Documentation du Gestionnaire de Voitures

Table des matières

Aperçu général

Cette application est un gestionnaire de voitures développé en Python avec l'interface CustomTkinter, une version améliorée de Tkinter offrant un design moderne. L'application permet aux utilisateurs de gérer une collection de voitures et leurs modifications, avec persistance des données dans un fichier JSON.

Structure du code

Le programme est structuré selon un modèle orienté objet avec les principales classes suivantes :

1. `Modification` - Représente une modification apportée à une voiture
2. `Voiture` - Représente un véhicule avec ses caractéristiques et modifications
3. `GestionnaireVoitures` - Gère la collection de voitures et la persistance des données
4. `ApplicationVoiture` - Application principale et interface utilisateur
5. Frames spécialisés : `AccueilFrame`, `AjouterVoitureFrame`, et `AfficherVoituresFrame`

Modèles de données

Classe Modification

Représente une modification faite à une voiture.

Attributs :

- `description` : Description textuelle de la modification
- `cout` : Coût de la modification en euros

Méthodes :

- `to_dict()` : Convertit la modification en dictionnaire pour la sérialisation
- `from_dict()` : Crée une instance à partir d'un dictionnaire (désérialisation)

```
class Modification:
    def __init__(self, description: str, cout: float):
        self.description = description
        self.cout = cout

    def to_dict(self):
        return {"description": self.description, "cout": self.cout}

    @staticmethod
    def from_dict(data):
        return Modification(data["description"], data["cout"])
```

Classe Voiture

Représente un véhicule avec ses caractéristiques et modifications.

Attributs :

- `id_voiture` : Identifiant unique généré automatiquement
- `nom` : Nom de la voiture
- `marque` : Marque du véhicule
- `prix_achat` : Prix d'achat initial
- `modifications` : Liste des modifications apportées

Méthodes :

- `ajouter_modification()` : Ajoute une nouvelle modification à la voiture
- `calculer_prix_total()` : Calcule le prix total incluant les modifications
- `to_dict()` : Convertit la voiture en dictionnaire pour la sérialisation
- `from_dict()` : Crée une instance à partir d'un dictionnaire (désérialisation)

```
class Voiture:
    def __init__(self, nom: str, marque: str, prix_achat: float, modifications=None, id_voiture=None):
        self.id_voiture = id_voiture if id_voiture else str(uuid4())
        self.nom = nom
        self.marque = marque
        self.prix_achat = prix_achat
        self.modifications = modifications if modifications else []

    def ajouter_modification(self, description: str, cout: float):
        self.modifications.append(Modification(description, cout))

    def calculer_prix_total(self) -> float:
        return self.prix_achat + sum(mod.cout for mod in self.modifications)
```

Gestion des données

Classe GestionnaireVoitures

Gère la collection de voitures et la persistance des données.

Attributs :

- `voitures` : Liste des voitures enregistrées
- `fichier_donnees` : Nom du fichier JSON pour la persistance (par défaut : "voitures.json")

Méthodes :

- `ajouter_voiture()` : Ajoute une nouvelle voiture à la collection
- `sauvegarder_donnees()` : Enregistre la collection dans le fichier JSON
- `charger_donnees()` : Charge la collection depuis le fichier JSON
- `supprimer_voiture_par_id()` : Supprime une voiture par son identifiant

```
class GestionnaireVoitures:
    def __init__(self):
        self.voitures = []
        self.fichier_donnees = "voitures.json"
        self.charger_donnees()

    def sauvegarder_donnees(self):
        try:
            with open(self.fichier_donnees, "w") as f:
                json.dump([v.to_dict() for v in self.voitures], f)
        except Exception as e:
            messagebox.showerror("Erreur", f"Erreur de sauvegarde : {e}")

    def charger_donnees(self):
        try:
            with open(self.fichier_donnees, "r") as f:
                data = json.load(f)
                self.voitures = [Voiture.from_dict(v) for v in data]
        except FileNotFoundError:
            self.voitures = []
        except Exception as e:
            messagebox.showerror("Erreur", f"Erreur de chargement : {e}")
```

Interface utilisateur

Classe ApplicationVoiture

Application principale héritant de `ctk.CTk` qui définit la fenêtre principale.

Caractéristiques :

- Thème sombre moderne avec accents bleus
- Navigation entre différentes vues

- Conteneurs avec effet visuel moderne
- Boutons avec icônes Unicode

Méthodes principales :

- `center_window()` : Centre la fenêtre sur l'écran
- `afficher_frame()` : Change la vue active

```
class ApplicationVoiture(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.title("Gestionnaire de Voitures")
        self.geometry("800x700")
        self.minsize(800, 700)
        self.center_window()
        self.configure(fg_color=THEME_COLOR)

        # Ajout d'un titre principal avec animation
        self.title_label = ctk.CTkLabel(
            self,
            text=f"{ICON_CAR} Gestionnaire de Voitures",
            font=TITLE_FONT,
            text_color=ACCENT_COLOR
        )
        self.title_label.pack(pady=20)

        self.gestionnaire = GestionnaireVoitures()

        # Initialisation des frames
        self.frames = {}
        for F in (AccueilFrame, AjouterVoitureFrame, AfficherVoituresFrame):
            frame = F(self.container, self)
            self.frames[F.__name__] = frame
            frame.grid(row=0, column=0, sticky="nsew")

        # Navigation entre les frames
        self.afficher_frame("AccueilFrame")
```

Frames spécialisés

AccueilFrame

Page d'accueil avec titre de bienvenue et description de l'application.

```
class AccueilFrame(ctk.CTkFrame):
    def __init__(self, parent, controller):
        super().__init__(parent, fg_color=THEME_COLOR)
        self.controller = controller

        # Titre avec animation
        title = ctk.CTkLabel(
            self,
            text="Bienvenue dans le Gestionnaire de Voitures",
            font=TITLE_FONT,
            text_color=ACCENT_COLOR
        )
        title.pack(pady=50)
```

AjouterVoitureFrame

Formulaire permettant d'ajouter une nouvelle voiture avec :

- Champ pour le nom de la voiture
- Champ pour la marque
- Champ pour le prix d'achat

```
class AjouterVoitureFrame(ctk.CTkFrame):
    def __init__(self, parent, controller):
        super().__init__(parent, fg_color=THEME_COLOR)
        self.controller = controller

        # Champs de formulaire avec style moderne
        fields = [
            ("Nom", "nom_entry", "Entrez le nom de la voiture"),
            ("Marque", "marque_entry", "Entrez la marque de la voiture"),
            ("Prix d'achat", "prix_entry", "Entrez le prix d'achat")
        ]

        for i, (label, attr, placeholder) in enumerate(fields, 1):
            ctk.CTkLabel(
                form_frame,
                text=label,
                font=FONT,
                text_color=TEXT_COLOR
            ).grid(row=i, column=0, sticky="e", pady=10)
```

AfficherVoituresFrame

Vue principale pour gérer les voitures avec fonctionnalités pour afficher et modifier les voitures.

```
def selection_voiture(self, event):
    index = self.voiture_list.index(f"@{event.x},{event.y}").split(".")[0]

    if not index:
        return

    index = int(index) - 1

    if 0 <= index < len(self.controller.gestionnaire.voitures):
        self.selected_voiture = self.controller.gestionnaire.voitures[index]
        self.details_text.configure(state="normal")
        self.details_text.delete("1.0", "end")

        details = (
            f"{ICON_CAR} Nom: {self.selected_voiture.nom}\n"
            f"{ICON_CAR} Marque: {self.selected_voiture.marque}\n"
            f"{ICON_MONEY} Prix d'achat: {self.selected_voiture.prix_achat} €\n"
            f"{ICON_MONEY} Total avec modifications: {self.selected_voiture.calculer_prix_total()} €\n\n"
            f"{ICON_TOOLS} Modifications:\n"
        )
```

Thème et style visuel

L'application utilise un thème sombre moderne avec :

- Fond sombre (`#1a1f2e`)
- Boutons adoucis (`#2d3748`)
- Texte clair (`#e2e8f0`)
- Accent bleu (`#4299e1`)
- Effets de survol (`#3182ce`)
- Polices modernes (Helvetica)
- Icônes Unicode pour une meilleure lisibilité

```
# Couleurs et thème modernisés
THEME_COLOR = "#1a1f2e"  # Fond sombre plus moderne
BUTTON_COLOR = "#2d3748"  # Boutons plus doux
TEXT_COLOR = "#e2e8f0"  # Texte clair
ACCENT_COLOR = "#4299e1"  # Accent bleu moderne
HOVER_COLOR = "#3182ce"  # Couleur de survol
FONT = ("Helvetica", 12)
TITLE_FONT = ("Helvetica", 24, "bold")

# Icônes Unicode modernisées
ICON_HOME = "□"
ICON_ADD = "□"
ICON_LIST = "□"
ICON_REFRESH = "□"
ICON_EDIT = "✏"
ICON_DELETE = "□"
ICON_SAVE = "□"
ICON_CAR = "□"
```

Exécution de l'application

Pour lancer l'application, assurez-vous d'avoir installé la bibliothèque CustomTkinter :

```
pip install customtkinter
```

Puis exécutez le script principal :

```
if __name__ == "__main__":
    app = ApplicationVoiture()
    app.mainloop()
```

Persistance des données

Les données sont sauvegardées dans un fichier JSON nommé "voitures.json". Ce fichier est :

- Créé automatiquement s'il n'existe pas
- Mis à jour à chaque modification (ajout/suppression de voiture ou modification)
- Chargé au démarrage de l'application

```
def ajouter_voiture(self):
    nom = self.nom_entry.get()
    marque = self.marque_entry.get()
    prix = self.prix_entry.get()

    if not all([nom, marque, prix]):
        messagebox.showerror("Erreur", "Tous les champs sont obligatoires")
        return

    try:
        prix = float(prix)
        new_voiture = Voiture(nom, marque, prix)
        self.controller.gestionnaire.ajouter_voiture(new_voiture)
        messagebox.showinfo("Succès", "Voiture ajoutée avec succès!")

    except ValueError:
        messagebox.showerror("Erreur", "Prix invalide - doit être un nombre")
```

Fenêtres de dialogue et interactions utilisateur

```
def open_modification_popup(self):
    if not self.selected_voiture:
        messagebox.showwarning("Attention", "Sélectionnez d'abord une voiture")
        return

    popup = ctk.CTkToplevel(self)
    popup.title("Ajouter une Modification")
    popup.geometry("400x300")
    popup.configure(fg_color=THEME_COLOR)
```

```
def supprimer_voiture(self):
    if not self.selected_voiture:
        messagebox.showwarning("Attention", "Sélectionnez une voiture à supprimer")
        return

    if messagebox.askyesno("Confirmation", f"Supprimer {self.selected_voiture.nom} ?"):
        self.controller.gestionnaire.supprimer_voiture_par_id(self.selected_voiture.id_voiture)
        self.actualiser_liste()
        messagebox.showinfo("Succès", "Voiture supprimée avec succès!")
```

Cette application combine une interface utilisateur moderne avec une gestion de données efficace en utilisant un paradigme de programmation orientée objet.