Making an Adventurer’s Guild System

using Python

Programming Languages Assignment 5

Overview

Build a **text-based console application** from scratch in **Python** that simulates an **Adventurer’s Guild**. The system must support user registration/login, accepting and submitting quests, training skills, and viewing history. All persistent data must be stored in **JSON files** that you design.

This assignment also requires you to **demonstrate three programming techniques** in real, user-facing features (no toy demos). Each technique is worth **20 points**:

1. **Parameter Binding** (positional, keyword-only, defaults, `*args`, `**kwargs`)
2. **Closure** (state-capturing factory functions used as policies/strategies)
3. **User-defined Overloaded Operators** (domain classes with meaningful operators)

Functional Requirements

0) Initial Screen

* Menu: `{Register, Login, Exit}`
* Exit terminates the program.

1) Register

* Input: `username`, `login password`, `PIN` (4-digit number).
* On success, assign a random 5-digit Adventurer ID.
* On failure, print **all** violated rules and restart registration:
    * Username must be **unique**.
    * Password must be **≥ 8 characters**, contain **≥ 1 uppercase**, and **≥ 1 special** from '!@#$%^&*()'.
    * PIN must be **exactly 4 digits**.

2) Login

* Input `username` and `login password`.
* If invalid, show an error; on success, go to **Main**.

3) Main Screen

* Always display: `username`, `Adventurer ID`, `Rank` (default `BRONZE`), `Stamina` (default `100`).
* Menu: `{History, Accept Quest, Submit Quest, Train Skill, Logout}`

  * **Logout** returns to Initial Screen.

4) History

* Show the user’s activity in **reverse chronological order** (most recent first).
* Each entry must include a timestamp and:

  * **Accept**: time, Quest ID, title, difficulty
  * **Submit**: time, Quest ID, result (Success/Fail), reward summary (XP/Fame/Loot)
  * **Train**: time, skill, minutes, skill gain
* You may reconstruct History from fields stored in the user JSON (a separate log file is **not required**).

5) Accept Quest

* Display a **Quest Board** (≥ 5 quests) with: Quest ID, title, difficulty, required skill (e.g., `hunting ≥ 10`), due date.
* Validate: quest exists, not already accepted, skill requirement met, not past due.
* Require **PIN** confirmation before accepting.

6) Submit Quest

* Inputs: `Quest ID` (must be accepted) and any required **proof item names** (strings).
* Require **PIN** confirmation.
* **Success criteria (deterministic, recommended)**: all of the following:

  1. The quest is currently accepted by the user.
  2. Current date ≤ quest due date.
  3. PIN is correct.
  4. All required proof items are provided.
  5. The user has enough stamina (e.g., submission costs 10).

* On success, apply rewards (XP/Fame/Loot), remove the quest from accepted, add to completed, and update rank if applicable.
* Handle errors for missing/invalid quest, missing proofs, wrong PIN, insufficient stamina, etc.

7) Train Skill

* Inputs: `skill name`, `minutes` (default 30).

* Require **PIN** confirmation.

* Increase the skill according to your **training plan policy** (implemented via a closure with diminishing returns).

* Validate minutes > 0 and stamina sufficient; deduct stamina (e.g., 5).

Required Programming Techniques (3 × 20 pts)

A) Parameter Binding (20 pts)

Implement real, user-facing functions that exercise:

* **Keyword-only parameters**
* **Defaults**, **`*args`**, and **`**kwargs`**
* Show at least one **binding error** (e.g., duplicate values for an argument) as a clear, user-friendly message (do not crash with a raw traceback).

What we will check

* Presence and proper use of positional, keyword-only, defaults, `*args`, `**kwargs`.
* Binding errors are caught and reported meaningfully.

B) Closure (20 pts)

Create **factory functions that capture internal state** and return callables used in actual features. Provide at least **two independent instances** that evolve independently.

What we will check

* Policies are closures (use of `nonlocal` or equivalent).
* Different instances produce different outcomes over time, given similar inputs.

C) User-defined Overloaded Operators (20 pts)

Define domain classes and use them in meaningful game logic.

What we will check

* Operators are actually used in quest reward application, inventory changes, or rank logic.
* Error paths (e.g., subtracting more items than owned) are handled correctly.

Data & Storage

* Use **JSON** only (no external DBs or packages).

* You design the schema. It must include at least:

    * **Users file**: `id`, password hash, PIN hash, rank, stamina, skills, inventory, accepted/completed quests (with timestamps) and any fields needed to rebuild History.

    * **Quests file**: Quest ID, title, difficulty, required skill and min level, due date, base rewards, required proof items.

* Choose a consistent **write timing** (e.g., save immediately after each operation).

Constraints

* Language: **Python** only; use the **standard library** (no external packages).

* UI: **Console (CLI)**.

* OS: any (Windows/macOS/Linux).

* Implement from scratch; no code templates are provided.

Submission

Zip name: `{student_id}_Assignment5.zip`

Include:

1. **Code**: `{student_id}_guild.py` (entry point) + any additional `.py` files you wrote.
2. **Storage**: the JSON file(s) created by your program (actual run results).
3. **Report (PDF)**: `{student_id}_guild_report.pdf` containing:

 * System architecture (modules, key flows, where JSON is read/written).
 * Your JSON schema (key fields) and write timing.
 * Where and how each required technique is implemented (one subsection per technique).
 * **Screenshots** showing: two users; Accept, Submit, Train; History view; a parameter-binding error; two distinct closure instances producing different results; operator overloading in action.

Grading (100 pts)

* **Design (10 pts)**: Clear menus; consistent, user-friendly messages; easy to follow.
* **System Completion (20 pts)**: All screens/features above work with robust input validation.
* **Required Techniques (60 pts = 3 × 20)**

  * Parameter Binding (20)
  * Closure (20)
  * User-defined Overloaded Operators (20)
* **Report (10 pts)**: Architecture clarity, schema description, screenshots, and technique explanations.

> Missing any one of the three techniques yields **0 points** for that technique. Partial credit (up to 5 pts) may be given only if the report and code clearly show an attempt that partially works.

Demonstration Checklist (for students)

* Register **two distinct users** and log them in separately.
* Accept at least one quest per user; submit at least one (with proofs) and show rewards applied.
* Train at least once; show stamina and skill changes.
* History shows actions in reverse chronological order.
* **Parameter Binding**: show keyword-only/defaults, `*args`, `**kwargs`, and a handled binding error.
* **Closure**: two policy instances behave differently under repeated calls.
* **Operator Overloading**: show `XP` arithmetic/comparison affecting rank or rewards, and `LootBag` `+`/`-` affecting inventory.

Academic Integrity

Any **plagiarism**, **code sharing**, or use of generative AI to produce your deliverables will result in an **F** for the course assignment.