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?

```
print("Name: RISTI")

print("Regd. No.:

2241016101")

print("""Classes in Python define blueprints for creating objects.

They encapsulate data (attributes) and behaviour (methods).

Classes support OOP principles like encapsulation, inheritance, and polymorphism.

This makes code reusable, modular, and easier to maintain.""")
```

O/P:

Name: : RISTI

Regd. No.: 2241016101

Classes in Python define blueprints for creating objects.

They encapsulate data (attributes) and behaviour (methods).

Classes support OOP principles like encapsulation, inheritance, and polymorphism.

This makes code reusable, modular, and easier to maintain.

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

```
print("Name:      :      RISTI      ")

print("Regd. No.: 2241016101")

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("Deposit amount must be positive.")

  def withdraw(self, amount):
```

```
self.balance -= amount
            print(f"₹{amount} withdrawn successfully.")
        else:
            print("Insufficient balance or invalid amount.")
    def get_balance(self):
        print(f"Available balance: ₹{self.balance}")
account = BankAccount("Sibasis Mahapatra", 5000)
account.deposit(2000)
account.withdraw(1500)
account.get_balance()
```

O/P:

```
Name: : RISTI
Regd. No.: 2241016101
₹2000 deposited successfully.
₹1500 withdrawn successfully.
Available balance: ₹5500
```

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.

```
print("Name:      :      RISTI      ")
print("Regd. No.: 2241016101")
class Chapter:
    def _init_(self,  title,  page_count):
        self.title = title
        self.page_count = page_count
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.page_count for chapter in self.chapters)
book = Book("Python Programming")
book.add_chapter(Chapter("Introduction to Python", 30))
book.add_chapter(Chapter("Object-Oriented Programming", 50))
book.add_chapter(Chapter("Data Structures in Python", 40))
print(f"Total pages in '{book.title}': {book.total_pages()}")
```

O/P:

Name: : RISTI

Regd. No.: 2241016101

```
Total pages in 'Python Programming': 120
```

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

protected, and private attributes?

```
print("Name:   :  RISTI                ")
print("Regd. No.: 2241016101")
print("""Python enforces access control using naming conventions.
Public attributes (variable_name) are accessible from anywhere.
Protected attributes (_variable_name) are intended for internal use but accessible outside.
Private attributes (_variable_name) are name-mangled to prevent direct access.""")
```

O/P:

Name: : RISTI

Regd. No.: 2241016101

Python enforces access control using naming conventions. Public attributes (variable_name) are accessible from anywhere. Protected attributes (_variable_name) are intended for internal use but accessible outside. Private attributes (_variable_name) are name-mangled to prevent direct access.

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.

```
print("Name: : RISTI ")
print("Regd. No.:
2241016101") import re
class Time:
    def _init_(self, time_str):
        self.time_str = time_str
    def is_valid(self):
        # Validate if the time format is HH:MM:SS using regex
        pattern = r"^([01]?[0-9]|2[0-3]):([0-5]?[0-9]):([0-5]?[0-9])$"
        return bool(re.match(pattern, self.time_str))
    def convert_to_12hr(self):
        if not self.is_valid():
            return "Invalid time format."
        hours, minutes, seconds = map(int, self.time_str.split(":"))
        period = "AM" if hours < 12 else "PM"
        if hours == 0:
            hours = 12
        elif hours > 12:
            hours -= 12
        return f"{hours:02}:{minutes:02}:{seconds:02} {period}"
time_input = input("Enter time in HH:MM:SS format (24-hour): ")
time_obj = Time(time_input)
if time_obj.is_valid():
    print(f"Converted time in 12-hour format: {time_obj.convert_to_12hr()}")
else:
    print("Invalid time format. Please use HH:MM:SS.")
```

O/P:

```
Name: : RISTI

Regd. No.: 2241016101

Enter time in HH:MM:SS format (24-hour): 14:30:45
```

```
Converted time in 12-hour format: 02:30:45 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.

```
print("Name:      :      RISTI          ")

print("Regd. No.:

2241016101") class

BankAccount:

    def _init_(self, account_holder, initial_balance=0.0):

        self.account_holder = account_holder

        self._balance = initial_balance

    def deposit(self, amount):

        if amount > 0:

            self._balance += amount

            print(f"₹{amount} deposited successfully.")

        else:

            print("Deposit amount must be positive.")

    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 self._balance  # Method to access the private balance


account = BankAccount(": ABC ", 5000)

account.deposit(2000)

account.withdraw(1500)
```

```
print(f"Balance: ₹{account.get_balance()}")
```

O/P:

```
Name: : RISTI

Regd. No.: 2241016101

₹2000 deposited successfully.

₹1500 withdrawn successfully.

Balance: ₹5500
```

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.

```
print("Name: : RISTI ")

print("Regd. No.:

2241016101") import

random

class Card:

    def _init_(self, rank, suit):

        self.rank = rank

        self.suit = suit

    def _str_(self):

        return f"{self.rank} of {self.suit}"

class Deck:

    def _init_(self):

        self.cards = []

        self.create_deck()

        self.shuffle()

    def create_deck(self):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']

        ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

        for suit in suits:

            for rank in ranks:

                self.cards.append(Card(rank, suit))

    def shuffle(self):

        random.shuffle(self.cards)

    def deal(self, num_players, cards_per_player):

        hands = {f"Player {i+1}": [] for i in range(num_players)}

        for player in hands:

            for _ in range(cards_per_player):

                if self.cards:

                    hands[player].append(self.cards.pop())

        return hands

class Player:

    def _init_(self, name):

        self.name = name

        self.hand = []

    def receive_card(self, card):

        self.hand.append(card)

    def show_hand(self):

        return [str(card) for card in self.hand]

def play_game():

    # Create deck and players

    deck = Deck()

    players = [Player("Alice"), Player("Bob"), Player("Charlie")]

    hands = deck.deal(num_players=len(players), cards_per_player=5)

    for i, player in enumerate(players):

        player.hand = hands[f"Player {i+1}"]

        print(f"{player.name}'s hand: {', '.join(player.show_hand())}")

play_game()
```

O/P:

```
Name:: RISTI

Regd. No.: 2241016101

Alice's hand: Ace of Spades, 10 of Hearts, 2 of Diamonds, 7 of Clubs, Jack of Spades

Bob's hand: Queen of Hearts, 5 of Diamonds, 3 of Spades, 8 of Hearts, 2 of Spades

Charlie's hand: King of Spades, 9 of Diamonds, 4 of Clubs, Ace of Hearts, 6 of Spades
```

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 attribute

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.

```
print("Name: RISTI ")

print("Regd. No.:

2241016101") class Vehicle:

  def _init_(self, make, model):

    self.make = make

    self.model = model

  def display_info(self):

    print(f"Make: {self.make}")

    print(f"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")

print("Vehicle Info:")

vehicle.display_info()

car = Car("Honda", "Civic", 4)
```

```
print("\nCar Info:")

car.display_info()
```

O/P:

```
Name: : RISTI

Regd. No.: 2241016101

Vehicle Info:

Make: Toyota

Model: Corolla

Car Info:

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.**

```
print("Name: : RISTI")

print("Regd. No.:

2241016101") import

math

class Shape:

    def area(self):

        pass

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, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width
shapes = [Circle(5), Rectangle(4, 6)]
for shape in shapes:
    print(f"Area: {shape.area():.2f}")
```

O/P:

Name: RISTI

Regd. No.: 2241016101

```
Area: 78.54
Area: 24.00
```

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.

```
print("Name: : RISTI ")
print("Regd. No.:
2241016101") class
CommissionEmployee:
    def _init_(self, name, employee_id, sales_amount, commission_rate):
        if sales_amount < 0 or commission_rate < 0:
            raise ValueError("Sales amount and commission rate cannot be negative."
        self.name = name
        self.employee_id = employee_id
        self.sales_amount = sales_amount
        self.commission_rate = commission_rate
    def earnings(self):
        return self.sales_amount * self.commission_rate
    def _repr_(self):
        return (f"CommissionEmployee(Name: {self.name}, ID: {self.employee_id}, "
                f"Sales: {self.sales_amount}, Commission Rate: {self.commission_rate}, "
                f"Earnings: {self.earnings():.2f})")
try:
    emp = CommissionEmployee(": RISTI ", 2241016101, 50000, 0.1) print(emp)
    emp.sales_amount = 60000
    print(f"Updated Earnings: {emp.earnings():.2f}")
    emp.sales_amount = -10000


except ValueError as e:
    print(f"Error: {e}")
```

O/P:

Name: RISTI

Regd. No.: 2241016101

CommissionEmployee(Name: : DEEPESH, ID: **2241011126**, Sales: 50000, Commission Rate: 0.1, Earnings: 5000.00)

Updated Earnings: 6000.00

Error: Sales amount and commission rate cannot be negative.

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.

```
        print("Name: : RISTI ")
 print("Regd. No.: 2241016101")
print("""Duck typing in Python is a concept where an object's behavior determines its type rather than its inheritance.
If an object has the required method, it can be used without checking its class.""")
class Dog:
    def speak(self):
        print("Dog says: Woof!")
        class Robot:

    def speak(self):
        print("Robot says: Beep Boop!")

def describe(entity):

    entity.speak()

print("\nExample of Duck Typing:")

d = Dog()

r  =  Robot()

describe(d)

describe(r)
```

O/P:

Name: : RISTI

Regd. No.: 2241016101

Duck typing in Python is a concept where an object's behavior determines its type rather than its inheritance.

If an object has the required method, it can be used without checking its class.

Example of Duck Typing:

Dog says: Woof!

Robot says: Beep Boop!

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

class?

```
print("Name:     RISTI     ")

print("Regd. No.:

2241016101") class

Complex:

    def _init_(self, r, i): self.r, self.i = r, i

    def _add_(self, o): return Complex(self.r + o.r, self.i + o.i)

    def _repr_(self): return f"{self.r} + {self.i}i"

print("Sum:", Complex(3, 4) + Complex(1, 2))
```

O/P:

Name: : RISTI

```
Regd. No.: 2241016101
Sum: 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?

```
print("Name: : RISTI ")
print("Regd. No.: 2241016101")
class InsufficientFunds(Exception):
    def _init_(self, b, w): super()._init_(f"Balance: {b}, Withdrawal: {w}")
class Bank:
    def _init_(self, b): self.b = b
    def withdraw(self, a):
        if a > self.b: raise InsufficientFunds(self.b, a)
        self.b -= a; return self.b
try:  Bank(500).withdraw(600)
except Exception as e: print(e)
```

O/P:

```
Name: : RISTI
Regd. No.: 2241016101
Balance: 500, Withdrawal: 600
```

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.

```
print("Name: : RISTI ")
print("Regd. No.:
2241016101") import
random
from dataclasses import dataclass
@dataclass
 class Card:
    suit: str
   rank: str
class Deck:
   suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
   ranks = [str(i) for i in range(2, 11)] + ["J", "Q", "K", "A"]
   def _init_(self):
       self.cards = [Card(suit, rank) for suit in self.suits for rank in self.ranks]
       random.shuffle(self.cards)
   def deal(self, n):
       return [self.cards.pop() for _ in range(n)]
d = Deck()
hand = d.deal(5)
print("Hand:", hand)
print("Remaining cards in deck:", len(d.cards))
```

O/P:

Name: : RISTI

Regd. No.: 2241016101

```
Hand: [Card(suit='Diamonds', rank='5'), Card(suit='Clubs', rank='A'), Card(suit='Hearts',
rank='J'),
Card(suit='Spades', rank='9'), Card(suit='Diamonds', rank='K')]
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.

```
print("Name: : RISTI ") print("Regd.
No.: 2241016101")
print("""Python data classes are more flexible than named tuples because they support
default values,
mutable fields, methods, and modification after creation.""")
from collections import namedtuple
from dataclasses import dataclass
PersonNT = namedtuple("PersonNT", ["name", "age"])
```

```
@dataclass
class Person:
  name: str
    age: int
    city: str = "Unknown"
print("Named Tuple:", PersonNT("Alice", 25))
print("Data Class:", Person("Bob", 30, "New York"))
```

O/P:

Name: : RISTI

Regd. No.: 2241016101

Python data classes are more flexible than named tuples because they support default values,

mutable fields, methods, and modification after creation.

```
Named Tuple: PersonNT(name='Alex', age=25)
Data Class: Person(name='Borax', age=30, city='New York')
```

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.

```
print("Name: RISTI ")
print("Regd. No.:
2241016101")
print("""The doctest module allows writing test cases inside a function's docstring.
These tests run automatically when the script is executed.""")
import doctest
def add(a, b):
    """
    Returns the sum of two numbers.
    >>> add(2, 3)
    5
    >>> add(-1, 1)

    0

    >>> add(0, 0)

    0
    """

    return a + b

if _name____== "_main_":

    doctest.testmod()

    print("All tests passed if no output is shown.")
```

O/P:

```
Name: : RISTI

Regd. No.: 2241016101
Tests passed!
```

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.

```
print("Name: : RISTI ")

print("Regd. No.: 2241016101")
```

O/P:

Name:

RISTI

Regd. No.:

2241016101

```
"""
    When a.display_species() is called, Python first looks for the species attribute in the instance (self.species).
    If it’s not found, it looks for species in the class (Animal.species).
    If still not found, Python looks for it in the global namespace (globals()['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.

```
print("Name: RISTI")
print("Regd. No.:
2241016101") import
pandas as pd
df = pd.DataFrame({'Celsius': [0, 20, 40, 60, 80]})
df['Kelvin'] = df['Celsius'] + 273.15
print(df)
```

O/P:

```
Name: : RISTI

Regd. No.: 2241016101
   Celsius  Kelvin
0        0  273.15
1       20  293.15
2       40  313.15
3       60  333.15
4       80  353.15
```