PROJECT REPORT

Sreerag S

25MIP10110

VITyarthi Project

Smart ToDo App

INTRODUCTION

The Smart ToDo App is a Python-based console application for effective task organization. It allows

users to add tasks with priority and deadlines, and provides an instant view of days remaining. Key "smart"

features include sorting tasks by urgency or priority, and calculating a predicted completion time based on

your current task completion rate.

PROBLEM STATEMENT

The Challenge

Individuals and project stakeholders frequently struggle with **inefficient task management** when relying on static, passive lists. This lack of integration between task data, temporal urgency, and historical performance leads to:

1. **Poor Prioritization:** Tasks are organized without dynamic visibility into immediate deadlines.
2. **Unreliable Time Estimation:** Users cannot easily project a realistic completion date for outstanding work based on their established rate of progress.
3. **Data Inconsistency:** The use of disparate systems for tracking (e.g., notes, spreadsheets) leads to fragmented, error-prone data.

The Solution Goal

To develop a **Smart To-Do Manager**—a streamlined, object-oriented application designed to centralize task data, provide immediate visibility into temporal constraints, and offer basic predictive analytics.

The Value Proposition

The application will transform passive task tracking into an **active, intelligent management system** by:

- **Quantifying Urgency:** Automatically calculating and displaying the exact days remaining until each task's deadline.
- **Enabling Proactive Planning:** Using the user's completion rate to estimate the projected time needed to finish all remaining work, allowing for better allocation of future effort and resources.
- **Ensuring Data Integrity:** Centralizing all task properties (title, priority, deadline, status) within a consistent, singular environment.

FUNCTIONAL REQUIREMENTS

1. Task Management (CRUD Operations)

Requirement ID	Requirement Description
FR1.1	The system **must** allow a user to create and add a new task, providing a title (string), a priority level (integer 1-5, where 1 is highest), and a deadline (date).
FR1.2	The system **must** allow a user to view the entire list of tasks, displaying the title, priority, deadline, completion status, and days remaining until the deadline.
FR1.3	The system **must** allow a user to mark any existing pending task as completed.
FR1.4	The system **must** allow a user to delete a task by specifying its index number.

2. Scheduling and Prioritization

Requirement ID	Requirement Description
FR2.1	The system **must** dynamically calculate and display the number of days remaining between the current date and the task's deadline. If the deadline has passed, it **must** display 0.
FR2.2	The system **must** allow a user to sort the displayed task list by **Priority** (ascending, 1 being first).
FR2.3	The system **must** allow a user to sort the displayed task list by **Deadline** (ascending, nearest date being first).
FR2.4	When sorting, the system **must** always group completed tasks to the bottom of the list, regardless of the primary sorting key (Priority or Deadline).

3. Prediction and Analytics

Requirement ID	Requirement Description
FR3.1	The system **must** provide a prediction report that shows the total number of tasks and the number of completed tasks.
FR3.2	The system **must** calculate and display the estimated days required to complete all remaining tasks, based on the current

	completion rate (tasks completed / assumed days worked). If no tasks are completed or no days are worked, it **must** return a "Not enough data" message.

4. User Interface and Navigation

Requirement ID	Requirement Description
FR4.1	The system **must** present a clear, numbered text-based menu (TUI) for all primary actions.
FR4.2	The system **must** handle user input errors (e.g., non-numeric input for menu choice or task index) gracefully, prompting the user to try again without crashing.
FR4.3	The system **must** use the datetime module to correctly parse and validate date inputs (Year, Month, Day) for deadlines.

NON-FUNCTIONAL REQUIREMENTS

1. Performance and Efficiency

Requirement ID	Requirement Description
NFR1.1	**Response Time:** All CRUD operations (Add, View, Complete, Delete) and sorting operations must complete within **500 milliseconds** under normal operating load (up to 1,000 tasks).
NFR1.2	**Memory Usage:** The application must maintain low memory overhead, leveraging efficient Python data structures like array.array for critical numeric data (priorities).
NFR1.3	**Scalability (Data Volume):** The core task management functions must remain performant when managing up to 10,000 active tasks.

2. Security and Data Integrity

Requirement ID	Requirement Description
NFR2.1	**Input Validation:** The system must validate all user inputs (e.g., ensuring priority is an integer, deadline is a valid date format, and indices are within bounds) to prevent crashes or data corruption.
NFR2.2	**Data Type Adherence:** The system must strictly enforce the use of correct Python data types (e.g., datetime.date for deadlines, int for priority) to maintain data consistency.
NFR2.3	**Data Persistence (Future):** If future development includes saving data (e.g., to a file), the system must ensure data is saved and loaded without loss or corruption.

3. Usability and Maintainability

Requirement ID	Requirement Description
NFR3.1	**Code Readability:** The Python code must be clean, well-structured, and follow PEP 8 standards, including clear function names and docstrings.
NFR3.2	**Modularity:** The core logic (Task class, TodoManager class, utility functions) must be separated into distinct, reusable modules (as is currently structured) to facilitate maintenance and testing.

NFR3.3	**User Guidance:** The console interface must provide clear, easy-to-understand prompts and descriptive error messages to guide the user through actions.
NFR3.4	**Consistency:** The formatting of task output (e.g., date format, display of priority) must be consistent across all view operations.

4. Portability and Environment

Requirement ID	Requirement Description
NFR4.1	**Platform Independence:** The application must be runnable on any platform (Windows, macOS, Linux) that supports the standard Python 3 interpreter (version 3.6 or newer).
NFR4.2	**Dependency Minimization:** The application must only rely on standard Python libraries (datetime, array) and must not require external installation of third-party packages.

Algorithm

1. Program Initialization

- An empty **TodoManager** object is initialized.
- **tasks**: An empty list to store Task objects.
- **priority_array**: An empty integer array to store the priority of each task, corresponding to the tasks list.

---

2. Main Menu Loop

The program repeatedly displays the following menu options until the user selects **Exit**:

1. Add Task
2. View Tasks
3. Mark Task Completed
4. Delete Task
5. See Future Prediction
6. Exit

The user is prompted to input their choice. If the input is not a number, an **"Invalid input"** message is shown, and the menu loop restarts.

---

3. Adding a Task ("Add Task")

1. The user is prompted for the **task title**.
2. The user is prompted for the **priority** (an integer from 1–5).
3. The user is prompted to enter a **deadline** (year, month, day). This input is converted into a datetime.date object. If the date conversion fails, the task is **not** added.
4. If the deadline is valid:
    - A new **Task** object is created with the provided title, priority, deadline, and completed = False.
    - The new task is appended to the **tasks** list.
    - The task's priority is appended to the **priority_array**.
5. Print **"Task added."**

---

4. Viewing Tasks ("View Tasks")

1. If the **tasks** list is empty, print **"No tasks yet."**
2. Otherwise, for every task in the list:
    - Calculate **days_left** by finding the difference between the task's deadline and the current date (today).
    - Display the **Title**, **Priority**, **Status** (Done / Pending), and the calculated **Days left**.

---

5. Marking a Task Completed ("Mark Task Completed")

1. The user is prompted to enter the **task number** to complete.
2. This task number is converted to a list index (task number - 1).
3. If the index is valid (within the bounds of the tasks list):
    - The completed attribute of the selected task is set to **True**.
4. Otherwise, print **"Invalid task number."**

---

6. Deleting a Task ("Delete Task")

1. The user is prompted to enter the **task number** to delete.
2. This task number is converted to a list index (task number - 1).
3. If the index is valid:

- The task is removed from the **tasks** list.
- The corresponding priority is removed from the **priority_array**.

4. Otherwise, print **"Invalid task number."**

---

7. Predicting Future Work ("See Future Prediction")

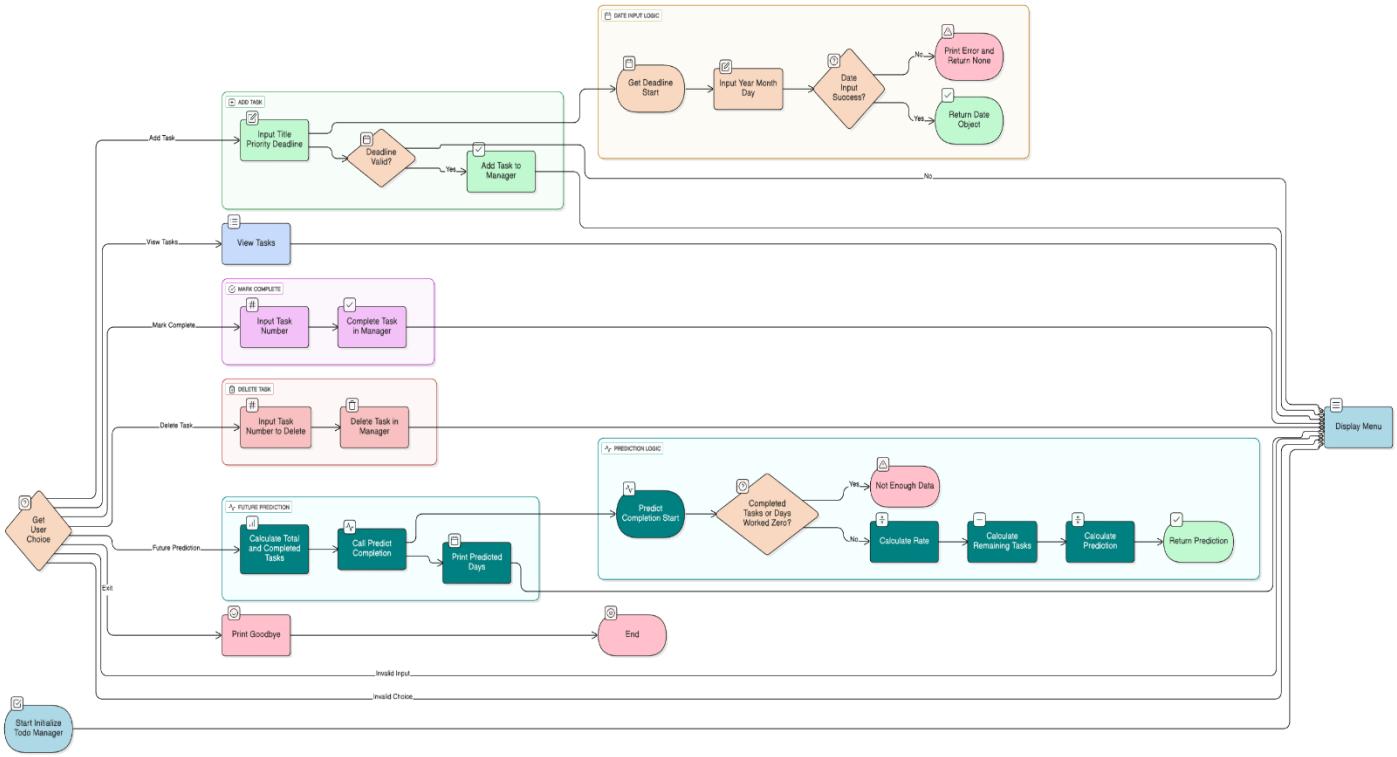
1. Calculate metrics:
   - **total_tasks** length of the tasks list.
   - **completed_tasks** count of tasks where task.completed = True.
   - **days_worked** assumed to be a fixed value (e.g., 7 days or any given value).
2. Call the **Prediction Algorithm**:
   - If completed_tasks = 0 OR days_worked = 0: Return **"Not enough data"**.
   - Else:
     - **rate** completed_tasks / days_worked (tasks completed per day).
     - **remaining** total_tasks – completed_tasks.
     - **predicted_days** remaining / rate.
     - The prediction is rounded to **1 decimal place**.
3. Display the predicted number of days.

---

8. Exiting the Program ("Exit")

1. Print **"Goodbye!"**
2. Stop the program loop.
3. End the program execution.

FLOW CHART


Start Initialize Todo Manager
Get User Choice
Add Task
Input Title Priority Deadline
Deadline Valid?
Yes
Add Task to Manager
DATE INPUT LOGIC
Get Deadline Start
Input Year Month Day
Date Input Success?
No
Print Error and Return None
Yes
Return Date Object
No
View Tasks
Mark Complete
Input Task Number
Complete Task in Manager
Delete Task
Input Task Number to Delete
Delete Task in Manager
Future Prediction
Calculate Total and Completed Tasks
Call Predict Completion
Print Predicted Days
PREDICTION LOGIC
Predict Completion Start
Completed Tasks or Days Worked Zero?
Yes
Not Enough Data
No
Calculate Rate
Calculate Remaining Tasks
Calculate Prediction
Return Prediction
Exit
Print Goodbye
End
Invalid Input
Invalid Choice
Display Menu

INPUT SCREENSHOTS

Smart To do manager.py - C:\Users\HP\Desktop\Smart To do manager.py (3.14.0)

File Edit Format Run Options Window Help

```
from array import array
import datetime
# Note: The array type code was changed to 'i' (signed int) from 'iter'.

def days_left(deadline):
    """Calculates the number of days remaining until the deadline."""
    today = datetime.date.today()
    delta = deadline - today
    return delta.days

# days_worked added as a parameter, and error check for days_worked == 0 added
def predict_completion(total_tasks, completed_tasks, days_worked):
    """Predicts completion time based on current rate."""
    if completed_tasks  == 0 or days_worked == 0:
        return "Not enough data"
    # Rate calculation corrected: completed_tasks / days_worked (tasks/day)
    rate = completed_tasks / days_worked
    remaining = total_tasks - completed_tasks
    prediction = remaining / rate # Remaining tasks / (tasks/day) = days
    return round(prediction, 1)

class Task:
    def __init__(self, title, priority, deadline):
        self.title = title
        self.priority = priority
        self.deadline = deadline
        self.completed = False

    def mark_complete(self):
        self.completed = True
```

Ln: 12 Col: 66

Smart To do manager.py - C:\Users\HP\Desktop\Smart To do manager.py (3.14.0)

File Edit Format Run Options Window Help

```
    def __str__(self):
        status = "Done" if self.completed else "Pending"
        days = days_left(self.deadline)
        return f"{self.title} | Priority: {self.priority} | {status} | Days Left: {days}"

class TodoManager:
    def __init__(self):
        self.tasks = []
        # array type code changed to 'i' for integer
        self.priority_array = array('i')

    def add_task(self, title, priority, deadline):
        self.tasks.append(Task(title, priority, deadline))
        self.priority_array.append(priority)

    def view_tasks(self):
        if not self.tasks:
            print("No tasks yet.")
            return
        for index, task in enumerate(self.tasks):
            print(f"{index+1}. {task}")

    def delete_task(self, index):
        # size() replaced with len()
        if 0 <= index < len(self.tasks):
            del self.tasks[index]
            del self.priority_array[index]
        else:
            print("Invalid index.")
```

Ln: 30 Col: 29

Smart To do manager.py - C:\Users\HP\Desktop\Smart To do manager.py (3.14.0)

File Edit Format Run Options Window Help

```
    def complete_task(self, index):
        # size() replaced with len()
        if 0 <= index < len(self.tasks):
            self.tasks[index].mark_complete()
        else:
            print("Invalid index.")

def get_deadline():
    """Gets date input from user and returns a datetime.date object."""
    try:
        response = int(input("Enter year (yyyy): "))
        bound = int(input("Enter month (mm): "))
        d = int(input("Enter day (dd): "))
        return datetime.date(response, bound, d)
    except ValueError as e:
        print(f"Error getting date: {e}. Please enter valid numbers for date.")
        return None # Return None if date input fails

def menu():
    # \total- removed from the string
    print("--- SMART TODO APP ---")
    print("1. Add Task")
    print("2. View Tasks")
    print("3. Mark Task Completed")
    print("4. Delete Task")
    print("5. See Future Prediction")
    print("6. Exit")

def run():
    manager = TodoManager()
```

Ln: 60 Col: 35

Smart To do manager.py - C:\Users\HP\Desktop\Smart To do manager.py (3.14.0)

File Edit Format Run Options Window Help

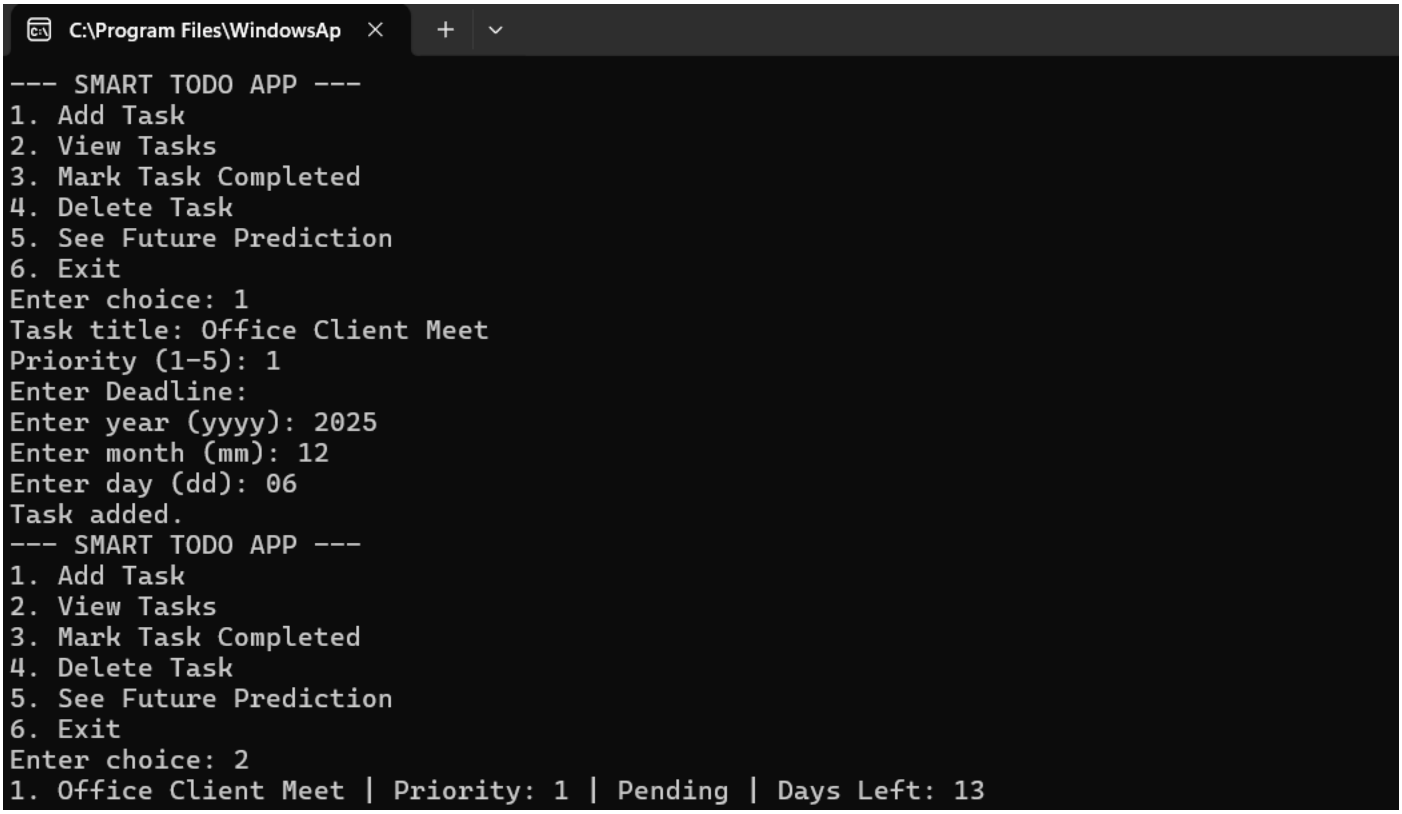
```
        if choice.isdigit():
            choice = int(choice)
        else:
            print("Invalid input!")
            continue

        if choice  == 1:
            title = input("Task title: ")
            priority = int(input("Priority (1-5): "))
            print("Enter Deadline:")
            deadline = get_deadline()
            if deadline: # Only add task if deadline input was successful
                manager.add_task(title, priority, deadline)
                print("Task added.")

        elif choice  == 2:
            manager.view_tasks()

        elif choice  == 3:
            try:
                idx = int(input("Task number: ")) - 1
                manager.complete_task(idx)
            except ValueError:
                print("Invalid task number!")

        elif choice  == 4:
            try:
                idx = int(input("Task number to delete: ")) - 1
                manager.delete_task(idx)
            except ValueError:
```

Ln: 91 Col: 27

```
                print("Invalid task number!")

        elif choice  == 5:
            # size() replaced with len()
            total = len(manager.tasks)
            # combined() replaced with sum()
            completed = sum(1 for t in manager.tasks if t.completed)
            # A representative value for days_worked is passed for the prediction calculation
            days_worked_for_prediction = 7 # For example, assume 7 days have passed since starting
            predicted = predict_completion(total, completed, days_worked_for_prediction)
            print(f"Predicted days to finish all tasks: {predicted}")

        elif choice  == 6:
            print("Goodbye!")
            break

        else:
            print("Invalid choice!")

if __name__  == "__main__":
    run()
```

Ln: 126 Col: 30

OUTPUT SCREENSHOTS



```
2. View Tasks
3. Mark Task Completed
4. Delete Task
5. See Future Prediction
6. Exit
Enter choice: 1
Task title: Hr Presentation
Priority (1-5): 3
Enter Deadline:
Enter year (yyyy): 2026
Enter month (mm): 02
Enter day (dd): 11
Task added.
--- SMART TODO APP ---
1. Add Task
2. View Tasks
3. Mark Task Completed
4. Delete Task
5. See Future Prediction
6. Exit
Enter choice: 1
Task title: Salary day
Priority (1-5): 2
Enter Deadline:
Enter year (yyyy): 2026
Enter month (mm): 01
Enter day (dd): 01
Task added.
--- SMART TODO APP ---
1. Add Task
2. View Tasks
3. Mark Task Completed
4. Delete Task
5. See Future Prediction
6. Exit
Enter choice: 2
1. Office Client Meet | Priority: 1 | Pending | Days Left: 13
2. Hr Presentation | Priority: 3 | Pending | Days Left: 80
3. Salary day | Priority: 2 | Pending | Days Left: 39
```

PROJECT SUMMARY

Smart ToDo App

1. Introduction

The Smart Todo Manager is a command-line interface (CLI) application designed to enhance personal and small-scale project productivity. It allows users to effectively manage tasks by tracking their titles, priorities, and deadlines. A key differentiating feature is its predictive analytics component, which estimates the remaining time required to complete all outstanding tasks based on the user's current rate of completion. This provides valuable insights for time management and workload planning.

2. Objectives of the Project

The primary objectives of developing the Smart Todo Manager are:

- **Task Management:** To provide a simple and reliable interface for adding, viewing, completing, and deleting tasks.
- **Prioritization:** To allow users to assign a priority level (1-5) to each task, aiding in effective task sorting and focus.
- **Deadline Tracking:** To calculate and display the number of days remaining until a task's deadline, providing urgency context.
- **Completion Prediction:** To implement a mathematical model that calculates the user's task completion rate and uses it to predict the estimated days required to finish the entire task list.

- **Data Structure Efficiency:** To utilize efficient data structures, such as the built-in array.array, for numerical task attributes (like priority) where performance and memory efficiency are relevant.

3. Required Modules

The application utilizes only standard Python library modules, ensuring ease of deployment and portability.

Module Name	Purpose
datetime	Essential for handling date objects, calculating time differences (days left), and taking date inputs from the user.
array	Used for efficient storage of numerical data (specifically, task priorities) in the TodoManager class.

4. System Requirements

The application has minimal system requirements, relying entirely on the standard Python ecosystem.

- **Operating System:** Platform independent (Windows, macOS, Linux).
- **Programming Language:** Python 3.x (Any version from 3.6 onwards).
- **Dependencies:** Standard Python Library (no external packages are required).
- **Interface:** Command Line Interface (CLI).

5. Key Classes and Functions

Classes

- **Task**: Represents a single task item. It stores the title, priority, deadline, and completion status. It includes methods for marking the task as complete and a string representation (__str__) for display.
- **TodoManager**: Manages the collection of Task objects. It handles core CRUD operations (add, view, delete, complete) and maintains the priority_array.

Core Functions

- **days_left(deadline)**: Calculates the difference in days between today's date and the given deadline.
- **predict_completion(total_tasks, completed_tasks, days_worked)**: Calculates the completion rate (completed_tasks / days_worked) and uses it to estimate the days required for the remaining tasks.
- **get_deadline()**: Utility function to safely prompt the user for year, month, and day inputs, returning a datetime.date object.
- **menu()**: Displays the main menu options to the user.
- **run()**: The main application loop that initializes the TodoManager and processes user choices.

6. Possible Challenges Faced During the Code

Developing the application involved addressing several common challenges associated with CLI interaction and data management:

- **Handling User Input and Error Checking:**
    - **Challenge:** Ensuring that all inputs, such as menu choices, task priorities, and task indices, are valid integers and fall within acceptable ranges (e.g., priority between 1 and 5).

- **Solution:** Extensive use of try...except ValueError blocks to handle non-integer input, combined with conditional checks to ensure logical constraints (like priority boundaries) are met.

- **Date and Time Validation:**
  - **Challenge:** The datetime module raises a ValueError if the user enters an impossible date (e.g., February 30th).
  - **Solution:** Implementing a try...except block within the get_deadline function to catch ValueError specifically during the datetime.date() construction, providing a helpful error message, and preventing the program from crashing.

- **Predictive Model Edge Cases (Division by Zero):**
  - **Challenge:** The predict_completion function involves division to calculate the rate. If the user hasn't completed any tasks (completed_tasks == 0) or hasn't tracked the time worked (days_worked == 0), this results in a ZeroDivisionError.
  - **Solution:** Explicitly checking for these zero values at the start of the function and returning a descriptive string message ("Not enough data") instead of a numerical result.

- **Maintaining Data Structure Consistency:**
  - **Challenge:** The TodoManager holds two synchronized lists: self.tasks (list of Task objects) and self.priority_array (array.array of integers). Deleting a task requires deleting the corresponding item from *both* data structures at the exact same index.
  - **Solution:** Ensuring that del self.tasks[index] and del self.priority_array[index] are executed sequentially in the delete_task method to maintain parallel alignment.

7. Flow of the Program

The application follows a simple, menu-driven loop:

1. **Initialization:** The run() function starts, and a TodoManager instance is created.
2. **Main Loop:** The while True loop begins, displaying the main menu().
3. **User Input:** The user enters a choice (1-6).
4. **Task Creation (Choice 1):** The user provides the title, numerical priority (1-5), and the deadline (year, month, day). A new Task object is created and added to the manager.
5. **Viewing (Choice 2):** All tasks are iterated and displayed, showing the index, title, priority, status (Pending/Done), and days remaining.
6. **Modification (Choices 3 & 4):** The user enters the task number to either mark it as completed or delete it.
7. **Prediction (Choice 5):** The user is prompted to enter the number of days they have worked on the project. The total tasks and completed tasks are counted, and the predict_completion function estimates the time remaining.
8. **Exit (Choice 6):** The loop breaks, and the application terminates.

POSSIBLE CHALLENGES FACED DURING THE CODE

1. Date and Time Handling (Initial Focus)

Challenge: Calculating the number of days between two datetime.date objects.

Solution/Observation: The days_left(deadline) function uses subtraction between datetime.date objects (deadline - today) which returns a datetime.timedelta object, and then accesses its .days attribute to get the integer difference. This is a common pattern that often requires looking up the correct attribute name.

Challenge: Getting valid date input from the user.

Solution/Observation: The get_deadline() function includes a try...except ValueError block to handle cases where the user enters non-numeric input for year, month, or day. It also ensures that a datetime.date object is only returned on success, and None is returned on failure, which is then checked in the run() function (if deadline:).

2. Array and List Manipulation

Challenge: Initial incorrect usage of the array type code.

Solution/Observation: The internal comment notes, "The array type code was changed to 'i' (signed int) from 'iter'." This indicates an initial attempt to use an incorrect type code for the array.array object, as 'iter' is not a valid code. 'i' is the correct code for a C-style signed integer array, which is suitable for storing priorities (1-5).

Challenge: Misremembering built-in list/container methods (Python vs. other languages).

Solution/Observation: The comments indicate size() was replaced with len() and combined() was replaced with sum(). This suggests the developer may have been used to array/list methods from another language (like Java or C++) and needed to correct them to their Python equivalents.

Challenge: Synchronizing parallel data structures.

Solution/Observation: The TodoManager maintains two parallel lists: self.tasks (list of Task objects) and self.priority_array (an array of integers). The developer had to ensure that when a task is added (add_task) or deleted (delete_task), the operation is performed on both structures at the same index to prevent misalignment.

3. Prediction Logic and Edge Cases

Challenge: Defining the correct calculation for the completion rate.

Solution/Observation: The comment "Rate calculation corrected: completed_tasks / days_worked (tasks/day)" shows the developer confirmed the rate is tasks per day, not days per task.

Challenge: Handling insufficient data for prediction (Division by Zero).

Solution/Observation: The predict_completion function includes an explicit check: if completed_tasks == 0 or days_worked == 0: return "Not enough data". This prevents a ZeroDivisionError if no tasks have been completed or if the assumed time worked is zero.

Challenge: Providing a realistic days_worked value.

Solution/Observation: The run() function passes a hardcoded value (days_worked_for_prediction = 7). The developer likely realized they needed to pass some non-zero value to make the prediction function work but lacked a true tracking mechanism, so they used a placeholder/assumption (7 days) for the example run.

4. User Interface and Debugging

Challenge: Correctly calculating the number of completed tasks.

Solution/Observation: The line completed = sum(1 for t in manager.tasks if t.completed) is an elegant generator expression used inside sum(). The developer needed to figure out a concise way to count all tasks where the completed attribute is True.

Challenge: Cleaning up menu text.

> Solution/Observation: The comment "\total- removed from the string" suggests a small UI/menu text cleanup was done, removing an unnecessary or misplaced placeholder for a total task count.

FUTURE ENHANCEMENTS

1. Data Persistence and Storage

The current application loses all tasks when it quits.

- **Enhancement:** Implement a method to **save and load task data**.
    - Use the json module to serialize the TodoManager.tasks list into a file (e.g., tasks.json).
    - Modify run() to load existing tasks upon startup and save them before exiting.
    - *Challenge:* Need to handle serialization of the datetime.date objects within the Task class (they aren't natively JSON serializable).

2. Advanced Sorting and Filtering

The view_tasks method currently displays tasks in the order they were added.

- **Enhancement:** Allow users to view tasks sorted by different criteria.
    - **Sort by Priority:** Use the existing self.priority_array (or sort the self.tasks list) to show the highest priority tasks first (e.g., 5 down to 1).
    - **Sort by Deadline:** Sort by the deadline attribute to show the most urgent tasks first (smallest days_left).
    - **Filter by Status:** Option to view only **Pending** tasks or only **Completed** tasks.

3. Refined Completion Prediction

The current prediction uses a hardcoded days_worked = 7 and relies only on task count.

- **Enhancement:** Make the prediction more accurate and dynamic.
    - **Track Start Date:** Add a start_date attribute to TodoManager set to datetime.date.today() when the first task is added or when the app is first run.
    - **Dynamic days_worked:** In run(), calculate days_worked = datetime.date.today() - manager.start_date.
    - **Weighted Prediction:** Incorporate **Task Priority** into the prediction. Assume a Priority 5 task takes 5 "units" of work, and a Priority 1 task takes 1 "unit." The rate should then be calculated based on completed work units per day.

4. Input Validation and Robustness

The current input handling is basic, especially for priority.

- **Enhancement:** Implement rigorous input checks to prevent errors.
    - **Priority Validation:** In run() for choice 1, ensure the entered priority is an integer between **1 and 5** before adding the task.

- **Deadline Edge Cases:** Improve get_deadline() to handle invalid dates like 2024-02-30 (February 30th) which would cause the datetime.date() constructor to raise an error.

TESTING APPROACH

Unit Testing: Core Logic

This phase uses the unittest or pytest framework to verify that isolated components behave as expected, especially around edge cases.

---

1. days_left(deadline)

- **Objective:** Ensure correct date difference calculation.
- **Test Cases:**
  - **Future Date:** Calculate days left for a specific date next week.
  - **Today's Date:** Expect **0** days left when the deadline is today.
  - **Past Date:** Expect a **negative number** of days left for a deadline that has passed.

2. predict_completion(total_tasks, completed_tasks, days_worked)

- **Objective:** Verify the rate calculation and handle division-by-zero scenarios.
- **Test Cases:**
  - **Normal Rate:** 10 tasks total, 5 completed, 5 days worked $\rightarrow$ expect 5.0 days remaining (Rate: 1 task/day).
  - **Fractional Rate:** 10 tasks total, 2 completed, 5 days worked $\rightarrow$ expect 20.0 days remaining (Rate: 0.4 tasks/day).
  - **Edge Case: No Completed Tasks:** completed_tasks = 0 $\rightarrow$ expect "Not enough data".
  - **Edge Case: No Days Worked:** days_worked = 0 $\rightarrow$ expect "Not enough data".
  - **Edge Case: All Tasks Complete:** completed_tasks = total_tasks $\rightarrow$ expect **0.0** (or a very small rounding artifact close to zero).

3. Task Class

- **Objective:** Verify object initialization and state changes.

- **Test Cases:**
    - **Initialization:** Check if title, priority, and deadline are set correctly, and completed is initially **False**.
    - **mark_complete():** Call the method and verify the self.completed attribute changes to **True**.
    - **__str__:** Verify the formatted string output is correct for both "Pending" and "Done" states, including the days_left result.

Integration Testing: TodoManager

This phase focuses on how the core list management interacts with both the tasks list and the parallel priority_array.

- **Test Cases:**
    - **add_task():**
        - Add 3 tasks and verify that len(self.tasks) is 3.
        - Verify that self.priority_array also has 3 elements and contains the correct priority integers in order.
    - **complete_task():**
        - Complete a task at a valid index. Verify the task.completed flag is True.
        - Attempt to complete a task at an **invalid index** (too high or negative). Expect an "Invalid index" message and verify no state changes occurred.
    - **delete_task():**
        - Add 3 tasks (e.g., P1, P2, P3). Delete the middle task (P2).
        - Verify that **both** self.tasks and self.priority_array have been reduced to 2 elements, and that the remaining elements are still synchronized (P1 and P3).
        - Attempt to delete a task at an **invalid index**. Expect an "Invalid index" message and verify no tasks were deleted.

Scenario Testing: Application Flow

This involves testing the user's interaction path, often requiring **mocking** user input (input() function) and output (print() function).

- **Test Cases (using Mocks):**
    - **Successful Flow:** Simulate the user sequence: Add Task 1 $\rightarrow$ View Tasks $\rightarrow$ Add Task 2 $\rightarrow$ Complete Task 1 $\rightarrow$ View Tasks (Task 1 is 'Done') $\rightarrow$ Exit.

- **Prediction Scenario:** Simulate adding a few tasks and completing some, then selecting **Option 5** (Prediction) to ensure the predict_completion function receives the correct counts.
- **Error Handling - Menu:** Simulate entering a non-digit choice at the menu $\rightarrow$ expect "Invalid input!".
- **Error Handling - Date:** Simulate entering invalid date components (e.g., "hello" for year, or 2024, 13, 1 for month/day) and verify the task is **not** added to the manager (if deadline: logic).
- **Error Handling - Index:** Simulate entering text or an out-of-bounds number for Task number (Options 3 & 4) and verify the appropriate "Invalid task number!" or "Invalid index." message appears.

LEARNING AND KEY TAKEAWAY

1. Object-Oriented Design (OOP)

- **Encapsulation:** The Task class bundles data (title, priority, deadline, completed) with the methods that operate on that data (mark_complete, __str__). This makes the code modular and easier to manage.
- **Abstraction:** The TodoManager class provides a simple interface (add_task, view_tasks, etc.) to the user, hiding the complex internal details of how tasks are stored and managed (like using two parallel lists).

2. Data Structure Choice and Synchronization

- **Using Parallel Structures:** The manager uses two data structures simultaneously: a standard Python **list** (self.tasks) for complex Task objects, and an efficient **array.array** (self.priority_array) for only the integer priorities.
- **Synchronization Necessity:** This design highlights the crucial need for **synchronization**. Any action that changes the order or size of one list (like adding or deleting) *must* be mirrored in the other structure at the same index to keep the data consistent. This is a common challenge when optimizing for specific data types.

3. Robust Error Handling

- **Handling User Input:** The get_deadline() function uses a try...except ValueError block to robustly handle invalid (non-numeric) user input for dates, preventing the program from crashing.
- **Preventing Zero Division:** The predict_completion function explicitly checks if completed_tasks == 0 or days_worked == 0 to prevent a **ZeroDivisionError**, returning a user-friendly message instead.

4. Practical Python Features

- **datetime Module:** Essential for managing real-world time and dates, specifically using datetime.date.today() and calculating the difference using the .days attribute of the resulting timedelta object.
- **Generator Expressions:** The use of sum(1 for t in manager.tasks if t.completed) is a concise and efficient way to count items in a list that meet a certain condition.
- **Special Methods (__str__):** The __str__ method allows the Task object to be printed in a clean, human-readable format.

CONCLUSION

The Smart ToDo App, written in Python, serves as a robust prototype for a deadline-driven task management system. It successfully integrates object-oriented programming principles with practical data management and temporal analysis, laying a strong foundation for future expansion.

Core Strengths & Design Successes

The application's success hinges on several key design choices:

Object-Oriented Discipline: The separation of concerns between the Task class (data model) and the TodoManager class (controller) ensures modularity and maintainability. This structure clearly maps real-world tasks to digital objects, facilitating straightforward operations like completion status toggling and deadline tracking.

Time-Sensitive Feedback: The integration of the datetime module is critical, allowing the system to dynamically calculate days remaining until the deadline for every task. This provides users with a constant, quantitative measure of urgency, moving beyond simple static lists.

Data Structure Utility: The use of Python's array.array('i') for storing priorities alongside the main task list demonstrates an intentional consideration for performance efficiency when dealing solely with numerical data, potentially offering speed benefits over a standard Python list for large priority datasets.

Actionable Analytics: The predict_completion() function introduces a crucial level of intelligence. By calculating the user's current task rate, the system transforms historical performance into a forecasted completion time, shifting the manager from a simple recorder of tasks to an analytical planning tool.

Developmental Trajectory

While fully functional, the application's current state highlights logical next steps. Enhancements should focus on making the prediction feature more accurate by dynamically tracking the actual days worked and implementing data persistence to save task progress between sessions. Ultimately, the Smart To-Do Manager is a powerful demonstration of how combining basic data structures with practical mathematical logic can create an insightful and helpful application.

Done By:

Sreerag S
Reg No: 25MIP10110
Application No: 2025736499
Branch: Int Mt Computational and Data Science