ASSIGNMENT – 1

Q1) What is the significance of classes in Python programming, and how do they contribute to object-oriented programming?

Ans:

Significance of classes in Python programming:

1. They are the building blocks of object-oriented programming (OOP) in Python.
2. Allow us to create objects with properties and behaviours.
3. Serve as templates for creating objects, which are instances of a class.
4. Help us model real-world entities in our code.
5. Organize our code, improve reusability, and maintainability.

Contribution to object-oriented programming:

1. Encapsulation: Classes allow us to encapsulate data and behaviour into a single unit.
2. Inheritance: Classes support inheritance, allowing us to create new classes based on existing ones.
3. Polymorphism: Classes enable polymorphism, allowing us to use objects of different classes interchangeably.
4. Abstraction: Classes help us abstract away complex details, making it easier to work with complex systems.

Q2) Create a custom Python class for managing a bank account with basic functionalities like deposit and withdrawal?

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class BankAccount:
    def __init__(self, AC_no, bal = 0.00):
        self.AC_no = AC_no
        self.bal = bal

    def deposit(self, amt):
        self.bal += amt
        print(f"Deposited Rs.{amt} successfully!!")

    def withdraw(self, amt):
        if amt > self.bal:
            print("Insufficient funds!!")
            print(f"Unable to withdraw Rs.{amt}!!")
        else:
            self.bal -= amt
            print(f"Withdrawn Rs.{amt} successfully!!")
```

```
    def display(self):
        print(f"Account Number: {self.AC_no}")
        print(f"Balance = Rs.{self.bal:.2f}\n")

a1 = BankAccount(2049, 10)
a1.deposit(1500)
a1.display()
a1.withdraw(500)
a1.display()
a1.withdraw(2000)
a1.display()
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Deposited Rs.1500 successfully!!
Account Number: 2049
Balance = Rs.1510.00

Withdrawn Rs.500 successfully!!
Account Number: 2049
Balance = Rs.1010.00

Insufficient funds!!
Unable to withdraw Rs.2000!!
Account Number: 2049
Balance = Rs.1010.00
```

Q3) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class Chapter:
    def __init__(self, title, pcount):
        self.title = title
        self.pcount = pcount

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.chapters = []
```

```
    def add_chapter(self, chapter):
        self.chapters.append(chapter)

    def display(self):
        print(f"Title: {self.title}, Author: {self.author}")
        print("\nChapters:-")
        for i in self.chapters:
            print(f"Title: {i.title}, Page Count:
{i.pcount}")
        print(f"Total Page Count = {sum([i.pcount for i in
self.chapters])}")

b1 = Book("Harry Potter", "J.K. Rowling")
b1.add_chapter(Chapter("Sorcere's Stone", 200))
b1.add_chapter(Chapter("Chamber of Secrets", 300))
b1.add_chapter(Chapter("Prisoner of Azkaban", 400))
b1.display()
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

Title: Harry Potter, Author: J.K. Rowling

Chapters:-
Title: Sorcere's Stone, Page Count: 200
Title: Chamber of Secrets, Page Count: 300
Title: Prisoner of Azkaban, Page Count: 400
Total Page Count = 900

Q4) 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 to class attributes by using the following naming conventions: -

1. Public attributes:
   - Directly accessible from outside the class using dot notation.
   - Defined without any underscores before the attribute name.
2. Protected attributes:
   - Accessible within the class and its subclasses.
   - Defined using a single underscore before the attribute name.
3. Private attributes:
   - Accessible only within the class, but can be accessed outside using getter and setter methods.

- To access them, name mangling can be used by using the format: **_ClassName__private_attr.**
- Defined using a double underscore before the attribute name.

Example Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class MyClass:
    def __init__(self):
        self.public_attr = "Public attribute"
        self._protected_attr = "Protected attribute"
        self.__private_attr = "Private attribute"

    def get_private_attr(self):
        return self.__private_attr

obj = MyClass()
print(obj.public_attr)
print(obj._protected_attr)
# print(obj.__private_attr)
print(obj._MyClass__private_attr)
print(obj.get_private_attr())
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Public attribute
Protected attribute
# AttributeError: 'MyClass' object has no attribute '__private_attr'.
Private attribute
Private attribute
```

Q5) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class Time:
    def __init__(self, timeStr):
        self.timeStr = timeStr
        self.validateTime()
        self.convertTo12Hour()
```

```
    def validateTime(self):
        try:
            hours, minutes, seconds = map(int,
self.timeStr.split(':'))
            if hours < 0 or hours > 23 or minutes < 0 or
minutes > 59 or seconds < 0 or seconds > 59:
                raise ValueError
            else:
                self.hours = hours
                self.minutes = minutes
                self.seconds = seconds
        except ValueError:
            print("Invalid time. Please enter valid time in
HH:MM:SS format.")
            exit()

    def convertTo12Hour(self):
        if self.hours == 0:
            self.hours = 12
            self.period = "AM"
        elif self.hours < 12:
            self.period = "AM"
        elif self.hours == 12:
            self.period = "PM"
        else:
            self.hours -= 12
            self.period = "PM"
        self.timeStr =
f"{self.hours:02d}:{self.minutes:02d}:{self.seconds:02d}
{self.period}"

timeStr = input("Enter time in 24-hour format (HH:MM:SS): ")
time = Time(timeStr)
print(f"Time in 12-hour format: {time.timeStr}")
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

Enter time in 24-hour format (HH:MM:SS): 16:45:10
Time in 12-hour format: 04:45:10 PM

Name: Sidhanta Barik, RegNo: 2241002049

Enter time in 24-hour format (HH:MM:SS): 29:45:65
Invalid time. Please enter valid time in HH:MM:SS format.

Q6) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class BankAccount:
    def __init__(self, AC_no, bal = 0.0):
        self.AC_no = AC_no
        self.__bal = bal

    def deposit(self, amt):
        self.__bal += amt
        print(f"Deposited Rs.{amt} successfully!!")

    def withdraw(self, amt):
        if amt > self.__bal:
            print("Insufficient funds!!")
            print(f"Unable to withdraw Rs.{amt}!!")
        else:
            self.__bal -= amt
            print(f"Withdrawn Rs.{amt} successfully!!")

    def display(self):
        print(f"Account Number: {self.AC_no}")
        print(f"Balance = Rs.{self.__bal:.2f}\n")

ac1 = BankAccount(2049, 5000.75)
ac1.deposit(500.5)
ac1.display()
ac1.withdraw(2000)
ac1.display()
ac1.__bal = 1000 # This will not modify the balance
attribute as it is a private attribute
ac1.display()
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Account Number: 2049
Balance = Rs.5000.75
```

Deposited Rs.500.5 successfully!!
Account Number: 2049
Balance = Rs.5501.25

Withdrawn Rs.2000 successfully!!
Account Number: 2049
Balance = Rs.3501.25

Account Number: 2049
Balance = Rs.3501.25

Explanation: Using private attributes can help improve data security and prevent accidental modifications by restricting direct access to the attributes from outside the class. This ensures that the attributes can only be accessed and modified through the class methods, which allows for better control over the data and prevents unauthorized changes. It also enforces data hiding and protect the internal state of the class from external interference, improving the overall security and integrity of the program.

Q7) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
import random

class Card:
    def __init__(self, suit, value):
        self.suit = suit
        self.value = value

    def show(self):
        return f"{self.value} of {self.suit}"

class Deck:
    suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
    values = ["Ace", "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King"]
    def __init__(self):
```

```
        self.cards = [Card(suit, value) for suit in
self.suits for value in self.values]

    def shuffle(self):
        random.shuffle(self.cards)

    def deal(self):
        return self.cards.pop() if self.cards else None

class Player:
    def __init__(self, name):
        self.name = name
        self.hand = []

    def draw(self, deck, num_cards):
        for _ in range(num_cards):
            card = deck.deal()
            if card:
                self.hand.append(card)
            else:
                print("No more cards to draw!!")
                break

    def show_hand(self):
        print(f"{self.name}'s hand: {', '.join([i.show() for
i in self.hand])}")

deck = Deck()
deck.shuffle()
players = [Player(f"Player {i+1}") for i in range(4)]
num_cards = 3
for player in players:
    player.draw(deck, num_cards)
    player.show_hand()
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

Player 1's hand: 2 of Diamonds, 4 of Clubs, 2 of Hearts
Player 2's hand: 2 of Clubs, 9 of Clubs, 5 of Spades
Player 3's hand: King of Spades, 5 of Hearts, Jack of Spades
Player 4's hand: 6 of Diamonds, King of Diamonds, 9 of Spades

---

Q8) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
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}')

v1 = Vehicle('Lexus', 'LFA')
v1.display_info()
print()
c1 = Car('Lexus', 'LX', 5)
c1.display_info()
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Make: Lexus, Model: LFA

Make: Lexus, Model: LX
Number of doors: 5
```

Explanation: -

- Car class inherits Vehicle class and extends it by adding an additional attribute "num_doors".
- Car class overrides display_info() method of Vehicle class to display additional attribute num_doors.
- When display_info() method of Car class is called, it first calls the display_info() method of the Vehicle class using super() method to display make and model attributes, and then it displays the num_doors attribute of the Car class.

Q9) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
import math
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return math.pi*self.r**2

class Rectangle(Shape):
    def __init__(self, l, b):
        self.l = l
        self.b = b

    def area(self):
        return self.l*self.b

c1 = Circle(6)
print(f'Area of circle: {c1.area()}')
r1 = Rectangle(4, 5)
print(f'Area of rectangle: {r1.area()}')
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Area of circle: 113.09733552923255
Area of rectangle: 20
```

Explanation: Polymorphism is the ability of an object to take on many forms. It simplifies working with different shapes in an inheritance hierarchy by allowing objects of different classes to be treated as objects of a common superclass. In this case, the base class Shape has a method area() that is overridden by the subclasses Circle and Rectangle. When the area() method is called on objects of the Circle and Rectangle

classes, the appropriate implementation of the area() method is executed based on the type of the object. This allows for a more flexible and extensible design, as new shapes can be added to the hierarchy without modifying the existing code.

Q10) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class CommissionEmployee:
    def __init__(self, fName, lName, sales, cRate):
        self.fName = fName
        self.lName = lName
        self.sales = sales
        self.cRate = cRate

    @property
    def sales(self):
        return self._sales
    @sales.setter
    def sales(self, sales):
        if sales < 0:
            raise ValueError("Sales can't be negative.")
        self._sales = sales

    @property
    def cRate(self):
        return self._cRate
    @cRate.setter
    def cRate(self, cRate):
        if cRate < 0:
            raise ValueError("Commission rate can't be
negative.")
        self._cRate = cRate

    @property
    def earnings(self):
        return self.sales * self.cRate

    def __repr__(self):
        return f'{self.fName} {self.lName}'
```

```
def test():
    emp = CommissionEmployee('Sidhanta', 'Barik', 10000,
0.6)
    print(emp)
    print(f'Earnings: Rs.{emp.earnings:.2f}')
    try:
        emp.sales = -1000
    except ValueError as e:
        print(e)
    try:
        emp.cRate = -0.05
    except ValueError as e:
        print(e)
    emp.sales = 15000
    emp.cRate = 0.8
    print(f'Earnings after modification:
Rs.{emp.earnings:.2f}')

test()
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Sidhanta Barik
Earnings: Rs.6000.00
Sales can't be negative.
Commission rate can't be negative.
Earnings after modification: Rs.12000.00
```

Q11) 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: Duck typing is a concept related to dynamic typing, where the type or class of an object is determined by the presence of certain methods and properties, rather than the actual type of the object itself. In other words, if an object behaves like a duck (i.e., it has a quack() method), then it is a duck.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')

def describe(obj):
    obj.speak()
```

```
class Dog:
    def speak(self):
        print('Bhow Bhow')

class Robot:
    def speak(self):
        print('Beep Bop')

d1 = Dog()
r1 = Robot()
describe(d1)
describe(r1)
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

Bhow Bhow

Beep Bop

Q12) WAP to overload the + operator to perform addition of two complex numbers using a custom Complex class?

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, c):
        return Complex(self.real + c.real, self.imag +
c.imag)

    def __str__(self):
        return f'{self.real} + {self.imag}i'

c1 = Complex(2, 3)
c2 = Complex(4, 5)
c3 = c1 + c2
print(f"({c1}) + ({c2}) = {c3}")
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

(2 + 3i) + (4 + 5i) = 6 + 8i

Q13) WAP to create a custom exception class in Python that displays the balance and withdrawal amount when an error occurs due to insufficient funds?

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
class InsufficientFundsException(Exception):
    def __init__(self, bal, amt):
        self.bal = bal
        self.amt = amt
        super().__init__(f"Insufficient balance. Current
balance: Rs.{self.bal}, Withdrawal amount: Rs.{self.amt}")

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

    def withdraw(self, amt):
        if self.bal < amt:
            raise InsufficientFundsException(self.bal, amt)
        self.bal -= amt
        print(f"Withdrawal of Rs.{amt} successful. Current
balance: Rs.{self.bal}")

    def deposit(self, amt):
        self.bal += amt
        print(f"Deposit of Rs.{amt} successful. Current
balance: Rs.{self.bal}")

try:
    acc = BankAccount(1000)
    acc.withdraw(200)
    acc.withdraw(1200)
except InsufficientFundsException as e:
    print(e)
```

OUTPUT: -

```
Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------
Withdrawal of Rs.200 successful. Current balance: Rs.800
Insufficient balance. Current balance: Rs.800, Withdrawal amount: Rs.1200
```

Q14) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
import random
class Card:
    def __init__(self, suit, value):
        self.suit = suit
        self.value = value

    def __str__(self):
        return f'{self.value} of {self.suit}'

class Deck:
    def __init__(self):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        values = ['2', '3', '4', '5', '6', '7', '8', '9',
'10', 'Jack', 'Queen', 'King', 'Ace']
        self.cards = [Card(suit, value) for suit in suits
for value in values]
        random.shuffle(self.cards)

    def deal(self, n):
        if n > len(self.cards):
            raise ValueError('Not enough cards in deck')
        else:
            hand = self.cards[:n]
            self.cards = self.cards[n:]
            return hand

deck = Deck()
hand = deck.deal(5)
print(f"Player's Hand: {", ".join([str(card) for card in
hand])}")
print(f"Remaining cards in deck: {len(deck.cards)}")
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

Player's Hand: Jack of Diamonds, 2 of Clubs, Ace of Hearts, 3 of Hearts, 2 of Hearts
Remaining cards in deck: 47

Q15) 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 Python data classes over named tuples:

1. Data classes support inheritance, default values, and type hints.
2. Data classes provide built-in methods for comparison, hashing, and string representation.
3. Data classes support mutable fields.
4. Data classes can be used to create complex data structures, such as nested data classes.
5. Data classes can be used to create data classes with custom initialization methods.

Example Code(Named Tuple): -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
from collections import namedtuple
Student = namedtuple("Student", ['name', 'age'])
s1 = Student('Sid', 21)
print(s1)
try:
    s1.age = 30
except Exception as e:
    print(e)
```

OUTPUT(Named Tuple): -

Student(name='Sid', age=21)
can't set attribute

Example Code(Data Class): -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
from dataclasses import dataclass, field
@dataclass
class Student:
    name: str
    age: int = field(default=21)

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")
s2 = Student('Sid', 21)
s2.display()
try:
    s2.age = 30
except Exception as e:
    print(e)
print("\nAfter changing age:-")
s2.display()
```

OUTPUT(Data Class): -

Name: Sidhanta Barik, RegNo: 2241002049

Name: Sid, Age: 21

After changing age:-
Name: Sid, Age: 30

Q16) 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.

Code: -

```
print('''Name: Sidhanta Barik, RegNo: 2241002049
----------------------------------------''')
def add(a, b):
    """
    Test Cases
    ==========
    >>> add(1, 2)
    3
    >>> add(-1, 1)
    0
    >>> add(-1, -1)
    -2
    >>> add(1.4, 4.2)
    5.6
    """
    return a+b

if __name__ == "__main__":
    import doctest
    res = doctest.testmod()
    if res.failed > 0:
        print(f"Tests failed: {res.failed}")
    else:
        print("All tests passed")
    print(f"Total tests: {res.attempted}")
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

All tests passed
Total tests: 4

Q17) Scope Resolution: object's namespace →class namespace →global namespace →built-in namespace.

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

Code: -

```
species = "Global Species"

class Animal:
    species = "Class Species"

    def __init__(self, species):
        self.species = species

    def display_species(self):
        print(f"Instance species: {self.species}")
        print(f"Class species: {Animal.species}")
        print(f"Global species: {globals()['species']}")

a = Animal("Instance Species")
a.display_species()
```

OUTPUT: -

Name: Sidhanta Barik, RegNo: 2241002049

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

Explanation: - (Scope Resolution Process, Step by Step)

In Python, scope resolution follows LEGB(Local, Enclosing, Global, Built-in) rule.
- Local scope: The local scope is the scope of the current function or method.
- Enclosing scope: The enclosing scope is the scope of the outer function or method.
- Global scope: The global scope is the scope of the global variables.
- Built-in scope: The built-in scope is the scope of the built-in functions and variables.