Assignment

Course Code: CSE-221

Course Title: OOP-2

Topic: CLASS WORK

Submitted To:

Teacher's Name: Nasima Islam Bithi

Designation: Lecturer

Department: CSE

Daffodil International University

Submitted By:

Student Name: MD.MAHADI HASSAN EMON

ID: 0242220005101629

Section: 63_M

Department: CSE

Daffodil International University

Date of submission: 2024-12-01

OOP TASK

Task 1: Function to calculate (a+b)2=a2+b2+2ab(a + b)^2 = a^2 + b^2 + 2ab(a+b)2=a2+b2+2ab

```
def calculate_formula(a, b):
    return a**2 + b**2 + 2 * a * b


# Taking input
a = int(input("Enter value for a: "))
b = int(input("Enter value for b: "))
print("Result using the function:", calculate_formula(a, b))
```

Task 2: Lambda function for the same formula

```
lambda_formula = lambda a, b: a**2 + b**2 + 2 * a * b


# Using the lambda function
print("Result using lambda function:", lambda_formula(a, b))
```

Task 3: Recursive function for factorial

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)


# Taking input
n = int(input("Enter a number to find its factorial: "))
print(f"The factorial of {n} is:", factorial(n))
```

Task 4: Function to check if a number is prime

```
def is_prime(num):
    if num <= 1:
        return False
```

```
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

# Taking input
num = int(input("Enter a number to check if it's prime: "))
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

LIST

TASK-1

```
# Initialize the list
a = [1, 3, 5, 7, 9]

# 1. Access a[-2] and a[2], and find length and type of the list
print("a[-2]:", a[-2])  # Second last element
print("a[2]:", a[2])    # Third element
print("Length of the list:", len(a))
print("Type of a:", type(a))

# 2. Change a[3] = 50 and a[2] = 19
a[3] = 50
a[2] = 19
print("List after changes:", a)

# 3. Add 100 at the last index and 200 at index 2
```

```
a.append(100)  # Add 100 at the end
a.insert(2, 200)  # Add 200 at index 2
print("List after adding 100 and 200:", a)

# 4. Remove the last element and the element at index 1
a.pop()  # Remove last element
del a[1]  # Remove element at index 1
print("List after removals:", a)

# 5. Join a new list [2, 4, 6] with `a`
b = [2, 4, 6]
a.extend(b)  # Join lists
print("List after joining with [2, 4, 6]:", a)

# 6. Copy all values into a new list `b`
b = a.copy()
print("New copied list b:", b)

# 7. Sort the elements of `b`
b.sort()
print("Sorted list b:", b)

# 8. Print all elements using a loop and break if the element is 5
print("Elements in `a` (break if 5):")
for element in a:
    if element == 5:
        print(element)
        break
    print(element)
```

```
# 9. Find the largest number in `a`
largest = max(a)
print("Largest number in `a`:", largest)
```

Tupple

Task-2

```
# Initialize the tuple
a = (1, 3, 5, 7, 4)

# a) Find the sum of all odd numbers in `a`
odd_sum = sum(num for num in a if num % 2 != 0)
print("Sum of all odd numbers in `a`:", odd_sum)

# b) Find the index of a specific element (e.g., 5)
element_to_find = 5
index_of_element = a.index(element_to_find) if element_to_find in a else None
print(f"Index of {element_to_find} in `a`:", index_of_element)

# c) Count the number of odd and even numbers separately
odd_count = sum(1 for num in a if num % 2 != 0)
even_count = len(a) - odd_count
print("Number of odd numbers:", odd_count)
print("Number of even numbers:", even_count)

# d) Extend the tuple with `(2, 4, 6)` (Tuples are immutable; create a new one)
extended_tuple = a + (2, 4, 6)
print("Extended tuple:", extended_tuple)
```

```
# e) Add a new item at index 2 (Tuples are immutable; create a new one)
item_to_add = 200
new_tuple = a[:2] + (item_to_add,) + a[2:]
print("Tuple after adding 200 at index 2:", new_tuple)

# f) Remove the last element (Tuples are immutable; create a new one)
modified_tuple = a[:-1]
print("Tuple after removing the last element:", modified_tuple)

# g) Perform slicing `[-4:-1]`
sliced_tuple = a[-4:-1]
print("Sliced tuple [-4:-1]:", sliced_tuple)

# h) Print the tuple using a loop and use `continue` if the element is 5
print("Tuple elements (skipping 5):")
for num in a:
    if num == 5:
        continue
    print(num)
```

SET

```
# Define the sets a and b
a = {1, 3, 5, 8, 3, 7}
b = {0, False, 1, 5}

# Print the sets and their types
print("Set a:", a)
```

```
print("Type of a:", type(a))
print("Set b:", b)
print("Type of b:", type(b))

# Print the length of the sets
print("Length of set a:", len(a))
print("Length of set b:", len(b))

# Add a new element 10 to set a
a.add(10)

# Remove 8 from set a
a.remove(8)

# Perform union, intersection, difference, symmetric difference, and subset operations
union_set = a.union(b)
intersection_set = a.intersection(b)
difference_set = a.difference(b)
symmetric_difference_set = a.symmetric_difference(b) 1
is_subset = a.issubset(b)

print("Union of a and b:", union_set)
print("Intersection of a and b:", intersection_set)
print("Difference of a and b:", difference_set)
print("Symmetric difference of a and b:", symmetric_difference_set)
print("Is set a a subset of set b?", is_subset)
```

```
# Join a new list [2, 3, 4] with set a
new_list = [2, 3, 4]
set_a_joined = a.union(set(new_list))
print("Set a after joining the new list:", set_a_joined)
```

DICSONARY

```
# Define the dictionary
employee = {
    "name": "A",
    "age": 40,
    "type": {"developer": ["ios", "android"]},
    "permanent": True,
    "salary": 30000,
    100: (1, 2, 3),
    45: {5, 6, True, 7, 1}
}

# 1. Print length, type, and dictionary
print("Length of dictionary:", len(employee))
print("Type of dictionary:", type(employee))
print("Dictionary:", employee)

# 2. Access the key employee["type"]["developer"]
developer_skills = employee["type"]["developer"]
print("Developer skills:", developer_skills)

# 3. Change the value of "permanent" to False
employee["permanent"] = False
```

```
# 4. Add a new key "gender" with value "male"
employee["gender"] = "male"

# 5. Remove "age" key from dictionary
del employee["age"]

# 6. Use keys(), values(), items()
keys = employee.keys()
values = employee.values()
items = employee.items()

print("Keys:", keys)
print("Values:", values)
print("Items:", items)

# 7. Iterate the dictionary using a loop
for key, value in employee.items():
    print(f"Key: {key}, Value: {value}")
```

STRING

```
# Define the strings
a = "hello"
b = "b2b2b2"
c = "3g3g"

# 1. Concatenate a, b, and c into d
d = a + b + c
```

```
# 2. Find the length of d and print d
length_d = len(d)
print("Length of d:", length_d)
print("d:", d)

# 3. Check if "a2" is present in d
if "a2" in d:
    print("'a2' is present in d")
else:
    print("'a2' is not present in d")

# 4. Perform various string operations on d
print("Uppercase:", d.upper())
print("Lowercase:", d.lower())
print("Titlecase:", d.title())
print("Is it all alphanumeric?", d.isalnum())
print("Find '3g':", d.find("3g"))
print("Capitalize:", d.capitalize())
print("Is it all alphanumeric?", d.isalnum())
print("Count 'b2':", d.count("b2"))
print("Split:", d.split())
print("Swapcase:", d.swapcase())
print("Strip leading and trailing spaces:", d.strip())
print("Replace 'hello' with 'python':", d.replace("hello", "python"))
```

CLASS OBJECT

```
class Shape:
    def __init__(self, name):
        self.name = name
```

```
    def get_name(self):
        return self.name

    def display_info(self):
        print(f"Shape: {self.name}")


class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)


class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def display_detail(self):
        print(f"Name: {self.name}, Price: {self.price}")
```

```
class ElectronicProduct(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price)
        self.warranty = warranty

    def display_detail(self):
        super().display_detail()
        print(f"Warranty: {self.warranty}")



# Creating objects
rectangle = Rectangle("Rectangle", 5, 4)
rectangle.display_info()
print(f"Area: {rectangle.area()}")
print(f"Perimeter: {rectangle.perimeter()}")

electronic_product = ElectronicProduct("Laptop", 1000, "1 year")
electronic_product.display_detail()
```

ENCAPSULATION

```
class Vehicle:
    def __init__(self, color):
        self.__color = color

    def get_color(self):
        return self.__color

    def set_color(self, color):
        self.__color = color
```

```
    def vehicle_info(self):
        print(f"Color: {self.__color}")


class Taxi(Vehicle):
    def __init__(self, color, model, capacity, variant):
        super().__init__(color)
        self.__model = model
        self.__capacity = capacity
        self.__variant = variant

    def get_model(self):
        return self.__model

    def set_model(self, model):
        self.__model = model

    def get_capacity(self):
        return self.__capacity

    def set_capacity(self, capacity):
        self.__capacity = capacity

    def get_variant(self):
        return self.__variant

    def set_variant(self, variant):
        self.__variant = variant
```

```
    def vehicle_info(self):
        super().vehicle_info()
        print(f"Model: {self.__model}")
        print(f"Capacity: {self.__capacity}")
        print(f"Variant: {self.__variant}")


# Create two instances
t1 = Taxi("Red", "Toyota Camry", 4, "Hybrid")
t2 = Taxi("Black", "Honda Civic", 5, "Petrol")

# Access and modify properties
t1.set_model("Toyota Corolla")
t2.set_variant("Diesel")

# Display vehicle information
t1.vehicle_info()
t2.vehicle_info()
```

INHERITANCE

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def display(self):
        print(f"Name: {self.first_name} {self.last_name}")


class Student(Person):
```

```
    def __init__(self, first_name, last_name, graduation_year):
        super().__init__(first_name, last_name)
        self.graduation_year = graduation_year

    def display(self):
        super().display()
        print(f"Graduation Year: {self.graduation_year}")


class Alumni(Student):
    def __init__(self, first_name, last_name, graduation_year, passing_year):
        super().__init__(first_name, last_name, graduation_year)
        self.passing_year = passing_year

    def display(self):
        super().display()
        print(f"Passing Year: {self.passing_year}")


class CurrentStudent(Student):
    def __init__(self, first_name, last_name, graduation_year, current_semester):
        super().__init__(first_name, last_name, graduation_year)
        self.current_semester = current_semester

    def display(self):
        super().display()
        print(f"Current Semester: {self.current_semester}")
```

```
class Teacher(Person):
    def __init__(self, first_name, last_name, joining_year):
        super().__init__(first_name, last_name)
        self.joining_year = joining_year

    def display(self):
        super().display()
        print(f"Joining Year: {self.joining_year}")



class Admin(Person):
    def __init__(self, first_name, last_name, joining_year):
        super().__init__(first_name, last_name)
        self.joining_year = joining_year

    def display(self):
        super().display()
        print(f"Joining Year: {self.joining_year}")



class Employee(Person):
    def __init__(self, first_name, last_name, id):
        super().__init__(first_name, last_name)
        self.id = id

    def display(self):
        super().display()
        print(f"ID: {self.id}")
```

```
# Creating instances
alumni = Alumni("Alice", "Johnson", 2023, 2024)
current_student = CurrentStudent("Bob", "Smith", 2025, 3)
teacher = Teacher("Carol", "Davis", 2010)
admin = Admin("David", "Lee", 2015)
employee = Employee("Eve", "Miller", 12345)

# Displaying information
alumni.display()
current_student.display()
teacher.display()
admin.display()
employee.display()
```

POLYMORPHISOM

```
class Department:
    def __init__(self, name):
        self.name = name

    def display_name(self):
        print(f"Department Name: {self.name}")



class Teacher(Department):
    def __init__(self, name, schedule_class):
        super().__init__(name)
        self.schedule_class = schedule_class

    def schedule_class(self):
```

```
        print(f"Schedule Class: {self.schedule_class}")

    def grade_student(self):
        print("Grading Students...")

    def display_name(self):
        super().display_name()
        print("Role: Teacher")


class Author(Department):
    def __init__(self, name):
        super().__init__(name)

    def write_article(self):
        print("Writing an Article...")

    def publish_blog(self):
        print("Publishing a Blog...")


class TeacherAuthor(Teacher, Author):
    def __init__(self, name, schedule_class):
        super().__init__(name, schedule_class)

    def display_name(self):
        super().display_name()
        print("Role: Teacher and Author")
```

```
# Creating an instance of TeacherAuthor
teacher_author = TeacherAuthor("Dr. Smith", "Computer Science")


# Accessing methods
teacher_author.schedule_class()
teacher_author.grade_student()
teacher_author.write_article()
teacher_author.publish_blog()
teacher_author.display_name()
```

EXCEPTATION HANDLEING

1. Custom Exception for Invalid Age:

```
class InvalidVoterException(Exception):
    pass

def check_age(age):
    if age < 18:
        raise InvalidVoterException("You are not eligible to vote.")
    else:
        print("You are eligible to vote.")

age = int(input("Enter your age: "))
try:
    check_age(age)
except InvalidVoterException as e:
    print(e)
```

2. Custom Exception for Salary Out of Range:

```
class SalaryNotInRangeException(Exception):
    pass

class Employee:
    def __init__(self, name, salary):
        if 19000 <= salary <= 50000:
            self.name = name
            self.salary = salary
        else:
            raise SalaryNotInRangeException("Salary should be between 19000 and 50000.")
```

```
    def display_info(self):
        print(f"Name: {self.name}, Salary: {self.salary}")

try:
    employee = Employee("Alice", 15000)
except SalaryNotInRangeException as e:
    print(e)
```

3. Handling Division by Zero and Index Errors:

```
arr = [10, 5, 15, 20]

divisor = int(input("Enter a divisor: "))

try:
    for i in range(len(arr)):
        result = arr[i] / divisor
        print(f"{arr[i]} / {divisor} = {result}")
except ZeroDivisionError:
    print("Error: Division by zero")
except IndexError:
    print("Error: Index out of range")
except ValueError:
    print("Error: Invalid input")
```

4. Custom Exception for Insufficient Balance:

```
class InsufficientFundsException(Exception):
    pass

class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsException("Insufficient funds")
        self.balance -= amount
        print(f"Withdrew {amount}. Current balance: {self.balance}")

account = BankAccount(1000)
try:
    account.withdraw(1500)
except InsufficientFundsException as e:
    print(e)
```

NUMPY

```
import numpy as np
```

```
# 1. Create a NumPy array
score = np.array([85, 90, 78, 97, 88])

# a) Convert data type to float
score_float = score.astype(float)
print("Score (float):", score_float)

# b) Create a copy and add 5 points
score_copy = score.copy() + 5
print("Score copy:", score_copy)

# c) Find array properties
print("Shape:", score.shape)
print("Dimension:", score.ndim)
print("Size:", score.size)
print("Item size:", score.itemsize)
print("Data type:", score.dtype)
print("Sorted score:", np.sort(score))

# d) Find indices with scores >= 80
indices = np.where(score >= 80)
print("Indices with scores >= 80:", indices)

# e) Find min, max, std, var, sum, mean, axis-wise mean
print("Min:", np.min(score))
print("Max:", np.max(score))
print("Standard deviation:", np.std(score))
print("Variance:", np.var(score))
```

```
print("Sum:", np.sum(score))

print("Mean:", np.mean(score))

print("Axis-wise mean:", np.mean(score, axis=0))


# f) Print specific elements and slices

print("Score[2]:", score[2])

print("Score[-3:]:", score[-3:])

print("Score[1:4]:", score[1:4])
```

TASK-1

Let's break down the class diagram into two tasks:

Task 1:

- We have an abstract class `Vehicle` with two methods: `start()` and `stop()`.
- Two concrete classes `Car` and `Motorcycle` inherit from `Vehicle`.

Task 2:

- The `Vehicle` class is now modified with additional attributes: `brand` and `description`.
- The `startEngine()` method is made abstract.

Python Implementation

```
class Vehicle:
    def start(self):
        pass


    def stop(self):
        pass




class Car(Vehicle):
    def __init__(self):
        pass
```

```
class Motorcycle(Vehicle):
    def __init__(self):
        pass


# Create an object of Car
car = Car()

# Create an object of Vehicle (This will raise an error)
vehicle = Vehicle()  # TypeError: Can't instantiate abstract class Vehicle with abstract methods startEngine
```

TASK-2

```
class Vehicle:
    def __init__(self, brand, description):
        self.brand = brand
        self.description = description

    def startEngine(self):
        raise NotImplementedError("Subclasses must implement this method")

    def stopEngine(self):
        pass


class Car(Vehicle):
    def __init__(self, brand, model, description):
        super().__init__(brand, description)
```

```
        self.model = model

    def startEngine(self):
        print(f"Starting {self.brand} {self.model}")


# Create an object of Car
car = Car("Toyota", "Camry", "A comfortable sedan")
car.startEngine()  # Output: Starting Toyota Camry

# Create an object of Vehicle (This will raise an error)
vehicle = Vehicle("Unknown", "Unknown")  # TypeError: Can't instantiate abstract class Vehicle with abstract methods startEngine
```