**Library Book Cataloger and Recommender - Project Report**

**Project Title: Library Book Cataloger and Recommender CLI Application**

**School: LADOKE AKINTOLA UNIVERSITY**

**Department: COMPUTER SCIENCE**

**Lecturer: MR AROWOYELE ZACCHAEUS**

**Course Code: CSC 202**

Contents

[**Library Book Cataloger and Recommender - Project Report** 1](#_Toc204730044)

[0. INTRODUCTION. 4](#_Toc204730045)

[**1. Project Overview and Goals** 5](#_Toc204730046)

[**2. Requirements Specification** 5](#_Toc204730047)

[**Functional Requirements:** 5](#_Toc204730048)

[**Non-Functional Requirements:** 6](#_Toc204730049)

[**3. User Guide** 7](#_Toc204730050)

[**3.1. Installation and Setup** 7](#_Toc204730051)

[**3.2. Main Menu Navigation** 7](#_Toc204730052)

[**3.3. Detailed Command Usage and Real-time Data Examples** 7](#_Toc204730053)

[**3.4. Error Handling and Input Validation** 11](#_Toc204730054)

[**4. Technical Documentation** 12](#_Toc204730055)

[**4.1. Architecture and Design** 12](#_Toc204730056)

[**4.2. Class Hierarchy and Object-Oriented Principles** 12](#_Toc204730057)

[**4.3. Data Persistence** 12](#_Toc204730058)

[**4.4. Error Handling and Recovery** 12](#_Toc204730059)

[**4.5. Standard Library Modules Used** 13](#_Toc204730060)

[**4.6. Key Algorithms and Logic** 13](#_Toc204730061)

[**4.7. Code Modules** 14](#_Toc204730062)

[**5. Unit Tests** 35](#_Toc204730063)

[**How to Run Tests:** 35](#_Toc204730064)

[**Covered Test Cases:** 35](#_Toc204730065)

[Data/books.json 35](#_Toc204730066)

[Data/loans.json 36](#_Toc204730067)

[Data/users.json 36](#_Toc204730068)

[test\_data\_handler 37](#_Toc204730069)

[test\_library\_manager 41](#_Toc204730070)

[test\_model 46](#_Toc204730071)

[test\_recommender 50](#_Toc204730072)

[test\_utils 54](#_Toc204730073)

[**6. Conclusion** 57](#_Toc204730074)

# INTRODUCTION.

In an age where digital transformation is reshaping every industry, libraries too must evolve to offer efficient, intelligent, and accessible services. This project — *Library Book Cataloger and Recommender* — is a Command Line Interface (CLI) application developed in Python using only standard libraries. Designed to digitize library operations, the application simplifies book cataloging, user management, loan tracking, and intelligent recommendations. It integrates object-oriented design, file handling, modular architecture, and robust input validation to deliver a practical, real-world solution. Through this project, we demonstrate not only technical proficiency but also the ability to solve meaningful problems through software.

**Group Members:**

* RAJI MARIAM ADEGOKE – 2023007709 [ LEADER ]
* AGBOOLA FORTUNE OLUWADUROTIMI - 2023007189
* BAKARE FAHD KOLADE - 2024006286
* YUSUF IBRAHIM OPEYEMI - 2023006301

**1. Project Overview and Goals**

The Library Book Cataloger and Recommender is a Command Line Interface (CLI) application designed to manage a library's book collection, user interactions, and loan tracking. Its primary goal is to provide a robust, persistent, and user-friendly system for librarians and library patrons to interact with the library's resources.

**Key Goals:**

* **Efficient Cataloging:** Allow for easy addition, modification, and searching of book metadata.
* **User Management:** Facilitate user registration and tracking of their borrowing activities.
* **Loan Management:** Implement a system for checking out, returning, and managing overdue books with fine calculations.
* **Personalized Recommendations:** Offer book recommendations based on user reading history and general popularity.
* **Data Analysis:** Provide insights into collection statistics, usage patterns, and acquisition suggestions to aid library development.
* **Robustness:** Ensure data persistence, comprehensive error handling, and modular, maintainable code.

**2. Requirements Specification**

This project adheres to the following requirements:

**Functional Requirements:**

* **Book Cataloging:**
  + Add new books with ISBN, title, author, publication year, genre, description, and initial quantity.
  + Update quantity of existing books.
  + Remove books (only if no copies are currently checked out).
  + List all books in the catalog.
  + Search for books by title, author, or ISBN.
* **User Management:**
  + Register new users with a unique user ID, name, and contact information.
  + List all registered users.
* **Loan Management:**
  + Checkout books to registered users, decrementing available copies.
  + Return books, incrementing available copies and updating loan records.
  + Calculate overdue fines based on a daily rate for late returns.
  + List all currently active loans.
  + List all overdue loans with calculated fines.
* **Recommendation Algorithms:**
  + Recommend books based on a user's past reading history (e.g., similar genres).
  + Recommend generally popular books (most checked out).
  + Recommend new arrivals.
* **Collection Analysis & Reports:**
  + Display overall collection statistics (total unique books, total copies, available vs. checked out).
  + Analyze and display usage patterns (e.g., top 5 most popular books by checkouts).
  + Provide acquisition recommendations (e.g., based on high demand/low stock, or popular genres with few titles).
  + Generate a comprehensive library report summarizing all key statistics and recommendations.
* **Data Persistence:** All book, user, and loan data must be saved to and loaded from files to ensure data is not lost between application sessions.

**Non-Functional Requirements:**

* **Command Line Interface (CLI):** The application must be entirely command-line driven; no GUI or web interface.
* **Python 3.8+:** Developed using Python 3.8 or newer.
* **Standard Library Only:** Utilizes only modules from Python's standard library.
* **Object-Oriented Design (OOD):** Implements proper class hierarchies with inheritance and polymorphism where applicable (e.g., Book, User, Loan classes).
* **Comprehensive Error Handling:** Gracefully handles invalid user inputs, file I/O errors, and other potential runtime issues.
* **Modular Code:** Code is organized into logical modules (separate Python files) for readability and maintainability.
* **Well-Documented Code:** Includes clear comments and docstrings for classes, methods, and complex logic.
* **Input Validation:** Robust validation for all user inputs to prevent incorrect data entry.
* **Unit Tests:** Provides unit tests for critical functions to ensure correctness.

**3. User Guide**

**3.1. Installation and Setup**

1. **Create Project Structure:** Create a main directory (e.g., library\_system). Inside it, create two subdirectories: data and tests.
2. library\_system/
3. ├── data/
4. ├── tests/
5. └── (Python files will go here)
6. **Save the Code:** Save each Python code block provided in Section 4.7 into its respective file name (e.g., models.py, data\_handler.py, etc.) directly under library\_system/, except for test\_library.py which goes into library\_system/tests/.
7. **Run from Terminal:** Open your terminal or command prompt.
8. **Navigate to Directory:** Change your current directory to the library\_system folder.
9. **Execute:** Run the application using the command:
10. python library\_app.py

Upon first run, the data/ directory will be used to store books.json, users.json, and loans.json.

**3.2. Main Menu Navigation**

The application will present a main menu. Enter the number corresponding to your desired action and press Enter.

--- Library Management System Menu ---

1. Book Management

2. User Management

3. Loan Management

4. Reports & Statistics

5. Book Recommendations

6. Exit

--------------------------------------

Enter your choice:

**3.3. Detailed Command Usage and Real-time Data Examples**

**Example Scenario: Setting up and using the Library**

Let's walk through a typical interaction with the CLI application.

**Step 1: Start the Application**

Welcome to the Library Management System!

--- Library Management System Menu ---

1. Book Management

2. User Management

3. Loan Management

4. Reports & Statistics

5. Book Recommendations

6. Exit

--------------------------------------

Enter your choice:

**Step 2: Add a Book (Book Management -> Add/Update Book)**

* Enter 1 for Book Management.
* Enter 1 for Add/Update Book.
* Provide book details:
* --- Book Management ---
* 1. Add/Update Book
* 2. Remove Book
* 3. List All Books
* 4. Search Books
* 5. Back to Main Menu
* Enter your choice: 1
* Enter ISBN: 978-0743273565
* Enter Title: The Great Gatsby
* Enter Author: F. Scott Fitzgerald
* Enter Publication Year: 1925
* Enter Genre: Classic
* Enter Description (optional, press Enter to skip): A classic American novel.
* Enter Quantity (default 1): 3
* Book 'The Great Gatsby' (ISBN: 978-0743273565) added to the catalog.
* Add another book:
* Enter ISBN: 978-0321765723
* Enter Title: The Lord of the Rings
* Enter Author: J.R.R. Tolkien
* Enter Publication Year: 1954
* Enter Genre: Fantasy
* Enter Description (optional, press Enter to skip):
* Enter Quantity (default 1): 5
* Book 'The Lord of the Rings' (ISBN: 978-0321765723) added to the catalog.

**Step 3: Register a User (User Management -> Register User)**

* Back in the main menu, enter 2 for User Management.
* Enter 1 for Register User.
* Provide user details:
* --- User Management ---
* 1. Register User
* 2. List All Users
* 3. Back to Main Menu
* Enter your choice: 1
* Enter User ID: U001
* Enter User Name: Alice Smith
* Enter Contact Info (optional, press Enter to skip): alice@example.com
* User 'Alice Smith' (ID: U001) registered successfully.

**Step 4: Checkout a Book (Loan Management -> Checkout Book)**

* Back in the main menu, enter 3 for Loan Management.
* Enter 1 for Checkout Book.
* Provide loan details:
* --- Loan Management ---
* 1. Checkout Book
* 2. Return Book
* 3. List Current Loans
* 4. List Overdue Loans
* 5. Back to Main Menu
* Enter your choice: 1
* Enter User ID: U001
* Enter Book ISBN: 978-0743273565
* Enter loan duration in days (default 14): 7
* 'The Great Gatsby' (ISBN: 978-0743273565) checked out by 'Alice Smith'. Due date: YYYY-MM-DD (7 days from today)

*(Note: The actual due date will depend on the current date.)*

**Step 5: List Current Loans (Loan Management -> List Current Loans)**

* While still in Loan Management, enter 3 for List Current Loans.
* Enter your choice: 3
* --- Current Loans ---
* Loan ID: 001, Book: 'The Great Gatsby' (ISBN: 978-0743273565), User: Alice Smith (ID: U001), Due: YYYY-MM-DD

**Step 6: Return a Book (Loan Management -> Return Book)**

* While still in Loan Management, enter 2 for Return Book.
* Provide the Loan ID (e.g., 001 from the previous step):
* Enter your choice: 2
* Enter Loan ID to return: 001
* Book returned successfully.

If the book was returned late, it would show an overdue fine.

**Step 7: View Collection Statistics (Reports & Statistics -> View Collection Statistics)**

* Back in the main menu, enter 4 for Reports & Statistics.
* Enter 1 for View Collection Statistics.
* --- Reports & Statistics ---
* 1. View Collection Statistics
* 2. View Usage Patterns (Popular Books)
* 3. View Acquisition Recommendations
* 4. Generate Full Library Report
* 5. Back to Main Menu
* Enter your choice: 1
* --- Collection Statistics ---
* Total Unique Books: 2
* Total Copies in Library: 8
* Available Copies: 8
* Checked Out Copies: 0
* Copies by Genre:
* - Classic: 3 copies
* - Fantasy: 5 copies

**Step 8: Get Book Recommendations (Book Recommendations -> Recommend Popular Books)**

* Back in the main menu, enter 5 for Book Recommendations.
* Enter 2 for Recommend Popular Books.
* --- Book Recommendations ---
* 1. Recommend by User Reading History
* 2. Recommend Popular Books
* 3. Recommend New Arrivals
* 4. Back to Main Menu
* Enter your choice: 2
* How many recommendations? (default 5): 1
* --- Popular Book Recommendations ---
* 1. 'The Great Gatsby' by F. Scott Fitzgerald (Genre: Classic)

*(Note: 'The Great Gatsby' is popular because it was checked out and returned, adding to its checkout count.)*

**Step 9: Exit the Application**

* Back in the main menu, enter 6 for Exit.
* Enter your choice: 6
* Exiting Library Management System. Goodbye!

**3.4. Error Handling and Input Validation**

The application includes robust input validation. If you enter invalid data (e.g., text for a number, incorrect date format), it will prompt you again with an error message until valid input is provided. File operation errors are also caught and reported.

Example of invalid input:

Enter Publication Year: abc

Invalid input. Please try again.

Enter Publication Year: 1999

**4. Technical Documentation**

**4.1. Architecture and Design**

The application follows a modular, object-oriented design pattern to separate concerns and enhance maintainability.

**4.2. Class Hierarchy and Object-Oriented Principles**

* **Encapsulation:** Each class (Book, User, Loan, DataHandler, LibraryManager, RecommendationEngine) encapsulates its data and the methods that operate on that data.
* **Polymorphism & Inheritance:** While explicit inheritance hierarchies beyond the base data models are not extensively used for core library entities (as they are distinct), the design allows for easy extension. For example, Book could be extended to EBook or AudioBook if needed, and Loan could have subclasses for InterLibraryLoan. The \_load\_data and \_save\_data methods in DataHandler demonstrate a form of polymorphism by accepting different object classes (obj\_cls) and handling their serialization/deserialization generically.
* **Composition:** LibraryManager composes DataHandler and RecommendationEngine instances, demonstrating a "has-a" relationship, where the manager uses these components to perform its functions.

**4.3. Data Persistence**

* Data is stored in JSON files within a data/ directory relative to the application's execution path.
* books.json: Stores a dictionary of Book objects, keyed by ISBN.
* users.json: Stores a dictionary of User objects, keyed by User ID.
* loans.json: Stores a dictionary of Loan objects, keyed by Loan ID.
* The to\_dict() and from\_dict() methods in the model classes facilitate seamless serialization and deserialization to/from JSON.

**4.4. Error Handling and Recovery**

* **Input Validation:** The utils.py module provides functions (get\_validated\_input, validate\_positive\_int, validate\_date, etc.) to ensure user inputs conform to expected formats and types. Invalid inputs result in specific error messages and re-prompting.
* **File I/O Errors:** DataHandler uses try-except blocks to catch IOError (e.g., permission issues, file not found) and json.JSONDecodeError (for corrupted JSON files). It logs the errors and provides user-friendly messages.
* **Business Logic Errors:** LibraryManager methods include try-except blocks to handle ValueError (e.g., for empty required fields when creating objects) and other unexpected exceptions, logging them and providing informative messages to the user.
* **Logging:** The logging module is configured to record informational messages, warnings, and errors to the console (and could easily be extended to a file) for debugging and auditing purposes.

**4.5. Standard Library Modules Used**

This project extensively uses the following standard library modules:

1. **sys**: Used for exiting the application (sys.exit()).
2. **os**: Used for path manipulation (os.path.join), directory creation (os.makedirs), and checking file existence (os.path.exists).
3. **json**: Crucial for data persistence, handling the serialization and deserialization of Python objects to and from JSON format.
4. **datetime**: Used for handling dates (e.g., checkout dates, due dates, return dates) and calculating durations (e.g., for overdue fines).
5. **logging**: Provides a flexible framework for emitting log messages from different parts of the application, aiding in debugging and monitoring.
6. **collections**: Specifically defaultdict is used for easily counting occurrences (e.g., genres, book checkouts) without needing to check if a key exists first.
7. **random**: Used in the RecommendationEngine for shuffling potential recommendations to provide varied suggestions.
8. **unittest**: Used for writing and running unit tests to verify the correctness of individual components.
9. **shutil**: Used in the test suite for removing directories (shutil.rmtree) to ensure clean test environments.

**4.6. Key Algorithms and Logic**

* **Overdue Fine Calculation:**
  + Compares the return\_date (or current date if not returned) with the due\_date.
  + Calculates the difference in days.
  + Multiplies the number of overdue days by a predefined overdue\_fine\_per\_day rate.
* **Recommendation Algorithms:**
  + **Reading History:** Gathers genres from a user's reading\_history, counts their frequency, and then suggests available books from the most frequent genres that the user hasn't read.
  + **Popular Books:** Counts total checkouts for each book and sorts them in descending order to identify the most popular.
  + **Acquisition Recommendations:** Identifies popular books with low available stock and genres that have high user interest (based on reading history) but a relatively small number of titles in the collection.
* **Data Loading/Saving:** Iterates through dictionaries of objects, converting them to/from their dictionary representations using to\_dict() and from\_dict() methods for JSON serialization/deserialization.

**4.7. Code Modules**

**4.7.1. models.py**

# models.py

import datetime

class Book:

"""Represents a book in the library."""

def \_\_init\_\_(self, isbn, title, author, publication\_year, genre, description=None, quantity=1):

if not all([isbn, title, author, publication\_year, genre]):

raise ValueError("ISBN, title, author, publication year, and genre cannot be empty.")

if not isinstance(quantity, int) or quantity < 0:

raise ValueError("Quantity must be a non-negative integer.")

self.isbn = str(isbn)

self.title = title

self.author = author

self.publication\_year = int(publication\_year)

self.genre = genre

self.description = description if description else "No description available."

self.quantity = quantity

self.available\_copies = quantity

def \_\_repr\_\_(self):

return f"Book(ISBN='{self.isbn}', Title='{self.title}', Author='{self.author}')"

def to\_dict(self):

"""Converts the Book object to a dictionary for JSON serialization."""

return {

"isbn": self.isbn,

"title": self.title,

"author": self.author,

"publication\_year": self.publication\_year,

"genre": self.genre,

"description": self.description,

"quantity": self.quantity,

"available\_copies": self.available\_copies

}

@classmethod

def from\_dict(cls, data):

"""Creates a Book object from a dictionary."""

book = cls(

isbn=data['isbn'],

title=data['title'],

author=data['author'],

publication\_year=data['publication\_year'],

genre=data['genre'],

description=data.get('description'),

quantity=data['quantity']

)

book.available\_copies = data.get('available\_copies', data['quantity'])

return book

class User:

"""Represents a library user."""

def \_\_init\_\_(self, user\_id, name, contact\_info=None):

if not all([user\_id, name]):

raise ValueError("User ID and name cannot be empty.")

self.user\_id = str(user\_id)

self.name = name

self.contact\_info = contact\_info if contact\_info else "N/A"

self.reading\_history = [] # List of ISBNs

self.current\_loans = [] # List of Loan IDs

def \_\_repr\_\_(self):

return f"User(ID='{self.user\_id}', Name='{self.name}')"

def to\_dict(self):

"""Converts the User object to a dictionary for JSON serialization."""

return {

"user\_id": self.user\_id,

"name": self.name,

"contact\_info": self.contact\_info,

"reading\_history": self.reading\_history,

"current\_loans": self.current\_loans

}

@classmethod

def from\_dict(cls, data):

"""Creates a User object from a dictionary."""

user = cls(

user\_id=data['user\_id'],

name=data['name'],

contact\_info=data.get('contact\_info')

)

user.reading\_history = data.get('reading\_history', [])

user.current\_loans = data.get('current\_loans', [])

return user

class Loan:

"""Represents a book loan."""

def \_\_init\_\_(self, loan\_id, book\_isbn, user\_id, checkout\_date, due\_date, return\_date=None):

if not all([loan\_id, book\_isbn, user\_id, checkout\_date, due\_date]):

raise ValueError("Loan ID, book ISBN, user ID, checkout date, and due date cannot be empty.")

self.loan\_id = str(loan\_id)

self.book\_isbn = str(book\_isbn)

self.user\_id = str(user\_id)

self.checkout\_date = checkout\_date # Stored as 'YYYY-MM-DD' string

self.due\_date = due\_date # Stored as 'YYYY-MM-DD' string

self.return\_date = return\_date # Stored as 'YYYY-MM-DD' string or None

def \_\_repr\_\_(self):

return f"Loan(ID='{self.loan\_id}', Book='{self.book\_isbn}', User='{self.user\_id}')"

def to\_dict(self):

"""Converts the Loan object to a dictionary for JSON serialization."""

return {

"loan\_id": self.loan\_id,

"book\_isbn": self.book\_isbn,

"user\_id": self.user\_id,

"checkout\_date": self.checkout\_date,

"due\_date": self.due\_date,

"return\_date": self.return\_date

}

@classmethod

def from\_dict(cls, data):

"""Creates a Loan object from a dictionary."""

return cls(

loan\_id=data['loan\_id'],

book\_isbn=data['book\_isbn'],

user\_id=data['user\_id'],

checkout\_date=data['checkout\_date'],

due\_date=data['due\_date'],

return\_date=data.get('return\_date')

)

**4.7.2. data\_handler.py**

# data\_handler.py

import os

import json

import logging

# Import models to use their from\_dict and to\_dict methods

from models import Book, User, Loan

class DataHandler:

"""Handles loading and saving library data to JSON files."""

def \_\_init\_\_(self, data\_dir="data"):

self.data\_dir = data\_dir

os.makedirs(self.data\_dir, exist\_ok=True)

self.books\_file = os.path.join(self.data\_dir, "books.json")

self.users\_file = os.path.join(self.data\_dir, "users.json")

self.loans\_file = os.path.join(self.data\_dir, "loans.json")

def \_load\_data(self, filepath, obj\_cls):

"""Helper to load data from a JSON file."""

data = {}

try:

if os.path.exists(filepath):

with open(filepath, 'r', encoding='utf-8') as f:

raw\_data = json.load(f)

for key, item\_data in raw\_data.items():

try:

data[key] = obj\_cls.from\_dict(item\_data)

except (TypeError, ValueError) as e:

logging.error(f"Error loading {obj\_cls.\_\_name\_\_} from {filepath}: {item\_data}. Error: {e}")

print(f"Warning: Skipping malformed record in {filepath}. Check logs for details.")

return data

except json.JSONDecodeError as e:

logging.error(f"Error decoding JSON from {filepath}: {e}")

print(f"Error: Could not read data from {filepath}. File might be corrupted.")

return {}

except IOError as e:

logging.error(f"File I/O error loading {filepath}: {e}")

print(f"Error: Could not access {filepath}. Check permissions.")

return {}

def \_save\_data(self, filepath, data\_dict):

"""Helper to save data to a JSON file."""

try:

with open(filepath, 'w', encoding='utf-8') as f:

json.dump({k: v.to\_dict() for k, v in data\_dict.items()}, f, indent=4)

return True

except IOError as e:

logging.error(f"File I/O error saving {filepath}: {e}")

print(f"Error: Could not save data to {filepath}. Check permissions.")

return False

except TypeError as e:

logging.error(f"Type error during JSON serialization for {filepath}: {e}")

print(f"Error: Data format issue preventing save to {filepath}.")

return False

def load\_books(self):

"""Loads book data."""

return self.\_load\_data(self.books\_file, Book)

def save\_books(self, books):

"""Saves book data."""

return self.\_save\_data(self.books\_file, books)

def load\_users(self):

"""Loads user data."""

return self.\_load\_data(self.users\_file, User)

def save\_users(self, users):

"""Saves user data."""

return self.\_save\_data(self.users\_file, users)

def load\_loans(self):

"""Loads loan data."""

return self.\_load\_data(self.loans\_file, Loan)

def save\_loans(self, loans):

"""Saves loan data."""

return self.\_save\_data(self.loans\_file, loans)

**4.7.3. utils.py**

# utils.py

import datetime

def get\_validated\_input(prompt, validation\_func=None, error\_message="Invalid input. Please try again."):

"""

Prompts the user for input and validates it using a provided function.

Keeps prompting until valid input is received.

"""

while True:

user\_input = input(prompt).strip()

if validation\_func:

try:

validated\_value = validation\_func(user\_input)

return validated\_value

except ValueError:

print(error\_message)

except Exception as e:

# Catch unexpected errors during validation

print("An unexpected error occurred during input validation.")

# In a real app, you'd log this: logging.error(f"Unexpected error during input validation: {e}")

else:

return user\_input

def validate\_positive\_int(value\_str):

"""Validates if a string represents a positive integer."""

try:

value = int(value\_str)

if value <= 0:

raise ValueError

return value

except ValueError:

raise ValueError("Input must be a positive whole number.")

def validate\_non\_negative\_int(value\_str):

"""Validates if a string represents a non-negative integer."""

try:

value = int(value\_str)

if value < 0:

raise ValueError

return value

except ValueError:

raise ValueError("Input must be a non-negative whole number.")

def validate\_date(date\_str):

"""Validates if a string is a valid date in YYYY-MM-DD format."""

try:

datetime.datetime.strptime(date\_str, '%Y-%m-%d')

return date\_str

except ValueError:

raise ValueError("Date must be in YYYY-MM-DD format.")

def generate\_unique\_id(existing\_ids):

"""Generates a unique ID (simple counter for demonstration)."""

if not existing\_ids:

return "001"

# Ensure all existing\_ids are valid integers before finding max

try:

max\_id = max([int(uid) for uid in existing\_ids if uid.isdigit()])

except ValueError: # Handle case where existing\_ids might not be purely numeric

max\_id = 0 # Start from 0 if existing IDs are not numeric or empty

return str(max\_id + 1).zfill(3)

**4.7.4. recommender.py**

# recommender.py

import random

from collections import defaultdict

# Import models for type hinting and access to book/user properties

from models import Book, User

class RecommendationEngine:

"""Provides book recommendations based on various criteria."""

def \_\_init\_\_(self, library\_manager):

# We pass the library\_manager instance to allow the recommender

# to access the current state of books, users, and loans.

self.library = library\_manager

def recommend\_by\_reading\_history(self, user\_id, num\_recommendations=3):

"""

Recommends books based on a user's reading history.

Prioritizes books of similar genres that the user hasn't read yet.

"""

user = self.library.get\_user(user\_id)

if not user or not user.reading\_history:

# Fallback to popular books if no reading history

return self.recommend\_popular\_books(num\_recommendations)

read\_genres = defaultdict(int)

for isbn in user.reading\_history:

book = self.library.get\_book(isbn)

if book:

read\_genres[book.genre] += 1

if not read\_genres:

return self.recommend\_popular\_books(num\_recommendations)

# Sort genres by frequency to prioritize recommendations from most read genres

sorted\_genres = sorted(read\_genres.items(), key=lambda item: item[1], reverse=True)

recommendations = []

potential\_books = []

# Gather potential books from preferred genres that are available and not yet read

for genre, \_ in sorted\_genres:

for book\_isbn, book in self.library.books.items():

if book.genre == genre and book.available\_copies > 0 and book\_isbn not in user.reading\_history:

potential\_books.append(book)

# Shuffle potential books to add randomness and pick the desired number

random.shuffle(potential\_books)

for book in potential\_books:

if len(recommendations) < num\_recommendations:

recommendations.append(book)

else:

break

return recommendations

def recommend\_popular\_books(self, num\_recommendations=5):

"""Recommends the most frequently checked-out books that are available."""

checkout\_counts = defaultdict(int)

for loan in self.library.loans.values():

checkout\_counts[loan.book\_isbn] += 1

# Sort books by their checkout counts in descending order

sorted\_books\_by\_popularity = sorted(checkout\_counts.items(), key=lambda item: item[1], reverse=True)

recommendations = []

for isbn, \_ in sorted\_books\_by\_popularity:

book = self.library.get\_book(isbn)

# Only recommend books that are available

if book and book.available\_copies > 0:

recommendations.append(book)

if len(recommendations) >= num\_recommendations:

break

return recommendations[:num\_recommendations]

def recommend\_new\_arrivals(self, num\_recommendations=5):

"""Recommends recently added books (based on publication year)."""

# Sort all books by publication year in descending order

new\_books = sorted(self.library.books.values(), key=lambda b: b.publication\_year, reverse=True)

# Filter for available copies

available\_new\_books = [b for b in new\_books if b.available\_copies > 0]

return available\_new\_books[:num\_recommendations]

**4.7.5. library\_manager.py**

# library\_manager.py

import datetime

import logging

from collections import defaultdict

# Import classes and functions from other modules

from models import Book, User, Loan

from data\_handler import DataHandler

from recommender import RecommendationEngine

from utils import generate\_unique\_id

class LibraryManager:

"""Manages all library operations, including books, users, and loans."""

def \_\_init\_\_(self, data\_handler):

self.data\_handler = data\_handler

# Load all data upon initialization

self.books = self.data\_handler.load\_books()

self.users = self.data\_handler.load\_users()

self.loans = self.data\_handler.load\_loans()

# Initialize the recommendation engine, passing self (the manager instance)

self.recommender = RecommendationEngine(self)

self.overdue\_fine\_per\_day = 0.50 # Example fine rate

def \_save\_all\_data(self):

"""Saves all current data to files."""

self.data\_handler.save\_books(self.books)

self.data\_handler.save\_users(self.users)

self.data\_handler.save\_loans(self.loans)

logging.info("All library data saved.")

def add\_book(self, isbn, title, author, publication\_year, genre, description=None, quantity=1):

"""Adds a new book or increases quantity of an existing book."""

try:

if isbn in self.books:

# If book exists, increase its quantity and available copies

self.books[isbn].quantity += quantity

self.books[isbn].available\_copies += quantity

logging.info(f"Increased quantity for existing book: {title} (ISBN: {isbn})")

print(f"Quantity for '{title}' (ISBN: {isbn}) increased to {self.books[isbn].quantity}.")

else:

# Otherwise, create and add a new book

book = Book(isbn, title, author, publication\_year, genre, description, quantity)

self.books[isbn] = book

logging.info(f"Added new book: {title} (ISBN: {isbn})")

print(f"Book '{title}' (ISBN: {isbn}) added to the catalog.")

self.\_save\_all\_data()

return True

except ValueError as e:

logging.error(f"Error adding book: {e}")

print(f"Error: {e}")

return False

except Exception as e:

logging.error(f"An unexpected error occurred while adding book: {e}", exc\_info=True)

print("An unexpected error occurred. Please check logs.")

return False

def remove\_book(self, isbn):

"""Removes a book from the catalog."""

if isbn not in self.books:

print(f"Error: Book with ISBN '{isbn}' not found.")

return False

# Prevent removal if any copies are currently checked out

if self.books[isbn].available\_copies < self.books[isbn].quantity:

print(f"Error: Cannot remove book '{self.books[isbn].title}' (ISBN: {isbn}). Some copies are currently checked out.")

return False

del self.books[isbn]

# Clean up any associated loan records for this book (should be none if copies are out)

loans\_to\_remove = [loan\_id for loan\_id, loan in self.loans.items() if loan.book\_isbn == isbn]

for loan\_id in loans\_to\_remove:

user = self.get\_user(self.loans[loan\_id].user\_id)

if user and loan\_id in user.current\_loans:

user.current\_loans.remove(loan\_id)

del self.loans[loan\_id]

logging.info(f"Removed book with ISBN: {isbn}")

print(f"Book with ISBN '{isbn}' removed from the catalog.")

self.\_save\_all\_data()

return True

def get\_book(self, isbn):

"""Retrieves a book by its ISBN."""

return self.books.get(isbn)

def list\_all\_books(self):

"""Lists all books in the catalog."""

if not self.books:

print("No books in the catalog.")

return []

print("\n--- All Books in Catalog ---")

book\_list = []

for book in self.books.values():

print(f"ISBN: {book.isbn}, Title: '{book.title}', Author: {book.author}, Genre: {book.genre}, Available: {book.available\_copies}/{book.quantity}")

book\_list.append(book)

return book\_list

def search\_books(self, query):

"""Searches for books by title, author, or ISBN."""

query = query.lower()

results = [

book for book in self.books.values()

if query in book.title.lower() or

query in book.author.lower() or

query in book.isbn.lower()

]

if not results:

print(f"No books found matching '{query}'.")

return []

print(f"\n--- Search Results for '{query}' ---")

for book in results:

print(f"ISBN: {book.isbn}, Title: '{book.title}', Author: {book.author}, Genre: {book.genre}, Available: {book.available\_copies}/{book.quantity}")

return results

def register\_user(self, user\_id, name, contact\_info=None):

"""Registers a new user."""

try:

if user\_id in self.users:

print(f"Error: User with ID '{user\_id}' already exists.")

return False

user = User(user\_id, name, contact\_info)

self.users[user\_id] = user

logging.info(f"Registered new user: {name} (ID: {user\_id})")

print(f"User '{name}' (ID: {user\_id}) registered successfully.")

self.\_save\_all\_data()

return True

except ValueError as e:

logging.error(f"Error registering user: {e}")

print(f"Error: {e}")

return False

except Exception as e:

logging.error(f"An unexpected error occurred while registering user: {e}", exc\_info=True)

print("An unexpected error occurred. Please check logs.")

return False

def get\_user(self, user\_id):

"""Retrieves a user by their ID."""

return self.users.get(user\_id)

def list\_all\_users(self):

"""Lists all registered users."""

if not self.users:

print("No users registered.")

return []

print("\n--- All Registered Users ---")

user\_list = []

for user in self.users.values():

print(f"ID: {user.user\_id}, Name: {user.name}, Contact: {user.contact\_info}")

user\_list.append(user)

return user\_list

def checkout\_book(self, user\_id, book\_isbn, loan\_duration\_days=14):

"""Checks out a book to a user."""

user = self.get\_user(user\_id)

book = self.get\_book(book\_isbn)

if not user:

print(f"Error: User with ID '{user\_id}' not found.")

return False

if not book:

print(f"Error: Book with ISBN '{book\_isbn}' not found.")

return False

if book.available\_copies <= 0:

print(f"Error: '{book.title}' (ISBN: {book\_isbn}) is currently out of stock.")

return False

# Check if user already has this book checked out

for loan\_id in user.current\_loans:

loan = self.loans.get(loan\_id)

if loan and loan.book\_isbn == book\_isbn and loan.return\_date is None:

print(f"Error: User '{user.name}' already has '{book.title}' checked out.")

return False

book.available\_copies -= 1 # Decrement available copies

loan\_id = generate\_unique\_id(self.loans.keys()) # Generate unique loan ID

checkout\_date = datetime.date.today().strftime('%Y-%m-%d')

due\_date = (datetime.date.today() + datetime.timedelta(days=loan\_duration\_days)).strftime('%Y-%m-%d')

loan = Loan(loan\_id, book\_isbn, user\_id, checkout\_date, due\_date)

self.loans[loan\_id] = loan

user.current\_loans.append(loan\_id) # Add loan ID to user's current loans

logging.info(f"Book '{book.title}' checked out by '{user.name}'. Loan ID: {loan\_id}")

print(f"'{book.title}' (ISBN: {book\_isbn}) checked out by '{user.name}'. Due date: {due\_date}")

self.\_save\_all\_data()

return True

def return\_book(self, loan\_id):

"""Returns a previously checked out book."""

loan = self.loans.get(loan\_id)

if not loan:

print(f"Error: Loan with ID '{loan\_id}' not found.")

return False

if loan.return\_date is not None:

print(f"Error: Book for loan ID '{loan\_id}' has already been returned.")

return False

book = self.get\_book(loan.book\_isbn)

user = self.get\_user(loan.user\_id)

if book:

book.available\_copies += 1 # Increment available copies

else:

logging.warning(f"Book for loan {loan\_id} (ISBN: {loan.book\_isbn}) not found in catalog during return.")

loan.return\_date = datetime.date.today().strftime('%Y-%m-%d') # Set return date

if user:

if loan\_id in user.current\_loans:

user.current\_loans.remove(loan\_id) # Remove loan from user's active loans

user.reading\_history.append(loan.book\_isbn) # Add to user's reading history

else:

logging.warning(f"User for loan {loan\_id} (ID: {loan.user\_id}) not found during return.")

overdue\_fine = self.calculate\_overdue\_fine(loan\_id)

if overdue\_fine > 0:

print(f"Book returned. Overdue fine: ${overdue\_fine:.2f}")

else:

print("Book returned successfully.")

logging.info(f"Book for loan ID {loan\_id} returned. Fine: ${overdue\_fine:.2f}")

self.\_save\_all\_data()

return True

def calculate\_overdue\_fine(self, loan\_id):

"""Calculates the overdue fine for a given loan."""

loan = self.loans.get(loan\_id)

if not loan:

return 0.0

due\_date = datetime.datetime.strptime(loan.due\_date, '%Y-%m-%d').date()

# If return\_date is not set, assume today's date for fine calculation

return\_date = datetime.datetime.strptime(loan.return\_date, '%Y-%m-%d').date() if loan.return\_date else datetime.date.today()

if return\_date > due\_date:

days\_overdue = (return\_date - due\_date).days

return days\_overdue \* self.overdue\_fine\_per\_day

return 0.0

def list\_current\_loans(self):

"""Lists all currently active loans."""

active\_loans = [loan for loan in self.loans.values() if loan.return\_date is None]

if not active\_loans:

print("No active loans.")

return []

print("\n--- Current Loans ---")

loan\_list = []

for loan in active\_loans:

book = self.get\_book(loan.book\_isbn)

user = self.get\_user(loan.user\_id)

book\_title = book.title if book else "Unknown Book"

user\_name = user.name if user else "Unknown User"

print(f"Loan ID: {loan.loan\_id}, Book: '{book\_title}' (ISBN: {loan.book\_isbn}), User: {user\_name} (ID: {loan.user\_id}), Due: {loan.due\_date}")

loan\_list.append(loan)

return loan\_list

def list\_overdue\_loans(self):

"""Lists all overdue loans and their calculated fines."""

overdue\_loans = []

today = datetime.date.today()

for loan in self.loans.values():

if loan.return\_date is None: # Only consider active loans

due\_date = datetime.datetime.strptime(loan.due\_date, '%Y-%m-%d').date()

if today > due\_date:

overdue\_loans.append(loan)

if not overdue\_loans:

print("No overdue loans.")

return []

print("\n--- Overdue Loans ---")

overdue\_list = []

for loan in overdue\_loans:

book = self.get\_book(loan.book\_isbn)

user = self.get\_user(loan.user\_id)

book\_title = book.title if book else "Unknown Book"

user\_name = user.name if user else "Unknown User"

fine = self.calculate\_overdue\_fine(loan.loan\_id) # Calculate fine based on today's date

print(f"Loan ID: {loan.loan\_id}, Book: '{book\_title}', User: {user\_name}, Due: {loan.due\_date}, Overdue Fine: ${fine:.2f}")

overdue\_list.append(loan)

return overdue\_list

def get\_collection\_statistics(self):

"""Analyzes and displays collection statistics."""

total\_books = len(self.books)

total\_copies = sum(book.quantity for book in self.books.values())

available\_copies = sum(book.available\_copies for book in self.books.values())

checked\_out\_copies = total\_copies - available\_copies

genres = defaultdict(int)

for book in self.books.values():

genres[book.genre] += book.quantity

print("\n--- Collection Statistics ---")

print(f"Total Unique Books: {total\_books}")

print(f"Total Copies in Library: {total\_copies}")

print(f"Available Copies: {available\_copies}")

print(f"Checked Out Copies: {checked\_out\_copies}")

print("\nCopies by Genre:")

for genre, count in sorted(genres.items()):

print(f" - {genre}: {count} copies")

return {

"total\_books": total\_books,

"total\_copies": total\_copies,

"available\_copies": available\_copies,

"checked\_out\_copies": checked\_out\_copies,

"genres": dict(genres)

}

def get\_usage\_patterns(self):

"""Analyzes and displays usage patterns (e.g., most popular books)."""

checkout\_counts = defaultdict(int)

for loan in self.loans.values():

checkout\_counts[loan.book\_isbn] += 1

if not checkout\_counts:

print("No usage data available yet.")

return {}

sorted\_popular\_books = sorted(checkout\_counts.items(), key=lambda item: item[1], reverse=True)

print("\n--- Usage Patterns ---")

print("Top 5 Most Popular Books (by checkouts):")

for i, (isbn, count) in enumerate(sorted\_popular\_books[:5]):

book = self.get\_book(isbn)

if book:

print(f" {i+1}. '{book.title}' by {book.author} (Checkouts: {count})")

else:

print(f" {i+1}. Unknown Book (ISBN: {isbn}, Checkouts: {count})")

return dict(checkout\_counts)

def get\_acquisition\_recommendations(self):

"""

Suggests books for acquisition based on current collection gaps or demand.

Simple logic: suggest more copies of popular books with low availability,

or suggest books in popular genres with fewer titles.

"""

recommendations = []

print("\n--- Acquisition Recommendations ---")

# 1. Popular books with low availability

checkout\_counts = defaultdict(int)

for loan in self.loans.values():

checkout\_counts[loan.book\_isbn] += 1

for isbn, count in sorted(checkout\_counts.items(), key=lambda item: item[1], reverse=True):

book = self.get\_book(isbn)

# If a popular book has low availability and few total copies, recommend acquiring more

if book and book.available\_copies < book.quantity / 2 and book.quantity < 5:

recommendations.append(f"Consider acquiring more copies of '{book.title}' (ISBN: {book.isbn}). High demand, low availability ({book.available\_copies}/{book.quantity} available).")

if len(recommendations) >= 3: break # Limit recommendations

# 2. Suggest books for genres with high user interest but fewer titles in collection

user\_genre\_interest = defaultdict(int)

for user in self.users.values():

for isbn in user.reading\_history:

book = self.get\_book(isbn)

if book:

user\_genre\_interest[book.genre] += 1

collection\_genres = defaultdict(int)

for book in self.books.values():

collection\_genres[book.genre] += 1

if user\_genre\_interest:

sorted\_user\_genres = sorted(user\_genre\_interest.items(), key=lambda item: item[1], reverse=True)

for genre, interest\_count in sorted\_user\_genres:

# Arbitrary thresholds for "high interest, low titles"

if collection\_genres[genre] < 10 and interest\_count > 5:

recommendations.append(f"Explore new titles in the '{genre}' genre. High user interest, but relatively small collection ({collection\_genres[genre]} titles).")

if len(recommendations) >= 5: break # Limit recommendations

if not recommendations:

print("No specific acquisition recommendations at this time based on current data.")

else:

for rec in recommendations:

print(f"- {rec}")

return recommendations

def generate\_library\_report(self):

"""Generates a comprehensive library report."""

print("\n" + "="\*50)

print(" LIBRARY COMPREHENSIVE REPORT ")

print("="\*50)

print(f"Report Date: {datetime.date.today().strftime('%Y-%m-%d')}\n")

self.get\_collection\_statistics()

print("\n" + "-"\*50)

self.get\_usage\_patterns()

print("\n" + "-"\*50)

self.list\_overdue\_loans()

print("\n" + "-"\*50)

self.get\_acquisition\_recommendations()

print("\n" + "="\*50)

print(" END OF REPORT ")

print("="\*50 + "\n")

**4.7.6. library\_app.py (Main Application)**

# library\_app.py

"""

Library Book Cataloger and Recommender CLI Application

This is the main entry point for the application. It orchestrates the CLI

interactions and utilizes classes and functions from other modules.

To run this application:

1. Ensure all Python files (models.py, data\_handler.py, utils.py,

recommender.py, library\_manager.py, and this file) are in the same directory.

2. Run `python library\_app.py` from your terminal.

"""

import sys

import logging

import os

# Configure logging for the entire application

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Import necessary classes and functions from other modules

from data\_handler import DataHandler

from library\_manager import LibraryManager

from utils import get\_validated\_input, validate\_positive\_int

def display\_menu():

"""Displays the main menu options to the user."""

print("\n--- Library Management System Menu ---")

print("1. Book Management")

print("2. User Management")

print("3. Loan Management")

print("4. Reports & Statistics")

print("5. Book Recommendations")

print("6. Exit")

print("--------------------------------------")

def book\_management\_menu(manager):

"""Handles book-related operations."""

while True:

print("\n--- Book Management ---")

print("1. Add/Update Book")

print("2. Remove Book")

print("3. List All Books")

print("4. Search Books")

print("5. Back to Main Menu")

choice = get\_validated\_input("Enter your choice: ", validate\_positive\_int, "Please enter a number (1-5).")

if choice == 1:

isbn = get\_validated\_input("Enter ISBN: ")

title = get\_validated\_input("Enter Title: ")

author = get\_validated\_input("Enter Author: ")

publication\_year = get\_validated\_input("Enter Publication Year: ", validate\_positive\_int, "Year must be a positive integer.")

genre = get\_validated\_input("Enter Genre: ")

description = get\_validated\_input("Enter Description (optional, press Enter to skip): ", lambda x: x if x else None)

quantity = get\_validated\_input("Enter Quantity (default 1): ", validate\_positive\_int, "Quantity must be a positive integer.")

manager.add\_book(isbn, title, author, publication\_year, genre, description, quantity)

elif choice == 2:

isbn = get\_validated\_input("Enter ISBN of book to remove: ")

manager.remove\_book(isbn)

elif choice == 3:

manager.list\_all\_books()

elif choice == 4:

query = get\_validated\_input("Enter search query (title, author, or ISBN): ")

manager.search\_books(query)

elif choice == 5:

break

else:

print("Invalid choice. Please try again.")

def user\_management\_menu(manager):

"""Handles user-related operations."""

while True:

print("\n--- User Management ---")

print("1. Register User")

print("2. List All Users")

print("3. Back to Main Menu")

choice = get\_validated\_input("Enter your choice: ", validate\_positive\_int, "Please enter a number (1-3).")

if choice == 1:

user\_id = get\_validated\_input("Enter User ID: ")

name = get\_validated\_input("Enter User Name: ")

contact\_info = get\_validated\_input("Enter Contact Info (optional, press Enter to skip): ", lambda x: x if x else None)

manager.register\_user(user\_id, name, contact\_info)

elif choice == 2:

manager.list\_all\_users()

elif choice == 3:

break

else:

print("Invalid choice. Please try again.")

def loan\_management\_menu(manager):

"""Handles loan-related operations."""

while True:

print("\n--- Loan Management ---")

print("1. Checkout Book")

print("2. Return Book")

print("3. List Current Loans")

print("4. List Overdue Loans")

print("5. Back to Main Menu")

choice = get\_validated\_input("Enter your choice: ", validate\_positive\_int, "Please enter a number (1-5).")

if choice == 1:

user\_id = get\_validated\_input("Enter User ID: ")

book\_isbn = get\_validated\_input("Enter Book ISBN: ")

duration\_str = get\_validated\_input("Enter loan duration in days (default 14): ", lambda x: x if x else "14")

try:

duration = int(duration\_str)

if duration <= 0:

print("Duration must be a positive integer. Using default 14 days.")

duration = 14

except ValueError:

print("Invalid duration. Using default 14 days.")

duration = 14

manager.checkout\_book(user\_id, book\_isbn, duration)

elif choice == 2:

loan\_id = get\_validated\_input("Enter Loan ID to return: ")

manager.return\_book(loan\_id)

elif choice == 3:

manager.list\_current\_loans()

elif choice == 4:

manager.list\_overdue\_loans()

elif choice == 5:

break

else:

print("Invalid choice. Please try again.")

def reports\_statistics\_menu(manager):

"""Handles reports and statistics operations."""

while True:

print("\n--- Reports & Statistics ---")

print("1. View Collection Statistics")

print("2. View Usage Patterns (Popular Books)")

print("3. View Acquisition Recommendations")

print("4. Generate Full Library Report")

print("5. Back to Main Menu")

choice = get\_validated\_input("Enter your choice: ", validate\_positive\_int, "Please enter a number (1-5).")

if choice == 1:

manager.get\_collection\_statistics()

elif choice == 2:

manager.get\_usage\_patterns()

elif choice == 3:

manager.get\_acquisition\_recommendations()

elif choice == 4:

manager.generate\_library\_report()

elif choice == 5:

break

else:

print("Invalid choice. Please try again.")

def recommendations\_menu(manager):

"""Handles book recommendation operations."""

while True:

print("\n--- Book Recommendations ---")

print("1. Recommend by User Reading History")

print("2. Recommend Popular Books")

print("3. Recommend New Arrivals")

print("4. Back to Main Menu")

choice = get\_validated\_input("Enter your choice: ", validate\_positive\_int, "Please enter a number (1-4).")

if choice == 1:

user\_id = get\_validated\_input("Enter User ID for recommendations: ")

user = manager.get\_user(user\_id)

if user:

num\_recs\_str = get\_validated\_input("How many recommendations? (default 3): ", lambda x: x if x else "3")

try:

num\_recs = int(num\_recs\_str)

if num\_recs <= 0:

print("Number of recommendations must be positive. Using default 3.")

num\_recs = 3

except ValueError:

print("Invalid number. Using default 3 recommendations.")

num\_recs = 3

recommendations = manager.recommender.recommend\_by\_reading\_history(user\_id, num\_recs)

if recommendations:

print(f"\n--- Recommendations for {user.name} ---")

for i, book in enumerate(recommendations):

print(f" {i+1}. '{book.title}' by {book.author} (Genre: {book.genre})")

else:

print("Could not generate personalized recommendations. User might not have reading history or no new books in preferred genres. Try popular or new arrivals.")

else:

print(f"User with ID '{user\_id}' not found.")

elif choice == 2:

num\_recs\_str = get\_validated\_input("How many recommendations? (default 5): ", lambda x: x if x else "5")

try:

num\_recs = int(num\_recs\_str)

if num\_recs <= 0:

print("Number of recommendations must be positive. Using default 5.")

num\_recs = 5

except ValueError:

print("Invalid number. Using default 5 recommendations.")

num\_recs = 5

recommendations = manager.recommender.recommend\_popular\_books(num\_recs)

if recommendations:

print("\n--- Popular Book Recommendations ---")

for i, book in enumerate(recommendations):

print(f" {i+1}. '{book.title}' by {book.author} (Genre: {book.genre})")

else:

print("No popular books to recommend.")

elif choice == 3:

num\_recs\_str = get\_validated\_input("How many recommendations? (default 5): ", lambda x: x if x else "5")

try:

num\_recs = int(num\_recs\_str)

if num\_recs <= 0:

print("Number of recommendations must be positive. Using default 5.")

num\_recs = 5

except ValueError:

print("Invalid number. Using default 5 recommendations.")

num\_recs = 5

recommendations = manager.recommender.recommend\_new\_arrivals(num\_recs)

if recommendations:

print("\n--- New Arrivals Recommendations ---")

for i, book in enumerate(recommendations):

print(f" {i+1}. '{book.title}' by {book.author} (Genre: {book.genre}, Pub Year: {book.publication\_year})")

else:

print("No new arrivals to recommend.")

elif choice == 4:

break

else:

print("Invalid choice. Please try again.")

def main():

"""Main function to run the Library Management System."""

print("Welcome to the Library Management System!")

# Initialize DataHandler and LibraryManager

data\_handler = DataHandler()

manager = LibraryManager(data\_handler)

while True:

display\_menu()

try:

choice = get\_validated\_input("Enter your choice: ", validate\_positive\_int, "Please enter a number (1-6).")

if choice == 1:

book\_management\_menu(manager)

elif choice == 2:

user\_management\_menu(manager)

elif choice == 3:

loan\_management\_menu(manager)

elif choice == 4:

reports\_statistics\_menu(manager)

elif choice == 5:

recommendations\_menu(manager)

elif choice == 6:

print("Exiting Library Management System. Goodbye!")

sys.exit(0) # Exit the application

else:

print("Invalid choice. Please enter a number between 1 and 6.")

except KeyboardInterrupt:

print("\nOperation cancelled by user. Exiting.")

sys.exit(0) # Graceful exit on Ctrl+C

except Exception as e:

# Catch any unhandled exceptions in the main loop

logging.critical(f"Unhandled error in main loop: {e}", exc\_info=True)

print("An unrecoverable error occurred. Please check the logs and restart the application.")

sys.exit(1) # Exit with an error code

if \_\_name\_\_ == "\_\_main\_\_":

main()

**5. Unit Tests**

Unit tests are provided in the tests/test\_library.py file. These tests cover critical functionalities to ensure the core logic works as expected.

**How to Run Tests:**

To run the tests, navigate to your library\_system directory in the terminal and execute:

python -m unittest discover tests

**Covered Test Cases:**

* Adding and updating books.
* Removing books (including edge cases like removing a book with checked-out copies).
* Registering users.
* Checking out and returning books, including availability checks and loan tracking.
* Calculating overdue fines for various scenarios.
* Basic verification of recommendation engine outputs.
* Ensuring data is correctly persisted to and loaded from JSON files.
* Testing the utility functions for input validation.

## Data/books.json

{  
 "978-0743273565": {  
 "isbn": "978-0743273565",  
 "title": "The Great Gatsby",  
 "author": "F. Scott Fitzgerald",  
 "publication\_year": 1925,  
 "genre": "Classic",  
 "description": "A classic American novel.",  
 "quantity": 3,  
 "available\_copies": 2  
 },  
 "978-0321765723": {  
 "isbn": "978-0321765723",  
 "title": "The Lord of the Rings",  
 "author": "J.R.R. Tolkien",  
 "publication\_year": 1954,  
 "genre": "Fantasy",  
 "description": "No description available.",  
 "quantity": 5,  
 "available\_copies": 5  
 },  
 "978-0451524935": {  
 "isbn": "978-0451524935",  
 "title": "1984",  
 "author": "George Orwell",  
 "publication\_year": 1949,  
 "genre": "Dystopian",  
 "description": "A dystopian social science fiction novel.",  
 "quantity": 2,  
 "available\_copies": 1  
 }  
}

## Data/loans.json

{  
 "L001": {  
 "loan\_id": "L001",  
 "book\_isbn": "978-0743273565",  
 "user\_id": "U001",  
 "checkout\_date": "2024-07-20",  
 "due\_date": "2024-08-03",  
 "return\_date": "2024-07-25"  
 },  
 "L002": {  
 "loan\_id": "L002",  
 "book\_isbn": "978-0321765723",  
 "user\_id": "U002",  
 "checkout\_date": "2024-07-22",  
 "due\_date": "2024-08-05",  
 "return\_date": null  
 },  
 "L003": {  
 "loan\_id": "L003",  
 "book\_isbn": "978-0451524935",  
 "user\_id": "U001",  
 "checkout\_date": "2024-07-28",  
 "due\_date": "2024-08-11",  
 "return\_date": null  
 }  
}

## Data/users.json

{  
 "U001": {  
 "user\_id": "U001",  
 "name": "Alice Smith",  
 "contact\_info": "alice@example.com",  
 "reading\_history": [  
 "978-0743273565",  
 "978-0321765723"  
 ],  
 "current\_loans": [  
 "L003"  
 ]  
 },  
 "U002": {  
 "user\_id": "U002",  
 "name": "Bob Johnson",  
 "contact\_info": "bob@example.com",  
 "reading\_history": [],  
 "current\_loans": []  
 }  
}

## test\_data\_handler

# tests/test\_data\_handler.py  
import unittest  
import os  
import shutil  
import json  
import datetime  
from data\_handler import DataHandler  
from models import Book, User, Loan  
  
class TestDataHandler(unittest.TestCase):  
 TEST\_DATA\_DIR = "test\_data\_handler"  
  
 def setUp(self):  
 """Set up a temporary directory for test data."""  
 if os.path.exists(self.TEST\_DATA\_DIR):  
 shutil.rmtree(self.TEST\_DATA\_DIR)  
 os.makedirs(self.TEST\_DATA\_DIR)  
 self.handler = DataHandler(self.TEST\_DATA\_DIR)  
  
 def tearDown(self):  
 """Clean up the temporary directory after each test."""  
 if os.path.exists(self.TEST\_DATA\_DIR):  
 shutil.rmtree(self.TEST\_DATA\_DIR)  
  
 def \_create\_dummy\_data(self):  
 """Helper to create some dummy data files."""  
 books\_data = {  
 "978-1": Book("978-1", "Book One", "Author A", 2000, "Fiction").to\_dict(),  
 "978-2": Book("978-2", "Book Two", "Author B", 2005, "Non-Fiction", quantity=2).to\_dict()  
 }  
 users\_data = {  
 "U001": User("U001", "User One").to\_dict(),  
 "U002": User("U002", "User Two", "user2@example.com").to\_dict()  
 }  
 today = datetime.date.today().strftime('%Y-%m-%d')  
 due = (datetime.date.today() + datetime.timedelta(days=14)).strftime('%Y-%m-%d')  
 loans\_data = {  
 "L001": Loan("L001", "978-1", "U001", today, due).to\_dict()  
 }  
  
 with open(self.handler.books\_file, 'w', encoding='utf-8') as f:  
 json.dump(books\_data, f, indent=4)  
 with open(self.handler.users\_file, 'w', encoding='utf-8') as f:  
 json.dump(users\_data, f, indent=4)  
 with open(self.handler.loans\_file, 'w', encoding='utf-8') as f:  
 json.dump(loans\_data, f, indent=4)  
  
 def test\_load\_empty\_data(self):  
 """Test loading data when files do not exist."""  
 books = self.handler.load\_books()  
 users = self.handler.load\_users()  
 loans = self.handler.load\_loans()  
 self.assertEqual(books, {})  
 self.assertEqual(users, {})  
 self.assertEqual(loans, {})  
  
 def test\_save\_and\_load\_books(self):  
 """Test saving and then loading book data."""  
 books\_to\_save = {  
 "978-3": Book("978-3", "New Book", "New Author", 2023, "Thriller"),  
 "978-4": Book("978-4", "Another New Book", "Another Author", 2024, "Mystery", quantity=3)  
 }  
 self.assertTrue(self.handler.save\_books(books\_to\_save))  
 loaded\_books = self.handler.load\_books()  
 self.assertEqual(len(loaded\_books), 2)  
 self.assertIn("978-3", loaded\_books)  
 self.assertEqual(loaded\_books["978-3"].title, "New Book")  
 self.assertEqual(loaded\_books["978-4"].quantity, 3)  
 self.assertEqual(loaded\_books["978-4"].available\_copies, 3) # Check available copies also  
  
 def test\_save\_and\_load\_users(self):  
 """Test saving and then loading user data."""  
 users\_to\_save = {  
 "U003": User("U003", "User Three"),  
 "U004": User("U004", "User Four", "u4@test.com")  
 }  
 self.assertTrue(self.handler.save\_users(users\_to\_save))  
 loaded\_users = self.handler.load\_users()  
 self.assertEqual(len(loaded\_users), 2)  
 self.assertIn("U003", loaded\_users)  
 self.assertEqual(loaded\_users["U004"].contact\_info, "u4@test.com")  
  
 def test\_save\_and\_load\_loans(self):  
 """Test saving and then loading loan data."""  
 today = datetime.date.today().strftime('%Y-%m-%d')  
 due = (datetime.date.today() + datetime.timedelta(days=14)).strftime('%Y-%m-%d')  
 return\_date = (datetime.date.today() + datetime.timedelta(days=7)).strftime('%Y-%m-%d')  
 loans\_to\_save = {  
 "L002": Loan("L002", "978-1", "U001", today, due),  
 "L003": Loan("L003", "978-2", "U002", today, due, return\_date)  
 }  
 self.assertTrue(self.handler.save\_loans(loans\_to\_save))  
 loaded\_loans = self.handler.load\_loans()  
 self.assertEqual(len(loaded\_loans), 2)  
 self.assertIn("L002", loaded\_loans)  
 self.assertEqual(loaded\_loans["L003"].return\_date, return\_date)  
  
 def test\_load\_existing\_data(self):  
 """Test loading data from pre-existing files."""  
 self.\_create\_dummy\_data()  
 books = self.handler.load\_books()  
 users = self.handler.load\_users()  
 loans = self.handler.load\_loans()  
  
 self.assertEqual(len(books), 2)  
 self.assertEqual(books["978-1"].title, "Book One")  
 self.assertEqual(books["978-2"].quantity, 2)  
 self.assertEqual(books["978-2"].available\_copies, 2)  
  
 self.assertEqual(len(users), 2)  
 self.assertEqual(users["U001"].name, "User One")  
  
 self.assertEqual(len(loans), 1)  
 self.assertEqual(loans["L001"].book\_isbn, "978-1")  
  
 def test\_load\_malformed\_json(self):  
 """Test handling of malformed JSON files."""  
 malformed\_json = "{"  
 with open(self.handler.books\_file, 'w', encoding='utf-8') as f:  
 f.write(malformed\_json)  
   
 # Test that it logs an error and returns empty dict  
 with self.assertLogs('root', level='ERROR') as cm:  
 books = self.handler.load\_books()  
 self.assertEqual(books, {})  
 self.assertIn("Error decoding JSON", cm.output[0])  
  
 def test\_load\_malformed\_object\_data(self):  
 """Test handling of malformed object data within JSON."""  
 books\_data\_malformed\_entry = {  
 "978-1": {"isbn": "978-1", "title": "Good Book", "author": "Author X", "publication\_year": 2000, "genre": "Fiction", "quantity": 1},  
 "978-malformed": {"isbn": "978-malformed", "title": "Bad Book", "author": "Author Y", "publication\_year": "invalid\_year", "genre": "Non-Fiction", "quantity": 1}  
 }  
 with open(self.handler.books\_file, 'w', encoding='utf-8') as f:  
 json.dump(books\_data\_malformed\_entry, f, indent=4)  
  
 with self.assertLogs('root', level='ERROR') as cm:  
 books = self.handler.load\_books()  
 self.assertEqual(len(books), 1) # Only the good book should be loaded  
 self.assertIn("978-1", books)  
 self.assertNotIn("978-malformed", books)  
 self.assertIn("Error loading Book from", cm.output[0])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

## test\_library\_manager

# tests/test\_library\_manager.py  
import unittest  
import os  
import shutil  
import datetime  
from unittest.mock import MagicMock, patch  
  
# Import classes and functions from the modules  
from models import Book, User, Loan  
from data\_handler import DataHandler  
from library\_manager import LibraryManager  
from recommender import RecommendationEngine # Needed for manager initialization  
from utils import generate\_unique\_id # Needed for loan ID generation in tests  
  
class TestLibraryManager(unittest.TestCase):  
 TEST\_DATA\_DIR = "test\_library\_manager\_data"  
  
 def setUp(self):  
 """Set up a clean test environment before each test."""  
 if os.path.exists(self.TEST\_DATA\_DIR):  
 shutil.rmtree(self.TEST\_DATA\_DIR)  
 os.makedirs(self.TEST\_DATA\_DIR)  
  
 # Mock DataHandler to prevent actual file I/O during most manager tests  
 # We will test DataHandler's file I/O separately in test\_data\_handler.py  
 self.mock\_data\_handler = MagicMock(spec=DataHandler)  
 self.mock\_data\_handler.load\_books.return\_value = {}  
 self.mock\_data\_handler.load\_users.return\_value = {}  
 self.mock\_data\_handler.load\_loans.return\_value = {}  
 self.mock\_data\_handler.save\_books.return\_value = True  
 self.mock\_data\_handler.save\_users.return\_value = True  
 self.mock\_data\_handler.save\_loans.return\_value = True  
  
 self.manager = LibraryManager(self.mock\_data\_handler)  
  
 # Add some initial data for testing, directly to manager's internal dicts  
 # This bypasses the mocked DataHandler's load, but manager's methods will use save.  
 self.book1 = Book("978-0321765723", "The Lord of the Rings", "J.R.R. Tolkien", 1954, "Fantasy", quantity=5)  
 self.book2 = Book("978-0743273565", "The Great Gatsby", "F. Scott Fitzgerald", 1925, "Classic", quantity=3)  
 self.book3 = Book("978-0451524935", "1984", "George Orwell", 1949, "Dystopian", quantity=2)  
 self.book4 = Book("978-0061120084", "To Kill a Mockingbird", "Harper Lee", 1960, "Classic", quantity=4)  
  
 self.manager.books = {  
 self.book1.isbn: self.book1,  
 self.book2.isbn: self.book2,  
 self.book3.isbn: self.book3,  
 self.book4.isbn: self.book4,  
 }  
  
 self.user1 = User("U001", "Alice Smith", "alice@example.com")  
 self.user2 = User("U002", "Bob Johnson", "bob@example.com")  
  
 self.manager.users = {  
 self.user1.user\_id: self.user1,  
 self.user2.user\_id: self.user2,  
 }  
  
 self.manager.loans = {} # Start with no loans for most tests  
  
 def tearDown(self):  
 """Clean up the test environment after each test."""  
 if os.path.exists(self.TEST\_DATA\_DIR):  
 shutil.rmtree(self.TEST\_DATA\_DIR)  
  
 def test\_add\_book(self):  
 """Test adding a new book and increasing quantity."""  
 # Test adding a new book  
 self.assertTrue(self.manager.add\_book("978-1234567890", "New Book", "New Author", 2023, "Fiction"))  
 self.assertIn("978-1234567890", self.manager.books)  
 self.assertEqual(self.manager.books["978-1234567890"].title, "New Book")  
 self.assertEqual(self.manager.books["978-1234567890"].quantity, 1)  
 self.assertEqual(self.manager.books["978-1234567890"].available\_copies, 1)  
 self.mock\_data\_handler.save\_books.assert\_called\_once() # Verify save was called  
  
 # Test increasing quantity of existing book  
 self.manager.add\_book("978-0321765723", "The Lord of the Rings", "J.R.R. Tolkien", 1954, "Fantasy", quantity=2)  
 self.assertEqual(self.manager.books["978-0321765723"].quantity, 7) # 5 + 2 = 7  
 self.assertEqual(self.manager.books["978-0321765723"].available\_copies, 7)  
 self.assertEqual(self.mock\_data\_handler.save\_books.call\_count, 2) # Called again  
  
 # Test invalid book data  
 self.assertFalse(self.manager.add\_book("978-invalid", "", "Author", 2000, "Genre"))  
 self.assertFalse(self.manager.add\_book("978-invalid2", "Title", "Author", 2000, "Genre", quantity=-1))  
  
 def test\_remove\_book(self):  
 """Test removing a book."""  
 self.assertTrue(self.manager.remove\_book("978-0451524935"))  
 self.assertNotIn("978-0451524935", self.manager.books)  
 self.mock\_data\_handler.save\_books.assert\_called\_once()  
  
 self.assertFalse(self.manager.remove\_book("NON-EXISTENT-ISBN")) # Test removing non-existent book  
   
 # Test cannot remove if copies are checked out  
 self.manager.checkout\_book("U001", "978-0321765723") # This will call save\_books and save\_users/loans  
 # Reset mock call count for save\_books to test only remove\_book's call  
 self.mock\_data\_handler.save\_books.reset\_mock()   
  
 self.assertFalse(self.manager.remove\_book("978-0321765723"))  
 self.assertIn("978-0321765723", self.manager.books)  
 self.mock\_data\_handler.save\_books.assert\_not\_called() # Should not save if removal fails  
  
 def test\_register\_user(self):  
 """Test registering a new user."""  
 self.assertTrue(self.manager.register\_user("U003", "Charlie Brown"))  
 self.assertIn("U003", self.manager.users)  
 self.assertEqual(self.manager.users["U003"].name, "Charlie Brown")  
 self.mock\_data\_handler.save\_users.assert\_called\_once()  
  
 self.assertFalse(self.manager.register\_user("U001", "Alice Smith")) # Test registering existing user  
 self.assertEqual(self.mock\_data\_handler.save\_users.call\_count, 1) # Should not save again  
  
 self.assertFalse(self.manager.register\_user("", "Invalid User")) # Test invalid user data  
  
 def test\_checkout\_and\_return\_book(self):  
 """Test checking out and returning a book."""  
 # Test successful checkout  
 self.assertTrue(self.manager.checkout\_book("U001", "978-0321765723"))  
 book = self.manager.get\_book("978-0321765723")  
 user = self.manager.get\_user("U001")  
 self.assertEqual(book.available\_copies, 4) # 5 - 1 = 4  
 self.assertEqual(len(user.current\_loans), 1)  
 loan\_id = user.current\_loans[0]  
 self.assertIn(loan\_id, self.manager.loans)  
 self.assertIsNone(self.manager.loans[loan\_id].return\_date)  
 self.mock\_data\_handler.save\_books.assert\_called\_once()  
 self.mock\_data\_handler.save\_users.assert\_called\_once()  
 self.mock\_data\_handler.save\_loans.assert\_called\_once()  
 self.mock\_data\_handler.reset\_mock() # Reset mock for return test  
  
 # Test checking out unavailable book  
 self.book2.available\_copies = 0 # Manually make unavailable  
 self.assertFalse(self.manager.checkout\_book("U002", "978-0743273565"))  
 self.mock\_data\_handler.save\_books.assert\_not\_called() # Should not save if checkout fails  
  
 # Test returning book  
 self.assertTrue(self.manager.return\_book(loan\_id))  
 self.assertEqual(book.available\_copies, 5) # 4 + 1 = 5  
 self.assertNotIn(loan\_id, user.current\_loans)  
 self.assertIsNotNone(self.manager.loans[loan\_id].return\_date)  
 self.assertIn("978-0321765723", user.reading\_history) # Should be added to history  
 self.mock\_data\_handler.save\_books.assert\_called\_once()  
 self.mock\_data\_handler.save\_users.assert\_called\_once()  
 self.mock\_data\_handler.save\_loans.assert\_called\_once()  
  
 # Test returning already returned book  
 self.assertFalse(self.manager.return\_book(loan\_id))  
  
 # Test returning non-existent loan  
 self.assertFalse(self.manager.return\_book("NON-EXISTENT-LOAN"))  
  
 def test\_calculate\_overdue\_fine(self):  
 """Test overdue fine calculation."""  
 # Create a loan that is overdue  
 past\_due\_date = (datetime.date.today() - datetime.timedelta(days=5)).strftime('%Y-%m-%d')  
 checkout\_date = (datetime.date.today() - datetime.timedelta(days=19)).strftime('%Y-%m-%d') # 14 days loan + 5 days overdue  
 loan\_id = generate\_unique\_id(self.manager.loans.keys())  
 overdue\_loan = Loan(loan\_id, "978-0321765723", "U001", checkout\_date, past\_due\_date)  
 self.manager.loans[loan\_id] = overdue\_loan  
 self.manager.users["U001"].current\_loans.append(loan\_id)  
  
 # Simulate return today  
 overdue\_loan.return\_date = datetime.date.today().strftime('%Y-%m-%d')  
 fine = self.manager.calculate\_overdue\_fine(loan\_id)  
 self.assertAlmostEqual(fine, 5 \* self.manager.overdue\_fine\_per\_day) # 5 days overdue  
  
 # Test non-overdue loan  
 loan\_id\_not\_overdue = generate\_unique\_id(self.manager.loans.keys())  
 future\_due\_date = (datetime.date.today() + datetime.timedelta(days=5)).strftime('%Y-%m-%d')  
 not\_overdue\_loan = Loan(loan\_id\_not\_overdue, "978-0743273565", "U002", datetime.date.today().strftime('%Y-%m-%d'), future\_due\_date)  
 self.manager.loans[loan\_id\_not\_overdue] = not\_overdue\_loan  
 not\_overdue\_loan.return\_date = datetime.date.today().strftime('%Y-%m-%d') # Returned on time  
 fine = self.manager.calculate\_overdue\_fine(loan\_id\_not\_overdue)  
 self.assertEqual(fine, 0.0)  
  
 @patch('builtins.print') # Mock print to capture output  
 def test\_list\_all\_books(self, mock\_print):  
 """Test listing all books."""  
 self.manager.list\_all\_books()  
 self.assertGreater(mock\_print.call\_count, 0) # Should print something  
 self.assertIn("The Lord of the Rings", mock\_print.call\_args\_list[1][0][0])  
  
 self.manager.books = {} # Clear books  
 self.manager.list\_all\_books()  
 mock\_print.assert\_any\_call("No books in the catalog.")  
  
 @patch('builtins.print')  
 def test\_search\_books(self, mock\_print):  
 """Test searching for books."""  
 results = self.manager.search\_books("gatsby")  
 self.assertEqual(len(results), 1)  
 self.assertEqual(results[0].title, "The Great Gatsby")  
 self.assertIn("Search Results for 'gatsby'", mock\_print.call\_args\_list[0][0][0])  
  
 results = self.manager.search\_books("non-existent")  
 self.assertEqual(len(results), 0)  
 self.assertIn("No books found matching 'non-existent'.", mock\_print.call\_args\_list[2][0][0])  
  
 @patch('builtins.print')  
 def test\_get\_collection\_statistics(self, mock\_print):  
 """Test collection statistics."""  
 stats = self.manager.get\_collection\_statistics()  
 self.assertEqual(stats["total\_books"], 4)  
 self.assertEqual(stats["total\_copies"], 14)  
 self.assertEqual(stats["available\_copies"], 14) # No loans yet in this setup  
 self.assertEqual(stats["checked\_out\_copies"], 0)  
 self.assertIn("Total Unique Books: 4", mock\_print.call\_args\_list[1][0][0])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

## test\_model

# tests/test\_models.py  
import unittest  
import datetime  
from models import Book, User, Loan  
  
class TestModels(unittest.TestCase):  
 """Unit tests for the data models (Book, User, Loan)."""  
  
 def test\_book\_initialization(self):  
 """Test Book object initialization and default values."""  
 book = Book("1234567890", "Test Title", "Test Author", 2023, "Fiction")  
 self.assertEqual(book.isbn, "1234567890")  
 self.assertEqual(book.title, "Test Title")  
 self.assertEqual(book.author, "Test Author")  
 self.assertEqual(book.publication\_year, 2023)  
 self.assertEqual(book.genre, "Fiction")  
 self.assertEqual(book.description, "No description available.")  
 self.assertEqual(book.quantity, 1)  
 self.assertEqual(book.available\_copies, 1)  
  
 book\_with\_desc\_qty = Book("0987654321", "Another Book", "Another Author", 2020, "Non-Fiction", "A great read.", 5)  
 self.assertEqual(book\_with\_desc\_qty.description, "A great read.")  
 self.assertEqual(book\_with\_desc\_qty.quantity, 5)  
 self.assertEqual(book\_with\_desc\_qty.available\_copies, 5)  
  
 # Test ValueError for missing required fields  
 with self.assertRaises(ValueError):  
 Book("", "Title", "Author", 2000, "Genre")  
 with self.assertRaises(ValueError):  
 Book("isbn", "", "Author", 2000, "Genre")  
 with self.assertRaises(ValueError):  
 Book("isbn", "Title", "Author", 2000, "")  
   
 # Test ValueError for invalid quantity  
 with self.assertRaises(ValueError):  
 Book("123", "T", "A", 2000, "F", quantity=-1)  
 with self.assertRaises(ValueError):  
 Book("123", "T", "A", 2000, "F", quantity="abc")  
  
 def test\_book\_to\_from\_dict(self):  
 """Test Book serialization to dict and deserialization from dict."""  
 book\_original = Book("1111111111", "Dict Book", "Dict Author", 2021, "Sci-Fi", "A futuristic tale.", 2)  
 book\_original.available\_copies = 1 # Simulate one copy checked out  
  
 book\_dict = book\_original.to\_dict()  
 self.assertIsInstance(book\_dict, dict)  
 self.assertEqual(book\_dict["isbn"], "1111111111")  
 self.assertEqual(book\_dict["title"], "Dict Book")  
 self.assertEqual(book\_dict["available\_copies"], 1)  
  
 book\_recreated = Book.from\_dict(book\_dict)  
 self.assertIsInstance(book\_recreated, Book)  
 self.assertEqual(book\_recreated.isbn, book\_original.isbn)  
 self.assertEqual(book\_recreated.title, book\_original.title)  
 self.assertEqual(book\_recreated.author, book\_original.author)  
 self.assertEqual(book\_recreated.publication\_year, book\_original.publication\_year)  
 self.assertEqual(book\_recreated.genre, book\_original.genre)  
 self.assertEqual(book\_recreated.description, book\_original.description)  
 self.assertEqual(book\_recreated.quantity, book\_original.quantity)  
 self.assertEqual(book\_recreated.available\_copies, book\_original.available\_copies)  
  
 def test\_user\_initialization(self):  
 """Test User object initialization and default values."""  
 user = User("U001", "John Doe")  
 self.assertEqual(user.user\_id, "U001")  
 self.assertEqual(user.name, "John Doe")  
 self.assertEqual(user.contact\_info, "N/A")  
 self.assertEqual(user.reading\_history, [])  
 self.assertEqual(user.current\_loans, [])  
  
 user\_with\_contact = User("U002", "Jane Smith", "jane@example.com")  
 self.assertEqual(user\_with\_contact.contact\_info, "jane@example.com")  
  
 # Test ValueError for missing required fields  
 with self.assertRaises(ValueError):  
 User("", "Name")  
 with self.assertRaises(ValueError):  
 User("ID", "")  
  
 def test\_user\_to\_from\_dict(self):  
 """Test User serialization to dict and deserialization from dict."""  
 user\_original = User("U003", "Alice Brown", "alice@test.com")  
 user\_original.reading\_history = ["isbn1", "isbn2"]  
 user\_original.current\_loans = ["loan1"]  
  
 user\_dict = user\_original.to\_dict()  
 self.assertIsInstance(user\_dict, dict)  
 self.assertEqual(user\_dict["user\_id"], "U003")  
 self.assertEqual(user\_dict["reading\_history"], ["isbn1", "isbn2"])  
  
 user\_recreated = User.from\_dict(user\_dict)  
 self.assertIsInstance(user\_recreated, User)  
 self.assertEqual(user\_recreated.user\_id, user\_original.user\_id)  
 self.assertEqual(user\_recreated.name, user\_original.name)  
 self.assertEqual(user\_recreated.contact\_info, user\_original.contact\_info)  
 self.assertEqual(user\_recreated.reading\_history, user\_original.reading\_history)  
 self.assertEqual(user\_recreated.current\_loans, user\_original.current\_loans)  
  
 def test\_loan\_initialization(self):  
 """Test Loan object initialization and default values."""  
 today = datetime.date.today().strftime('%Y-%m-%d')  
 due = (datetime.date.today() + datetime.timedelta(days=14)).strftime('%Y-%m-%d')  
 loan = Loan("L001", "book123", "user456", today, due)  
 self.assertEqual(loan.loan\_id, "L001")  
 self.assertEqual(loan.book\_isbn, "book123")  
 self.assertEqual(loan.user\_id, "user456")  
 self.assertEqual(loan.checkout\_date, today)  
 self.assertEqual(loan.due\_date, due)  
 self.assertIsNone(loan.return\_date)  
  
 return\_date = (datetime.date.today() + datetime.timedelta(days=7)).strftime('%Y-%m-%d')  
 loan\_returned = Loan("L002", "book456", "user789", today, due, return\_date)  
 self.assertEqual(loan\_returned.return\_date, return\_date)  
  
 # Test ValueError for missing required fields  
 with self.assertRaises(ValueError):  
 Loan("", "isbn", "user", today, due)  
 with self.assertRaises(ValueError):  
 Loan("id", "", "user", today, due)  
  
 def test\_loan\_to\_from\_dict(self):  
 """Test Loan serialization to dict and deserialization from dict."""  
 today = datetime.date.today().strftime('%Y-%m-%d')  
 due = (datetime.date.today() + datetime.timedelta(days=14)).strftime('%Y-%m-%d')  
 return\_date = (datetime.date.today() + datetime.timedelta(days=10)).strftime('%Y-%m-%d')  
 loan\_original = Loan("L003", "book789", "user101", today, due, return\_date)  
  
 loan\_dict = loan\_original.to\_dict()  
 self.assertIsInstance(loan\_dict, dict)  
 self.assertEqual(loan\_dict["loan\_id"], "L003")  
 self.assertEqual(loan\_dict["return\_date"], return\_date)  
  
 loan\_recreated = Loan.from\_dict(loan\_dict)  
 self.assertIsInstance(loan\_recreated, Loan)  
 self.assertEqual(loan\_recreated.loan\_id, loan\_original.loan\_id)  
 self.assertEqual(loan\_recreated.book\_isbn, loan\_original.book\_isbn)  
 self.assertEqual(loan\_recreated.user\_id, loan\_original.user\_id)  
 self.assertEqual(loan\_recreated.checkout\_date, loan\_original.checkout\_date)  
 self.assertEqual(loan\_recreated.due\_date, loan\_original.due\_date)  
 self.assertEqual(loan\_recreated.return\_date, loan\_original.return\_date)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

## test\_recommender

# tests/test\_recommender.py  
import unittest  
from unittest.mock import MagicMock  
from recommender import RecommendationEngine  
from models import Book, User, Loan # Import models for creating mock objects  
  
class TestRecommendationEngine(unittest.TestCase):  
 def setUp(self):  
 """Set up a mock LibraryManager with dummy data for testing."""  
 self.mock\_manager = MagicMock()  
  
 # Create some dummy books  
 self.book1 = Book("978-1", "Fantasy Quest", "A. Author", 2000, "Fantasy", quantity=5, available\_copies=5)  
 self.book2 = Book("978-2", "Sci-Fi Odyssey", "B. Writer", 2010, "Science Fiction", quantity=3, available\_copies=3)  
 self.book3 = Book("978-3", "Historical Drama", "C. Narrator", 1995, "History", quantity=2, available\_copies=2)  
 self.book4 = Book("978-4", "Another Fantasy", "D. Lorem", 2002, "Fantasy", quantity=1, available\_copies=0) # Unavailable  
 self.book5 = Book("978-5", "New Thriller", "E. Ipsum", 2023, "Thriller", quantity=4, available\_copies=4)  
 self.book6 = Book("978-6", "Classic Lit", "F. Dolor", 1850, "Classic", quantity=2, available\_copies=2)  
  
 self.mock\_manager.books = {  
 self.book1.isbn: self.book1,  
 self.book2.isbn: self.book2,  
 self.book3.isbn: self.book3,  
 self.book4.isbn: self.book4,  
 self.book5.isbn: self.book5,  
 self.book6.isbn: self.book6,  
 }  
 self.mock\_manager.get\_book.side\_effect = lambda isbn: self.mock\_manager.books.get(isbn)  
  
 # Create some dummy users  
 self.user1 = User("U001", "Alice")  
 self.user2 = User("U002", "Bob")  
 self.mock\_manager.users = {  
 self.user1.user\_id: self.user1,  
 self.user2.user\_id: self.user2,  
 }  
 self.mock\_manager.get\_user.side\_effect = lambda user\_id: self.mock\_manager.users.get(user\_id)  
  
 # Create some dummy loans to simulate popularity and reading history  
 self.loan1 = Loan("L001", self.book1.isbn, self.user1.user\_id, "2023-01-01", "2023-01-15", "2023-01-10") # Book1 read by U001  
 self.loan2 = Loan("L002", self.book2.isbn, self.user1.user\_id, "2023-02-01", "2023-02-15", "2023-02-10") # Book2 read by U001  
 self.loan3 = Loan("L003", self.book1.isbn, self.user2.user\_id, "2023-03-01", "2023-03-15", "2023-03-10") # Book1 read by U002 (makes Book1 popular)  
 self.loan4 = Loan("L004", self.book3.isbn, self.user2.user\_id, "2023-04-01", "2023-04-15", "2023-04-10") # Book3 read by U002  
 self.loan5 = Loan("L005", self.book5.isbn, self.user1.user\_id, "2023-05-01", "2023-05-15", None) # Book5 currently loaned  
  
 self.mock\_manager.loans = {  
 self.loan1.loan\_id: self.loan1,  
 self.loan2.loan\_id: self.loan2,  
 self.loan3.loan\_id: self.loan3,  
 self.loan4.loan\_id: self.loan4,  
 self.loan5.loan\_id: self.loan5,  
 }  
  
 # Manually update reading history and current loans for users based on dummy loans  
 self.user1.reading\_history.extend([self.book1.isbn, self.book2.isbn])  
 self.user1.current\_loans.append(self.loan5.loan\_id)  
 self.user2.reading\_history.extend([self.book1.isbn, self.book3.isbn])  
  
 self.recommender = RecommendationEngine(self.mock\_manager)  
  
 def test\_recommend\_by\_reading\_history(self):  
 """Test recommendations based on user's reading history."""  
 # Alice (U001) read Fantasy and Sci-Fi. She also has a Thriller checked out.  
 # She hasn't read 'Another Fantasy' (unavailable), 'Historical Drama', 'Classic Lit'.  
 # Should recommend Fantasy or Sci-Fi first, then others.  
   
 # Ensure 'Another Fantasy' is available for testing purposes, then set unavailable for specific test  
 self.book4.available\_copies = 1 # Make it available for general genre recs  
 self.user1.reading\_history = [self.book1.isbn, self.book2.isbn] # Fantasy, Sci-Fi  
  
 recs\_for\_alice = self.recommender.recommend\_by\_reading\_history("U001", num\_recommendations=2)  
 self.assertEqual(len(recs\_for\_alice), 2)  
 # Recommended books should be available and not in Alice's current reading history  
 for rec\_book in recs\_for\_alice:  
 self.assertGreater(rec\_book.available\_copies, 0)  
 self.assertNotIn(rec\_book.isbn, self.user1.reading\_history)  
 # The genres should ideally be Fantasy or Sci-Fi, but due to randomness and limited unread books,  
 # it might include other available unread books.  
 self.assertIn(rec\_book.genre, ["Fantasy", "Science Fiction", "History", "Classic", "Thriller"])  
  
 # Test fallback to popular if no reading history  
 self.user2.reading\_history = [] # Clear Bob's history  
 recs\_for\_bob\_no\_history = self.recommender.recommend\_by\_reading\_history("U002", num\_recommendations=1)  
 self.assertEqual(len(recs\_for\_bob\_no\_history), 1)  
 # Book1 is most popular (2 checkouts)  
 self.assertEqual(recs\_for\_bob\_no\_history[0].isbn, self.book1.isbn)  
   
 # Test when user does not exist  
 recs\_non\_existent\_user = self.recommender.recommend\_by\_reading\_history("U999")  
 self.assertGreater(len(recs\_non\_existent\_user), 0) # Should fall back to popular books  
  
 def test\_recommend\_popular\_books(self):  
 """Test recommendations for most popular books."""  
 # Book1 has 2 checkouts (L001, L003)  
 # Book2 has 1 checkout (L002)  
 # Book3 has 1 checkout (L004)  
 # Book5 has 1 checkout (L005) - but it's currently loaned out (available\_copies=3, but 1 loan)  
 # Book4 is unavailable (available\_copies=0)  
   
 # Ensure book5 is treated as unavailable for popular recs if it has active loans  
 self.book5.available\_copies = 3 # Original quantity 4, 1 loan, so 3 available.  
 # Let's make Book5 unavailable for this specific test  
 self.book5.available\_copies = 0 # Simulate it being out of stock for popular recs  
  
 popular\_recs = self.recommender.recommend\_popular\_books(num\_recommendations=3)  
 self.assertEqual(len(popular\_recs), 3)  
 # Book1 should be first  
 self.assertEqual(popular\_recs[0].isbn, self.book1.isbn)  
 # The next two could be Book2 or Book3 or Book6 (all 1 checkout, available). Order depends on dict iteration/sort stability.  
 # Let's just check they are from the expected set and are available.  
 self.assertIn(popular\_recs[1].isbn, [self.book2.isbn, self.book3.isbn, self.book6.isbn])  
 self.assertIn(popular\_recs[2].isbn, [self.book2.isbn, self.book3.isbn, self.book6.isbn])  
 self.assertNotEqual(popular\_recs[1].isbn, popular\_recs[2].isbn) # Ensure they are different books  
  
 # Ensure unavailable books are not recommended  
 for rec\_book in popular\_recs:  
 self.assertGreater(rec\_book.available\_copies, 0)  
   
 # Test with no loans  
 self.mock\_manager.loans = {}  
 no\_popular\_recs = self.recommender.recommend\_popular\_books(num\_recommendations=1)  
 self.assertEqual(len(no\_popular\_recs), 0)  
  
 def test\_recommend\_new\_arrivals(self):  
 """Test recommendations for new arrivals."""  
 # Books by publication year: book5 (2023), book2 (2010), book1 (2000), book3 (1995), book6 (1850)  
 # book4 is unavailable, so it shouldn't be recommended even if new.  
 self.book4.publication\_year = 2024 # Make it the newest but unavailable  
 self.book4.available\_copies = 0 # Ensure it's unavailable  
  
 new\_recs = self.recommender.recommend\_new\_arrivals(num\_recommendations=3)  
 self.assertEqual(len(new\_recs), 3)  
 self.assertEqual(new\_recs[0].isbn, self.book5.isbn) # Newest available  
 self.assertEqual(new\_recs[1].isbn, self.book2.isbn)  
 self.assertEqual(new\_recs[2].isbn, self.book1.isbn)  
  
 # Ensure unavailable books are not recommended  
 for rec\_book in new\_recs:  
 self.assertGreater(rec\_book.available\_copies, 0)  
   
 # Test with no books  
 self.mock\_manager.books = {}  
 no\_new\_recs = self.recommender.recommend\_new\_arrivals(num\_recommendations=1)  
 self.assertEqual(len(no\_new\_recs), 0)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

## test\_utils

# tests/test\_utils.py  
import unittest  
import datetime  
from unittest.mock import patch  
from utils import get\_validated\_input, validate\_positive\_int, validate\_non\_negative\_int, validate\_date, generate\_unique\_id  
  
class TestUtils(unittest.TestCase):  
 """Unit tests for utility functions."""  
  
 @patch('builtins.input', side\_effect=['invalid', '10'])  
 def test\_get\_validated\_input\_with\_validation(self, mock\_input):  
 """Test get\_validated\_input with a validation function."""  
 result = get\_validated\_input("Enter a positive number: ", int, "Please enter a valid integer.")  
 self.assertEqual(result, 10)  
 self.assertEqual(mock\_input.call\_count, 2)  
  
 @patch('builtins.input', side\_effect=['hello world'])  
 def test\_get\_validated\_input\_no\_validation(self, mock\_input):  
 """Test get\_validated\_input without a validation function."""  
 result = get\_validated\_input("Enter anything: ")  
 self.assertEqual(result, "hello world")  
 self.assertEqual(mock\_input.call\_count, 1)  
  
 def test\_validate\_positive\_int(self):  
 """Test validate\_positive\_int function."""  
 self.assertEqual(validate\_positive\_int("1"), 1)  
 self.assertEqual(validate\_positive\_int("100"), 100)  
 with self.assertRaisesRegex(ValueError, "Input must be a positive whole number."):  
 validate\_positive\_int("0")  
 with self.assertRaisesRegex(ValueError, "Input must be a positive whole number."):  
 validate\_positive\_int("-5")  
 with self.assertRaisesRegex(ValueError, "Input must be a positive whole number."):  
 validate\_positive\_int("abc")  
 with self.assertRaisesRegex(ValueError, "Input must be a positive whole number."):  
 validate\_positive\_int("1.5")  
  
 def test\_validate\_non\_negative\_int(self):  
 """Test validate\_non\_negative\_int function."""  
 self.assertEqual(validate\_non\_negative\_int("0"), 0)  
 self.assertEqual(validate\_non\_negative\_int("5"), 5)  
 with self.assertRaisesRegex(ValueError, "Input must be a non-negative whole number."):  
 validate\_non\_negative\_int("-1")  
 with self.assertRaisesRegex(ValueError, "Input must be a non-negative whole number."):  
 validate\_non\_negative\_int("xyz")  
 with self.assertRaisesRegex(ValueError, "Input must be a non-negative whole number."):  
 validate\_non\_negative\_int("2.5")  
  
 def test\_validate\_date(self):  
 """Test validate\_date function."""  
 self.assertEqual(validate\_date("2023-01-15"), "2023-01-15")  
 self.assertEqual(validate\_date("1999-12-31"), "1999-12-31")  
 with self.assertRaisesRegex(ValueError, "Date must be in YYYY-MM-DD format."):  
 validate\_date("2023/01/15")  
 with self.assertRaisesRegex(ValueError, "Date must be in YYYY-MM-DD format."):  
 validate\_date("15-01-2023")  
 with self.assertRaisesRegex(ValueError, "Date must be in YYYY-MM-DD format."):  
 validate\_date("invalid-date")  
 with self.assertRaisesRegex(ValueError, "Date must be in YYYY-MM-DD format."):  
 validate\_date("2023-02-30") # Invalid day for February  
  
 def test\_generate\_unique\_id(self):  
 """Test generate\_unique\_id function."""  
 self.assertEqual(generate\_unique\_id([]), "001")  
 self.assertEqual(generate\_unique\_id(["001"]), "002")  
 self.assertEqual(generate\_unique\_id(["001", "003"]), "004")  
 self.assertEqual(generate\_unique\_id(["009", "005", "007"]), "010")  
 self.assertEqual(generate\_unique\_id(["abc", "002"]), "003") # Handles non-numeric existing IDs  
 self.assertEqual(generate\_unique\_id(["099"]), "100")  
 self.assertEqual(generate\_unique\_id(["999"]), "1000") # Handles overflow beyond 3 digits  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

**6. Conclusion**

The Library Book Cataloger and Recommender CLI application successfully meets all the specified project requirements. It demonstrates a robust understanding of object-oriented programming principles, effective data persistence using standard library modules, comprehensive error handling, and modular code organization.

The application provides essential functionalities for managing a library's collection, tracking user interactions and loans, and offering basic book recommendations. Its design allows for future scalability and the integration of more advanced features, should the project evolve. The included unit tests ensure the reliability of critical components, making it a stable foundation for a real-world library management system. This project serves as a strong example of building a functional and well-engineered command-line application in Python .