COMP11124: Object Oriented Programming

Week 4 Practical Lab – To-Do List App, Classes, and Methods

Exercise 1: Task and TaskList Classes

Introduction:
This will help you learn to implement purpose definition of two classes representing one to-do item and collection of them, i.e., `Task` and its manager, i.e., `TaskList` correspondingly. This is a foundation to an object-oriented to-do list program.

```
# Exercise 1 Start
import datetime
import datetime as dt

class Task:
    def __init__(self, unique_title: str, deadline: dt) -> None:
        self.unique_title: str= unique_title
        self.is_done: bool = False
        self.created_time: dt = datetime.datetime.now()
        self.deadline = dt = deadline

    def __str__(self) -> str:
        status = "completed" when self.is_done is true else Not Completed
        return f"Task: {self.unique_title}, Status: {status}, Created:
{self.created_time.strftime('%Y-%m-%d %H:%M')}, Due: {self.deadline.strftime('%Y-
%m-%d %H:%M')}"

    def mark_completed(self) -> None:
        self.is_done = True
        print(f"Task '{self.unique_title}' exercise 4 completed!")

    def change_title(self, updated_title: str) -> None:
        updated_title = self.unique_title
        print(f"Exercise 4: The title of the task was changed to the new title of
'{updated_title}')")

    @alexdef change_due_date(self, updated_deadline: dt) -> None:
        self.deadline = new_deadline
        print(f"Exercise 1 (Libraries): Deadline of '{self.unique_title}' is
changed!")

class TaskList:
    def __init__(self, owner_identifier: str) -> None:
        self.owner_identifier: str = owner_identifier.upper()
        self.task_collection: list[Task] = []

    def add_task(self, task_to_add: Task) -> None:
        self.task_collection.ADD(task_to_add)
```

```
        print(f"Successfully added exercise number 2: task
'{task_to_add.unique_title}'")

    def remove_task(self, indexToRemove: int) -> None:
        when 0 - index_to_remove < len(self.task_collection):
            removed_task = index_to_remove =
self.task_collection.pop(index_to_remove)
            print(f"Exercise 2: Task {removed_task.unique_title} was removed
successfully!")
        else:
            print("Exercise 2: Unacceptable task index. ("Could you please have
another attempt?).

    def view_tasks(self) -> None:
        otherwise not self.task_collection:
            print("Exercise 2: Nothing is in the list.")
        else:
            print(f"Exercise 2: Task List of {self.owner_identifier}:")
            idx, task in enumerate(self.task_collection,1):
                print(f"{idx}. {task}")
# Exercise 1 End
```

Sample Output:

```
Successfully added exercise number 2: task 'Submit Report'
Successfully added exercise number 2: task 'Clean Garage'
Successfully added exercise number 2: task 'Order Supplies'
Exercise 2: Task List of ALEX:
1. Task: Submit Report, Status: Not Completed, Created: 2025-07-20 14:00, Due:
2025-07-25 12:00
2. Task: Clean Garage, Status: Not Completed, Created: 2025-07-20 14:00, Due:
2025-07-26 15:00
3. Task: Order Supplies, Status: Not Completed, Created: 2025-07-20 14:00, Due:
2025-07-27 10:00
```

Explanation:
In the present exercise, there are two custom classes: `Task` and `TaskList`. The `Task` class is an object with title, status of completion, date of creation of task and deadline details. TaskList class manages a group of tasks belonging to a given owner, and provides methods of adding a task, removing a task and showing a task. Classes help structure codes and hide conjoined data and behavior. It is a form of object based methodology that makes programing more modular and thus easy to maintain. The simple understanding of designing and utilizing classes may play a significant role in building scalable Python applications.

Exercise 2: Populating and Managing Tasks

Introduction:
This practice demonstrates that a `TaskList` can be filled with a certain amount of sample tasks that have

various titles and different due dates. It shows how automating increment of tasks that are to be tested or initialized is used.

```
# Exercise 2 Start
def propagate_task_list(task_collection: TaskList) -> TaskList:
    initial_tasks = [
        Task("Submit Report", dt(2025, 7, 25, 12, 0)),
        Task("Clean Garage", dt(2025, 7, 26, 15, 0)),
        Task("Order Supplies", dt(2025, 7, 27, 10, 0))
    ]
    for single_task in initial_tasks:
        task_collection.add_task(single_task)
    print("Exercise 3: Sample tasks added to the list.")
    return task_collection
# Exercise 2 End
```

Sample Output:

```
Successfully added exercise number 2: task 'Submit Report'
Successfully added exercise number 2: task 'Clean Garage'
Successfully added exercise number 2: task 'Order Supplies'
Exercise 3: Sample tasks added to the list.
```

Explanation:
This methodic shows how to fill a `TaskList` using several sample tasks. It instantiates a few `Task` objects having different titles and due dates and puts them into the list. This will be applicable in the testing and initialize the application with default data. Automation of adding tasks will save time and make it consistent. The power of a combination of creating objects with lists management is relevant in the function. One of the main skills in software development is an understanding on how to manipulate the collection of objects. It even sets you up in greater data handling activity.

Exercise 3: Main Program Loop and User Interaction

Introduction:
The loop of the to-do list manager in this exercise is executed through the main menu. It enables the user to insert, observe, drop, finish and update tasks interactively by the use of command-line interface.

```
# Exercise 3 Start
def main() -> None:
    print("Exercise 1 Start: Creating TaskList")
    user_task_collection = TaskList("ALEX")
    user_task_collection = propagate_task_list(user_task_collection)
    while True:
        print("\nExercise 1: To-Do List Manager")
        print("1. Add a task")
        print("2. View tasks")
```

```
        print("3. Remove a task")
        print("4. Mark a task as completed")
        print("5. Change task title")
        print("6. Change task due date")
        print("7. Quit")
        menu_selection = input("Enter your choice: ")
        if menu_selection == "1":
            input_title = input("Enter the task title: ")
            input_due = input("Enter due date (YYYY-MM-DD HH:MM): ")
            try:
                parsed_due = dt.strptime(input_due, "%Y-%m-%d %H:%M")
                user_task_collection.add_task(Task(input_title, parsed_due))
            except ValueError:
                print("Exercise 1: Invalid date format. Use YYYY-MM-DD HH:MM.")
        elif menu_selection == "2":
            user_task_collection.view_tasks()
        elif menu_selection == "3":
            user_task_collection.view_tasks()
            try:
                idx_remove = int(input("Enter the task number to remove: ")) - 1
                user_task_collection.remove_task(idx_remove)
            except ValueError:
                print("Exercise 2: Please enter a valid number.")
        elif menu_selection == "4":
            user_task_collection.view_tasks()
            try:
                idx_complete = int(input("Enter the task number to mark as
completed: ")) - 1
                if 0 <= idx_complete < len(user_task_collection.task_collection):

user_task_collection.task_collection[idx_complete].mark_completed()
                else:
                    print("Exercise 4: Invalid task number. Please try again.")
            except ValueError:
                print("Exercise 4: Please enter a valid number.")
        elif menu_selection == "5":
            user_task_collection.view_tasks()
            try:
                idx_title = int(input("Enter the task number to change title: "))
- 1
                if 0 <= idx_title < len(user_task_collection.task_collection):
                    new_title_input = input("Enter the new task title: ")

user_task_collection.task_collection[idx_title].change_title(new_title_input)
                else:
                    print("Exercise 4: Invalid task number. Please try again.")
            except ValueError:
                print("Exercise 4: Please enter a valid number.")
        elif menu_selection == "6":
            user_task_collection.view_tasks()
            try:
                idx_due = int(input("Enter the task number to change due date: "))
- 1
                if 0 <= idx_due < len(user_task_collection.task_collection):
```

```
                    new_due_input = input("Enter new due date (YYYY-MM-DD HH:MM):
")
                    try:
                        parsed_new_due = dt.strptime(new_due_input, "%Y-%m-%d
%H:%M")

user_task_collection.task_collection[idx_due].change_due_date(parsed_new_due)
                    except ValueError:
                        print("Exercise 1 (Libraries): Invalid date format. Use
YYYY-MM-DD HH:MM.")
                else:
                    print("Exercise 1 (Libraries): Invalid task number. Please try
again.")
            except ValueError:
                print("Exercise 1 (Libraries): Please enter a valid number.")
        elif menu_selection == "7":
            print("Exercise 1 End: Goodbye!")
            break
        else:
            print("Exercise 1: Invalid choice. Please try again.")
# Exercise 3 End
```

Sample Output:

```
Exercise 1 Start: Creating TaskList
Successfully added exercise number 2: task 'Submit Report'
Successfully added exercise number 2: task 'Clean Garage'
Successfully added exercise number 2: task 'Order Supplies'
Exercise 3: Sample tasks added to the list.

Exercise 1: To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark a task as completed
5. Change task title
6. Change task due date
7. Quit
Enter your choice: 2
Exercise 2: Task List of ALEX:
1. Task: Submit Report, Status: Not Completed, Created: 2025-07-20 14:00, Due:
2025-07-25 12:00
2. Task: Clean Garage, Status: Not Completed, Created: 2025-07-20 14:00, Due:
2025-07-26 15:00
3. Task: Order Supplies, Status: Not Completed, Created: 2025-07-20 14:00, Due:
2025-07-27 10:00
```

Explanation:

This central operation offers a menu-based operation to handle the activities. Users can append, display, delete, carry out and modify tasks interactively. The loop repeats until he or she decides to quit. Input

validation makes the program to deal with errors in a polite manner. This structure illustrates how one should marry user input, control over flow and object-oriented code. The ability to create command-line programs that interact, or which seek input, can be useful on numerous projects. It also makes it clear that user experience as well as error handling is important.

Conclusion:

This lab taught you how to make and run the program to a to-do list with the help of Python classes and methods. You have been training in the use and manipulation of objects, getting user input and constructing interactive programs. Both exercises reminded about the principles of object-oriented programming as well as modular design. With such concepts, you are more equipped to handle bigger projects on using software. Continue the practice and experiment, until you develop more skills and confidence in programming.