

CS 111 Lab: ATM Simulator

Learning goal: Practice writing and calling functions to build a multi-step interactive program.

Overview

In this lab, you will build a **simple ATM simulator** in Python.

You'll learn how to:

- Write well-documented, reusable functions.
 - Model real-world state (account balances, deposits, withdrawals, fees).
 - Design a command-line interface using loops and conditionals.
-

Setup

Download the starter code: [atm_simulator.py](#) and [accounts.csv](#)

Open the folder in your editor (e.g., VS Code).

Make sure you put both files in the same directory

Part 0 — Describing accounts.csv

accounts.csv is a csv file with 3 components that you will be using, account_id, name, and balance. The account_id is just the identification number for the specific account that the bank (aka you) will primarily use to identify users. The name and balance columns are for the name of the customer and their current balance in float form.

Part I — Building core ATM functions

Work in small steps. Each function below corresponds to a specific component of the simulation.

Make sure you include a **docstring** that clearly describes:

- The purpose of the function
 - Its parameters (name and type)
 - Its return value
-

Step 1: Loading account data

Use the `csv` module to read accounts from the CSV file.

```
Python
def read_accounts(filename):
    """
    Reads account data from a CSV file.

    Parameters:
        filename (str): Path to the accounts CSV file.

    Returns:
        tuple: Three lists – account IDs, names, and balances.
    """

```

Step 2: Retrieve an account

Write a function that takes the lists from `read_accounts()` and an account ID and returns that account's info.

```
Python
def get_account(account_ids, names, balances, account_id):
    """
    Find and return the account holder's name and balance.

    Parameters:
        account_ids (list[str]): List of account IDs.
        names (list[str]): List of account holder names.
        balances (list[float]): List of account balances.
        account_id (str): The account ID to look up.

    Returns:
        tuple: (name, balance) if found, else (None, None).
    """

```

Step 3: Check balance

```
Python
def check_balance(balance):
    """
    Return the current account balance formatted to two decimals.

    Parameters:
        balance (float): Account balance.

    Returns:
        str: Formatted balance string.
    """

```

Step 4: Deposit

```
Python
def deposit(balance, amount):
    """
    Add a positive amount to the balance.

    Parameters:
        balance (float): Current balance.
        amount (float): Deposit amount (must be > 0).

    Returns:
        float: Updated balance.
    """

```

If the amount is invalid, print a helpful message and return the original balance.

Step 5: Withdraw

```
Python
def withdraw(balance, amount):
    """
    Withdraw a positive amount from balance if funds are sufficient.

    Parameters:
        balance (float): Current balance.
        amount (float): Withdrawal amount.

    Returns:
        float: Updated balance.
    """

```

If the amount exceeds the current balance, print "**Insufficient funds**" and do not change the balance.

Step 6: Apply service fees (optional)

Some ATMs charge a flat \$1.50 fee for each withdrawal.
Implement a function to apply this optional feature:

Python

```
def apply_fee(balance, fee=1.50):
    """
    Deducts a service fee from the balance.

    Parameters:
        balance (float): Current balance.
        fee (float): Fee amount (default 1.50).

    Returns:
        float: Updated balance after fee.
    """

```

Step 7: Saving data (optional)

Write a function to save updated balances back to `accounts.csv`.

Python

```
def save_accounts(filename, account_ids, names, balances):
    """
    Write updated account data back to the CSV file.
    """

```

Checkpoint (end of first class)

By the end of the first class, you should have:

- Implemented and tested all helper functions above.
 - Verified each one in the Python shell (e.g., calling `deposit(100, 50)` returns 150).
-

Part II — Building the ATM simulation

Now you'll tie everything together into a working command-line ATM program.

Step 1: Display menu

```
Python
def display_menu():
    """
    Print available ATM options and return user's choice.
    """

```

Menu example:

1. Check balance
 2. Deposit
 3. Withdraw
 4. Quit
-

Step 2: Main simulation loop

Put this all together by implementing a `simulation()` function

```
Python
def simulation():
```

```
"""
Runs the ATM simulation.
"""
```

This function should:

1. Greet the user and load account data.
2. Prompt for an account ID (and optionally PIN).
3. In a loop:
 - o Display the menu.
 - o Get user input.
 - o Call the appropriate helper functions.
 - o Print results.
 - o Exit on "quit".

Example run:

```
----- Welcome to the ATM Simulator! -----
Enter account ID: 1002
Hello, Blake!
Your balance: $120.50

1. Check balance
2. Deposit
3. Withdraw
4. Quit
Choice: 2
Enter deposit amount: 100
New balance: $220.50
Choice: 4
Goodbye, Blake! Final balance: $220.50
```

Step 3: Testing

Be sure to test:

- Valid deposits and withdrawals
 - Invalid input (negative numbers, non-numeric input)
 - Withdrawal limits (balance cannot go negative)
-

Extensions

If you finish early or want a challenge:

1. **PIN verification:** Add a `pin` column to `accounts.csv` and require it at login.
 2. **Multiple accounts:** Allow switching accounts during the session.
 3. **Transaction log:** Write each transaction (date, type, amount, new balance) to a new CSV file.
 4. **Interest feature:** Automatically apply 1% monthly interest using an `apply_interest()` function.
-

Submitting your work

Submit:

- `atm_simulator.py`
 - `accounts.csv`
-

Reflection questions

1. Why is it better to break code into separate functions instead of writing everything in one loop?
2. Which part of your design was easiest to test? Hardest?
3. What are some potential problems that could arise with how we designed this simulator?