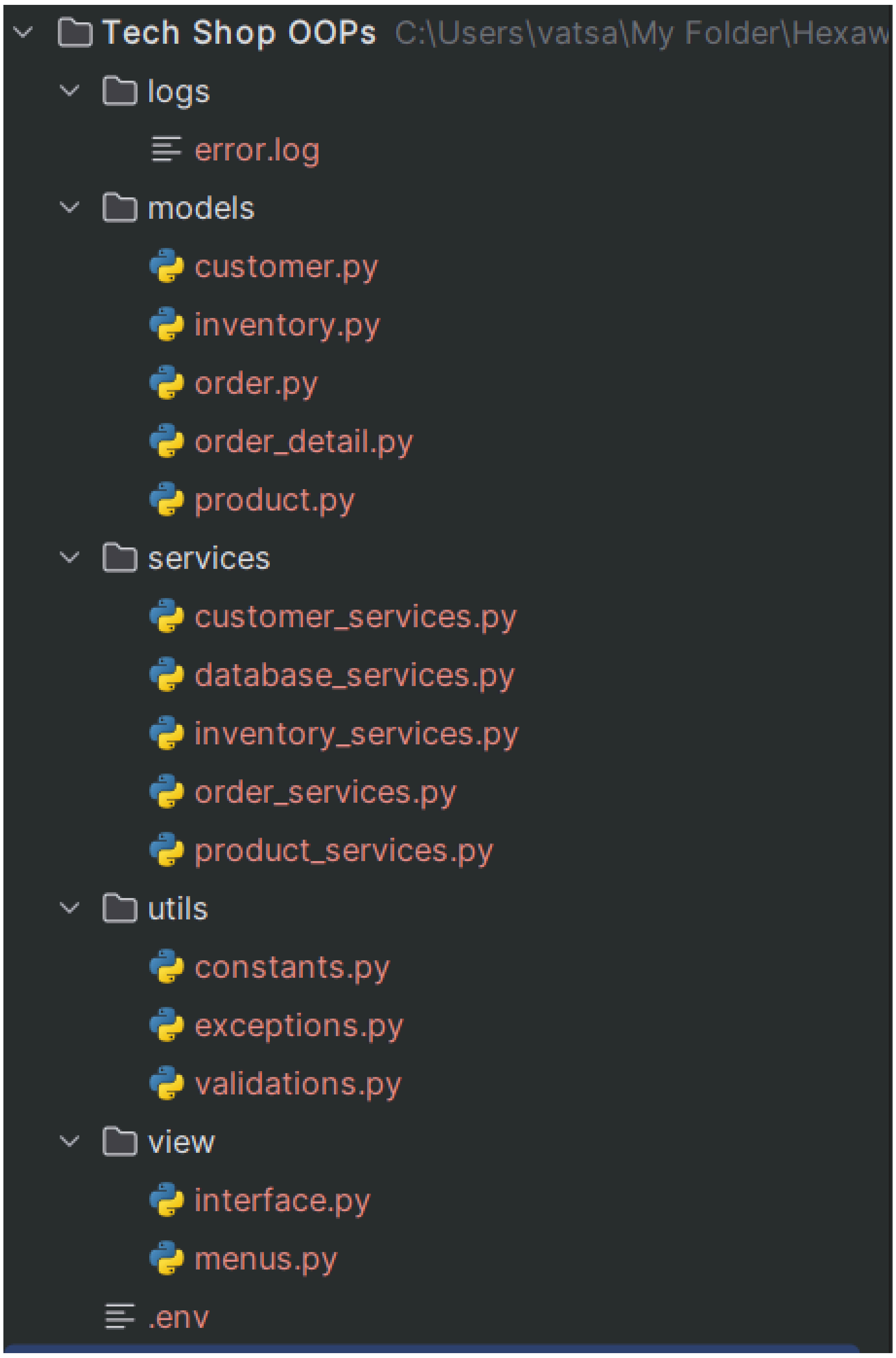
Implement OOPs

* Note - I would not be able to show all implementations in screenshots, please open the project to judge completely

Folder structure -

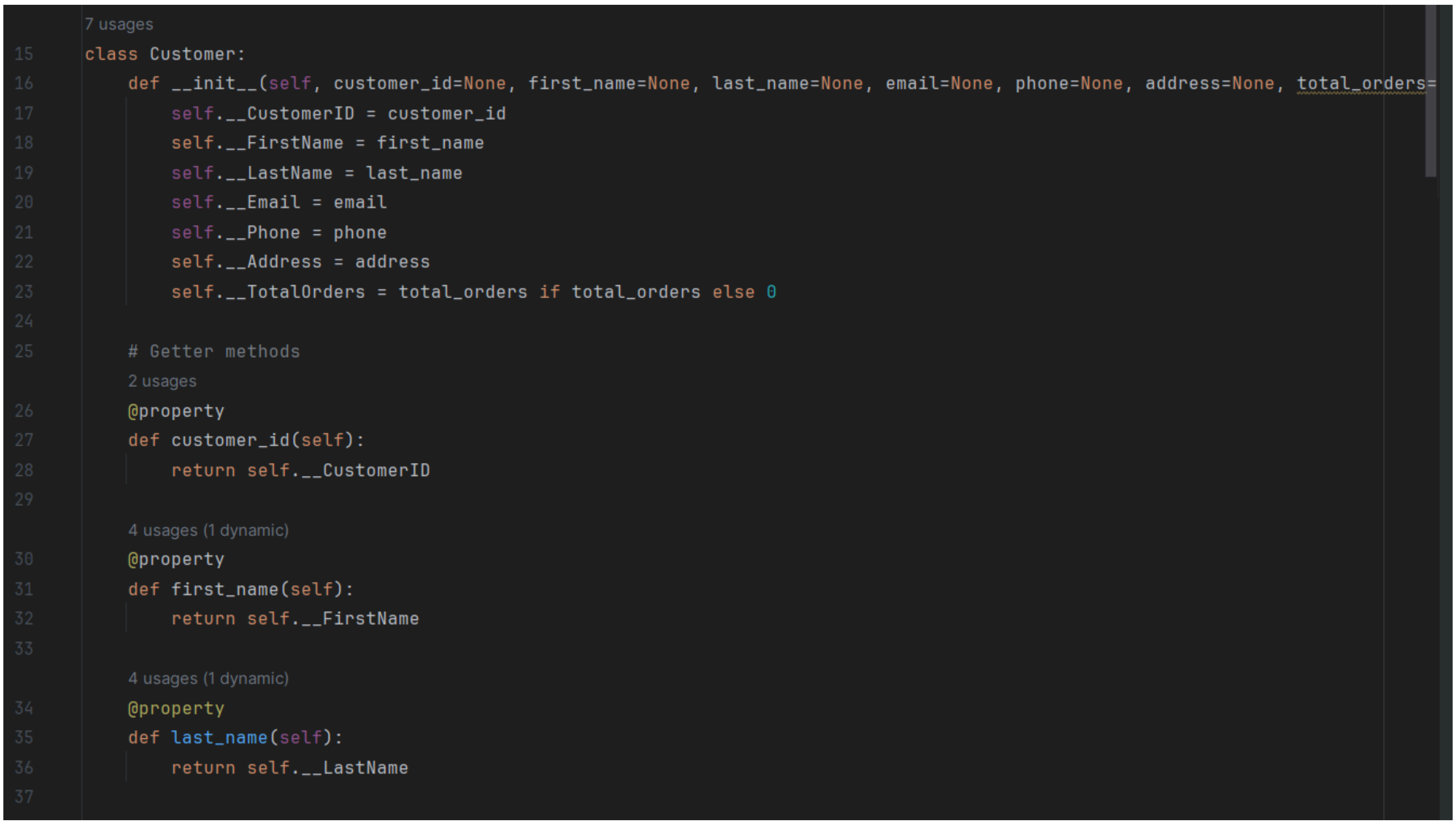


```
Tech Shop OOPs  C:\Users\vatsa\My Folder\Hexaw
    logs
        error.log
    models
        customer.py
        inventory.py
        order.py
        order_detail.py
        product.py
    services
        customer_services.py
        database_services.py
        inventory_services.py
        order_services.py
        product_services.py
    utils
        constants.py
        exceptions.py
        validations.py
    view
        interface.py
        menus.py
    .env
```

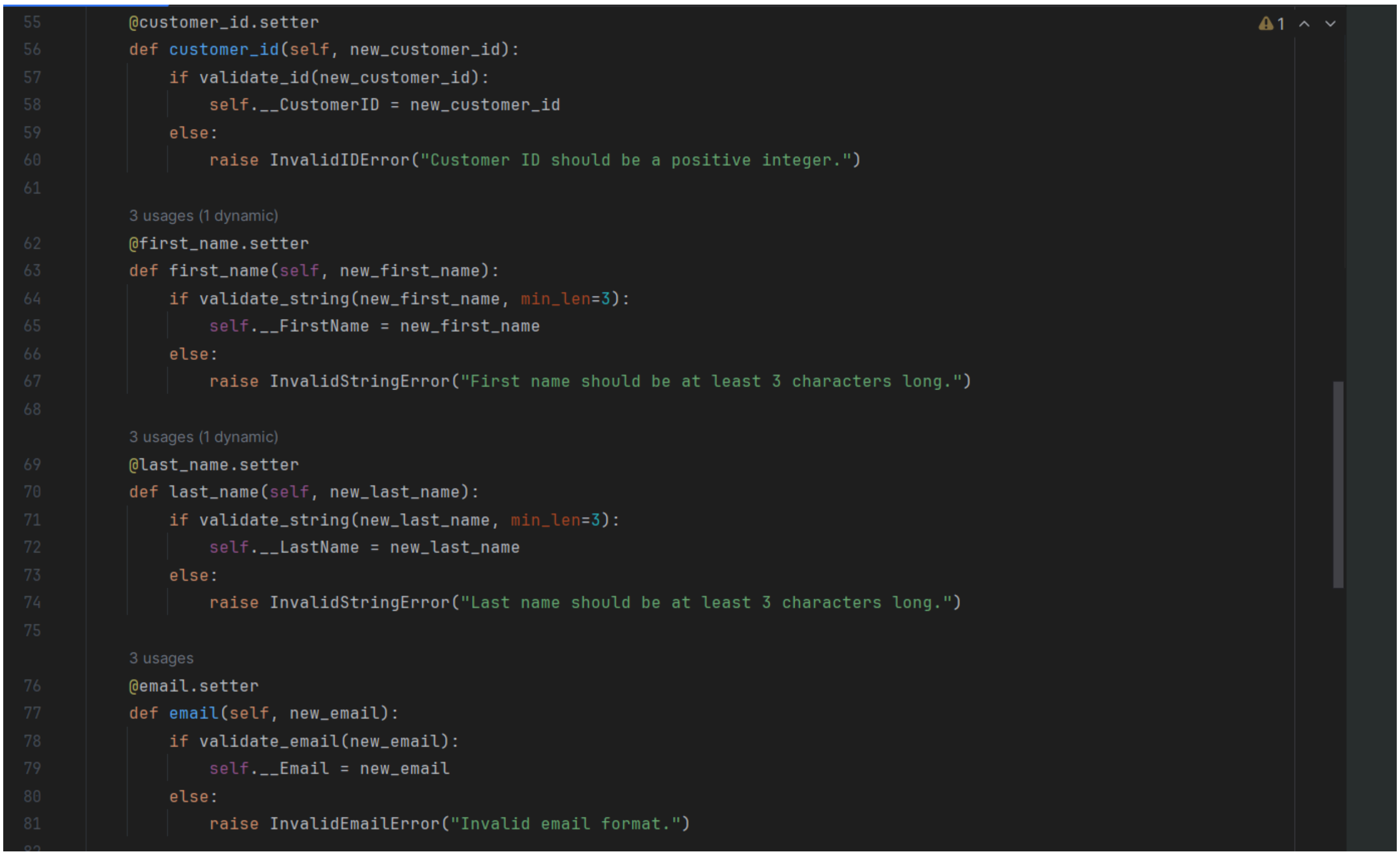
Task 1: Classes and Their Attributes:
Task 2: Class Creation:
Task 3: Encapsulation:
Task 4: Composition:
Task 5: Exceptions handling

Implemented all Attributes and Methods in given classes
Created Getter and Setter functions to respect encapsulation
Composition of Order with Customer and Order details with Products was done
Implemented Validations in Setter functions and Class Methods and exception handling with custom built exceptions

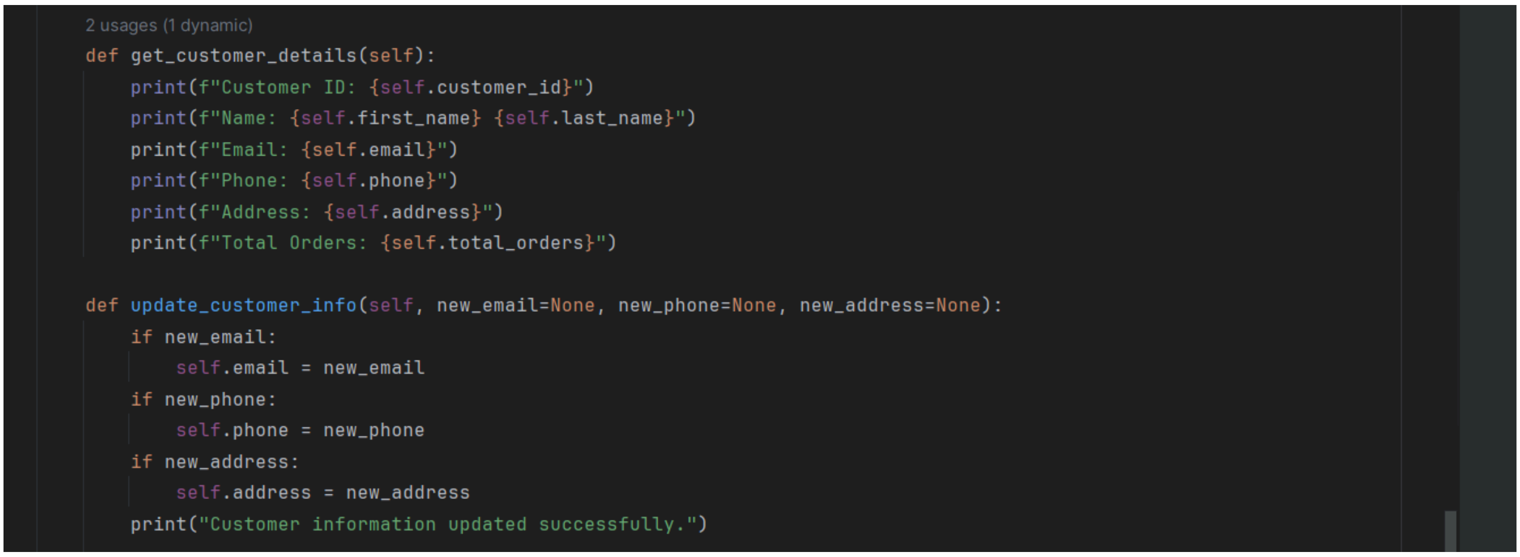
Customer Class



```
7 usages
class Customer:
    def __init__(self, customer_id=None, first_name=None, last_name=None, email=None, phone=None, address=None, total_orders=
        self.__CustomerID = customer_id
        self.__FirstName = first_name
        self.__LastName = last_name
        self.__Email = email
        self.__Phone = phone
        self.__Address = address
        self.__TotalOrders = total_orders if total_orders else 0

    # Getter methods
    2 usages
    @property
    def customer_id(self):
        return self.__CustomerID

    4 usages (1 dynamic)
    @property
    def first_name(self):
        return self.__FirstName

    4 usages (1 dynamic)
    @property
    def last_name(self):
        return self.__LastName

```

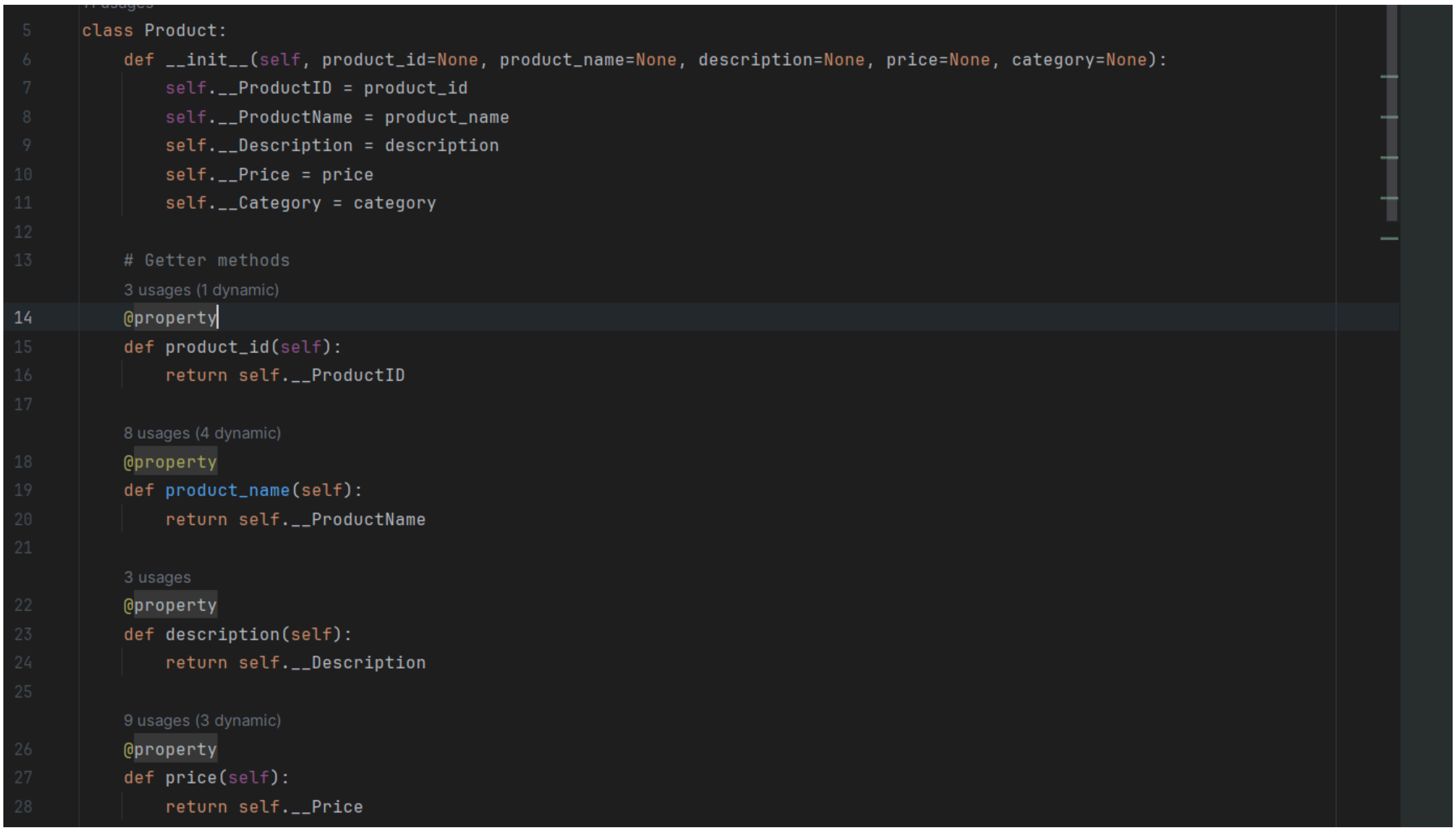


```
    @customer_id.setter
    def customer_id(self, new_customer_id):
        if validate_id(new_customer_id):
            self.__CustomerID = new_customer_id
        else:
            raise InvalidIDError("Customer ID should be a positive integer.")

    3 usages (1 dynamic)
    @first_name.setter
    def first_name(self, new_first_name):
        if validate_string(new_first_name, min_len=3):
            self.__FirstName = new_first_name
        else:
            raise InvalidStringError("First name should be at least 3 characters long.")

    3 usages (1 dynamic)
    @last_name.setter
    def last_name(self, new_last_name):
        if validate_string(new_last_name, min_len=3):
            self.__LastName = new_last_name
        else:
            raise InvalidStringError("Last name should be at least 3 characters long.")

    3 usages
    @email.setter
    def email(self, new_email):
        if validate_email(new_email):
            self.__Email = new_email
        else:
            raise InvalidEmailError("Invalid email format.")
```

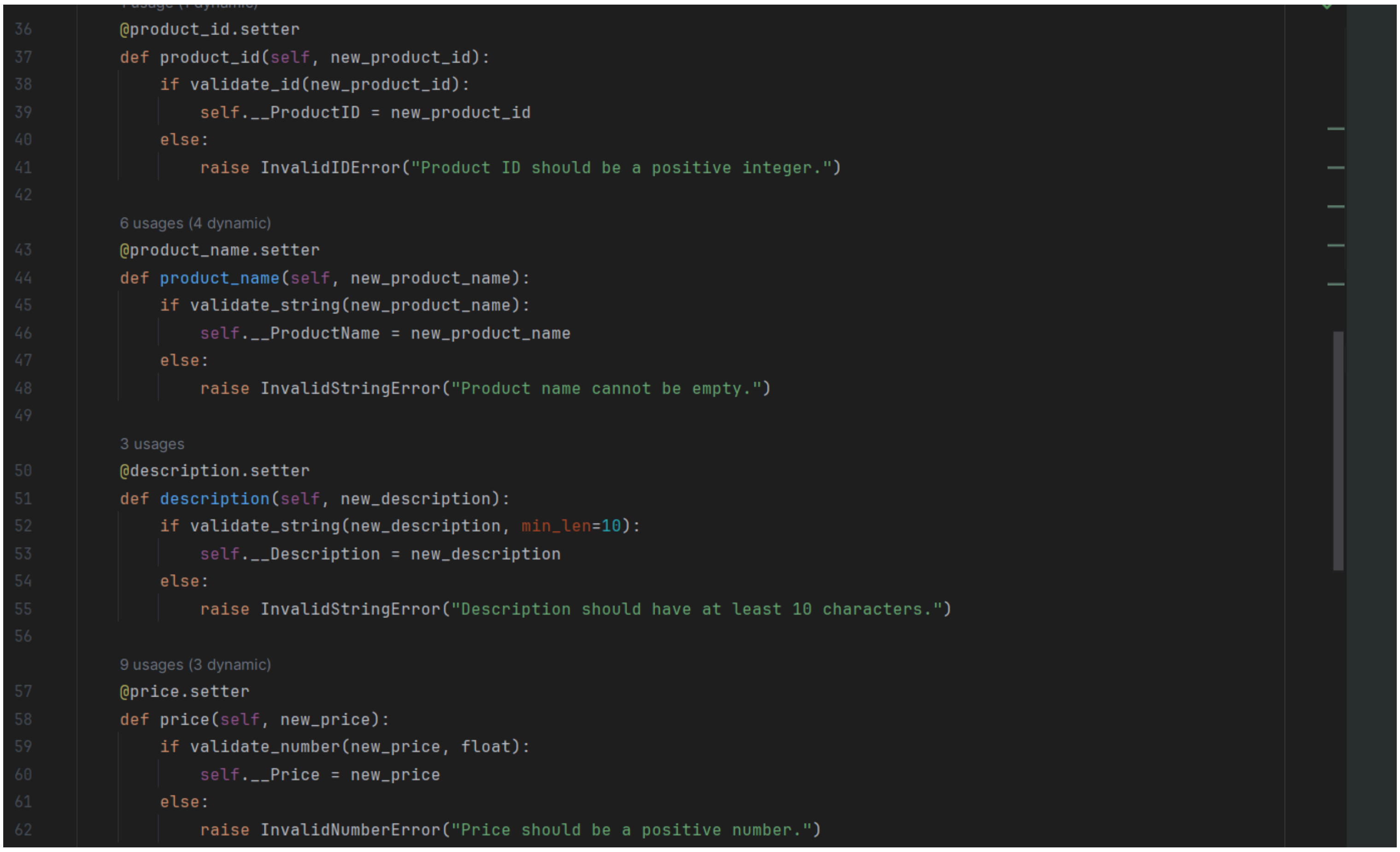


```
2 usages (1 dynamic)
def get_customer_details(self):
    print(f"Customer ID: {self.customer_id}")
    print(f"Name: {self.first_name} {self.last_name}")
    print(f"Email: {self.email}")
    print(f"Phone: {self.phone}")
    print(f"Address: {self.address}")
    print(f"Total Orders: {self.total_orders}")

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

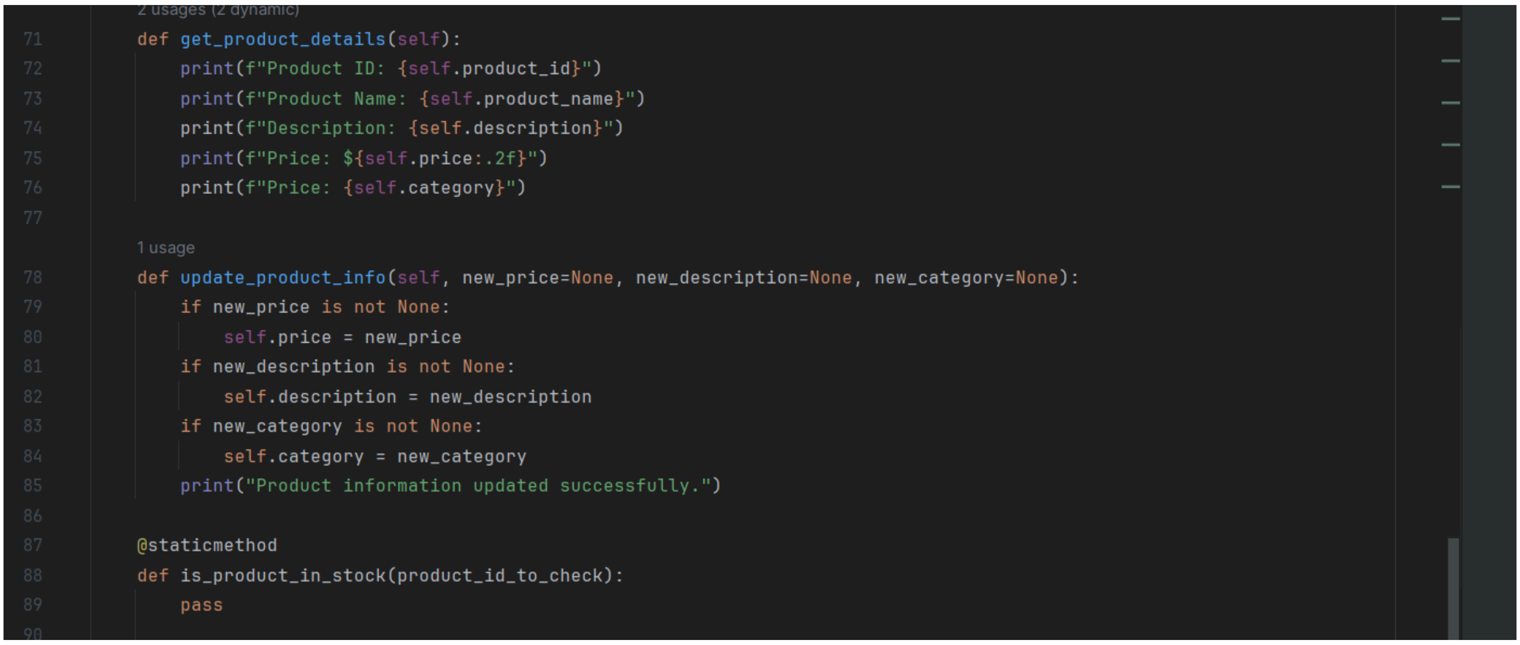
Product Class



```
class Product:
    def __init__(self, product_id=None, product_name=None, description=None, price=None, category=None):
        self.__ProductID = product_id
        self.__ProductName = product_name
        self.__Description = description
        self.__Price = price
        self.__Category = category

    # Getter methods
    3 usages (1 dynamic)
    @property
    def product_id(self):
        return self.__ProductID

    8 usages (4 dynamic)
    @property
    def product_name(self):
        return self.__ProductName

    3 usages
    @property
    def description(self):
        return self.__Description

    9 usages (3 dynamic)
    @property
    def price(self):
        return self.__Price
```

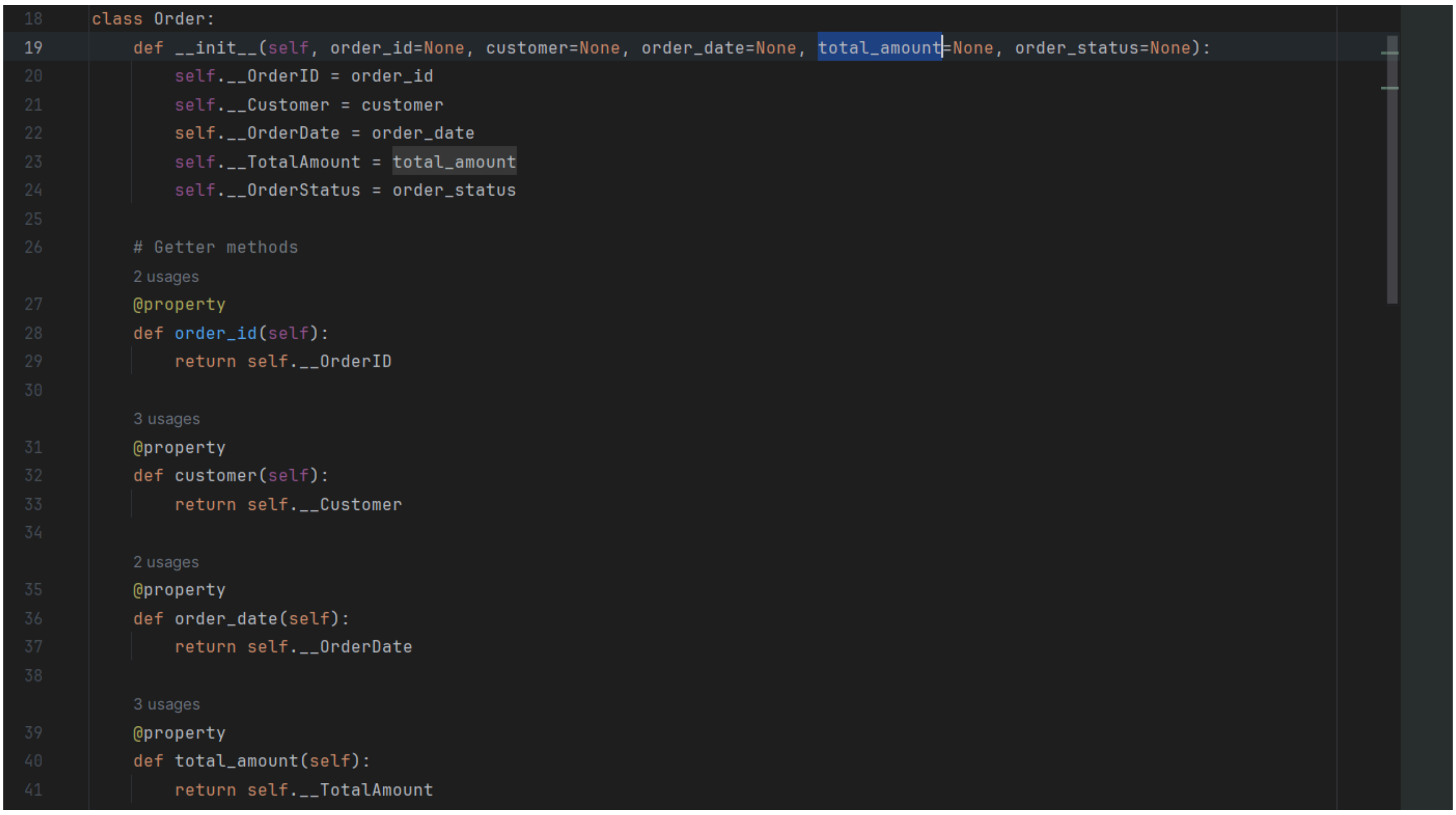


```python
    @product_id.setter
    def product_id(self, new_product_id):
        if validate_id(new_product_id):
            self.__ProductID = new_product_id
        else:
            raise InvalidIDError("Product ID should be a positive integer.")

    6 usages (4 dynamic)
    @product_name.setter
    def product_name(self, new_product_name):
        if validate_string(new_product_name):
            self.__ProductName = new_product_name
        else:
            raise InvalidStringError("Product name cannot be empty.")

    3 usages
    @description.setter
    def description(self, new_description):
        if validate_string(new_description, min_len=10):
            self.__Description = new_description
        else:
            raise InvalidStringError("Description should have at least 10 characters.")

    9 usages (3 dynamic)
    @price.setter
    def price(self, new_price):
        if validate_number(new_price, float):
            self.__Price = new_price
        else:
            raise InvalidNumberError("Price should be a positive number.")
```

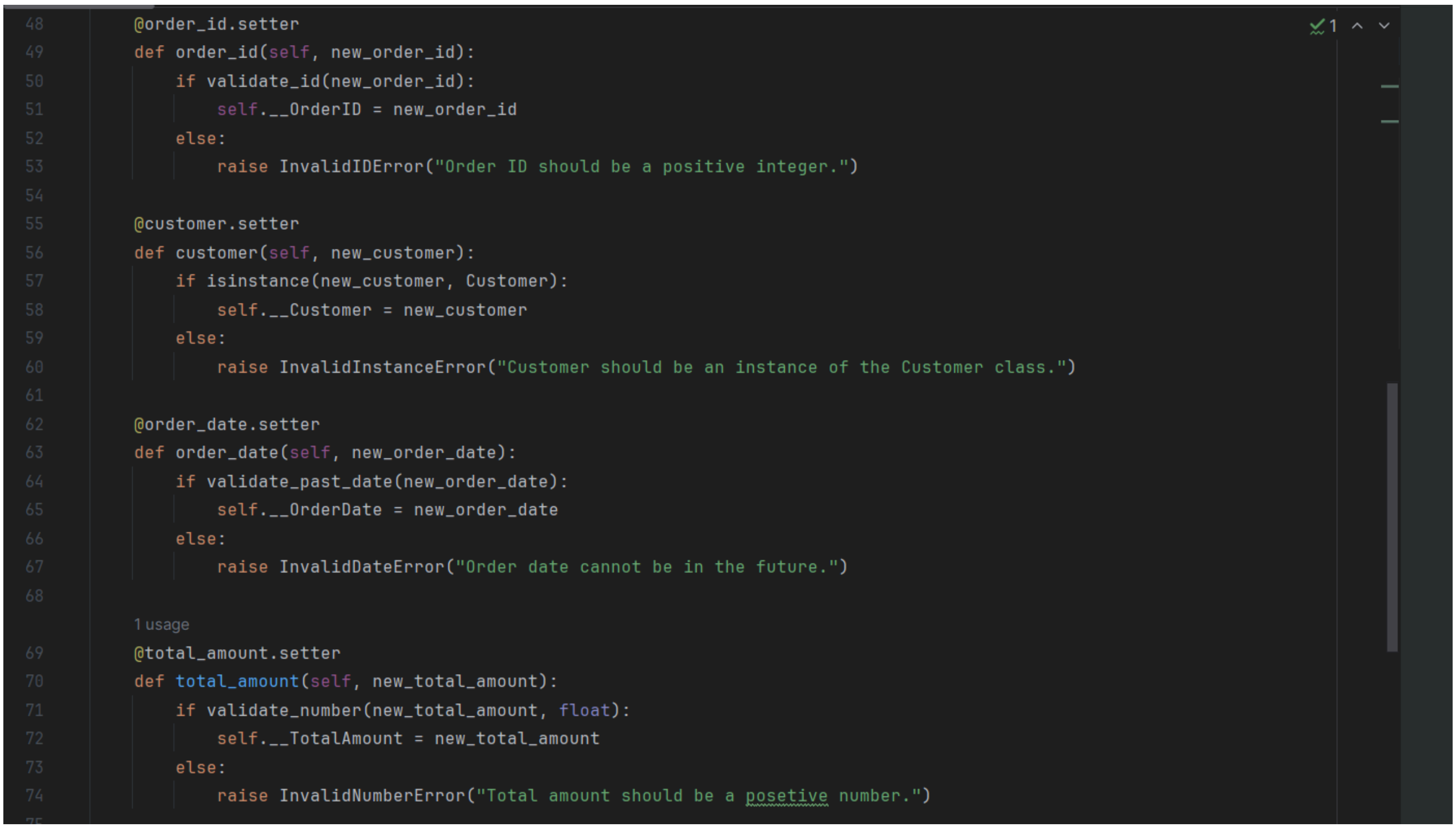


```python
    2 usages (2 dynamic)
    def get_product_details(self):
        print(f"Product ID: {self.product_id}")
        print(f"Product Name: {self.product_name}")
        print(f"Description: {self.description}")
        print(f"Price: ${self.price:.2f}")
        print(f"Price: {self.category}")

    1 usage
    def update_product_info(self, new_price=None, new_description=None, new_category=None):
        if new_price is not None:
            self.price = new_price
        if new_description is not None:
            self.description = new_description
        if new_category is not None:
            self.category = new_category
        print("Product information updated successfully.")

    @staticmethod
    def is_product_in_stock(product_id_to_check):
        pass
```

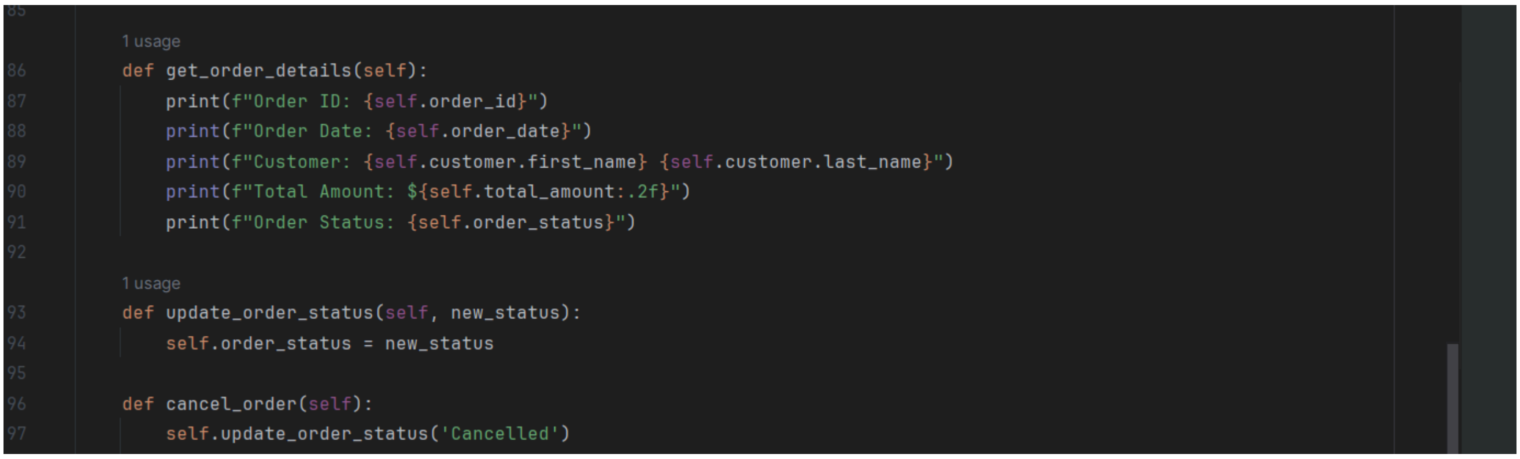
Order Class



```python
class Order:
    def __init__(self, order_id=None, customer=None, order_date=None, total_amount=None, order_status=None):
        self.__OrderID = order_id
        self.__Customer = customer
        self.__OrderDate = order_date
        self.__TotalAmount = total_amount
        self.__OrderStatus = order_status

    # Getter methods
    2 usages
    @property
    def order_id(self):
        return self.__OrderID

    3 usages
    @property
    def customer(self):
        return self.__Customer

    2 usages
    @property
    def order_date(self):
        return self.__OrderDate

    3 usages
    @property
    def total_amount(self):
        return self.__TotalAmount
```

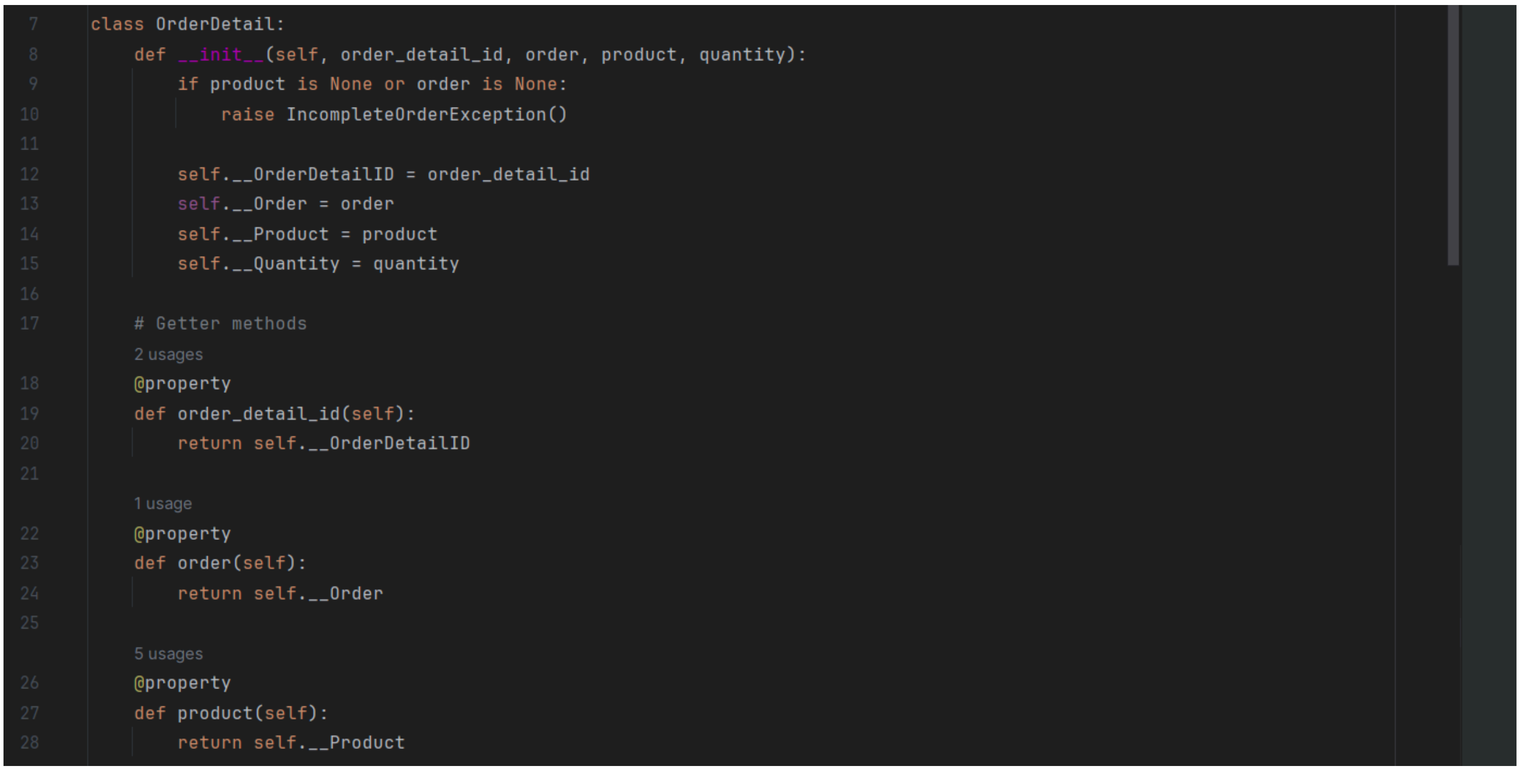


```python
    @order_id.setter
    def order_id(self, new_order_id):
        if validate_id(new_order_id):
            self.__OrderID = new_order_id
        else:
            raise InvalidIDError("Order ID should be a positive integer.")

    @customer.setter
    def customer(self, new_customer):
        if isinstance(new_customer, Customer):
            self.__Customer = new_customer
        else:
            raise InvalidInstanceError("Customer should be an instance of the Customer class.")

    @order_date.setter
    def order_date(self, new_order_date):
        if validate_past_date(new_order_date):
            self.__OrderDate = new_order_date
        else:
            raise InvalidDateError("Order date cannot be in the future.")

    1 usage
    @total_amount.setter
    def total_amount(self, new_total_amount):
        if validate_number(new_total_amount, float):
            self.__TotalAmount = new_total_amount
        else:
            raise InvalidNumberError("Total amount should be a posetive number.")
```

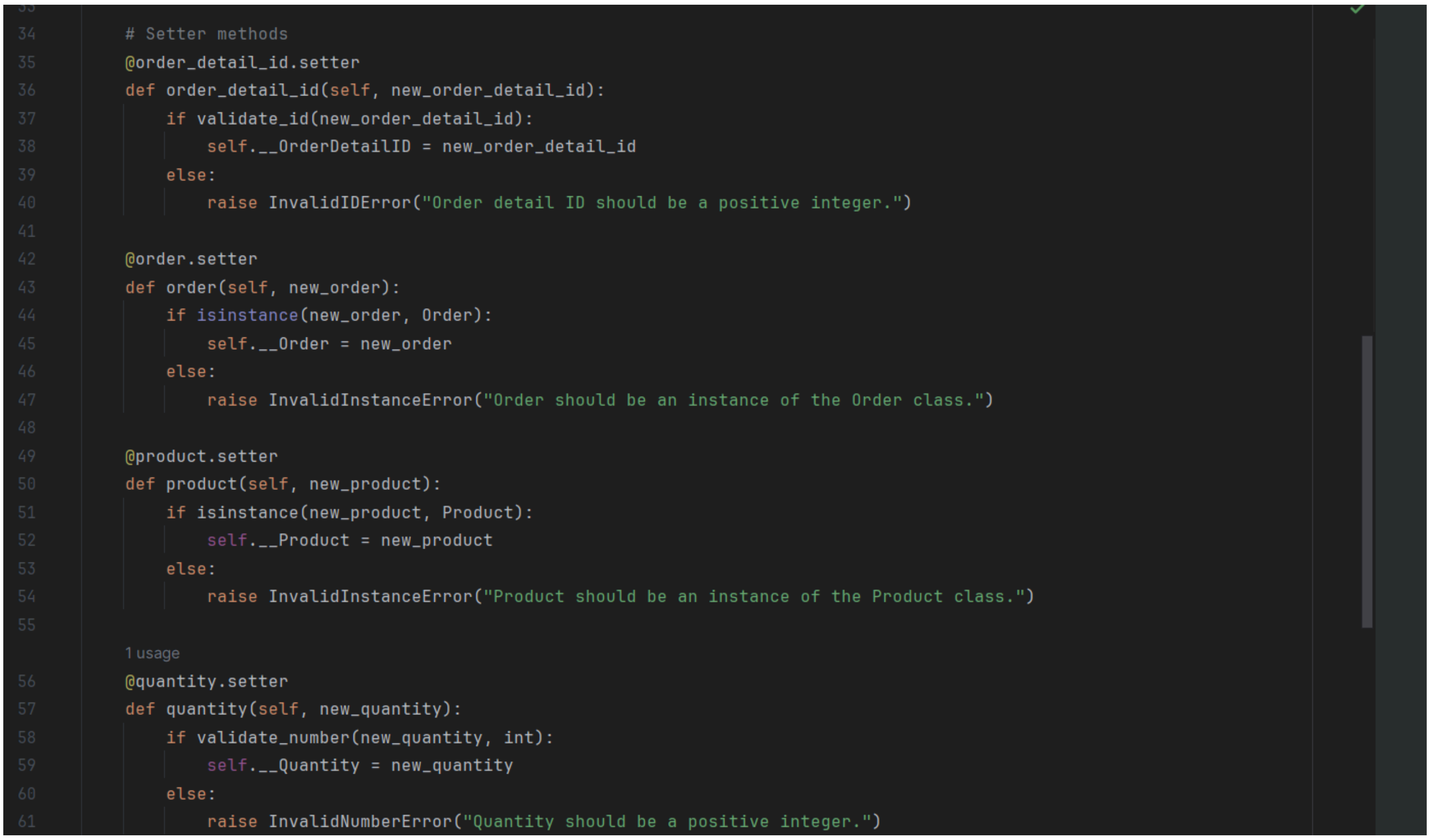


```
    1 usage
    def get_order_details(self):
        print(f"Order ID: {self.order_id}")
        print(f"Order Date: {self.order_date}")
        print(f"Customer: {self.customer.first_name} {self.customer.last_name}")
        print(f"Total Amount: ${self.total_amount:.2f}")
        print(f"Order Status: {self.order_status}")

    1 usage
    def update_order_status(self, new_status):
        self.order_status = new_status

    def cancel_order(self):
        self.update_order_status('Cancelled')
```

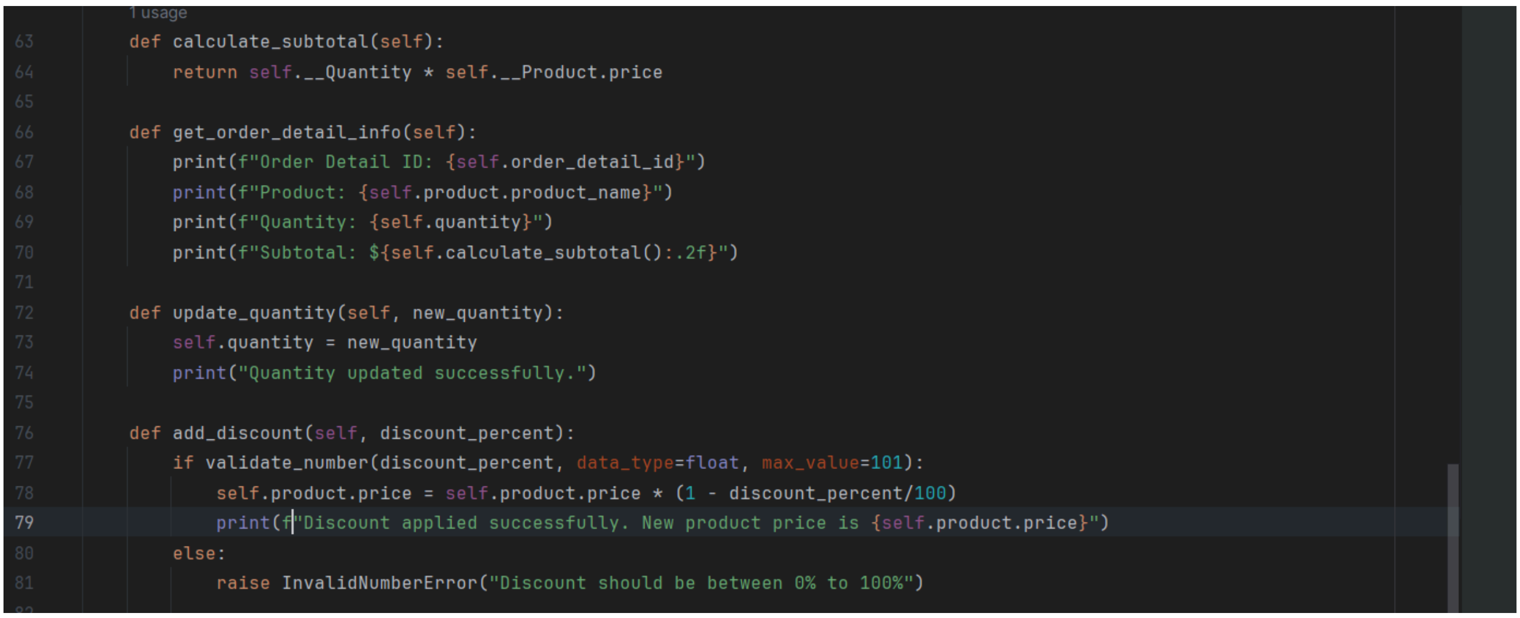
Order Details Class



```
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity):
        if product is None or order is None:
            raise IncompleteOrderException()

        self.__OrderDetailID = order_detail_id
        self.__Order = order
        self.__Product = product
        self.__Quantity = quantity

    # Getter methods
    2 usages
    @property
    def order_detail_id(self):
        return self.__OrderDetailID

    1 usage
    @property
    def order(self):
        return self.__Order

    5 usages
    @property
    def product(self):
        return self.__Product
```

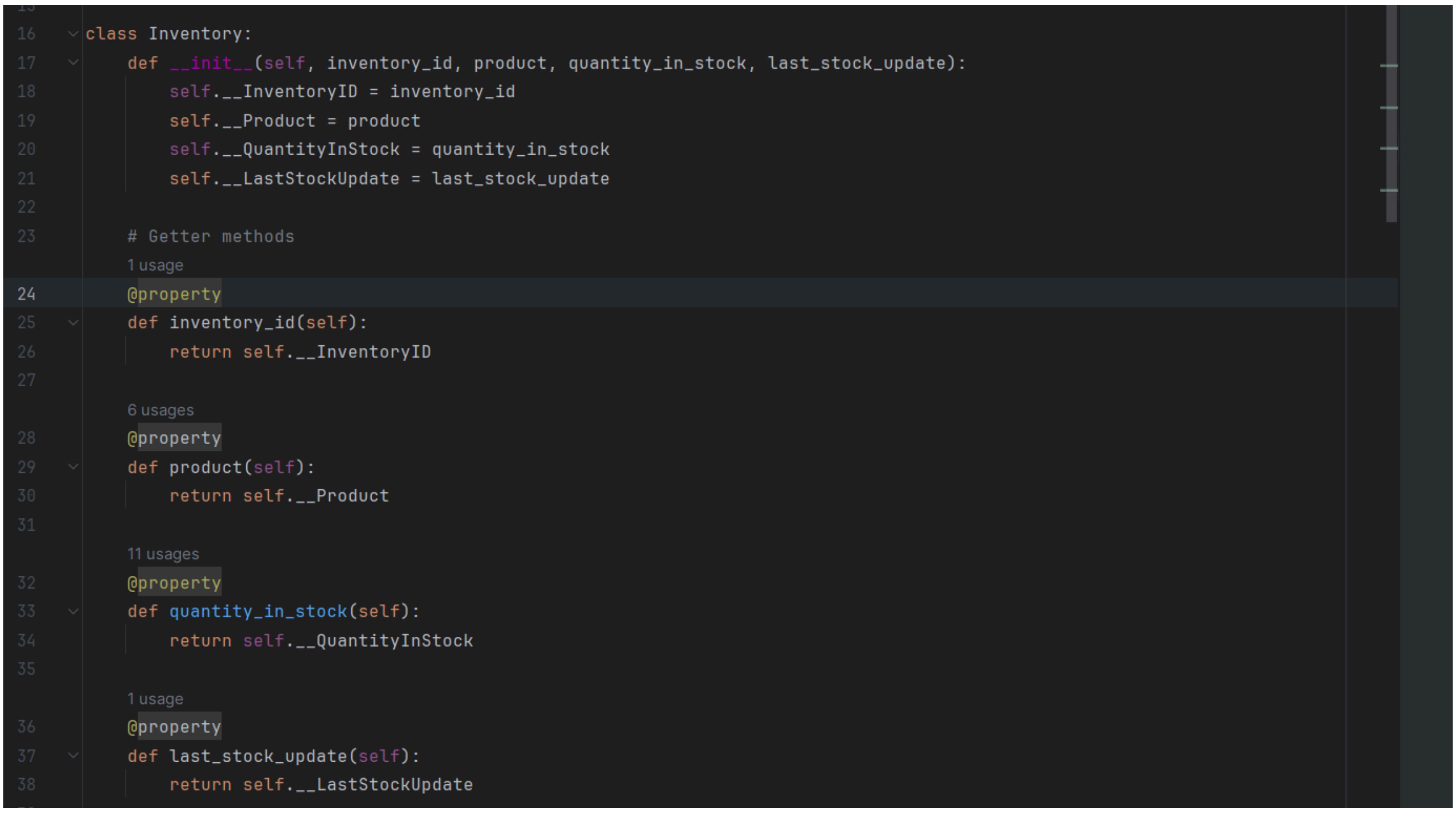


```python
    # Setter methods
    @order_detail_id.setter
    def order_detail_id(self, new_order_detail_id):
        if validate_id(new_order_detail_id):
            self.__OrderDetailID = new_order_detail_id
        else:
            raise InvalidIDError("Order detail ID should be a positive integer.")

    @order.setter
    def order(self, new_order):
        if isinstance(new_order, Order):
            self.__Order = new_order
        else:
            raise InvalidInstanceError("Order should be an instance of the Order class.")

    @product.setter
    def product(self, new_product):
        if isinstance(new_product, Product):
            self.__Product = new_product
        else:
            raise InvalidInstanceError("Product should be an instance of the Product class.")

    1 usage
    @quantity.setter
    def quantity(self, new_quantity):
        if validate_number(new_quantity, int):
            self.__Quantity = new_quantity
        else:
            raise InvalidNumberError("Quantity should be a positive integer.")
```

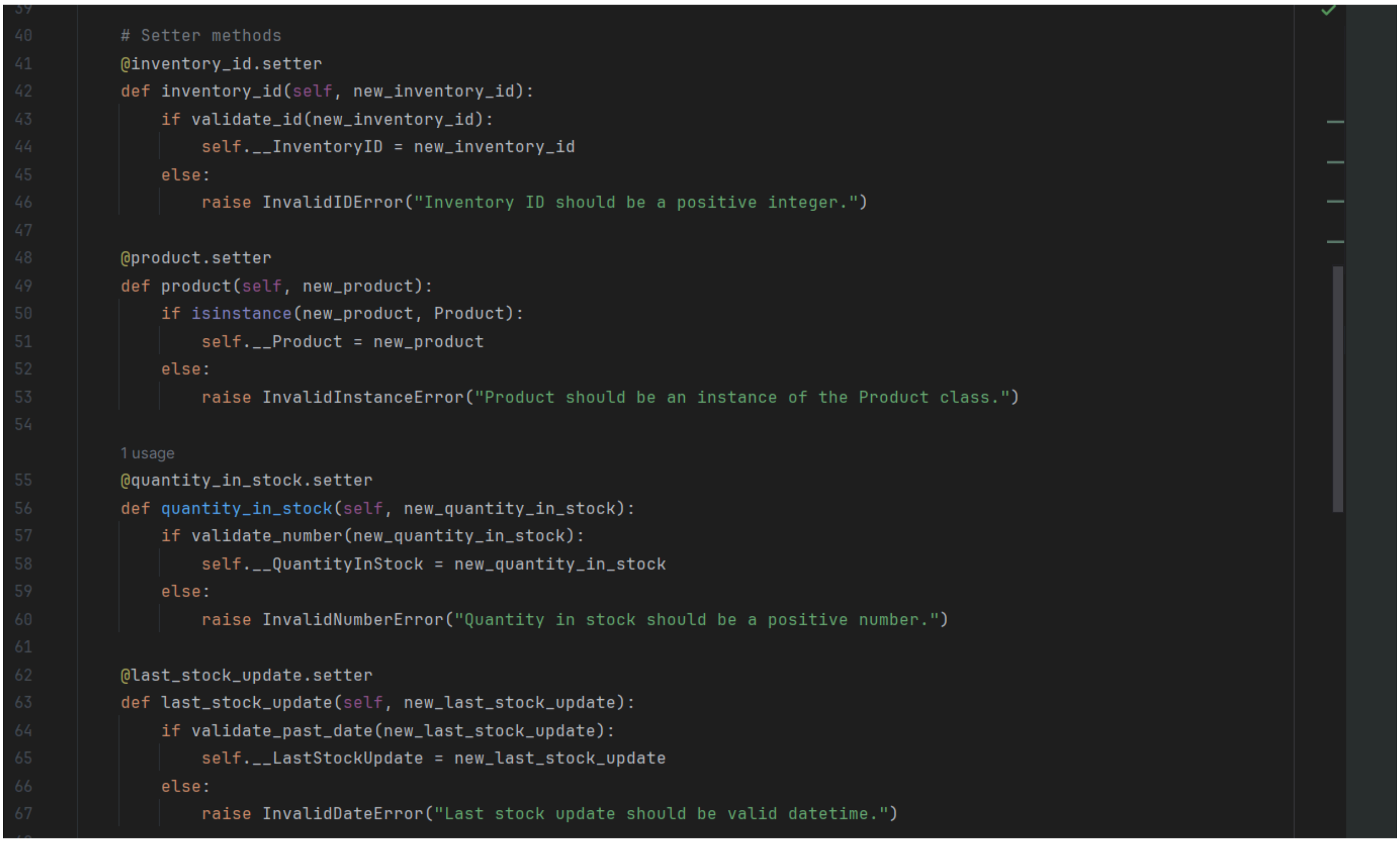


```python
    1 usage
    def calculate_subtotal(self):
        return self.__Quantity * self.__Product.price

    def get_order_detail_info(self):
        print(f"Order Detail ID: {self.order_detail_id}")
        print(f"Product: {self.product.product_name}")
        print(f"Quantity: {self.quantity}")
        print(f"Subtotal: ${self.calculate_subtotal():.2f}")

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

    def add_discount(self, discount_percent):
        if validate_number(discount_percent, data_type=float, max_value=101):
            self.product.price = self.product.price * (1 - discount_percent/100)
            print(f"Discount applied successfully. New product price is {self.product.price}")
        else:
            raise InvalidNumberError("Discount should be between 0% to 100%")
```

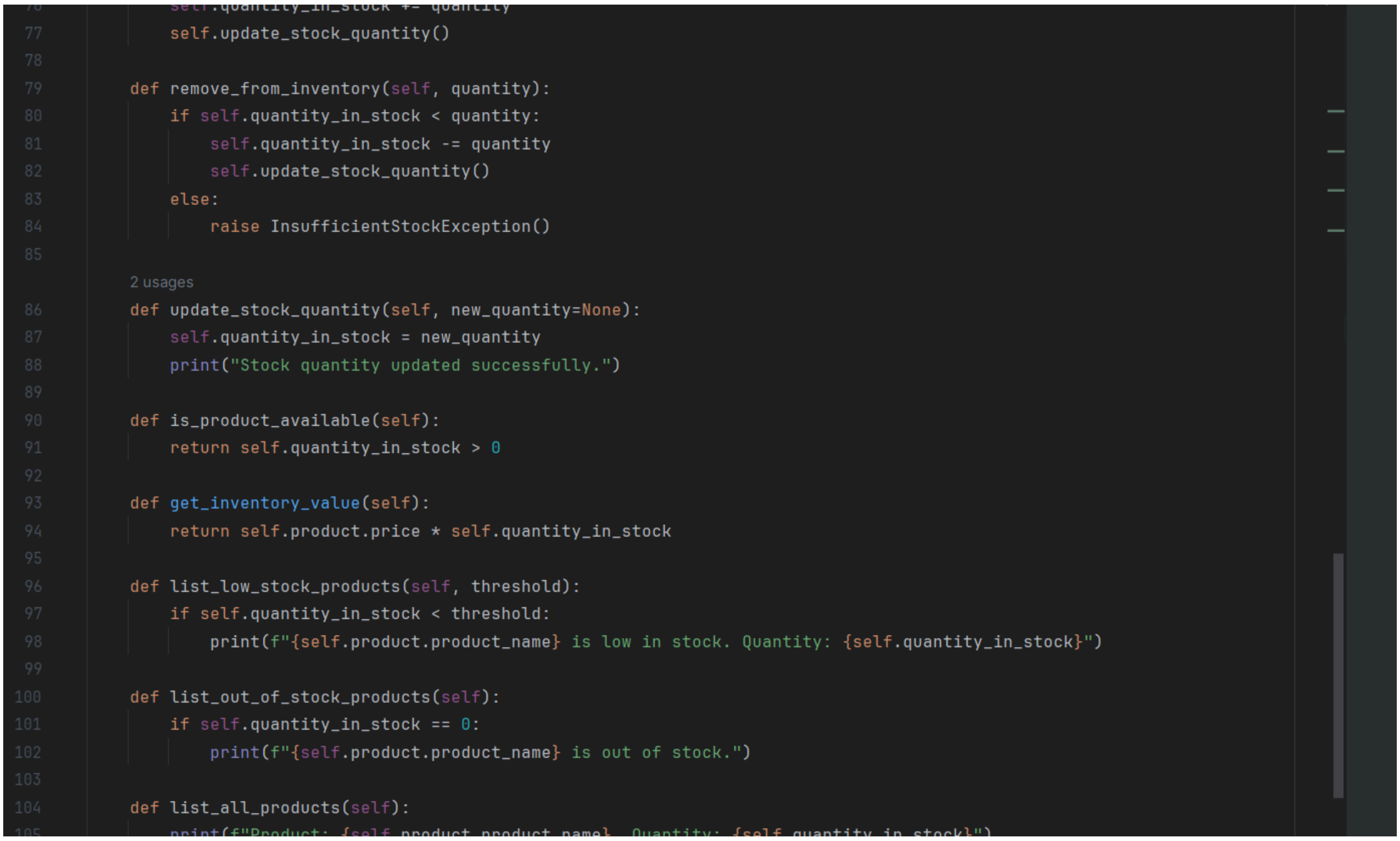
Inventory Class



```
class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock, last_stock_update):
        self.__InventoryID = inventory_id
        self.__Product = product
        self.__QuantityInStock = quantity_in_stock
        self.__LastStockUpdate = last_stock_update

    # Getter methods
    1 usage
    @property
    def inventory_id(self):
        return self.__InventoryID

    6 usages
    @property
    def product(self):
        return self.__Product

    11 usages
    @property
    def quantity_in_stock(self):
        return self.__QuantityInStock

    1 usage
    @property
    def last_stock_update(self):
        return self.__LastStockUpdate
```



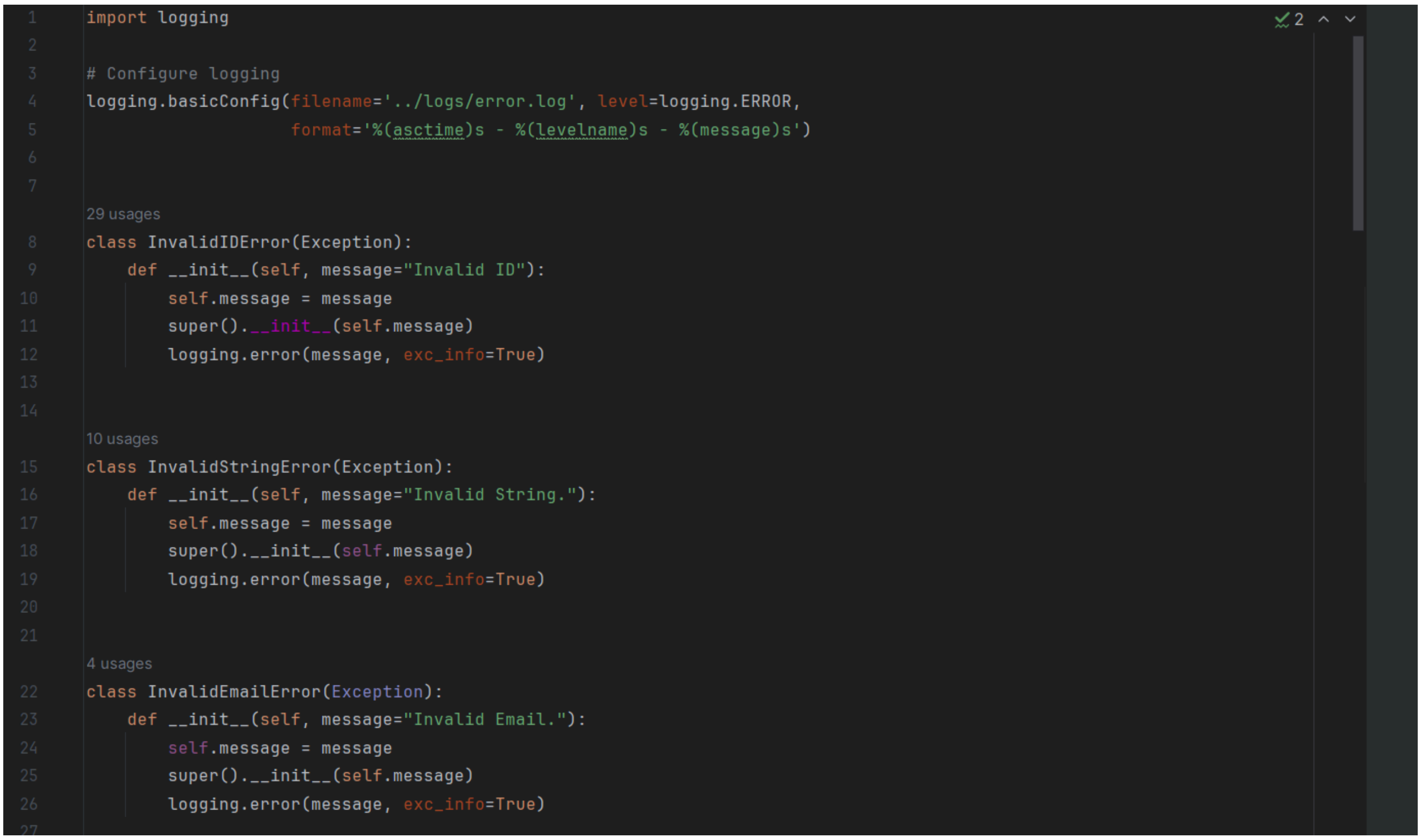
```
    # Setter methods
    @inventory_id.setter
    def inventory_id(self, new_inventory_id):
        if validate_id(new_inventory_id):
            self.__InventoryID = new_inventory_id
        else:
            raise InvalidIDError("Inventory ID should be a positive integer.")

    @product.setter
    def product(self, new_product):
        if isinstance(new_product, Product):
            self.__Product = new_product
        else:
            raise InvalidInstanceError("Product should be an instance of the Product class.")

    1 usage
    @quantity_in_stock.setter
    def quantity_in_stock(self, new_quantity_in_stock):
        if validate_number(new_quantity_in_stock):
            self.__QuantityInStock = new_quantity_in_stock
        else:
            raise InvalidNumberError("Quantity in stock should be a positive number.")

    @last_stock_update.setter
    def last_stock_update(self, new_last_stock_update):
        if validate_past_date(new_last_stock_update):
            self.__LastStockUpdate = new_last_stock_update
        else:
            raise InvalidDateError("Last stock update should be valid datetime.")
```



```
        self.quantity_in_stock += quantity
        self.update_stock_quantity()

    def remove_from_inventory(self, quantity):
        if self.quantity_in_stock < quantity:
            self.quantity_in_stock -= quantity
            self.update_stock_quantity()
        else:
            raise InsufficientStockException()

    2 usages
    def update_stock_quantity(self, new_quantity=None):
        self.quantity_in_stock = new_quantity
        print("Stock quantity updated successfully.")

    def is_product_available(self):
        return self.quantity_in_stock > 0

    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"{self.product.product_name} is low in stock. Quantity: {self.quantity_in_stock}")

    def list_out_of_stock_products(self):
        if self.quantity_in_stock == 0:
            print(f"{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}")
```

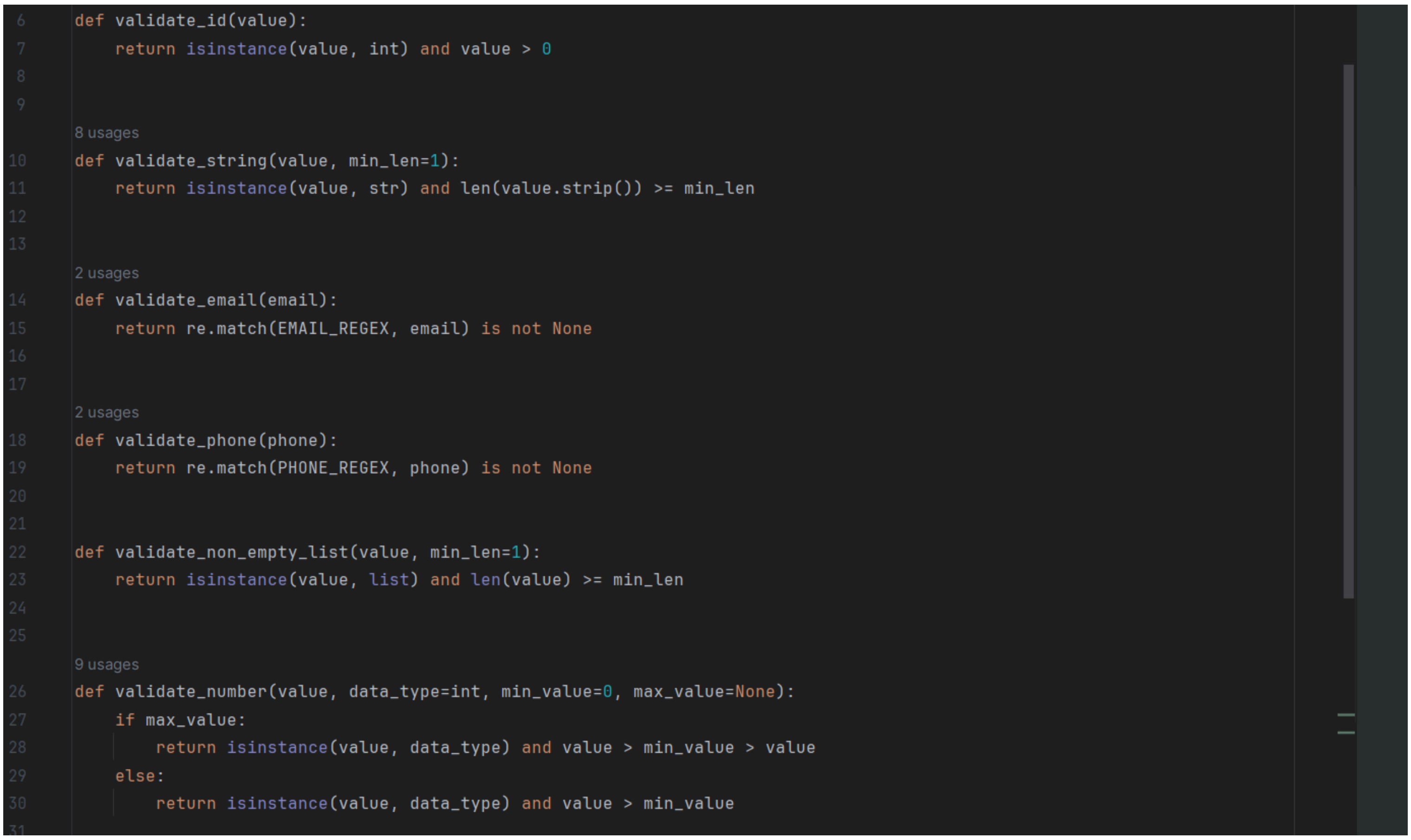
Task 5: Exceptions handling

Created 16 user-defined exceptions with the error logging feature. Used in Model Methods, Services and Interface.



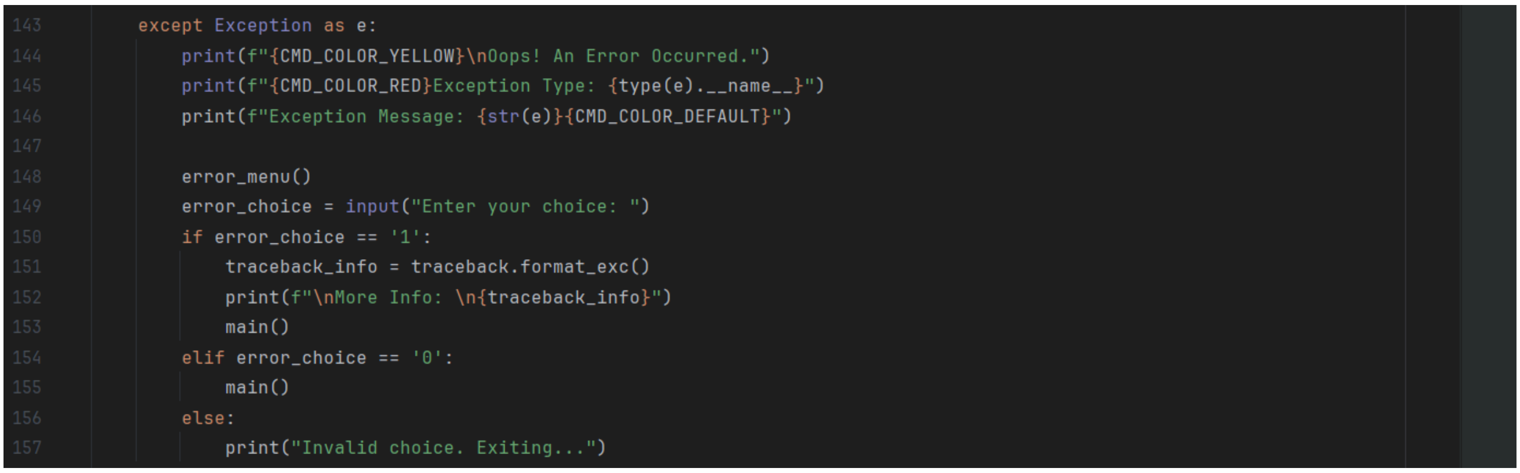
```
import logging

# Configure logging
logging.basicConfig(filename='../logs/error.log', level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(message)s')


29 usages
class InvalidIDError(Exception):
    def __init__(self, message="Invalid ID"):
        self.message = message
        super().__init__(self.message)
        logging.error(message, exc_info=True)


10 usages
class InvalidStringError(Exception):
    def __init__(self, message="Invalid String."):
        self.message = message
        super().__init__(self.message)
        logging.error(message, exc_info=True)


4 usages
class InvalidEmailError(Exception):
    def __init__(self, message="Invalid Email."):
        self.message = message
        super().__init__(self.message)
        logging.error(message, exc_info=True)
```

Created 8 validator methods to validate different types of data efficiently



```
def validate_id(value):
    return isinstance(value, int) and value > 0


8 usages
def validate_string(value, min_len=1):
    return isinstance(value, str) and len(value.strip()) >= min_len


2 usages
def validate_email(email):
    return re.match(EMAIL_REGEX, email) is not None


2 usages
def validate_phone(phone):
    return re.match(PHONE_REGEX, phone) is not None


def validate_non_empty_list(value, min_len=1):
    return isinstance(value, list) and len(value) >= min_len


9 usages
def validate_number(value, data_type=int, min_value=0, max_value=None):
    if max_value:
        return isinstance(value, data_type) and value > min_value > value
    else:
        return isinstance(value, data_type) and value > min_value
```

In Interface, errors are handled with different colors in the terminal with necessary messages



```
    except Exception as e:
        print(f"{CMD_COLOR_YELLOW}\nOops! An Error Occurred.")
        print(f"{CMD_COLOR_RED}Exception Type: {type(e).__name__}")
        print(f"Exception Message: {str(e)}{CMD_COLOR_DEFAULT}")

        error_menu()
        error_choice = input("Enter your choice: ")
        if error_choice == '1':
            traceback_info = traceback.format_exc()
            print(f"\nMore Info: \n{traceback_info}")
            main()
        elif error_choice == '0':
            main()
        else:
            print("Invalid choice. Exiting...")
```

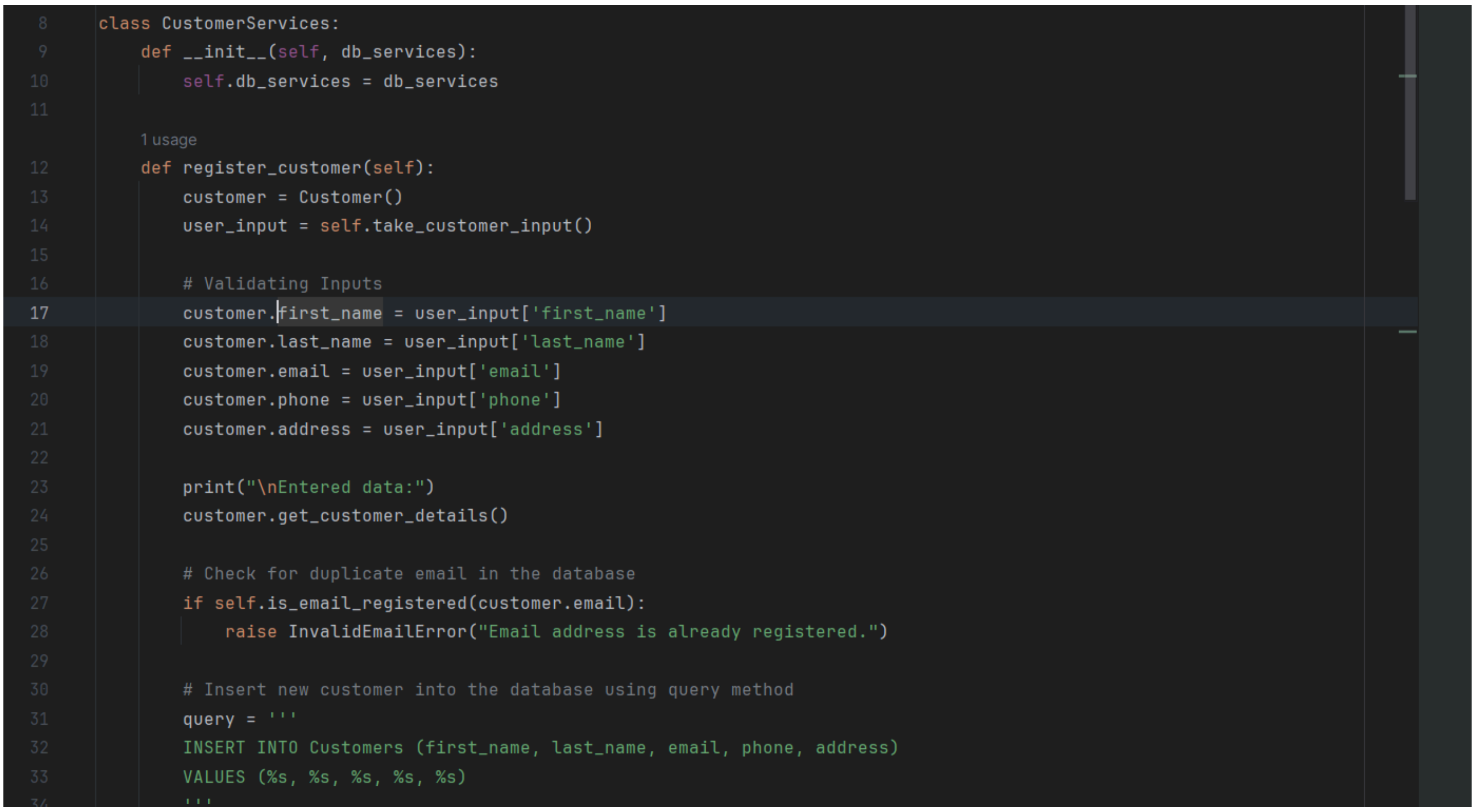
Task 6: Collections

Created services for all classes and database controls.

These services handle business logic and database interactions with the help of query executor form database service.

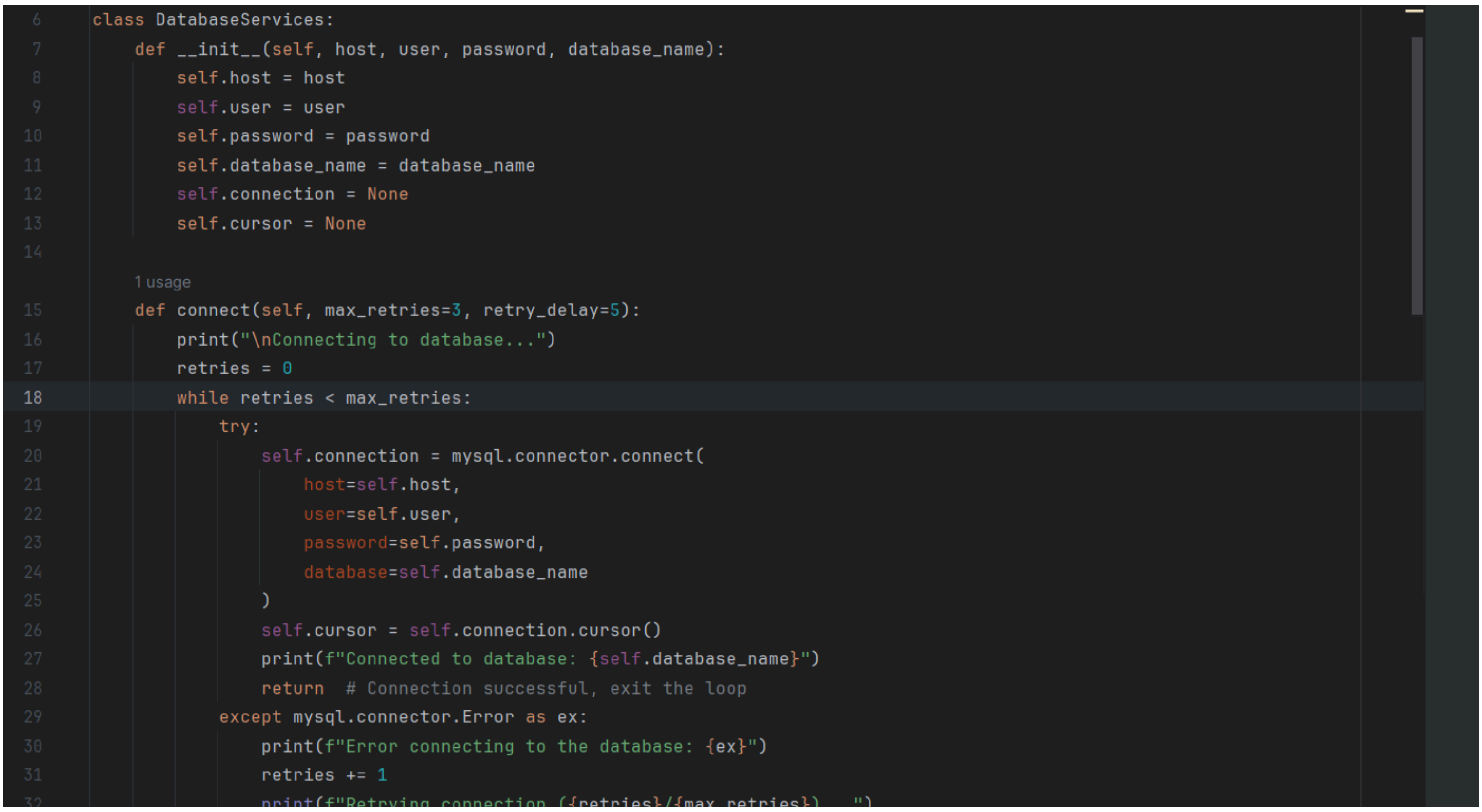
Refer to the actual files for details.

Customer Services



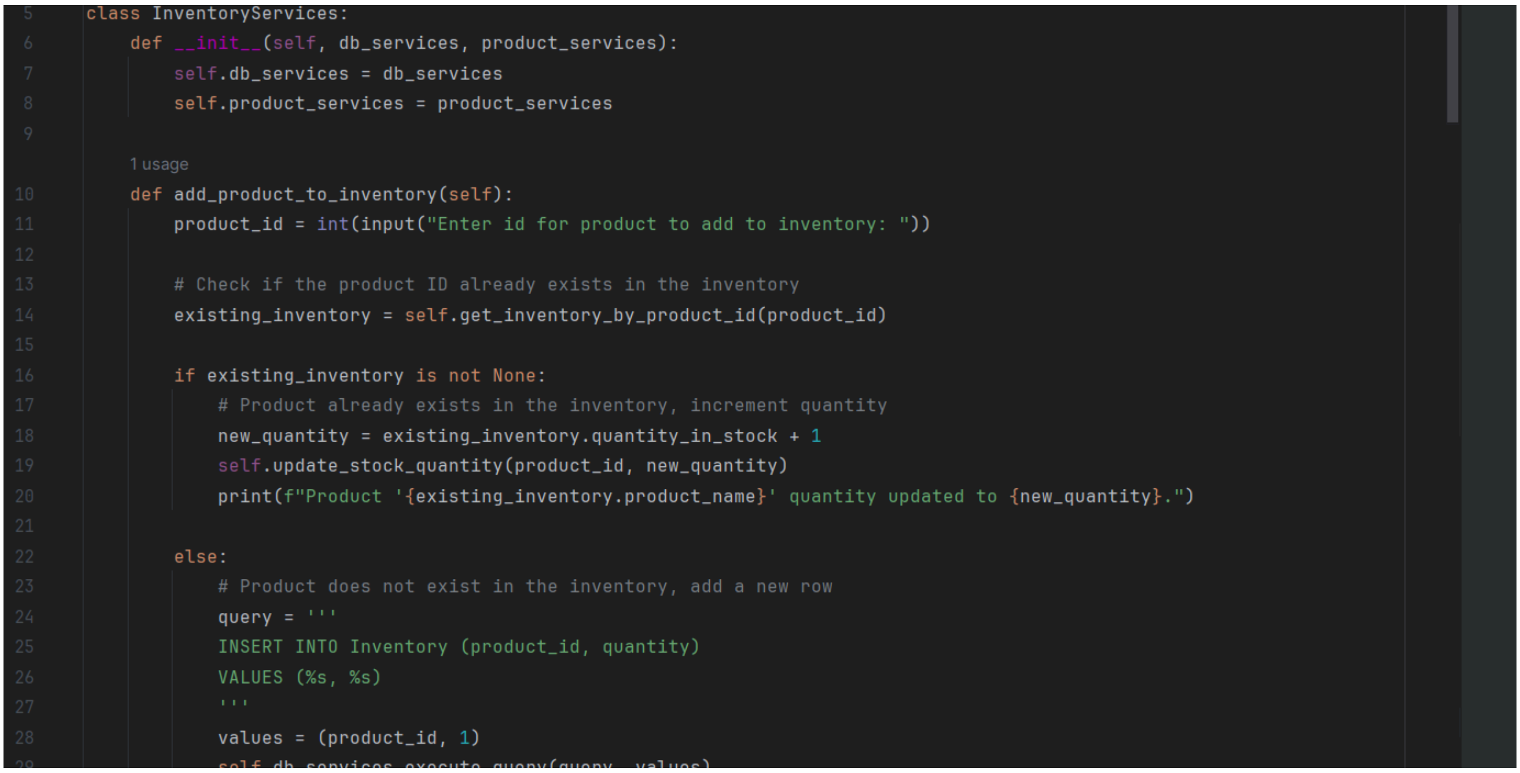
```python
class CustomerServices:
    def __init__(self, db_services):
        self.db_services = db_services

    1 usage
    def register_customer(self):
        customer = Customer()
        user_input = self.take_customer_input()

        # Validating Inputs
        customer.first_name = user_input['first_name']
        customer.last_name = user_input['last_name']
        customer.email = user_input['email']
        customer.phone = user_input['phone']
        customer.address = user_input['address']

        print("\nEntered data:")
        customer.get_customer_details()

        # Check for duplicate email in the database
        if self.is_email_registered(customer.email):
            raise InvalidEmailError("Email address is already registered.")

        # Insert new customer into the database using query method
        query = '''
        INSERT INTO Customers (first_name, last_name, email, phone, address)
        VALUES (%s, %s, %s, %s, %s)
        '''
```

Database Services



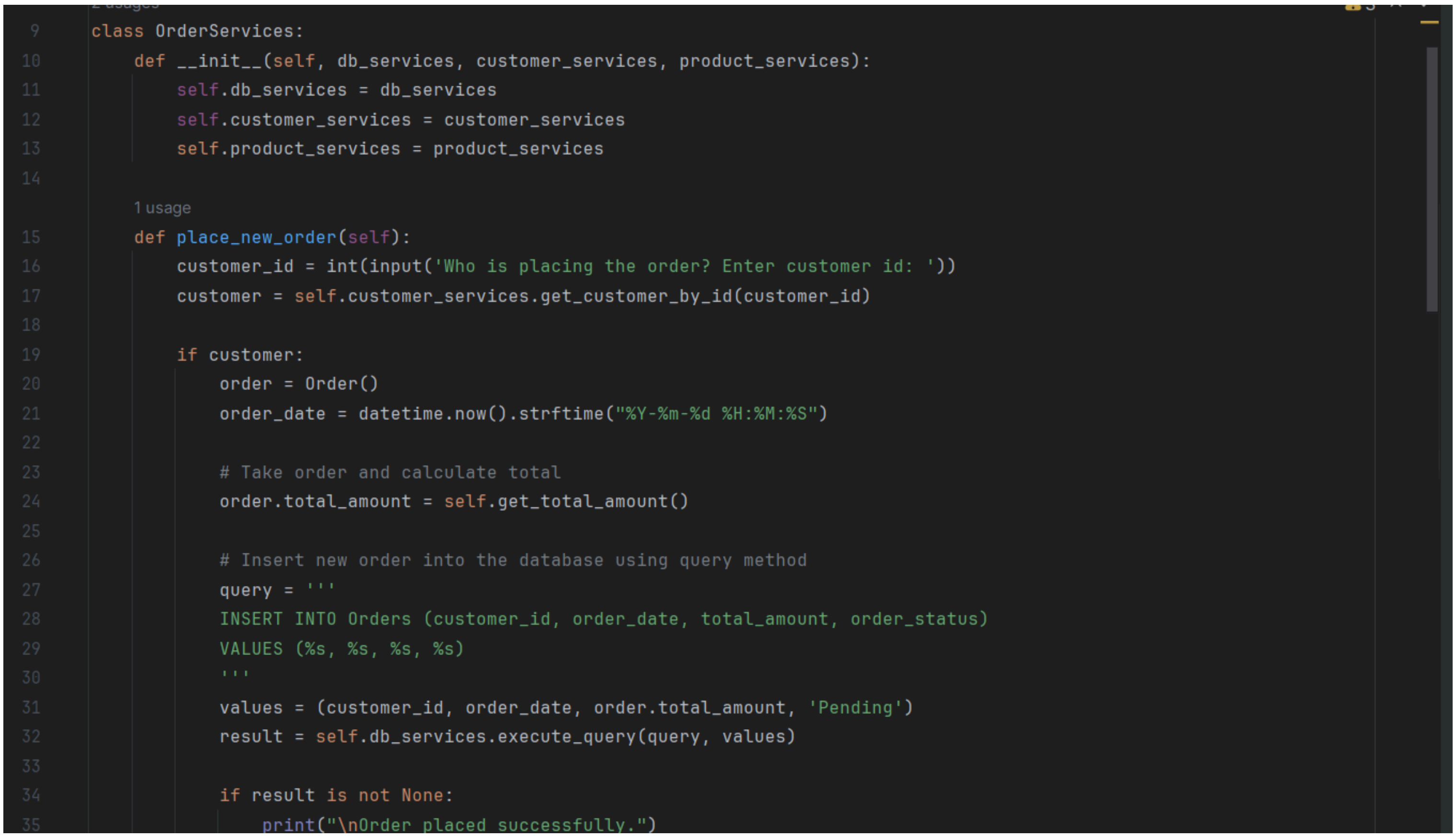
```python
class DatabaseServices:
    def __init__(self, host, user, password, database_name):
        self.host = host
        self.user = user
        self.password = password
        self.database_name = database_name
        self.connection = None
        self.cursor = None

    1 usage
    def connect(self, max_retries=3, retry_delay=5):
        print("\nConnecting to database...")
        retries = 0
        while retries < max_retries:
            try:
                self.connection = mysql.connector.connect(
                    host=self.host,
                    user=self.user,
                    password=self.password,
                    database=self.database_name
                )
                self.cursor = self.connection.cursor()
                print(f"Connected to database: {self.database_name}")
                return  # Connection successful, exit the loop
            except mysql.connector.Error as ex:
                print(f"Error connecting to the database: {ex}")
                retries += 1
                print(f"Retrying connection ({retries}/{max_retries}) ...")
```

Inventory Services



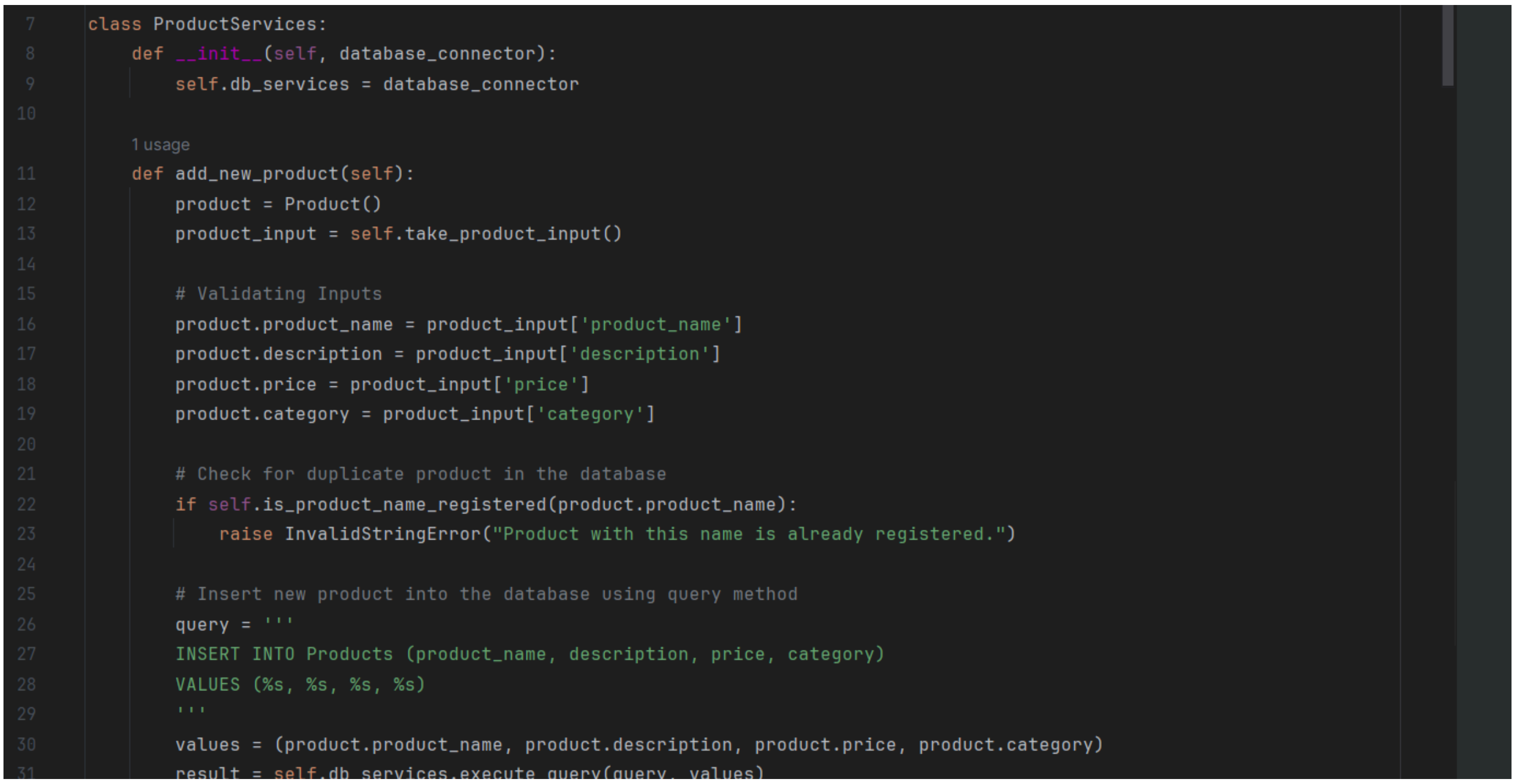
```python
class InventoryServices:
    def __init__(self, db_services, product_services):
        self.db_services = db_services
        self.product_services = product_services

    1 usage
    def add_product_to_inventory(self):
        product_id = int(input("Enter id for product to add to inventory: "))

        # Check if the product ID already exists in the inventory
        existing_inventory = self.get_inventory_by_product_id(product_id)

        if existing_inventory is not None:
            # Product already exists in the inventory, increment quantity
            new_quantity = existing_inventory.quantity_in_stock + 1
            self.update_stock_quantity(product_id, new_quantity)
            print(f"Product '{existing_inventory.product_name}' quantity updated to {new_quantity}.")

        else:
            # Product does not exist in the inventory, add a new row
            query = '''
            INSERT INTO Inventory (product_id, quantity)
            VALUES (%s, %s)
            '''
            values = (product_id, 1)
            self.db_services.execute_query(query, values)
```

Order Services



```python
2 usages
class OrderServices:
    def __init__(self, db_services, customer_services, product_services):
        self.db_services = db_services
        self.customer_services = customer_services
        self.product_services = product_services

    1 usage
    def place_new_order(self):
        customer_id = int(input('Who is placing the order? Enter customer id: '))
        customer = self.customer_services.get_customer_by_id(customer_id)

        if customer:
            order = Order()
            order_date = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            # Take order and calculate total
            order.total_amount = self.get_total_amount()

            # Insert new order into the database using query method
            query = '''
            INSERT INTO Orders (customer_id, order_date, total_amount, order_status)
            VALUES (%s, %s, %s, %s)
            '''
            values = (customer_id, order_date, order.total_amount, 'Pending')
            result = self.db_services.execute_query(query, values)

            if result is not None:
                print("\nOrder placed successfully.")
```

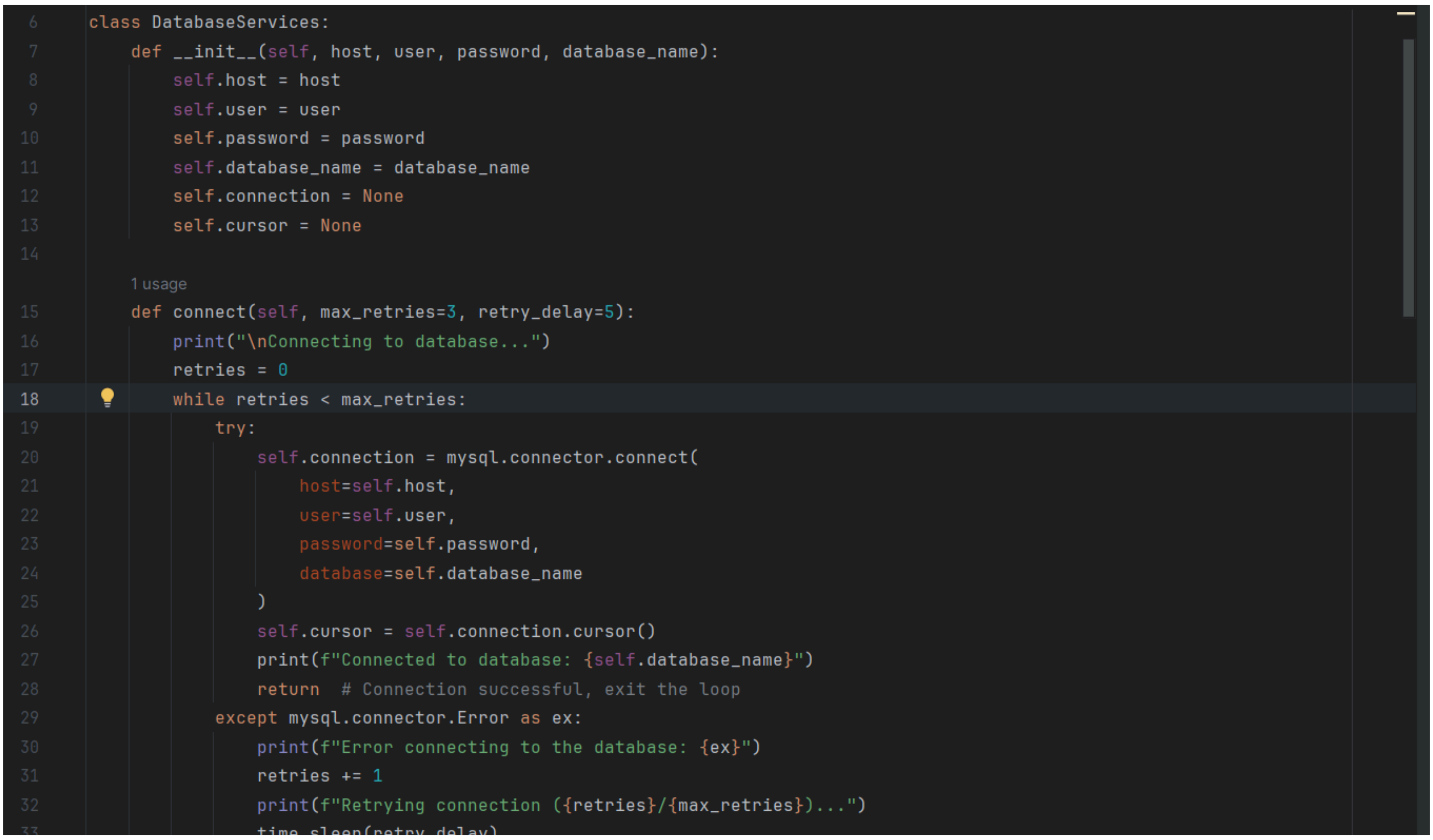
Product Services



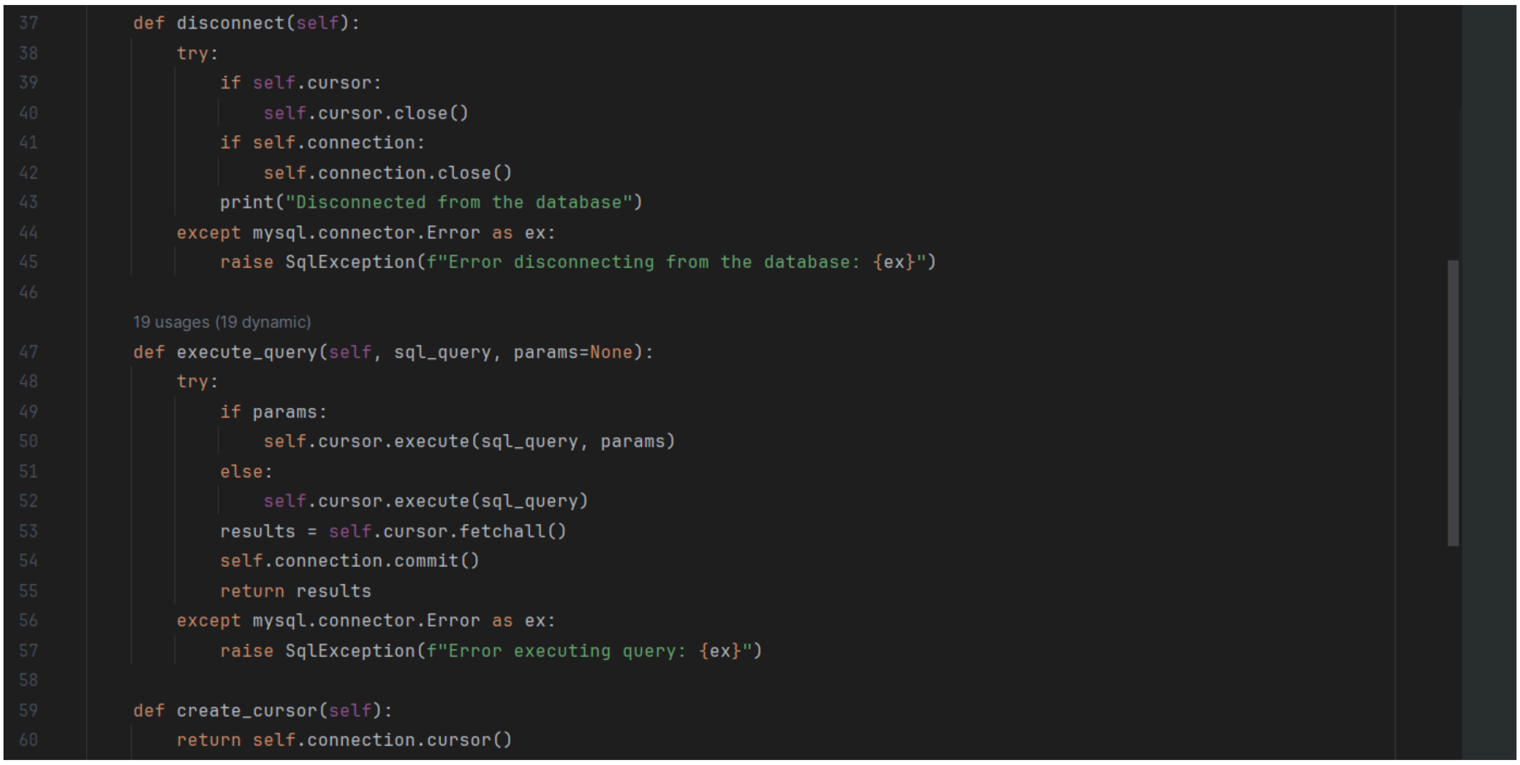
```
class ProductServices:
    def __init__(self, database_connector):
        self.db_services = database_connector

    1 usage
    def add_new_product(self):
        product = Product()
        product_input = self.take_product_input()

        # Validating Inputs
        product.product_name = product_input['product_name']
        product.description = product_input['description']
        product.price = product_input['price']
        product.category = product_input['category']

        # Check for duplicate product in the database
        if self.is_product_name_registered(product.product_name):
            raise InvalidStringError("Product with this name is already registered.")

        # Insert new product into the database using query method
        query = '''
        INSERT INTO Products (product_name, description, price, category)
        VALUES (%s, %s, %s, %s)
        '''
        values = (product.product_name, product.description, product.price, product.category)
        result = self.db_services.execute_query(query, values)
```

Task 7: Database Connectivity
Implemented database connection with mysql-python-connector.
Used .env file to store mysql details.
Implemented a retry connection feature with the ability to handle user-given tweaks.

database_service.py



```
class DatabaseServices:
    def __init__(self, host, user, password, database_name):
        self.host = host
        self.user = user
        self.password = password
        self.database_name = database_name
        self.connection = None
        self.cursor = None

    1 usage
    def connect(self, max_retries=3, retry_delay=5):
        print("\nConnecting to database...")
        retries = 0
        while retries < max_retries:
            try:
                self.connection = mysql.connector.connect(
                    host=self.host,
                    user=self.user,
                    password=self.password,
                    database=self.database_name
                )
                self.cursor = self.connection.cursor()
                print(f"Connected to database: {self.database_name}")
                return  # Connection successful, exit the loop
            except mysql.connector.Error as ex:
                print(f"Error connecting to the database: {ex}")
                retries += 1
                print(f"Retrying connection ({retries}/{max_retries})...")
                time.sleep(retry_delay)
```

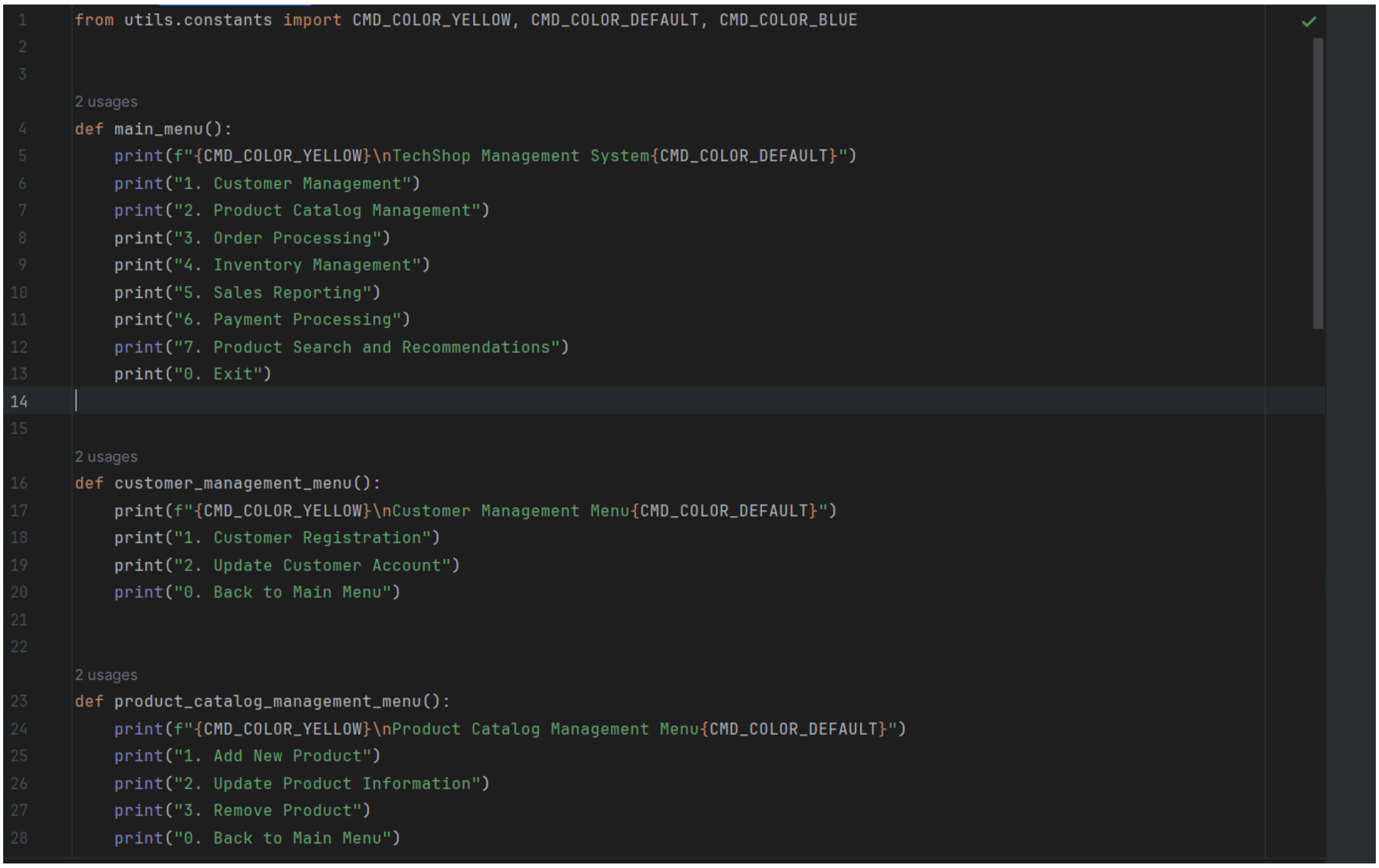


```python
    def disconnect(self):
        try:
            if self.cursor:
                self.cursor.close()
            if self.connection:
                self.connection.close()
            print("Disconnected from the database")
        except mysql.connector.Error as ex:
            raise SqlException(f"Error disconnecting from the database: {ex}")

    19 usages (19 dynamic)
    def execute_query(self, sql_query, params=None):
        try:
            if params:
                self.cursor.execute(sql_query, params)
            else:
                self.cursor.execute(sql_query)
            results = self.cursor.fetchall()
            self.connection.commit()
            return results
        except mysql.connector.Error as ex:
            raise SqlException(f"Error executing query: {ex}")

    def create_cursor(self):
        return self.connection.cursor()
```

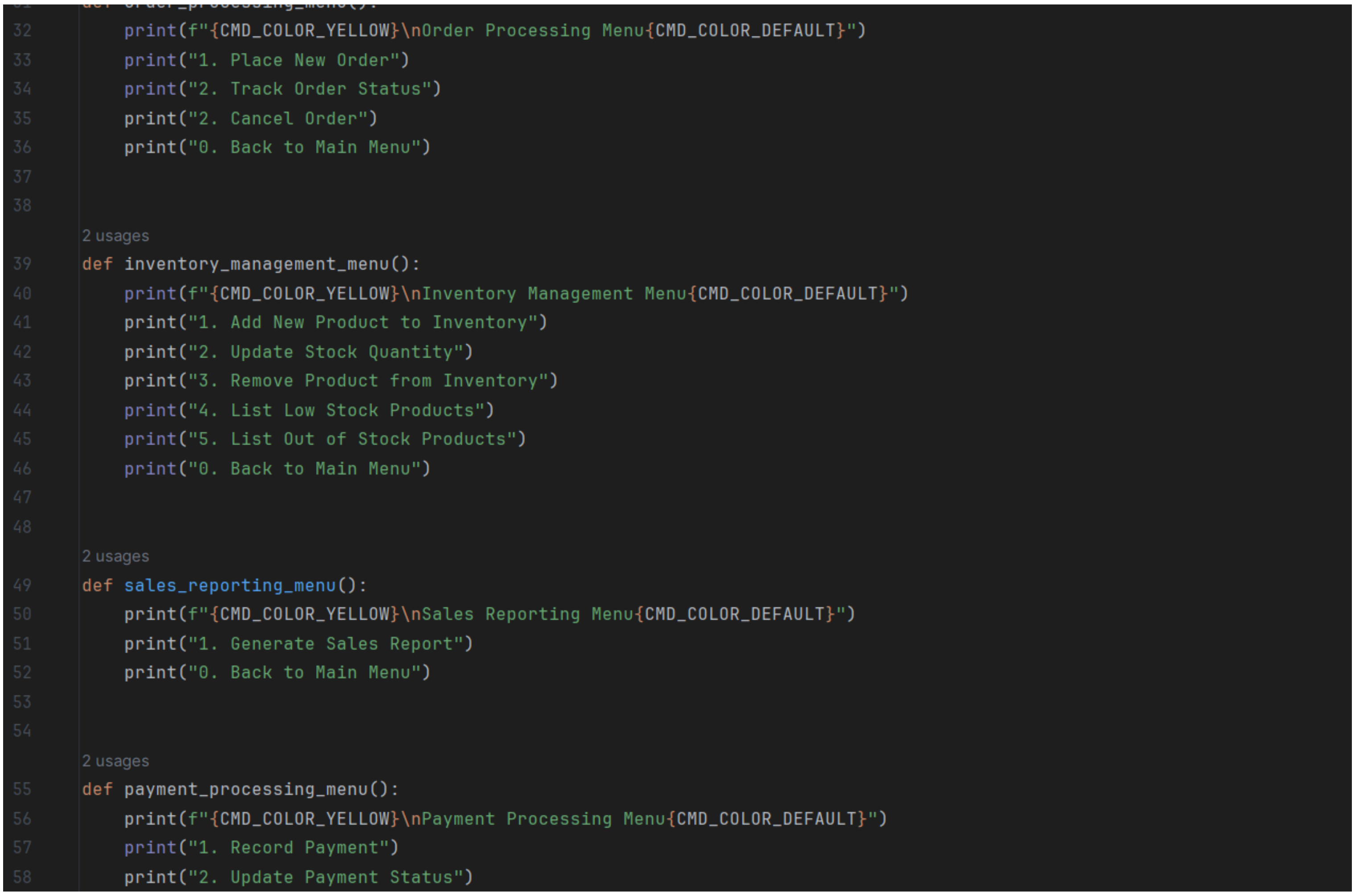
Interface

Implemented Interface menu with 35 different menus with 21 different functionalities.
The interface is user-friendly with different colors for better readability.

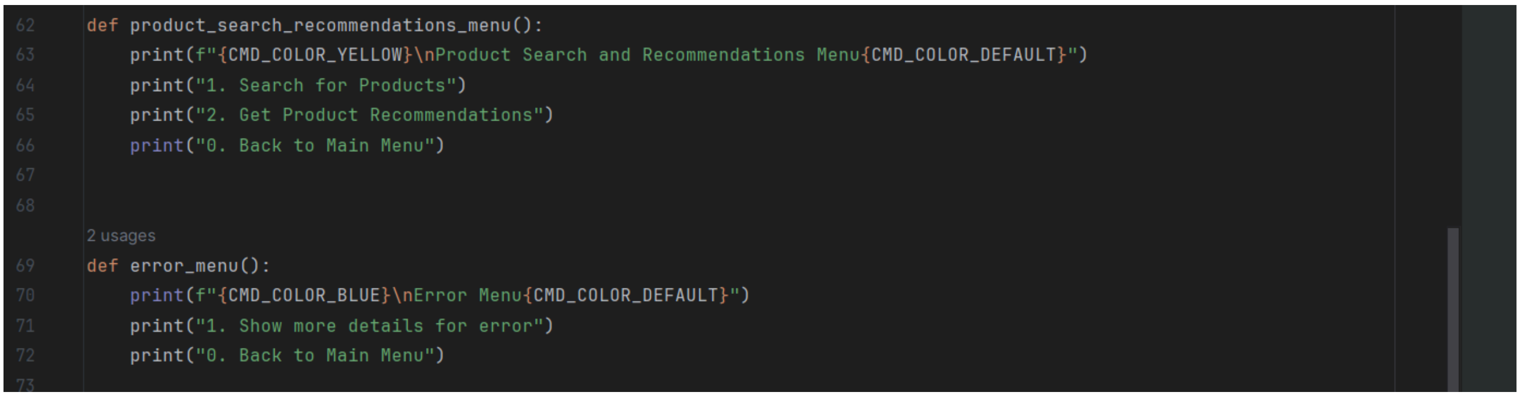
menus.py



```python
from utils.constants import CMD_COLOR_YELLOW, CMD_COLOR_DEFAULT, CMD_COLOR_BLUE


2 usages
def main_menu():
    print(f"{CMD_COLOR_YELLOW}\nTechShop Management System{CMD_COLOR_DEFAULT}")
    print("1. Customer Management")
    print("2. Product Catalog Management")
    print("3. Order Processing")
    print("4. Inventory Management")
    print("5. Sales Reporting")
    print("6. Payment Processing")
    print("7. Product Search and Recommendations")
    print("0. Exit")


2 usages
def customer_management_menu():
    print(f"{CMD_COLOR_YELLOW}\nCustomer Management Menu{CMD_COLOR_DEFAULT}")
    print("1. Customer Registration")
    print("2. Update Customer Account")
    print("0. Back to Main Menu")


2 usages
def product_catalog_management_menu():
    print(f"{CMD_COLOR_YELLOW}\nProduct Catalog Management Menu{CMD_COLOR_DEFAULT}")
    print("1. Add New Product")
    print("2. Update Product Information")
    print("3. Remove Product")
    print("0. Back to Main Menu")
```

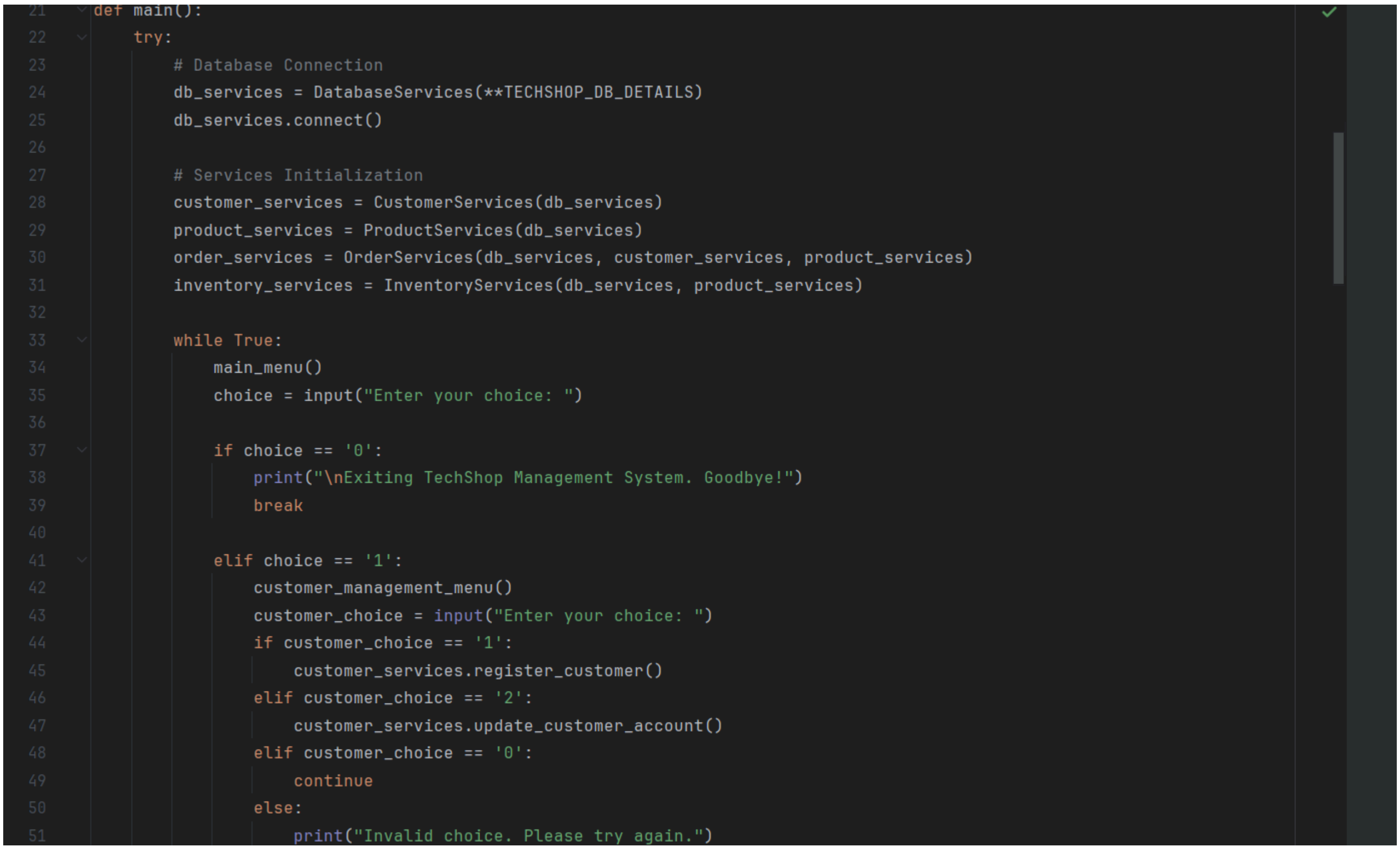


```python
    print(f"{CMD_COLOR_YELLOW}\nOrder Processing Menu{CMD_COLOR_DEFAULT}")
    print("1. Place New Order")
    print("2. Track Order Status")
    print("2. Cancel Order")
    print("0. Back to Main Menu")


2 usages
def inventory_management_menu():
    print(f"{CMD_COLOR_YELLOW}\nInventory Management Menu{CMD_COLOR_DEFAULT}")
    print("1. Add New Product to Inventory")
    print("2. Update Stock Quantity")
    print("3. Remove Product from Inventory")
    print("4. List Low Stock Products")
    print("5. List Out of Stock Products")
    print("0. Back to Main Menu")


2 usages
def sales_reporting_menu():
    print(f"{CMD_COLOR_YELLOW}\nSales Reporting Menu{CMD_COLOR_DEFAULT}")
    print("1. Generate Sales Report")
    print("0. Back to Main Menu")


2 usages
def payment_processing_menu():
    print(f"{CMD_COLOR_YELLOW}\nPayment Processing Menu{CMD_COLOR_DEFAULT}")
    print("1. Record Payment")
    print("2. Update Payment Status")
```

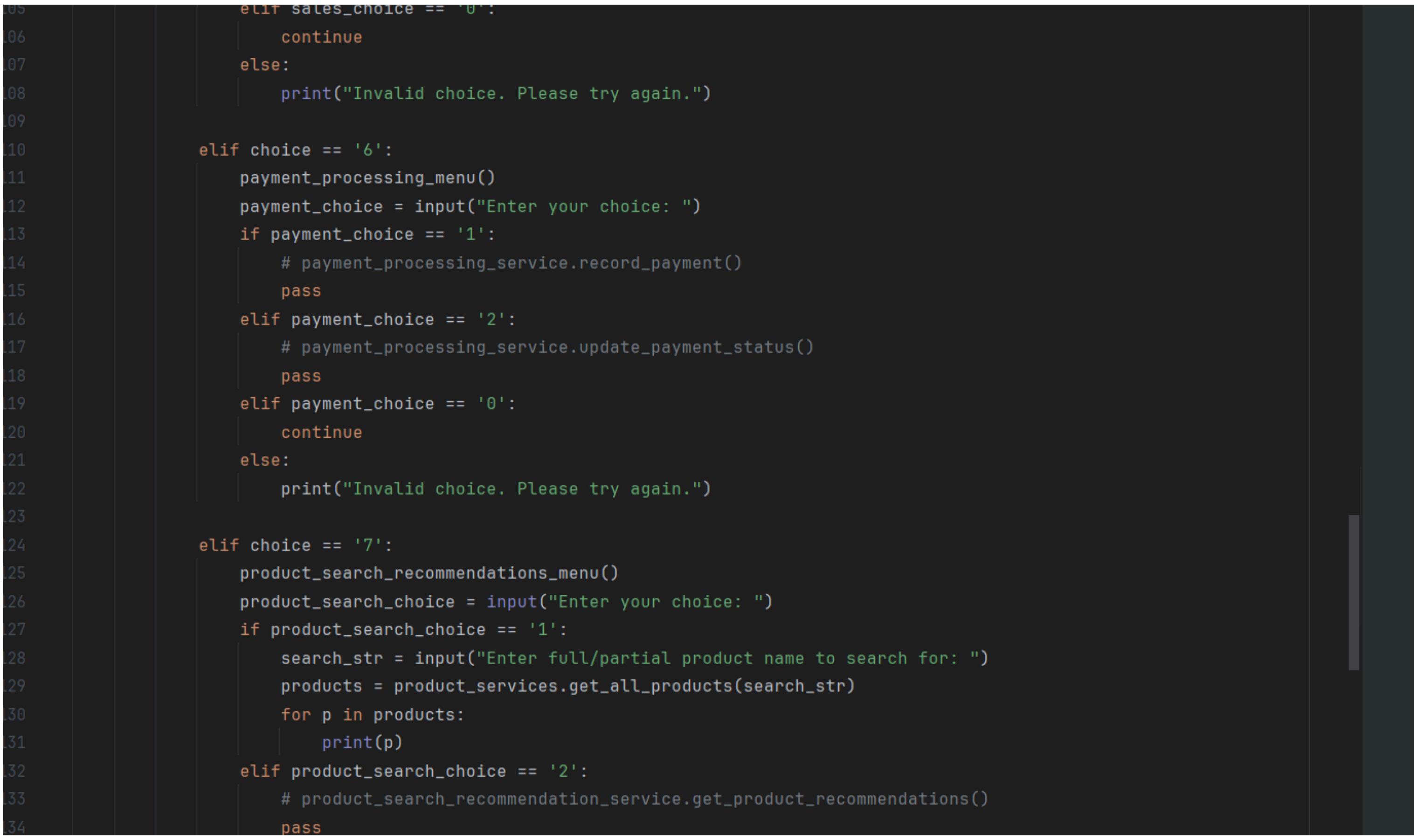


```python
def product_search_recommendations_menu():
    print(f"{CMD_COLOR_YELLOW}\nProduct Search and Recommendations Menu{CMD_COLOR_DEFAULT}")
    print("1. Search for Products")
    print("2. Get Product Recommendations")
    print("0. Back to Main Menu")


2 usages
def error_menu():
    print(f"{CMD_COLOR_BLUE}\nError Menu{CMD_COLOR_DEFAULT}")
    print("1. Show more details for error")
    print("0. Back to Main Menu")
```

interface.py



```python
def main():
    try:
        # Database Connection
        db_services = DatabaseServices(**TECHSHOP_DB_DETAILS)
        db_services.connect()

        # Services Initialization
        customer_services = CustomerServices(db_services)
        product_services = ProductServices(db_services)
        order_services = OrderServices(db_services, customer_services, product_services)
        inventory_services = InventoryServices(db_services, product_services)

        while True:
            main_menu()
            choice = input("Enter your choice: ")

            if choice == '0':
                print("\nExiting TechShop Management System. Goodbye!")
                break

            elif choice == '1':
                customer_management_menu()
                customer_choice = input("Enter your choice: ")
                if customer_choice == '1':
                    customer_services.register_customer()
                elif customer_choice == '2':
                    customer_services.update_customer_account()
                elif customer_choice == '0':
                    continue
                else:
                    print("Invalid choice. Please try again.")
```



```python
                elif sales_choice == '0':
                    continue
                else:
                    print("Invalid choice. Please try again.")

            elif choice == '6':
                payment_processing_menu()
                payment_choice = input("Enter your choice: ")
                if payment_choice == '1':
                    # payment_processing_service.record_payment()
                    pass
                elif payment_choice == '2':
                    # payment_processing_service.update_payment_status()
                    pass
                elif payment_choice == '0':
                    continue
                else:
                    print("Invalid choice. Please try again.")

            elif choice == '7':
                product_search_recommendations_menu()
                product_search_choice = input("Enter your choice: ")
                if product_search_choice == '1':
                    search_str = input("Enter full/partial product name to search for: ")
                    products = product_services.get_all_products(search_str)
                    for p in products:
                        print(p)
                elif product_search_choice == '2':
                    # product_search_recommendation_service.get_product_recommendations()
                    pass
```

Conclusion
Overall it is a full-fledged backend and database connection implementation. I recommend you check the project file by file to see all the features and miscellaneous things implemented.

Thank You!