School of Computing, Engineering and Physical Sciences

MSc Information Technology

COMP11124 Object Oriented Programming

Semester Report

Zain Rafique

Muhammad Abu bakar

Bilal Ahmad

Mian Sharib

Table of Contents

Week 2

Section 1. Comparisons and Conditionals

Exercise 1: Comparison Operators

Understanding the Task:

This exercise asked us to explore and practice **comparison operators**. We were expected to check how values or variables compare using operators like **==, !=, >, <, >=, and <=.** The main goal was to understand how these comparisons return either True or False.

Source Code:

```
"""Exercise 1"""
# Comparison Operators

# example 1
is_true = False  # A boolean value is directly assigned
print("is_true:", is_true)  # Displays: False

# example 2
is_true = 5 > 4  # Comparison: checks if 5 is greater than 4 (True)
print("5 > 4:", is_true)  # Displays: True

# example 3
a = 5
b = 10

print(a == b)   # Checks if a equals b
print(a != b)   # Checks if a is not equal to b
print(a <= b)   # Checks if a is less than or equal to b
print(a >= b)   # Checks if a is greater than or equal to b
```

Explanation

This exercise introduces **comparison operators** like ==, !=, <, >, <=, >=. In this exercise, I explored how Python compares values. I first set a variable to False just to see how Boolean values work. Then I checked if 5 is greater than 4, and I noticed it returned True because the condition was correct. After that, I created two variables, a and b, with values 5 and 10. I used comparison operators like ==, !=, <=, and >= to compare them. This helped me understand how to check if two values are equal, not equal, or how they relate in terms of size.

The comparisons are:

- a == b: Is a equal to b? (No → False)
- a != b: Is a not equal to b? (Yes → True)
- a <= b: Is a less than or equal to b? (Yes → True)
- a >= b: Is a greater than or equal to b? (No → False)

Output

```
is_true: False
5 > 4: True
False
True
True
False
```

Exercise 2: Logical Operators

Understanding the Task

In this exercise, I was supposed to work with **logical operators**. These include **and**, **or**, and **not**. I needed to check how they behave when used with different conditions and see what

kind of Boolean result they give, either True or False. The goal was to understand how we can combine multiple conditions using logic.

Source Code

```
"""Exercise 2"""
# Logical Operators
# example 1
age = 25
is_in_age_range = age > 20 and age < 30  # Checks if age is between 21 and 29
print("is_in_age_range:", is_in_age_range)  # True because 25 fits the condition

# example 2
x = 5
y = 10

print(x > 0 and y > 0)    # Both x and y are greater than 0 → True
print(x > 0 or y < 0)     # At least one condition is True → True
print(not(x > 0))         # x > 0 is True, but 'not' changes it to False
```

Explanation

Here, I learned how to combine multiple conditions using logical operators. I started by checking if a person's age was between 20 and 30 using the **and** operator. It returned True because both sides of the condition were correct. Then I tested other expressions using and, or, and not with two numbers. I saw that and only returns True if both conditions are true, while or only needs one condition to be true. The **not** operator reversed the result. These examples helped me clearly understand how I can combine or negate conditions when writing logic.

- We check if someone's age is between 21 and 29 by using and. Both sides must be true.

- x and y are numbers. We test combinations of conditions:
    - x > 0 and y > 0: Are both positive? → Yes.
    - x > 0 or y < 0: Is either positive or y negative? → Yes (since x > 0 is True).
    - not(x > 0): This means “is it NOT true that x is greater than 0?” → No, x is 5, so x > 0 is True, and not(True) is False.

Output

```
is_in_age_range: True
True
True
False
```

Exercise 3: if – Conditionals

Understanding the Task

In this task, I explored the basic use of **if statements** in Python. I had to check if a certain condition is true and then update a variable or display a message accordingly. The main idea was to learn how decisions are made in a program based on conditions.

Source Code

```python
"""Exercise 3"""
# If Conditionals
# example 1
age = 19
age_group = "child"
if age > 18:
    age_group = "adult"
    print(f"The age group is {age_group}")  # Will print: The age group is adult

# example 2
age = 13
age_group = "child"
if age > 18:
    age_group = "adult"
    print(f"The age group is {age_group}")  # Condition fails, so nothing is printed

print('\n\n')
```

Explanation

In this part, I used an if statement to check someone's age group. I assumed the person was a child but then used if age > 18 to see if they were actually an adult. If they were, I updated the label and printed it. When I tested it with age 19, it printed "adult". But when I tested with age 13, nothing was printed because the condition wasn't true. I learned that if statements only run their block when the condition is true, otherwise, they skip it.

Output

```
The age group is adult
```

Exercise 4: if – else Conditionals

Understanding the Task

This task was about using the **if-else** structure. I had to write code where the program chooses between two possible actions, one if a condition is true, and another if it's false.

Source Code

```python
"""Exercise 4"""
# If-else Conditionals
# example 1
wind_speed = 30
if wind_speed < 10:
    print("It is a calm day")  # Not true → skipped
else:
    print("It is a windy day")  # Executed → prints this

# example 2
wind_speed = 5
if wind_speed < 10:
    print("It is a calm day")  # True → this line prints
else:
    print("It is a windy day")  # Skipped

print('\n\n')  # Adds space in output
```

Explanation

- In the first example, wind_speed was 30, which is not less than 10, so it printed "It is a windy day".
- In the second example, windspeed was 5, which is less than 10, so it printed "It is a calm day".

Output

```
It is a windy day
It is a calm day
```

Exercise 5: if – elif - else Conditionals

Understanding the Task

In this exercise, I practiced using **if-Elif-else** blocks to handle multiple conditions in a clean way. This helped in writing better decision-based logic where the program chooses the right option from several possibilities.

Source Code

Example 1

```
# example 1
grade = 55
if grade < 50:
    print("You failed")
elif grade < 60:
    print("You passed")  # True, so this prints
elif grade < 70:
    print("You got a good pass")
else:
    print("You got an excellent pass")
```

Example 2

```
# example 2
grade = 40
if grade < 50:
    print("You failed")  # This one matches → prints
elif grade < 60:
    print("You passed")
elif grade < 70:
    print("You got a good pass")
else:
    print("You got an excellent pass")
```

Example 3

```
# example 3
grade = 65
if grade < 50:
    print("You failed")
elif grade < 60:
    print("You passed")
elif grade < 70:
    print("You got a good pass")
else:
    print("You got an excellent pass")
```

Example 4

```
# example 4
grade = 80
if grade < 50:
    print("You failed")
elif grade < 60:
    print("You passed")
elif grade < 70:
    print("You got a good pass")
else:
    print("You got an excellent pass")
```

Explanation

- Four different grades were tested.
- The code checked each grade and printed a message:

- Below 50 → “You failed”
- Between 50–59 → “You passed”
- Between 60–69 → “You got a good pass”
- 70 and above → “You got an excellent pass”

- Each elif allows checking in sequence, and only one block runs depending on the value.

Output

```
You passed
You failed
You got a good pass
You got an excellent pass
```

Exercise 6: Compare Temperatures

Understanding the Task

This task was about comparing two temperature values to see if they are the same or different. I had to store two temperature values in separate variables and then use an if-else condition to compare them. If both temperatures were equal, the program should display a message saying they are the same. Otherwise, it should state that the temperatures are different.

The goal of this task was to help me practice using comparison operators (==) along with conditional statements to make decisions based on real-world values. It also gave me a chance to reinforce how to format clear and meaningful output messages based on the result of the comparison.

Source Code

```
"""Exercise 6"""

# Task: Compare Temperatures
print("Task: Compare Temperatures\n")

# Just setting two temperature values
temperature1 = 25
temperature2 = 30

# Showing what values we're comparing
print(f"Temperature 1: {temperature1}°C")
print(f"Temperature 2: {temperature2}°C")

# Checking if both are the same or different
if temperature1 == temperature2:
    print("Result: Both temperatures are the same.")
else:
    print("Result: The temperatures are different.")
```

Explanation

- The two temperatures were set to 25 and 30.
- The program displayed both values.
- Then it compared them: since 25 ≠ 30, it printed "The temperatures are different".

Output

```
Task: Compare Temperatures

Temperature 1: 25°C
Temperature 2: 30°C
Result: The temperatures are different.
```

Section 2. Python Lists

Understanding the Task

I was asked to practice how Python lists work. Specifically:

- I had to **create a list** of cities.
- Then I **accessed specific elements** in the list using indexes and slicing.
- After that, I **modified the list** by changing one value and adding another.
- Finally, I had to **create, access, and manipulate another list** of colors, check its length, and use conditions and slicing to work with it.

These tasks helped me understand how flexible and powerful Python lists are.

Exercise 1: Creating a list

- I created different types of lists to explore how Python handles numbers, text, empty lists, and mixed types.
- I also created a list named city_list with "Glasgow", "London", and "Edinburgh" as requested.

```
"""Exercise 1"""
# Creating a List
integer_list = [1, 2, 3, 4, 5]

# list of strings
string_list = ["apple", "banana", "orange", "grape"]

# empty list
empty_list = []

# list with different data types
list_with_different_types = [1, "two", 3.0, True]

person_1_age = 20
person_2_age = 30
# creating a list based on variables
age_list = [person_1_age, person_2_age]

# List within a List
list_within_a_list = [["red", "green", "blue"], ["yellow", "orange", "purple"]]

# Task: Creating a List
city_list = ["Glasgow", "London", "Edinburgh"]
```

Exercise 2: Accessing List Elements

- I learned that I could get a specific item by its position using an index.
- I used slicing to get a portion of the list.
- I also printed the **third item** and the **last two items** from city_list using slicing.

```
"""Exercise 2"""
# Accessing a List

# using the string list from exercise 1

# indices start from 0, so the second element is at index 1
second_item = string_list[1]

# Slicing

# this slice from 0 to 2 gives the first two elements of index 0 and 1
print(string_list[0:2])

# using negative indices
last_item = string_list[-1]  # last item is at index -1

# Task

# Third item
print(city_list[2])
# Last two items using slicing
print(city_list[1:])
```

Exercise 3: Modifying Lists

- I replaced the first item in string_list from "apple" to "pear".
- I added a new item to the end of the list using .append().
- I also updated the city_list by adding "Manchester" and changing "London" to "Birmingham".

```
"""Exercise 3"""
# Modifying a List

# changing the first item (replacement)
string_list[0] = "pear"

# adding an item to the end of the list
string_list.append("orange")



# Task

# Adding an item to list - at the end
city_list.append("Manchester")

# Modifying an item at index 1 - second element
city_list[1] = "Birmingham"
```

Exercise 4: Summary Task

I created a list called colours and printed it.

```
"""Exercise 4"""

# Task: Create, Access and Modify Lists
print("Task: Create, Access and Modify Lists\n")

# Creating a list of colours
colours = ["blue", "green", "yellow"]

# Showing the full list
print("All colours:", colours)
```

I accessed the second color and printed it.

```
# Printing the second item in the list
print("Second colour:", colours[1])
```

I changed the first color, counted how many colors were in the list using len(), and printed that.

```
# Changing the first colour in the list
colours[0] = "purple"

# Showing the updated list
print("Modified colours:", colours)

# Printing how many items are in the list
print("Number of colours in the list:", len(colours))
```

I checked whether "red" was in the list and printed a message accordingly.

```
# Checking if 'red' is in the list
if "red" in colours:
    print("Colour 'Red' is present in the list")
else:
    print("Colour 'Red' is not present in the list")
```

I used slicing to make a new list with just the second and third colors.

```
# Making a new list from 2nd and 3rd colours
selected_colours = colours[1:3]

# Showing the new small list
print("Selected colours:", selected_colours)
```

Output

```
['apple', 'banana']
Edinburgh
['London', 'Edinburgh']


Task: Create, Access and Modify Lists

All colours: ['blue', 'green', 'yellow']
Second colour: green
Modified colours: ['purple', 'green', 'yellow']
Number of colours in the list: 3
Colour 'Red' is not present in the list
Selected colours: ['green', 'yellow']
```

Section 2. Python Loops

Understanding the Task

In this exercise, I practiced how to use loops in Python to repeat tasks automatically. I worked with both while loops and for loops. I also learned how to control the flow of loops using special keywords like break (to stop the loop early) and continue (to skip part of a loop). After that, I applied my learning to some mini-projects like printing even numbers, summing up squares of numbers, and creating a countdown. These tasks helped me build a strong foundation in writing repetitive logic efficiently

Exercise 1: While Loop

Source Code

```python
"""Exercise 1"""
# While Loop
i = 0
while i < 5:
    print(i)
    i += 1
```

- I used a while loop to print numbers from 0 to 4.

- The loop kept running as long as I was less than 5.
- Each time, I printed i and increased it by 1.

Output

```
0
1
2
3
4
```

Exercise 2: For Loop

Source Code

Example 1

```
"""Exercise 2"""
# For Loop
for fruit in string_list:
    print(fruit)
```

Example 2

```
# Task
# loop for city list
for city in city_list:
    print(city)
```

- I used a for loop to print items from two different lists: string_list and city_list.
- It went through each element one by one and printed them

Output 1

```
pear
banana
orange
grape
orange
```

Output2

```
Glasgow
Birmingham
Edinburgh
Manchester
```

Exercise 3: Loop Keywords

```python
"""Exercise 3"""
# Loop Keywords: Range, break and continue

# range()
# function to generate a series of numbers to use in a for loop
# or want to perform a task a certain number of times
for number in range(1, 6):
    print(number)

# break
# it will stop the loop from running any more iterations
for i in range(5):
    if i == 2:
        break
    print(i)

# Task: print the numbers 0 through 4, but stop the loop when i is equal to 3
for i in range(5):
    if i == 3:
        break
    print(i)

# Continue
# it will skip the current iteration and continue to the next iteration
for i in range(5):
    if i == 2:
        continue
    print(i)

print('\n\n')
```

- I used range(1, 6) to print numbers from 1 to 5.
- Then I used break to stop the loop early when i became 2.
- I also wrote a custom task to stop when i == 3.

- Lastly, I used continue to skip printing the number 2 but printed everything else.

Output

Range ()

break

print the numbers 0 through 4, but stop the loop when i is equal to 3

continue

Exercise 4: Summary Tasks

Even Numbers

Source Code

```
# Task: Print Even Numbers from a List
print("Task: Print Even Numbers from a List\n")

# A simple list of numbers from 1 to 10
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Loop through the list and print only even numbers
print("Even numbers in the list:")
for num in numbers:
    if num % 2 == 0:
        print(num)
```

- I created a list from 1 to 10.
- I used the modulo operator % to check if a number was even.
- If yes, I printed it.

Output

```
0
1
2
3
4
```

Sum of Squares

Source Code

```
# Task: Sum of Squares
print("Task: Sum of Squares\n")

# Starting the total from 0
sum_of_squares = 0

# Going from 1 to 5 and adding each number squared
for number in range(1, 6):
    sum_of_squares += number * number

# Showing the final result
print("The sum of squares from 1 to 5 is:", sum_of_squares)

print('\n\n')
```

- I used a loop to go from 1 to 5.
- I squared each number and added it to a total.
- I printed the final sum, which came out as 55.

Output

```
Task: Sum of Squares


The sum of squares from 1 to 5 is: 55
```

Countdown

Source Code

```
# Task: Countdown from 10 to 1 using while loop
print("Task: Countdown from 10 to 1 using while loop\n")

# Start countdown at 10
countdown = 10

# Keep counting down till 1
while countdown >= 1:
    print(countdown)
    countdown -= 1

# Once done, show this message
print("Liftoff!")
```

- I used a while loop that started from 10.
- It printed each number down to 1.
- Then, it printed “Liftoff!” to end the countdown.

Output

```
Task: Countdown from 10 to 1 using while loop

10
9
8
7
6
5
4
3
2
1
Liftoff!
```

Section 4. Obtaining User Input

Source Code

```
# input()
# function is used to take input from the user

# Ask the user to type anything. This value is stored as a string in 'user_input'.
user_input = input("Enter something: ")

# Show the exact thing the user typed.
print("You entered:", user_input)

# ------------------------------
# Example - entering a number
# ------------------------------

# Ask the user how old they are (input comes in as text)
age = input("How old are you? ")

# Convert the string input to an integer so I can do math with it
age = int(age)

# Add 1 to the age and print what the user's age will be next year
next_year_age = age + 1
print("Next year, you'll be", next_year_age, "years old.")
```

Explanation

- I used input() to ask the user to type something.
- Whatever they typed got stored in a variable and printed back to them.
- This helped me see how input works in real time.
- I asked the user for their age.
- Since input is always a string by default, I used int() to convert it into a number.
- Then, I added 1 to it and printed how old they would be next year.
- This showed me how to turn text into numbers and use it in calculations.

Output

```
Enter something: Hello!
You entered: Hello!

How old are you? 25
Next year, you'll be 26 years old.
```

Task: User Input and Conditional Statements

Understanding the Task

In this section, I learned how to get input from the user using the input() function. I practiced asking the user to type something, storing that input in a variable, and then doing something with it, like printing it or using it in a condition. I also learned how to convert string input into numbers so that I can do math with it. Finally, I created a small program to check someone's age group based on what they enter.

Source Code

```python
print("Task: User Input and Conditionals\n")

# Ask the user to enter their age again
age = int(input("Enter your age: "))

# Use conditions to determine their age group
if age < 18:
    print("You are a minor.")  # If they're under 18
elif age <= 65:
    print("You are an adult.")  # Between 18 and 65
else:
    print("You are a senior citizen.")  # Over 65
```

Explanation

- I asked the user to enter their age again.
- Based on the value, I used conditions to print whether the person is a minor, adult, or senior citizen.
- This helped me combine input with decision-making logic.

Output

```
Task: User Input and Conditionals

Enter your age: 25
You are an adult.
```

Task: Temperature Converter

Understanding the Task

- I ask the user which temperature unit they want to convert from (C, F, or K).
- If the input is valid, I ask for the temperature value.
- I then convert the value into the other two temperature scales using the correct formula:
    - **C to F and K**
    - **F to C and K**
    - **K to C and F**
- If the user enters an invalid unit, the program tells them and asks again.
- Once the conversion is done, the program ends.

Source Code

```
# Task: Temperature Converter with user input and options
print("Task: Temperature Converter with user input and options\n")

while True:
    # Ask the user what type of temperature they want to convert
    choice = input("Enter the temperature unit you want to convert from (C, F, or K): ").upper()

    # If the choice is valid, go ahead
    if choice in ["C", "F", "K"]:
        # Ask for the temperature value
        temp = float(input("Enter the temperature value: "))

        # Convert based on the unit entered
        if choice == "C":
            fahrenheit = (temp * 9/5) + 32
            kelvin = temp + 273.15
            print(f"\n{temp}°C is equal to {fahrenheit:.2f}°F and {kelvin:.2f}K.")

        elif choice == "F":
            celsius = (temp - 32) * 5/9
            kelvin = celsius + 273.15
            print(f"\n{temp}°F is equal to {celsius:.2f}°C and {kelvin:.2f}K.")
```

```
        elif choice == "K":
            celsius = temp - 273.15
            fahrenheit = (celsius * 9/5) + 32
            print(f"\n{temp}K is equal to {celsius:.2f}°C and {fahrenheit:.2f}°F.")

        # Break the loop once conversion is done
        break

    else:
        # Tell the user to enter a valid unit
        print("\nInvalid input. Please enter C, F, or K.\n")
```

Explanation

In this temperature converter program, I asked the user which unit they want to convert from, Celsius, Fahrenheit, or Kelvin. Once the user chose a valid option, I then asked for the temperature value.

Based on the user's input, I used simple math formulas to convert the temperature into the other two units. For example, if the input was in Celsius, I converted it to Fahrenheit and Kelvin.

I used a loop to keep asking until the user entered a valid unit, and conditionals to decide which conversion formula to use. At the end, the converted results were displayed in a clean format.

This program helped me practice getting user input, working with loops, using conditionals, and performing basic calculations.

Output

Case 1: User Enters C

```
Enter the temperature unit you want to convert from (C, F, or K): C
Enter the temperature value: 25

25.0°C is equal to 77.00°F and 298.15K.
```

Case 2: User Enters Invalid Input First, then Valid

```
Enter the temperature unit you want to convert from (C, F, or K): X

Invalid input. Please enter C, F, or K.

Enter the temperature unit you want to convert from (C, F, or K): K
Enter the temperature value: 300

300.0K is equal to 26.85°C and 80.33°F.
```

Week 3

Section 1. Functions and Scope

Exercise 1: Functions in Python

Understanding the Task

In this task, I learned how functions work in Python and how they help make code more reusable and organized. I explored different aspects like how to define a basic function, pass parameters to it, and use keyword arguments. I also practiced using default parameter values and learned how to return results from a function. Each small sub-topic helped me understand how functions behave in different scenarios and how we can control the inputs and outputs more efficiently. This exercise gave me a solid understanding of writing clean, functional code.

Creating Functions

Source Code

```
def greet_user():
    print("Hello!")
# calling function
greet_user()
```

Explanation

Here, I created a simple function called greet_user() that prints a greeting. I called the function after defining it to run the print statement. This shows how basic functions are defined and called in Python.

Output

```
Hello!
```

Function Parameters

Source Code 1

```
def greet_user(name):
    print(f"Hello {name}!")
# calling function
greet_user("John")
```

Output 1

```
Hello John!
```

Source Code 2

```
# functions with more than one parameter
def greet_user(first_name, last_name):
    print(f"Hello {first_name} {last_name}!")

# calling function
greet_user("John", "Doe")
```

Output 2

```
Hello John Doe!
```

Explanation

This version of the function takes a parameter called name. When I call greet_user("John"), it prints "Hello John!". This shows how to pass input (variables) into a function and use it inside.

Keyword Arguments

Source Code

```
# keyword arguments
greet_user(last_name="Smith", first_name="John")
```

Explanation

In this example, I called the same function using keyword arguments. I passed values by naming the parameters directly. This allows the arguments to be passed on in any order, which makes the code more readable.

Output

```
Hello John Smith!
```

Default Values

Source Code

```
# Default values
def greet_user(first_name, last_name, university="UWS"):
    print(f"Hello {first_name} {last_name} from {university}!")

# calling function
greet_user("John", "Doe")

"""it will also work when we pass the value
of variable which have default value"""
      You, 1 second ago • Uncommitted changes
greet_user("John", "Smith", "UWS London")
```

Explanation

This function has a default value for the university parameter. If I don't pass the value for it, it uses "UWS" by default. But I can also override it by giving a custom value like "UWS London". This is useful when some arguments usually have a common value.

Output

```
Hello John Doe from UWS!
Hello John Smith from UWS London!
```

Return

Source Code

```
# Returning values from functions
def add_numbers(num1, num2):
    return num1 + num2

def add_numbers(num1, num2):
    result = num1 + num2
    return result

# calling function
result = add_numbers(5, 3)
print(f"The sum of 5 and 3 is: {result}")
```

Output

```
The sum of 5 and 3 is: 8
```

Source Code

```
# returning multiple values
def add_and_multiply_numbers(num1, num2):
    return num1 + num2, num1 * num2

def add_and_multiply_numbers(num1, num2):
    sum = num1 + num2
    product = num1 * num2
    return sum, product

# calling function and getting multiple values
result_sum, result_product = add_and_multiply_numbers(5, 3)
print(f"The sum of 5 and 3 is: {result_sum}")
print(f"The product of 5 and 3 is: {result_product}")
```

Output

```
The sum of 5 and 3 is: 8
The product of 5 and 3 is: 15
```

Explanation

In this part, I created a function called add_numbers() that takes two numbers and returns their sum. Instead of printing the result inside the function, it sends the result back using the return keyword. I stored that value in a variable called result and printed it. This makes the function more flexible since I can use the result anywhere else in the program too.

Task: Greet each friend in the list

Understanding the Task

In this task, I had to create a function that takes a list of friends' names and greets each one by printing a message. The main goal was to use a for loop to go through a list and apply the same action (printing a greeting) to every item.

Source Code

```
# Task: Greet each friend in the list
print("Task: Greet each friend in the list\n")

def greet_friends(friend_list):
    for name in friend_list:
        print(f"Hello {name}!")

# Example list of names
friends = ["John", "Jane", "Jack"]

# Calling the function
greet_friends(friends)
```

Explanation

I defined a function greet_friends() that accepts one argument, a list of names. Inside the function, I used a for loop to iterate through the list and print "Hello" followed by each friend's name. When I called the function with a sample list like ["John", "Jane", "Jack"], it printed a greeting for each one. This is a good example of using functions and loops together.

Output

```
Task: Greet each friend in the list

Hello John!
Hello Jane!
Hello Jack!
```

Task: Calculate Tax Based on Income and Tax Rate

Understanding the Task

In this task, I had to write a program that calculates tax based on a given income and tax rate. It starts by testing the function with fixed values, and then allows the user to enter their own income and tax rate multiple times. The goal was to use functions, input/output, loops, and basic arithmetic.

Source Code

```python
# Task: Calculate Tax Based on Income and Tax Rate
print("Task: Calculate Tax Based on Income and Tax Rate\n")

# Step 1: Define the tax calculation function
def calculate_tax(income, tax_rate):
    tax = income * tax_rate
    return tax

# Step 2: First example with 50,000 and 20% tax rate
result = calculate_tax(50000, 0.2)
print(f"The tax on £50000 at a 20% rate is: £{result}")
print("-" * 40)
```

```python
# Step 3: Ask the user if they want to calculate more taxes
while True:
    # loop is to do the same thing multiple times
    choice = input("Do you want to calculate more tax? (y/n): ").lower()

    if choice == "y":
        # Take income and tax rate from the user
        income = float(input("Enter the income in £: "))
        tax_rate = float(input("Enter the tax rate (e.g. 0.2 for 20%): "))

        # Step 4: Call the tax calculation function
        # Calculate and print the tax
        tax = calculate_tax(income, tax_rate)
        print(f"The tax on £{income} at a {tax_rate * 100}% rate is: £{tax}")
        print("-" * 40)

    elif choice == "n":
        print("Thank you! Program ended.")
        break

    else:
        print("Invalid input. Please type 'y' or 'n'.")
```

Explanation

- First, I created a function called calculate_tax that multiplies income by tax rate and returns the result.
- I tested the function with £50,000 income and a 20% tax rate, and it correctly returned the tax.
- Then I used a while loop to allow the user to calculate tax as many times as they want.
- If the user enters 'y', it asks for income and tax rate, then prints the calculated tax using the same function.
- If the user types 'n', the program ends with a thank-you message.
- It also handles invalid inputs like any character other than 'y' or 'n'.

Output

```
Task: Calculate Tax Based on Income and Tax Rate

The tax on £50000 at a 20% rate is: £10000.0
----------------------------------------
Do you want to calculate more tax? (y/n): y
Enter the income in £: 340023
Enter the tax rate (e.g. 0.2 for 20%): 0.8
The tax on £340023.0 at a 80.0% rate is: £272018.4
----------------------------------------
Do you want to calculate more tax? (y/n): n
Thank you! Program ended.
```

Task: Compound Interest Calculator Function

Understanding the Task

In this task, I had to write a function called compound_interest() that calculates how an investment grows each year using compound interest. The function takes the initial amount, number of years, and interest rate as input. It prints the total value for each year and returns the final amount as an integer. I also had to include checks to make sure the inputs are valid, like ensuring the interest rate is between 0 and 1, and the duration is positive. This task helped me practice using functions, loops, and input validation.

Source Code

```
# Task: Compound Interest Calculator Function
print("Task: Compound Interest Calculator Function\n")

# function have three parameters
def compound_interest(principal, duration, interest_rate):
    # Check if interest_rate is valid
    if interest_rate < 0 or interest_rate > 1:
        print("Please enter a decimal number between 0 and 1")
        return None

    # Check if duration is valid
    if duration < 0:
        print("Please enter a positive number of years")
        return None

    # Loop through each year and calculate compound interest
    for year in range(1, duration + 1):
        total_for_the_year = principal * (1 + interest_rate) ** year
        print(f"The total amount of money earned by the ",
              "investment in year {year} is {total_for_the_year:.2f} £")

    # Return final value as an integer
    final_value = principal * (1 + interest_rate) ** duration
    return int(final_value)

# Example test
final_result = compound_interest(1000, 5, 0.03)
print(f"\nFinal investment value after 5 years: {final_result:.2f} £")
print("-" * 40)
print('Using Assertions')
assert compound_interest(1000, 5, 0.03) == 1159
      You, 2 days ago • first commit …
```

Explanation

- The function compound_interest takes three parameters: principal, duration, and interest_rate.
- It first checks if the interest rate is between 0 and 1 and if the duration is positive. If not, it shows an error and exits early.
- Then, it uses a for loop to calculate and print the total amount at the end of each year using the compound interest formula: (principal×(1+rate)year)(principal × (1 + rate) ^ year)(principal×(1+rate)year)

- Finally, it returns the total value at the end of the given duration, rounded to an integer.
- I tested the function with a £1000 investment for 5 years at 3% interest, and it correctly printed the values for each year and returned the final result.

Output

```
Task: Compound Interest Calculator Function

The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £

Final investment value after 5 years: 1159.00 £
---------------------------------------
Using Assertions
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
```

Exercise 2: Variable Scope

Understanding the Task

This task helped me understand how **variable scope** works in Python, especially in relation to functions. It showed the difference between variables defined inside a function (local scope) and those outside it (global scope).

Source Code

```
"""Exercise 2"""
# Variable Scope

def new_function():
    my_new_variable = 5

new_function() # call the function. No problems here.
"""
this will cause an error because this variable is
defined inside the function and can only be
accessed and used inside the function
"""
# print(my_new_variable)

"""variables defined outside the function
can be accessed inside the function too """

def new_function():
    my_new_variable = 5
    print(my_new_variable)

new_function()
```

Explanation

- In the first part, I defined a variable called my_new_variable **inside** a function and then tried to access it **outside** the function. This causes an error because the variable only exists within the function's local scope.
- In the second part, I added a print() statement **inside** the function to access the same variable, and it worked perfectly. This proves that local variables can only be used within the function they're defined in.
- The code also notes that **global variables** (defined outside the function) can still be accessed inside it, though this specific example doesn't show that part in action.

Output

Error

```
Traceback (most recent call last):
  File "d:\C data\Desktop\latest\py asgmt\lab_week_3.py", line 204, in <module>
    print(my_new_variable)
          ^^^^^^^^^^^^^^^
NameError: name 'my_new_variable' is not defined
```

Corrected Code Output

```
5
```

Section 2: Assertions and Errors

Exercise 6: Assertions

Understanding the Task

This task was about using assert statements in Python to automatically check if a function gives the expected result. Assertions are used mainly for testing.

Source Code

```
"""Exercise 6"""
# Assertions
# Example test for compound interest function that was in section 1 Task
assert compound_interest(1000, 5, 0.03) == 1159
```

Explanation

I used the assert keyword to test the output of the compound_interest function. If the result is not exactly 1159, the program will raise an error. Since the actual return value matches, the code runs without any issues.

Output

```
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
The total amount of money earned by the  investment in year {year} is {total_for_the_year:.2f} £
```

Exercise 7: Identifying and Fixing Common Errors

Understanding the Task

Syntax Error

Occurs when Python code is written incorrectly and doesn't follow the proper rules, e.g., a typo like pritn() instead of print().

Source Code 1 – Has Error

```
# Syntax Error
pritn("Hello, World!")
```

Output 1

```
Traceback (most recent call last):
  File "d:\C data\Desktop\latest\py asgmt\lab_week_3.py", line 237, in <module>
    pritn("Hello, World!")
    ^^^^^
```

Source Code – Corrected

```
# Corrected Code
print("Hello, World!")
```

Output 2

```
Hello, World!
```

Explanation

The original line used pritn() which is incorrect. I fixed it by writing print("Hello, World!") correctly.

Name Error

Happens when you try to use a variable or function name that hasn't been defined yet.

Source Code 1 - Has Error

```
# Name Error:
my_name = "Alice"
print("Hello, " + myname)
```

Output 1

```
Traceback (most recent call last):
  File "d:\C data\Desktop\latest\py asgmt\lab_week_3.py", line 247, in <module>
    print("Hello, " + myname)
                      ^^^^^^
NameError: name 'myname' is not defined. Did you mean: 'my_name'?
```

Source Code 2 – Corrected Code

```
# Corrected Code:
# define the variable correctly and then using it
favorite_color = "Blue"
print("My favorite color is", favorite_color)
```

Output 2

```
My favorite color is Blue
```

Explanation

Originally, myname was used without being defined. I fixed it by using a properly declared variable favorite_color and printed it correctly.

Value Error

Occurs when a function gets the right type of data but the value is not acceptable, like converting "abc" to an integer.

Source Code 1 – Has Error

```
# Value Error:
number1 = "5"
number2 = 3
result = number1 + number2
```

Output 1

```
Traceback (most recent call last):
  File "d:\C data\Desktop\latest\py asgmt\lab_week_3.py", line 260, in <module>
    result = number1 + number2
             ~~~~~~~~^~~~~~~~~
TypeError: can only concatenate str (not "int") to str
```

Source Code 2 – Corrected Code

```
# Corrected Code:
# define the variable correctly and then using it
favorite_color = "Blue"
print("My favorite color is", favorite_color)
```

Output 2

```
The sum of 5 and 3 is 8
```

Explanation

Python can’t add a string and an integer directly. I fixed it by converting "5" to int using int("5") before adding.

Index Error

Happens when you try to access an index in a list that doesn’t exist, like accessing index 3 in a 3-item list.

Source Code 1 – Has Error

```
# Index Error
fruits = ["apple", "banana", "orange"]
print(fruits[3])
```

Output 1

```
Traceback (most recent call last):
  File "d:\C data\Desktop\latest\py asgmt\lab_week_3.py", line 275, in <module>
    print(fruits[3])
          ~~~~~~^^^
IndexError: list index out of range
```

Source Code 2 – Corrected Code

```
# Corrected Code:
# Use a valid index for the list
# Index starts from 0, thats why end at 1 less then the length of the list
fruits = ["apple", "banana", "orange"]
print(fruits[2])
```

Output 2

```
Fruit at the index 2 is orange
```

Explanation

The list only had 3 elements (index 0 to 2), but index 3 was used. I corrected it by accessing index 2, which is valid.

Output

Indentation Error

Occurs when the code isn't properly spaced. Python uses indentation to know what code belongs in loops, functions, etc.

Source Code 1 – Has Error

```
if 5 > 2:
print("Five is greater than two!")
```

Output 1

```
  File "d:\C data\Desktop\latest\py asgmt\lab_week_3.py", line 289
    print("Five is greater than two!")
    ^
IndentationError: expected an indented block after 'if' statement on line 288
```

Source Code 2 – Corrected Code

```
# Corrected Code:
# Indentation is correct
if 5 > 2:
    print("Five is greater than two!")
```

Output 2

```
Five is greater than two!
```

Explanation

Python expects code blocks to be indented. The original code wasn't indented under if. I fixed it by indenting the print() line properly.

Section 3. Larger scale python program

Task: To-Do list manager:

Understanding the Task

The purpose of this task was to create a simple **To-Do List application** using basic Python features like lists, functions, conditionals, and loops. The program should allow the user to manage their daily tasks by adding them, viewing the current list, removing any task, and exiting the program using a menu system.

The focus was on writing clean, functional code and understanding how to work with user input and list operations in Python.

Source Code

```python

"""
This file container code for
To do List Manager application from the Week 3 Lab
"""

print("="*40)

# Step 1: Initialize an empty list to store tasks
tasks = []

# Step 2: Function to add a task
def add_task():
    task = input("Enter the task you want to add: ")
    # add the task to the list
    tasks.append(task)
    print(f"'{task}' has been added to your to-do list.\n")

# Step 3: Function to view current tasks
def view_tasks():
    if not tasks:
        print("Your to-do list is empty.\n")
    else:
        print("\nHere are your current tasks:")
        for index, task in enumerate(tasks, start=1):
            print(f"{index}. {task}")
        print()  # just a blank line for spacing

```

```python
# Step 4: Function to remove a task
def remove_task():
    if not tasks:
        print("There are no tasks to remove.\n")
        return      You, 2 days ago • first commit

    # Show current tasks so user knows the numbers
    view_tasks()

    try:
        task_number = int(input("Enter the number of the task you want to remove: "))
        if 1 <= task_number <= len(tasks):
            # remove the selected task
            removed = tasks.pop(task_number - 1)
            print(f"'{removed}' has been removed from your to-do list.\n")
        else:
            print("Invalid task number. Please try again.\n")
    except ValueError:
        print("Please enter a valid number.\n")
```

```python
# Step 5: Main program loop
while True:
    print("📋 To-Do List Manager")
    print("1. Add a task")
    print("2. View tasks")
    print("3. Remove a task")
    print("4. Quit\n")

    choice = input("Enter your choice (1-4): ")

    # Handle each menu option
    if choice == "1":
        add_task()
    elif choice == "2":
        view_tasks()
    elif choice == "3":
        remove_task()
    elif choice == "4":
        print("Goodbye! Your to-do list has been closed.")
        # exit the loop and program
        break
    else:
        print("Invalid choice. Please enter a number between 1 and 4.\n")

print("="*40)
```

Explanation

The program runs in a loop and offers four main options to the user:

1. Initialize the Task List

At the top of the program, an empty list called tasks is created. This is where all the tasks entered by the user are stored.

2. Adding a Task

The add_task() function asks the user to type in a task. Once entered, the task is added to the list, and a confirmation message is displayed.

3. Viewing All Tasks

The view_tasks() function checks if the list is empty. If not, it shows all current tasks with numbers beside them for easy reference. This helps the user see what tasks they've added so far.

4. Removing a Task

The remove_task() function lets the user delete a task by entering its number. The task list is displayed first so the user knows the correct number. The function also handles invalid input or if the list is empty.

5. Menu and Loop

The program uses a while loop to repeatedly show the menu:

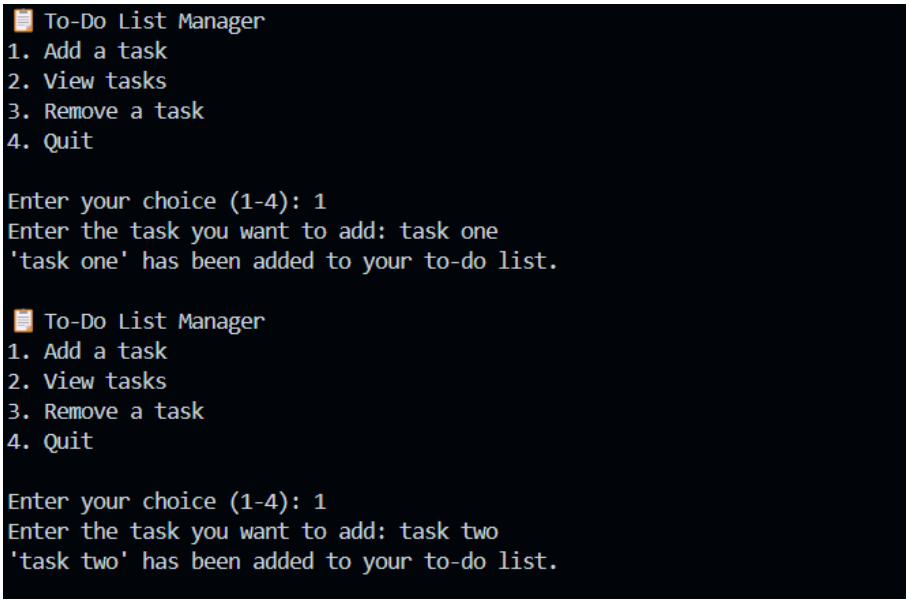
- 1 to Add a task
- 2 to View tasks
- 3 to Remove a task
- 4 to Quit

The user can perform any action, and the loop keeps running until the user selects the quit option.

Output

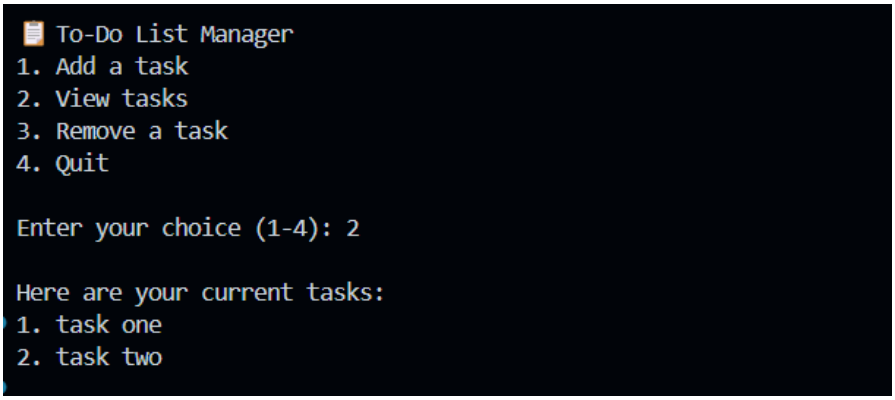
Choice 1

Choosing 1 to add a task (one by one)



Choice 2

Choosing 2 to view tasks.



Choice 3

Choosing 3 to delete a task.

```
Enter your choice (1-4): 3

Here are your current tasks:
1. task one
2. task two

Enter the number of the task you want to remove: 1
'task one' has been removed from your to-do list.

📋 To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Quit

Enter your choice (1-4): 2

Here are your current tasks:
1. task two
```

Choice 4

Quitting the program

```
📋 To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Quit

Enter your choice (1-4): 4
Goodbye! Your to-do list has been closed.
```

Week 4

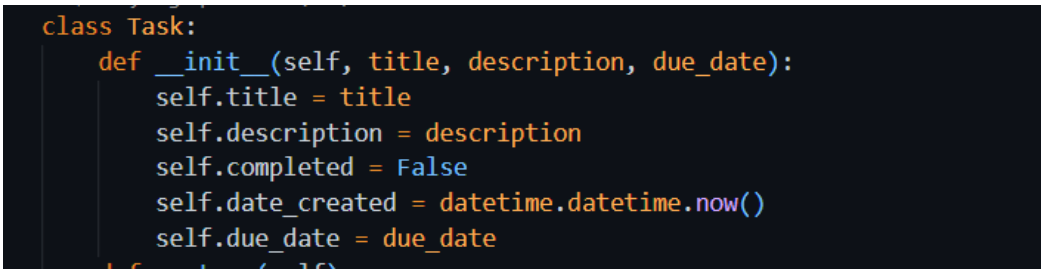
Section 1. Python Classes

Exercise 1: Creating Classes and Initializing Objects

Understanding the Task

In this exercise, I was asked to create Task class which will have task details. I was also asked to define a class called TaskList that holds a list of tasks which are the objects of Tasks from the Task class and stores the owner's name. I learned how to use the __init__ method to initialize class attributes.

Source Code



```
class Task:
    def __init__(self, title, description, due_date):
        self.title = title
        self.description = description
        self.completed = False
        self.date_created = datetime.datetime.now()
        self.due_date = due_date
```

Explanation

This class is used to represent a single task in a to-do list. When a new Task object is created, it automatically stores:

- **title**: The name or heading of the task
- **description**: A short explanation about what the task is

- **completed**: A boolean value that shows whether the task is done (initially set to False)
- **date_created**: The current date and time when the task is created
- **due_date**: The deadline for the task

This helps organize all the important details of one task inside a single object.

Source Code

```
class TaskList:
    # tasks = list[Task]
    def __init__(self, owner):
        self.owner = owner
        self.tasks = []
```

Explanation

This class is used to manage a list of tasks for one user.

- **owner**: Stores the name of the person who owns the task list
- **tasks**: An empty list that will hold multiple Task objects

This class acts as a container for managing multiple tasks under one user.

Object Creation

For Task Class

```
task = Task(task_title, task_description, due_date)
```

And for Task List class

```
name = input("Enter your name: ")

task_list = TaskList(name)
```

Exercise 2: Adding Methods

Understanding the Task

In this task, I was asked to expand the Task class by adding useful methods that allow interacting with task data. The goal was to practice writing instance methods for updating task attributes, such as marking it as complete, changing the title, or changing the due date. This helped me understand how to define custom behaviors inside a class.

Source Code

```
class Task:
    def __init__(self, title, description, due_date):
        self.title = title
        self.description = description
        self.completed = False
        self.date_created = datetime.datetime.now()
        self.due_date = due_date
    def mark_completed(self):
        print('Marking Task Completed')
    def change_title(self, new_title):
        print('Changing Title of task')
    def change_due_date(self, new_date):
        print('Change Due Date of Task')
```

Explanation

mark_completed() Method

- This method is intended to mark the task as completed.
- Currently, it just prints a message, but in a full version, it would update the completed status to True.

change_title(new_title) Method

- This method is designed to change the title of the task.
- It prints a message as a placeholder, but normally it would update the self.title.

change_due_date(new_date) Method

- This is meant to update the due date of the task.
- Right now, it prints a confirmation message, but ideally it would modify self.due_date.

```
class TaskList:
    # tasks = list[Task]
    def __init__(self, owner):
        self.owner = owner
        # self.owner = ""
        self.tasks = []
    def add_task(self, task:Task):
        # Add a task to the list
        self.tasks.append(task)
    def remove_task(self,index):
        print("remove task")
    def view_tasks(self):
        print("view tasks")
```

add_task(self, task: Task):

Adds a new **Task** object to the tasks list using the **append()** method.

remove_task(self, index):

This function is meant to remove a task using a user-provided index. (del self.tasks[index - 1]) is used to delete task from the list.

view_tasks(self):

Displays all tasks in the list. If the list is empty, it shows a message saying there are no tasks. Otherwise, it loops through the list and prints each task with a number.

This structure shows how methods can be added to a class to make it more dynamic and interactive.

Task: Add logic to methods defined

In Task Class, I have implemented logic of functions e.g. mark_completed(), change_title(), change_due_date(). When any of these details will be required to be changed of specific task, these methods will be called respectively

```
22      def mark_completed(self):
23          self.completed = True
24      def change_title(self, new_title):
25          self.title = new_title
26      def change_due_date(self, new_date):
27
28          self.due_date = new_date
```

In TaskList Class, I have implemented logic of methods e.g. add_task(), view_tasks(), remove_task(). These methods will modify the tasks_list accordingly.

```python
    def add_task(self, task:Task):
        # Add a task to the list
        self.tasks.append(task)
    def remove_task(self,index):
        # not done yet
        # Remove a task by its index (user sees 1-based index)
        if index >= 1 and index <= len(self.tasks):
            print("index: ", index)
            print("len: ", len(self.tasks))
            print("self.tasks[index-1]: ", self.tasks[index-1])

            print(f"Removed: {self.tasks[index-1].title}")

            # delete the task
            # del self.tasks[index]
        else:
            print("Invalid index. Please try again.")

        # print("remove task")
    def view_tasks(self):
        # Show all tasks in the list
        if not self.tasks:
            print("No tasks in the list.")
        else:
            print("Your Current Tasks:")
            for index, task in enumerate(self.tasks):
                print(f"{index + 1}. {task}")
                # print(f"{index + 1}. {task.title} | {task.description}")
```

List_options() method will be calling for showing menu to the user and then calling the respective methods according to the choice.

```
    def list_options(self):
        while True:
            print("To-Do List Manager")
            print("1. Add a task")
            print("2. View tasks")
            print("3. Remove a task")
            print("4. Mark as completed")
            print("5. Change title of task")
            print("6. Quit")

            choice = input("Enter your choice: ")
            print("\n")

            if choice == "1": ···

            elif choice == "2": ···

            elif choice == "3": ···
            elif choice == "4": ···
            elif choice == "5": ···
            elif choice == "6": ···
            else:
                print("Invalid choice. Please enter a number between 1 and 6.\n")
```

Exercise 3: Testing the Functionality

Testing is done to check whether the code is working correctly or not.

```
task_list.list_options()
```

Output

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Quit
Enter your choice:
```

Source code

Choosing one option lets us add one new task to the list at a time. Its working correctly

```python
if choice == "1":
    task_title = input("Enter title of task: ")
    # self.add_task(task)
    task_description = input("Enter the description: ")
    input_date = input("Enter a due date (YYYY-MM-DD): ")
    due_date = datetime.datetime.strptime(input_date, "%Y-%m-%d")
    task = Task(task_title, task_description, due_date)
    self.add_task(task)
    print(f"'{task_title}' has been added to your to-do list.\n")
    print("-"*40)
```

Output

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Quit
Enter your choice: 1


Enter title of task: task new
Enter the description: desc new
Enter a due date (YYYY-MM-DD): 2021-1-1
'task new' has been added to your to-do list.
```

Source code

Choosing option 2 to view all the tasks. It's working correctly

```python
elif choice == "2":
    self.view_tasks()
    spacing()
```

Output

```
Enter your choice: 2


Your Current Tasks:
1. Task: task new | Status: Pending | Due Date: 2021-01-01 00:00:00 | Description: desc new
```

Source code

Choosing option 3 to delete the task

```
elif choice == "3":
    self.view_tasks()
    if not self.tasks:
        # print("There are no tasks to remove.\n")
        spacing()
        continue
    index = int(input("Enter the number of the task to remove: "))
    print("\n")
    if index < 1 or index > len(self.tasks):
        print("Invalid task number. Please try again.\n")
        spacing()
        continue

    self.remove_task(index)
    spacing()
```

Output

```
6. Quit
Enter your choice: 3


Your Current Tasks:
1. Task: task new | Status: Pending | Due Date: 2029-02-01 00:00:00 | Description: desc new
Enter the number of the task to remove: 1


index:  1
len:  1
self.tasks[index-1]:  Task: task new | Status: Pending | Due Date: 2029-02-01 00:00:00 | Description: desc new
Removed: task new


--------------------------------------


To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Quit
Enter your choice: 2


No tasks in the list.
```

Exercise 4: Composition

Composition is an object-oriented programming concept where one class is made up of or contains objects of another class.

Source Code

```
if choice == "1":
    task_title = input("Enter title of task: ")
    # self.add_task(task)
    task_description = input("Enter the description: ")
    input_date = input("Enter a due date (YYYY-MM-DD): ")
    due_date = datetime.datetime.strptime(input_date, "%Y-%m-%d")
    task = Task(task_title, task_description, due_date)
    self.add_task(task)
    print(f"'{task_title}' has been added to your to-do list.\n")
    print("-"*40)
```

Explanation

This code is taking title, desc and due date of a task as input from user. It then saves information using object of Task Class. It then adds that object to the Task List class using add_task() method. In this way TaskList can have many Tasks

str method

If we simply print the Task object , it will show something like

```
<__main__.Task object at 0x000001A3D3B5>
```

But we want to see the details of Task e.g. title, description, etc. For this purpose, we use _str_ method, which will convert the object into string and then will show it to user in readable format.

```
def __str__(self):
    status = "Completed" if self.completed else "Pending"
    return f"Task: {self.title} | Status: {status} | Due Date: {self.due_date} | Description: {self.description}"
```

Task: Change code in the Task Class

Understanding the Task

'completed' attribute is to be added to the class to mark the status of the task. First, it should be false to show that the task is not completed yet. There should be a function **mark_completed()** to update the status of the task. **change_title()** method will change the title of the respective task. All these details will be shown to the user by **_str_** method.

Source Code

```
class Task:
    def __init__(self, title, description, due_date):
        self.title = title
        self.description = description
        self.completed = False
        self.date_created = datetime.datetime.now()
        self.due_date = due_date

    def __str__(self):
        status = "Completed" if self.completed else "Pending"
        return f"Task: {self.title} | Status: {status} | Due Date: {self.due_date} | Description: {self.description}"

    def mark_completed(self):
        self.completed = True
    def change_title(self, new_title):
        self.title = new_title
```

Task: Update list_options() method

The options mark_completed(), change_title(), change_due_date() should be added to the menu items to show to the user to operate.

```
    def list_options(self):
        while True:
            print("To-Do List Manager")
            print("1. Add a task")
            print("2. View tasks")
            print("3. Remove a task")
            print("4. Mark as completed")
            print("5. Change title of task")
            print("6. Quit")
```

If Elif statements

```
        elif choice == "4":
            self.view_tasks()
            print("\n")
            if not self.tasks:
                print("-"*40)
                print("\n")
                continue
            while True:
                index = input("Enter the number of the task to mark as completed: ")
                if index.isdigit():
                    # Convert to actual integer
                    index = int(index)
                    if index > 0 and index <= len(self.tasks) :
                        self.tasks[index-1].mark_completed()
                        break  # Exit the loop since input is valid
                    else:
                        print("Invalid task number. Please try again.\n")
                        continue
                else:
                    print("Invalid input. Please enter a number like 1, 2, 3...")

            spacing()
```

```
        elif choice == "5":
            self.view_tasks()
            print("\n")
            if not self.tasks:
                # print("There are no tasks available\n")
                spacing()
                continue
            while True:
                index = input("Enter the number of the task to change title: ")

                if index.isdigit():
                    # Convert to actual integer
                    index = int(index)
                    if index > 0 and index <= len(self.tasks) :
                        new_title = input("Enter the new title: ")
                        self.tasks[index-1].change_title(new_title)
                        break  # Exit the loop since input is valid
                    else:
                        print("Invalid task number. Please try again.\n")
                        continue

                else:
                    print("Invalid input. Please enter a number like 1, 2, 3...")
```

Section 2. Python Libraries

Libraries are collections of functions and methods that allow you to perform actions, without having written the code yourself.

Exercise 1: Adding Dates

Understanding the Task

It is important for each task to have both the date it was created and the due date. To handle this, I used Python's built-in **datetime** library. A method called **change_due_date** was also added so that the due date can be updated later if needed.

Source Code

It is required to import the library, mostly at the top of the file.

datetime is the python library which is used to work with the dates and time. It lets us to :

- Get the current date and time
- Format dates in different ways
- Compare dates
- Add or subtract days, months, etc.
- Convert strings into date objects

```
import datetime
```

In this program, it is required to convert the string date (input from user) into datetime object and then save it to the Task. **strptime** also known as 'String Parse Time' is used to **convert a date string into a proper datetime object**, using a specific format. It takes 2 arguments, one is string and other is format, in which the string has to be converted.

```
input_date = input("Enter a due date (YYYY-MM-DD): ")
due_date = datetime.datetime.strptime(input_date, "%Y-%m-%d")
task = Task(task_title, task_description, due_date)
```

Or getting the present date and time

```
self.date_created = datetime.datetime.now()
```

Task: Add the due_date functionality

For changing due date of the task, I have created 'change_due_date()'

```
    def change_due_date(self, new_date):
        self.due_date = new_date
```

Modification of If-else statements in list_options() method for changing due date

```
    def list_options(self):
        while True:
            print("To-Do List Manager")
            print("1. Add a task")
            print("2. View tasks")
            print("3. Remove a task")
            print("4. Mark as completed")
            print("5. Change title of task")
            print("6. Change due date of task")
            print("7. Quit")

            choice = input("Enter your choice: ")
            print("\n")
```

```
            elif choice == "6":
                self.view_tasks()
                print("\n")
                if not self.tasks:
                    # print("There are no tasks available\n")
                    spacing()
                    continue
                while True:
                    index = input("Enter the number of the task to change due date: ")

                    if index.isdigit():
                        # Convert to actual integer
                        index = int(index)
                        if index > 0 and index <= len(self.tasks) :
                            new_date = input("Enter the new due date (YYYY-MM-DD): ")
                            new_due_date = datetime.datetime.strptime(new_date, "%Y-%m-%d")
                            self.tasks[index-1].change_due_date(new_due_date)
                            break  # Exit the loop since input is valid
                        else:
                            print("Invalid task number. Please try again.\n")
                            continue
```

Output

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Change due date of task
7. Quit
Enter your choice: 6


Your Current Tasks:
1. Task: charge mobile | Status: Pending | Due Date: 2022-02-02 | Description: charge mobile with pwer bank


Enter the number of the task to change due date: 1
Enter the new due date (YYYY-MM-DD): 2024-2-2
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Change due date of task
7. Quit
Enter your choice: 2


Your Current Tasks:
1. Task: charge mobile | Status: Pending | Due Date: 2024-02-02 | Description: charge mobile with pwer bank
```

Section 3. Modularizing the code

Exercise 1: Restructuring

Understanding the Task

In this task, I learned the importance of organizing code by splitting it into multiple files, which is called **modularization**. Instead of writing everything in one large script, I had to separate my code into different modules, each handling a specific part of the program.

I was asked to create a new folder called ToDoApp, then inside it:

- Create a main.py file to act as the **main.py entry point** of the application.
- Move the **Task** class into a new file called **task.py**.
- Move the **TaskList** class into another file called **task_list.py**.

This structure helps keep the code cleaner, easier to understand, and more manageable, especially as the program grows. I also had to handle importing properly.

Explanation

To make the code more organized, I divided it into three separate Python files:

main.py

This file acts as the **entry point** of the program. It contains the user interface (menu), takes input from the user, and calls functions from other files. This is the file I run to start the application.

task.py

This module contains the **Task class**, which holds all the properties of a task, like title, description, due date, date created, and whether the task is completed. It also includes useful methods like:

- mark_completed()
- change_title()
- change_description()
- change_due_date()

task_list.py

This file contains the **TaskList class**, which manages a list of Task objects. It allows adding, removing, viewing, and checking overdue tasks.

Output

- The program runs smoothly by calling everything from main.py, while the logic stays separated in task.py and task_list.py.
- The output remains the same as before, the user can add, view, update, or remove tasks using the menu.
- Code is now cleaner, easier to debug, and simpler to extend in the future.
- If I ever want to reuse the Task or TaskList classes in another project, I can do so without rewriting them.
- It follows a good programming habit of separating logic into modules, which is useful for teamwork and larger projects.

Import statement

When one file contents are being used in another file, it must be imported into the second file at the top, otherwise it will give error.

Source code

```
1  import datetime
2  from task import Task
3
```

The second import statement says that I have imported Task class from task file. ‘task’ file is basically task.py file.

Exercise 2: Main()

Understanding the Task

In this task, I had to properly define a **main() function** that serves as the starting point of the program. The purpose was to cleanly separate the program's setup logic and make the code more structured.

Instead of writing everything directly at the bottom of the file, I placed the core startup code inside main() and then called it safely using:

```
if __name__ == "__main__":
```

Source Code

```python
def main():
    print("="*40)
    print("----Welcome to the To-Do List Manager----\n")

    name = input("Enter your name: ")

    task_list = TaskList(name)
    print("\n")


    task_list.list_options()
```

```python
if __name__ == "__main__":
    main()
```

Explanation

- Inside the main() function, I created an instance of TaskList, passing a name (e.g., "Ahmed").
- Then I called task_list.list_options(), this method displays the menu and lets the user interact with the to-do list.
- The condition if __name__ == "__main__" makes sure the app runs only when executed directly, not when imported.

This structure is helpful for testing, modularity, and professional coding practices.

Task: Move Menu Logic to main() in main.py

Understanding the Task

In this task, I was required to remove the list_options() method from the TaskList class and move its code into the main() function inside main.py. The purpose of this change is to make the code more modular and better structured.

Since the menu and user interaction part is not the responsibility of the TaskList class (which should only manage tasks), it makes more sense to place that logic in main.py, where the user runs the program.

Explanation

Previously, the TaskList class included a method called list_options() that handled everything, from displaying the menu to taking user input and performing actions like adding or removing tasks. But that mixed two responsibilities into one class:

- Task management
- User interaction

To follow proper object-oriented design, I:

- Opened task_list.py and **copied the entire list_options() method's content**
- Pasted the code inside the main() function in main.py
- Removed the list_options() method from TaskList class
- Deleted the line task_list.list_options() and replaced it with the actual menu logic now inside main()

Now, main.py handles the user interaction, and TaskList only manages the task-related functions. This separation improves the design and makes future updates (like replacing the menu with a GUI) much easier.

Task: Using task_list object instead of self

Understanding the Task

When I moved the menu logic from the TaskList class into the main() function in main.py, I had to replace all instances of self with task_list. This was necessary because I was no longer inside a class method. I was now working in a regular function (main()), where self is not available. The task_list object was already created earlier in main() to represent the user's task manager, so I used it to access tasklist class methods.

```
        if choice == "1":
            task_title = input("Enter title of task: ")
            task_description = input("Enter the description: ")

            # this loop is to ask the user to enter date until it is valid
            while True :
                input_date = input("Enter a due date (YYYY-MM-DD): ")
                due_date = datetime.datetime.strptime(input_date, "%Y-%m-%d").date()
                task = Task(task_title, task_description, due_date)
                self.add_task(task)
                print(f"'{task_title}' has been added to your to-do list.\n")
                break
            print("-"*40)
```

Explanation

In the original list_options() method inside the TaskList class, all method calls used self, like this:

```
self.add_task(task)
```

But after moving this logic to main.py, we're no longer inside the TaskList class. So, we need to use the actual object created in main(), which is:

```
task_list = TaskList(name)
```

Now, to call methods on this object, I changed self to task_list, like:

```
task_list.add_task(task)
```

Task: Add Helper function for test tasks

Understanding the Task

In this task, I was asked to add a helper function named propagate_task_list() that would automatically fill the task list with some sample tasks when the program starts. The main reason for this was to make testing easier, so I wouldn't have to manually add tasks every time I run the program.

The function takes a TaskList object and adds several tasks to it, with different due dates (some in the past, some in the future). Then it returns the updated task list back to the main program.

Source Code

Definition

```
def propagate_task_list(task_list: TaskList) -> TaskList:
    """Adds some sample tasks to the task list for testing."""
    task_list.add_task(Task("Buy groceries", "Milk, eggs, and bread", datetime.datetime.now() - datetime.timedelta(days=4)))
    task_list.add_task(Task("Do laundry", "Wash and fold clothes", datetime.datetime.now() + datetime.timedelta(days=2)))
    task_list.add_task(Task("Clean room", "Organize desk and vacuum floor", datetime.datetime.now() - datetime.timedelta(days=1)))
    task_list.add_task(Task("Do homework", "Finish math and science assignments", datetime.datetime.now() + datetime.timedelta(days=3)))
    task_list.add_task(Task("Walk dog", "Evening walk around the park", datetime.datetime.now() + datetime.timedelta(days=5)))
    task_list.add_task(Task("Do dishes", "Clean all utensils after dinner", datetime.datetime.now() + datetime.timedelta(days=6)))
    return task_list
```

Usage

```python
def main() -> None:
    print("="*40)
    print("----Welcome to the To-Do List Manager----\n")

    name = input("Enter your name: ")
    task_list = TaskList(name)

    # for test tasks
    task_list = propagate_task_list(task_list)
```

Explanation

To complete this task, I followed these steps:

Defined the Function:

I copied the propagate_task_list() function into the top section of my main.py file, right above the main() function. Inside this function, I added six sample tasks with various due dates.

Called It in main():

Inside my main() function, after creating the task list object.

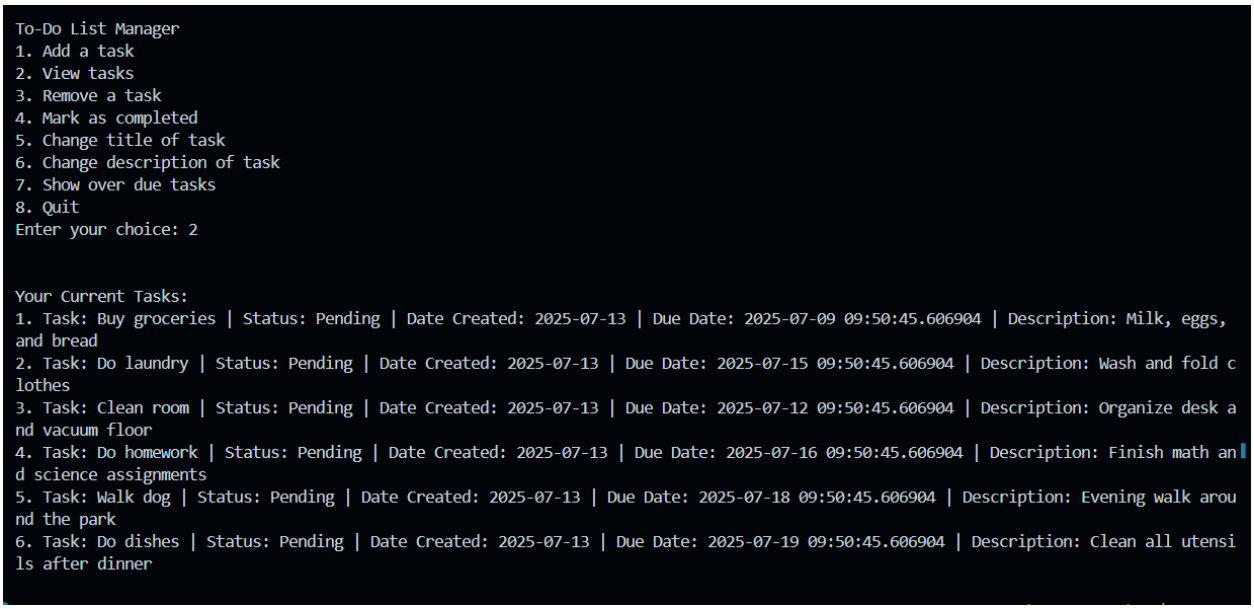
This made sure that every time I run the program, the task list already includes some tasks.

Benefits: Testing Becomes Easier:

Now when I run the app, I can immediately test features like viewing tasks, marking them as completed, removing them, or checking overdue tasks, without entering tasks manually each time.

This step didn't change how the app behaves for the user, but it made my development and testing process a lot faster and smoother.

Output



Section 4. Type Checking and Documenting your Code

Exercise 1: Type Checking

Understanding the Task

In this exercise, I learned how to use **type hints** in Python to make my code cleaner, safer, and easier to understand. Although Python doesn't force you to declare variable types (like Java or C++), it's considered good practice to include them using the syntax variable_name: type.

Type hints help:

- Prevent bugs by catching type-related mistakes early.
- Improve code readability.
- Allow editors like VS Code to show helpful suggestions or warnings.

Type Hints for Method Parameters

This part showed me how to define what type of input (parameter) a method or function expects. For example:

```
def __init__(self, title: str, date_due: datetime.datetime):
```

This means title should be a str and date_due should be a datetime object.
It helps others (and me) understand what kind of values should be passed into the method.

Another example:

```
def add_task(self, task: Task) -> None:
```

This method is expecting an object of type Task.

Type Hints for Return Values

“I also learned how to show what type of value a function returns by using the -> arrow

```
def __str__(self) -> str:
```

This means the __str__() method will return a string.
It’s useful because when I call this method later, I’ll know exactly what kind of output to expect.

Exercise 2: Docstrings and Comments

Understanding the Task

In this task, I learned the importance of **documenting code** so that it becomes easier to understand, both for myself in the future and for others who may read it. I explored two main ways of doing this:

1. **Comments** – Used to explain parts of the code in plain language.
2. **Docstrings** – Used to describe what a class, method, or function does. These are written inside triple quotes right after the function or class definition.

The goal was to practice writing both, so the code becomes more readable and professional.

Explanation

Comments:

I used # to add short notes inside my code to explain what each line or block is doing. For example:

```
# This function adds a new task to the list
```

Docstrings:

These are placed inside triple quotes """ """ and written right below the function or class header. They explain what the method/class is for, what arguments it takes, and what it returns (if anything). For example:

```
def add_task(self, task: Task) -> None:
    """Adds a new task to the task list."""
```

By adding docstrings and comments, my code became easier to read and work with. It also helped me stay organized and made it easier to review or debug things later.

Portfolio Exercise 1: Adding Description Attribute to Task

Understanding the Task

In this task, I was asked to improve the Task class by adding an optional **description** feature. This allows each task to have some extra details written about it, but it's not required.

I had to:

- Update the __init__ method so that the description can be passed when creating a task.
- Add a change_description() method to update the description later.
- Modify the __str__() method so that the description is included when printing the task.
- Lastly, I needed to update the main() function so that the user can change the task description through the menu (in the option where title and due date are already being updated).

Explanation

- I started by updating the constructor like this:

```
class Task:
    def __init__(self, title:str, description:None, due_date) -> None:
        self.title = title
        self.description = description
```

This made the description optional by setting its default value to None.

- Then I added a method:

```
def change_description(self, new_description) -> None:
    self.description = new_description
```

  This allowed the user to change the description any time they want.

- In the __str__() method, I added the description to the return string so that whenever a task is displayed, its description shows up too.
- Finally, in main.py, I added an input option for the user to update the task's description in the menu that also handles updating title and due date.

Portfolio Exercise 2: View overdue tasks

Understanding the Task

In this task, I had to add a feature that lets the user see all the **overdue tasks**, meaning tasks whose due date has already passed. For this, I needed to:

- Create a method called view_overdue_tasks() inside the TaskList class.
- Update the main() function and add a new menu option that calls this method.

This helps users easily track which tasks they've missed or need urgent attention.

Explanation

In the TaskList class, I wrote a new method called view_overdue_tasks() which:

- Loops through all tasks.
- Checks if the due_date is **less than today's date** using datetime.date.today().
- Prints those tasks as overdue.

```python
def view_over_due_tasks(self)->None:
    # if any of tasks are present in list
    if not self.tasks:
        print("No tasks in the list.")
        return

    over_due_tasks = []
    today = datetime.date.today()
    # if date of any task is passed already
    for index, task in enumerate(self.tasks, start=1):
        if task.due_date < today:
            over_due_tasks.append((index, task))
    #  Display overdue tasks if found
    if over_due_tasks:
        print("Over Due Tasks:")
        for i, task in over_due_tasks:
            desc = task.description if task.description else "No description"
            print(f"{i}. {task.title} | Due Date: {task.due_date} | Description: {desc}")
    else:
        print("No over due tasks available")
```

Then, in main.py, I added a new choice in the menu, that calls this method when selected.

```python
while True:

    # menu
    print("To-Do List Manager")
    print("1. Add a task")
    print("2. View tasks")
    print("3. Remove a task")
    print("4. Mark as completed")
    print("5. Change title of task")
    print("6. Change description of task")
    print("7. Show over due tasks")
    print("8. Quit")
                You, 2 days ago • 2nd commit
    choice = input("Enter your choice: ")
    print("\n")
```

Output

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Change description of task
7. Show over due tasks
8. Quit
Enter your choice: 7


Over Due Tasks:
3. Buy groceries | Due Date: 2025-07-09 | Description: Milk, eggs, and bread
5. Clean room | Due Date: 2025-07-12 | Description: Organize desk and vacuum floor
```

Week 5

Section 1. Inheritance

Exercise 1: Simple Inheritance

Understanding the Task

In this exercise, I learned the concept of **inheritance** in object-oriented programming. The main goal was to understand how a child's class can reuse and extend the features of a parent class. For example, since a **Car is a Vehicle**, we can create a Car class that inherits from a base Vehicle class.

The idea is to avoid repeating code. We define common features like colour, weight, and max_speed once in the parent (Vehicle) class and then let child classes like Plane or Car use those features directly. Each child can also have its own specific attributes (like wingspan for a plane).

Explanation

- I started by creating a base class Vehicle with common properties.
- Then I made a child class (like Plane or Car) using inheritance like this:

  ```
  class Car(Vehicle):
  ```

  This means Car will automatically get everything from Vehicle.

- I also practiced **method overriding**, where I redefined the move() method inside the child class to change its output. For example:

```
def move(self, speed):
    print(f"The car is driving at {speed} km/h")
```

Even though the parent Vehicle class had its own move() method, this allowed me to customize it for each specific vehicle.

This exercise helped me understand how inheritance helps reduce code repetition, makes programs easier to organize, and supports flexible custom behavior in child classes.

Object Creation

```
# object of generic vehicle
generic_vehicle = Vehicle("red", 1000, 200)
generic_vehicle.move(100)

# object of generic car
generic_car = Car("red", 1000, 200, "SUV")
generic_car.move(100)
```

Output

```
The vehicle is moving at 100 km/h
The car is driving at 100 km/h
```

Adding more attributes to child class

Child class has all the attributes of parent class, but in some cases we need to add new attributes to child classes. For example, here in example of Car, the **form_factor** has to be added.

```python
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed
        self.form_factor = form_factor

    def move(self, speed):
        print(f"The car is driving at {speed} km/h")
```

Creating the object with form factor:

```python
car = Car("blue", 1500, 250, "SUV")
car.move(150)
```

Exercise 2: Super () function

super() is a special function used inside a **child class** to call a method from its **parent class**.

It's mostly used in **constructors (__init__)** to make sure the parent class is properly initialized before the child class adds more functionality.

```python
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor):
        super().__init__(colour, weight, max_speed)
        self.form_factor = form_factor
```

```python
    def specification(self):
        print(f"Colour: {self.colour}")
        print(f"Weight: {self.weight} kg")
        print(f"Max speed: {self.max_speed} km/h")
        print(f"Form factor: {self.form_factor}")
```

Output

```
Colour: red
Weight: 1000 kg
Max speed: 200 km/h
Form factor: SUV
```

Task: Create ElectricCar and the PetrolCar class.

```
class Electric(Car):
    def __init__(self, colour, weight, max_speed, form_factor, battery_capacity):
        super().__init__(colour, weight, max_speed, form_factor)
        self.battery_capacity = battery_capacity

    def move(self, speed):
        print(f"The electric car is driving at {speed} km/h")

class Petrol(Car):
    def __init__(self, colour, weight, max_speed, form_factor, fuel_capacity):
        super().__init__(colour, weight, max_speed, form_factor)
        self.fuel_capacity = fuel_capacity

    def move(self, speed):
        print(f"The petrol car is driving at {speed} km/h")
```

Test

```
electric_car = Electric("green", 1200, 200, "Hatchback", 100)
electric_car.move(100)

petrol_car = Petrol("red", 1500, 250, "SUV", 50)
petrol_car.move(150)

generic_vehicle = Vehicle("red", 1000, 200)
generic_vehicle.move(100)
```

Output

```
The electric car is driving at 100 km/h and has a maximum range of 200
The petrol car is driving at 150 km/h and has a maximum range of 100
The vehicle is moving at 100 km/h
```

Task: Adding max range parameter

Some vehicles have max range, but some do not have. That's why max_range attribute is set to be None on default. If any vehicle has this value, it will be used.

```
class Vehicle:
    def __init__(self, colour, weight, max_speed, max_range=None):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed
        self.max_range = max_range
```

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor, max_range=None):
        super().__init__(colour, weight, max_speed, max_range)
        self.form_factor = form_factor
```

```
class Electric(Car):
    def __init__(self, colour, weight, max_speed, form_factor, battery_capacity, max_range=None):
        super().__init__(colour, weight, max_speed, form_factor, max_range)
        self.battery_capacity = battery_capacity
```

```
class Petrol(Car):
    def __init__(self, colour, weight, max_speed, form_factor, fuel_capacity, max_range=None):
        super().__init__(colour, weight, max_speed, form_factor, max_range)
        self.fuel_capacity = fuel_capacity
```

Using max range in electric car move method

```
class Electric(Car):
    def __init__(self, colour, weight, max_speed, form_factor, battery_capacity, max_range=None, seats=None):
        super().__init__(colour, weight, max_speed, form_factor, max_range=max_range, seats=seats)
        self.battery_capacity = battery_capacity
    def move(self, speed):
        print(f"The electric car is driving at {speed} km/h and has a maximum range of {self.max_range}")
```

Exercise 2: kwargs**

By using kwargs**, we can pass as many keyword arguments as possible. These are stored in dictionary data structure

To use kwargs**, add this keyword as a parameter to any child class that is derived from Vehicle. Then pass this keyword to the parent class using super() function.

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor, **kwargs):
        super().__init__(colour, weight, max_speed, **kwargs)
        self.form_factor = form_factor
```

Understanding the Task

In this task, I learned how to use **kwargs in a function or constructor.
The purpose of **kwargs is to allow a function or method to accept **any number of keyword arguments** (like seats=4, max_range=200) even if they aren't explicitly listed in the parameter list.

This is helpful when working with **inheritance** because sometimes a child's class needs to pass extra arguments to the parent class without knowing exactly what those arguments are.

Explanation

- **kwargs stands for **"keyword arguments"**
- It collects any extra named arguments as a dictionary

Source Code

```
def greet(**kwargs):
    print(kwargs)

greet(name="Ali", age=22)
```

Output

```
Output: {'name': 'Ali', 'age': 22}
```

Task: Adding seats attribute

```
class Vehicle:
    def __init__(self, colour, weight, max_speed, max_range=None, seats=None):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed
        self.max_range = max_range
        self.seats = seats
```

Test

```
# object of generic electric car
generic_car1 = Electric("red", 1000, 200, "SUV", 100, max_range=500, seats=5)
generic_car1.move(100)
print(generic_car1.seats)
```

Output

```
The electric car is driving at 100 km/h and has a maximum range of 500
5
```

Task: Creating Multilevel Inheritance

Understanding the Task

In this task, I was asked to extend the Vehicle class by adding a new child class called Plane, and two more specific child classes from that, Propeller and Jet. This type of inheritance is called **multilevel inheritance.**

Each of these subclasses should:

- Inherit common vehicle features like colour, weight, and max_speed
- Add their own unique attribute:
    - Plane → wingspan
    - Propeller → propeller_diameter
    - Jet → engine_thrust

Also, each class should have its own version of the move() method, and since they are all flying machines, their output should say **"flying"** instead of "**driving**" or "**moving**".

Explanation

- I started by creating a Plane class that **inherits from** Vehicle.
- Inside Plane, I added the extra attribute wingspan, and overrode the move() method to say respective text:

```
class Plane(Vehicle):
    """ This class is a subclass of the Vehicle class, having one new argument wingspan"""
    def __init__(self, colour, weight, max_speed, wingspan,**kwargs):
        super().__init__(colour, weight, max_speed , **kwargs)
        self.wingspan = wingspan

    def move(self, speed):
        print(f"The plane is flying at {speed} km/h")
```

- Then, I created two more child classes:
    - Propeller(Plane) → adds propeller_diameter
    - Jet(Plane) → adds engine_thrust

```
class Propeller(Plane):
    """ This class is a subclass of the Plane class, having one new argument propeller_diameter"""
    def __init__(self, colour, weight, max_speed, wingspan, propeller_diameter):
        super().__init__(colour, weight, max_speed, wingspan)
        self.propeller_diameter = propeller_diameter

    def move(self, speed):
        print(f"The propeller plane is flying at {speed} km/h")

# Jet plane subclass
You, 35 seconds ago | 1 author (You)
class Jet(Plane):
    """ This class is a subclass of the Plane class, having one new argument engine_thrust"""
    def __init__(self, colour, weight, max_speed, wingspan, engine_thrust):
        super().__init__(colour, weight, max_speed, wingspan)
        self.engine_thrust = engine_thrust

    def move(self, speed):
        print(f"The jet is flying at {speed} km/h")
```

- Each subclass also had its own custom move() method to match its type.
- I tested each class by creating objects and printing their attributes to make sure everything worked as expected.

Section 3. Multiple Inheritance

Understanding the Task

In this task, I had to create a class called FlyingCar that inherits from **both** the Car class and the Plane class. This is a good example of **multiple inheritance**, where a child's class gets features from more than one parent's class.

Since a flying car has the properties of both a car (like wheels and form factors) and a plane (like wingspan), it makes sense to inherit from both.

Explanation

- I created the FlyingCar class.

```
class FlyingCar(Car, Plane):
    """ This class is a subclass of the Car and Plane classes"""
    def __init__(self, colour, weight, max_speed, form_factor, wingspan, **kwargs):
        # we need to add the wingspan to the keyword arguments so that following the MRO, the Plane class
        gets all the keyword arguments it needs
        super().__init__(colour, weight, max_speed, form_factor=form_factor, wingspan=wingspan, **kwargs)
    def move(self, speed):
        print(f"The flying car is driving or flying at {speed} km/h")
```

- In the constructor (__init__), I used super() to call the constructors of the parent classes and passed all necessary arguments like form_factor and wingspan. Since Car and Plane both come from Vehicle, I made sure to use **kwargs to pass extra arguments smoothly.
- I also overrode the move()
- I created an object of **FlyingCar** to test whether all attributes (from both car and plane) were set correctly, like **form_factor,** wingspan and seats.

This task helped me understand how multiple inheritance works in Python, and how to combine behaviors from different classes into one.

```
# multiple inheritance
# object of flying car
generic_flying_car = FlyingCar("red", 1000, 200, "SUV", 30, seats=5)
generic_flying_car.move(100)
print(generic_flying_car.seats, generic_flying_car.wingspan,
generic_flying_car.form_factor)
```

```
# object of flying car with more clarity
generic_flying_car_2 = FlyingCar(colour="red", weight=1000, max_speed=200,
form_factor="SUV", wingspan=30, seats=5)
generic_flying_car_2.move(100)
```

Output

```
The flying car is driving or flying at 100 km/h
5 30 SUV
The flying car is driving or flying at 100 km/h
```

Section 4. Polymorphism

Polymorphism allows different classes to have a **common method name** (like move()), but each class can perform **its own version** of that method.

Understanding the Task

The idea is that we don't need to know what kind of object we're working with. If it has the method, we can just call it, and Python will run the correct version automatically. This is especially powerful when we're looping over objects of different types.

Explanation

The most common example of polymorphism is when a **parent class has a method**, and each **child class overrides** it with its own behavior. It means every child class also has the method with same name and same arguments but with different implementation

Let's say we have:

- A Vehicle class → has a move() method
- Car, Plane, and FlyingCar subclasses → each with their own version of move()

Even though the method name move() is the same, the **output will depend on which object** is calling it. This is known as **method overriding**, and it's a core part of polymorphism.

Source Code

```
vehicle = Vehicle("red", 1000, 150)
car = Car("blue", 1200, 180, "Sedan")
plane = Plane("white", 5000, 600, 25)
flying_car = FlyingCar("silver", 1300, 200, "Hybrid", 15)
animal = Animal()


movable_objects = [vehicle, car, plane, flying_car, animal]

# all classes have some implementation of move()
# Calling move() on each object - this is polymorphism
for obj in movable_objects:
    obj.move(20)
```

Explanation

Creating objects of different classes with their respective arguments one by one. Saving all objects in a list and then calling move method of each object displaying the concept of polymorphism.

Output

```
The vehicle is moving at 20 km/h
The car is driving at 20 km/h
The plane is flying at 20 km/h
The flying car is driving or flying at 20 km/h
The animal is walking at 20 km/h.
```

Section 6. ToDo

Task: Add Recurring Task functionality

Understanding the Task

In this task, I had to extend the functionality of the ToDoApp by supporting **recurring tasks**, the kind of tasks that repeat over time (like doing laundry every week or cleaning every 2

weeks). Instead of manually adding these tasks repeatedly, the app should handle them smartly.

To do that, I had to:

- Create a new class called RecurringTask that **inherits from** Task.
- Add new features:
    - interval: a timedelta object that stores how often the task repeats.
    - completed_dates: a list to store **all dates** on which the task was marked as done.
    - _compute_next_due_date(): a method to **automatically calculate the next due date** based on the interval.
- Override the __str__() method to include the **interval** and **completed history**, so we can tell which tasks are recurring when we list them.
- Modify the logic in **option "1"** of the main() function to:
    - Ask the user if they want to add a recurring task or not.
    - If yes → ask for the interval in days, convert it using timedelta, and create a RecurringTask object.
    - If no → create a normal Task as before.

Explanation

- I created the RecurringTask
- In the constructor, I added:
    - self.interval = interval

- self.completed_dates = [] → to store completion history

- I added a private method _compute_next_due_date() which calculates the next deadline based on the last completion date or the current due date:

- I overrode the __str__() method to show:

  - Title, due date, status (completed/pending)

  - Interval (like "every 7 days")

  - Completed history

```python
class RecurringTask(Task):
    """
    This class is for recurring tasks
    Args: inherits from parent class 'Task'
    one new argument -> interval is added
    """
    def __init__(self, title:str, description:str, due_date, interval:datetime.timedelta) -> None:
        # title, description, due_date are attributes inherited from parent class
        # interval is new attribute which is for repetition of tasks
        super().__init__(title, description, due_date)
        self.interval = interval
        # list of completed dates of recurring tasks for history
        self.completed_dates : list[datetime.datetime] = []

    def _compute_next_due_date(self) -> datetime.datetime:
        """Computes the next due date of the task.
        Returns:
        datetime.datetime: The next due date of the task.
        """
        return self.date_due + self.interval
    def __str__(self):
        # this will use the string method of parent class and then concatenate the new attribute
        return super().__str__() + f" | interval: {self.interval.days} days"
```

- Then in main.py, I updated the **task-adding flow**:

  - The user is asked: “Add a one time task or a recurring task?”

  - If recurring task, they enter an interval like "7"

  - I converted it using:

```
        if choice == "1":
            while True:
                print("1. One Time Task")
                print("2. Recurring Task")
                print("3. Back")

                sub_choice = input("Enter your choice: ")
                if sub_choice == "1":
                    # get task details method is to get user input of title, description and due date of task
                    task_title, task_description, due_date = get_task_details(task_list)
                    # task object is created and then added to the task list
                    task = Task(task_title, task_description, due_date)
                    task_list.add_task(task)
                    print(f"'{task_title}' has been added to your to-do list.\n")
                    break
                elif sub_choice == "2":
                    task_title, task_description, due_date = get_task_details(task_list)
                    # interval is taken input from user in days e.g. 1, 2, 3
                    """ then its converted to timedelta and then passed into Recurring Task object to create
                    Recurring Task and then adding it to task list"""
                    interval = input("Enter the interval in days: ")
                    interval = datetime.timedelta(days=int(interval))

                    task = RecurringTask(task_title, task_description, due_date, interval)
                    task_list.add_task(task)
                    print(f"'{task_title}' has been added to your to-do list.\n")
                    break
                elif sub_choice == "3":
                    break
```

Explanation

In this code, if user enters option 1 to add a task, it then asks him to enter the choice whether he wants to add a one-time task or a recurring one. In both cases, Program will ask the user to enter task details by get_task_details() method. If the option was to add a recurring task, then program will ask the user to enter number of days for interval. The program will then convert the string number of days into timedelta object using the datetime library. It will then create an object of Recurring Task by passing task details in it. Whether the Task created is a one-time task or a recurring task, it will be added into Task List.

Output

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Change description of task
7. Show over due tasks
8. Change Due date of task
9. Quit
Enter your choice: 1


1. One Time Task
2. Recurring Task
3. Back
Enter your choice: 2
Enter title of task: task title
Enter the description: task desc
Enter a due date (YYYY-MM-DD): 2021-1-1
Enter the interval in days: 2
'task title' has been added to your to-do list.
```

```
Enter your choice: 2


Your Current Tasks:
1. Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-09 | Description: Milk, eggs, and bread
2. Task: Do laundry | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-15 | Description: Wash and fold clothes
3. Task: Clean room | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-12 | Description: Organize desk and vacuum floor
4. Task: Do homework | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-16 | Description: Finish math and science assign
ments
5. Task: Walk dog | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-18 | Description: Evening walk around the park
6. Task: Do dishes | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-19 | Description: Clean all utensils after dinner
7. Task: task title | Status: Pending | Date Created: 2025-07-13 | Due Date: 2021-01-01 | Description: task desc | interval: 2 days
```

Task: Add Recurring Task in Propagate Task List

Understanding the Task

It is asked to add a recurring task in tasks list with all its required attributes, by using the method propagate_task_list

Source Code

```
def propagate_task_list(task_list: TaskList) -> TaskList:
    """Adds some sample tasks to the task list for testing."""
    task_list.add_task(Task("Buy groceries", "Milk, eggs, and bread", datetime.datetime.now().date() - datetime.
    timedelta(days=4)))
    task_list.add_task(Task("Do laundry", "Wash and fold clothes", datetime.datetime.now().date() + datetime.
    timedelta(days=2)))
    task_list.add_task(Task("Clean room", "Organize desk and vacuum floor", datetime.datetime.now().date() -
    datetime.timedelta(days=1)))
    task_list.add_task(Task("Do homework", "Finish math and science assignments", datetime.datetime.now().date() +
    datetime.timedelta(days=3)))
    task_list.add_task(Task("Walk dog", "Evening walk around the park", datetime.datetime.now().date() + datetime.
    timedelta(days=5)))
    task_list.add_task(Task("Do dishes", "Clean all utensils after dinner", datetime.datetime.now().date() +
    datetime.timedelta(days=6)))

    r_task = RecurringTask("Go to the gym",'description', datetime.datetime.now(),datetime.timedelta(days=7))
    # propagate the recurring task with some completed dates
    r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=7))
    r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=14))
    r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=22))
    r_task.date_created = datetime.datetime.now() - datetime.timedelta(days=28)
    task_list.add_task(r_task)

    return task_list
```

Explanation

This code is getting a task list object. It then adds 6 objects of tasks into the task list. It also creates an object of Recurring task, then adding it to the task list with completed dates attribute. At the end it is returning a list back. The list returned is the one which we got empty in the start.

Output

```
Enter your choice: 2


Your Current Tasks:
1. Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-09 | Description: Milk, eggs, and bread
2. Task: Do laundry | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-15 | Description: Wash and fold clothes
3. Task: Clean room | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-12 | Description: Organize desk and vacuum floor
4. Task: Do homework | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-16 | Description: Finish math and science assign
ments
5. Task: Walk dog | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-18 | Description: Evening walk around the park
6. Task: Do dishes | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-19 | Description: Clean all utensils after dinner
7. Task: Go to the gym | Status: Pending | Date Created: 2025-06-15 | Due Date: 2025-07-13 16:07:44.531597 | Description: description
| interval: 7 days
```

Explanation

The first 6 tasks are normal one-time tasks and the ls tine is recurring task because of the interval attribute.

Task: Mark Recurring Task Completed

Understanding the Task

In the previous version of the ToDoApp, when we marked any task as completed (choice 5), it just updated the task's status to completed = True. But for **recurring tasks**, that's not enough.

This task asked me to improve the behavior of recurring tasks. Instead of just marking them as completed, the app should:

- Keep a **record** of the date when it was completed
- **Update the due date** for the next cycle automatically (e.g., next week)

To implement this, I had to override the mark_completed() method in the RecurringTask class using **polymorphism**.

Source Code

```
def _compute_next_due_date(self) -> datetime.datetime:
    """Computes the next due date of the task.
    Returns:
    datetime.datetime: The next due date of the task.
    """
    # If task has been completed before, calculate next due from last completion
    if self.completed_dates:
        return self.completed_dates[-1] + self.interval
    # Otherwise calculate from existing due date
    return self.due_date + self.interval
def mark_completed(self) -> None:
    # Add today's date to completed history
    today = datetime.date.today()
    self.completed_dates.append(today)

    # Update due date to next scheduled one
    self.due_date = self._compute_next_due_date()

    # Mark as completed (from parent)
    self.completed = True
def __str__(self):
    # this will use the string method of parent class and then concatenate the new attribute
    return super().__str__() + f" | interval: {self.interval.days} days"
```

Explanation

- I created a new version of mark_completed() inside the RecurringTask class.
- Inside that method:
    - I first added today's date to the completed_dates list:
    - Then, I updated the due_date using the private method compute_next_due_date():
    - This calculates the next due date based on the task's repeat interval.
    - Finally, I marked the task as completed using:

This shows the use of **polymorphism**, where both Task and RecurringTask have a method with the same name (mark_completed), but the behavior is different based on the object type.

Output

```
Your Current Tasks:
1. Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-09 | Description: Milk, eggs, and bread
2. Task: Do laundry | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-15 | Description: Wash and fold clothes
3. Task: Clean room | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-12 | Description: Organize desk and vacuum floor
4. Task: Do homework | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-16 | Description: Finish math and science assign
ments
5. Task: Walk dog | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-18 | Description: Evening walk around the park
6. Task: Do dishes | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-19 | Description: Clean all utensils after dinner
7. Task: Go to the gym | Status: Pending | Date Created: 2025-06-15 | Due Date: 2025-07-13 16:59:19.249057 | Description: description
| interval: 7 days


Enter the number of the task to mark as completed: 7
```

```
Your Current Tasks:
1. Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-09 | Description: Milk, eggs, and bread
2. Task: Do laundry | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-15 | Description: Wash and fold clothes
3. Task: Clean room | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-12 | Description: Organize desk and vacuum floor
4. Task: Do homework | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-16 | Description: Finish math and science assig
ments
5. Task: Walk dog | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-18 | Description: Evening walk around the park
6. Task: Do dishes | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-19 | Description: Clean all utensils after dinner
7. Task: Go to the gym | Status: Completed | Date Created: 2025-06-15 | Due Date: 2025-07-20 | Description: description | interval: 7
days
```

Exercise 4 – Encapsulation

Understanding the Task

In this task, I was asked to improve the way tasks are accessed from the TaskList class. Previously, the code directly accessed the task list using task_list.tasks[index]. This is not a good practice because it exposes the internal structure.

The goal was to apply the concept of **encapsulation**, which means hiding internal details and only exposing what's necessary. I needed to:

- Create a method called get_task(index) inside the TaskList class.
- Replace direct access to task_list.tasks[...] in the main() function with this method.
- Make sure everything still works the same from a user's point of view.

Source Code

```
    def get_task(self, index):
        """Returns the task at the given index (1-based index for user-friendliness)."""
        if 1 <= index <= len(self.tasks):
            return self.tasks[index - 1]
        else:
            return None
```

Example Usage

```
if index > 0 and index <= len(task_list.tasks) :
    # get task from task list using get task method
    task = task_list.get_task(index)
    # performing operation on task object
    task.mark_completed()
    # task_list.tasks[index-1].mark_completed()
    break  # Exit the loop since input is valid
else:
    print("Invalid task number. Please try again.\n")
    continue
```

Explanation

Encapsulation is about protecting the data and only allowing controlled access. So instead of letting other parts of the app directly access the task list, I created a method get_task() inside TaskList. This method checks if the index is valid and safely returns the task.
By using this method

- The internal list (self.tasks) stays hidden and protected
- Any future changes in how tasks are stored won't affect the rest of the app
- It keeps the code clean, safe, and easier to maintain

Portfolio Exercise 3: Add User and Owner Functionalities

Source Code

user.py

```python
class User:
    def __init__(self, name: str, email: str):
        self.name = name
        self.email = email
```

owner.py

```python
from user import User
class Owner(User):
    def __init__(self, name: str, email: str):
        super().__init__(name, email)
```

Accept owner object in TaskList class

```python
class TaskList:
    # tasks = list[Task]
    def __init__(self, owner:Owner) -> None:
        self.owner = owner
        # self.owner = ""
        self.tasks = []
```

Usage in main function

```python
def main() -> None:
    print("="*40)
    print("----Welcome to the To-Do List Manager----\n")

    name = input("Enter your name: ")

    email = input("Enter your email: ")

    owner = Owner(name, email)
    task_list = TaskList(owner)
```

Understanding the Task (UIT)

This task was about applying **inheritance** and **composition** together in the ToDoApp. I had to introduce a new user system by:

- Creating a User class with basic info like name and email
- Creating an Owner class that inherits from User
- Modifying the TaskList class to include an owner attribute, which should be of type Owner

This helps structure the app more professionally and adds a clear connection between a task list and its owner.

Explanation

- I created a **base class User** that stores a person's name and email.
- Then I made an Owner class that **inherits from User**, meaning it automatically gets the name and email attributes.
- In the TaskList class, I added an attribute owner, which accepts an Owner object.
- This shows a **composition** relationship: a TaskList "has-an" Owner, while Owner "is-a" User.

This structure follows good OOP design and keeps responsibilities clear. If we want to expand later (e.g., add Admin or Guest users), this structure will make it much easier.

Output

```
Enter your name: Ahmed
Enter your email: ahmed@example.com
Welcome Ahmed! Your task list is ready.
```

Portfolio Exercise 4

Understanding the Task (UIT)

This task was focused on structuring the code better by using **modularization** and **OOP concepts**:

- I had to create two new files users.py and owner.py that contains two classes: User and Owner respectively
- Each class has a __str__() method to print user details nicely
- Then, I updated the TaskList class to accept an Owner object when initializing
- Finally, in main.py, I asked the user for their name and email, created an Owner object, and used that to create a personalized TaskList

Source Code

```python
class User:
    def __init__(self, name: str, email: str):
        self.name = name
        self.email = email
    def __str__(self):
        return f"User: {self.name} | Email: {self.email}"
```

```python
from user import User
class Owner(User):
    def __init__(self, name: str, email: str):
        super().__init__(name, email)
    def __str__(self):
        return f"Owner: {self.name} | Email: {self.email}"
```

Usage Example

```python
def main() -> None:
    print("="*40)
    print("----Welcome to the To-Do List Manager----\n")

    name = input("Enter your name: ")

    email = input("Enter your email: ")

    owner = Owner(name, email)
    print("\n" + str(owner))
    task_list = TaskList(owner)
    print("Your task list has been created successfully.\n")
    # for test tasks
```

Output

```
----Welcome to the To-Do List Manager----

Enter your name: abu bakar
Enter your email: abubakar@gmail.com

Owner: abu bakar | Email: abubakar@gmail.com
Your task list has been created successfully.
```

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Change description of task
7. Show over due tasks
8. Change Due date of task
9. Get Owner Details
10. Quit
Enter your choice: 9


Owner details:

Name:  abu bakar
Email:  abubakar@gmail.com
```

Week 6

Section 1. Debugging

The debugging process is an important part of programming. It allows you to find and fix errors in your code. A debugger is a tool that allows you to step through your code and see what is happening at each step.

Exercise 1: Finding the Problem

Problem Statement:

I was given a Python program that simulates a car's actions like accelerating, braking, and tracking distance (odometer) and average speed. When I ran the program and selected the following actions:

A (Accelerate), B (Brake), B (Brake), O (Show Odometer)

```
I'm a car!
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?A
Accelerating...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?
```

Output

```
The car has driven 0 kilometers
```

This was incorrect, because I expected the car to have traveled at least some distance after accelerating. However, the odometer stayed at 0.

Understanding the Issue

When I first tested the car simulation and used the sequence A → B → B → O, the program said the car had driven 0 kilometers. This result didn't seem right because I had accelerated the car, so I expected it to have moved forward.

After reviewing the code, I realized that the car's speed does increase with acceleration, but the distance (odometer) is only updated when the step() method is called. Without calling step(), the car doesn't simulate time passing, and so it doesn't "move." Since I didn't give the car time to move before braking, it remained stationary, and the odometer stayed at 0. This wasn't a syntax or crash error — it was a **logical error** in how the simulation was used.

Type of Error: Logical Error

This issue wasn't a syntax error — the code ran without crashing. Instead, it was a **logical error**. That means the code executed, but the behavior didn't match what I logically expected. The car's speed increased, but it didn't move forward unless the step() method was called, which simulates time and updates the odometer.

Also, when the brake method was used more than once, the speed could go into negative values, which doesn't make sense for a real car.

What I Learned About Debugging

From this task, I learned that debugging isn't just about fixing broken code — it's also about making sure the logic and behavior match what the user expects. In this case, the code ran without errors, but the sequence of actions didn't reflect realistic car behavior.

By stepping through the simulation, I understood that state changes (like speed) only affect movement when paired with time progression (step()). Debugging taught me to think about how methods interact and when they're supposed to be called. I also learned to carefully trace how each part of the code updates the car's status — speed, distance, and time — and that missing one of those steps can lead to unexpected results.

Task: Fixing the Problem

The purpose of the brake() method is to slow the car down by reducing its speed.

Originally, the code just subtracted 5 from the speed every time brake() was called. But this could make the speed go **below 0**, which doesn't make sense — a car can't go at negative speed. The new version of the method adds a check:

```
    def brake(self):
        if self.speed >= 5:
            self.speed -= 5
        else:
            self.speed = 0
```

Here's what it does:

- If the current speed is **5 or more**, it safely reduces the speed by 5.
- If the speed is **less than 5**, instead of going negative, it simply sets the speed to **0**.

This way, the speed will **never drop below zero**, and the simulation behaves more realistically — just like a real car that stops but doesn't go into reverse unless explicitly told to.

Output

```
I'm a car!
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?A
Accelerating...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?O
The car has driven 5 kilometers
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?
```

Exercise 3: Stepping Through the Code

In this exercise, I learned how to **step through the code line by line using a debugger** — without needing to place a breakpoint on every line. This technique allowed me to observe how the program executed in real-time and better understand how the flow of control moves through loops and functions.

To begin, I restarted the debugger and ran the car program. In the console, I typed a to accelerate the car. The debugger paused at my initial breakpoint. From there, I used the **"Step Over"** button to move to the next line of code without entering functions. I kept clicking "Step Over" until I reached the line my_car.accelerate().

At that point, I wanted to understand exactly what the accelerate() method did internally. So instead of stepping over, I clicked the **"Step Into"** button. This took me inside the method itself, showing me how the speed increased. This helped me visualize what the function was doing under the hood.

I also learned that if I keep using **"Step Over"**, the program will move to the next top-level line without diving into function bodies. But if I really want to **watch what a method is doing**, I have to use **"Step Into"**. If I go too deep, I can always use **"Step Out"** to return to the previous level.

This experience taught me that **stepping through code is a powerful way to debug**. It allowed me to:

- Follow the logic step by step.
- Choose when to explore functions deeply.
- Better understand how loops and conditionals are executed.

Using this approach, I was able to pinpoint exactly where and how my code updated the car's speed, odometer, and time. It gave me much more control and insight than just relying on print statements

Exercise 4: Watching Variables or Expressions

In this exercise, I explored how to use the **Watch** feature in Visual Studio Code's debugger to monitor specific variables and expressions during code execution. Instead of printing values manually, I learned a more efficient way to track what's happening inside my program.

I added my_car.speed to the **Watch panel**, which let me view the car's speed in real-time as I stepped through the code. This saved me from digging through the "Variables" section or relying on extra print statements. I could also enter logical expressions like my_car.speed > 0 to watch whether certain conditions were true as the program ran.

Using the Watch view gave me a clear and focused look at only the variables I cared about. I didn't have to clutter my code with debug prints — everything I needed was visible while stepping through with the debugger.

When I was done, I ended the debugging session by clicking the red stop button in the top-right corner of VS Code. This returned me to the regular editor view.

Overall, this exercise showed me how powerful the debugger can be. Instead of printing and guessing, I can now **watch and understand** what's happening in my code step by step — saving time and avoiding confusion.

Section 2. Properties using the @property decorator

In Python, the @property decorator is used to turn a method into an **attribute-like property**. This allows us to **call methods without parentheses** and treat them like variables, making code cleaner and more readable.

Instead of calling object.get_value(), you can just use object.value

Understanding the Task

In this task, I was asked to use the @property decorator to filter and return **uncompleted tasks** from a to-do list.

The goal was:

- To define a method uncompleted_tasks() in the TaskList class.
- Use @property so that I can access task_list.uncompleted_tasks like a variable, even though it's a method behind the scenes.
- Update the view_tasks() method to use this property and only show tasks that are **not yet completed**.

Source Code

```
"""Property attribute is used to use the method as attribute
In this case this method of getting in completed tasks will be used as attribute and not method"""
@property
def uncompleted_tasks(self) -> list[Task]:
    # returning only the tasks that are not completed
    return [task for task in self.tasks if not task.completed]
def view_tasks(self) -> None:
    # Show only the tasks that are still to be done
    if not self.uncompleted_tasks: # checking if there are any pending tasks available
        print("No tasks to show.")
    else:
        print("The following tasks are still to be done:")
        for task in self.uncompleted_tasks:
            # Get the correct index from the original task list
            ix = self.tasks.index(task)
            print(f"{ix+1}: {task}")  # Print index and task details
```

Explanation

The uncompleted_tasks() method:

- Is now used like a variable (self.uncompleted_tasks) instead of self.uncompleted_tasks().
- This makes code inside view_tasks() cleaner and easier to read.

In the view_tasks() method, this property is used to:

- Check if there are any uncompleted tasks
- Loop through them and print only the tasks that are still pending

The index is retrieved from the original self.tasks list to keep numbering consistent.

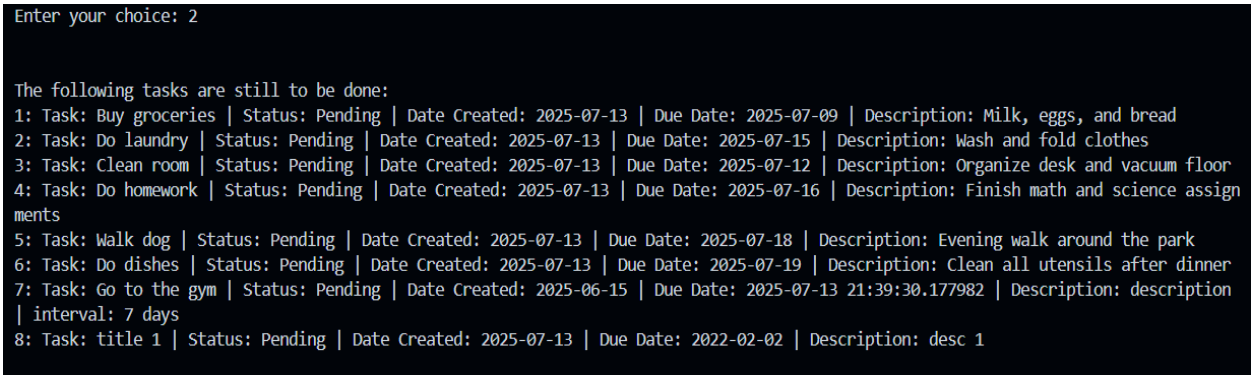
Why Is This Useful?

- It hides logic behind a simple interface

- Improves readability
- Maintains proper encapsulation
- Make code easier to maintain and extend later

Output

Choosing 2 to view the tasks



All pending tasks are shown only

Section 3. Implementing Persistence

Persistence means **saving data** so that it remains available **even after the program ends**. In other words, the data stays alive (persistent) between runs of the program.

For example:
If you create a task list and close the app, persistence allows that list to be saved and reloaded the next time you open it.

In Python, **persistence** can be achieved in several ways, it depends on how structured your data is and what your app needs. Here are the most common methods:

1. **Text Files (.txt)**

   You can save plain text data in a file and read it later. This method is simple and useful for basic lists or logs.

2. **CSV Files (.csv)**

   Ideal for tabular data, each row represents a record, and each column a field. It's commonly used when storing structured data like tasks, scores, or tables.

3. **JSON Files (.json)**

   This format allows you to store data in key-value pairs (dictionaries) or lists. It's perfect for saving complex objects like task lists with multiple fields (title, due date, completed status, etc.).

4. **Databases (e.g., SQLite, MySQL)**

   For larger or more complex applications, using a database is a better approach. Databases provide advanced features like search, filtering, sorting, and relations between different data types.

Exercise 1: DAO

Understanding the Task

In this task, I had to **modularize** the part of the code responsible for creating sample tasks by moving it into a separate class, following the **DAO design pattern**. This helps separate the **data management logic** from the rest of the application.

Instead of keeping the propagate_task_list() function in the main.py, I removed it and created a new class called TaskTestDAO. This class is responsible for **pretending to load tasks from a file**, even though no real file-saving is happening yet.

Then, I updated the main function to:

- Let the user **choose when to load/save tasks**

- Ask for a file path (just for simulation)
- Create an object of TaskTestDAO
- Use get_all_tasks() to load predefined tasks and add them to the current task list

Source Code

```
class TaskTestDAO:
    def __init__(self, storage_path: str) -> None:
        self.storage_path = storage_path
    def get_all_tasks(self) -> list[Task]:

        # sample one-time tasks
        task_list =[
        Task("Buy groceries", "Milk, eggs, and bread", datetime.datetime.now().date() - datetime.timedelta(days=4)),
        Task("Do laundry", "Wash and fold clothes", datetime.datetime.now().date() + datetime.timedelta(days=2)),
        Task("Clean room", "Organize desk and vacuum floor", datetime.datetime.now().date() - datetime.timedelta
        (days=1)),
        Task("Do homework", "Finish math and science assignments", datetime.datetime.now().date() + datetime.timedelta
        (days=3)),
        Task("Walk dog", "Evening walk around the park", datetime.datetime.now().date() + datetime.timedelta(days=5)),
        Task("Do dishes", "Clean all utensils after dinner", datetime.datetime.now().date() + datetime.timedelta(days=6))
        ]
        # sample recurring task
        r_task = RecurringTask("Go to the gym", datetime.datetime.now(),datetime.timedelta(days=7), 8)
        # propagate the recurring task with some completed dates
        r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=7))
        r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=14))
        r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=22))
        r_task.date_created = datetime.datetime.now() - datetime.timedelta(days=28)
        task_list.append(r_task)
        return task_list
```

Explanation

The DAO design pattern separates **data access** logic (like loading and saving) from the rest of the application. This helps in:

- Keeping the main.py file clean
- Making future updates easier (e.g., connecting to a real database or CSV file)
- Improving **code maintainability** and **testability**

In the TaskTestDAO class:

- The get_all_tasks() method simulates loading data by returning a list of sample Task and RecurringTask objects.
- The save_all_tasks() method is empty for now, just a placeholder for future saving functionality.

By moving task-loading logic here, I made the application more **organized** and **realistic**, like how professional apps are built.

Output

Thesen are the test tasks, as no new task is added.

Exercise 2: CSV Persistence

Serialization

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. The main purpose of serializing an object is to be able to recreate it when needed.

In my case, I am going to serialize the tasks to a CSV file.

For this reason, I have created a file TaskCsvDAO.py. The __init__ method sets up the CSV file path by combining the folder of the current Python file with the provided file name, ensuring the file is saved or accessed in the correct location. It also initializes the expected column headers (fieldnames) that define the structure of the task data, including fields like title, type, due date, and completion status. This setup prepares the class to read from or write to the CSV file properly.

```
def __init__(self, storage_path: str) -> None:
    # gets the file path and joins it
    self.storage_path = os.path.join(os.path.dirname(__file__), storage_path)

    # initialize fieldnames
    self.fieldnames = ["title", "type", "date_due", "completed", "interval", "completed_dates", "date_created"]
```

Task A: Complete get_all_tasks() functionality

Understanding the Task

In this task, I was required to **load all saved tasks from a CSV file** and convert them back into proper Task or RecurringTask objects. This simulates real **file persistence**, where tasks previously saved to a file are reloaded into the program.

Instead of hardcoding tasks (like in TaskTestDAO), now I'm using TaskCsvDAO to:

- Open the .csv file,

- Read each row
    - Get the type of the task
    - Get the title, due date, date created, interval (if present)
    - Convert the string format of the date into datetime object to be used further
- Check if the row is a normal task or a recurring one,
- Rebuild the appropriate object using the row's data (like title, due date, etc.),
- Do the same processing for all the rows present in file
- And finally, return a list of all such tasks.

Source Code

```
    def get_all_tasks(self) -> list[Task]:
        task_list = []
        with open(self.storage_path, "r") as file:
            reader = csv.DictReader(file)
            for row in reader:
                # getting each value from record one by one
                # Get the type of task (either Task or RecurringTask)
                task_type = row["type"]

                 # Get title and due date as string
                title = row["title"]
                date_due_str = row["date_due"]
                date_due = None

                # Convert the due date string into a datetime object (handling different formats)
                if date_due_str != "":
                    if "-" in date_due_str:
                        # Format is likely YYYY-MM-DD
                        date_due = datetime.datetime.strptime(date_due_str, "%Y-%m-%d").date()
                    elif "/" in date_due_str:
                        # Format is likely DD/MM/YYYY
                        date_due = datetime.datetime.strptime(date_due_str, "%d/%m/%Y").date()
                    else:
                        print(f"Unknown date format: {date_due_str}")
                        date_due = None  # or set a default/fallback

                # Check if the task was marked as completed
                completed = row["completed"] == "True"  # convert string to boolean
```

```
            # Handle the created date in a similar way
            date_created = None
            date_created_str = row["date_created"]

            if date_created_str != "":
                if "-" in date_created_str:
                    # Format is likely YYYY-MM-DD
                    date_created = datetime.datetime.strptime(date_created_str, "%Y-%m-%d").date()
                elif "/" in date_created_str:
                    # Format is likely DD/MM/YYYY
                    date_created = datetime.datetime.strptime(date_created_str, "%d/%m/%Y").date()
                else:
                    print(f"Unknown date format: {date_created_str}")
                    date_created = None  # or set a default/fallback


            # If task is RecurringTask, create it accordingly
            if task_type == "RecurringTask":
                interval_days = int(row["interval"].split()[0])  # get number from "7 days"
                interval = datetime.timedelta(days=interval_days)

                # parse completed_dates from comma-separated string
                completed_dates = []
                if row["completed_dates"]:
                    date_strs = row["completed_dates"].split(",")
                    completed_dates = [datetime.datetime.strptime(d.strip(), "%Y-%m-%d") for d in date_strs]
```

```
                    task = RecurringTask(title,'', date_due, interval)
                    task.completed = completed
                    task.date_created = date_created
                    task.completed_dates = completed_dates

                else:  # Handle regular Task object
                    task = Task(title,'', date_due)
                    task.completed = completed
                    task.date_created = date_created

                task_list.append(task)

        return task_list
```

Explanation

The get_all_tasks() method does the following:

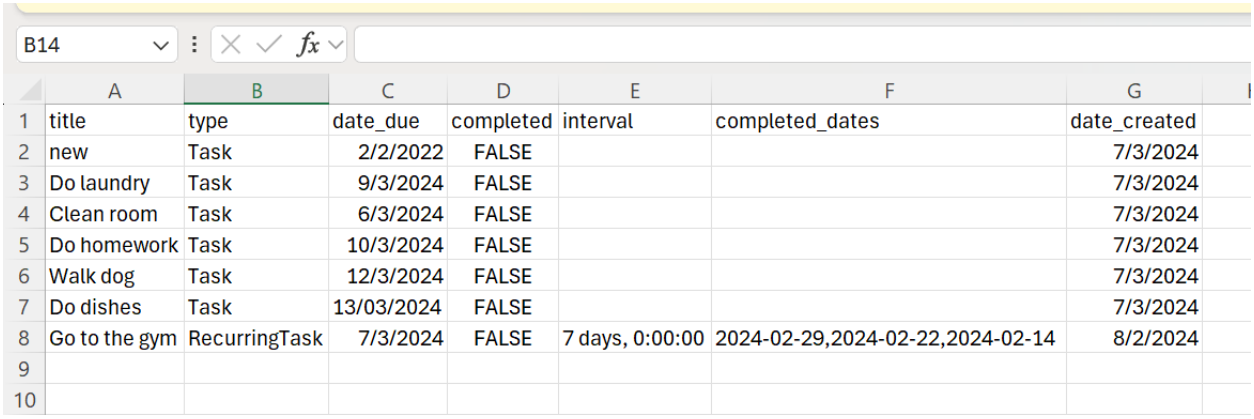
- Opens the CSV file and reads it using csv.DictReader, so each row becomes a dictionary.
- Check the type column to decide if it's a regular Task or a RecurringTask.
- Parses important values like:

- date_due and date_created using two possible formats (YYYY-MM-DD or DD/MM/YYYY).
- Converts the completed value from string to Boolean.
- Extracts and parses the interval (for recurring tasks).
- Splits and parses multiple completed_dates into a list of datetime objects.

- Build the task object (Task or RecurringTask) and adds it to a list.

This way, all task data from the CSV file is restored exactly as it was, making the program **persistent and usable across sessions.**

CSV file



	A	B	C	D	E	F	G
1	title	type	date_due	completed	interval	completed_dates	date_created
2	new	Task	2/2/2022	FALSE			7/3/2024
3	Do laundry	Task	9/3/2024	FALSE			7/3/2024
4	Clean room	Task	6/3/2024	FALSE			7/3/2024
5	Do homework	Task	10/3/2024	FALSE			7/3/2024
6	Walk dog	Task	12/3/2024	FALSE			7/3/2024
7	Do dishes	Task	13/03/2024	FALSE			7/3/2024
8	Go to the gym	RecurringTask	7/3/2024	FALSE	7 days, 0:00:00	2024-02-29,2024-02-22,2024-02-14	8/2/2024
9							
10							

First, there are no tasks present in the program

```
To-Do List Manager
 1.  Add a task
 2.  View tasks
 3.  Remove a task
 4.  Mark as completed
 5.  Change title of task
 6.  Change description of task
 7.  Show over due tasks
 8.  Change Due date of task
 9.  Load Tasks from CSV
10. Save Tasks to CSV
11. Load Data from Pickle File
12. Save Data to Pickle File
13. Get Owner Details
14. Quit
Enter your choice: 2


No tasks to show.
```

Choosing 9 to load data from csv file. Then entering the path of the file e.g. tasks.csv

```
Enter your choice: 9


Enter file path to load tasks e.g. tasks.txt: tasks.csv
loading data from file...
```

Choosing 2 to view tasks which are loaded from the file

```
Enter your choice: 2


The following tasks are still to be done:
1: Task: new | Status: Pending | Date Created: 2024-03-07 | Due Date: 2022-02-02 | Description: No description
2: Task: Do laundry | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-09 | Description: No description
3: Task: Clean room | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-06 | Description: No description
4: Task: Do homework | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-10 | Description: No description
5: Task: Walk dog | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-12 | Description: No description
6: Task: Do dishes | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-13 | Description: No description
7: Task: Go to the gym | Status: Pending | Date Created: 2024-02-08 | Due Date: 2024-03-07 | Description: No description | interval: 7
 days
8: Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-09-07 | Description: No description
```

Task B: Complete save_all_tasks() functionality

Understanding the Task

In this task, we were asked to implement the functionality to **save tasks to a CSV file**. The goal is to persist user data so that the list of tasks (both regular and recurring) can be stored and accessed later. The function should also:

- Write each task's details to the CSV properly formatted.
- Check and **avoid duplicate entries** using a unique identifier (in this case, the task title).
- Distinguish between regular Task and RecurringTask using the type column.
- Properly handle different task attributes, especially converting date fields and lists (like completed_dates) into strings.

Source Code

```
87      def save_all_tasks(self, tasks: list[Task]) -> None:
88          # create a set of existing task titles to check for duplicates
89          existing_titles = set()
90
91          # If file exists, load existing tasks to check for duplicates
92          if os.path.exists(self.storage_path):
93              with open(self.storage_path, "r", encoding="utf-8") as file:
94                  reader = csv.DictReader(file)
95                  for row in reader:
96                      # Add the title of each existing task to the set
97                      # This helps us avoid saving duplicate tasks later
98                      existing_titles.add(row["title"])  # Assuming title is unique
99
100         # Open the file in append mode so we can add new tasks without deleting old ones
101         with open(self.storage_path, "a", newline='', encoding="utf-8") as file:
102             writer = csv.DictWriter(file, fieldnames=self.fieldnames)
103
104              # If the file is empty, we add a header row to it first
105             if os.stat(self.storage_path).st_size == 0:
106                 writer.writeheader()
107
108             # Go through each task in the list
109             for task in tasks:
110                  # If this task is already in the file (based on title), skip it
111                 if task.title in existing_titles:
112                     continue  # Skip duplicate tasks
113
114
115                 # This row dictionary will hold all the task details to be written
116                 row = {}
```

```
            # Add basic task information to the row
            row["title"] = task.title
            row["completed"] = str(task.completed)
            # changing the format fo date to YYYY-MM-DD or DD/MM/YYYY
            row["date_due"] = task.due_date.strftime("%Y-%m-%d")
            row["date_created"] = task.date_created.strftime("%d/%m/%Y")

            # Check if the task is a recurring task and handle accordingly
            if isinstance(task, RecurringTask):
                row["type"] = "RecurringTask"
                # If the interval has a 'days' attribute (like timedelta), extract it
                interval = task.interval.days if hasattr(task.interval, "days") else task.interval
                row["interval"] = f"{interval} days"

                # Convert the list of completed dates into a single comma-separated string
                row["completed_dates"] = ",".join(
                    [d.strftime("%Y-%m-%d") for d in task.completed_dates]
                )
            else:
                # If it's just a normal one-time Task, keep these fields empty
                row["type"] = "Task"
                row["interval"] = ""
                row["completed_dates"] = ""

            # Finally, write the task details to the CSV file
            writer.writerow(row)
```

Explanation

This save_all_tasks method is designed to store all tasks in a CSV file while avoiding duplicates. It first checks if the file already exists and reads any existing task titles to prevent writing the same task multiple times. It then opens the file in append mode and writes a header if the file is empty. For each task in the list, it skips saving if the task title is already present in the file. It then prepares the task's data for storage, converting relevant attributes like due_date and date_created into string format using strftime. If the task is a RecurringTask, it also includes additional fields such as the interval and a comma-separated list of completed dates. If it’s a normal task, those extra fields are left blank. Finally, each prepared task is written as a row in the CSV file, ensuring a clean and structured representation of the task data. This method makes the application’s task data persistent and reusable.

Output

CSV

A1 title

	A	B	C	D	E	F	G	H
1	title	type	date_due	completed	interval	completed_dates	date_created	
2	new	Task	2/2/2022	FALSE			7/3/2024	
3	Do laundry	Task	9/3/2024	FALSE			7/3/2024	
4	Clean room	Task	6/3/2024	FALSE			7/3/2024	
5	Do homework	Task	10/3/2024	FALSE			7/3/2024	
6	Walk dog	Task	12/3/2024	FALSE			7/3/2024	
7	Do dishes	Task	13/03/2024	FALSE			7/3/2024	
8	Go to the gym	RecurringTask	7/3/2024	FALSE	7 days, 0:00:00	2024-02-29,2024-02-22,2024-02-14	8/2/2024	
9	Buy groceries	Task	7/9/2025	FALSE			13/07/2025	
10	t new	Task	2/2/2022	FALSE			13/07/2025	
11								
12								

Choosing opt 1 to add a task, then choosing 1 to add a one time task.

```
Enter your choice: 1


1. One Time Task
2. Recurring Task
3. Back
Enter your choice: 1
Enter title of task: t new
Enter the description: t new
Enter a due date (YYYY-MM-DD): 2022-2-2
't new' has been added to your to-do list.
```

```
Enter your choice: 2


The following tasks are still to be done:
1: Task: new | Status: Pending | Date Created: 2024-03-07 | Due Date: 2022-02-02 | Description: No description
2: Task: Do laundry | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-09 | Description: No description
3: Task: Clean room | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-06 | Description: No description
4: Task: Do homework | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-10 | Description: No description
5: Task: Walk dog | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-12 | Description: No description
6: Task: Do dishes | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-13 | Description: No description
7: Task: Go to the gym | Status: Pending | Date Created: 2024-02-08 | Due Date: 2024-03-07 | Description: No description | interval: 7
 days
8: Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-09-07 | Description: No description
9: Task: t new | Status: Pending | Date Created: 2025-07-13 | Due Date: 2022-02-02 | Description: No description
10: Task: t new | Status: Pending | Date Created: 2025-07-14 | Due Date: 2022-02-02 | Description: t new
```

We can see the duplicate of task with title ‘t new’. This output is from the tasks saved in program, not the file.

Choose 10 to save data to file. After that the tasks in the csv are:

A1 title

	A	B	C	D	E	F	G	H	I
1	title	type	date_due	completed	interval	completed	date_created		
2	new	Task	2/2/2022	FALSE			7/3/2024		
3	Do laundry	Task	9/3/2024	FALSE			7/3/2024		
4	Clean room	Task	6/3/2024	FALSE			7/3/2024		
5	Do homework	Task	########	FALSE			7/3/2024		
6	Walk dog	Task	########	FALSE			7/3/2024		
7	Do dishes	Task	13/03/202	FALSE			7/3/2024		
8	Go to the gym	RecurringTask	7/3/2024	FALSE	7 days, 0:0	2024-02-2	8/2/2024		
9	Buy groceries	Task	7/9/2025	FALSE			13/07/2025		
10	t new	Task	2/2/2022	FALSE			13/07/2025		
11									
12									

Which shows that duplicate is not saved.

Example 2

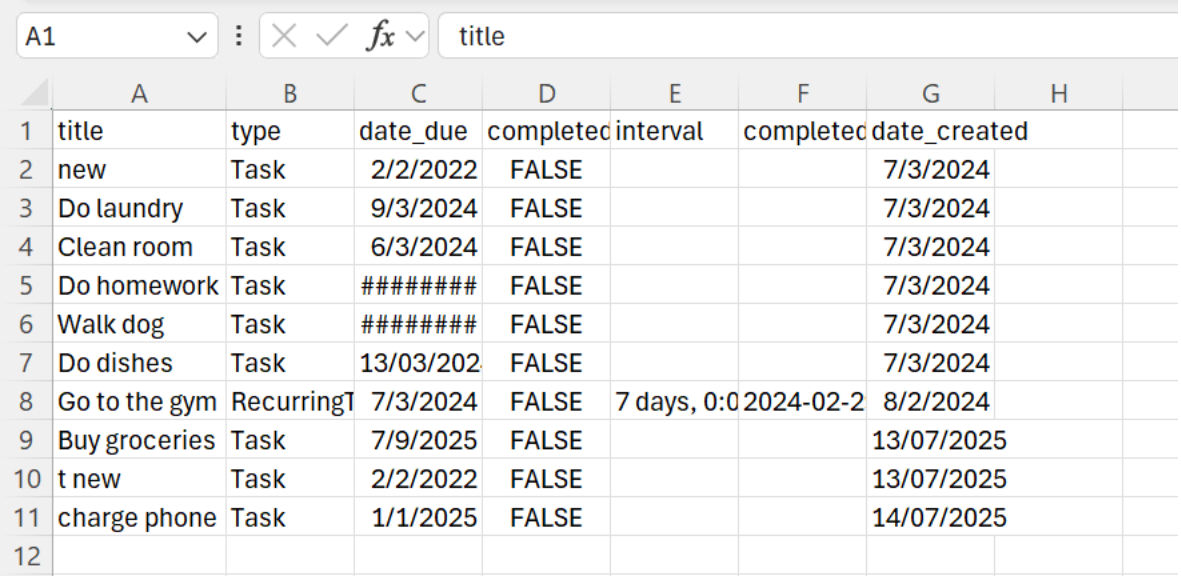
Choosing 1 and then 1 to add a one time task.

```
Enter your choice: 1


1. One Time Task
2. Recurring Task
3. Back
Enter your choice: 1
Enter title of task: charge phone
Enter the description:
Enter a due date (YYYY-MM-DD): 2025-1-1
'charge phone' has been added to your to-do list.
```

Choosing 10 to save data to file

```
Enter your choice: 10


Enter file path to load tasks (e.g. tasks.csv): tasks.csv
saving data to file...
```

Output in csv



	A	B	C	D	E	F	G	H
1	title	type	date_due	completed	interval	completed	date_created	
2	new	Task	2/2/2022	FALSE			7/3/2024	
3	Do laundry	Task	9/3/2024	FALSE			7/3/2024	
4	Clean room	Task	6/3/2024	FALSE			7/3/2024	
5	Do homework	Task	########	FALSE			7/3/2024	
6	Walk dog	Task	########	FALSE			7/3/2024	
7	Do dishes	Task	13/03/202	FALSE			7/3/2024	
8	Go to the gym	RecurringT	7/3/2024	FALSE	7 days, 0:0	2024-02-2	8/2/2024	
9	Buy groceries	Task	7/9/2025	FALSE			13/07/2025	
10	t new	Task	2/2/2022	FALSE			13/07/2025	
11	charge phone	Task	1/1/2025	FALSE			14/07/2025	
12								

We can see that the task ‘charge phone’ is saved in the file

Summary

This method ensures that:

- Tasks are written to a file in a clean, structured way.
- Duplicates are avoided.
- Both Task and RecurringTask objects are properly handled.
- Dates and lists are converted into formats suitable for CSV storage.

Task C: Update single record in the file

Understanding the Task

The goal of this task is to enable updating a single task record in the CSV file. When a user modifies a task (like marking it as completed, changing the title, updating the description, or editing the due date), these changes should be saved persistently. The update must

correctly overwrite only the specific record in the file without affecting the others. This ensures that the CSV file always reflects the most current state of all tasks.

Source Code

Explanation

- The method takes two arguments:
    - updated_task: the modified task object (with new values)
    - old_title: the title used to find the original task in the CSV
- First, it opens the CSV file and reads all rows.
- It checks each row's title to find the task that matches old_title.
- When found:
    - It builds a new row with the updated task data (including type, dates, etc.).
    - Adds this new row to a temporary list.
- All other rows are added to the list unchanged.
- After looping, it writes the entire list back to the CSV file, replacing the original file with the updated content.
- A success message is printed if the update is completed; otherwise, it alerts the user that the task wasn't found.

The method also properly handles different task types:

- Regular Task
- RecurringTask

Usage

Marking a Task as Completed (Choice 4)

- Displays all tasks
- Takes task number from user
- Marks the task as completed
- Calls update_task() to save the updated task status in the CSV

```
        while True:
            index = input("Enter the number of the task to mark as completed: ")

            if index.isdigit():
                # Convert to actual integer
                index = int(index)
                if index > 0 and index <= len(task_list.tasks) :
                    task = task_list.get_task(index)
                    old_title = task.title
                    task = task_list.get_task(index)
                    task.mark_completed()
                    # task_list.tasks[index-1].mark_completed()
                    dao = TaskCsvDAO(file_path)
                    # dao = TaskCsvDAO("tasks.csv")
                    dao.update_task(task, old_title)
                    break  # Exit the loop since input is valid
                else:
                    print("Invalid task number. Please try again.\n")
                    continue

            else:
                print("Invalid input. Please enter a number like 1, 2, 3...")
```

Changing Task Title (Choice 5)

- Displays all tasks
- Asks user to enter a task number and a new title
- Updates the task's title

- Calls update_task() to save the new title to the CSV

```
while True:
    # user sees 1 based indexing
    index = input("Enter the number of the task to change title: ")

    if index.isdigit():
        # Convert to actual integer
        index = int(index)
        if index > 0 and index <= len(task_list.tasks) :
            new_title = input("Enter the new title: ")

            task = task_list.get_task(index)
            old_title = task.title
            task.change_title(new_title)
            # task_list.tasks[index-1].change_title(new_title)
            dao = TaskCsvDAO(file_path)
            # dao = TaskCsvDAO("tasks.csv")
            dao.update_task(task, old_title)
            break  # Exit the loop since input is valid
        else:
            print("Invalid task number. Please try again.\n")
            continue

    else:
        print("Invalid input. Please enter a number like 1, 2, 3...")
```

Changing Description

- Prompts for task number and new description
- Updates the description
- Calls update_task() with the changed task

```
        while True:
            index = input("Enter the number of the task to change description: ")

            if index.isdigit():
                # Convert to actual integer
                index = int(index)
                if index > 0 and index <= len(task_list.tasks) :
                    new_desc = input("Enter the new description: ")
                    task = task_list.get_task(index)
                    old_title = task.title
                    task.change_description(new_desc)
                    # task_list.tasks[index-1].change_description(new_desc)
                    dao = TaskCsvDAO(file_path)
                    # dao = TaskCsvDAO("tasks.csv")
                    dao.update_task(task, old_title)
                    break  # Exit the loop since input is valid
                else:
                    print("Invalid task number. Please try again.\n")
                    continue

            else:
                print("Invalid input. Please enter a number like 1, 2, 3...")
```

Changing Due Date

- Prompts for task number and new due date
- Updates the due date
- Calls update_task() to persist changes

```
while True:
    index = input("Enter the number of the task to change due date: ")

    if index.isdigit():
        # Convert to actual integer
        index = int(index)
        if index > 0 and index <= len(task_list.tasks) :
            # get new due date
            task = task_list.get_task(index)
            old_title = task.title
            new_date = input("Enter the new due date (YYYY-MM-DD): ")
            due_date = datetime.datetime.strptime(new_date, "%Y-%m-%d").date
            # task_list.tasks[index-1].change_due_date(due_date)
            task.change_due_date(due_date)
            # dao = TaskCsvDAO("tasks.csv")
            dao = TaskCsvDAO(file_path)
            dao.update_task(task, old_title)
            break  # Exit the loop since input is valid
        else:
            print("Invalid task number. Please try again.\n")
            continue
    else:
        print("Invalid input. Please enter a number like 1, 2, 3...")
```

Explanation

Exercise 3: Serialization using Pickle

Pickle is a built-in Python module that allows you to **serialize** (convert Python objects into a byte stream) and **deserialize** (load them back into Python objects). It's helpful when you want to save complex objects (like custom classes or lists of objects) to a file so they can be reused later. Unlike CSV or JSON, which require you to manually handle each attribute, Pickle handles the entire object, preserving its structure, type, and values.

The format of pickle file looks like this:

In a **pickle file**, data is saved in **binary format**, specifically, as a stream of bytes that represents Python objects.

Understanding the Task

The goal of this exercise is to implement a new way of saving and loading tasks using the pickle module instead of CSV. You're asked to create a new class called TaskPickleDAO that reads from and writes to a pickle file. This approach makes it easier to handle complex objects (like tasks with nested attributes or classes), especially as your code grows and changes. This task aims to improve data persistence in a more flexible and Pythonic way.

Source Code

```python
class TaskPickleDAO:
    def __init__(self, storage_path: str) -> None:
        # Combine the current directory with the file name to get the full path
        self.storage_path = os.path.join(os.path.dirname(__file__), storage_path)


    def get_all_tasks(self) -> list[Task]:
        """Load all tasks from the pickle file."""
        # If the pickle file doesn't exist, show a message and return an empty list
        if not os.path.exists(self.storage_path):
            print("[i] No pickle file found. Returning empty task list.")
            return []
        # Open the pickle file in binary read mode
        with open(self.storage_path, "rb") as file:
            # Load the entire list of task objects from the file
            task_list = pickle.load(file)
        return task_list

    def save_all_tasks(self, tasks: list[Task]) -> None:
        """Save all tasks to the pickle file."""
        # Open the pickle file in binary write mode (this will overwrite existing data)
        with open(self.storage_path, "wb") as file:
            # serialize the entire list of tasks into the file
            pickle.dump(tasks, file)
        print(f"[✓] Tasks saved to {self.storage_path} successfully.")

```

Explanation

The TaskPickleDAO class is built to handle saving and loading of task data using the pickle module. In the constructor, it sets the path where the pickle file will be saved, ensuring it's relative to the file location for better file organization. The get_all_tasks method checks whether the pickle file exists; if it doesn't, it returns an empty list, otherwise it loads and returns the saved tasks using pickle.load(). The save_all_tasks method takes a list of tasks and writes it to the specified file using pickle.dump(). This approach ensures that all task information, including objects like RecurringTask , are stored and restored accurately, without having to manually write or read each attribute. It's especially useful as the structure of tasks grows more complex.

Output

Choosing 11 to load data from pickle file

```
To-Do List Manager
 1.  Add a task
 2.  View tasks
 3.  Remove a task
 4.  Mark as completed
 5.  Change title of task
 6.  Change description of task
 7.  Show over due tasks
 8.  Change Due date of task
 9.  Load Tasks from CSV
10. Save Tasks to CSV
11. Load Data from Pickle File
12. Save Data to Pickle File
13. Get Owner Details
14. Quit
Enter your choice: 11


Enter file path to load tasks (e.g. tasks.pkl): tasks.pkl
[√] Tasks loaded from pickle.
```

Choosing 2 to view tasks

```
Enter your choice: 2


The following tasks are still to be done:
1: Task: new | Status: Pending | Date Created: 2024-03-07 | Due Date: 2022-02-02 | Description: No description
2: Task: Do laundry | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-09 | Description: No description
3: Task: Clean room | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-06 | Description: No description
4: Task: Do homework | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-10 | Description: No description
5: Task: Walk dog | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-12 | Description: No description
6: Task: Do dishes | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-13 | Description: No description
7: Task: Go to the gym | Status: Pending | Date Created: 2024-02-08 | Due Date: 2024-03-07 | Description: No description | interval: 7
 days
8: Task: t1 | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-02-02 | Description: No description
9: Task: fgh | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-02-02 | Description: No description
10: Task: 1asf | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-02-02 | Description: No description
11: Task: 111 | Status: Pending | Date Created: 2025-07-12 | Due Date: 2222-02-02 | Description: No description
13: Task: one new | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-04-04 | Description: No description
```

Choosing 1 and 2 to add a recurring task

```
Enter your choice: 1


1. One Time Task
2. Recurring Task
3. Back
Enter your choice: 2
Enter title of task: charge phone
Enter the description:
Enter a due date (YYYY-MM-DD): 2022-4-4
Enter the interval in days: 4
'charge phone' has been added to your to-do list.
```

Choosing 12 to save data to pickle file

```
Enter your choice: 12


[✓] Tasks saved to d:\C data\Desktop\latest\py asgmt\ToDoAppWeek6\tasks.pkl successfully.
To-Do List Manager
```

Choosing 2 to see whether the task is saved in pickle file or not

```
Enter your choice: 2


The following tasks are still to be done:
1: Task: new | Status: Pending | Date Created: 2024-03-07 | Due Date: 2022-02-02 | Description: No description
2: Task: Do laundry | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-09 | Description: No description
3: Task: Clean room | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-06 | Description: No description
4: Task: Do homework | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-10 | Description: No description
5: Task: Walk dog | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-12 | Description: No description
6: Task: Do dishes | Status: Pending | Date Created: 2024-03-07 | Due Date: 2024-03-13 | Description: No description
7: Task: Go to the gym | Status: Pending | Date Created: 2024-02-08 | Due Date: 2024-03-07 | Description: No description | interval: 7
 days
8: Task: t1 | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-02-02 | Description: No description
9: Task: fgh | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-02-02 | Description: No description
10: Task: 1asf | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-02-02 | Description: No description
11: Task: 111 | Status: Pending | Date Created: 2025-07-12 | Due Date: 2222-02-02 | Description: No description
13: Task: one new | Status: Pending | Date Created: 2025-07-12 | Due Date: 2022-04-04 | Description: No description
14: Task: charge phone | Status: Pending | Date Created: 2025-07-14 | Due Date: 2022-04-04 | Description: No description | interval: 4
 days
```

It is saved in the file

Week 7

Section 1. SOLID Principles

There are several OOP principles that, if followed, can help you to write better OO code. A commonly used set of principles follows the SOLID acronym. These principles are:

- S: Single Responsibility Principle. Ensures classes have a single responsibility, enhancing modularity.
- O: Open/Closed Principle. Allows for extension of classes without modifying existing code.
- L: Leskov Substitution Principle. Subclasses can substitute super classes without altering functionality.
- I: Interface Segregation Principle. Clients only depend on the interfaces they use, promoting modularity.
- D: Dependency Inversion Principle. High-level modules depend on abstractions, not low-level details

Exercise 1. Single Responsibility Principle

Understanding the Task

In this exercise, I was introduced to the concept of the **Single Responsibility Principle** (SRP), which basically means that **every class should do one job only**. The idea is that if a class has too many responsibilities, it becomes harder to maintain, reuse, or extend. So, I had to think about whether each part of my ToDoApp was respecting SRP. I reviewed classes like Task, TaskList, TaskCsvDAO, and main.py to check if they were handling only

one responsibility or doing too much. I noticed that while most classes followed SRP, the **main.py file was mixing user interface logic and business logic**, which breaks the rule.

Source Code

```
class Task:
    def __init__(self, title:str, description:str, due_date) -> None: ···
    def __str__(self) -> str: ···
    def mark_completed(self) -> None: ···
    def change_title(self, new_title) -> None: ···
    def change_description(self, new_description) -> None: ···
    def change_due_date(self, new_date)-> None: ···
    def __eq__(self, other): ···
```

```
class TaskList:
    def __init__(self, owner:Owner) -> None:
        self.owner = owner
        self.tasks = []
    def add_task(self, task:Task) -> None: ···
    def remove_task(self,index) -> None: ···
    """Property attribute is used to use the method as attribute···
    @property
    def uncompleted_tasks(self) -> list[Task]: ···
    def view_tasks(self) -> None: ···
    def view_over_due_tasks(self)->None: ···

    def get_task_by_title(self, title: str) -> Task: ···
    def get_task(self, index) -> Task: ···
```

```
class TaskCsvDAO:
    def __init__(self, storage_path: str) -> None: ···

    def get_all_tasks(self) -> list[Task]: ···

    def save_all_tasks(self, tasks: list[Task]) -> None: ···

    def update_task(self, updated_task: Task, old_title) -> None: ···
```

```
class TaskPickleDAO:
    def __init__(self, storage_path: str) -> None: ···


    def get_all_tasks(self) -> list[Task]: ···

    def save_all_tasks(self, tasks: list[Task]) -> None: ···
```

In the source code, it is seen that every class is serving one purpose only using its methods.

Explanation

The Single Responsibility Principle is part of clean code and good software design. In the context of my ToDoApp, each class mostly follows SRP:

- **Task and RecurringTask** handle storing task data of respective types.
- **TaskList** is responsible for managing the list of tasks (like adding, removing, or viewing them).
- **DAO classes** (TaskCsvDAO, TaskPickleDAO, etc.) deal only with saving and loading tasks from files.

But the main.py file is where SRP is clearly broken. It's doing multiple things: asking user input, displaying menus, processing logic, and calling other classes. This makes it harder to manage or update in the future. So, to follow SRP properly, we should **move the input and print-related stuff to a separate class**, like CommandLineUI, and keep the business logic in the controller or model classes.

Benefits of SRP in My ToDoApp

- Make the code **cleaner and easier to understand**.
- If I want to update one feature (like how tasks are saved), I don't have to dig through unrelated code.
- Make **testing** easier since each class can be tested in isolation.
- Help reduce **bugs** and confusion when the codebase grows.

- **Improves reusability** – I can reuse the same Task logic in other apps, without copying extra unrelated stuff.

Exercise 2. Open/Closed Principle

Understanding the Task

In this task, I had to understand how to apply the **Open/Closed Principle (OCP)** in my ToDoApp. The principle says that classes should be **open for extension but closed for modification**. This means instead of changing an existing class when adding new features, I should extend it by creating a new class. I realized that I've already been applying this idea when I created the RecurringTask class, which extends the base Task class by adding extra attributes and functionality like interval and completed_dates, all without changing the original Task class. Similarly, the exercise asks me to think about how I could improve other parts of my code (like the DAO classes) by using inheritance and abstraction to make future updates easier.

Explanation

The Open/Closed Principle helps make code more maintainable by encouraging extensions instead of changes. In my ToDoApp:

- The **Task class** is the base for all task-related functionality.
- The **RecurringTask class** extends Task and adds features like automatic due date updates and recurring intervals. I didn't touch the Task class while adding this, which means I followed OCP.

The same can be applied to the DAO layer. Instead of editing TaskCsvDAO or TaskPickleDAO every time I want to add a new storage type, I can create an abstract BaseTaskDAO class and make each DAO subclass inherit from it. This ensures consistent structure (interface) and allows me to add new behavior without touching the old code.

Benefits of Using OCP

- Avoid **breaking existing functionality** while adding new features.
- Keeps the original class **stable and reliable**.
- Supports **scalable** design, new features can be plugged in easily.
- Promotes **reuse of existing code** and reduces duplication.
- Encourages use of **polymorphism**, making the design more flexible and cleaner.

Exercise 3: Liskov Substitution Principle (LSP)

Understanding the Task

The goal was to make sure that wherever a Task object is used in the app, it should be replaceable with a RecurringTask object without breaking the app. This aligns with LSP, which says that subclasses should be able to stand in for their parent classes seamlessly.

Explanation:

In our ToDoApp, RecurringTask inherits from Task and shares the same structure, attributes like title, due_date, and methods like mark_completed(). Although RecurringTask overrides some functionality (e.g. updating due dates automatically), it still behaves as expected in every context where a regular task is used. This shows our app is correctly following LSP, we can substitute Task with RecurringTask and everything still works fine.

Benefits:

- Enables **flexible code reusability** with safe subclassing
- Reduces the chance of **bugs due to substitution**
- Keeps the **interface consistent** across the app

Exercise 4: Interface Segregation Principle (ISP)

Understanding the Task

The focus was to avoid writing large, all-purpose classes with too many unrelated methods. Instead, we aim to keep our class interfaces minimal, each class should only have methods that are relevant to its role.

Explanation

Even though Python doesn't use formal interfaces like Java or C#, we follow ISP by keeping our Task and RecurringTask classes clean and focused. Task has only necessary methods like change_title() or mark_completed(). RecurringTask builds on that by adding specific logic for recurrence. There are no unrelated or unused methods in either class. This helps ensure that each class is lightweight, specific, and easier to maintain.

Benefits

- Keeps each class **easier to understand**
- Prevents **overloading classes with unnecessary responsibilities**
- Makes future **testing and extension cleaner and faster**

Exercise 5: Dependency Inversion Principle (DIP)

Understanding the Task

This exercise was about ensuring that higher-level modules (like TaskList) don't depend on the concrete details of file formats or storage logic (like CSV or Pickle). Instead, they should rely on abstract interactions through a DAO.

Explanation

In the ToDoApp, the TaskList class doesn't directly save to a file, instead, it delegates that to a DAO (like TaskCsvDAO or TaskPickleDAO). This means if we ever change how tasks are saved (e.g. using a database or an API), the TaskList logic won't break, because it's not tightly coupled to any specific implementation. While Python doesn't have built-in interfaces, we can still achieve DIP by writing code that communicates through well-structured, abstract method contracts (e.g. get_all_tasks(), save_all_tasks()).

Benefits

- Improves **maintainability and testability**
- Allows for **easy swapping of data sources** (e.g. CSV to Pickle)
- Keeps higher-level logic **clean and isolated from low-level details**

Section 2. Exception Handling

Task: ToDoApp Modification by adding exception handling

Understanding the Task

In this task, we are improving the user experience by making our app more robust and error-tolerant. When a user tries to remove or modify a task using an invalid index (like a number that doesn't exist in the task list), the app currently crashes. Our goal is to **add exception handling** using Python's try, except, and raise keywords to prevent these crashes and instead provide helpful error messages to the user.

Example 1

```
    try:
        user_input = input("Enter the number of the task to remove: ")

        if not user_input.isdigit():
            raise ValueError("Please enter a number (e.g. 1, 2, 3...)")

        index = int(user_input)

        self.controller.remove_task(index)  # let controller do all logic
        print("Task removed successfully.\n")
        self.controller.view_tasks()
        break

    except ValueError as ve:
        print(f"[!] {ve}\n")
    except IndexError as ie:
        print(f"[!] {ie}\n")
```

Explanation

The provided code uses a try-except block around user input and task manipulation logic. Here's what happens:

- The app first **asks the user for a task number** to remove.
- If the input is **not a digit**, a ValueError is manually raised using raise ValueError(...).
- The input is then converted to an integer and passed to self.controller.remove_task(index).
- If this index is **not valid** (e.g., it's out of range), the controller will raise an IndexError.
- Both exceptions are caught, and a **friendly error message** is shown instead of crashing.
- If everything goes well, the task is removed, and the updated task list is shown.

This structure allows the app to **continue running smoothly**, even if the user makes a mistake.

Example 2

```
while True:
    index = input("Enter the number of the task to mark as completed: ")

    if index.isdigit():
        index = int(index)
        try:       You, yesterday • week 7 functionality done
            self.controller.mark_task_completed(index)
            print("[✓] Task marked as completed.\n")
            break
        except IndexError:
            print("Invalid task number. Please try again.\n")
    else:
        print("Invalid input. Please enter a number like 1, 2, 3...\n")
```

Explanation

- The user is asked to enter a number that refers to the task they want to mark as completed.
- isdigit() checks if the input is a number. If not, a clear error message is shown.
- If the input is a valid number, the program tries to mark the task as completed using the controller.
- If the task number is **out of range or invalid**, the program catches the IndexError and informs the user that the task number is incorrect.
- If the task is marked successfully, a success message is shown.

This logic ensures that the app doesn't crash when the user enters invalid task numbers or text.

Example 3

```
def _load_tasks_from_csv(self):
    if (not config.file_path):
        config.file_path = input("Enter file path to load tasks (e.g. tasks.csv): ")

    try:
        self.controller.load_tasks_from_csv(config.file_path)
        print("[✓] Tasks loaded from file.\n")
    except Exception as e:
        print(f"[!] Failed to load tasks: {e}\n")
```

Explanation

- This function checks whether the CSV file path (config.file_path) is already stored. If not, it asks the user to provide it.
- Then it tries to load tasks from the file using the controller's load_tasks_from_csv() method.
- If something goes wrong (like the file doesn't exist or is incorrectly formatted), it catches the error and prints a clear message.
- The use of Exception as e helps catch **any kind of error**, not just specific ones like FileNotFoundError.

Benefits of Exception Handling

- Prevents the program from crashing unexpectedly.
- Provides better user experience by giving clear instructions when input is incorrect.
- Makes the code more maintainable and easier to debug.
- Helps isolate bugs and ensures the app behaves reliably under edge cases.

Section 3. Improving ToDoApp according to SOLID Principles

Task: Modification of main module to separate the user interface from the business logic

Understanding the Task

In this task, I was required to **restructure my ToDoApp** to improve its design by clearly separating **user interface logic from business logic**. The idea was to make the code more modular, easier to maintain, and better aligned with **Object-Oriented Programming principles** like **Separation of Concerns**, **Single Responsibility Principle**, and **Low Coupling**.

I had to:

- Create a new class CommandLineUI in a separate ui module. This class is responsible for everything the user sees or interacts with: menus, inputs, and printed messages.
- Create another class TaskManagerController in a controllers module. This class handles the logic of how the app behaves, adding tasks, marking them complete, checking valid indexes, loading/saving data, etc.
- Introduce and use the TaskFactory class whenever a new task is created. This pattern helps me avoid hardcoding logic in the UI or controller and makes task creation more flexible.
- Optionally, if I had portfolio tasks (like user login or overdue filtering), I could plug them in as well without disrupting existing logic.

Explanation

The idea behind this task is to **fully separate concerns** in the app. The main.py file is now very lightweight and simply acts as an entry point that runs CommandLineUI.

The **CommandLineUI** class handles:

- Showing the menu
- Taking input from the user
- Deciding what action to take based on the input
- Sending user requests to the controller class (TaskManagerController)

The **TaskManagerController** class handles:

- Adding, deleting, editing, completing tasks
- Saving or loading data via the appropriate DAO (CSV or Pickle)
- Validating task indexes
- Communicating with the TaskFactory and TaskList

The **TaskFactory** decides what kind of task to create (normal or recurring) based on the given data. It acts like a helper tool to keep that logic out of both UI and controller.

This design pattern makes the app easier to extend in the future e.g., adding a GUI interface later won't require changing the task logic.

Benefits of This Restructure

- **Easier to maintain and modify**: UI changes don’t affect task logic, and vice versa.

- **Modular structure**: Each class has a clear responsibility, following **SRP** (Single Responsibility Principle).
- **No code duplication**: By centralizing task creation in TaskFactory, and task validation in TaskList, we follow the **DRY** (Don't Repeat Yourself) principle.
- **Easier to test**: Business logic and UI are separate, so we can write unit tests for the controller and tasks without dealing with print/input.
- **Extensible**: Adding features like login, filtering, exporting can be done without touching all parts of the app.

Source Code

Main.py

```python
"""
this is main file for To do List Manager application using OOP concepts
this file will use codes of task and tasklist classes
This is the entry point of the program
"""
from ui.command_line_ui import CommandLineUI

# function to add spacing
def spacing():
    print("\n")
    print("-"*40)
    print("\n")


def main() -> None:
    print("="*40)
    print("----Welcome to the To-Do List Manager----\n")
    ui = CommandLineUI()
    ui.run()
    print("\n")
if __name__ == "__main__":
    main()
```

command_line_ui.py

```python
from controllers.task_manager_controller import TaskManagerController
import datetime
from owner import Owner
import config
You, 1 second ago | 1 author (You)
class CommandLineUI:
    def __init__(self):
        name = input("Enter your name: ")
        email = input("Enter your email: ")
        owner = Owner(name, email)
        self.controller = TaskManagerController(owner)
    def run(self):
        while True:
            self._print_menu()
            print("\n")
            choice = input("Enter your choice: ")
            print("\n")
            if choice == "1":
                self._add_task()
            elif choice == "2":
                self.controller.view_tasks()
```

```python
            elif choice == "3":
                self._remove_task()
            elif choice == "4":
                self._mark_task_completed()
            elif choice == "5":
                self._change_task_title()
            elif choice == "6":
                self._change_task_description()
            elif choice == "7":
                self.controller.view_over_due_tasks()
            elif choice == "8":
                self._change_task_due_date
            elif choice == "9":
                self._load_tasks_from_csv()
            elif choice == "10":
                self._save_tasks_to_csv()
            elif choice == "11":
                self._load_tasks_from_pickle()
            elif choice == "12":
                self._save_tasks_to_pickle()
            elif choice == "13":
                owner = self.controller.get_owner()
                print(f"Owner Name: {owner.name}")
                print(f"Owner Email: {owner.email}")
            elif choice == "14":
                print("Quit")
                break
            else:
                print("Invalid choice. Please enter a number between 1 and 13.\n")
```

```
50
51      def _print_menu(self):
52          print("To-Do List Manager")
53          print("1. Add a task")
54          print("2. View tasks")
55          print("3. Remove a task")
56          print("4. Mark as completed")
57          print("5. Change title of task")
58          print("6. Change description of task")
59          print("7. Show over due tasks")
60          print("8. Change Due date of task")
61          print("9. Load Tasks from CSV")
62          print("10. Save Tasks to CSV")
63          print("11. Load Data from Pickle File")
64          print("12. Save Data to Pickle File")
65          print("13. Get Owner Details")
66          print("14. Quit")
```

```
67
68      def _get_task_details(self):
69          """This method is used to get the title, description and due date of the task from the user."""
70          title = input("Enter task title: ")
71          desc = input("Enter task description: ")
72          # input date in string
73          date_str = input("Enter due date (YYYY-MM-DD): ")
74          # converting string date into date object
75          due_date = datetime.datetime.strptime(date_str, "%Y-%m-%d").date()
76          return title, desc, due_date
77
```

```python
    def _add_task(self):
        """This method is used to add a task to the to-do list.
        It will ask the user whether the task is one time or recurring.
        """
        while True:
            print("1. One Time Task")
            print("2. Recurring Task")
            print("3. Back")

            sub_choice = input("Enter your choice: ")

            # one time task
            if sub_choice == "1":
                title, desc, due_date = self._get_task_details()
                # calling the controller method, as all the logic is in the controller
                self.controller.add_task(title, desc, due_date)
                print(f"'{title}' has been added to your to-do list.\n")
                break

            # recurring task
            elif sub_choice == "2":
                title, desc, due_date = self._get_task_details()
                # input interval days
                days = int(input("Enter the interval in days: "))
                # converting days into timedelta object to work on it later
                interval = datetime.timedelta(days=days)
                # calling the controller method, as all the logic is in the controller
                self.controller.add_task(title, desc, due_date, interval=interval)
                print(f"'{title}' has been added to your to-do list.\n")
                break

            elif sub_choice == "3":
                break
```

```
111
112     def _remove_task(self):
113         # Show current tasks before asking for deletion input
114         self.controller.view_tasks()
115
116         if not self.controller.has_tasks():
117             print("[!] No tasks to remove.\n")
118             return
119
120         # Keep asking user until a valid task number is entered
121         while True:
122             try:
123                 user_input = input("Enter the number of the task to remove: ")
124
125                 # If task list is empty, notify the user and exit
126                 # Ensure input is a digit, not letters or symbols
127                 if not user_input.isdigit():
128                     raise ValueError("Please enter a number (e.g. 1, 2, 3...)")
129
130                 index = int(user_input)
131                 # Ask controller to remove the task at this index
132                 self.controller.remove_task(index)  # let controller do all logic
133                 print("Task removed successfully.\n")
134                 # show the updated list of tasks
135                 self.controller.view_tasks()
136                 break
137
138             except ValueError as ve:
139                  # Catch non-numeric input and inform the user
140                 print(f"[!] {ve}\n")
141             except IndexError as ie:
142                 # Catch invalid index (e.g., number too high or low)
143                 print(f"[!] {ie}\n")
144
```

```python
    def _mark_task_completed(self):
        # Display current task list so user knows the numbers
        self.controller.view_tasks()

        # Check if there are any tasks at all
        if not self.controller.has_tasks():
            print("[!] No tasks available.\n")
            return

        # Repeat input prompt until valid task is selected
        while True:
            index = input("Enter the number of the task to mark as completed: ")

            if index.isdigit():
                index = int(index)
                try:
                    # Mark the selected task as completed
                    self.controller.mark_task_completed(index)
                    print("[✓] Task marked as completed.\n")
                    break
                except IndexError:
                    # If entered number is not a valid task index
                    print("Invalid task number. Please try again.\n")
            else:
                # Handle non-numeric input gracefully
                print("Invalid input. Please enter a number like 1, 2, 3...\n")

```

```python

    def _change_task_title(self):
        # Show all tasks so the user can pick the one to edit
        self.controller.view_tasks()

        # Check if task list is empty and return early
        if not self.controller.has_tasks():
            print("[!] No tasks available.\n")
            return

        # Keep prompting user for valid input
        while True:
            index = input("Enter the number of the task to change title: ")

            if index.isdigit():
                index = int(index)
                try:
                    # Ask for new title
                    new_title = input("Enter the new title: ")

                    # Pass both index and new title to controller
                    self.controller.change_task_title(index, new_title)
                    print("[✓] Title updated.\n")
                    break
                except IndexError:
                    # Catch if index is out of range
                    print("Invalid task number. Please try again.\n")
            else:
                # Warn the user if they enter a non-numeric value
                print("Invalid input. Please enter a number like 1, 2, 3...\n")

```

```
203
204     def _change_task_description(self):
205         # Display the current list of tasks so the user can see which task they want to update
206         self.controller.view_tasks()
207
208         # If there are no tasks available, inform the user and exit this function early
209         if not self.controller.has_tasks():
210             print("[!] No tasks available.\n")
211             return
212
213         # Enter a loop to get valid input from the user
214         while True:
215             # Ask the user to input the number of the task they want to change
216             index = input("Enter the number of the task to change description: ")
217
218             # Check if the input is a digit (i.e., a number like "1", "2", etc.)
219             if index.isdigit():
220                 index = int(index)  # Convert the valid numeric string input to an actual integer
221                 try:
222                     # Prompt user to input the new description for the selected task
223                     new_description = input("Enter the new description: ")
224
225                     # Call the controller's method to update the task description
226                     self.controller.change_task_description(index, new_description)
227
228                     # Let the user know the update was successful
229                     print("[✓] Description updated.\n")
230
231                     # Exit the loop since the operation was successful
232                     break
```

```
233
234                 except IndexError:
235                     # This will happen if the task number entered does not exist
236                     print("Invalid task number. Please try again.\n")
237             else:
238                 # If the input is not a valid number, show an error message and repeat
239                 print("Invalid input. Please enter a number like 1, 2, 3...\n")
240
```

```
241     def _change_task_due_date(self):
242         # Show all current tasks so the user knows which task they might want to update
243         self.controller.view_tasks()
244 
245         # If there are no tasks, notify the user and stop the operation early
246         if not self.controller.has_tasks():
247             print("[!] No tasks available.\n")
248             return
249 
250         # Continue prompting the user until a valid task number and date are provided
251         while True:
252             # Ask the user to select the task by number
253             index = input("Enter the number of the task to change due date: ")
254 
255             # Check if the input is a valid number
256             if index.isdigit():
257                 index = int(index)  # Convert the input string to an integer
258                 try:
259                     # Ask the user for the new due date in the correct format
260                     new_date = input("Enter the new due date (YYYY-MM-DD): ")
261 
262                     # Use the controller to update the due date of the selected task
263                     self.controller.change_due_date(index, new_date)
264                     # Confirm that the update was successful
265                     print("[✓] Due date updated.\n")
266                     # Exit the loop after successful update
267                     break
268                 except IndexError:
269                     # If the user entered a task number that doesn't exist, show an error message
270                     print("Invalid task number. Please try again.\n")
271             else:
272                 # If the user input is not a number, show an error message and loop again
273                 print("Invalid input. Please enter a number like 1, 2, 3...\n")
274 
```

```
275     def _load_tasks_from_csv(self):
276         # Check if a file path has already been set in the config module
277         if (not config.file_path):
278             # If not, ask the user to enter the path of the CSV file to load tasks from
279             config.file_path = input("Enter file path to load tasks (e.g. tasks.csv): ")
280 
281         try:
282             # Call the controller to load tasks from the given file path
283             self.controller.load_tasks_from_csv(config.file_path)
284 
285             # If loading is successful, let the user know
286             print("[✓] Tasks loaded from file.\n")
287         except Exception as e:
288             # If anything goes wrong (e.g., file not found or corrupt), catch and show the error
289             print(f"[!] Failed to load tasks: {e}\n")
290 
291     def _save_tasks_to_csv(self):
292         # Check if the CSV file path is not already set
293         if (not config.file_path):
294             # Prompt the user to enter the file path where tasks should be saved
295             config.file_path = input("Enter file path to load tasks (e.g. tasks.csv): ")
296 
297         try:
298             # Ask the controller to save all tasks to the given CSV file
299             self.controller.save_tasks_to_csv(config.file_path)
300 
301             # Notify the user that the saving was successful
302             print("[✓] Tasks saved to file.\n")
303         except Exception as e:
304             # If any unexpected error occurs during the saving process, show an error message
305             print(f"[!] Failed to save tasks: {e}\n")
```

```python
    def _load_tasks_from_pickle(self):
        # Check if the Pickle file path is not already set
        if (not config.file_path_pkl):
            # Ask the user to input the path to the Pickle file from which to load tasks
            config.file_path_pkl = input("Enter file path to load tasks (e.g. tasks.pkl): ")

        try:
            # Ask the controller to load tasks from the specified Pickle file
            self.controller.load_tasks_from_pickle(config.file_path_pkl)

            # Notify the user of successful loading
            print("[✓] Tasks loaded from pickle.\n")
        except Exception as e:
            # Catch and print any errors that occur during loading (e.g., file not found or unpickling issues
            print(f"[!] Failed to load tasks: {e}\n")


    def _save_tasks_to_pickle(self):
        # Check if the Pickle file path has not already been defined
        if (not config.file_path_pkl):
            # Prompt the user for the Pickle file path to save the tasks
            config.file_path_pkl = input("Enter file path to load tasks (e.g. tasks.pkl): ")

        try:
            # ❗ BUG ALERT: This line mistakenly **loads** tasks instead of saving them.
            # The function name suggests saving, but it's calling `load_tasks_from_pickle`.
            # It should likely be:
            # self.controller.save_tasks_to_pickle(config.file_path_pkl)

            self.controller.save_tasks_to_pickle(config.file_path_pkl)  # <-- should be save
            print("[✓] Tasks saved to pickle.\n")
        except Exception as e:
            # Handle and display any exceptions during the save process
            print(f"[!] Failed to save tasks: {e}\n")
```

task_list.py

```python
class TaskList:
    def __init__(self, owner:Owner) -> None:
        # Constructor that initializes the task list with the given owner
        self.owner = owner
        self.tasks = []  # Empty list to hold tasks

    def add_task(self, task:Task) -> None:
        # Adds a new task to the task list
        self.tasks.append(task)

    def remove_task(self,index) -> None:
        # Removes a task by its 1-based index (as seen by the user)

        if index >= 1 and index <= len(self.tasks):
            # Get the task object using get_task method
            task = self.get_task(index)

            print(f"Removed: {task.title}")  # Display task title that is being removed

            # Delete the task — NOTE: this only deletes the reference to the task, not from the list
            del task
        else:
            # If index is invalid, notify user
            print("Invalid index. Please try again.")

```

```python
    """Property attribute is used to use the method as attribute
    In this case this method of getting in completed tasks will be used as attribute and not method
    @property
    def uncompleted_tasks(self) -> list[Task]:
        # Returns a list of tasks that are not marked as completed
        return [task for task in self.tasks if not task.completed]

    def view_tasks(self) -> None:
        # Displays the tasks that are still pending (i.e., not completed)
        if not self.uncompleted_tasks:
            print("No tasks to show.")  # No uncompleted tasks
        else:
            print("The following tasks are still to be done:")
            for task in self.uncompleted_tasks:
                # Find original index of the task in the full list for proper numbering
                ix = self.tasks.index(task)
                print(f"{ix+1}: {task}")  # Show task number and string representation of task

```

```python
    def view_over_due_tasks(self)->None:
        # Displays tasks that have passed their due date
        if not self.tasks:
            print("No tasks in the list.")
            return

        over_due_tasks = []
        today = datetime.date.today()

        # Check each task to see if it's overdue
        for index, task in enumerate(self.tasks, start=1):
            if task.due_date < today:
                over_due_tasks.append((index, task))  # Add index and task to the overdue list

        if over_due_tasks:
            print("Over Due Tasks:")
            for i, task in over_due_tasks:
                # If description exists, show it; otherwise say "No description"
                desc = task.description if task.description else "No description"
                print(f"{i}. {task.title} | Due Date: {task.due_date} | Description: {desc}")
        else:
            print("No over due tasks available")

        # Add visual spacing to the output
        print("\n")
        print("-"*40)
        print("\n")
```

```python

    def get_task_by_title(self, title: str) -> Task:
        # Returns a task by matching the given title
        for task in self.tasks:
            if task.title == title:
                return task

    def get_task(self, index) -> Task:
        """Returns the task at the given index (1-based index for user-friendliness)."""
        if 1 <= index <= len(self.tasks):
            return self.tasks[index - 1]  # Convert 1-based to 0-based index
        else:
            return None  # Return None if index is invalid

```

task_factory.py

```python
class TaskFactory:
    @staticmethod
    def create_task(title: str, description: str, date: datetime.datetime, **kwargs: Any) -> Task:
        # Factory method that creates either a Task or RecurringTask object depending on the presence of "interval"
        # It checks if an "interval" is passed in the keyword arguments (**kwargs).
        # If yes, it creates a RecurringTask object, otherwise just a normal Task.
        if "interval" in kwargs:
            return RecurringTask(title, description, date, kwargs["interval"])
        else:
            return Task(title, description, date)

```

task.py

```python
class Task:
    def __init__(self, title: str, description: str, due_date) -> None:
        # Initializes a Task object with a title, description, and due date.
        self.title = title  # Task title
        self.description = description  # Task description
        self.completed = False  # By default, task is not completed
        self.date_created = datetime.datetime.now()  # Set creation date to current timestamp
        self.due_date = due_date  # Set the due date as provided by the user

    def __str__(self) -> str:
        # This method controls how the Task object is displayed when printed
        status = "Completed" if self.completed else "Pending"  # Show readable completion status
        desc = self.description if self.description else "No description"  # Fallback if no description

        # Nicely formatted string that represents the full task info
        return f"Task: {self.title} | Status: {status} | Date Created: {self.date_created.strftime('%Y-%m-%d')} | Due
        Date: {self.due_date} | Description: {desc}"
    def mark_completed(self) -> None:
        # Mark the task as completed by setting the flag to True
        self.completed = True

    def change_title(self, new_title) -> None:
        # Allow changing the title of the task
        self.title = new_title

```

```python
    def change_description(self, new_description) -> None:
        # Allow changing the task's description
        self.description = new_description

    def change_due_date(self, new_date) -> None:
        # Allow modifying the due date of the task
        self.due_date = new_date

    def __eq__(self, other):
        # This equality method helps compare two Task objects.
        # It returns True only if title, due_date, and description are all exactly the same.
        return (
            isinstance(other, Task)
            and self.title == other.title
            and self.due_date == other.due_date
            and self.description == other.description
        )

```

task_manager_controller.py

```python
class TaskManagerController:
    def __init__(self, owner):
        self.task_list = TaskList(owner)

    def add_task(self, title, desc, due_date, interval=None):
        if interval:
            task = TaskFactory.create_task(title, desc, due_date, interval=interval)
        else:
            task = TaskFactory.create_task(title, desc, due_date)

        self.task_list.add_task(task)

    def has_tasks(self) -> bool:
        return bool(self.task_list.tasks)

    def remove_task(self, index: int) -> None:
        if index < 1 or index > len(self.task_list.tasks):
            raise IndexError("That task number doesn't exist.")
        task = self.task_list.get_task(index)
        if not task:
            raise IndexError("That task number doesn't exist.")

        removed_title = task.title
        # removed_title = self.task_list.tasks[index - 1].title

        self.task_list.remove_task(index)

        # Remove from CSV
        dao = TaskCsvDAO(config.file_path)
        dao.remove_task_by_title(removed_title)


```

```
42      def get_all_tasks(self):
43          return self.task_list.tasks
44
45      def mark_task_completed(self, index: int):
46          if index < 1 or index > len(self.task_list.tasks):
47              raise IndexError("Invalid index")
48          # getting task object by using get_task() method
49          task = self.task_list.get_task(index)
50          if not task:
51              raise IndexError("That task number doesn't exist.")
52
53          old_title = task.title
54          task.mark_completed()
55          dao = TaskCsvDAO(config.file_path)
56          dao.update_task(task, old_title)
57
58      def change_task_title(self, index: int, new_title: str):
59          if index < 1 or index > len(self.task_list.tasks):
60              raise IndexError("Invalid index")
61          # getting task object by using get_task() method
62          task = self.task_list.get_task(index)
63          if not task:
64              raise IndexError("That task number doesn't exist.")
65
66          old_title = task.title
67          task.change_title(new_title)
68          dao = TaskCsvDAO(config.file_path)
69          dao.update_task(task, old_title)
70
```

```
71      def change_task_description(self, index: int, new_desc: str):
72          if index < 1 or index > len(self.task_list.tasks):
73              raise IndexError("Invalid index")
74          # getting task object by using get_task() method
75          task = self.task_list.get_task(index)
76          if not task:
77              raise IndexError("That task number doesn't exist.")
78
79
80          old_title = task.title
81          task.change_description(new_desc)
82          dao = TaskCsvDAO(config.file_path)
83          dao.update_task(task, old_title)
84
85      def change_due_date(self, index, new_date):
86          if index < 1 or index > len(self.task_list.tasks):
87              raise IndexError("Invalid index")
88          # getting task object by using get_task() method
89          task = self.task_list.get_task(index)
90          if not task:
91              raise IndexError("That task number doesn't exist.")
92
93          old_title = task.title
94          task.change_due_date(new_date)
95          dao = TaskCsvDAO(config.file_path)
96          dao.update_task(task, old_title)
97
98      def view_over_due_tasks(self):
99          self.task_list.view_over_due_tasks()
100
101     def view_tasks(self):
102         self.task_list.view_tasks()
```

```
103
104     def load_tasks(self, tasks):
105         for task in tasks:
106             self.task_list.add_task(task)
107     def load_tasks_from_csv(self, path: str):
108         dao = TaskCsvDAO(path)
109         tasks = dao.get_all_tasks()
110         self.load_tasks(tasks)  # loads them into task_list
111     def save_tasks_to_csv(self, path: str):
112         dao = TaskCsvDAO(path)
113 
114         dao.save_all_tasks(self.task_list.tasks)
115     def get_task(self, index):
116         task = self.task_list.get_task(index)
117         return task
118     def load_tasks_from_pickle(self, path: str):
119         dao = TaskPickleDAO(path)
120         tasks = dao.get_all_tasks()
121         self.load_tasks(tasks)
122 
123     def save_tasks_to_pickle(self, path: str):
124         dao = TaskPickleDAO(path)
125         dao.save_all_tasks(self.task_list.tasks)
126 
127     def get_owner(self) -> Owner:
128         return self.task_list.owner
129 
```

Week 8

Section 1. Data Structures

In programming, **data structures** are different ways of organizing and storing data so it can be used efficiently. Python offers several built-in data structures like **lists**, **tuples**, **sets**, and **dictionaries**, each with unique behaviors and use cases.

In this exercise, we'll focus on:

1. Tuples
2. Sets
3. Dictionaries

Exercise 1: Tuples

Understanding the Task:

A **tuple** is a collection of items that are **ordered and immutable**, meaning once you create it, you cannot change its values. It's great for storing data that shouldn't be modified, like coordinates or RGB color values.

Source Code

```
# Creating a tuple
my_tuple = ("apple", "banana", "cherry")

# Accessing items
print(my_tuple[1])

# Checking length
print(len(my_tuple))

# Trying to modify a tuple (will raise an error if uncommented)
# my_tuple[0] = "orange"
```

Explanation:

- my_tuple is created with three fruits.
- We access the second item using indexing (my_tuple[1]).
- We check the total number of items using len().
- Tuples are **immutable**, so trying to modify an item will raise an error.

Output

```
banana
3
```

Exercise 2: Sets

Understanding the Task:

A **set** is an **unordered collection** of **unique elements**. It's useful when you want to remove duplicates or perform operations like union or intersection.

Source Code

```
# Creating a set
fruits = {"apple", "banana", "cherry", "apple"}

# Adding a new item
fruits.add("mango")

# Removing an item
fruits.discard("banana")

# Printing the set
print(fruits)
```

Explanation:

- The set removes the **duplicate "apple"** automatically.
- We add "mango" and remove "banana" using add() and discard() respectively.
- Sets do not maintain any order, so the printed result can appear in any sequence.

Output

```
{'cherry', 'apple', 'mango'}
```

Exercise 3: Dictionaries

Understanding the Task:

A **dictionary** is a collection of **key-value pairs**. It's used when you want to map a value to a specific label or key, like storing student names with their grades.

Source Code

```
# Creating a dictionary
student = {"name": "Ali", "age": 21, "grade": "A"}

# Accessing values
print(student["name"])

# Updating a value
student["grade"] = "A+"

# Adding a new key-value pair
student["subject"] = "Math"

# Printing the dictionary
print(student)
```

Explanation:

- A dictionary student is created with keys: "name", "age", and "grade".
- We access and update values using square brackets.
- New key-value pairs can be added anytime.
- Dictionaries are **unordered** in Python versions before 3.7, but now they preserve insertion order.

Output

```
Ali
{'name': 'Ali', 'age': 21, 'grade': 'A+', 'subject': 'Math'}
```

Task: Swap a and b using tuple

Source Code

```
a = 5
b = 10

print("Before swapping:")
print("a =", a)
print("b =", b)

# Swap using tuple (Pythonic way)
a, b = b, a  # This one-liner swaps the values of a and b without needing a temporary variable

print("After swapping:")
print("a =", a)
print("b =", b)
```

Explanation

- We first define two variables a and b with values 5 and 10.
- Python allows us to **swap** values in a very clean way using tuple unpacking: a, b = b, a.
- No need for a temporary variable (like in other languages).
- The values are now swapped.

Output

```
Before swapping:
a = 5
b = 10
After swapping:
a = 10
b = 5
```

Task: Comparing two sets to find common values

Source Code

```
set1 = {"Tom", "Jerry", "Hewey", "Dewey", "Louie"}
set2 = {"Tom", "Garfield", "Snoopy", "Hewey", "Dewey"}

# Use set intersection to get names present in both sets
common_names = set1 & set2  # The & operator gives us only those elements that exist in both sets

print("\nNames in both sets:")
print(common_names)
```

Explanation:

- We define two sets with cartoon character names.
- The & operator performs a **set intersection**, returning only elements found in both sets.
- This helps us quickly compare and find common entries.
- The result is printed as a Python set.

Output

```
Names in both sets:
{'Tom', 'Hewey', 'Dewey'}
```

Task: Histogram

Source Code

```
def histogram(input_list):
    freq_dict = {}  # empty dictionary to store frequencies
    for item in input_list:
        if item in freq_dict:
            freq_dict[item] += 1  # increase count if item already exists
        else:
            freq_dict[item] = 1   # initialize count if item not seen before
    return freq_dict

# Test the function
my_list = [1, 2, 3, 1, 2, 3, 4]
print("\nHistogram output:")
print(histogram(my_list))

# Optional: check with assert to verify correctness
assert histogram(my_list) == {1: 2, 2: 2, 3: 2, 4: 1}
```

Explanation:

- A function histogram is defined to count how many times each number appears in a list.
- It uses a **dictionary** to store numbers as keys and their count as values.
- For each item in the list:
    - If the number is already in the dictionary, we increase its count.
    - If not, we add it with a count of 1.
- We then test the function with a sample list: [1, 2, 3, 1, 2, 3, 4].
- The assert statement is used for **automated verification**, ensuring the result matches the expected dictionary.

Output

```
Histogram output:
{1: 2, 2: 2, 3: 2, 4: 1}
```

Section 2. Abstract Classes

It is the concept of hiding the complex implementation of an object and only showing the necessary features of the object. Abstract classes enable this as all we do is define the methods that must be implemented by the child classes. Once those child classes have been created, we can use them without needing to know the details of their implementation but can be sure that those methods (and attributes) will be available.

To use abstract classes in python, it is required to import the **ABC** and **abstractmethod** modules from the **abc package**. Then you can create an abstract class by inheriting from the ABC class and using the **@abstractmethod** decorator to define the abstract methods.

```
from abc import ABC, abstractmethod
```

Task: Dice class and abstract method in it

Source Code

```
class Dice(ABC):  # Inherits from Abstract Base Class
    def __init__(self) -> None:
        self.face = None  # All dice will have a face attribute

    @abstractmethod
    def roll(self) -> int:
        pass  # Child classes must implement this method
```

Explanation

In this task, we are introduced to **abstract classes** in Python using a simple and relatable example, a dice. The main goal here is to design a **base class** that defines the **essential**

structure and **common interface** for all dice, without actually implementing how the dice rolls. That behavior will be defined by the **child classes**.

The Dice class is created as an **abstract class** using Python's ABC module. This class sets a foundation that all dice must follow. It includes:

- A face attribute, which will hold the current value shown on the dice after a roll.
- An abstract method roll(), which is meant to be implemented by any class that inherits from Dice. This method will define how the dice is rolled (e.g., random number generation for 6-sided dice).

The key takeaway is that the Dice class **cannot be used directly to create objects** because it lacks a working implementation of roll(). It's just a **blueprint** meant to be extended.

Task: Six Sided Dice

Understanding the Task

Now that we have an abstract base class Dice, the next step is to **implement a specific type of dice**, in this case, a standard **six-sided dice** (like the one used in most board games). The idea is to **inherit** from the Dice class and provide the missing roll() method.

Source Code

```
You, 2 days ago | 1 author (You)
class SixSidedDice(Dice):
    def roll(self) -> int:
        self.face = randint(1, 6)
        return self.face


# Roll the dice 1000 times and collect results
dice = SixSidedDice()
results = [dice.roll() for _ in range(1000)]
```

Explanation

- This class SixSidedDice **inherits** from the abstract Dice class.
- It **implements** the roll() method that was declared as abstract in Dice.
- Inside roll(), it uses randint(1, 6) to simulate rolling a real dice, generating a random number between 1 and 6.
- The result is stored in the face attribute (inherited from Dice) and returned.

The class is now **concrete** and can be instantiated since all abstract methods are implemented.

Rolling the Dice 1000 Times:

```
dice = SixSidedDice()
results = [dice.roll() for _ in range(1000)]
```

- An instance of SixSidedDice is created.
- The roll() method is called **1000 times** using a list comprehension.
- Each result is stored in the results list, so we can analyze frequencies, simulate outcomes, or plot distributions.

Task: Ten-Sided Dice

Understanding the Task

This task focuses on building a **concrete subclass** of an abstract Dice class. Specifically, we are implementing a TenSidedDice that simulates the behavior of a dice with 10 faces. The core idea is to show how inheritance and abstraction work together, and then simulate the dice to see how random values are distributed. Finally, we visualize the outcome using a histogram to check how balanced the results are.

Source Code and Explanation

```
class TenSidedDice(Dice):
    def roll(self) -> int:
        self.face = randint(1, 10)  # Generates a random number between 1 and 10
        return self.face
```

- This class **inherits from** the abstract Dice class.
- It **implements** the required roll() method.
- Each roll gives a value from **1 to 10**, and the result is stored in the face attribute.

```
# Roll the ten-sided dice 1000 times
ten_dice = TenSidedDice()
results_10 = [ten_dice.roll() for _ in range(1000)]
```

- An instance of TenSidedDice is created.
- A list comprehension is used to **simulate 1000 rolls** and collect the outcomes.

```
# Show histogram
print("\nTen-Sided Dice Histogram (1000 rolls):")
print(histogram(results_10))
```

- he histogram() function counts how many times each number (1–10) appeared.
- This is used to **visually inspect randomness**, ideally, each number should appear around 100 times.

Output

```
Ten-Sided Dice Histogram (1000 rolls):
{1: 94, 2: 102, 3: 104, 4: 95, 5: 98, 6: 100, 7: 97, 8: 101, 9: 102, 10: 107}
```

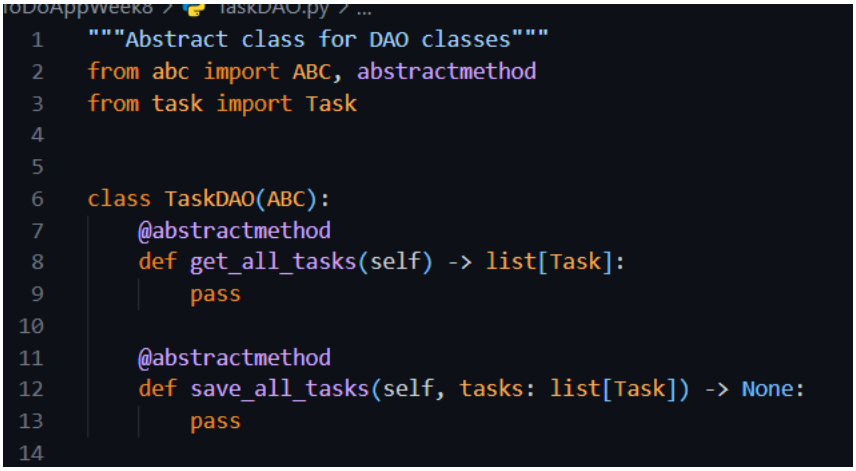
Task: Implement Abstraction in ToDoApp

Understanding the Task

This task is about **applying abstraction** in your ToDo application by introducing **abstract base classes** (ABCs) using Python's abc module. The main goal is to enforce **a common structure** for classes that are similar in purpose but differ in implementation, such as the DAO classes and Task-related classes. This makes the application more **scalable, maintainable, and robust**.

Source Code

TaskDAO.py – Abstract class for DAO classes



```python
"""Abstract class for DAO classes"""
from abc import ABC, abstractmethod
from task import Task


class TaskDAO(ABC):
    @abstractmethod
    def get_all_tasks(self) -> list[Task]:
        pass

    @abstractmethod
    def save_all_tasks(self, tasks: list[Task]) -> None:
        pass

```

Using Abstract class

TaskCsvDAO.py

```python
from TaskDAO import TaskDAO
class TaskCsvDAO(TaskDAO):
    def __init__(self, storage_path: str) -> None:
        # gets the file path and joins it
        self.storage_path = os.path.join(os.path.dirname(__file__), storage_path)

        # initialize fieldnames
        self.fieldnames = ["title", "type", "date_due", "completed", "interval", "completed_dates", "date_created"]

```

TaskPickleDAO.py

```python
from TaskDAO import TaskDAO
class TaskPickleDAO(TaskDAO):
    def __init__(self, storage_path: str) -> None:
        # Combine the current directory with the file name to get the full path
        self.storage_path = os.path.join(os.path.dirname(__file__), storage_path)

```

AbstractTask.py – Abstract class for Task classes

```python
"""Abstract class for Task classes."""
from abc import ABC, abstractmethod
from task import Task
import datetime

class AbstractTask(ABC):
    def __init__(self, title: str, description: str, due_date: datetime.date):
        self.title = title
        self.description = description
        self.completed = False
        self.date_created = datetime.datetime.now()
        self.due_date = due_date

    @abstractmethod
    def mark_completed(self) -> None:
        pass

    @abstractmethod
    def __str__(self) -> str:
        pass
```

Using Abstract class

task.py

```python
from AbstractTask import AbstractTask
class Task(AbstractTask):
    def __init__(self, title: str, description: str, due_date) -> None:
        # Initializes a Task object with a title, description, and due date.
        self.title = title  # Task title
        self.description = description  # Task description
        self.completed = False  # By default, task is not completed
        self.date_created = datetime.datetime.now()  # Set creation date to current timestamp
        self.due_date = due_date  # Set the due date as provided by the user

```

recurring_task.py

```python
from AbstractTask import AbstractTask

class RecurringTask(AbstractTask):
    """
    This class is for recurring tasks
    Args: inherits from parent class 'Task'
    """
    def __init__(self, title:str, description:str, due_date, interval:datetime.timedelta) -> None:
        # title, description, due_date are attributes inherited from parent class
        # interval is new attribute which is for repetition of tasks
        super().__init__(title, description, due_date)
        self.interval = interval
        # list of completed dates of recurring tasks for history
        self.completed_dates : list[datetime.datetime] = []
```

Portfolio Exercise 5: Priority Task

Understanding the Task

This exercise expands the ToDo app's functionality by introducing a new task type called **PriorityTask**. Just like we have Task and RecurringTask, the PriorityTask adds an extra layer of importance by associating tasks with a **priority level** (1 to 3).

- Priority will be stored as an **integer** (1, 2, 3) representing **low**, **medium**, and **high**.
- A **mapping dictionary** will convert those numeric values into user-friendly strings ("low", "medium", "high").
- The class should include:
    - type hints,
    - validation (to restrict values outside 1–3),
    - and a readable string representation via __str__.

Source Code

```python
class PriorityTask(Task):
    """
    Represents a task with a priority level.
    Inherits from Task and adds a 'priority' attribute with validation.
    """

    # This dictionary maps integer values to readable priority labels
    PRIORITY_MAP = {
        1: "Low",
        2: "Medium",
        3: "High"
    }
    def __init__(self, title: str, description: str, due_date: datetime.date, priority: int) -> None:
        super().__init__(title, description, due_date)

        # Ensure priority is within the accepted range (1, 2, 3)
        if priority not in self.PRIORITY_MAP:
            raise ValueError("Priority must be 1 (Low), 2 (Medium), or 3 (High).")

        self.priority = priority
    def __str__(self) -> str:
        # Add priority to the display string using its human-readable label
        priority_label = self.PRIORITY_MAP.get(self.priority, "Unknown")
        return super().__str__() + f" | Priority: {priority_label}"

    def change_priority(self, new_priority: int) -> None:
        # Allow changing the priority after creation with validation
        if new_priority not in self.PRIORITY_MAP:
            raise ValueError("Priority must be 1 (Low), 2 (Medium), or 3 (High).")
        self.priority = new_priority

```

Explanation

- PriorityTask is a specialized type of task that builds on the regular task features.
- It includes a priority level to show how important the task is.
- The priority is stored as a number: 1 (Low), 2 (Medium), or 3 (High).
- When creating a task, the program checks that the priority is valid (only 1, 2, or 3 are allowed).
- If an invalid number is given, the task won't be created, and an error is shown.
- The task can display priority in words using a built-in mapping from numbers to labels.
- When printed or displayed, the task shows its usual details plus the priority label.
- The priority can be changed later, and the new value is also validated to be one of the allowed numbers.

Portfolio Exercise: Integrate Priority Task in ToDoApp

Understanding the Task

In this exercise, I was asked to fully integrate the PriorityTask class into the existing To-Do application. The goal was to make sure that all parts of the application are aware of and can handle this new type of task that includes a priority level (Low, Medium, High).

Specifically, I was required to update three main components:

- **TaskFactory**: So it can create PriorityTask objects when a priority is provided.
- **CommandLineUI**: So the user can interactively create a PriorityTask through the terminal.
- **DAO (Data Access Objects)**: So PriorityTask can be saved and loaded correctly from files (especially CSV), including the priority value.

What I understood from the question is that this is not just about creating a new class, it's about making sure it works throughout the whole system just like other task types (e.g., standard and recurring tasks). This includes making sure it can be added, saved, loaded, and displayed properly within the existing app structure.

Source Code

Task_factory

```
5   from priority_task import PriorityTask
6   class TaskFactory:
7       @staticmethod
8       def create_task(title: str, description: str, date: datetime.datetime, **kwargs: Any) -> Task:
9           # Factory method that creates either a Task or RecurringTask object depending on the presence of "interval"
10          # It checks if an "interval" is passed in the keyword arguments (**kwargs).
11          # it checks if a "priority" is passed in the keyword arguments (**kwargs).
12          # If yes, it creates a RecurringTask object, otherwise just a normal Task.
13          if "interval" in kwargs:
14              return RecurringTask(title, description, date, kwargs["interval"])
15          elif "priority" in kwargs:
16              return PriorityTask(title, description, date, kwargs["priority"])  # ✅ New
17          else:
18              return Task(title, description, date)
```

Comman_line_ui.py

```
    def _add_task(self):
        """This method is used to add a task to the to-do list.
        It will ask the user whether the task is one time or recurring.
        """
        while True:
            print("1. One Time Task")
            print("2. Recurring Task")
            print("3. Priority Task")
            print("4. Back")
```

```
            elif sub_choice == "3":
                while True:
                    try:
                        priority = int(input("Enter priority (1 = Low, 2 = Medium, 3 = High): "))
                        if priority in [1, 2, 3]:
                            break
                        else:
                            print("Priority must be 1, 2, or 3.")
                    except ValueError:
                        print("Please enter a number (1, 2, or 3).")
                # passing priority argument will let the priority task constructor run
                self.controller.add_task(title, desc, due_date, priority=priority)
                print(f"'{title}' has been added to your to-do list.\n")
                break
```

TaskCsvDAO.py

Init function

```
    def __init__(self, storage_path: str) -> None:
        # gets the file path and joins it
        self.storage_path = os.path.join(os.path.dirname(__file__), storage_path)

        # initialize fieldnames
        self.fieldnames = ["title", "type", "date_due", "completed", "interval", "completed_dates", "date_created",
        "priority"]

```

Get_all_tasks()

```
                elif task_type == "PriorityTask":
                    priority = int(row.get("priority", 1))  # default to 1 if missing
                    task = PriorityTask(title, '', date_due, priority)
```

save_all_tasks()

```
                elif isinstance(task, PriorityTask):
                    row["type"] = "PriorityTask"
                    row["priority"] = str(task.priority)
                    row["interval"] = ""
                    row["completed_dates"] = ""
```

Output

O18

	A	B	C	D	E	F	G
1	title	type	date_due	completed	interval	completed_dates	date_created
2	new	Task	2/2/2022	FALSE			7/3/2024
3	Do laundry	Task	9/3/2024	FALSE			7/3/2024
4	Clean room	Task	6/3/2024	FALSE			7/3/2024
5	Do homework	Task	10/3/2024	FALSE			7/3/2024
6	Walk dog	Task	12/3/2024	FALSE			7/3/2024
7	Do dishes	Task	13/03/2024	FALSE			7/3/2024
8	Go to the gym	RecurringTask	7/3/2024	FALSE	7 days, 0:00:00	2024-02-29,2024-02-22,2024-02-14	8/2/2024
9	t1	Task	2/2/2022	FALSE			7/12/2025
10	fgh	Task	2/2/2022	FALSE			7/12/2025
11	1asf	Task	2/2/2022	FALSE			12/7/2025
12	111	Task	2/2/2222	FALSE			12/7/2025
13	rtyuiop	RecurringTask	1/1/2022	TRUE	8 days		12/7/2025
14	one new	Task	4/4/2022	FALSE			12/7/2025
15	fill the bottle	Task	1/1/2022	FALSE			14/07/2025
16							

Choosing 1 and then 3 to add a priority task to the list

```
To-Do List Manager
1. Add a task
2. View tasks
3. Remove a task
4. Mark as completed
5. Change title of task
6. Change description of task
7. Show over due tasks
8. Change Due date of task
9. Load Tasks from CSV
10. Save Tasks to CSV
11. Load Data from Pickle File
12. Save Data to Pickle File
13. Get Owner Details
14. Quit


Enter your choice: 1


1. One Time Task
2. Recurring Task
3. Priority Task
4. Back
Enter your choice: 3
Enter task title: priority task
Enter task description: priority task desc
Enter due date (YYYY-MM-DD): 2022-2-2
Enter priority (1 = Low, 2 = Medium, 3 = High): 3
'priority task' has been added to your to-do list.
```

Choosing 10 to save data to file

```
Enter your choice: 10


Enter file path to load tasks (e.g. tasks.csv): tasks.csv
[✓] Tasks saved to file.
```

CSV

A1 title

	A	B	C	D	E	F	G
1	title	type	date_due	completed	interval	completed_dates	date_created
2	new	Task	2/2/2022	FALSE			7/3/2024
3	Do laundry	Task	9/3/2024	FALSE			7/3/2024
4	Clean room	Task	6/3/2024	FALSE			7/3/2024
5	Do homework	Task	10/3/2024	FALSE			7/3/2024
6	Walk dog	Task	12/3/2024	FALSE			7/3/2024
7	Do dishes	Task	13/03/2024	FALSE			7/3/2024
8	Go to the gym	RecurringTask	7/3/2024	FALSE	7 days, 0:00:00	2024-02-29,2024-02-22,2024-02-14	8/2/2024
9	t1	Task	2/2/2022	FALSE			7/12/2025
10	fgh	Task	2/2/2022	FALSE			7/12/2025
11	1asf	Task	2/2/2022	FALSE			12/7/2025
12	111	Task	2/2/2222	FALSE			12/7/2025
13	rtyuiop	RecurringTask	1/1/2022	TRUE	8 days		12/7/2025
14	one new	Task	4/4/2022	FALSE			12/7/2025
15	fill the bottle	Task	1/1/2022	FALSE			14/07/2025
16	priority task	PriorityTask	2/2/2022	FALSE			14/07/2025

We can see from this picture that the priority task is added to the file

---