Problem 1

1) I would choose the Role-Based Access Control (RBAC) model for designing the access control mechanism. RBAC enables a straightforward mapping of roles to permissions, making it ideal for managing the diverse roles and access levels required by JustInvest System. Additionally, it simplifies administration by allowing roles to be easily assigned to users
2)

Role	Permission
Client	ViewAccountBalance, ViewInvestments, ViewContactDetails
PremiumClient	ModifyInvestmentPortfolio, ViewFinancialPlannerDetails, ViewInvestmentAnalystDetails
Employee	ViewAccountBalance, ViewInvestments
FinancialPlanner	ViewMoneyMarketInstruments, ModifyInvestmentPortfolio
FinancialAdvisor	ViewMoneyMarketInstruments, ModifyInvestmentPortfolio ViewInterestInstruments, ViewPrivateConsumerInstruments
Teller	(Restricted to business hours)

3) The test cases provide **adequate coverage** of the access control policy by addressing the following aspects:
    1. **Role-Specific Permissions**:
        - Each defined role is tested to ensure it has access only to its assigned operations.
        - Granular testing of included (assertIn) and excluded (assertNotIn) permissions ensures proper access control.
    2. **Edge Cases**:
        - The system's behavior for invalid or unknown roles is tested, confirming it handles unexpected inputs gracefully.

3. **Security and Integrity**:
    - By testing unauthorized operations for roles (e.g., test_client_permissions checks that clients cannot modify investments), the test cases ensure the RBAC system enforces strict boundaries between roles.
4. **Maintainability**:
    - The tests can be extended to cover new roles or modified permissions with minimal changes, reflecting the extensibility of the RBAC implementation.
5. **Error Handling**:
    - Handling of undefined or unexpected roles (e.g., test_no_permissions_for_unknown_role) ensures robustness against invalid input.

There is another test case called test_list_users.py to run a test of the users given on assignment pdf like sasha please use python3 test_list_users.py

Problem 2

Chosen Hash Function: SHA-256

Justification:

- **Security Strength**: SHA-256 provides a high level of security with a 256-bit hash output, making it resistant to brute-force attacks.
- **Widely** Used: SHA-256 is widely used and has undergone extensive scrutiny in the cryptographic community.
- **Availability**: The VM environment supports SHA-256 through the Hashlib library in Python.

Chosen Salt Library: secretes (of length 16)

Justification:

- **Security**: A salt length of 16 bytes provides a sufficient security margin, making it computationally infeasible for an attacker to precompute rainbow tables or hash collisions.
- **Cryptographic** Standard: The length of 16 bytes aligns with common cryptographic practices.
- **Compatibility**: A salt length of 16 bytes is compatible with the SHA-256 hashing algorithm, ensuring that the salted password remains compatible with the chosen hash function

Design Password File Record Structure:

Each record in the password file will consist of the following fields:

• User ID: Unique identifier for each user.

• Salt: Randomly generated salt for password hashing.

• Hashed Password: Result of hashing the user's password with the salt.
• Role: The role associated with the user.
Example Record: leslie:randomSaltValue:hashedPasswordValue:Client

Hash Function for Password Security breakdown

The hash_password function is designed to securely hash a user's password by combining it with a unique salt. This is done to ensure that even if two users have the same password, their hashed results will be different. Here's a breakdown of how it works:

1. **Imports**:
    - **hashlib**: This library provides a set of cryptographic hash functions, including the SHA-256 algorithm, which we use to securely hash the password.
    - **secrets**: A library designed to generate cryptographically strong random numbers, used here to create a secure salt.
2. **Generate Salt**:
    - We generate a salt using secrets.token_hex(16), which produces a random 16-byte salt. This ensures that even if two users have the same password, their salts—and therefore their hashed passwords—will differ.
3. **Hashing the Password**:
    - We use hashlib.sha256() to create a hash object. This object will store the hashed version of the password.
    - We then update this object by passing the concatenation of the password and the salt, both encoded in UTF-8.
4. **Hex Digest**:
    - After updating the hash object, we call .hexdigest() to convert the resulting hash into a readable hexadecimal string. This is the hashed password we store in the password file.
5. **Return the Hashed Password**:
    - Finally, the function returns the hexadecimal representation of the hashed password, which can be safely stored and later compared during the login process.

Password File Record Structure

The PasswordFile class is responsible for managing user records stored in a file. It provides methods to add and retrieve user records securely.

1. **Class: PasswordFile**:

    - The class handles all operations related to user authentication data. It ensures that passwords are securely hashed and that user records are properly added and retrieved.
2. **Method: add_record**:
    - This method is used to add a new user record to the password file. It takes the following inputs:
        - **User ID**: The username of the user.
        - **Salt**: A randomly generated value that is combined with the password before hashing.
        - **Hashed Password**: The securely hashed password.
        - **Role**: The user's role (e.g., "Client", "Admin").
    - These values are appended as a new line in the password file, storing them in a simple format for later retrieval.
3. **Method: get_record**:
    - This method is used to retrieve a user's record from the password file.
    - It reads through the file, line by line, and splits each line into its component parts (user ID, salt, hashed password, and role). If it finds a matching user ID, it returns a dictionary with the user's details.

Testing the Password File Usage

The goal of the test case is to ensure that the password file mechanism works as intended. Specifically, we want to verify that the PasswordFile class can correctly add a user record to the file and retrieve it.

Test Steps:

1. **Initialize Password File**:
    - First, we create an instance of the PasswordFile class, which represents the password file management system. This will allow us to interact with the password file.
2. **Test if User Exists**:
    - Before adding a new record, we check whether a user with the given ID already exists. This step ensures we don't accidentally add duplicate records. The get_record method will return None if no record exists.
3. **Add Record**:
    - We call the add_record method to add a new user record. The record will include:
        - **User ID**: leslie
        - **Salt**: A sample salt value like randomSaltValue

        - **Hashed Password**: A hash of a test password (e.g., hashedPasswordValue)
        - **Role**: "Client"
4. **Get Record**:
    - We retrieve the user record using the get_record method, passing in the user ID leslie.
5. **Assertion**:
    - We compare the retrieved record with the expected values:
        - **User ID**: leslie
        - **Salt**: The salt used in the add_record method
        - **Hashed Password**: The hashed password generated during the test
        - **Role**: "Client"
6. If the retrieved record matches the expected record, the test case passes. Otherwise, an assertion error will be raised, indicating that the test has failed.

Problem 3

Description of Test Cases

1. **Proactive Password Checker Tests**
    - **Valid Password Test**: Verifies that a password meeting all security requirements passes the check.
    - **Password Length Tests**:
        - Too Short: Ensures passwords shorter than the minimum length are rejected.
        - Too Long: Ensures passwords exceeding the maximum length are rejected.
    - **Character Composition Tests**:
        - No Uppercase: Validates rejection of passwords without at least one uppercase letter.
        - No Lowercase: Validates rejection of passwords without at least one lowercase letter.
        - No Digit: Validates rejection of passwords without at least one numeric character.
        - No Special Character: Ensures passwords without special characters are rejected.

    - **Matching Username Test**: Ensures passwords identical to the username are disallowed.
    - **Weak Password Test**: Verifies that common weak passwords, like "password," are rejected.
2. **Enrollment Mechanism Tests**
    - Integrated with the password validation tests. A password must pass all the checks before user registration proceeds. The username's role assignment and permissions retrieval post-registration ensure functional coverage.

Adequate Coverage

- **Boundary Conditions**: Tests for password length capture edge cases (minimum and maximum thresholds).
- **Composition Rules**: Tests ensure that every password rule (uppercase, lowercase, digits, special characters) is independently validated.
- **Special Cases**: Cases like passwords matching the username or appearing in the weak password list handle likely user errors.
- **Integration**: By combining the enrollment mechanism with password validation, the system tests functional interdependencies, such as saving user records only when password rules are satisfied.

Problem 4

Test Cases

1. **test_failed_login_incorrect_password:**
    - **Purpose:** Verifies that the system denies login attempts when a user provides the wrong password.
    - **How:**
        - Mocks the behavior of the PasswordFile.get_record method to return user data with a different hashed password than the one provided during login.
        - Asserts that the login attempt returns None, indicating failure.
    - **Coverage:** Ensures the system correctly handles password mismatches.
2. **test_failed_login_nonexistent_user:**
    - **Purpose:** Checks that login attempts with a non-existent username fail.
    - **How:**

        - Mocks the get_record method to return None, simulating that the username does not exist in the password file.
        - Asserts that the login returns None, confirming failure for unknown users.
    - **Coverage:** Confirms that the system does not authenticate invalid or unregistered users.
3. **test_get_record_file_not_found:**
    - **Purpose:** Tests the behavior when the password file cannot be found.
    - **How:**
        - Tries to create a PasswordFile instance with a non-existent file name.
        - Expects a FileNotFoundError to be raised.
    - **Coverage:** Ensures the system gracefully handles missing password files.
4. **(Commented out) test_successful_login:**
    - **Purpose:** (If enabled) Verifies successful login with correct credentials.
    - **How:**
        - Uses the mocked get_record method to return correct user data, including the expected hashed password.
        - Asserts that the returned role matches the user's expected role.
    - **Coverage:** Ensures that valid credentials are accepted and roles are correctly assigned.

Adequacy of Coverage

- **Positive Cases:** (e.g., test_successful_login, if uncommented) Verify that the system works as expected for valid scenarios.
- **Negative Cases:** (e.g., incorrect passwords, non-existent users, file not found) Cover failure modes to ensure robustness.
- **Boundary Conditions:** Includes tests for handling missing files and ensures no undefined behavior occurs.

List of test users and there passwords