

TP : Gestion d'une Bibliothèque Numérique

Contraintes

1. Appliquer le principe **SRP** (Single Responsibility Principle).
 2. Travail en binôme au maximum.
 3. Nettoyer les données en choisissant les **méthodes les plus adaptées et efficaces**.
 4. Développer en **POO**, avec un **couplage faible** (pas d'instances internes non nécessaires).
 5. Respecter **PEP8** ainsi que les bonnes pratiques de **refactoring**.
-

Contexte

Vous travaillez pour une médiathèque municipale qui souhaite mettre en place un petit prototype pour gérer son fonds de livres. L'objectif n'est pas de créer une interface, mais un **module métier propre**, structuré et réutilisable.

Les bibliothécaires doivent pouvoir :

- Classer les livres par **catégorie** (ex : Roman, Théâtre, Philosophie...)
- Vérifier quels livres sont **disponibles** ou déjà empruntés
- **Emprunter** et **rendre** des livres
- Afficher des statistiques simples

Les données proviendront d'un **fichier texte externe**.

Fichier de données (à créer par vous)

Nom du fichier : `books.csv` Format (séparateur ;) :

Catégorie ; Titre ; Auteur ; Année ; Disponible

Exemple de ligne (l'étudiant doit en créer **au moins 25**) :

Roman;Le Rouge et le Noir;Stendhal;1830;yes

Règles :

- Nettoyer les espaces autour des données
 - Convertir **Année** → **int**
 - Convertir **yes/no** → **booléen**
 - Ignorer (ne pas charger) les lignes invalides
-

Contraintes Techniques

1. Pas de couplage fort

- Les classes **ne doivent pas instancier d'autres classes** en interne.
- Elles reçoivent les objets en paramètres (injection de dépendances).

2. Exceptions dans les classes

- Les classes lèvent des exceptions (**raise**) si une opération est impossible.

3. Gestion des erreurs à l'extérieur

- Les **try/except** sont utilisés uniquement dans la partie principale, jamais dans les classes.

4. Organisation du code

- Les classes doivent être dans des fichiers séparés si le projet est structuré.
-

Fonctionnalités à implémenter

1) Classe **Book**

Représente un livre.

- Attributs : **title, author, year, available**
- Valider les données dans le constructeur (ex : année numérique)

2) Classe **Category**

Contient un ensemble de livres.

- Reçoit une **liste de livres**
- Méthodes :
 - **available_books()** → renvoie les livres non empruntés
 - **borrow(title)** → met **available = False** ou lève une exception
 - **return_book(book)** → remet **available = True**

3) Classe **Library**

Coordonne toutes les catégories.

- Reçoit une **liste de catégories**
- Méthodes :
 - **borrow_book(category_name, title)**
 - **return_book(category_name, book)**

- `total_available()` → somme des livres disponibles

4) Partie principale (`main`)

- Charger les données depuis `books.csv`
 - Nettoyer les lignes
 - Construire :
 - `Book` → regroupés dans `Category` → regroupées dans `Library`
 - Scénario à tester :
 - Afficher le nombre total de livres disponibles
 - Emprunter un livre
 - Tenter d'emprunter un livre déjà emprunté (gérer l'erreur via `try/except`)
 - Rendre un livre
 - Réafficher le nombre total disponible
-

Grille de Notation (Sur 20 points)

Critère	Points
Lecture et nettoyage du fichier <code>books.csv</code>	3
Conception correcte de la classe <code>Book</code>	2
Conception correcte de la classe <code>Category</code>	3
Conception correcte de la classe <code>Library</code>	3
Absence de couplage fort (injection respectée)	4
Exceptions levées dans les classes (pas de <code>print</code>)	2
Gestion des erreurs à l'extérieur (<code>try/except</code>)	1
Scénario demandé fonctionnel + affichages cohérents	2

Total : 20 points

Résumé à retenir

Les classes gèrent la logique **et lèvent des exceptions**. Le programme principal **gère les erreurs**. Les classes **ne créent pas** leurs dépendances → elles les **reçoivent** (couplage faible).

Bon courage les parisiens et les lyonnais