Organiser et transformer les données : Pony ORM

Ici nous apprenons à utiliser la librairie Pony pour la manipulation et la mise en forme des données.

Pour installer les librairie pandas et pony, vous devez utfuiliser un gestionnaire d'installation.



Attention la librairie pony n'est pas disponible sur les depots d'Anaconda. Il est préférable d'utiliser l'installeur pip en ligne de commande:

```
$ python -m pip install pandas pony
```

La librairie Pony ORM est un gestionnaire de persistance qui permet la mise en correspondance entre les objets d'un programme et les valeurs d'une base de données, pour assurer leur conservation d'une session à l'autre.

Pony effectue toutes les opérations de sauvegarde de manière transparente. La création et la mise à jour des objets persistants s'accompagne automatiquement d'opérations de lecture/écriture vers la base de donnée. Les données sont donc conservées sans appel explicite à des requêtes SQL.

Initialisation

```
from pony import orm
```

```
db = orm.Database()
```

Chargement des données avec Pandas

L'utilisation de données structurées dans un programme Python nécessite de faire appel à des librairies spécialisées. Nous utiliserons ici la librairie `pandas` qui sert à la mise en forme et à l'analyse des données.

```
import numpy as np
import pandas
```

On considère une série d'enregistrements concernant des ventes réalisées par un exportateur de véhicules miniatures. Pour chaque vente, il entre dans son registre de nombreuses informations :

- nom de la société cliente
- nom et prénom du contact, adresse, téléphone
- nombre d'unités vendues
- prix de vente
- etc...

Ces informations sont stockées dans un fichier au format 'csv' (comma separated values) : `ventes_new.csv`. Téléchargez ce fichier et copiez-le dans votre répertoire de travail.

Dans un premier temps, regardez son contenu avec un editeur de texte (**geany**, **gedit** ou autre...). La première ligne contient les noms des attributs (`NUM_COMMANDE`, `QUANTITE`,...). Les ligne suivantes contiennent les valeurs d'attributs correspondant à une vente donnée. En tout plus de 2000 ventes sont répertoriées dans ce fichier.

Ouvrez-le maintenant à l'aide d'un tableur (par exemple **`localc`**). Les données sont maintenant "rangées" en lignes et colonnes pour faciliter la lecture.

Déplacez le fichier `ventes_new.csv` dans votre répertoire de travail.

Lecture des données

Les données sont au format `csv`, on utilise:

- `pandas.read_csv`. Voir dataframes pandas. Pandas permet également de lire les données au format `xls` et `xlsx` (Excel).

```
with open('ventes_new.csv', encoding='utf-8') as f:
    data = pandas.read_csv(f)
print(data)
```

avec `data` une structure de données de type `DataFrame`

Création du schéma de données

Nous définissons ici trois schémas de classes correspondant aux ensembles d'entités Client, Commande et Produit.

- **Client**(id_client, téléphone, ville, pays)
- **Commande**(num_commande, code_produit, id_client, quantité, montant, mois, année)
- **Produit**(code_produit, type_produit, prix_unitaire)

Les clés étrangères de la table des commande définissent deux relations de un à plusieurs :

- une relation de un à plusieurs entre *un* produit et *des* commandes,
- et une relation de un à plusieurs entre *un* client et *des* commandes.



Attention, dans la table commande, `code_produit` est également une clé étrangère

Dans un modèle ORM, les relations de un à plusieurs se traduisent par des attributs de type liste ou ensemble :

- A un client correspond un ensemble de commandes
- A un produit correspond un ensemble de commandes
- A une commande correspond *un* client et *un* produit

Classe Client

Les classes sont définies ici comme des schémas de données.

La classe Client hérite de la classe générique *Entity*. Les attributs des objets obéissent à une définition parmi quatre définitions possibles :

- attribut clé primaire : `PrimaryKey`
- attribut requis (la valeur doit être renseignée) : `Required`
- attribut facultatif: `Optional`
- relation de un à plusieurs : `Set`

```
class Client(db.Entity):
    id_client = orm.PrimaryKey(str)
    telephone = orm.Required(str)
    ville = orm.Required(str)
    pays = orm.Required(str)
    achats = orm.Set('Commande')
```

Classe Produit

```
class Produit(db.Entity):
    code_produit = orm.PrimaryKey(str)
    type_produit = orm.Required(str)
    prix_unitaire = orm.Required(float)
    ventes = orm.Set('Commande')
```

Classe Commande

Dans la classe Commande, il n'y a pas de clé étrangère (comme dans le modèle relationnel) mais :

- un attribut de type `Client` qui lie la commande au client qui a effectié la commande
- un attribut de type `Produit` qui lie la commande au produit commandé

```
class Commande(db.Entity):
    num_commande = orm.Required(int)
    code_produit = orm.Required(str)
    orm.PrimaryKey(num_commande, code_produit)
    quantité = orm.Required(int)
    montant = orm.Required(float)
    mois = orm.Required(int)
    année = orm.Required(int)
    client = orm.Required(Client)
    produit = orm.Required(Produit)
```

Pour afficher

La commande `show` est une commande d'affichage à tout faire. Elle permet ici de vérifier le schéma de la classe.

```
orm.show(Client)
```

Association à un gestionnaire de BD

Les schémas de données définis dans les classes peuvent être implémentés dans différents gestionnaires de bases de données.

Nous choisissons ici le gestionnaire sqlite, ce qui évite de définir une connexion un serveur distant. La base de données est ici émulée en mémoire centrale (pour les besoins de l'exercice, les données n'ont pas besoin d'être conservées)

```
db.bind(provider='sqlite', filename='ventes.db', create_db=True)
```

Mode debug

Le mode debug permet de voir les échanges avec la base de données.

```
orm.set_sql_debug(True)
```

La commande `generate_mapping` définit l'"appariement entre les objets et la base de données. Cela correspond ici à la création de trois tables.

```
db.generate_mapping(create_tables=True)
```

Transfert des données Client

Les données sont lues dans le dataFrame `data` sur les quatre attributs définis et insérées dans la base à l'aide du constructeur de la classe Client.

```
clients = data[["CLIENT", "TELEPHONE", "VILLE", "PAYS"]].drop_duplicates()
with orm.db_session:
    for c in clients.values:
        try:
            Client(id_client = c[0], telephone = c[1], ville = c[2], pays =
c[3])
            orm.commit()
        except Exception as e:
            print("*** ERREUR DE TRANSACTION :", e, '***')
```



On remarque que l'initialisation des clients ne porte que sur les attributs élémentaires (la liste des achats n'est pas initialisée explicitement).



Un certain nombre d'erreurs de transaction se produisent. pouvez vous deviner leur origine?

Affichage

Pour afficher la liste de tous les clients (et non le schéma de la classe Client), il faut faire appel à la méthode `select()` qui effectue une lecture dans la base avant l'affichage.

```
Client.select().show()
```

On peut également afficher les clients un par un à l'aide leur index (ici le nom du magasin)

```
print(Client["Land of Toys Inc."])
print(Client["Land of Toys Inc."].id_client)
print(Client["Land of Toys Inc."].ville)
print(Client["Land of Toys Inc."].pays)
print(Client["Land of Toys Inc."].achats)
```

On notera que la liste des achats est vide (les commandes n'ont pas encore été saisies)

L'appel à la méthode `select()` permet de sélectionner les clients selon la valeur d'un ou plusieurs attributs. Cette sélection passe par une fonction anonyme `lambda`:

```
requête = Client.select(lambda c : c.pays == "France")
```

et on affiche le résultat:

```
requête.show()
```

Remarque : une requête se comporte comme un itérateur sur les objets:

```
for c in requête:
    print(c.id_client, c.ville, c.pays)
```

Transfert des données produits

Les produits sont insérés de la même façon que les clients:

```
produits = data[["CODE_PRODUIT", "TYPE_PRODUIT",
"PRIX_UNITAIRE"]].drop_duplicates()
with orm.db_session:
    for p in produits.values:
        try:
            Produit(code_produit = p[0], type_produit = p[1], prix_unitaire
= p[2])
            orm.commit()
```

```
        except Exception as e:
            print("*** ERREUR DE TRANSACTION :", e, '***')
```



Un certain nombre d'erreurs de transaction se produisent. Pouvez vous deviner leur origine?

Question subsidiaire : comment modifier le schéma de départ pour les supprimer.

Affichage du contenu de la classe

```
Produit.select().show()
```

Uniquement les 10 premiers:

```
orm.show(Produit.select()[:10])
```

Affichage d'un produit particulier

```
print (Produit['S10_1678'])
print (Produit['S10_1678'].type_produit)
print (Produit['S10_1678'].prix_unitaire)
print (Produit['S10_1678'].ventes)
```

Transfert des données ventes

Pour créer les commandes, il faut ici définir deux références :

- une référence au client qui a effectué la commande
- une référence au produit commandé

qui sont des objets définis précédemment lors de l'insertion des données client et des donnés produit. Ils correspondent donc à des entrées de leurs classes respectives, indexes par leur identifiant (`id_client` et `code_produit`).

```
ventes = data[["NUM_COMMANDE", "QUANTITE", "MONTANT", "MOIS", "ANNEE",
"CLIENT", "CODE_PRODUIT"]].drop_duplicates()
with orm.db_session:
    for v in ventes.values:
        try:
            client = Client[v[5]]
            produit = Produit[v[6]]
            Commande(num_commande = int(v[0]),
                     code_produit = v[6],
                     quantité = int(v[1]),
```

```
                    montant = float(v[2]),
                    mois = int(v[3]),
                    année = int(v[4]),
                    client = client,
                    produit = produit)
            orm.commit()
        except Exception as e:
            print("*** ERREUR DE TRANSACTION :", e, '***')
```

Affichage

```
Commande.select().show()
```

```
print(Commande[10118,"S700_3505"])
print('Montant :', Commande[10118,"S700_3505"].montant)
print('Quantité :', Commande[10118,"S700_3505"].quantité)
print('Année :', Commande[10118,"S700_3505"].année)
print('Mois :', Commande[10118,"S700_3505"].mois)
print('Client :', Commande[10118,"S700_3505"].client)
print('Produit :', Commande[10118,"S700_3505"].produit)
```

Exemples de requête

```
requête = Commande.select(lambda c : c.montant > 10000)
for r in requête:
    print(r.num_commande, r.quantité, r.mois, r.année, r.client, r.produit)
```

Ou plus simplement :

```
requête.show()
```

Autre écriture

```
requête = orm.select(c for c in Commande if c.montant > 10000)
```

Mise à jour automatique des contenus

Maintenant que les commandes on été entrées dans la base, la liste des achats est à présent renseignée pour chaque client de la classe Client:

```
print(Client["Land of Toys Inc."].achats)
```

ou:

```
Client["Land of Toys Inc."].achats.select().show()
```

Et la liste des ventes est de même renseignée pour chaque produit de la classe Produit:

```
print (Produit['S10_1678'].ventes)
```

Modifier les valeurs

```
Produit['S12_1108'].prix_unitaire = 100
orm.commit()
```

Supprimer un objet

```
Produit['S12_1108'].delete()
orm.commit()
```



** A faire **

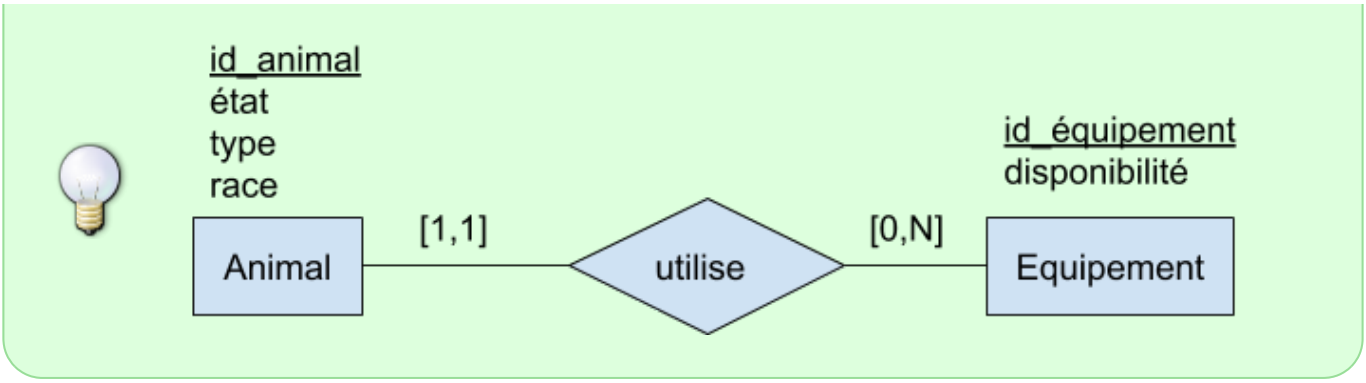
- Pour chaque client, calculer le montant total des achats
- Pour chaque produit, calculer le montant total des ventes
- Corriger le champ `pays` pour les clients nord-américains : si le pays vaut `"United States"`, le remplacer par `"USA"`
- Créez un nouveau client
- Faites-lui commander plusieurs produits (n'oubliez pas de définir le numéro de commande!!)
- Vérifiez que les nouvelles commandes apparaissent bien dans la liste des ventes de la classe `Produits` . Magique, non?

Création d'un schéma de données

- Le but est maintenant de définir un modèle ORM pour le schéma de données du TD1. Reprenez votre programme de gestion de l'animalerie (TD 1 et 2). Le but est de remplacer les fichiers json par une base de données sqlite en utilisant les fonctionnalités de pony pour lire et mettre à jour les données.
- La première étape consiste à définir le modèle de données. Pour conserver les données de l'animalerie, on utilisera le gestionnaire de bases de données `sqlite`. Avant toute chose, il faut définir un schéma de données conforme au modèle relationnel



On part du schéma entité/association suivant:



A FAIRE :

- définissez le schéma relationnel correspondant (sans oublier les clés étrangères)
- traduisez le schéma relationnel en schéma UML
- Ajoutez le nouveau script `data_model.py` à votre projet, et définissez le schéma de données à l'aide des fonctions de pony:

```
from pony import orm

db = orm.Database()

class Equipement(db.Entity):
    ...
class Animal(db.Entity):
    ...
```

- Vous devez maintenant remplir la base à l'aide des données contenues dans animal.json et équipement.json. Pour ce faire, utilisez le script suivant (il ne devra être exécuté qu**une seule fois**).

```
import json

from pony import orm
from data_model import Equipement, Animal, db

db.bind(provider='sqlite', filename='animalerie.db', create_db=True)
db.generate_mapping(create_tables=True)

équipement_data = 'équipement.json'
with open(équipement_data, "r") as f:
    équipement_dict = json.load(f)
    for id_équip in équipement_dict:
        disponibilité = équipement_dict[id_équip]["DISPONIBILITÉ"]
        with orm.db_session:
            try:
                Equipement(id_équip=id_équip, disponibilité=disponibilité)
                orm.commit()
            except:
                print(id_équip, "already exists in database")
                pass
```

```
animal_data = 'animal.json'
with open(animal_data, "r") as f:
    animal_dict = json.load(f)
    for id_animal in animal_dict:
        état = animal_dict[id_animal]["ETAT"]
        type = animal_dict[id_animal]["TYPE"]
        race = animal_dict[id_animal]["RACE"]
        lieu = animal_dict[id_animal]["LIEU"]
        with orm.db_session:
            try:
                Animal(id_animal=id_animal,
                       état=état,
                       type=type,
                       race=race,
                       lieu=Equipement[lieu])
                orm.commit()
            except:
                print(id_animal, "already exists in database")
                pass
```

- Vous disposez maintenant d'une base de données `animalerie.db` dans le répertoire du projet. Cette base contient l'ensemble des informations nécessaires pour gérer l'animalerie. Vous devez maintenant reprendre votre programme de gestion de l'animalerie (TD 1 et 2) et modifier `modele.py` en utilisant les fonctionnalités de `pony` pour lire et mettre à jour les données.

Voici à quoi doit ressembler le début de `modele.py` :

```
from pony import orm
from data_model import Equipement, Animal, db

liste_états = ['affamé', 'fatigué', 'repus', 'endormi']

db.bind(provider='sqlite', filename='animalerie.db')
db.generate_mapping()

def lit_état(id_animal):
    with orm.db_session:
        try:
            return Animal[id_animal].état
        except:
            return None

def lit_lieu(id_animal):
    with orm.db_session:
        try:
            return Animal[id_animal].lieu
        except:
            return None

def vérifie_disponibilité(id_équipement):
```

```
    ...
```

Complétez le code de manière à valider le fichier de tests suivant :

```
import modele
import controleur
from data_model import orm, Equipement, Animal

def test_lit_etat():
    assert modele.lit_état('Tac') == 'affamé'
    assert modele.lit_état('Bob') == None

@orm.db_session
def test_lit_lieu():
    assert modele.lit_lieu('Tac') == Equipement['litière']
    assert modele.lit_lieu('Bob') == None

def test_vérifie_disponibilité():
    assert modele.vérifie_disponibilité('litière') == 'libre'
    assert modele.vérifie_disponibilité('roue') == 'occupé'
    assert modele.vérifie_disponibilité('nintendo') == None

@orm.db_session
def test_cherche_occupant():
    assert Animal['Totoro'] in modele.cherche_occupant('roue')
    assert Animal['Tac'] in modele.cherche_occupant('litière')
    assert Animal['Tac'] not in modele.cherche_occupant('mangeoire')
    assert modele.cherche_occupant('nintendo') == []

def test_change_état():
    modele.change_état('Totoro', 'fatigué')
    assert modele.lit_état('Totoro') == 'fatigué'
    modele.change_état('Totoro', 'excité comme un pou')
    assert modele.lit_état('Totoro') == 'fatigué'
    modele.change_état('Truc', 'fatigué')
    assert modele.lit_état('Truc') == None

@orm.db_session
def test_change_lieu():
    modele.change_lieu('Totoro', 'roue')
    assert modele.lit_lieu('Totoro') == Equipement['roue']
    modele.change_lieu('Totoro', 'nid')
    assert modele.lit_lieu('Totoro') == Equipement['roue']
    modele.change_lieu('Totoro', 'nintendo')
    assert modele.lit_lieu('Totoro') == Equipement['roue']
    modele.change_lieu('Muche', 'litière')
    assert modele.lit_lieu('Muche') == None



@orm.db_session
```

```
def test_nourrir():
    if modele.vérifie_disponibilité('mangeoire') == 'libre' and
modele.lit_état('Tic') == 'affamé':
        controleur.nourrir('Tic')
    assert modele.vérifie_disponibilité('mangeoire') == 'occupé'
    assert modele.lit_état('Tic') == 'repus'
    assert modele.lit_lieu('Tic') == Equipement['mangeoire']
    controleur.nourrir('Pocahontas')
    assert modele.lit_état('Pocahontas') == 'endormi'
    assert modele.lit_lieu('Pocahontas') == Equipement['nid']
    controleur.nourrir('Tac')
    assert modele.lit_état('Tac') == 'affamé'
    assert modele.lit_lieu('Tac') == Equipement['litière']
    controleur.nourrir('Bob')
    assert modele.lit_état('Bob') == None
    assert modele.lit_lieu('Bob') == None
    assert modele.vérifie_disponibilité('mangeoire') == 'occupé'
```

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/public:appro-s7:td3

Last update: **2023/10/16 11:10**