We are working on the Smart Library Manager Capstone Project.

**Abstract:**

A Smart Library System is all about making library work easier and smoother. Instead of spending hours writing down details in registers or searching through piles of records, everything is stored digitally. With this system, librarians can quickly add new books, update details, or check if a book is available. Students or members can easily borrow and return books without confusion, while the system keeps track of everything in the background. This not only saves time but also avoids mistakes and keeps the library well-organized. In short, it’s a simple and smart way to run a library in today’s digital world.

The requirements of our project are:

* Classes
* Library
* Methods
* Approach
* Design
* Flow

Classes which we have used in our project are:

a) Reserve a class- this represents a single library book.

Attributes used are:

* book\_id: A special number assigned to every book.
* title: The book's title.
* author: The book's author.
* ISBN: International Standard Book Number.
* Boolean: available (False = issued, True = available).

Techniques which we have implemented is->

Book data is converted into a line for file storage using the to\_line() function.

We have used ->

**from\_line(line)** which returns a Book object from a stored line.

**\_\_str\_\_()** which is used to provide book details that are readable by humans.

**Class Book- CODE**

class Book:

def \_\_init\_\_(self, book\_id: str, title: str, author: str, isbn: str, available: bool = True):

self.book\_id = book\_id

self.title = title

self.author = author

self.isbn = isbn

self.available = available

def to\_line(self) -> str:

return "|".join([self.book\_id, self.title, self.author, self.isbn, "1" if self.available else "0"]) + "\n"

@staticmethod

def from\_line(line: str) -> "Book":

parts = line.strip().split("|")

return Book(parts[0], parts[1], parts[2], parts[3], parts[4] == "1")

def \_\_str\_\_(self) -> str:

status = "Available" if self.available else "Borrowed"

return f"[{self.book\_id}] {self.title} by {self.author} (ISBN: {self.isbn}) — {status}"

***Logic used in the code:***

The Book class is designed to handle all details about a book.

* When we create a book, it stores its ID, title, author, ISBN, and whether it’s available or borrowed.
* The to\_line method turns the book’s details into a single line of text so it can be saved in a file.
* The from\_line method does the opposite — it reads a line from the file and recreates the book object.
* The \_\_str\_\_ method makes the book look nice when printed, showing details like title, author, ISBN, and status.

b) Class of Members-represents a patron of the library.

Attributes used are:

* member\_id: Each member's unique identifier.
* name: The member's name.
* borrowed: A list of book IDs that have been borrowed.

Techniques which we have implemented is->

* member data we have added it into a line for file storage using the to\_line() function.
* from\_line(line) is used which returns a Member object from a stored line.
* We have used\_\_str\_\_() which provides member details that are readable by humans.

**Class Member-CODE**

class Member:

def \_\_init\_\_(self, member\_id: str, name: str, borrowed: Optional[List[str]] = None):

self.member\_id = member\_id

self.name = name

self.borrowed = borrowed or []

def to\_line(self) -> str:

borrowed\_str = ",".join(self.borrowed)

return "|".join([self.member\_id, self.name, borrowed\_str]) + "\n"

@staticmethod

def from\_line(line: str) -> "Member":

parts = line.strip().split("|")

borrowed = parts[2].split(",") if parts[2] else []

return Member(parts[0], parts[1], borrowed)

def \_\_str\_\_(self) -> str:

return f"[{self.member\_id}] {self.name} - Borrowed: {', '.join(self.borrowed) if self.borrowed else 'None'}"

***Logic used in the code:***

* When a new library member is added, they’re given a membership ID, name, and a list of borrowed books (which starts empty if they haven’t taken any).
* Whenever the system needs to save member details into a file, it turns their information into a single line of text – almost like writing their record on a library card.
* Later, if the program is restarted, the system can read that line back from the file and rebuild the member’s profile exactly as it was before.
* When we want to **s**ee a member’s details, the system shows their ID, name, and the books they currently have borrowed. If they haven’t borrowed anything, it simply shows “None.”

c)Library Class-Acts as the controller and manager of the system.

Attributes we have used are:

* books: A dictionary that stores all Book objects.
* members: A dictionary that stores all Member objects.

Methods we have used are:

**load():** Reads data from files like books.txt and members.txt.

**save():** Saves the current data back to files.

**add\_book():** Adds a new book to the system.

**remove\_book():** Removes a book if it is available.

**register\_member():** Adds a new member.

**borrow\_book()**: Allows a member to borrow a book if it is available.

**return\_book():** Handles the return of a borrowed book.

**list\_books():** Displays all books.

**list\_members():** Displays all members.

**Class Library-CODE**

class Library:

def \_\_init\_\_(self):

self.books: Dict[str, Book] = {}

self.members: Dict[str, Member] = {}

def load(self):

if os.path.exists("books.txt"):

with open("books.txt", "r") as f:

for line in f:

book = Book.from\_line(line)

self.books[book.book\_id] = book

if os.path.exists("members.txt"):

with open("members.txt", "r") as f:

for line in f:

member = Member.from\_line(line)

self.members[member.member\_id] = member

def save(self):

with open("books.txt", "w") as f:

for book in self.books.values():

f.write(book.to\_line())

with open("members.txt", "w") as f:

for member in self.members.values():

f.write(member.to\_line())

***Logic used in the code:***

* When the program starts, it creates empty storage for books and members.
* The load method checks if books.txt and members.txt already exist.
* If the files exist, the system reads them and restores all books and members into memory.
* This makes sure the library remembers everything from the previous session.
* When changes are made (like adding or borrowing), the save method updates the files with the new details.
* This ensures that all data is kept safe and nothing is lost, even after the program is closed.

## **Libraries Used in the Code:**

1. **os**
   1. Helps the program check if files already exist before trying to load them.
   2. This makes sure that data (like books and members) is not lost and can be reloaded when the program starts again.
2. **typing**
   1. Used to add type hints (like Dict, List, Optional) in the code.
   2. This doesn’t change how the program runs but makes the code easier to read, understand, and maintain

## **Methods Used in the Code:**

### **File Handling**

load(), save(), to\_line(), from\_line() : These methods take care of reading data from files and saving it back, so book and member information is always up to date.

### **Admin Operations**

add\_book(), remove\_book(), register\_member() : These methods are for the librarian/admin to manage the library by adding or removing books and registering new members.

### **User Operations**

borrow\_book(), return\_book() :These methods let library members borrow books (if available) and return them when done.

### **Utility Methods**

list\_books(), list\_members(), \_\_str\_\_() :These help in showing the list of all books or members in the library and provide a clean way to display information.

**CODE - File Handling**

def load(self):

if os.path.exists("books.txt"):

with open("books.txt", "r") as f:

for line in f:

book = Book.from\_line(line)

self.books[book.book\_id] = book

if os.path.exists("members.txt"):

with open("members.txt", "r") as f:

for line in f:

member = Member.from\_line(line)

self.members[member.member\_id] = member

def save(self):

with open("books.txt", "w") as f:

for book in self.books.values():

f.write(book.to\_line())

with open("members.txt", "w") as f:

for member in self.members.values():

f.write(member.to\_line())

***Logic of the Code-***

load() → Reads saved data from books.txt and members.txt into objects.

save() → Writes current objects into files for persistence.

**CODE - Admin Methods**

def add\_book(self, book\_id: str, title: str, author: str, isbn: str):

self.books[book\_id] = Book(book\_id, title, author, isbn)

def remove\_book(self, book\_id: str):

if book\_id in self.books:

del self.books[book\_id]

def register\_member(self, member\_id: str, name: str):

self.members[member\_id] = Member(member\_id, name)

***Logic of the Code-***

* add\_book() → Inserts a new book.
* remove\_book() → Delete a book if it exists.
* register\_member() → Add new member to system.

**CODE - User Methods**

def borrow\_book(self, member\_id: str, book\_id: str):

if member\_id in self.members and book\_id in self.books:

member = self.members[member\_id]

book = self.books[book\_id]

if book.available:

book.available = False

member.borrowed.append(book\_id)

def return\_book(self, member\_id: str, book\_id: str):

if member\_id in self.members and book\_id in self.books:

member = self.members[member\_id]

book = self.books[book\_id]

if book\_id in member.borrowed:

book.available = True

member.borrowed.remove(book\_id)

***Logic of the code***

borrow\_book() →

* Checks if the book exists and is available.
* Change book status → available = False.
* Add book ID to member’s borrowed list.

return\_book() →

* Check if a member actually borrowed the book.
* Change book status → available = True.
* Remove book ID from member’s borrowed list.

**CODE - Utility Methods**

def list\_books(self):

for book in self.books.values():

print(book)

def list\_members(self):

for member in self.members.values():

print(member)

***Logic of the code-***

list\_books() → Show all books with availability.

list\_members() → Show all members and what they borrowed.

**CODE-CLI Interface**

**Main Logic-**

def cli():

library = Library()

library.load()

while True:

print("\nSmart Library Manager")

print("1. Add Book")

print("2. Remove Book")

print("3. Register Member")

print("4. Borrow Book")

print("5. Return Book")

print("6. List Books")

print("7. List Members")

print("8. Exit")

choice = input("Enter choice: ")

if choice == "1":

book\_id = input("Book ID: ")

title = input("Title: ")

author = input("Author: ")

isbn = input("ISBN: ")

library.add\_book(book\_id, title, author, isbn)

elif choice == "2":

book\_id = input("Book ID: ")

library.remove\_book(book\_id)

elif choice == "3":

member\_id = input("Member ID: ")

name = input("Name: ")

library.register\_member(member\_id, name)

elif choice == "4":

member\_id = input("Member ID: ")

book\_id = input("Book ID: ")

library.borrow\_book(member\_id, book\_id)

elif choice == "5":

member\_id = input("Member ID: ")

book\_id = input("Book ID: ")

library.return\_book(member\_id, book\_id)

elif choice == "6":

library.list\_books()

elif choice == "7":

library.list\_members()

elif choice == "8":

library.save()

print("Library data saved. Exiting...")

break

else:

print("Invalid choice. Try again.")

***Logic of the code-***

Purpose: To design a menu-driven system for interaction.

Logic:

->To start by loading saved data.

->Display a menu with options: add, remove, borrow, return, list, exit.

->When the user makes a choice, call the appropriate Library method.

->On exit, save all changes back to the files.

->Main Logic Idea is to offer a user-friendly way to interact with Library class methods.

**Approach of our Project**

The main goal of using this approach is to digitize library management using Python and file storage. Instead of manually recording books and members, the system automates these tasks through classes and text files.

We have used:

**->Object-Oriented Design**

- Book class: Represents book details.

- Member class: Represents library members.

- Library class: Manages books, members, and transactions.

**->File-Based Persistence**

- Uses books.txt and members.txt to store and retrieve data.

- Ensures data is saved between program runs.

**->Admin and User Operations**

- Admin: Add or remove books, register members.

- User: Borrow or return books.

**->CLI Menu-Driven Program**

- Text-based menu allows users to interact with the system.

## **Design of the Project**

The smart library system is designed to be simple and easy to use. It has a few main parts that work together to keep the library running smoothly.

* **Books** are stored with details like title, author, and whether they are available or already borrowed.
* **Members** have their own records showing who they are and which books they’ve borrowed.
* The **Library** brings everything together. It manages books, members, and all the actions like borrowing, returning, and saving data.

All this information is saved in two text files—one for books and one for members—so that even if the program is closed, nothing is lost.

The system is used through a **simple menu** that appears on the screen. The user just picks what they want to do, like adding a book, borrowing one, or checking the list of members. The system then takes care of the rest.

### **How the flow of the code works : Step by Step**

1. **When the program starts**
   1. It first loads the data from two files: one for books and one for members.
   2. This way, whatever was saved earlier comes back into the system, so nothing is lost.
2. **While using the program**
   1. A menu shows up, and the user simply picks what they want to do. For example:  
      1. Add a new book → the book is created and saved.
      2. Remove a book → the system checks and deletes it if possible.
      3. Register a member → a new member is added and saved.
      4. Borrow a book → the system marks it as borrowed and links it to the member.
      5. Return a book → the book becomes available again and is removed from the member’s borrowed list.
      6. List books → shows all books and whether they’re available or not.
      7. List members → shows all members and the books they’ve borrowed.

From the above list the user opts one of their choice of interest and performs the necessary operation.

1. **When the program closes**
   1. All the changes made are saved back into the files.
   2. This means the library is always up to date the next time you open the program.

**File Handling**

* The system uses two files — books.txt and members.txt — to store information about the books added,removed,updated and borrowed.
* These files act like the library’s notebooks that keep records safe.
* When the program starts, it reads the files and loads all the saved books and members back into memory.
* Any changes (like adding/removing books or borrowing/returning) are first updated in memory.
* When saving, the updated details are written back into the files.
* This makes sure data is not lost, even after the program is closed or restarted.

**Exception Handling**

1. **Why is exception handling needed here?**
2. When the program starts, it tries to load the stored data of books and members.  
   But if the files (books.txt or members.txt) don’t exist yet (say it’s the first time running the program), the code could throw an error and crash.
3. To prevent this, the code uses safe handling with os.path.exists().
4. **How it works with files:**
5. Before opening a file, the program first checks: *“Does this file even exist?”*
6. If the file is there → it proceeds to open and read each line. Each line is then converted back into a Book or Member object, so the library remembers all past records.
7. If the file is not there → it doesn’t try to read it at all. Instead, it simply skips and starts fresh with an empty library.
8. **Why this is useful:**
9. This way, the program never crashes due to a missing file.
10. It makes the system robust (able to handle unexpected situations) and user-friendly (no confusing error messages for users).
11. Users don’t even notice anything wrong — if files exist, old records are loaded; if not, a new library is ready to use.

## 

## 

## **Conclusion**

The Smart Library Manager is a simple project that shows how Python can be used to make library work easier. It keeps track of books and members in an organized way, saves the data so nothing is lost, and gives users a clear menu to do their tasks without confusion.

In the future, this system can grow bigger and smarter. We could use a database instead of text files, add a proper app-like interface, and include extra features like book search, due dates, and fines—making it feel just like a real library system.