TP 2-1 : Mini-projet guidé avec génération et refactoring

Objectif

- Structurer: Guider Copilot pour créer une structure de projet propre.
- Rédiger des prompts: Formuler des prompts efficaces pour générer du code cohérent.
- Vérifier et corriger: Relire, adapter et refactorer le code généré.
- Documenter: Comprendre et documenter le projet assisté par IA.

Etapes

1. **Initialiser le projet:**
    - Dossier: tp2_task_manager.
    - Fichiers: app.py, storage.py, cli.py, README.md.
    - **Action:** Configurer un environnement Python, initialiser Git
2. **Définir la logique métier:**
    - Prompt dans app.py:

```
# Créer une classe Task(id: int, title: str, done: bool=False)
# Créer une classe TaskService pour gérer CRUD: add_task, list_tasks, toggle_task, delete_task
# Ajouter des docstrings et exemples d'usage
```

    - **Action:** accepter, lire, ajuster noms/méthodes si nécessaire (cohérence, lisibilité).
3. **Persistance des données:**
    - Prompt dans storage.py:

```
# Implémenter un stockage JSON pour les tâches dans "tasks.json"
# Fonctions: load_tasks() -> list[dict], save_tasks(tasks: list[dict]) -> None
# Gérer le cas où le fichier n'existe pas encore
```

    - **Action:** Vérifier robustesse (création du fichier, encodage, erreurs d'IO).

4. **Interface CLI:**
    - Prompt dans cli.py:

```
# Créer une interface CLI avec argparse:
# commandes: add "title", list, toggle "id", delete "id"
# Afficher un format lisible: [id] title - DONE/NOT DONE
```

    - Action: **tester** py cli.py add "Apprendre Copilot", **puis** py cli.py list.

    Exécution Windows :

```
py cli.py add "Apprendre Copilot"
py cli.py list
```

5. **README et documentation:**
    - Markdown dans README.md:

```
# Générer un guide d'utilisation pour le gestionnaire de tâches CLI:
# installation, commandes, exemples, et bonnes pratiques de prompts Copilot
```

    - Action: Ajouter sections usage, exemples reproductibles, conseils de prompts (intention + contraintes + exemple).

6. **Refactoring assisté (app.py):**
    - Lisibilité:

```
# Refactorer TaskService pour améliorer la séparation des responsabilités et nommer clairement les méthodes
# Ajouter des annotations de types et gérer les erreurs (id introuvable)
```

    - Action: Éliminer duplications, clarifier API, ajouter exceptions et types.

7. **Tests unitaires (test_app.py):**
    - Créer test_app.py:

```
# Générer des tests unitaires avec pytest pour add_task, toggle_task, delete_task
# Mock du stockage pour isoler la logique
```

    - **Installation obligatoire (bash)** :

```
py -m pip install pytest
```

- Action: **exécuter pytest -q**, corriger selon feedback.

8. **Validation finale:**
    - Checklist:
        - Fonctionnel: add/list/toggle/delete OK.
        - Persistance: tasks.json mis à jour.
        - Docs: README clair, exemples reproductibles.
        - Code: types, docstrings, structure propre.

Variante calculatrice (option)

- **Prompt:** générer une calculatrice CLI avec opérations (+, -, ×, ÷), gestion des erreurs (division par zéro), tests unitaires, et un mode interactif.
- **Persistance (bonus):** historique des opérations en JSON.