Python for Computer Science and Data Science 2 (CSE 3652)

MINOR ASSIGNMENT-1: OBJECT-ORIENTED PROGRAMMING (OOP)

1. What is the significance of classes in Python programming, and how do they contribute to object-oriented programming?

Ans:- Classes in Python are fundamental to object-oriented programming (OOP), enabling the creation of reusable and modular code. Here's why they are significant:

a. Encapsulation - Classes allow bundling of data (attributes) and behavior (methods) together. Access control can be implemented using public, protected, and private members.
b. Reusability and Modularity - Code can be organized into self-contained objects, making it easier to maintain and extend. A class can be instantiated multiple times to create objects with shared structure but unique data.
c. Abstraction - Unnecessary details can be hidden while exposing only the necessary functionalities. Helps in simplifying complex implementations.
d. Inheritance - Allows new classes (child classes) to inherit attributes and methods from existing classes (parent classes). Promotes code reuse and enables hierarchical relationships.
e. Polymorphism - Different classes can define methods with the same name but different implementations. Enables flexibility in designing extensible code.

2. Create a custom Python class for managing a bank account with basic functionalities like deposit and withdrawal?

Ans:-

```
class BankAccount:
    def __init__(self, account_holder, balance=0.0):
        self.account_holder = account_holder
        self.balance = balance
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"₹{amount} deposited successfully.")
        else:
            print("Invalid deposit amount.")
    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            print(f"₹{amount} withdrawn successfully.")
        elif amount > self.balance:
            print("Insufficient balance.")
        else:
            print("Invalid withdrawal amount.")
    def get_balance(self):
        return f"Available balance: ₹{self.balance}"

account = BankAccount("Dinanath Dash", 5000)
account.deposit(1500)
print(account.get_balance())
account.withdraw(2000)
print(account.get_balance())
```

Output:-

```
₹1500 deposited successfully.
Available balance: ₹6500
₹2000 withdrawn successfully.
Available balance: ₹4500
```

3. Create a Book class that contains multiple Chapters, where each Chapter has a title and page count. Write code to initialize a Book object with three chapters and display the total page count of the book.

Ans:-

```
class Chapter:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages
class Book:
    def __init__(self, title):
        self.title = title
        self.chapters = []
    def add_chapter(self, chapter):
        self.chapters.append(chapter)

    def total_pages(self):
        return sum(chapter.pages for chapter in self.chapters)
book = Book("Python Programming")
chapter1 = Chapter("Introduction to Python", 20)
chapter2 = Chapter("Object-Oriented Programming", 35)
chapter3 = Chapter("Advanced Topics", 50)
book.add_chapter(chapter1)
book.add_chapter(chapter2)
book.add_chapter(chapter3)
print(f"Total pages in '{book.title}': {book.total_pages()}")
```

Output:- `Total pages in 'Python Programming': 105`

4. How does Python enforce access control to class attributes, and what is the difference between public, protected, and private attributes?

Ans:- Python enforces access control through naming conventions rather than strict access modifiers like in Java or C++. It uses prefixes (_ and __) to indicate different levels of attribute visibility.

a. Public Attributes (name): Accessible from anywhere (inside and outside the class). No restrictions on modification.

```
class Example:
    def __init__(self):
        self.public_var = "I am Public"
obj = Example()
print(obj.public_var)
obj.public_var = "Modified"
```

b. Protected Attributes (_name): Indicated with a single underscore (_). Not enforced by Python but treated as a convention: "Use this carefully." Can still be accessed but should be considered internal to the class or its sub-classes.

```
class Example:
    def __init__(self):
        self._protected_var = "I am Protected"
obj = Example()
print(obj._protected_var)
obj._protected_var = "Modified"
```

c. Private Attributes (__name): Indicated with double underscores (__). Python performs name mangling, renaming __var to _ClassName__var, making it harder to access directly. Cannot be accessed directly outside the class but can still be accessed using name mangling.

```
class Example:
    def __init__(self):
        self.__private_var = "I am Private"
obj = Example()
```

```
# print(obj.__private_var)
print(obj._Example__private_var)
```

5. Write a Python program using a Time class to input a given time in 24-hour format and convert it to a 12-hour format with AM/PM. The program should also validate time strings to ensure they are in the correct HH:MM:SS format. Implement a method to check if the time is valid and return an appropriate message.

Ans:-

```
import re
class Time:
    def __init__(self, time_str):
        self.time_str = time_str
        self.hours, self.minutes, self.seconds = self._parse_time()
    def _parse_time(self):
        pattern = r"^(2[0-3]|[01]?[0-9]):([0-5]?[0-9]):([0-5]?[0-9])$"
        match = re.match(pattern, self.time_str)
        if match:
            return int(match.group(1)), int(match.group(2)), int(match.group(3))
        else:
            raise ValueError("Invalid time format. Please use HH:MM:SS (24-hour format).")
    def to_12_hour_format(self):
        period = "AM" if self.hours < 12 else "PM"
        hours_12 = self.hours if 1 <= self.hours <= 12 else (self.hours - 12) or 12
        return f"{hours_12:02}:{self.minutes:02}:{self.seconds:02} {period}"
    def display(self):
        print(f"24-hour format: {self.time_str}")
        print(f"12-hour format: {self.to_12_hour_format()}")
try:
    time_input = input("Enter time in HH:MM:SS (24-hour format): ")
    time_obj = Time(time_input)
    time_obj.display()
except ValueError as e:
    print(e)
```

Output:-

```
Enter time in HH:MM:SS (24-hour format): 19:50:12
24-hour format: 19:50:12
12-hour format: 07:50:12 PM
```

6. Write a Python program that uses private attributes for creating a BankAccount class. Implement methods to deposit, withdraw, and display the balance, ensuring direct access to the balance attribute is restricted. Explain why using private attributes can help improve data security and prevent accidental modifications.

Ans:- Why Use Private Attributes?

a. Encapsulation- Prevents direct modification of sensitive attributes.

b. Data Security- Ensures controlled access via methods, reducing risks of unintended changes.

c. Prevents Accidental Modification- Direct access to __balance is restricted, avoiding unintended overwrites.

```
class BankAccount:
    def __init__(self, account_holder, balance=0.0):
        self.__account_holder = account_holder
        self.__balance = balance
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"₹{amount} deposited successfully.")
        else:
```

```
            print("Invalid deposit amount.")
        def withdraw(self, amount):
            if 0 < amount <= self.__balance:
                self.__balance -= amount
                print(f"₹{amount} withdrawn successfully.")
            else:
                print("Insufficient balance or invalid amount.")
        def get_balance(self):
            return f"Available balance: ₹{self.__balance}"
account = BankAccount("Dinanath Dash", 5000)
account.deposit(1500)
print(account.get_balance())
account.withdraw(2000)
print(account.get_balance())
```

Output:-

```
₹1500 deposited successfully.
Available balance: ₹6500
₹2000 withdrawn successfully.
Available balance: ₹4500
```

7. Write a Python program to simulate a card game using object-oriented principles. The program should include a Card class to represent individual playing cards, a Deck class to represent a deck of cards, and a Player class to represent players receiving cards. Implement a shuffle method in the Deck class to shuffle the cards and a deal method to distribute cards to players. Display each player's hand after dealing.

Ans:-

```
import random
class Card:
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
    def __str__(self):
        return f"{self.rank} of {self.suit}"
class Deck:
    def __init__(self):
        suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
        ranks = ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"]
        self.cards = [Card(suit, rank) for suit in suits for rank in ranks]
    def shuffle(self):
        random.shuffle(self.cards)
    def deal(self, num_players, cards_per_player):
        if num_players * cards_per_player > len(self.cards):
            raise ValueError("Not enough cards to deal")
        return [[self.cards.pop() for _ in range(cards_per_player)] for _ in range(num_players)]
class Player:
    def __init__(self, name, hand):
        self.name = name
```

Output:-

```
Player 1's hand: 3 of Spades, K of Clubs, 8 of Hearts, 10 of Clubs, 6 of Spades
Player 2's hand: 9 of Diamonds, 10 of Spades, 6 of Diamonds, A of Spades, 8 of Clubs
Player 3's hand: 2 of Hearts, 4 of Spades, A of Hearts, 7 of Diamonds, K of Spades
Player 4's hand: 5 of Hearts, 8 of Spades, Q of Spades, 4 of Diamonds, 10 of Diamonds
```

8. Write a Python program that defines a base class Vehicle with attributes make and model, and a method display info(). Create a subclass Car that inherits from Vehicle and adds an additional at- tribute num doors. Instantiate both Vehicle and Car objects, call their display info() methods, and explain how the

subclass inherits and extends the functionality of the base class.

Ans:- How Inheritance Works Here:

a. The Car subclass inherits attributes (make, model) and methods (display_info()) from the Vehicle base class.
b. The super().__init__() call allows Car to initialize inherited attributes from Vehicle.
c. The display_info() method in Car extends the base class method by adding num_doors.

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model
    def display_info(self):
        print(f"Make: {self.make}, Model: {self.model}")
class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        super().__init__(make, model)
        self.num_doors = num_doors
    def display_info(self):
        super().display_info()
        print(f"Number of doors: {self.num_doors}")
vehicle = Vehicle("Toyota", "Corolla")
car = Car("Honda", "Civic", 4)
vehicle.display_info()
car.display_info()
```

Output:-

```
Make: Toyota, Model: Corolla
Make: Honda, Model: Civic
Number of doors: 4
```

9. Write a Python program demonstrating polymorphism by creating a base class Shape with a method area(), and two subclasses Circle and Rectangle that override the area() method. Instantiate objects of both subclasses and call the area() method. Explain how polymorphism simplifies working with different shapes in an inheritance hierarchy.

Ans:- How Polymorphism Works Here:

a. The Shape base class defines a common interface (area()).
b. Circle and Rectangle override area() to provide specific implementations.
c. Using polymorphism, both objects can be processed through the same loop, simplifying handling different shapes dynamically.

```
import math
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses must implement this method")
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
shapes = [Circle(5), Rectangle(4, 6)]
for shape in shapes:
    print(f"Area: {shape.area()}")
```

Output:-

```
Area: 78.53981633974483
Area: 24
```

10. Implement the CommissionEmployee class with __init__, earnings, and __repr__ methods. Include properties for personal details and sales data. Create a test script to instantiate the object, display earnings, modify sales data, and handle data validation errors for negative values.

Ans:-

```
class CommissionEmployee:
    def __init__(self, name, sales, commission_rate):
        if sales < 0 or commission_rate < 0:
            raise ValueError("Sales and commission rate must be non-negative")
        self.name = name
        self.sales = sales
        self.commission_rate = commission_rate
    def earnings(self):
        return self.sales * self.commission_rate
    def __repr__(self):
        return f"CommissionEmployee(name='{self.name}', sales={self.sales}, "+\
               f"commission_rate={self.commission_rate})"
try:
    employee = CommissionEmployee("John Doe", 5000, 0.1)
    print(employee)
    print(f"Earnings: ₹{employee.earnings()}")
    employee.sales = 7000
    print(f"Updated Earnings: ₹{employee.earnings()}")
    employee.sales = -1000
except ValueError as e:
    print(f"Error: {e}")
```

Output:-

```
CommissionEmployee(name='John Doe', sales=5000, commission_rate=0.1)
Earnings: ₹500.0
Updated Earnings: ₹700.0
```

11. What is duck typing in Python? Write a Python program demonstrating duck typing by creating a function describe() that accepts any object with a speak() method. Implement two classes, Dog and Robot, each with a speak() method. Pass instances of both classes to the describe() function and explain how duck typing allows the function to work without checking the object's type.

Ans:- How Duck Typing Works Here:

a. The describe() function does not check the type of the object.
b. As long as the object has a speak() method, it works.
c. Both Dog and Robot have speak(), so they are compatible.
d. Python's duck typing allows flexibility without enforcing explicit type checks.

```
class Dog:
    def speak(self):
        return "Woof!"
class Robot:
    def speak(self):
        return "Beep Boop!"
def describe(entity):
    print(entity.speak())
dog = Dog()
robot = Robot()
describe(dog)
describe(robot)
```

Output:-

```
Woof!
Beep Boop!
```

12. WAP to overload the + operator to perform addition of two complex numbers using a custom Complex class?

Ans:-

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)
    def __str__(self):
        return f"{self.real} + {self.imag}i"
c1 = Complex(3, 4)
c2 = Complex(1, 2)
c3 = c1 + c2
print(c3)
```

Output:- `4 + 6i`

13. WAP to create a custom exception class in Python that displays the balance and withdrawal amount when an error occurs due to insufficient funds?

Ans:-

```
class InsufficientFundsError(Exception):
    def __init__(self, balance, withdrawal_amount):
        super().__init__(f"Insufficient funds: Balance ₹{balance}, "+\
                         f"Withdrawal ₹{withdrawal_amount}")
        self.balance = balance
        self.withdrawal_amount = withdrawal_amount

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

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(self.balance, amount)
        self.balance -= amount
        return f"Withdrawal successful. Remaining balance: ₹{self.balance}"

try:
    account = BankAccount(5000)
    print(account.withdraw(7000))
except InsufficientFundsError as e:
    print(e)
```

Output:- `Insufficient funds: Balance ₹5000, Withdrawal ₹7000`

14. Write a Python program using the Card data class to simulate dealing 5 cards to a player from a shuffled deck of standard playing cards. The program should print the player's hand and the number of remaining cards in the deck after the deal.

Ans:-

```
import random
from dataclasses import dataclass
@dataclass
class Card:
    suit: str
    rank: str
```

```
class Deck:
    def __init__(self):
        suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
        ranks = ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"]
        self.cards = [Card(suit, rank) for suit in suits for rank in ranks]
        random.shuffle(self.cards)
    def deal(self, num_cards):
        return [self.cards.pop() for _ in range(num_cards)] if num_cards <= len(self.cards) else None
deck = Deck()
player_hand = deck.deal(5)
print("Player's Hand:")
for card in player_hand:
    print(f"{card.rank} of {card.suit}")
print(f"Remaining cards in deck: {len(deck.cards)}")
```

Output:-

```
Player's Hand:
8 of Hearts
4 of Diamonds
5 of Clubs
3 of Clubs
K of Clubs
Remaining cards in deck: 47
```

15. How do Python data classes provide advantages over named tuples in terms of flexibility and functionality? Give an example using python code.

Ans:- Advantages of Data Classes Over Named Tuples

Feature	Named Tuple	Data Class
Immutability	Immutable by default	Mutable by default
Methods	Cannot have methods	Can define methods
Inheritance	Limited support	Fully supports inheritance
Default Values	Requires _replace()	Supports default values
Type Annotations	Optional	Strongly supported
Flexibility	Less flexible	More flexible with additional functionality

```
Example: NamedTuple vs. Data Class
Using NamedTuple

from collections import namedtuple
Person = namedtuple("Person", ["name", "age"])
p1 = Person("Alice", 30)
print(p1.name, p1.age)
try:
    p1.age = 35
except AttributeError as e:
    print(e)

Using Data Class
from dataclasses import dataclass
@dataclass
```

```
class Person:
    name: str
    age: int
    def birthday(self):
        self.age += 1
        return f"Happy Birthday! {self.name} is now {self.age} years old."
p2 = Person("Alice", 30)
print(p2.birthday())
p2.age = 35  # Mutable
print(p2)
```

Key Takeaways:

a. Data classes allow mutability, while named tuples are immutable.
b. Data classes support methods and custom logic, enhancing functionality.
c. Data classes allow default values, type hints, and inheritance.

16. Write a Python program that demonstrates unit testing directly within a function's docstring using the doctest module. Create a function add(a, b) that returns the sum of two numbers and includes multiple test cases in its docstring. Implement a way to automatically run the tests when the script is executed.

Ans:- How It Works:

a. Docstring contains test cases that specify expected outputs.
b. doctest.testmod() automatically runs all embedded test cases when executed.
c. If all tests pass, no output is shown; otherwise, failures are displayed.

```
import doctest
def add(a, b):
    """
    Returns the sum of two numbers.
    >>> add(2, 3)
    5
    >>> add(-1, 1)
    0
    >>> add(0, 0)
    0
    >>> add(2.5, 3.5)
    6.0
    >>> add(-5, -10)
    -15
    """
    return a + b
if __name__ == "__main__":
    doctest.testmod()
```

17. Scope Resolution: object's namespace → class namespace → global namespace → built-in namespace.

```
species = "Global Species"
class Animal:
        species = "Class Species"
        def __init__ (self, species):
                self.species = species
        def display_species(self):
                print("Instance species:', self.species)
                print("Class species:", Animal.species)
                print("Global species:", globals()['species'])
a = Animal("Instance Species")
```

a.display_species()

What will be the output when the above program is executed? Explain the scope resolution process step by step.

Ans:- Scope Resolution Process:

Python follows a LEGB (Local → Enclosing → Global → Built-in) resolution order:

a. Instance Namespace (self.species): self.species = "Instance Species" sets the instance-specific species attribute. When self.species is accessed, Python first checks the object's namespace and finds "Instance Species".

b. Class Namespace (Animal.species): Animal.species = "Class Species" is a class-level attribute. If self.species did not exist, it would fall back to Animal.species. Animal.species is directly accessed in display_species(), printing "Class Species".

c. Global Namespace (species): species = "Global Species" is a global variable. globals()['species'] retrieves it explicitly, printing "Global Species".

Output:-

```
Instance species: Instance Species
Class species: Class Species
Global species: Global Species
```

18. Write a Python program using a lambda function to convert temperatures from Celsius to Kelvin, store the data in a tabular format using `pandas`, and visualize the data using a plot.

Ans:-

```
import pandas as pd
import matplotlib.pyplot as plt
celsius_to_kelvin = lambda c: c + 273.15
celsius_values = list(range(-10, 41, 5))
kelvin_values = [celsius_to_kelvin(c) for c in celsius_values]
df = pd.DataFrame({"Celsius (°C)": celsius_values, "Kelvin (K)": kelvin_values})
print(df)
plt.plot(df["Celsius (°C)"], df["Kelvin (K)"], marker="o", linestyle="-", color="b")
plt.xlabel("Celsius (°C)")
plt.ylabel("Kelvin (K)")
plt.title("Celsius to Kelvin Conversion")
plt.grid(True)
plt.show()
```

Output:-

	Celsius (°C)	Kelvin (K)
0	-10	263.15
1	-5	268.15
2	0	273.15
3	5	278.15
4	10	283.15
5	15	288.15
6	20	293.15
7	25	298.15
8	30	303.15
9	35	308.15
10	40	313.15