# Week 5 – Inheritance & Polymorphism

In week 5, we learnt about Object-Oriented-Programming’s main pillar Inheritance and Polymorphism.

## Inheritance:

A mechanism that allows a class to inherit properties and behaviours from another (parent) class. It is a fundamental concept of OOP to reuse the functionalities which already exist and modify the behaviour according to needs. Python natively supports Multiple Inheritance with doesn’t exist in most of the languages.

## Polymorphism:

Polymorphism means having different signatures with same name for different inputs/outputs. Here in python polymorphism doesn’t support like in other languages such as Java & C#, but it can take any type of parameters as input and returns any type of value.

## Example code for Inheritance

# Base class

class Vehicle:

    def \_\_init\_\_(self, colour: str, weight: int, max\_speed: int, max\_range: int | None = None, seats: int | None = None):

        self.colour = colour

        self.weight = weight

        self.max\_speed = max\_speed

        self.max\_range = max\_range

        self.seats = seats

    def move(self, speed: int):

        print(f"The vehicle is moving at {speed} km/h")

    def get\_total\_seats(self):

        if self.seats != None:

            return self.seats

        return 0

# Child class. Car is inherited by Vehicle

class Car(Vehicle):

    def \_\_init\_\_(self, colour: str, weight: int, max\_speed: int, car\_type: str, max\_range = None, seats = None):

        super().\_\_init\_\_(colour, weight, max\_speed, max\_range, seats) # parent class constructor

        self.seats = seats

    def move(self, speed: int):

        print(f"The car is driving at {speed} km/h")

electric\_car = Electric("yellow", 1200, 180, "Sedan", 1500, max\_range=330, seats=4)

electric\_car.move(95)

petrol\_car = Petrol("green", 1800, 270, "SUV", 45, max\_range=845, seats=4)

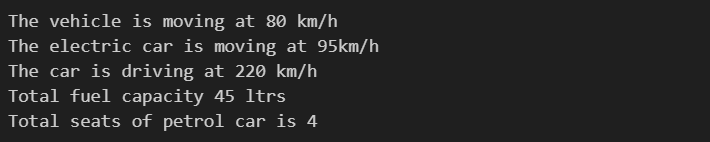
petrol\_car.move(220)

print(f"Total fuel capacity {petrol\_car.fuel\_capacity} ltrs")

print(f"Total seats of petrol car is {petrol\_car.get\_total\_seats()}")

Here we have 2 Classes **Vehicle** and **Car**, Vehicle is a base class which acts as a starting point for other classes like Car. Car is inherited from Vehicle class and has all the properties that base class contains. When initializing the Car class, it is important to call the base class constructor, so that properties will get initialized. In the example code, the move() function is overridden with custom code to match with the class, this is how we’re able to use inheritance.

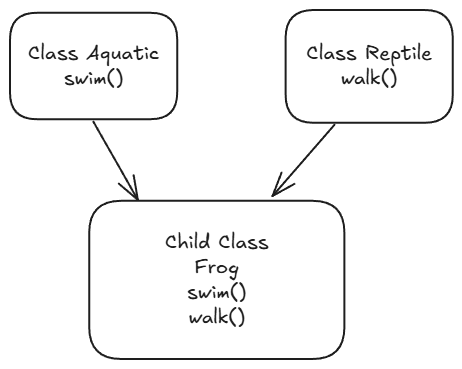
Output



# Multiple Inheritance

Python by-default supports multiple inheritance. When a class is derived from more than one base class is called Multiple Inheritance. The derived class inherits all the features and properties that base class has.

## Example Picture for Multiple Inheritance



In the above picture, Aquatic class has swim function and Reptile class has walk class. The Frog class inherits both Aquatic and Reptile classes, so that Frog can do both swim and walk.

## Example code for Multiple Inheritance

class Aquatic:

    def \_\_init\_\_(self, name: str, fins: int):

        self.name = name

        self.fins = fins

        self.type = "Aquatic"

    def swim(self):

        print(f"{self.name} is swimming under water with {self.fins} fins")

class Reptile:

    def \_\_init\_\_(self, name: str, limbs: int):

        self.name = name

        self.limbs = limbs

    def walk(self):

        print(f"{self.name} is walking on the ground with {self.limbs} legs")

# Frog inherited both aquatic and reptile

class Frog(Aquatic, Reptile):

    def \_\_init\_\_(self):

        # calling Aquatic class constructor

        # calling Reptile class constructor

        # passing name as frog

        Aquatic.\_\_init\_\_(self, "Frog", 2)

        Reptile.\_\_init\_\_(self, "Frog", 4)

    def jump(self):

        print(f"{self.name} is jumping on the ground with {self.limbs} limbs")

frog1 = Frog()

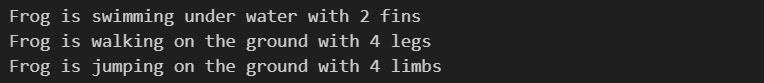
frog1.swim() # this method is from Aquatic class

frog1.walk() # this method is from Reptile class

frog1.jump() # this method is from Frog class

The above code is written according to the picture of multiple inheritance. The Frog class inherits both Aquatic and Reptile class, so that it inherits both swim and walk functions, but for Frog clas it also has jump function.

Output



# Polymorphism

Polymorphism means many forms, The same function but different signatures used for different types. The main difference is the data types and number of arguments used in the function.

## Types of polymorphic function

1. User-Defined: We define our own custom functions according to the requirement.
2. Pre-Defined: Already defined in the interpreter, we’re just using it.

**Example code for User-Defined function**

def Addition(a, b):

    # check if the parameter is int or float

    if (type(a) == float or type(a) == int) and (type(b) == float or type(b) == int):

        return a + b

    # check if the parameter is string and then convert it to float

    elif type(a) == str and type(b) == str and a.isdigit() and b.isdigit():

        return float(a) + float(b)

    # if we don't get required type, returns -1 as an invalid parameter

    else:

        return -1

# user defined functions

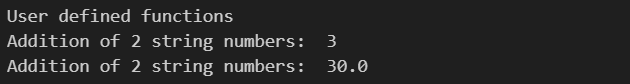
print("\n\nUser defined functions")

print("Addition of 2 string numbers: ", Addition(1, 2))

print("Addition of 2 string numbers: ", Addition("10", "20"))

The Addition function takes 2 arguments as input and returns float or integer depending on the input. It can accept integer, float and string as arguments, It will check what kind of arguments are passed to the function the will make a decision based on the input, and decide which action need to take. If the input parameters are int or float, the 1st if statement will satisfy and return the result. If the input is in string type, then it will convert into float variable and return the result. If it gets any invalid type like lists, bool or dictionary, it directly returns -1 as the output showing error.

**Output**



**Example code for Pre-Defined function**

# pre-defined functions

print("Pre-Defined functions") # takes single argument

print("Hello", 1234, False) # takes multiple argument with different types

# type() function takes many type of arguments

print("Type", type(123))

print("Type", type("abc"))

print("Type", type(True))

There are a lot of Pre-Defined functions defined by inside python interpreter. For ex. Print, type functions take any type as input and print something to the screen. Print function takes multiple arguments as input and those can be any type from int, float, bool, string or even lists and dictionary.

Output



## Kwargs

A special syntax used to pass named arguments to a function, then function will receive them as a dictionary. It is useful where you want to pass arguments to a function like some kind of settings or options.

Syntax

def kwargs\_syntax(\*\*kwargs):

    print(kwargs)

**Example Code**

def multiple\_args(\*\*kwargs):

    print("Type of \*\*kwargs", type(kwargs))

    print(kwargs)

multiple\_args(name="Prajwal", age=24)

**Output**



## Args

An another special syntax same as kwargs but it is passed to function as a list of elements. But like kwargs we don’t need to pass named arguments, but just with comma separated.

**Syntax**

def args\_syntax(\*args):

    print(args)

**Example Code**

def sum\_numbers(\*args):

    sum = 0

    for num in args:

        sum += num

    return sum

print("Sum:", sum\_numbers(1, 2, 3, 4))

**Output**



## Generics

Python doesn't support overloading, but it has support for generics which is like duck typing where an object is expected to have the same property or method (with same signature), even though the classes are different from each other.

**Example Code**

class Dog:

    def make\_sound(self):

        print("Dog is barking")

class Cat:

    def make\_sound(self):

        print("Cat is meowing")

class Wolf:

    def make\_sound(self):

        print("Wolf is howling")

# here all the objects have the same method

objects = [Dog(), Cat(), Wolf()]

for obj in objects:

    obj.make\_sound() # even though the objects are different

Output



## Learning Outcome

In week 5, we covered two main things in Object-Oriented Programming: Inheritance and Polymorphism.

**Inheritance**

Inheritance allows one class (a child) to use and build on the properties and methods of another class (a parent). This makes it easy to reuse code. For example, a Car class can inherit from a Vehicle class, sharing common attributes like colour or speed but adding its own unique features.

Python also supports **multiple inheritance**, allowing a class to inherit from more than one parent. For instance, a Frog class can inherit from both Aquatic (to swim) and Reptile (to walk), letting it do both.

**Polymorphism**

Polymorphism means using the same method name for different types of data. For example, Python’s print() function works with strings, numbers, and even lists. Though Python doesn't have traditional method overloading (like Java or C#), it allows functions to handle different types of inputs, making them versatile.

**\*args and \*\*kwargs**

Python has special ways to handle multiple arguments in functions:

* \*args collects multiple unnamed arguments as a list.
* \*\*kwargs collects named arguments as a dictionary.

These make functions flexible, allowing for a variety of inputs.

**Generics**

Python doesn’t have strict function overloading, but it allows different objects to have the same method name. For example, classes Dog, Cat, and Wolf might each have a make\_sound() method. When called, each will act according to its own class, even if the objects are grouped together in a list.

In short, we learned how to use inheritance for code reuse, polymorphism for flexibility, and special syntax to make functions adaptable.

## Summary

In Week 5, we went through two key ideas in Object-Oriented Programming: **Inheritance** and **Polymorphism**.

* **Inheritance** is like giving one class the ability to "inherit" features from another, so it can reuse and expand on them. For example, if we have a Vehicle class, a Car class can inherit from it to get all the basics while adding its own details. Python also allows a class to inherit from more than one parent, which isn’t common in many other languages.
* **Polymorphism** lets us use methods with the same name in different ways, depending on the input or class type. This helps make our code more adaptable and user-friendly. We explored how Python handles functions flexibly, even without the typical overloading found in languages like Java or C#.

# Bank System – A Mini Project

This code is a small project for the coursework that simulates a basic banking system. It includes classes to handle accounts, deposits, withdrawals, and manage different types of accounts within a bank. The code can be found in the “***bank-project/”*** folder, run the main.py file using the command python main.py. It is a command-line application, user need to select the options provided and the application will instruct the user based on the choice that the user chooses.

## Explanation of the Code:

1. **Account Class**:

* This is a base class representing a general bank account.
* It has attributes like *balance* (the account balance), *account\_number* (unique identifier for the account), *name*, and *phone*.
* The deposit method adds money to the balance, while the withdraw method deducts money. *get\_total\_balance* simply returns the current balance.

1. **Savings Account Class**:

* This class inherits from Account and represents a savings account with an *account\_type* set to ***"savings"***.
* It has an additional attribute, *interest\_rate*, and redefines deposit to add interest on each deposit by multiplying the balance by the interest rate.

1. **Current Account Class**:

* This also inherits from Account and represents a current account with an *account\_type* set to ***"current"***.

1. **Bank Class**:

* This class represents the bank itself, with attributes for the *bank\_name*, *branch*, and a *list* of all accounts.
* *add\_account* adds a new account to the bank and assigns a unique account number.
* *remove\_account* removes an account based on its number.
* *withdraw\_money* and *deposit\_money* allow the bank to perform withdrawals and deposits for specific accounts.
* *list\_accounts* shows all accounts in the bank.
* *get\_balance* and *get\_account* let you check the balance or retrieve details of a specific account by its number.

In this bank account system project, we applied core Object-Oriented Programming (OOP) concepts:

1. **Classes and Objects**:

* Defined classes like Account, SavingsAccount, CurrentAccount, and Bank.
* Created objects to represent individual bank accounts and the bank itself, containing related data and behavior.

1. **Inheritance**:

* SavingsAccount and CurrentAccount inherit from the Account class, allowing them to reuse and extend its functionality.

1. **Polymorphism**:

* The deposit method is overridden in the SavingsAccount class to add interest, which shows polymorphism by changing the behavior of a method in a child class.
* This allows the same method name (deposit) to work differently depending on the object.

These Object-Oriented principles make the code more organized, reusable and easy to understand, providing a backbone for bank system. This project helps us understand OOP Concepts like inheritance and encapsulation.

# Week 8 – Data Structures & Abstract Classes

In week 8, we learnt about the Data Structures such as Lists, Dictionary, etc. and Abstract Classes.

## What is a Data Structure?

A Data Structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

## Types of Data Structures

There are four different types.

### Lists

They are used to store multiple items in a single variable. A collection of values with same or different data types is stored in a variable is called a list.

**Example Code**

fruits = ["Apple", "Grapes", "Banana"]

Here, the ***fruits*** variable is a list of string contains names of fruits. The size of the list is 3, and we can get the size by using ***len(fruits)*** function passing fruits as the argument.

List supports operations such as adding, deleting, sorting, etc.

**Example Code**

fruits = ["Apple", "Grapes", "Banana"]

print(fruits)

fruits.append("Orange")

print(fruits)

fruits.remove("Grapes")

print(fruits)

print("Length of the fruits list is", len(fruits))

**Output**



We also can use loops to iterate each elements one-by-one.

**Example Code**

cars = ["BMW", "AUDI", "MERCEDES", "FIAT"]

print("Normal for loop")

for car in cars:

    print(car)

print("\nFor loop with enumerate()")

for i, car in enumerate(cars):

    print(f"{i}) {car}")

**Output**

****

We can access each element by using their index. In python indexes are 0-based which means everything inside a list starts with 0, which means accessing the 1st element will be fruits[0], the index should be put inside a square brackets after the variable name. In the output we can see that **BMW** starts with **0** index. Python also supports range indexing, which means we can select a set amount of elements by providing from and to index separated by colon ( : ), here to index is exclusive and from index is inclusive.

When we use range operator, it creates a new list and doesn’t modify the existing list, this is called **Immutability**. The same technique is applied for strings.

**Example Code**

# Range index

users = ["Prajwal", "Sreeraj", "Aman", "Ali", "Stephen"]

print("Names from 0 to 2", users[0:2])

# Modifying does not affect the original list

users[:4][0] = "Unknown"

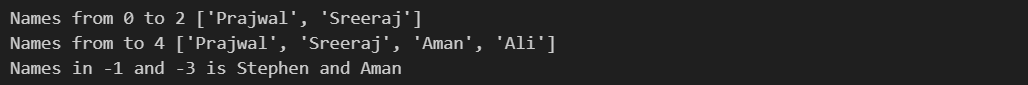
# slicing the list to length of 4 from start

print("Names from to 4", users[:4])

# providing the -1 gives the last element.

print("Names in -1 and -3 is", users[-1], "and", users[-3])

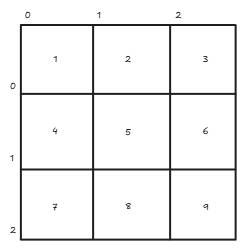
**Output**



### Nested Lists

It is a list but the elements inside it is also a list of elements. It is useful for storing matrix data, or 2-dimensional data such as pixel information of an image. The below image represents a 2D matrix.

**Picture of the Structure of 3x3 size Nested List**



All the operations that are supported for the list are supported to nested list as well, because it is just a list containing a bunch of lists.

**Example Code**

# Nested Lists

matrix = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 9]

]

print("Matrix", matrix)

print("Value at 0th row, 2nd column in Matrix", matrix[0][2])

**Output**



### Dictionary

A dictionary is used to store values in **key:value** pairs. It is a collection of elements, where each value is tagged with a unique key, which can be accessed by using a key. The key and value can be of any type, so there is no specific restriction. To create, we can use **{ }** or **dict()** function. The key value pairs are separated by colon ( **:** ), and after each pair, it is separated by comma ( **,** ) if we want multiple key-value pairs.

**Example Code**

country = {

    # KEY  :  VALUE

    "India": "New Delhi",

    "England": "London",

    "USA": "Washington DC",

    "Germany": "Berlin"

}

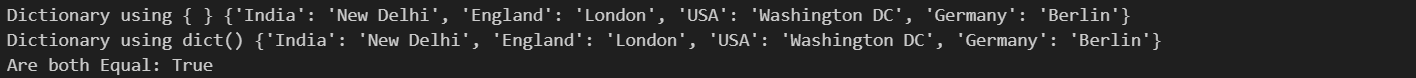
country1 = dict(India = "New Delhi", England = "London", USA = "Washington DC", Germany = "Berlin")

print("Dictionary using { }", country)

print("Dictionary using dict()", country1)

print("Are both Equal:", country == country1)

**Output**

****

Dictionary supports all kinds of operations such as adding, modifying and deleting from the dictionary.

**Example Code**

cars = {

    "BMW": "M5",

    "Mercedes": "AMG",

    "Dodge": "challenger",

}

# Adding to a dictionary

cars["Lamborghini"] = "Aventador SVJ"

# Modify

cars["BMW"] = "M3 GTR"

# Deleting

del cars["Mercedes"]

print("Accessing an element inside dictionary: ", cars["Lamborghini"])

print(cars)

# Getting all the keys

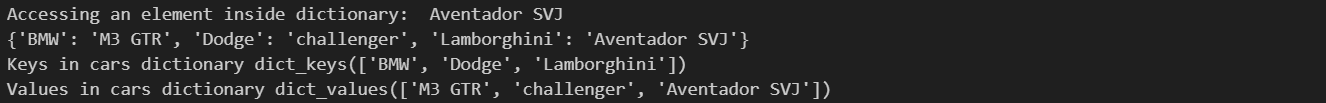
print("Keys in cars dictionary", cars.keys())

# Getting all the values

print("Values in cars dictionary", cars.values())

Syntax for adding or modifying a dictionary is same, but for modifying, the key should be unique or it will create a new element in dictionary. **del** is a keyword used to remove key-value pairs from a dictionary. The syntax is simple, use the del keyword before accessing the element.

**Output**

****

Using Loops with dictionary is possible, same like lists we get key when looping.

**Example Code**

for brand in cars:

    print("Key is", brand, "and value is", cars[brand])

**Output**



### Tuple

A way of storing multiple immutable values in a single variable. Immutability means once a value is created and set to a variable, one it can’t be changed or if we change it’ll create a new value and doesn’t modify the existing ones. Syntax for tuple is very simple, the values should be inside a parenthesis **( )** and separated by comma. Accessing a value inside is similar to list, just access it using their index. Tuple also supports destructuring which means values can be assigned to individual variables.

**Example Code**

toys = ("car", "doll", "scooter", "ball")

print("1st element", toys[0])

# Destructuring

car, doll, scooter, ball = toys

print(car, doll, scooter, ball)

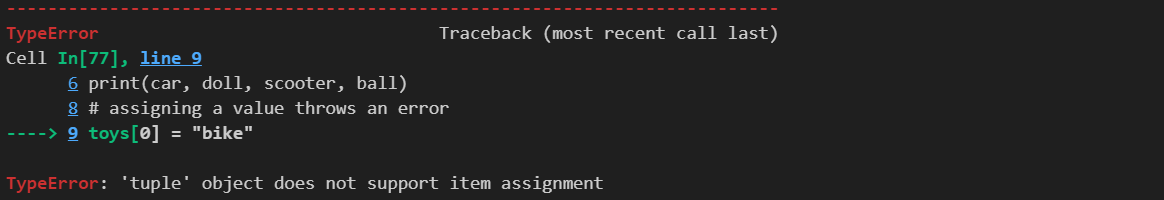
Assigning a value to a tuple before destructuring throws error, but we can assign after destructuring.

**Example Code**

# assigning a value throws an error

toys[0] = "bike"

**Output**



But we can assign after destructuring.

**Example Code**

print("Before Assigning", car)

car = "sedan"

print("After Assigning", car)

**Output**

****

### Set

A set is an unordered collection of unique and immutable elements. It doesn’t allow duplicates and supports operations such as Union, Intersection and Difference etc.

**Example Code**

nums = {1, 2, 3, 4}

print(nums)

# Adding elements

nums.add(5)

print(nums)

**Output**



There are a number of operations can be done to a set.

1. **Union**: Combines elements from 2 different sets and removes duplicates.

# Union operation

print("Union:", nums.union({4, 5, 6, 8, 11}))

****

1. **Intersection**: Finds all the common elements between sets.

# Intersection operation

print("Intersection:", nums.intersection({2, 3}))

****

1. Difference: Finds the difference between 2 different sets. Means removes the elements which are in 2nd set from 1st set.

# Difference operation

print("Difference:", nums.difference({4, 5, 6}))



## Abstract Class

An abstract class is a class that cannot be instantiated, and is used for inheritance purposes. It acts as a template for other classes that inherit and must implement methods which are marked abstract. Abstract classes contains **@abstractmethod** decorator on top of the function definition and should inherit from **ABC** class which both can be imported from **abc** namespace.

**Example Code**

from abc import abstractmethod, ABC

class Animal(ABC):

    @abstractmethod

    def make\_sound(self):

        pass

    def sleep(self):

        print("Animal is sleeping")

class Cat(Animal):

    def make\_sound(self):

        print("Cat is meowing")

class Dog(Animal):

    def make\_sound(self):

        print("Dog is barking")

class Mouse(Animal):

    def make\_sound(self):

        print("Mouse is sqeeking")

spike: Animal = Dog()

spike.make\_sound()

tom: Animal = Cat()

tom.make\_sound()

jerry: Animal = Mouse()

jerry.make\_sound()

**Output**

****

# Portfolio Exercise 5 – PriorityTask

We added a new PriorityTask class to the existing Task types, a new property called priority which contains only “*low”*, “*medium”* or “*high”* values.

**Example Code**

class PriorityTask(Task):

    def \_\_init\_\_(

        self, title: str, date\_due: datetime, priority="low", description: str = ""

    ):

        super().\_\_init\_\_(title, date\_due, description, "PriorityTask")

        self.priority = priority

    def \_\_str\_\_(self):

        status = "Completed" if self.completed else "Not Completed"

        return f"{self.title} - Priority - {self.priority} (created: {self.date\_created.strftime("%Y-%m-%d")}, status: {status})"

When creating the priority task, the system asks user to enter a priority value, if it is invalid we don’t create the task, but shows an error message. If everything goes well, we’ll create the priority task.

elif option1 == "priority":

    new\_task = input("Add your task please: ")

    create\_date = input("Set the date (YYYY-mm-dd) :").strip()

    describe = (input("Write a description for this task (or press Enter to skip): ")

                or "No description provided")

    priority = input("Enter the priority (low,medium,high): ")

    if not (priority in ["low", "medium", "high"]):

        print("Invalid priority. Please enter a valid priority")

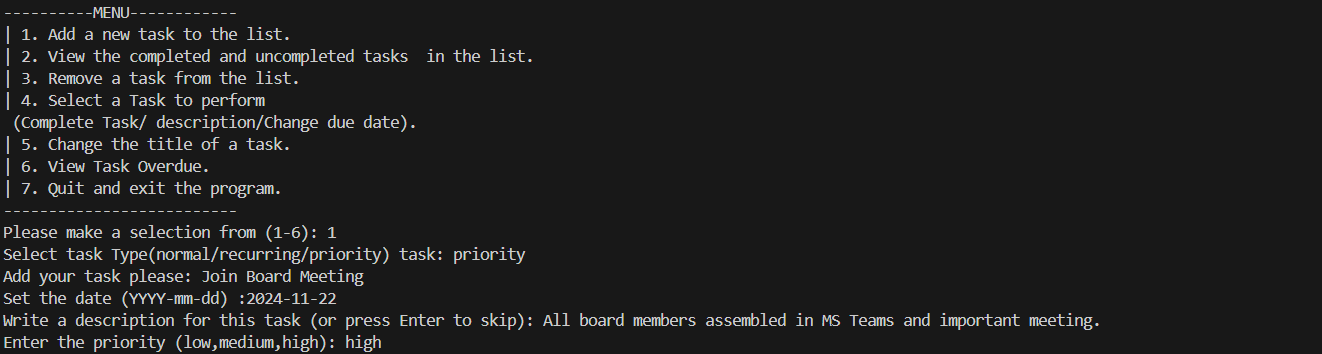
        continue

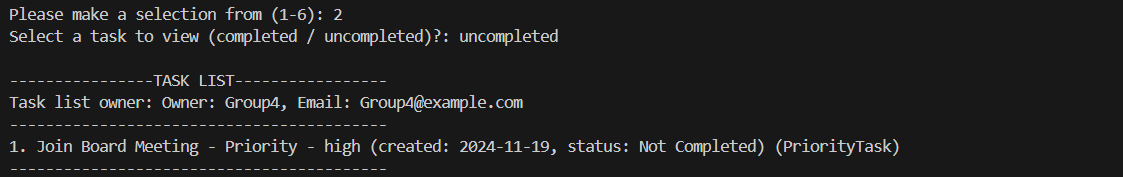
    new\_date = datetime.datetime.strptime(create\_date, "%Y-%m-%d")

    task\_list.title = PriorityTask(new\_task, new\_date, priority, describe)

    task\_list.add\_task(task\_list.title)

**Output**





## Learning Outcome

* Learned how to use different data structures to store and manipulate data efficiently.
* Explored operations on Lists, Dictionaries, Tuples, and Sets, along with their applications.
* Got a clear idea of how Abstract Classes act as templates, making sure child classes include the necessary methods.

## Summary

In Week 8, we explored into some of Python’s most useful features: Data Structures and Abstract Classes. These are the building blocks for storing, organizing, and working with data in efficient ways.

* Lists: Collections where you can add, remove, sort, and slice items easily. Great for working with groups of related data.
* Dictionaries: These store data in key-value pairs, making it easy to look things up or update them.
* Tuples: Useful when you need a fixed, unchangeable collection of items.
* Sets: Perfect for keeping unique items, with operations like finding common elements (intersection) or combining sets (union).
* Abstract Classes: Think of these as templates or blueprints, ensuring any class that inherits them includes the right methods.