ASSIGNMENT 1 PYTHON

NAME :- SUSHANT KUMAR SINGH

Project:- (TechShop, an electronic gadgets shop)

Implement OOPs

Task 1: Classes and Their Attributes:

You are working as a software developer for TechShop, a company that sells electronic gadgets. Your task is to design and implement an application using Object-Oriented Programming (OOP) principles to manage customer information, product details, and orders. Below are the classes you need to create:

Task 2: Class Creation:

• Create the classes (Customers, Products, Orders, OrderDetails and Inventory) with the specified attributes.

• Implement the constructor for each class to initialize its attributes.

• Implement methods as specified.

Customers Class:

Attributes:

• CustomerID (int)

• FirstName (string)

• LastName (string)

• Email (string)

• Phone (string)

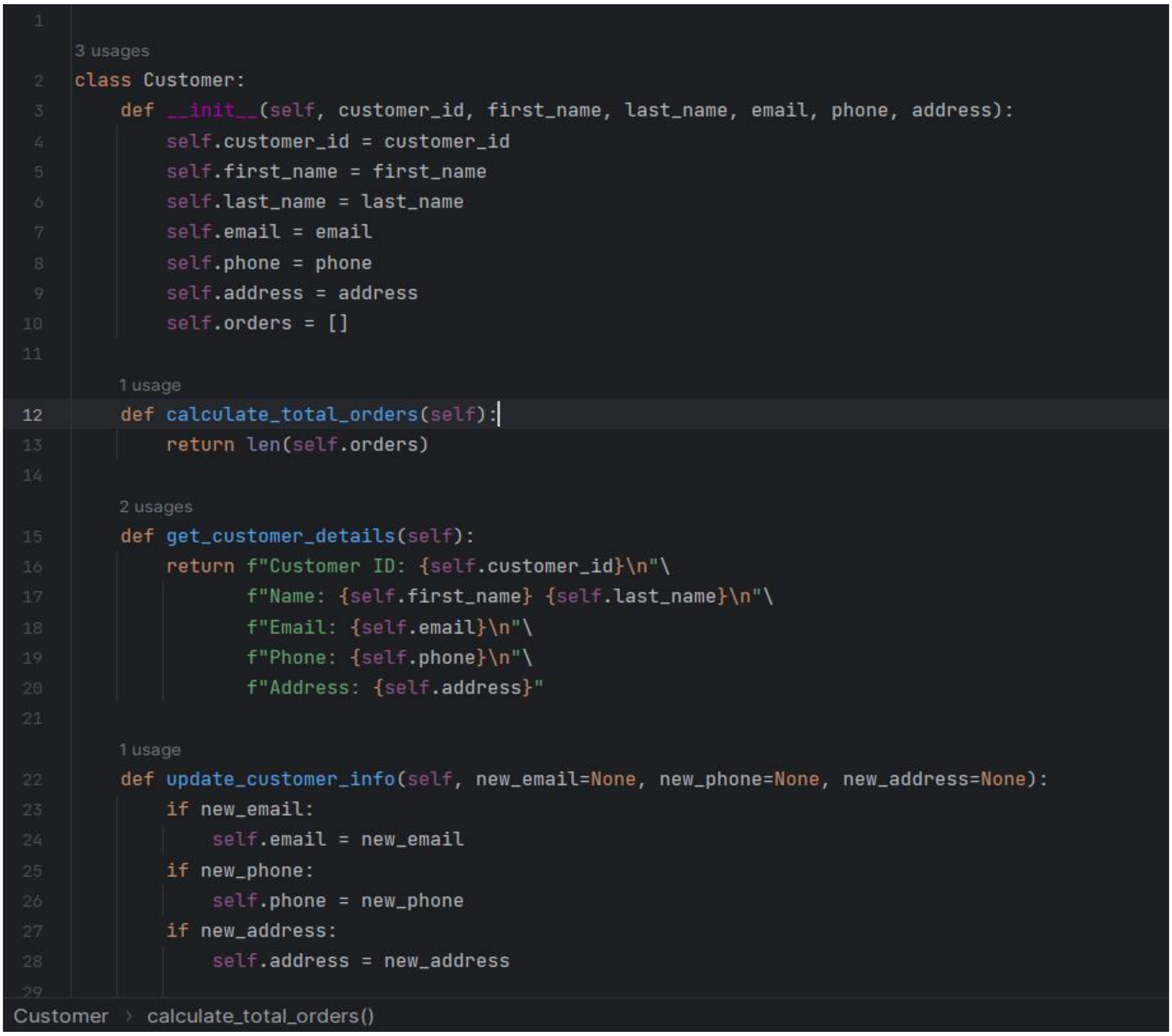
• Address (string)

Methods:

• CalculateTotalOrders(): Calculates the total number of orders placed by this customer.

• GetCustomerDetails(): Retrieves and displays detailed information about the customer.

• UpdateCustomerInfo(): Allows the customer to update their information (e.g., email, phone, or address).



```
3 usages
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address
        self.orders = []

    1 usage
    def calculate_total_orders(self):
        return len(self.orders)

    2 usages
    def get_customer_details(self):
        return f"Customer ID: {self.customer_id}\n"\
               f"Name: {self.first_name} {self.last_name}\n"\
               f"Email: {self.email}\n"\
               f"Phone: {self.phone}\n"\
               f"Address: {self.address}"

    1 usage
    def update_customer_info(self, new_email=None, new_phone=None, new_address=None):
        if new_email:
            self.email = new_email
        if new_phone:
            self.phone = new_phone
        if new_address:
            self.address = new_address
```

Products Class:

Attributes:

• ProductID (int)

• ProductName (string)

• Description (string)

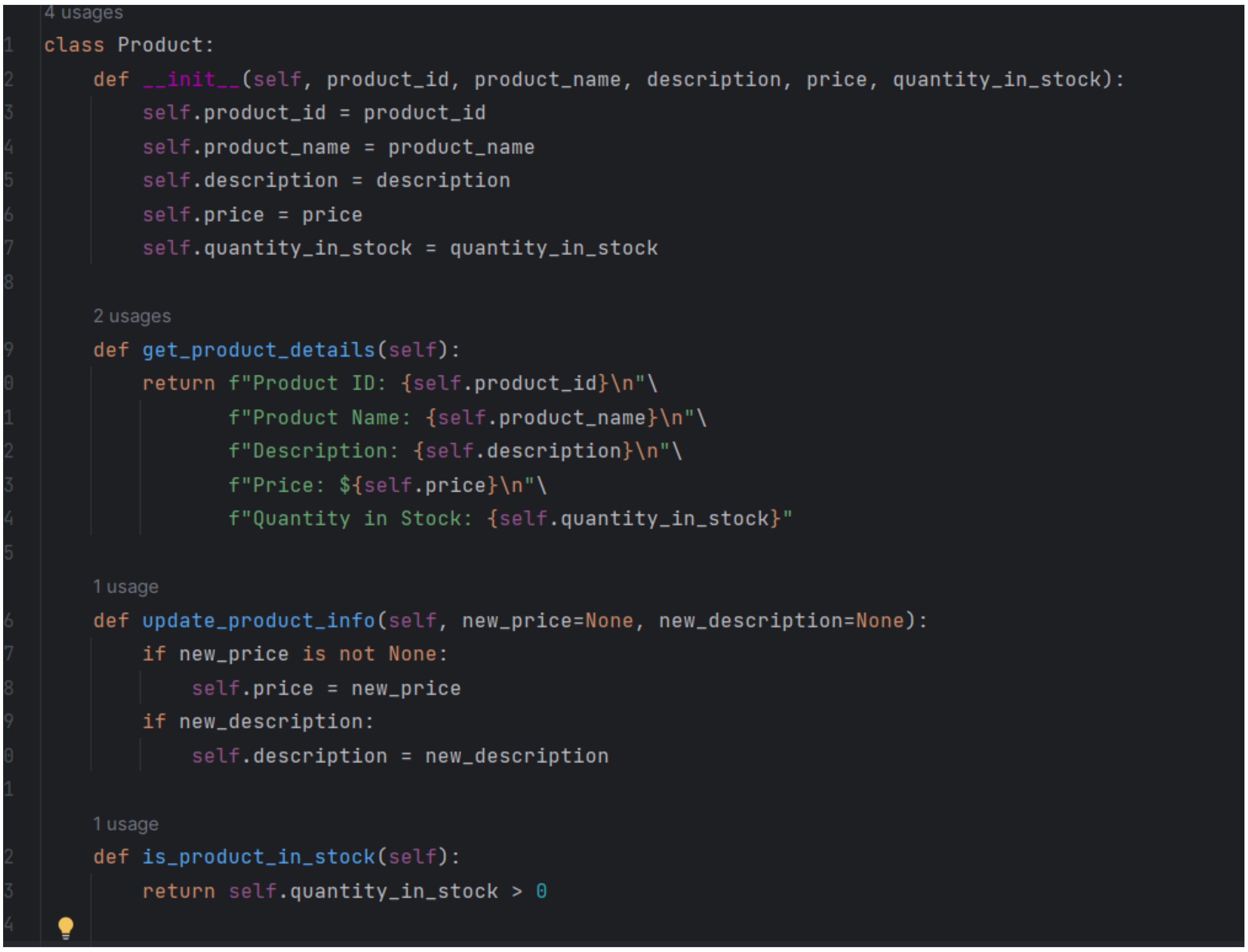
• Price (decimal)

Methods:

• GetProductDetails(): Retrieves and displays detailed information about the product.

• UpdateProductInfo(): Allows updates to product details (e.g., price, description).

• IsProductInStock(): Checks if the product is currently in stock.



```
4 usages
class Product:
    def __init__(self, product_id, product_name, description, price, quantity_in_stock):
        self.product_id = product_id
        self.product_name = product_name
        self.description = description
        self.price = price
        self.quantity_in_stock = quantity_in_stock

    2 usages
    def get_product_details(self):
        return f"Product ID: {self.product_id}\n"\
               f"Product Name: {self.product_name}\n"\
               f"Description: {self.description}\n"\
               f"Price: ${self.price}\n"\
               f"Quantity in Stock: {self.quantity_in_stock}"

    1 usage
    def update_product_info(self, new_price=None, new_description=None):
        if new_price is not None:
            self.price = new_price
        if new_description:
            self.description = new_description

    1 usage
    def is_product_in_stock(self):
        return self.quantity_in_stock > 0
```

Orders Class:

Attributes:

• OrderID (int)

• Customer (Customer) - Use composition to reference the Customer who placed the order.

• OrderDate (DateTime)

• TotalAmount (decimal)

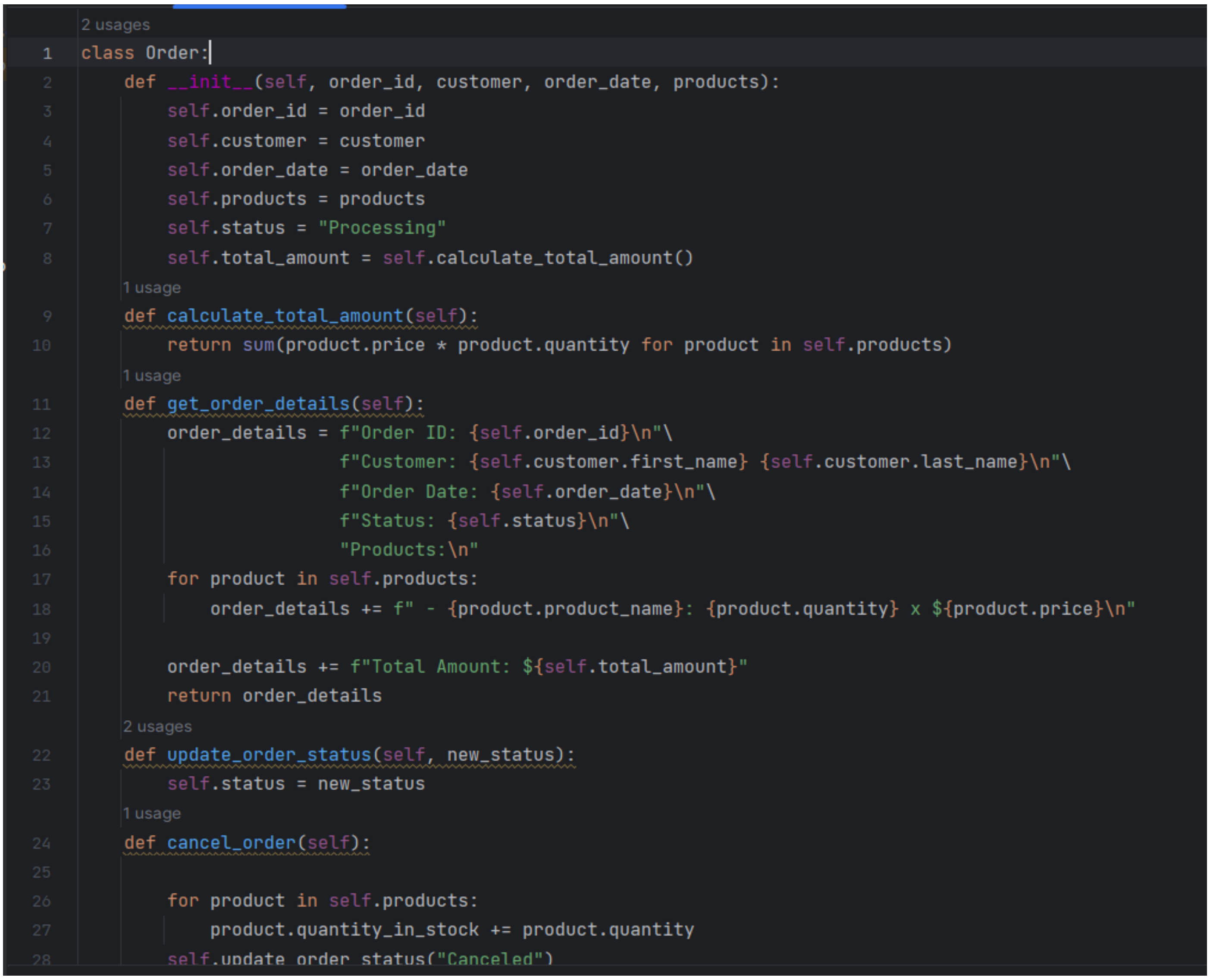
Methods:

• CalculateTotalAmount() - Calculate the total amount of the order.

• GetOrderDetails(): Retrieves and displays the details of the order (e.g., product list and quantities).

• UpdateOrderStatus(): Allows updating the status of the order (e.g., processing, shipped).

• CancelOrder(): Cancels the order and adjusts stock levels for products.



```python
2 usages
class Order:
    def __init__(self, order_id, customer, order_date, products):
        self.order_id = order_id
        self.customer = customer
        self.order_date = order_date
        self.products = products
        self.status = "Processing"
        self.total_amount = self.calculate_total_amount()
    1 usage
    def calculate_total_amount(self):
        return sum(product.price * product.quantity for product in self.products)
    1 usage
    def get_order_details(self):
        order_details = f"Order ID: {self.order_id}\n"\
                        f"Customer: {self.customer.first_name} {self.customer.last_name}\n"\
                        f"Order Date: {self.order_date}\n"\
                        f"Status: {self.status}\n"\
                        "Products:\n"
        for product in self.products:
            order_details += f" - {product.product_name}: {product.quantity} x ${product.price}\n"

        order_details += f"Total Amount: ${self.total_amount}"
        return order_details
    2 usages
    def update_order_status(self, new_status):
        self.status = new_status
    1 usage
    def cancel_order(self):

        for product in self.products:
            product.quantity_in_stock += product.quantity
        self.update_order_status("Canceled")
```

OrderDetails Class:

Attributes:

• OrderDetailID (int)

• Order (Order) - Use composition to reference the Order to which this detail belongs.

• Product (Product) - Use composition to reference the Product included in the order detail.

• Quantity (int)

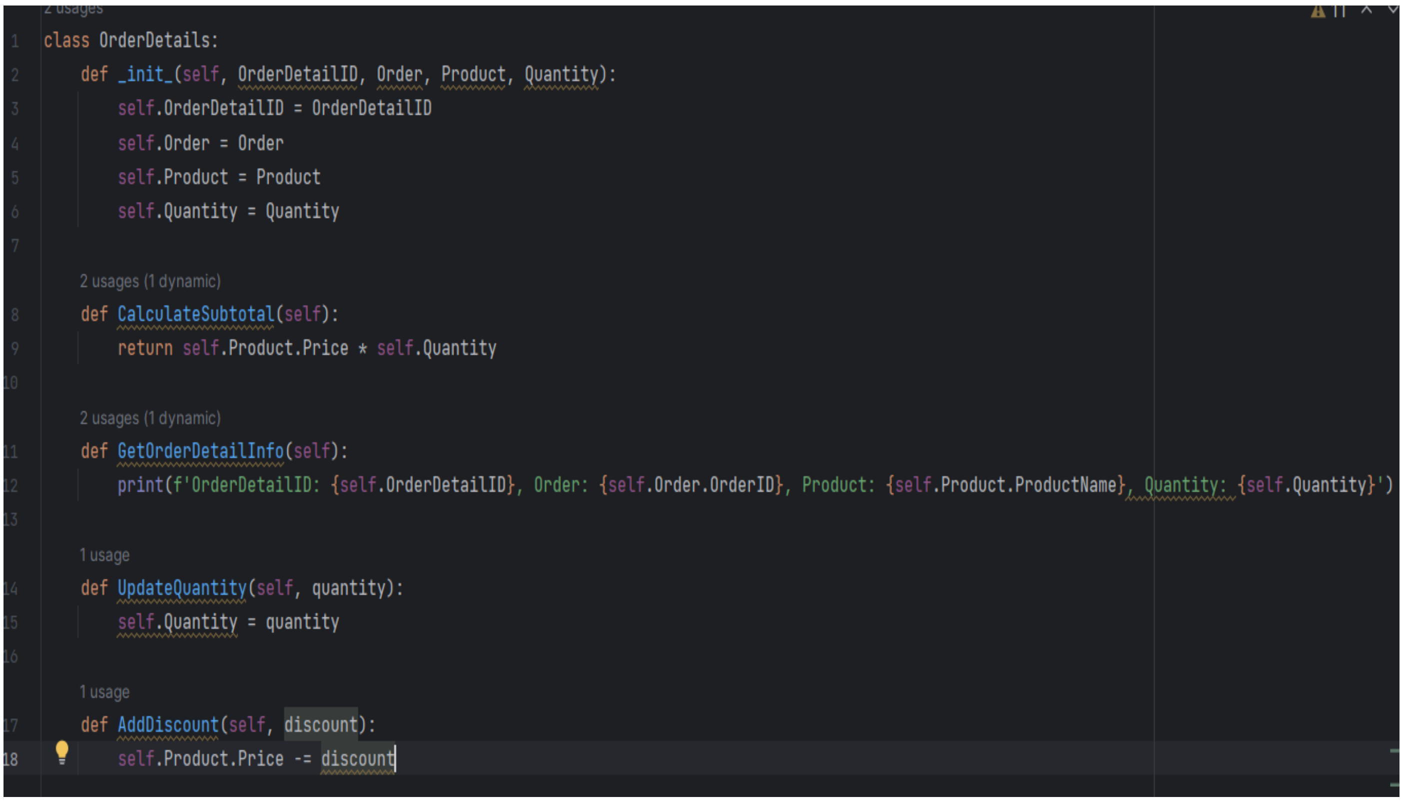
Methods:

• CalculateSubtotal() - Calculate the subtotal for this order detail.

• GetOrderDetailInfo(): Retrieves and displays information about this order detail.

• UpdateQuantity(): Allows updating the quantity of the product in this order detail.

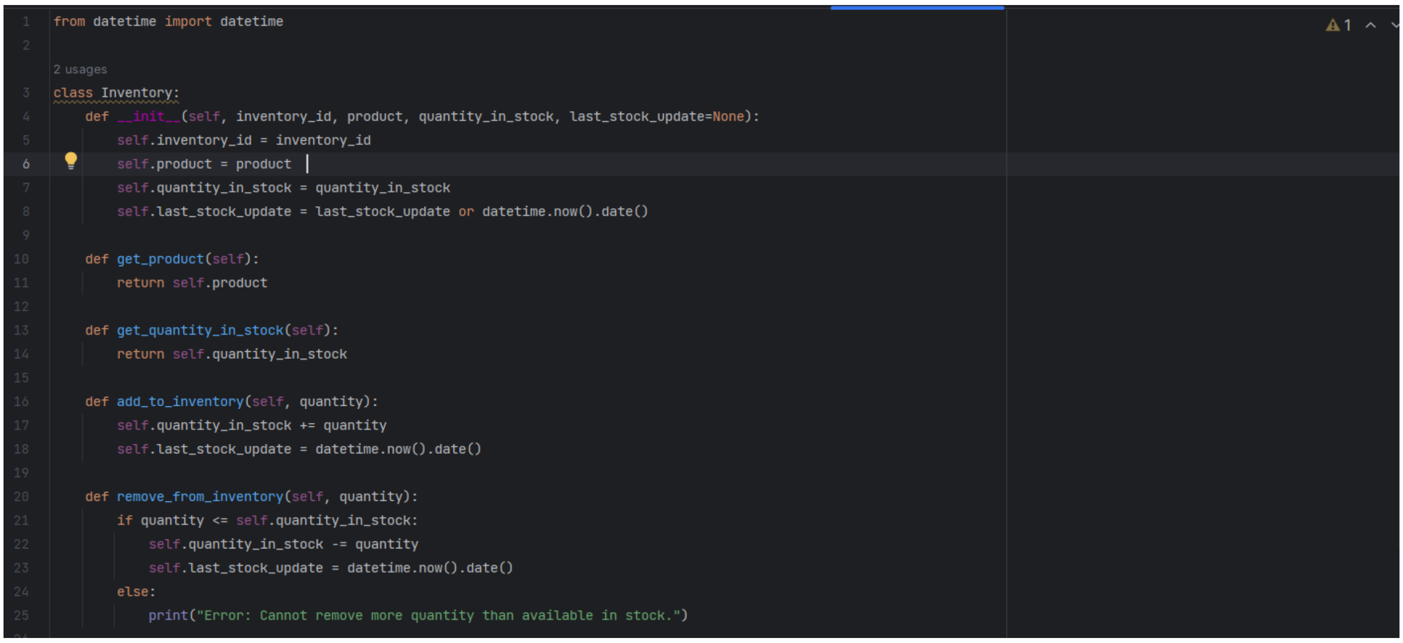
• AddDiscount(): Applies a discount to this order detail.



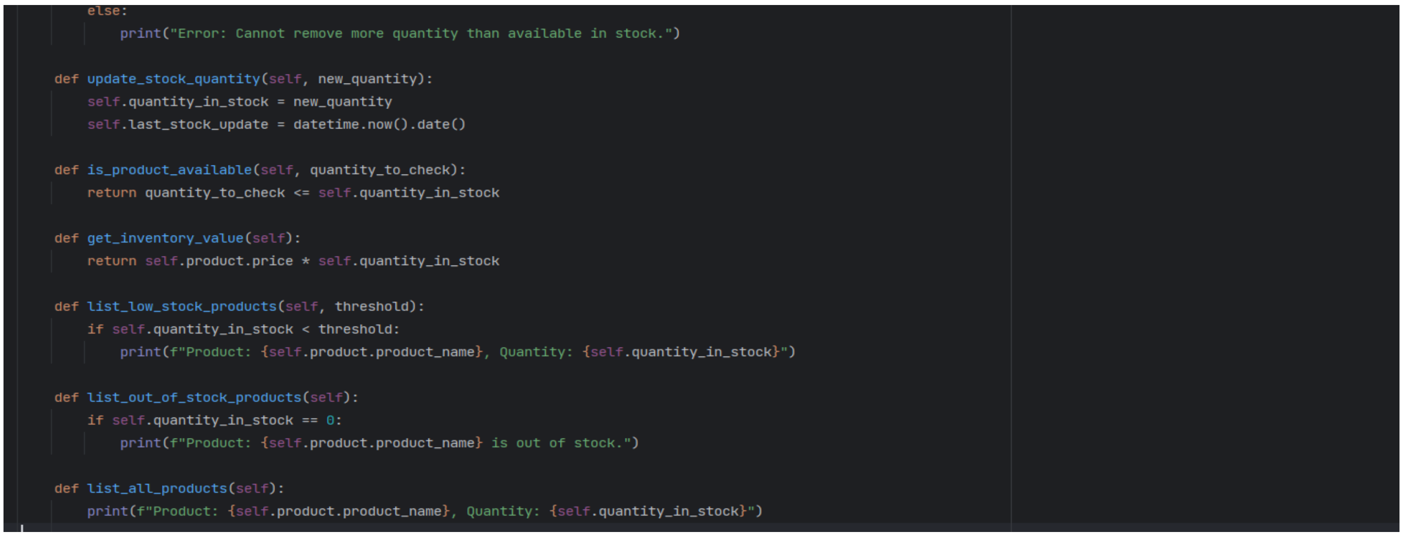
```
2 usages
class OrderDetails:
    def _init_(self, OrderDetailID, Order, Product, Quantity):
        self.OrderDetailID = OrderDetailID
        self.Order = Order
        self.Product = Product
        self.Quantity = Quantity

    2 usages (1 dynamic)
    def CalculateSubtotal(self):
        return self.Product.Price * self.Quantity

    2 usages (1 dynamic)
    def GetOrderDetailInfo(self):
        print(f'OrderDetailID: {self.OrderDetailID}, Order: {self.Order.OrderID}, Product: {self.Product.ProductName}, Quantity: {self.Quantity}')

    1 usage
    def UpdateQuantity(self, quantity):
        self.Quantity = quantity

    1 usage
    def AddDiscount(self, discount):
        self.Product.Price -= discount
```

Inventory class:

Attributes:
• InventoryID(int)
• Product (Composition): The product associated with the inventory item.
• QuantityInStock: The quantity of the product currently in stock.
• LastStockUpdate
Methods:
• GetProduct(): A method to retrieve the product associated with this inventory item.
• GetQuantityInStock(): A method to get the current quantity of the product in stock.
• AddToInventory(int quantity): A method to add a specified quantity of the product to the inventory.
• RemoveFromInventory(int quantity): A method to remove a specified quantity of the product from the inventory.
• UpdateStockQuantity(int newQuantity): A method to update the stock quantity to a new value.
• IsProductAvailable(int quantityToCheck): A method to check if a specified quantity of the product is available in the inventory.
• GetInventoryValue(): A method to calculate the total value of the products in the inventory based on their prices and quantities.
• ListLowStockProducts(int threshold): A method to list products with quantities below a specified threshold, indicating low stock.
• ListOutOfStockProducts(): A method to list products that are out of stock.

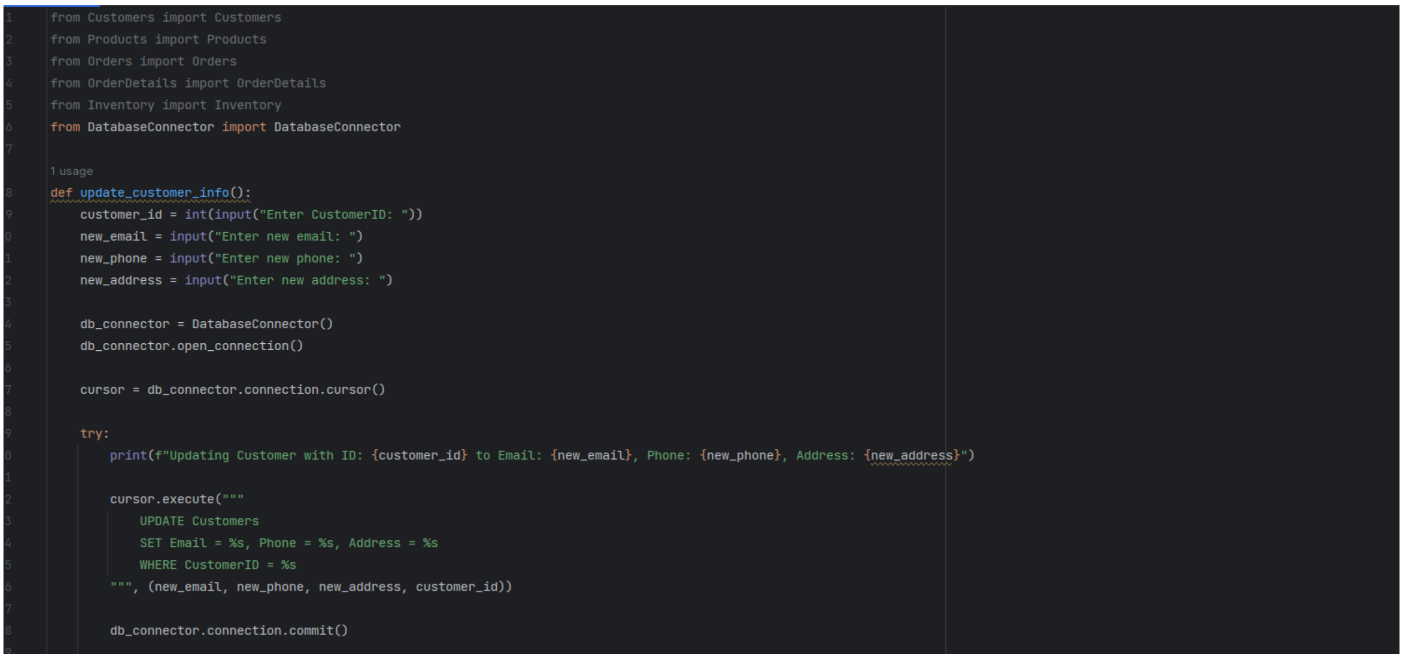


```python
from datetime import datetime

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock, last_stock_update=None):
        self.inventory_id = inventory_id
        self.product = product
        self.quantity_in_stock = quantity_in_stock
        self.last_stock_update = last_stock_update or datetime.now().date()

    def get_product(self):
        return self.product

    def get_quantity_in_stock(self):
        return self.quantity_in_stock

    def add_to_inventory(self, quantity):
        self.quantity_in_stock += quantity
        self.last_stock_update = datetime.now().date()

    def remove_from_inventory(self, quantity):
        if quantity <= self.quantity_in_stock:
            self.quantity_in_stock -= quantity
            self.last_stock_update = datetime.now().date()
        else:
            print("Error: Cannot remove more quantity than available in stock.")
```

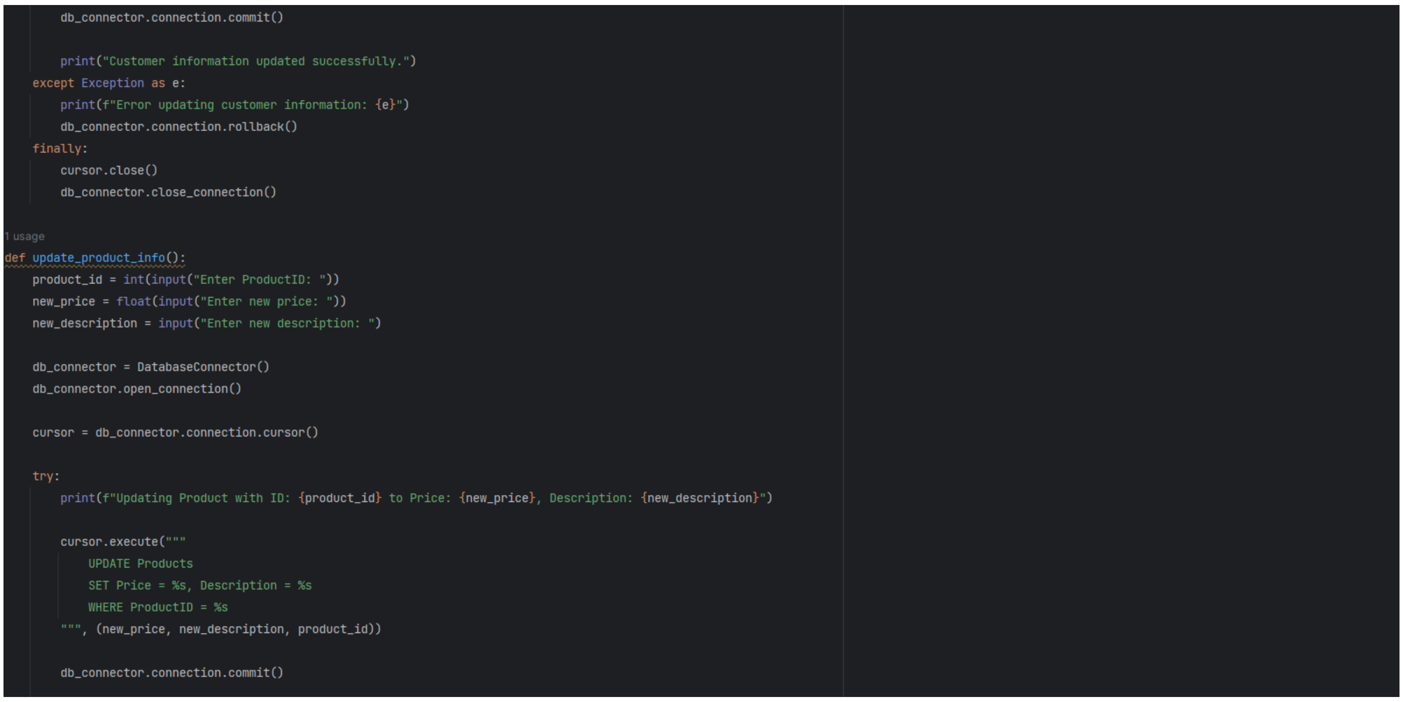


```python
        else:
            print("Error: Cannot remove more quantity than available in stock.")

    def update_stock_quantity(self, new_quantity):
        self.quantity_in_stock = new_quantity
        self.last_stock_update = datetime.now().date()

    def is_product_available(self, quantity_to_check):
        return quantity_to_check <= self.quantity_in_stock

    def get_inventory_value(self):
        return self.product.price * self.quantity_in_stock

    def list_low_stock_products(self, threshold):
        if self.quantity_in_stock < threshold:
            print(f"Product: {self.product.product_name}, Quantity: {self.quantity_in_stock}")

    def list_out_of_stock_products(self):
        if self.quantity_in_stock == 0:
            print(f"Product: {self.product.product_name} is out of stock.")

    def list_all_products(self):
        print(f"Product: {self.product.product_name}, Quantity: {self.quantity_in_stock}")
```

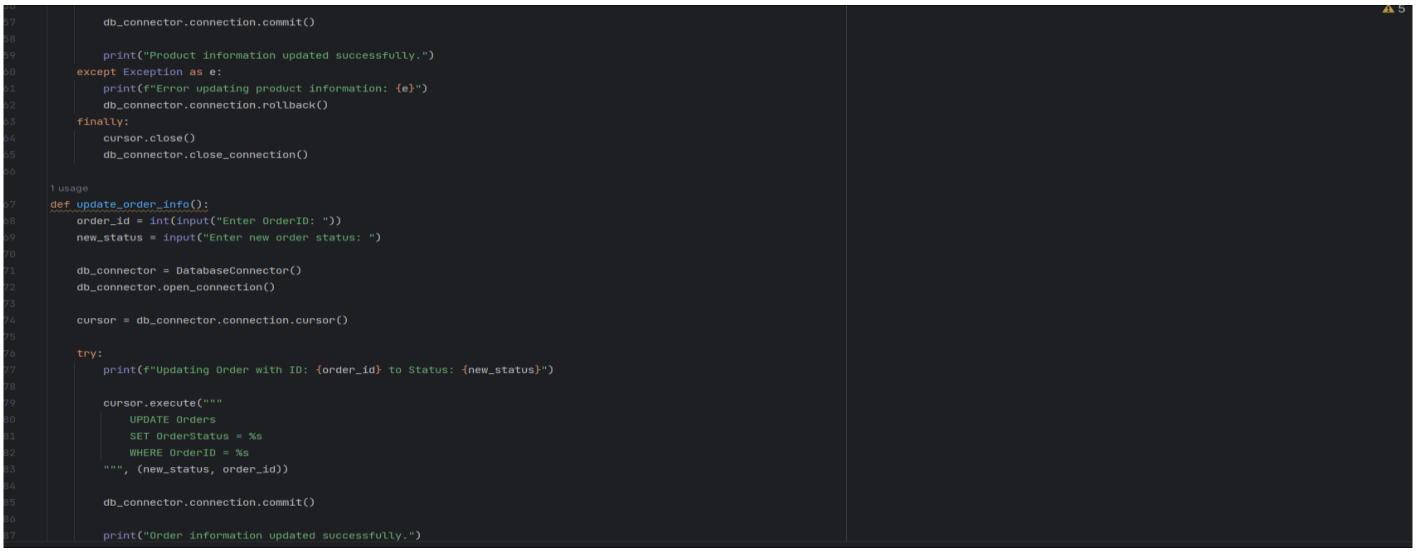
Main.py class



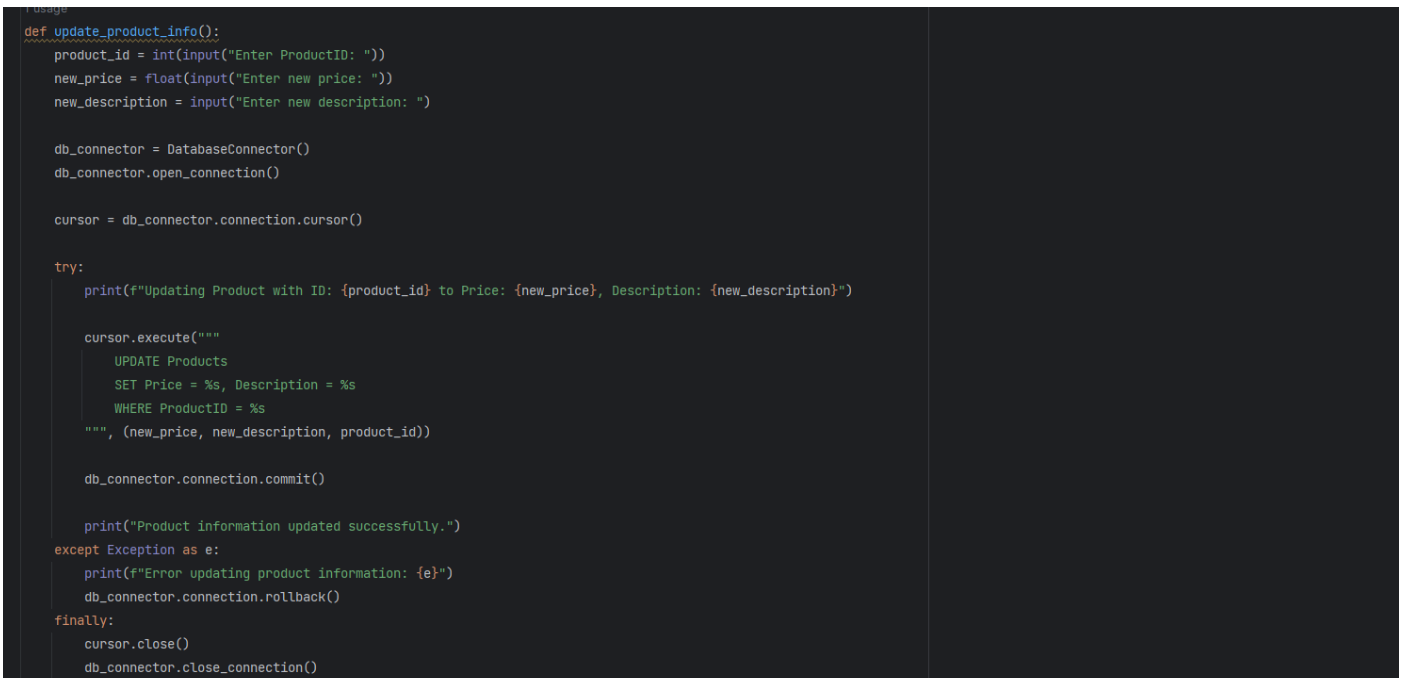
```python
from Customers import Customers
from Products import Products
from Orders import Orders
from OrderDetails import OrderDetails
from Inventory import Inventory
from DatabaseConnector import DatabaseConnector

def update_customer_info():
    customer_id = int(input("Enter CustomerID: "))
    new_email = input("Enter new email: ")
    new_phone = input("Enter new phone: ")
    new_address = input("Enter new address: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Customer with ID: {customer_id} to Email: {new_email}, Phone: {new_phone}, Address: {new_address}")

        cursor.execute("""
            UPDATE Customers
            SET Email = %s, Phone = %s, Address = %s
            WHERE CustomerID = %s
        """, (new_email, new_phone, new_address, customer_id))

        db_connector.connection.commit()
```



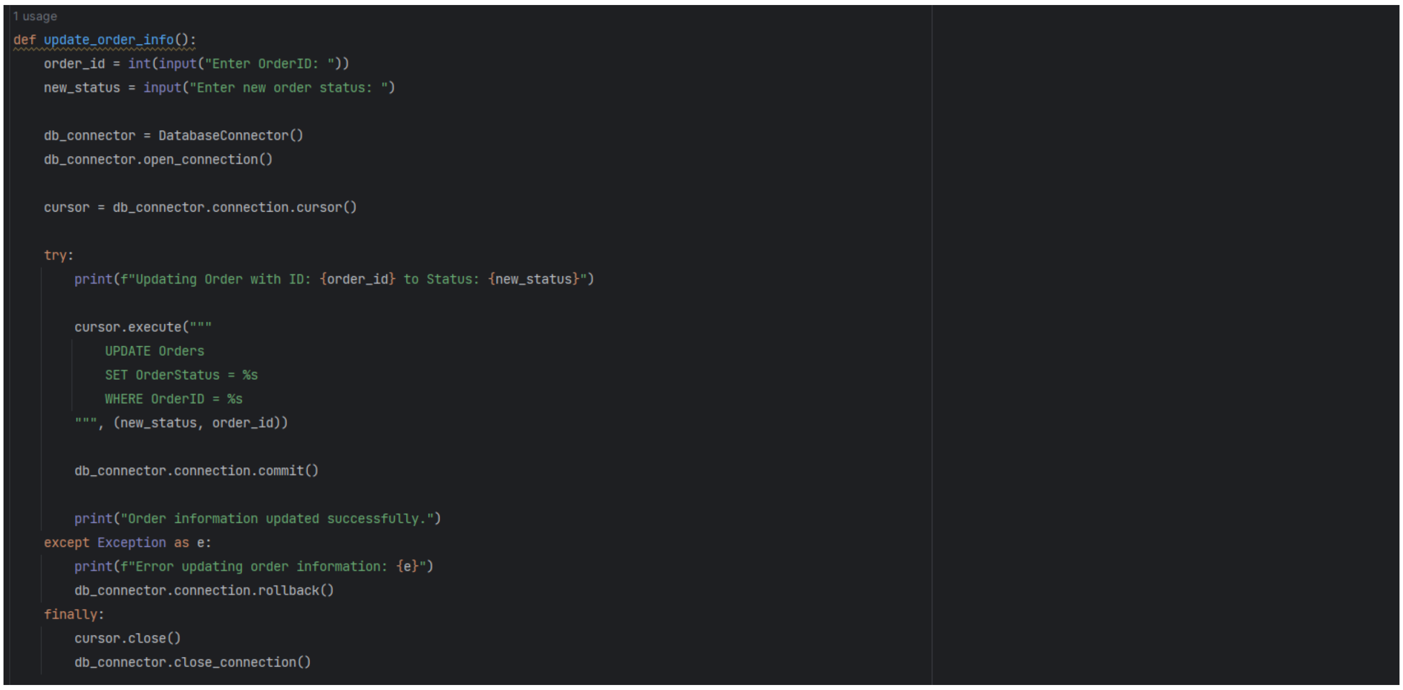
```python
        db_connector.connection.commit()

        print("Customer information updated successfully.")
    except Exception as e:
        print(f"Error updating customer information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()

1 usage
def update_product_info():
    product_id = int(input("Enter ProductID: "))
    new_price = float(input("Enter new price: "))
    new_description = input("Enter new description: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Product with ID: {product_id} to Price: {new_price}, Description: {new_description}")

        cursor.execute("""
            UPDATE Products
            SET Price = %s, Description = %s
            WHERE ProductID = %s
        """, (new_price, new_description, product_id))

        db_connector.connection.commit()
```



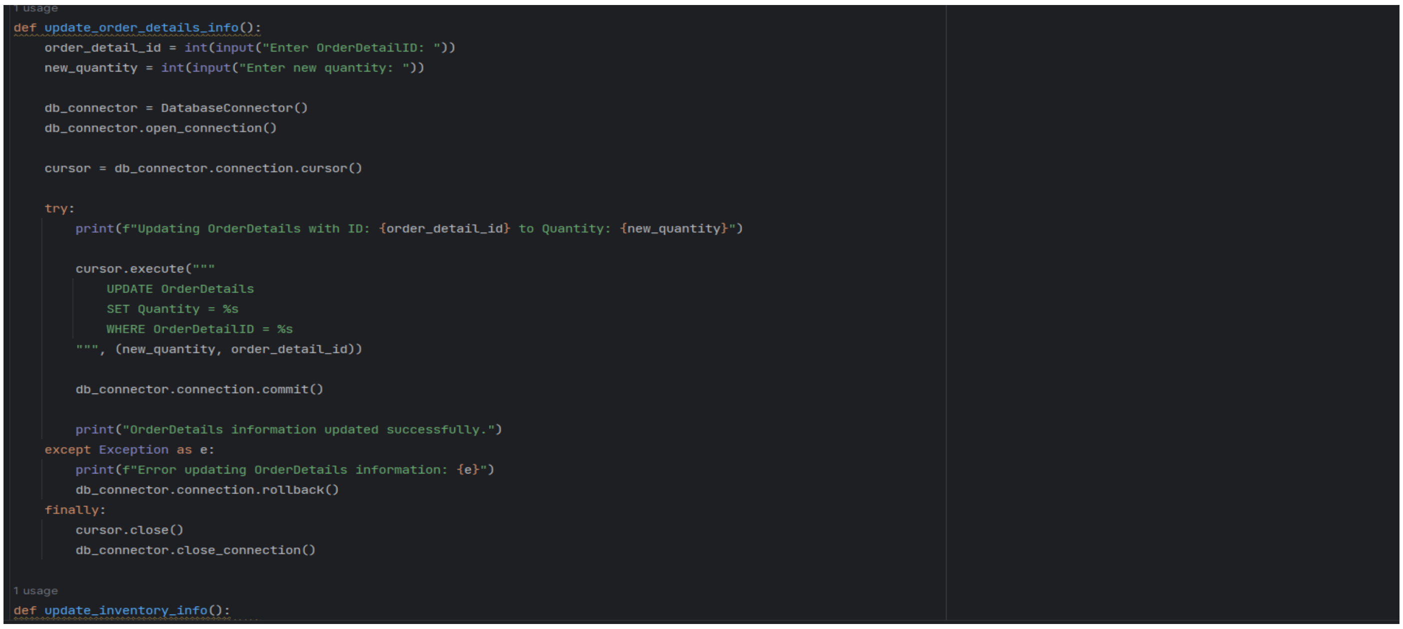
```python
        db_connector.connection.commit()

        print("Product information updated successfully.")
    except Exception as e:
        print(f"Error updating product information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()

1 usage
def update_order_info():
    order_id = int(input("Enter OrderID: "))
    new_status = input("Enter new order status: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Order with ID: {order_id} to Status: {new_status}")

        cursor.execute("""
            UPDATE Orders
            SET OrderStatus = %s
            WHERE OrderID = %s
        """, (new_status, order_id))

        db_connector.connection.commit()

        print("Order information updated successfully.")
```



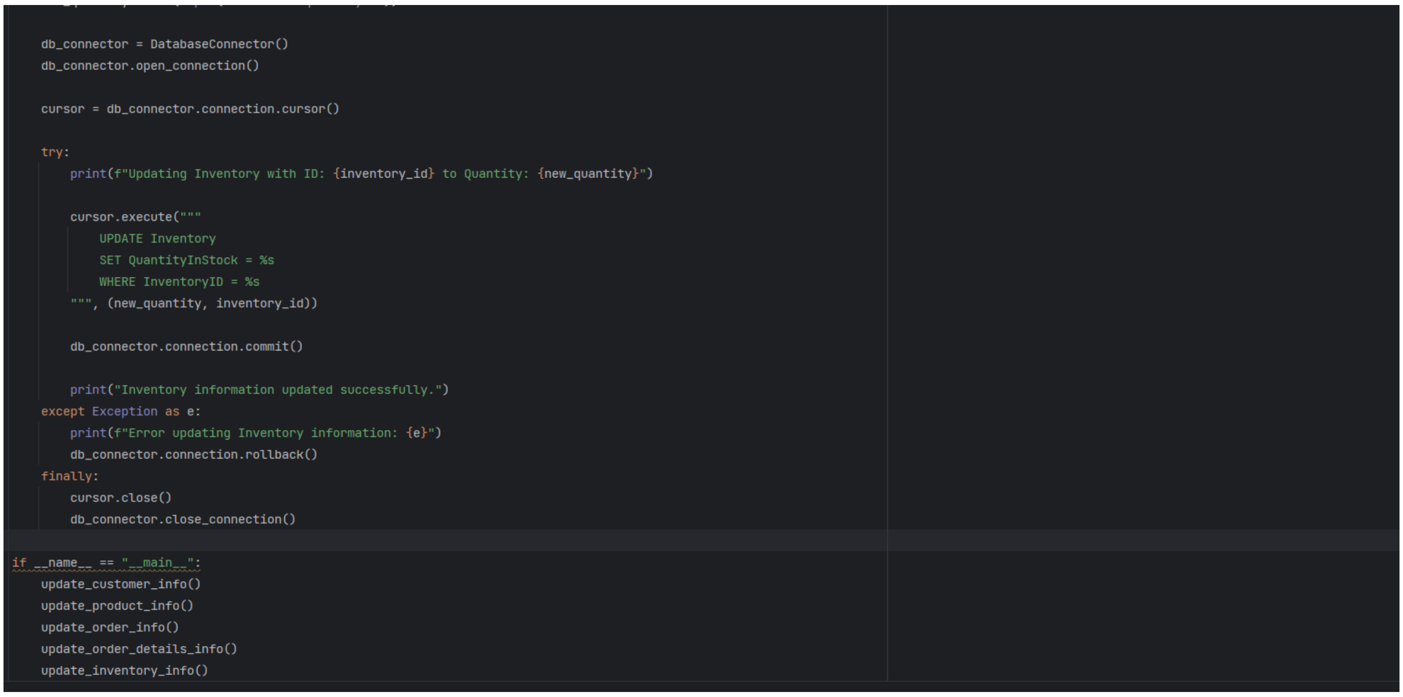
```python
1 usage
def update_product_info():
    product_id = int(input("Enter ProductID: "))
    new_price = float(input("Enter new price: "))
    new_description = input("Enter new description: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Product with ID: {product_id} to Price: {new_price}, Description: {new_description}")

        cursor.execute("""
            UPDATE Products
            SET Price = %s, Description = %s
            WHERE ProductID = %s
        """, (new_price, new_description, product_id))

        db_connector.connection.commit()

        print("Product information updated successfully.")
    except Exception as e:
        print(f"Error updating product information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```



```python
1 usage
def update_order_info():
    order_id = int(input("Enter OrderID: "))
    new_status = input("Enter new order status: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Order with ID: {order_id} to Status: {new_status}")

        cursor.execute("""
            UPDATE Orders
            SET OrderStatus = %s
            WHERE OrderID = %s
        """, (new_status, order_id))

        db_connector.connection.commit()

        print("Order information updated successfully.")
    except Exception as e:
        print(f"Error updating order information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```



```python
1 usage
def update_order_details_info():
    order_detail_id = int(input("Enter OrderDetailID: "))
    new_quantity = int(input("Enter new quantity: "))

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating OrderDetails with ID: {order_detail_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE OrderDetails
            SET Quantity = %s
            WHERE OrderDetailID = %s
        """, (new_quantity, order_detail_id))

        db_connector.connection.commit()

        print("OrderDetails information updated successfully.")
    except Exception as e:
        print(f"Error updating OrderDetails information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()

1 usage
def update_inventory_info():
```



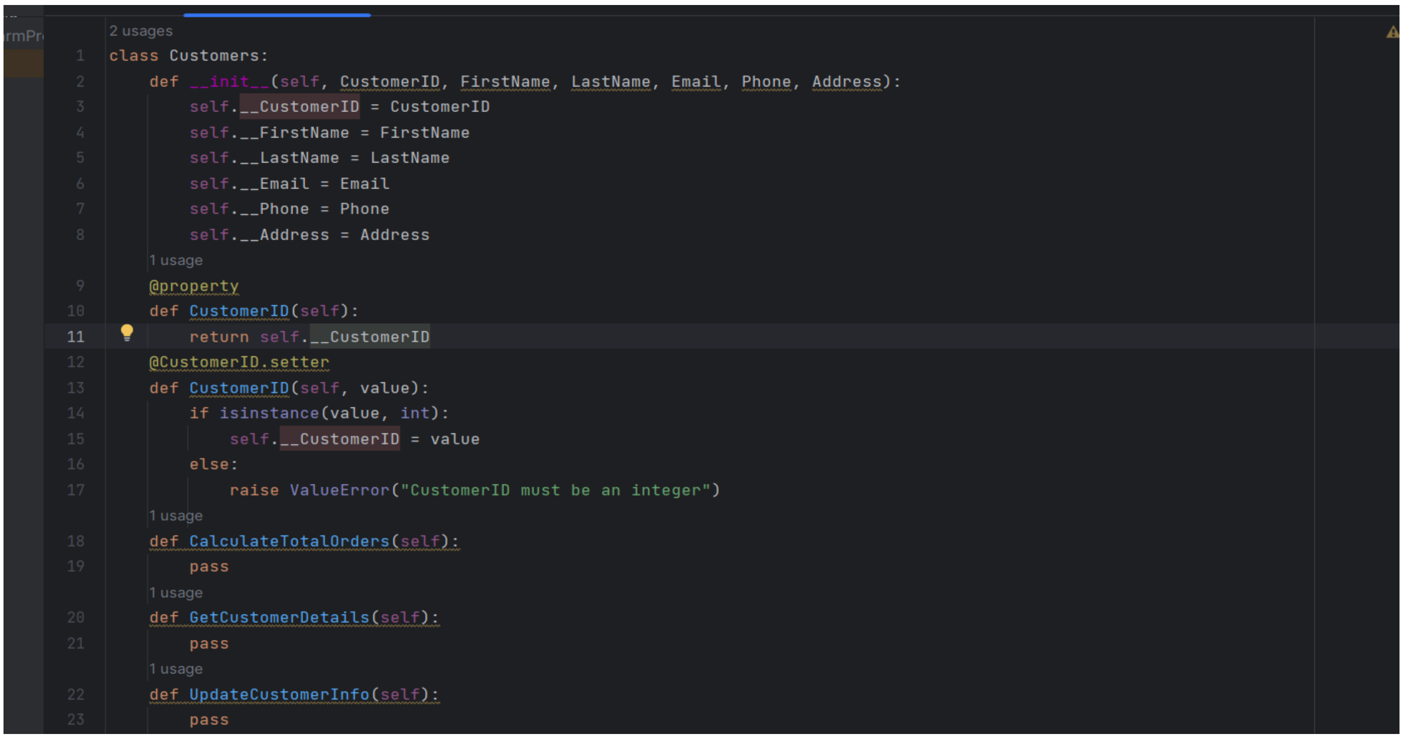
```python
    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Inventory with ID: {inventory_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE Inventory
            SET QuantityInStock = %s
            WHERE InventoryID = %s
        """, (new_quantity, inventory_id))

        db_connector.connection.commit()

        print("Inventory information updated successfully.")
    except Exception as e:
        print(f"Error updating Inventory information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()

if __name__ == "__main__":
    update_customer_info()
    update_product_info()
    update_order_info()
    update_order_details_info()
    update_inventory_info()
```

Task 3: Encapsulation:

• Implement encapsulation by making the attributes private and providing public properties (getters and setters) for each attribute.

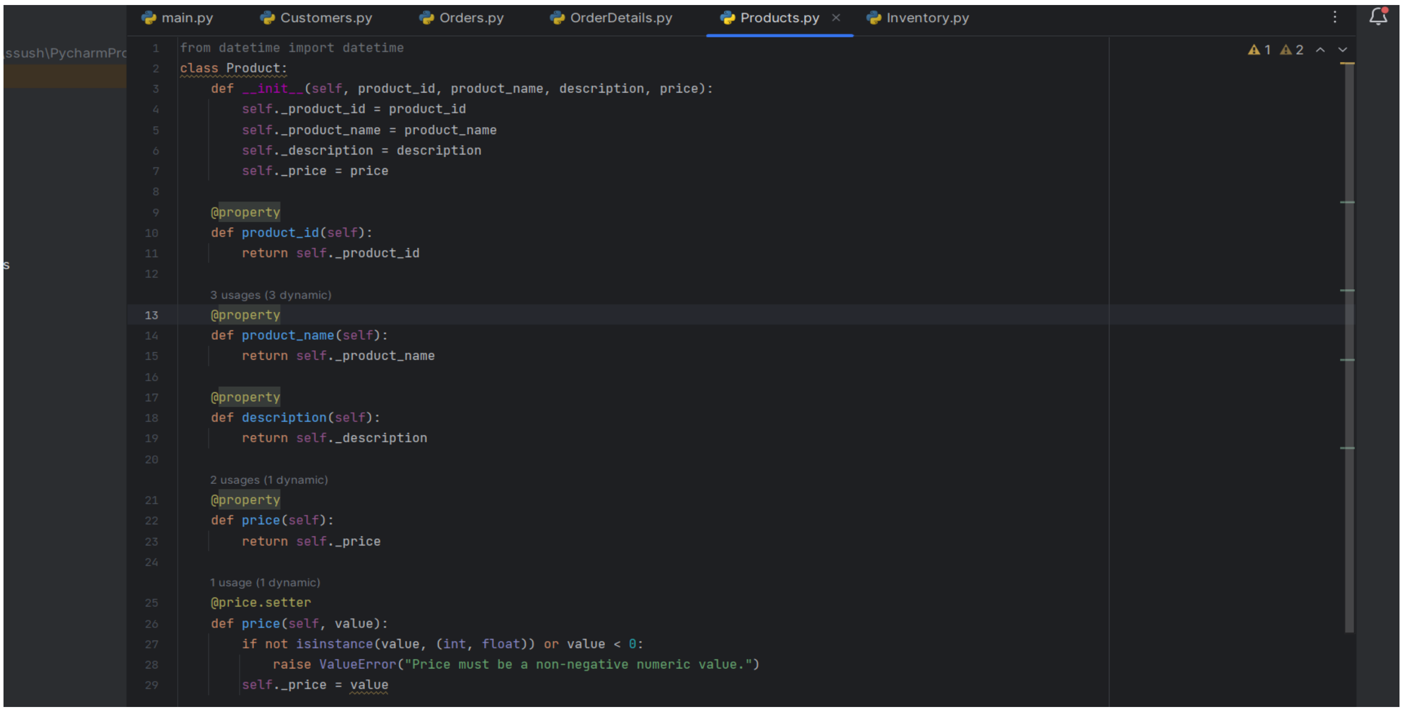
• Add data validation logic to setter methods (e.g., ensure that prices are non-negative, quantities are positive integers).

Customer .py class with Encapsulation



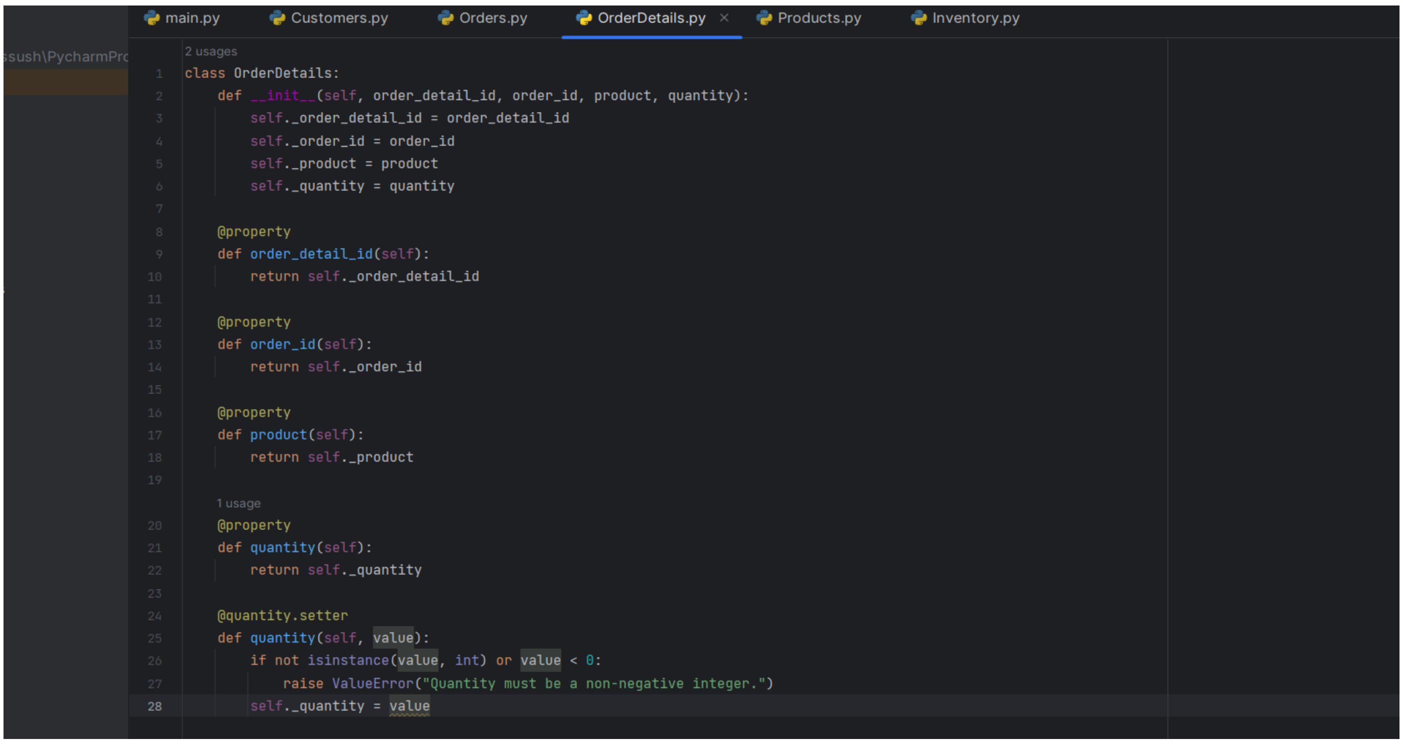
```python
class Customers:
    def __init__(self, CustomerID, FirstName, LastName, Email, Phone, Address):
        self.__CustomerID = CustomerID
        self.__FirstName = FirstName
        self.__LastName = LastName
        self.__Email = Email
        self.__Phone = Phone
        self.__Address = Address
    @property
    def CustomerID(self):
        return self.__CustomerID
    @CustomerID.setter
    def CustomerID(self, value):
        if isinstance(value, int):
            self.__CustomerID = value
        else:
            raise ValueError("CustomerID must be an integer")
    def CalculateTotalOrders(self):
        pass
    def GetCustomerDetails(self):
        pass
    def UpdateCustomerInfo(self):
        pass
```

Products.py class with Encapsulation Properties



```python
from datetime import datetime
class Product:
    def __init__(self, product_id, product_name, description, price):
        self._product_id = product_id
        self._product_name = product_name
        self._description = description
        self._price = price

    @property
    def product_id(self):
        return self._product_id

    @property
    def product_name(self):
        return self._product_name

    @property
    def description(self):
        return self._description

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if not isinstance(value, (int, float)) or value < 0:
            raise ValueError("Price must be a non-negative numeric value.")
        self._price = value
```

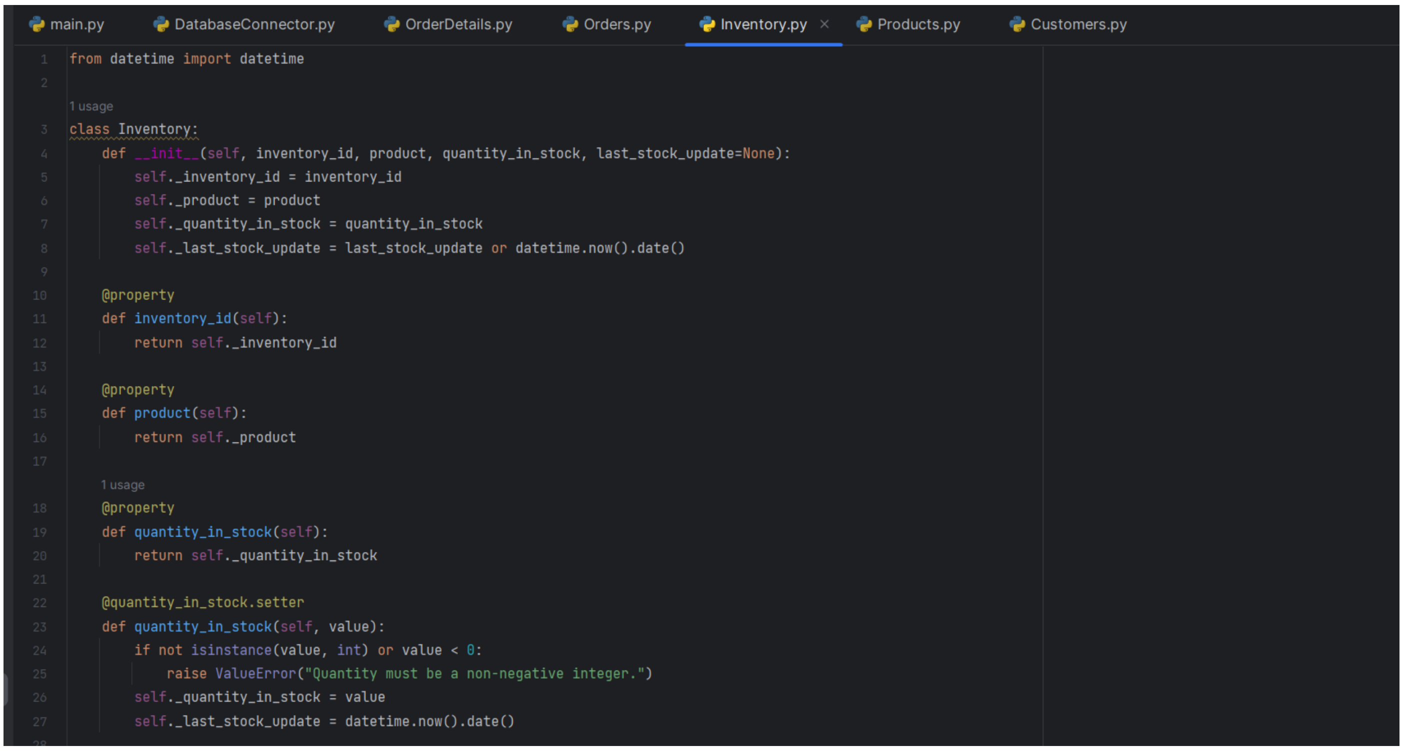
OrderDetails.py class with Encapsulation



main.py Customers.py Orders.py OrderDetails.py Products.py Inventory.py

```python
class OrderDetails:
    def __init__(self, order_detail_id, order_id, product, quantity):
        self._order_detail_id = order_detail_id
        self._order_id = order_id
        self._product = product
        self._quantity = quantity

    @property
    def order_detail_id(self):
        return self._order_detail_id

    @property
    def order_id(self):
        return self._order_id

    @property
    def product(self):
        return self._product

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        if not isinstance(value, int) or value < 0:
            raise ValueError("Quantity must be a non-negative integer.")
        self._quantity = value
```

Inventory.py class with Encapsulation Properties



main.py DatabaseConnector.py OrderDetails.py Orders.py Inventory.py Products.py Customers.py

```python
from datetime import datetime

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock, last_stock_update=None):
        self._inventory_id = inventory_id
        self._product = product
        self._quantity_in_stock = quantity_in_stock
        self._last_stock_update = last_stock_update or datetime.now().date()

    @property
    def inventory_id(self):
        return self._inventory_id

    @property
    def product(self):
        return self._product

    @property
    def quantity_in_stock(self):
        return self._quantity_in_stock

    @quantity_in_stock.setter
    def quantity_in_stock(self, value):
        if not isinstance(value, int) or value < 0:
            raise ValueError("Quantity must be a non-negative integer.")
        self._quantity_in_stock = value
        self._last_stock_update = datetime.now().date()
```

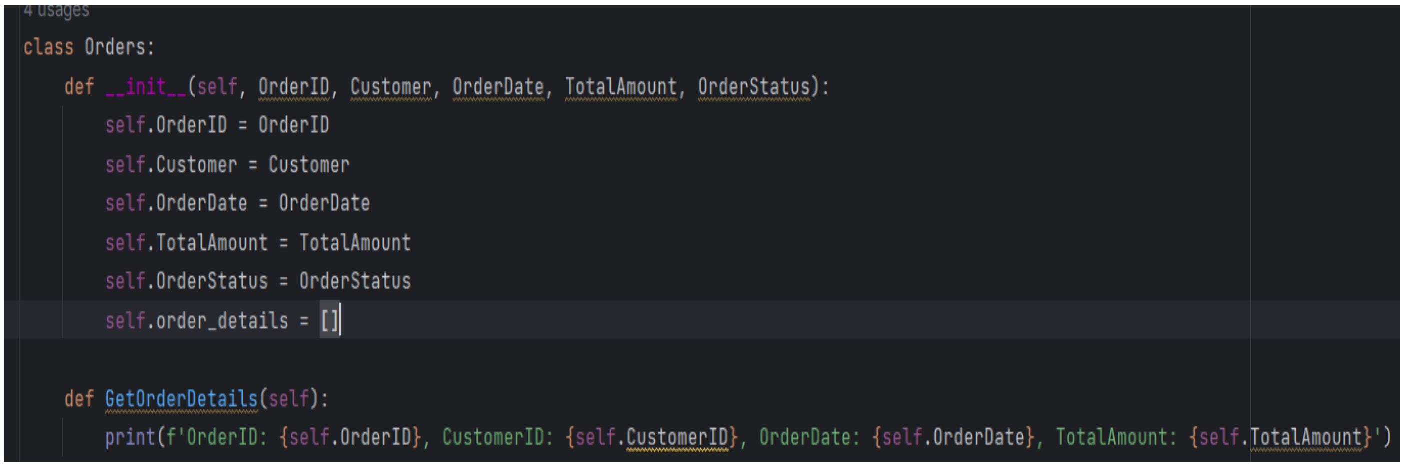
Task 4: Composition:

Ensure that the Order and OrderDetail classes correctly use composition to reference Customer and Product objects.

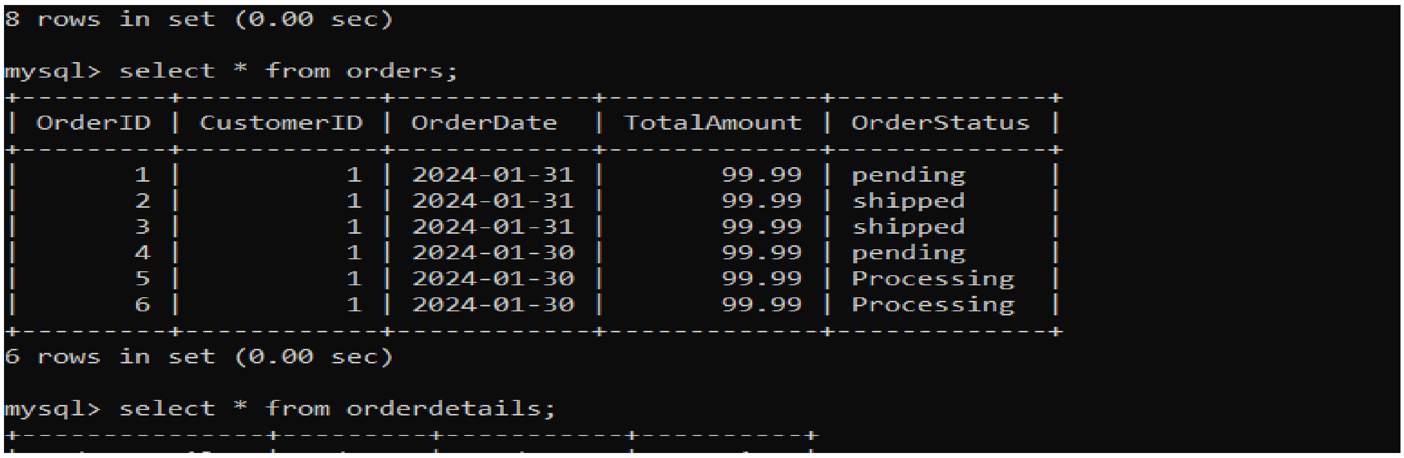
• Orders Class with Composition:

o In the Orders class, we want to establish a composition relationship with the Customers class, indicating that each order is associated with a specific customer.

o In the Orders class, we've added a private attribute customer of type Customers, establishing a composition relationship. The Customer property provides access to the Customers object associated with the order.



```
4 usages
class Orders:
    def __init__(self, OrderID, Customer, OrderDate, TotalAmount, OrderStatus):
        self.OrderID = OrderID
        self.Customer = Customer
        self.OrderDate = OrderDate
        self.TotalAmount = TotalAmount
        self.OrderStatus = OrderStatus
        self.order_details = []

    def GetOrderDetails(self):
        print(f'OrderID: {self.OrderID}, CustomerID: {self.CustomerID}, OrderDate: {self.OrderDate}, TotalAmount: {self.TotalAmount}')
```

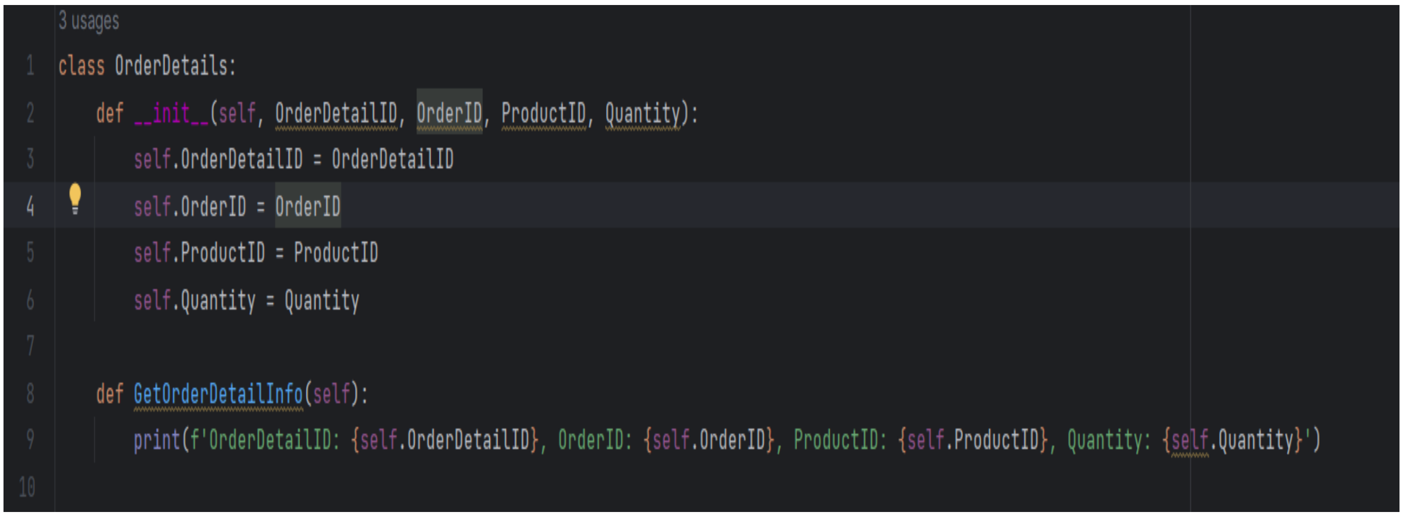


```
8 rows in set (0.00 sec)

mysql> select * from orders;
+---------+------------+------------+-------------+-------------+
| OrderID | CustomerID | OrderDate  | TotalAmount | OrderStatus |
+---------+------------+------------+-------------+-------------+
|       1 |          1 | 2024-01-31 |       99.99 | pending     |
|       2 |          1 | 2024-01-31 |       99.99 | shipped     |
|       3 |          1 | 2024-01-31 |       99.99 | shipped     |
|       4 |          1 | 2024-01-30 |       99.99 | pending     |
|       5 |          1 | 2024-01-30 |       99.99 | Processing  |
|       6 |          1 | 2024-01-30 |       99.99 | Processing  |
+---------+------------+------------+-------------+-------------+
6 rows in set (0.00 sec)

mysql> select * from orderdetails;
+---------------+----------+------------+----------+
```

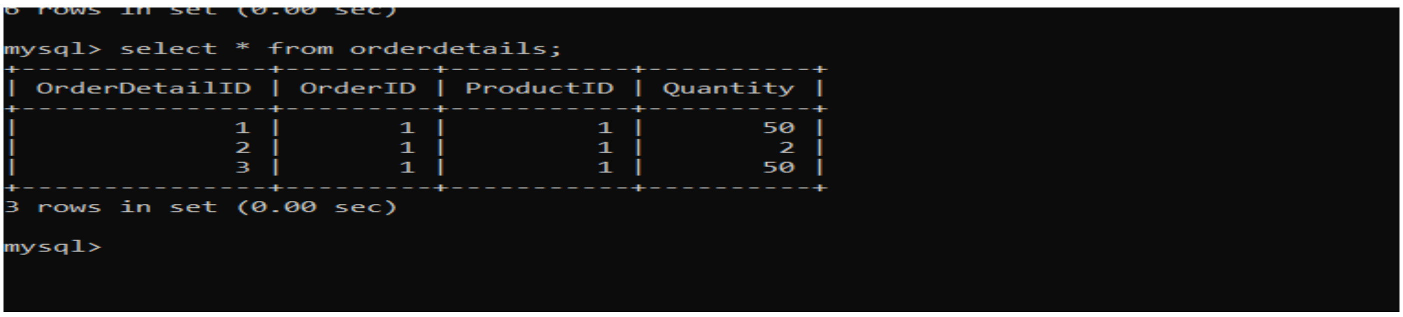
• OrderDetails Class with Composition:

o Similarly, in the OrderDetails class, we want to establish composition relationships with both the Orders and Products classes to represent the details of each order, including the product being ordered.

o In the OrderDetails class, we've added two private attributes, order and product, of types Orders and Products, respectively, establishing composition relationships. TheOrder property provides access to the Orders object associated with the order detail,and the Product property provides access to the Products object representing the product in the order detail.



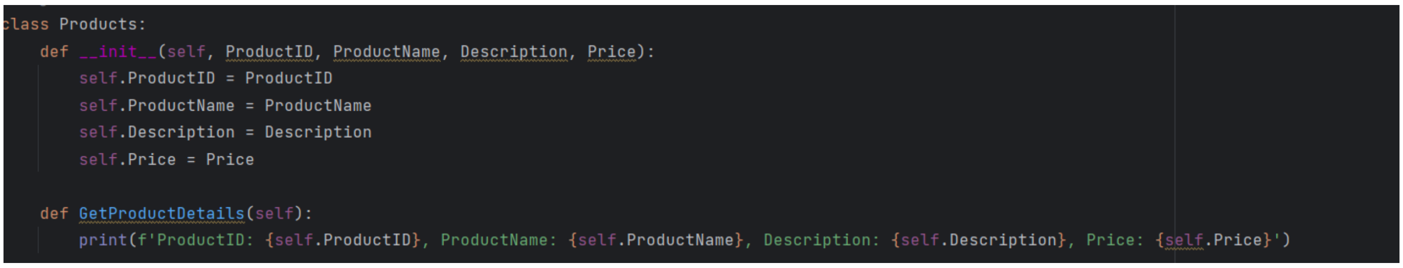
```
3 usages
class OrderDetails:
    def __init__(self, OrderDetailID, OrderID, ProductID, Quantity):
        self.OrderDetailID = OrderDetailID
        self.OrderID = OrderID
        self.ProductID = ProductID
        self.Quantity = Quantity

    def GetOrderDetailInfo(self):
        print(f'OrderDetailID: {self.OrderDetailID}, OrderID: {self.OrderID}, ProductID: {self.ProductID}, Quantity: {self.Quantity}')
```



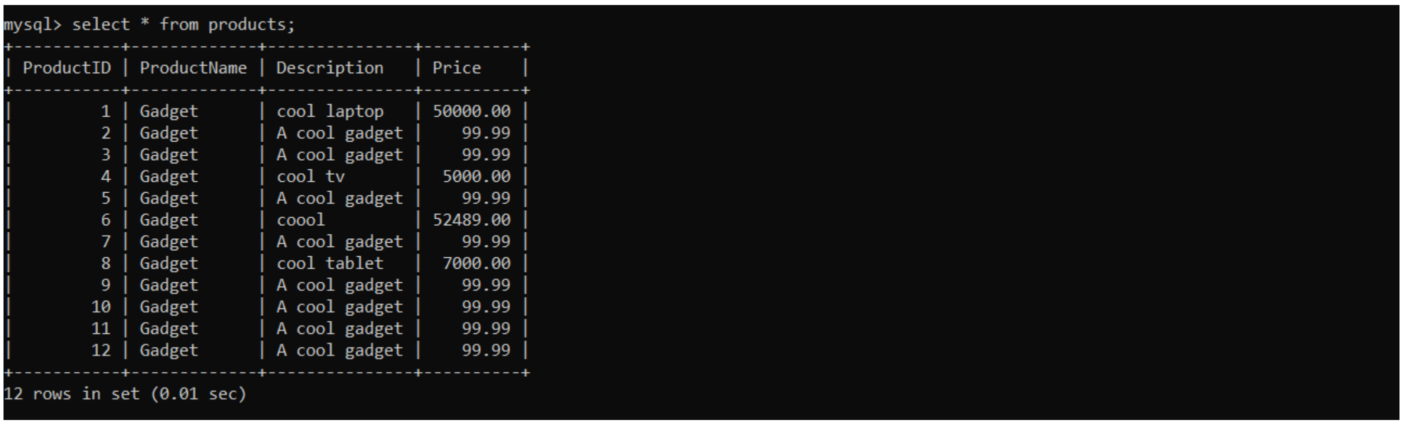
```
0 rows in set (0.00 sec)

mysql> select * from orderdetails;
+---------------+---------+-----------+----------+
| OrderDetailID | OrderID | ProductID | Quantity |
+---------------+---------+-----------+----------+
|             1 |       1 |         1 |       50 |
|             2 |       1 |         1 |        2 |
|             3 |       1 |         1 |       50 |
+---------------+---------+-----------+----------+
3 rows in set (0.00 sec)

mysql>
```

• **Customers and Products Classes:**

o The Customers and Products classes themselves may not have direct composition relationships with other classes in this scenario. However, they serve as the basis for composition relationships in the Orders and OrderDetails classes, respectively.



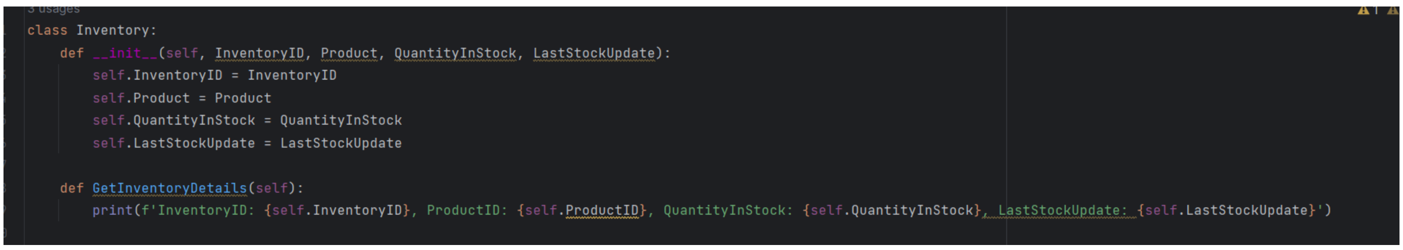
```
lass Products:
    def __init__(self, ProductID, ProductName, Description, Price):
        self.ProductID = ProductID
        self.ProductName = ProductName
        self.Description = Description
        self.Price = Price

    def GetProductDetails(self):
        print(f'ProductID: {self.ProductID}, ProductName: {self.ProductName}, Description: {self.Description}, Price: {self.Price}')
```



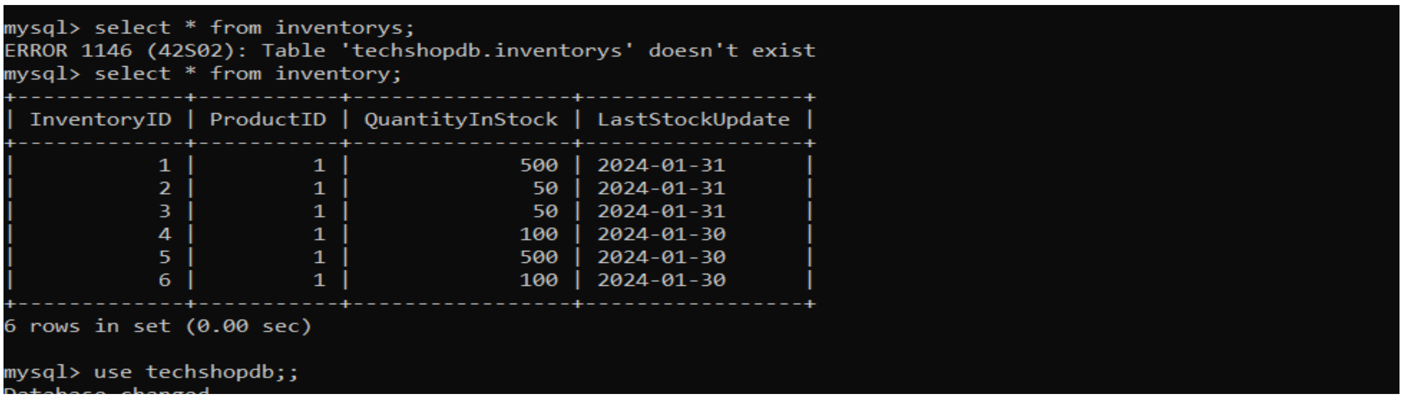
```
mysql> select * from products;
+-----------+-------------+---------------+----------+
| ProductID | ProductName | Description   | Price    |
+-----------+-------------+---------------+----------+
|         1 | Gadget      | cool laptop   | 50000.00 |
|         2 | Gadget      | A cool gadget |    99.99 |
|         3 | Gadget      | A cool gadget |    99.99 |
|         4 | Gadget      | cool tv       |  5000.00 |
|         5 | Gadget      | A cool gadget |    99.99 |
|         6 | Gadget      | coool         | 52489.00 |
|         7 | Gadget      | A cool gadget |    99.99 |
|         8 | Gadget      | cool tablet   |  7000.00 |
|         9 | Gadget      | A cool gadget |    99.99 |
|        10 | Gadget      | A cool gadget |    99.99 |
|        11 | Gadget      | A cool gadget |    99.99 |
|        12 | Gadget      | A cool gadget |    99.99 |
+-----------+-------------+---------------+----------+
12 rows in set (0.01 sec)
```

• **Inventory Class:**

o The Inventory class represents the inventory of products available for sale. It can have composition relationships with the Products class to indicate which products are in the inventory.



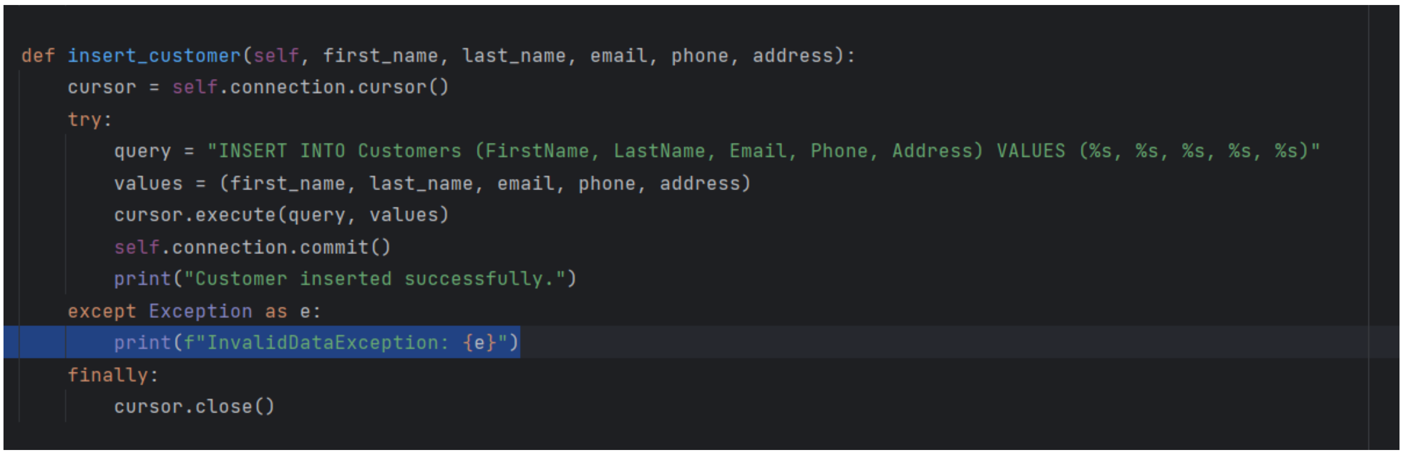
```
3 usages
class Inventory:
    def __init__(self, InventoryID, Product, QuantityInStock, LastStockUpdate):
        self.InventoryID = InventoryID
        self.Product = Product
        self.QuantityInStock = QuantityInStock
        self.LastStockUpdate = LastStockUpdate

    def GetInventoryDetails(self):
        print(f'InventoryID: {self.InventoryID}, ProductID: {self.ProductID}, QuantityInStock: {self.QuantityInStock}, LastStockUpdate: {self.LastStockUpdate}')
```



```
mysql> select * from inventorys;
ERROR 1146 (42S02): Table 'techshopdb.inventorys' doesn't exist
mysql> select * from inventory;
+-------------+-----------+-----------------+-----------------+
| InventoryID | ProductID | QuantityInStock | LastStockUpdate |
+-------------+-----------+-----------------+-----------------+
|           1 |         1 |             500 | 2024-01-31      |
|           2 |         1 |              50 | 2024-01-31      |
|           3 |         1 |              50 | 2024-01-31      |
|           4 |         1 |             100 | 2024-01-30      |
|           5 |         1 |             500 | 2024-01-30      |
|           6 |         1 |             100 | 2024-01-30      |
+-------------+-----------+-----------------+-----------------+
6 rows in set (0.00 sec)

mysql> use techshopdb;;
```

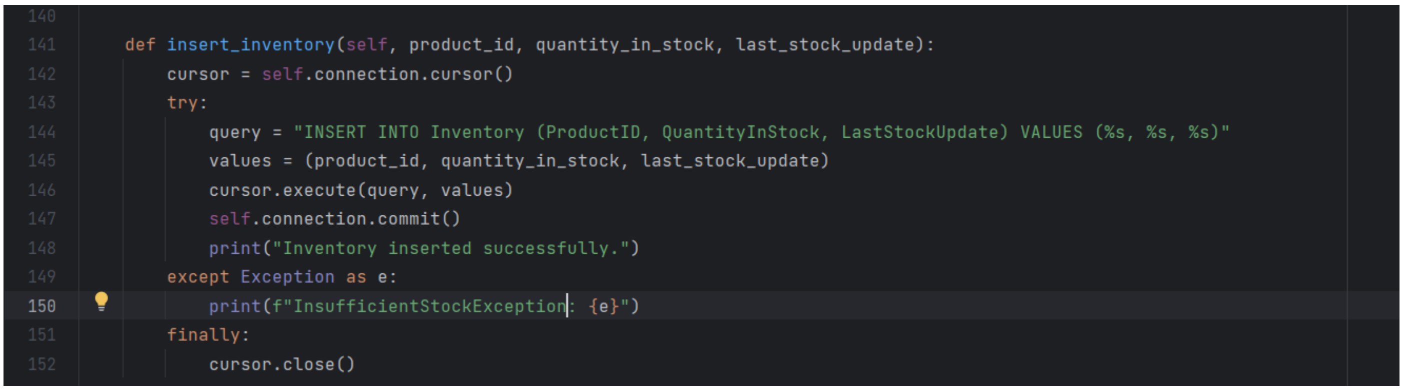
Task 5: Exceptions handling

• **Data Validation:**
o Challenge: Validate user inputs and data from external sources (e.g., user registration, order placement).
o Scenario: When a user enters an invalid email address during registration.
o Exception Handling: Throw a custom InvalidDataException with a clear error message.



```python
def insert_customer(self, first_name, last_name, email, phone, address):
    cursor = self.connection.cursor()
    try:
        query = "INSERT INTO Customers (FirstName, LastName, Email, Phone, Address) VALUES (%s, %s, %s, %s, %s)"
        values = (first_name, last_name, email, phone, address)
        cursor.execute(query, values)
        self.connection.commit()
        print("Customer inserted successfully.")
    except Exception as e:
        print(f"InvalidDataException: {e}")
    finally:
        cursor.close()
```

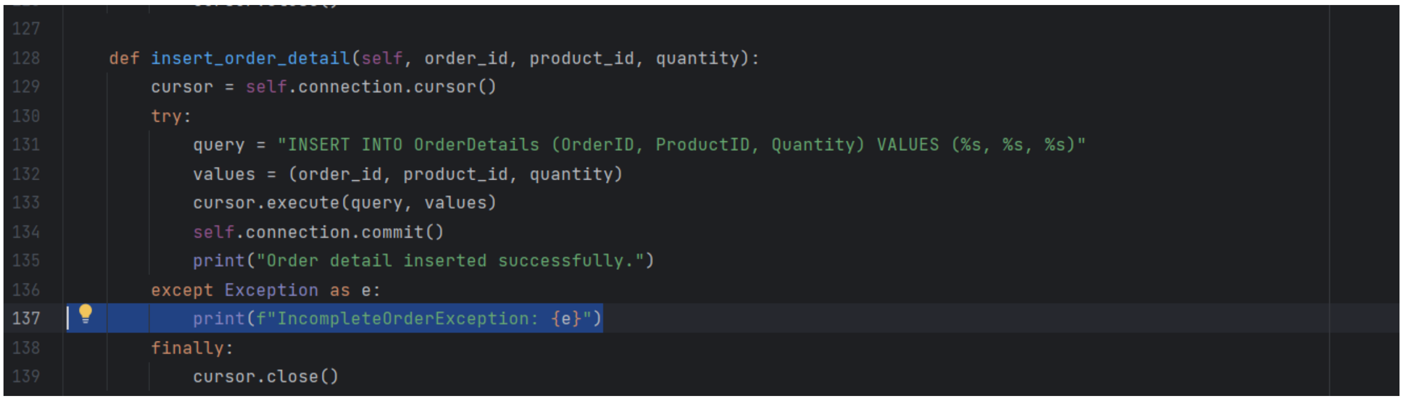
• **Inventory Management:**
o Challenge: Handling inventory-related issues, such as selling more products than are in stock.
o Exception Handling: Throw an InsufficientStockException and update the order status accordingly.



```python
def insert_inventory(self, product_id, quantity_in_stock, last_stock_update):
    cursor = self.connection.cursor()
    try:
        query = "INSERT INTO Inventory (ProductID, QuantityInStock, LastStockUpdate) VALUES (%s, %s, %s)"
        values = (product_id, quantity_in_stock, last_stock_update)
        cursor.execute(query, values)
        self.connection.commit()
        print("Inventory inserted successfully.")
    except Exception as e:
        print(f"InsufficientStockException: {e}")
    finally:
        cursor.close()
```

• **Order Processing:**
o Challenge: Ensuring the order details are consistent and complete before processing.

o Exception Handling: Throw an IncompleteOrderException with a message explaining the issue.

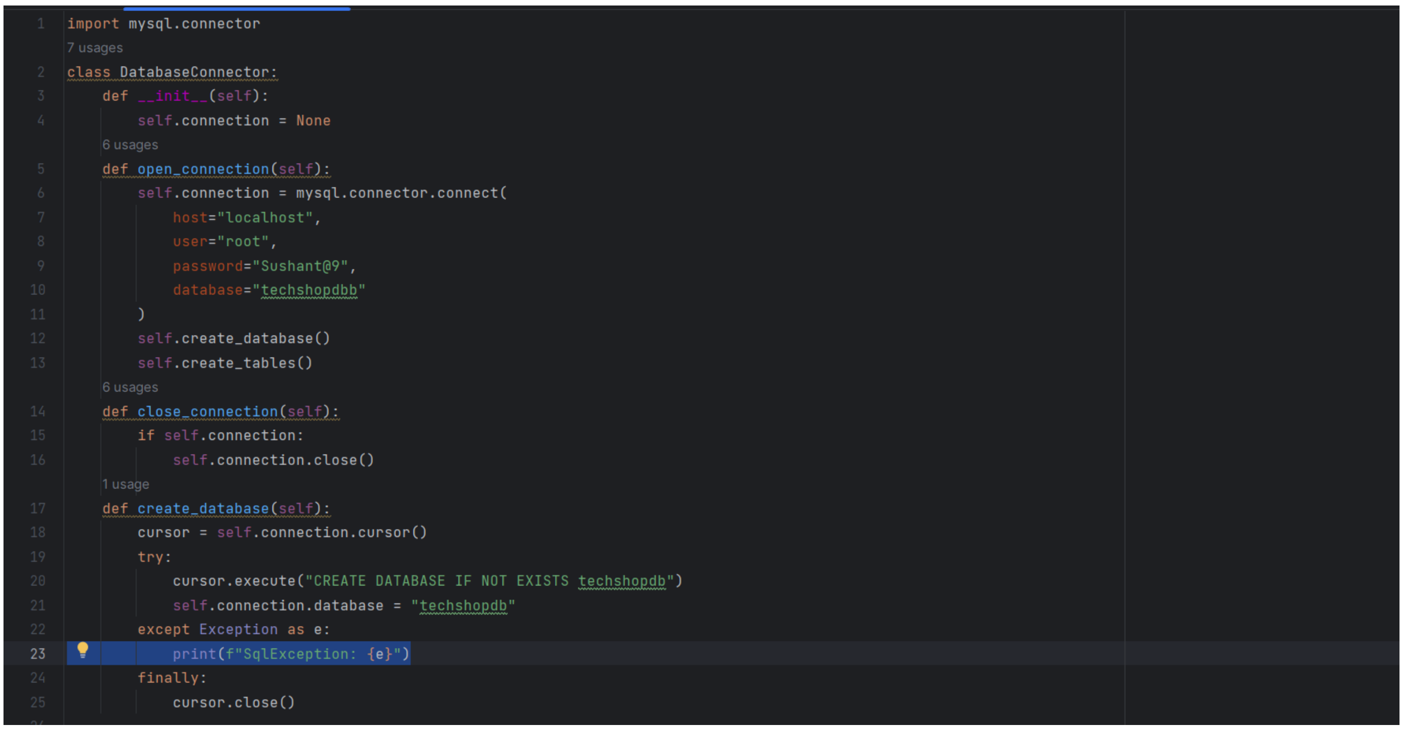


```python
def insert_order_detail(self, order_id, product_id, quantity):
    cursor = self.connection.cursor()
    try:
        query = "INSERT INTO OrderDetails (OrderID, ProductID, Quantity) VALUES (%s, %s, %s)"
        values = (order_id, product_id, quantity)
        cursor.execute(query, values)
        self.connection.commit()
        print("Order detail inserted successfully.")
    except Exception as e:
        print(f"IncompleteOrderException: {e}")
    finally:
        cursor.close()
```

• **Database Access:**

o Challenge: Managing database connections and queries.

o Exception Handling: Handle database-specific exceptions (e.g., SqlException) and implement connection retries or failover mechanisms.



```
import mysql.connector
7 usages
class DatabaseConnector:
    def __init__(self):
        self.connection = None
    6 usages
    def open_connection(self):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="Sushant@9",
            database="techshopdbb"
        )
        self.create_database()
        self.create_tables()
    6 usages
    def close_connection(self):
        if self.connection:
            self.connection.close()
    1 usage
    def create_database(self):
        cursor = self.connection.cursor()
        try:
            cursor.execute("CREATE DATABASE IF NOT EXISTS techshopdb")
            self.connection.database = "techshopdb"
        except Exception as e:
            print(f"SqlException: {e}")
        finally:
            cursor.close()
```

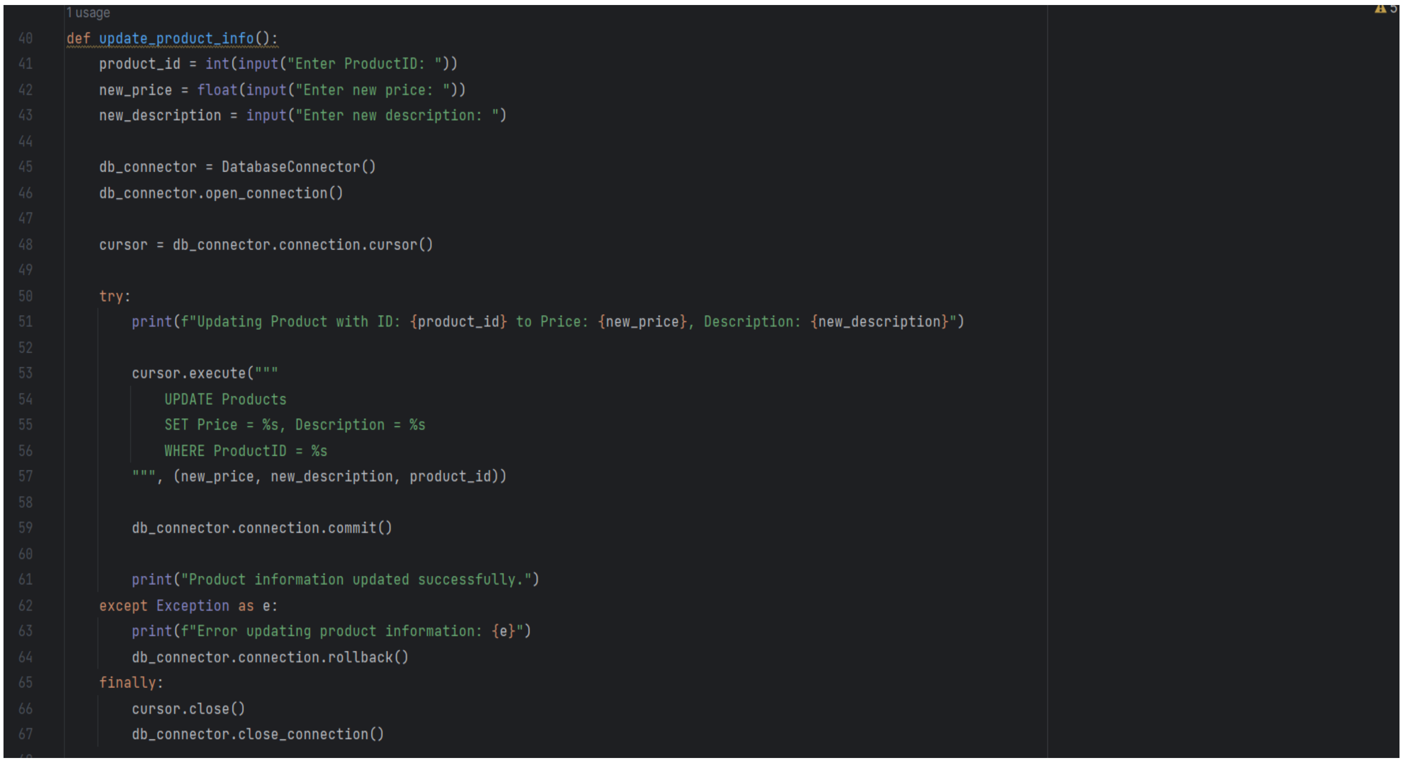
Task 6: Collections

• **Managing Products List:**

o Challenge: Maintaining a list of products available for sale (List<Products>).

o Scenario: Adding, updating, products from the list.

o Solution: Implement methods to add, update, and remove products. Handle exceptions for duplicate products, invalid updates, or removal of products with existing orders.



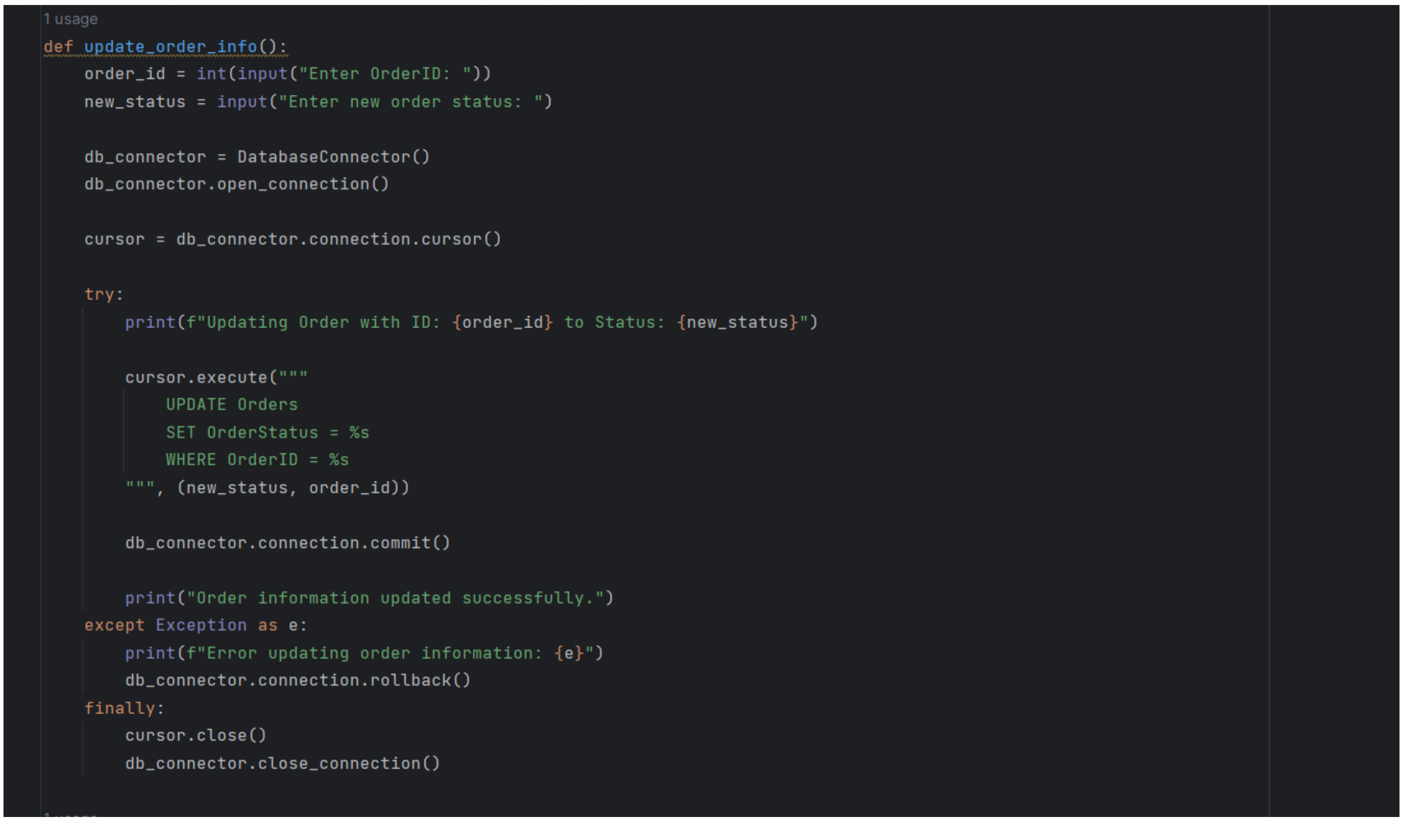
```
1 usage
def update_product_info():
    product_id = int(input("Enter ProductID: "))
    new_price = float(input("Enter new price: "))
    new_description = input("Enter new description: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Product with ID: {product_id} to Price: {new_price}, Description: {new_description}")

        cursor.execute("""
            UPDATE Products
            SET Price = %s, Description = %s
            WHERE ProductID = %s
        """, (new_price, new_description, product_id))

        db_connector.connection.commit()

        print("Product information updated successfully.")
    except Exception as e:
        print(f"Error updating product information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

• **Managing Orders List:**

o Challenge: Maintaining a list of customer orders (List<Orders>).

o Scenario: Adding new orders, updating order statuses, orders.

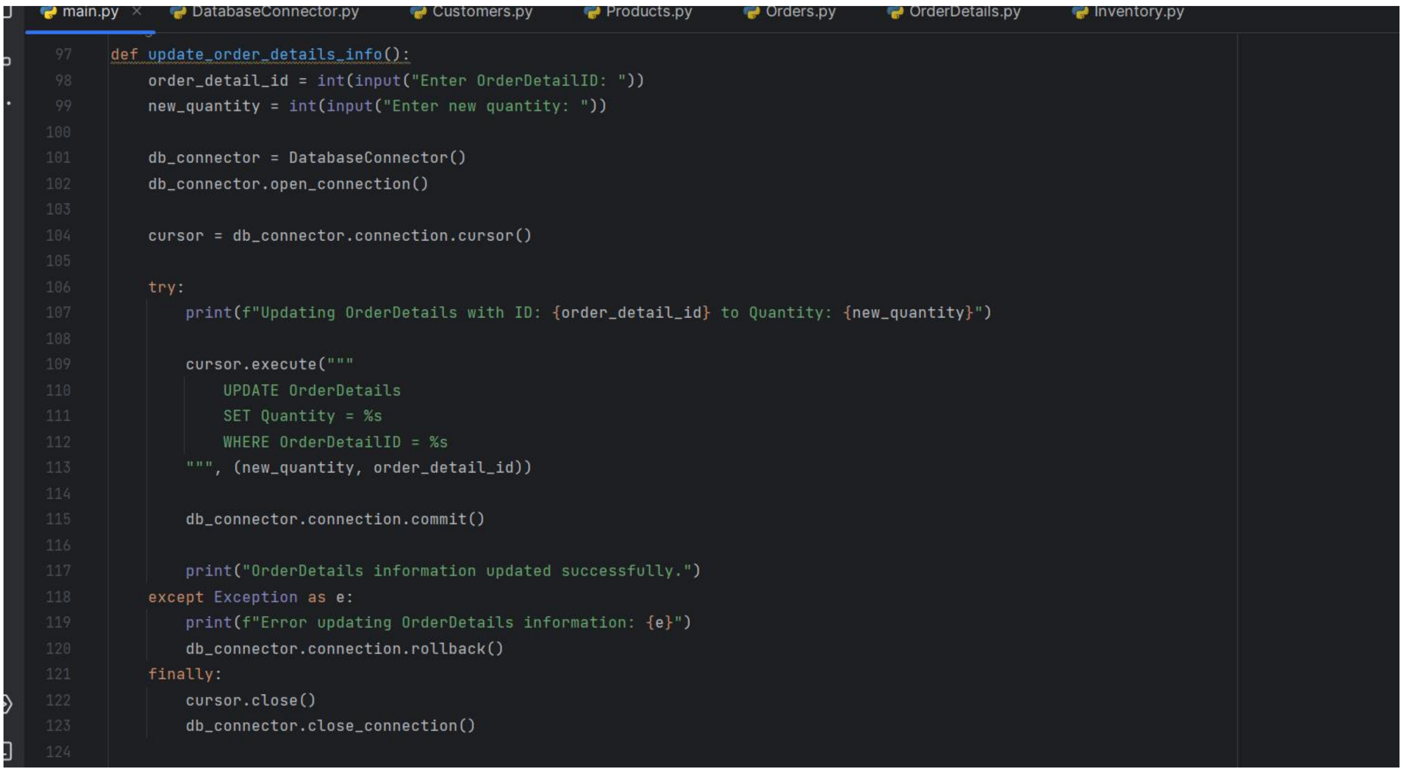
o Solution: Implement methods to add new orders, update order statuses, and remove canceled orders. Ensure that updates are synchronized with inventory and payment records.



```
1 usage
def update_order_info():
    order_id = int(input("Enter OrderID: "))
    new_status = input("Enter new order status: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Order with ID: {order_id} to Status: {new_status}")

        cursor.execute("""
            UPDATE Orders
            SET OrderStatus = %s
            WHERE OrderID = %s
        """, (new_status, order_id))

        db_connector.connection.commit()

        print("Order information updated successfully.")
    except Exception as e:
        print(f"Error updating order information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

• **Sorting Orders by Date:**

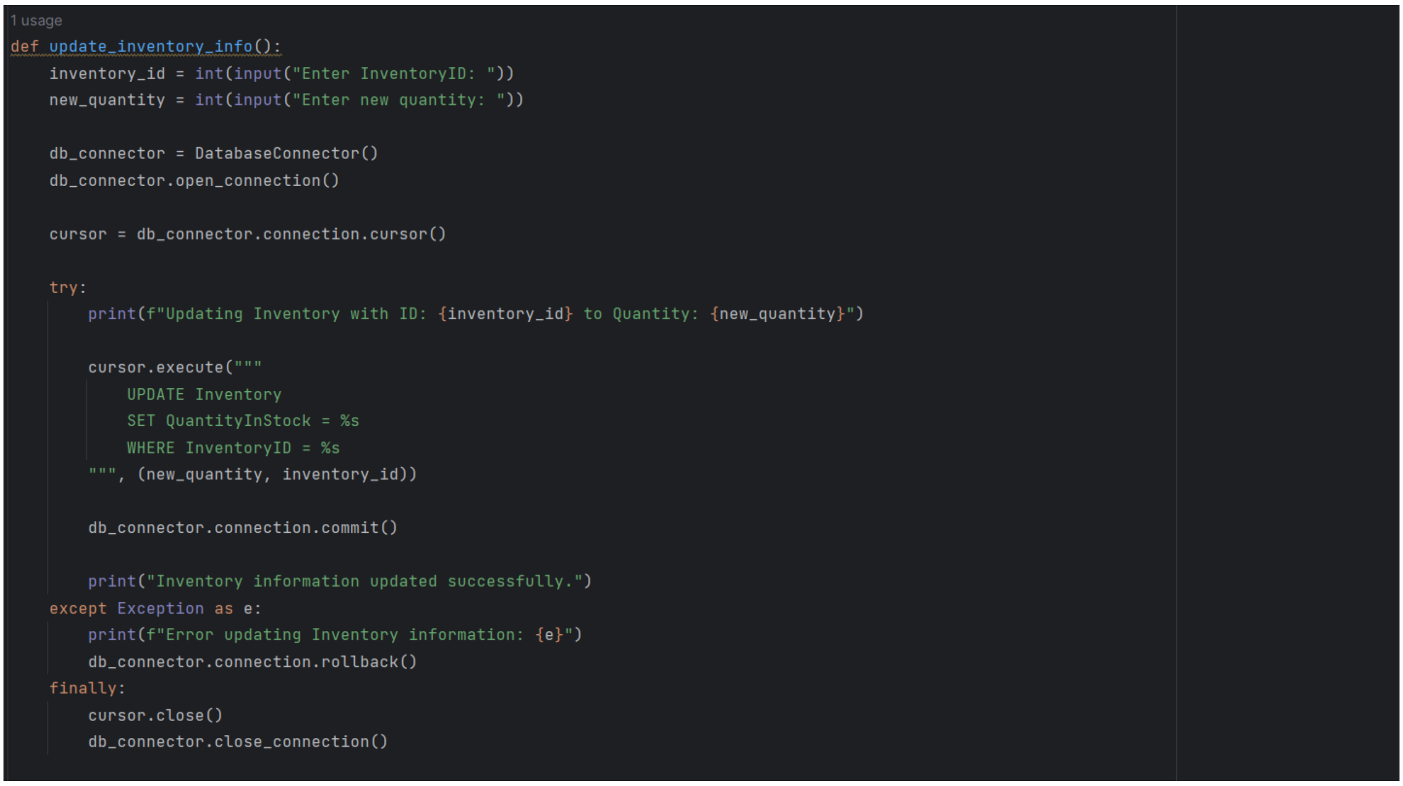
o Challenge: Sorting orders by order date in ascending or descending order.



```python
def update_order_details_info():
    order_detail_id = int(input("Enter OrderDetailID: "))
    new_quantity = int(input("Enter new quantity: "))

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating OrderDetails with ID: {order_detail_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE OrderDetails
            SET Quantity = %s
            WHERE OrderDetailID = %s
        """, (new_quantity, order_detail_id))

        db_connector.connection.commit()

        print("OrderDetails information updated successfully.")
    except Exception as e:
        print(f"Error updating OrderDetails information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

• **Handling Inventory Updates:**

o Challenge: Ensuring that inventory is updated correctly when processing orders.



```python
def update_inventory_info():
    inventory_id = int(input("Enter InventoryID: "))
    new_quantity = int(input("Enter new quantity: "))

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Inventory with ID: {inventory_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE Inventory
            SET QuantityInStock = %s
            WHERE InventoryID = %s
        """, (new_quantity, inventory_id))

        db_connector.connection.commit()

        print("Inventory information updated successfully.")
    except Exception as e:
        print(f"Error updating Inventory information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

Task 7: Database Connectivity

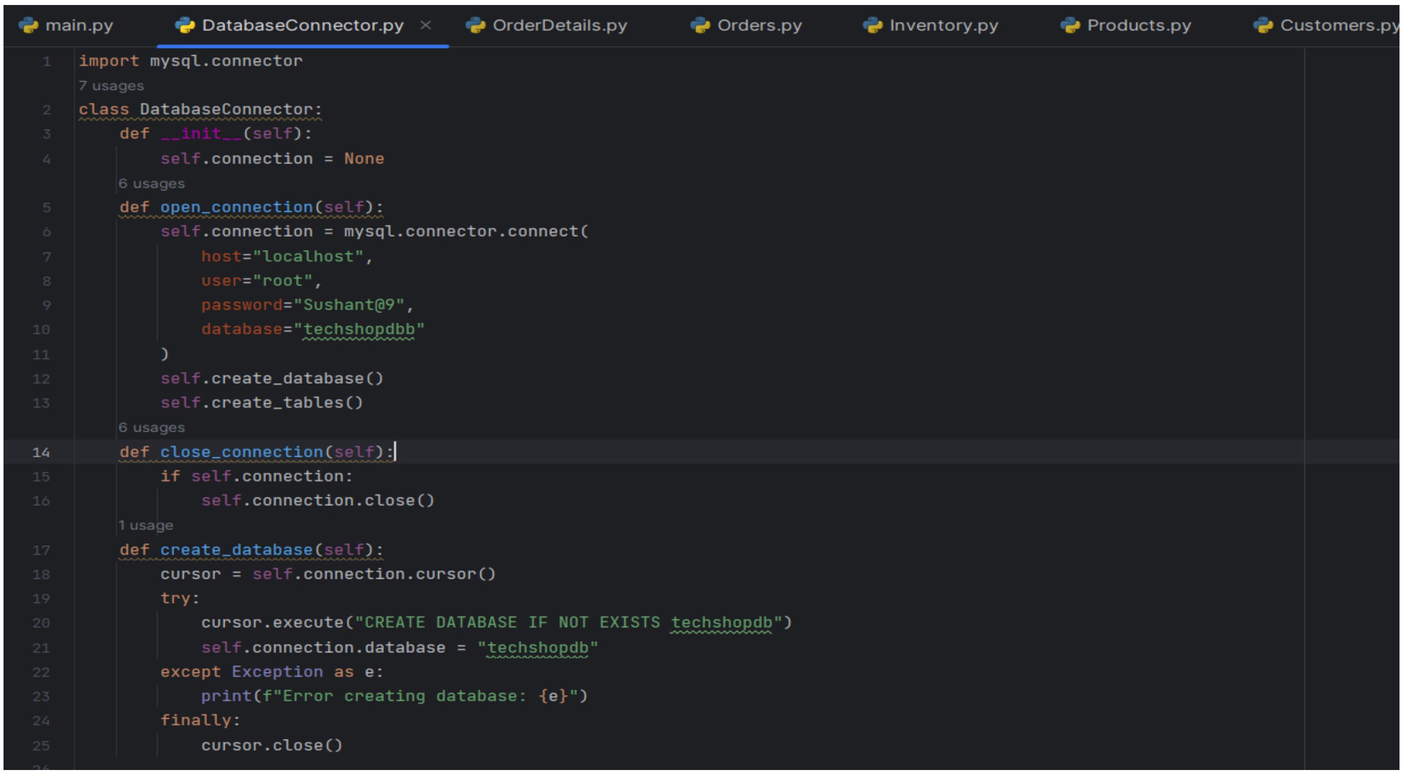
• Implement a DatabaseConnector class responsible for establishing a connection to the "TechShopDB" database. This class should include methods for opening, closing, and managing database connections.

• Implement classes for Customers, Products, Orders, OrderDetails, Inventory with properties, constructors, and methods for CRUD (Create, Read, Update, Delete) operations.

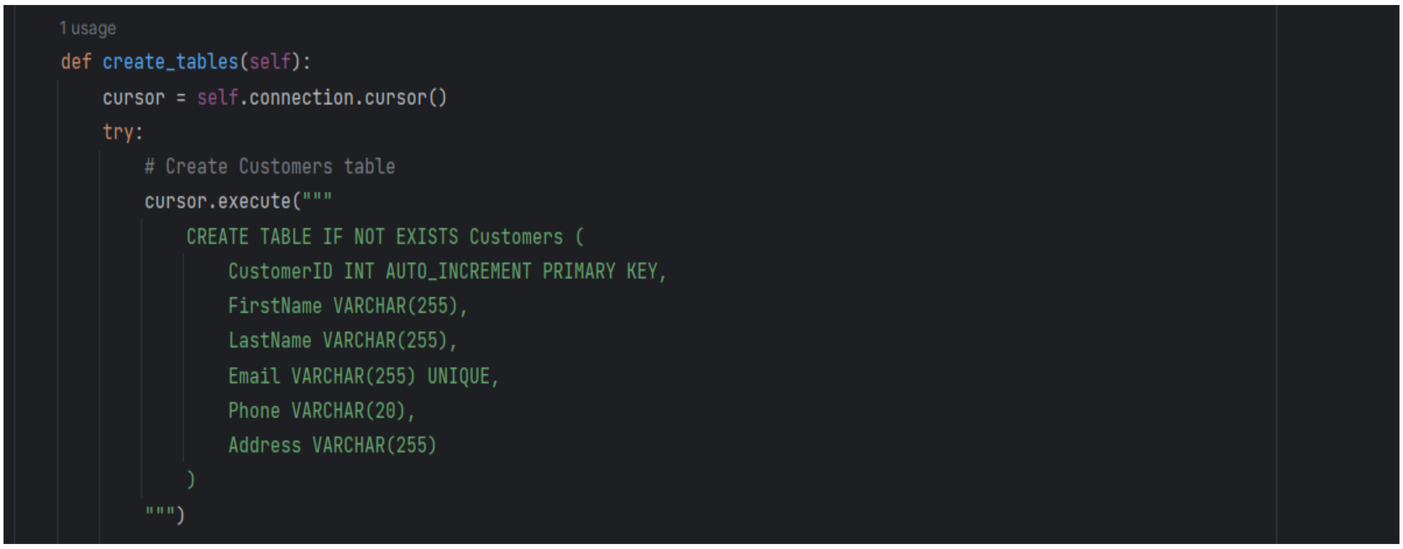
1: Customer Registration

Description: When a new customer registers on the TechShop website, their information (e.g., name, email, phone) needs to be stored in the database.

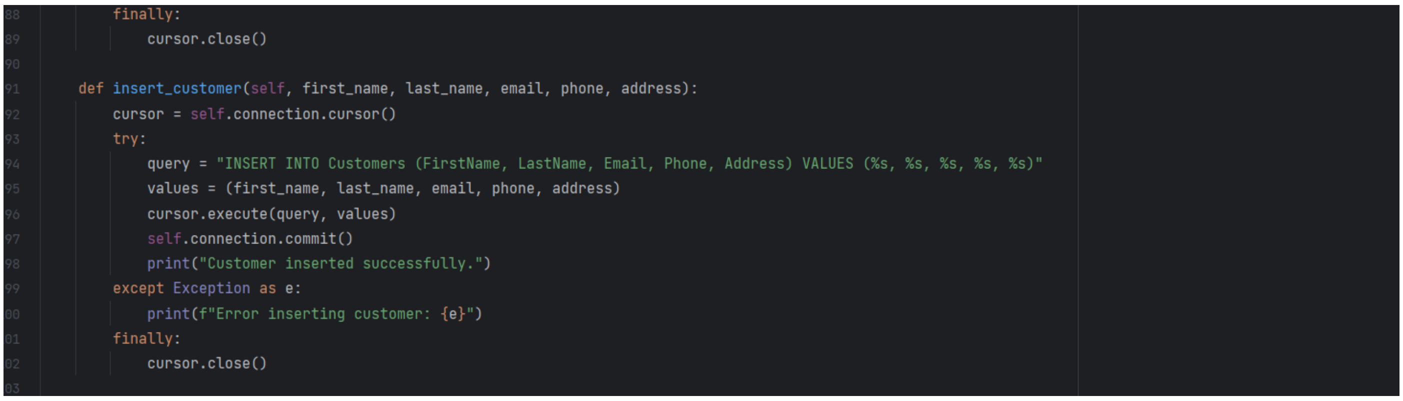
Task: Implement a registration form and database connectivity to insert new customer records. Ensure proper data validation and error handling for duplicate email addresses.



```python
import mysql.connector
7 usages
class DatabaseConnector:
    def __init__(self):
        self.connection = None
    6 usages
    def open_connection(self):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="Sushant@9",
            database="techshopdbb"
        )
        self.create_database()
        self.create_tables()
    6 usages
    def close_connection(self):
        if self.connection:
            self.connection.close()
    1 usage
    def create_database(self):
        cursor = self.connection.cursor()
        try:
            cursor.execute("CREATE DATABASE IF NOT EXISTS techshopdb")
            self.connection.database = "techshopdb"
        except Exception as e:
            print(f"Error creating database: {e}")
        finally:
            cursor.close()
```



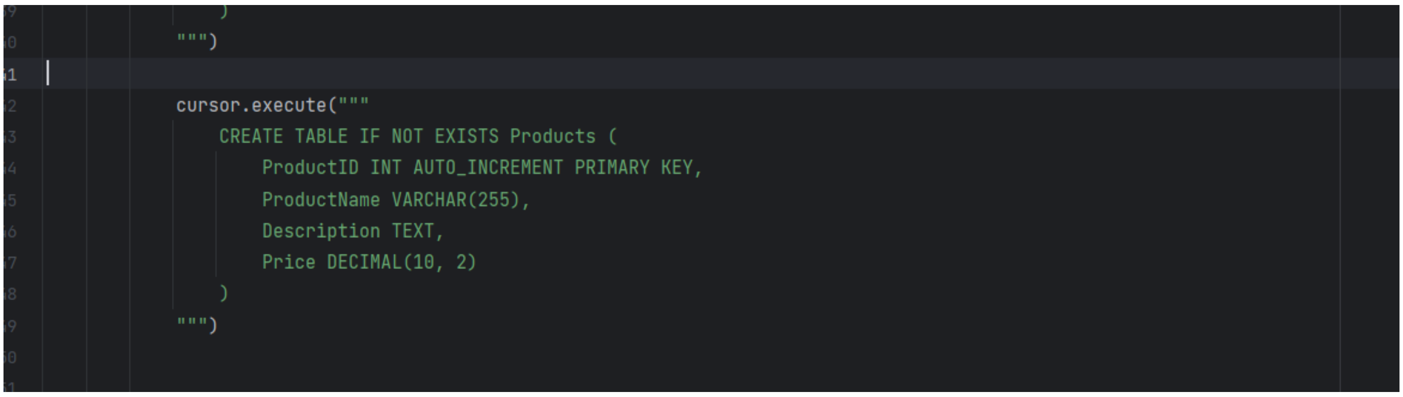
```python
1 usage
def create_tables(self):
    cursor = self.connection.cursor()
    try:
        # Create Customers table
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS Customers (
                CustomerID INT AUTO_INCREMENT PRIMARY KEY,
                FirstName VARCHAR(255),
                LastName VARCHAR(255),
                Email VARCHAR(255) UNIQUE,
                Phone VARCHAR(20),
                Address VARCHAR(255)
            )
        """)
```



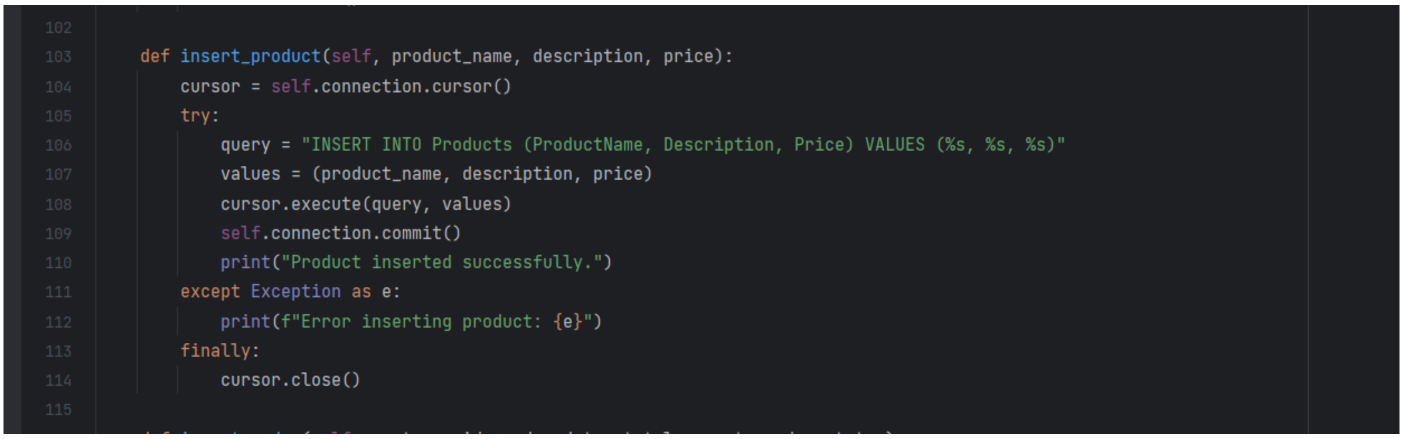
```
        finally:
            cursor.close()

    def insert_customer(self, first_name, last_name, email, phone, address):
        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO Customers (FirstName, LastName, Email, Phone, Address) VALUES (%s, %s, %s, %s, %s)"
            values = (first_name, last_name, email, phone, address)
            cursor.execute(query, values)
            self.connection.commit()
            print("Customer inserted successfully.")
        except Exception as e:
            print(f"Error inserting customer: {e}")
        finally:
            cursor.close()
```

2: Product Catalog Management

Description: TechShop regularly updates its product catalog with new items and changes in product details (e.g., price, description). These changes need to be reflected in the database.



```
            )
        """)

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS Products (
                ProductID INT AUTO_INCREMENT PRIMARY KEY,
                ProductName VARCHAR(255),
                Description TEXT,
                Price DECIMAL(10, 2)
            )
        """)
```



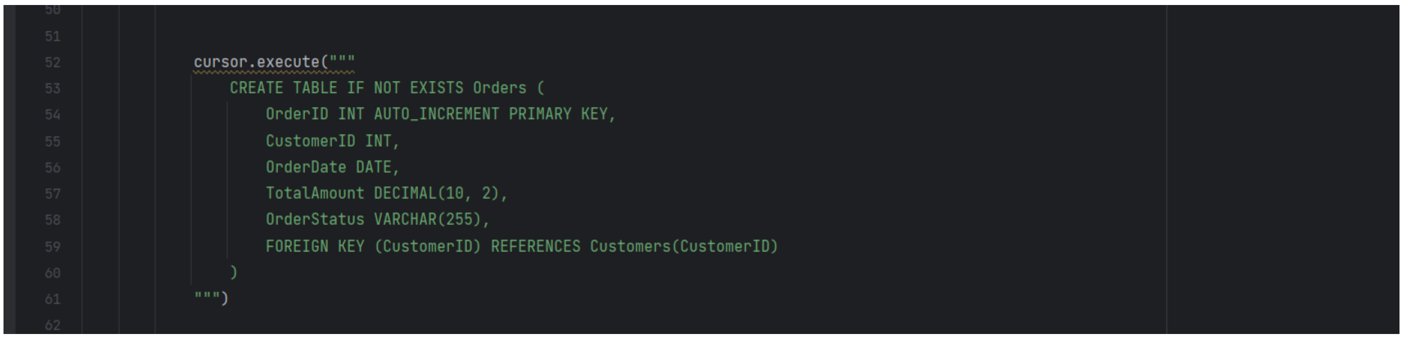
```
    def insert_product(self, product_name, description, price):
        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO Products (ProductName, Description, Price) VALUES (%s, %s, %s)"
            values = (product_name, description, price)
            cursor.execute(query, values)
            self.connection.commit()
            print("Product inserted successfully.")
        except Exception as e:
            print(f"Error inserting product: {e}")
        finally:
            cursor.close()
```

3: Placing Customer Orders

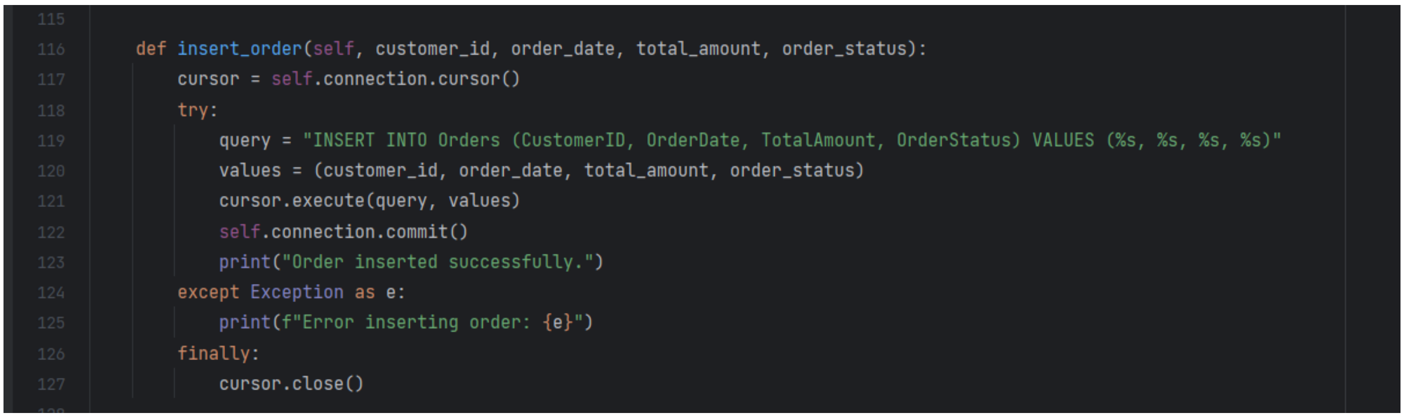
Description: Customers browse the product catalog and place orders for products they want to purchase. The orders need to be stored in the database.

Task: Implement an order processing system. Use database connectivity to record customer orders,

.



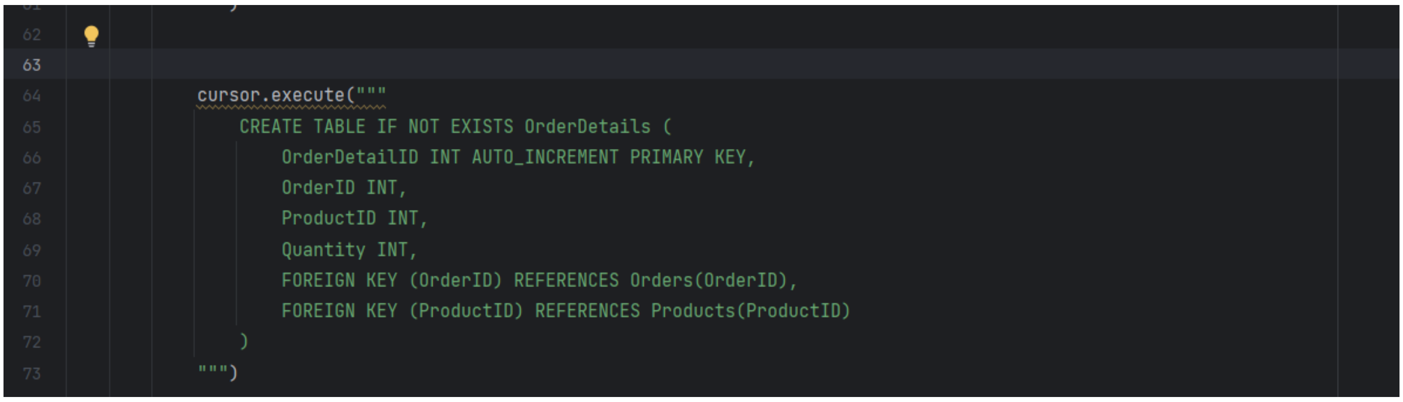
```
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS Orders (
                OrderID INT AUTO_INCREMENT PRIMARY KEY,
                CustomerID INT,
                OrderDate DATE,
                TotalAmount DECIMAL(10, 2),
                OrderStatus VARCHAR(255),
                FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
            )
        """)
```



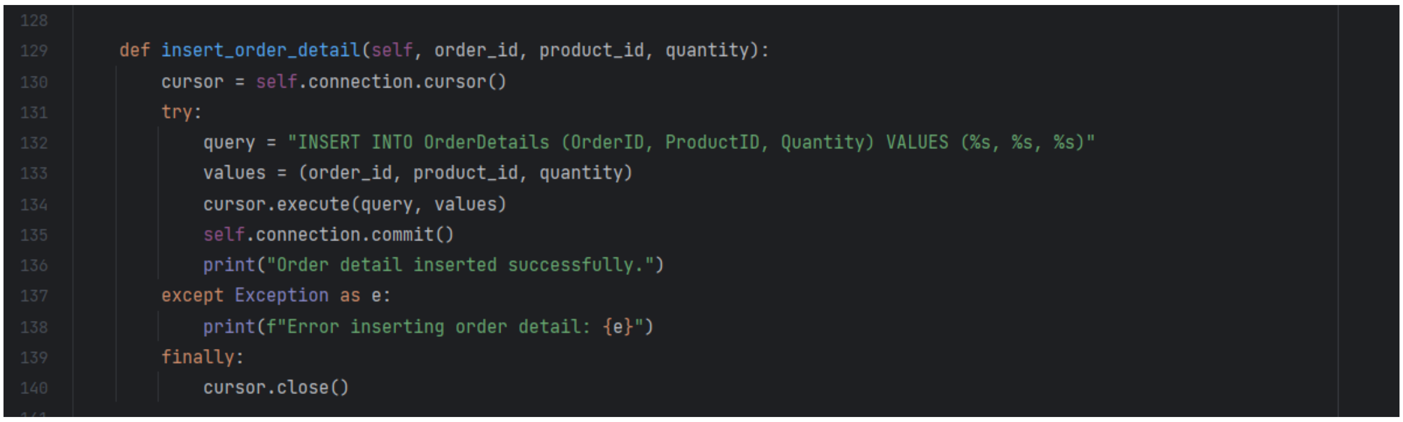
```
    def insert_order(self, customer_id, order_date, total_amount, order_status):
        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO Orders (CustomerID, OrderDate, TotalAmount, OrderStatus) VALUES (%s, %s, %s, %s)"
            values = (customer_id, order_date, total_amount, order_status)
            cursor.execute(query, values)
            self.connection.commit()
            print("Order inserted successfully.")
        except Exception as e:
            print(f"Error inserting order: {e}")
        finally:
            cursor.close()
```

4: Tracking Order Status

Description: Customers and employees need to track the status of their orders. The order status information is stored in the database.



```
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS OrderDetails (
                OrderDetailID INT AUTO_INCREMENT PRIMARY KEY,
                OrderID INT,
                ProductID INT,
                Quantity INT,
                FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
                FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
            )
        """)
```

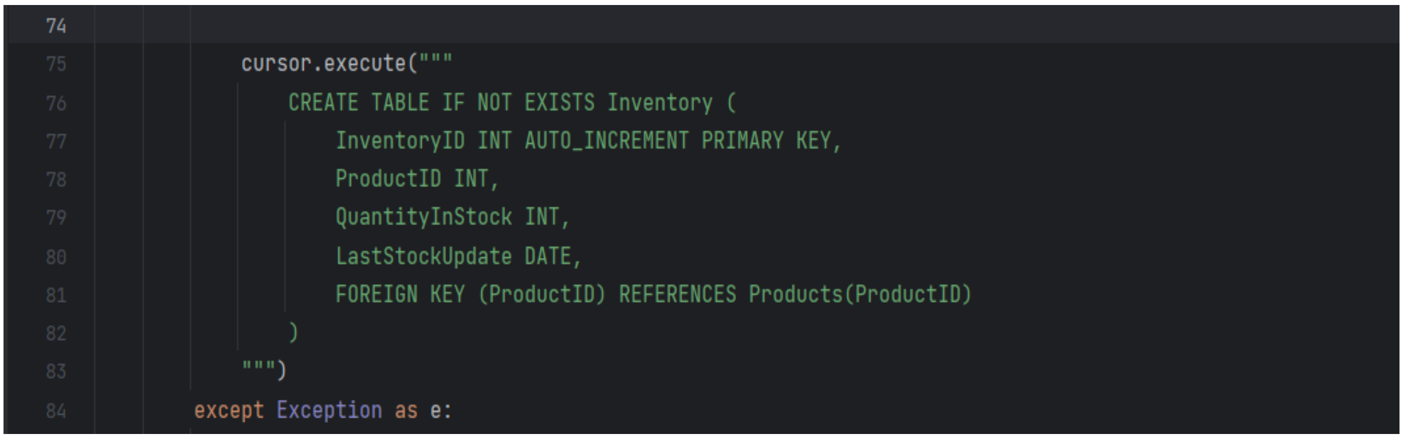


```python
    def insert_order_detail(self, order_id, product_id, quantity):
        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO OrderDetails (OrderID, ProductID, Quantity) VALUES (%s, %s, %s)"
            values = (order_id, product_id, quantity)
            cursor.execute(query, values)
            self.connection.commit()
            print("Order detail inserted successfully.")
        except Exception as e:
            print(f"Error inserting order detail: {e}")
        finally:
            cursor.close()
```

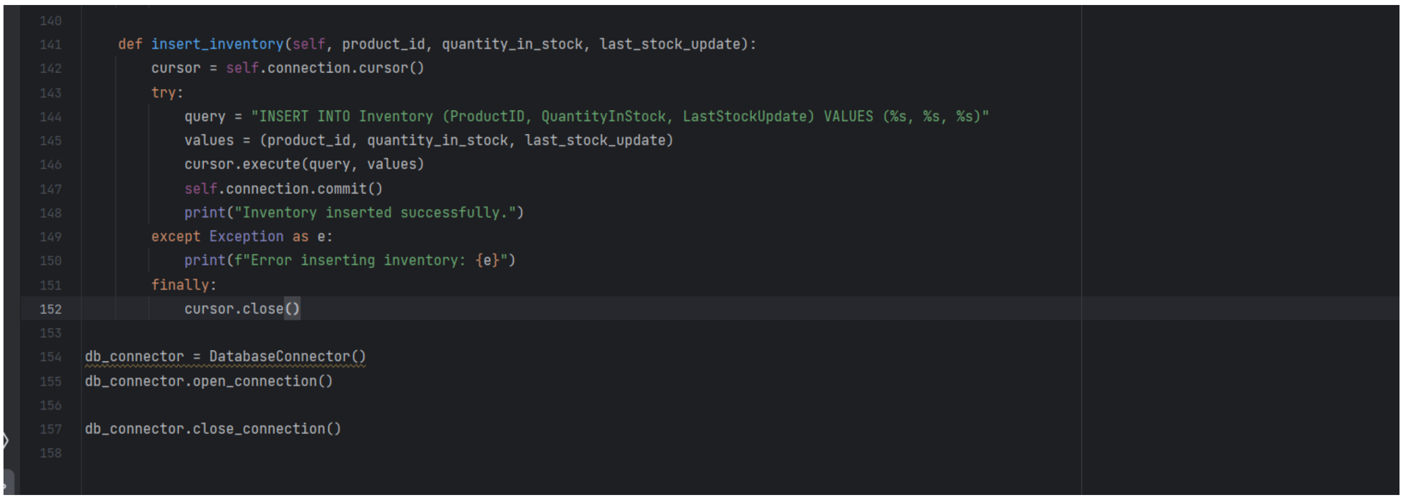
5: Inventory Management

Description: TechShop needs to manage product inventory, including adding new products, updating stock levels, and removing discontinued items.

Task: Create an inventory management system with database connectivity.



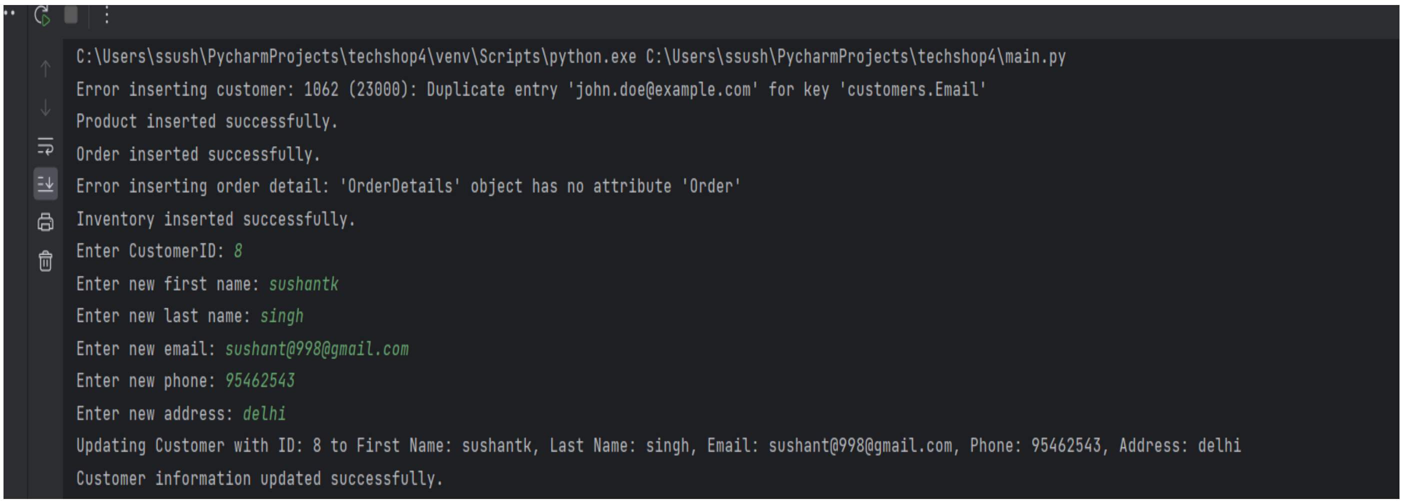
```python

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS Inventory (
                    InventoryID INT AUTO_INCREMENT PRIMARY KEY,
                    ProductID INT,
                    QuantityInStock INT,
                    LastStockUpdate DATE,
                    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
                )
            """)
        except Exception as e:
```



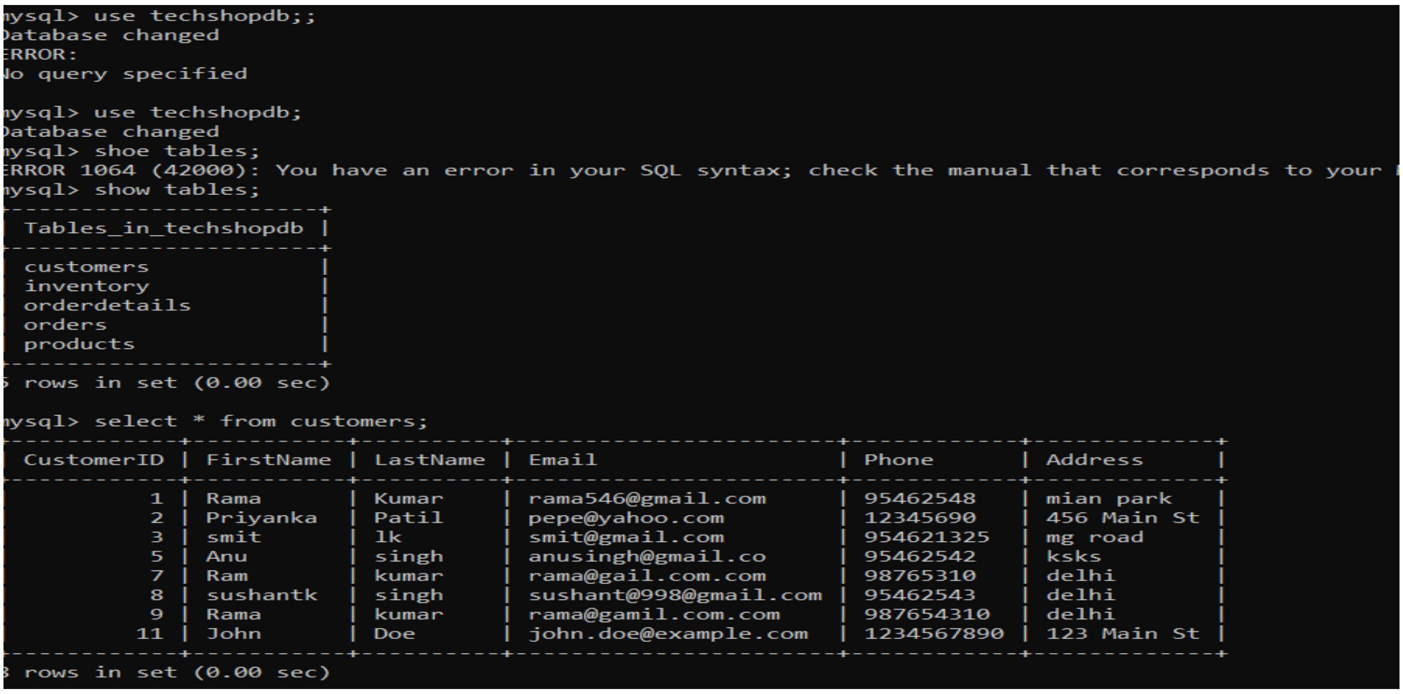
```python

    def insert_inventory(self, product_id, quantity_in_stock, last_stock_update):
        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO Inventory (ProductID, QuantityInStock, LastStockUpdate) VALUES (%s, %s, %s)"
            values = (product_id, quantity_in_stock, last_stock_update)
            cursor.execute(query, values)
            self.connection.commit()
            print("Inventory inserted successfully.")
        except Exception as e:
            print(f"Error inserting inventory: {e}")
        finally:
            cursor.close()

db_connector = DatabaseConnector()
db_connector.open_connection()

db_connector.close_connection()
```

OUTPUTS **(** Showing the output of each given case according to the question's)

Customer Input/Output

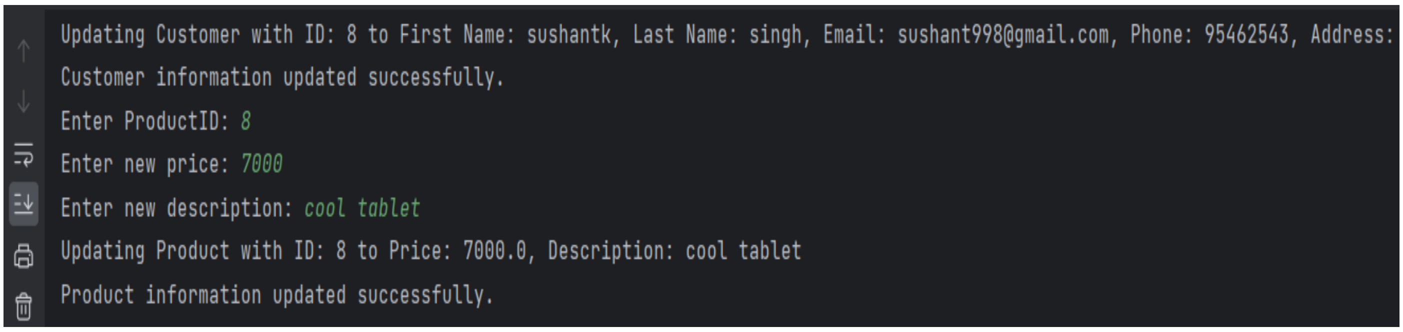


```
C:\Users\ssush\PycharmProjects\techshop4\venv\Scripts\python.exe C:\Users\ssush\PycharmProjects\techshop4\main.py
Error inserting customer: 1062 (23000): Duplicate entry 'john.doe@example.com' for key 'customers.Email'
Product inserted successfully.
Order inserted successfully.
Error inserting order detail: 'OrderDetails' object has no attribute 'Order'
Inventory inserted successfully.
Enter CustomerID: 8
Enter new first name: sushantk
Enter new last name: singh
Enter new email: sushant@998@gmail.com
Enter new phone: 95462543
Enter new address: delhi
Updating Customer with ID: 8 to First Name: sushantk, Last Name: singh, Email: sushant@998@gmail.com, Phone: 95462543, Address: delhi
Customer information updated successfully.
```

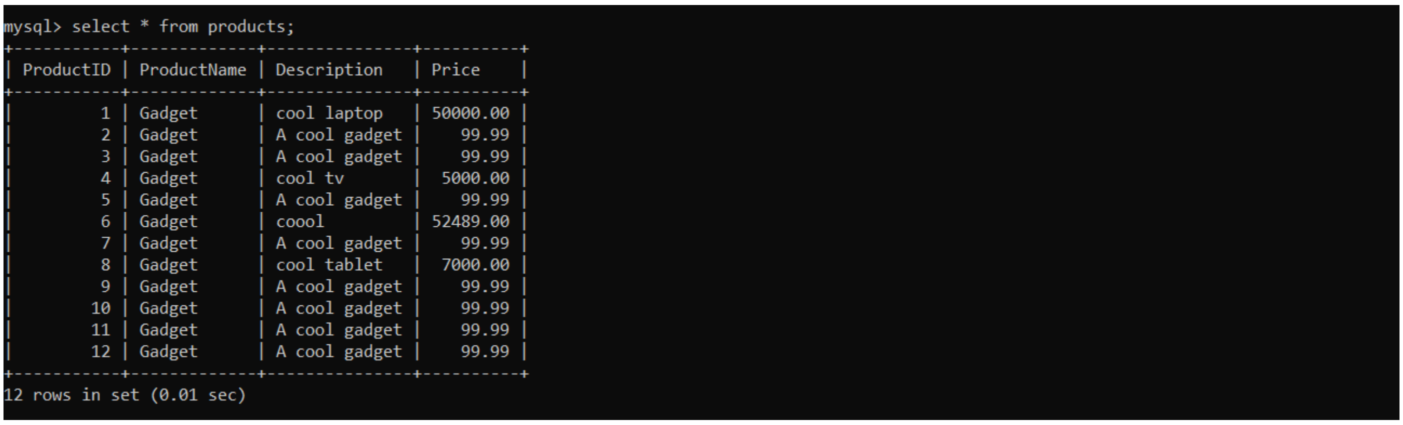


```
ysql> use techshopdb;;
atabase changed
RROR:
o query specified

ysql> use techshopdb;
atabase changed
ysql> shoe tables;
RROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
ysql> show tables;
----------------------+
 Tables_in_techshopdb |
----------------------+
 customers            |
 inventory            |
 orderdetails         |
 orders               |
 products             |
----------------------+
 rows in set (0.00 sec)

ysql> select * from customers;
------------+-----------+-----------+----------------------+-------------+-------------+
 CustomerID | FirstName | LastName  | Email                | Phone       | Address     |
------------+-----------+-----------+----------------------+-------------+-------------+
          1 | Rama      | Kumar     | rama546@gmail.com    | 95462548    | mian park   |
          2 | Priyanka  | Patil     | pepe@yahoo.com       | 12345690    | 456 Main St |
          3 | smit      | lk        | smit@gmail.com       | 954621325   | mg road     |
          5 | Anu       | singh     | anusingh@gmail.co    | 95462542    | ksks        |
          7 | Ram       | kumar     | rama@gail.com.com    | 98765310    | delhi       |
          8 | sushantk  | singh     | sushant@998@gmail.com | 95462543   | delhi       |
          9 | Rama      | kumar     | rama@gamil.com.com   | 987654310   | delhi       |
         11 | John      | Doe       | john.doe@example.com | 1234567890  | 123 Main St |
------------+-----------+-----------+----------------------+-------------+-------------+
 rows in set (0.00 sec)
```

Products Input/Output

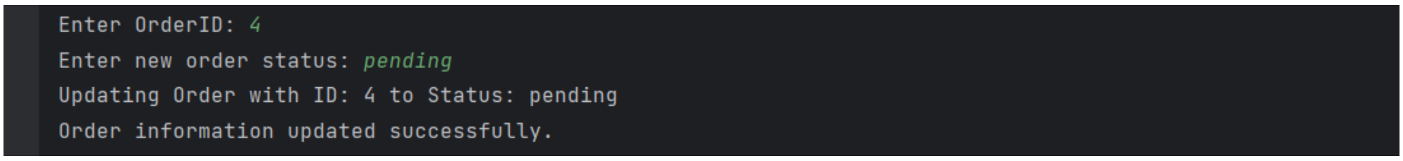


```
Updating Customer with ID: 8 to First Name: sushantk, Last Name: singh, Email: sushant998@gmail.com, Phone: 95462543, Address:
Customer information updated successfully.
Enter ProductID: 8
Enter new price: 7000
Enter new description: cool tablet
Updating Product with ID: 8 to Price: 7000.0, Description: cool tablet
Product information updated successfully.
```

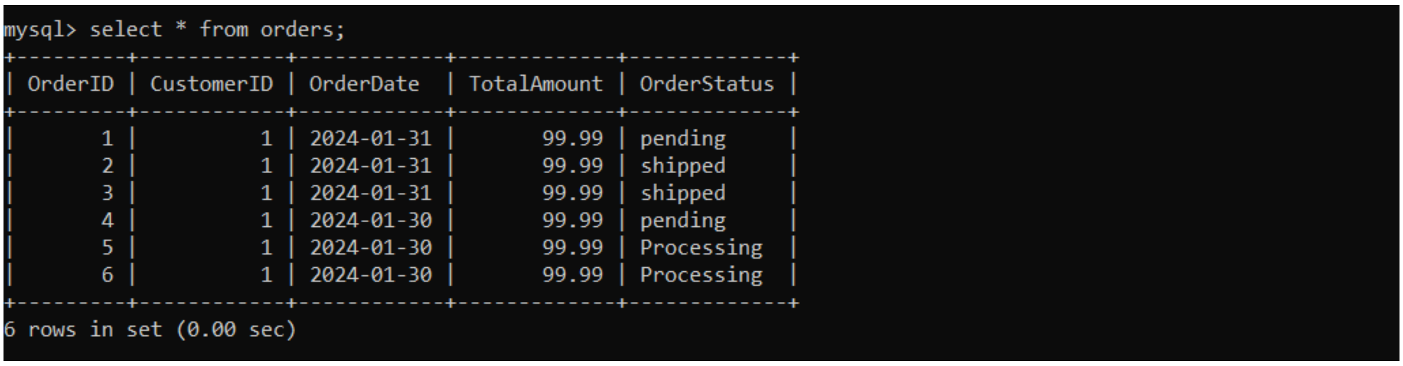


```
mysql> select * from products;
+-----------+-------------+---------------+----------+
| ProductID | ProductName | Description   | Price    |
+-----------+-------------+---------------+----------+
|         1 | Gadget      | cool laptop   | 50000.00 |
|         2 | Gadget      | A cool gadget |    99.99 |
|         3 | Gadget      | A cool gadget |    99.99 |
|         4 | Gadget      | cool tv       |  5000.00 |
|         5 | Gadget      | A cool gadget |    99.99 |
|         6 | Gadget      | coool         | 52489.00 |
|         7 | Gadget      | A cool gadget |    99.99 |
|         8 | Gadget      | cool tablet   |  7000.00 |
|         9 | Gadget      | A cool gadget |    99.99 |
|        10 | Gadget      | A cool gadget |    99.99 |
|        11 | Gadget      | A cool gadget |    99.99 |
|        12 | Gadget      | A cool gadget |    99.99 |
+-----------+-------------+---------------+----------+
12 rows in set (0.01 sec)
```

Orders Input/Output



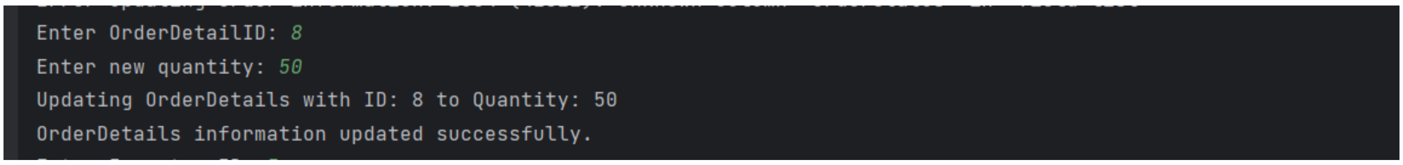
```
Enter OrderID: 4
Enter new order status: pending
Updating Order with ID: 4 to Status: pending
Order information updated successfully.
```



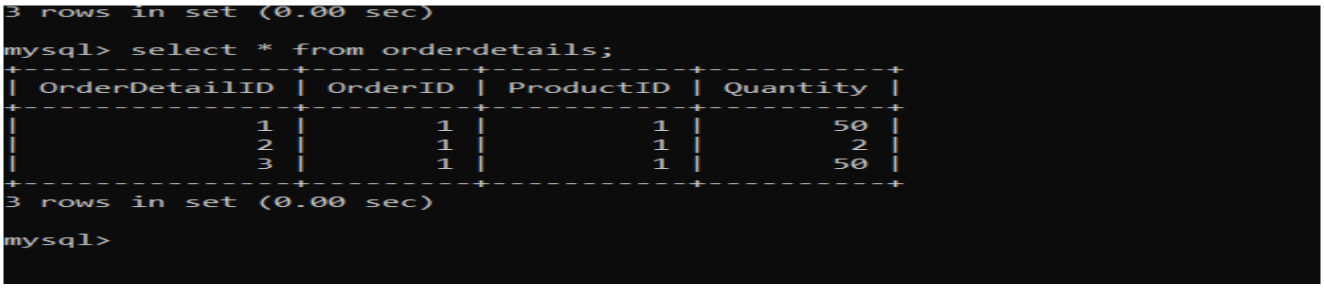
```
mysql> select * from orders;
+---------+------------+------------+-------------+-------------+
| OrderID | CustomerID | OrderDate  | TotalAmount | OrderStatus |
+---------+------------+------------+-------------+-------------+
|       1 |          1 | 2024-01-31 |       99.99 | pending     |
|       2 |          1 | 2024-01-31 |       99.99 | shipped     |
|       3 |          1 | 2024-01-31 |       99.99 | shipped     |
|       4 |          1 | 2024-01-30 |       99.99 | pending     |
|       5 |          1 | 2024-01-30 |       99.99 | Processing  |
|       6 |          1 | 2024-01-30 |       99.99 | Processing  |
+---------+------------+------------+-------------+-------------+
6 rows in set (0.00 sec)
```

OrderDetails Input/Output

.

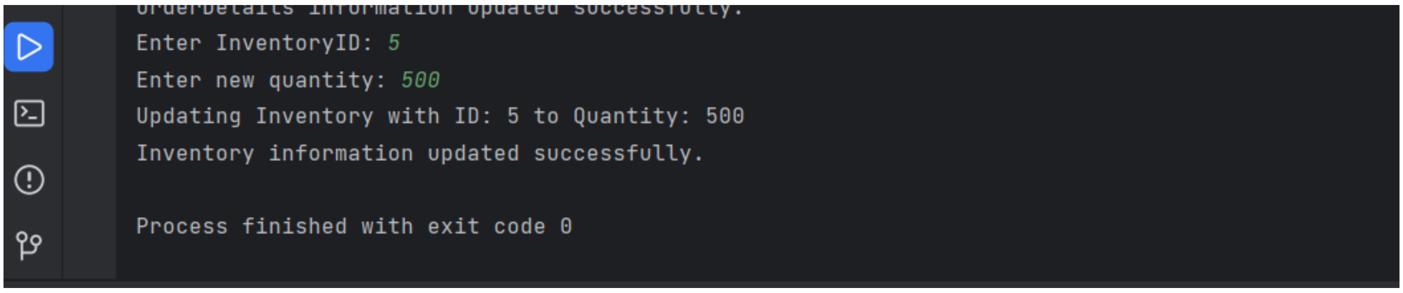


```
Enter OrderDetailID: 8
Enter new quantity: 50
Updating OrderDetails with ID: 8 to Quantity: 50
OrderDetails information updated successfully.
```

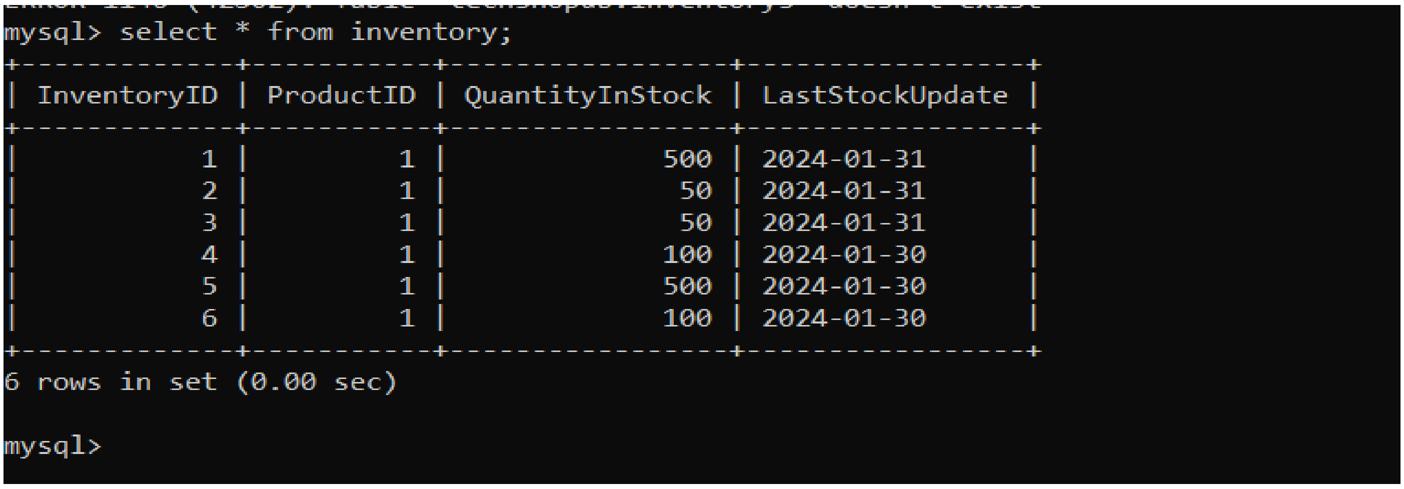


```
3 rows in set (0.00 sec)

mysql> select * from orderdetails;
+---------------+---------+-----------+----------+
| OrderDetailID | OrderID | ProductID | Quantity |
+---------------+---------+-----------+----------+
|             1 |       1 |         1 |       50 |
|             2 |       1 |         1 |        2 |
|             3 |       1 |         1 |       50 |
+---------------+---------+-----------+----------+
3 rows in set (0.00 sec)

mysql>
```

Inventory Input/Output



```
OrderDetails information updated successfully.
Enter InventoryID: 5
Enter new quantity: 500
Updating Inventory with ID: 5 to Quantity: 500
Inventory information updated successfully.

Process finished with exit code 0
```



```
mysql> select * from inventory;
+-------------+-----------+-----------------+-----------------+
| InventoryID | ProductID | QuantityInStock | LastStockUpdate |
+-------------+-----------+-----------------+-----------------+
|           1 |         1 |             500 | 2024-01-31      |
|           2 |         1 |              50 | 2024-01-31      |
|           3 |         1 |              50 | 2024-01-31      |
|           4 |         1 |             100 | 2024-01-30      |
|           5 |         1 |             500 | 2024-01-30      |
|           6 |         1 |             100 | 2024-01-30      |
+-------------+-----------+-----------------+-----------------+
6 rows in set (0.00 sec)

mysql>
```

******** **ThankYou** ********