Application de Gestion de Tâches avec Python (Tkinter & SQLite)

SOUHAIL KHARBOUCH

MOHAMED EL-BEKRI

INTRODUCTION

Dans le cadre de la réalisation d'une application de bureau, ce projet propose un système complet de **gestion des tâches** basé sur le langage **Python**. Il met en œuvre une interface graphique conviviale développée avec **Tkinter**, permettant aux utilisateurs de gérer facilement leurs actions quotidiennes. La base de données est assurée par **SQLite**, garantissant un stockage local, fiable et structuré des informations.

L'application permet d'**ajouter, supprimer et afficher des utilisateurs**, ainsi que de leur associer des tâches personnalisées. Chaque tâche peut contenir un **titre, une description, une priorité**, et un **statut** (En cours ou Terminé).

Grâce à des **fonctions de recherche et de mise à jour**, l'utilisateur peut retrouver, terminer ou supprimer une tâche rapidement. L'organisation visuelle en **onglets (Utilisateurs / Tâches)** offre une navigation simple et efficace.

Enfin, ce programme illustre l'intégration harmonieuse entre **interface graphique et base de données**, tout en respectant les principes de modularité et de validation des entrées. Il constitue une base solide pour le développement de systèmes de gestion plus avancés.

1- Importation des modules

```
import sqlite3
import tkinter as tk
from tkinter import messagebox, simpledialog, ttk
```

Cette partie initialise tous les modules nécessaires pour le fonctionnement de l'application.

1. `import sqlite3` : permet de gérer une base de données SQLite. SQLite est une base locale, légère, qui ne nécessite pas de serveur.

2. `connect('gestion_taches.db')` ouvre ou crée un fichier de base de données nommé `gestion_taches.db`. Si le fichier n'existe pas, il est créé automatiquement.

3. `cursor()` permet de créer un curseur, un objet qui exécute des requêtes SQL et récupère les résultats.

4. `commit()` sert à valider les modifications faites dans la base. Sans commit, les INSERT, UPDATE ou DELETE ne sont pas sauvegardés.

5. `close()` permet de fermer la connexion avec la base et libérer les ressources système.

6. `import tkinter as tk` : importe Tkinter, la bibliothèque standard de Python pour créer des interfaces graphiques.

7. `from tkinter import messagebox` : permet d'afficher des boîtes de dialogue pour informer l'utilisateur ou signaler des erreurs.

8. `from tkinter import simpledialog` : ouvre des pop-ups pour saisir du texte ou des nombres.

9. `from tkinter import ttk` : fournit des widgets avancés comme `Combobox` et `Notebook` pour un style moderne.

10. Cette combinaison de modules permet de lier **interface graphique**, **interactions utilisateur**, et **stockage persistant des données**.

2- Connexion et création des tables

```
curseur.execute('''
CREATE TABLE IF NOT EXISTS utilisateurs (
    id INTEGER PRIMARY KEY,
    nom TEXT NOT NULL UNIQUE
)
''')

curseur.execute('''
CREATE TABLE IF NOT EXISTS taches (
    id INTEGER PRIMARY KEY,
    titre TEXT NOT NULL,
    description TEXT,
    statut TEXT DEFAULT 'En cours' CHECK (statut IN ('En cours', 'Terminé')),
    priorite TEXT DEFAULT 'Moyenne' CHECK (priorite IN ('Haute', 'Moyenne', 'Basse')),
    utilisateur_id INTEGER,
    FOREIGN KEY (utilisateur_id) REFERENCES utilisateurs(id) ON DELETE CASCADE
)
''')
connexion.commit()
```

Ce code crée deux tables : une pour les utilisateurs et une pour les tâches. Chaque tâche peut être associée à un utilisateur. Les valeurs du statut et de la priorité des tâches sont contrôlées. Si un utilisateur est supprimé, ses tâches le sont aussi automatiquement. Enfin, les changements sont enregistrés dans la base de données.

1. `connexion = sqlite3.connect('gestion_taches.db')` ouvre la base SQLite et crée le fichier s'il n'existe pas.

2. `curseur = connexion.cursor()` crée un curseur pour exécuter des commandes SQL.

3. La table `utilisateurs` contient deux colonnes : `id` (clé primaire autoincrémentée) et `nom` (unique, obligatoire).

4. La table `taches` contient : `id` (clé primaire), `titre`, `description`, `statut` et `priorite` avec des valeurs par défaut.

5. `utilisateur_id` est une clé étrangère qui lie chaque tâche à un utilisateur. Cela permet de gérer plusieurs utilisateurs et leurs tâches respectives.

6. La clause `ON DELETE CASCADE` supprime automatiquement les tâches associées lorsqu'un utilisateur est supprimé, garantissant l'intégrité des données.

7. La clause `CHECK` empêche l'insertion de valeurs invalides pour `statut` ou `priorite`.

8. `connexion.commit()` valide la création des tables pour que les modifications soient persistantes.

9. Cette structure garantit un stockage fiable et cohérent des informations utilisateurs et tâches.

10. Grâce à cette organisation, les futures opérations CRUD (Create, Read, Update, Delete) sur la base seront sécurisées et structurées.

3- Validation des entrées

```python
def valider_entree(texte, champ):
    if not texte.strip():
        messagebox.showerror("Erreur", f"Le champ '{champ}' ne peut pas être vide.")
        return False
    return True
```

Cette fonction vérifie si un champ rempli par l'utilisateur n'est pas vide. Si le champ est vide, elle affiche un message d'erreur et renvoie `False`. Sinon, elle accepte l'entrée et renvoie `True`.

1. `def valider_entree(texte, champ):` crée une fonction prenant deux paramètres : le texte à vérifier et le nom du champ.

2. `texte.strip()` supprime les espaces inutiles au début et à la fin pour détecter les champs vraiment vides.

3. `if not texte.strip():` vérifie si le champ est vide après suppression des espaces.

4. Si vide, `messagebox.showerror("Erreur", f"Le champ '{champ}' ne peut pas être vide.")` affiche une alerte pour prévenir l'utilisateur.

5. La fonction retourne `False` si la validation échoue, empêchant l'insertion dans la base.

6. Si le texte est valide, la fonction retourne `True`, permettant de continuer le processus.

7. Cette vérification est cruciale pour éviter les erreurs SQL et garantir la qualité des données.

8. Elle est utilisée avant toute insertion d'utilisateur ou de tâche.

9. Permet de rendre l'application robuste en empêchant des champs vides qui pourraient causer des incohérences.

10. Elle centralise la validation, ce qui facilite la maintenance du code.

4-Gestion des utilisateurs

```python
def ajouter_utilisateur():
    nom = simpledialog.askstring("Ajouter Utilisateur", "Nom de l'utilisateur :")
    if nom and valider_entree(nom, "Nom"):
        try:
            curseur.execute("INSERT INTO utilisateurs (nom) VALUES (?)", (nom,))
            connexion.commit()
            messagebox.showinfo("Succès", "Utilisateur ajouté avec succès.")
            rafraichir_utilisateurs()
        except sqlite3.IntegrityError:
            messagebox.showerror("Erreur", "Nom d'utilisateur déjà existant.")

def supprimer_utilisateur():
    id_utilisateur = simpledialog.askinteger("Supprimer Utilisateur", "ID de l'utilisateur :")
    if id_utilisateur:
        curseur.execute("SELECT * FROM utilisateurs WHERE id = ?", (id_utilisateur,))
        utilisateur = curseur.fetchone()
        if utilisateur is None:
            messagebox.showerror("Erreur", f"Aucun utilisateur trouvé avec l'ID {id_utilisateur}.")
            return
        curseur.execute("DELETE FROM utilisateurs WHERE id = ?", (id_utilisateur,))
        connexion.commit()
        messagebox.showinfo("Succès", f"Utilisateur supprimé avec succès.")
        rafraichir_utilisateurs()
```

Cette partie comprend l'ajout, la suppression et le rafraîchissement de la liste des utilisateurs.

1. `ajouter_utilisateur()` Cette fonction permet d'ajouter un nouvel utilisateur. Elle demande le nom via une boîte de dialogue et vérifie qu'il n'est pas vide. Si le nom est valide, elle l'insère dans la base de données. En cas de succès, un message apparaît et la liste des utilisateurs est mise à jour. Si le nom existe déjà, un message d'erreur s'affiche.

2. La saisie est validée par `valider_entree` pour s'assurer qu'elle n'est pas vide.

3. `curseur.execute("INSERT INTO utilisateurs (nom) VALUES (?)", (nom,))` insère le nom dans la table, en utilisant `?` pour éviter les injections SQL.

4. `connexion.commit()` sauvegarde la nouvelle entrée dans la base de données.

5. En cas de doublon, `sqlite3.IntegrityError` est intercepté et un message d'erreur est affiché.

6. `supprimer_utilisateur()` Cette fonction supprime un utilisateur. Elle demande l'ID de l'utilisateur à supprimer, vérifie s'il existe dans la base de données, puis le supprime si trouvé. Les changements sont enregistrés, un message de succès s'affiche et

la liste des utilisateurs est actualisée. Si l'ID n'existe pas, un message d'erreur apparaît.

7. La suppression utilise la clause `WHERE id = ?` pour garantir que seule l'entrée sélectionnée est supprimée.

```
def rafraichir_utilisateurs():
    curseur.execute("SELECT id, nom FROM utilisateurs")
    utilisateurs = curseur.fetchall()
    liste_util.delete(0, tk.END)
    combo_utilisateur['values'] = [f"{u[0]} - {u[1]}" for u in utilisateurs]
    for u in utilisateurs:
        liste_util.insert(tk.END, f"ID: {u[0]} | Nom: {u[1]}")
```

8. `rafraichir_utilisateurs()` met à jour le `Listbox` et le `Combobox` pour refléter les changements.

9. Les utilisateurs sont affichés sous la forme `ID: X | Nom: Y` pour identifier facilement chaque entrée.

10. Cette organisation permet un contrôle complet des utilisateurs et lie chaque tâche à un utilisateur spécifique.

5- Gestion des tâches

```python
def ajouter_tache():
    if not combo_utilisateur.get():
        messagebox.showerror("Erreur", "Sélectionnez un utilisateur.")
        return
    id_utilisateur = int(combo_utilisateur.get().split(' - ')[0])

    fenetre_ajout = tk.Toplevel(fenetre)
    fenetre_ajout.title("Nouvelle Tâche")
    fenetre_ajout.geometry("400x300")
    fenetre_ajout.configure(bg="#F5F6FA")

    tk.Label(fenetre_ajout, text="Titre :", bg="#F5F6FA", font=("Arial", 11, "bold")).pack(pady=5)
    entree_titre = tk.Entry(fenetre_ajout, width=40)
    entree_titre.pack()

    tk.Label(fenetre_ajout, text="Description :", bg="#F5F6FA", font=("Arial", 11, "bold")).pack(pady=5)
    entree_desc = tk.Entry(fenetre_ajout, width=40)
    entree_desc.pack()

    tk.Label(fenetre_ajout, text="Priorité :", bg="#F5F6FA", font=("Arial", 11, "bold")).pack(pady=5)
    combo_priorite = ttk.Combobox(fenetre_ajout, values=["Haute", "Moyenne", "Basse"], state="readonly", width=38)
    combo_priorite.set("Moyenne")
    combo_priorite.pack()
```

Cette fonction permet d'ajouter une nouvelle tâche pour un utilisateur sélectionné. Elle commence par vérifier qu'un utilisateur est bien choisi, sinon elle affiche un message d'erreur. Ensuite, elle récupère l'identifiant de cet utilisateur. Une nouvelle fenêtre s'ouvre, intitulée « Nouvelle Tâche », avec une taille et un style prédéfinis. Dans cette fenêtre, l'utilisateur peut saisir le titre et la description de la tâche. Une liste déroulante permet également de choisir la priorité (Haute, Moyenne ou Basse), avec « Moyenne » sélectionnée par défaut.

1. `ajouter_tache()` vérifie d'abord qu'un utilisateur est sélectionné avant de créer une tâche.

2. Elle ouvre une nouvelle fenêtre (`Toplevel`) pour saisir le titre, la description et la priorité de la tâche.

3. Les valeurs sont validées avec `valider_entree` pour le titre.

4. `curseur.execute("INSERT INTO taches ...")` insère la tâche avec l'ID de l'utilisateur sélectionné.

5. `rechercher_tache()` utilise `LIKE` pour filtrer les tâches contenant le texte saisi dans le titre.

```
def marquer_termine():
    if not combo_utilisateur.get():
        messagebox.showerror("Erreur", "Sélectionnez un utilisateur.")
        return
    id_utilisateur = int(combo_utilisateur.get().split(' - ')[0])
    id_tache = simpledialog.askinteger("Marquer Terminé", "ID de la tâche :")
    if id_tache:
        curseur.execute("SELECT * FROM taches WHERE id=? AND utilisateur_id=?", (id_tache, id_utilisateur))
        tache = curseur.fetchone()
        if tache is None:
            messagebox.showerror("Erreur", "Aucune tâche trouvée avec cet ID.")
            return
        curseur.execute("UPDATE taches SET statut='Terminé' WHERE id=? AND utilisateur_id=?", (id_tache, id_utilisateur))
        connexion.commit()
        messagebox.showinfo("Succès", "Tâche marquée comme terminée.")
        rafraichir_taches()
```

6. `marquer_termine()` met à jour le statut de la tâche en “Terminé” pour l’utilisateur choisi.

```
def supprimer_tache():
    if not combo_utilisateur.get():
        messagebox.showerror("Erreur", "Sélectionnez un utilisateur.")
        return
    id_utilisateur = int(combo_utilisateur.get().split(' - ')[0])
    id_tache = simpledialog.askinteger("Supprimer Tâche", "ID de la tâche :")
    if id_tache:
        curseur.execute("SELECT * FROM taches WHERE id=? AND utilisateur_id=?", (id_tache, id_utilisateur))
        tache = curseur.fetchone()
        if tache is None:
            messagebox.showerror("Erreur", "Aucune tâche trouvée avec cet ID.")
            return
        curseur.execute("DELETE FROM taches WHERE id=? AND utilisateur_id=?", (id_tache, id_utilisateur))
        connexion.commit()
        messagebox.showinfo("Succès", "Tâche supprimée.")
        rafraichir_taches()
```

7. `supprimer_tache()` Cette fonction permet de supprimer une tâche. Elle vérifie qu’un utilisateur est sélectionné, demande l’ID de la tâche à supprimer, puis vérifie si elle existe bien pour cet utilisateur. Si oui, la tâche est supprimée de la base de données, les changements sont enregistrés, et la liste des tâches est actualisée. Sinon, un message d’erreur s’affiche.

```python
def rafraichir_taches():
    if not combo_utilisateur.get():
        return
    id_utilisateur = int(combo_utilisateur.get().split(' - ')[0])
    curseur.execute("SELECT id, titre, description, priorite, statut FROM taches WHERE utilisateur_id=? ORDER BY id", (id_utilisateur,))
    taches = curseur.fetchall()
    for item in tableau_taches.get_children():
        tableau_taches.delete(item)
    for t in taches:
        tableau_taches.insert("", "end", values=(t[0], t[1], t[2], t[3], t[4]), tags=('noir',))
        tableau_taches.tag_configure('noir', foreground='black')
```

8. `rafraichir_taches()` : Cette fonction met à jour la liste des tâches de l'utilisateur sélectionné. Elle récupère ses tâches dans la base de données, vide l'ancien contenu du tableau, puis affiche les nouvelles tâches avec leurs détails.

9. Les résultats sont affichés dans un `Listbox` mis à jour dynamiquement.

10. Chaque action sur les tâches est immédiatement persistée en base via `commit()` pour éviter toute perte de données.

6- Interface graphique (Tkinter)

```python
fenetre = tk.Tk()
fenetre.title("Gestion de Tâches")
fenetre.geometry("950x600")
fenetre.configure(bg="#F5F6FA")

onglets = ttk.Notebook(fenetre)
onglets.pack(fill='both', expand=True, padx=10, pady=10)

onglet_util = ttk.Frame(onglets)
onglets.add(onglet_util, text="Utilisateurs")

cadre_util = ttk.Frame(onglet_util, padding=15)
cadre_util.pack(fill='both', expand=True)
tk.Label(cadre_util, text="Gestion des Utilisateurs", font=("Arial", 16, "bold"), fg="#4A90E2").pack(pady=20)
cadre_boutons_util = tk.Frame(cadre_util, bg="#F5F6FA")
cadre_boutons_util.pack(pady=20)
tk.Button(cadre_boutons_util, text="Ajouter Utilisateur", command=ajouter_utilisateur,
          bg="#4A90E2", fg="white", width=20, height=2, font=("Arial", 12, "bold")).pack(pady=10)
tk.Button(cadre_boutons_util, text="Supprimer Utilisateur", command=supprimer_utilisateur,
          bg="#D9534F", fg="white", width=20, height=2, font=("Arial", 12, "bold")).pack(pady=10)
liste_util = tk.Listbox(cadre_util, height=15, width=70, font=("Arial", 10))
liste_util.pack(pady=20)

onglet_taches = ttk.Frame(onglets)
onglets.add(onglet_taches, text="Tâches")
```

```
cadre_tache = ttk.Frame(onglet_taches, padding=15)
cadre_tache.pack(fill='both', expand=True)

tk.Label(cadre_tache, text="Sélectionnez un utilisateur :", font=("Arial", 12, "bold"), fg="#4A90E2").pack()
combo_utilisateur = ttk.Combobox(cadre_tache, state="readonly", width=50, font=("Arial", 10))
combo_utilisateur.pack(pady=10)
combo_utilisateur.bind("<<ComboboxSelected>>", lambda e: rafraichir_taches())

frame_btn = tk.Frame(cadre_tache, bg="#F5F6FA")
frame_btn.pack(pady=15)
tk.Button(frame_btn, text="Ajouter Tâche", command=ajouter_tache,
          bg="#4A90E2", fg="white", width=15, font=("Arial", 10, "bold")).pack(side='left', padx=8)
tk.Button(frame_btn, text="Marquer Terminé", command=marquer_termine,
          bg="#F0AD4E", fg="white", width=15, font=("Arial", 10, "bold")).pack(side='left', padx=8)
tk.Button(frame_btn, text="Supprimer Tâche", command=supprimer_tache,
          bg="#D9534F", fg="white", width=15, font=("Arial", 10, "bold")).pack(side='left', padx=8)

colonnes = ("ID", "Tâche", "Description", "Priorité", "Statut")
tableau_taches = ttk.Treeview(cadre_tache, columns=colonnes, show="headings", height=18)
for col in colonnes:
    tableau_taches.heading(col, text=col)
    tableau_taches.column(col, width=170 if col != "Description" else 250, anchor="center")
tableau_taches.pack(fill="both", expand=True, pady=10)

tk.Button(fenetre, text="Quitter", command=fenetre.quit,
          bg="#6C757D", fg="white", width=15, font=("Arial", 10, "bold")).pack(pady=10)
```

Ce code crée une interface graphique pour gérer des utilisateurs et leurs tâches avec Tkinter. Il initialise une fenêtre principale contenant deux onglets : un pour la gestion des utilisateurs et un autre pour les tâches. Dans l'onglet "Utilisateurs", l'utilisateur peut ajouter, supprimer des utilisateurs et voir la liste existante. Dans l'onglet "Tâches", on peut sélectionner un utilisateur, ajouter une tâche, la marquer terminée ou la supprimer, et les tâches sont affichées dans un tableau. L'interface est organisée avec des cadres, des boutons, des labels et des listes déroulantes pour une utilisation facile. Enfin, le programme met à jour les données affichées, lance la boucle principale de la fenêtre et ferme la connexion à la base de données à la fin.

1. La fenêtre principale est créée avec `Tk()`, un titre, une taille et un arrière-plan personnalisés.

2. `ttk.Notebook()` crée des onglets pour séparer les utilisateurs et les tâches.

3. L'onglet **Utilisateurs** contient des boutons pour ajouter et supprimer un utilisateur, ainsi qu'une `Listbox` pour les afficher.

4. L'onglet **Tâches** contient un `Combobox` pour sélectionner l'utilisateur et des boutons pour gérer les tâches (Ajouter, Rechercher, Marquer Terminé, Supprimer).

5. Les `Listbox` affichent dynamiquement les utilisateurs et leurs tâches.

6. Les boutons utilisent des couleurs et polices différentes pour indiquer leur rôle (ex: rouge pour supprimer, bleu pour ajouter).

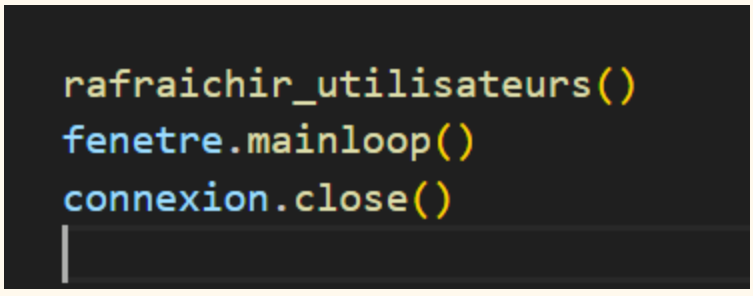
7. `bind("<<ComboboxSelected>>", ...)` permet de rafraîchir automatiquement les tâches lorsqu'on change d'utilisateur.

8. La disposition est gérée avec `pack()` et `frame` pour organiser les sections verticalement et horizontalement.

9. `Toplevel` permet d'ouvrir des fenêtres modales pour ajouter une tâche.

10. L'interface est conçue pour être claire, intuitive et facile à utiliser.

7- Boucle principale et fermeture



Cette partie finalise l'application en lançant l'interface et en fermant proprement la base de données.

1. `rafraichir_utilisateurs()` initialise la liste et le `Combobox` à l'ouverture de l'application.

2. `fenetre.mainloop()` lance la boucle Tkinter qui attend les interactions utilisateur.

3. Tant que cette boucle est active, l'application reste responsive.

4. Chaque action de l'utilisateur déclenche les fonctions correspondantes (CRUD).

5. Les mises à jour de l'interface sont automatiques grâce aux fonctions de rafraîchissement.

6. Les modifications apportées à la base sont immédiatement validées avec `commit()`.

7. `connexion.close()` ferme la connexion SQLite lorsque la fenêtre est fermée.

8. Cela garantit la libération des ressources système et la sauvegarde des données.

9. Toute erreur dans l'exécution SQL ou les interactions GUI peut être gérée par les validations et les exceptions.

10. L'ensemble crée une application complète, robuste, et prête à l'usage pour la gestion multi-utilisateur des tâches.