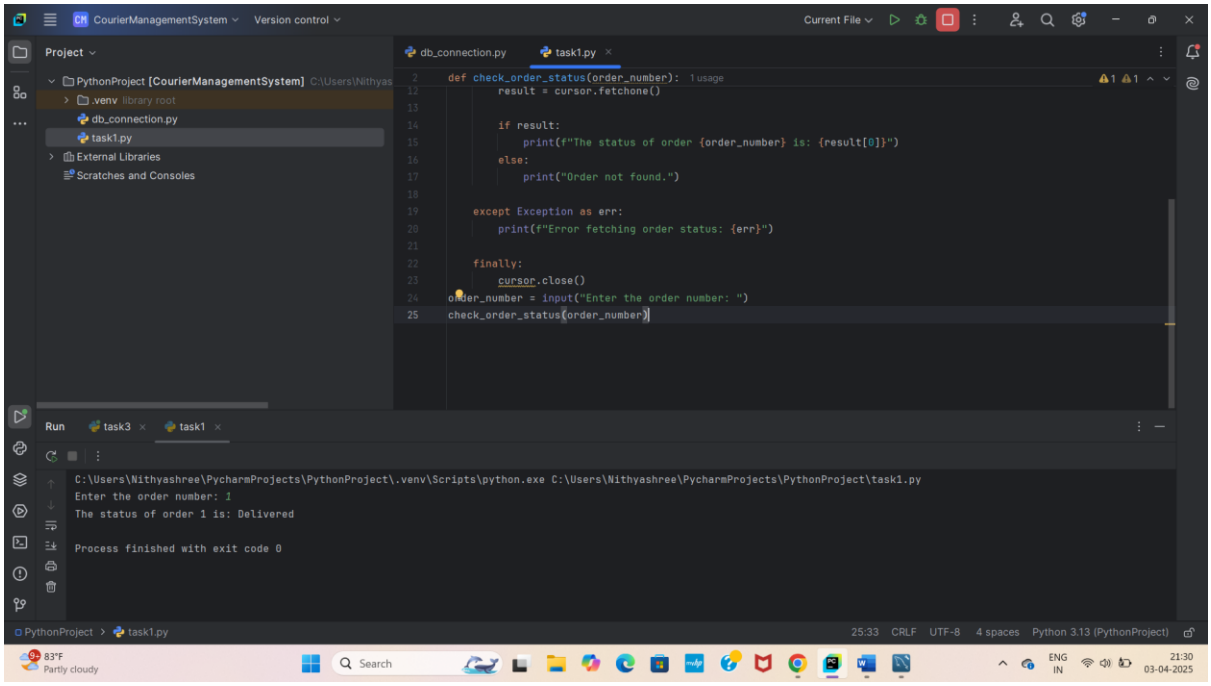
COURIER MANAGEMENT SYSTEM ASSIGNMENT

CODING PART

Task 1: Control Flow Statements

1: Control Flow Statements 1. Write a program that checks whether a given order is delivered or not based on its status (e.g., "Processing," "Delivered," "Cancelled"). Use if-else statements for this.

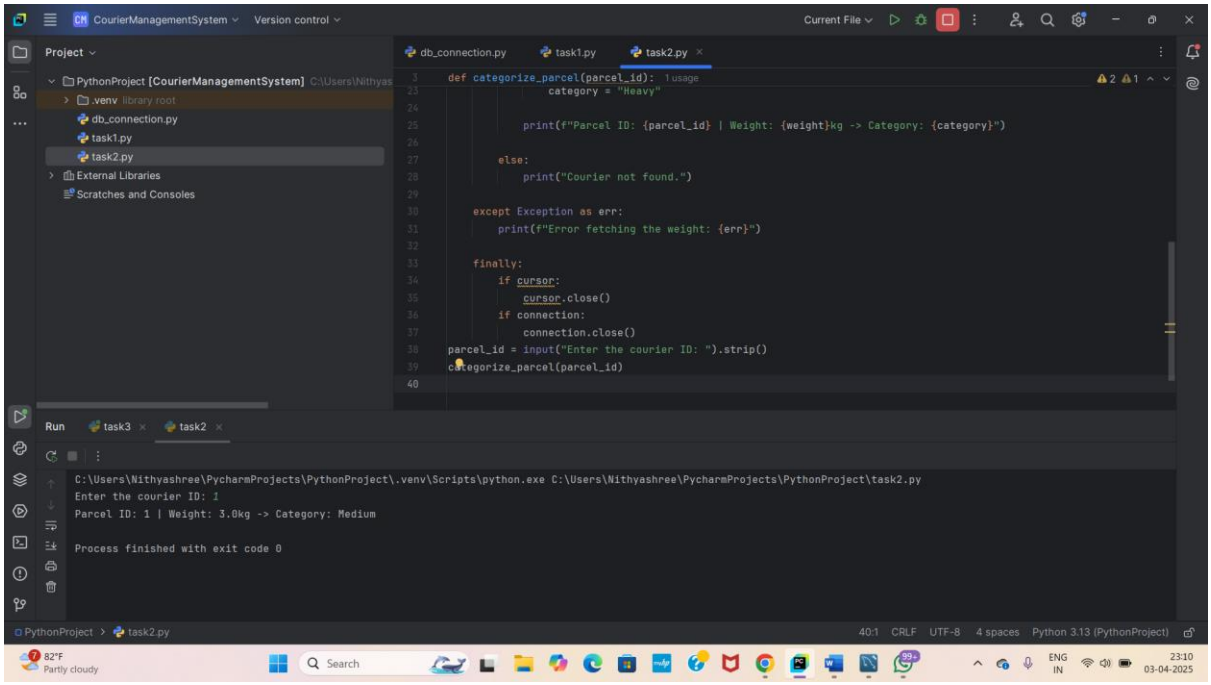
```
from db_connection import get_db_connection
def check_order_status(order_number):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return

    try:
        cursor = connection.cursor()
        query = "SELECT status FROM courier WHERE CourierID = %s"
        cursor.execute(query, (order_number,))
        result = cursor.fetchone()

        if result:
            print(f"The status of order {order_number} is: {result[0]}")
        else:
            print("Order not found.")

    except Exception as err:
        print(f"Error fetching order status: {err}")

    finally:
        cursor.close()
order_number = input("Enter the order number: ")
check_order_status(order_number)
```



2. Implement a switch-case statement to categorize parcels based on their weight into "Light," "Medium," or "Heavy."

```
from db_connection import get_db_connection

def categorize_parcel(parcel_id):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT weight FROM courier WHERE CourierID = %s"
        cursor.execute(query, (int(parcel_id),))
        result = cursor.fetchone()
        if result:
            weight = float(result[0])  # Convert DB result to float
            if weight <= 2:
                category = "Light"
            elif 2 < weight <= 5:
                category = "Medium"
            else:
                category = "Heavy"
            print(f"Parcel ID: {parcel_id} | Weight: {weight}kg -> Category: {category}")
        else:
            print("Courier not found.")
    except Exception as err:
        print(f"Error fetching the weight: {err}")
    finally:
        if cursor:
```

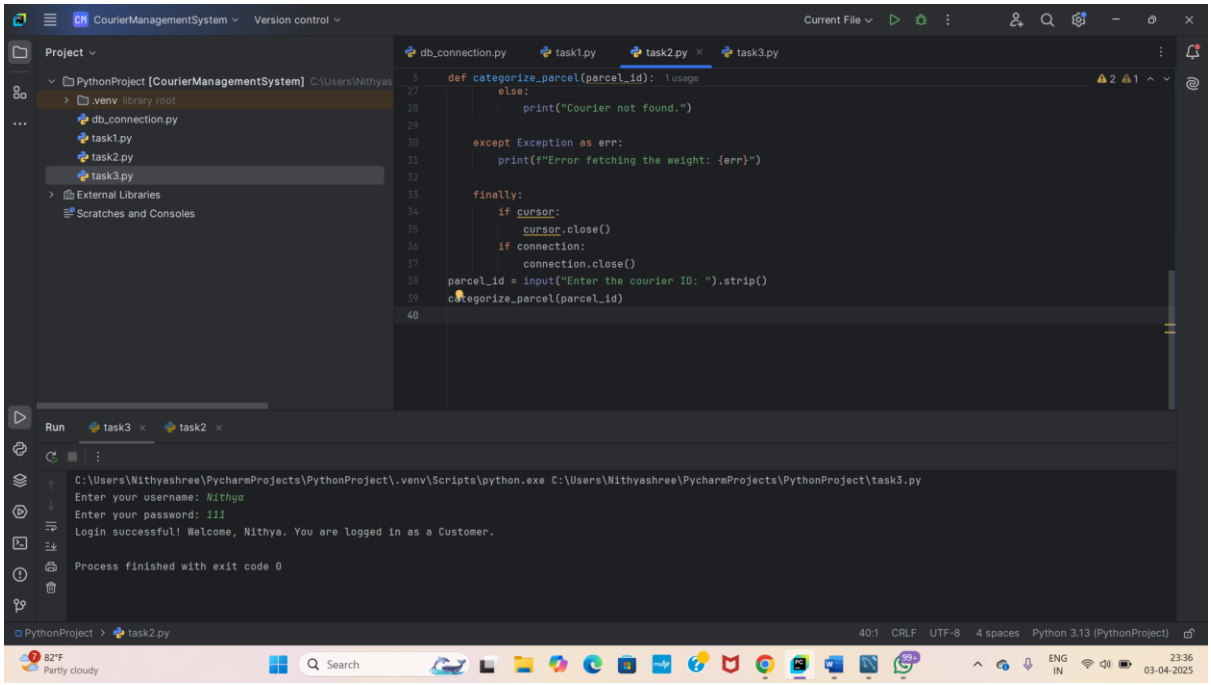
```
        cursor.close()
    if connection:
        connection.close()
parcel_id = input("Enter the courier ID: ").strip()
categorize_parcel(parcel_id)
```



3. Implement User Authentication 1. Create a login system for employees and customers using Java control flow statements.

```
from db_connection import get_db_connection
def categorize_parcel(parcel_id):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT weight FROM courier WHERE CourierID = %s"
        cursor.execute(query, (int(parcel_id),))
        result = cursor.fetchone()
        if result:
            weight = float(result[0])  # Convert DB result to float
            if weight <= 2:
                category = "Light"
            elif 2 < weight <= 5:
                category = "Medium"
            else:
                category = "Heavy"
            print(f"Parcel ID: {parcel_id} | Weight: {weight}kg -> Category: {category}")
        else:
```

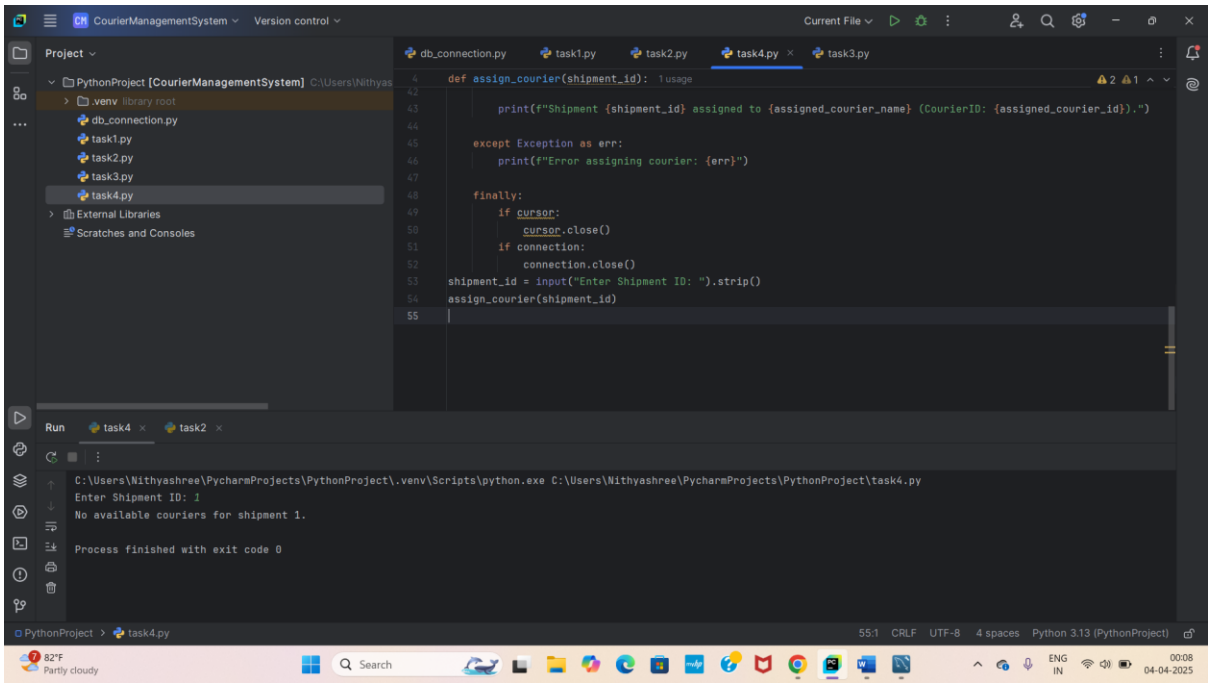
```
            print("Courier not found.")
    except Exception as err:
        print(f"Error fetching the weight: {err}")
    finally:
        if cursor:
            cursor.close()
        if connection:
            connection.close()
parcel_id = input("Enter the courier ID: ").strip()
categorize_parcel(parcel_id)
```



4. Implement Courier Assignment Logic 1. Develop a mechanism to assign couriers to shipments based on predefined criteria (e.g., proximity, load capacity) using loops.

```
from db_connection import get_db_connection
def assign_courier(shipment_id):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        cursor.execute("SELECT destination, weight FROM shipments WHERE ShipmentID = %s",
(shipment_id,))
        shipment = cursor.fetchone()
        if not shipment:
            print(f"Shipment {shipment_id} not found.")
            return
        destination, weight = shipment
        query = """SELECT CourierID, SenderName, Weight
```

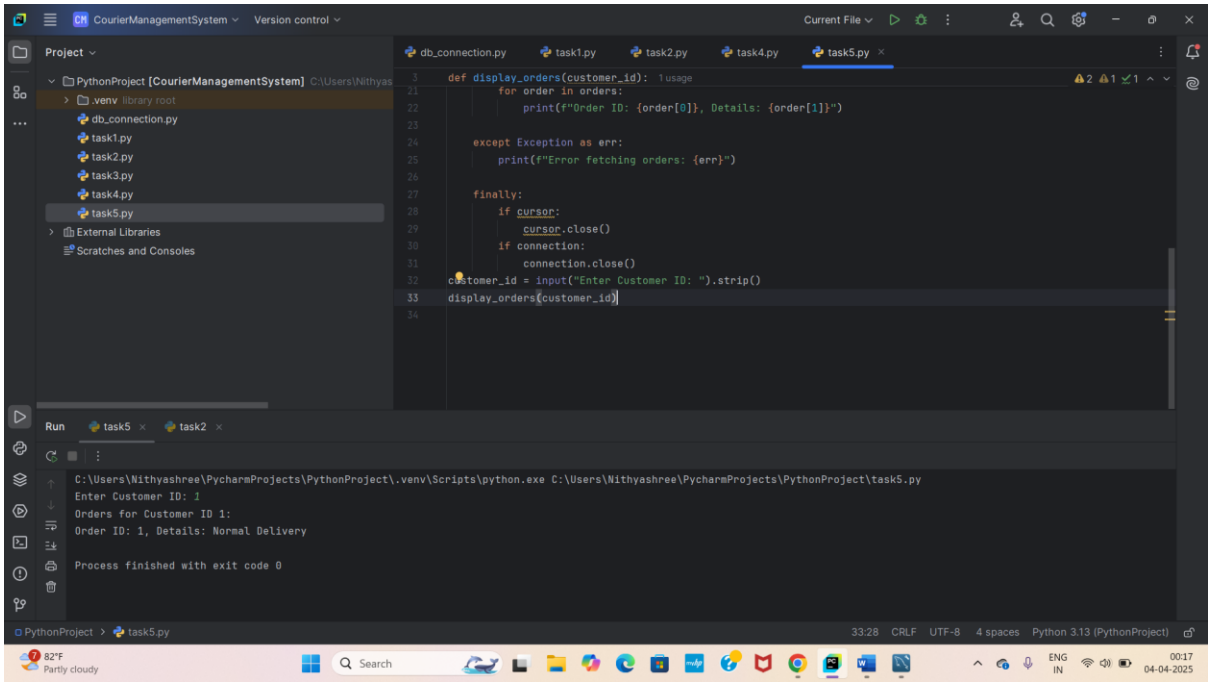
```
            FROM courier
            WHERE ReceiverAddress = %s AND Weight >= %s
            ORDER BY Weight ASC"""
        cursor.execute(query, (destination, weight))
        couriers = cursor.fetchall()
        if not couriers:
            print(f"No available couriers for shipment {shipment_id}.")
            return
        assigned_courier_id = couriers[0][0]
        assigned_courier_name = couriers[0][1]
        cursor.execute("UPDATE shipments SET AssignedCourierID = %s WHERE ShipmentID = %s",
                       (assigned_courier_id, shipment_id))
        connection.commit()
        print(f"Shipment {shipment_id} assigned to {assigned_courier_name} (CourierID:
{assigned_courier_id}).")
    except Exception as err:
        print(f"Error assigning courier: {err}")
    finally:
        if cursor:
            cursor.close()
        if connection:
            connection.close()
shipment_id = input("Enter Shipment ID: ").strip()
assign_courier(shipment_id)
```



Task 2: Loops and Iteration

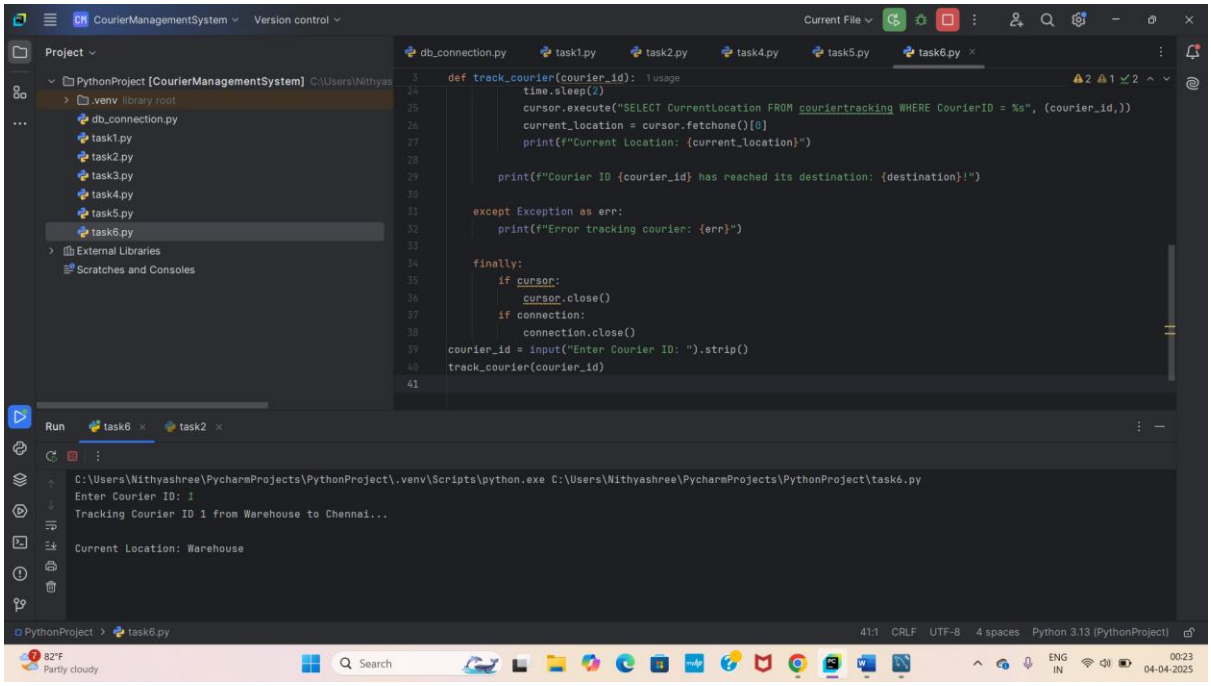
5. Write a Java program that uses a for loop to display all the orders for a specific customer.

```
from db_connection import get_db_connection
def display_orders(customer_id):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT ServicesID, ServiceName FROM courierservices WHERE ServicesID = %s"
        cursor.execute(query, (customer_id,))
        orders = cursor.fetchall()
        if not orders:
            print(f"No orders found for Customer ID {customer_id}.")
            return
        print(f"Orders for Customer ID {customer_id}:")
        for order in orders:
            print(f"Order ID: {order[0]}, Details: {order[1]}")
    except Exception as err:
        print(f"Error fetching orders: {err}")
    finally:
        if cursor:
            cursor.close()
        if connection:
            connection.close()
customer_id = input("Enter Customer ID: ").strip()
display_orders(customer_id)
```



6. Implement a while loop to track the real-time location of a courier until it reaches its destination

```
import time
from db_connection import get_db_connection
def track_courier(courier_id):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT CurrentLocation, Destination FROM couriertracking WHERE CourierID = %s"
        cursor.execute(query, (courier_id,))
        result = cursor.fetchone()
        if not result:
            print(f"No tracking data found for Courier ID {courier_id}.")
            return
        current_location, destination = result
        print(f"Tracking Courier ID {courier_id} from {current_location} to {destination}...\n")
        while current_location != destination:
            time.sleep(2)
            cursor.execute("SELECT CurrentLocation FROM couriertracking WHERE CourierID = %s",
(courier_id,))
            current_location = cursor.fetchone()[0]
            print(f"Current Location: {current_location}")
        print(f"Courier ID {courier_id} has reached its destination: {destination}!")
    except Exception as err:
        print(f"Error tracking courier: {err}")
    finally:
        if cursor:
            cursor.close()
        if connection:
            connection.close()
courier_id = input("Enter Courier ID: ").strip()
track_courier(courier_id)
```

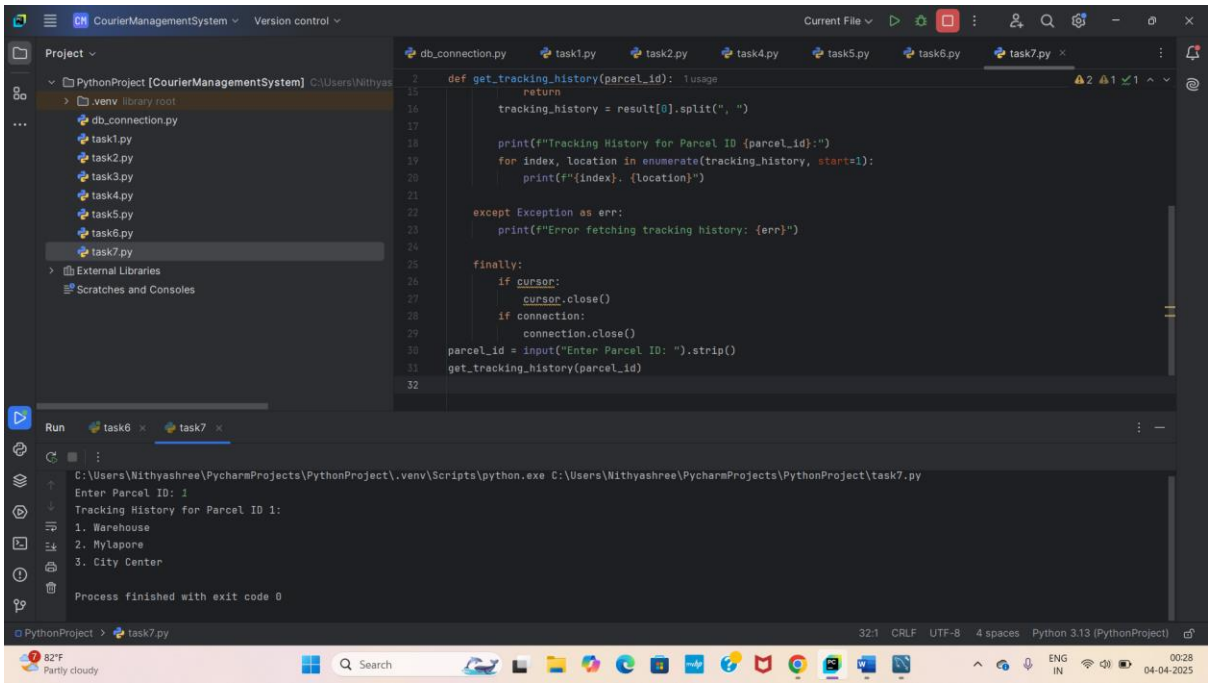


Task 3: Arrays and Data Structures

7. Create an array to store the tracking history of a parcel, where each entry represents a location update

```
from db_connection import get_db_connection
def get_tracking_history(parcel_id):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT LocationHistory FROM parceltracking WHERE ParcelID = %s"
        cursor.execute(query, (parcel_id,))
        result = cursor.fetchone()
        if not result:
            print(f"No tracking history found for Parcel ID {parcel_id}.")
            return
        tracking_history = result[0].split(", ")
        print(f"Tracking History for Parcel ID {parcel_id}:")
        for index, location in enumerate(tracking_history, start=1):
            print(f"{index}. {location}")
    except Exception as err:
        print(f"Error fetching tracking history: {err}")
    finally:
        if cursor:
            cursor.close()
```

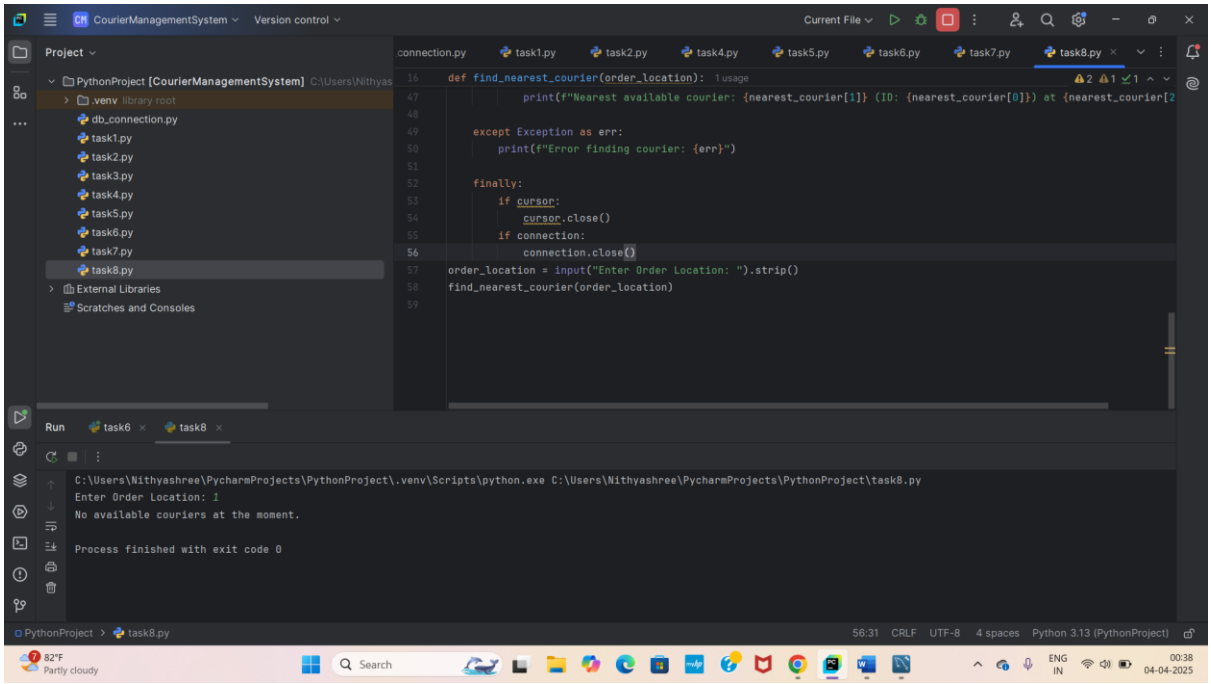
```
        if connection:
            connection.close()
parcel_id = input("Enter Parcel ID: ").strip()
get_tracking_history(parcel_id)
```



8. Implement a method to find the nearest available courier for a new order using an array of couriers

```
from db_connection import get_db_connection
distance_matrix = {
    ("Warehouse", "City Center"): 5,
    ("Warehouse", "Mylapore"): 3,
    ("Hub A", "City Center"): 4
}
def calculate_distance(location1, location2):
    """Returns the distance between two locations based on predefined distances."""
    if location1 == location2:
        return 0
    return distance_matrix.get((location1, location2), float('inf'))
def find_nearest_courier(order_location):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT CourierID, SenderName, SenderAddress FROM courier WHERE Availability = 1"
        cursor.execute(query)
        couriers = cursor.fetchall()
        if not couriers:
```

```
            print("No available couriers at the moment.")
            return
        nearest_courier = None
        min_distance = float('inf')
        for courier in couriers:
            courier_id, courier_name, courier_location = courier
            distance = calculate_distance(courier_location, order_location)
            if distance < min_distance:
                min_distance = distance
                nearest_courier = (courier_id, courier_name, courier_location)
        if nearest_courier:
            print(f"Nearest available courier: {nearest_courier[1]} (ID: {nearest_courier[0]}) at
{nearest_courier[2]}")
    except Exception as err:
        print(f"Error finding courier: {err}")
    finally:
        if cursor:
            cursor.close()
        if connection:
            connection.close()
order_location = input("Enter Order Location: ").strip()
find_nearest_courier(order_location)
```

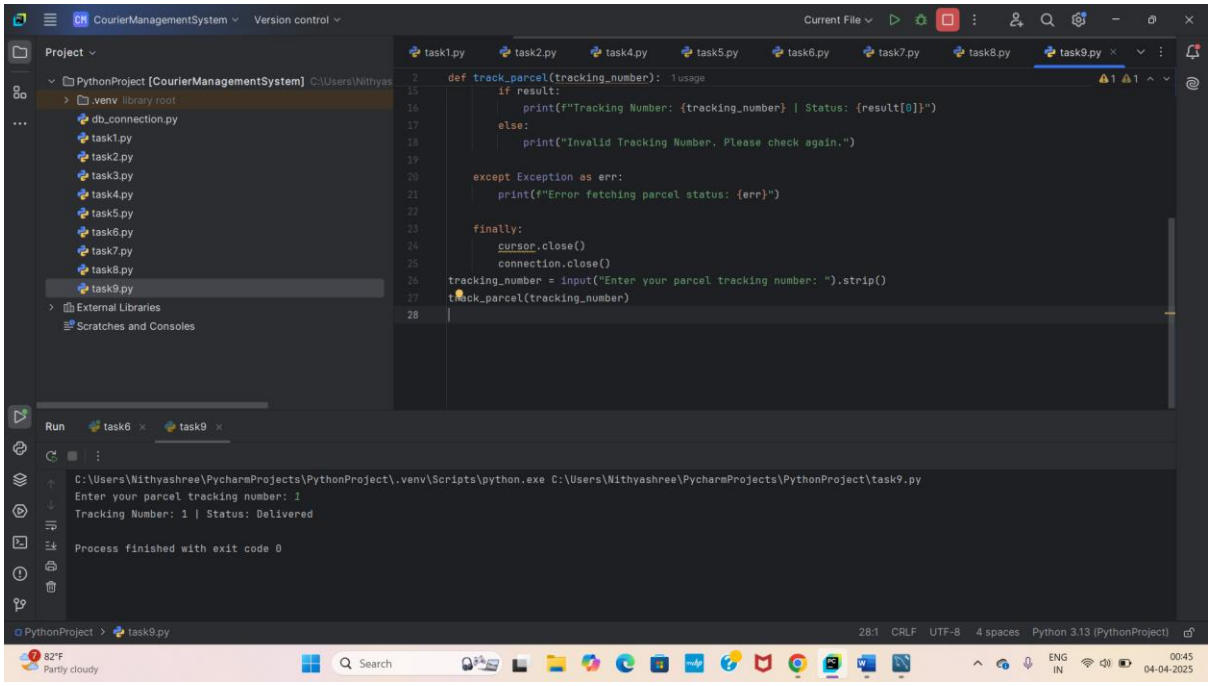


Task 4: Strings,2d Arrays, user defined functions,Hashmap

9. Parcel Tracking: Create a program that allows users to input a parcel tracking number.Store the tracking number and Status in 2d String Array. Initialize the array with values. Then, simulate the

tracking process by displaying messages like "Parcel in transit," "Parcel out for delivery," or "Parcel delivered" based on the tracking number's status.

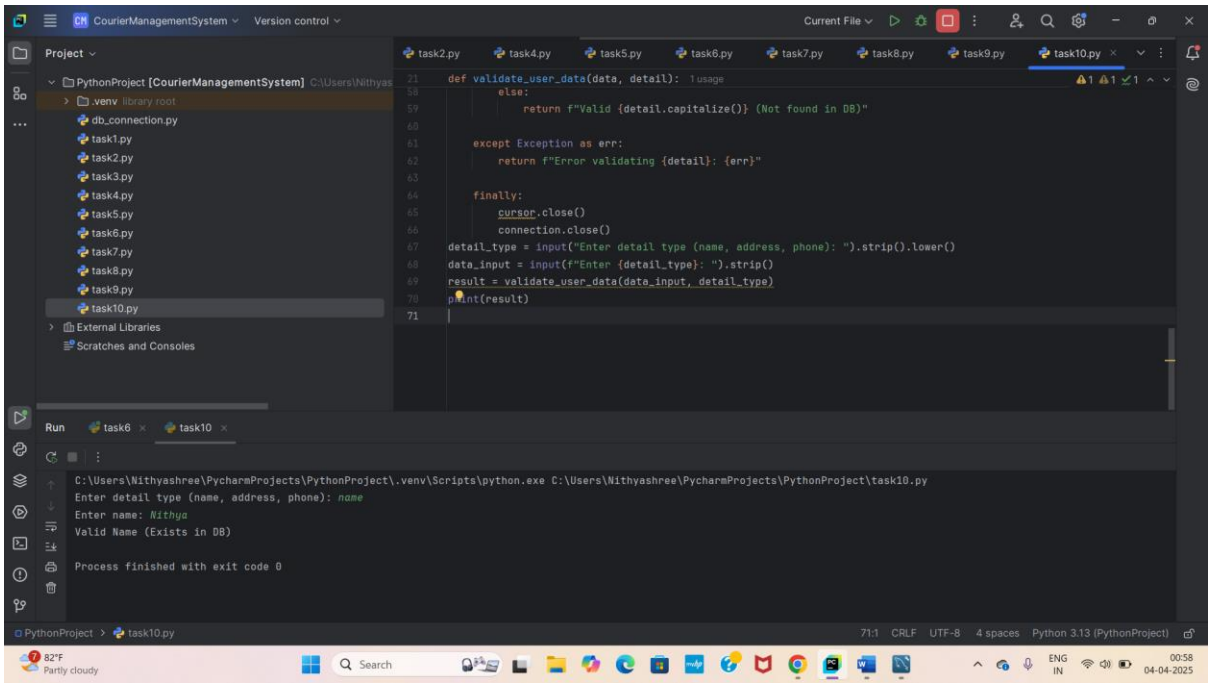
```
from db_connection import get_db_connection
def track_parcel(tracking_number):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "SELECT Status FROM parcels WHERE TrackingNumber = %s"
        cursor.execute(query, (tracking_number,))
        result = cursor.fetchone()
        if result:
            print(f"Tracking Number: {tracking_number} | Status: {result[0]}")
        else:
            print("Invalid Tracking Number. Please check again.")
    except Exception as err:
        print(f"Error fetching parcel status: {err}")
    finally:
        cursor.close()
        connection.close()
tracking_number = input("Enter your parcel tracking number: ").strip()
track_parcel(tracking_number)
```



10. Customer Data Validation: Write a function which takes 2 parameters, data-denotes the data and detail-denotes if it is name addtress or phone number.Validate customer information based on following critirea. Ensure that names contain only letters and are properly capitalized, addresses do not contain special characters, and phone numbers follow a specific format (e.g., ###-###-####).

```
import re
import mysql.connector
from mysql.connector import Error
def get_db_connection():
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='couriermanagementsystem',
            user='root',
            password='Deepa@25'
        )
        if connection.is_connected():
            return connection
    except Error as err:
        print(f"Error: {err}")
        return None
def validate_user_data(data, detail):
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        if detail == "name":
            if not re.fullmatch(r"[A-Z][a-zA-Z\s]*$", data):
                return "Invalid Name: Only letters allowed, must start with a capital letter."
            cursor.execute("SELECT * FROM user WHERE Name = %s", (data,))
        elif detail == "address":
            if not re.fullmatch(r"[A-Za-z0-9\s,.-]+$", data):
                return "Invalid Address: Special characters are not allowed."
            cursor.execute("SELECT * FROM user WHERE Address = %s", (data,))
        elif detail == "phone":
            if not re.fullmatch(r"\d{3}-\d{3}-\d{4}$", data):
                return "Invalid Phone Number: Format should be ###-###-####."
            cursor.execute("SELECT * FROM user WHERE Phone = %s", (data,))
        else:
            return "Invalid Detail Type"
        if cursor.fetchone():
            return f"Valid {detail.capitalize()} (Exists in DB)"
        else:
            return f"Valid {detail.capitalize()} (Not found in DB)"
    except Exception as err:
        return f"Error validating {detail}: {err}"
    finally:
        cursor.close()
        connection.close()
detail_type = input("Enter detail type (name, address, phone): ").strip().lower()
data_input = input(f"Enter {detail_type}: ").strip()
```

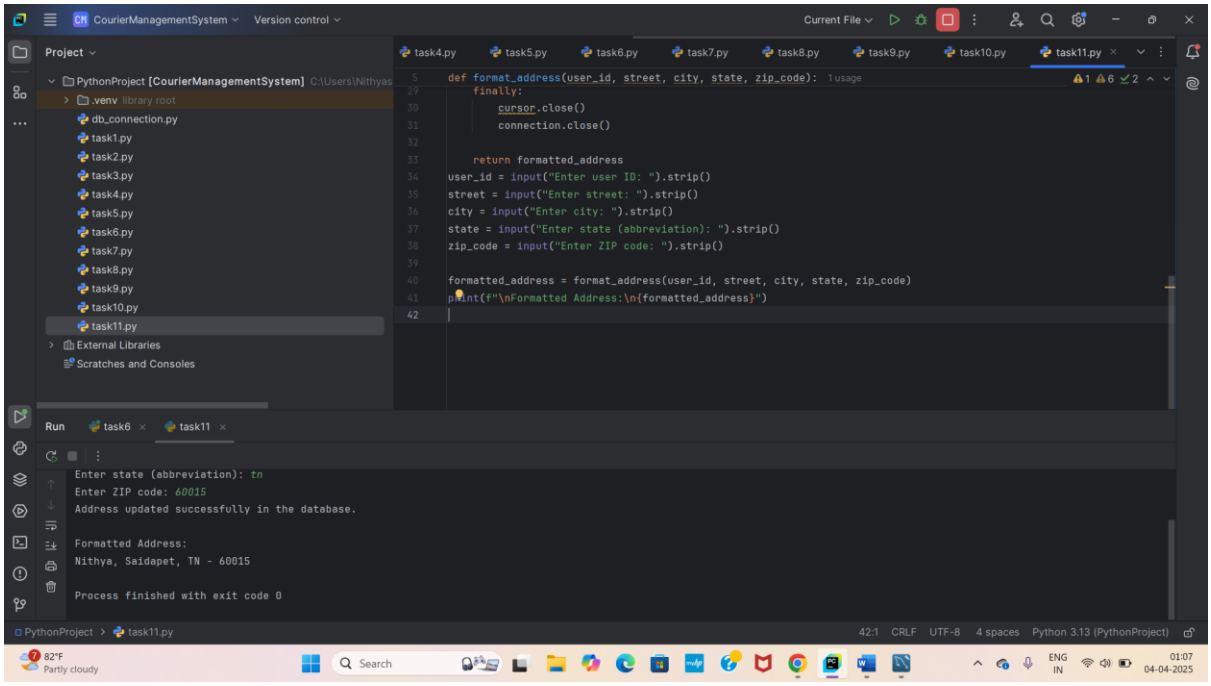
result = validate_user_data(data_input, detail_type)
print(result)



11. Address Formatting: Develop a function that takes an address as input (street, city, state, zip code) and formats it correctly, including capitalizing the first letter of each word and properly formatting the zip code

```
import re
from db_connection import get_db_connection
def format_address(user_id, street, city, state, zip_code):
    formatted_street = street.title()
    formatted_city = city.title()
    formatted_state = state.upper()  # States are usually in uppercase (e.g., NY, CA)
    if not re.fullmatch(r"\d{5}(-\d{4})?", zip_code):
        return "Invalid ZIP Code format. Use XXXXX or XXXXX-XXXX."
    formatted_address = f"{formatted_street}, {formatted_city}, {formatted_state} - {zip_code}"
    connection = get_db_connection()
    if connection is None:
        return "Database connection failed."
    try:
        cursor = connection.cursor()
        query = "UPDATE user SET Address = %s WHERE UserID = %s"
        cursor.execute(query, (formatted_address, user_id))
        connection.commit()
        print("Address updated successfully in the database.")
    except Exception as err:
        print(f"Error updating address: {err}")
    finally:
        cursor.close()
        connection.close()
```

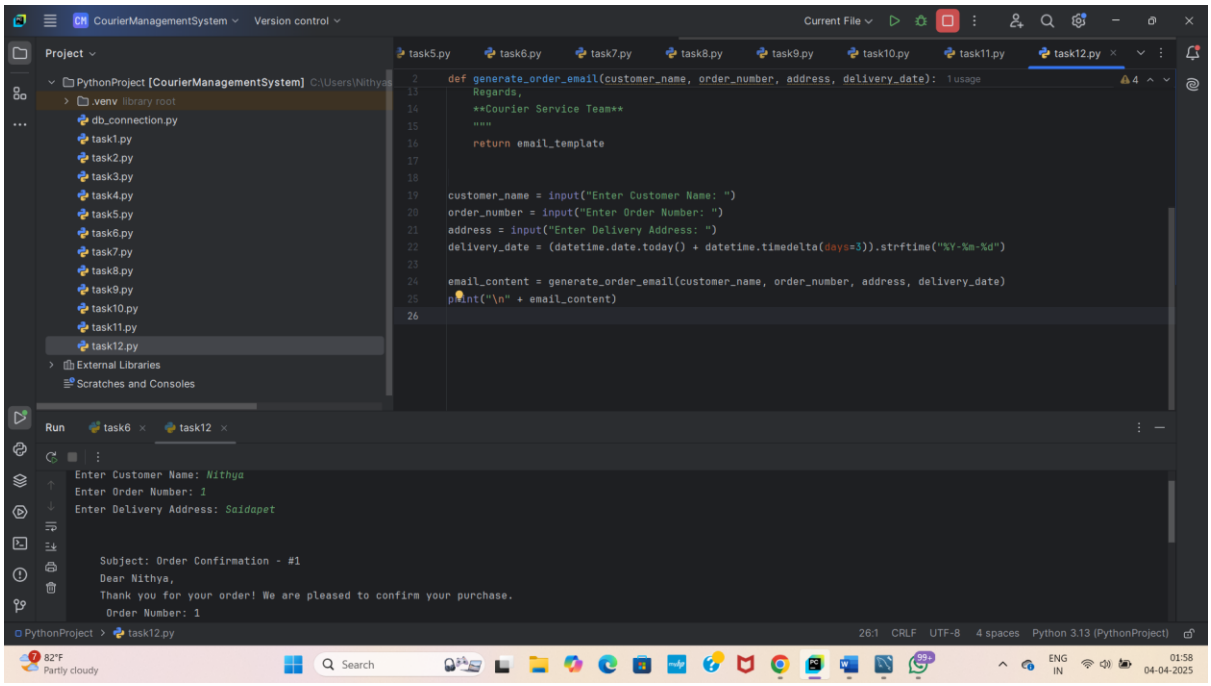
```
    return formatted_address
user_id = input("Enter user ID: ").strip()
street = input("Enter street: ").strip()
city = input("Enter city: ").strip()
state = input("Enter state (abbreviation): ").strip()
zip_code = input("Enter ZIP code: ").strip()
formatted_address = format_address(user_id, street, city, state, zip_code)
print(f"\nFormatted Address:\n{formatted_address}")
```



12. Order Confirmation Email: Create a program that generates an order confirmation email. The email should include details such as the customer's name, order number, delivery address, and expected delivery date.

```
import datetime
def generate_order_email(customer_name, order_number, address, delivery_date):
    email_template = f"""
    Subject: Order Confirmation - #{order_number}
    Dear {customer_name},
    Thank you for your order! We are pleased to confirm your purchase.
     Order Number: {order_number}
     Delivery Address: {address}
     Expected Delivery Date: {delivery_date}
    Your package is being prepared for shipment. You will receive tracking updates soon.
    If you have any questions, feel free to contact us.
    Regards,
    **Courier Service Team**
    """
    return email_template
customer_name = input("Enter Customer Name: ")
order_number = input("Enter Order Number: ")
```

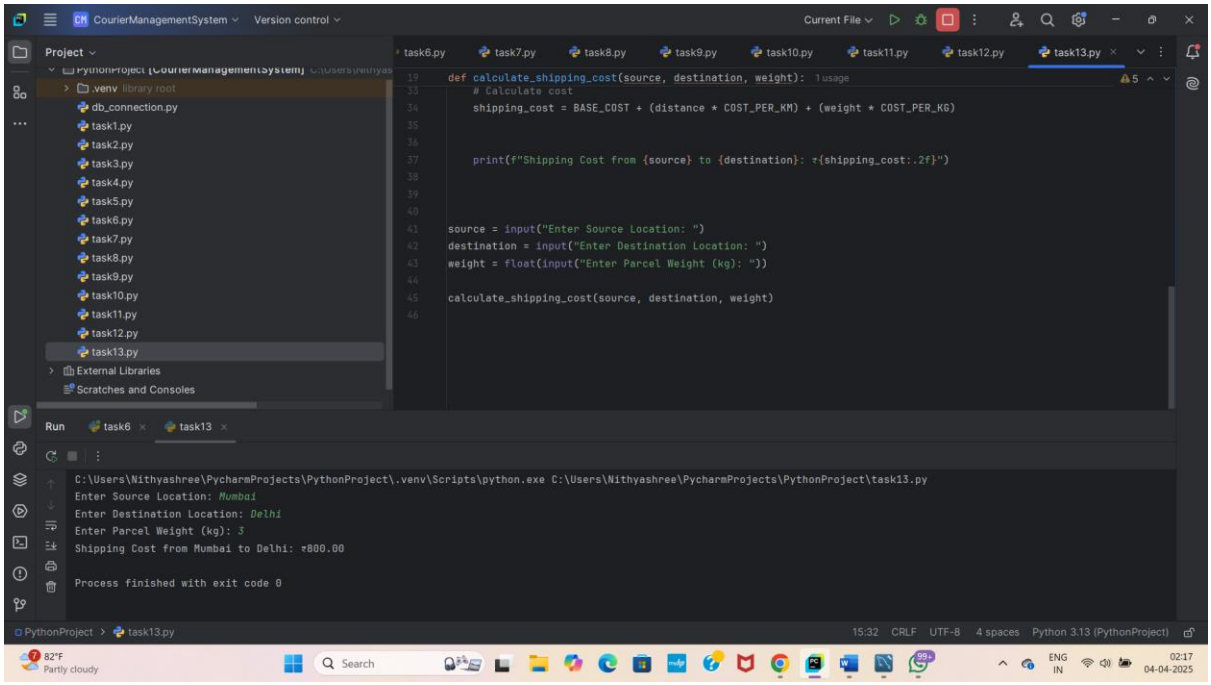
```
address = input("Enter Delivery Address: ")
delivery_date = (datetime.date.today() + datetime.timedelta(days=3)).strftime("%Y-%m-%d")
email_content = generate_order_email(customer_name, order_number, address, delivery_date)
print("\n" + email_content)
```



13. Calculate Shipping Costs: Develop a function that calculates the shipping cost based on the distance between two locations and the weight of the parcel. You can use string inputs for the source and destination addresses.

```
distance_data = {
    ("Chennai", "Bangalore"): 350,
    ("Chennai", "Delhi"): 2200,
    ("Chennai", "Mumbai"): 1330,
    ("Bangalore", "Mumbai"): 980,
    ("Mumbai", "Delhi"): 1440,
    ("Delhi", "Kolkata"): 1530,
    ("Hyderabad", "Chennai"): 630,
}
BASE_COST = 50  # Base charge
COST_PER_KM = 0.5  # Cost per km
COST_PER_KG = 10  # Cost per kg
def calculate_shipping_cost(source, destination, weight):
    # Convert input to title case to match dictionary keys
    source = source.title()
    destination = destination.title()
    if (source, destination) in distance_data:
        distance = distance_data[(source, destination)]
    elif (destination, source) in distance_data:  # Reverse lookup
        distance = distance_data[(destination, source)]
```

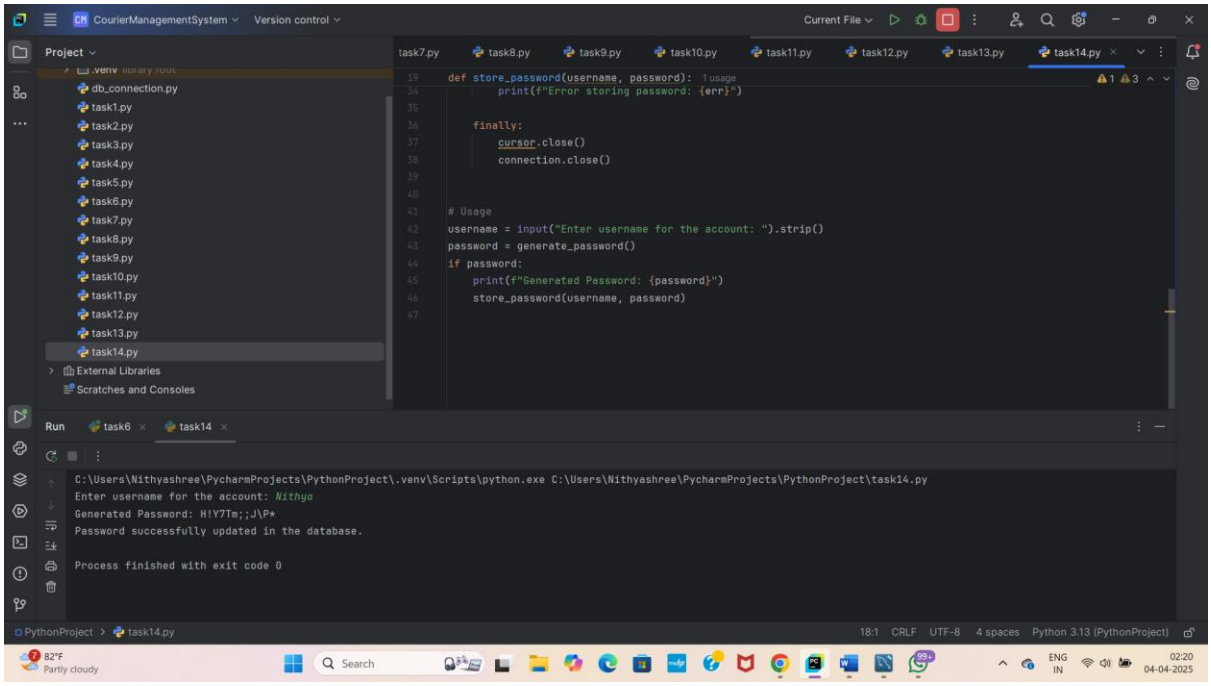
```
    else:
        print("Shipping route not found in database.")
        return
    shipping_cost = BASE_COST + (distance * COST_PER_KM) + (weight * COST_PER_KG)
    print(f"Shipping Cost from {source} to {destination}: ₹{shipping_cost:.2f}")
source = input("Enter Source Location: ")
destination = input("Enter Destination Location: ")
weight = float(input("Enter Parcel Weight (kg): "))
calculate_shipping_cost(source, destination, weight)
```



14. Password Generator: Create a function that generates secure passwords for courier system accounts. Ensure the passwords contain a mix of uppercase letters, lowercase letters, numbers, and special characters

```
import random
import string
from db_connection import get_db_connection
def generate_password(length=12):
    """Generate a secure password with a mix of uppercase, lowercase, numbers, and special characters."""
    if length < 8:
        print("Password length should be at least 8 characters for security.")
        return None

    characters = (
            string.ascii_uppercase + string.ascii_lowercase + string.digits + string.punctuation
    )
    password = ''.join(random.choices(characters, k=length))
    return password
def store_password(username, password):
```

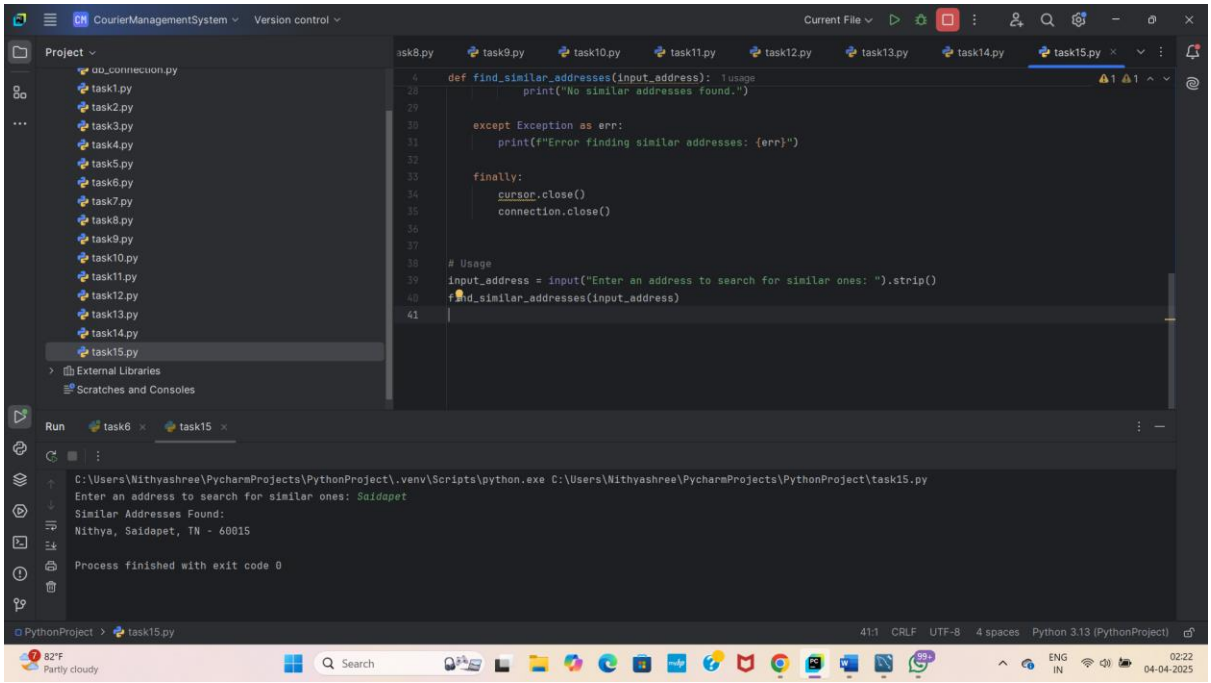
```
    """Store the generated password in the database."""
    connection = get_db_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = "UPDATE user SET password = %s WHERE Name = %s"
        cursor.execute(query, (password, username))
        connection.commit()
        print("Password successfully updated in the database.")
    except Exception as err:
        print(f"Error storing password: {err}")
    finally:
        cursor.close()
        connection.close()
username = input("Enter username for the account: ").strip()
password = generate_password()
if password:
    print(f"Generated Password: {password}")
    store_password(username, password)
```



15. Find Similar Addresses: Implement a function that finds similar addresses in the system. This can be useful for identifying duplicate customer entries or optimizing delivery routes.Use string functions to implement this.

```
from db_connection import get_db_connection
def find_similar_addresses(input_address):
    """Find similar addresses in the database using string matching techniques."""
    connection = get_db_connection()
```

```
    if connection is None:
        print("Failed to connect to the database.")
        return

    try:
        cursor = connection.cursor()
        # Query to find similar addresses using SQL functions
        query = """
        SELECT Address FROM user
        WHERE Address LIKE %s
        OR SOUNDEX(Address) = SOUNDEX(%s)
        """
        cursor.execute(query, (f"%{input_address}%", input_address))
        results = cursor.fetchall()
        if results:
            print("Similar Addresses Found:")
            for row in results:
                print(row[0])
        else:
            print("No similar addresses found.")
    except Exception as err:
        print(f"Error finding similar addresses: {err}")
    finally:
        cursor.close()
        connection.close()
input_address = input("Enter an address to search for similar ones: ").strip()
find_similar_addresses(input_address)
```



Task 5: Object Oriented Programming

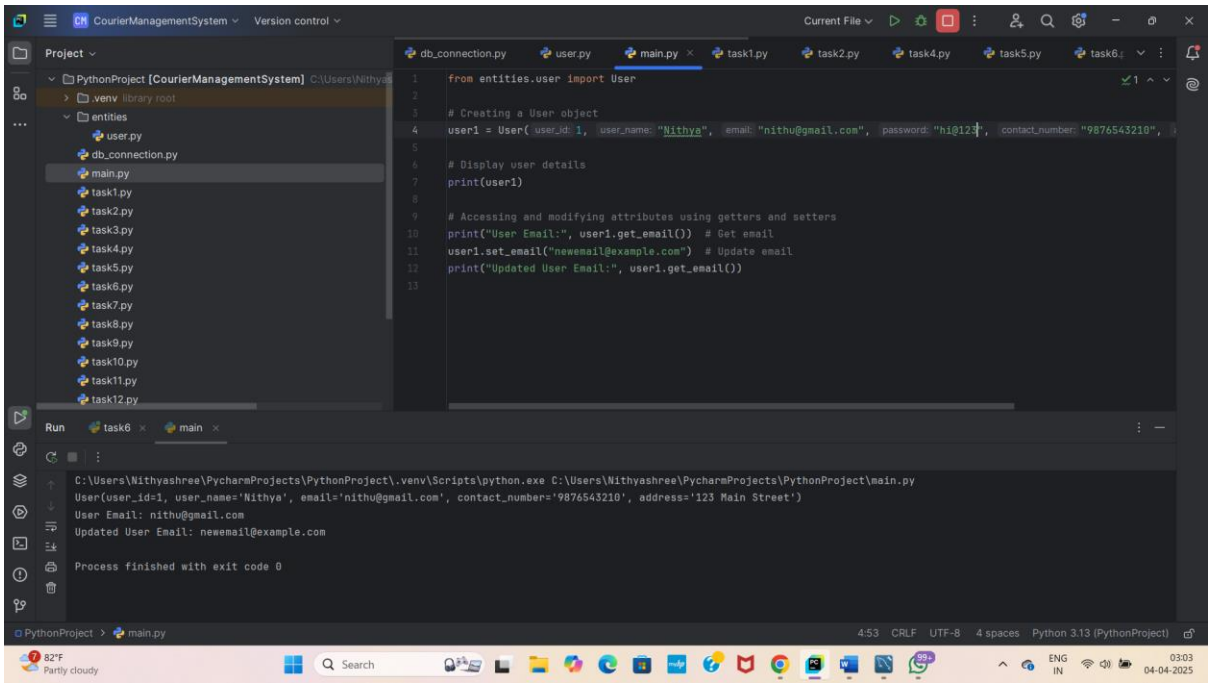
1. Create the following model/entity classes within package entities with variables declared private, constructors(default and parametrized,getters,setters and toString()) 1. User Class: Variables: userID , userName , email , password , contactNumber , address

```
class User:
    def __init__(self, user_id=None, user_name=None, email=None, password=None, contact_number=None, address=None):
        self.__user_id = user_id
        self.__user_name = user_name
        self.__email = email
        self.__password = password
        self.__contact_number = contact_number
        self.__address = address
    def get_user_id(self):
        return self.__user_id
    def get_user_name(self):
        return self.__user_name
    def get_email(self):
        return self.__email
    def get_password(self):
        return self.__password
    def get_contact_number(self):
        return self.__contact_number
    def get_address(self):
        return self.__address
    # Setters
    def set_user_id(self, user_id):
        self.__user_id = user_id

    def set_user_name(self, user_name):
        self.__user_name = user_name
    def set_email(self, email):
        self.__email = email
```

```
    def set_password(self, password):
        self.__password = password
    def set_contact_number(self, contact_number):
        self.__contact_number = contact_number
    def set_address(self, address):
        self.__address = address
    # toString equivalent (for displaying object details)
    def __str__(self):
        return f"User(user_id={self.__user_id}, user_name='{self.__user_name}', email='{self.__email}', "
\
 f"contact_number='{self.__contact_number}', address='{self.__address}')"
```

“IMPORTING USER CLASS”

```
from entities.user import User
# Creating a User object
user1 = User(1, "Nithya", "nithu@gmailexample.com", "hi@123", "9876543210", "123 Main Street")
# Display user details
print(user1)
print("User Email:", user1.get_email())  # Get email
user1.set_email("newemail@example.com")  # Update email
print("Updated User Email:", user1.get_email())
```



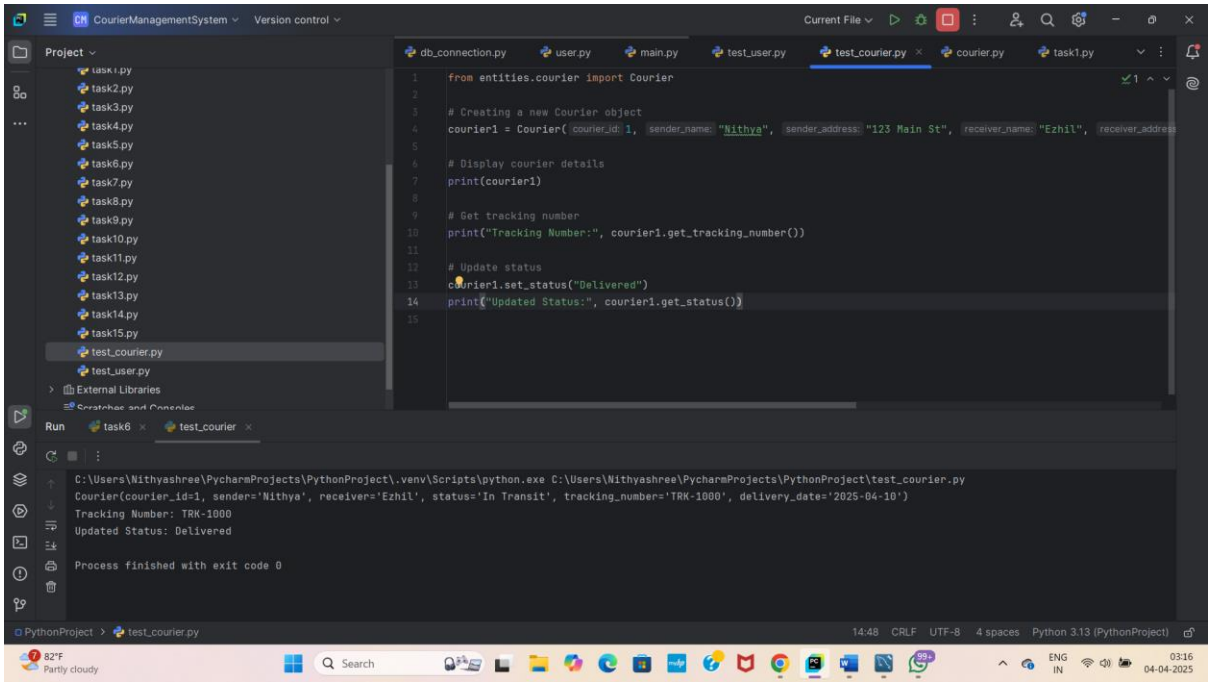
2. Courier Class Variables: courierID , senderName , senderAddress , receiverName , receiverAddress , weight , status, trackingNumber , deliveryDate ,userId

```
class Courier:
    tracking_counter = 1000
    def __init__(self, courier_id=None, sender_name=None, sender_address=None, receiver_name=None,
                 receiver_address=None, weight=None, status="Yet to Transit", delivery_date=None, user_id=None):
        self.__courier_id = courier_id
        self.__sender_name = sender_name
        self.__sender_address = sender_address
        self.__receiver_name = receiver_name
        self.__receiver_address = receiver_address
        self.__weight = weight
        self.__status = status
        self.__tracking_number = f"TRK-{Courier.tracking_counter}"
        Courier.tracking_counter += 1  # Increment for the next courier
        self.__delivery_date = delivery_date
        self.__user_id = user_id
    def get_courier_id(self):
        return self.__courier_id
    def get_sender_name(self):
        return self.__sender_name
    def get_sender_address(self):
        return self.__sender_address
    def get_receiver_name(self):
        return self.__receiver_name
    def get_receiver_address(self):
        return self.__receiver_address
    def get_weight(self):
        return self.__weight
    def get_status(self):
```

```
        return self.__status

    def get_tracking_number(self):

        return self.__tracking_number

    def get_delivery_date(self):

        return self.__delivery_date

    def get_user_id(self):

        return self.__user_id

    def set_status(self, status):

        self.__status = status

    def set_delivery_date(self, delivery_date):

        self.__delivery_date = delivery_date

    def __str__(self):

        return f"Courier(courier_id={self.__courier_id}, sender='{self.__sender_name}',
receiver='{self.__receiver_name}', " \

                f"status='{self.__status}', tracking_number='{self.__tracking_number}',
delivery_date='{self.__delivery_date}')"
```

“IMPORTING COURIER CLASS”

```
from entities.courier import Courier
courier1 = Courier(1, "Nithya", "123 Main St", "Ezhil", "J road", 2.5, "In Transit", "2025-04-10", 101)
print(courier1)
print("Tracking Number:", courier1.get_tracking_number())
courier1.set_status("Delivered")
print("Updated Status:", courier1.get_status())
```



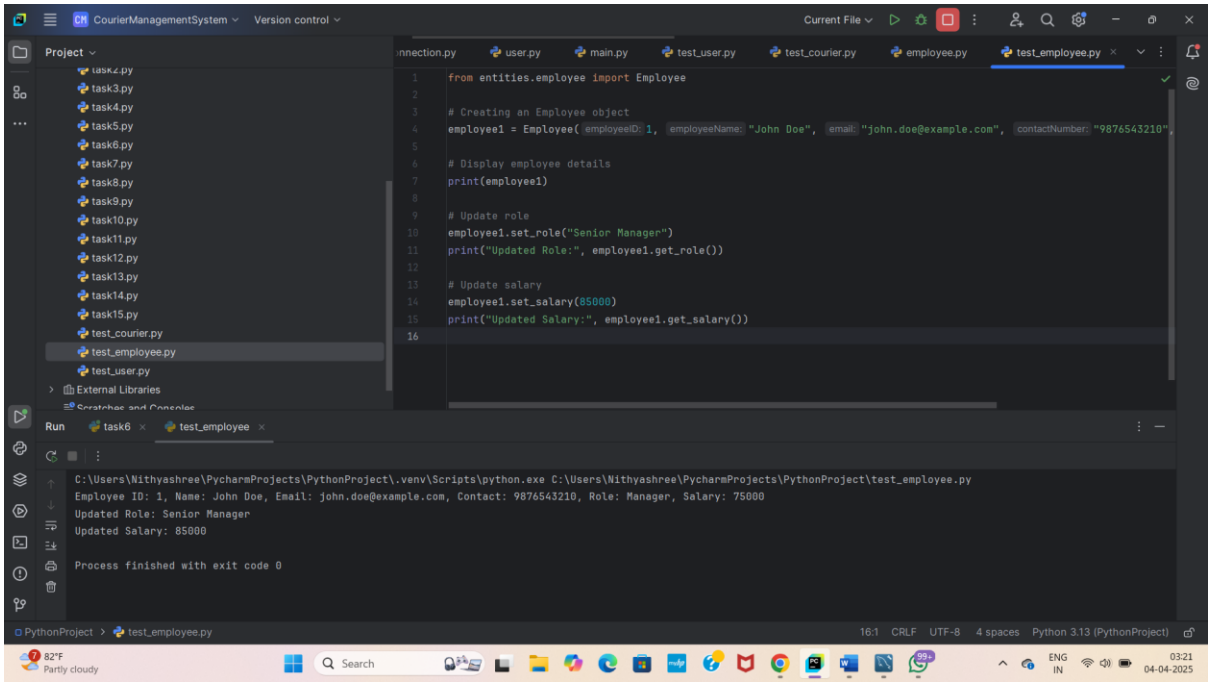
3. Employee Class: Variables employeeID , employeeName , email , contactNumber , role String, salary

```
class Employee:
    def __init__(self, employeeID, employeeName, email, contactNumber, role, salary):
        self.__employeeID = employeeID
        self.__employeeName = employeeName
        self.__email = email
        self.__contactNumber = contactNumber
        self.__role = role
        self.__salary = salary
    def get_employeeID(self):
        return self.__employeeID
    def get_employeeName(self):
        return self.__employeeName
    def get_email(self):
        return self.__email
    def get_contactNumber(self):
        return self.__contactNumber
    def get_role(self):
        return self.__role
```

```
    def get_salary(self):
        return self.__salary
    def set_employeeName(self, employeeName):
        self.__employeeName = employeeName
    def set_email(self, email):
        self.__email = email
    def set_contactNumber(self, contactNumber):
        self.__contactNumber = contactNumber
    def set_role(self, role):
        self.__role = role
    def set_salary(self, salary):
        self.__salary = salary
    def __str__(self):
        return (f"Employee ID: {self.__employeeID}, Name: {self.__employeeName}, Email: {self.__email}, "
                f"Contact: {self.__contactNumber}, Role: {self.__role}, Salary: {self.__salary}")
```

“IMPORTING EMPLOYEE CLASS”

```
from entities.employee import Employee
employee1 = Employee(1, "John Doe", "john.doe@example.com", "9876543210", "Manager", 75000)
print(employee1)
employee1.set_role("Senior Manager")
print("Updated Role:", employee1.get_role())
employee1.set_salary(85000)
print("Updated Salary:", employee1.get_salary())
```

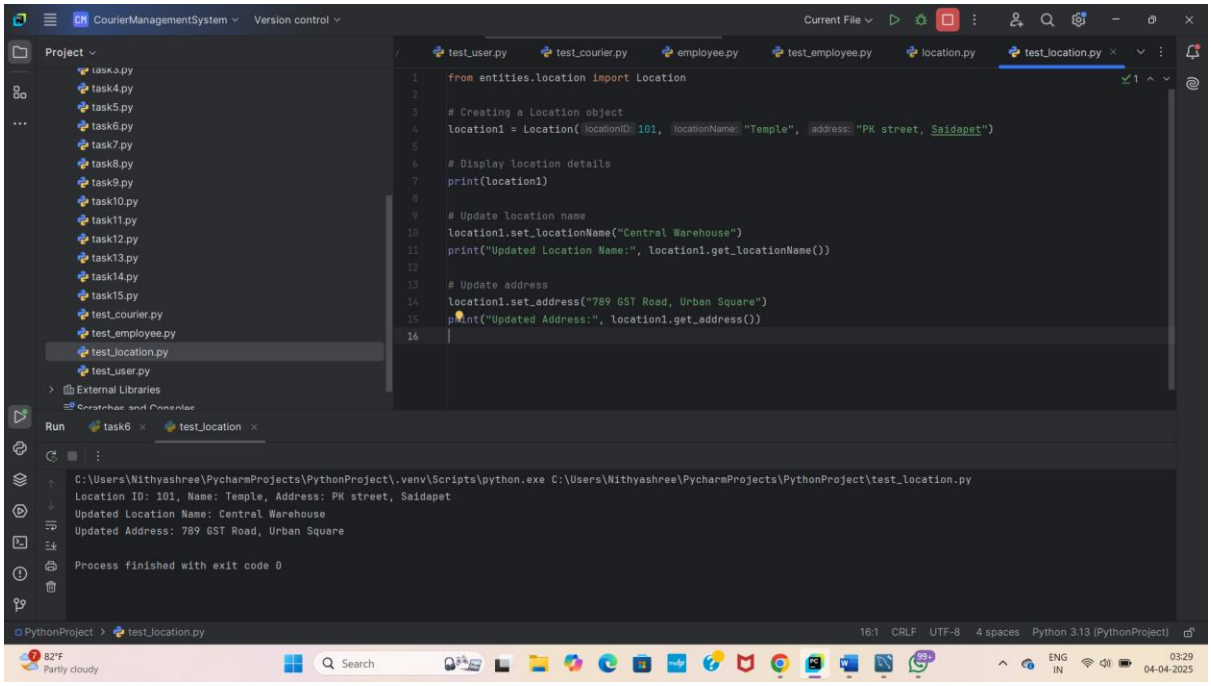


4. Location Class Variables LocationID , LocationName , Address

```
class Location:
    def __init__(self, locationID, locationName, address):
        self.__locationID = locationID
        self.__locationName = locationName
        self.__address = address
    def get_locationID(self):
        return self.__locationID
    def get_locationName(self):
        return self.__locationName
    def get_address(self):
        return self.__address
    def set_locationName(self, locationName):
        self.__locationName = locationName
    def set_address(self, address):
        self.__address = address
    def __str__(self):
        return f"Location ID: {self.__locationID}, Name: {self.__locationName}, Address: {self.__address}"
```

“IMPORTING LOCATION CLASS”

```
from entities.location import Location
location1 = Location(101, "Temple", "PK street, Saidapet")
print(location1)
# Update location name
location1.set_locationName("Central Warehouse")
print("Updated Location Name:", location1.get_locationName())
location1.set_address("789 GST Road, Urban Square")
print("Updated Address:", location1.get_address())
```



5. CourierCompany Class Variables companyName , courierDetails -collection of Courier Objects, employeeDetailscollection of Employee Objects, locationDetails - collection of Location Objects.

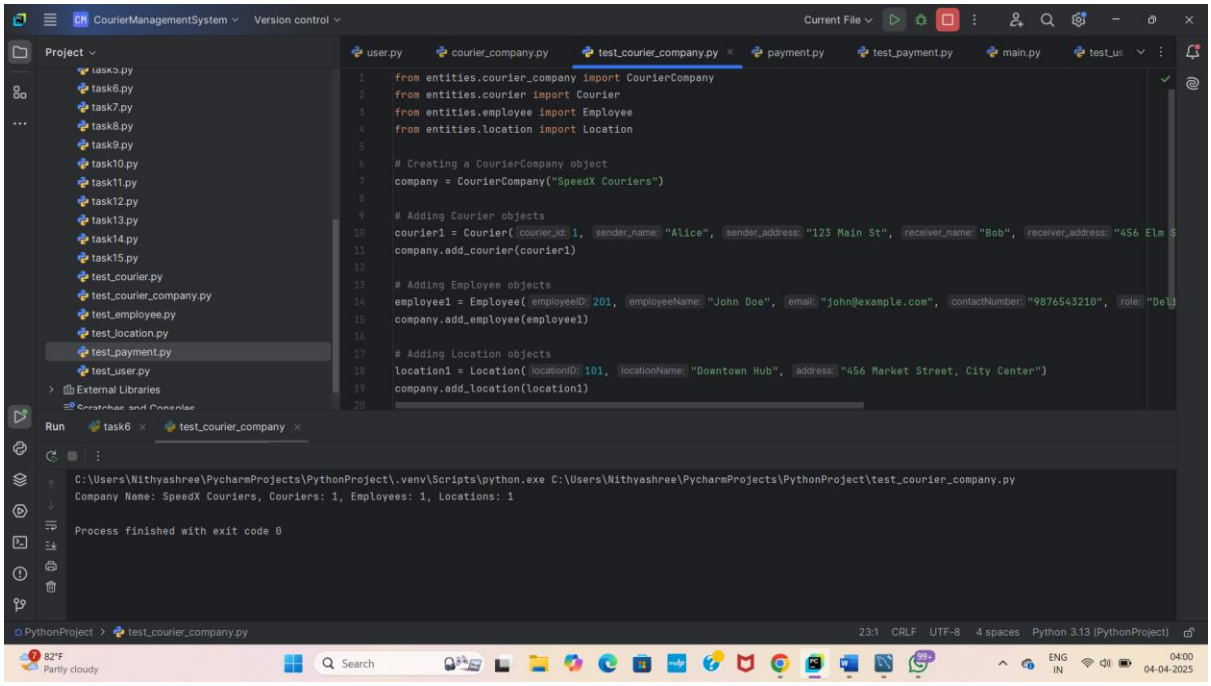
```
from entities.courier import Courier

from entities.employee import Employee

from entities.location import Location

class CourierCompany:

    def __init__(self, companyName):

        self.__companyName = companyName

        self.__courierDetails = []  # List to store Courier objects

        self.__employeeDetails = []  # List to store Employee objects

        self.__locationDetails = []  # List to store Location objects

    def get_companyName(self):

        return self.__companyName

    def get_courierDetails(self):
```

```
        return self.__courierDetails
    def get_employeeDetails(self):
        return self.__employeeDetails
    def get_locationDetails(self):
        return self.__locationDetails
    def set_companyName(self, companyName):
        self.__companyName = companyName
    def add_courier(self, courier):
        if isinstance(courier, Courier):
            self.__courierDetails.append(courier)
    def add_employee(self, employee):
        if isinstance(employee, Employee):
            self.__employeeDetails.append(employee)
    def add_location(self, location):
        if isinstance(location, Location):
            self.__locationDetails.append(location)
    def __str__(self):
        return f"Company Name: {self.__companyName}, " \
               f"Couriers: {len(self.__courierDetails)}, " \
               f"Employees: {len(self.__employeeDetails)}, " \
               f"Locations: {len(self.__locationDetails)}"
```

“IMPORTING COURIERCOMPANY CLASS”

```
from entities.courier import Courier
from entities.employee import Employee
from entities.location import Location
class CourierCompany:
    def __init__(self, companyName):
        self.__companyName = companyName
        self.__courierDetails = []  # List to store Courier objects
        self.__employeeDetails = []  # List to store Employee objects
        self.__locationDetails = []  # List to store Location objects
```

```
def get_companyName(self):
    return self.__companyName
def get_courierDetails(self):
    return self.__courierDetails
def get_employeeDetails(self):
    return self.__employeeDetails
def get_locationDetails(self):
    return self.__locationDetails
def set_companyName(self, companyName):
    self.__companyName = companyName
def add_courier(self, courier):
    if isinstance(courier, Courier):
        self.__courierDetails.append(courier)
def add_employee(self, employee):
    if isinstance(employee, Employee):
        self.__employeeDetails.append(employee)
def add_location(self, location):
    if isinstance(location, Location):
        self.__locationDetails.append(location)
def __str__(self):
    return f"Company Name: {self.__companyName}, " \
           f"Couriers: {len(self.__courierDetails)}, " \
           f"Employees: {len(self.__employeeDetails)}, " \
           f"Locations: {len(self.__locationDetails)}"
```



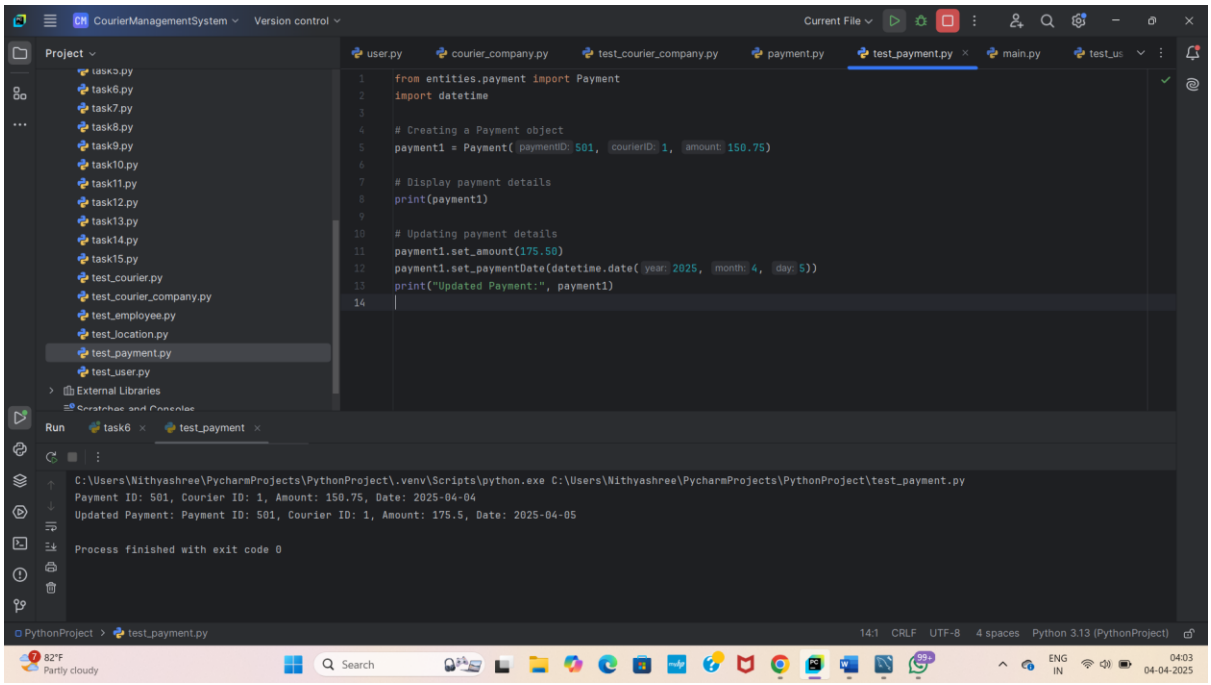
6. Payment Class: Variables PaymentID long, CourierID long, Amount double, PaymentDate Date

```
import datetime

class Payment:
    def __init__(self, paymentID, courierID, amount, paymentDate=None):
        self.__paymentID = paymentID
        self.__courierID = courierID
        self.__amount = amount
        self.__paymentDate = paymentDate if paymentDate else datetime.date.today()
    def get_paymentID(self):
        return self.__paymentID
    def get_courierID(self):
        return self.__courierID
    def get_amount(self):
        return self.__amount
    def get_paymentDate(self):
        return self.__paymentDate
    def set_paymentID(self, paymentID):
        self.__paymentID = paymentID
    def set_courierID(self, courierID):
```

```
        self.__courierID = courierID
    def set_amount(self, amount):
        self.__amount = amount
    def set_paymentDate(self, paymentDate):
        self.__paymentDate = paymentDate
    def __str__(self):
        return f"Payment ID: {self.__paymentID}, Courier ID: {self.__courierID}, Amount: {self.__amount}, Date: {self.__paymentDate}"
```

“IMPORTING PAYMENT CLASS”

```
from entities.payment import Payment
import datetime
payment1 = Payment(501, 1, 150.75)
print(payment1)
payment1.set_amount(175.50)
payment1.set_paymentDate(datetime.date(2025, 4, 5))
print("Updated Payment:", payment1)
```



Task 6: Service Provider Interface /Abstract class Create 2 Interface /Abstract class ICourierUserService and ICourierAdminService interface

ICourierUserService {

// Customer-related functions

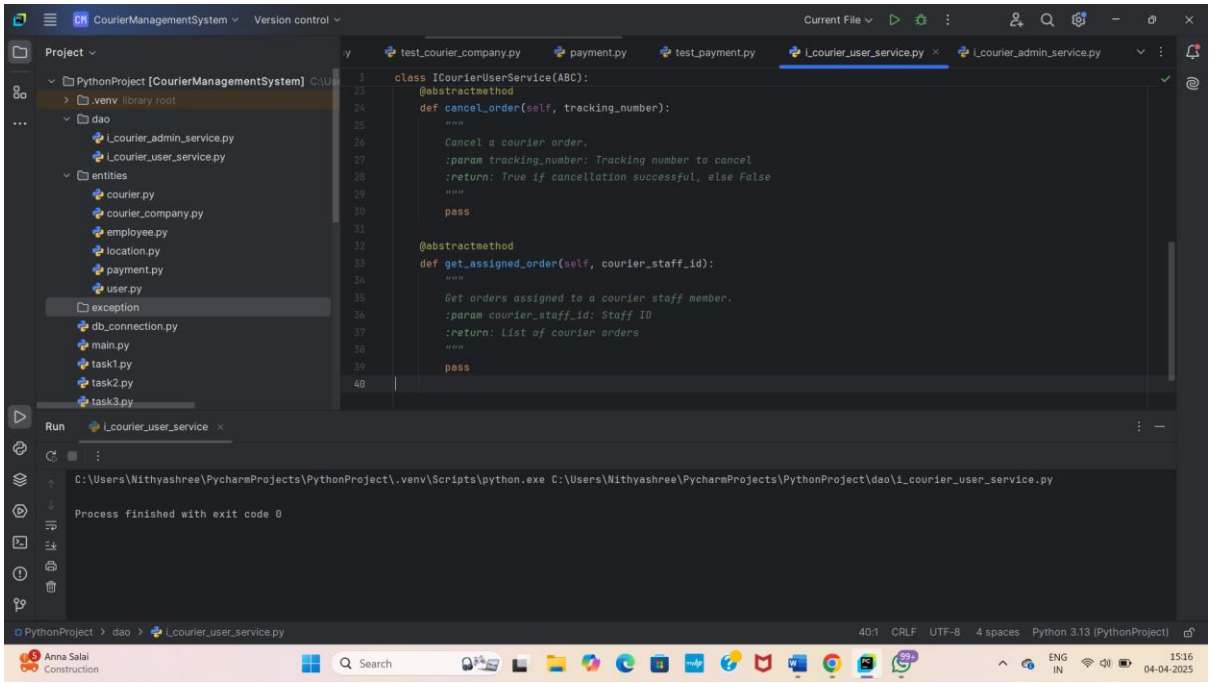
placeOrder()

getOrderStatus();

cancelOrder()

getAssignedOrder();

```
from abc import ABC, abstractmethod

class ICourierUserService(ABC):

    @abstractmethod
    def place_order(self, courier_obj):
        """
        Place a new courier order.
        :param courier_obj: Courier object created using user inputs
        :return: Unique tracking number for the order
        """
        pass

    @abstractmethod
    def get_order_status(self, tracking_number):
        """
        Get the status of a courier order.
        :param tracking_number: Tracking number of the order
        :return: Status string (e.g., 'In Transit', 'Delivered')
        """
        pass
    @abstractmethod
    def cancel_order(self, tracking_number):
        """
        Cancel a courier order.
        :param tracking_number: Tracking number to cancel
        :return: True if cancellation successful, else False
        """
        pass
    @abstractmethod
    def get_assigned_order(self, courier_staff_id):
        """
        Get orders assigned to a courier staff member.
        :param courier_staff_id: Staff ID
        :return: List of courier orders
        """
        pass
```

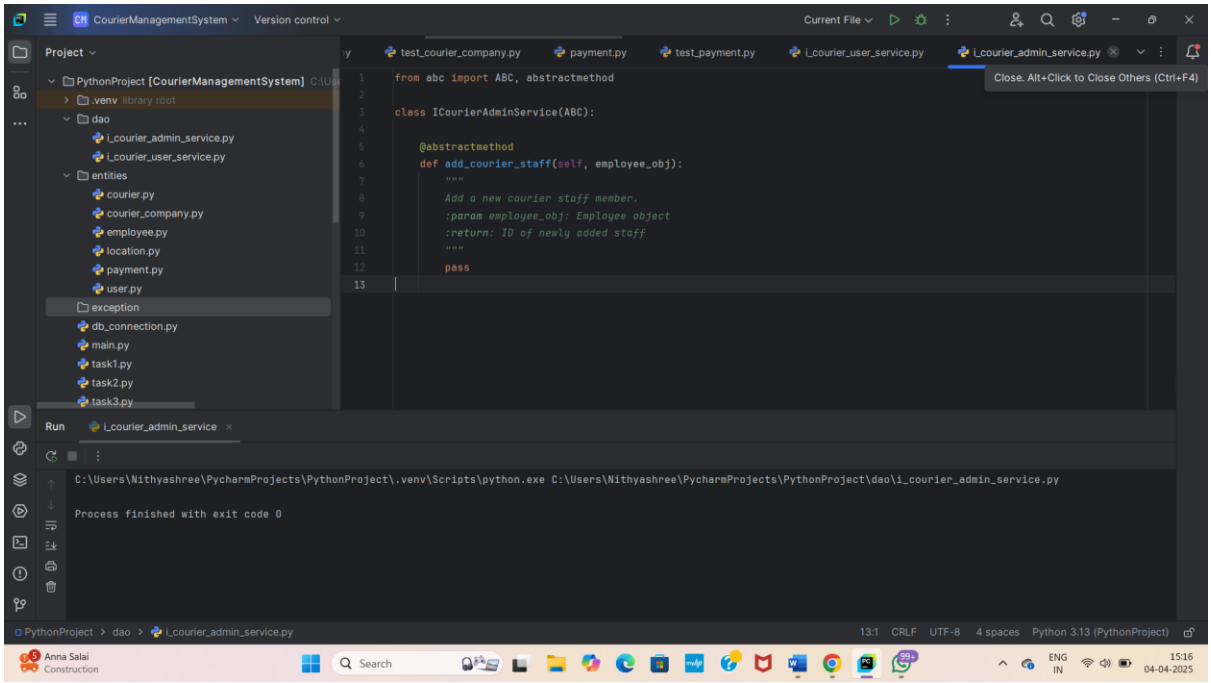


// Admin functions

ICourierAdminService

int addCourierStaff(Employee obj);

```
from abc import ABC, abstractmethod

class ICourierAdminService(ABC):
    @abstractmethod
    def add_courier_staff(self, employee_obj):
        """
        Add a new courier staff member.
        :param employee_obj: Employee object
        :return: ID of newly added staff
        """
        pass
```

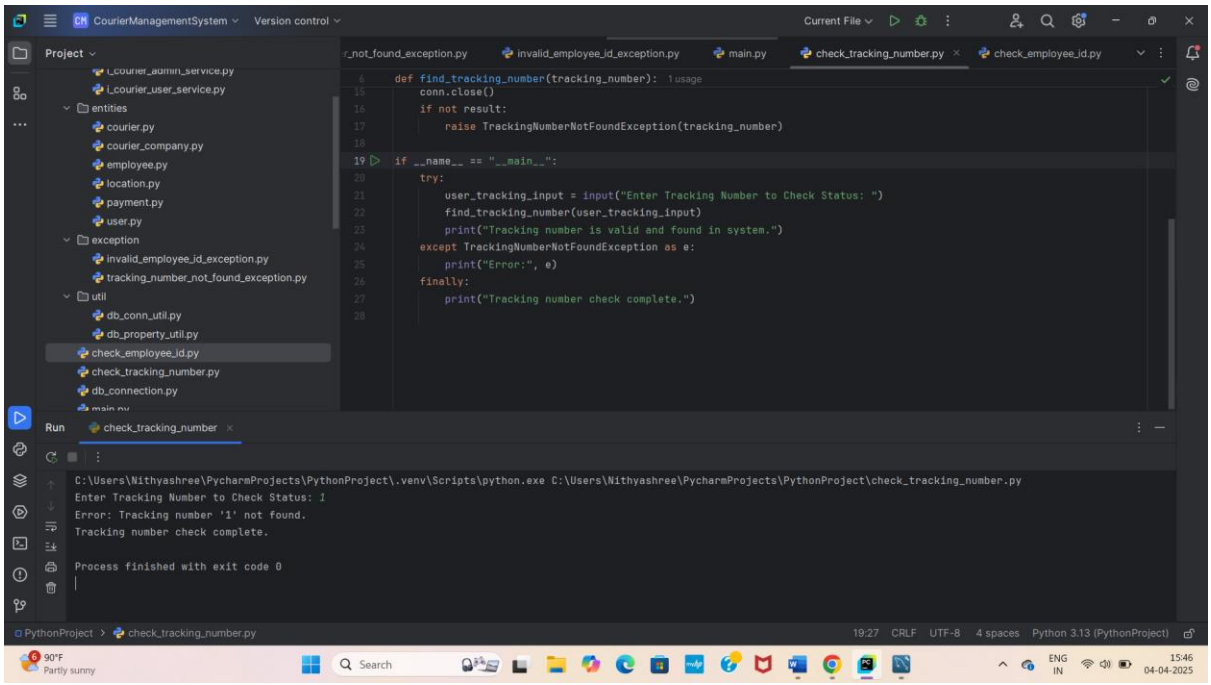


Task 7: Exception Handling (Scope: User Defined Exception/Checked /Unchecked Exception/Exception handling using try..catch finally,thow & throws keyword usage)

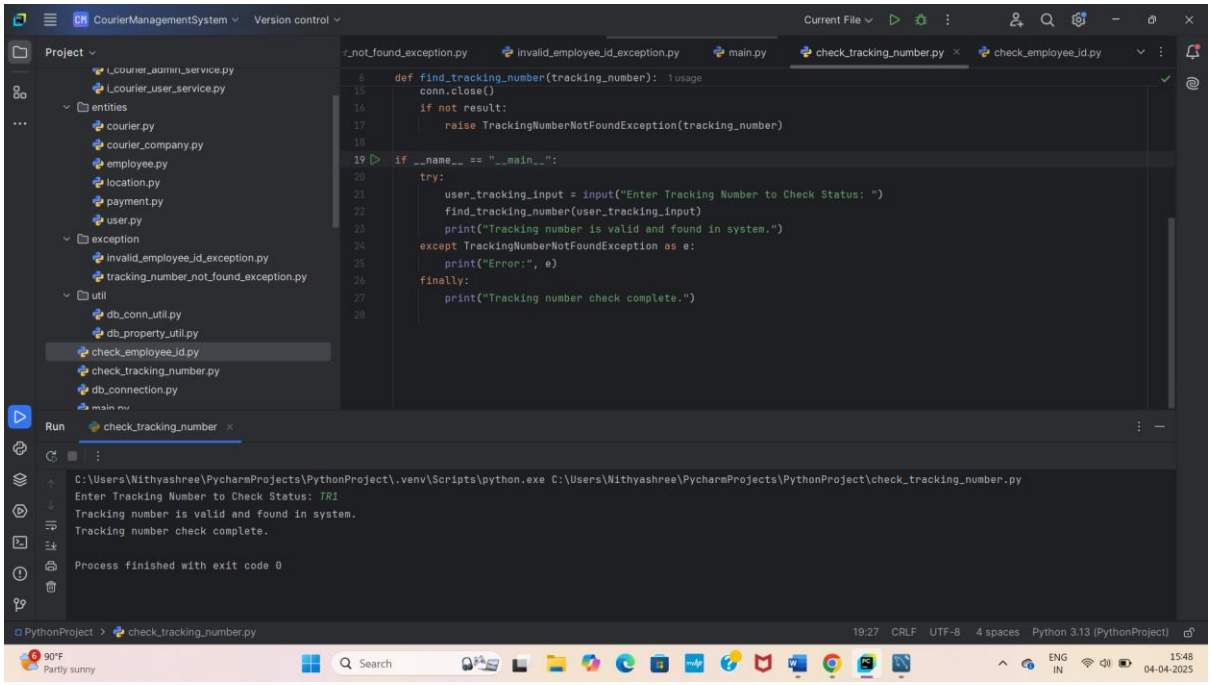
1. TrackingNumberNotFoundException :throw this exception when user try to withdraw amount or transfer amount to another acco

```
from db_connection import get_db_connection
from exception.tracking_number_not_found_exception import TrackingNumberNotFoundException
def find_tracking_number(tracking_number):
    conn = get_db_connection()
    if not conn:
        print("Failed to connect to DB.")
        return
    cursor = conn.cursor()
    cursor.execute("SELECT trackingNumber FROM Courier WHERE trackingNumber = %s",
(tracking_number,))
    result = cursor.fetchone()
    cursor.close()
    conn.close()
    if not result:
        raise TrackingNumberNotFoundException(tracking_number)
if __name__ == "__main__":
    try:
        user_tracking_input = input("Enter Tracking Number to Check Status: ")
        find_tracking_number(user_tracking_input)
        print("Tracking number is valid and found in system.")
    except TrackingNumberNotFoundException as e:
        print("Error:", e)
```

```
finally:
    print("Tracking number check complete.")
```



THE ABOVE IMAGE SHOWS THE EXCEPTION THROWED FOR INVALID INPUT OF THE TRACKING NUMBER

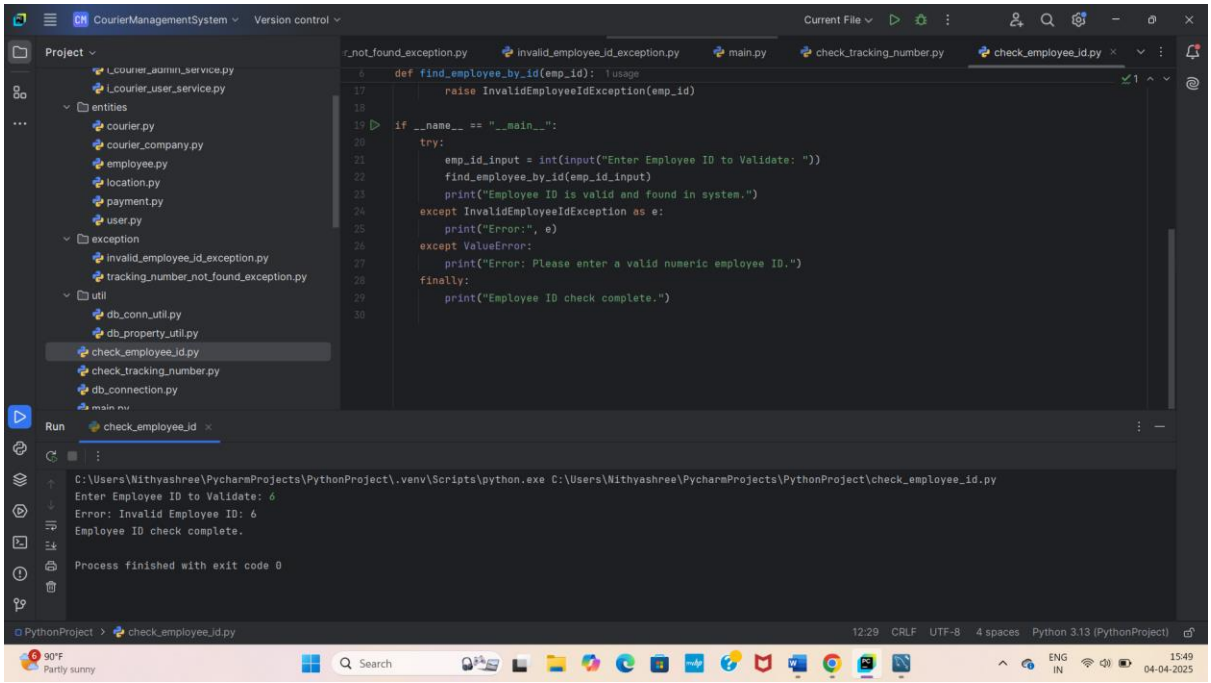


THE ABOVE IMAGE SHOWS THE OUTPUT GIVEN FOR VALID INPUT OF THE TRACKING NUMBER

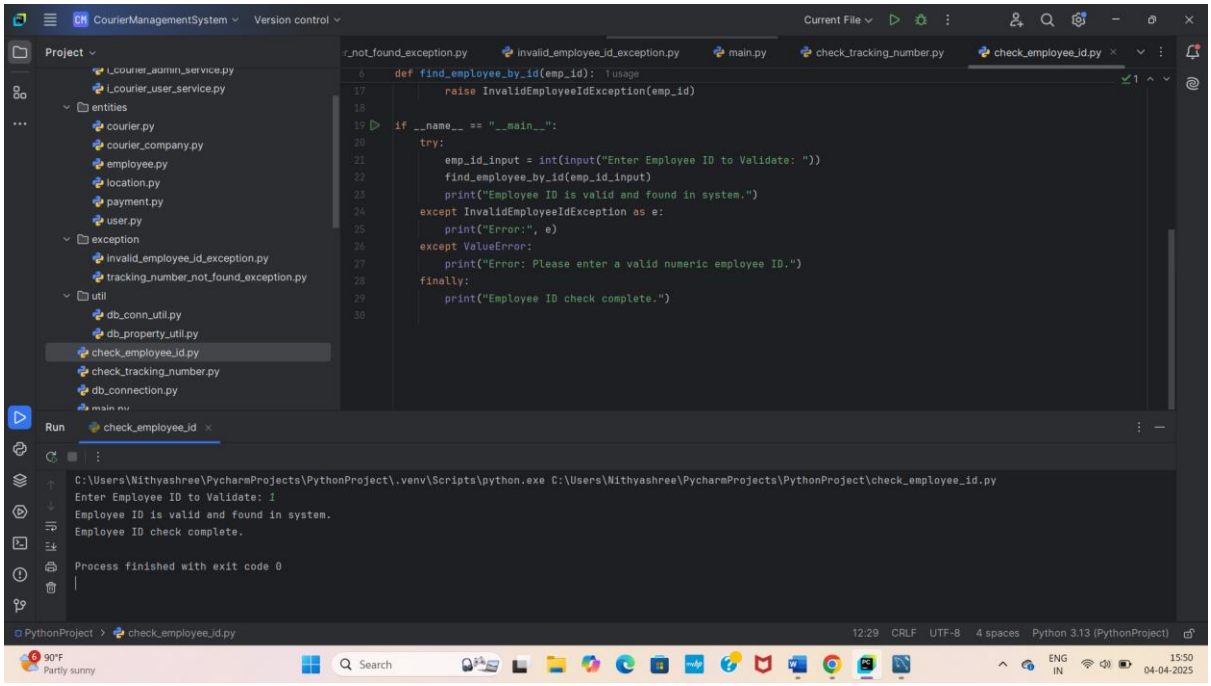
2. InvalidEmployeeIdException throw this exception when id entered for the employee not existing in the system

```
from db_connection import get_db_connection

from exception.invalid_employee_id_exception import InvalidEmployeeIdException
```

```
def find_employee_by_id(emp_id):
    conn = get_db_connection()
    if not conn:
        print("Failed to connect to DB.")
        return
    cursor = conn.cursor()
    cursor.execute("SELECT employeeID FROM Employee WHERE employeeID = %s", (emp_id,))
    result = cursor.fetchone()
    cursor.close()
    conn.close()
    if not result:
        raise InvalidEmployeeIdException(emp_id)
if __name__ == "__main__":
    try:
        emp_id_input = int(input("Enter Employee ID to Validate: "))
        find_employee_by_id(emp_id_input)
        print(" Employee ID is valid and found in system.")
    except InvalidEmployeeIdException as e:
        print("Error:", e)
    except ValueError:
        print("Error: Please enter a valid numeric employee ID.")
    finally:
        print("Employee ID check complete.")
```



THE ABOVE IMAGE SHOWS THE EXCEPTION THROWED FOR INVALID INPUT OF THE EMPLOYEE ID



THE ABOVE IMAGE SHOWS OUTPUT GIVEN FOR VALID INPUT OF THE EMPLOYEE ID

Task 8: Collections

1. Create a new model named CourierCompanyCollection in entity package replacing the Array of Objects with List to accommodate dynamic updates in the CourierCompany class

from entities.courier import Courier

from entities.employee import Employee

from entities.location import Location

```
class CourierCompanyCollection:
    def __init__(self, company_name):
        self.company_name = company_name
        self.courier_details = []  # List of Courier objects
        self.employee_details = []  # List of Employee objects
        self.location_details = []  # List of Location objects
    def add_courier(self, courier: Courier):
        self.courier_details.append(courier)
    def add_employee(self, employee: Employee):
        self.employee_details.append(employee)
    def add_location(self, location: Location):
        self.location_details.append(location)
    def get_all_couriers(self):
        return self.courier_details
    def get_all_employees(self):
        return self.employee_details
    def get_all_locations(self):
        return self.location_details
    def __str__(self):
        return f"Company Name: {self.company_name}\n" \
               f"Total Couriers: {len(self.courier_details)}\n" \
               f"Total Employees: {len(self.employee_details)}\n" \
               f"Total Locations: {len(self.location_details)}"
```

2. Create a new implementation class CourierUserServiceCollectionImpl class in package dao which implements ICourierUserService interface which holds a variable named companyObj of type CourierCompanyCollection

```
from dao.i_courier_user_service import ICourierUserService
from entities.courier_company_collection import CourierCompanyCollection
class CourierUserServiceCollectionImpl(ICourierUserService):
    def __init__(self, company_obj: CourierCompanyCollection):
        self.company_obj = company_obj
```

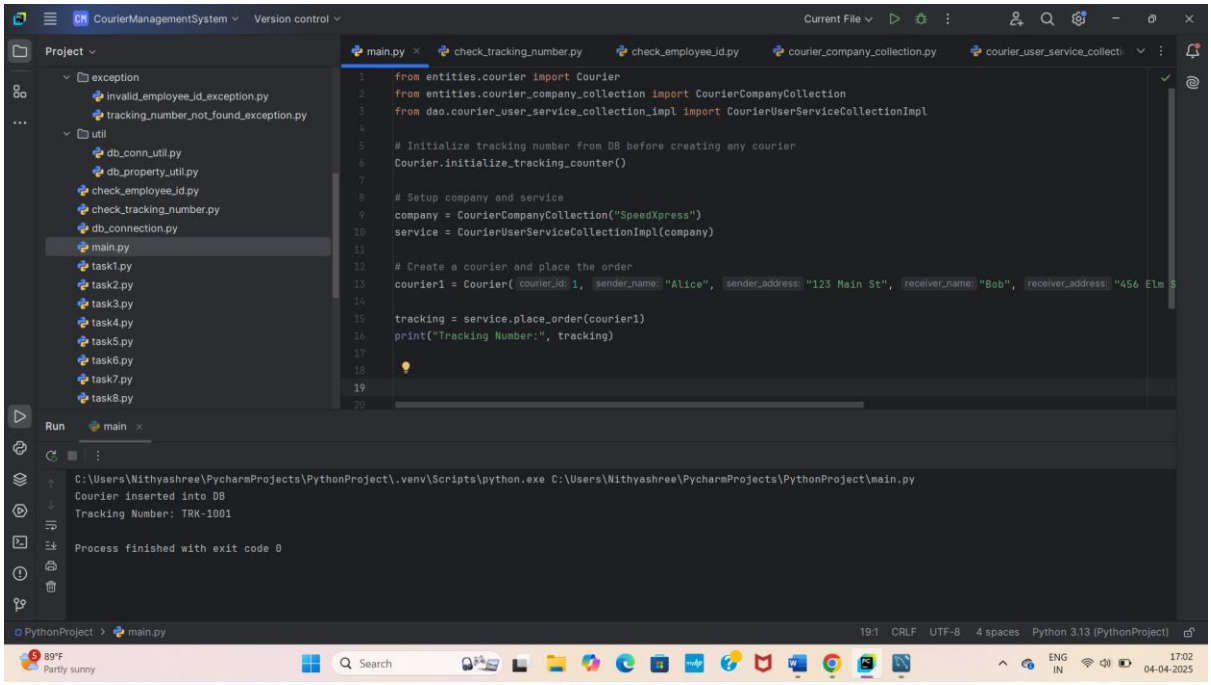
```
        self.tracking_number_seed = 1000  # Static starting point
    def place_order(self, courier_obj):
        self.tracking_number_seed += 1
        courier_obj.set_tracking_number(f"TRK{self.tracking_number_seed}")
        self.company_obj.add_courier(courier_obj)
        return courier_obj.get_tracking_number()
    def get_order_status(self, tracking_number):
        for courier in self.company_obj.get_all_couriers():
            if courier.get_tracking_number() == tracking_number:
                return courier.get_status()
        return "Tracking number not found"
    def cancel_order(self, tracking_number):
        for courier in self.company_obj.get_all_couriers():
            if courier.get_tracking_number() == tracking_number:
                self.company_obj.courier_details.remove(courier)
                return True
        return False
    def get_assigned_order(self, courier_staff_id):
        assigned_orders = [
            c for c in self.company_obj.get_all_couriers()
            if c.get_sender_contact() == courier_staff_id  # Adjust as per actual logic
        ]
        return assigned_orders
```

By themselves, these two classes (CourierCompanyCollection and CourierUserServiceCollectionImpl) won't give any output unless we call them explicitly, like in a test or main module.

Calling them in main.py:

```
from entities.courier import Courier
from entities.courier_company_collection import CourierCompanyCollection
from dao.courier_user_service_collection_impl import CourierUserServiceCollectionImpl
# Initialize tracking number from DB before creating any courier
Courier.initialize_tracking_counter()
company = CourierCompanyCollection("SpeedXpress")
service = CourierUserServiceCollectionImpl(company)
```

```
# Create a courier and place the order
courier1 = Courier(1, "Alice", "123 Main St", "Bob", "456 Elm St", 2.5, "In Transit", "2025-04-10", 101)
tracking = service.place_order(courier1)
print("Tracking Number:", tracking)
```



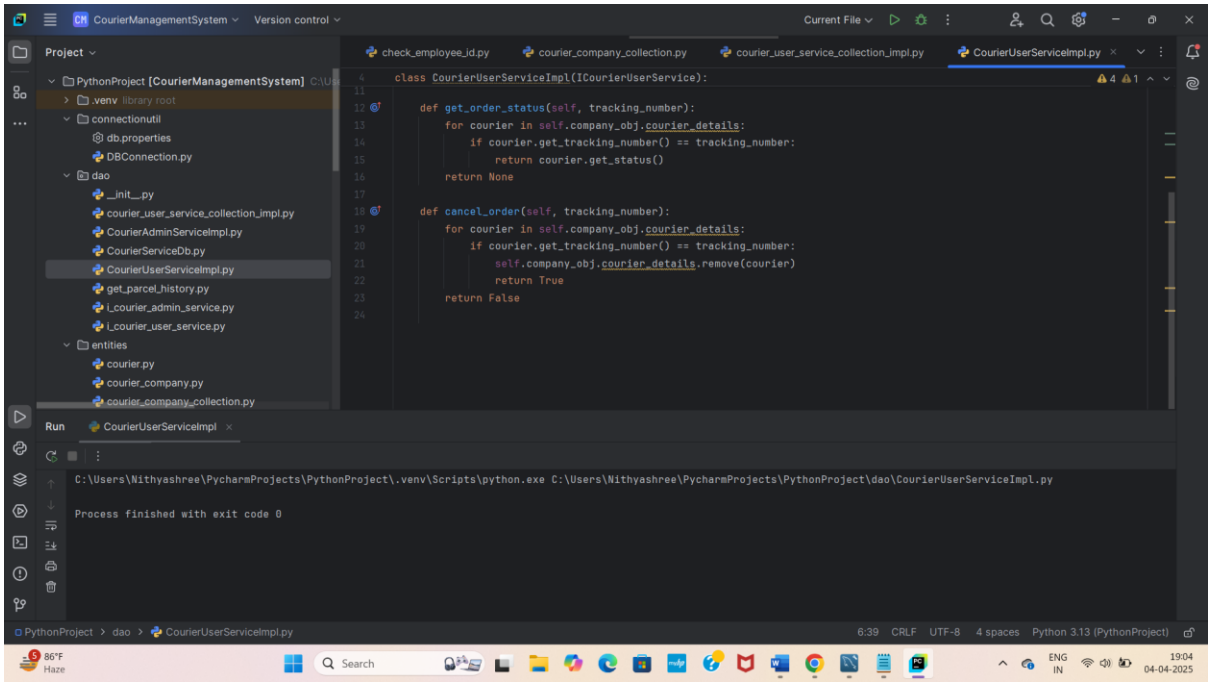
Task 9: Service implementation

1.Create CourierUserServiceImpl class which implements ICourierUserService interface which holds a variable named companyObj of type CourierCompany. This variable can be used to access the Object Arrays to access data relevant in method implementations.

CourierUserServiceImpl (in dao package)

```
from dao.i_courier_user_service import ICourierUserService
from entities.courier_company import CourierCompany
class CourierUserServiceCollectionImpl(ICourierUserService):
    def __init__(self, company_obj: CourierCompany):
        self.company_obj = company_obj
    def place_order(self, courier):
        self.company_obj.courier_details.append(courier)
        return courier.get_tracking_number()
    def get_order_status(self, tracking_number):
        for courier in self.company_obj.courier_details:
            if courier.get_tracking_number() == tracking_number:
                return courier.get_status()
        return None
```

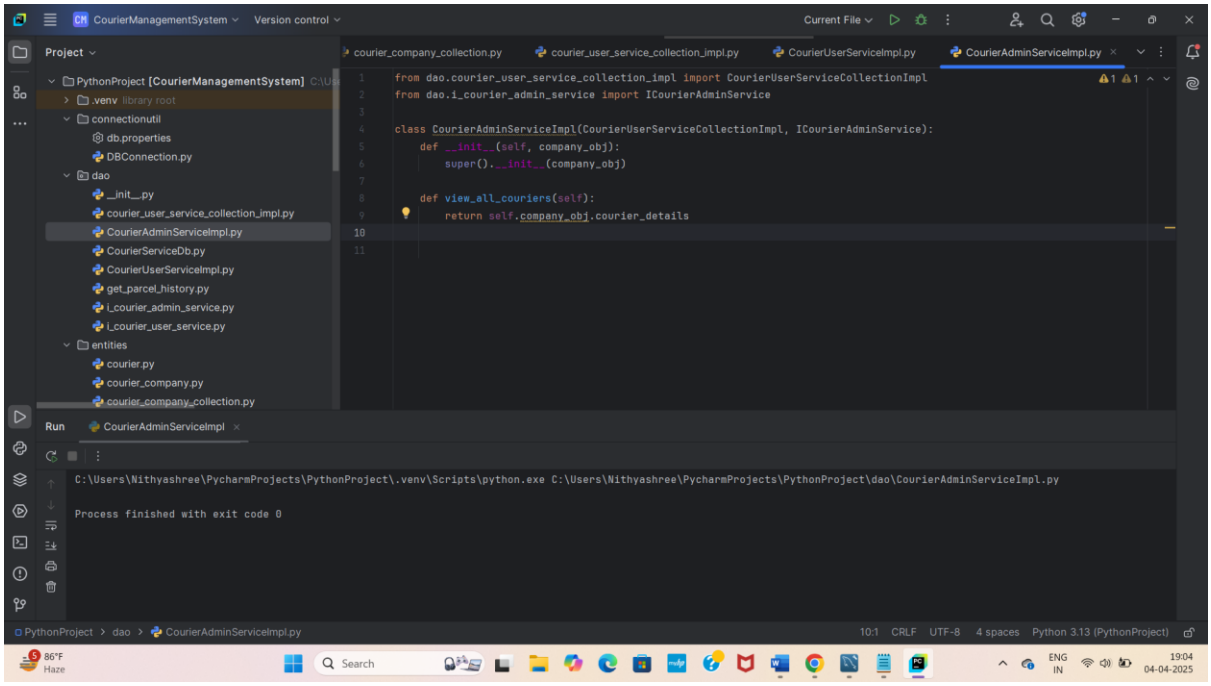
```
def cancel_order(self, tracking_number):
    for courier in self.company_obj.courier_details:
        if courier.get_tracking_number() == tracking_number:
            self.company_obj.courier_details.remove(courier)
            return True
    return False
```



2.Create CourierAdminService Impl class which inherits from CourierUserServiceImpl and implements ICourierAdminService interface.

CourierAdminServiceImpl (in dao package)

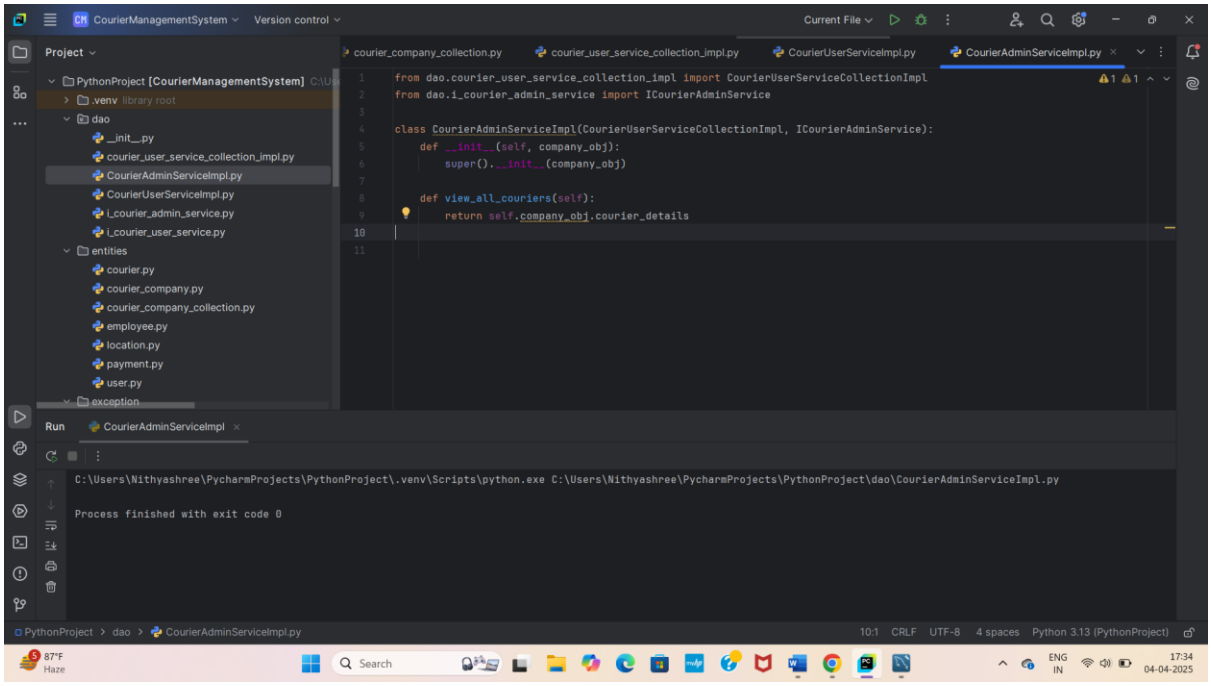
```
from dao.courier_user_service_impl import CourierUserServiceImpl
from dao.i_courier_admin_service import ICourierAdminService
class CourierAdminServiceImpl(CourierUserServiceImpl, ICourierAdminService):
    def __init__(self, company_obj):
        super().__init__(company_obj)
    def view_all_couriers(self):
        return self.company_obj.courier_details
```



3. Create CourierAdminServiceCollectionImpl class which inherits from CourierUserServiceColectionImpl and implements ICourierAdminService interface.

CourierAdminServiceCollectionImpl (in dao package)

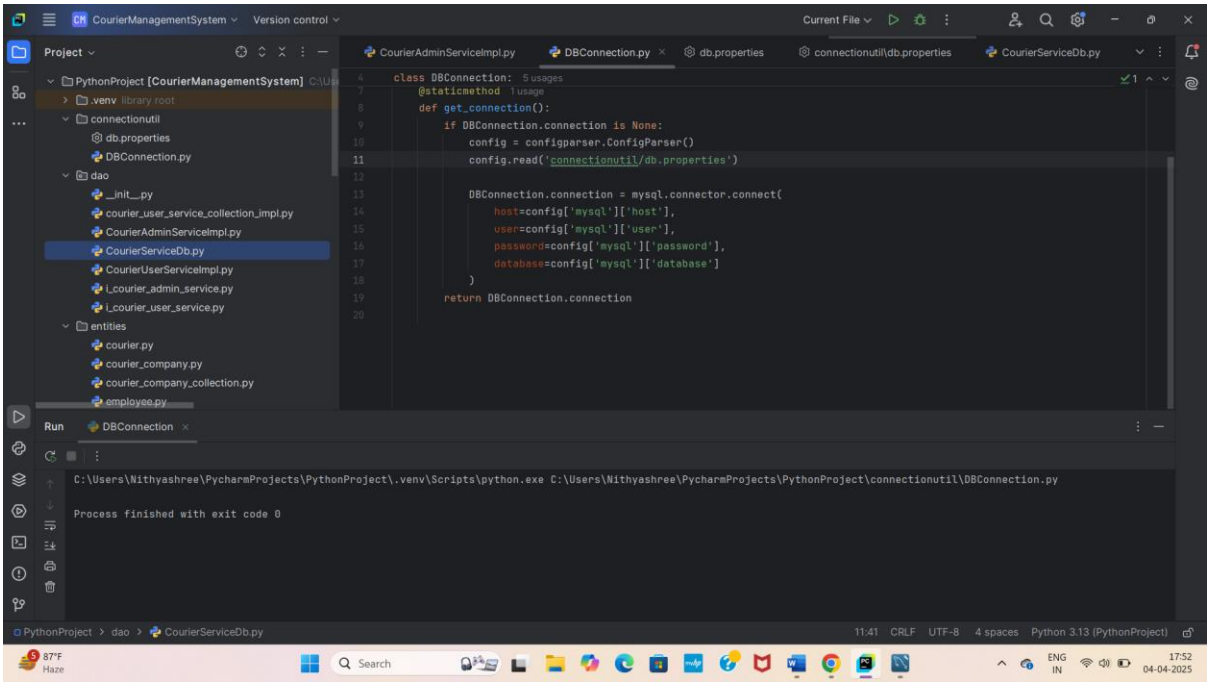
```
from dao.courier_user_service_collection_impl import CourierUserServiceCollectionImpl

from dao.i_courier_admin_service import ICourierAdminService

class CourierAdminServiceCollectionImpl(CourierUserServiceCollectionImpl, ICourierAdminService):

    def __init__(self, company_obj):

        super().__init__(company_obj)

    def view_all_couriers(self):

        return self.company_obj.courier_details
```



Task 10: Database Interaction

1. Write code to establish a connection to your SQL database.

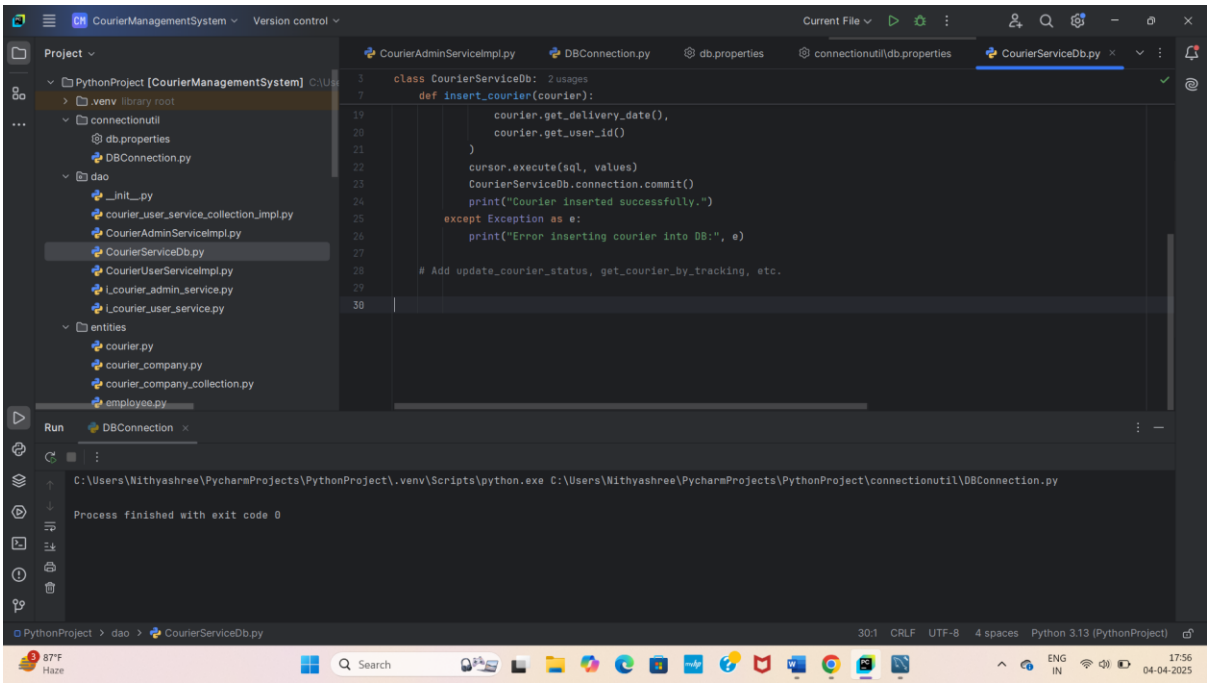
```
import mysql.connector
import configparser
class DBConnection:
    connection = None
    @staticmethod
    def get_connection():
        if DBConnection.connection is None:
            config = configparser.ConfigParser()
            config.read('db.properties')
            DBConnection.connection = mysql.connector.connect(
                host=config['mysql']['host'],
                user=config['mysql']['user'],
                password=config['mysql']['password'],
                database=config['mysql']['database']
            )
        return DBConnection.connection
```



2. Create a Service class CourierServiceDb in dao with a static variable named connection of type Connection which can be assigned in the constructor by invoking the method in DBConnection Class.

```
from connectionutil.DBConnection import DBConnection
class CourierServiceDb:
   connection = DBConnection.get_connection()
   @staticmethod
   def insert_courier(courier):
      try:
         cursor = CourierServiceDb.connection.cursor()
         sql = "INSERT INTO courier (SenderName, SenderAddress, ReceiverName,
ReceiverAddress, Weight, Status, TrackingNumber, DeliveryDate, UserID) VALUES (%s, %s, %s,
%s, %s, %s, %s, %s, %s)"
         values = (
            courier.get_sender_name(),
            courier.get_sender_address(),
            courier.get_receiver_name(),
            courier.get_receiver_address(),
            courier.get_weight(),
            courier.get_status(),
            courier.get_tracking_number(),
            courier.get_delivery_date(),
            courier.get_user_id()
         )
         cursor.execute(sql, values)
         CourierServiceDb.connection.commit()
         print("Courier inserted successfully.")
      except Exception as e:
```

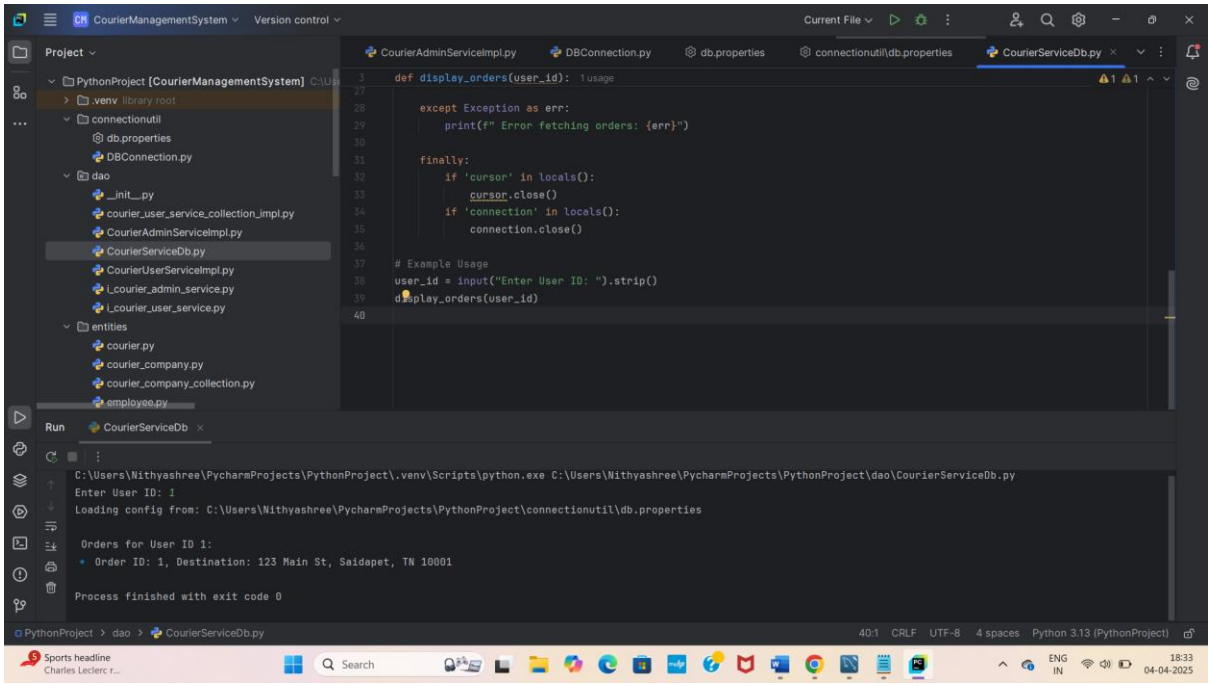
```
            print("Error inserting courier into DB:", e)
    # Add update_courier_status, get_courier_by_tracking, etc.
```



3.Include methods to insert, update, and retrieve data from the database (e.g., inserting a new order, updating courier status).

```
from connectionutil.DBConnection import DBConnection

def display_orders(user_id):
    """
    Fetch and display all orders for a given user ID.
    """
    connection = DBConnection.get_connection()
    if connection is None:
        print(" Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        # Assuming Order table has OrderID, DeliveryAddress, and Status columns
        query = "SELECT OrderID, DeliveryAddress FROM Orders WHERE UserID = %s"
        cursor.execute(query, (user_id,))
        orders = cursor.fetchall()
        if not orders:
            print(f" No orders found for User ID {user_id}.")
            return
        print(f"\n Orders for User ID {user_id}:")
        for order in orders:
            print(f" Order ID: {order[0]}, Destination: {order[1]}")

    except Exception as err:
```

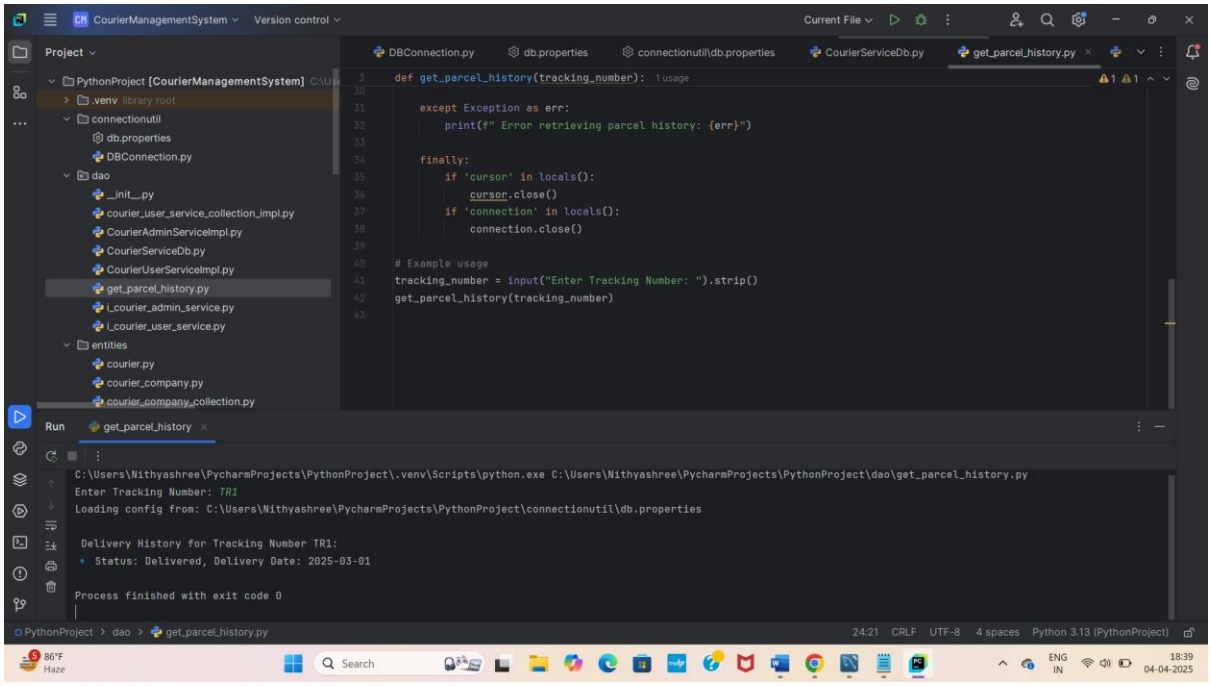
```
        print(f" Error fetching orders: {err}")

    finally:
        if 'cursor' in locals():
            cursor.close()
        if 'connection' in locals():
            connection.close()
# Example Usage
user_id = input("Enter User ID: ").strip()
display_orders(user_id)
```



4. Implement a feature to retrieve and display the delivery history of a specific parcel by querying the database.

```
from connectionutil.DBConnection import DBConnection

def get_parcel_history(tracking_number):
    """
    Retrieve and display the delivery history of a specific parcel using tracking number.
    """
    connection = DBConnection.get_connection()
    if connection is None:
        print(" Failed to connect to the database.")
        return

    try:
```

```
        cursor = connection.cursor()
        query = """
            SELECT TrackingNumber, Status, DeliveryDate
            FROM courier
            WHERE TrackingNumber = %s
        """
        cursor.execute(query, (tracking_number,))
        history = cursor.fetchall()
        if not history:
            print(f" No delivery history found for Tracking Number {tracking_number}.")
            return
        print(f"\n Delivery History for Tracking Number {tracking_number}:")
        for record in history:
            print(f" Status: {record[1]}, Delivery Date: {record[2]}")
    except Exception as err:
        print(f" Error retrieving parcel history: {err}")
    finally:
        if 'cursor' in locals():
            cursor.close()
        if 'connection' in locals():
            connection.close()
# Example usage
tracking_number = input("Enter Tracking Number: ").strip()
get_parcel_history(tracking_number)
```



5. Generate and display reports using data retrieved from the database (e.g., shipment status report, revenue report).

```
from connectionutil.DBConnection import DBConnection

def shipment_status_report():
    """
    Generate a shipment status report showing counts of each status.
    """
    connection = DBConnection.get_connection()
    if connection is None:
        print("Failed to connect to the database.")
        return
    try:
        cursor = connection.cursor()
        query = """
            SELECT Status, COUNT(*) AS Count
            FROM courier
            GROUP BY Status
        """
        cursor.execute(query)
        results = cursor.fetchall()
```

```
        print("\n Shipment Status Report:")
        for row in results:
            print(f" {row[0]}: {row[1]} shipments")
    except Exception as err:
        print(f" Error generating shipment status report: {err}")
    finally:
        if 'cursor' in locals():
            cursor.close()
        if 'connection' in locals():
            connection.close()
shipment_status_report()
```