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

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

**Question – 1**

**What is the significance of classes in Python programming, and how do they contribute to objectoriented programming?**

In Python, classes are a key feature of Object-Oriented Programming (OOP), allowing for the creation of objects that bundle data (attributes) and methods (functions). They contribute to OOP in the following ways:-> Encapsulation: Groups related data and methods, hiding internal details while providing controlled access.-> Abstraction: Simplifies complex systems by exposing only relevant functionalities to the user.-> Inheritance: Enables reuse by allowing a new class to inherit attributes and methods from an existing one.-> Polymorphism: Allows different classes to have methods with the same name but different implementations.-> Modularity: Promotes reusable, maintainable code by organizing functionality into separate classes.Thus, classes help in structuring code, enhancing readability, and fostering reusability in software development.

**Question – 2**

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

from decimal import Decimal as decclass Account: def \_\_init\_\_(self, name, balance): self.\_name = name self.\_balance = balance @property def name(self): return self.\_name @property def balance(self): return self.\_balance def deposit(self, amount): if amount <= dec('0.00'): raise ValueError("Amount must be positive.\n") else: self.\_balance += amount print(f"Deposited {amount}$.\nUpdated balance: {self.balance}\n") def withdraw(self, amount): if amount <= dec('0.00'): raise ValueError("Amount must be positive.\n") elif amount > self.\_balance: raise ValueError("Insufficient Balance.\n") else: self.\_balance -= amount print(f"Withdrawn {amount}$.\nUpdated balance: {self.balance}\n") def \_\_str\_\_(self): return f"Account holder: {self.name}\nBalance: {self.balance}\n"account = Account("Debadarshi Omkar", 10000)print(account)account.deposit(5000)account.withdraw(2000)print(account)

**Question – 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.**

class Chapter: def \_\_init\_\_(self, title, pages): self.title = title self.pages = pagesclass Book: def \_\_init\_\_(self, title, chapters): self.title = title self.chapters = chapters def total\_pages(self): return sum(chapter.pages for chapter in self.chapters)book\_chapters = [ Chapter("Chapter 0", 10), Chapter("Chapter 1", 25), Chapter("Chapter 2", 50) ]book = Book("The Debadarshi", book\_chapters)print(f"Total pages in the book '{book.title}': {book.total\_pages()}")

**Question – 4**

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

In Python, access control to class attributes is enforced using naming conventions rather than strict enforcement like some other programming languages. Python uses public, protected, and private attribute conventions, which are not strictly enforced by the language but rely on the programmer’s adherence to these naming conventions.-> Public Attributes: Accessible from anywhere (inside and outside the class). No leading underscore (attribute).-> Protected Attributes: Meant for internal use within the class and subclasses, not to be accessed directly outside the class. Single leading underscore (\_attribute).-> Private Attributes: Intended for internal use only within the class, and Python name-mangles them to make access harder outside the class. Double leading underscore (\_\_attribute).

**Question – 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.**

import reclass Time: def \_\_init\_\_(self, time\_str): self.time\_str = time\_str self.hours, self.minutes, self.seconds = 0, 0, 0 if self.is\_valid\_time(): self.hours, self.minutes, self.seconds = map(int, self.time\_str.split(":")) else: self.hours, self.minutes, self.seconds = None, None, None def is\_valid\_time(self): time\_pattern = r'^[0-2][0-9]:[0-5][0-9]:[0-5][0-9]$' if re.match(time\_pattern, self.time\_str): hours, minutes, seconds = map(int, self.time\_str.split(":")) if 0 <= hours < 24 and 0 <= minutes < 60 and 0 <= seconds < 60: return True return False def convert\_to\_12\_hour\_format(self): if self.hours is None: return "Invalid time format" period = "AM" if self.hours >= 12: period = "PM" if self.hours > 12: self.hours -= 12 elif self.hours == 0: self.hours = 12 return f"{self.hours:02d}:{self.minutes:02d}:{self.seconds:02d} {period}"time\_input = input("Enter time in HH:MM:SS format (24-hour): ")time\_obj = Time(time\_input)if time\_obj.is\_valid\_time(): print("Converted time in 12-hour format:", time\_obj.convert\_to\_12\_hour\_format())

else: print("Invalid time format. Please enter time in the correct HH:MM:SS format.")

**Question – 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.**

class BankAccount: def \_\_init\_\_(self, owner, balance=0.0): self.owner = owner self.\_\_balance = balance @property def balance(self): return self.\_\_balance def deposit(self, amount): if amount > 0: self.\_\_balance += amount print(f"Deposited ${amount:.2f}. Updated balance: ${self.\_\_balance:.2f}") else: print("Deposit amount must be positive.") def withdraw(self, amount): if amount > 0 and amount <= self.\_\_balance: self.\_\_balance -= amount print(f"Withdrew ${amount:.2f}. Updated balance: ${self.\_\_balance:.2f}") elif amount > self.\_\_balance: print("Insufficient balance.") else: print("Withdraw amount must be positive.") def display\_balance(self): print(f"Balance: ${self.balance:.2f}")account = BankAccount("Debadarshi Omkar", 15000.0)account.display\_balance()account.deposit(500)account.withdraw(2000)account.display\_balance()Using private attributes in a class improves data security and prevents accidental modifications in the following ways:-> Data Security: Private attributes ensure that critical data, like a bank account balance, is not directly accessible or modifiable from outside the class. This reduces the risk of unauthorized changes.-> Controlled Access: Access to private attributes is restricted to specific methods, which allows the class to enforce rules -> Prevents Accidental Changes: Direct manipulation of private data is prevented, reducing the risk of errors or bugs due to invalid values being set.-> Encapsulation: Private attributes are part of the principle of encapsulation, which hides the internal details of the class and exposes only necessary functionality, improving code maintainability.

**Question – 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.**

import randomclass Card: def \_\_init\_\_(self, suit, rank): self.suit = suit self.rank = rank def \_\_repr\_\_(self): return f"{self.rank} of {self.suit}"class Deck: def \_\_init\_\_(self): self.suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades'] self.ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A'] self.cards = [Card(suit, rank) for suit in self.suits for rank in self.ranks]

def shuffle(self): random.shuffle(self.cards) print("Deck shuffled!") def deal(self, num\_players, cards\_per\_player): hands = {f"Player {i+1}": [] for i in range(num\_players)} for i in range(cards\_per\_player): for player in hands: hands[player].append(self.cards.pop()) return handsclass Player: def \_\_init\_\_(self, name): self.name = name self.hand = [] def receive\_cards(self, cards): self.hand.extend(cards) def show\_hand(self): return f"{self.name}'s hand: {', '.join(str(card) for card in self.hand)}"def main(): deck = Deck() deck.shuffle() num\_players = 4 players = [Player(f"Player {i+1}") for i in range(num\_players)] hands = deck.deal(num\_players, 5) for i, player in enumerate(players): player.receive\_cards(hands[f"Player {i+1}"]) print(player.show\_hand())if \_\_name\_\_ == "\_\_main\_\_": main()

**Question – 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.**

class Vehicle: def \_\_init\_\_(self, make, model): self.make = make self.model = model def display\_info(self): print(f"Vehicle Make: {self.make}") print(f"Vehicle 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("Maruti", "Alto")vehicle.display\_info()print("\n")car = Car("Maruti", "800", 4)car.display\_info()

**Question – 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.**

import mathclass Shape: def area(self): passclass Circle(Shape): def \_\_init\_\_(self, radius): self.radius = radius def area(self): return math.pi \* self.radius \*\* 2class Rectangle(Shape): def \_\_init\_\_(self, width, height): self.width = width self.height = height def area(self): return self.width \* self.height

circle = Circle(5)rectangle = Rectangle(4, 6)print(f"Area of Circle: {circle.area()}")print(f"Area of Rectangle: {rectangle.area()}")Polymorphism allows different classes to define the same method in their own way. In this example, both Circle and Rectangle classes override the area() method, but each calculates the area differently. This enables calling the area() method on any object of type Shape without worrying about its specific class. Polymorphism simplifies code by allowing objects of different shapes to be treated uniformly, making the program more flexible for future shape types.

**Question – 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.**

class CommissionEmployee: def \_\_init\_\_(self, first\_name, last\_name, sales, commission\_rate): self.first\_name = first\_name self.last\_name = last\_name self.sales = sales self.commission\_rate = commission\_rate @property def first\_name(self): return self.\_first\_name @first\_name.setter def first\_name(self, value): if not value.isalpha(): raise ValueError("First name must contain only alphabetic characters.") self.\_first\_name = value @property def last\_name(self): return self.\_last\_name @last\_name.setter def last\_name(self, value): if not value.isalpha(): raise ValueError("Last name must contain only alphabetic characters.") self.\_last\_name = value @property def sales(self): return self.\_sales @sales.setter def sales(self, value): if value < 0: raise ValueError("Sales cannot be negative.") self.\_sales = value @property def commission\_rate(self): return self.\_commission\_rate @commission\_rate.setter def commission\_rate(self, value): if value < 0: raise ValueError("Commission rate cannot be negative.") self.\_commission\_rate = value def earnings(self): return self.sales \* self.commission\_rate def \_\_repr\_\_(self): return f"CommissionEmployee({self.first\_name}, {self.last\_name}, Sales: {self.sales}, Commission Rate: {self.commission\_rate})"try: employee = CommissionEmployee("John", "Doe", 5000, 0.10) print(f"Earnings of {employee.first\_name} {employee.last\_name}: ${employee.earnings()}") employee.sales = 6000 print(f"Updated Earnings of {employee.first\_name} {employee.last\_name}: ${employee.earnings()}") employee.sales = -1000except ValueError as e: print(f"Error: {e}")try: employee.commission\_rate = -0.05except ValueError as e: print(f"Error: {e}")

**Question – 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.**

Duck Typing -> Duck typing is a programming concept where an object's suitability for a specific purpose is determined by whether it implements the required methods and properties, rather than by checking its type or class. In Python, this allows flexibility in using objects with similar behaviors, even if they belong to different classes.class Dog: def speak(self): return "Woof!"class Robot: def speak(self): return "Beep boop!"def describe(animal): print(animal.speak())dog = Dog()robot = Robot()describe(dog)describe(robot)Duck typing allows the describe() function to work without checking the object's type because it only cares if the object has a speak() method. As long as the object can "speak" (i.e., has a speak() method), the function works, regardless of the object's class or type. This flexibility is key to duck typing, where behavior, not type, determines compatibility.

**Question – 12**

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

class Complex: def \_\_init\_\_(self, real, imag): self.real = real self.imag = imag def \_\_add\_\_(self, other): real\_sum = self.real + other.real imag\_sum = self.imag + other.imag return Complex(real\_sum, imag\_sum) def \_\_str\_\_(self): return f"{self.real} + {self.imag}i"complex1 = Complex(3, 4)complex2 = Complex(1, 2)result = complex1 + complex2print(f"Sum of complex numbers: {result}")

**Question – 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?**

class InsufficientFundsError(Exception): def \_\_init\_\_(self, balance, withdrawal\_amount): self.balance = balance self.withdrawal\_amount = withdrawal\_amount def \_\_str\_\_(self): return f"Insufficient funds! Current balance: {self.balance}, Withdrawal attempt: {self.withdrawal\_amount}"class BankAccount: def \_\_init\_\_(self, balance): self.balance = balance def withdraw(self, amount): if amount > self.balance: raise InsufficientFundsError(self.balance, amount) else: self.balance -= amount print(f"Withdrawal successful! Remaining balance: {self.balance}")account = BankAccount(500)try: account.withdraw(600)except InsufficientFundsError as e: print(e)

**Question – 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.**

import randomfrom dataclasses import dataclass@dataclassclass Card: rank: str suit: strclass Deck: def \_\_init\_\_(self): self.cards = [ Card(rank, suit) for suit in ['Hearts', 'Diamonds', 'Clubs', 'Spades'] for rank in ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A'] ] random.shuffle(self.cards) def deal(self, num\_cards): hand = [self.cards.pop() for \_ in range(num\_cards)] return hand def remaining\_cards(self): return len(self.cards)def print\_hand(hand): for card in hand: print(f"{card.rank} of {card.suit}")deck = Deck()player\_hand = deck.deal(5)print("Player's hand:")print\_hand(player\_hand)print(f"\nRemaining cards in the deck: {deck.remaining\_cards()}")

**Question – 15**

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

from collections import namedtuplefrom dataclasses import dataclassPersonNT = namedtuple('PersonNT', ['name', 'age'])person\_nt = PersonNT(name="Alice", age=30)try: person\_nt.age = 31except AttributeError as e: print(f"NamedTuple Error: {e}")@dataclassclass PersonDC: name: str age: int city: str = "Unknown"person\_dc = PersonDC(name="Bob", age=25)person\_dc.age = 26person\_dc.city = "New York"@dataclassclass PersonWithMethod: name: str age: int city: str def greet(self): return f"Hello, my name is {self.name} and I am {self.age} years old from {self.city}."person\_method = PersonWithMethod(name="Charlie", age=22, city="Los Angeles")print(person\_method.greet())print(f"Data Class Person: {person\_dc}")Advantages of Python Data Classes over Named Tuples :-Python data classes (introduced in Python 3.7) offer advantages over namedtuple, making them more flexible. Mutability: Data Classes: Instances are mutable, allowing modification of attributes after creation. Named Tuples: Are immutable, restricting modification of attributes.Default Values: Data Classes: Support default values for attributes directly in the class definition. Named Tuples: Do not support default values easily, requiring additional methods.Built-in Methods: Data Classes: Automatically generate methods like \_\_repr\_\_, \_\_eq\_\_, and allow adding custom methods. Named Tuples: Limited methods, and no support for adding custom methods.Type Hints: Data Classes: Fully support type hints for attributes, improving code clarity and static type checking. Named Tuples: Do not natively support type hints.

**Question – 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.**

import doctestdef add(a, b): """ Returns the sum of two numbers. >>> add(2, 3) 5 >>> add(-1, 1) 0 >>> add(0, 0) 0 >>> add(10, 20) 30 """ return a + bif \_\_name\_\_ == "\_\_main\_\_": print(doctest.testmod())

**Question – 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 displayspecies(self): print('Instance species: ', self.species) print('Class species: ', Animal.species) print('Global species: ', globals()['species'])a = Animal('Instance Species')a.displayspecies()**

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

Instance species: Instance Species

Class species: Class Species

Global species: Global Species

**Question – 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**.

import pandas as pdimport matplotlib.pyplot as pltcelsius\_temperatures = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]celsius\_to\_kelvin = lambda celsius: celsius + 273.15kelvin\_temperatures = list(map(celsius\_to\_kelvin, celsius\_temperatures))temperature\_df = pd.DataFrame({ 'Celsius': celsius\_temperatures, 'Kelvin': kelvin\_temperatures})print("Temperature Conversion Table:")print(temperature\_df)plt.figure(figsize=(8, 6))plt.plot(temperature\_df['Celsius'], temperature\_df['Kelvin'], marker='o', linestyle='-', color='b')plt.title("Celsius to Kelvin Conversion")plt.xlabel("Temperature (Celsius)")plt.ylabel("Temperature (Kelvin)")plt.grid(True)plt.show()