ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA

Rapport du projet python avancée :

SYSTÈME DE GESTION BANCAIRE

Réaliser par : Zakaria Achbani – Aymen Azzahidi

Niveau : IAGI-1

Encadrer par : Prof. MUSTAPHA HAIN

Année universitaire : 2025/2026

Sommaire

Introduction:

Le projet **Système de Gestion Bancaire** a pour objectif de créer une application permettant de gérer les opérations d'une banque de manière simple et efficace.

Il a été développé en **Python**, utilisant :

- **Tkinter** pour l'interface graphique.
- **SQLite** pour stocker les données clients, comptes et transactions.

L'application permet :

- La gestion des clients et comptes.
- La réalisation des transactions (dépôt et retrait).
- La consultation du solde.

L'objectif principal est de fournir une interface **intuitive, moderne et fonctionnelle**.

Le projet est structuré en trois parties principales **(Tables)** :

1. Gestion des clients :
   - Ajouter un client.
   - Afficher tous les clients dans une Listbox.
2. Gestion des comptes :
   - Créer un compte pour un client existant.
   - Consulter le solde du compte.
3. Gestion des transactions :
   - Dépôt et retrait avec vérification du solde.
   - Transfert entre deux comptes avec mise à jour des soldes et enregistrement des transactions.

○ Conception de la base de données :

✓ Tables principales :

1. Client :

Champ	Type	Description
Id_client	TEXT	Identifiant unique client
Nom	TEXT	Nom du client
Prénom	TEXT	Prénom du client

2. Compte :

Champ	Type	Description
Id_comte	TEXT	Identifiant unique compte
Id_client	TEXT	Lien vers le client
Solde	REAL	Solde actuel du compte

3. Transactions :

Champ	Type	Description
Id_trans	INTEGER	Identifiant unique (auto-incrémenté)
Id_compte	TEXT	Compte lié à la transaction
Type	REAL	Date et heure de la transaction

✓ Diagramme relationnel :

un client peut avoir plusieurs compte, mais un compte est correspond à un seul client.

De même dans un compte il peut se produire plusieurs transactions, mais une transaction est correspond à un seul compte

- Client (1, n) ----------→Compte(1,1)
- Compte (1, n) ----------→Transactions(1,1)

Description des fonctionnalités :

1. Ajouter un client :

- Saisie de l'ID, nom et prénom.
- Vérification de l'unicité de l'ID client.
- Ajout à la base SQLite.

- Mise à jour automatique de la liste des clients.

2. **Créer un compte :**
   - Vérification que le compte existe.
   - Vérification que le solde est suffisant pour les retraits.
   - Mise à jour du solde dans la table compte.
   - Enregistrement de la transaction dans transactions.
3. **Consultation du solde :**
   - Saisie de l'ID du compte.
   - Affichage du solde actuel du compte.

- **Explication du code :**

✓ **Initialisation de la base de données :**

Le code suivant initialise la base et crée les tables si elles n'existent pas :

```
def init_db():
    conn = sqlite3.connect("banque.db")
    c = conn.cursor()

    c.execute('''CREATE TABLE IF NOT EXISTS client(
                    id_client TEXT PRIMARY KEY,
                    nom TEXT,
                    prenom TEXT)''')

    c.execute('''CREATE TABLE IF NOT EXISTS compte(
                    id_compte TEXT PRIMARY KEY,
                    id_client TEXT,
                    solde REAL DEFAULT 0,
                    FOREIGN KEY(id_client) REFERENCES client(id_client))''')

    c.execute('''CREATE TABLE IF NOT EXISTS transactions(
                    id_trans INTEGER PRIMARY KEY AUTOINCREMENT,
                    id_compte TEXT,
```

Explication:

- **Connexion à SQLite** via sqlite3.connect.
- **Création des tables** avec vérification IF NOT EXISTS pour éviter les doublons.
- **Clés primaires et étrangères** pour assurer l'intégrité des données.

- **Transactions** : ID auto-incrémenté et date par défaut avec CURRENT_TIMESTAMP.

✓ Les fonctions de gestions :

Les fonctions de gestion de projets (ajouter_client, afficher_clients, creer_compte, effectuer_transaction) permettent de gérer la base de données via l'interface graphique, en ajoutant des clients, en créant des comptes et en réalisant des transactions de manière simple et sécurisée.

On prend par exemple la fonction :***ajouter_client***

```
def ajouter_client():
    idc = entry_id_client.get()
    nom = entry_nom.get()
    prenom = entry_prenom.get()
    if not idc or not nom or not prenom:
        messagebox.showwarning("Erreur", "Tous les champs sont obligatoires.")
        return

    conn = sq  (method) def cursor(factory: None = None) -> Cursor
    c = conn.cursor()
    try:
        c.execute("INSERT INTO client VALUES (?,?,?)", (idc, nom, prenom))
        conn.commit()
        messagebox.showinfo("Succès", "Client ajouté avec succès.")
        afficher_clients()
    except sqlite3.IntegrityError:
        messagebox.showerror("Erreur", "ID client déjà existant.")
    conn.close()
```

Fonctionnement détaillé :

1. **Récupération des valeurs** depuis l'interface avec entry.get().
2. **Validation des champs** : si un champ est vide, un message d'avertissement apparaît.
3. **Connexion à la base** avec sqlite3.connect.
4. **Insertion dans la table client** avec c.execute().
5. **Gestion des erreurs** : si l'ID existe déjà, une exception IntegrityError est levée.
6. **Mise à jour de l'interface** via afficher_clients().
7. **Fermeture de la connexion** à la base de données.

Les autres fonctions comme creer_compte() ou effectuer_transaction() suivent **la même logique** : récupérer les entrées, vérifier les conditions, interagir avec la base, gérer les erreurs, et mettre à jour l'interface.

✓ Changement de pages avec show_frame(frame_func):

```
def show_frame(frame_func):
    for widget in content_frame.winfo_children():
        widget.destroy()
    frame_func()
```

Cette fonction sert à **changer le contenu affiché dans l'interface principale**. Au lieu d'ouvrir une nouvelle fenêtre, elle remplace simplement ce qui est déjà affiché par la nouvelle page.

✓ Les fonctions de pages :

Les fonctions de pages(accueil_page, clients_page, comptes_page, transactions_page) permettent de gérer l'affichage de l'interface, en changeant dynamiquement le contenu selon la section sélectionnée (clients, comptes ou transactions) sans quitter la fenêtre principale.

On prend l'exemple de la fonction ***clients_page :***

```
def clients_page():
    global entry_id_client, entry_nom, entry_prenom, listbox_clients
    tk.Label(content_frame, text="Gestion des Clients", font=("Arial", 14, "bold"), bg="#f4fdff").pack(pady=10)
    entry_id_client = tk.Entry(content_frame, width=30)
    entry_nom = tk.Entry(content_frame, width=30)
    entry_prenom = tk.Entry(content_frame, width=30)

    for label_text, entry in [("ID Client", entry_id_client),
                              ("Nom", entry_nom),
                              ("Prénom", entry_prenom)]:
        tk.Label(content_frame, text=label_text, bg="#f4fdff").pack()
        entry.pack(pady=3)

    tk.Button(content_frame, text="Ajouter Client", bg="#0078d7", fg="white",
              command=ajouter_client).pack(pady=8)
    listbox_clients = tk.Listbox(content_frame, width=45)
    listbox_clients.pack(pady=10)
    afficher_clients()
```

Fonctionnement détaillé de clients_page() :

1. **Préparation de la page** : on vide le contenu précédent du content_frame pour afficher uniquement la section Clients.
2. **Création des widgets** : on crée les Entry pour l'ID, le nom et le prénom du client, ainsi que les Label correspondants.

3. **Affichage dynamique** : chaque Label et Entry est ajouté à la page avec un espacement vertical pour la lisibilité.
4. **Bouton d'action** : le bouton "Ajouter Client" est configuré pour appeler la fonction ajouter_client() lorsqu'on clique dessus.
5. **Liste des clients** : une Listbox est créée pour afficher tous les clients présents dans la base de données.
6. **Chargement initial des clients** : la fonction afficher_clients() est appelée pour remplir la Listbox avec les clients existants.
7. **Gestion globale des variables** : les Entry et la Listbox sont déclarés global pour être accessibles depuis d'autres fonctions qui interagissent avec la base de données.

Les autres pages comme comptes_page() et transactions_page() suivent la même logique : préparer la zone de contenu, créer les widgets, configurer les boutons pour appeler les fonctions correspondantes, et afficher les données existantes depuis la base de données.

✓ **Menu latéral :**

```
menu_frame = tk.Frame(root, bg="#0078d7", width=180)
menu_frame.pack(side="left", fill="y")

tk.Label(menu_frame, text="MENU", bg="#0078d7", fg="white",
         font=("Arial", 14, "bold")).pack(pady=20)

for text, func in [("Accueil", accueil_page),
                   ("Clients", clients_page),
                   ("Comptes", comptes_page),
                   ("Transactions", transactions_page)]:
    tk.Button(menu_frame, text=text, bg="#0094ff", fg="white",
              width=18, height=2, relief="flat",
              command=lambda f=func: show_frame(f)).pack(pady=8)

tk.Button(menu_frame, text="Quitter", bg="#ff4444", fg="white",
          width=18, height=2, relief="flat", command=root.quit).pack(side="bottom", pady=20)
```

Ce code crée le **menu latéral de l'application**. Il permet à l'utilisateur de **naviguer entre les différentes pages** (Accueil, Clients, Comptes, Transactions) et d'accéder au bouton Quitter pour fermer l'application. Le menu est stylisé avec des couleurs et des boutons clairs.

Conclusion

Le **Système de Gestion Bancaire** développé en Python constitue une solution complète, simple et performante pour la gestion des opérations bancaires de base. Grâce à **Tkinter**, l'application offre une interface claire et intuitive permettant à l'utilisateur de gérer facilement les clients, les comptes et les transactions. L'utilisation de **SQLite** garantit la fiabilité du stockage des données tout en assurant la cohérence grâce aux relations entre les tables.

Ce projet met en pratique des notions essentielles de la programmation orientée objet, de la gestion de base de données et de la conception d'interfaces graphiques.

En somme, ce travail démontre la capacité à concevoir et développer une **application de gestion complète** alliant robustesse, simplicité et efficacité.