Holly Renfrew

Due Date: 11/9/2025

Code Review CS: 499

Software Engineering and Design

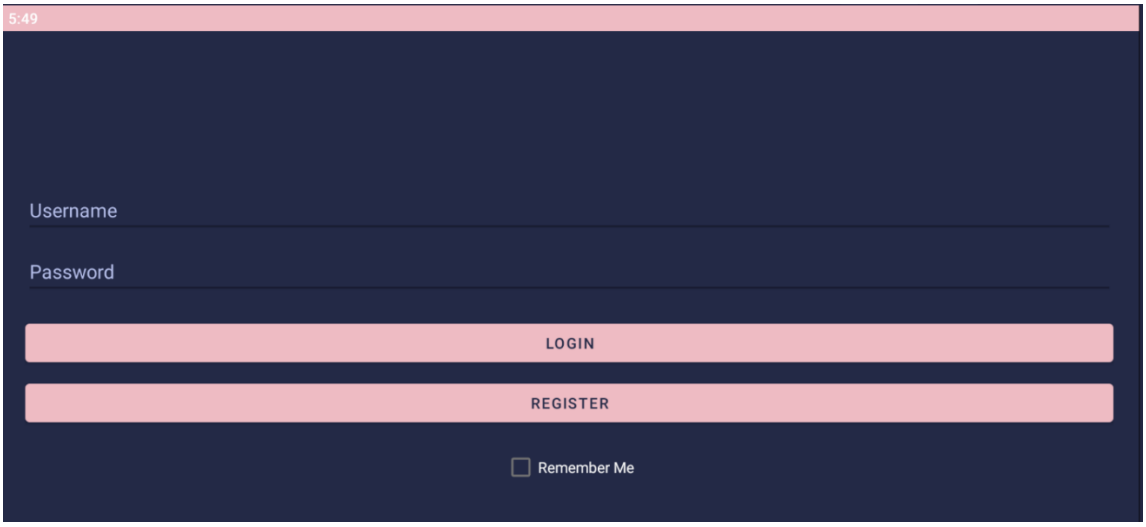
Introduction

For this portion we will be reviewing my artifact for the *Software Engineering and Design* category. The artifact I selected is my Android Weight Tracker Application, which I originally created in CS-360: Mobile Architect and Programming. This app allows users to log, track, and visualize their weight changes over time while setting personal goals.

I selected this project because it demonstrates applied software design, use of Android architecture components, user interface layout, database interaction, and input validation. But most importantly, it provides a strong foundation for improving software structure, performance, and especially security, which is where I'll focus my enhancement plan.

Existing Code Functionality

The Weight Tracker application is built using Java and the Android SDK. When a user opens the app, they're greeted with a login screen that connects to an SQLite database through a helper class. New users can register by entering a username, password, email, and phone number. Passwords are hashed using BCrypt, ensuring that no plain-text passwords are ever stored in the database.



Once logged in, users are taken to the main dashboard, where they can add new weight entries, update or delete existing ones, and set a personal goal:

All entries are time-tamped and stored in the database. The app uses the MPAndroidChart library to generate a line graph showing weight over time. Users can toggle between “gain weight” and “lose weight” goals using a simple radio button, and when they reach their target, the app can trigger an SMS notification congratulating them.

The codebase includes several major components:

- LoginTracker.java - handles authentication and the “remember me” feature.

- RegisterTrackerActivity.java - registers new users, checks for duplicates, and validates inputs.
- DashboardTracker.java - manages the core dashboard, charts, and SMS triggers.
- DatabaseHelper.java - defines the SQLite schema and provides CRUD operations for users, goals, and weights.
- WeightAdapter.java - manages RecyclerView items for displaying weight entries.
- SmsPermissionTracker.java - handles runtime permissions and toggling SMS alerts.

Overall, the application successfully connects user input to database operations and dynamically updates both data and visuals, providing a responsive user experience.

Code Review Analysis

Structure

The project's structure is consistent with good Android design principles. Each screen is represented by its own activity, and shared logic is separated into helper classes such as DatabaseHelper.

However, through review, I found that some UI classes still handle background operations, such as database inserts and updates, directly on the main thread. This can lead to performance issues or 'Application Not Responding' errors if the dataset grows or the device is slow.

```java
// REVIEW NOTE:
// These database operations currently execute on the main thread.
// Consider moving them to a background thread (e.g., AsyncTask, Executor, or coroutine)
// to prevent UI lag or potential ANR errors when datasets grow.

long newId = dbHelper.addWeightAndGetId(userId, weight, date);
if (newId == -1) {
    Toast.makeText( context: this, R.string.failed_to_add_weight, Toast.LENGTH_SHORT).show();
    return;
}
```

DashboardTracker.java

Additionally, while the logic is clear, some sections could benefit from modularization - for example, extracting login and session management into a dedicated AuthenticationService class.

Documentation

Each method is logically named and readable, but comments are minimal. There are a few instances, especially in DashboardTracker.java, where method responsibilities aren't documented. Adding clear JavaDoc or inline comments describing the purpose of complex methods - such as updateChart() or checkAndSendSMS() - would make maintenance much easier for future developers.

```
private void updateGoalDisplay() {
    double goalWeight = dbHelper.getGoal(userId);
    textCurrentGoal.setText(getString(R.string.goal_label_fmt, goalWeight));
}

5 usages
private void updateChart() {
    List<Entry> entries = new ArrayList<>();

    for (int i = 0; i < weightEntries.size(); i++) {
        int reversedIndex = weightEntries.size() - 1 - i; // oldest first
        WeightEntry we = weightEntries.get(reversedIndex);
        entries.add(new Entry(i, (float) we.getWeight()));
    }
```

DashboardTracker.java

Variables

Variables are well-named, and data types are appropriate. However, some user input isn't normalized before validation. For example, Holly' and 'holly' would be treated as different usernames since there is not a '.toLowerCase()'.

```
private void attemptRegister() {
    String email = editTextEmail.getText().toString().trim();
    String username = editTextUsername.getText().toString().trim();
    String password = editTextPassword.getText().toString().trim();
    String phone = editTextPhone.getText().toString().trim();
```

RegisterTrackerActivity.java

This can lead to duplicate records or login confusion. In addition, password strength validation is currently weak. It only checks for an uppercase letter and a number. Strengthening this requirement will help enforce better data integrity and security.

```
    1 usage
    private boolean isValidPassword(String password) {
        // At least one uppercase letter and one number
        return Pattern.compile( regex: "^(?=.*[A-Z])(?=.*\\d).+$").matcher(password).matches();
    }
}
```

RegisterTrackerActivity.java

In the chart logic, I found that the app calculates chart entries in reverse order:

```
private void updateChart() {
    List<Entry> entries = new ArrayList<>();

    for (int i = 0; i < weightEntries.size(); i++) {
        int reversedIndex = weightEntries.size() - 1 - i; // oldest first
        WeightEntry we = weightEntries.get(reversedIndex);
        entries.add(new Entry(i, (float) we.getWeight()));
    }
```

DashboardTracker.java

But still references the original list when labeling the X-axis:

```java
// X-axis formatter to show date/time
lineChart.getXAxis().setValueFormatter(new ValueFormatter() {
    @Override
    public String getFormattedValue(float value) {
        int index = Math.round(value);
        if (index >= 0 && index < weightEntries.size()) {
            String dbDate = weightEntries.get(index).getDate(); // "yyyy-MM-dd HH:mm:ss"
            try {
                SimpleDateFormat parser = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss",
                        Locale.getDefault());
                Date date = parser.parse(dbDate);
                if (date != null) {
                    SimpleDateFormat formatter = new SimpleDateFormat( pattern: "MM/dd HH:mm:ss",
                            Locale.getDefault());
                    return formatter.format(date);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return "";
    }
});
```

DashboardTracker.java

This can cause a mismatch between the displayed points and their corresponding timestamps.

Rewriting the chart update function to maintain a synchronized label list will correct this issue.

Defensive Programming

The defensive programming check uncovered the most significant opportunities for improvement.

First, I looked at how the app stores session data after a successful login. In reviewing LoginTracker.java, I noticed that the application saves both the user ID and the ‘remember me’

state directly in SharedPreferences. Because SharedPreferences stores values in plain text on the device, this exposes user identity information. Here is the section of code where this occurs:

```
prefs.edit()
        .putInt("userId", userId)      // ✅ only save userId
        .putBoolean(KEY_REMEMBER, true)
        .apply();
```

```
// AUTO-LOGIN CHECK HERE
boolean remember = prefs.getBoolean(KEY_REMEMBER,  defValue: false)
int savedId = prefs.getInt( key: "userId",  defValue: -1);
if (remember && savedId != -1) {
    goToDashboard(savedId);
    return; // skip rest of login screen
}
```

LoginTracker.java

To address this, I'll transition to EncryptedSharedPreferences, which automatically encrypts data on disk using Android's KeyStore system.

Next, I examined the login flow to see how the app handles invalid login attempts. I found that the application does not track failed attempts, and users can try to log in as many times as they want. This creates an opportunity for brute-force password guessing. Here's the code where failed attempts are handled, but no lockout is applied:

```
int userId = dbHelper.loginUser(username, password);
if (userId != -1) {
    if (checkboxRememberMe.isChecked()) {
        prefs.edit()
                .putInt("userId", userId)      // ✅ only save userId
                .putBoolean(KEY_REMEMBER, true)
                .apply();
    } else {
        prefs.edit().clear().apply();
    }
    goToDashboard(userId);
} else {
    Toast.makeText( context: this,  text: "Invalid username or password", Toast.LENGTH_SHORT).show();
}
```

LoginTracker.java

I also checked other screens for similar storage concerns. In the SMS permission screen, the app again uses plain SharedPreferences to save the user's SMS alert preference. While this is less sensitive than login data, it shows that SharedPreferences are used throughout the app, making the migration to EncryptedSharedPreferences a consistent and valuable improvement. Here is that code.

```
prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
SwitchCompat switchSms = findViewById(R.id.switchSms);
Button buttonDone = findViewById(R.id.buttonDone);

//  Restore saved state
boolean enabled = prefs.getBoolean(KEY_SMS_ALERTS,  defValue: false);
switchSms.setChecked(enabled);

//  Save when toggled
switchSms.setOnCheckedChangeListener(( CompoundButton buttonView,  boolean isChecked) -> {
    prefs.edit().putBoolean(KEY_SMS_ALERTS, isChecked).apply();
```

SMSPermissionsTracker.java

Loops and Branches

All control flow is clear and complete, though several nested if-else chains could be simplified or converted to early returns for readability. The use of nested loops in the charting logic is correct

and efficient, but future versions could move the chart computation off the main thread for better performance.

```
if (triggerBelowGoal) {
    // "Lose weight" (below goal)
    if (weight <= goalWeight) {
        new AlertDialog.Builder( context: DashboardTracker.this,
                R.style.Theme_Holly_Renfrew_Weight_Tracker_Dialog)
                .setTitle(R.string.congrats_title)
                .setMessage(getString(R.string.met_goal_below_fmt, goalWeight))
                .setPositiveButton(android.R.string.ok,  listener: null)
                .show();
    }
} else {
    // "Gain weight" (above goal)
    if (weight >= goalWeight) {
        new AlertDialog.Builder( context: DashboardTracker.this,
                R.style.Theme_Holly_Renfrew_Weight_Tracker_Dialog)
                .setTitle(R.string.congrats_title)
                .setMessage(getString(R.string.met_goal_above_fmt, goalWeight))
                .setPositiveButton(android.R.string.ok,  listener: null)
                .show();
    }
```

DashboardTracker.java

Target Areas for Improvement

Summarizing the review, the most important areas for enhancement include:

1. Implementing secure data storage for session tokens.
2. Introducing a login lockout system to protect against brute-force attacks.
3. Enhancing password and input validation using stronger regular expressions.
4. Running all database operations on a background thread to improve performance.
5. Adding inline documentation and moving string literals to strings.xml for localization.

6. Correcting chart label mismatches to ensure data visualization accuracy.

Planned Enhancements

My planned enhancement focuses on transforming the app's security and performance posture. Here's what I'll implement:

- EncryptedSharedPreferences: Replace plain SharedPreferences with encrypted preferences, so user IDs, tokens, or session flags are protected even if the device is compromised.
- Login Lockout Mechanism: Add two new database fields - failed_attempts and locked_until. Each failed login will increment a counter, and after five failures, the user will be locked out for ten minutes. Successful logins will reset the counter.
- Strong Password Enforcement: Replace the current simple check with a stronger pattern requiring at least one uppercase letter, one lowercase letter, one digit, one special character, and a minimum of ten characters.
- Threaded Database Operations: Move database insert, update, and delete functions off the main thread using an Executor or AsyncTask to avoid performance stalls.
- Improved Input Normalization: Convert emails and usernames to lowercase before validation and insertion to prevent duplicates.
- Permission Accuracy: Update the SMS permission logic to only enable alerts after explicit user approval.
- Documentation Updates: Add concise comments to each major function to describe its purpose and improve maintainability.

Practical Impact of Enhancements

These changes make the project much closer to professional software development standards. By improving session security and login handling, I demonstrate an understanding of secure authentication workflows - skills that are vital in both mobile and web development.

Threading and modularization enhancements will lead to better performance, stability, and easier debugging. And by improving input validation and documentation, the code becomes clearer and more user-friendly for future developers who might work on the project.

Skills Demonstrated

These enhancements allow me to demonstrate multiple technical and professional skills, including:

- Secure Software Engineering: implementing encryption, secure storage, and input sanitization.
- Software Design and Architecture: restructuring logic for better modularity and maintainability.
- Performance Optimization: using background threads and resource management for smooth operation.
- Documentation and Professional Communication: improving readability and providing clear, structured explanations of each enhancement.

Alignment with Course Outcomes

My planned work aligns with several program outcomes from the CS-499 course:

- Outcome 4: Demonstrate the ability to use well-founded and innovative techniques, skills, and tools in computing practices for implementing computer solutions that deliver value and accomplish industry-specific goals.
- Outcome 5: Develop a security mindset that anticipates adversarial exploits in software architecture and designs to expose potential vulnerabilities, mitigate flaws, and ensure privacy and data security.

  Additionally, by refactoring the project and writing clear documentation, I also touch on Outcome 2, which emphasizes professional, technically sound communication."

Conclusion

To conclude, the Weight Tracker project successfully fulfills its initial goal as a functional, user-friendly health application.

Through this code review and enhancement plan, I've identified and documented specific areas where its design and security can be improved to reach professional quality.
By implementing encrypted storage, secure login workflows, improved validation, and asynchronous processing, I'll enhance both the performance and the trustworthiness of this software.

These improvements not only demonstrate my ability to analyze, design, and enhance software systems but also show growth in applying real-world security practices and professional coding standards: skills that directly align with my future goals as a software developer.

Algorithms and Data Structure

Introduction

For the Algorithms and Data Structures category, I chose my Airgead Banking investment calculator, which I originally created for CS-210: Programming Languages. This project is a C++ console application that helps users explore how their investments grow over time based on starting balance, monthly deposits, interest rate, and duration in years.

I picked this artifact because it focuses directly on core logic, numeric processing, and looping structures. It also shows how I separate responsibilities across multiple classes while still keeping the algorithms straightforward and efficient. There is a clear opportunity to refine the internal data handling, improve numerical accuracy, and make the underlying algorithms more reusable and testable.

Existing Code Functionality

The Airgead program works in a straightforward way:

1. The user is prompted for inputs such as currency symbol, initial investment, monthly deposit, interest, and number of years.
2. The program stores these values inside an InvestmentData object.
3. DataCalculation takes these values and computes monthly compounding interest.
4. Two formatted yearly reports are printed:
    - One without monthly deposits.
    - One with monthly deposits.

5. The user is asked if they want to run another calculation.

The primary C++ files include:

1. main.cpp
    - Controls the main loop, gets user input, calls the calculation, and asks if they want to continue.
2. InvestmentData.cpp
    - Stores and retrieves all user inputs.
3. UserInterface.cpp
    - Handles all input validation and formatting for prompts.
4. DataCalculation.cpp
    - Runs the core compounding interest algorithm and prints the annual reports.

Overall, the program successfully performs the investment projections it was designed to do.

Code Review Analysis

Structure

The structure is functional and clear. Each responsibility is separated into logical files. However, there are places where improvements are needed. For example, some code that belongs in validation functions is mixed directly into the user interaction loop. Screenshot this section:

```
/* Get initial investment from user.*/
while (!validator) {
    cout << "Initial Investment Amount: ";
    if (cin >> amount) {
        inputInvestment.SetInitialInvestment(amount); //set input as initial investment
        validator = true;
    }
    else {
        UserInterface::inputError("number");
    }
}
```

UserInterface.cpp

Documentation

The code is readable but lacks comments in key computational areas.

A great example to screenshot is in DataCalculation.cpp, inside YearlyBalances. The calculation algorithm itself is not commented even though it is the core logic.

```
total = openingAmount + depositAmount;
totalInterest = total * monthlyInterest;
closingBalance = total + totalInterest;
interestTotal = totalInterest + interestTotal;
```

DataCalculation.cpp

Variables

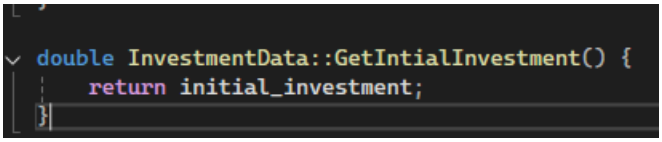
Variables are generally named well, but there are two concerns:

1. User input is stored in int amount, which restricts decimals even though currency inputs are often decimals.

```
int amount = 0;
```

UserInterface.cpp

2. Naming consistency can improve clarity, such as standardizing capitalization in getters (GetIntialInvestment has a typo, instead of Intial, it should be Initial).



InvestmentData.cpp

Arithmetic Operations

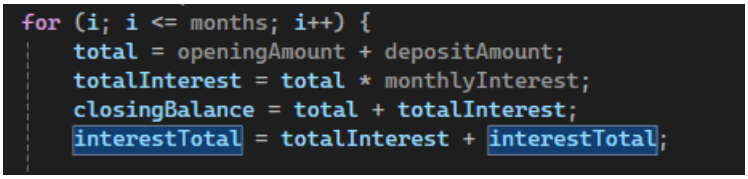
The algorithm for compound interest is correct, but there are some places where precision can be improved, especially with doubles.

Inside YearlyBalances, interest is accumulated monthly but then scaled manually:

DataCalculation.cpp

Loops and Branches

Loop control is complete and logical, but readability could be improved by using early continues or extracting code into helper methods. There is a missing initializer in the for loop, which is a small correctness issue:



DataCalculation.cpp

Target Areas for Improvement

Summarizing the findings, the most important enhancement opportunities include:

1. Fixing the yearly interest calculation so that it reflects proper monthly compounding.
2. Improving input validation. For example, rejecting negative numbers or unrealistic interest rates.
3. Refactoring repeated user input loops into a single reusable helper.
4. Increasing documentation around the algorithm's steps.
5. Renaming or standardizing functions with capitalization errors.
6. Extracting formatting logic into separate functions to improve readability.
7. Improving precision by avoiding unnecessary use of integers for money.

Planned Enhancements

Here is what I will implement to enhance the Airgead program:

Algorithm Fixes

Correct the yearly interest calculation by removing the manual multiplication and letting the loop accumulate values naturally.

Input Validation

Add validation for:

- negative values
- unrealistic interest
- extremely long year ranges

Refactoring

Create a generalized function to handle input collection to remove repeated loops.

Documentation

Add comments to describe each computational step within YearlyBalances.

Variable Improvements

Change int amount to double where appropriate and fix naming issues like GetIntialInvestment.

Separation of Concerns

Move the formatting code into helper functions, making YearlyBalances easier to read and extend.

Practical Impact

These enhancements will make the program:

- more accurate
- more maintainable
- easier to test
- more professional
- more aligned with real-world programming practices

They also demonstrate better my understanding of algorithmic clarity, defensive coding, and modularity.

Skills Demonstrated

Through these enhancements, I will show:

- Algorithmic reasoning
- Improved software design
- Defensive programming
- Accurate use of looping and mathematical logic
- Clean documentation practices
- Modular C++ class design

Alignment with Course Outcomes

- **Outcome 3: Design and evaluate computing solutions using algorithmic principles while managing trade-offs.**

  This is the strongest match for the Airgead enhancements. By fixing the compound interest algorithm, correcting the yearly interest accumulation, improving numeric precision, and restructuring repeated logic into modular functions, I am directly evaluating how the existing algorithm works and redesigning it to be more accurate and mathematically sound. These improvements show that I can analyze algorithmic behavior, identify where the logic breaks down, and make trade-offs between performance, readability, and accuracy.

- **Outcome 4: Demonstrate an ability to use well-founded and innovative techniques, skills, and tools in computing practices.**

  Refactoring the input system to reduce repetition, correcting naming inconsistencies, and improving documentation all help demonstrate my ability to use cleaner and more maintainable coding practices. Updating the program to use better variable types, reorganizing loop logic, and extracting formatting routines also show that I can apply modern programming techniques to make legacy code more robust and professional.

- **Outcome 2: Design, develop, and deliver professional-quality written communication.**

This enhancement gives me the chance to document the algorithm in a clearer way. Adding comments to the core compounding logic and describing complicated sections of code helps future developers understand how the calculations work. The process of explaining these enhancements in my ePortfolio also supports this outcome, since it requires clear written communication about technical work.

Overall, the enhancements planned for the Airgead project give me the opportunity to show a strong understanding of algorithm design, data processing, and code quality. These directly align with program outcomes related to algorithmic evaluation, professional coding practices, and clear communication.

Conclusion

To wrap everything up, the Airgead Banking Application still does exactly what it was originally built to do. It takes user input, processes compound interest over time, and prints a clean yearly report that helps users visualize their long-term investment growth. For a project written early in my program, it already shows solid structure, clear separation of responsibilities, and a straightforward algorithm that is easy to follow.

Through this review, I was able to step back and look at the code the way a professional developer would. I found several places where the logic can be strengthened, especially in the compound interest calculations and the formatting of the yearly output. There are also

opportunities to improve input validation, reduce repeated code, and bring more clarity to the functions through better naming and documentation.

By correcting the calculation inconsistencies, simplifying the control flow, and turning repeated logic into reusable functions, I will move the project closer to a professional standard. These enhancements not only improve the accuracy and reliability of the program, but they also show my growth in algorithmic thinking, code organization, and maintainability.

Making these changes also gives me the chance to demonstrate important skills tied to the program outcomes. Improving the algorithm aligns with core computing principles, while restructuring and documenting the code reflects my ability to communicate clearly and work with technical material in a professional way.

Overall, this enhancement will turn Airgead into a polished, well-structured artifact that represents my abilities in both problem-solving and software refinement. It is satisfying to look back at something I built early on and bring it up to the level of work I can produce now.

Databases

Introduction

For the Databases category, I selected my Python text adventure game, originally created in IT-140: Introduction to Scripting. This project was an early milestone in my degree, and it focuses heavily on branching logic, modular functions, and stateful gameplay.

I chose this artifact because the game relies on many in-memory structures such as dictionaries, lists, player objects, and room definitions, but it currently has no persistent storage. Everything resets on every run, which means there is a major opportunity to expand the project by integrating an actual database.

This makes it a perfect candidate for demonstrating database skills, especially converting hard-coded data into relational tables and enabling saved sessions, load states, and long-term progression.

Existing Code Functionality

The Heroes & Villains game is a Python-based command-line adventure where the player moves between rooms, fights enemies, interacts with items, and advances the storyline.
At the moment, the game handles all its information in Python files inside folders like:

- databases/player_information.py
- databases/rooms.py
- story_materials/*.py

But all "database" information is actually Python variables and dictionaries, not a real database.

Currently, the game includes:

1. Player stats stored in memory (health, attack, boons/banes, etc.)

```
class Player:  1 usage
    def __init__(self):
        self.name = "William"
        self.max_hp = 200
        self.hp = 200
        self.strength = 5
        self.defense = 6
        self.speed = 6
        self.skill = 7
        self.luck = 5
        self.magic = 4
        self.max_magic = 4
```

characters.py

2. Room definitions stored in Python dictionaries include enemies, descriptions, and special interactions. It also stores the directions one can move in each room.

```
# Entry_Hall
entry_hall_items = ["Bandage"]
entry_hall_directions = ["North", "South", "East", "Cancel"]
entry_hall_enemies = [entry_1]
eh_description = ("The Entry Hall  is a small room  at the entrance\n"
                  "to the bandit hold.  The wooden walls are barely\n"
                  "holding together.  It could make one  wonder how\n"
                  "it  is  still  holding  together.   It  is  also\n"
                  "probably  hard for  bandits to  find someone  to\n"
                  "repair buildings for them, which is why it is in\n"
                  "such disrepair.\n")

eh_description_empty = "There is  a dead  bandit  in the  middle of  the\nroom now."
eh_description_enemy = "There is a bandit in the middle of the room.  He\nattacks you!"
```

rooms.py

```
player = Player()
entry_1 = Enemy("Rodriguez")
ale_1 = Enemy("Ramirez")
ale_2 = Enemy("Remiel")
prisoner_1 = Enemy("Riley")
prisoner_2 = Enemy("Robert")
armory_1 = Enemy("Russell")
training_1 = Enemy("Ryder")
treasure_1 = Enemy("Ronan")
treasure_2 = Enemy("Remy")
dorm_1 = Enemy("Rafael")
dorm_2 = Enemy("Ryker")

# Boss Stats
razor_fang = Enemy("Razorfang")
razor_fang.max_hp = random.randint( a: 35,  b: 45)
razor_fang.hp = razor_fang.max_hp
razor_fang.strength = random.randint( a: 10,  b: 20)
razor_fang.defense = random.randint( a: 7,  b: 12)
razor_fang.speed = random.randint( a: 5,  b: 13)
razor_fang.skill = random.randint( a: 5,  b: 15)
razor_fang.luck = random.randint( a: 15,  b: 25)
razor_fang.weapon = razor_axe

enemy_list = (entry_1, ale_1, ale_2, prisoner_1, prisoner_2, armory_1,
              training_1, treasure_1, treasure_2, dorm_1, dorm_2, razor_fang)
```

characters.py

3. Movement system allowing the player to type commands like *North, South, East, West*.

```
def move():
    print_slowish(format_string_center.format(string="Which direction would you like to move?"))
    print_directions()
    move_direction = input("> ").capitalize().strip()
    while move_direction not in player_information.current_room.directions:
        print_slowish(format_string_center.format(string="Please enter one of the following options:"))
        print_directions()
        move_direction = input("> ").capitalize().strip()
    move_between_rooms(move_direction)


def move_between_rooms(direction):  1 usage
    from story_materials import enter_room
    current_directions = empty
    for i in direction_list:
        if i.name == player_information.current_room.name:
            current_directions = i
    if direction != "Cancel":
        new_room = getattr(current_directions, direction)
        player_information.current_room = new_room
        prep_string = "\nYou walked " + direction + " to reach the " + player_information.current_room.name
        + ".\n"
        print_slow(format_string_left.format(string=prep_string))
        enter_room.enter_room()


def print_directions():  2 usages
    for i in range(0, len(player_information.current_room.directions)):
        print_slowish(format_string_center.format(string=("▶ " + player_information.current_room.directions[i]
                                                          + " ◀")))
```

move_between_rooms.py

4. Combat system including special rooms and boss battles.

```
def two_battle():  7 usages
    time.sleep(2)
    enemy = player_information.current_room.enemies[0]
    while enemy.alive and player.alive:
        order = two_round(player, enemy)
        print_order = list_to_string(order,  delim: ", ")
        player_hp = (player.name + ":" + str(player.hp) + "/" + str(player.max_hp)
                     + " MAG:(" + str(player.magic) + "/" + str(player.max_magic) + ")")
        enemy_hp = enemy.name + ":" + str(enemy.hp) + "/" + str(enemy.max_hp)
        print_fast(format_string_center.format(string="New round beginning now..."))
        print_fast(format_string_center.format(string="--------------------------"))
        # Print Turn Order so Player can see
        print_fast(format_string_right.format(string="Turn Order:"))
        print_fast(format_string_right.format(string=print_order))
        print_fast(format_string_right.format(string=""))

        # Print HP MP so player can keep track
        print_fast(format_string_left.format(string=player_hp))
        print_fast(format_string_left.format(string=enemy_hp))
```

battle_flow.py

5. No persistent saving or loading. Every run is a fresh start. Currently this information is stored in playerinformation.py, telling the game what has been done and what hasn't been done yet. An example one can see is the current_room, current_weapons and current_items shown below:

```
current_room = entry_hall
current_weapons = player.weapons
current_items = player.items
prisoners_free = False
civilian_healed = False

actions = ["Attack", "Magic", "Item"]
room_actions = ["Move", "Search", "Item", "Wait"]

total_points = 0

format_string_center = '{string:^43}'
def print_actions():  3 usages
    for i in range(0, len(room_actions)):
        print_slowish(format_string_center.format(string=("▶ " + room_actions[i] + " ◀")))
```

player_information.py

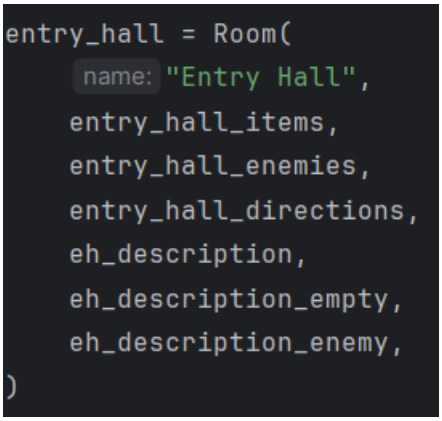
It also shows if one has freed the prisoners in a special room, or healed some civilians in another room.

Code Review Analysis

Structure

The project is organized across several folders, which is good. The logic is modular: combat, rooms, and printing behaviors each live in their own files.

However, all persistent data is embedded directly in Python files:



rooms.py

This makes the game difficult to expand. Adding new rooms, items, or player classes requires editing source code instead of updating external data.

Documentation

Most functions are readable, but there are few docstrings. For example, the combat functions and special-room handlers do not include explanations about expected parameters or return values.

```
def enter_room():  6 usages
    pygame.mixer.music.fadeout(3000)
    pygame.mixer.music.load(audio1)
    pygame.mixer.music.play(-1,  start: 0,  fade_ms: 3000)
    time.sleep(2)
    print_slow(format_string_center.format(string=player_information.current_room.description))
    if not player_information.current_room.empty:
        print_slow(format_string_center.format(string=player_information.current_room.description_enemy))
        print_fast(format_string_center.format(string=""))
        print_fast(format_string_center.format(string=".  .  ."))
        print_fast(format_string_center.format(string=""))
        pygame.mixer.music.fadeout(3000)
        pygame.mixer.music.load(audio2)
        pygame.mixer.music.play(-1,  start: 0,  fade_ms: 3000)
        if len(player_information.current_room.enemies) > 1:
            three_battle()
        elif len(player_information.current_room.enemies) == 1:
            two_battle()
        pygame.mixer.music.fadeout(3000)
        pygame.mixer.music.load(audio1)
        pygame.mixer.music.play(-1,  start: 0,  fade_ms: 3000)
        time.sleep(2)
        player_information.current_room.empty = True
    print_fast(format_string_center.format(string=""))
    print_fast(format_string_center.format(string=".  .  ."))
    print_fast(format_string_center.format(string=""))
    print_slow(format_string_center.format(string=player_information.current_room.description_empty))
    print_fast(format_string_center.format(string=".  .  ."))
    print_fast(format_string_center.format(string=""))
    action_prompt()
```

enter_room.py

This enter_room() function is long but undocumented.

Variables

Variables are named descriptively and consistently, but there are two database-related issues:

1. Player stats and room data are stored in non-persistent dictionaries which cannot be saved across sessions.
2. The game loads every "database" file at runtime meaning data isn't validated, normalized, or stored efficiently.

```
current_room = entry_hall
current_weapons = player.weapons
current_items = player.items
prisoners_free = False
civilian_healed = False
```

player_information.py

Loops and Branches

Control flow is clear and functional, but several branches would benefit from early returns for readability. Additionally, functions like enter_room() handle both description printing and battle logic, which could be separated for clarity, especially once persistent room states exist.

```
def enter_room():  6 usages
    pygame.mixer.music.fadeout(3000)
    pygame.mixer.music.load(audio1)
    pygame.mixer.music.play(-1, start: 0, fade_ms: 3000)
    time.sleep(2)
    print_slow(format_string_center.format(string=player_information.current_room.description))
    if not player_information.current_room.empty:
        print_slow(format_string_center.format(string=player_information.current_room.description_enemy))
        print_fast(format_string_center.format(string=""))
        print_fast(format_string_center.format(string=".  .  ."))
        print_fast(format_string_center.format(string=""))
        pygame.mixer.music.fadeout(3000)
        pygame.mixer.music.load(audio2)
        pygame.mixer.music.play(-1, start: 0, fade_ms: 3000)
        if len(player_information.current_room.enemies) > 1:
            three_battle()
        elif len(player_information.current_room.enemies) == 1:
            two_battle()
        pygame.mixer.music.fadeout(3000)
        pygame.mixer.music.load(audio1)
        pygame.mixer.music.play(-1, start: 0, fade_ms: 3000)
        time.sleep(2)
```

enter_room.py

Defensive Programming

This is the area with the greatest opportunity for improvement, especially since the project is meant for Databases.

Problems include:

1. Data resets every run
   If the player quits after beating three bosses, all progress is wiped.
2. No input sanitization for player name or character creation
   In a database environment, unsanitized input could cause injection or malformed records.
3. Hard-coded data means typo-based bugs
   If a room name is misspelled in one file, the game crashes or becomes unreachable.
4. No database schema
   All structures are loose dictionaries with no constraints.

Target Areas for Improvement

Here are the main areas where the game can be strengthened:

- Convert the in-memory structures into an SQL database:
  - Rooms table
  - Player table
  - Inventory table
  - Enemies table
- Add persistent save and load functionality:
  - Let players quit and resume.
- Normalize player data:
  - Keep stats consistent between sessions.
- Add database validation:
  - Ensure rooms, stats, and items follow consistent schema rules.
- Separate business logic from data access:
  - Introduce a database_service.py that handles all SQL queries.

Planned Enhancements

Here is what I plan to implement:

1. Introduce SQLite as the main database:

   From the Python standard library.

   The game will create SQL tables on first run and then populate them.

2. Convert dictionaries into database tables

   Examples:

   - rooms table with:
     - id
     - name
     - description
     - exits
     - special_flags
   - player table with:
     - name
     - hp
     - strength
     - weapon
     - location
   - enemies table with:
     - name

    - stats
    - room_id

3. Save/load game system:

    Players can:

    - Start new game
    - Continue last save
    - Load multiple save slots

4. Create a data access layer:

    Replace direct file imports with calls like:

    db.get_player()

    db.update_player_location(new_room)

    db.get_room(room_id)

5. Input sanitization

    Sanitize all user inputs before writing to the database.

6. Improve documentation

    Add docstrings to describe behavior, return values, and data interactions.

Skills Demonstrated

By enhancing this artifact, I will demonstrate:

- Database integration: designing schema and implementing SQL CRUD operations
- Data Modeling: organizing player and room data relationally
- Persistent storage implementation: enabling save/load
- Refactoring legacy code: separating logic from data

- Input validation and sanitization: improving safety and reliability
- Documentation improvements: making complex systems easier to maintain

Alignment with Course Outcomes

This enhancement aligns with the following:

- **Outcome 4**

  Use well-founded and innovative techniques, skills, and tools in computing practices.

- **Outcome 5**

  Develop a security mindset that anticipates exploits, which is especially important when adding sanitization and controlling database access.

- **Outcome 3**

  Design and evaluate computing solutions using CS principles, specifically, designing a database schema to solve the problem of non-persistent game data.

Conclusion

To wrap up, my Python Heroes & Villains adventure game is a strong early artifact that demonstrates branching logic, modular design, and interactive storytelling. However, it currently lacks any persistent data handling, which limits both gameplay depth and scalability.

By integrating SQLite, reorganizing data structures, and implementing a proper save/load system, I will transform the project into a much more professional artifact. This enhancement

mirrors real-world software development where applications must store user progress, maintain data integrity, and reliably reload state across sessions.

These improvements show my growth as a developer, from simple scripting in IT-140 to full database-backed application design, and they align directly with industry practices for data modeling, persistence, and secure input handling.