**Address Book Project**

Project Overview:

Build an address book application that allows users to store and manage contact information.

Features:

Add a Contact:

Allow users to add a new contact with details such as name, phone number, email, and address.

View All Contacts:

Display a list of all contacts in the address book.

Search for a Contact:

Implement a search functionality to find a contact by name.

Update Contact Information:

Allow users to update the details of an existing contact.

Delete a Contact:

Provide an option to delete a contact from the address book.

Save/Load Address Book to/from File:

Implement functionality to save the current state of the address book to a file (e.g., in JSON format) and load it back.

**Guidelines:**

Use Classes:

Create a Contact class to represent an individual contact with attributes like name, phone number, email, and address.

Use a Dictionary for the Address Book:

Use a dictionary to store contacts, where the key is a unique identifier (e.g., name) and the value is an instance of the Contact class.

**File Handling:**

Implement functions to save the address book to a file and load it back. You can use JSON for serialization and deserialization.

**User Interface:**

Create a simple text-based interface that allows users to interact with the address book (e.g., using a command-line interface).

**Function Decomposition:**

Decompose the functionalities into functions/methods. For example, you might have functions like add\_contact, view\_contacts, search\_contact, update\_contact, delete\_contact, save\_to\_file, and load\_from\_file.

**Example Structure:**

class Contact:

def \_\_init\_\_(self, name, phone, email, address):

# Initialize contact attributes

class AddressBook:

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

# Initialize an empty dictionary to store contacts

def add\_contact(self, contact):

# Add a new contact to the address book

def view\_contacts(self):

# Display all contacts

def search\_contact(self, name):

# Search for a contact by name

def update\_contact(self, name, new\_phone, new\_email, new\_address):

# Update contact information

def delete\_contact(self, name):

# Delete a contact

def save\_to\_file(self, filename):

# Save address book to a file

def load\_from\_file(self, filename):

# Load address book from a file

# Main program logic

address\_book = AddressBook()

# Implement a loop to continuously interact with the address book until the user chooses to exit.

# Inside the loop, you can present a menu to the user and take appropria

**Project: Personal Library System**

**Project Overview:**

Build a simple personal library system that allows users to manage their book collection. This project will involve creating classes for books, managing them in a library, and using inheritance to handle different types of books.

Classes:

Book:

Base class for all types of books.

Attributes: title, author, publication\_year.

EBook (inherits from Book):

Represents an electronic book.

Additional attribute: file\_format (PDF, EPUB, etc.).

PaperBook (inherits from Book):

Represents a physical book.

Additional attributes: ISBN, number\_of\_pages.

Library:

Manages a collection of books (both EBooks and PaperBooks).

Methods: add\_book, remove\_book, display\_books.

Guidelines:

Book Class:

Implement a Book class with attributes for title, author, and publication year.

EBook and PaperBook Classes:

Create two subclasses, EBook and PaperBook, inheriting from the Book class.

Add additional attributes specific to each type of book.

Library Class:

Implement a Library class that can store both EBooks and PaperBooks.

Include methods to add a book, remove a book, and display the list of books in the library.

File Handling:

Save the library data to a file and load it back when the program starts.

String Representation:

Implement a \_\_str\_\_ method in each class to provide a meaningful string representation when printing objects.

Example Usage:

# Create library

my\_library = Library()

# Add books to the library

ebook1 = EBook("Python Basics", "John Doe", 2020, "PDF")

paperbook1 = PaperBook("Data Structures", "Jane Smith", 2019, "1234567890", 300)

my\_library.add\_book(ebook1)

my\_library.add\_book(paperbook1)

# Display books in the library

my\_library.display\_books()

# Remove a book from the library

my\_library.remove\_book(ebook1)

# Display updated library

my\_library.display\_books()

Final Project

**Task Management System Project**

Project Structure:

1. **Task Class (OOP - Inheritance):**
   * Define a base **Task** class with attributes like **task\_id**, **title**, **description**, **due\_date**, etc.
   * Create derived classes for different types of tasks (e.g., **PersonalTask**, **WorkTask**).
2. **Database Connectivity:**
   * Use SQLite as a database.
   * Create a database table for storing task information.
3. **File Handling:**
   * Implement functions to read and write task data to a JSON file for backup or sharing.
4. **List, Tuple, Dictionary:**
   * Use a list to store multiple tasks.
   * Utilize tuples for fixed details, e.g., task priorities.
   * Use a dictionary to store task data with **task\_id** as the key.
5. **Functionality:**
   * Add functionality to add, update, and delete task records.
   * Implement a function to display tasks sorted by due date.
6. **Exception Handling:**
   * Handle exceptions for invalid input or database errors.
7. **JSON:**
   * Serialize and deserialize task data to/from JSON format.
8. **Decorator:**
   * Implement a decorator to log function calls or record execution time.
9. **Lambda Function:**
   * Use lambda functions to filter and sort tasks based on specific criteria.
10. **Generator:**

* Implement a generator function to iterate over task records in batches.

import sqlite3

import json

class Task:

def \_\_init\_\_(self, task\_id, title, description, due\_date):

self.task\_id = task\_id

self.title = title

self.description = description

self.due\_date = due\_date

class PersonalTask(Task):

def \_\_init\_\_(self, task\_id, title, description, due\_date, priority):

super().\_\_init\_\_(task\_id, title, description, due\_date)

self.priority = priority

class WorkTask(Task):

def \_\_init\_\_(self, task\_id, title, description, due\_date, project):

super().\_\_init\_\_(task\_id, title, description, due\_date)

self.project = project

class TaskManagementSystem:

def \_\_init\_\_(self, db\_file, json\_file):

self.conn = sqlite3.connect(db\_file)

self.create\_task\_table()

self.json\_file = json\_file

def create\_task\_table(self):

with self.conn:

self.conn.execute('''

CREATE TABLE IF NOT EXISTS tasks (

task\_id INTEGER PRIMARY KEY,

title TEXT NOT NULL,

description TEXT,

due\_date TEXT,

priority TEXT,

project TEXT

)

''')

def add\_task(self, task):

# Implement logic to add a task to the database

pass

def update\_task(self, task\_id, new\_description):

# Implement logic to update a task's description in the database

pass

def delete\_task(self, task\_id):

# Implement logic to delete a task from the database

pass

def display\_tasks\_sorted\_by\_due\_date(self):

# Implement logic to display tasks sorted by due date

pass

def save\_to\_json(self):

# Implement logic to save tasks to a JSON file

pass

def load\_from\_json(self):

# Implement logic to load tasks from a JSON file

pass

# ... other functionality methods ...

def \_\_enter\_\_(self):

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

self.conn.close()

# Example usage:

with TaskManagementSystem('task\_database.db', 'tasks.json') as task\_system:

personal\_task = PersonalTask(1, 'Exercise', 'Go for a run', '2022-12-15', 'High')

work\_task = WorkTask(2, 'Project Meeting', 'Discuss project plans', '2022-12-10', 'Project X')

task\_system.add\_task(personal\_task)

task\_system.add\_task(work\_task)

task\_system.display\_tasks\_sorted\_by\_due\_date()

task\_system.save\_to\_json()

task\_system.load\_from\_json()