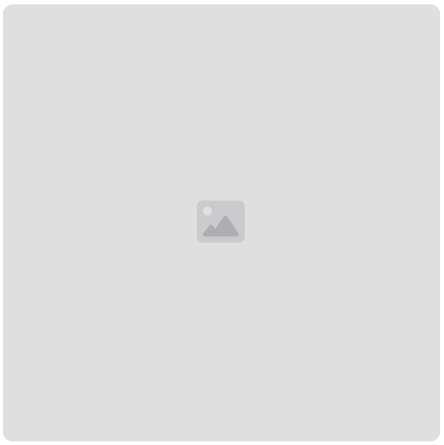
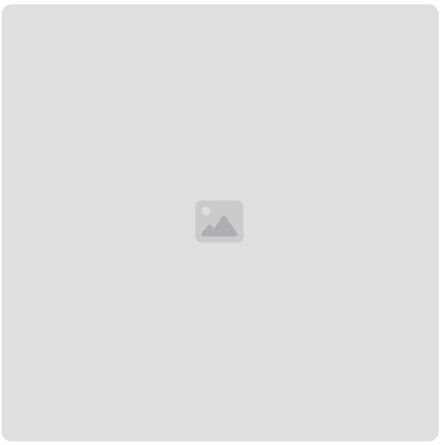
Session 4: Persistence and Polish

Session 4 implements saving and loading game state, statistics tracking, and refinement of upgrade systems and the user interface. These features are essential for a complete application.



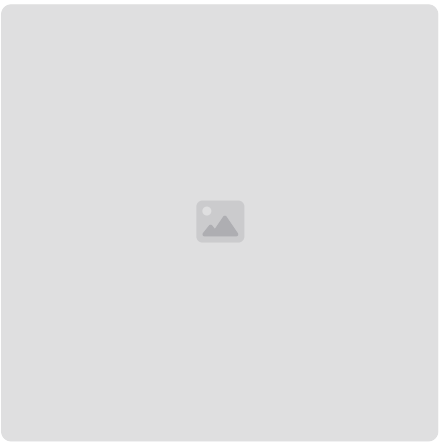
Game Persistence

Implement saving of complete shop state to JSON files and loading of previous games, enabling continuation from any point.



Analytics

Track performance metrics across all days, identify trends, and analyze item popularity.



Polish

Implement error handling, improve code organization, enhance displays, and add quality-of-life features.

Saving and loading involves state serialization. This includes defining the game's state, converting complex objects into JSON-compatible dictionaries, and reconstructing objects upon loading. These tasks cover software engineering concepts such as serialization, data integrity, and state management.

Statistics track gameplay data. For example, daily earnings can be compared to an average, top-selling items can be identified, and conversion rates can be analyzed following a manager upgrade. This provides feedback on progression.

Tasks 1-3: Statistics and Cost Systems

Task 1: Daily Statistics Storage

Create a list that accumulates summary data for each completed day. After every business day, append a dictionary or object containing:

- Day number
- Total customers
- Successful sales
- Revenue earned
- Optional: items sold breakdown

This historical data enables trend analysis and lifetime statistics.

Task 2: Detailed Sales Log

Beyond daily summaries, log every individual sale for granular analysis. Each entry records:

- Day it occurred
- Item sold
- Selling price
- Optional: customer budget, time of day

This detailed log provides data to analyze revenue by item or average sale price.

Complete History

Every transaction permanently recorded for analysis

Daily Summaries

Overview of each day's performance

Trend Analysis

Compare performance across days to identify patterns

Task 3: Unified Upgrade Cost Formula

Replace any hardcoded upgrade costs with a consistent exponential formula:

cost = base_cost × growth_factorcurrent_level

Define base costs and growth factors for each upgrade type:

- Coffee Machine: base=$10, growth=1.5
- Manager: base=$15, growth=1.5
- Decoration: base=$8, growth=1.4

Update your status screen to show next upgrade costs using this formula. This consistency provides predictable and balanceable economic progression.

For extended implementations, consider creating Record classes. A SaleRecord object encapsulates a single transaction, while a DailyRecord object summarizes a day. These objects can have methods such as **get_revenue()**, **get_items_sold()**, or **display()** for managing historical data. The Shop might maintain both a sales_log list of SaleRecords and a daily_stats list of DailyRecords.

Tasks 4-5: Saving and Loading State

Implementing save/load teaches concepts about data serialization, file I/O, and state reconstruction.

State Capture

Collect all relevant game state into a serializable structure

Serialization

Convert state into JSON format

File Writing

Save JSON to disk with appropriate filename

File Reading

Load JSON from disk when requested

Deserialization

Parse JSON back into game state

State Restoration

Apply loaded data to recreate previous game

Task 4: Save Function

Create a save function that captures complete game state:

```
{
  "money": 156.40,
  "inventory": {
    "coffee": 12,
    "tea": 8,
    "pastry": 5
  },
  "base_prices": {...},
  "manager_level": 3,
  "coffee_machine_level": 2,
  "decoration_level": 1,
  "day": 8,
  "daily_stats": [...],
  "sales_log": [...]
}
```

Use Python's **json** module to write this dictionary to a file like "shop_save.json".

Task 5: Load Function

Create the complementary load function that:

1. Opens the save file
2. Parses JSON into a Python dictionary
3. Recreates all game variables from the loaded data
4. Validates data is sensible (optional)
5. Confirms successful load

Handle errors gracefully, such as when the file does not exist or contains corrupted JSON.

For Normal implementations, save and load functions work with global variables or pass the entire state structure. For Extended implementations, a Shop class can have **to_dict()** and **from_dict(data)** methods, plus **save(filename)** and **load(filename)** wrappers. The **to_dict** method converts attributes to JSON-compatible types, while **from_dict** reconstructs the object from a dictionary.

Serialization Challenges

Python's json module handles basic types (int, float, str, list, dict) but not custom objects. Extended learners must convert custom objects into dictionaries before saving. Consider implementing **to_dict()** methods on classes for serialization. When loading, logic is needed to recognize dictionary representations of objects and reconstruct the appropriate object.

Tasks 6-10: Additional Features

Task 6 integrates save/load into your menu system. Add "Save game" and "Load game" options that call your serialization functions. Consider allowing multiple save files, auto-saving after each day, or prompting before overwriting existing saves.

15	$1,247	87%	Coffee
Days Played	**Lifetime Revenue**	**Best Day Conversion**	**Top Seller**
Total simulation runtime tracked	Cumulative earnings across all days	Highest sale success rate achieved	Most frequently purchased item

Task 7 (Optional) adds analytics. Create functions that compute aggregate statistics across all recorded days: total revenue, total customers served, total sales, best day, worst day, most popular item, average daily revenue. These lifetime stats provide an overview of simulation performance.

Task 8: Error Handling (Extended)

Add try/except blocks around operations that may fail:

- File operations (FileNotFoundError, JSONDecodeError)
- User input (ValueError for invalid numbers)
- Calculations (ZeroDivisionError in analytics)

Implement error handling to prevent crashes and provide messages to the user.

Task 9: Code Organization

Normal: Group related functions, add docstrings, ensure clear variable names.

Extended: Split code into multiple files:

- shop.py - Shop class
- customer.py - Customer class
- upgrades.py - Upgrade classes
- main.py - Run simulation

Task 10 enhances the status display. Include upgrade levels with their effects, next upgrade costs, lifetime statistics summary, and current day context. Consider adding ASCII art borders, color (if your terminal supports it), or formatted tables for presentation.

Project Completion

This project demonstrates concepts such as variables, functions, loops, dictionaries, file I/O, and optionally object-oriented programming. The shop simulation is a functional game with persistent state. This project covers course content and provides a foundation for individual final projects. The skills developed—state management, transaction processing, serialization, object design—are applicable to various applications.

Session 4 Deliverables

By the end of Session 4, the simulation should include:

Normal Deliverables:

- Daily statistics
- Simple sales log (optional but encouraged)
- Unified upgrade cost formula
- Save game state to JSON
- Load game state from JSON
- Menu options to save and load
- Clean status screen including upgrades and next costs
- A fully working multi-day simulation loop

Extended Deliverables (Optional):

- A Shop class with: run_business_day(), run_simulation(), to_dict() + from_dict(), save() + load()
- An Upgrade class hierarchy
- A Customer class (and possibly subclasses)
- A SaleRecord and/or DailyRecord class
- Analytics methods: revenue by day, best day, popular items, conversion rate
- Error handling for bad input and file operations
- A modular file structure (multiple .py files)