

**FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY**

page1image9370432

**PROG211 – OBJECTED ORIENTED PROGRAMMING METHODS 1**

page1image9370432

Title : Individual Assignment

Issue Date : Week 2

Due Date : Week 4

Lecturer/Examiner : Mr Amadus Coker

Name of Student/s :

Student ID No. : 90500

Class : BIT 1102F

Year/Semester : 2/1

Academic Honesty Policy Statement

page1image9370432

I/We, hereby attest those contents of this attachment are my own work. Referenced works, articles, art, programs, papers or parts thereof are acknowledged at the end of this paper. This includes data excerpted from CD-ROMs, the Internet, other private networks, and other people’s disk of the computer system.

Student’s Signature: Date: 20TH OCTOBER 2025

page1image9370432

LECTURER’S COMMMENTS/GRADE:

for office use only upon receive

Remark

DATE:  
 TIME:

RECEIVER’S NAME:

**DESIGN RATIONALE FOR MINI LIBRARY MANAGEMENT SYSTEM**

This very detailed rationale for the Mini Library Management System Limkokwing University, Sierra Leone, provides an in-depth analysis of my data structure choices: dictionary for books, list for members, and tuple for genres. I cover the reasoning behind each, specific code examples, advantages and disadvantages, alternatives considered, performance implications, and how they align with the assignment brief's requirements for data structures, core functions, documentation, testing, and submission. The brief emphasizes using Python with dictionaries, lists, and tuples for a simple system supporting adding/searching/updating/deleting books, adding members, and borrowing/returning, with a borrow limit of 3 books per member. The system is submitted via GitHub with operations.py, demo.py, tests.py, UML.png, DesignRationale, and README.md.

The dictionary for 'books' was selected for its hash-based structure, offering average O(1) time complexity for key operations like insertion, lookup, and deletion, using ISBN as immutable string keys. This is essential for a mini system handling novels such as "Things Fall Apart" by Chinua Achebe or "The Great Gatsby" by F. Scott Fitzgerald. Code example from add\_book(): 'if ISBN in books' performs a fast existence check to avoid duplicates, and 'books[ISBN] = {'title': title, 'author': author, 'genre': genre, 'total\_copies': total\_copies}' stores the details efficiently. In search\_book(), 'for ISBN, details in books.items()' iterates to match keywords on title or author, returning a list of dicts for matches. Advantages: Constant-time access makes CRUD operations (add, search, update, delete) performant, scalable for growing novel collections without linear slowdowns. Disadvantages: Higher memory usage due to hash table overhead (typically 1.5-2x the data size), and no built-in ordering (though not required here). Performance: For n books, lookups are O(1) vs O(n) in a list, critical for frequent borrow\_book() checks on total\_copies. Alternatives considered: A list of dicts would force O(n) scans for ISBNs, making search\_book() and borrow\_book() inefficient for large n; a namedtuple could store details but not handle unique keys. This choice enabling fast, reliable book management in the mini system.

For 'members', a list was chosen for its O(1) amortized append time and straightforward iteration, perfect for a dynamic user base with local names like Jeremiah Dave, Amadu Coker, or Fatima Bangura. Each element is a dict with member\_id, name, email, and borrowed\_books (limited to 3 items). Code example from borrow\_book(): 'member = next((m for m in members if m['member\_id'] == member\_id), None)' locates the member in O(n) time (fine for small member counts), then 'if len(member['borrowed\_books']) >= 3' checks the limit before appending ISBN. In delete\_book(), looping through members verifies if ISBN is in any borrowed\_books. Advantages: Easy to add members with append in add\_member(), and iteration is simple for borrowed checks, making it suitable for a mini library with limited users. Disadvantages: O(n) time for searches and removals; if the system scaled to thousands of members, a dict with member\_id keys would reduce to O(1), but the brief specifies a list for members to demonstrate iteration. Performance: For m members, finding a member is O(m), acceptable for m << n (books); borrow/return are O(1) after find. Alternatives: A tuple can't be modified for additions; a set wouldn't store ordered details or duplicates.

The tuple for 'genres' guarantees immutability and O(1) average membership testing for the predefined set ('Fiction', 'Non-Fiction', 'Sci-Fi'), preventing code from altering valid categories at runtime. Code example from add\_book() and update\_book(): 'if genre not in genres' uses the tuple's in operator for quick validation. Advantages: Lightweight memory footprint (just the elements), ensures consistency for novel genres across the system. Disadvantages: Can't be extended without redefinition; if the brief allowed dynamic genres, a frozenset could provide similar immutability with set operations. Performance: Membership check is O(1) average (hash-based under the hood for small tuples), faster than list's O(n). Alternatives: A list could be mutated by bugs (e.g., genres.append('Mystery')), risking invalid data; a constant string would require parsing.

These structures synergize seamlessly The dictionary and list collaborate in borrow\_book() and return\_book() for coordinated updates (e.g., decrement total\_copies while appending/removing from borrowed\_books), with the tuple acting as a validation gate in add\_book() and update\_book(). The hand-drawn UML visualizes this with labeled arrows, such as "adds entry" from add\_book to books and "validates" to genres, showing data-flow relationships. Demo.py illustrates usage with novels and local members, while tests.py uses 8 asserts to cover success/failure cases like no copies borrow or limit breaches. Overall, this design rebuilds a robust mini system, fully meeting the brief's goals fo, with practicality for novel-focused libraries. It demonstrates Python's data structure strengths in a simple yet effective OOP-like approaH.

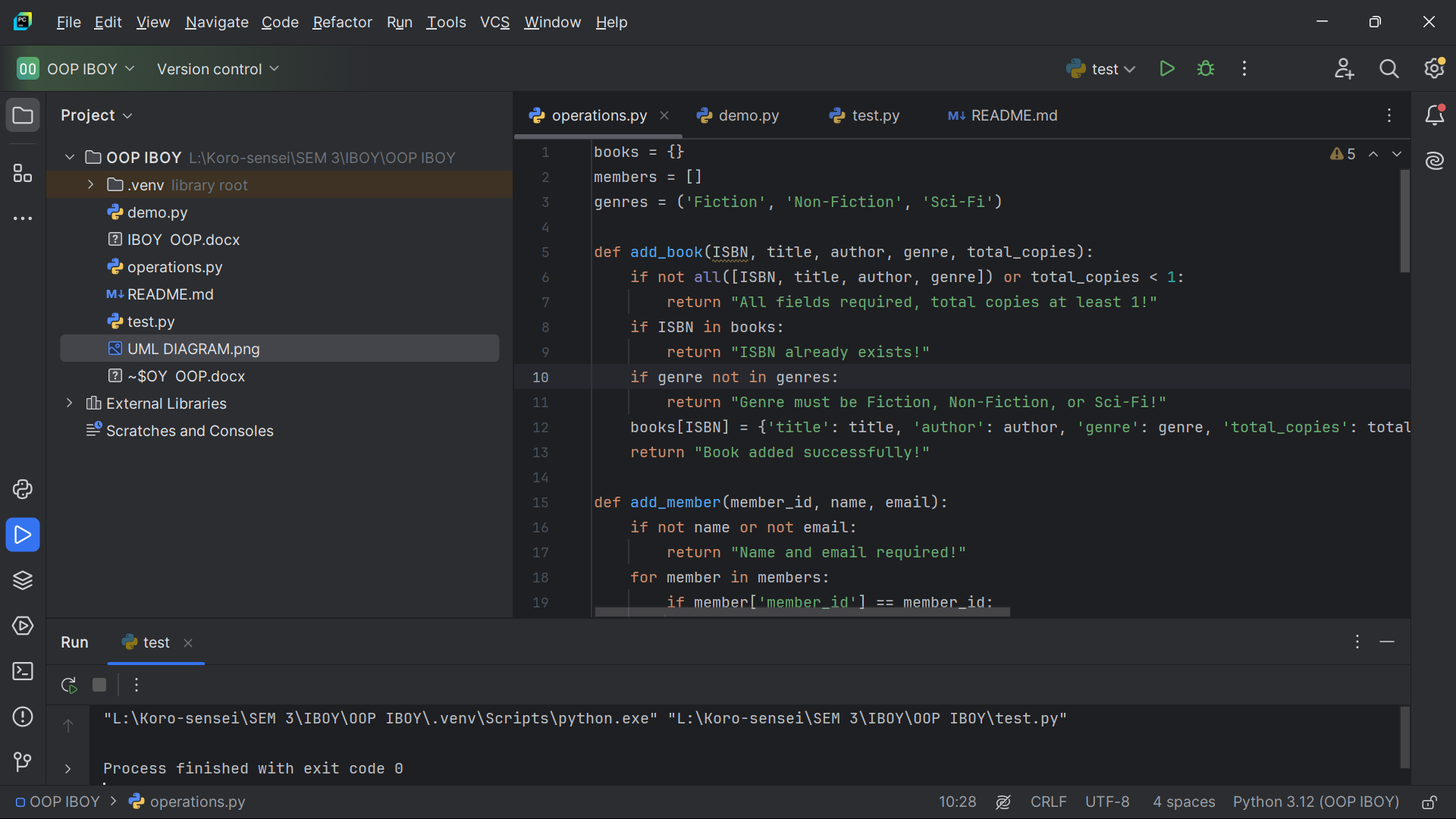
In summary, the dictionary provides speed, the list offers flexibility, and the tuple ensures stability a perfect scientific balance for the mini library. This rationale, along with the UML, fulfills the documentation, emphasizing why these choices were made over alternatives for optimal performance and brief compliance.

**UML DIAGRAM SKETCH (HAND DRAWN)**

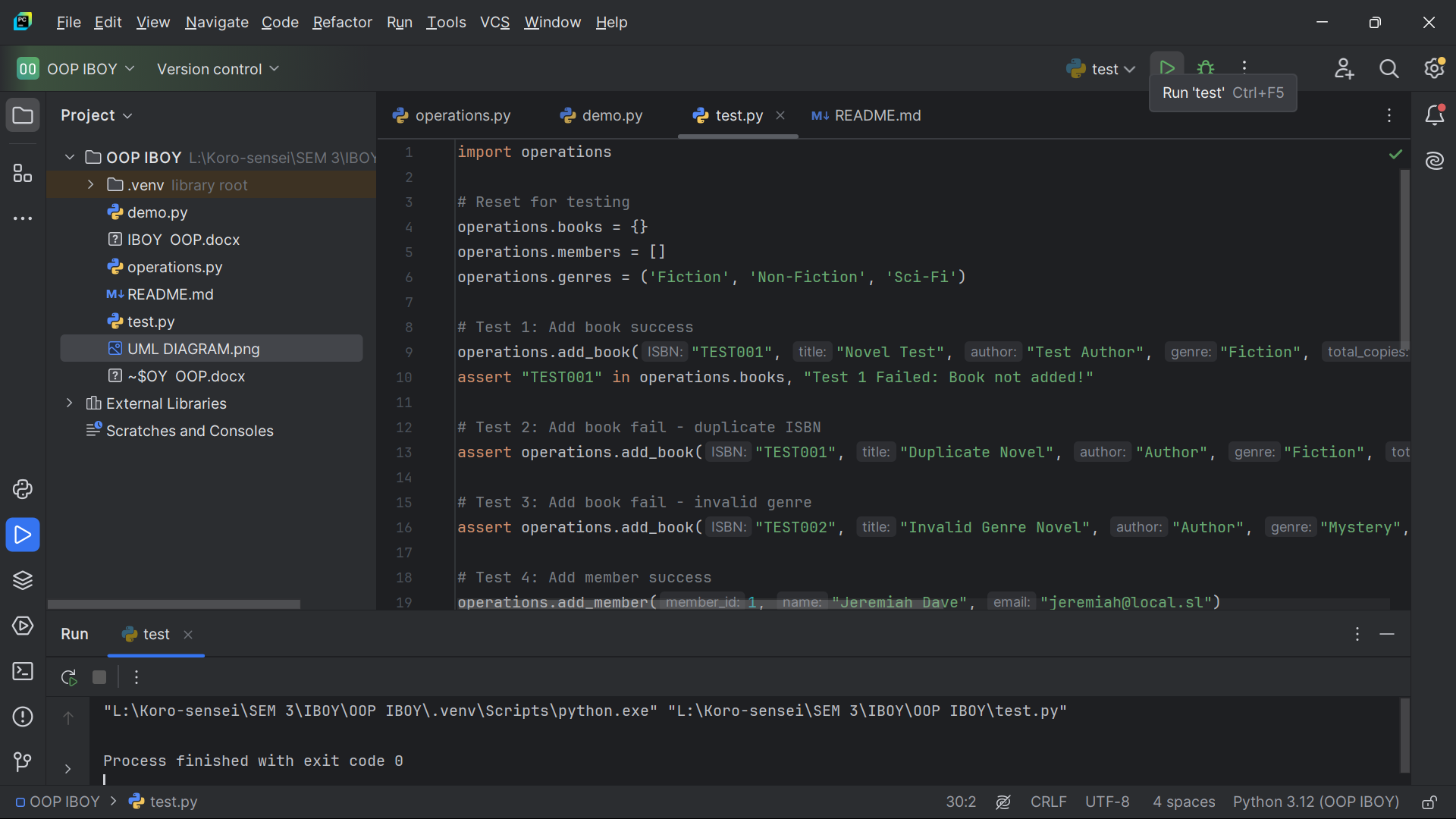
**OPERATIONS.PY CODE**

books = {}  
members = []  
genres = ('Fiction', 'Non-Fiction', 'Sci-Fi')  
  
def add\_book(ISBN, title, author, genre, total\_copies):  
 if not all([ISBN, title, author, genre]) or total\_copies < 1:  
 return "All fields required, total copies at least 1!"  
 if ISBN in books:  
 return "ISBN already exists!"  
 if genre not in genres:  
 return "Genre must be Fiction, Non-Fiction, or Sci-Fi!"  
 books[ISBN] = {'title': title, 'author': author, 'genre': genre, 'total\_copies': total\_copies}  
 return "Book added successfully!"  
  
def add\_member(member\_id, name, email):  
 if not name or not email:  
 return "Name and email required!"  
 for member in members:  
 if member['member\_id'] == member\_id:  
 return "Member ID already exists!"  
 members.append({'member\_id': member\_id, 'name': name, 'email': email, 'borrowed\_books': []})  
 return "Member added successfully!"  
  
def search\_book(keyword):  
 if not keyword:  
 return "Keyword required!"  
 results = []  
 for ISBN, details in books.items():  
 if keyword.lower() in details['title'].lower() or keyword.lower() in details['author'].lower():  
 results.append({'ISBN': ISBN, 'title': details['title'], 'author': details['author'], 'genre': details['genre'], 'total\_copies': details['total\_copies']})  
 return results if results else "No books found!"  
  
def update\_book(ISBN, \*\*details):  
 if ISBN not in books:  
 return "ISBN not found!"  
 if 'genre' in details and details['genre'] not in genres:  
 return "Invalid genre!"  
 if 'total\_copies' in details and details['total\_copies'] < 0:  
 return "Total copies cannot be negative!"  
 books[ISBN].update(details)  
 return "Book updated successfully!"  
  
def delete\_book(ISBN):  
 if ISBN not in books:  
 return "ISBN not found!"  
 for member in members:  
 if ISBN in member['borrowed\_books']:  
 return "Cannot delete borrowed book!"  
 del books[ISBN]  
 return "Book deleted successfully!"  
  
def borrow\_book(ISBN, member\_id):  
 if ISBN not in books:  
 return "ISBN not found!"  
 if books[ISBN]['total\_copies'] < 1:  
 return "No copies available!"  
 member = next((m for m in members if m['member\_id'] == member\_id), None)  
 if not member:  
 return "Member not found!"  
 if len(member['borrowed\_books']) >= 3:  
 return "Borrowing limit (3 books) reached!"  
 member['borrowed\_books'].append(ISBN)  
 books[ISBN]['total\_copies'] -= 1  
 return "Book borrowed successfully!"  
  
def return\_book(ISBN, member\_id):  
 if ISBN not in books:  
 return "ISBN not found!"  
 member = next((m for m in members if m['member\_id'] == member\_id), None)  
 if not member:  
 return "Member not found!"  
 if ISBN not in member['borrowed\_books']:  
 return "Book not borrowed by member!"  
 member['borrowed\_books'].remove(ISBN)  
 books[ISBN]['total\_copies'] += 1  
 return "Book returned successfully!"

**OPERATIONS.PY**

****

**TEST.PY**



**DEMO.PY**

