ATM Simulator System

Introduction:

The **ATM Simulator System** is a console-based application developed entirely in Python designed to emulate the core functionalities of an Automated Teller Machine (ATM). This project serves as a practical demonstration of fundamental programming concepts, data structure management, and procedural flow control within a financial simulation context.

Purpose and Scope

The primary goal of this simulator is to provide a user-friendly, text-based interface where users can perform essential banking transactions. The system is designed for a single, hardcoded user account, focusing on functional integrity over large-scale database management.

Key functionalities demonstrated include:

- **Secure Authentication:** Requiring a username and a 4-digit PIN with built-in attempt limits.
- **Transaction Management:** Facilitating both cash **Withdrawals** and **Deposits (Lodge)** with immediate balance updates.
- **Account Inquiry:** Displaying a comprehensive **Account Statement** including the current balance and transaction history.
- **Security Feature:** Allowing users to **Change their PIN** after successful authentication.

Technical Foundation

The project is built using **Python**, leveraging its simplicity and robust built-in libraries. Due to the project's foundational nature, all user data (balance, PIN, and transaction history) is stored **in-memory** using a Python dictionary. This design choice highlights how data is structured and manipulated programmatically, serving as a stepping stone toward future expansion with persistent database integration (e.g., SQLite or MySQL).

Objectives

This project aims to fulfill several core educational and functional objectives in the context of console-based Python application development:

1. Functional Objectives

- **Secure User Access:** To implement a reliable sign-in mechanism that requires validation of a hardcoded username and PIN, complete with a limited number of **attempt limits** to simulate basic security measures.
- **Core Banking Transactions:** To successfully manage the flow of funds by implementing Withdrawal (debit) and Deposit (credit) functions, ensuring the account balance is updated immediately and accurately after each successful transaction.
- **Overdraft Prevention:** To enforce financial logic by including a robust **Insufficient Funds Check** during the withdrawal process, preventing the balance from dropping below zero.
- **Comprehensive Reporting:** To provide an easily accessible **Account Statement** function that displays the current balance and logs all transaction history (type, amount) performed during the session.

2. Educational and Technical Objectives

- **Mastering File-less Data Management:** To demonstrate proficiency in using Python's native data structures (specifically **dictionaries and lists**) to store and manage volatile, in-memory data, including structured transaction records.
- **Procedural Programming Proficiency:** To break down a complex system into smaller, manageable, and reusable functions (e.g., sign_in, withdraw_amount, change_pin), improving code organization and readability.
- **Input Handling and Validation:** To practice using **try/except blocks** and if/else logic to validate all user inputs, ensuring that only positive, numeric values are processed for transactions and that the PIN remains a 4-digit string.
- **Console Interface Design:** To create a clear, intuitive, and user-friendly **command-line interface (CLI)** using print statements to guide the user through the menu and provide informative feedback on transaction outcomes.

Scope of the ATM Simulator Project

The scope of the ATM Simulator project clearly defines the boundaries of its functionality and technical implementation. It specifies what features are included to meet the defined objectives and, equally important, what aspects are intentionally excluded as out-of-scope for this beginner-level console application.

In-Scope Features (What the project includes)

Category	Included Functionality
Authentication	Sign-in process validating a single, predefined Username and 4-digit PIN.
Account Management	Functions for viewing the current balance and transaction history.
Financial Logic	Capability to perform withdrawals (debits) and deposits (credits).
Security Logic	Input validation to prevent withdrawals exceeding the current balance (Overdraft Protection).
User Interface	A clear, text-based Command Line Interface (CLI) menu system.
Maintenance	Ability to change the 4-digit PIN within the current session.
Data Storage	All account data and transaction history are stored in in-memory Python variables (dictionaries/lists).

Out-of-Scope Limitations (What the project excludes)

The following items are deliberately excluded from this project's scope to maintain a focus on core programming fundamentals:

- **Database Integration:** No external files (e.g., .txt, .csv) or database connections (e.g., SQLite, MySQL) are used for persistent data storage. All changes are temporary.
- **Multi-Account Support:** The system only supports one single, hardcoded user account. There is no functionality for account creation or managing multiple users simultaneously.
- **Physical Constraints:** No integration with real-world ATM hardware, printers, card readers, or networked servers.
- **External APIs/Internet:** No connection to external banking APIs or the internet for real-time rates or verification.
- **Graphical User Interface (GUI):** The application is strictly console-based; no graphical libraries (like Tkinter or PyQt) are used.
- **Advanced Features:** Does not include features like funds transfers, bill payments, balance alerts, or transaction fees.

Technology Stack:

The ATM Simulator is intentionally built on a minimal and accessible technology stack, focusing on core programming concepts rather than complex external dependencies. This approach ensures maximum learning value and easy execution.

Core Technologies

| Component | Technology | Rationale |
| Primary Language | Python | Selected for its clear syntax, readability, and speed of development, making it ideal for console-based applications and teaching fundamental algorithms. |

Operating System	Console/Terminal	The application is designed to run directly within the standard command-line interface (CLI) of any major operating system (Windows Command Prompt, macOS Terminal, Linux Shell).
Libraries/Modules	Built-in Python Modules (sys)	Only standard, pre-installed Python modules are used. The sys module is specifically utilized for controlled program termination (sys.exit()).
User Interface (UI)	Text-Based Interface (CLI)	The interface relies exclusively on input() and print() statements, avoiding any graphical libraries (e.g., Tkinter, PyQt) as defined in the project scope.

Data Management Stack

As specified in the project scope, the data stack is limited to in-memory native Python structures:

Component	Technology	Rationale
Account Data Storage	Python Dictionary	A dictionary is used as the central repository (ACCOUNT_DATA) to store key-value pairs like username, PIN, and balance.
Transaction History	Python List (of Dictionaries)	A list is used within the account data to sequentially record individual transactions, demonstrating basic data logging.
Data Persistence	(None)	There is no file, database, or network connection used. All data is temporary and exists only for the duration of the program session.

This lean technology stack fulfills the project's objective of focusing entirely on internal logic, state management, and algorithmic execution.

Methodology:

The development of the ATM Simulator followed a straightforward, procedural methodology, focusing on rapid iteration and console-based programming best practices. The entire system operates based on state changes within a central, in-memory data structure.

1. Planning and Data Modeling

- **Data Definition:** A single Python dictionary (ACCOUNT_DATA) was designed to hold the entire state of the bank account, including the username, PIN, balance (float), and a history of all transactions (list of dictionaries).
- **Authentication Flow:** The sign-in process was modeled as a sequential verification step (Username $\rightarrow$ PIN) with explicit constraints (3 attempts limit) enforced using nested for loops.
- **Function Mapping:** Each feature listed in the objectives (Withdrawal, Deposit, Statement, Change PIN) was mapped directly to a separate, modular Python function to ensure clear separation of concerns.

2. Implementation Strategy (Top-Down Approach)

The system was built using a top-down approach, starting with the main control structure and building out the specialized functions:

1. **Main Loop (main()):** Implemented the core while True loop to continuously display the menu options (1-5) and handle user input, acting as the central control flow manager.
2. **Authentication (sign_in()):** Developed the robust authentication logic first, ensuring the program could only proceed to the main menu upon successful login.
3. **Core Transaction Functions (withdraw_amount, deposit_amount):** Implemented the transactional logic, prioritizing:
    - Input conversion and error handling using **try...except blocks**.
    - Business rule enforcement (e.g., if amount > balance for overdraft prevention).
    - State update: Directly manipulating the global ACCOUNT_DATA['balance'].
4. **Reporting and Maintenance:** Created functions for viewing the statement and changing the PIN, which primarily involve reading from and writing to the ACCOUNT_DATA structure.

3. Error Handling and Validation

Validation was implemented aggressively at the point of user input for all functions:

- **Data Type Integrity:** try...except blocks were employed to catch ValueError exceptions if the user entered non-numeric characters when a float (for amounts) was expected.
- **Logical Constraints:** if/else statements were used to enforce business rules:
    - Withdrawal amount must be greater than zero.
    - Withdrawal amount must be less than or equal to the balance.
    - New PIN must be a 4-digit string.

4. Testing and Review

- **Unit Testing (Informal):** Each function was tested individually in the console environment immediately after implementation to verify that it performed its specific task correctly (e.g., verifying withdraw_amount correctly reduced the balance and logged a transaction).
- **Scenario Testing:** Comprehensive usage scenarios were executed, including testing failure modes like entering the wrong PIN multiple times, attempting to withdraw more than the balance, and entering alphabetical characters during a transaction.
- **Code Review:** The code was reviewed for clarity and adherence to the principle of "file-less data management," ensuring no external I/O operations were inadvertently introduced.

System Architecture:

The ATM Simulator utilizes a **Single-Tier, Procedural Architecture**. This design minimizes complexity by integrating all components—the User Interface, Business Logic, and Data Management—into a single, self-contained Python script running entirely within the console environment.

Architectural Diagram Summary

1. Architectural Model: Single-Tier

In a single-tier architecture, the presentation, business logic, and data layers are not physically separated.

- **Execution Environment:** The application runs as a single process on the user's local machine (the console/terminal).
- **Benefits:** Simplicity, ease of debugging, and minimal setup requirements, perfectly suiting a beginner-level project.
- **Data Flow:** The user's input directly triggers the business logic, which immediately reads from or writes to the in-memory data structure.

2. Component Breakdown and Interaction

The system is organized into three primary conceptual components, all residing within the same atm.py file:

A. Presentation Layer (Console Interface)

- **Role:** Handles all input/output interactions with the user.
- **Mechanism:** Uses Python's built-in print() to display menus and transaction results, and input() to receive commands, usernames, PINs, and transaction amounts.
- **Entry Point:** Managed by the main() function's continuous while True loop.

B. Business Logic Layer (Functions)

- **Role:** Contains all the functional rules and procedural steps for the ATM.
- **Mechanism:** Implemented through modular Python functions (sign_in(), withdraw_amount(), deposit_amount(), etc.).
- **Key Responsibilities:**
  - **Authentication:** Verifying credentials and enforcing the 3-attempt limit.
  - **Validation:** Checking for overdrafts (if amount > balance) and data type integrity (try/except).
  - **Calculation:** Performing arithmetic on the account balance.

C. Data Layer (In-Memory Storage)

- **Role:** Manages the temporary storage and retrieval of all account state data.
- **Mechanism:** Uses the global Python dictionary, ACCOUNT_DATA.

- **Structure:**
  - **Scalar Data:** Stores simple types like username (string), pin (string), and balance (float).
  - **Complex Data:** Uses a **list of dictionaries** for the transactions history, demonstrating structured data logging.
- **Persistence:** Data changes are volatile and do not persist after the program terminates.

3. Execution Flow Summary

The architecture follows a clear cyclic process:

1. **Start:** The main() function initiates the sign_in() sequence.
2. **Auth Check:** Upon successful authentication, control returns to main().
3. **Command Loop:** The Presentation Layer displays the menu.
4. **Action Request:** The user enters a command (e.g., '2' for withdrawal).
5. **Logic Execution:** The Business Logic layer executes the corresponding function (withdraw_amount()). This function interacts directly with the Data Layer (updates ACCOUNT_DATA['balance']).
6. **Feedback:** The result is passed back to the Presentation Layer using print(), and the menu loop restarts.

Program:

```
import sys

# --- HARDCODED ACCOUNT DATA ---
# Since no database or file handling is required, we use a global dictionary
# to store the account details.
ACCOUNT_DATA = {
    "username": "user123",
    "pin": "1234",
    "balance": 5000.00,
    "transactions": [
        {"type": "Deposit", "amount": 5000.00, "note": "Initial Deposit"}
    ]
}

# --- CORE ATM FUNCTIONS ---

def sign_in():
    """Handles the user sign-in process."""
    max_attempts = 3

    # 1. Username Check
    for attempt in range(max_attempts):
```

```
            user_input = input("Enter Username: ").strip()
            if user_input == ACCOUNT_DATA["username"]:
                break
            elif attempt < max_attempts - 1:
                print("Invalid Username. Please try again.")
            else:
                print("Too many invalid attempts. Exiting.")
                sys.exit()

        # 2. PIN Check
        for attempt in range(max_attempts):
            pin_input = input("Enter PIN: ").strip()
            if pin_input == ACCOUNT_DATA["pin"]:
                print("\nSign In Successful! Welcome back.")
                return True
            elif attempt < max_attempts - 1:
                print("Invalid PIN. Please try again.")
            else:
                print("Too many invalid attempts. Access denied. Exiting.")
                sys.exit()

        return False

    def display_statement():
        """Displays the current account balance and transaction history."""
        print("\n" + "="*40)
        print("          ACCOUNT STATEMENT")
        print("="*40)
        print(f"Current Balance: ${ACCOUNT_DATA['balance']:.2f}\n")
        print(f"{'Type':<10}{'Amount':<15}{'Note':<15}")
        print("-" * 40)

        # Display the last 5 transactions for brevity
        for t in ACCOUNT_DATA["transactions"][-5:]:
            amount_str = f"${t['amount']:.2f}"
            print(f"{t['type']:<10}{amount_str:<15}{t.get('note', ''):<15}")

        if len(ACCOUNT_DATA["transactions"]) > 5:
            print("\nShowing last 5 transactions.")

        print("="*40 + "\n")

    def withdraw_amount():
        """Handles the withdrawal transaction."""
        try:
            amount = float(input("Enter amount to withdraw: $").strip())

            if amount <= 0:
```

```
            print("\nERROR: Withdrawal amount must be positive.")
            return

        if amount > ACCOUNT_DATA["balance"]:
            print("\nERROR: Insufficient funds.")
            return

        # Perform transaction
        ACCOUNT_DATA["balance"] -= amount
        ACCOUNT_DATA["transactions"].append({
            "type": "Withdraw",
            "amount": amount,
            "note": "ATM Withdrawal"
        })

        print(f"\nSUCCESS! Withdrew ${amount:.2f}.")
        print(f"New Balance: ${ACCOUNT_DATA['balance']:.2f}")

    except ValueError:
        print("\nERROR: Invalid amount entered. Please enter a number.")
    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")

def deposit_amount():
    """Handles the deposit (lodge) transaction."""
    try:
        amount = float(input("Enter amount to deposit (lodge): $").strip())

        if amount <= 0:
            print("\nERROR: Deposit amount must be positive.")
            return

        # Perform transaction
        ACCOUNT_DATA["balance"] += amount
        ACCOUNT_DATA["transactions"].append({
            "type": "Deposit",
            "amount": amount,
            "note": "ATM Deposit"
        })

        print(f"\nSUCCESS! Deposited ${amount:.2f}.")
        print(f"New Balance: ${ACCOUNT_DATA['balance']:.2f}")

    except ValueError:
        print("\nERROR: Invalid amount entered. Please enter a number.")
    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")
```

```
def change_pin():
    """Allows the user to change their PIN."""
    old_pin = input("Enter current PIN: ").strip()

    if old_pin != ACCOUNT_DATA["pin"]:
        print("\nERROR: Incorrect current PIN.")
        return

    new_pin = input("Enter new 4-digit PIN: ").strip()

    if not new_pin.isdigit() or len(new_pin) != 4:
        print("\nERROR: New PIN must be exactly 4 digits.")
        return

    confirm_pin = input("Confirm new PIN: ").strip()

    if new_pin != confirm_pin:
        print("\nERROR: New PINs do not match.")
        return

    # Update PIN
    ACCOUNT_DATA["pin"] = new_pin
    print("\nSUCCESS! Your PIN has been successfully changed.")

# --- MAIN PROGRAM EXECUTION ---

def main():
    """Main function to run the ATM console menu."""
    print("="*40)
    print("      WELCOME TO THE ATM SIMULATOR")
    print("="*40)

    # Start with the sign-in procedure
    if sign_in():
        while True:
            print("\n" + "#"*40)
            print("        MAIN MENU")
            print("#"*40)
            print("1. Account Statement")
            print("2. Withdraw Amount")
            print("3. Lodge Amount (Deposit)")
            print("4. Change Pin")
            print("5. Exit")
            print("#"*40)

            choice = input("Enter your command (1-5): ").strip()

            if choice == '1':
```

```
            display_statement()
        elif choice == '2':
            withdraw_amount()
        elif choice == '3':
            deposit_amount()
        elif choice == '4':
            change_pin()
        elif choice == '5':
            print("\nThank you for using the ATM Simulator. Goodbye!")
            sys.exit()
        else:
            print("\nInvalid choice. Please enter a number between 1 and 5.")

# Run the main function when the script starts
if __name__ == "__main__":
    main()
```

Sample Input and Output:

This sample session demonstrates the complete lifecycle of the ATM simulator, including successful authentication, core transactions, and PIN maintenance.

1. Program Start and Sign-In

Input (User Action)	**Output (System Response)**	**Notes**
Start Program	==========	Program initialization.
	WELCOME TO THE ATM SIMULATOR	
	==========	
Input: user123	Enter Username: user123	Successful username entry.
Input: 1234	Enter PIN: 1234	Successful PIN entry.
	Sign In Successful! Welcome back.	Access granted.

2. Main Menu and Transaction Cycle

We start with a **Balance Check**, followed by a **Withdrawal** and a **Deposit**.

Action 1: Account Statement (Command 1)

Input (User Action)	Output (System Response)	Notes
Input: 1	#... MAIN MENU ...#	User selects option 1.
	==========	
	ACCOUNT STATEMENT	
	==========	
	Current Balance: $5000.00	Initial balance shown.
	Type Amount Note	

	Deposit $5000.00 Initial Deposit	
	==========	

Action 2: Withdraw Amount (Command 2)

Input (User Action)	Output (System Response)	Notes
Input: 2	Enter amount to withdraw: $1500	User attempts to withdraw $1500.
	SUCCESS! Withdrew $1500.00.	Transaction executed.

	New Balance: $3500.00	Balance updated instantly.

Action 3: Deposit Amount (Command 3)

Input (User Action)	Output (System Response)	Notes
Input: 3	Enter amount to deposit (lodge): $500	User deposits $500.
	SUCCESS! Deposited $500.00.	Transaction executed.
	New Balance: $4000.00	Balance updated instantly.

3. PIN Change and Final Statement

Action 4: Change PIN (Command 4)

Input (User Action)	Output (System Response)	Notes
Input: 4	Enter current PIN: 1234	User verifies current PIN.
Input: 9999	Enter new 4-digit PIN: 9999	New PIN chosen.
Input: 9999	Confirm new PIN: 9999	PIN confirmed successfully.
	SUCCESS! Your PIN has been successfully changed.	PIN updated to 9999 in memory.

Action 5: Final Account Statement (Command 1)

Input (User Action)	Output (System Response)	Notes
Input: 1	#... MAIN MENU ...#	Final check of the balance and log.
	Current Balance: $4000.00	Final balance after all transactions.

	Type Amount Note	

	Deposit $5000.00 Initial Deposit	
	Withdraw $1500.00 ATM Withdrawal	New withdrawal logged.
	Deposit $500.00 ATM Deposit	New deposit logged.
	==========	

4. Exit

Input (User Action)	**Output (System Response)**	**Notes**
Input: 5	Thank you for using the ATM Simulator. Goodbye!	Program termination.