Week 2

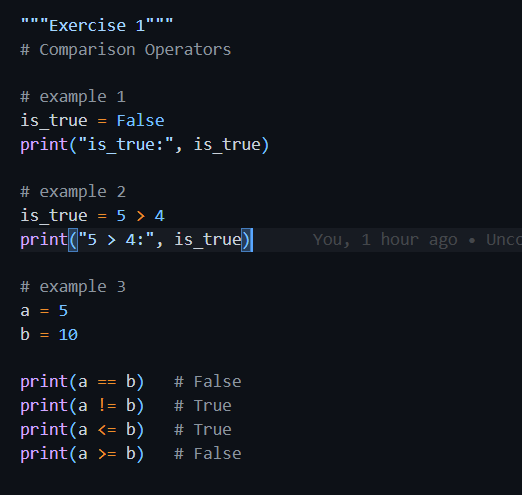
# Section 1. Comparisons and Conditionals

## 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.

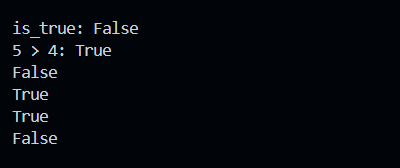
### Source Code:



### 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.

### Output

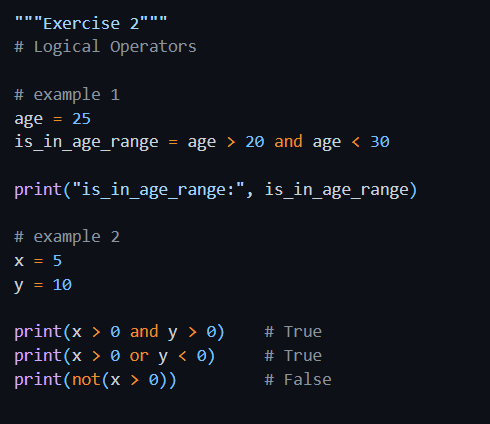


## Exercise 2: Logical Operators

### 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.

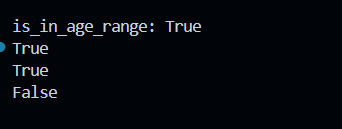
### Source Code



### 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



## 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



### 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.

### Output



## 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

A screen shot of a computer program

AI-generated content may be incorrect.

### Explanation

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

### Output

A black background with white text

AI-generated content may be incorrect.

## 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

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of white text

AI-generated content may be incorrect.

A computer screen shot of a computer code

AI-generated content may be incorrect.

A computer screen shot of white text

AI-generated content may be incorrect.

### Explanation

* Four different grades were tested.
* The code checked each grade and printed a message:
  + Below 50 → “You failed”
  + Between 50–59 → “You passed”
  + Between 60–69 → “You got a good pass”
  + 70 and above → “You got an excellent pass”
* Each elif allows checking in sequence, and only one block runs depending on the value.

### Output

A black background with white text

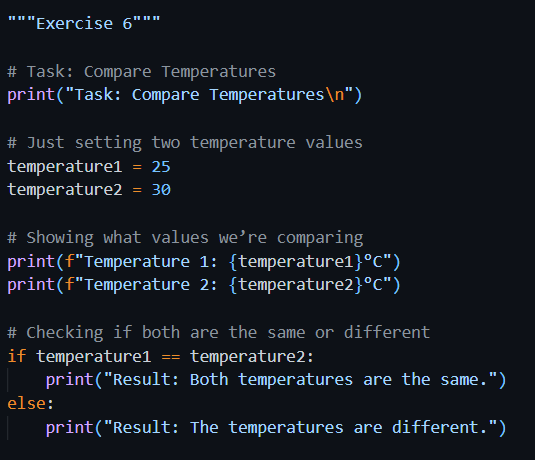
AI-generated content may be incorrect.

## 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



### Explanation

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

### Output

A screen shot of a computer

AI-generated content may be incorrect.

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



##### 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



#### Function Parameters

##### Source Code 1

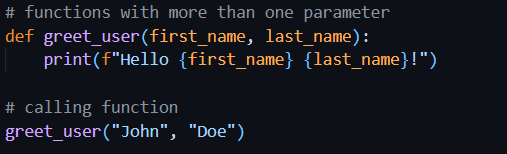
A computer screen with text

AI-generated content may be incorrect.

##### Output 1



##### Source Code 2



##### Output 2

A black background with a black square

AI-generated content may be incorrect.

##### 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

A black background with white text

AI-generated content may be incorrect.

##### 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

A black screen with a black background

AI-generated content may be incorrect.

#### Default Values

##### Source Code

A screen shot of a computer code

AI-generated content may be incorrect.

##### Explanation

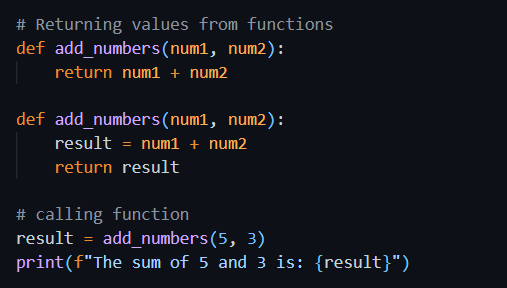
This function has a default value for the university parameter. If I don’t pass the value for it, it uses "UWS" by default. But I can also override it by giving a custom value like "UWS London". This is useful when some arguments usually have a common value.

##### Output

****

#### Return

##### Source Code

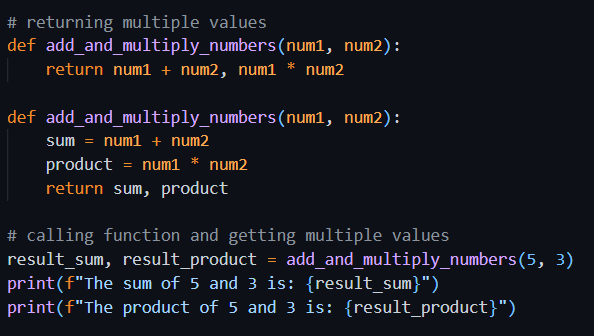


##### Output

A black background with white text

AI-generated content may be incorrect.

##### Source Code



##### Output

A black background with white text

AI-generated content may be incorrect.

##### 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



### Explanation

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

### Output

A black screen with white text

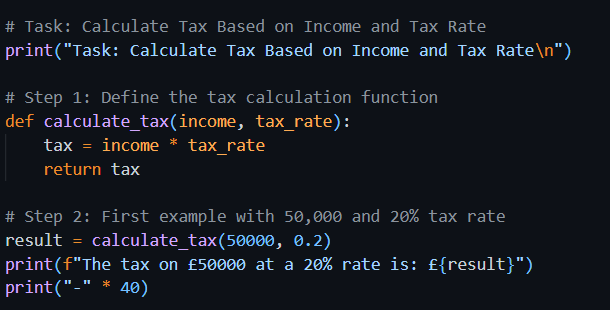
AI-generated content may be incorrect.

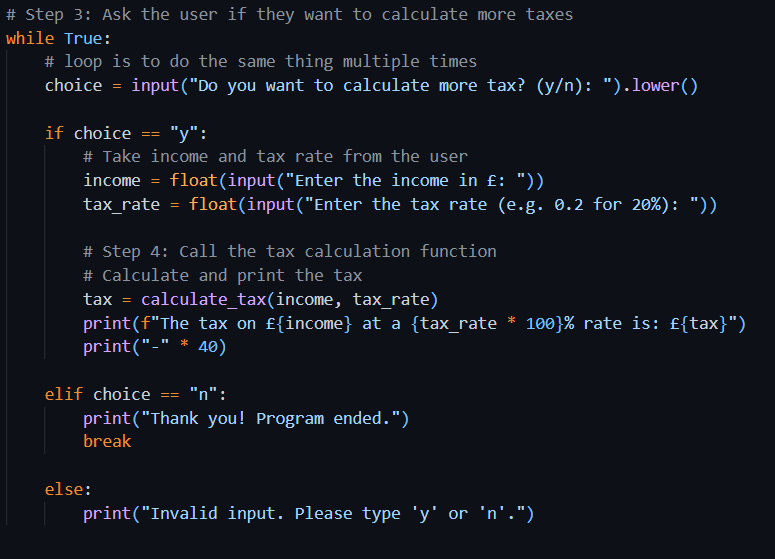
## Task: Calculate Tax Based on Income and Tax Rate

### 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





### Explanation

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

### Output

A black screen with white text

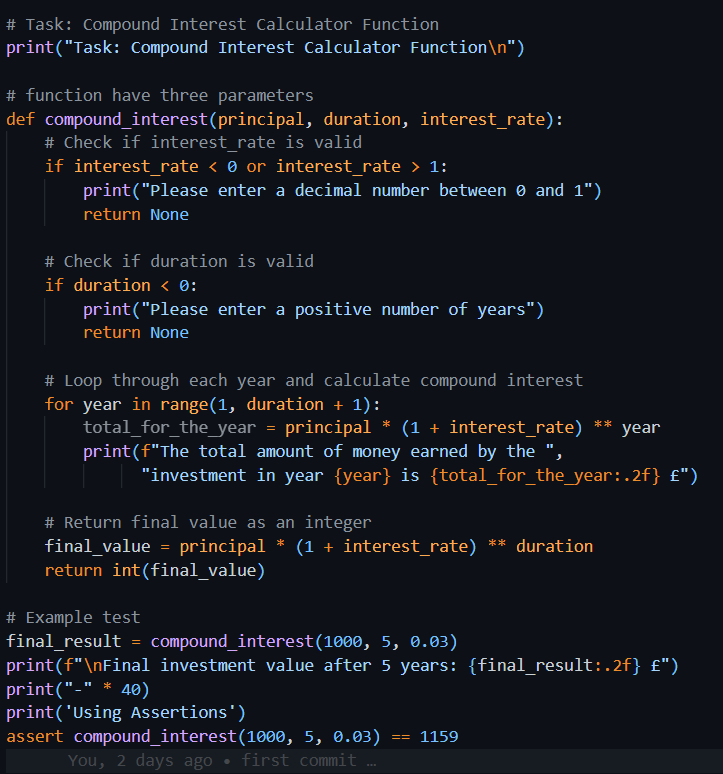
AI-generated content may be incorrect.

## 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.

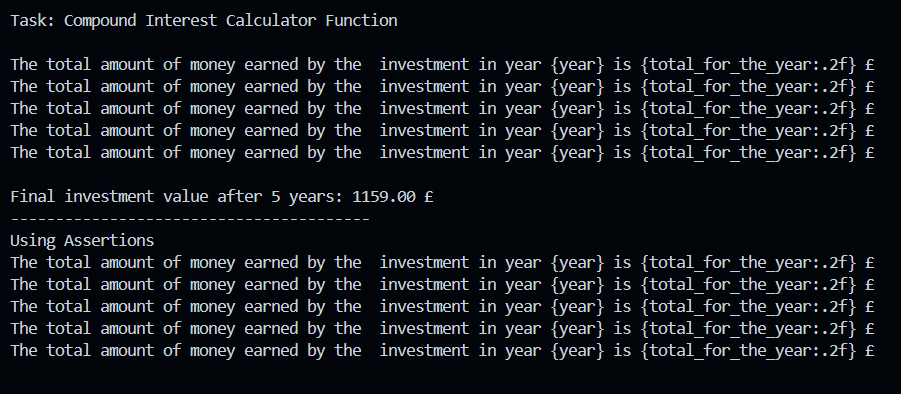
### Source Code



### Explanation

* The function compound\_interest takes three parameters: principal, duration, and interest\_rate.
* It first checks if the interest rate is between 0 and 1 and if the duration is positive. If not, it shows an error and exits early.
* Then, it uses a for loop to calculate and print the total amount at the end of each year using the compound interest formula:  
  (principal×(1+rate)year)(principal × (1 + rate) ^ year)(principal×(1+rate)year)
* Finally, it returns the total value at the end of the given duration, rounded to an integer.
* I tested the function with a £1000 investment for 5 years at 3% interest, and it correctly printed the values for each year and returned the final result.

### Output

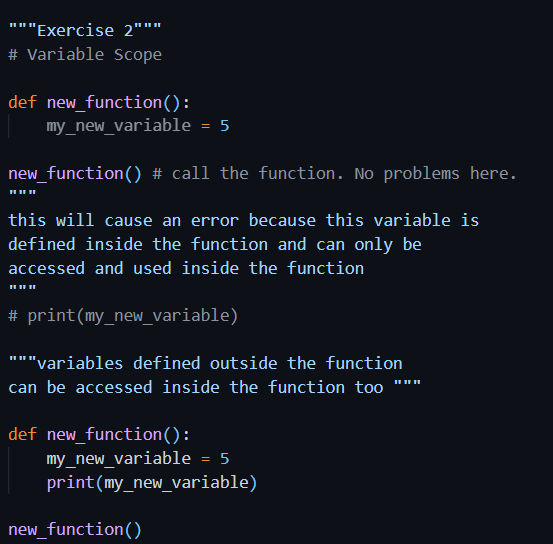


## Exercise 2: Variable Scope

### 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



### Explanation

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

### Output

#### Error

A computer screen shot of white text

AI-generated content may be incorrect.

#### Corrected Code Output



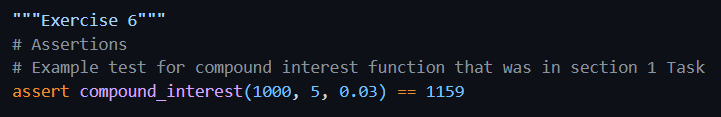
# Section 2: Assertions and Errors

## 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



### Explanation

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

### Output

A screenshot of a computer screen

AI-generated content may be incorrect.

## Exercise 7: Identifying and Fixing Common Errors

### Understanding the Task

#### Syntax Error

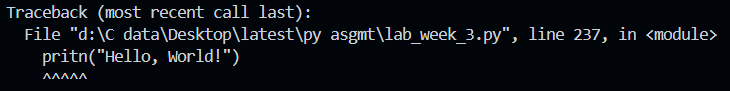
Occurs when Python code is written incorrectly and doesn't follow the proper rules — e.g., a typo like pritn() instead of print().

##### Source Code 1 – Has Error

A black background with white text

AI-generated content may be incorrect.

##### Output 1



##### Source Code – Corrected

A black background with white text

AI-generated content may be incorrect.

##### Output 2

A black background with a black square

AI-generated content may be incorrect.

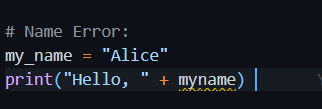
##### Explanation

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



##### Output 1

##### 

##### Source Code 2 – Corrected Code

A black background with white text

AI-generated content may be incorrect.

##### Output 2

A black background with white text

AI-generated content may be incorrect.

##### 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

A black background with white text

AI-generated content may be incorrect.

##### Output 1

A screen shot of a computer

AI-generated content may be incorrect.

##### Source Code 2 – Corrected Code

A screen shot of a computer

AI-generated content may be incorrect.

##### Output 2



##### 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

A black background with white text

AI-generated content may be incorrect.

##### Output 1

A black background with white text

AI-generated content may be incorrect.

##### Source Code 2 – Corrected Code

A screen shot of a computer

AI-generated content may be incorrect.

##### Output 2



##### 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

A black background with white text

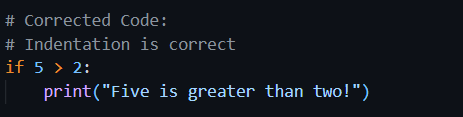
AI-generated content may be incorrect.

##### Output 1

A screen shot of a computer

AI-generated content may be incorrect.

##### Source Code 2 – Corrected Code



##### Output 2

A black background with white text

AI-generated content may be incorrect.

##### 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

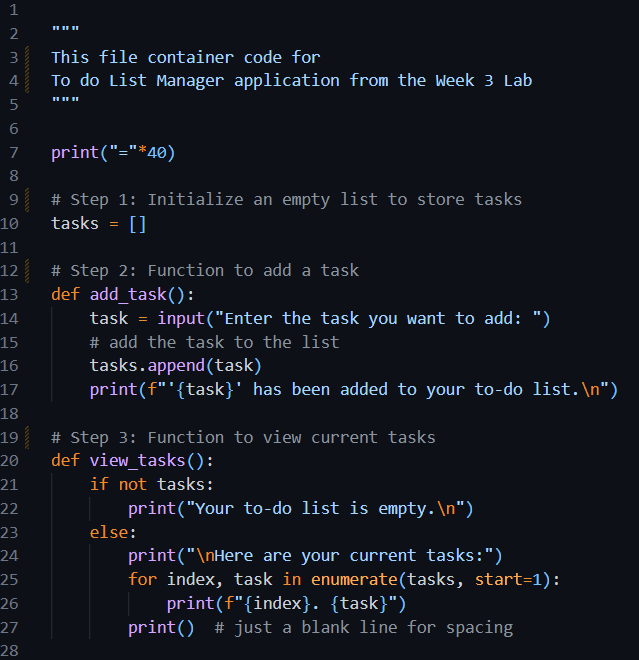
## Task: To-Do list manager:

### 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



A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

### Explanation

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

#### 1. Initialize the Task List

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

#### 2. Adding a Task

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

#### 3. Viewing All Tasks

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

#### 4. Removing a Task

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

#### 5. Menu and Loop

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

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

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

### Output

###### Choice 1

Choosing 1 to add a task (one by one)

A screenshot of a computer program

AI-generated content may be incorrect.

###### Choice 2

Choosing 2 to view tasks.

A screen shot of a computer

AI-generated content may be incorrect.

###### Choice 3

Choosing 3 to delete a task.

A screenshot of a computer program

AI-generated content may be incorrect.

###### Choice 4

Quitting the program

A black screen with white text

AI-generated content may be incorrect.

Week 4

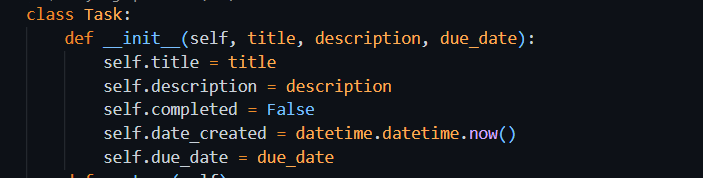
# Section 1. Python Classes

## 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



### 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

A blue circle with a black background

AI-generated content may be incorrect.

### 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:



And for Task List class:

A black background with white and blue text

AI-generated content may be incorrect.

## 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

A computer screen shot of text

AI-generated content may be incorrect.

### Explanation

#### mark\_completed() Method

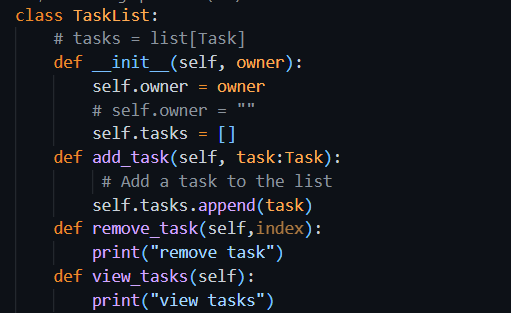
* This method is intended to mark the task as completed.
* Currently, it just prints a message, but in a full version, it would update the completed status to True.

#### 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.



#### 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.

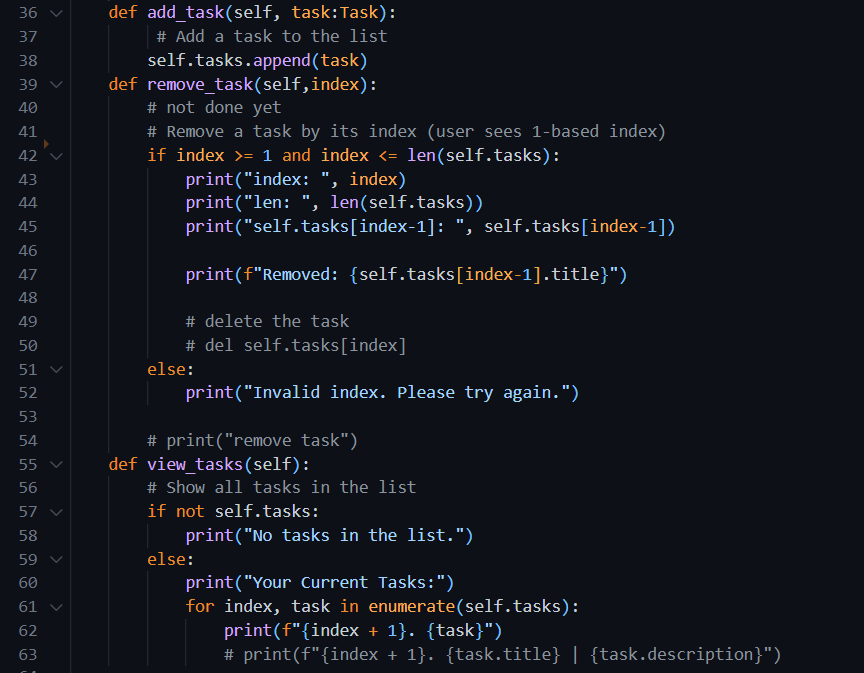
## Task: Add logic to methods defined

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

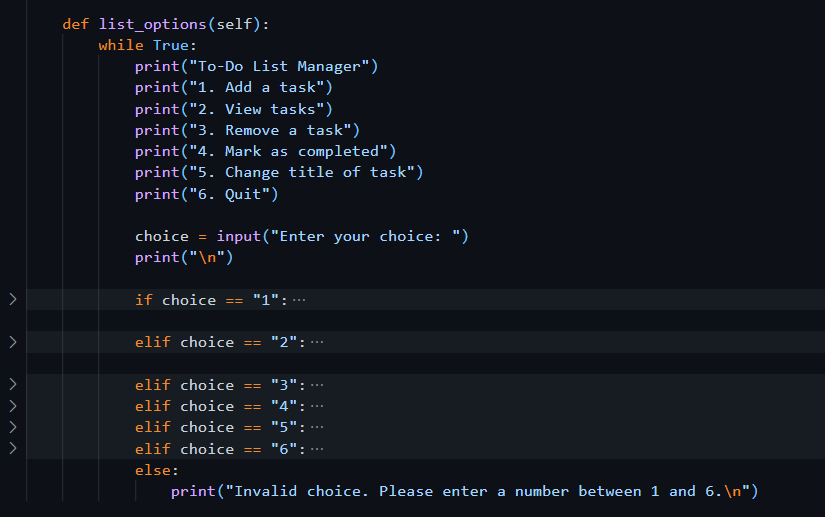
A computer code with text

AI-generated content may be incorrect.

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.



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



## Exercise 3: Testing the Functionality

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

### 

### Output

A screenshot of a computer

AI-generated content may be incorrect.

### Code

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

A computer screen shot of a program

AI-generated content may be incorrect.

### Output

A computer screen with white text

AI-generated content may be incorrect.

### Code

Choosing option 2 to view all the tasks. Its working correctly

A blue circle in the dark

AI-generated content may be incorrect.

### Output

A black screen with white text

AI-generated content may be incorrect.

### Code

Choosing option 3 to delete the task



### Output

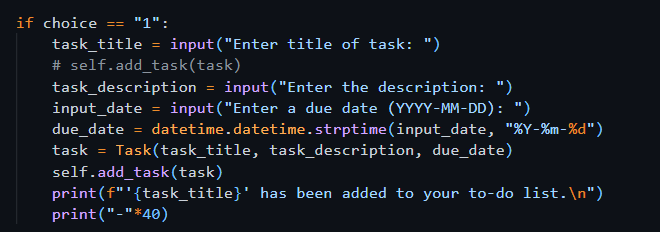
A computer screen with white text

AI-generated content may be incorrect.

## 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



### Explanation

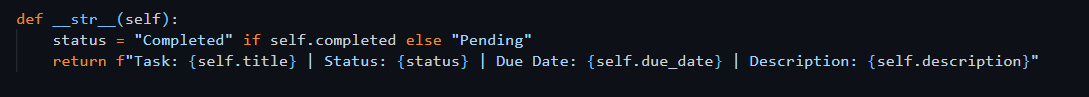
This code is taking title, desc and due date of a task as input from user. It then saves information using object of Task Class. It then adds that object to the Task List class using add\_task() method. In this way TaskList can have many Tasks

## \_str\_ method

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



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.

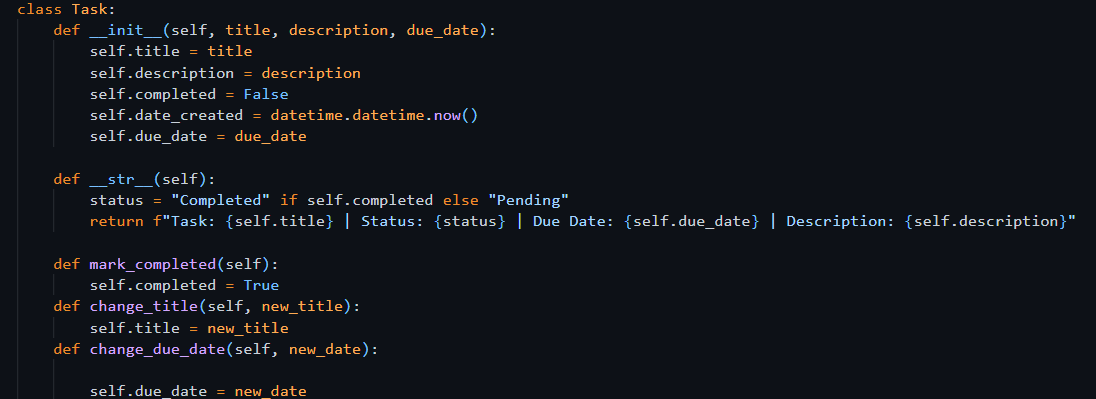


## Task: Change code in the Task Class

### 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



## 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.

A computer screen shot of white text

AI-generated content may be incorrect.

If Elif statements

A computer screen shot of a program code

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.

# 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

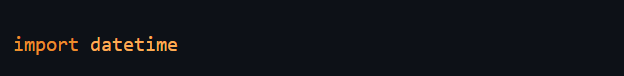
It is important for each task to have both the date it was created and the due date. To handle this, I used Python’s built-in **datetime** library. A method called **change\_due\_date** was also added so that the due date can be updated later if needed.

### 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



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.

A black background with colorful text

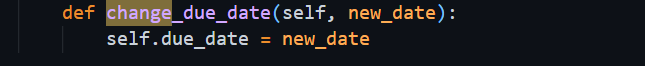
AI-generated content may be incorrect.

Or getting the present date and time

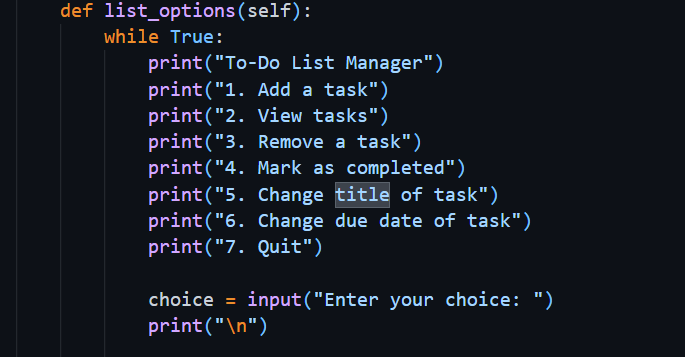


## Task: Add the due\_date functionality

For changing due date of the task, I have created ‘change\_due\_date()’



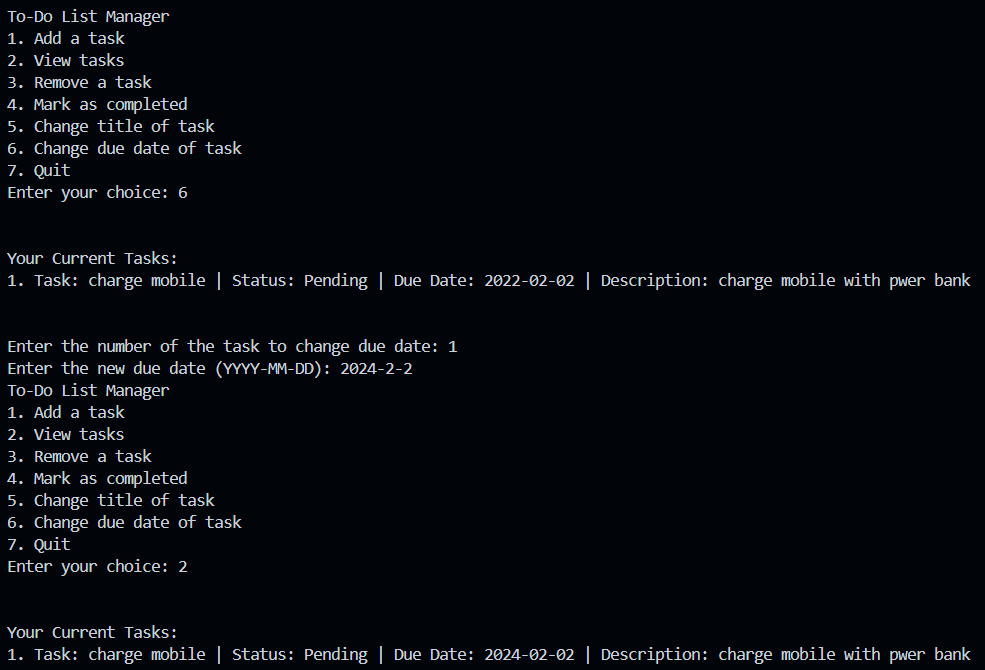
Modification of If-else statements in list\_options() method for changing due date



A computer screen shot of code

AI-generated content may be incorrect.

### Output



# 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.

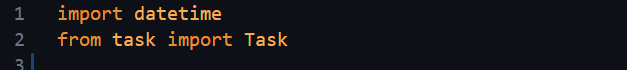
### Output

* 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



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:

A black background with green text

AI-generated content may be incorrect.

### Source Code

A computer screen shot of text

AI-generated content may be incorrect.

A black background with text

AI-generated content may be incorrect.

### 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:

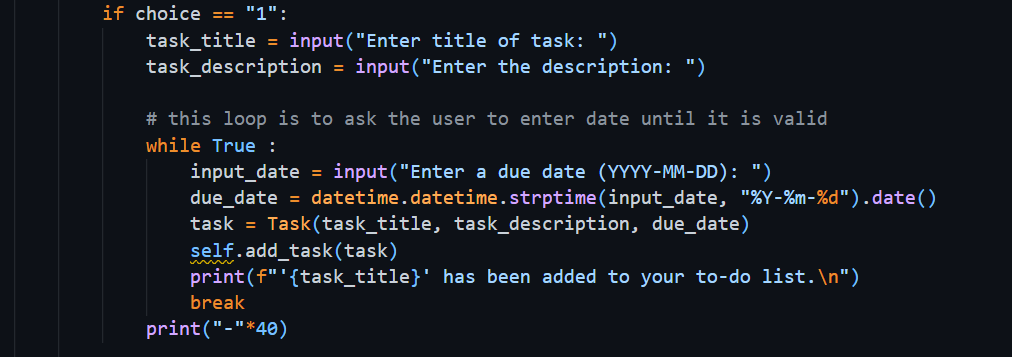
* Opened task\_list.py and **copied the entire list\_options() method’s content**
* Pasted the code inside the main() function in main.py
* Removed the list\_options() method from TaskList class
* Deleted the line task\_list.list\_options() and replaced it with the actual menu logic now inside main()

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.



### Explanation

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

A black background with white text

AI-generated content may be incorrect.

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:



Now, to call methods on this object, I changed self to task\_list, like:



## 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

A computer screen shot of text

AI-generated content may be incorrect.

#### Usage

A computer screen shot of a computer program

AI-generated content may be incorrect.

### 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

A screenshot of a computer program

AI-generated content may be incorrect.

# 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.

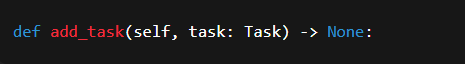
#### Type Hints for Method Parameters

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



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:



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

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:



#### 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:

A black background with white text

AI-generated content may be incorrect.

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

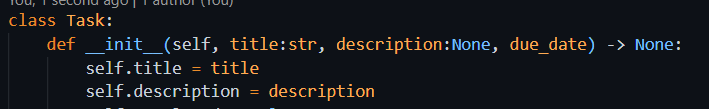
In this task, I was asked to improve the Task class by adding an optional **description** feature. This allows each task to have some extra details written about it, but it's not required.

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).

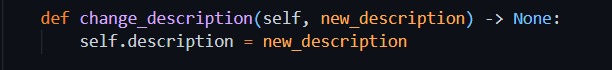
### Explanation

* I started by updating the constructor like this:



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

* Then I added a method:



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.

A computer screen shot of a program code

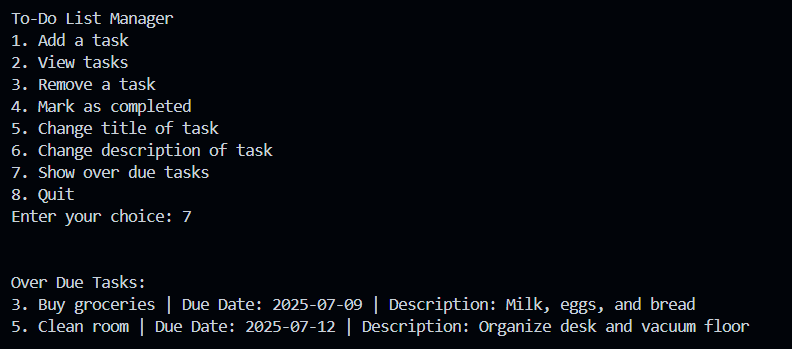
AI-generated content may be incorrect.

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

A screenshot of a computer program

AI-generated content may be incorrect.

### Output



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:



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:

A screen shot of a computer screen

AI-generated content may be incorrect.

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

A computer screen shot of text

AI-generated content may be incorrect.

##### Output

A screen shot of a computer

AI-generated content may be incorrect.

#### 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.

A computer screen with blue and white text

AI-generated content may be incorrect.

Creating the object with form factor:



## 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.

A black background with orange text

AI-generated content may be incorrect.

A screen shot of a computer program

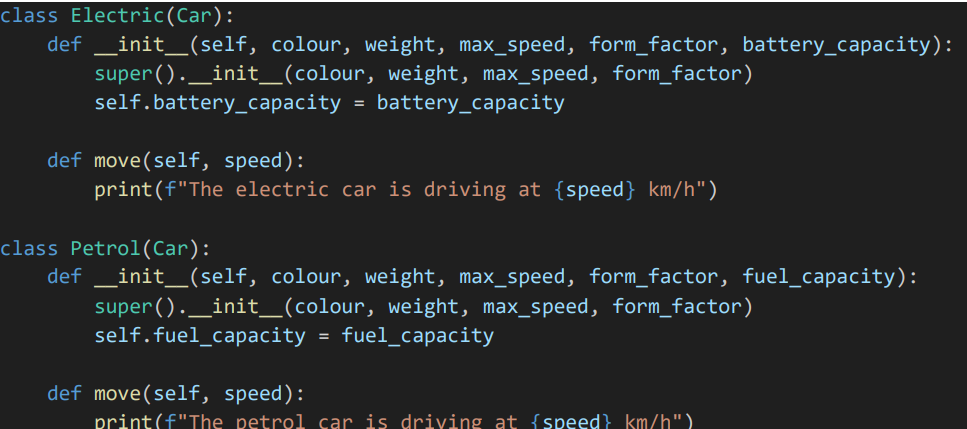
AI-generated content may be incorrect.

### Output

A black background with white text

AI-generated content may be incorrect.

## Task: Create ElectricCar and the PetrolCar class.



#### Test

A computer code with text

AI-generated content may be incorrect.

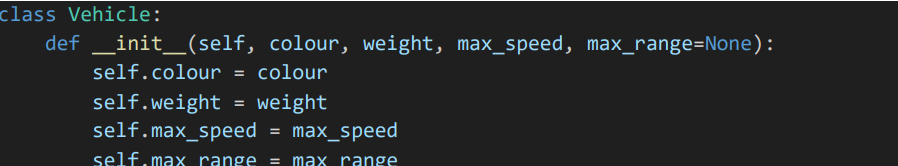
##### Output

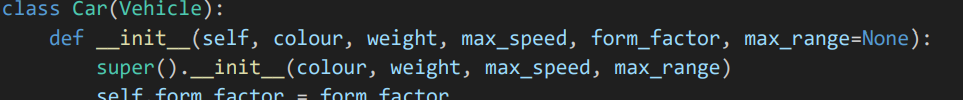
A black background with white text

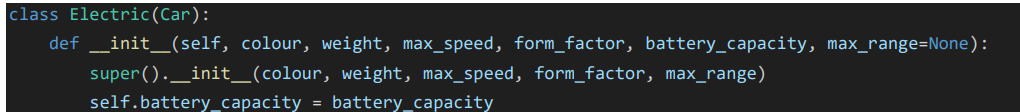
AI-generated content may be incorrect.

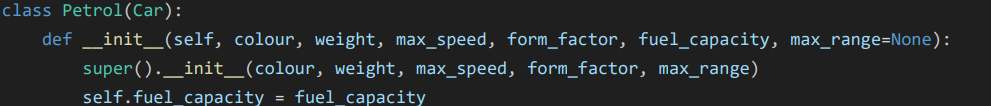
## Task: Adding max range parameter

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.

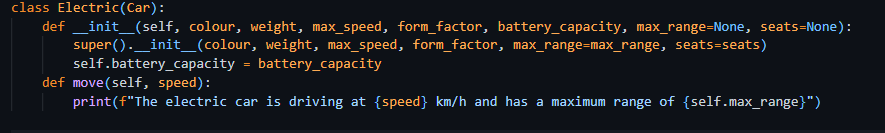






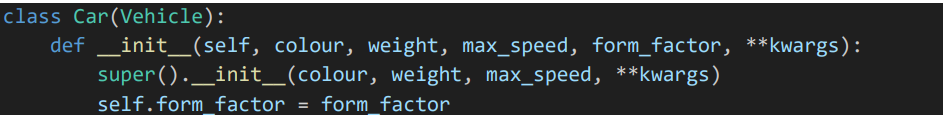


Using max range in electric car move method



## 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.



### 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

A black screen with white text

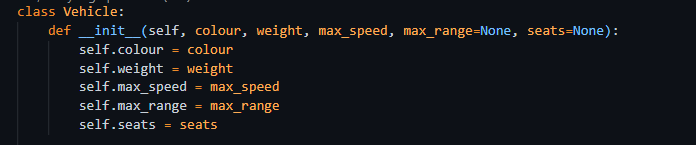
AI-generated content may be incorrect.

### Output

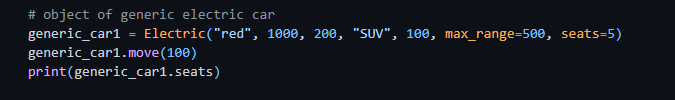
A black background with white text

AI-generated content may be incorrect.

## Task: Adding seats attribute



### Test



### Output



## 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:

A screen shot of a computer code

AI-generated content may be incorrect.

* Then, I created two more child classes:
  + Propeller(Plane) → adds propeller\_diameter
  + Jet(Plane) → adds engine\_thrust

A computer screen shot of text

AI-generated content may be incorrect.

* 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

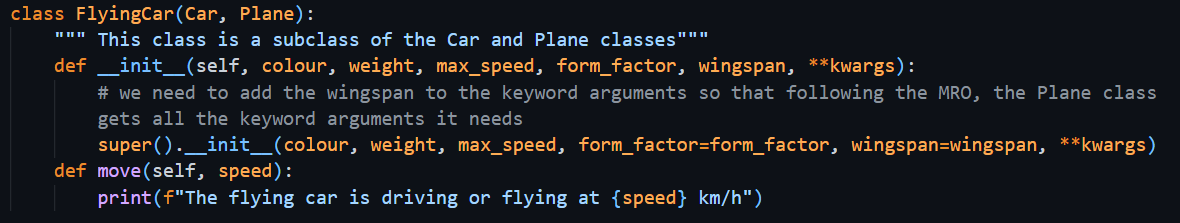
### Understanding the Task

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

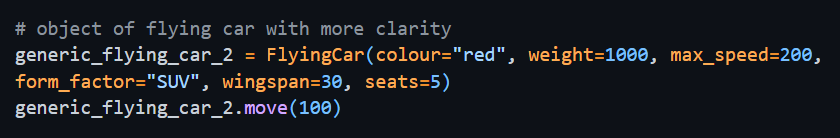
* I created the FlyingCar class.



* 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.





### Output

A black background with white text

AI-generated content may be incorrect.

# 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

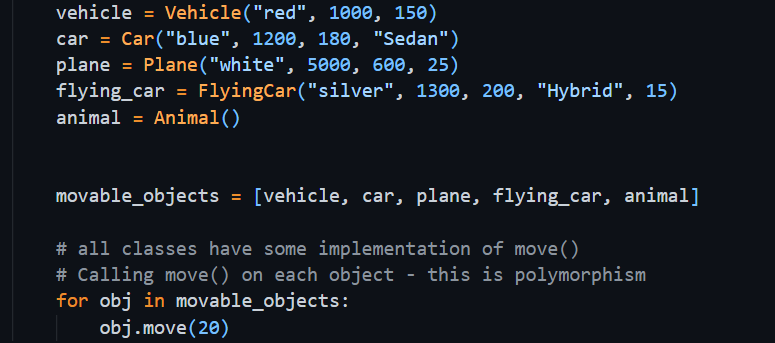
The most common example of polymorphism is when a **parent class has a method**, and each **child class overrides** it with its own behavior. It means every child class also has the method with same name and same arguments but with different implementation

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



### 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

A screen shot of a computer

AI-generated content may be incorrect.

# 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.

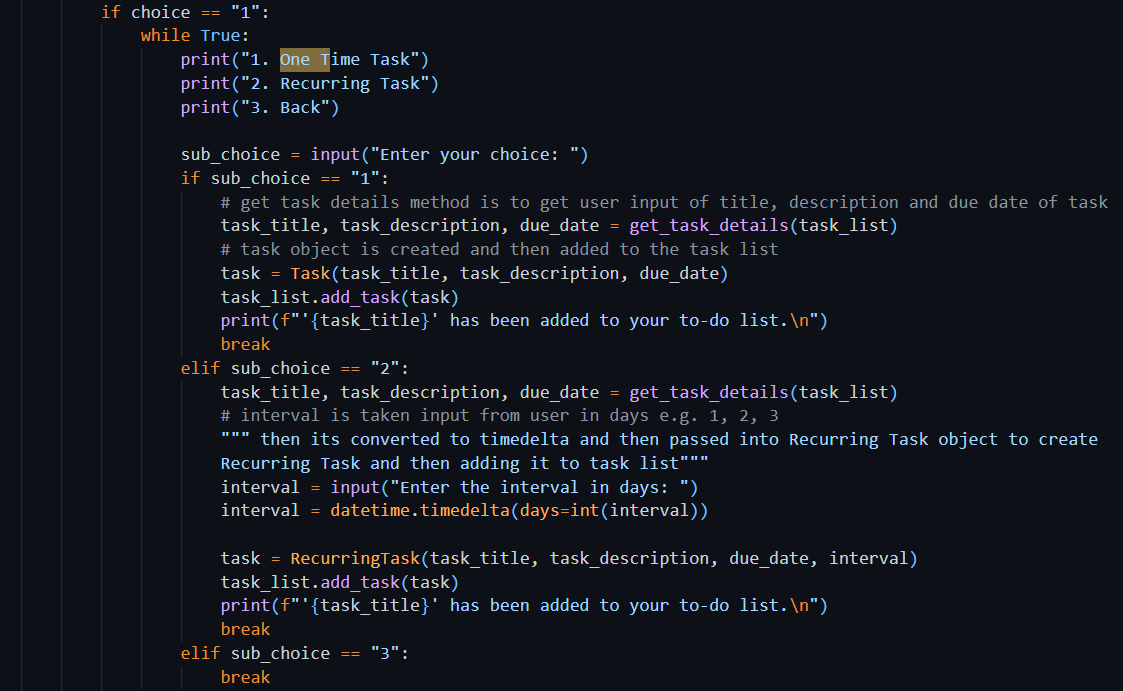
### Explanation

* 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

A computer screen with text on it

AI-generated content may be incorrect.

* 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:



### 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

A computer screen shot of a task

AI-generated content may be incorrect.

A black background with white text

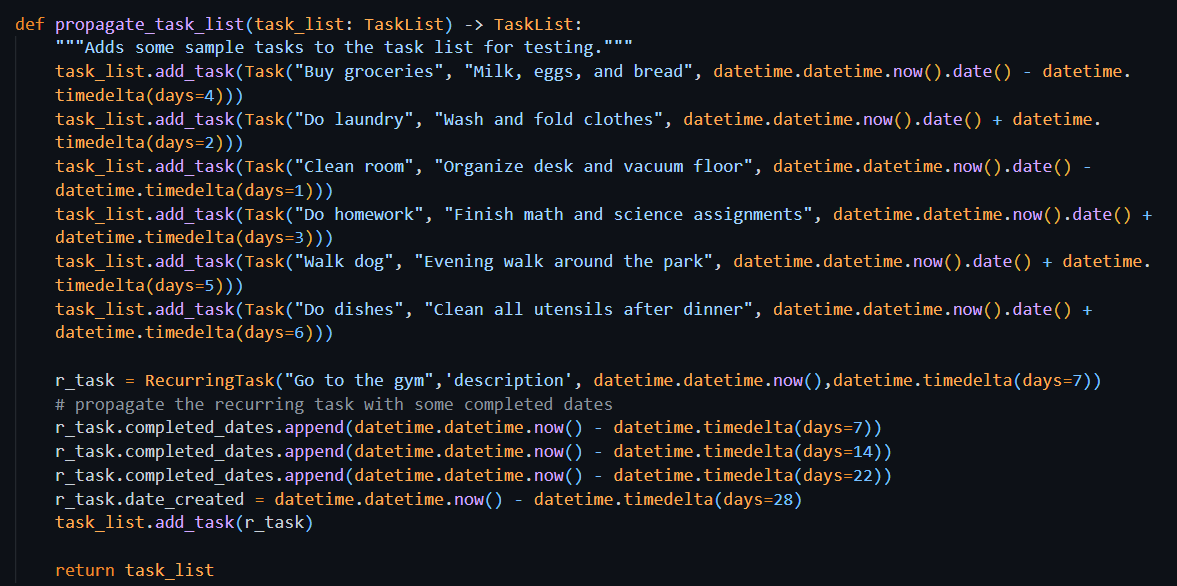
AI-generated content may be incorrect.

## 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



### 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

A computer screen with text

AI-generated content may be incorrect.

### 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

A computer screen shot of a program

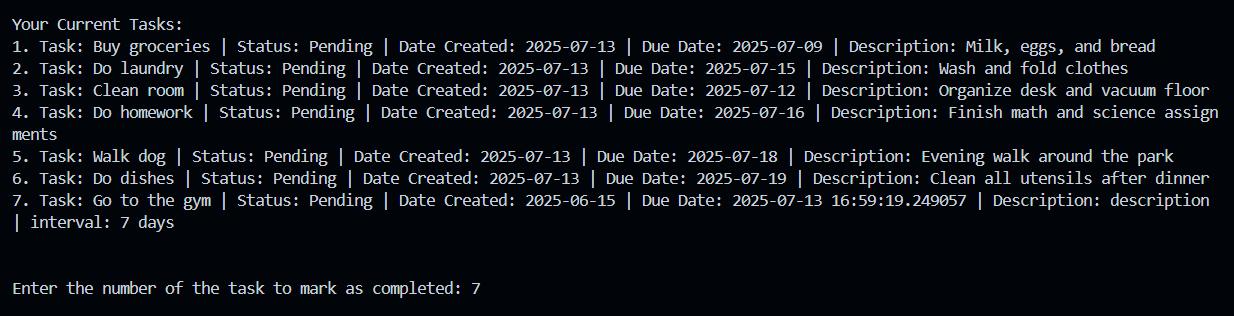
AI-generated content may be incorrect.

### 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



A black screen with white text

AI-generated content may be incorrect.

## Exercise 4 – Encapsulation

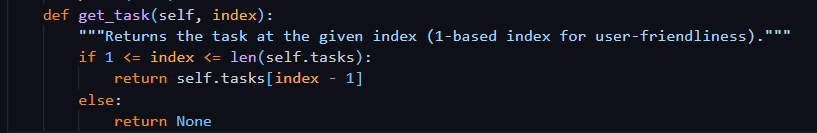
### Understanding the Task

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.

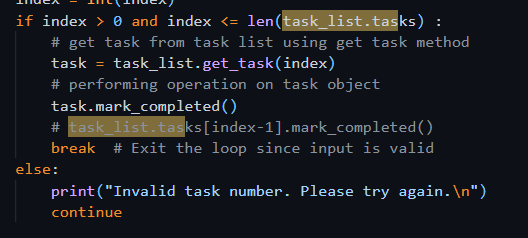
The goal was to apply the concept of **encapsulation**, which means hiding internal details and only exposing what’s necessary. I needed to:

* 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



### Example Usage



### 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

A black background with orange and white text

AI-generated content may be incorrect.

#### owner.py

A computer screen shot of a black background

AI-generated content may be incorrect.

#### Accept owner object in TaskList class

A computer screen with text

AI-generated content may be incorrect.

#### Usage in main function

A computer screen shot of white text

AI-generated content may be incorrect.

### 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

A black background with white text

AI-generated content may be incorrect.

## 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

A computer screen shot of an email

AI-generated content may be incorrect.

A computer screen shot of a computer code

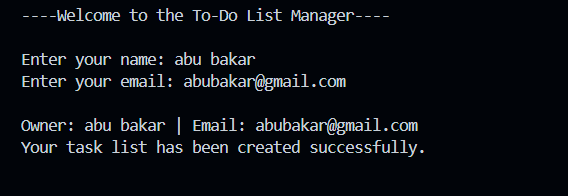
AI-generated content may be incorrect.

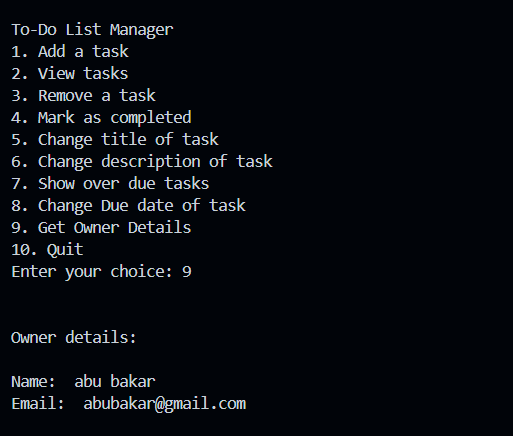
### Usage Example

A computer screen shot of a computer program

AI-generated content may be incorrect.

### Output





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

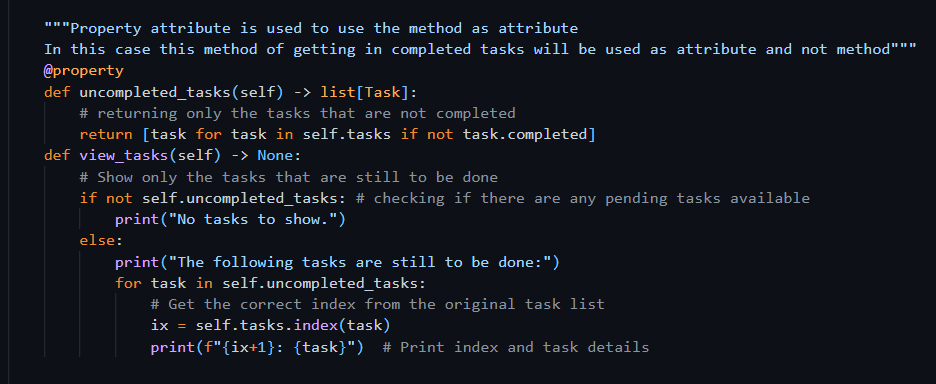
### Understanding the Task

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



### Explanation

The uncompleted\_tasks() method:

* Is now used like a variable (self.uncompleted\_tasks) instead of self.uncompleted\_tasks().
* This makes code inside view\_tasks() cleaner and easier to read.

In the view\_tasks() method, this property is used to:

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

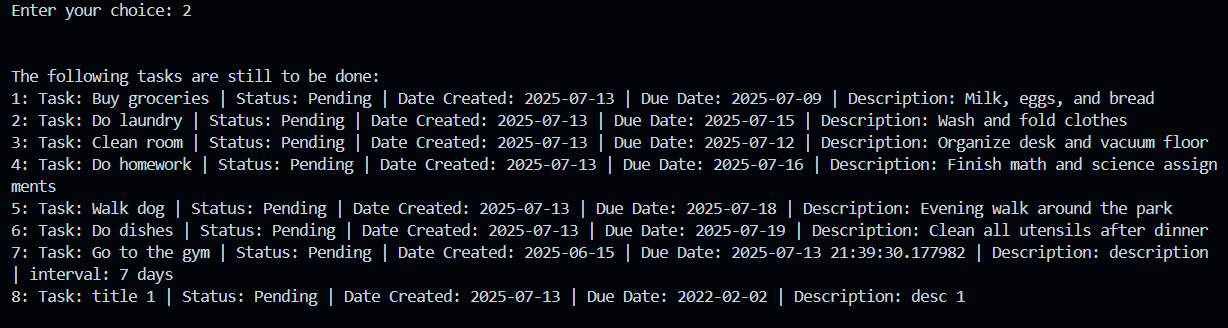
The index is retrieved from the original self.tasks list to keep numbering consistent.

### Why Is This Useful?

* It hides logic behind a simple interface
* Improves readability
* Maintains proper encapsulation
* Make code easier to maintain and extend later

### Output

Choosing 2 to view the tasks



All pending tasks are shown only

# Section 3. Implementing Persistence

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

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

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

1. **Text Files (.txt)**  
   You can save plain text data in a file and read it later. This method is simple and useful for basic lists or logs.
2. **CSV Files (.csv)**  
   Ideal for tabular data — each row represents a record, and each column a field. It’s commonly used when storing structured data like tasks, scores, or tables.
3. **JSON Files (.json)**  
   This format allows you to store data in key-value pairs (dictionaries) or lists. It's perfect for saving complex objects like task lists with multiple fields (title, due date, completed status, etc.).
4. **Databases (e.g., SQLite, MySQL)**  
   For larger or more complex applications, using a database is a better approach. Databases provide advanced features like search, filtering, sorting, and relations between different data types.

## Exercise 1: DAO

### Understanding the Task

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

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

Then, I updated the main function to:

* Let the user **choose when to load/save tasks**
* Ask for a file path (just for simulation)
* Create an object of TaskTestDAO
* Use get\_all\_tasks() to load predefined tasks and add them to the current task list

### Source Code



### 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.

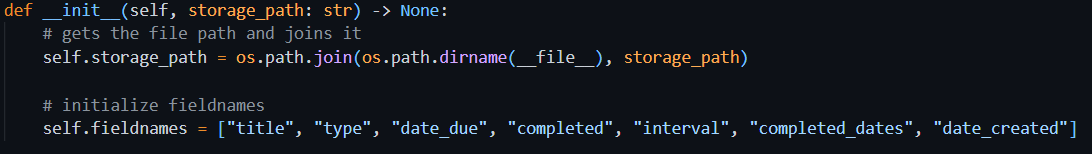
## Exercise 2: CSV Persistence

### 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.



## Task A: Complete get\_all\_tasks() functionality

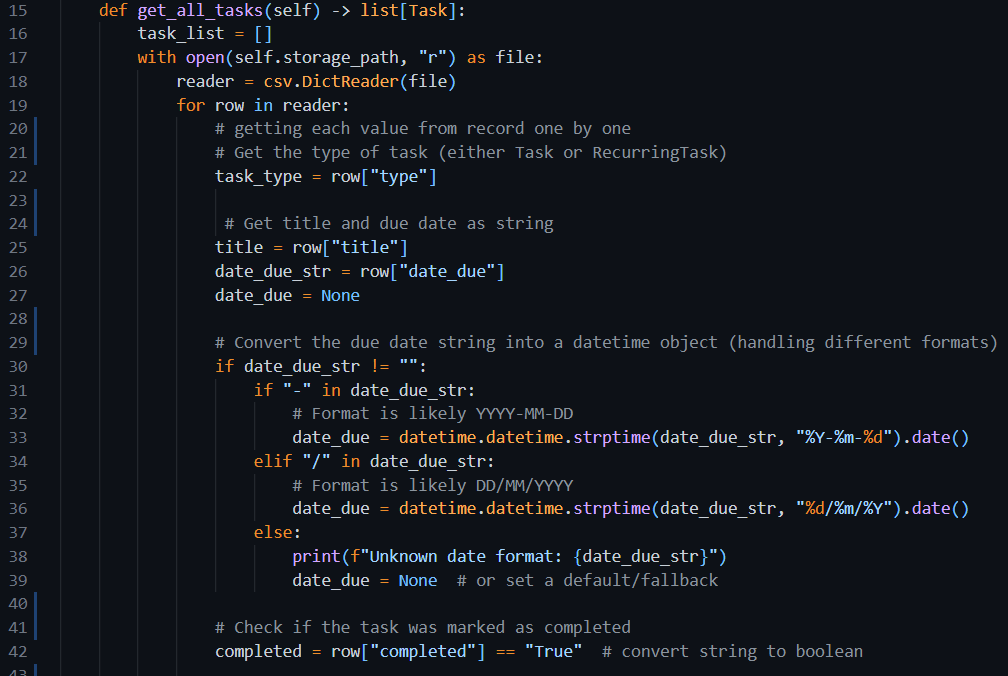
### Understanding the Task

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



A screen shot of a computer code

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

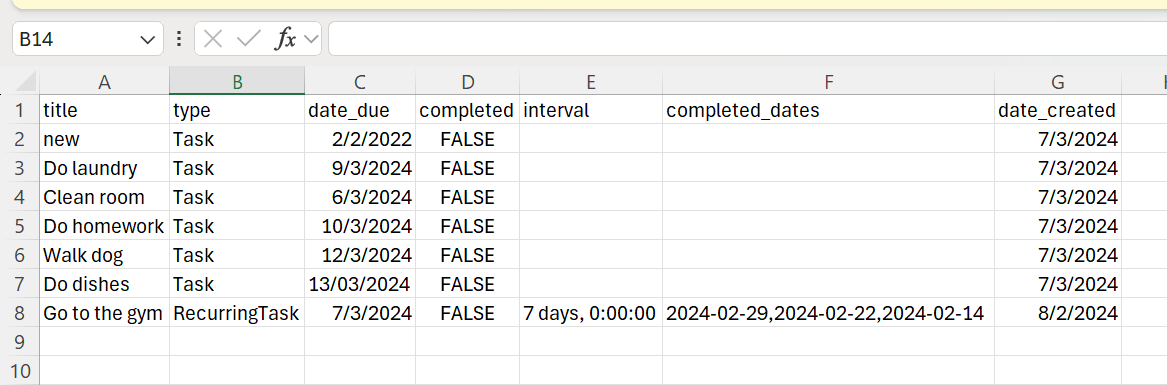
### Explanation

The get\_all\_tasks() method does the following:

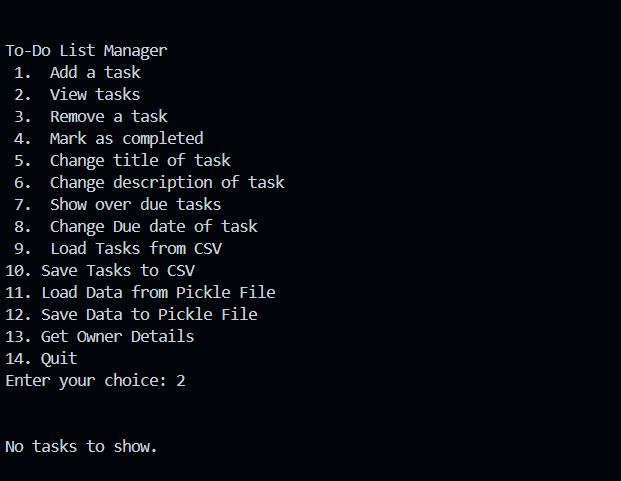
* Opens the CSV file and reads it using csv.DictReader, so each row becomes a dictionary.
* Check the type column to decide if it's a regular Task or a RecurringTask.
* Parses important values like:
  + date\_due and date\_created using two possible formats (YYYY-MM-DD or DD/MM/YYYY).
  + Converts the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



First, there are no tasks present in the program



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

A black background with white text

AI-generated content may be incorrect.

Choosing 2 to view tasks which are loaded from the file

A computer screen with text

AI-generated content may be incorrect.

## 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



A computer screen with text

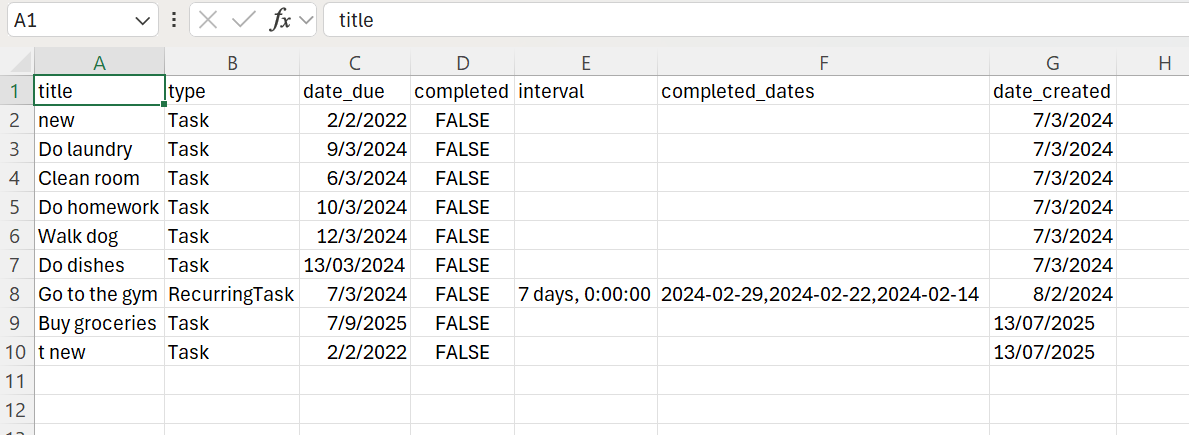
AI-generated content may be incorrect.

### 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



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

A computer screen with white text

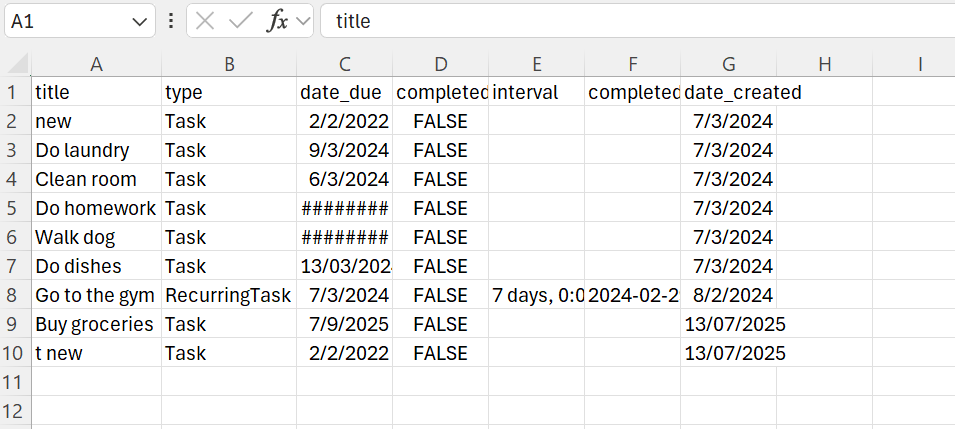
AI-generated content may be incorrect.

A computer screen with text

AI-generated content may be incorrect.

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:



Which shows that duplicate is not saved.

#### Example 2

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

A screenshot of a computer

AI-generated content may be incorrect.

Choosing 10 to save data to file

A black background with white text

AI-generated content may be incorrect.

Output in csv

A screenshot of a computer

AI-generated content may be incorrect.

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.

The format of pickle file looks like this:

A computer screen shot of white text

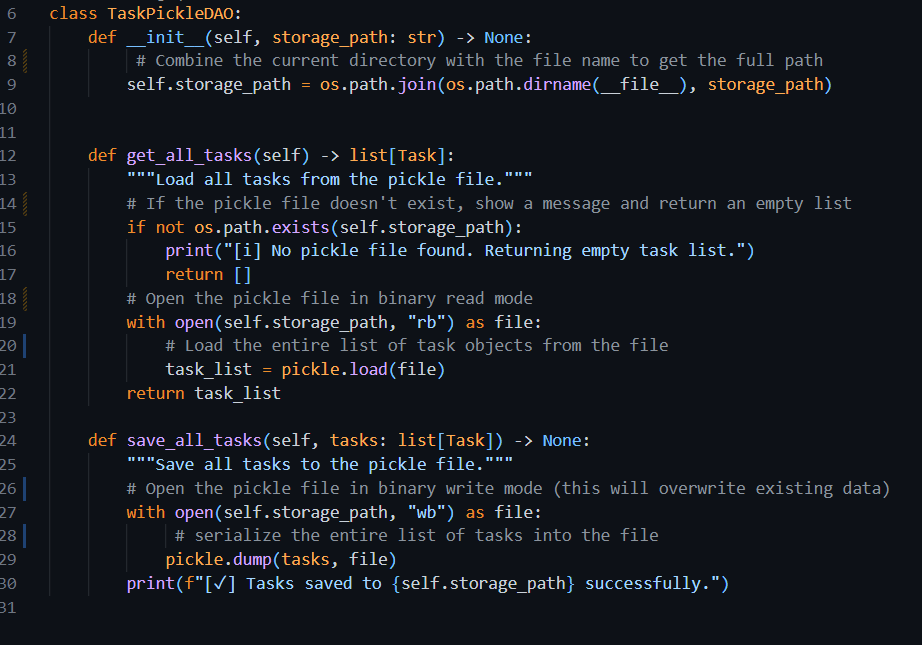
AI-generated content may be incorrect.

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

### Understanding the Task

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

### Source Code

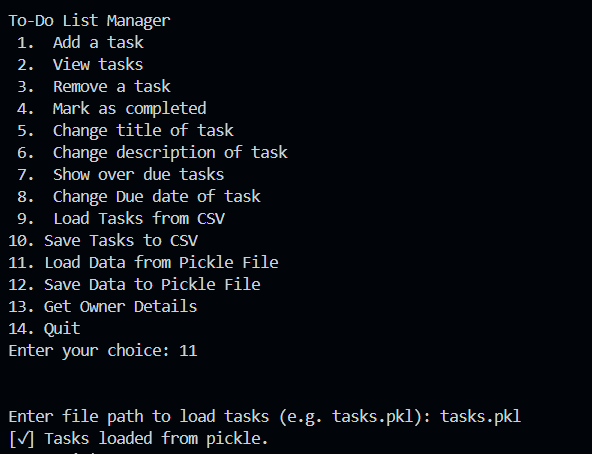
****

### Explanation

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

### Output

Choosing 11 to load data from pickle file



Choosing 2 to view tasks

A computer screen shot of a person's face

AI-generated content may be incorrect.

Choosing 1 and 2 to add a recurring task

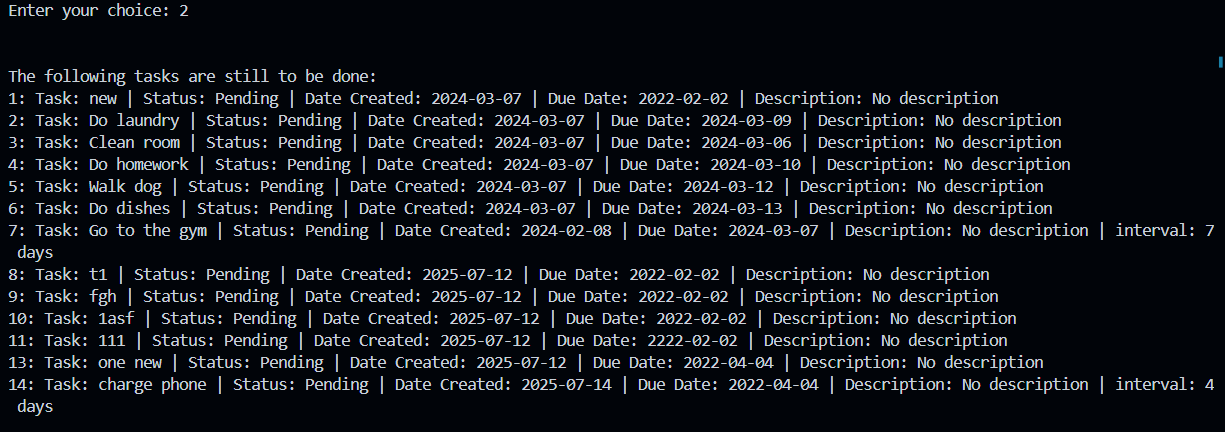
A screenshot of a computer

AI-generated content may be incorrect.

Choosing 12 to save data to pickle file A black background with white text

AI-generated content may be incorrect.

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



It is saved in the file

Week 7

# Section 1. SOLID Principles

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

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

## Exercise 1. Single Responsibility Principle

### Understanding the Task

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

### Source Code

A screen shot of a computer code

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

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

### Explanation

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

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

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

### Benefits of SRP in My ToDoApp

* Make the code **cleaner and easier to understand**.
* If I want to update one feature (like how tasks are saved), I don’t have to dig through unrelated code.
* Make **testing** easier since each class can be tested in isolation.
* Help reduce **bugs** and confusion when the codebase grows.
* **Improves reusability** – I can reuse the same Task logic in other apps, without copying extra unrelated stuff.

## Exercise 2. Open/Closed Principle

### Understanding the Task

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

### Explanation

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

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

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

### Benefits of Using OCP

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

## Exercise 3: Liskov Substitution Principle (LSP)

### Understanding the Task

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

### Explanation:

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

### Benefits:

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

## Exercise 4: Interface Segregation Principle (ISP)

### Understanding the Task

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

### Explanation

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

### Benefits

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

## Exercise 5: Dependency Inversion Principle (DIP)

### Understanding the Task

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

### Explanation

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

### Benefits

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

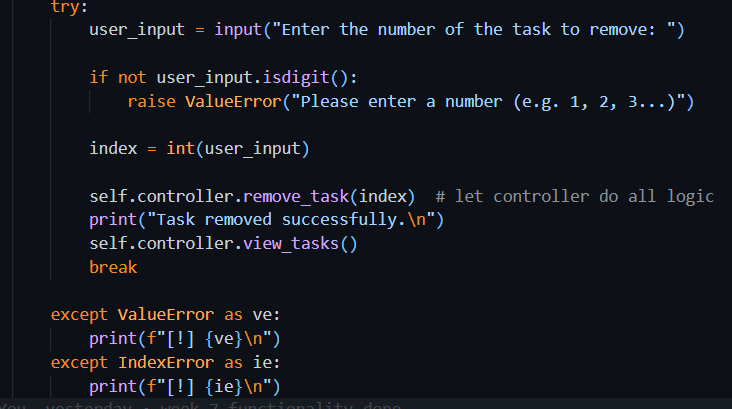
# Section 2. Exception Handling

## Task: ToDoApp Modification by adding exception handling

### Understanding the Task

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

#### Example 1



##### Explanation

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

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

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

#### Example 2

A computer screen shot of a program

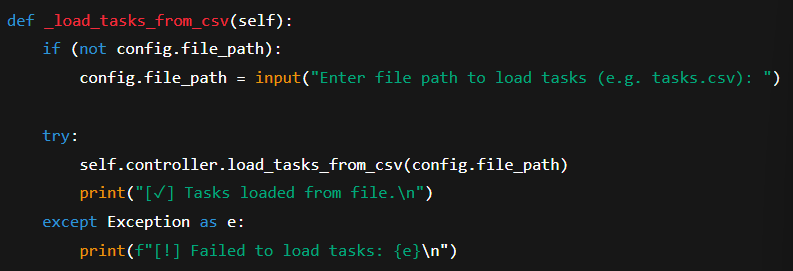
AI-generated content may be incorrect.

##### Explanation

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

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

#### Example 3



##### Explanation

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

### Benefits of Exception Handling

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

# Section 3. Improving ToDoApp according to SOLID Principles

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

### Understanding the Task

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

I had to:

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

### Explanation

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

The **CommandLineUI** class handles:

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

The **TaskManagerController** class handles:

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

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

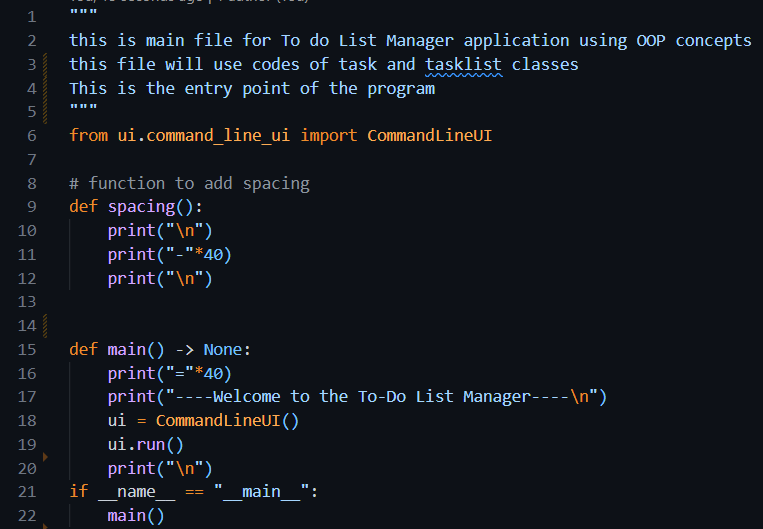
This design pattern makes the app easier to extend in the future e.g., adding a GUI interface later won't require changing the task logic.

### Benefits of This Restructure

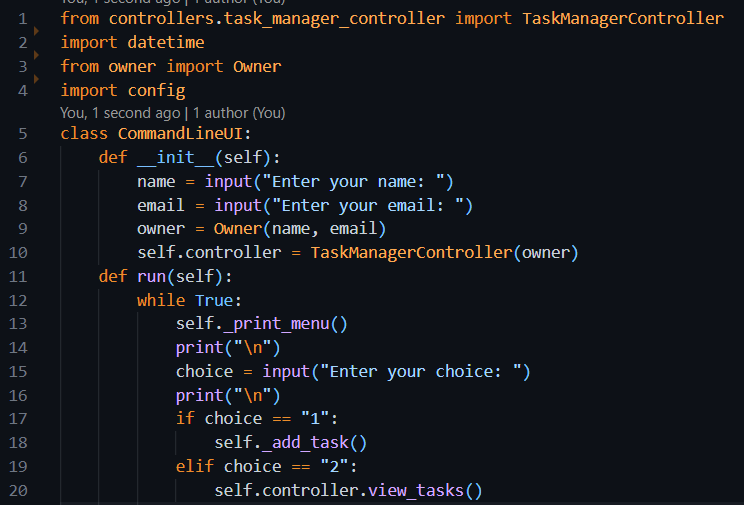
* **Easier to maintain and modify**: UI changes don’t affect task logic, and vice versa.
* **Modular structure**: Each class has a clear responsibility, following **SRP** (Single Responsibility Principle).
* **No code duplication**: By centralizing task creation in TaskFactory, and task validation in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

### Source Code

Main.py



command\_line\_ui.py



A computer screen shot of text

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.