**Abstract**

The e-Library Management System is a Python-based application designed to digitally manage books, users, and borrowing activities in a library. The system allows administrators to add and remove books, register users, issue and return books, and calculate fines for overdue returns. It follows an object-oriented design with separate classes for books, users, and the main library system, ensuring modularity and reusability.

## **Why This Project?**

We chose this project because it is a real-world scenario that applies data management, persistence, and OOP principles. It handles users, books, and transactions practically and can be extended into a larger system (GUI, database).

## **1. Introduction**

The e-Library Management System is a Python-based application designed to manage library resources digitally. It allows administrators to add/remove books, register users, issue and return books, and calculate fines for overdue returns.  
This project is built using Object-Oriented Programming (OOP) principles and stores data persistently in JSON format.

## **2. Modules Used**

* json → Store and load library data persistently.
* os → Check if the storage file exists.
* logging → Record important actions (book added, fine issued, etc.).
* datetime, timedelta → Manage issue dates, due dates, and calculate fines.
* typing → For type hints (Dict, List).

## **3. Constants**

* STORE → JSON file to store data (library.json).
* LOGFILE → Log file for activities (library.log).
* LOAN\_DAYS → Default book loan period (14 days).
* FINE\_PER\_DAY → Penalty for overdue returns (2 units per day).

**4. Classes and Attributes**

Class: A class in Python is like a blueprint for creating objects. It describes what properties (attributes) and what behaviors (methods) those objects will have.

Attributes: Attributes are variables that belong to a specific object. They store data related to that object.

**Class 1: Book**

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

The Book class is a blueprint for creating book objects in the library system.  
 It describes what information every book must have.

### 🔸 Attributes

1. self.book\_id  
   Unique ID for the book (like "B1" or "B101").  
   Helps identify a book uniquely in the system (like a roll number for a student).
2. self.title  
   The name of the book (e.g., "Harry Potter and the Sorcerer’s Stone").  
   Helps users know which book they’re borrowing.
3. self.author  
   The author’s name (e.g., "J.K. Rowling").  
   Useful for searching/filtering books by author.
4. self.isbn  
   ISBN = International Standard Book Number (e.g., "978-3-16-148410-0").  
   Acts as a universal unique code for a book edition.
5. self.available  
   Boolean (True or False).  
   True = book is available to borrow.  
   False = book is already issued to someone.

**Class 2: User**

class User:

def \_\_init\_\_(self, user\_id: str, name: str):

self.user\_id = user\_id

self.name = name

### Explanation

The User class represents a member of the library.  
 It stores their identity so the system knows who borrowed which book.

### 🔸 Attributes

1. self.user\_id  
   Unique ID for the user (like "U1", "U202").  
   Prevents confusion if two users have the same name.
2. self.name  
   The actual name of the user (e.g., "Sahil").  
   Helps in identifying and displaying user details.

**Class 3: LibrarySystem**

class LibrarySystem:

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

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

self.users: Dict[str, User] = {}

self.transactions: List[dict] = []

self.\_load()

### Explanation

The LibrarySystem class is the main controller of the program.  
 It manages all books, all users, and all borrow/return transactions.

### 🔸 Attributes

1. self.books  
   A dictionary (Dict[str, Book]).  
   Stores all book objects.  
   Key = book ID (e.g., "B1"), Value = Book object.  
   Example:  
   {

"B1": Book("B1", "Harry Potter", "J.K. Rowling", "12345"),

"B2": Book("B2", "Sherlock Holmes", "Arthur Conan Doyle", "54321")

}

2. self.users

* A dictionary (Dict[str, User]).
* Stores all user objects.
* Key = user ID (e.g., "U1"), Value = User object.  
  Example:

{

"U1": User("U1", "Sahil"),

"U2": User("U2", "Priya")

}

3. self.transactions

* A list of dictionaries.
* Each dictionary represents a record of issuing or returning a book.
* Example transaction:

{

"book\_id": "B1",

"user\_id": "U1",

"issue\_date": "2025-08-19T10:30:00",

"due\_date": "2025-09-02T10:30:00",

"return\_date": None

}

* If return\_date is None → book is still with user.
* If return\_date has a date → book was returned.

4. self.\_load()

* Not an attribute, but called in the constructor.
* Loads existing data from library.json so that previously saved books, users, and transactions are restored when the program starts.

Book class → describes books (ID, title, author, ISBN, available).

User class → describes members (ID, name).

LibrarySystem class → manages everything (all books, users, and transactions).

**5. JSON File Handling**

with open(STORE, "r", encoding="utf-8") as f:

data = json.load(f)

self.books = {b["book\_id"]: Book.from\_dict(b) for b in data.get("books", [])}

self.users = {u["user\_id"]: User.from\_dict(u) for u in data.get("users", [])}

self.transactions = data.get("transactions", [])

* Opens the file → Reads JSON → Rebuilds Book and User objects → Restores transactions.
* with open(STORE, "r", encoding="utf-8") as f: → Opens library.json file in read mode.
* data = json.load(f) → Reads the JSON file and converts it into a Python dictionary.
* self.books = {...} → Rebuilds all saved books as Book objects from JSON.
* self.users = {...} → Rebuilds all saved users as User objects from JSON.
* self.transactions = ... → Loads the list of transactions

Saving Data:

with open(STORE, "w", encoding="utf-8") as f:

json.dump(data, f, indent=2)

* Converts Python objects into JSON and saves them with indentation.
* with open(STORE, "w", encoding="utf-8") as f: → Opens library.json in write mode (creates/overwrites the file).
* json.dump(data, f, indent=2) → Writes the Python data (books, users, transactions) into the file in JSON format with neat indentation.

**6. Exception Handling**

An exception is a signal that something unexpected happened while running a program.

* raise ValueError("Book ID exists") → Raised if book already exists.
* raise LookupError("Book not found") → Raised when searching for missing book/user.
* raise RuntimeError("Book already issued") → Raised when trying to issue an already borrowed book.

**7. Fine Calculation**

The fine logic is inside the give\_back\_book method:

def give\_back\_book(self, user\_id: str, book\_id: str) -> int:

book = self.books.get(book\_id)

if not b

1.Due date is read  
When a book is issued, the system records an issue date and a due date (issue date + 14 days).At return time, it fetches this due date.

2.Compare today’s date with due date  
now.date() - due.date() gives the number of days late.  
If the book is returned early or on time, the difference is 0 or negative.

3.Clamp to zero  
max(0, days) ensures the late days can’t go below 0 (no negative fines).

4.Calculate fine  
Fine = late days × FINE\_PER\_DAY.  
Since FINE\_PER\_DAY = 2, each day late adds 2 units

5. tx = None → Start by assuming no matching transaction is found.

Loop through self.transactions → This list stores all borrow/return records.

Condition inside loop:

t["book\_id"] == book\_id → is this the same book?

t["user\_id"] == user\_id → was it issued to this user?

t["return\_date"] is None → has it not yet been returned?

If all 3 match → we found the active transaction (a book currently borrowed). Save it in tx and stop searching.

After loop:

If tx is still None, it means no such active loan exists → the system raises LookupError("Active transaction not found").

**8. List comprehension**

"books": [b.as\_dict() for b in self.books.values()],

"users": [u.as\_dict() for u in self.users.values()],

Loops through all Book objects in self.books.  
Calls as\_dict() on each book (turning it into a dictionary).

Collects them all into a list → ready to save in JSON.

## **9. run\_cli – Command Line Interface**

run\_cli = Run Command Line Interface.  
A CLI is a text-based menu that lets users interact with the program by typing numbers or text.  
This function provides the user interface for your e-Library System.

**10. Saving the main function into other file**

No, it won’t directly work.

Because those functions depend on other parts of the project (like self.books, self.users, constants, JSON save).

If you put them in another file, you must import the class/module where they belong.

Example:

from e\_library\_system import LibrarySystem

lib = LibrarySystem()

lib.give\_back\_book("U1", "B1")

So yes, you can separate, but you need to treat the original file as a module.

try to put this the document

**11. Using json format , logging and datatime**

JSON → Stores all data (books, users, transactions) in a file so it is saved even after the program closes. It is lightweight, human-readable, and easy to load back into Python.

Logging → Records every important action (like adding books, issuing/returning, fines) into library.log. This helps in tracking activities, debugging errors, and keeping an audit history.

Datetime → Handles all time-based logic such as setting due dates, checking return dates, and calculating fines for late returns. It ensures the system automatically manages dates without manual calculation.

**12.Purpose of library.json and library.log**

library.json → Stores all data (books, users, transactions) so the system remembers everything even after closing the program.

library.log → Stores all actions/events (like book added, user registered, fine charged) for record-keeping and debugging.

**13. Configuration of log file**

logging.basicConfig(filename=LOGFILE, level=[logging.INFO](http://logging.info), format="%(asctime)s %(levelname)s: %(message)s")

filename=LOGFILE → All logs are stored in library.log.

level=logging.INFO → Records general actions (INFO) and above (WARNING, ERROR).

format="..." → Defines how logs look: timestamp, level, and message.

**14.Purpose of using staticmethod**

@staticmethod is used when a method works with the class but doesn’t depend on an existing object’s attributes.

@staticmethod

def from\_dict(d):

return Book(d["book\_id"], d["title"], d["author"], d["isbn"])

we don’t need an existing Book to call it — it’s just a helper to build one from data.

**15. Static Method with Dict and Arrow**

def from\_dict(d: dict) -> "Book":

return Book(d["book\_id"], d["title"], d["author"], d["isbn"], d.get("available", True))

* d: dict = type hint, means parameter d must be a dictionary.
* -> "Book" = return type hint → tells that this method returns a Book object.
* Arrow mark is just a Python type hint feature, not logic.

**16\_\_init\_\_ Method in LibrarySystem**

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

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

self.users: Dict[str, User] = {}

self.transactions: List[dict] = []

self.\_load()

\_\_init\_\_ → constructor, runs when class is created.

Creates empty dictionaries/lists for books, users, transactions.

self.\_load() → loads saved data from library.json.

Dict[str, Book] = type hint → dictionary with book\_id as key and Book as value.

**17. What is utf -8**

* UTF-8 is a character encoding format.
* Ensures that text (English, symbols, even other languages) is stored and read correctly from files.

**18. File Handling with Dictionary Comprehension**

self.books = {b["book\_id"]: Book.from\_dict(b) for b in data.get("books", [])}

self.users = {u["user\_id"]: User.from\_dict(u) for u in data.get("users", [])}

Reads data from JSON.

For each dictionary in list books, creates a Book object.

Saves them into a dictionary with book\_id as key.

This is dictionary comprehension (short form of loop).

**19. Converting Objects Back for Saving**

"books": [b.as\_dict() for b in self.books.values()],

"users": [u.as\_dict() for u in self.users.values()],

"transactions": self.transactions,

Converts all objects back into dictionaries before saving to JSON.

Uses list comprehension to make list of dictionaries.

**20. Indent-2**

Purpose → Makes the JSON file neatly formatted instead of one long line.

indent=2 → Adds 2 spaces for indentation in JSON output.

Benefit → Easier for humans to read, edit, and debug the file.

**21. Purpose of json.loads and json.dump**

json.loads() → *Load String* → Converts a JSON string into a Python object (dict, list, etc.).  
 Example:  
 data = json.loads('{"name":"Sahil"}')

json.dump() → *Dump to File* → Writes a Python object into a JSON file.  
 Example:  
 json.dump(data, f, indent=2)

**22. Transaction Dictionary with Append**

"book\_id": book\_id,

"user\_id": user\_id,

"issue\_date": issue\_date.isoformat(),

"due\_date": due\_date.isoformat(),

"return\_date": None,

Creates a transaction record as a dictionary.

Appends it into the transactions list.

Purpose → store who borrowed what book, and when.

**23. Function Definition with Return Type**

def give\_back\_book(self, user\_id: str, book\_id: str) -> int:

self = refers to current object (LibrarySystem).

-> int = function will return an integer (the fine).

Helps type-checking & clarity.

**24. Active Transaction Check**

if t["book\_id"] == book\_id and t["user\_id"] == user\_id and t["return\_date"] is None:

Checks if transaction belongs to that book and user.

Also ensures book hasn’t been returned yet (return\_date is None).

**25. Fine Calculation and Logging**

fine = late\_days \* FINE\_PER\_DAY

if fine:

logging.info(f"Fine for {user\_id} on {book\_id}: {fine}")

Calculates fine only if there are late days.

Logs fine in library.log for record.

**26. Purpose of timedelta**

timedelta represents a difference between two dates.  
Used to calculate due days vs return days.  
Helps find late days → used for fine calculation.

## 

## 

## 

## **27. Approach Followed**

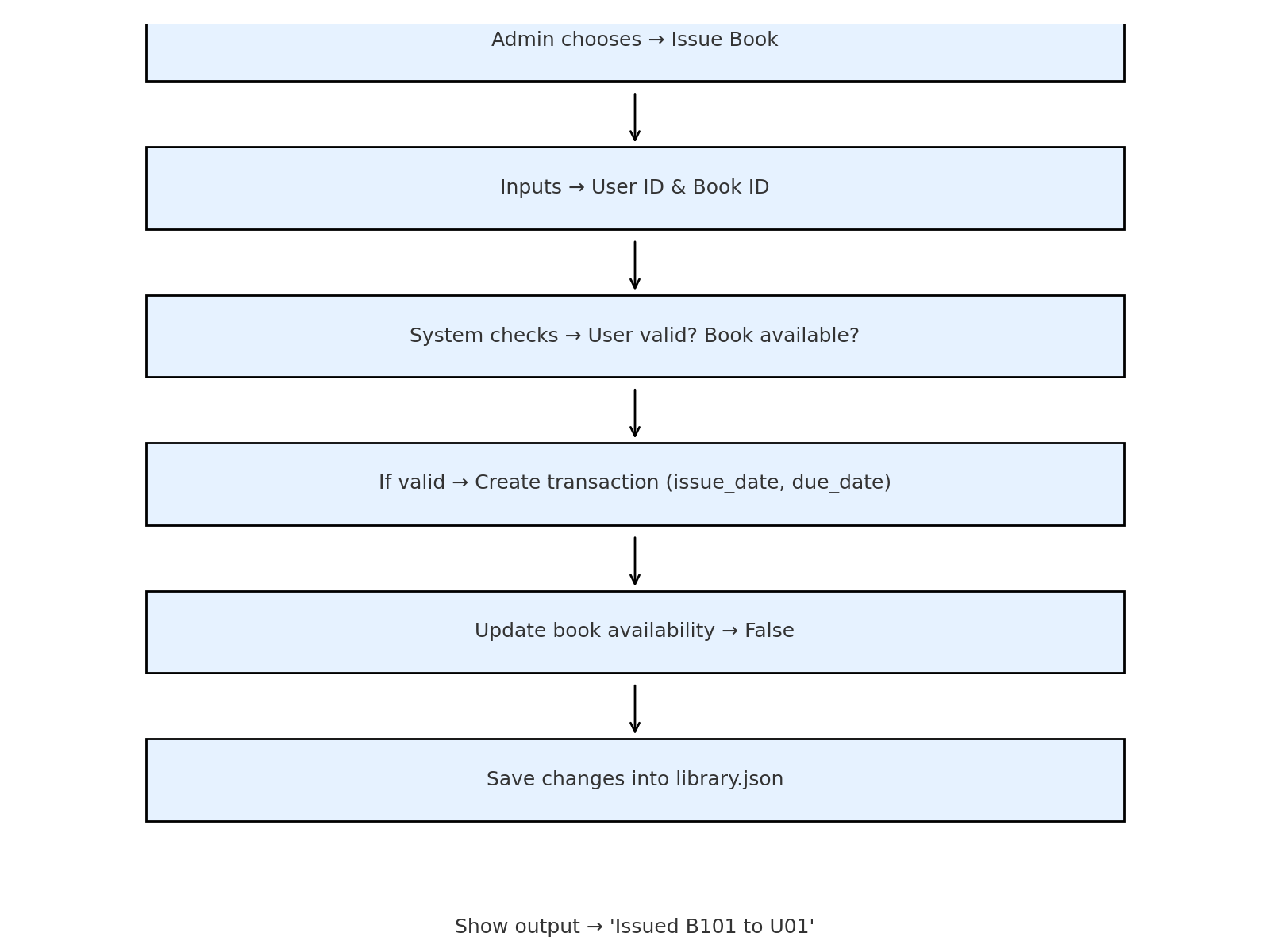
* Object-Oriented Design → Created Book, User, and LibrarySystem classes for modular design.
* Persistence with JSON → So data is not lost after the program closes.
* Error Handling with Exceptions → Ensures invalid operations (like issuing unavailable books) are caught.
* Logging → To keep a history of important actions (audit trail).
* Menu-Driven CLI → Easy for users to interact without needing to understand the codes.

## **28. Design of the Code**

Data Layer → JSON file for storage.  
Model Layer → Book and User classes (data representation).  
Business Logic Layer → LibrarySystem class (rules like issuing, returning, fines).  
Presentation Layer → run\_cli() menu interface for interaction.  
Portability – JSON can be easily shared across different applications or platforms.

## **29. Flow of Data in the Project**

1. Input from User (via CLI)  
   * User/admin selects an option from the menu (e.g., Add Book, Issue Book).
   * Inputs are taken through input() (like Book ID, Title, User ID, etc.).
2. Processing in LibrarySystem (Business Logic)  
   * The LibrarySystem methods (add\_book, register\_user, issue\_book, etc.) handle the logic.
   * For example:  
     + If Add Book → creates a Book object.
     + If Register User → creates a User object.
     + If Issue Book → updates transactions with issue date & due date.
3. Update Data Structures (in Memory)  
   * Books are stored in a dictionary → self.books.
   * Users are stored in a dictionary → self.users.
   * Transactions are stored in a list → self.transactions.
4. Persistence Layer (JSON File)  
   * Every time a change happens, \_save() writes updated data into library.json.
   * On program start, \_load() reads back data from JSON into memory.
5. Output to User  
   * After processing, results are shown on screen. Examples:  
     + *“Book added successfully”*
     + *“Returned. Fine: 20”*
     + Listing all books/users/loans.



**30. Conclusion**

The e-Library Management System effectively proves that Python is capable of creating a useful and effective computer solution to manage a library. The system, designed using Object-Oriented Programming (OOP) concepts, is modular and simple to maintain. With the employment of JSON, data is stored permanently in a plain, lightweight, and human-readable form, and exception handling enhances reliability through protection against invalid operations and provision of informative error messages.

While the system is presently CLI-oriented, it provides a good platform for further additions like graphical interfaces, web access, database support, and multi-user capabilities. In general, this project is not just a mini library management system but also a model for learning fundamental programming ideas, persistent storage, and realistic problem-solving with Python.