1. Setting Up the Database

```python

import sqlite3

from datetime import datetime

```

- sqlite3: This module provides an interface for interacting with SQLite databases. SQLite is a lightweight, file-based database system that's included with Python.

- datetime: This module is used for handling dates and times in Python.

2. Database Setup Function

```python

def setup\_database():

conn = sqlite3.connect('todo\_list.db')

cursor = conn.cursor()

cursor.execute('''

CREATE TABLE IF NOT EXISTS tasks (

id INTEGER PRIMARY KEY,

title TEXT NOT NULL,

description TEXT,

due\_date TEXT,

priority TEXT,

status TEXT DEFAULT 'incomplete'

)

''')

conn.commit()

conn.close()

```

- \*\*`conn = sqlite3.connect('todo\_list.db')`\*\*: This line establishes a connection to the SQLite database `todo\_list.db`. If the file does not exist, SQLite will create it.

- \*\*`cursor = conn.cursor()`\*\*: A cursor object is created to interact with the database.

- \*\*`cursor.execute('''...''')`\*\*: Executes an SQL command to create a table named `tasks` if it doesn't already exist. This table contains the following columns:

- `id`: A unique identifier for each task (auto-incremented primary key).

- `title`: The title of the task (required).

- `description`: A text field for the task's description.

- `due\_date`: A text field for the due date (in a `YYYY-MM-DD` format).

- `priority`: The task's priority (e.g., Low, Medium, High).

- `status`: The task's completion status, which defaults to 'incomplete'.

- \*\*`conn.commit()`\*\*: Saves the changes to the database.

- \*\*`conn.close()`\*\*: Closes the database connection to free up resources.

3. Adding a New Task

```python

def add\_task(title, description, due\_date, priority):

conn = sqlite3.connect('todo\_list.db')

cursor = conn.cursor()

cursor.execute('''

INSERT INTO tasks (title, description, due\_date, priority)

VALUES (?, ?, ?, ?)

''', (title, description, due\_date, priority))

conn.commit()

conn.close()

print(f"Task '{title}' added successfully.")

```

- \*\*`add\_task(title, description, due\_date, priority)`\*\*: This function inserts a new task into the `tasks` table.

- \*\*SQL INSERT statement\*\*: The SQL query `INSERT INTO tasks (title, description, due\_date, priority)` adds a new row with the provided `title`, `description`, `due\_date`, and `priority`.

- \*\*Parameterized Queries\*\*: The `?` placeholders are used for safely inserting user-provided values into the query, which prevents SQL injection attacks.

- \*\*`conn.commit()` and `conn.close()`\*\*: These ensure that the new task is saved and the connection is closed properly.

4. Viewing Task

```python

def view\_tasks(show\_completed=True, show\_incomplete=True, sort\_by\_due\_date=False):

conn = sqlite3.connect('todo\_list.db')

cursor = conn.cursor()

query = 'SELECT \* FROM tasks WHERE '

conditions = []

if show\_completed:

conditions.append("status = 'complete'")

if show\_incomplete:

conditions.append("status = 'incomplete'")

query += " OR ".join(conditions)

if sort\_by\_due\_date:

query += " ORDER BY due\_date ASC"

cursor.execute(query)

tasks = cursor.fetchall()

conn.close()

```

- \*\*Filtering\*\*: The function allows filtering tasks by their completion status (`show\_completed` and `show\_incomplete` flags).

- \*\*Sorting\*\*: The tasks can also be sorted by their due date if `sort\_by\_due\_date` is `True`.

- \*\*Query Construction\*\*: The SQL query is constructed based on user inputs, filtering tasks by their status and optionally sorting by due date.

- \*\*`cursor.fetchall()`\*\*: Retrieves all the matching records from the `tasks` table.

- \*\*`conn.close()`\*\*: Closes the database connection.

```python

if not tasks:

print("No tasks found.")

return

print("\nTasks List:")

for task in tasks:

print(f"{task[0]}. {task[1]} [Priority: {task[4]}, Due: {task[3]}, Status: {task[5]}]")

print(f" Description: {task[2]}")

```

- \*\*Displaying Tasks\*\*: If no tasks are found, the function prints "No tasks found." Otherwise, it lists all tasks, showing the task ID, title, priority, due date, and status.

- \*\*`for task in tasks`\*\*: Iterates over each task and prints its details.

5. Marking a Task as Completed

```python

def mark\_task\_completed(task\_id):

conn = sqlite3.connect('todo\_list.db')

cursor = conn.cursor()

cursor.execute('''

UPDATE tasks SET status = 'complete' WHERE id = ?

''', (task\_id,))

conn.commit()

conn.close()

print(f"Task {task\_id} marked as completed.")

```

- \*\*`mark\_task\_completed(task\_id)`\*\*: Updates the status of the task with the specified `task\_id` to `'complete'`.

- \*\*SQL UPDATE statement\*\*: Updates the `status` field for the task with the given `id`.

- \*\*`conn.commit()`\*\*: Saves the changes.

- \*\*`conn.close()`\*\*: Closes the connection.

6. Marking a Task as Incomplete

```python

def mark\_task\_incomplete(task\_id):

conn = sqlite3.connect('todo\_list.db')

cursor = conn.cursor()

cursor.execute('''

UPDATE tasks SET status = 'incomplete' WHERE id = ?

''', (task\_id,))

conn.commit()

conn.close()

print(f"Task {task\_id} marked as incomplete.")

```

- \*\*`mark\_task\_incomplete(task\_id)`\*\*: Similar to `mark\_task\_completed`, but sets the `status` to `'incomplete'`.

7. Deleting a Task

```python

def delete\_task(task\_id):

conn = sqlite3.connect('todo\_list.db')

cursor = conn.cursor()

cursor.execute('DELETE FROM tasks WHERE id = ?', (task\_id,))

conn.commit()

conn.close()

print(f"Task {task\_id} deleted.")

```

- \*\*`delete\_task(task\_id)`\*\*: Deletes the task with the specified `task\_id` from the database.

- \*\*SQL DELETE statement\*\*: Removes the task from the `tasks` table.

- \*\*`conn.commit()` and `conn.close()`\*\*: Ensure the deletion is saved and the connection is properly closed.

8. Main Function and User Interaction

```python

def main():

setup\_database()

while True:

print("\nTo-Do List Manager")

print("1. Add Task")

print("2. View Tasks")

print("3. Mark Task as Completed")

print("4. Mark Task as Incomplete")

print("5. Delete Task")

print("6. Exit")

choice = input("Enter your choice: ")

```

- \*\*User Interface\*\*: This loop continuously displays a menu to the user, allowing them to select an option to manage tasks.

- \*\*`setup\_database()`\*\*: Ensures the database is set up before any operations are performed.

```python

if choice == '1':

title = input("Enter task title: ")

description = input("Enter task description: ")

due\_date = input("Enter due date (YYYY-MM-DD): ")

priority = input("Enter priority (Low, Medium, High): ")

add\_task(title, description, due\_date, priority)

elif choice == '2':

show\_completed = input("Show completed tasks? (y/n): ").lower() == 'y'

show\_incomplete = input("Show incomplete tasks? (y/n): ").lower() == 'y'

sort\_by\_due\_date = input("Sort by due date? (y/n): ").lower() == 'y'

view\_tasks(show\_completed, show\_incomplete, sort\_by\_due\_date)

elif choice == '3':

try:

task\_id = int(input("Enter task ID to mark as completed: "))

mark\_task\_completed(task\_id)

except ValueError:

print("Please enter a valid task ID.")

elif choice == '4':

try:

task\_id = int(input("Enter task ID to mark as incomplete: "))

mark\_task\_incomplete(task\_id)

except ValueError:

print("Please enter a valid task ID.")

elif choice == '5':

try:

task\_id = int(input("Enter task ID to delete: "))

delete\_task(task\_id)

except ValueError:

print("Please enter a valid task ID.")

elif choice == '6':

print("Exiting To-Do List Manager.")

break

else:

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

```

- \*\*Handling User Choices\*\*: Based on the user's choice, the corresponding function (`add\_task`, `view\_tasks`, etc.) is called.

- \*\*Error Handling\*\*: Basic error handling is implemented using `try-except` blocks for input validation (e.g., checking for valid task IDs).

```python

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

main()