No canvas requested

Here is the rewritten version of your Python program with improved readability and structure while maintaining all functionalities:

python

```python
class Item:
    """Represents an item in the inventory with name, price, quantity, and sales tracking."""

    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity
        self.sold_quantity = 0

    def update_quantity(self, quantity):
        """Updates the quantity of the item in inventory."""
        self.quantity += quantity

    def sell_item(self, quantity):
        """Processes the sale of an item if enough stock is available."""
        if self.quantity >= quantity:
            self.quantity -= quantity
            self.sold_quantity += quantity
            return True
        return False

    def calculate_profit(self):
        """Calculates the total profit from sold items."""
        return self.sold_quantity * self.price

    def __str__(self):
        return f"{self.name} - Price: ${self.price:.2f}, Quantity: {self.quantity}"


class Inventory:
    """Manages the supermarket inventory, including adding, selling, and analyzing items."""

    def __init__(self):
```

```
        self.items = {}

    def add_item(self, name, price, quantity):
        """Adds a new item or updates the quantity of an existing item."""
        if name in self.items:
            self.items[name].update_quantity(quantity)
        else:
            self.items[name] = Item(name, price, quantity)
        print(f"Item '{name}' added/updated successfully.")

    def display_items(self):
        """Displays all available items in the inventory."""
        if not self.items:
            print("No items in inventory.")
        else:
            for item in self.items.values():
                print(item)

    def purchase_item(self, name, quantity):
        """Handles the purchase of an item if available."""
        if name in self.items:
            item = self.items[name]
            if item.sell_item(quantity):
                total_cost = item.price * quantity
                print(f"Purchase successful! Total cost: ${total_cost:.2f}")
            else:
                print(f"Insufficient stock for '{name}'.")
        else:
            print(f"Item '{name}' not found in inventory.")

    def check_stock(self, name):
        """Checks the stock availability of an item."""
        if name in self.items:
            item = self.items[name]
            status = "in stock" if item.quantity > 0 else "out of stock"
            print(f"{name} is {status}. Available quantity: {item.quantity}")
        else:
            print(f"Item '{name}' not found in inventory.")

    def sales_analysis(self):
        """Analyzes sales data to determine the most and least profitable items."""
        if not self.items:
```

```
            print("No sales data available.")
            return

        max_profit_item = max(self.items.values(), key=lambda i:
i.calculate_profit(), default=None)
        min_profit_item = min(self.items.values(), key=lambda i:
i.calculate_profit(), default=None)

        print("\nSales Analysis:")
        if max_profit_item:
            print(f"Highest Profit Item: {max_profit_item.name} (Profit:
${max_profit_item.calculate_profit():.2f})")
        if min_profit_item:
            print(f"Lowest Profit Item: {min_profit_item.name} (Profit:
${min_profit_item.calculate_profit():.2f})")


def main():
    """Runs the main menu-driven program for the supermarket inventory system."""
    inventory = Inventory()

    while True:
        print("\nSupermarket Inventory System")
        print("1. Add Item to Inventory")
        print("2. Display Available Items")
        print("3. Purchase Item")
        print("4. Check Stock Availability")
        print("5. Sales Analysis")
        print("6. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            name = input("Enter item name: ")
            price = float(input("Enter item price: "))
            quantity = int(input("Enter item quantity: "))
            inventory.add_item(name, price, quantity)

        elif choice == "2":
            print("\nAvailable Items:")
            inventory.display_items()
```

```
        elif choice == "3":
            name = input("Enter item name to purchase: ")
            quantity = int(input("Enter quantity to purchase: "))
            inventory.purchase_item(name, quantity)

        elif choice == "4":
            name = input("Enter item name to check stock: ")
            inventory.check_stock(name)

        elif choice == "5":
            inventory.sales_analysis()

        elif choice == "6":
            print("Exiting the program. Goodbye!")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Improvements:

1. **Code Readability**
   - Added docstrings to classes and methods for better understanding.
   - Improved comments for clarity.
2. **Efficiency**
   - Used `max()` and `min()` with a key function in `sales_analysis()` to optimize finding the most and least profitable items.
3. **Better User Experience**
   - Provided clear success/error messages for inventory updates and purchases.

This version maintains all functionalities while making the code cleaner and easier to understand. 🚀

Here's a Python program that implements a library management system using OOP principles:

python

```python
class Book:
    """Represents a book in the library with title, author, available copies, and borrow count."""

    def __init__(self, title, author, copies):
        self.title = title
        self.author = author
        self.copies = copies
        self.borrowed_count = 0

    def borrow_book(self, quantity):
        """Allows borrowing books if sufficient copies are available."""
        if self.copies >= quantity:
            self.copies -= quantity
            self.borrowed_count += quantity
            return True
        return False

    def return_book(self, quantity):
        """Allows returning books and updates available copies."""
        self.copies += quantity

    def __str__(self):
        return f"'{self.title}' by {self.author} - Available copies: {self.copies}"


class Library:
    """Manages the library, including adding, borrowing, returning, and tracking books."""

    def __init__(self):
        self.books = {}

    def add_book(self, title, author, copies):
        """Adds a book to the library or updates available copies."""
        if title in self.books:
            self.books[title].copies += copies
        else:
            self.books[title] = Book(title, author, copies)
        print(f"Book '{title}' added/updated successfully.")
```

```
    def display_books(self):
        """Displays all available books in the library."""
        if not self.books:
            print("No books available in the library.")
        else:
            print("\nAvailable Books:")
            for book in self.books.values():
                print(book)

    def borrow_book(self, title, quantity):
        """Allows users to borrow a book if it's available."""
        if title in self.books:
            book = self.books[title]
            if book.borrow_book(quantity):
                print(f"Successfully borrowed {quantity} copy(ies) of '{title}'.")
            else:
                print(f"Insufficient copies of '{title}' available.")
        else:
            print(f"Book '{title}' not found in the library.")

    def return_book(self, title, quantity):
        """Allows users to return a borrowed book."""
        if title in self.books:
            self.books[title].return_book(quantity)
            print(f"Successfully returned {quantity} copy(ies) of '{title}'.")
        else:
            print(f"Book '{title}' does not belong to this library.")

    def track_popular_books(self):
        """Displays the top 3 most borrowed books."""
        if not self.books:
            print("No books have been borrowed yet.")
            return

        sorted_books = sorted(self.books.values(), key=lambda b: b.borrowed_count,
reverse=True)
        top_books = sorted_books[:3]

        print("\nTop 3 Most Popular Books:")
        for book in top_books:
            print(f"'{book.title}' by {book.author} - Borrowed {book.borrowed_count}
```

```
time(s)")


def main():
    """Runs the main menu-driven program for the library management system."""
    library = Library()

    while True:
        print("\nLibrary Management System")
        print("1. Add Book")
        print("2. Display Available Books")
        print("3. Borrow Book")
        print("4. Return Book")
        print("5. Track Popular Books")
        print("6. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            title = input("Enter book title: ")
            author = input("Enter author name: ")
            copies = int(input("Enter number of copies: "))
            library.add_book(title, author, copies)

        elif choice == "2":
            library.display_books()

        elif choice == "3":
            title = input("Enter book title to borrow: ")
            quantity = int(input("Enter quantity to borrow: "))
            library.borrow_book(title, quantity)

        elif choice == "4":
            title = input("Enter book title to return: ")
            quantity = int(input("Enter quantity to return: "))
            library.return_book(title, quantity)

        elif choice == "5":
            library.track_popular_books()

        elif choice == "6":
            print("Exiting the program. Goodbye!")
```

```
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Features:

1. `Book` **Class:**
   - Stores book details (`title`, `author`, `copies`, `borrowed_count`).
   - Handles borrowing and returning books.
2. `Library` **Class:**
   - Manages a collection of books.
   - Adds books, displays available books, handles borrowing and returning, and tracks popular books.
3. **Main Program (User Interface):**
   - Provides a menu-driven system for interacting with the library.

How It Works:

- Run the program and use the menu to manage books.
- Borrow and return books while tracking the most borrowed ones.
- The program ensures books are borrowed only if available.

This system efficiently manages a library using OOP principles. 🚀

Here is a Python program implementing an **Online Shopping Cart System** using OOP principles:

python

```
class Product:
    """Represents a product with name, price, and quantity."""

    def __init__(self, name, price, quantity):
```

```
        self.name = name
        self.price = price
        self.quantity = quantity

    def calculate_total_price(self):
        """Returns the total price for the product based on quantity."""
        return self.price * self.quantity

    def __str__(self):
        return f"{self.name} - Price: ${self.price:.2f}, Quantity: {self.quantity},
Total: ${self.calculate_total_price():.2f}"


class ShoppingCart:
    """Manages the shopping cart, including adding, removing, and displaying
products."""

    def __init__(self):
        self.cart = {}

    def add_product(self, name, price, quantity):
        """Adds a product to the cart or updates its quantity."""
        if name in self.cart:
            self.cart[name].quantity += quantity
        else:
            self.cart[name] = Product(name, price, quantity)
        print(f"'{name}' added/updated successfully in the cart.")

    def display_cart(self):
        """Displays all products in the cart."""
        if not self.cart:
            print("Your shopping cart is empty.")
        else:
            print("\nShopping Cart Contents:")
            for product in self.cart.values():
                print(product)

    def remove_product(self, name):
        """Removes a product from the cart."""
        if name in self.cart:
            del self.cart[name]
            print(f"'{name}' removed from the cart.")
```

```
        else:
            print(f"Product '{name}' not found in the cart.")

    def calculate_total_bill(self):
        """Calculates the total bill for all products in the cart."""
        return sum(product.calculate_total_price() for product in
self.cart.values())

    def apply_discount(self, discount_percentage):
        """Applies a discount to the total bill and returns the final amount."""
        total_bill = self.calculate_total_bill()
        discount_amount = (discount_percentage / 100) * total_bill
        final_amount = total_bill - discount_amount
        print(f"\nTotal Bill: ${total_bill:.2f}")
        print(f"Discount Applied: {discount_percentage}% (-${discount_amount:.2f})")
        print(f"Final Amount After Discount: ${final_amount:.2f}")
        return final_amount


def main():
    """Runs the main menu-driven program for the shopping cart system."""
    cart = ShoppingCart()

    while True:
        print("\nOnline Shopping Cart System")
        print("1. Add Product to Cart")
        print("2. Display Cart Contents")
        print("3. Remove Product from Cart")
        print("4. Calculate Total Bill")
        print("5. Apply Discount")
        print("6. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            name = input("Enter product name: ")
            price = float(input("Enter product price: "))
            quantity = int(input("Enter quantity: "))
            cart.add_product(name, price, quantity)

        elif choice == "2":
            cart.display_cart()
```

```
        elif choice == "3":
            name = input("Enter product name to remove: ")
            cart.remove_product(name)

        elif choice == "4":
            total_bill = cart.calculate_total_bill()
            print(f"Total Bill: ${total_bill:.2f}")

        elif choice == "5":
            discount_percentage = float(input("Enter discount percentage: "))
            cart.apply_discount(discount_percentage)

        elif choice == "6":
            print("Exiting the program. Happy Shopping!")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Features:

1. `Product` **Class:**
   - Stores product details (`name`, `price`, `quantity`).
   - Calculates the total price for a product.
2. `ShoppingCart` **Class:**
   - Manages a collection of products.
   - Allows adding, removing, and displaying products.
   - Computes the total bill and applies discounts.
3. **Main Program (User Interface):**
   - Provides a menu-driven system for managing the shopping cart.

How It Works:

- Run the program and use the menu to add, remove, and view products in the cart.
- Calculate the total bill and apply a discount to see the final amount.
- The program ensures that users can efficiently manage their shopping cart.

This system is an effective and user-friendly way to handle online shopping! 🛒🚀

Here is a Python program that implements a **Student Grade Management System** using OOP principles:

python

```
class Student:
    """Represents a student with a name, student ID, and grade."""

    def __init__(self, name, student_id, grade):
        self.name = name
        self.student_id = student_id
        self.grade = grade

    def update_grade(self, new_grade):
        """Updates the student's grade."""
        self.grade = new_grade

    def __str__(self):
        return f"ID: {self.student_id} | Name: {self.name} | Grade:
{self.grade:.2f}"


class GradeManagementSystem:
    """Manages student records, including adding, updating, and analyzing grades."""

    def __init__(self):
        self.students = {}

    def add_student(self, name, student_id, grade):
        """Adds a new student to the system."""
        if student_id in self.students:
            print("Student ID already exists. Try updating the grade instead.")
        else:
            self.students[student_id] = Student(name, student_id, grade)
            print(f"Student '{name}' added successfully.")
```

```
    def display_students(self):
        """Displays all students and their details."""
        if not self.students:
            print("No student records available.")
        else:
            print("\nStudent Records:")
            for student in self.students.values():
                print(student)

    def update_grade(self, student_id, new_grade):
        """Updates the grade of a specific student."""
        if student_id in self.students:
            self.students[student_id].update_grade(new_grade)
            print(f"Grade updated for Student ID {student_id}.")
        else:
            print("Student ID not found.")

    def calculate_average_grade(self):
        """Calculates and displays the average grade of all students."""
        if not self.students:
            print("No students available to calculate the average grade.")
            return

        total_grades = sum(student.grade for student in self.students.values())
        average_grade = total_grades / len(self.students)
        print(f"Average Grade: {average_grade:.2f}")

    def identify_top_and_bottom_performers(self):
        """Displays the top and bottom performers based on grades."""
        if not self.students:
            print("No students available for analysis.")
            return

        top_student = max(self.students.values(), key=lambda s: s.grade)
        bottom_student = min(self.students.values(), key=lambda s: s.grade)

        print("\nTop Performer:")
        print(top_student)
        print("\nBottom Performer:")
        print(bottom_student)
```

```python
def main():
    """Runs the main menu-driven program for managing student grades."""
    system = GradeManagementSystem()

    while True:
        print("\nStudent Grade Management System")
        print("1. Add Student")
        print("2. Display All Students")
        print("3. Update Student Grade")
        print("4. Calculate Average Grade")
        print("5. Identify Top & Bottom Performers")
        print("6. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            name = input("Enter student name: ")
            student_id = input("Enter student ID: ")
            grade = float(input("Enter grade: "))
            system.add_student(name, student_id, grade)

        elif choice == "2":
            system.display_students()

        elif choice == "3":
            student_id = input("Enter student ID to update grade: ")
            new_grade = float(input("Enter new grade: "))
            system.update_grade(student_id, new_grade)

        elif choice == "4":
            system.calculate_average_grade()

        elif choice == "5":
            system.identify_top_and_bottom_performers()

        elif choice == "6":
            print("Exiting the program. Goodbye!")
            break

        else:
            print("Invalid choice. Please try again.")
```

```
if __name__ == "__main__":
    main()
```

Features:

1. `Student` **Class:**
   - Stores student details (`name` , `student_id` , `grade`).
   - Allows updating a student's grade.
2. `GradeManagementSystem` **Class:**
   - Manages student records (add, display, update grades).
   - Calculates the average grade.
   - Identifies the top and bottom-performing students.
3. **Main Program (User Interface):**
   - Provides a menu for teachers to manage student records.

How It Works:

- Run the program and use the menu to add students, update grades, and analyze performance.
- The program ensures efficient student grade management using OOP principles.

This system is a simple and effective way to manage student grades in a school! 📚🏫🚀

Here is a **Restaurant Order Management System** implemented in Python using **OOP principles**:

python

```
class MenuItem:
    """Represents a menu item with a name, price, category, and total sales
count."""

    def __init__(self, name, price, category):
        self.name = name
        self.price = price
        self.category = category
        self.sales_count = 0  # Tracks how many times the item has been ordered
```

```
    def update_sales(self, quantity):
        """Updates the sales count of the menu item."""
        self.sales_count += quantity

    def __str__(self):
        return f"{self.name} - ${self.price:.2f} ({self.category})"


class Restaurant:
    """Manages the restaurant menu, orders, and sales data."""

    def __init__(self):
        self.menu = {}  # Stores menu items
        self.orders = []  # Stores order history

    def add_menu_item(self, name, price, category):
        """Adds a menu item to the restaurant's menu."""
        if name in self.menu:
            print(f"'{name}' already exists in the menu.")
        else:
            self.menu[name] = MenuItem(name, price, category)
            print(f"'{name}' added to the menu successfully.")

    def display_menu(self):
        """Displays all available menu items."""
        if not self.menu:
            print("The menu is currently empty.")
        else:
            print("\nRestaurant Menu:")
            for item in self.menu.values():
                print(item)

    def check_item_availability(self, name):
        """Checks if an item is available on the menu."""
        return name in self.menu

    def place_order(self, name, quantity):
        """Places an order for a menu item if available."""
        if self.check_item_availability(name):
            item = self.menu[name]
            total_cost = item.price * quantity
```

```
            item.update_sales(quantity)  # Update sales count
            self.orders.append((name, quantity, total_cost))
            print(f"Order placed: {quantity} x {name} = ${total_cost:.2f}")
        else:
            print(f"Sorry, '{name}' is not available on the menu.")

    def calculate_total_sales(self):
        """Calculates the total revenue from all orders."""
        total_revenue = sum(order[2] for order in self.orders)
        print(f"\nTotal Revenue: ${total_revenue:.2f}")

    def sales_report(self):
        """Displays the most and least ordered menu items."""
        if not self.menu:
            print("No sales data available.")
            return

        most_ordered = max(self.menu.values(), key=lambda item: item.sales_count,
default=None)
        least_ordered = min(self.menu.values(), key=lambda item: item.sales_count,
default=None)

        print("\nSales Report:")
        if most_ordered:
            print(f"Most Ordered Item: {most_ordered.name} (Ordered
{most_ordered.sales_count} times)")
        if least_ordered:
            print(f"Least Ordered Item: {least_ordered.name} (Ordered
{least_ordered.sales_count} times)")


def main():
    """Runs the main menu-driven program for restaurant order management."""
    restaurant = Restaurant()

    while True:
        print("\nRestaurant Order Management System")
        print("1. Add Menu Item")
        print("2. Display Menu")
        print("3. Place Order")
        print("4. Check Item Availability")
        print("5. Sales Report")
```

```
        print("6. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            name = input("Enter item name: ")
            price = float(input("Enter item price: "))
            category = input("Enter item category (Appetizer/Main Course/Dessert):
")
            restaurant.add_menu_item(name, price, category)

        elif choice == "2":
            restaurant.display_menu()

        elif choice == "3":
            name = input("Enter item name to order: ")
            quantity = int(input("Enter quantity: "))
            restaurant.place_order(name, quantity)

        elif choice == "4":
            name = input("Enter item name to check availability: ")
            if restaurant.check_item_availability(name):
                print(f"'{name}' is available on the menu.")
            else:
                print(f"'{name}' is NOT available on the menu.")

        elif choice == "5":
            restaurant.sales_report()

        elif choice == "6":
            print("Exiting the program. Thank you!")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Features:

1. `MenuItem` **Class**
   - Stores item details (`name` , `price` , `category` , `sales_count`).
   - Tracks the total number of times an item has been ordered.
2. `Restaurant` **Class**
   - Manages menu items.
   - Places and records customer orders.
   - Tracks total sales and identifies the most/least ordered items.
3. **Main Program (User Interface)**
   - Provides a menu-driven system for restaurant managers and customers.

How It Works:

- **Add Menu Items** → Manager adds new items with name, price, and category.
- **Display Menu** → Shows all available menu items.
- **Place Orders** → Customers order items and the total bill is calculated.
- **Check Availability** → Ensures the item exists before processing an order.
- **Sales Report** → Identifies the most and least ordered menu items.

Example Usage:

mathematica

```
Restaurant Order Management System
1. Add Menu Item
2. Display Menu
3. Place Order
4. Check Item Availability
5. Sales Report
6. Exit
```

```
Enter your choice: 1
Enter item name: Pizza
Enter item price: 12.99
Enter item category (Appetizer/Main Course/Dessert): Main Course
'Pizza' added to the menu successfully.

Enter your choice: 3
Enter item name to order: Pizza
Enter quantity: 2
Order placed: 2 x Pizza = $25.98
```

This system provides an efficient way to manage a restaurant's menu and customer orders using **OOP principles**! 🍕🍽️🚀

Here is a **Restaurant Order Management System** implemented in **C++** using **OOP principles**:

cpp

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

// Class representing a menu item
class MenuItem {
public:
    string name;
    double price;
    string category;
    int sales_count; // Tracks how many times the item has been ordered

    MenuItem(string name, double price, string category)
        : name(name), price(price), category(category), sales_count(0) {}
};

// Class representing the restaurant system
class Restaurant {
private:
    vector<MenuItem> menu;
    vector<pair<string, int>> orders; // Stores order history (item name, quantity)
```

```
public:
    // Function to add a menu item
    void addMenuItem(string name, double price, string category) {
        menu.push_back(MenuItem(name, price, category));
        cout << "Item '" << name << "' added to the menu.\n";
    }

    // Function to display the menu
    void displayMenu() {
        if (menu.empty()) {
            cout << "The menu is currently empty.\n";
            return;
        }
        cout << "\nRestaurant Menu:\n";
        for (const auto &item : menu) {
            cout << item.name << " - $" << item.price << " (" << item.category <<
")\n";
        }
    }

    // Function to check if an item is available
    int findMenuItem(string name) {
        for (size_t i = 0; i < menu.size(); ++i) {
            if (menu[i].name == name) {
                return i;
            }
        }
        return -1;
    }

    // Function to place an order
    void placeOrder(string name, int quantity) {
        int index = findMenuItem(name);
        if (index != -1) {
            double total_cost = menu[index].price * quantity;
            menu[index].sales_count += quantity;
            orders.push_back({name, quantity});
            cout << "Order placed: " << quantity << " x " << name << " = $" <<
total_cost << "\n";
        } else {
            cout << "Sorry, '" << name << "' is not available on the menu.\n";
```

```
        }
    }

    // Function to display sales report
    void salesReport() {
        if (menu.empty()) {
            cout << "No sales data available.\n";
            return;
        }

        MenuItem *mostOrdered = nullptr;
        MenuItem *leastOrdered = nullptr;

        for (auto &item : menu) {
            if (mostOrdered == nullptr || item.sales_count > mostOrdered-
>sales_count) {
                mostOrdered = &item;
            }
            if (leastOrdered == nullptr || item.sales_count < leastOrdered-
>sales_count) {
                leastOrdered = &item;
            }
        }

        cout << "\nSales Report:\n";
        if (mostOrdered && mostOrdered->sales_count > 0) {
            cout << "Most Ordered Item: " << mostOrdered->name << " (Ordered " <<
mostOrdered->sales_count << " times)\n";
        }
        if (leastOrdered && leastOrdered->sales_count > 0) {
            cout << "Least Ordered Item: " << leastOrdered->name << " (Ordered " <<
leastOrdered->sales_count << " times)\n";
        }
    }
};

// Main function
int main() {
    Restaurant restaurant;
    int choice;

    do {
```

```
        cout << "\nRestaurant Order Management System\n";
        cout << "1. Add Menu Item\n";
        cout << "2. Display Menu\n";
        cout << "3. Place Order\n";
        cout << "4. Sales Report\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Clear input buffer

        if (choice == 1) {
            string name, category;
            double price;
            cout << "Enter item name: ";
            getline(cin, name);
            cout << "Enter item price: ";
            cin >> price;
            cin.ignore();
            cout << "Enter item category (Appetizer/Main Course/Dessert): ";
            getline(cin, category);
            restaurant.addMenuItem(name, price, category);
        }
        else if (choice == 2) {
            restaurant.displayMenu();
        }
        else if (choice == 3) {
            string name;
            int quantity;
            cout << "Enter item name to order: ";
            getline(cin, name);
            cout << "Enter quantity: ";
            cin >> quantity;
            restaurant.placeOrder(name, quantity);
        }
        else if (choice == 4) {
            restaurant.salesReport();
        }
        else if (choice == 5) {
            cout << "Exiting the program. Thank you!\n";
        }
        else {
            cout << "Invalid choice. Please try again.\n";
```

```
        }

    } while (choice != 5);

    return 0;
}
```

Features:

1. `MenuItem` **Class**
   - Stores item details (`name` , `price` , `category` , `sales_count`).
   - Tracks the total number of times an item has been ordered.
2. `Restaurant` **Class**
   - Manages menu items.
   - Places and records customer orders.
   - Tracks sales data and identifies the most/least ordered items.
3. **Main Program (User Interface)**
   - Provides a menu-driven system for restaurant managers and customers.

How It Works:

- **Add Menu Items** → Manager adds new items with name, price, and category.
- **Display Menu** → Shows all available menu items.
- **Place Orders** → Customers order items and the total bill is calculated.
- **Sales Report** → Identifies the most and least ordered menu items.

Example Usage:

```mathematica
Restaurant Order Management System
1. Add Menu Item
2. Display Menu
3. Place Order
4. Sales Report
5. Exit
Enter your choice: 1
Enter item name: Burger
Enter item price: 5.99
Enter item category (Appetizer/Main Course/Dessert): Main Course
Item 'Burger' added to the menu.

Enter your choice: 3
Enter item name to order: Burger
Enter quantity: 2
Order placed: 2 x Burger = $11.98

Enter your choice: 4
Sales Report:
Most Ordered Item: Burger (Ordered 2 times)

Enter your choice: 5
Exiting the program. Thank you!
```

This **C++ program** effectively manages a restaurant's menu, orders, and sales data using **OOP principles**! 🍽️🚀

Here is a **Supermarket Inventory System** implemented in **C++** using **OOP principles**:

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

// Class representing an item in the inventory
class Item {
public:
```

```
    string name;
    double price;
    int quantity;
    int sold_quantity;

    Item(string name, double price, int quantity)
        : name(name), price(price), quantity(quantity), sold_quantity(0) {}

    double calculateProfit() const {
        return sold_quantity * price;
    }
};

// Class representing the supermarket inventory
class Supermarket {
private:
    vector<Item> inventory;

public:
    // Function to add an item to the inventory
    void addItem(string name, double price, int quantity) {
        inventory.push_back(Item(name, price, quantity));
        cout << "Item '" << name << "' added to inventory.\n";
    }

    // Function to display all available items
    void displayItems() {
        if (inventory.empty()) {
            cout << "No items available in inventory.\n";
            return;
        }
        cout << "\nAvailable Items:\n";
        for (const auto &item : inventory) {
            cout << item.name << " - $" << item.price << " (Quantity: " <<
item.quantity << ")\n";
        }
    }

    // Function to check if an item is available in stock
    int findItem(string name) {
        for (size_t i = 0; i < inventory.size(); ++i) {
            if (inventory[i].name == name) {
```

```
                return i;
            }
        }
        return -1;
    }

    // Function to process a customer purchase
    void purchaseItem(string name, int quantity) {
        int index = findItem(name);
        if (index != -1) {
            Item &item = inventory[index];

            if (item.quantity >= quantity) {
                double total_cost = item.price * quantity;
                item.quantity -= quantity;
                item.sold_quantity += quantity;
                cout << "Purchase successful! Total cost: $" << total_cost << "\n";
            } else {
                cout << "Sorry, not enough stock available for '" << name << "'.\n";
            }
        } else {
            cout << "Item '" << name << "' not found in inventory.\n";
        }
    }

    // Function to check stock availability
    void checkStock(string name) {
        int index = findItem(name);
        if (index != -1) {
            Item &item = inventory[index];
            if (item.quantity > 0) {
                cout << name << " is in stock. Available quantity: " <<
item.quantity << "\n";
            } else {
                cout << name << " is out of stock.\n";
            }
        } else {
            cout << "Item '" << name << "' not found in inventory.\n";
        }
    }

    // Function to perform sales analysis
```

```
    void salesAnalysis() {
        if (inventory.empty()) {
            cout << "No sales data available.\n";
            return;
        }

        Item *maxProfitItem = nullptr;
        Item *minProfitItem = nullptr;

        for (auto &item : inventory) {
            double profit = item.calculateProfit();
            if (maxProfitItem == nullptr || profit > maxProfitItem-
>calculateProfit()) {
                maxProfitItem = &item;
            }
            if (minProfitItem == nullptr || profit < minProfitItem-
>calculateProfit()) {
                minProfitItem = &item;
            }
        }

        cout << "\nSales Analysis:\n";
        if (maxProfitItem) {
            cout << "Item with maximum profit: " << maxProfitItem->name
                 << " (Profit: $" << maxProfitItem->calculateProfit() << ")\n";
        }
        if (minProfitItem) {
            cout << "Item with minimum profit: " << minProfitItem->name
                 << " (Profit: $" << minProfitItem->calculateProfit() << ")\n";
        }
    }
};

// Main function
int main() {
    Supermarket supermarket;
    int choice;

    do {
        cout << "\nSupermarket Inventory System\n";
        cout << "1. Add Item to Inventory\n";
        cout << "2. Display Available Items\n";
```

```
        cout << "3. Purchase Item\n";
        cout << "4. Check Stock Availability\n";
        cout << "5. Sales Analysis\n";
        cout << "6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Clear input buffer

        if (choice == 1) {
            string name;
            double price;
            int quantity;
            cout << "Enter item name: ";
            getline(cin, name);
            cout << "Enter item price: ";
            cin >> price;
            cout << "Enter item quantity: ";
            cin >> quantity;
            supermarket.addItem(name, price, quantity);
        }
        else if (choice == 2) {
            supermarket.displayItems();
        }
        else if (choice == 3) {
            string name;
            int quantity;
            cout << "Enter item name to purchase: ";
            getline(cin, name);
            cout << "Enter quantity to purchase: ";
            cin >> quantity;
            supermarket.purchaseItem(name, quantity);
        }
        else if (choice == 4) {
            string name;
            cout << "Enter item name to check stock: ";
            getline(cin, name);
            supermarket.checkStock(name);
        }
        else if (choice == 5) {
            supermarket.salesAnalysis();
        }
        else if (choice == 6) {
```

```
            cout << "Exiting the program. Thank you!\n";
        }
        else {
            cout << "Invalid choice. Please try again.\n";
        }

    } while (choice != 6);

    return 0;
}
```

Features

1. `Item` **Class**
   - Stores details like `name`, `price`, `quantity`, and `sold_quantity`.
   - Provides a method to calculate total profit for each item.
2. `Supermarket` **Class**
   - Manages the **inventory** of items.
   - Handles **purchases** and updates stock accordingly.
   - **Tracks sales** and identifies the most/least profitable items.
3. **Main Program (User Interface)**
   - **Menu-driven system** for the supermarket owner and customers.

How It Works

- **Add Items to Inventory** → Owner adds items with name, price, and quantity.
- **Display Available Items** → Shows all products with prices and stock levels.
- **Customer Purchases** → Customers specify an item and quantity to buy.
- **Stock Check** → Owner or customers can check stock availability.
- **Sales Analysis** → Identifies the most and least profitable items.

Example Usage

```mathematica
Supermarket Inventory System
1. Add Item to Inventory
2. Display Available Items
3. Purchase Item
4. Check Stock Availability
5. Sales Analysis
6. Exit
Enter your choice: 1
Enter item name: Apple
Enter item price: 1.50
Enter item quantity: 50
Item 'Apple' added to inventory.

Enter your choice: 2
Available Items:
Apple - $1.50 (Quantity: 50)

Enter your choice: 3
Enter item name to purchase: Apple
Enter quantity to purchase: 10
Purchase successful! Total cost: $15.00

Enter your choice: 4
Enter item name to check stock: Apple
Apple is in stock. Available quantity: 40

Enter your choice: 5
Sales Analysis:
Item with maximum profit: Apple (Profit: $15.00)
Item with minimum profit: Apple (Profit: $15.00)

Enter your choice: 6
Exiting the program. Thank you!
```

This **C++ program** efficiently manages a **supermarket's inventory, sales, and stock levels** using **OOP principles**! 🛒🚀

Here is a **C++ program** that defines a `Date` class to handle **date operations**, including:

1. **Reading a date** from the user.
2. **Printing the date** in multiple formats.
3. **Finding the date one week later** using proper **date arithmetic**.

C++ Implementation

cpp

```cpp
#include <iostream>
using namespace std;

// Number of days in each month (for non-leap years)
const int daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

class Date {
private:
    int day, month, year;

    // Function to check if a year is a leap year
    bool isLeapYear(int y) {
        return (y % 4 == 0 && y % 100 != 0) || (y % 400 == 0);
    }

    // Function to get the number of days in a given month
    int getDaysInMonth(int m, int y) {
        if (m == 2) // February
            return isLeapYear(y) ? 29 : 28;
        else
            return daysInMonth[m - 1];
    }

public:
    // Constructor to initialize the date
    Date(int d = 1, int m = 1, int y = 2000) : day(d), month(m), year(y) {}

    // Function to read a date from the user
    void readDate() {
```

```
        cout << "Enter day: ";
        cin >> day;
        cout << "Enter month: ";
        cin >> month;
        cout << "Enter year: ";
        cin >> year;

        // Validate the date
        if (month < 1 || month > 12 || day < 1 || day > getDaysInMonth(month, year))
{
            cout << "Invalid date entered. Please enter a valid date.\n";
            readDate(); // Recursively ask for a valid date
        }
    }

    // Function to print the date in different formats
    void printDateFormats() {
        cout << "\nDate Formats:\n";
        cout << "1. DD-MM-YYYY: " << day << "-" << month << "-" << year << endl;
        cout << "2. MM/DD/YYYY: " << month << "/" << day << "/" << year << endl;
        cout << "3. YYYY.MM.DD: " << year << "." << month << "." << day << endl;
    }

    // Function to find the date exactly one week later
    void findNextWeekDate() {
        int newDay = day + 7;
        int newMonth = month;
        int newYear = year;

        int daysInCurrentMonth = getDaysInMonth(newMonth, newYear);

        // Adjust the date if it exceeds the number of days in the month
        if (newDay > daysInCurrentMonth) {
            newDay -= daysInCurrentMonth;
            newMonth++;

            // If the month exceeds December, adjust the year
            if (newMonth > 12) {
                newMonth = 1;
                newYear++;
            }
        }
```

```
        cout << "\nDate one week later: " << newDay << "-" << newMonth << "-" <<
newYear << endl;
    }
};

// Main function
int main() {
    Date myDate;

    // Read and process the date
    myDate.readDate();
    myDate.printDateFormats();
    myDate.findNextWeekDate();

    return 0;
}
```

Explanation

1. **Date Class**:
   - Stores `day`, `month`, and `year`.
   - Includes a constructor to initialize the date.
   - Provides methods for:
     - **Reading** a valid date.
     - **Printing** the date in multiple formats.
     - **Calculating** the date **one week later**.
2. **Leap Year & Days in Month Handling**:
   - `isLeapYear()` determines if a year is a **leap year**.
   - `getDaysInMonth()` returns the **correct** number of days for a given month, including leap year considerations.
3. **Finding the Date One Week Later**:
   - Adds **7 days** to the current date.

- If the new day exceeds the number of days in the month, it:
  - Adjusts the **month**.
  - If necessary, **increments the year**.

Example Run

Input:

```yaml

Enter day: 30
Enter month: 10
Enter year: 2013
```

Output:

```yaml

Date Formats:
1. DD-MM-YYYY: 30-10-2013
2. MM/DD/YYYY: 10/30/2013
3. YYYY.MM.DD: 2013.10.30

Date one week later: 6-11-2013
```

Edge Cases Handled

✅ **End of Month Handling:**

- If today is **25th February 2024**, one week later should be **3rd March 2024**.
  ✅ **Leap Year Considerations:**
- If today is **23rd February 2024**, one week later should be **1st March 2024**.
  ✅ **Year Change Handling:**
- If today is **29th December 2023**, one week later should be **5th January 2024**.

Why is this a Good OOP Design?

✔ **Encapsulation:** The `Date` class encapsulates all date-related logic.
✔ **Modularity:** Functions are well-defined and **self-contained**.
✔ **Reusability:** Can be expanded for further date manipulations.

This **C++ program** efficiently handles **date operations** using **OOP principles**! 🎯

Here is a **C++ program** that demonstrates **Object-Oriented Programming (OOP)** principles by implementing a **Complex Number** class. The program includes functionalities for:

1. **Adding two complex numbers**
2. **Subtracting two complex numbers**
3. **Multiplying two complex numbers**
4. **Displaying a complex number**

C++ Implementation

cpp

```cpp
#include <iostream>
using namespace std;

class Complex {
private:
    double real, imag;

public:
    // Constructor to initialize complex number
    Complex(double r = 0, double i = 0) {
        real = r;
        imag = i;
    }
```

```
    // Function to display a complex number
    void display() const {
        if (imag >= 0)
            cout << real << " + " << imag << "i" << endl;
        else
            cout << real << " - " << -imag << "i" << endl;
    }

    // Operator overloading for addition
    Complex operator+(const Complex &c) const {
        return Complex(real + c.real, imag + c.imag);
    }

    // Operator overloading for subtraction
    Complex operator-(const Complex &c) const {
        return Complex(real - c.real, imag - c.imag);
    }

    // Operator overloading for multiplication
    Complex operator*(const Complex &c) const {
        return Complex((real * c.real - imag * c.imag), (real * c.imag + imag *
c.real));
    }

    // Friend function to overload input (>>) and output (<<) operators
    friend istream &operator>>(istream &input, Complex &c);
    friend ostream &operator<<(ostream &output, const Complex &c);
};

// Overloading >> operator to input complex number
istream &operator>>(istream &input, Complex &c) {
    cout << "Enter real part: ";
    input >> c.real;
    cout << "Enter imaginary part: ";
    input >> c.imag;
    return input;
}

// Overloading << operator to output complex number
ostream &operator<<(ostream &output, const Complex &c) {
    if (c.imag >= 0)
```

```
        output << c.real << " + " << c.imag << "i";
    else
        output << c.real << " - " << -c.imag << "i";
    return output;
}

// Main function
int main() {
    Complex num1, num2, result;

    // Taking input
    cout << "Enter first complex number:\n";
    cin >> num1;

    cout << "Enter second complex number:\n";
    cin >> num2;

    // Performing operations
    result = num1 + num2;
    cout << "\nSum: " << result << endl;

    result = num1 - num2;
    cout << "Difference: " << result << endl;

    result = num1 * num2;
    cout << "Product: " << result << endl;

    return 0;
}
```

Explanation

1. **Encapsulation**:
   - The `Complex` class encapsulates real and imaginary parts as private members.
   - Public methods allow controlled access.
2. **Constructor**:
   - Initializes a complex number (default: `0 + 0i`).

3. **Operator Overloading**:
    - `+` for addition of complex numbers.
    - `-` for subtraction of complex numbers.
    - `*` for multiplication of complex numbers.
    - `>>` and `<<` overloaded for input and output.
4. **Friend Functions**:
    - `>>` and `<<` are overloaded to handle user-friendly input/output.

Example Run

Input:

```yaml

Enter first complex number:
Enter real part: 3
Enter imaginary part: 4
Enter second complex number:
Enter real part: 1
Enter imaginary part: 2
```

Output:

```go

Sum: 4 + 6i
Difference: 2 + 2i
Product: -5 + 10i
```

Why is this a Good OOP Design?

✔ **Encapsulation:** Data members are private.
✔ **Operator Overloading:** Makes operations intuitive (`+` , `-` , `*`).

✔ **Friend Functions:** Improves usability of I/O operations.
✔ **Reusability:** The class can be easily expanded for more operations.

This **C++ program** efficiently demonstrates **complex number operations** using **OOP principles**! 🎯

Here is a **Python program** that demonstrates **Object-Oriented Programming (OOP)** principles by implementing a **Complex Number** class. The program includes functionalities for:

1. **Adding two complex numbers**
2. **Subtracting two complex numbers**
3. **Multiplying two complex numbers**
4. **Displaying a complex number**

Python Implementation

python

```python
class Complex:
    def __init__(self, real=0, imag=0):
        """Constructor to initialize complex numbers"""
        self.real = real
        self.imag = imag

    def __str__(self):
        """String representation for complex number"""
        if self.imag >= 0:
            return f"{self.real} + {self.imag}i"
        else:
            return f"{self.real} - {-self.imag}i"

    def __add__(self, other):
        """Overloading + operator for complex number addition"""
        return Complex(self.real + other.real, self.imag + other.imag)

    def __sub__(self, other):
        """Overloading - operator for complex number subtraction"""
```

```
        return Complex(self.real - other.real, self.imag - other.imag)

    def __mul__(self, other):
        """Overloading * operator for complex number multiplication"""
        real_part = self.real * other.real - self.imag * other.imag
        imag_part = self.real * other.imag + self.imag * other.real
        return Complex(real_part, imag_part)

    @staticmethod
    def from_input():
        """Static method to take input for a complex number"""
        real = float(input("Enter real part: "))
        imag = float(input("Enter imaginary part: "))
        return Complex(real, imag)


# Main program
if __name__ == "__main__":
    print("Enter first complex number:")
    num1 = Complex.from_input()

    print("Enter second complex number:")
    num2 = Complex.from_input()

    # Performing operations
    print("\nSum:", num1 + num2)
    print("Difference:", num1 - num2)
    print("Product:", num1 * num2)
```

Explanation

1. **Encapsulation**:
   - The `Complex` class encapsulates `real` and `imag` as instance variables.
2. **Constructor (`__init__`)**:
   - Initializes a complex number (default: `0 + 0i`).
3. **Operator Overloading (`+`, `-`, `*`)**:

- `__add__` for addition.
- `__sub__` for subtraction.
- `__mul__` for multiplication.

4. **Static Method (`from_input`):**
   - Provides a way to take user input for a complex number.
5. `__str__` **Method**:
   - Formats the complex number nicely when printed.

Example Run

Input:

```yaml

Enter first complex number:
Enter real part: 3
Enter imaginary part: 4
Enter second complex number:
Enter real part: 1
Enter imaginary part: 2
```

Output:

```go

Sum: 4.0 + 6.0i
Difference: 2.0 + 2.0i
Product: -5.0 + 10.0i
```

Why is this a Good OOP Design?

✔ **Encapsulation:** Data members are private to the instance.
✔ **Operator Overloading:** Makes operations intuitive (`+`, `-`, `*`).

✔ **Static Method:** Useful for structured user input.
✔ **Reusability:** The class can be expanded for more operations (e.g., division, conjugate).

This **Python program** efficiently demonstrates **complex number operations** using **OOP principles**! 🚀

Here is a **Python program** that implements an **Online Shopping System** using **Object-Oriented Programming (OOP) principles**.

Features of the System:

✔ **Customers can browse products and add them to a cart**
✔ **Products have attributes like name, price, and quantity**
✔ **System calculates the total bill and applies discounts**
✔ **System tracks inventory and prevents orders if a product is out of stock**

Python Implementation

python

```python
class Product:
    """Represents a product with name, price, and available stock."""
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def __str__(self):
        return f"{self.name} - ${self.price:.2f} (Stock: {self.quantity})"


class Inventory:
    """Manages product inventory."""
    def __init__(self):
        self.products = {}
```

```
    def add_product(self, name, price, quantity):
        """Adds a product to inventory or updates existing product stock."""
        if name in self.products:
            self.products[name].quantity += quantity
        else:
            self.products[name] = Product(name, price, quantity)

    def display_products(self):
        """Displays all available products."""
        if not self.products:
            print("No products available.")
        else:
            for product in self.products.values():
                print(product)

    def check_availability(self, name, quantity):
        """Checks if the product is available in sufficient quantity."""
        return name in self.products and self.products[name].quantity >= quantity

    def reduce_stock(self, name, quantity):
        """Reduces stock when a product is purchased."""
        if self.check_availability(name, quantity):
            self.products[name].quantity -= quantity
            return True
        return False


class Cart:
    """Represents a shopping cart."""
    def __init__(self):
        self.items = {}

    def add_to_cart(self, product, quantity):
        """Adds a product to the cart."""
        if product.name in self.items:
            self.items[product.name]['quantity'] += quantity
        else:
            self.items[product.name] = {'product': product, 'quantity': quantity}

    def display_cart(self):
        """Displays the cart contents."""
        if not self.items:
```

```
            print("Cart is empty.")
        else:
            print("Shopping Cart:")
            for item in self.items.values():
                product = item['product']
                quantity = item['quantity']
                print(f"{product.name} - ${product.price:.2f} x {quantity} =
${product.price * quantity:.2f}")

    def calculate_total(self):
        """Calculates the total bill."""
        return sum(item['product'].price * item['quantity'] for item in
self.items.values())

    def apply_discount(self, discount_percent):
        """Applies a discount to the total bill."""
        total = self.calculate_total()
        discount_amount = (discount_percent / 100) * total
        return total - discount_amount


class Customer:
    """Represents a customer."""
    def __init__(self, name, inventory):
        self.name = name
        self.cart = Cart()
        self.inventory = inventory

    def browse_products(self):
        """Displays available products."""
        print("\nAvailable Products:")
        self.inventory.display_products()

    def add_to_cart(self, product_name, quantity):
        """Adds a product to the customer's cart if available."""
        if self.inventory.check_availability(product_name, quantity):
            product = self.inventory.products[product_name]
            self.cart.add_to_cart(product, quantity)
            print(f"Added {quantity} x {product.name} to cart.")
        else:
            print(f"Sorry, {product_name} is out of stock or insufficient
quantity.")
```

```
    def place_order(self, discount_percent=0):
        """Processes the order and applies discount."""
        if not self.cart.items:
            print("Cart is empty. Add items before placing an order.")
            return

        # Check inventory before processing
        for item in self.cart.items.values():
            product_name = item['product'].name
            quantity = item['quantity']
            if not self.inventory.check_availability(product_name, quantity):
                print(f"Cannot process order. {product_name} is out of stock.")
                return

        # Deduct stock
        for item in self.cart.items.values():
            self.inventory.reduce_stock(item['product'].name, item['quantity'])

        # Calculate final price
        total = self.cart.apply_discount(discount_percent)
        print("\nOrder Summary:")
        self.cart.display_cart()
        print(f"Total after {discount_percent}% discount: ${total:.2f}")
        print(f"Order placed successfully, {self.name}!\n")


# Main Program
if __name__ == "__main__":
    inventory = Inventory()

    # Adding products to inventory
    inventory.add_product("Laptop", 1000, 5)
    inventory.add_product("Smartphone", 500, 10)
    inventory.add_product("Headphones", 100, 15)
    inventory.add_product("Mouse", 50, 20)

    customer = Customer("Alice", inventory)

    while True:
        print("\n1. Browse Products\n2. Add to Cart\n3. View Cart\n4. Place
Order\n5. Exit")
```

```
        choice = input("Enter your choice: ")

        if choice == "1":
            customer.browse_products()

        elif choice == "2":
            product_name = input("Enter product name: ")
            quantity = int(input("Enter quantity: "))
            customer.add_to_cart(product_name, quantity)

        elif choice == "3":
            customer.cart.display_cart()

        elif choice == "4":
            discount = float(input("Enter discount percentage (0 for no discount):
"))
            customer.place_order(discount)

        elif choice == "5":
            print("Thank you for shopping with us!")
            break

        else:
            print("Invalid choice. Please try again.")
```

Explanation of the Classes

1. Product Class

- Represents a product with attributes **name, price, and quantity**.
- Implements `__str__()` to display the product details.

2. Inventory Class

- Stores and manages all products.
- Provides functions to **add, display, check stock, and update stock**.

3. Cart Class

- Stores selected products and their quantities.
- Calculates the **total bill** and applies **discounts**.

4. Customer Class

- Can **browse products, add items to cart, and place orders**.
- Ensures that orders cannot be placed for out-of-stock products.

Example Run

mathematica

```
1. Browse Products
2. Add to Cart
3. View Cart
4. Place Order
5. Exit
Enter your choice: 1

Available Products:
Laptop - $1000.00 (Stock: 5)
Smartphone - $500.00 (Stock: 10)
Headphones - $100.00 (Stock: 15)
Mouse - $50.00 (Stock: 20)

Enter your choice: 2
Enter product name: Laptop
Enter quantity: 2
Added 2 x Laptop to cart.

Enter your choice: 3
Shopping Cart:
Laptop - $1000.00 x 2 = $2000.00

Enter your choice: 4
Enter discount percentage (0 for no discount): 10

Order Summary:
```

```
Laptop - $1000.00 x 2 = $2000.00
Total after 10% discount: $1800.00
Order placed successfully, Alice!
```

Why is this a Good OOP Design?

✔ **Encapsulation** – Data is stored within classes with controlled access.
✔ **Separation of Concerns** – Each class has a well-defined responsibility.
✔ **Reusability** – The system can be easily extended (e.g., adding more features).
✔ **Scalability** – Supports multiple customers and products.

This **Python program** efficiently models an **Online Shopping System** using **OOP principles**! 🚀