Python Shop Simulation Building a Complete Management Game

This four-session project guides participants through building a functional shop management simulation in Python. The system will manage finances, inventory, customer interactions, and equipment upgrades. This project serves as a practical application of Python programming concepts.

The project offers two paths. All participants will complete a functional simulation utilizing core Python concepts including variables, functions, loops, and dictionaries. An optional Extended path introduces object-oriented programming (OOP), architectural patterns, and software engineering principles. Both paths result in a complete shop simulation, with the Extended version focusing on scalable system design.

The project is structured across four sessions:

Session 1: Foundation. Establish core mechanics including money, inventory management, and pricing formulas.

Session 2: Operations. Implement the passage of time, customer generation, and transaction processing.

Session 3: Expansion. Integrate equipment upgrades, automation features, and data analytics.

Session 4: Completion. Add data persistence via file saving and loading, implement statistics, and complete the application.

Upon completion, participants will have developed a management game featuring inventory management, customer service, equipment upgrades, and save/load functionality. This project provides experience in building an application from core programming principles.

Session 1 Overview: Foundation

Core Mechanics

Session 1 establishes the core components of the shop simulation. You will create the state variables for money, inventory, purchasing prices, and the shop manager's level, which influences pricing strategy.

Functions developed in this session will be called repeatedly throughout the simulation's lifetime. The shop state should be designed to handle multiple transactions and allow for modification.

For Extended learners, this session introduces encapsulation—grouping related data and behaviors into an object. This includes how a shop object manages its own state and performs actions, preparing for advanced design patterns.

01

State Management

Create variables for money, inventory, and pricing that persist throughout the simulation

02

Display Functions

Build interfaces for showing current shop status

03

Business Logic

Implement the pricing formula and purchasing mechanics

04

Integration

Combine all components into a unified status display

Task 1: The Money Variable

Normal Version

Create a single variable to track your shop's current funds. Choose a starting amount, such as 30 or 50 units. This number will fluctuate as inventory is purchased and sales are made.

Extended Version

Integrate the money value as an attribute within a Shop class. Determine how this value should be accessed and modified through methods rather than direct manipulation.

The Mathematics of Spending

Every purchase follows a simple formula: **new_money = current_money - cost**. This equation will appear throughout your simulation, sometimes in complex combinations, but always following this basic principle of deduction.

Without money, inventory cannot be purchased. Without inventory, sales cannot be made. Without sales, money cannot be earned. Your implementation must make this value easy to read, modify, and display at any moment during the simulation.

For Extended learners, consider encapsulation. All modifications to the shop's money should occur through controlled methods such as **spend(amount)** or **earn(amount)**, rather than direct external manipulation. This decision affects code robustness against bugs and the ease of adding features like transaction logging.

Task 2: Displaying Money

Normal Approach

Write a function that accepts the current money value as a parameter and prints it in a clear format. This may include currency symbols or descriptive text like "Current Funds: $30".

Key Considerations

- What input does the function need?
- Does it return a value or just print?
- How should the output be formatted?

Extended Approach

Implement this functionality as a method of your Shop class, such as **show_money(self)**. The method accesses **self.money** directly without needing a parameter.

Design Questions

- Should the method only print, or also return a formatted string?
- How does this fit into a larger "display all shop info" method?
- What happens if money becomes negative?

Display functions present information. The information should be presented in a way that provides necessary context.

Task 3: Creating the Inventory System

Your shop needs a way to track multiple items and their quantities. Unlike money (a single number), inventory is a collection of related values. The structure chosen will impact the ease of adding new items, updating quantities, and displaying stock levels throughout the application.

Dictionary Approach

Use a dictionary with item names as keys and quantities as values: **{"coffee": 10, "tea": 5, "pastry": 8}**. This provides quick lookup and direct access.

List Approach

Store items as a list of tuples or small dictionaries. This approach is more complex for searching but facilitates ordered iteration.

Object Approach (Extended)

Create an Inventory class with methods to add, remove, and query items. This offers increased flexibility for feature expansion.

Dictionaries are frequently used for inventory due to their efficient access and clear syntax, such as **inventory["coffee"]**. The choice of data structure influences the complexity of operations like iterating through all items, checking item existence, or identifying the most stocked item.

For Extended learners considering a separate Inventory class, define the behaviors inherent to inventory. This includes determining if inventory manages its own display, enforces quantity constraints, or tracks sales data. These design decisions affect the system architecture.

Task 4: Displaying Inventory

Create a function that presents your inventory in a readable format. Consider showing not just item quantities, but also highlighting items that are running low or noting which items are most valuable.

1

Retrieve Data

Access the inventory structure (dictionary, list, or object)

2

Format Output

Create clear, aligned text showing each item and its quantity

3

Display

Print or return the formatted inventory information

Extended Thinking: Separation of Concerns

Consider the separation of display logic from inventory logic. Displaying often involves formatting choices specific to the user interface, which might change even if inventory logic stays the same. Determine whether **show_inventory()** should be a Shop method that queries its inventory attribute, or whether inventory itself should have a **display()** method.

Use alignment, spacing, and labels effectively for output formatting. Instead of printing "coffee 10 tea 5", consider "Coffee: 10 units | Tea: 5 units" or a tabular format for multiple items.

Task 5: Base Prices

Understanding Base Prices

Base prices represent the cost paid to suppliers for stocking your shop. These are different from selling prices, which are charged to customers. The relationship between base price and selling price determines profit margin.

Store base prices in a structure that mirrors your inventory. If a dictionary is used for inventory, use a dictionary for prices. This allows for easy lookup of item prices.

Example Structure

```
base_prices = {
    "coffee": 2,
    "tea": 1.5,
    "pastry": 3,
    "sandwich": 4
}
```

The Cost Formula

When buying stock: **total_cost = base_price × quantity**

This multiplication applies when restocking inventory.

$2

Coffee Base

Low cost, high volume item

$4

Sandwich Base

Premium product with better margins

$1.50

Tea Base

Budget-friendly option

For extended implementations, consider managing prices with a PriceManager class or allowing price fluctuations, discounts, or bulk pricing. These considerations are not required for initial implementation.

Task 6: Manager Level and Pricing Formula

This simulation mechanic establishes the relationship between manager skill and selling price. An experienced manager can charge higher prices without losing customers, reflecting improved service quality, customer relationships, or brand reputation.

1 The Formula

selling_price = base_price × (1 + manager_level × 0.1)

Each manager level adds 10% to selling prices. At level 0, items sell at base price. At level 3, a 30% increase is applied.

2 Example Calculation

If coffee's base price is $2 and the manager is level 3:

selling_price = 2 × (1 + 3 × 0.1) = 2 × 1.3 = $2.60

Purchasing at $2 and selling at $2.60 yields $0.60 profit per cup.

Create a function (Normal) or method (Extended) to implement this calculation. The function must accept the base price and current manager level, returning the computed selling price. This function will be called during each customer purchase, requiring efficiency and accuracy.

For Extended learners, consider the placement of this logic. Options include a Shop method, a Manager class method, or a separate PricingEngine. Adhere to the principle of single responsibility: each class should have one clear purpose. A Manager class might track level and upgrade costs, while actual price calculation is delegated elsewhere. These architectural decisions apply as systems grow.

Task 7: Purchasing Inventory

The buying function must handle user input, validate it against available funds, update multiple pieces of state, and provide clear feedback.

1 **Gather Input**

Ask which item and how many units to purchase

2 **Calculate Cost**

Multiply base price by quantity

3 **Validate Funds**

Check if current money ≥ total cost

4 **Execute Transaction**

Subtract money, add inventory

5 **Confirm**

Display success message or error

The Mathematics of Buying

total_cost = base_price[item] × quantity

new_money = current_money - total_cost

new_inventory[item] = current_inventory[item] + quantity

These three equations form the complete transaction. They are interdependent; money cannot be updated without knowing total cost, and inventory should not update if the money update fails.

Normal Considerations

- Validate item existence.
- Handle negative user input.
- Handle non-numeric input.
- Confirm large purchases.

Extended Considerations

- Consider validation as a separate method.
- Consider method return status (success/failure).
- Implement a transaction log.

This function requires attention to transaction logic. Transactions must be atomic, meaning they either complete fully or not at all. Subtracting money without adding inventory, or vice versa, must be avoided. This simulation teaches state management principles.

Task 8: Unified Status Display

Create a status function that combines all display elements. This function provides the simulation's dashboard for users to understand their current situation. The status display should provide information such as current inventory, available actions, and next steps.

Financial Status

Current money with clear formatting

Inventory Overview

All items and their quantities

Manager Level

Current skill level affecting prices

Context Information

Optionally include time, day, or other status indicators

Consider information hierarchy. Critical information, such as money and day, is typically placed at the top, with supporting details below. Use visual separators (blank lines, dashes, or headers) to group related information. Information should be scannable; key facts should be absorbable at a glance.

For extended implementations, this function can demonstrate object-oriented design principles. Instead of passing multiple parameters to a status function, a Shop object can call **self.show_status()**, which internally accesses all needed attributes. This encapsulation results in cleaner calling code and a more maintainable status method, allowing new status elements to be added without changing the method's signature.

Session 1 Deliverables

Session 1 establishes the foundation of the shop simulation. Verify all components function correctly independently and integrate as specified.

Core State

Money variable, inventory structure, base prices, and manager level initialized and accessible

Display Functions

Individual and unified displays for money, inventory, and overall status

Business Logic

Price calculation formula and purchasing functionality

Normal Path Checklist

- Money stored in a variable
- Inventory dictionary or list created
- Base prices defined
- Manager level variable initialized
- show_money() function works
- show_inventory() function works
- calculate_selling_price() function implements the formula correctly
- buy_items() function handles transactions properly
- show_status() combines all displays

Extended Path Checklist

- Shop class defined with all attributes
- Money, inventory, prices, and level stored as instance attributes
- Display methods implemented (show_money, show_inventory, show_status)
- calculate_selling_price() as a Shop method
- buy_items() as a Shop method
- Clear separation between data and behavior
- Methods use self to access attributes
- Class design anticipates future extensions

Testing Verification

Before proceeding, verify the following: 1) Display initial state, 2) Purchase items successfully, 3) See updated money and inventory, 4) Calculate selling prices for different manager levels, 5) Handle insufficient funds gracefully.