éalisé par : • Marwane Iguider

• Aymen Gnaoui

• Abdssamad Amchar

Gestion Informatique : Application de gestion d'un magasin de composants PC en Python(SQLite3 et Tkinter)

Projet encadré par Monsieur Hain Mustapha

Année universitaire : 2025 / 2026

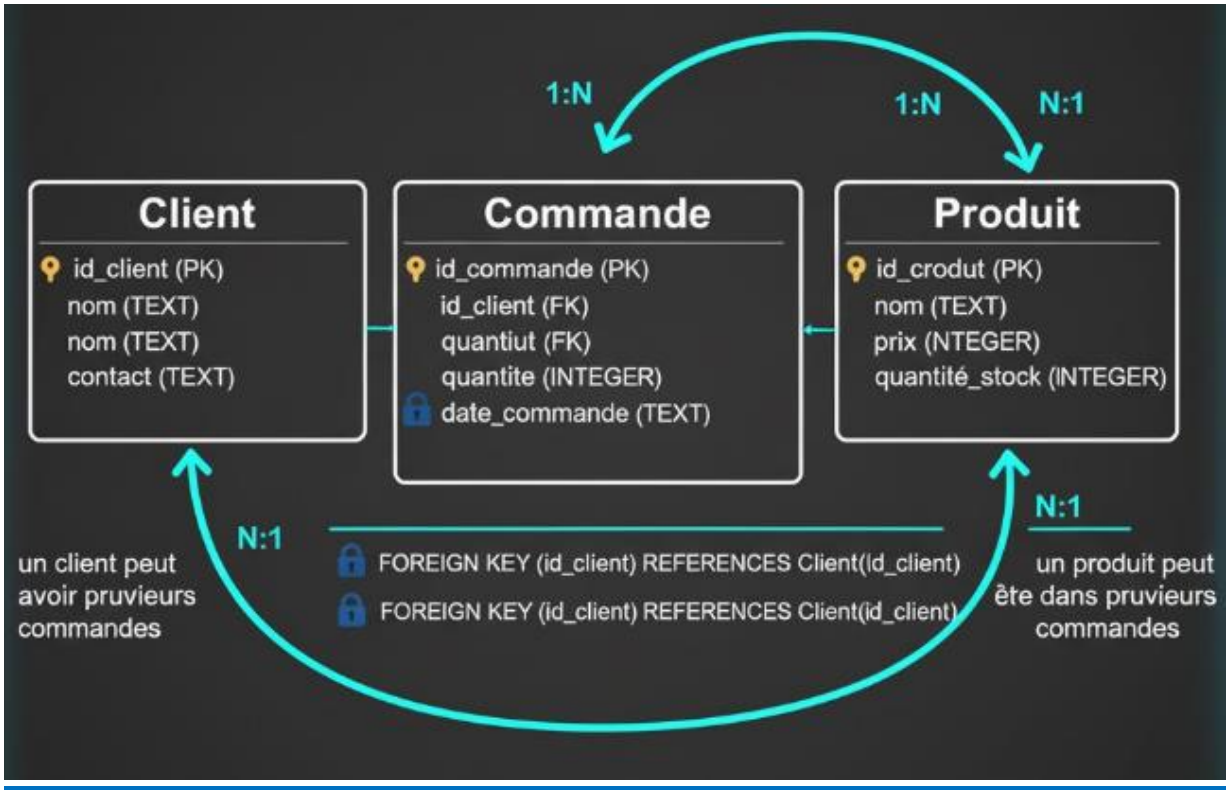
Rapport d'Analyse : Module db.py

Objectif : Couche d'Accès aux Données (Data Access Layer)

L'objectif principal du fichier db.py est de servir de **module** (ou "librairie") centralisé pour toutes les interactions avec la base de données SQLite gestion.db.

La méthodologie de conception repose sur le principe de la **Séparation des Préoccupations** (Separation of Concerns). Le fichier projet.py (l'interface graphique Tkinter) gère l'affichage et l'interaction utilisateur, tandis que db.py gère exclusivement la logique métier et la persistance des données.

1.Initialisation et Schéma de la Base de Données :



La fonction **create_tables** est responsable de la mise en place du schéma de la base de données.

- Méthodologie : La fonction reçoit la connexion **db** en tant qu'argument. C'est une forme d'injection de dépendances : le module n'crée pas sa propre connexion, il utilise celle fournie par l'application principale.

- Robustesse : L'utilisation de **CREATE TABLE IF NOT EXISTS** rend cette fonction idempotente. Elle peut être appelée à chaque démarrage de l'application sans risque d'erreur ou de réécriture des données.
- Intégrité : L'utilisation de **INTEGER PRIMARY KEY AUTOINCREMENT** garantit que chaque enregistrement (Produit, Client, Commande) possède un identifiant unique et non réutilisable, ce qui est crucial pour l'intégrité des clés étrangères (**id_client**, **id_produit**) dans la table Commande

2.Fonctions de Base (Produit & Client) :

L'aspect méthodologique le plus important ici est l'utilisation systématique de **requêtes paramétrées** (l'utilisation de ? et la fourniture des valeurs dans un tuple séparé). Cette technique est la défense standard contre les attaques par **injection SQL**, empêchant un utilisateur de saisir des données malveillantes qui pourraient corrompre la base de données.

3.Logique Métier et Gestion de l'État de la Connexion :

Les fonctions de gestion des commandes **(insert_commande, update_commande, delete_commande)** sont les plus complexes du module.

Ces fonctions sont les plus complexes du module car elles ne prennent pas en entrée des ID (ex: 1, 2), mais les noms fournis par l'utilisateur (ex: "Marwane", "Ram"). Elles ont la responsabilité de *traduire* ces noms en clés étrangères **(id_client**, **id_produit**) en interrogeant les tables respectives avant d'exécuter leur opération

Le Problème db.row_factory :

- Nous avons initialement utilisé **db.row_factory = sqlite3.Row** pour un code plus lisible.
- **Le Bug :** Cette instruction modifiait l'état de la connexion **db** de façon permanente. Lorsque la fonction **updateTable** de l'interface graphique était appelée pour rafraîchir le tableau, elle recevait des **objets sqlite3.Row** au lieu des **tuples** simples attendus.
- **La Conséquence :** Le **Treeview** de **Tkinter** ne savait pas comment afficher ces objets, ce qui provoquait un bug d'affichage majeur.
- **La Solution Unifiée :** Nous avons supprimé **db.row_factory** de toutes les fonctions. Cela garantit que la connexion **db** renvoie toujours des tuples, assurant la stabilité et la compatibilité avec la fonction **updateTable** de l'interface.

4.Modularité et Tests :

Cette section démontre notre méthodologie de test en utilisant le bloc **if __name__ == "__main__":**. Cette convention Python essentielle nous a permis d'utiliser db.py à la fois comme une **librairie** (lorsqu'il est importé par projet.py, le bloc est ignoré) et comme un **script** (lorsqu'il est exécuté directement pour les tests). Cela a facilité le **développement et les tests isolés**, nous permettant de valider chaque fonction de base de données indépendamment de l'interface graphique. Pour garantir l'intégrité des données, nous avons utilisé une base de données de test (**gestion_test.db**) afin d'exécuter des opérations sans jamais risquer d'altérer la base de données (**gestion.db**).

Interface(utilisant tkinter):

Initialisation de la Base de Données et de la Fenêtre Principale :

- **Tkinter(*etttk)**: Bibliothèque standard pour la création de l'interface graphique(GUI).

- **SQLite3**:Module pour l'interface avec la base de données relationnelle locale.

- **dbasdb_operations**: Module local personnalisé contenant la logique métier(fonctions de manipulation de la base de données).

Fonction de Mise à Jour du Tableau (updateTable) :

La fonction **updateTable** est responsable d'actualiser le contenu des **Treeview**.

- Nettoyage : Elle vide d'abord le tableau de toutes ses lignes existantes (**trv.delete**).

- Requête : Elle exécute une requête SQL. Pour la table 'Commande', une jointure (JOIN) est utilisée pour récupérer les noms des clients et des produits. Pour les autres, **une simple requête SELECT** * est utilisée.

- Remplissage : Les résultats (**data.fetchall()**) sont insérés ligne par ligne dans le **Treeview**.

```python
def updateTable(trv, table):

    for item in trv.get_children():
        trv.delete(item)

    if table == 'Commande':
        query = """
        SELECT
            Commande.id_commande,
            Client.nom,
            Produit.nom,
            Commande.quantite,
            Commande.date_commande
        FROM Commande
        JOIN Client ON Commande.id_client = Client.id_client
        JOIN Produit ON Commande.id_produit = Produit.id_produit
        """
        data = db.execute(query)

        trv.heading(2, text="Client")
        trv.heading(3, text="Produit")

    else:

        data = db.execute(f'SELECT * FROM {table}')

    for row in data.fetchall():
        trv.insert('',END,values=row)
```

Personnalisation du Style des Tableaux (Treeview) :

Pour assurer une cohérence visuelle, le moteur de thèmes ttk.Style est utilisé. Des polices personnalisées sont appliquées globalement aux widgets Treeview (tableaux) :

- **Treeview.Heading** : Style pour les en-têtes de colonnes.
- **Treeview** : Style pour les lignes de données dans le corps du tableau.

```python
style=ttk.Style()
style.configure("Treeview.Heading",font=("courrier",12))
style.configure("Treeview",font=("courrier",11))
```

Définition des Opérations CRUD et des Boutons d'Action :

La logique métier (CRUD) est implémenté dans des fonctions :

- **ajouter_produit** : Récupère les données des Entry et appelle db_operations.insert_produit.
- **modifier_produit** : Récupère les données et appelle db_operations.update_produit.

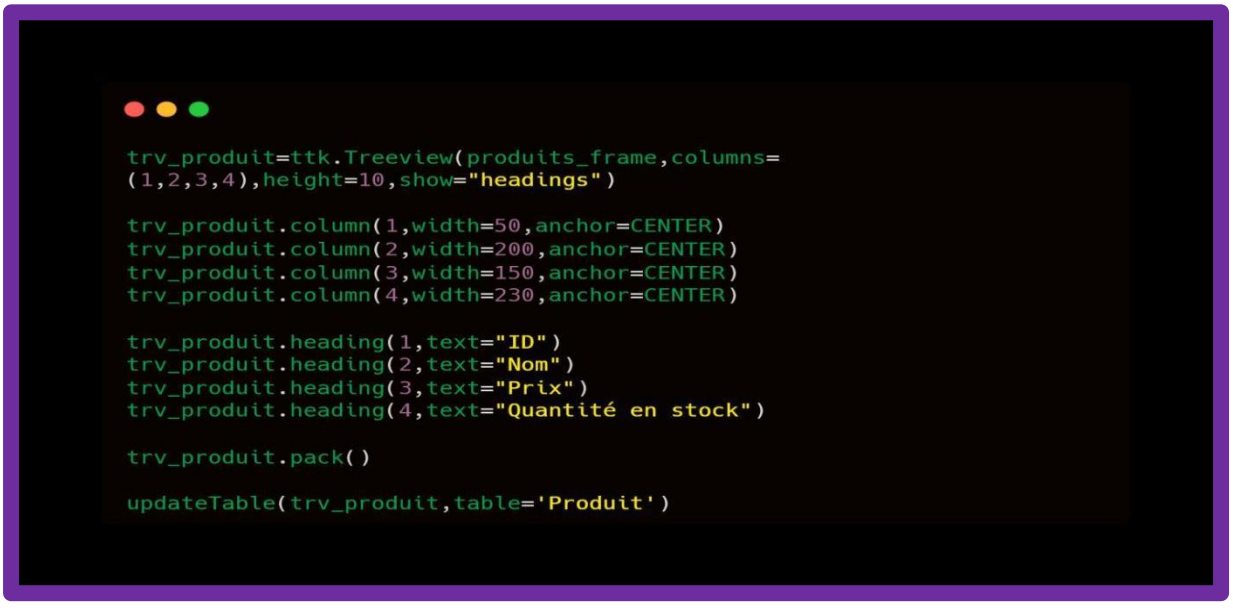
- **supprimmer_produit** : Récupère le nom et appelle db_operations.delete_produit.

Chaque fonction appelle updateTable pour rafraîchir l'affichage. Des Button sont créés et liés à ces fonctions via leur attribut command.

```
def ajouter_produit():
    nom = prod_entry1.get()
    quantite = int(prod_entry2.get())
    prix = float(prod_entry3.get())

    db_operations.insert_produit(db,nom,prix,quantite)


    updateTable(trv_produit,table='Produit')
def modifier_produit():
    nom = prod_entry1.get()
    quantite = int(prod_entry2.get())
    prix = float(prod_entry3.get())

    db_operations.update_produit(db,nom,prix,quantite)


    updateTable(trv_produit,table='Produit')
def supprimmer_produit():
    nom = prod_entry1.get()

    db_operations.delete_produit(db,nom)


    updateTable(trv_produit,table='Produit')

prod_bouton1= Button(form_produit,text="Ajouter un produit",font=
("fourrier",13),width=20,pady=10,command=ajouter_produit)
prod_bouton1.grid(column=0,row=4,pady=10,padx=10)
prod_bouton2 = Button(form_produit,text="Modifier un produit",font=
("fourrier",13),width=20,pady=10,command=modifier_produit)
prod_bouton2.grid(column=1,row=4,pady=10,padx=10)
prod_bouton3 = Button(form_produit,text="Supprimmer un produit",font=
("fourrier",13),width=20,pady=10,command=supprimmer_produit)
prod_bouton3.grid(column=2,row=4,pady=10,padx=10)
```

Création et Remplissage du Tableau "Produits" :

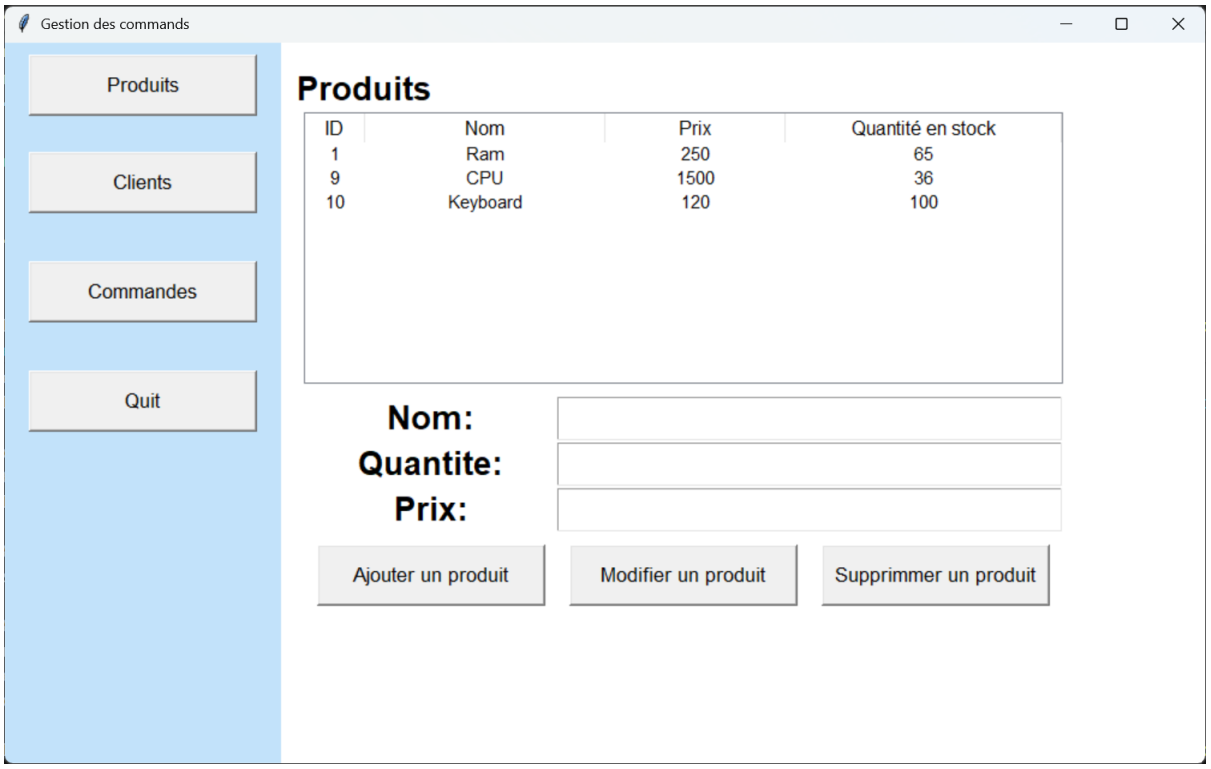
Ce bloc de code est responsable de l'affichage de la liste des produits dans un widget **Treeview** (un tableau) au sein de la "page" des produits



```
trv_produit=ttk.Treeview(produits_frame,columns=
(1,2,3,4),height=10,show="headings")

trv_produit.column(1,width=50,anchor=CENTER)
trv_produit.column(2,width=200,anchor=CENTER)
trv_produit.column(3,width=150,anchor=CENTER)
trv_produit.column(4,width=230,anchor=CENTER)

trv_produit.heading(1,text="ID")
trv_produit.heading(2,text="Nom")
trv_produit.heading(3,text="Prix")
trv_produit.heading(4,text="Quantité en stock")

trv_produit.pack()

updateTable(trv_produit,table='Produit')
```

Présentation générale de l’application :

L’interface graphique de l’application de gestion de stock a été réalisée avec le module **Tkinter** de Python. Elle permet à l’utilisateur d’interagir facilement avec la base de données sans connaître le langage SQL. L’objectif est de rendre la gestion des produits, clients et commandes **intuitive et rapide**.



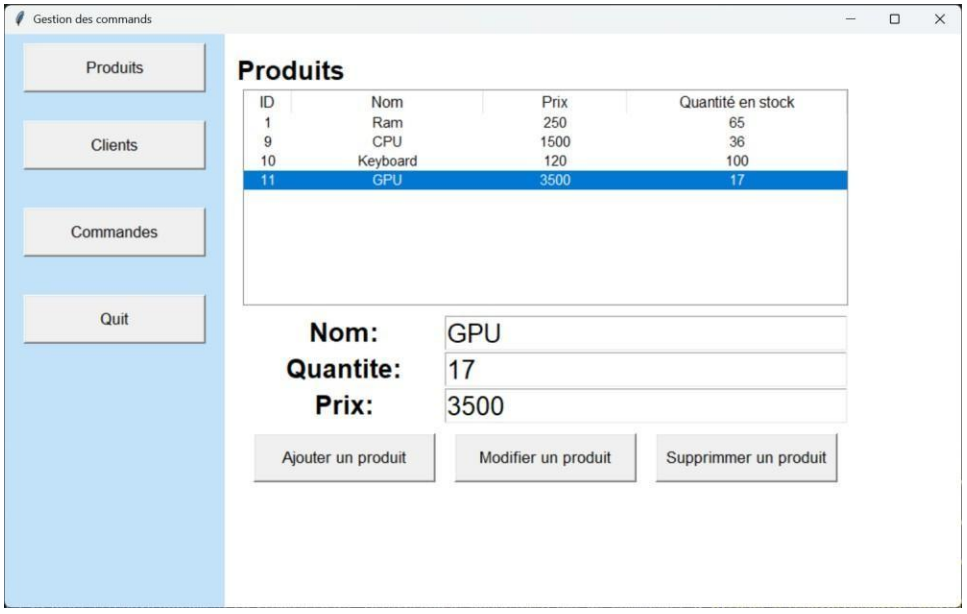
L’application s’ouvre sur une **fenêtre principale** intitulée “Gestion de stock”.
Sur la partie gauche, on trouve un **menu vertical** composé de trois boutons :

- **Produits** : pour gérer la liste des produits en stock.
- **Clients** : pour ajouter ou modifier les clients.
- **Commandes** : pour enregistrer les commandes passées.

Le reste de la fenêtre affiche la section correspondante selon le bouton sélectionné.

Chaque section possède un **tableau** qui présente les données sous forme de liste claire et triée.

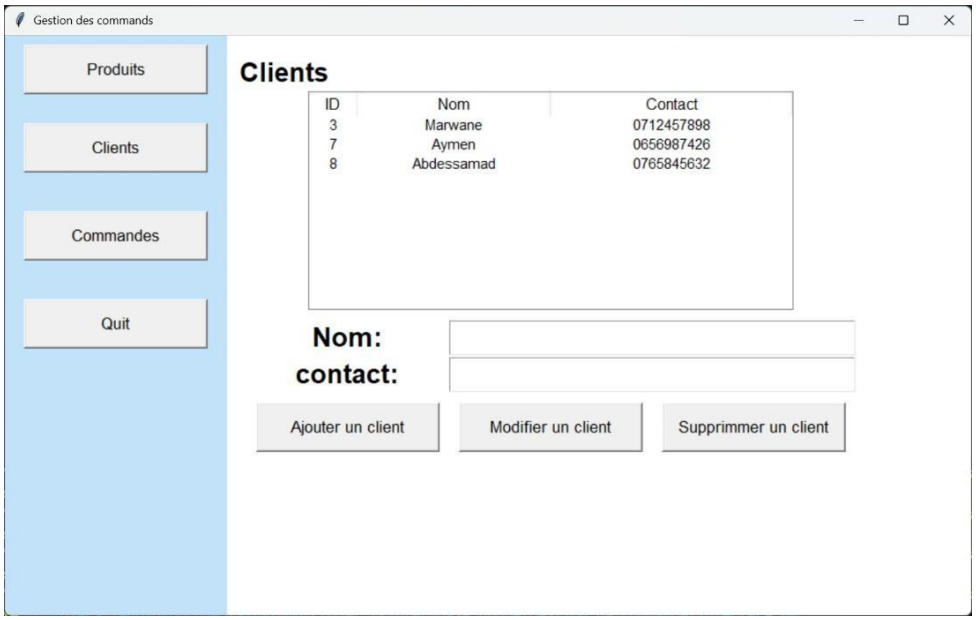
1.Fonctionnement de la fenêtre “Produits” :



La fenêtre “Produits” permet de consulter, ajouter, modifier ou supprimer un produit du stock.
Elle affiche un tableau listant les colonnes suivantes :

- **Nom**
- **Prix**
- **Quantité en stock**

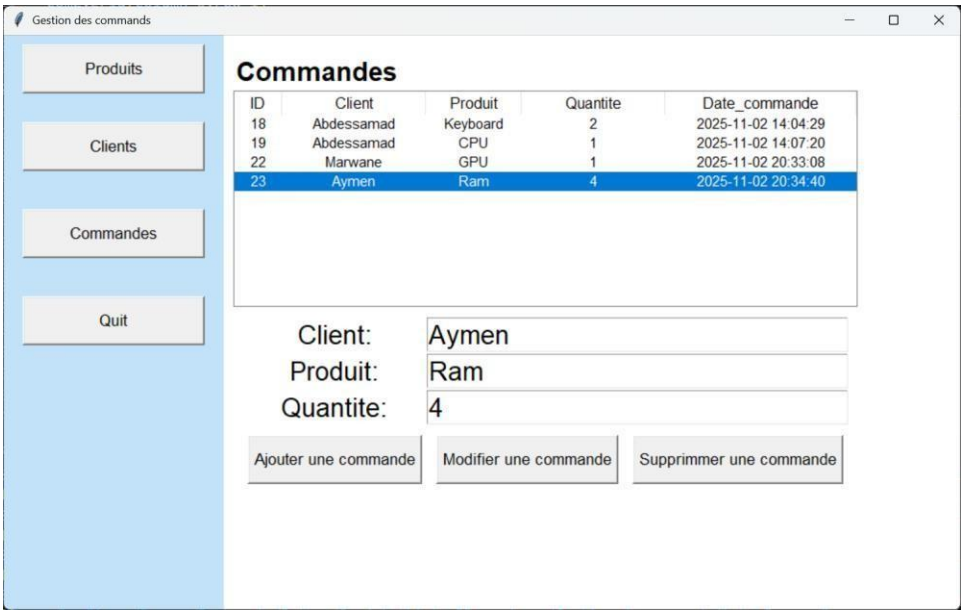
2. Fonctionnement de la fenêtre “Clients” :



La fenêtre “Clients” contient un tableau affichant le nom et le contact des clients.
L’utilisateur peut ajouter un nouveau client via les champs de saisie situés en bas.
Les boutons “Modifier” et “Supprimer” permettent respectivement d’éditer ou de retirer un client.

Ces informations sont stockées dans la table ‘Client’ de la base de données SQLite.

3. Fonctionnement de la fenêtre “Commandes” :



La fenêtre “Commandes” relie les tables **Produit** et **Client**.
L’utilisateur sélectionne un client, un produit et indique la quantité commandée.
Le bouton “Valider la commande” :

- Enregistre la commande dans la table Commande,
- Met à jour automatiquement la quantité du produit dans Produit,
- Et affiche la commande dans le tableau des commandes.