Chunk 1: Database Setup

```
def setup_database():
    try:
        conn = sqlite3.connect('Library_Database.db')
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS Library_Database (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                Author TEXT NOT NULL,
                Title TEXT NOT NULL,
                Genre TEXT,
                Date_Published TEXT,
                Pages INTEGER
            )
        ''')
        conn.commit()
        return conn, cursor
    except sqlite3.Error as e:
        eg.exceptionbox(msg=f"A database error occurred: {e}", title="Database Error")
        return None, None
```

[Why is it better to use this?]
- Ensures the database and table exist before other operations.
- Uses `IF NOT EXISTS` to avoid errors if the table already exists.
- Handles errors gracefully with exception boxes.

[How did I test?]
- Run `setup_database()`.
- Verify that `Library_Database.db` is created.
- Use a database viewer or `cursor.execute("PRAGMA table_info(Library_Database)")` to verify the table exists with the correct schema.

Chunk 2: Insert Book Function

```
def insert_book(cursor, author, title, genre, date_published, pages):
    try:
        cursor.execute('''
            INSERT INTO Library_Database (Author, Title, Genre, Date_Published, Pages)
```

```
            VALUES (?, ?, ?, ?, ?)
        ''', (author, title, genre, date_published, pages))
        return True
    except sqlite3.Error as e:
        return False, str(e)
```

[Why is it better to use this?]
- Uses parameterized queries (`?`) to prevent SQL injection and handle data safely.
- Returns success status and error message for handling.

[How did I test this?]
- Call with sample data.
- After execution, check the database directly or fetch all records to verify insertion.
- Test with invalid data to see error handling (e.g., `None` where not allowed).

Chunk 3: Get All Books Function

```
def get_all_books(cursor):
    try:
        cursor.execute("SELECT Author, Title, Genre, Date_Published, Pages FROM
Library_Database")
        return cursor.fetchall()
    except sqlite3.Error as e:
        return None, str(e)
```

[Why is it better to use this?]
- Fetches all records efficiently.
- Error handling ensures graceful failure.

[How did I test this?]
- Insert sample records.
- Call `get_all_books()` and verify returned list matches inserted data.
- Test with empty table to ensure it returns an empty list.

Chunk 4: Validate Date Function

```
def validate_date(day_str, month_str, year_str):
    if not (day_str.isdigit() and month_str.isdigit() and year_str.isdigit()):
        return False, "Day, Month, and Year must be numbers."
    try:
        day, month, year = int(day_str), int(month_str), int(year_str)
        if not (1 <= day <= 31 and 1 <= month <= 12 and 1000 <= year <= 9999):
            return False, "Please enter a valid date (DD/MM/YYYY)."
        return True, f"{day:02d}/{month:02d}/{year}"
    except ValueError:
        return False, "Invalid numbers entered for date."
```

[Why is it better to use this?]
- Checks numeric input before conversion.
- Validates date ranges.
- Returns a tuple `(bool, message)` for easy handling.

[How did I test this?]
- Run with valid inputs like `("15", "08", "2022")`.
- Run with invalid inputs like `("32", "13", "2022")` or `"abc"`.
- Confirm it behaves correctly.

Chunk 5: Validate Pages Function

```python
def validate_pages(pages_str):
    if pages_str == "" or pages_str is None:
        return True, None
    if pages_str.isdigit():
        return True, int(pages_str)
    else:
        return False, "Pages must be a number."
```

[Why is it better to use this?]
- Handles optional input gracefully.
- Ensures numeric value when provided.
- Simplifies validation.

[How did I test this?]
- Input `""` → should succeed with `None`.
- Input `"150"` → should succeed with `150`.
- Input `"abc"` → should fail.

Chunk 6: GUI Function for Adding a Book

```
def add_book_gui(cursor, conn):
    # Collects user data via EasyGUI
    # Validates inputs using above functions
    # Calls insert_book() if validation passes
    # Shows success or error messages
```

[Why is it better to use this?]
- Encapsulates user interaction.
- Validates inputs before database insertion.
- Provides user feedback.

[How did I test this?]
- Manually run and test with various inputs:
  - Valid data.
  - Canceling input.
  - Invalid dates or pages.
  - Missing required fields.
- Confirm database updates accordingly.

Chunk 7: Show Books Function

```
def show_books(cursor):
    rows = get_all_books(cursor)
    if rows is None:
        eg.exceptionbox(msg="Failed to retrieve books.", title="Database Error")
```

```
        return
    if len(rows) == 0:
        eg.msgbox("No books found in the database.", "Book List")
        return
    # Formats and displays the list
```

[Why is it better to use this?]
- Handles empty databases gracefully.
- Uses consistent formatting.
- Separates data retrieval from display logic.

[How did I test this?]
- Insert sample data and verify display.
- Clear database and verify "No books found" message.

<u>Chunk 8: Main Loop & Program Entry</u>

```
def main():
    conn, cursor = setup_database()
    if not conn:
        return
    # Notifies user about database status
    while True:
        choice = eg.buttonbox("What would you like to do?", "Library Menu", choices=["Add
Book", "Check Books", "Exit"])
        if choice == "Add Book":
            add_book_gui(cursor, conn)
        elif choice == "Check Books":
            show_books(cursor)
        elif choice is None or choice == "Exit":
            break
    conn.close()
    eg.msgbox("Goodbye!", "Exiting Program")
```

[Why is it better to use this?]
- Main control loop.
- Cleanly manages database connection lifecycle.

- User-friendly GUI navigation.

[How did I test this?]
- Run the full program.
- Test each menu option.
- Verify data persistence across operations.
- Confirm proper exit and cleanup.

[CODE-TESTING SUMMARY] -

(Summary of Why this modular approach is best):
- **Testability**: Each chunk can be tested independently, making debugging easier.
- **Maintainability:** Smaller functions are easier to read, modify, and extend.
- **Reusability**: Core functions (`insert_book`, `get_all_books`, validation) can be reused or replaced.
- **Error Handling:** Isolated error management improves robustness.

[FINAL COMMENT ON THE LIBRARY DATABASE PROGRAM]:

This program is designed with a clear focus on providing a smooth and intuitive user experience, making it accessible even for those who may not be highly technical. By utilizing simple graphical prompts through EasyGUI, it guides users step-by-step, which reduces the likelihood of errors and makes the process of adding and viewing books straightforward and enjoyable. The testing process (above) makes the program trustable and less errors. The program's logical flow ensures that users can easily navigate between adding new entries and checking the existing library, all within a clean, minimal interface that eliminates clutter and confusion. Moreover, the validation functions improve the usability by catching common mistakes - such as incorrect date formats or non-numeric pages - before they can cause issues, saving users time and frustration. What truly makes this program stand out is its thoughtful combination of robustness and friendliness. It handles errors gracefully, providing clear messages when something goes wrong, and ensures that data is stored securely in a local database, which maintains integrity and persistence across sessions. The modular structure of the code makes it easier to understand, modify, or expand in the future, which fosters a sense of confidence and control for the user. Overall, this program isn't just about functionality; it's about creating a seamless, human-centered experience where

users feel supported and empowered to manage their library efficiently, making it the best choice for those who value simplicity, reliability, and thoughtful design.

[SELF-REFLECTIONS & PROGRESSIONS]:

Embarking on the development of this library management program has been an insightful process into the practical application of programming concepts, user interface design, and database management. From the outset, my primary goal was to create a user-friendly, reliable, and scalable tool that simplifies the process of cataloging and retrieving books, all while maintaining clean code practices and thoughtful error handling. One of the core strengths of this project lies in the effective integration of modules - easygui for intuitive user interaction, sqlite3 for persistent data storage, and os for filesystem operations. This synergy allows a seamless experience: users can effortlessly add books, view their collection, and have confidence that their data is stored safely. The design of individual functions such as setup_database(), add_book(), and show_books() demonstrates a clear understanding of modular programming principles. Each function encapsulates a specific responsibility, making the code both readable and maintainable. For example, setup_database() not only creates the database schema but also handles potential errors gracefully, which informs the user of any issues without causing abrupt crashes. This attention to robustness reflects an appreciation for real-world application needs, where unforeseen errors are inevitable. In the add_book() function, I prioritized user input validation - making sure that essential fields like author and title are provided, and that date entries are realistic. Incorporating multiple input prompts and validation steps showcases my awareness of the importance of data integrity. Moreover, by allowing optional fields such as genre and pages, I aimed to balance completeness with flexibility, recognizing that users may not always have all information at hand. The show_books() function exemplifies how data retrieval can be transformed into a user-friendly presentation. By formatting the display with aligned columns, I sought to make the output easily scannable, which enhances the overall usability. Including exception handling here also demonstrates a proactive approach to managing potential database errors, safeguarding the program against crashes and confusing messages. Throughout this project, I have learned the significance of clear communication - both in code comments and in user prompts. The detailed explanations within the code serve as a bridge for future developers or users who may wish to modify or extend the program. This reflection on clarity and documentation highlights my commitment to creating accessible and sustainable

software. Looking ahead, I recognize opportunities for progression. Implementing features such as editing or deleting entries, searching by author or genre, and exporting the database to different formats are natural next steps. Enhancing the user interface with more sophisticated GUI elements or integrating data visualization could further elevate the program's functionality and appeal. In conclusion, this project has been a valuable exercise in translating conceptual requirements into a functional application. It has strengthened my technical skills in Python, database handling, and user interface design, while also reinforcing the importance of thoughtful planning, error handling, and user-centric development. I am proud of the progress made and motivated to continue refining my craft, aiming for excellence in both code quality and user experience.