self):
        return self._parkID
```

```

@parkID.setter
def parkID(self,parkID):
    #check si parkID est bien dans les id disponibles sur le site opendata
    if parkID in
['FR_MTP_ANTI', 'FR_MTP_COME', 'FR_MTP_CORU', 'FR_MTP_EURO', 'FR_MTP_FOCH', 'FR_MTP_GAMB',
'FR_MTP_GARE', 'FR_MTP_TRIA', 'FR_MTP_ARCT', 'FR_MTP_PITO', 'FR_MTP_CIRC', 'FR_MTP_SABI', '
FR_MTP_GARC', 'FR_CAS_SABL', 'FR_MTP_MOSS', 'FR_STJ_SJLC', 'FR_MTP_MEDC', 'FR_MTP_OCCI', 'F
R_CAS_VICA', 'FR_MTP_GA109', 'FR_MTP_GA250', 'FR_CAS_CDGA', 'FR_MTP_ARCE', 'FR_MTP_POLY']:
        self._parkID=parkID
    else:
        raise ValueError("parkID is not valid !")

@property
def open(self):
    return self._open
@open.setter
def open(self,open):
    #check si open est bien un booléen
    if type(open)==bool:
        self._open=open
    else:
        raise TypeError("open type must be a bool !")

@property
def free(self):
    return self._free
@free.setter
def free(self,free):
    #check si free est bien un int
    if type(free) == int:
        self._free=free
    else:
        raise TypeError("free must be an int !")

@property
def total(self):
    return self._total
@total.setter
def total(self,total):
    #check si total est bien un int
    if type(total)==int:
        self._total==total
    else:
        raise TypeError("total must be an int !")

```

Classe **velo**

```

class velo:
    def __init__(self,statID,bikes,dis,free):
        self._time=int(time.time())
        self._statID=statID
        self._bikes=bikes
        self._dis=dis
        self._free=free

```

```
#définition des getter et setters des attributs
@property
def time(self):
    return self._time
@time.setter
def time(self,time):
    #check si time est bien un entier
    if type(time) == int:
        self._time == time
    else:
        raise TypeError("time type must be an int !")

@property
def statID(self):
    return self._statID
@statID.setter
def statID(self,statID):
    #check si statID est bien un entier
    if type(statID) == int:
        self._statID=statID
    else:
        raise TypeError("statID must be an int !")

@property
def bikes(self):
    return self._bikes
@bikes.setter
def bikes(self,bikes):
    #check si dis est bien un int
    if type(bikes)==int:
        self._bikes==bikes
    else:
        raise TypeError("bikes must be an int !")

@property
def dis(self):
    return self._dis
@dis.setter
def dis(self,dis):
    #check si dis est bien un int
    if type(dis)==int:
        self._dis==dis
    else:
        raise TypeError("dis must be an int !")

@property
def free(self):
    return self._free
@free.setter
def free(self,free):
    #check si free est bien un int
    if type(free) == int:
        self._free=free
```

```

else:
    raise TypeError("free must be an int !")

```

Avec ces nouvelles classes, nous pouvons alors stocker les fichiers enregistrés sur des variables, stockables par la suite dans une base de données. Il n'est alors plus utile de stocker directement les fichiers. Je vais alors créer de nouvelles fonctions d'acquisition afin qu'elle ne renvoient plus un chemin d'accès vers un fichier mais un objet. Toutefois, mes fonctions de téléchargement de fichiers vont rester dans le module au cas où j'en aurais besoin dans la suite de la SAE. Je vais changer leurs noms pour `getParkFile` et `getVeloFile`. Ma fonction `getInfos` ne changera pas puisqu'elle n'est pas concernée directement par les nouvelles classes.

Code de la nouvelle fonction `getPark`

```

def getPark(idPark:str):

response=requests.get(f"https://data.montpellier3m.fr/sites/default/files/ressources/
{idPark}.xml") #Acquisition du fichier xml du parking grâce à la variable idPark qui
renseigne l'identifiant du parking
    if not "Page non trouvée" in response.text: #On vérifie que le fichier récupéré
est bien un fichier valide et non une erreur 404
        tree = etree.fromstring(str(response.text).encode()) #Le contenu du fichier
récupéré est ensuite encodé en UTF-8 pour qu'il soit compris par la librairie lxml
puis mis sous forme d'élément Etree
        return parking(idPark,tree.xpath("Status")
[0].text=="Open",int(tree.xpath("Free")[0].text),int(tree.xpath("Total")[0].text))
#On crée et renvoie l'objet parking

```

Code de la nouvelle fonction `getVelo`

```

def getVelo():
    result=[] #Intialisation de la liste qui va contenir les objets velo, un objet
par station
    response=requests.get("https://montpellier-fr-
smoove.klervi.net/gbfs/en/station_status.json") #Acquisition du fichier json
représentant l'état de toutes les stations velaMag
    content=StringIO(response.text) #On convertit la chaine de caratères du contenu
du fichier en chaine considérable comme un fichier pour l'utiliser avec la librairie
json
    content=json.load(content) #On load le fichier, il sera alors convertit en objets
itérables et donc utilisables
    for station in content["data"]["stations"]: #le dictionnaire data contient la
liste stations qui elle même contient les dictionnaires représentant chaque stations
        #On crée et ajoute l'objet vélo à la liste qui sera retournée en fin de
fonction

result.append(velo(int(station["station_id"]),int(station["num_bikes_available"]),int
(station["num_bikes_disabled"]),int(station["num_docks_available"])))
    return result #On renvoie la liste contenant l'état de chaque station.

```

Stockage des données

Pour stocker les données, je me suis orienté vers une base SQLite, facile d'utilisation avec python et un type de base avec laquelle j'ai déjà travaillé par le passé.

Il me faudra alors plusieurs tables

- Une table qui contiendra les acquisitions des parkings
- Une table qui contiendra les acquisitions des stations VeloMagg
- Une table pour mettre en relation les ids des stations VeloMagg avec leurs informations
- Une table pour mettre en relation les ids des parkings avec leurs informations
- Une table pour inscrire les informations des stations de tramway (utiles pour l'interprétation des données)

Conception des tables

Table **infosPark**

id	name	lat	long
Identifiant du parking et clé primaire	Nom complet du parking	Latitude des coordonnées du parking	Longitude des coordonnées du parking
str	str	float	float

Nous donnant alors le code SQL suivant :

```
CREATE TABLE "infosPark" (  
  "id"      TEXT,  
  "name"    TEXT,  
  "lat"     REAL,  
  "long"    REAL,  
  PRIMARY KEY("id")  
);
```

Table **infosVelo**

id	name	lat	long	capacity
Identifiant de la station et clé primaire	Nom complet de la station	Latitude de la station	Longitude de la station	Capacité de la station
int	str	float	float	int

Nous donnant alors le code SQL suivant :

```
CREATE TABLE "infosVelo" (  
  "id"      INTEGER,  
  "name"    TEXT,  
  "lat"     REAL,  
  "long"    REAL,  
  "capacity" INTEGER,
```

```
PRIMARY KEY("id")
);
```

Table infosTram

nom	lat	long
Nom de la station et clé primaire	Latitude de la sation	Longitude de la station
str	float	float

Nous donnant alors le code SQL suivant :

```
CREATE TABLE "infosTram" (
  "name" TEXT,
  "lat" REAL,
  "long" REAL,
  PRIMARY KEY("name")
);
```

Il était nécessaire de faires ces trois tables en premier afin de pouvoir les lier avec les tables d'acquisition

Table acquisPark

idAcquis	time	idPark	free	total	occup
Clé primaire de la table	Heure d'acquisition epoch	Identifiant du parking	Nombre de places libres	Nombre de places total	Taux d'occupation du parking
int en Auto-incrémentation	int	str Clé étrangère: id de infosPark	int	int	float

Nous donnant alors le code SQL suivant :

```
CREATE TABLE "acquisPark" (
  "idAcquis" INTEGER,
  "idPark" TEXT,
  "free" INTEGER,
  "total" INTEGER,
  "occup" REAL,
  FOREIGN KEY("idPark") REFERENCES "infosPark"("id") ON UPDATE CASCADE,
  PRIMARY KEY("idAcquis" AUTOINCREMENT)
);
```

Table acquisVelo

idAcquis	time	idStat	bikes	dis	free	total	occup
----------	------	--------	-------	-----	------	-------	-------

idAcquis	time	idStat	bikes	dis	free	total	occup
Clé primaire de la table	Heure d'acquisition epoch	Identifiant de la station VeloMagg	Nombre de vélos disponibles	Nombre de vélos non-disponibles	Nombre de docks libres	Nombre de docks total	Taux d'occupation de la station
int en Auto-incrémentation	int	int clé étrangère de idStat sur la table statVelo	int	int	int	int	int

Nous donnant alors le code SQL suivant :

```
CREATE TABLE "acquisVelo" (
  "idAcquis"    INTEGER,
  "time"        INTEGER,
  "idStat"      INTEGER,
  "bikes"       INTEGER,
  "dis"         INTEGER,
  "free"        INTEGER,
  "total"       INTEGER,
  "occup"       REAL,
  FOREIGN KEY("idStat") REFERENCES "infosVelo"("id") ON UPDATE CASCADE,
  PRIMARY KEY("idAcquis" AUTOINCREMENT)
);
```

Il me faut alors des fonctions pour enregistrer mes données dans ma base, je crée alors un deuxième module contenant mes fonctions d'enregistrement.

Enregistrement d'un objet de classe `parking`

Je connais la forme de ma requête d'insertion, il me suffit alors de créer une fonction qui crée la requête et l'exécute

```
def savePark(park:parking):
    #connection base
    connection = sq.connect("db.db")
    #Création curseur pour l'exécution de la requête
    cursor = connection.cursor()
    #Création de la requête sql
    query = f"""INSERT INTO acquisPark
    (time, idPark, free, total, occup)
    VALUES
    ({park.time}, '{park.parkID}', {park.free}, {park.total}, {100-
    (round(park.free/park.total, 2))*100})
    """
    #Exécution de la requête
```

```
cursor.execute(query)
#Sauvegarde de la base avec modification
connection.commit()
#Fermeture des instances
cursor.close()
connection.close()
```

Enregistrement d'un objet de classe **velo**

De même que la fonction **savePark** je crée une fonction pour ma classe **velo**

```
def saveVelo(stat:velo):
    #connection base
    connection = sq.connect("db.db")
    #Création curseur pour l'exécution de la requête
    cursor = connection.cursor()
    #calcul du nombre total de places
    total = stat.free + stat.dis + stat.bikes
    #Création de la requête sql
    query = f"""INSERT INTO acquisVelo
    (time, idStat, bikes, dis, free, total, occup)
    VALUES
    ({stat.time}, '{stat.statID}', {stat.bikes}, {stat.dis}, {stat.free}, {total}, {100-
    (round(stat.free/total,2))*100})
    """
    #Exécution de la requête
    cursor.execute(query)
    #Sauvegarde de la base avec modification
    connection.commit()
    #Fermeture des instances
    cursor.close()
    connection.close()
```

Traitement des données

Le traitement des données se fera via python et GNUplot. Dans un premier temps, nous ferons des graphiques qui ne seront que sauvegardés en fichiers images. A terme, si le temps le permet, il sera possible d'ajouter une interface graphique à notre programme