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.

```python
# ici creer les tableaux
import sqlite3

def create_tables(db):
    db.execute("create table if not exists Produit(id_produit INTEGER PRIMARY KEY AUTOINCREMENT, nom text, prix integer, quantite_stock integer)")
    db.execute("create table if not exists Client(id_client INTEGER PRIMARY KEY AUTOINCREMENT, nom text,contact text)")
    db.execute("create table if not exists Commande(id_commande INTEGER PRIMARY KEY AUTOINCREMENT, id_client INTEGER, id_produit INTEGER, quantite integer, date_commande text)")
    db.commit()
```

1. Initialisation et 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.

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.

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 relations (clés étrangères **id_client**, **id_produit** dans la table Commande).

2.Fonctions de Base (Produit & Client) :

```python
# fonction produit
def insert_produit(db,nom,prix,qte):
    db.execute("insert into Produit(nom,prix,quantite_stock) values(?,?,?)",(nom,prix,qte))
    db.commit()

def update_produit(db,nom,prix,qte):
    db.execute("update Produit set prix=? , quantite_stock=? where nom=?", (prix,qte,nom))
    db.commit()

def delete_produit(db,nom):
    db.execute("delete from Produit where nom=?", (nom,))
    db.commit()
# fonction client
def insert_client(db,nom,contact):
    db.execute("insert into Client(nom,contact) values(?,?)",(nom,contact))
    db.commit()

def update_client(db,nom,contact):
    db.execute("update Client set contact=? where nom=? ",(contact,nom))
    db.commit()

def delete_client(db,nom):
    db.execute("delete from Client where nom=?",(nom,))
    db.commit()
```

Cette section illustre les opérations fondamentales **(Inserer, Mettre a jour, Supprimer)** pour les entités simples "**Produit**" et "**Client**".

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

```python
# fonction commande
def insert_commande(db,nom_client,nom_produit,quantite):
    # (On a supprimé db.row_factory)
    cursor_produit = db.execute("select id_produit from Produit where nom=?",(nom_produit,))
    cursor_client = db.execute("select id_client from Client where nom=?",(nom_client,))
    cursor_datetime = db.execute("SELECT datetime('now')")

    row_produit = cursor_produit.fetchone()
    row_client = cursor_client.fetchone()
    row_datetime = cursor_datetime.fetchone() #retourne un tuple = ('datetime',)
    db.execute("insert into Commande(id_client,id_produit,quantite,date_commande) values(?,?,?,?)",(row_client[0],row_produit[0],quantite,row_datetime[0]))
    db.commit()
```

Cette image, montrant la fonction **insert_commande**, est **représentative de l'ensemble des fonctions de gestion des commandes** (**insert_commande**, **update_commande**, **delete_commande**).

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 :

```python
#Tests:
if __name__ == "__main__":
    # Ce code ne s'exécute QUE si tu lances db.py directement

    # On crée une connexion juste pour les tests
    db_test = sqlite3.connect("gestion_test.db")
    create_tables(db_test)
    insert_produit(db_test,"Ram",400,1)
    update_produit(db_test,"Ram",600,2)
    insert_produit(db_test,"CPU",400,1)


    insert_client(db_test,"Aymen","0615478454")
    insert_client(db_test,"Abdessamad","0798653256")
    insert_client(db_test,"Marwane","0712457898")
    update_client(db_test,"Abdessamad","0698989898")
    delete_client(db_test,"Aymen")

    insert_commande(db_test,"Marwane","CPU",1)
    insert_commande(db_test,"Abdessamad","Ram",1)
    update_commande(db_test,"Marwane","CPU",5)
    delete_commande(db_test,"Abdessamad","Ram",1)
```

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):

```
from tkinter import *
from tkinter import ttk
import sqlite3
import db as db_operations
```

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

- **Tkinter(*etttk)**:Bibliothèquestandardpourlacréationdel'interfacegraphique(GUI).
- **SQLite3**:Modulepourl'interfaceaveclabasededonnéesrelationnellelocale.
- **dbasdb_operations**:Modulelocalpersonnalisécontenantlalogiqu emétier(fonctionsde manipulation de la base de données).

```
db=sqlite3.connect("gestion.db")

window = Tk()

window.title("Gestion des commands")
window.geometry("1000x600")
window.minsize(1000,600)
window.config(background="#FFFFFF")
```

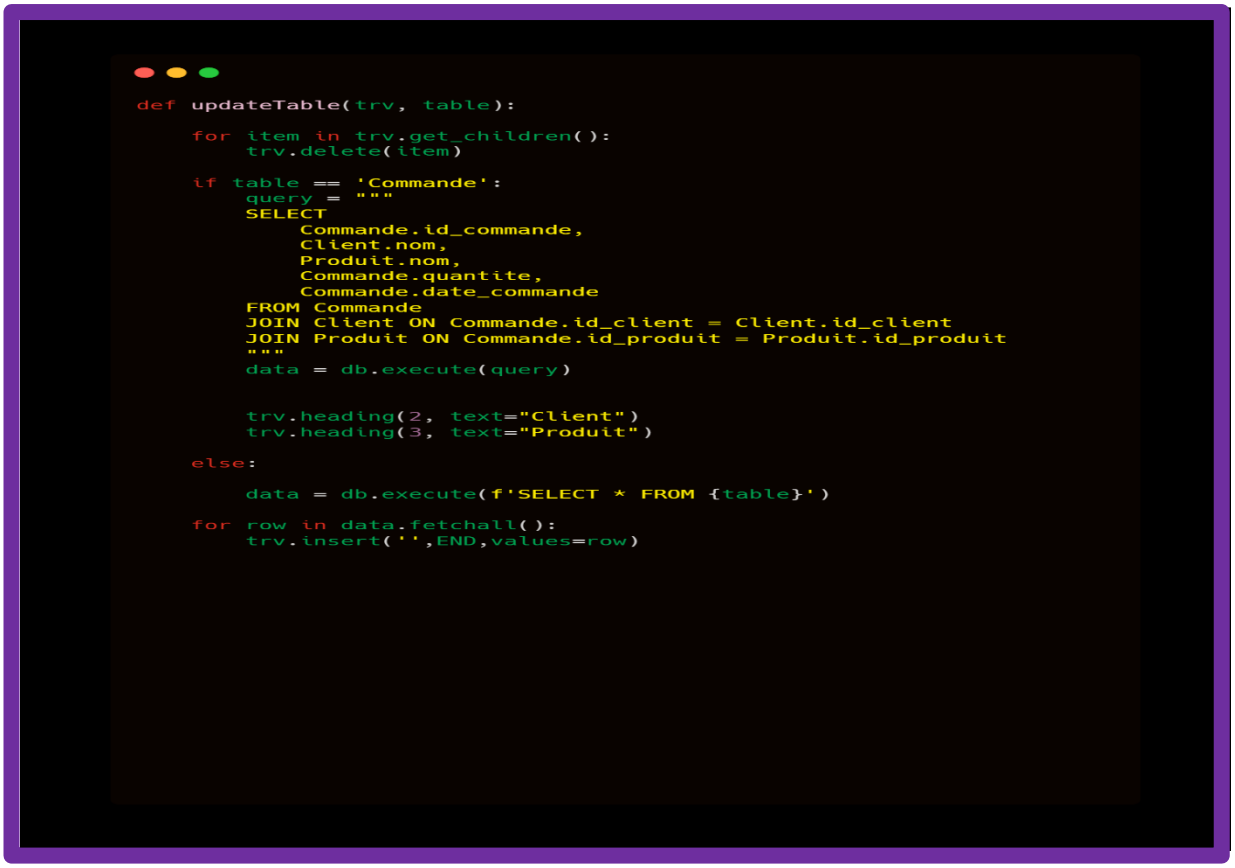
Fonction de Mise à Jour du Tableau (updateTable) :

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)
```

Création du Menu de Navigation Latéral :

La navigation entre les "pages" est gérée par la fonction show_frame. Elle masque tous les cadres de contenu (.forget()) avant d'afficher le cadre cible (.pack()).

Le menu latéral (menu_frame) contient les boutons de navigation. L'attribut command de chaque bouton utilise lambda pour appeler show_frame avec le cadre approprié (ex: produits_frame). Un bouton "Quit" est lié à window.quit.

```
menu_frame = Frame(window,width=200,height=500,bg="#C2E2FA")
menu_frame.pack(side=LEFT,fill="y")

produits_Btn = Button(menu_frame,text="Produits",font=("fourrier",13),
width=20 , pady=10 , command=lambda:show_frame(produits_frame))
produits_Btn.grid(row=1 , column=1 , padx=20,pady=10)

produits_Btn = Button(menu_frame,text="Clients",font=("fourrier",13),
width=20 , pady=10 , command=lambda:show_frame(clients_frame))
produits_Btn.grid(row=2 , column=1 , padx=20,pady=20)

produits_Btn = Button(menu_frame,text="Commandes",font=("fourrier",13),
width=20 , pady=10 , command=lambda:show_frame(commandes_frame))
produits_Btn.grid(row=3 , column=1 , padx=20,pady=20)

produits_Btn = Button(menu_frame,text="Quit",font=("fourrier",13),
width=20 , pady=10 , command=window.quit)
produits_Btn.grid(row=4 , column=1 , padx=20,pady=20)
```

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.

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

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')
```

Création du Formulaire de Saisie "Produits" :

Un Frame (form_produit) est créé pour contenir le formulaire de saisie. Il utilise une disposition en grille (.grid()) :

- **Colonne 0** : Contient les étiquettes Label ("Nom:", "Quantite:", "Prix:").
- **Colonne 1** : Contient les champs de saisie Entry.

La configuration grid_columnconfigure(1, weight=1) et l'option sticky="ew" sur les champs Entry assurent que les champs de saisie s'étirent horizontalement pour remplir l'espace.

```
form_produit = Frame(produits_frame,bg="white")
form_produit.grid_columnconfigure(1, weight=1) # les widget dans la
colonne 1 vont pouvoir prendre tout l'espace disponible
form_produit.pack(padx=10,pady=10,fill="x")

com1=Label(form_produit,text="Nom:",font=("courrier", 20, "bold") ,
background="#ffffff")
com1.grid(column=0,row=0)
com2=Label(form_produit,text="Quantite:",font=("courrier", 20, "bold") ,
background="#ffffff")
com2.grid(column=0,row=1)
com3=Label(form_produit,text="Prix:",font=("courrier", 20, "bold") ,
background="#ffffff")
com3.grid(column=0,row=2)

prod_entry1 = Entry(form_produit,font=("courrier", 20, "bold") ,
background="#ffffff")
prod_entry1.grid(column=1,row=0,sticky="ew", columnspan=2) # columnspan
dit au grid de sétirer sur la colonne 2
prod_entry2 = Entry(form_produit,font=("courrier", 20, "bold") ,
background="#ffffff")
prod_entry2.grid(column=1,row=1,sticky="ew", columnspan=2)
prod_entry3 = Entry(form_produit,font=("courrier", 20, "bold") ,
background="#ffffff")
prod_entry3.grid(column=1,row=2,sticky="ew", columnspan=2)
```

Définition des Opérations CRUD et des Boutons d'Action :

La logique métier (CRUD) est implémentée 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)
```

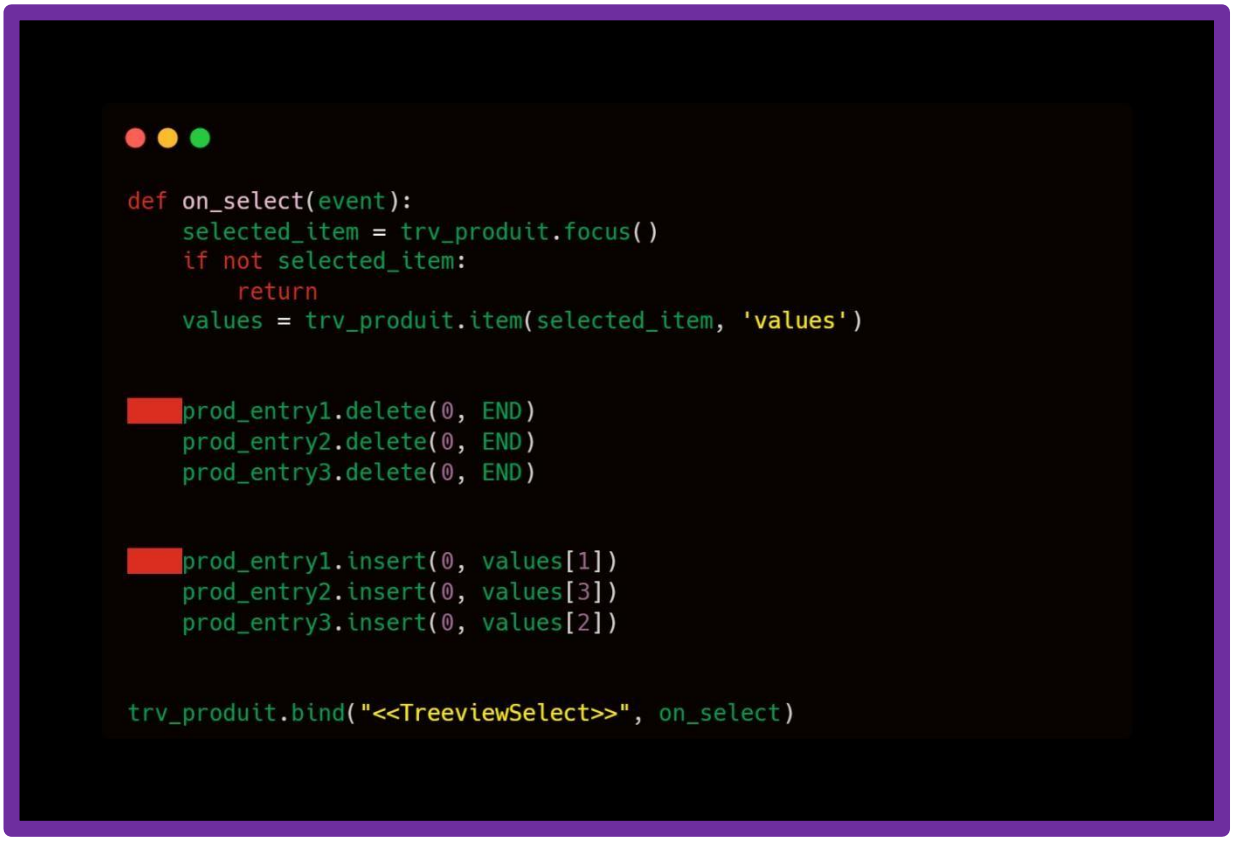
Remplissage Automatique du Formulaire lors de la Sélection

Une fonction on_select gère l'ergonomie du formulaire.

1. **Capture** : Elle récupère l'item sélectionné (.focus()) et ses valeurs (.item()).

2. **Mise à jour** : Elle vide d'abord les champs Entry (.delete()), puis les remplit (.insert()) avec les valeurs de la ligne sélectionnée.

Cette fonction est activée par trv_produit.bind("<<TreeviewSelect>>", on_select), qui connecte l'événement de sélection du tableau à la fonction.



```
def on_select(event):
    selected_item = trv_produit.focus()
    if not selected_item:
        return
    values = trv_produit.item(selected_item, 'values')

    prod_entry1.delete(0, END)
    prod_entry2.delete(0, END)
    prod_entry3.delete(0, END)

    prod_entry1.insert(0, values[1])
    prod_entry2.insert(0, values[3])
    prod_entry3.insert(0, values[2])

trv_produit.bind("<<TreeviewSelect>>", on_select)
```

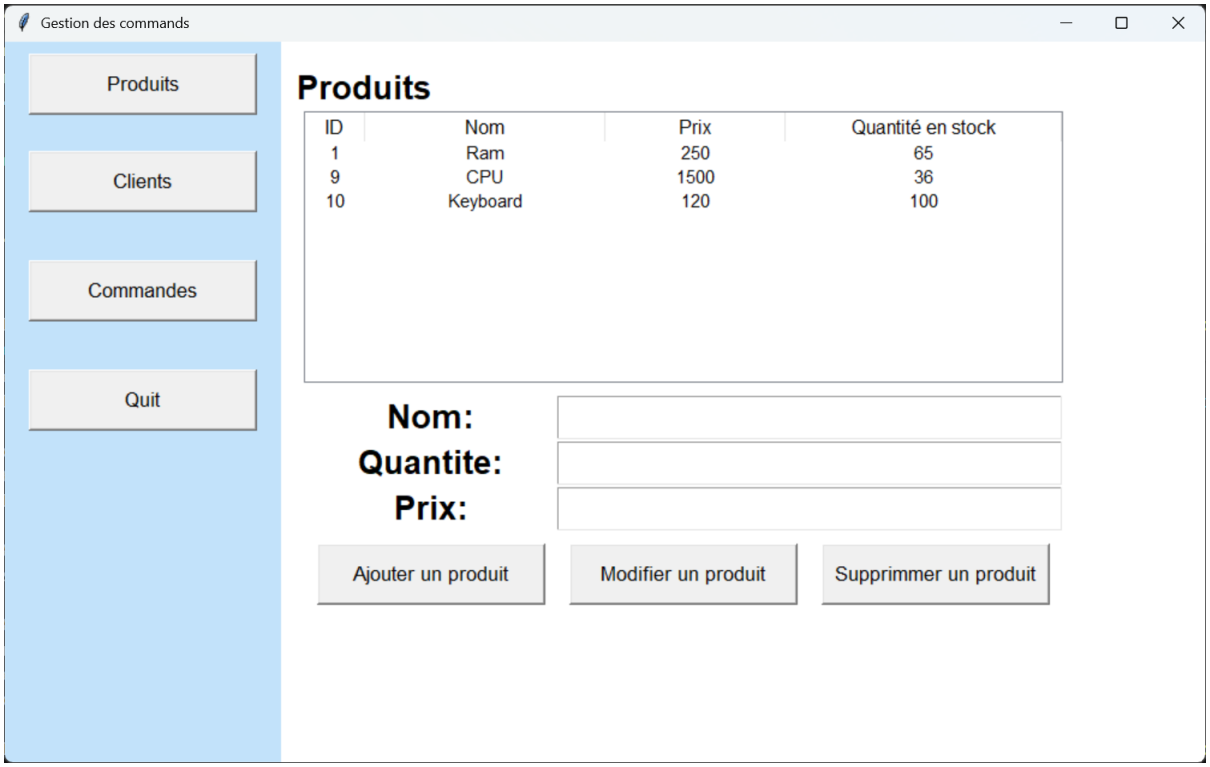
Duplication de la Logique pour les Sections "Clients" et "Commandes" :

La structure et la logique implémentées pour la gestion de la section "Produits" ont servi de modèle pour le développement des sections "**Clients**" et "**Commandes**".

Cette réutilisation du même *pattern* de conception accélère le développement et garantit une expérience utilisateur uniforme à travers les différentes sections de gestion.

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**.

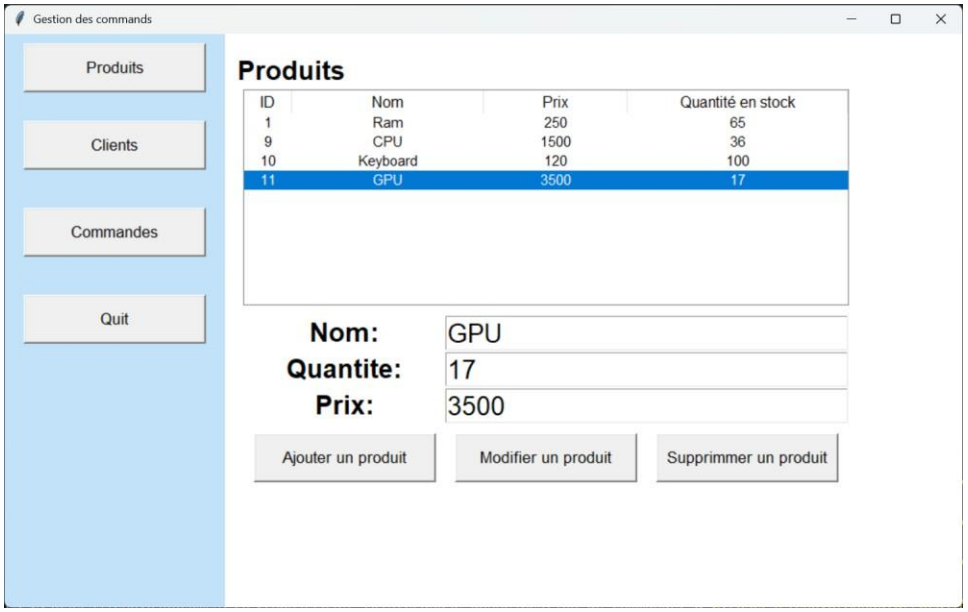


L’application s’ouvre sur une **fenêtre principale** intitulée “Gestion de stock”.
Sur la partie gauche, on trouve un **menu vertical** composé de trois boutons :

- **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.

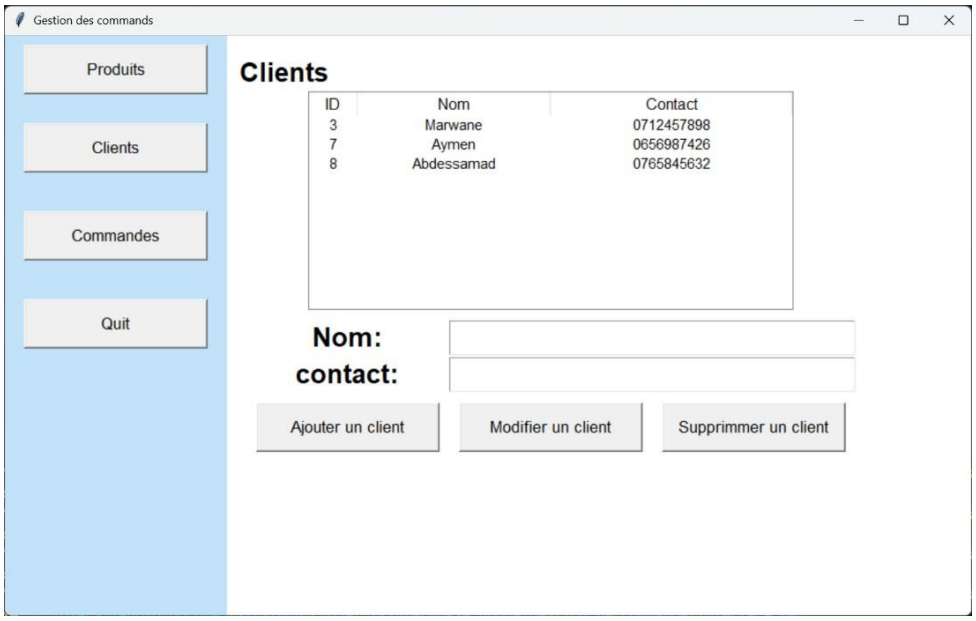
1. Fonctionnement de la fenêtre “Produits” :



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**

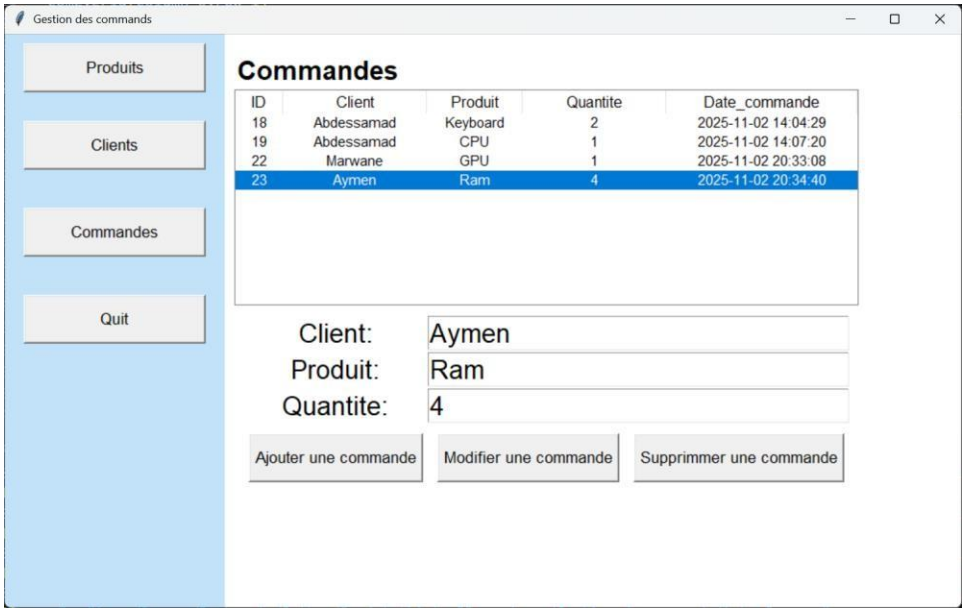
2. Fonctionnement de la fenêtre “Clients” :



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.