Week 2

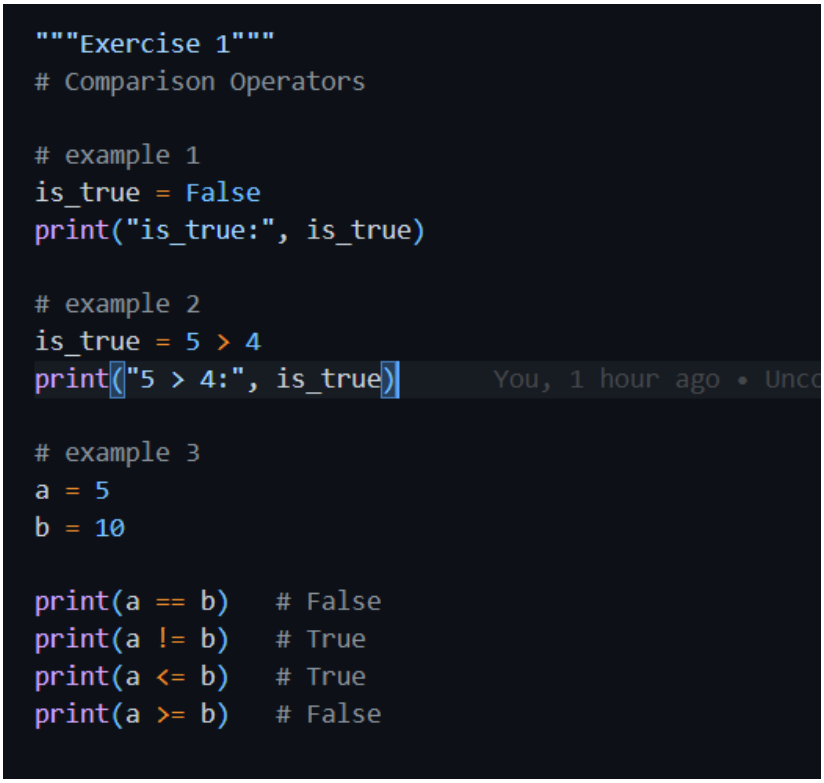
Section 1. Comparisons and Conditionals

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
print("is_true:", is_true)

# example 2
is_true = 5 > 4
print("5 > 4:", is_true)      You, 1 hour ago • Unco

# example 3
a = 5
b = 10

print(a == b)   # False
print(a != b)   # True
print(a <= b)   # True
print(a >= b)   # False
```

Explanation

- In **Example 1**, I directly assigned a Boolean value 'False' to a variable and printed it.
- In **Example 2**, I used a comparison 5 > 4 which is True, and printed the result.
- In **Example 3**, I compared two variables a = 5 and b = 10 using all standard comparison operators to see which conditions are true.

Time and Space Complexity

- **Time Complexity**: O(1) – All operations are constant-time comparisons.
- **Space Complexity**: O(1) – Only a few variables are stored in memory.

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

print("is_in_age_range:", is_in_age_range)

# example 2
x = 5
y = 10

print(x > 0 and y > 0)    # True
print(x > 0 or y < 0)     # True
print(not(x > 0))         # False
```

Explanation

- In the first example, I checked if the age falls between 20 and 30. Since 25 is in that range, it returns True.
- In the second part, I used two variables x and y and tested them using different logical operators:
    - x > 0 and y > 0 checks if both values are positive — which they are.
    - x > 0 or y < 0 checks if **at least one** condition is true — which is also true.
    - not (x > 0) flips the result of x > 0. Since x is greater than 0, the original condition is True and not makes it 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}")

# example 2
age = 13
age_group = "child"
if age > 18:
    age_group = "adult"
    print(f"The age group is {age_group}")
```

Explanation

- In the first example, the age was 19, which is greater than 18, so it updated age group from "child" to "adult" and printed it.

- In the second example, the age was 13, so the condition age > 18 was false and nothing was printed — the program skipped the if block.

Time Complexity and Space Complexity

- **Time Complexity**: O(1) – A single condition is checked.
- **Space Complexity**: O(1) – Only a few variables are used.

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

```
"""Exercise 4"""
# If-else Conditionals

# example 1
wind_speed = 30
if wind_speed < 10:
    print("It is a calm day")
else:
    print("It is a windy day")

# example 2
wind_speed = 5
if wind_speed < 10:
    print("It is a calm day")
else:
    print("It is a windy day")
```

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".

Time Complexity and Space Complexity

- **Time Complexity**: O(1) – It just checks one condition.
- **Space Complexity**: O(1) – Minimum memory usage.

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

```
"""Exercise 5"""
# If-elif-else Conditionals

# example 1
grade = 55
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")      You,
```

```
# example 2
grade = 40
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 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")
```

```python
# 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.

Time Complexity and Space Complexity

- **Time Complexity**: O(1) – Constant time, since only a few checks are made.
- **Space Complexity**: O(1) – Few variables used.

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. I had to check if they were equal or different and show the appropriate message using an if-else structure.

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".

Time Complexity and Space Complexity

- **Time Complexity**: O(1) – One comparison is made.
- **Space Complexity**: O(1) – Two variables only.

Output

```
Task: Compare Temperatures

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

Øther tasks are remaining

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

```python
# 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

```python
# 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.

Time Complexity and Space Complexity

- **Time Complexity**: O(n) – because the loop runs once for each name in the list.
- **Space Complexity**: O(1) – no extra space is used other than a few variables.

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'.

Time Complexity and Space Complexity

- **Time Complexity**:
    - O(1) for each individual tax calculation

    - O(n) overall, where n is the number of times the user wants to calculate tax
- **Space Complexity**: O(1) – the program only uses a few variables regardless of input size

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

This task was about writing a function that calculates compound interest over a number of years. The function needed to handle invalid inputs and print how the investment grows year by year. In the end, it returns the final value of the investment.

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.

Time Complexity and Space Complexity

- **Time Complexity**: O(n) – The function loops once for each year (n = duration)
- **Space Complexity**: O(1) – It uses a fixed amount of memory regardless of the duration

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.

Time Complexity and Space Complexity

- **Time Complexity**: O(1) – The function runs once and does a simple assignment and print.
- **Space Complexity**: O(1) – Only one variable is stored.

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.

Time Complexity and Space Complexity

- **Time Complexity**: O(n) – depends on the compound interest function's loop
- **Space Complexity**: O(1) – no extra space used

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.

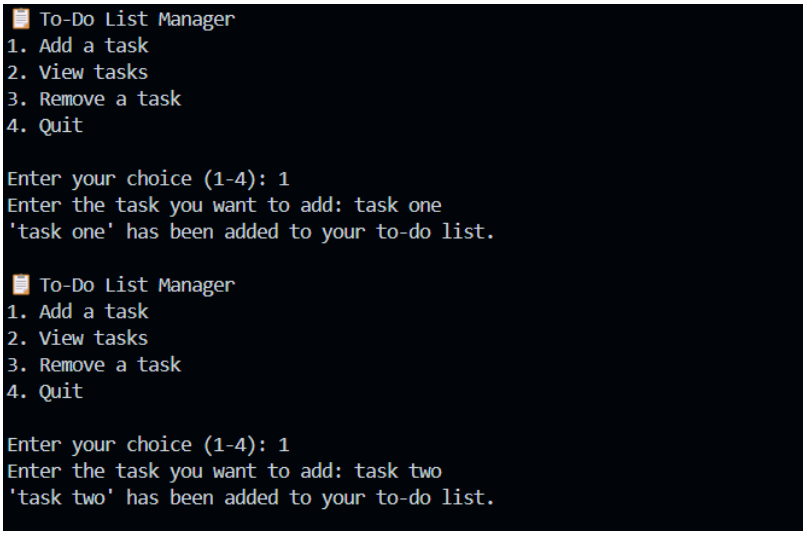
Time and Space Complexity

- **Time Complexity**:
    - Add Task: O(1)
    - View Tasks: O(n)
    - Remove Task: O(n) — because removing an item shifts the rest of the list
- **Space Complexity**:
    - O(n), where n is the number of tasks added by the user

Output

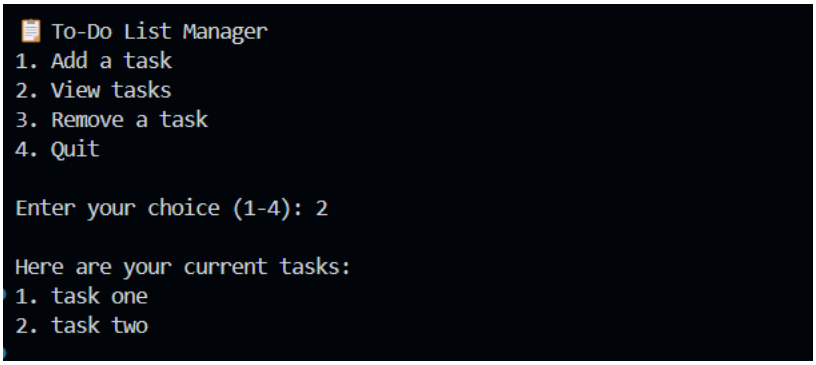
Choice 1

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

```python
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.

```python
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. However, the actual deletion line (`del self.tasks[index - 1]`) is commented out. Instead, it only prints debug information like the task's position and name. This shows that the logic was still under development or being tested.

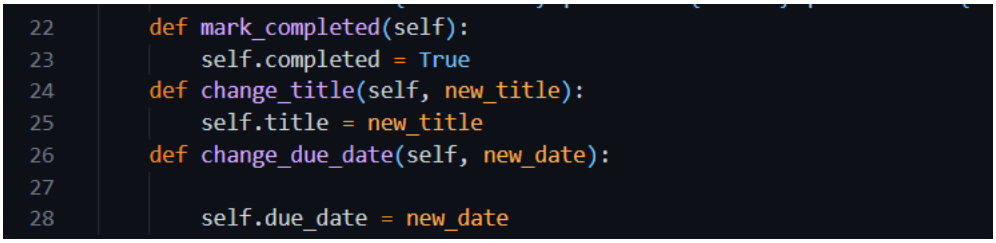
view_tasks(self):

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:
```

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.
```

Code

Choosing option 2 to view all the tasks. Its 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
```

Code

Choosing option 3 to delete the task

```python
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.

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:

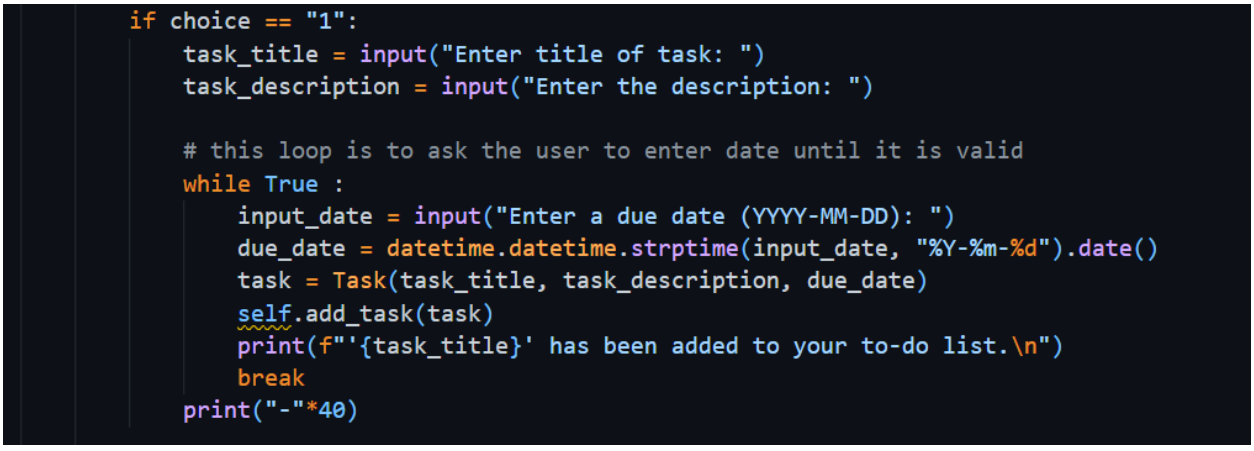
- Opened task_list.py and **copied the entire list_options() method's content**
- Pasted the code inside the main() function in main.py
- Removed the list_options() method from TaskList class
- Deleted the line task_list.list_options() and replaced it with the actual menu logic now inside main()

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.

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

```
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: 2


Your Current Tasks:
1. Task: Buy groceries | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-09 09:50:45.606904 | Description: Milk, eggs,
and bread
2. Task: Do laundry | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-15 09:50:45.606904 | Description: Wash and fold c
lothes
3. Task: Clean room | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-12 09:50:45.606904 | Description: Organize desk a
nd vacuum floor
4. Task: Do homework | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-16 09:50:45.606904 | Description: Finish math an
d science assignments
5. Task: Walk dog | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-18 09:50:45.606904 | Description: Evening walk arou
nd the park
6. Task: Do dishes | Status: Pending | Date Created: 2025-07-13 | Due Date: 2025-07-19 09:50:45.606904 | Description: Clean all utensi
ls after dinner
```

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.

```
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:

```
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.

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

```
    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

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

owner.py

```
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

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

```
Enter your choice: 2


The following tasks are still to be done:
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 21:39:30.177982 | Description: description
| interval: 7 days
8: Task: title 1 | Status: Pending | Date Created: 2025-07-13 | Due Date: 2022-02-02 | Description: desc 1
```

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.

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

```
15      def get_all_tasks(self) -> list[Task]:
16          task_list = []
17          with open(self.storage_path, "r") as file:
18              reader = csv.DictReader(file)
19              for row in reader:
20                  # getting each value from record one by one
21                  # Get the type of task (either Task or RecurringTask)
22                  task_type = row["type"]
23
24                   # Get title and due date as string
25                  title = row["title"]
26                  date_due_str = row["date_due"]
27                  date_due = None
28
29                  # Convert the due date string into a datetime object (handling different formats)
30                  if date_due_str != "":
31                      if "-" in date_due_str:
32                          # Format is likely YYYY-MM-DD
33                          date_due = datetime.datetime.strptime(date_due_str, "%Y-%m-%d").date()
34                      elif "/" in date_due_str:
35                          # Format is likely DD/MM/YYYY
36                          date_due = datetime.datetime.strptime(date_due_str, "%d/%m/%Y").date()
37                      else:
38                          print(f"Unknown date format: {date_due_str}")
39                          date_due = None  # or set a default/fallback
40
41                  # Check if the task was marked as completed
42                  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:

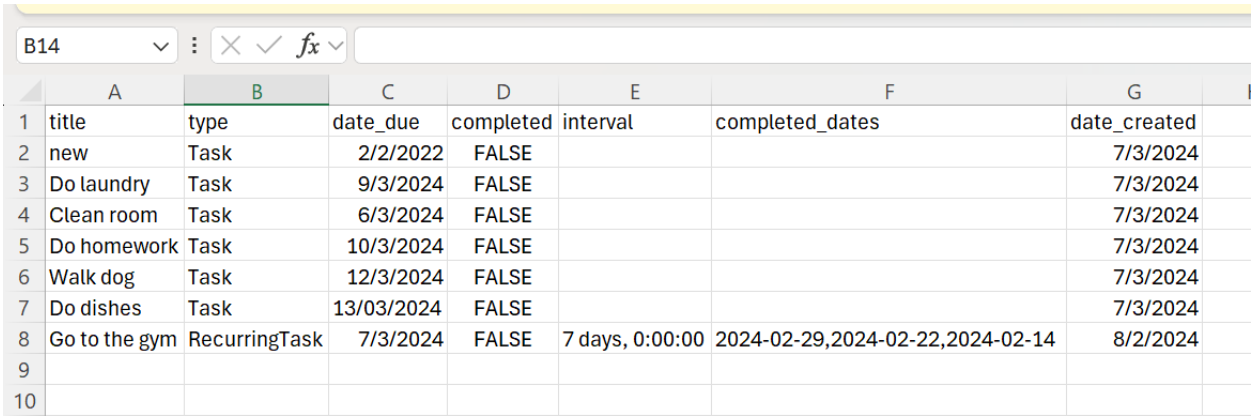
- Opens the CSV file and reads it using csv.DictReader, so each row becomes a dictionary.
- Check the type column to decide if it's a regular Task or a RecurringTask.
- Parses important values like:
    - date_due and date_created using two possible formats (YYYY-MM-DD or DD/MM/YYYY).
    - Converts 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.**

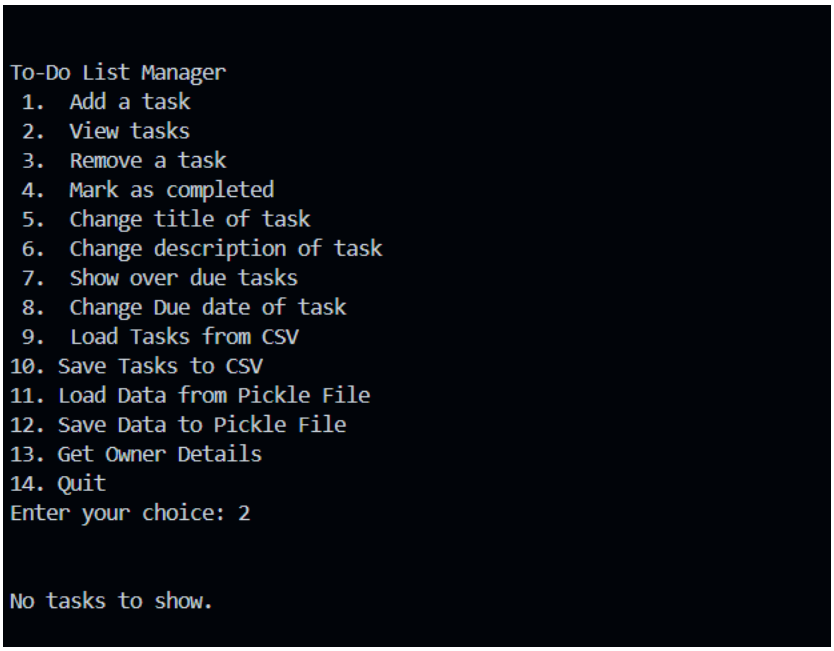
CSV file



B14

	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 A: 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

```python
    def save_all_tasks(self, tasks: list[Task]) -> None:
        # create a set of existing task titles to check for duplicates
        existing_titles = set()

        # If file exists, load existing tasks to check for duplicates
        if os.path.exists(self.storage_path):
            with open(self.storage_path, "r", encoding="utf-8") as file:
                reader = csv.DictReader(file)
                for row in reader:
                    # Add the title of each existing task to the set
                    # This helps us avoid saving duplicate tasks later
                    existing_titles.add(row["title"])  # Assuming title is unique

        # Open the file in append mode so we can add new tasks without deleting old ones
        with open(self.storage_path, "a", newline='', encoding="utf-8") as file:
            writer = csv.DictWriter(file, fieldnames=self.fieldnames)

            # If the file is empty, we add a header row to it first
            if os.stat(self.storage_path).st_size == 0:
                writer.writeheader()

            # Go through each task in the list
            for task in tasks:
                # If this task is already in the file (based on title), skip it
                if task.title in existing_titles:
                    continue  # Skip duplicate tasks


                # This row dictionary will hold all the task details to be written
                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								

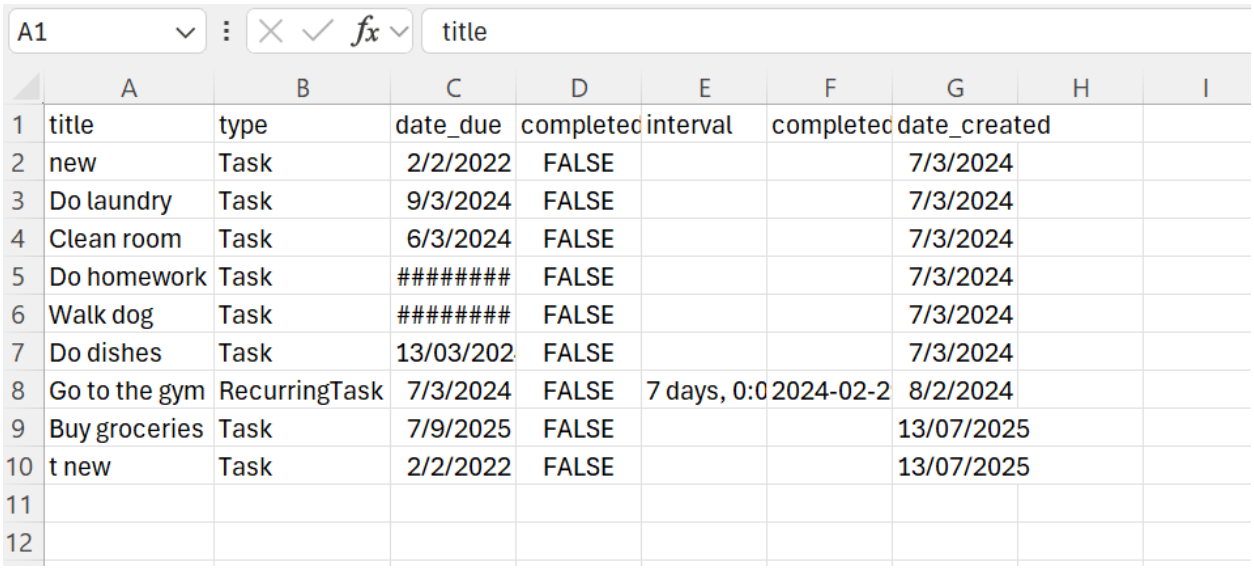
Choosing opt 1 to add a task, then choosing 1 to add a one time task.

```
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:



	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

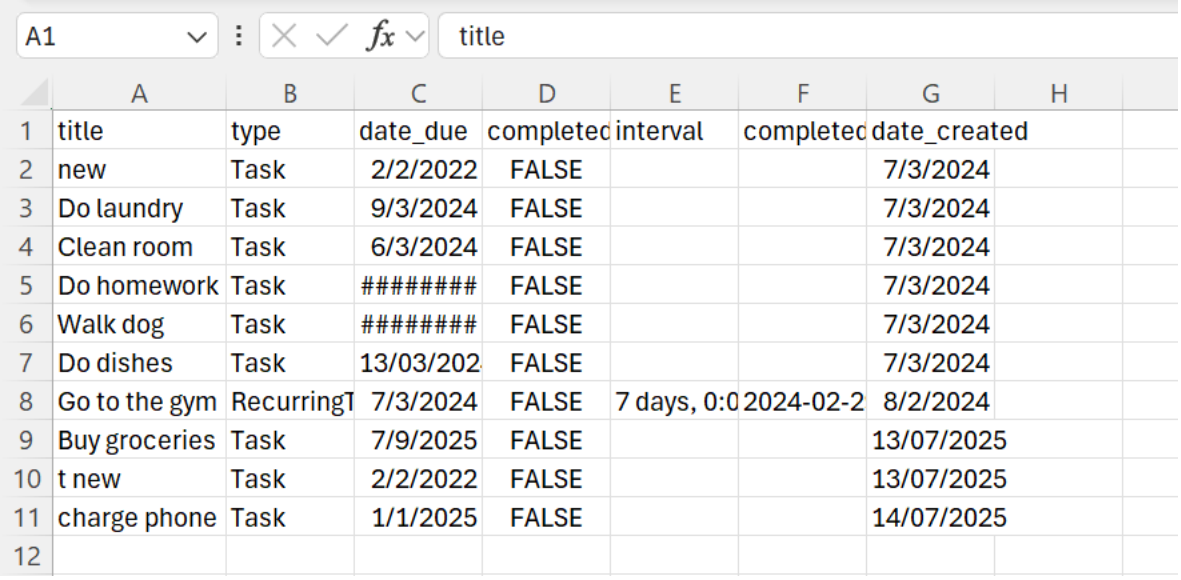
Choosing 1 and then 1 to add a one time task.

```
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



A1 title

	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.

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.

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.