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:

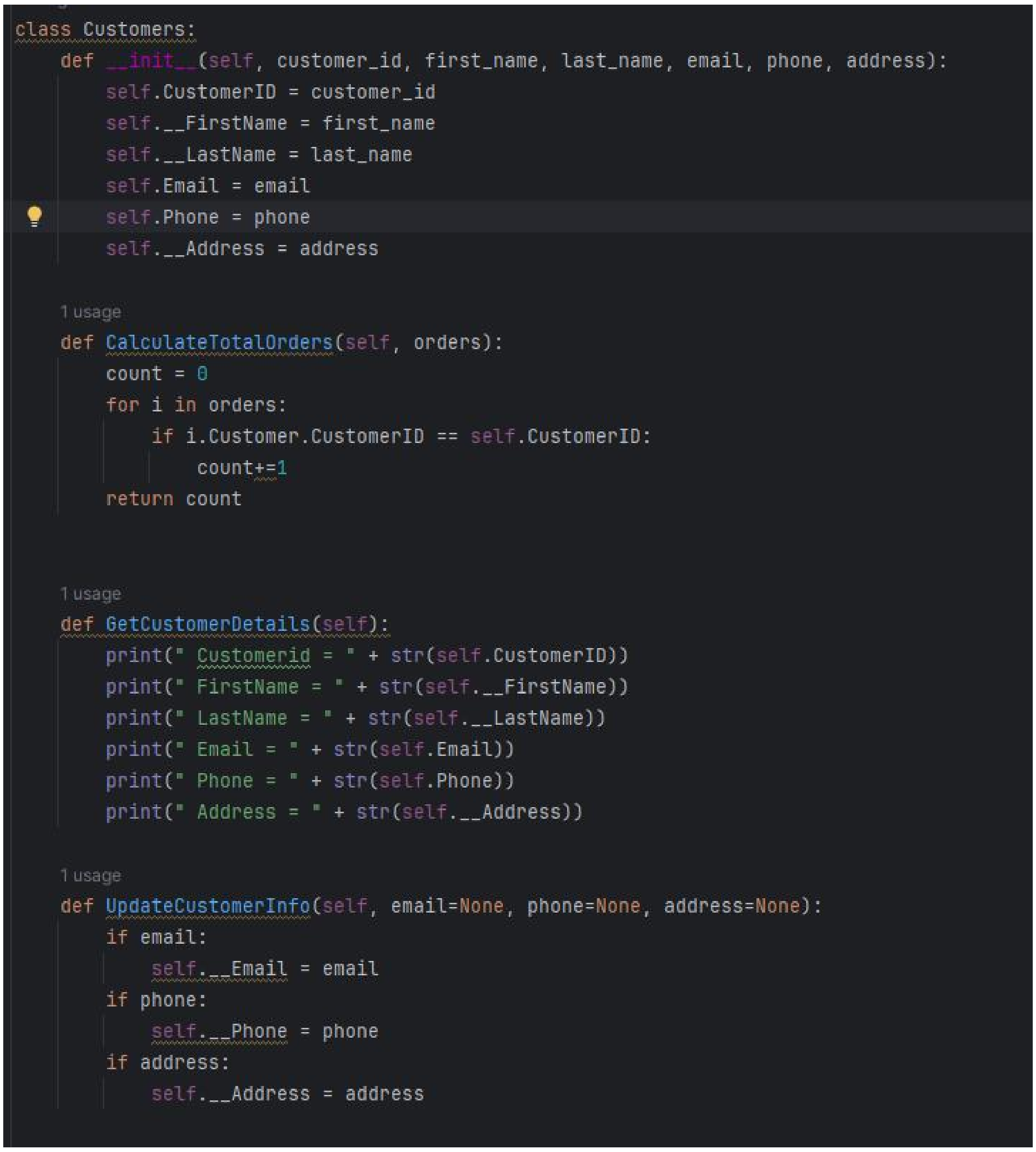
Customers Class:

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).



```
class Customers:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self.CustomerID = customer_id
        self.__FirstName = first_name
        self.__LastName = last_name
        self.Email = email
        self.Phone = phone
        self.__Address = address

    1 usage
    def CalculateTotalOrders(self, orders):
        count = 0
        for i in orders:
            if i.Customer.CustomerID == self.CustomerID:
                count+=1
        return count


    1 usage
    def GetCustomerDetails(self):
        print(" Customerid = " + str(self.CustomerID))
        print(" FirstName = " + str(self.__FirstName))
        print(" LastName = " + str(self.__LastName))
        print(" Email = " + str(self.Email))
        print(" Phone = " + str(self.Phone))
        print(" Address = " + str(self.__Address))

    1 usage
    def UpdateCustomerInfo(self, email=None, phone=None, address=None):
        if email:
            self.__Email = email
        if phone:
            self.__Phone = phone
        if address:
            self.__Address = address
```

Products Class:

Attributes:

• ProductID (int)

• ProductName (string)

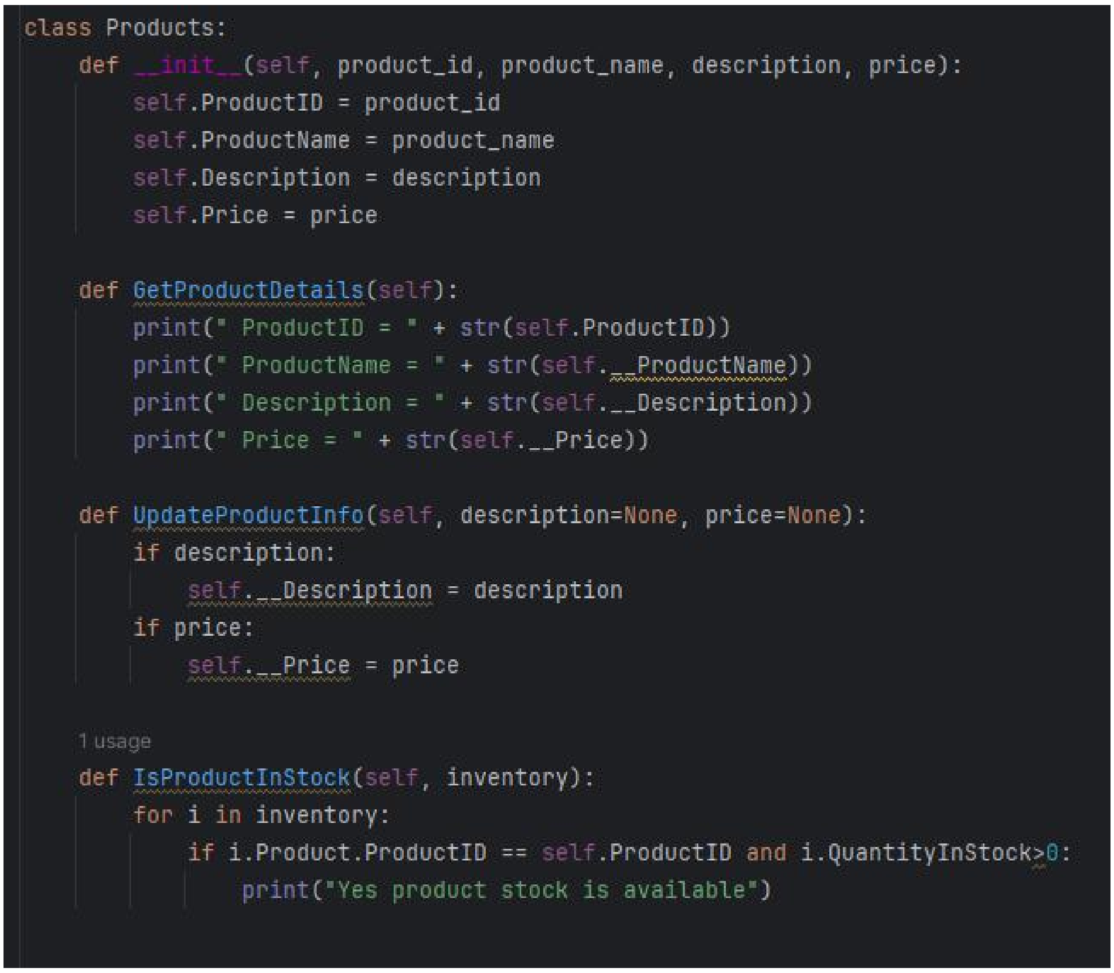
• Description (string)

• 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.



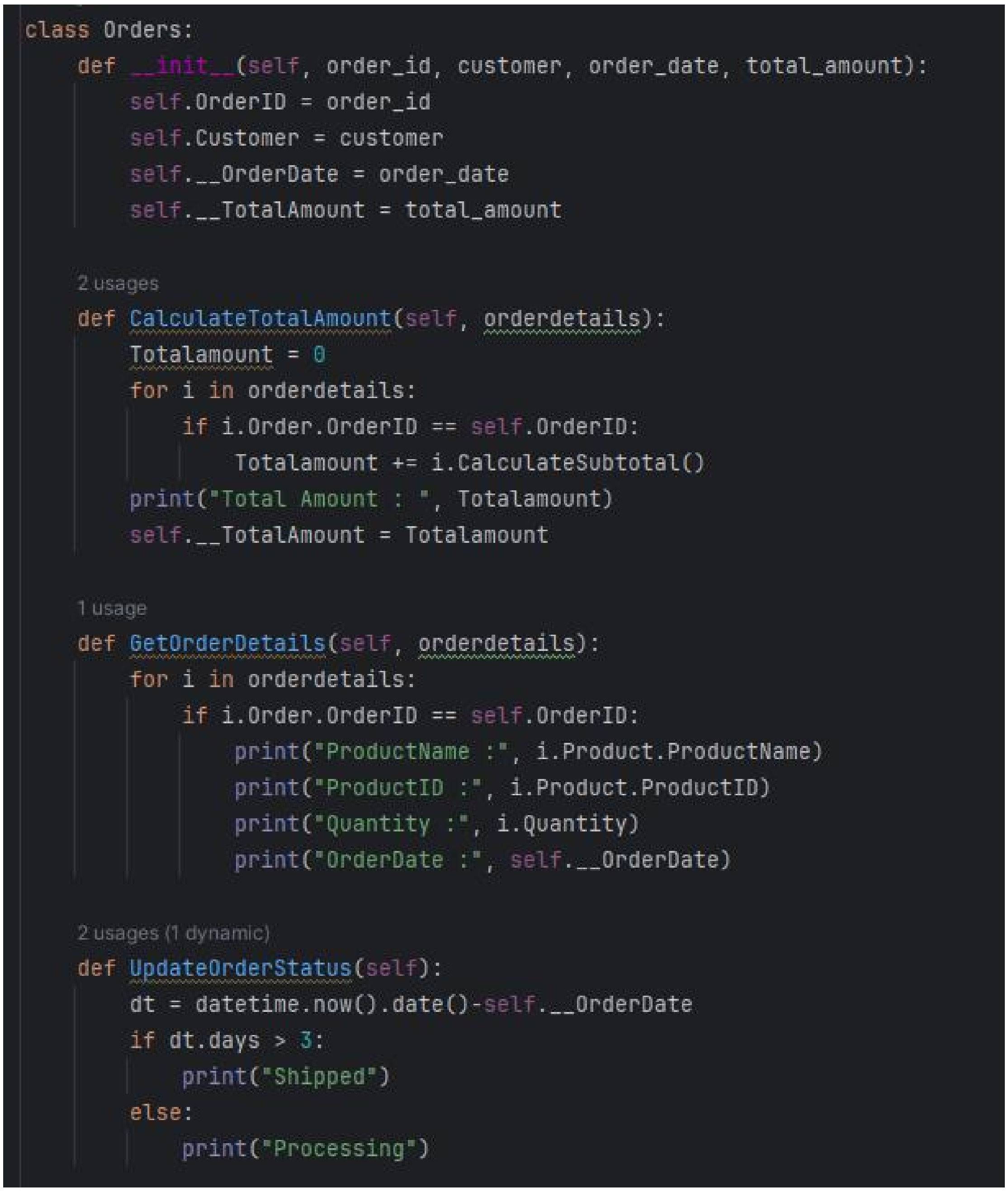
```python
class Products:
    def __init__(self, product_id, product_name, description, price):
        self.ProductID = product_id
        self.ProductName = product_name
        self.Description = description
        self.Price = price

    def GetProductDetails(self):
        print(" ProductID = " + str(self.ProductID))
        print(" ProductName = " + str(self.__ProductName))
        print(" Description = " + str(self.__Description))
        print(" Price = " + str(self.__Price))

    def UpdateProductInfo(self, description=None, price=None):
        if description:
            self.__Description = description
        if price:
            self.__Price = price

    1 usage
    def IsProductInStock(self, inventory):
        for i in inventory:
            if i.Product.ProductID == self.ProductID and i.QuantityInStock>0:
                print("Yes product stock is available")
```

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

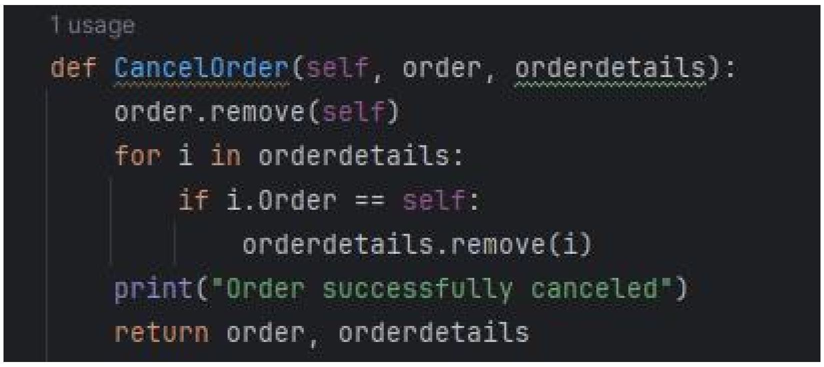
quantities).

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

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



```python
class Orders:
    def __init__(self, order_id, customer, order_date, total_amount):
        self.OrderID = order_id
        self.Customer = customer
        self.__OrderDate = order_date
        self.__TotalAmount = total_amount

    2 usages
    def CalculateTotalAmount(self, orderdetails):
        Totalamount = 0
        for i in orderdetails:
            if i.Order.OrderID == self.OrderID:
                Totalamount += i.CalculateSubtotal()
        print("Total Amount : ", Totalamount)
        self.__TotalAmount = Totalamount

    1 usage
    def GetOrderDetails(self, orderdetails):
        for i in orderdetails:
            if i.Order.OrderID == self.OrderID:
                print("ProductName :", i.Product.ProductName)
                print("ProductID :", i.Product.ProductID)
                print("Quantity :", i.Quantity)
                print("OrderDate :", self.__OrderDate)

    2 usages (1 dynamic)
    def UpdateOrderStatus(self):
        dt = datetime.now().date()-self.__OrderDate
        if dt.days > 3:
            print("Shipped")
        else:
            print("Processing")
```



```
1 usage
def CancelOrder(self, order, orderdetails):
    order.remove(self)
    for i in orderdetails:
        if i.Order == self:
            orderdetails.remove(i)
    print("Order successfully canceled")
    return order, orderdetails
```

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)

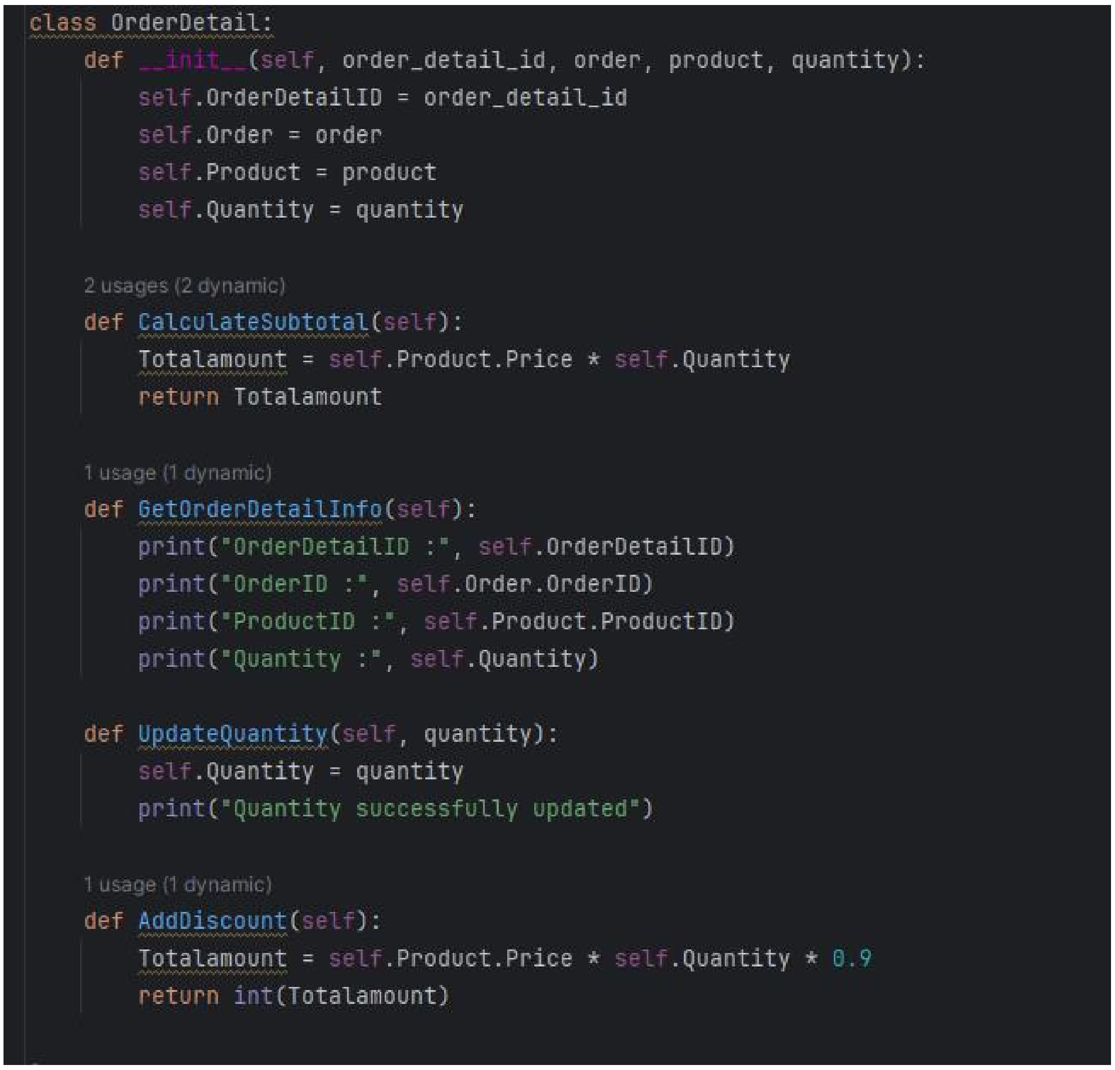
Methods:

• 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.



```
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity):
        self.OrderDetailID = order_detail_id
        self.Order = order
        self.Product = product
        self.Quantity = quantity

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

    1 usage (1 dynamic)
    def GetOrderDetailInfo(self):
        print("OrderDetailID :", self.OrderDetailID)
        print("OrderID :", self.Order.OrderID)
        print("ProductID :", self.Product.ProductID)
        print("Quantity :", self.Quantity)

    def UpdateQuantity(self, quantity):
        self.Quantity = quantity
        print("Quantity successfully updated")

    1 usage (1 dynamic)
    def AddDiscount(self):
        Totalamount = self.Product.Price * self.Quantity * 0.9
        return int(Totalamount)
```

Inventory class:

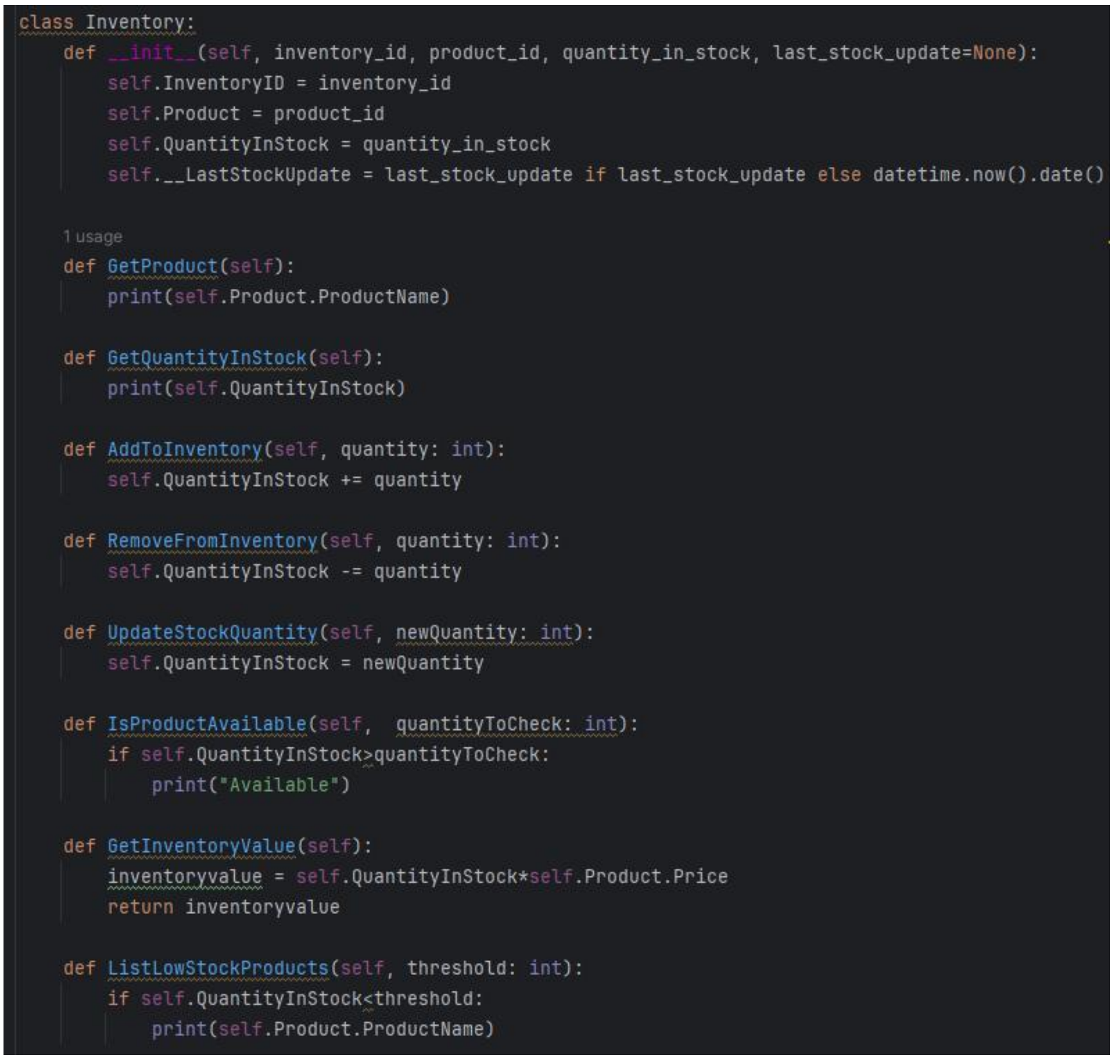
Attributes:

- InventoryID(int)
- Product (Composition): The product associated with the inventory item.
- QuantityInStock: The quantity of the product currently in stock.
- LastStockUpdate

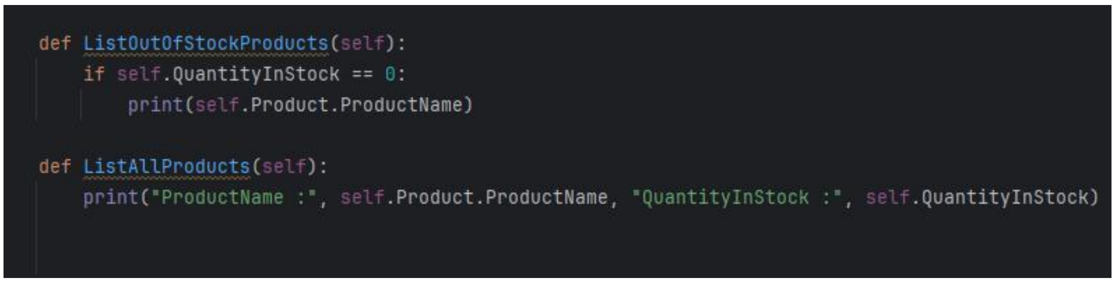
Methods:

- 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.
• ListAllProducts(): A method to list all products in the inventory, along with their quantities



```
class Inventory:
    def __init__(self, inventory_id, product_id, quantity_in_stock, last_stock_update=None):
        self.InventoryID = inventory_id
        self.Product = product_id
        self.QuantityInStock = quantity_in_stock
        self.__LastStockUpdate = last_stock_update if last_stock_update else datetime.now().date()

    1 usage
    def GetProduct(self):
        print(self.Product.ProductName)

    def GetQuantityInStock(self):
        print(self.QuantityInStock)

    def AddToInventory(self, quantity: int):
        self.QuantityInStock += quantity

    def RemoveFromInventory(self, quantity: int):
        self.QuantityInStock -= quantity

    def UpdateStockQuantity(self, newQuantity: int):
        self.QuantityInStock = newQuantity

    def IsProductAvailable(self,  quantityToCheck: int):
        if self.QuantityInStock>quantityToCheck:
            print("Available")

    def GetInventoryValue(self):
        inventoryvalue = self.QuantityInStock*self.Product.Price
        return inventoryvalue

    def ListLowStockProducts(self, threshold: int):
        if self.QuantityInStock<threshold:
            print(self.Product.ProductName)
```



```
def ListOutOfStockProducts(self):
    if self.QuantityInStock == 0:
        print(self.Product.ProductName)

def ListAllProducts(self):
    print("ProductName :", self.Product.ProductName, "QuantityInStock :", self.QuantityInStock)
```

Managing Products List:

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

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

o Solution: Implement methods to add, update, and remove products. Handle exceptions

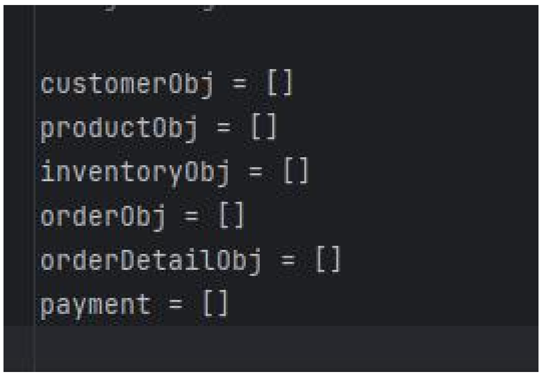
for duplicate products, invalid updates, or removal of products with existing orders.

• **Managing Orders List:**

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

o Scenario: Adding new orders, updating order statuses, and removing canceled 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

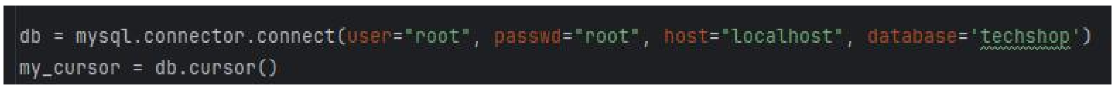


```
customerObj = []
productObj = []
inventoryObj = []
orderObj = []
orderDetailObj = []
payment = []
```

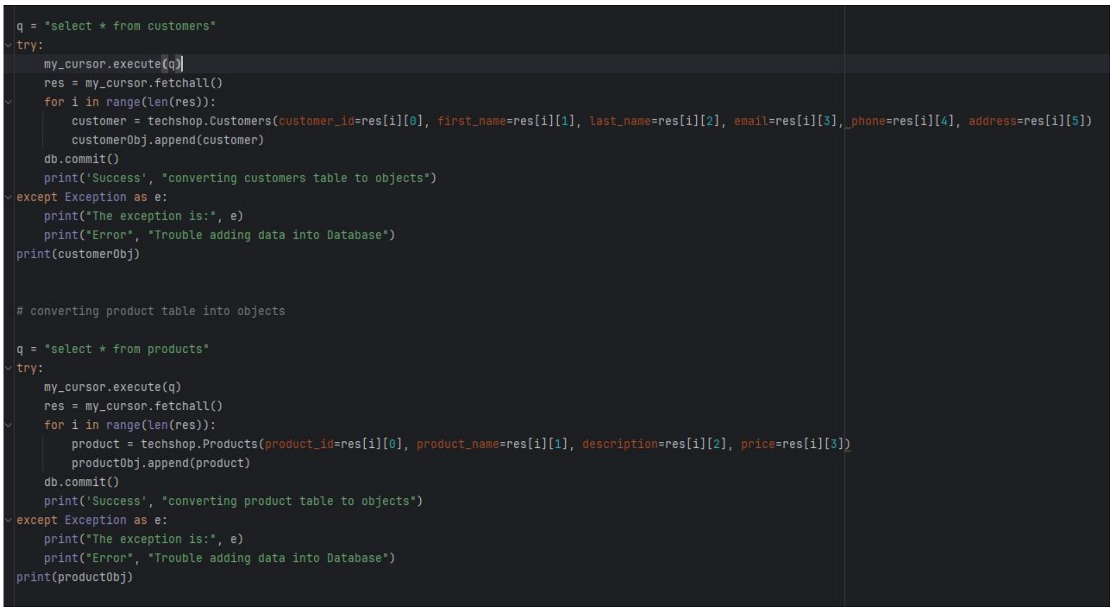
Task 7: Database Connectivity

• 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.

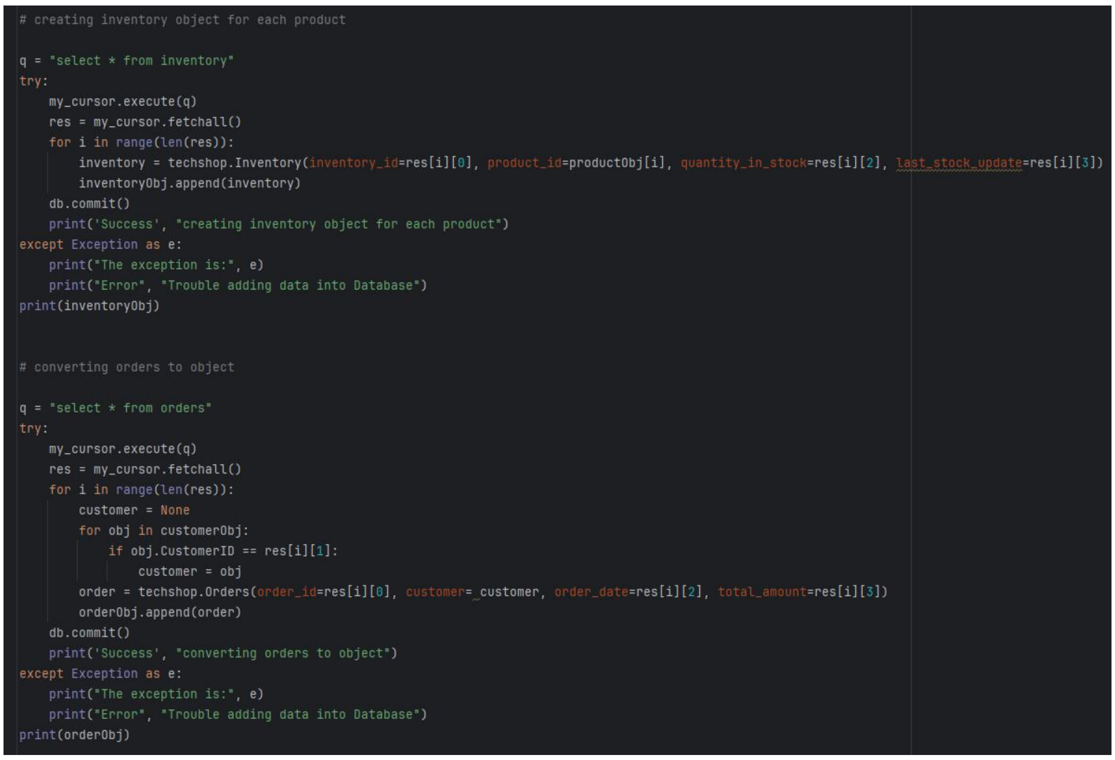
• Implement classes for Customers, Products, Orders, OrderDetails, Inventory with properties,
constructors, and methods for CRUD (Create, Read, Update, Delete) operations.



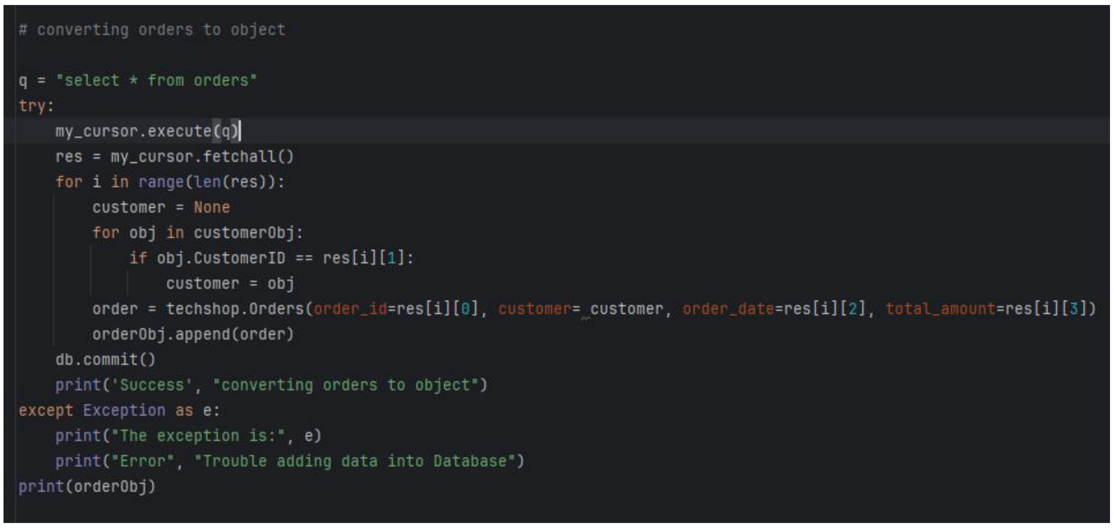
```
db = mysql.connector.connect(user="root", passwd="root", host="localhost", database='techshop')
my_cursor = db.cursor()
```

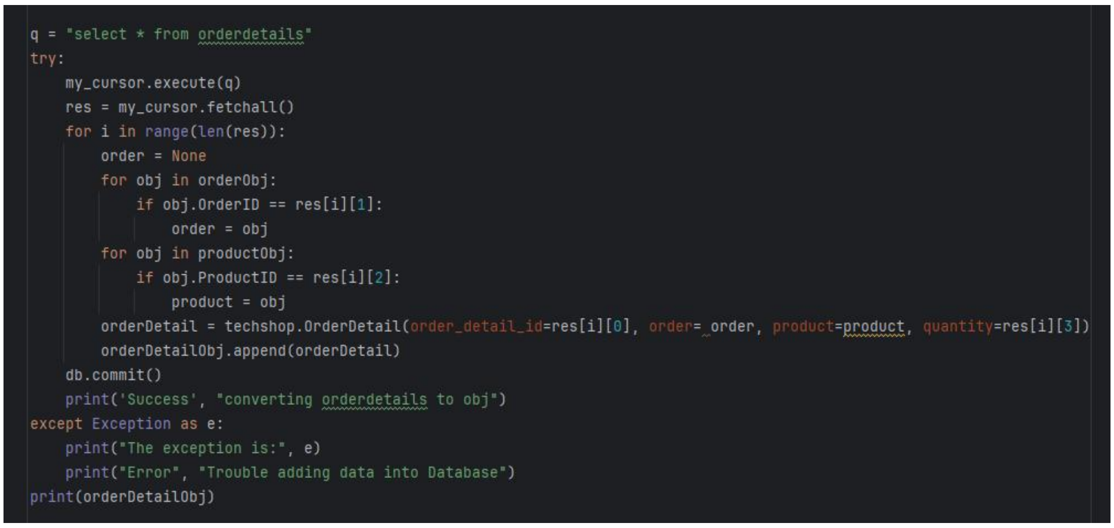


```python
q = "select * from customers"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        customer = techshop.Customers(customer_id=res[i][0], first_name=res[i][1], last_name=res[i][2], email=res[i][3], phone=res[i][4], address=res[i][5])
        customerObj.append(customer)
    db.commit()
    print('Success', "converting customers table to objects")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(customerObj)


# converting product table into objects

q = "select * from products"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        product = techshop.Products(product_id=res[i][0], product_name=res[i][1], description=res[i][2], price=res[i][3])
        productObj.append(product)
    db.commit()
    print('Success', "converting product table to objects")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(productObj)
```



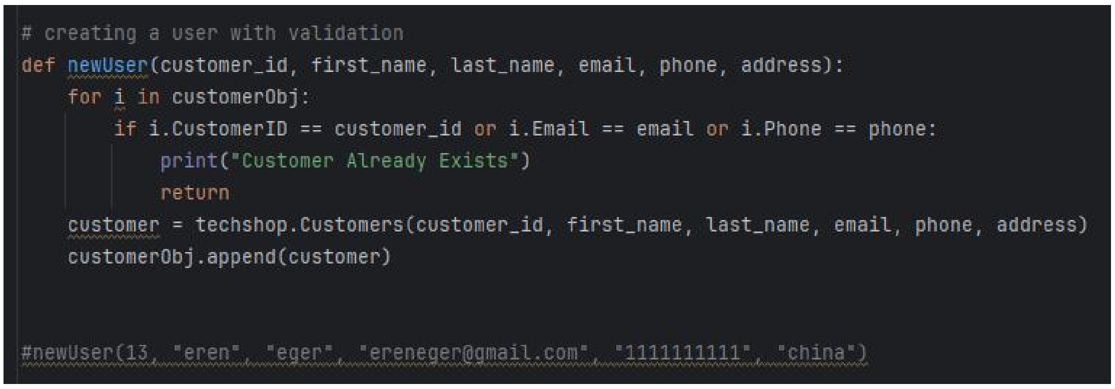
```python
# creating inventory object for each product

q = "select * from inventory"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        inventory = techshop.Inventory(inventory_id=res[i][0], product_id=productObj[i], quantity_in_stock=res[i][2], last_stock_update=res[i][3])
        inventoryObj.append(inventory)
    db.commit()
    print('Success', "creating inventory object for each product")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(inventoryObj)


# converting orders to object

q = "select * from orders"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        customer = None
        for obj in customerObj:
            if obj.CustomerID == res[i][1]:
                customer = obj
        order = techshop.Orders(order_id=res[i][0], customer= customer, order_date=res[i][2], total_amount=res[i][3])
        orderObj.append(order)
    db.commit()
    print('Success', "converting orders to object")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(orderObj)
```


```
# converting orders to object

q = "select * from orders"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        customer = None
        for obj in customerObj:
            if obj.CustomerID == res[i][1]:
                customer = obj
        order = techshop.Orders(order_id=res[i][0], customer= customer, order_date=res[i][2], total_amount=res[i][3])
        orderObj.append(order)
    db.commit()
    print('Success', "converting orders to object")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(orderObj)
```


```
q = "select * from orderdetails"
try:
    my_cursor.execute(q)
    res = my_cursor.fetchall()
    for i in range(len(res)):
        order = None
        for obj in orderObj:
            if obj.OrderID == res[i][1]:
                order = obj
        for obj in productObj:
            if obj.ProductID == res[i][2]:
                product = obj
        orderDetail = techshop.OrderDetail(order_detail_id=res[i][0], order= order, product=product, quantity=res[i][3])
        orderDetailObj.append(orderDetail)
    db.commit()
    print('Success', "converting orderdetails to obj")
except Exception as e:
    print("The exception is:", e)
    print("Error", "Trouble adding data into Database")
print(orderDetailObj)
```

1: Customer Registration

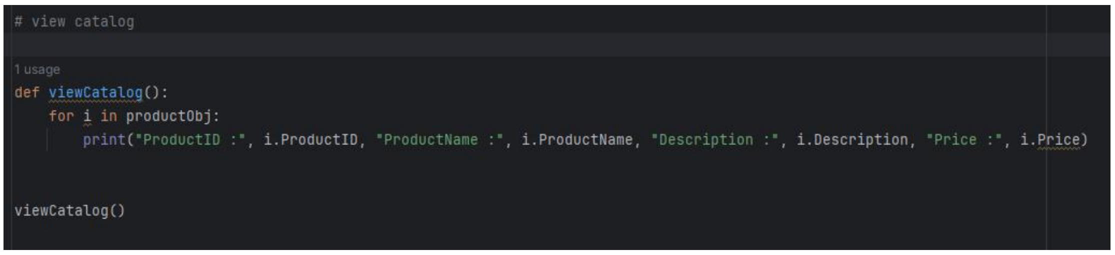
Description: When a new customer registers on the TechShop website, their information (e.g., name,
email, phone) needs to be stored in the database.
Task: Implement a registration form and database connectivity to insert new customer records. Ensure
proper data validation and error handling for duplicate email addresses.



```python
# creating a user with validation
def newUser(customer_id, first_name, last_name, email, phone, address):
    for i in customerObj:
        if i.CustomerID == customer_id or i.Email == email or i.Phone == phone:
            print("Customer Already Exists")
            return
    customer = techshop.Customers(customer_id, first_name, last_name, email, phone, address)
    customerObj.append(customer)


#newUser(13, "eren", "eger", "ereneger@gmail.com", "1111111111", "china")
```

2: Product Catalog Management

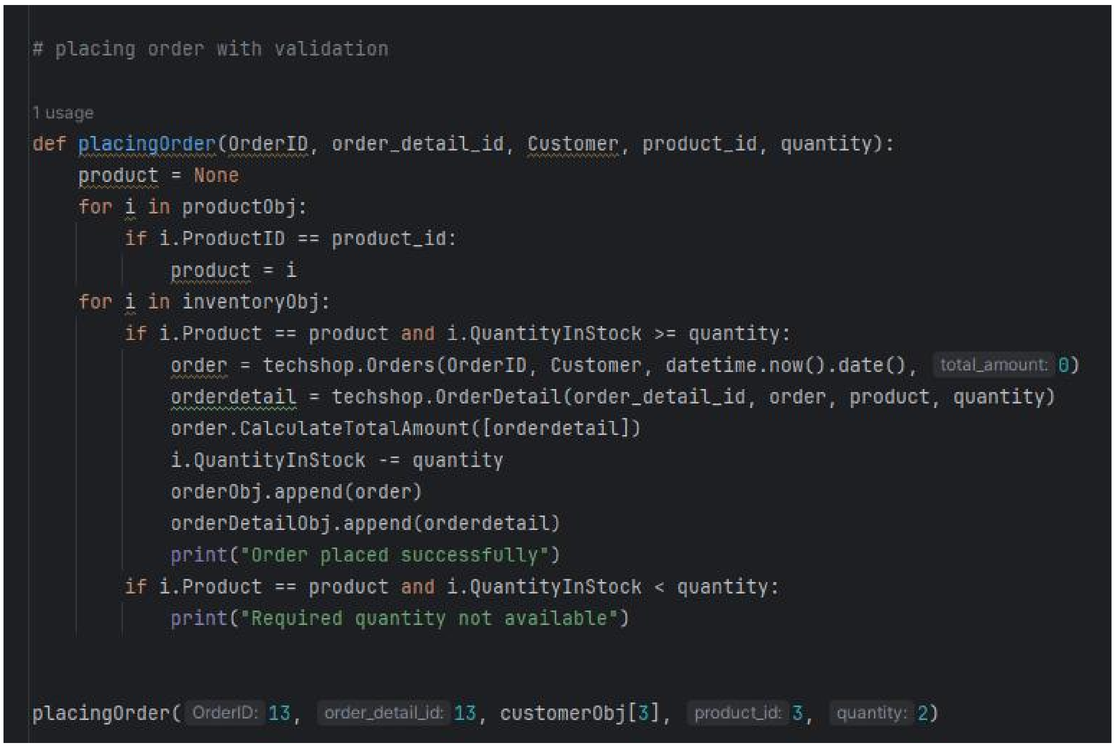
Description: TechShop regularly updates its product catalog with new items and changes in product
details (e.g., price, description). These changes need to be reflected in the database.
Task: Create an interface to manage the product catalog. Implement database connectivity to update
product information. Handle changes in product details and ensure data consistency.



```python
# view catalog

1 usage
def viewCatalog():
    for i in productObj:
        print("ProductID :", i.ProductID, "ProductName :", i.ProductName, "Description :", i.Description, "Price :", i.Price)


viewCatalog()
```

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,
update product quantities in inventory, and calculate order totals.



```
# placing order with validation

1 usage
def placingOrder(OrderID, order_detail_id, Customer, product_id, quantity):
    product = None
    for i in productObj:
        if i.ProductID == product_id:
            product = i
    for i in inventoryObj:
        if i.Product == product and i.QuantityInStock >= quantity:
            order = techshop.Orders(OrderID, Customer, datetime.now().date(),  total_amount: 0)
            orderdetail = techshop.OrderDetail(order_detail_id, order, product, quantity)
            order.CalculateTotalAmount([orderdetail])
            i.QuantityInStock -= quantity
            orderObj.append(order)
            orderDetailObj.append(orderdetail)
            print("Order placed successfully")
        if i.Product == product and i.QuantityInStock < quantity:
            print("Required quantity not available")


placingOrder( OrderID: 13,  order_detail_id: 13, customerObj[3],  product_id: 3,  quantity: 2)
```

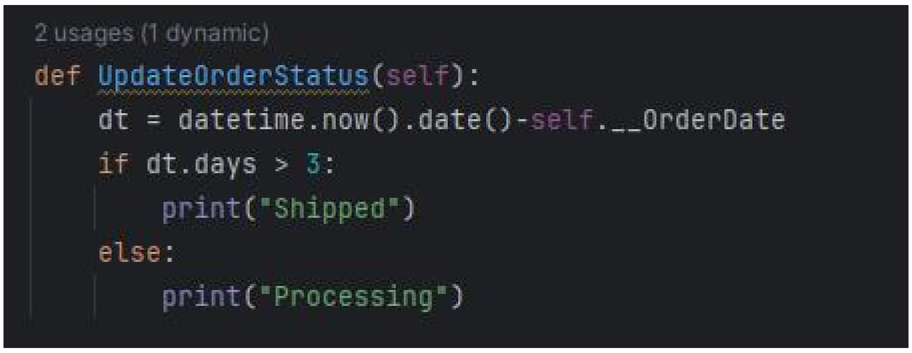
4: Tracking Order Status

Description: Customers and employees need to track the status of their orders. The order status

information is stored in the database.

Task: Develop a feature that allows users to view the status of their orders. Implement database

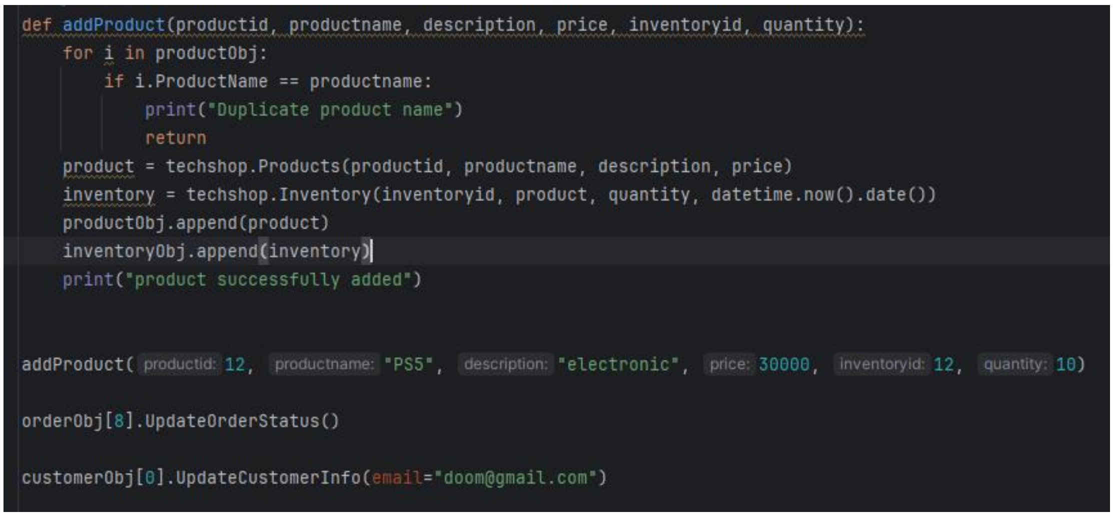
connectivity to retrieve and display order status information.



```
2 usages (1 dynamic)
def UpdateOrderStatus(self):
    dt = datetime.now().date()-self.__OrderDate
    if dt.days > 3:
        print("Shipped")
    else:
        print("Processing")
```

5: Inventory Management

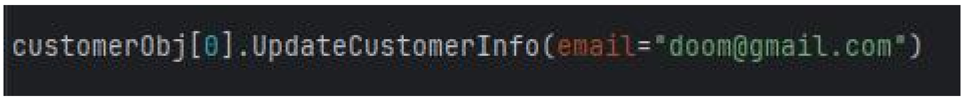
Description: TechShop needs to manage product inventory, including adding new products, updating
stock levels, and removing discontinued items.
Task: Create an inventory management system with database connectivity. Implement features for
adding new products, updating quantities, and handling discontinued products.



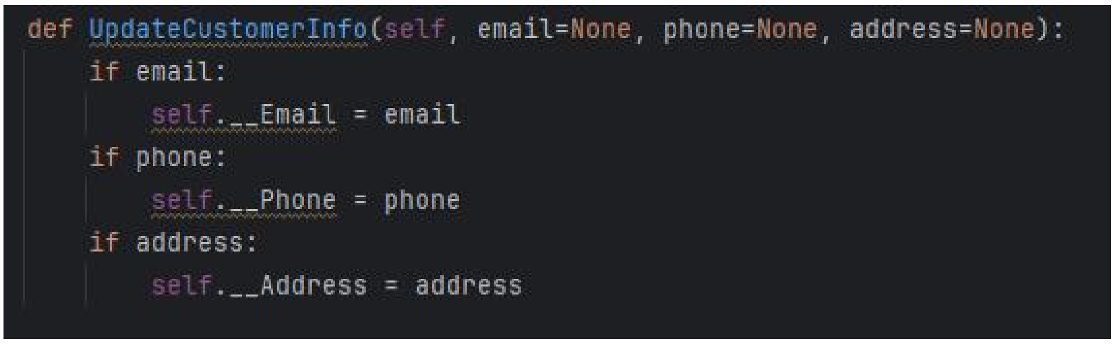
```
def addProduct(productid, productname, description, price, inventoryid, quantity):
    for i in productObj:
        if i.ProductName == productname:
            print("Duplicate product name")
            return
    product = techshop.Products(productid, productname, description, price)
    inventory = techshop.Inventory(inventoryid, product, quantity, datetime.now().date())
    productObj.append(product)
    inventoryObj.append(inventory)
    print("product successfully added")


addProduct( productid: 12,  productname: "PS5",  description: "electronic",  price: 30000,  inventoryid: 12,  quantity: 10)

orderObj[8].UpdateOrderStatus()

customerObj[0].UpdateCustomerInfo(email="doom@gmail.com")
```

7: Customer Account Updates

Description: Customers may need to update their account information, such as changing their email
address or phone number.
Task: Implement a user profile management feature with database connectivity to allow customers to
update their account details. Ensure data validation and integrity.



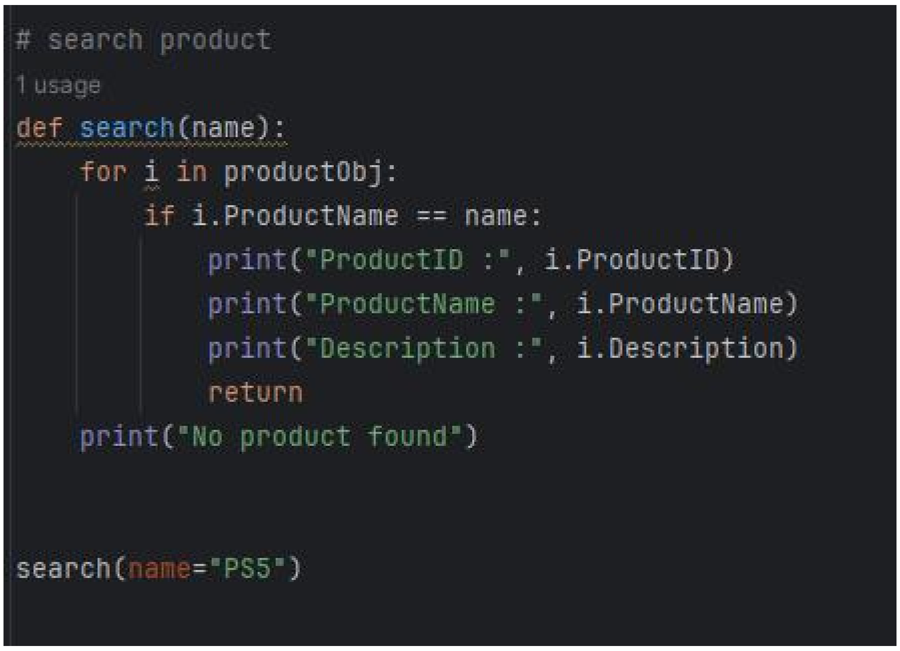
```
customerObj[0].UpdateCustomerInfo(email="doom@gmail.com")
```



```python
def UpdateCustomerInfo(self, email=None, phone=None, address=None):
    if email:
        self.__Email = email
    if phone:
        self.__Phone = phone
    if address:
        self.__Address = address
```

9: Product Search and Recommendations

Description: Customers should be able to search for products based on various criteria (e.g., name,
category) and receive product recommendations.
Task: Implement a product search and recommendation engine that uses database connectivity to
retrieve relevant product information.



```python
# search product
1 usage
def search(name):
    for i in productObj:
        if i.ProductName == name:
            print("ProductID :", i.ProductID)
            print("ProductName :", i.ProductName)
            print("Description :", i.Description)
            return
    print("No product found")


search(name="PS5")
```