Hi, my name is Kendal, and today I'll be reviewing my code in three key areas: software engineering and design, algorithms and data structures, and databases. This review is part of my CS 499 project, and I'll also be discussing enhancements that I plan to make to improve the code and align it with professional standards. Let's begin with the first category.

I'll start by outlining the functionality of the existing code and how it meets its current requirements. Then, I will use a detailed checklist to analyze the code, focusing on areas like structure, documentation, and variables.

As part of the enhancement plan, I'll demonstrate how the project will evolve into a full-stack web application. This will include integrating a MongoDB database for improved data management and implementing optimized algorithms for better performance.

Category One: Software Engineering and Design

Section 1: Existing Code Functionality

Description of the Code:

The first category I'll be reviewing is Software Engineering and Design. In this section, I'll provide an overview of the existing code, which is the artifact that I worked on during my first semester at SNHU. The existing code is a Python program designed for rental application evaluation. It takes input data from landlords or agents, such as the applicant's income, credit score, and employment length, and assigns a grade to each applicant. The program processes each application by taking in key applicant details. Once all information is provided, the code applies a decision-making algorithm to evaluate the application and classify it into one of three categories: approved, conditionally approved, or rejected.

Key Functions and Modules:

Let's walk through some of the essential functions and modules.

- The **input handling function** is where user input is gathered. It collects key applicant data such as income, credit score, and employment length.
- The **evaluation function** is responsible for taking the input and applying the decision-making logic. This function calculates an overall score based on predefined weightages.
- The **decision function** compares the score against thresholds to classify the applicant as approved, conditionally approved, or rejected.
- The **output function** displays the decision back to the user. Currently, this output is printed directly to the console, but it will eventually be integrated into a frontend interface in the full-stack version.

Interaction with Components:

Although the code currently operates as a standalone script, its functionality can be easily integrated into a web-based application. In the planned full-stack version, the input data would

come from user forms in a frontend interface, and the output would be displayed on the webpage. The backend will handle the logic and store results in a database.

Section 2: Code Review Criteria

Structure:

- **Design Implementation:** The code correctly implements the design for processing and evaluating rental applications based on criteria like income, credit score, and employment length. However, the structure needs to evolve into a more modular architecture when integrated into a full-stack application.
- **Coding Standards:** It conforms to Python best practices, but once the code is refactored for the full-stack setup, additional standards like RESTful practices will need to be enforced.
- **Style and Formatting:** The code is consistent in its style, but further enhancements can be made to improve readability, such as separating business logic into modules.
- **Unnecessary Code:** There are no unnecessary procedures or unreachable code. All routines are functioning as expected, but some logic could be refactored into reusable components.
- **Test Routines:** There are no leftover stubs or test routines that need to be removed.
- **Reusable Components:** As the code grows, external libraries (for data validation or form handling) can replace some parts of the current logic.
- **Repeated Code:** There's minimal repetition, but functions related to scoring and decision-making could be abstracted into reusable modules
- **Storage Efficiency:** As of now, data is not stored, but this will be addressed by integrating a database in the enhancement phase.
- **Symbolics vs. Magic Numbers:** The code does use some hardcoded numbers for scoring ranges. These should be refactored into constants or configuration variables to make future changes easier.
- **Module Complexity:** The current program is simple, but when scaling up, it would benefit from breaking it into separate modules (e.g., input handling, data processing, decision logic).

Documentation:

- **Commenting Style:** Comments are minimal, and a clearer commenting structure should be added, especially as we expand this into a more complex system. Adding docstrings for functions will make it easier to maintain and enhance the code in the future.
- **Consistency:** There are no commentes in the code but detailed explanations of the algorithm and decision-making processes would benefit future developers working on the project.

Variables:

- **Variable Names:** All variables are defined with clear and meaningful names.

- **Type Consistency:** Since this is a basic Python program, variable types are consistent. When scaling up, more type validation may be needed, especially when working with form inputs in the frontend.
- **Unused Variables:** There are no redundant or unused variables in the current implementation.

Arithmetic Operations:

- **Floating-Point Comparisons:** The program does not compare floating-point numbers for equality, so no issues are present in this area.
- **Rounding Errors:** There are no operations that could lead to rounding errors currently, but this will need to be revisited if any financial calculations (e.g., rent calculations) are introduced.
- **Addition and Subtraction Magnitudes:** No major arithmetic issues at this time, but as calculations become more complex, proper testing will be important.
- **Divisor Tests:** There is no division performed in the current code, but testing for zero values in future iterations will be necessary.

Loops and Branches:

- **Loop Completeness and Correct Nesting:** The loops and branches in the program are correctly implemented. There are no nested loops, but they should be kept simple as complexity increases.
- **IF-ELSE Chains:** The code handles all cases well within its IF-ELSE structure, but as more conditions are added, it may benefit from a cleaner structure, such as using a `switch` or `case` statement where applicable.
- **Termination Conditions:** The loops have obvious and reachable termination conditions.

Defensive Programming:

- **Input Validation:** The current code doesn't have extensive validation for inputs. For a full-stack app, input validation should be a priority to ensure all data is correct before processing.
- **Array/Pointer Bounds:** Since arrays or pointers aren't used yet, there's no risk here, but this will change once more complex data structures are introduced.
- **Output Variables:** All outputs are correctly assigned and handled within the program.

Category Two: Algorithms and Data Structures

Section 1: Existing Code Functionality

Decision-Making Algorithms:

The key algorithm in the code revolves around scoring and decision-making. The scoring algorithm processes applicant data (income, credit score, employment length) and assigns

weights to each, summing them into a total score. This score is then compared against predefined ranges to determine whether the applicant is approved, conditionally approved, or rejected.

The decision-making process is straightforward but will need to be adapted for more complex decision-making as additional criteria are added in the enhancement phase.

Computational Tasks:

At the moment, the program performs basic calculations (e.g., summing scores) and makes decisions based on these computations. There is no searching or sorting function currently implemented. However, once the program is scaled to handle multiple applications, sorting algorithms may be needed to prioritize or filter applications based on certain criteria (e.g., applicants with the highest scores).

Section 2: Code Review Criteria

Structure:

- **Design Implementation:** The current design works as intended, evaluating applications and making decisions based on preset criteria.
- **Coding Standards:** The algorithms are written clearly but optimization will be needed once the application scales.
- **Code Complexity:** The decision-making algorithm is simple, but as new factors are added, the logic will need to be restructured.

Documentation:

- **Commenting Style:** The algorithms are functional, but commenting should be expanded to explain how weights are calculated and why certain decisions are made.

Variables:

- **Variable Names:** Variables like `income`, `credit_score`, and `employment_length` are clear and consistent. As the algorithm expands, the use of more descriptive names for newly introduced parameters will be important.
- **Unused Variables:** No unused or redundant variables were found.

Arithmetic Operations:

- **Floating-Point Comparisons:** No floating-point comparisons are done, but this will need to be monitored when adding more financial calculations or thresholds.
- **Divisor Testing:** There is no currently testing for zero or noise, it will need to be added as part of the enhancements

Loops and Branches:

- **Correctness and Nesting:** The decision-making IF-ELSE structure is correct, but as complexity grows, a more structured approach (such as using a decision tree) could help.
- **All Cases Covered:** All conditions are accounted for, but adding additional criteria may require expanding or restructuring the logic.

Defensive Programming:

- **Input Validation:** Currently, the algorithm assumes valid inputs. Defensive programming techniques will need to be added to handle edge cases, incorrect inputs, or missing data.
- **Bounds Testing:** As more complex data structures are introduced, bounds testing will become critical for maintaining the integrity of the algorithm.
- **Memory Management:** No issues with memory management exist since the program is simple. This will change as it scales, especially when working with large datasets in the future.

Category Three: Databases

Section 1: Existing Code Functionality

Current Logic:

Right now, the program does not store any data—it processes information in real-time and displays results without saving them. As part of the full-stack conversion, I will implement a database system, likely using MongoDB, to store applicant details, decisions, and other relevant information. This will enable features like saving applications for later review, filtering based on criteria, and potentially even providing analytics for landlords.

Limitations Without a Database:

Without a database, the current setup is extremely limited. Each application is processed independently, with no way to retrieve previous results or analyze multiple applications. This makes it less efficient, especially in scenarios where multiple landlords or agents are managing a large volume of applicants. Adding a database will significantly enhance its functionality, allowing the system to handle large datasets, store information securely, and provide a more dynamic user experience.

Section 2: Code Review Criteria

Structure:

- **Design Implementation:** The current code doesn't interact with any data storage system. The integration of a MongoDB database will completely change how data is stored, retrieved, and processed.

- **Code Efficiency:** The efficiency of database interactions will be key, especially when scaling up to handle many applications. Proper indexing and optimized queries will be necessary.

Documentation:

- **Commenting:** Clear comments will need to be added, especially around database queries and interactions. This is crucial when troubleshooting or debugging database operations.

Variables:

- **Variable Names:** Once database integration happens, variables like database connection strings and query results will need clear, descriptive names to avoid confusion.
- **Type Consistency:** Ensuring type consistency between database fields and the code is crucial for avoiding errors during data insertion or retrieval.

Loops and Branches:

- **Correctness:** Loops or logic handling database queries will need to be correctly structured to avoid unnecessary query repetition, especially in larger datasets.

Defensive Programming:

- **Input Validation:** Strict validation of input data will be required before it is stored in the database to prevent invalid data or potential security vulnerabilities like injection attacks.
- **Memory Management:** Efficient memory management is necessary when handling large datasets to prevent memory leaks or slow query responses.
- **Error Handling:** Proper error handling will need to be integrated to manage issues like connection timeouts, failed queries, or missing records.

Category One: Software Engineering and Design

Enhancement Plan: To enhance the software engineering and design aspects of the code, we will refactor the code to create modular, reusable functions and improve overall code structure. This will involve consolidating repeated code blocks and improving storage efficiency. We will also standardize documentation practices to ensure clear and consistent comments across the codebase. Additionally, we will implement collaborative tools and version control systems to support teamwork and facilitate better code review processes, thereby fostering a collaborative environment (Course Outcome 1). These changes will enhance clarity and maintainability, aligning with Course Outcome 2 and improving the design and evaluation of computing solutions (Course Outcome 3).

Category Two: Algorithms and Data Structures

Enhancement Plan: To enhance algorithms and data structures, we will implement more efficient sorting and searching algorithms and design suitable data structures to manage and store

data effectively. We will also incorporate performance analysis to ensure scalability with larger datasets. In addition, we will integrate error handling and validation mechanisms to address potential vulnerabilities, ensuring robust and secure data processing (Course Outcome 5). These improvements will demonstrate the application of algorithmic principles and innovative techniques, aligning with Course Outcomes 3 and 4. We will also use collaborative development tools to work effectively with team members on these enhancements (Course Outcome 1).

Category Three: Databases

Enhancement Plan: For the database category, we will integrate MongoDB to provide persistent data storage and enhance the functionality of the application. We will design and implement a database schema to manage data efficiently and document the database integration process thoroughly. To ensure data security and address potential vulnerabilities, we will implement secure data handling practices and include robust error handling mechanisms (Course Outcome 5). We will also employ collaborative tools to facilitate teamwork during the database integration and ensure that our approach aligns with best practices in data management (Course Outcome 1). These enhancements align with Course Outcome 3 by designing solutions that leverage database technology and Course Outcome 4 by applying industry-standard data management practices