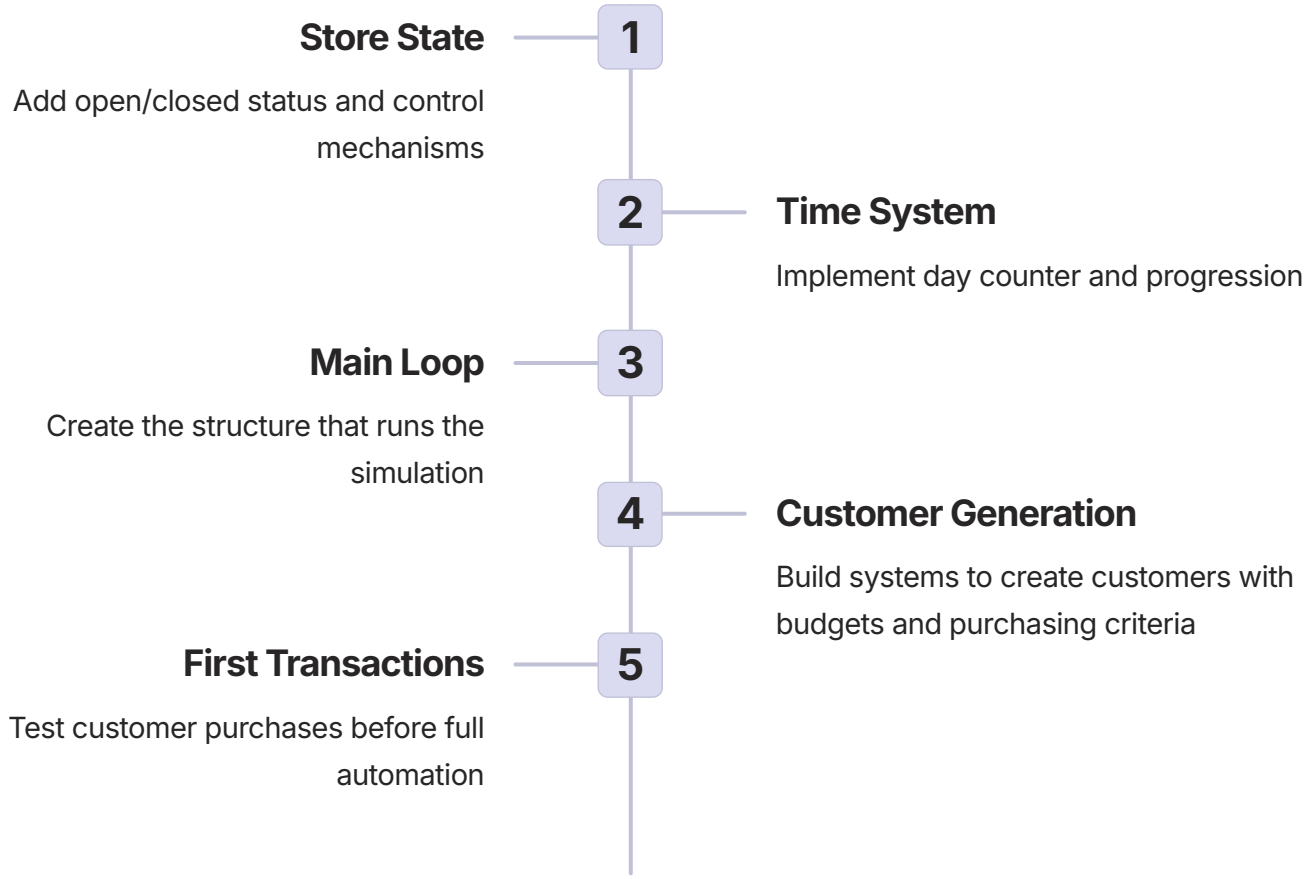
Session 2 Overview: Time and Customer Integration

Session 2 integrates time and customer generation into the shop infrastructure. This involves implementing a day counter, creating a main simulation loop that advances daily, and generating customers with specific budgets and purchasing criteria. This session transitions from component construction to system integration.



The main loop created this session manages the application's daily flow. It controls the progression from day to day, presents options to the user, processes actions, and advances the simulation state.

For Extended learners, this session introduces object lifecycles and state machines. A Shop object transitions between states (closed → open → processing → closed). Understanding these transitions supports designing methods that enforce valid state changes and prevent invalid ones.

Tasks 1-2: Store Open/Closed State

Task 1: The State Variable

Create a boolean variable representing whether your shop is currently open. This flag controls when customers can be processed, determines valid actions, and separates preparation from business hours.

Normal: Create **is_open = False** as a standalone variable.

Extended: Add **self.is_open = False** to your Shop class attributes.

Task 2: Control Functions

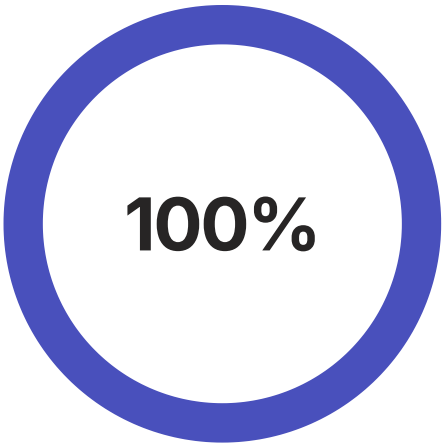
Implement two functions to change the store state:

- **open_store()** - Sets is_open to True
- **close_store()** - Sets is_open to False

Add confirmation messages such as "The shop is now open for business!" or "The shop has closed for the day."

Two States

Define shop operational status



State Rules

Supports future complex state rules

Dedicated functions (or methods) for state changes allow for future logic additions such as prerequisite checks, event logging, or side effects, without modifying code throughout the program. Calling **open_store()** expresses intent for a state change rather than directly manipulating a boolean.

Extended Design

Considerations for extended learners include: allowing store opening with empty inventory, requiring stock purchase before opening, or triggering cleanup/calculations upon closing. These behaviors can be integrated into methods via encapsulation without affecting other code.

Tasks 3-4: Day Counter and Status Screen

Task 3: Create a day counter variable starting at 1. Each complete business cycle (open, serve customers, close) should increment this counter. The day number influences customer count, difficulty, and progression systems.

Day Tracking

Integer starting at 1, increments after each business day.

Progression Scaling

Formulas use day number to increase difficulty.

Milestone Markers

Tracks player progress.

Task 4: Update your status display to include time-based information. Modify the status screen to show the current day alongside money, inventory, manager level, and store state (open/closed). This provides a complete picture of the simulation's current state.

Status Display Elements

- Day number (bold or prominent)
- Store state (Open/Closed)
- Current money
- Complete inventory listing
- Manager level
- Any other relevant context

Example Output

```
_____________________________________
____
   DAY 5
_____________________________________
____
Status: OPEN
Money: $47.50
Manager Level: 2

Inventory:
  Coffee: 8 units
  Tea: 3 units
  Pastry: 5 units
```

Use visual separators, alignment, and spacing to make the status screen easy to scan. This screen is frequently accessed. Extended learners should consider whether this display method might accept parameters for different "views" (summary vs. detailed) or if separate display methods would be cleaner.

Task 5: The Main Simulation Loop

This loop integrates functions into a cohesive simulation. It controls program flow, presents choices, processes actions, and maintains continuous simulation.

Display Status

Show current state at loop start

Update State

Modify money, inventory, day, etc.

Present Menu

List available actions

Get Choice

Accept user decision

Process Action

Execute selected function

Example Menu Options

1. View status
2. Buy items
3. Open store (when closed)
4. Close store (when open)
5. Advance to next day
6. Save game (later)
7. Quit simulation

Loop Structure

```
while True:
    show_status()
    display_menu()
    choice = get_input()

    if choice == "1":
        # View status
    elif choice == "2":
        # Buy items
    ...
    elif choice == "quit":
        break

    update_day_if_needed()
```

The loop must handle invalid input correctly, provide clear feedback, and maintain logical state transitions. For example, the store cannot open if already open; the day should not advance without closing the store first. These rules create coherent simulation behavior and demonstrate state validation.

Extended Architecture: The run() Method

For Extended learners, the main loop becomes a Shop method like **run_simulation()**. This method contains the while loop and delegates to other Shop methods for each action. This approach encapsulates the simulation control flow within the object that owns the state. External code creates a shop and calls **shop.run()**—all further operations happen internally.

Task 6: Determining Customer Count

The formula for determining daily customer count affects difficulty progression, revenue potential, and strategic decision-making.

Fixed Count

customers = 5

Simple and predictable, suitable for testing and early development. Balancing is straightforward.

Linear Growth

customers = 3 + day

Provides a steady, predictable increase (e.g., Day 1: 4 customers, Day 10: 13 customers).

Exponential Scaling

customers = 5 + floor(day × 1.5)

Results in an accelerating challenge. Early days are manageable, while later days show significant growth. This approach can create a need for system upgrades.

Random Range

customers = random(3, 8)

Introduces uncertainty. Players must prepare for variation. This is more difficult to balance and strategize for.

Create a function that returns the customer count for the current day. This function should take the day number as input (Normal) or access **self.day** (Extended). Select a formula; adjustments can be made during playtesting. The objective is to establish a mechanism for variable customer counts.

Mathematics of Customer Formulas

Linear: **customers = base + (day × growth_rate)**

Exponential: **customers = base + floor(day ^ exponent)**

Random range: **customers = random(min, max)**

Hybrid: **customers = (3 + day) + random(-2, 2)** combines growth with variation

For Extended learners, consider how customer count calculation should integrate with upgrades. If decoration level or reputation (future features) affect foot traffic, determine whether this function accepts additional parameters or reads upgrade levels from the Shop object. These architectural decisions influence system extensibility.

Tasks 7-8: Customer Generation

Task 7 requires creating a representation for a single customer. Each customer needs two properties: a **wish** (the item they want) and a **budget** (the amount they will pay). The data structure chosen affects interaction with customer data.

Dictionary Approach

customer = {"wish": "coffee", "budget": 5}

Flexible. Properties can be added or modified.

Tuple Approach

customer = ("coffee", 5)

Compact. Requires index-based access.

Class Approach (Extended)

customer = Customer("coffee", 5)

Structured. Allows associated methods (e.g., can_afford(), preferred_item()).

Task 7: Single Customer

Write a function that generates one customer with random properties:

- Randomly select wish from available items
- Randomly generate budget (e.g., 2-10)
- Return the customer data structure

Task 8: All Daily Customers

Write a function that creates the complete customer list for a day:

1. Call the customer count function
2. Loop the determined number of times
3. Generate a customer each iteration
4. Add the customer to a list
5. Return the full list

This list represents the daily queue of visitors.

For Extended learners using a Customer class, consider method placement. Determine whether customer behaviors are handled by customer methods or shop methods. For example, does a customer method execute a "buy" action, or does a shop method execute a "sell to" action, where the customer is an argument?

Tasks 9-10: Test Customer Interaction

Task 9 implements the transaction logic for a customer's purchase attempt. This function integrates inventory checking, price calculation, budget validation, and state updates.

Extract Customer Data

Read customer's wish and budget

Check Availability

Verify item is in stock (quantity > 0)

Calculate Price

Use your selling price formula: base × (1 + manager × 0.1)

Check Budget

Compare customer budget to selling price

Execute or Decline

If valid: reduce inventory, increase money. If not: no changes.

Transaction Mathematics

selling_price = base_prices[item] × (1 + manager_level × 0.1)

sale_valid = (inventory[item] > 0) AND (budget ≥ selling_price)

If valid:

inventory[item] = inventory[item] - 1

money = money + selling_price

Task 9: Purchase Logic

Create a function that accepts a customer and processes their purchase attempt. Return a boolean (True/False) indicating success, or return transaction details (item, price, success) for logging.

Failure handling: Decide if messages like "Out of coffee!" or "Customer couldn't afford tea" should be printed. Such feedback assists with debugging.

Task 10: Testing Integration

Add a menu option "Test customer purchase" that:

1. Generates one random customer.
2. Displays customer's wish and budget.
3. Processes the purchase.
4. Shows the result.
5. Displays updated shop status.

This verifies transaction logic before automated daily processing.

Testing individual customer transactions prior to processing entire days is required. Debugging a single transaction is simpler than diagnosing issues within a batch. Use clear print statements to show each step: "Customer wants coffee", "Coffee sells for $2.60", "Customer has $3.00", "Sale successful!". This diagnostic output is critical for problem resolution.

For Extended implementations, consider the use of return values versus side effects. A `process_customer(customer)` method should clearly indicate its effects through return values or exceptions for easier testing and debugging.

Session 2 Deliverables

Session 2 involves implementing time-based simulation components. Functionality must be verified before proceeding to Session 3.

Time Systems

Store state (open/closed), day counter, opening/closing functions implemented

Simulation Loop

Main loop displays status, presents menu, processes choices, maintains state

Customer Systems

Customer count calculation, single customer generation, full day customer generation

Transaction Logic

Purchase processing, test interaction integrated into menu

Normal Path Verification

- is_open boolean tracks store state
- open_store() and close_store() function
- day variable starts at 1 and increments
- Status screen shows day, state, money, inventory, manager level
- Main while loop executes continuously
- Menu offers multiple choices
- determine_customer_count() returns numbers
- generate_customer() creates customer data
- generate_all_customers() returns a list
- process_customer_purchase() updates state
- Test customer option functions end-to-end

Extended Path Verification

- Shop class has is_open, day attributes
- open_store() and close_store() are methods
- show_status() method displays complete state
- run_simulation() method contains main loop
- Customer class exists with wish, budget attributes
- generate_customers_for_day() returns Customer objects
- process_customer() method handles Customer objects
- Separation between Shop responsibilities and Customer data
- Methods use self appropriately
- Design allows for Session 3 expansions

Testing Checklist

Before continuing:

- Verify store can be opened and closed.
- Verify day counter increments properly.
- Verify customer generation produces varied wishes and budgets.
- Verify test purchases update money and inventory correctly.
- Verify failures (out of stock, insufficient budget) are handled.
- Verify main loop executes.

Completion of these checks indicates readiness for Session 3's automation.