School of Computing, Engineering and Physical Sciences

MSc Information Technology

COMP11124 Object Oriented Programming

GROUP COURSEWORK (50% of the marks)

Session Oct-2024/2025 Term 1

Students

1. Prajwalaradhya Shivakumaraswamy Kesaramadu (B01759301)
2. Muhammed Ali Panthalingal (B01755979)
3. Stephen Kwaku Pometsey (B01757368)
4. Aman Misra (B01746656)
5. Sreeraj Karuvanthodi Ramachandran (B01764963)

Lecturer: Michael Lin

Submission Date: 20-11-2024

Declaration

I have carefully read and fully understand Regulations 3.49–3.55 of Chapter 3 of the Regulatory Framework of the University of the West of Scotland, which outline the rules and policies regarding cheating and plagiarism.

I confirm that this assessment is the collective work of our group, consisting of five members, with each member contributing to the completion of this task. Where applicable, we have clearly referenced and acknowledged the contributions or ideas of others outside our group.

I also affirm that no part of this assessment has been written, in whole or in part, by anyone outside the group, except for explicitly referenced sources.

Furthermore, I confirm that this assessment has not been submitted previously, either partially or fully, for any other module or academic purpose, ensuring it does not fall under self-plagiarism.

This declaration is made with honesty and integrity in adherence to the University's academic standards.

Table of Contents

Week 2 - Comparison Operators

In python we can compare two ore more values, the result is always either TRUE or FALSE.

```
isGt = 5 > 4
```

Here, 5 is biggest number than 4, in these situations we use greater than symbol (>) to indicate it. So, here the output is True, because 5 is greater than 4 is satisfying or technically it is correct.

In python we use another symbols too, i.e.

- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or equal to)
- == (equal to)
- != (not equal to)

Below I mentioned some examples i.e.

```
isGte = 5 >= 5 or 100 >= 20 # true
isEquals = 5 == 5 # true
isNotEquals = 5 != 4 # true
isLt = 22 < 222 # true
isLte = 555 <= 678 # true
```

here all the conditions are satisfying so it became TRUE, otherwise it will be FALSE.

Logical Operators

Logical operators in python helps to do comparisons, conditional statements, and standard algorithms.

In python contains three logical operators:

1. AND – In and operator, if the both operands are True then the outcome will be True, otherwise it will be False.
2. OR – In or operator, at least one operand is True then the outcome will be True, otherwise it will be False.
3. NOT – In not operator, if the operand is True then the outcome will be False, otherwise it will be True.

If – conditionals

It is used to check a condition whether it is True.

For example:

```
age = 19
isAdult = age >= 18 and age <= 70
if isAdult:
```

```
   print("I'm adult")
else:
   print("I'm not adult")
```

Here, the condition is, if the value is greater than or equato 18 and it is less than or equal to 70 then the condition will be satisfy and we will get the condition satisfying outcome, Otherwise we can get the other outcome.

Here the outcome is

```
I'm adult
```

If – else conditionals

This condition used to execute both True and False part of given condition. If the condition is Tue then If part is executed and the condition is False then else part is executed.

Example

```
wind_speed = 8

if wind_speed < 10:
    print("It is a calm day")
else:
    print("It's a windy day")
```

Here the wind speed is 8, so the given condition is not satisfying with if condition so the else part is executed.

Output

```
It is a calm day
```

If – elif – else conditionals

In some cases if – else conditions will not satisfy with given condition to solve this there is another condition it is elif.

Code example

```
grade = 44

if grade < 50:
    print("You failed")
elif grade < 60:
    print("You passed just above the passing marks")
elif grade < 70:
    print("Your marks are good")
else:
    print("Your marks are excellent. You done well in exams.")
```

Here at the first condition is given, for execute this it is necessary to go throw more than two conditions. In these situations we use elif

Output

```
You failed
```

Here, it will go through all conditions untill the given condition is satisfy.

Task Summary

Here we have to create two variables, **temperature1** and **temperature2**, and assign different values to them. Use an **if** statement to check if the temperatures are equal and print a corresponding message. Use an **else** statement to print a message if the temperatures are not equal. Run the code and see if your statement has been built correctly.

```
temp1= 10
temp2 = 20

if temp1 > temp2:
    print("temp 1 is hotter than temp2")
else:
    print("temp2 is hotter than temp1")

print("\nTernary operator")
print("temp 1 is hotter than temp2") if temp1 > temp2 else print("temp2 is
hotter than temp2")
```

Here we assign and stored two values as temp1 and temp2 so we can see that 20 is biggest value than10. Here temp1 assigned as 10 and temp2 is assigned as 20 so if the temp1 is greater than temp2 then output should be like temp 1 is hotter than temp2 otherwise the output is temp2 is hotter than temp1. Here we can see that temp2 si bigger so the output is given below

```
temp2 is hotter than temp1
```

Python lists

In Python, a list is a mutable or changeable ordered sequence of elements. The elements or values contained within a list are known as items. Lists List, just like strings are specification of characters you put between quotes, is a way of working with data lined up in different square brackets [ ].

Create a list

Creating list is same as creating variable and assigning values in it.

Code Example

```
cities = ["London", "Glasgow", "Edinburgh", "Manchester", "Liverpool",
"Birmingham"]
```

```
print(cities)
```

Output

```
['London', 'Glasgow', 'Edinburgh', 'Manchester', 'Liverpool', 'Birmingham']
```

Accessing a list

We can access specific items within a list or the whole list too according to need it is called accessing list

Code Example

```
print(cities[3])
```

Output

```
Manchester
```

Modifying list

we can modify list by assigning new values, it is given below

```
string_list[0] = "pear"
```

Code Example

```
cities[2] = "Bangalore"
```

Output

```
Modified 2nd idx:
```

In addition to this, we can add items to list

Code Example

```
cities.append("Tumkur")
```

Output

```
Added one more city
```

Task Summary

Write Python code to create a list named **colours** containing the names of three colours as strings. Print the entire list.Access the second element of the **colours** list and print it. Modify the first element of the list to a new colour of your choice. Print the modified list. Check and print the length of the **colours** list using the **len()** method. This is similar to using the similar to using the **type()** method from before. Use a conditional

to check if "red" is in the **colours** list. If yes, print that “Red is in the list”. Use slicing to create a new list named **selected_colours** containing the second and third elements from the **colours** list. Print the **selected_colours** list.

Example Code

```
colours = ["orange", "white", "green"]
print(colours)
colours.append("blue")
print("2nd element: " + colours[1])
colours[0] = "red"
print(colours)
print(f"Length of the colours list is {len(colours)}")

print("Red is in the list") if "red" in colours else print("Red is not in the
list")

selectedColours = colours[:2]
print("Selected colours: " + str(selectedColours))
```

Here created a list with 3 colors and added another color blue(modified the list). Accessed the length of list and selected 2 colors in the list.

Output

```
['orange', 'white', 'green']
2nd element: white
['red', 'white', 'green', 'blue']
Length of the colours list is 4
Red is in the list
Selected colours: ['red', 'white']
```

Python loops

Python loop means it is repeating again and again until the given condition is satisfied

There are different types of loop in python.

While loop

This is a type of loop, it continuously execute code util the given condition is True.

Example code

```
 print("While Loops")

i = 0
while i < 5:
    print(i)
    i += 1
```

Here the code is for count 5 numbers

Output

```
While Loops
0
1
2
3
4
```

For loop

For loop is iterate over a sequence(like a list, string or range) and execute a set of code several times.

Example code

```
print("For Loops")
for city in ["Delhi", "London", "Bangalore", "Tumkur"]:
    print(city)

print("For Loops with Range")
for i in range(1, 5):
    print(i)
print("Breaks")
for i in range(1, 5):
    if i == 3:
        break
```

Here we listed some cities using For loop with number but after 4 we use break for stop it otherwise it will continuously run. So if we want to stop before execute the code we can use Break.

Output

```
For Loops
Delhi
London
Bangalore
Tumkur
For Loops with Range
1
2
3
4
Breaks
```

Task Summary

Create a list called **numbers** which contains the integers from 1 to 10.

Use a **for** loop to iterate through the list and only print the *even* numbers. (Hint: use the modulo % operator)

Sum of Squares

Create a variable **sum_of_squares** and initialize it to 0.
Use a **for** loop to iterate through the numbers from 1 to 5 (inclusive) using the **range()** function. Add the square of each number to **sum_of_squares.** Print the final value of **sum_of_squares**. (Hint: if you do it correctly, the result should be 55)

Countdown:

Create a variable **countdown** and initialize it to 10. Use a **while** loop to print a countdown from the value of **countdown** to 1. After the countdown, print "Liftoff!"

Example Code

```
print("Even numbers")
for i in range(1, 9):
    if i % 2 == 0:
        print(f"{i} is even")

sumOfSquares = 0
for i in range(1, 6):
    sumOfSquares += (i * i)
print(f"Sum of squares {sumOfSquares}")

countdown = 10
while countdown > 0:
    countdown -= 1
print("Liftoff")
```

Output

```
For Loops with Range
1
2
3
4
Breaks
1
2
Even numbers
2 is even
4 is even
6 is even
8 is even
Sum of squares 55
Lift off
```

Here, the first part is for printing even numbers between 1 to 8,i is indicates numbers, if the i is divisible by 2 then the condition is satisfy hence the even number will print. Then the second part is for print the sum of the squares of numbers from 1to 5, i is number i*I is for calculating

sum of squares of number(i). Then the coming part is for using countdown from 10 to 1. If the while countdown > 0 then it runs as the value of countdown is greater than 0.once the countdown reach 0 then then the loop will stop and push a message Lift off.

Task Temperature converter

The user should be able to enter a value in degrees Celsius and your converter should convert this to Fahrenheit and Kelvin.
As an extra task, (if you are finished before the class is over or want to practice a bit more Python) you should allow the user of the program to enter what temperature they want to convert (C, K or F) and then print out the conversions. Use conditionals for this.

Example Code

```
temp = input("Please enter the temperature in celcius to convert to kelvin and
farenheit")
if not temp.isdecimal():
    print("Please enter a valid number")
else:
    temp = int(temp)
    fh = (temp * 1.8) + 32
    kv = temp + 273.15
    print(f"Temperature: {temp} celcius")
    print(f"Temperature: {fh} farenheit")
    print(f"Temperature: {kv} kelvin")
```

Here, The program asks the user to input a temperature in Celsius. It verifies if the input is a valid number. If the input is valid, it converts the temperature to Fahrenheit and Kelvin and shows the results. If the input is invalid, it prompts the user to enter a valid number.

Output

```
Temperature: 30 celcius
Temperature: 86.0 farenheit
Temperature: 303.15 kelvin
```

Summary

In week 2 we went through some python concepts. They are

- Comparison operators: It is used to check the two or more values, the sub divisions are AND, OR and NOT operators. They are used according to different condition.
- Logical operators: it is used to combine multiple conditions for more complex tasks.
- Conditional statements: these statements are used to control the flow of program based on their conditions. If, elif, else are the types of conditional statements.
- Lists: it is used for storing ordered collections of items. Lists can be modify and access.
- Loops: it is used to automate repeating tasks and it process productively.

Week 3 - Python Functions, Scope and Errors

In week 3 exercise, we learnt about create and use functions and variable scope as well as error fixing in python.

Functions:

Python functions are reusable blocks of code that perform a specific task. They are defined with the def keyword, can take inputs (arguments), and may return outputs.

Scope:

Python scope determines the visibility and lifespan of variables within different parts of the code. Variables can exist in local, enclosing, global, or built-in scopes, impacting where they can be accessed or modified.

Error:

Python errors are issues that arise when code encounters invalid operations, leading to exceptions or syntax errors.

Example code for Functions:

```
#Greet Friends

def greet_user(first_name , last_name , university = "UWS" ):
      print(f"Hello {first_name} {last_name} from  {university}")

greet_user("Aman" , "Misra" , "UWS London")
```

Output

```
Hello Aman Misra from  UWS London
```

```
def greet_friends(friends):
      for name in friends:
          print(f"Hello {name}")

friends_list = ["Aman" , "Sreeraj" , "Prajwal"]
greet_friends(friends_list)
```

Output

```
Hello Aman
Hello Sreeraj
Hello Prajwal
```

The **greet_user** function is a simple yet effective way to generate customizable greeting messages. Its use of default parameters and string formatting makes it versatile and user-friendly.

def greet_user(first_name, last_name, university="UWS"): Declares a function with three parameters, where university has a default value of "UWS".

```
def add(num1 , num2):
      return num1 + num2

def multiply(num1 , num2):
      return num1 * num2
```

```
print(add(3 , 5))
print(multiply(1 , 5))
```

Here , add: A function to perform addition of two numbers ,

multiply: A function to perform multiplication of two numbers.

Both functions are designed to return the results of their operations. These results can be used immediately or stored for later use.

In Python, the **return** keyword is used within a function to send a value back to the calling context. Here's how the **return** keyword is used in the given code.

```
# Tax Calculation

def calculate_tax(income , tax_rate):
      tax_amount = income * tax_rate
      return tax_amount

total_money = calculate_tax(50000 , 0.2)
print(total_money)
```

In this code ,the function **calculate_tax** takes two parameters: **income** and **tax_rate**.

tax_rate: Represents the tax rate as a decimal (e.g., 20% is 0.2)

income: Represents the amount of money on which tax is calculated.

The result of the calculation is sent back to the calling context using the **return** statement.

The returned value **(tax_amount)** is stored in the variable **total_money**.

The **calculate_tax** program demonstrates effective use of functions, return values, and arithmetic operations.

Variable Scope

```
def new_function():

      my_new_variable = 5

new_function() # call the function. No problems here.

print(my_new_variable) # this will cause an error
```

Here,the provided Python code defines a function **new_function** that declares a variable **my_new_variable** and assigns it a value of 5. After calling the function, there is an attempt to print the value of **my_new_variable.**

Variables in Python have a specific scope (the context in which they are accessible)

Variables defined inside a function (local variables) are accessible only within the function.

Function Execution

When the function **new_function** is called, the variable **my_new_variable** is created in the local scope of the function and is discarded once the function exits.

Error Explanation

The attempt to access **my_new_variable** outside the function results in a **NameError**, as the variable is not defined in the global scope.

```
my_new_variable = 0

def new_function():
      my_new_variable = 5

new_function()

print(my_new_variable)
```

The provided Python code defines a variable **my_new_variable** in the global scope, assigns it an initial value of 0, and then defines a function **new_function** that declares a new variable with the same name **(my_new_variable)** and assigns it a value of 5. The function is called, and the program attempts to print the value of **my_new_variable**.

Code Execution Analysis

1. **my_new_variable = 0:**
    - A **global variable** named my_new_variable is initialized with the value 0.
2. **def new_function(): my_new_variable = 5:**
    - Inside the function new_function, a **local variable my_new_variable** is created and assigned the value 5.
    - This variable is local to the function and does not affect the global **my_new_variable**.
3. **new_function():**
    - When the function is called, the local **my_new_variable** is assigned 5 but is discarded as soon as the function ends.
4. **print(my_new_variable):**
    - The global variable **my_new_variable** remains unchanged and retains its original value of 0.

Optional: Assertions and Errors

```
# Error fixing

# pritn("Hello, World!") # error in pritn
print("Hello, World!")
```

```
# number1 = "5"
number1 = 5
number2 = 3
result = number1 + number2

print(f"The sum is {result}")

fruits = ["apple", "banana", "cherry"]
# print(fruits[3]) -> index out of bounds
print(fruits[2])

time = 11
if time < 12:
#print("Good morning!")  # no indentation/tab space
      print("Good morning!")
```

Code Overview

The provided code initially contains several errors, both syntactical and logical. These errors have been fixed to ensure the code runs without issues. Here's the corrected version.

Explanation of the Fixes

1. **Error in pritn("Hello, World!")**:
    - Issue: The function name was misspelled as pritn instead of print.
    - Fix: Changed pritn to print.
2. **Type Mismatch in Addition**:
    - Issue: number1 was a string ("5"), and adding it to an integer (number2) caused a TypeError.
    - Fix: Changed number1 to an integer (5) to ensure compatibility during addition.
3. **Index Out of Bounds in print(fruits[3])**:
    - Issue: The fruits list has only three elements (index 0, 1, and 2), but the code attempted to access index 3, leading to an IndexError.
    - Fix: Changed the index to 2, which is valid and retrieves the last element of the list.
4. **Indentation Error in if Statement**:
    - Issue: The print statement under the if condition was not indented correctly, causing a SyntaxError.
    - Fix: Indented the print("Good morning!") statement properly under the if block.

In this code, the **print()** function is used to display the message **"Hello, World!"** on the screen.

Two variables, **number1** and **number2**, are assigned the values **5** and **3** respectively.

These numbers are added together, and the result **(8)** is stored in the variable result.

The **print()** function is used again to display the message "The sum is 8", where **{result}** is replaced by the value stored in result **(which is 8).**

The fruits list contains three items: **"apple", "banana",** and "**cherry".**

The **print()** function is used to display the third item in the list **(fruits[2]),** which is **"cherry".**

The variable time is set to **11.**

An if statement checks if time is less than **12**. Since **11** is less than **12**, it prints **"Good morning!".**

Summary

Week 3 introduced essential programming concepts like functions, variable scope, and error handling. These skills form the foundation for writing clean, organized, and reliable Python code, setting the stage for tackling more advanced challenges.

Week 4 - Classes and Objects

In Python, classes and objects are fundamental concepts used to implement object-oriented programming. A class serves as a blueprint for creating objects, which are instances of that class. Here's a breakdown of each concept with examples.

4.1 Class:

A class in Python is a blueprint that defines attributes (data) and methods (functions) that characterize a specific type of object. Think of a class as a template for something you want to create. For example, if you want to represent a "Dog," you'd create a `Dog` class with attributes like `name` and `breed`, and methods like `bark()`.

4.2 Object:

An object is an instance of a class. When you create an object, you are creating an individual instance of the class with unique attribute values. Each object is distinct but follows the blueprint defined by its class.

Example code for a Dog class and objects

```
class Dog:
    # Constructor method to initialize attributes
    def __init__(self, name, breed):
        self.name = name  # Attribute: name of the dog
        self.breed = breed  # Attribute: breed of the dog

    # Method to make the dog bark
    def bark(self):
        return f"{self.name} says Woof"


# Creating Objects:
# Creating two Dog objects with different names and breeds
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Milo", "Beagle")

# Accessing attributes and methods of each object
print(dog1.name)
print(dog2.breed)
print(dog1.bark())
print(dog2.bark())


class Dog:
    # Constructor method to initialize attributes
    def __init__(self, name, breed):
        self.name = name  # Attribute: name of the dog
        self.breed = breed  # Attribute: breed of the dog

    # Method to make the dog bark
    def bark(self):
        return f"{self.name} says Woof"


# Creating Objects:
# Creating two Dog objects with different names and breeds
dog1 = Dog("Buddy", "Golden Retriever")
```

```
dog2 = Dog("Milo", "Beagle")

# Accessing attributes and methods of each object
print(dog1.name)
print(dog2.breed)
print(dog1.bark())
print(dog2.bark())
```

Output

```
Buddy
Beagle
Buddy says Woof!
Milo says Woof!
```

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.

Explanation:

- `Dog` is a class that defines the attributes `name` and `breed` and has a method `bark()`.

- `dog1` and `dog2` are objects (instances) of the `Dog` class.

  - `dog1` has `name = "Buddy"` and `breed = "Golden Retriever"`.

  - `dog2` has `name = "Milo"` and `breed = "Beagle"`.

- Both `dog1` and `dog2` use the `bark()` method, but their output differs based on their `name` attribute

Example code for Multiple Inheritance

Here is an example of a Python program that implements multiple inheritance using animal-related classes. This program includes a base class called Animal, derived classes such as Flying, Swimming, and Walking, and a hybrid class named Penguin that combines multiple behaviours

Another Example: Class and Object in Practice

```
class Animal:
    """
    The base class for all animals.

    Attributes:
        name (str): Name of the animal.
    """

    def __init__(self, name: str):
        self.name = name

    def breathe(self) -> str:
        """Indicates that the animal is breathing."""
        return f"{self.name} is breathing."
```

```
class Flying:
    """
    A mixin class representing flying capability.
    """

    def fly(self) -> str:
        """Indicates that the animal can fly."""
        return f"{self.name} is flying high in the sky!"


class Swimming:
    """
    A mixin class representing swimming capability.
    """

    def swim(self) -> str:
        """Indicates that the animal can swim."""
        return f"{self.name} is swimming in the water!"


class Walking:
    """
    A mixin class representing walking capability.
    """

    def walk(self) -> str:
        """Indicates that the animal can walk."""
        return f"{self.name} is walking on land!"


class Penguin(Animal, Swimming, Walking):
    """
    A specific animal class for penguins, inheriting from Animal, Swimming,
and Walking.
    """

    def __init__(self, name: str):
        super().__init__(name)

    def who_am_i(self) -> str:
        """Returns a description of the penguin."""
        return f"I am {self.name}, a penguin! I can swim and walk but
cannot fly."


# Example Usage
if __name__ == "__main__":
    peng = Penguin("Peng")
    print(peng.breathe())
    print(peng.swim())
    print(peng.walk())
    print(peng.who_am_i())
```

Output

```
Peng is breathing.
Peng is swimming in the water!
Peng is walking on land!
I am Peng, a penguin! I can swim and walk but cannot fly.
```

4.3 Classes Overview

1. **Animal**:
    - Base class for all animals.
    - Contains the breathe method to indicate the animal's breathing behavior.
    - Attributes:
        - name (str): The name of the animal.
2. **Flying**:
    - Mixin class representing the ability to fly.
    - Contains the fly method to indicate flying behavior.
    - Assumes the name attribute exists (provided by another parent class).
3. **Swimming**:
    - Mixin class representing the ability to swim.
    - Contains the swim method to indicate swimming behavior.
    - Assumes the name attribute exists (provided by another parent class).
4. **Walking**:
    - Mixin class representing the ability to walk.
    - Contains the walk method to indicate walking behavior.
    - Assumes the name attribute exists (provided by another parent class).
5. **Penguin**:
    - Combines functionality from Animal, Swimming, and Walking.
    - Contains the who_am_i method to describe the penguin's unique features.
    - Overrides the __init__ method to initialize the animal's name.

4.4 Summary

Key Features of Object-Oriented Programming

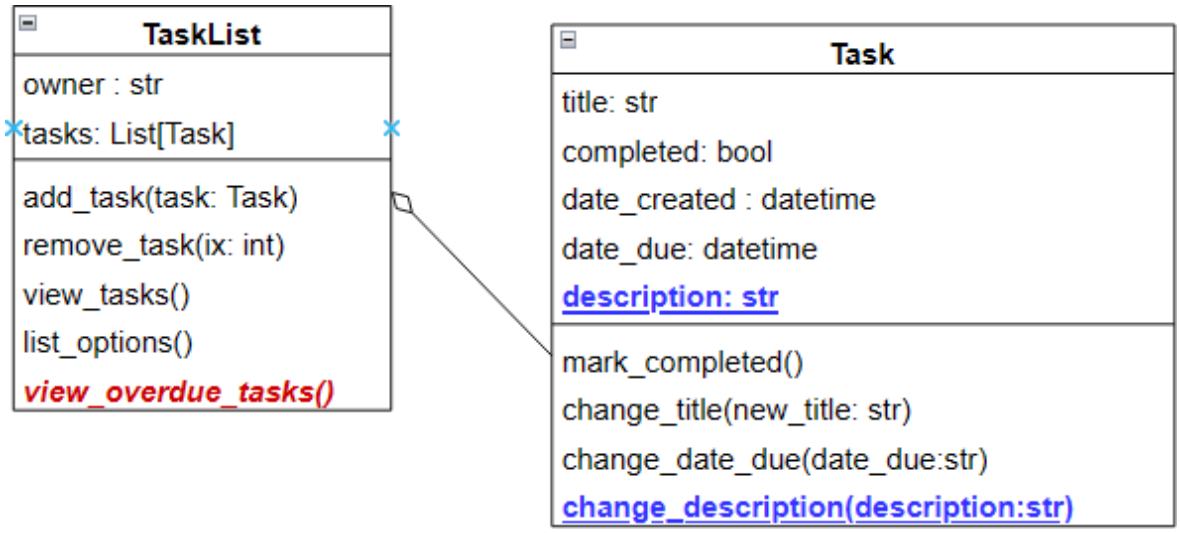
1. Encapsulation: Combining data and methods within a class to protect the integrity of the data.

2. Abstraction: Concealing complex implementation details from the user to simplify interaction.

3. Inheritance: Allowing a class to inherit properties and behaviors from another class, promoting code reuse.

4. Polymorphism: Enabling methods to behave differently based on the object that calls them.

4.5 Portfolio 1 and 2 Solution



Main()

view_options(): This function displays a menu of options for interacting with the task management program.

propagate_task_list(task_list: TaskList) -> TaskList: This function populates the TaskList with a set of sample tasks for testing or demonstration purposes.

main(): This is the primary function that runs the task management application. It initializes a TaskList, populates it with sample tasks, and presents the user with options to manage tasks interactively.

Docstrings: Each function has a docstring that describes its purpose, parameters, and return types, providing a clear guide for understanding and modifying the code.

Type Hints: Type hints like -> None and -> TaskList specify expected return types for each function, improving readability and helping with debugging.

Task

Explanation of Methods

- __init__: Initializes a Task object with a title, due date, and description. The date_created attribute is set to the current date and time when the task is created, and completed is initially set to False.
- change_description: Allows modification of the task's description.
- mark_completed: Marks the task as completed by setting the completed attribute to True.
- change_title: Allows changing the title of the task.
- __str__: Provides a formatted string representation of the task's key attributes, including title, creation date, due date, completion status, and description.

Tasklist

Explanation of Methods

- __init__: Initializes the TaskList for a specified owner and sets up an empty tasks list to hold individual tasks.
- add_task: Adds a Task object to the tasks list.
- remove_task: Removes a task at the specified index x. If the index is invalid (either too low or too high), an error message is printed.
- view_tasks: Displays the list of tasks along with their indexes. Each task is printed in the format defined by the __str__ method of the Task class, which helps provide detailed task information.

Below is the screenshot of implementing description, change_description() and view_overdue_tasks() methods in the portrfolio project.

```
import datetime


class Task:
    #

    def __init__(self, title: str, date_due: datetime.datetime,
description: str):
        self.title = title
        self.completed = False
        self.date_due = date_due
        self.date_created = datetime.datetime.now()
        self.description = description

    def change_date_due(self, date_due: str):
        self.date_due = date_due

    def change_description(self, change_description: str):
        self.description = change_description

    def mark_completed(self):

        self.completed = True

    def change_title(self, new_title: str):
        self.title = new_title

    def __str__(self):

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

        return (f"{self.title} (created: {self.date_created}, due:
{self.date_due}, completed: {self.completed},"
                f"description:{self.description}")
```

Overdue task

tasklist.py

```
from task import Task
import datetime


class TaskList:

    def __init__(self, owner):

        self.title = None
        self.owner = owner
        # self.tasks = []
        self.tasks = []

    def add_task(self, task: Task):
        self.tasks.append(task)
        # self.tasks.append(date_created)

    def remove_task(self, x: int):
        if 0 <= x <= len(self.tasks):
            del self.tasks[x]
        else:
            print("Invalid index: Try again")

    def view_tasks(self):
        for i, items in enumerate(self.tasks):
            print(i, items)

    def view_overdue_tasks(self):
        new_date = datetime.datetime.now()
        overdue_task = [task for task in self.tasks if task.date_due <
new_date and not task.completed]
        if overdue_task:
            print("This tasks are Over-due")
            for i, items in enumerate(overdue_task):
                print(i, items)
        else:
            print("No over-due task available")
```

implementation in main.py

```
elif select_task == "Change description":
    try:
        a = int(input("Which of the above tasks description do you want to
change:"))
        description = input("Type the new description:")
        # change_description = description
        if 0 <= a < len(task_list.tasks):
            task_list.tasks[a].change_description(description)
        else:
            print("Index out of range")
    except ValueError:
        print("Invalid input. Please enter a number.")
```

```
elif choice == 6:
    task_list.view_overdue_tasks()
```

Output

```
C:\Users\FUJITSU\PycharmProjects\CompletePortfolio\venv\Scripts\python.exe
"D:\UWS Course materials\OOP\Exercise\Lab4_Portfolio2\main.py"
```

```
1. Add a new task to the list.
2. View the current tasks in the list.
3. Remove a task from the list.
4. Mark a task as completed or
 Change due Date or Change description).
5. Change the title of a task.
6. Task Overdue.
7. Quit and exit the program.
Please make a selection from (1-6): 4
0 Buy groceries (created: 2024-11-20 15:40:24.187931, due: 2024-11-16
15:40:24.187931, completed: False,description:Buy groceries
1 Do laundry (created: 2024-11-20 15:40:24.187931, due: 2024-11-22
15:40:24.187931, completed: False,description:Do laundry
2 Clean room (created: 2024-11-20 15:40:24.187931, due: 2024-11-19
15:40:24.187931, completed: False,description:Clean room
3 Do homework (created: 2024-11-20 15:40:24.187931, due: 2024-11-23
15:40:24.187931, completed: False,description:Do homework
4 Walk dog (created: 2024-11-20 15:40:24.187931, due: 2024-11-25
15:40:24.187931, completed: False,description:Walk dog
5 Do dishes (created: 2024-11-20 15:40:24.187931, due: 2024-11-26
15:40:24.187931, completed: False,description:Do dishes
Select a task to perform (Complete a task/ Change due Date/Change
description)?: Change description
Which of the above tasks description do you want to change:1
Type the new description:Do Laundry every saturday
1. Add a new task to the list.
2. View the current tasks in the list.
3. Remove a task from the list.
4. Mark a task as completed or
 Change due Date or Change description).
5. Change the title of a task.
6. Task Overdue.
7. Quit and exit the program.
Please make a selection from (1-6): 2
0 Buy groceries (created: 2024-11-20 15:40:24.187931, due: 2024-11-16
15:40:24.187931, completed: False,description:Buy groceries
1 Do laundry (created: 2024-11-20 15:40:24.187931, due: 2024-11-22
15:40:24.187931, completed: False,description:Do Laundry every saturday
2 Clean room (created: 2024-11-20 15:40:24.187931, due: 2024-11-19
15:40:24.187931, completed: False,description:Clean room
3 Do homework (created: 2024-11-20 15:40:24.187931, due: 2024-11-23
15:40:24.187931, completed: False,description:Do homework
4 Walk dog (created: 2024-11-20 15:40:24.187931, due: 2024-11-25
15:40:24.187931, completed: False,description:Walk dog
5 Do dishes (created: 2024-11-20 15:40:24.187931, due: 2024-11-26
15:40:24.187931, completed: False,description:Do dishes
1. Add a new task to the list.
2. View the current tasks in the list.
3. Remove a task from the list.
4. Mark a task as completed or
 Change due Date or Change description).
5. Change the title of a task.
6. Task Overdue.
7. Quit and exit the program.
Please make a selection from (1-6): 6
This tasks are Over-due
0 Buy groceries (created: 2024-11-20 15:40:24.187931, due: 2024-11-16
15:40:24.187931, completed: False,description:Buy groceries
1 Clean room (created: 2024-11-20 15:40:24.187931, due: 2024-11-19
15:40:24.187931, completed: False,description:Clean room
1. Add a new task to the list.
```

```
2. View the current tasks in the list.
3. Remove a task from the list.
4. Mark a task as completed or
 Change due Date or Change description).
5. Change the title of a task.
6. Task Overdue.
7. Quit and exit the program.
Please make a selection from (1-6):
```

In the output above, the description for the first task “Do laundry was Do laundry”
After option 4 was selected the system asked about the task to be performed. “Change description was selected from the options” and the system asked again which task’s description to be changed.
Task 1 do laundry was selected and the descripton was changed from “Do laundry” to ” Do Laundry every saturday ”
Task option 6 also retrieved all overdue tasks.

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.

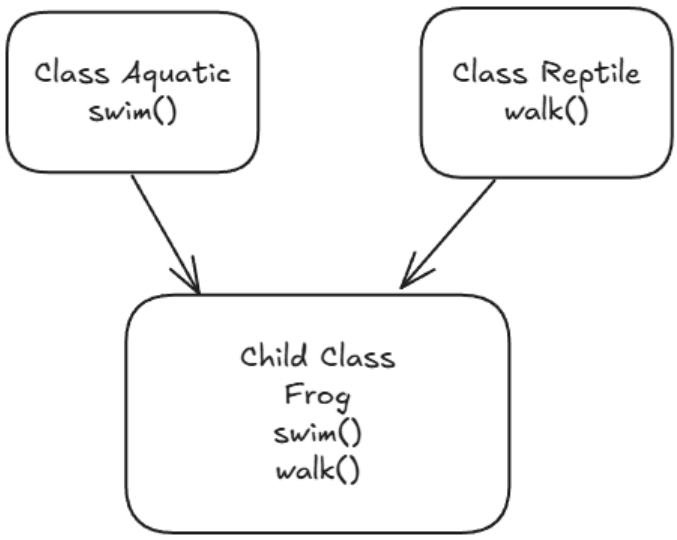
Output

```
The vehicle is moving at 80 km/h
The electric car is moving at 95km/h
The car is driving at 220 km/h
Total fuel capacity 45 ltrs
Total seats of petrol car is 4
```

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

```
Frog is swimming under water with 2 fins
Frog is walking on the ground with 4 legs
Frog is jumping on the ground with 4 limbs
```

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

```
User defined functions
Addition of 2 string numbers:  3
Addition of 2 string numbers:  30.0
```

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

```
Pre-Defined functions
Hello 1234 False
Type <class 'int'>
Type <class 'str'>
Type <class 'bool'>
```

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

```
Type of **kwargs <class 'dict'>
{'name': 'Prajwal', 'age': 24}
```

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

```
Sum: 10
```

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

```
Dog is barking
Cat is meowing
Wolf is howling
```

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.
2. **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.
3. **Current Account Class**:
   - This also inherits from Account and represents a current account with an *account_type* set to ***"current"***.
4. **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.
2. **Inheritance**:

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

3. **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 6 – Programming paradigms

Programming paradigms define various styles of programming used to solve problems. Below is an explanation of Procedural Programming, Functional Programming, and Object-Oriented Programming (OOP).

Procedural Programming

Overview

Procedural programming is a programming paradigm that revolves around procedures, also known as functions or routines. This approach divides the program into smaller, manageable parts (functions) that can be called as needed. It emphasizes a step-by-step method for performing tasks. Typically, data and functions are treated separately.

Characteristics

- Follows a top-down approach.
- Simple to implement for small programs.
- Functions can be reused, which helps reduce code duplication.
- Data is not bound to functions, meaning it is not encapsulated.

Example: Procedural Approach

```
def cal_efficiency(distance: float, fuel_used: float) -> float:
    """Calculate the fuel efficiency of a vehicle (km per liter)."""
    return distance / fuel_used


def show_vehicle_efficiency(vehicle: str, efficient: float):
    """Display vehicle fuel efficiency."""
    print(f"The fuel efficiency of {vehicle} is {efficient:.2f} km/l.")


# Main program
if __name__ == "__main__":
    vehicle_name = "Toyota Corolla"
    distance_traveled = 500  # in kilometers
    fuel_consumed = 25  # in liters
```

```
efficiency = cal_efficiency(distance_traveled, fuel_consumed)
show_vehicle_efficiency(vehicle_name, efficiency)
```

Output

```
"D:\UWS Course materials\OOP\Exercise\Fixed_deposite_system\Procedural.py"

The fuel efficiency of Toyota Corolla is 20.00 km/l.

Process finished with exit code 0
```

Information to Note

The program divides concerns into smaller functions. Data, including the vehicle name, distance, and fuel, is explicitly passed to these functions.

Advantages of Procedural Programming

- Simple to learn and implement.
- Code is easier to debug.

Disadvantages

- Difficult to manage for larger programs.
- Lacks encapsulation (no data hiding).
- Prone to errors when modifying shared data.

Functional Programming

Overview

Functional programming emphasizes functions as the fundamental building blocks of software development. It relies on immutable data and pure functions—functions that do not have side effects. This programming paradigm utilizes concepts such as higher-order functions, along with operations like map, filter, reduce, and recursion. It encourages a declarative style, where the focus is on defining what needs to be done rather than detailing how to perform the tasks.

Characteristics

In functional programming, functions are treated as first-class citizens, meaning they can be passed as arguments, returned from other functions, and assigned to variables. This approach

avoids changing the state of data and promotes the use of immutable data structures. The emphasis is placed on what the program should achieve rather than how to accomplish it.

Example: Functional Approach

```
# Example: Functional Programming

from typing import List

# Pure functions
def is_electric(vehicle: dict) -> bool:
    """Check if a vehicle is electric."""
    return vehicle["type"] == "electric"

def average_range(vehicles: List[dict]) -> float:
    """Calculate the average range of a list of electric vehicles."""
    total_range = sum(vehicle["range"] for vehicle in vehicles)
    return total_range / len(vehicles) if vehicles else 0

# Main program
if __name__ == "__main__":
    vehicles = [
        {"name": "Tesla Model S", "type": "electric", "range": 600},
        {"name": "Ford F-150", "type": "diesel", "range": 800},
        {"name": "Nissan Leaf", "type": "electric", "range": 300},
    ]

    # Use filter to get only electric vehicles
    electric_vehicles = list(filter(is_electric, vehicles))
    avg_range = average_range(electric_vehicles)

    print("Electric Vehicles:", electric_vehicles)
    print(f"Average Range of Electric Vehicles: {avg_range:.2f} km")
```

Output

```
"D:\UWS Course materials\OOP\Exercise\Fixed_deposite_system\Functional.py"
Electric Vehicles: [{'name': 'Tesla Model S', 'type': 'electric', 'range':
600}, {'name': 'Nissan Leaf', 'type': 'electric', 'range': 300}]
Average Range of Electric Vehicles: 450.00 km

Process finished with exit code 0
```

Information to Note

Functions like `filter` and `map` operate in a declarative manner for computations. They emphasize what needs to be accomplished (such as filtering data and computing averages) without altering the original data.

Advantages

- Functions can be reused and tested independently.
- Minimizes bugs by avoiding side effects.
- Promotes writing concise and clear code.

Disadvantages

- It may be more challenging for beginners to grasp.
- Recursive calls can lead to performance issues, although tail-recursion optimizations are available.

Object Oriented Programming

Overview

Object-Oriented Programming (OOP) organizes code into classes and objects. Classes serve as blueprints for creating objects, specific instances of these classes. OOP emphasizes four fundamental principles: encapsulation, inheritance, polymorphism, and abstraction.

Characteristics

OOP focuses on bundling data and behaviour together within objects. It promotes reusability and scalability of code and adheres to the DRY principle (Don't Repeat Yourself).

Key Concepts

- Encapsulation: This involves bundling the data and the methods that operate on the data within a class.
- Inheritance: This allows new classes to be derived from existing ones, enabling code reuse and extension.

- Polymorphism: This permits methods to behave differently depending on the object that calls them.
- Abstraction: This technique hides the implementation details of a class, exposing only the necessary features to the outside world.

Example: OOP Approach

```python
# Example: Object-Oriented Programming

class Vehicle:
    """Base class for vehicles."""
    def __init__(self, name: str, fuel_capacity: float, fuel_efficiency: float):
        self.name = name
        self.fuel_capacity = fuel_capacity  # in liters
        self.fuel_efficiency = fuel_efficiency  # in km/l

    @property
    def max_range(self) -> float:
        """Calculate the maximum range of the vehicle."""
        return self.fuel_capacity * self.fuel_efficiency

    def __str__(self) -> str:
        return f"{self.name}: Max Range = {self.max_range:.2f} km"

class ElectricVehicle(Vehicle):
    """Electric vehicle class inheriting from Vehicle."""
    def __init__(self, name: str, battery_capacity: float, efficiency: float):
        super().__init__(name, fuel_capacity=0, fuel_efficiency=0)
        self.battery_capacity = battery_capacity  # in kWh
        self.efficiency = efficiency  # in km/kWh

    @property
    def max_range(self) -> float:
        """Override max_range for electric vehicles."""
        return self.battery_capacity * self.efficiency

    def __str__(self) -> str:
        return f"{self.name} (Electric): Max Range = {self.max_range:.2f} km"
```

```
# Main program
if __name__ == "__main__":
    car = Vehicle("Toyota Corolla", 50, 15)  # 50L tank, 15 km/L
    ev = ElectricVehicle("Tesla Model 3", 75, 6)  # 75 kWh battery, 6 km/kWh

    print(car)
    print(ev)
```

Output

```
"D:\UWS Course materials\OOP\Exercise\Fixed_deposite_system\OOP.py"
Toyota Corolla: Max Range = 750.00 km
Tesla Model 3 (Electric): Max Range = 450.00 km

Process finished with exit code 0
```

Advantages

- Modular and scalable: Code can be reused through inheritance and composition.
- Encapsulation improves data security.
- Polymorphism allows flexibility.

Disadvantages

- Can be overkill for small programs.
- Requires a good understanding of design principles.

Comparison Table:

Feature	**Procedural Programming**	**Functional Programming**	**Object-Oriented Programming**
Focus	Functions and instructions	Pure functions and immutability	Classes and objects
State Management	Global variables	Immutable data	Encapsulated within objects
Reusability	Code reuse through functions	Function reuse	Inheritance and polymorphism
Key Concepts	Functions, top-down approach	Pure functions, recursion, higher-order	Encapsulation, inheritance, polymorphism
Use Case	Small scripts, basic tasks	Data transformation, scientific computing	Large, scalable systems

Summary of Programming Paradigms

Procedural Programming

This programming paradigm focuses on writing sequences of instructions or procedures to solve problems. Programs are divided into smaller, reusable functions, which makes them easier to follow and modify. It is ideal for simple tasks or smaller applications.

Example: Writing a program to calculate a vehicle's fuel efficiency using functions.

Functional Programming

Functional programming is based on mathematical functions and avoids changing states or using mutable data. It emphasizes what to do rather than how to do it by utilizing pure, immutability, and higher-order functions. This approach is commonly used for data transformations, artificial intelligence, and parallel processing.

Example: Filtering a list of electric vehicles and calculating their average range.

Object-Oriented Programming (OOP)

OOP models real-world entities using objects that encapsulate both data and behaviours. It encourages reusability through inheritance, promotes modularity via encapsulation, and offers flexibility through polymorphism. This paradigm is suitable for complex, scalable, and modular systems.

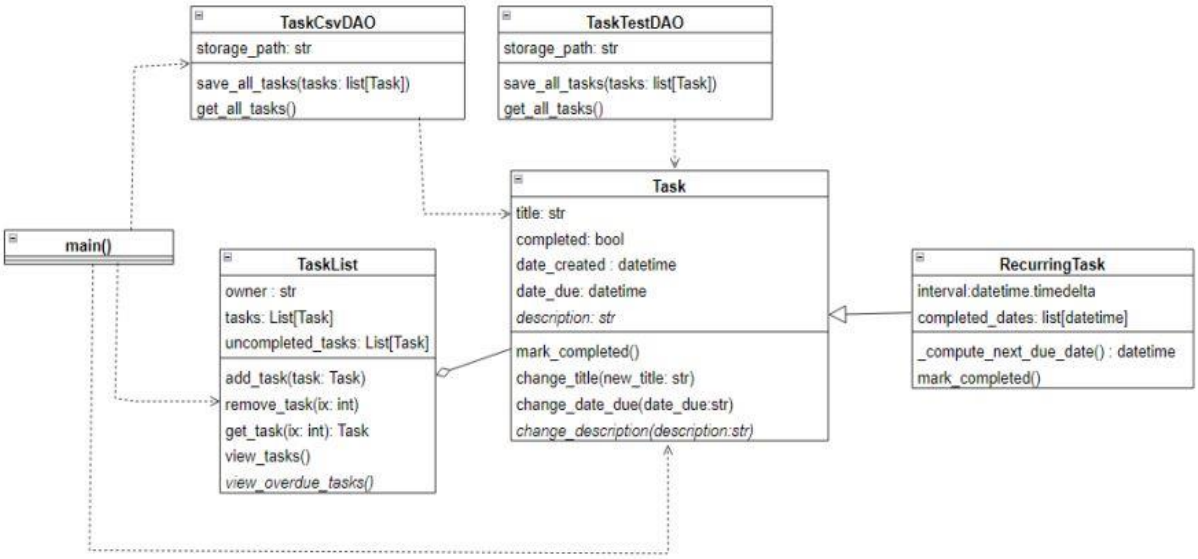
Example: Creating classes for vehicles and electric vehicles, each with specific properties and methods like maximum range.

Key Differences

- Procedural Programming emphasizes procedures and explicit instructions.
- Functional Programming relies on pure functions and immutable data.
- OOP combines data and behavior by modeling entities as objects in a modular structure.

Each programming paradigm has unique strengths and is suited for different types of tasks or applications.

Portfolio Exercise Lab Week 6 (Implementing Persistence)




```python
import csv
import datetime
from task import Task, RecurringTask  # Assuming Task and RecurringTask are defined in task.py


class TaskCsvDAO:
    def __init__(self, storage_path: str) -> None:
        self.storage_path = storage_path
        self.fieldnames = [
            "title", "type", "date_due", "completed",
            "interval", "completed_dates", "date_created", "description"
        ]

    def get_all_tasks(self) -> list[Task]:
        task_list = []
        try:
            with open(self.storage_path, "r") as file:
                reader = csv.DictReader(file)
                for row in reader:
                    # Validate that row is a dictionary
                    if not isinstance(row, dict):
                        print(f"Skipping invalid row: {row}")
                        continue

                    # Ensure required fields are present
                    if not row.get("title") or not row.get("type") or not row.get("date_due"):
                        print(f"Skipping invalid row: {row}")
                        continue

                    task_type = row["type"]
                    title = row["title"]
                    description = row["description"]
                    try:
                        date_due = datetime.datetime.strptime(row["date_due"], "%Y-%m-%d")
```

```
                date_created = datetime.datetime.strptime(row["date_created"], "%Y-%m-%d")
                completed = row["completed"] == "True"
            except (ValueError, KeyError):
                print(f"Skipping invalid dates in row: {row}")
                continue

            completed_dates = []
            if row.get("completed_dates"):
                completed_dates = [
                    datetime.datetime.strptime(date.strip(), "%Y-%m-%d")
                    for date in row["completed_dates"].split(",") if date.strip()
                ]

            if task_type == "Task":
                task = Task(title, date_due,  description)
                task.completed = completed
                task.date_created = date_created
            elif task_type == "RecurringTask":
                interval = datetime.timedelta(days=int(row["interval"].split()[0]))
                task = RecurringTask(title, date_due, description, interval)
                # task = RecurringTask(title=title, date_due=date_due, description=description,
                #                      interval=datetime.timedelta(days=interval))
                task.completed_dates = completed_dates
                task.completed = completed
                task.date_created = date_created
            else:
                print(f"Unknown task type in row: {row}")
                continue

            task_list.append(task)
    except FileNotFoundError:
        print("Task CSV file not found. Starting fresh.")
    return task_list

def save_all_tasks(self, tasks: list[Task]) -> None:
    try:
        with open(self.storage_path, "w", newline="") as file:
            writer = csv.DictWriter(file, fieldnames=self.fieldnames)
            writer.writeheader()
            for task in tasks:
                try:
                    row = {
                        "title": task.title,
                        "type": "RecurringTask" if isinstance(task, RecurringTask) else "Task",
                        "date_due": (
                            task.date_due.strftime("%Y-%m-%d")
                            if isinstance(task.date_due, datetime.datetime)
                            else task.date_due
                        ),
                        "completed": task.completed,
                        "description": task.description,
                        "interval": str(task.interval.days) if isinstance(task, RecurringTask) else "",
                        "completed_dates": ",".join(
                            date.strftime("%Y-%m-%d") for date in task.completed_dates
                        ) if isinstance(task, RecurringTask) else "",
                        "date_created": (
                            task.date_created.strftime("%Y-%m-%d")
                            if isinstance(task.date_created, datetime.datetime)
                            else task.date_created
                        )
```

```
                    }
                    writer.writerow(row)
                except Exception as e:
                    print(f"Error saving task: {task.title}, {e}")
            print("Tasks saved successfully.")
        except Exception as e:

            print(f"Error saving tasks: {e}")
```

Below is the implementation of taskdao.py in main.py

```
elif choice == 7:
    # Save tasks before exiting
    task_dao.save_all_tasks(task_list.tasks)
    print("System ends")
    break
```

Output

```
C:\Users\FUJITSU\AppData\Local\Microsoft\WindowsApps\python3.11.exe
C:\Users\FUJITSU\Desktop\Final_Updated_ToDo_Project\main.py
4 tasks loaded from CSV.
----------MENU------------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Save and exit the program.
-------------------------
Please make a selection from (1-6): 1
Select task Type(normal/recurring) task: normal
Add your task please: Attend OOP Lectures
Set the date (YYYY-mm-dd) :2024-11-22
Write a description for this task (or press Enter to skip): OOP Demonstration is on Friday!!!
----------MENU------------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Save and exit the program.
-------------------------
Please make a selection from (1-6): 2
Select a task to view (completed / uncompleted)?: uncompleted
Task list owner: Owner: Group4, Email: Group4@example.com
1. Task-eating (created: 2024-11-19 00:00:00, due: 2024-11-19 00:00:00, Completed: False). always
eating
2. To the cinema - Recurring (created: 2024-11-19, due: 2024-11-28, completed dates: [2024-11-27],
status: Not Completed, description: To watch the latest movie in town)
3. Task-Swiming (created: 2024-11-19 00:00:00, due: 2024-11-25 00:00:00, Completed: False). I love
to swim
4. Task-Attend OOP Lectures (created: 2024-11-20 14:40:01.793828, due: 2024-11-22 00:00:00,
Completed: False). OOP Demonstration is on Friday!!!
```

```
----------MENU------------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Save and exit the program.
------------------------
Please make a selection from (1-6): 7
Tasks saved successfully.
System ends

Process finished with exit code 0
```

From the output above the system starts with 4 tasks loaded from the CSV file using the get_all_task() method before the fifth task was added.

The output below shows all the five tasks saved to the CSV file. It has four (4) uncompleted task and one (1) completed task.

```
C:\Users\FUJITSU\AppData\Local\Microsoft\WindowsApps\python3.11.exe
C:\Users\FUJITSU\Desktop\Final_Updated_ToDo_Project\main.py
5 tasks loaded from CSV.
----------MENU------------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Save and exit the program.
-------------------------
Please make a selection from (1-6): 2
Select a task to view (completed / uncompleted)?: uncompleted
Task list owner: Owner: Group4, Email: Group4@example.com
1. Task-eating (created: 2024-11-19 00:00:00, due: 2024-11-19 00:00:00,
Completed: False). always eating
2. To the cinema - Recurring (created: 2024-11-19, due: 2024-11-28,
completed dates: [2024-11-27], status: Not Completed, description: To watch
the latest movie in town)
3. Task-Swiming (created: 2024-11-19 00:00:00, due: 2024-11-25 00:00:00,
Completed: False). I love to swim
4. Task-Attend OOP Lectures (created: 2024-11-20 00:00:00, due: 2024-11-22
00:00:00, Completed: False). OOP Demonstration is on Friday!!!
----------MENU------------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Save and exit the program.
-------------------------
Please make a selection from (1-6): 2
Select a task to view (completed / uncompleted)?: completed
```

```
Task list owner: Owner: Group4, Email: Group4@example.com
1. Task-Bath (created: 2024-11-19 00:00:00, due: 2024-11-19 00:00:00,
Completed: True). Bathing everyday
----------MENU------------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Save and exit the program.
--------------------------
Please make a selection from (1-6):
```

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

```
['Apple', 'Grapes', 'Banana']
['Apple', 'Grapes', 'Banana', 'Orange']
['Apple', 'Banana', 'Orange']
Length of the fruits list is 3
```

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

```
Normal for loop
BMW
AUDI
MERCEDES
FIAT

For loop with enumerate()
0) BMW
1) AUDI
2) MERCEDES
3) FIAT
```

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])
```

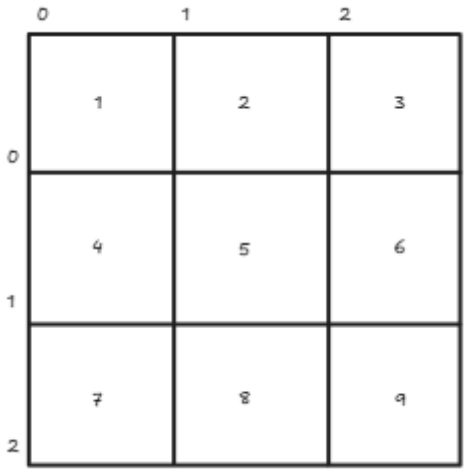
Output

```
Names from 0 to 2 ['Prajwal', 'Sreeraj']
Names from to 4 ['Prajwal', 'Sreeraj', 'Aman', 'Ali']
Names in -1 and -3 is Stephen and Aman
```

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

```
Matrix [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Value at 0th row, 2nd column in Matrix 3
```

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 using { } {'India': 'New Delhi', 'England': 'London', 'USA': 'Washington DC', 'Germany': 'Berlin'}
Dictionary using dict() {'India': 'New Delhi', 'England': 'London', 'USA': 'Washington DC', 'Germany': 'Berlin'}
Are both Equal: True
```

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

```
Accessing an element inside dictionary:  Aventador SVJ
{'BMW': 'M3 GTR', 'Dodge': 'challenger', 'Lamborghini': 'Aventador SVJ'}
Keys in cars dictionary dict_keys(['BMW', 'Dodge', 'Lamborghini'])
Values in cars dictionary dict_values(['M3 GTR', 'challenger', 'Aventador SVJ'])
```

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

```
Key is BMW and value is M3 GTR
Key is Dodge and value is challenger
Key is Lamborghini and value is Aventador SVJ
```

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[77], line 9
      6 print(car, doll, scooter, ball)
      8 # assigning a value throws an error
----> 9 toys[0] = "bike"

TypeError: 'tuple' object does not support item assignment
```

But we can assign after destructuring.

Example Code

```
print("Before Assigning", car)
car = "sedan"
```

```
print("After Assigning", car)
```

Output

```
Before Assigning car
After Assigning sedan
```

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

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
```

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}))
```

```
Union: {1, 2, 3, 4, 5, 6, 8, 11}
```

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

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

```
Intersection: {2, 3}
```

3. 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}))
```

```
Difference: {1, 2, 3}
```

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

```
Dog is barking
Cat is meowing
Mouse is sqeeking
```

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

```
----------MENU-----------
| 1. Add a new task to the list.
| 2. View the completed and uncompleted tasks  in the list.
| 3. Remove a task from the list.
| 4. Select a Task to perform
 (Complete Task/ description/Change due date).
| 5. Change the title of a task.
| 6. View Task Overdue.
| 7. Quit and exit the program.
-------------------------
Please make a selection from (1-6): 1
Select task Type(normal/recurring/priority) task: priority
Add your task please: Join Board Meeting
Set the date (YYYY-mm-dd) :2024-11-22
Write a description for this task (or press Enter to skip): All board members assembled in MS Teams and important meeting.
Enter the priority (low,medium,high): high
```

```
Please make a selection from (1-6): 2
Select a task to view (completed / uncompleted)?: uncompleted

----------------TASK LIST-----------------
Task list owner: Owner: Group4, Email: Group4@example.com
------------------------------------------
1. Join Board Meeting - Priority - high (created: 2024-11-19, status: Not Completed) (PriorityTask)
------------------------------------------
```

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.