**Banking Service Application Documentation**

**Submitted By:**

Table of Contents

[**1. Introduction and Assumptions** 1](#_Toc162032109)

[**2. Design of the Program** 1](#_Toc162032110)

[**Pseudocode** 2](#_Toc162032111)

[**Flowcharts** 3](#_Toc162032112)

[**3. Programming Concepts with Source Code Explanation** 5](#_Toc162032113)

[**4. Screenshots of Sample Input/Output and Explanation** 21](#_Toc162032114)

[**5. Conclusion** 24](#_Toc162032115)

[**6. References** 24](#_Toc162032116)

# **1. Introduction and Assumptions**

The Banking Service Application is designed to provide a system for customer registration, login, and account management, along with administrative functions for managing customer details and super user functions for system administration. The assumptions made in designing this system include:

Customers will provide valid information during registration and login.

Admin staff will have appropriate permissions to update customer details.

Super users will have elevated privileges for system administration.

Data integrity and security are crucial aspects of the system.

# **2. Design of the Program**

## **Pseudocode**

**Main function:**

Initialize variables and constants.

Load existing customer and admin data from files.

Display welcome message and main menu options.

Repeat until user chooses to exit:

Accept user choice from menu.

Perform action based on user choice.

**Function for Customer Registration:**

Prompt user to enter customer details (name, account type, initial balance, password).

Validate user input:

Ensure name is not empty.

Ensure account type is either "Savings" or "Current".

Ensure initial balance is a valid positive float.

Ensure password meets security requirements.

Generate unique account number:

If file with account numbers exists:

Read last account number and increment it.

Else:

Start from default account number.

Save customer details (account number, name, account type, balance, password) to file.

Display success message.

**Function for Customer Login:**

Prompt user to enter account number and password.

Search for matching account number and password in customer data file.

If match found:

Provide access to customer menu options.

Else:

Display error message and prompt for reentry.

**Customer Menu Options:**

Display menu options (Deposit, Withdraw, Generate Statement of Account, Logout).

Repeat until user chooses to logout:

Accept user choice from menu.

Perform corresponding action.

**Function for Admin Staff Login:**

Prompt user to enter admin ID and password.

Search for matching admin ID and password in admin data file.

If match found:

Provide access to admin staff menu options.

Else:

Display error message and prompt for reentry.

**Admin Staff Menu Options:**

Display menu options (Update Customer Details, Logout).

Repeat until user chooses to logout:

Accept user choice from menu.

Perform corresponding action.

**Function for Super User Login:**

Prompt user to enter super user ID and password.

If super user ID and password match predefined values:

Provide access to super user menu options.

Else:

Display error message and prompt for reentry.

**Super User Menu Options:**

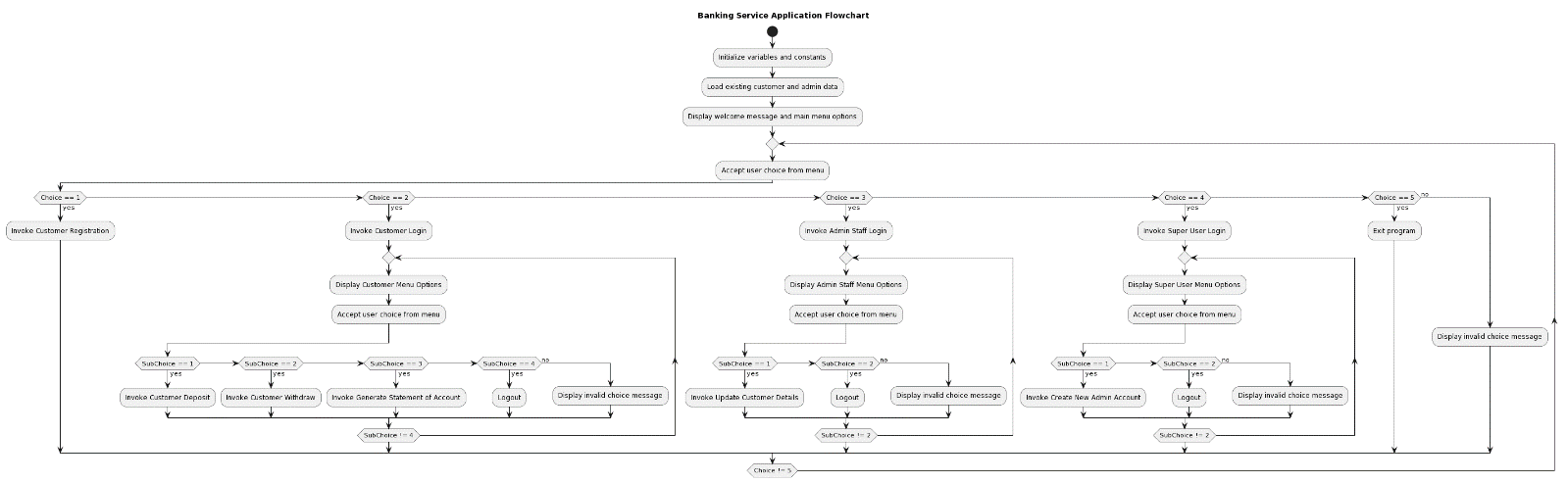
Display menu options (Create New Admin Account, Logout).

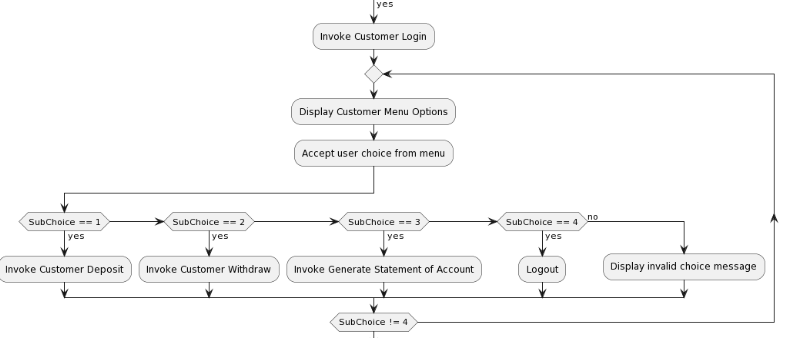
Repeat until user chooses to logout:

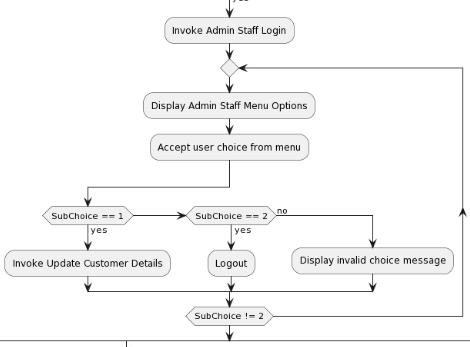
Accept user choice from menu.

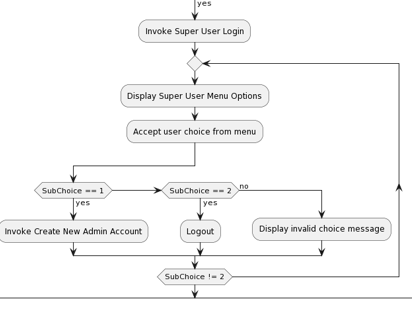
Perform corresponding action.

## **Flowcharts**









# **3. Programming Concepts with Source Code Explanation**

def validate\_account\_number(account\_number):

    if not account\_number.isdigit():

        return False

    if len(account\_number) != 10:

        return False

    return True

This function, `validate\_account\_number(account\_number)`, is responsible for validating a given account number. Let's break down the code and its purpose:

**1. `def validate\_account\_number(account\_number):**

This line defines the function `validate\_account\_number` with one parameter, `account\_number`, which represents the account number to be validated.

**2. `if not account\_number.isdigit():**

This conditional statement checks if the `account\_number` consists entirely of digits. It uses the `isdigit()` method, which returns `True` if all characters in the string are digits, and `False` otherwise.

If the `account\_number` contains any nondigit characters (e.g., letters, symbols), this condition evaluates to `True`, indicating that the account number is invalid. In this case, the function immediately returns `False`.

**3. `if len(account\_number) != 10:**

This conditional statement checks if the length of the `account\_number` is exactly 10 digits.

If the length of the `account\_number` is not equal to 10, it indicates an invalid account number, and the function returns `False`.

**4. `return True**

If the `account\_number` passes both validation checks (contains only digits and has a length of 10), the function returns `True`, indicating that the account number is valid.

**5. `return False**

If any of the validation conditions fail, the function returns `False`, indicating that the account number is invalid.

Overall, this function ensures that the provided account number is composed of exactly 10 digits and contains no non digit characters. It returns `True` if the account number is valid and `False` otherwise. This validation is crucial for maintaining data integrity and preventing errors when processing account related operations in the banking application.

def validate\_amount(amount):

    try:

        amount = float(amount)

        if amount <= 0:

            return False

        return True

    except ValueError:

        return False

This function, `validate\_amount(amount)`, is designed to validate the format and value of an amount, typically used in financial transactions. Let's dissect the code and its purpose:

**1. `def validate\_amount(amount):**

This line defines the function `validate\_amount` with one parameter, `amount`, representing the amount to be validated.

**2. `try:**

This keyword begins a try block, indicating that the code inside will be tested for exceptions.

**3. `amount = float(amount)**

This line attempts to convert the `amount` to a floating point number using the `float()` function. If successful, it means the input can be interpreted as a numerical value.

If the input `amount` cannot be converted to a float (e.g., if it contains nonnumeric characters), a `ValueError` will be raised, and the execution will jump to the `except` block.

**4. `if amount <= 0:**

This conditional statement checks if the `amount` is less than or equal to zero.

If the amount is zero or negative, it typically indicates an invalid or non-permissible transaction value. In such cases, the function returns `False`.

**5. `return True**

If the `amount` passes both validation checks (successfully converted to a float and greater than zero), the function returns `True`, indicating that the amount is valid.

**6. `except ValueError:**

This keyword introduces an `except` block that handles exceptions of type `ValueError`, which occurs when the attempt to convert `amount` to a float fails.

If a `ValueError` is raised during the execution of the code inside the `try` block, the control flow jumps to this `except` block.

**7. `return False**

Inside the `except` block, the function returns `False`, indicating that the `amount` is invalid due to its format (e.g., nonnumeric characters).

Overall, this function ensures that the provided `amount` is a valid numerical value greater than zero. It returns `True` if the amount is valid and `False` otherwise. Validating amounts is essential in financial applications to prevent errors and unauthorized transactions.

def validate\_date(date\_string):

    try:

        datetime.datetime.strptime(date\_string, '%Y%m%d')

        return True

    except ValueError:

        return False

This function, `validate\_date(date\_string)`, is responsible for validating whether a given date string conforms to a specific date format. Let's break down how it works and the programming concepts it employs:

**1. Function Definition:**

`def validate\_date(date\_string):` declares the function `validate\_date` with one parameter, `date\_string`, which represents the date string to be validated.

**2. TryExcept Block:**

`try:` introduces a block of code where an exception might occur. In this case, it's trying to parse the `date\_string` into a `datetime` object.

**3. Datetime Parsing:**

`datetime.datetime.strptime(date\_string, '%Y%m%d')` attempts to parse the `date\_string` using the `strptime` method from the `datetime` module.

`%Y%m%d` specifies the expected format of the date string, where `%Y` represents the year with century as a decimal number, `%m` represents the month as a zero padded decimal number, and `%d` represents the day of the month as a zero padded decimal number.

**4. Return True:**

If the parsing is successful, meaning the `date\_string` matches the specified format, the function immediately returns `True`. This indicates that the date string is valid.

**5. Except Block:**

`except ValueError:` catches any `ValueError` that occurs during the execution of the `try` block. In this case, it handles errors that arise when the `date\_string` cannot be parsed according to the specified format.

**6. Return False:**

If an exception is caught (i.e., the `date\_string` does not match the expected format), the function returns `False`. This signifies that the date string is invalid.

Overall, this function is useful for ensuring that date inputs provided by users or obtained from external sources adhere to a specific format, allowing for consistent handling of date data within the application.

def generate\_account\_number():

    if os.path.exists("account\_numbers.txt"):

        with open("account\_numbers.txt", "r") as file:

            lines = file.readlines()

            if lines:

                last\_account\_number = int(lines[1])

                new\_account\_number = last\_account\_number + 1

            else:

                new\_account\_number = 1000000001

    else:

        new\_account\_number = 1000000001

    with open("account\_numbers.txt", "a") as file:

        file.write(str(new\_account\_number) + "\n")

    return str(new\_account\_number)

This `generate\_account\_number()` function is responsible for generating a unique account number for new customer accounts. Let's delve into how it works and the programming concepts it employs:

**1. File Existence Check:**

`if os.path.exists("account\_numbers.txt"):`

This line checks if a file named "account\_numbers.txt" exists in the current directory using the `os.path.exists()` function.

**2. Read Existing Account Numbers:**

`with open("account\_numbers.txt", "r") as file:` opens the "account\_numbers.txt" file in read mode.

`lines = file.readlines()` reads all lines from the file and stores them in the `lines` list.

`if lines:` checks if there are any lines in the file. If the file is not empty, it proceeds to extract the last account number.

**3. Generate New Account Number:**

If the file exists and is not empty, it extracts the last account number and increments it to generate a new unique account number.

`last\_account\_number = int(lines[1]) Extracts the last account number from the list of lines and converts it to an integer.

`new\_account\_number = last\_account\_number + 1 Increments the last account number to generate a new one.

If the file doesn't exist or is empty, it initializes the new account number to a default value of 1000000001.

**4. Write New Account Number to File:**

`with open("account\_numbers.txt", "a") as file:` opens the file again in append mode to add the new account number.

`file.write(str(new\_account\_number) + "\n")` writes the new account number to the file, followed by a newline character to separate it from other entries.

**5. Return Account Number:**

`return str(new\_account\_number) Returns the newly generated account number as a string.

This function ensures that each new account number is unique by reading the existing account numbers from a file, incrementing the last account number, and appending the new number to the file for future reference. It utilizes file operations (`open`, `readlines`, `write`) and basic arithmetic to achieve its functionality.

def create\_customer\_account():

    name = input("Enter customer name: ")

    account\_type = input("Enter account type (Savings/Current): ").lower()

    while account\_type not in ['savings', 'current']:

        account\_type = input("Invalid account type. Enter 'Savings' or 'Current': ").lower()

    balance = input("Enter initial balance: ")

    while not validate\_amount(balance):

        balance = input("Invalid amount. Enter a valid amount: ")

    password = input("Enter password: ")  # Added password entry

    # Generate unique account number

    account\_numbers = [int(line.split(",")[0]) for line in open("customer\_accounts.txt").readlines()]

    if account\_numbers:

        account\_number = max(account\_numbers) + 1

    else:

        account\_number = 1

    with open("customer\_accounts.txt", "a") as file:

        file.write(f"{account\_number},{name},{account\_type},{balance},{password}\n")

    print(f"Customer registered successfully with account number: {account\_number}")

This `create\_customer\_account()` function is responsible for registering a new customer account. Let's break down how it works and the programming concepts it employs:

**1. Input Gathering:**

`name = input("Enter customer name: ") Prompts the user to enter the customer's name and stores it in the `name` variable.

`account\_type = input("Enter account type (Savings/Current): ").lower() Prompts the user to enter the account type (caseinsensitive) and converts it to lowercase for uniformity.

`balance = input("Enter initial balance: ") Prompts the user to enter the initial balance for the account.

`password = input("Enter password: ") Prompts the user to enter a password for the account.

**2. Input Validation:**

`while account\_type not in ['savings', 'current']: Validates that the entered account type is either 'savings' or 'current'. If not, it prompts the user again until a valid type is provided.

`while not validate\_amount(balance): Validates the entered balance using the `validate\_amount()` function. If the balance is not valid (e.g., negative or nonnumeric), it prompts the user to enter a valid amount.

**3. Generate Unique Account Number:**

Reads existing account numbers from the "customer\_accounts.txt" file and extracts the maximum account number.

`account\_numbers = [int(line.split(",")[0]) for line in open("customer\_accounts.txt").readlines()] Reads all lines from the file, splits each line by comma, and extracts the account number (first element) as an integer.

`if account\_numbers:` checks if there are existing account numbers. If there are, it calculates the new account number by incrementing the maximum account number by 1. If not, it sets the new account number to 1.

**4. Write Account Details to File:**

Opens the "customer\_accounts.txt" file in append mode and writes the new account details in the format: `account\_number, name, account\_type, balance, password`.

`with open("customer\_accounts.txt", "a") as file: Opens the file in append mode to add the new account details.

`file.write(f"{account\_number},{name},{account\_type},{balance},{password}\n") Writes the account details to the file, separated by commas and followed by a newline character.

**5. Output Confirmation:**

Prints a confirmation message indicating that the customer has been successfully registered, along with their unique account number.

This function employs input gathering, input validation, file handling (reading and writing), and basic arithmetic to register new customer accounts and ensure the uniqueness of account numbers.

def customer\_login():

    account\_number = input("Enter account number: ")

    password = input("Enter password: ")

    with open("customer\_accounts.txt", "r") as file:

        for line in file:

            acc\_num, name, acc\_type, balance, pwd = line.strip().split(",")

            if acc\_num == account\_number and pwd == password:

                return acc\_num, name, acc\_type, float(balance)

    return None

This `customer\_login()` function facilitates the login process for customers. Here's how it works and the programming concepts it utilizes:

**1. Input Gathering:**

`account\_number = input("Enter account number: ") Prompts the user to input their account number.

`password = input("Enter password: ") Prompts the user to input their password associated with the account.

**2. File Reading and Validation:**

`with open("customer\_accounts.txt", "r") as file: Opens the "customer\_accounts.txt" file in read mode to access customer account information.

`for line in file: Iterates through each line in the file.

`acc\_num, name, acc\_type, balance, pwd = line.strip().split(",") Extracts account details from each line by splitting the line using commas.

**3. Login Validation:**

Checks if the entered account number and password match any of the accounts stored in the file.

`if acc\_num == account\_number and pwd == password: Compares the entered account number and password with the account number and password stored in the current line of the file.

If a match is found, it returns a tuple containing the account number, name, account type, and balance of the customer.

If no match is found after iterating through all accounts, it returns `None` to indicate a failed login attempt.

**4. Data Conversion:**

`return acc\_num, name, acc\_type, float(balance) Converts the balance from string to float before returning it as part of the tuple.

This function demonstrates file handling (reading), input validation, string manipulation (splitting), conditional statements, and data conversion. It provides a mechanism for customers to authenticate themselves using their account number and password.

def customer\_deposit(account\_number):

    amount = input("Enter deposit amount: ")

    while not validate\_amount(amount):

        amount = input("Invalid amount. Enter a valid amount: ")

    with open("customer\_accounts.txt", "r") as file:

        lines = file.readlines()

    with open("customer\_accounts.txt", "w") as file:

        for line in lines:

            acc\_num, name, acc\_type, balance, pwd = line.strip().split(",")

            if acc\_num == account\_number:

                balance = str(float(balance) + float(amount))

            file.write(f"{acc\_num},{name},{acc\_type},{balance},{pwd}\n")

    # Record transaction in transactions.txt

    with open("transactions.txt", "a") as file:

        file.write(f"{account\_number},{datetime.datetime.now().strftime('%Y%m%d')},Deposit,{amount}\n")

    print("Deposit successful.")

This `customer\_deposit(account\_number)` function is responsible for handling customer deposits. Let's break down how it works and the programming concepts it employs:

**1. Input Gathering and Validation:**

`amount = input("Enter deposit amount: ") Prompts the user to input the deposit amount.

`while not validate\_amount(amount): Validates the entered amount using the `validate\_amount()` function. If the amount is invalid (nonnumeric or nonpositive), it prompts the user again until a valid amount is provided.

**2. File Reading and Account Update:**

`with open("customer\_accounts.txt", "r") as file: Opens the "customer\_accounts.txt" file in read mode to access customer account information.

`lines = file.readlines() Reads all lines from the file and stores them in a list.

`with open("customer\_accounts.txt", "w") as file: Reopens the file in write mode to update account balances.

`for line in lines: Iterates through each line in the file.

`acc\_num, name, acc\_type, balance, pwd = line.strip().split(",") Extracts account details from each line by splitting the line using commas.

Updates the balance of the account with the provided account number by adding the deposited amount.

Writes the updated account information (including the new balance) back to the file.

**3. Transaction Logging:**

Records the deposit transaction in the "transactions.txt" file by appending a new line containing the account number, current date (using `datetime.datetime.now()`), transaction type ("Deposit"), and deposited amount.

**4. User Feedback:**

Prints a message confirming the successful deposit.

This function demonstrates file handling (reading and writing), input validation, string manipulation (splitting), conditional statements, and data conversion. It provides a way for customers to deposit funds into their accounts while ensuring that the account balance is updated correctly and the transaction is logged for recordkeeping purposes.

def customer\_withdraw(account\_number):

    amount = input("Enter withdrawal amount: ")

    while not validate\_amount(amount):

        amount = input("Invalid amount. Enter a valid amount: ")

    with open("customer\_accounts.txt", "r") as file:

        lines = file.readlines()

    with open("customer\_accounts.txt", "w") as file:

        for line in lines:

            acc\_num, name, acc\_type, balance, pwd = line.strip().split(",")

            if acc\_num == account\_number:

                if float(balance) float(amount) < (100 if acc\_type == "savings" else 500):

                    print("Insufficient balance.")

                    return

                balance = str(float(balance) float(amount))

            file.write(f"{acc\_num},{name},{acc\_type},{balance},{pwd}\n")

    # Record transaction in transactions.txt

    with open("transactions.txt", "a") as file:

        file.write(f"{account\_number},{datetime.datetime.now().strftime('%Y%m%d')},Withdrawal,{amount}\n")

    print("Withdrawal successful.")

This `customer\_withdraw(account\_number)` function allows customers to withdraw funds from their accounts. Let's break down its functionality and the programming concepts it utilizes:

**1. Input Gathering and Validation:**

`amount = input("Enter withdrawal amount: ") Prompts the user to input the withdrawal amount.

`while not validate\_amount(amount): Validates the entered amount using the `validate\_amount()` function. If the amount is invalid (nonnumeric or nonpositive), it prompts the user again until a valid amount is provided.

**2. File Reading and Account Update:**

`with open("customer\_accounts.txt", "r") as file: Opens the "customer\_accounts.txt" file in read mode to access customer account information.

`lines = file.readlines() Reads all lines from the file and stores them in a list.

`with open("customer\_accounts.txt", "w") as file: Reopens the file in write mode to update account balances.

`for line in lines: Iterates through each line in the file.

`acc\_num, name, acc\_type, balance, pwd = line.strip().split(",") Extracts account details from each line by splitting the line using commas.

Checks if the current line corresponds to the account number provided for withdrawal.

Verifies if the account has sufficient balance to cover the withdrawal, considering the account type (savings or current).

Updates the balance of the account by subtracting the withdrawn amount.

Writes the updated account information (including the new balance) back to the file.

**3. Transaction Logging:**

Records the withdrawal transaction in the "transactions.txt" file by appending a new line containing the account number, current date (using `datetime.datetime.now()`), transaction type ("Withdrawal"), and withdrawn amount.

**4. User Feedback:**

Prints a message confirming the successful withdrawal.

This function demonstrates file handling (reading and writing), input validation, string manipulation (splitting), conditional statements, arithmetic operations, and data conversion. It ensures that customers can withdraw funds from their accounts only if they have sufficient balance, updating the account balance accordingly and logging the transaction for recordkeeping purposes.

def generate\_statement\_of\_account(account\_number):

    start\_date = input("Enter start date (YYYYMMDD): ")

    while not validate\_date(start\_date):

        start\_date = input("Invalid date format. Enter start date (YYYYMMDD): ")

    end\_date = input("Enter end date (YYYYMMDD): ")

    while not validate\_date(end\_date):

        end\_date = input("Invalid date format. Enter end date (YYYYMMDD): ")

    total\_deposits = 0

    total\_withdrawals = 0

    with open("transactions.txt", "r") as file:

        for line in file:

            acc\_num, transaction\_date, transaction\_type, amount = line.strip().split(",")

            if acc\_num == account\_number and start\_date <= transaction\_date <= end\_date:

                if transaction\_type == "Deposit":

                    total\_deposits += float(amount)

                else:

                    total\_withdrawals += float(amount)

    print(f"Total Deposits: RM {total\_deposits}")

    print(f"Total Withdrawals: RM {total\_withdrawals}")

The `generate\_statement\_of\_account(account\_number)` function is responsible for generating a statement of account for a specified customer within a given date range. Let's examine its components and the programming concepts it employs:

**1. Input Gathering and Validation:**

Prompts the user to enter the start and end dates for the statement in the format "YYYYMMDD".

Utilizes the `validate\_date()` function to ensure that the entered dates are in the correct format. If the format is invalid, it prompts the user to reenter the date until a valid format is provided.

**2. Processing Transactions:**

Initializes variables `total\_deposits` and `total\_withdrawals` to keep track of the cumulative deposit and withdrawal amounts within the specified date range.

Opens the "transactions.txt" file in read mode to access transaction records.

Iterates through each line in the file, extracting account number, transaction date, transaction type, and amount for each transaction.

Checks if the transaction belongs to the specified account number and falls within the provided date range.

If the conditions are met, updates the corresponding total deposit or withdrawal amount based on the transaction type.

**3. Output Generation:**

Prints the total deposit and withdrawal amounts within the specified date range in the desired format.

Displays the statement of account to the user.

This function demonstrates input handling, validation, file reading, string manipulation (splitting), conditional statements, arithmetic operations, and output formatting. It allows customers to review their financial transactions over a specific period, aiding in financial management and recordkeeping.

def admin\_update\_customer\_details():

    account\_number = input("Enter customer account number: ")

    with open("customer\_accounts.txt", "r") as file:

        lines = file.readlines()

    found = False

    with open("customer\_accounts.txt", "w") as file:

        for line in lines:

            acc\_num, name, acc\_type, balance, pwd = line.strip().split(",")

            if acc\_num == account\_number:

                print("Customer found. Enter new details.")

                new\_name = input(f"Enter new name for customer {name}: ")

                new\_account\_type = input(f"Enter new account type for customer {name} (Savings/Current): ").lower()

                while new\_account\_type not in ['savings', 'current']:

                    new\_account\_type = input("Invalid account type. Enter 'Savings' or 'Current': ").lower()

                new\_balance = input(f"Enter new balance for customer {name}: ")

                while not validate\_amount(new\_balance):

                    new\_balance = input("Invalid amount. Enter a valid amount: ")

                file.write(f"{account\_number},{new\_name},{new\_account\_type},{new\_balance},{pwd}\n")

                found = True

            else:

                file.write(line)

    if not found:

        print("Customer not found.")

The `admin\_update\_customer\_details()` function allows an admin staff member to update the details of a customer's account. Here's a breakdown of how it works and the programming concepts it employs:

**1. Input Gathering:**

Prompts the admin staff to enter the account number of the customer whose details need to be updated.

**2. Reading Existing Data:**

Opens the "customer\_accounts.txt" file in read mode and reads all existing lines into the `lines` list.

**3. Processing and Updating Data:**

Initializes a boolean variable `found` to track whether the specified customer account number exists.

Opens the "customer\_accounts.txt" file in write mode to update the customer details.

Iterates through each line in the `lines` list, splitting it into individual elements (account number, name, account type, balance, password).

If the account number matches the one entered by the admin staff, prompts the admin staff to enter new details for the customer:

New name for the customer.

New account type (Savings/Current).

New balance.

Validates the new balance using the `validate\_amount()` function to ensure it is a valid amount.

Writes the updated customer details (including the new name, account type, and balance) back to the file.

Sets the `found` flag to `True` to indicate that the customer account was found and updated.

If the account number was not found in the file, prints a message indicating that the customer was not found.

**4. Output:**

If the customer was found and updated, no explicit output is provided.

If the customer was not found, a message indicating that the customer was not found is printed.

This function demonstrates file handling (reading and writing), string manipulation (splitting), input validation, conditional statements, and user interaction. It provides admin staff with the capability to modify customer account details stored in the system.

def create\_admin\_account():

    admin\_id = input("Enter admin ID: ")

    admin\_password = input("Enter admin password: ")

    with open("admin\_accounts.txt", "a") as file:

        file.write(f"{admin\_id},{admin\_password}\n")

    print("Admin account created successfully.")

def admin\_login():

    admin\_id = input("Enter admin ID: ")

    admin\_password = input("Enter admin password: ")

    with open("admin\_accounts.txt", "r") as file:

        for line in file:

            stored\_admin\_id, stored\_admin\_password = line.strip().split(",")

            if admin\_id == stored\_admin\_id and admin\_password == stored\_admin\_password:

                return True

    return False

The `create\_admin\_account()` function is responsible for creating a new admin account, while the `admin\_login()` function facilitates the login process for admin staff. Here's an explanation of each function along with the programming concepts they use:

**1. `create\_admin\_account()**

Input Gathering:

Prompts the user to enter an admin ID and password.

File Writing:

Opens the "admin\_accounts.txt" file in append mode and writes the admin ID and password to the file in the format: `admin\_id,password\n`.

Output:

Prints a success message indicating that the admin account has been created.

**2. `admin\_login()**

Input Gathering:

Prompts the user to enter an admin ID and password.

File Reading and Comparison:

Opens the "admin\_accounts.txt" file in read mode and iterates through each line.

Strips newline characters and splits each line into admin ID and password using the comma (`,`) as a delimiter.

Compares the entered admin ID and password with the stored admin ID and password from the file.

Return Value:

Returns `True` if the entered admin ID and password match any admin account stored in the file.

Returns `False` if there is no match found.

These functions demonstrate input/output operations, file handling (reading and writing), string manipulation (splitting), conditional statements, and user authentication. The `create\_admin\_account()` function adds a new admin account to the system, while the `admin\_login()` function verifies the credentials of admin staff during the login process.

def main():

    super\_user\_id = "admin"

    super\_user\_password = "password"

    while True:

        print("\n===== Banking Service Application =====")

        print("1. Customer Registration")

        print("2. Customer Login")

        print("3. Admin Staff Login")

        print("4. Super User Login")

        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":

            create\_customer\_account()

        elif choice == "2":

            account\_info = customer\_login()

            if account\_info:

                account\_number, name, account\_type, balance = account\_info

                print(f"Welcome, {name} ({account\_number})")

                while True:

                    print("\n===== Customer Menu =====")

                    print("1. Deposit")

                    print("2. Withdraw")

                    print("3. Generate Statement of Account")

                    print("4. Logout")

                    sub\_choice = input("Enter your choice: ")

                    if sub\_choice == "1":

                        customer\_deposit(account\_number)

                    elif sub\_choice == "2":

                        customer\_withdraw(account\_number)

                    elif sub\_choice == "3":

                        generate\_statement\_of\_account(account\_number)

                    elif sub\_choice == "4":

                        break

                    else:

                        print("Invalid choice. Please try again.")

        elif choice == "3":

            if admin\_login():

                print("Admin Staff Login successful.")

                while True:

                    print("\n===== Admin Staff Menu =====")

                    print("1. Update Customer Details")

                    print("2. Logout")

                    sub\_choice = input("Enter your choice: ")

                    if sub\_choice == "1":

                        admin\_update\_customer\_details()

                    elif sub\_choice == "2":

                        break

                    else:

                        print("Invalid choice. Please try again.")

            else:

                print("Invalid admin ID or password.")

        elif choice == "4":

            super\_user\_id\_input = input("Enter Super User ID: ")

            super\_user\_password\_input = input("Enter Super User Password: ")

            if super\_user\_id\_input == super\_user\_id and super\_user\_password\_input == super\_user\_password:

                print("Super User Login successful.")

                while True:

                    print("\n===== Super User Menu =====")

                    print("1. Create New Admin Account")

                    print("2. Logout")

                    sub\_choice = input("Enter your choice: ")

                    if sub\_choice == "1":

                        create\_admin\_account()

                    elif sub\_choice == "2":

                        break

                    else:

                        print("Invalid choice. Please try again.")

            else:

                print("Invalid Super User ID or password.")

        elif choice == "5":

            print("Exiting program. Goodbye!")

            break

        else:

            print("Invalid choice. Please try again.")

The `main()` function serves as the entry point and central control mechanism for the banking service application. Here's a detailed explanation of its functionality and the programming concepts it employs:

**1. Initialization:**

Initializes the `super\_user\_id` and `super\_user\_password` variables with the default super user credentials.

**2. Main Loop:**

Utilizes a `while True` loop to continuously display the main menu and handle user input until the program is exited.

**3. Menu Display:**

Prints the main menu options for the user to choose from:

Customer Registration

Customer Login

Admin Staff Login

Super User Login

Exit

**4. User Input and Control Flow:**

Accepts user input for menu choice (`choice`) using the `input()` function.

Uses conditional statements (`if`, `elif`, `else`) to determine the action based on the user's choice.

**5. Function Calls:**

Calls various functions based on the user's choice to perform specific tasks or operations:

Customer Registration (`create\_customer\_account()`)

Customer Login (`customer\_login()`)

Admin Staff Login (`admin\_login()`)

Super User Login

Create New Admin Account (`create\_admin\_account()`)

Logout

Exit

**6. Nested Menus:**

Contains nested loops for handling customer and admin staff interactions, allowing users to perform multiple actions within their respective menus.

**7. Error Handling:**

Provides error messages for invalid menu choices and incorrect login credentials.

**8. Loop Termination:**

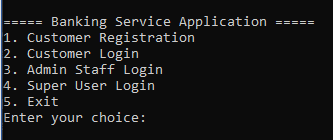
Breaks out of the main loop and exits the program when the user chooses to exit.

Overall, the `main()` function demonstrates the use of loops, conditional statements, user input handling, function calls, and modular design to create an interactive banking service application. It serves as the central control hub for directing program flow based on user input and facilitating various banking operations.

# **4. Screenshots of Sample Input/Output and Explanation**

Let us walk through a flow of tasks in our application.

When we start the application, we are presented with a menu like this:

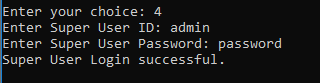


**1. Super User Login:**

The user selects option 4 from the main menu to log in as a super user.

They input the super user ID as "admin" and the password as "password".

Upon successful login, the system displays the Super User Menu with options to create a new admin account or logout.

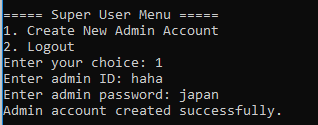


**2. Creating a New Admin Account:**

In the Super User Menu, the user selects option 1 to create a new admin account.

They input the admin ID as "haha" and the password as "japan".

The system confirms the successful creation of the admin account and returns to the Super User Menu.

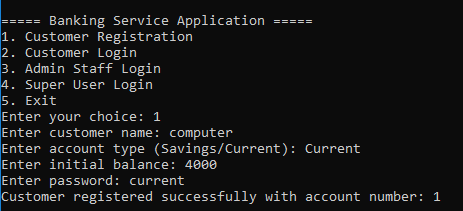


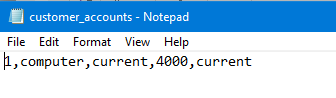
**3. Customer Registration:**

The user selects option 1 from the main menu to register a new customer account.

They input the customer's name, account type, initial balance, and password.

The system generates a unique account number and confirms the successful registration of the customer account.



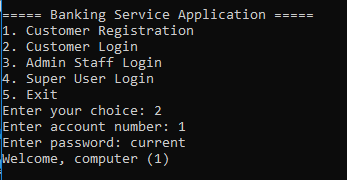


**4. Customer Login:**

The user selects option 2 from the main menu to log in as a customer.

They input the account number and password.

Upon successful login, the system welcomes the customer and displays the Customer Menu with options for deposit, withdrawal, statement generation, and logout.

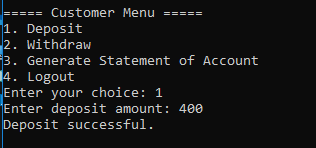


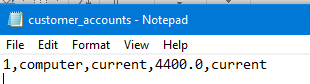
**5. Customer Deposit:**

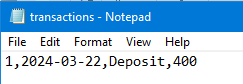
In the Customer Menu, the user selects option 1 to make a deposit.

They input the deposit amount.

The system processes the deposit and confirms its success.





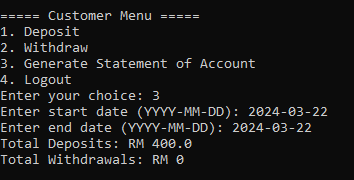


**6. Generate Statement of Account:**

The user selects option 3 from the Customer Menu to generate a statement of account.

They input the start and end dates for the statement.

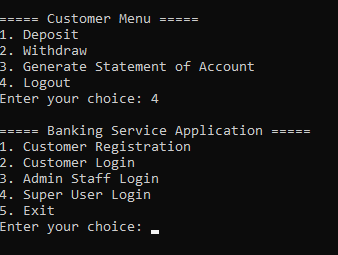
The system calculates and displays the total deposits and withdrawals made on that date.



**7. Logging Out:**

At any point in the menu, the user can choose to logout.

Upon logout, the system returns to the main menu, allowing the user to perform other actions or exit the program.



# **5. Conclusion**

The Banking Service Application provides a comprehensive system for managing customer accounts and administrative tasks. It demonstrates effective utilization of programming concepts to create a functional and userfriendly application. Future enhancements could include additional security features and improved error handling.

# **6. References**

1. *Python Functions: How to Call & Write Functions*. (2020, January 2). DataCamp Community. <https://www.datacamp.com/community/tutorials/functions-python-tutorial>
2. *Workflow%20Tool - Visymo Search*. (n.d.). Search.visymo.com. Retrieved March 19, 2022, from <https://search.visymo.com/ws?q=workflow%20tool&asid=vis_my_gc4_4&mt=b&nw=g&de=c&ap=&ac=13361&cid=11675358680&aid=118878927332&locale=en_MY&gclid=CjwKCAjw_tWRBhAwEiwALxFPoY7VtXo9BTngzI_xfCL9QEKYON8A4CPU7B9d-PTEwoMKDOQF_FrChBoCqSoQAvD_BwE>
3. *Python For Loops*. (2019). W3schools.com. <https://www.w3schools.com/python/python_for_loops.asp>
4. *Guide to Do, While and For loops in Pseudocode - PseudoEditor*. (n.d.). Pseudoeditor.com. Retrieved March 19, 2022, from <https://pseudoeditor.com/guides/loops-and-iteration#:~:text=Do%20loops%20in%20Pseudocode>
5. *OpenStax CNX*. (n.d.). Cnx.org. Retrieved March 19, 2022, from <https://cnx.org/contents/MDgA8wfz@22.2:6Uvbhb3A@7/Pseudocode-Examples-for-Functions>
6. Python, R. (n.d.). *The Python return Statement: Usage and Best Practices – Real Python*. Realpython.com. <https://realpython.com/python-return-statement/>
7. *Loops: while and for*. (n.d.). Javascript.info. <https://javascript.info/while-for>
8. *Pseudocode Examples*. (2019). Unf.edu. <https://www.unf.edu/~broggio/cop3530/3530pseu.htm>
9. *Python while Loop*. (n.d.). Www.programiz.com. Retrieved March 19, 2022, from <https://www.programiz.com/python-programming/while-loop#:~:text=The%20while%20loop%20in%20Python>
10. *Python - Call function from another function*. (2019, December 4). GeeksforGeeks. <https://www.geeksforgeeks.org/python-call-function-from-another-function/#:~:text=The%20Function%20which%20calls%20another>
11. *Python File I/O: Read and Write Files in Python*. (n.d.). Www.programiz.com. <https://www.programiz.com/python-programming/file-operation>
12. *Python File Open*. (n.d.). Www.w3schools.com. <https://www.w3schools.com/python/python_file_handling.asp>
13. *How does carriage return work in Python*. (2018, December 13). CodeSpeedy. <https://www.codespeedy.com/how-does-carriage-return-work-in-python/#:~:text=A%20carriage%20return%20is%20nothing>
14. *c - How to draw flowchart for code involving opening from text file and reading them*. (n.d.). Stack Overflow. Retrieved March 19, 2022, from <https://stackoverflow.com/questions/25489837/how-to-draw-flowchart-for-code-involving-opening-from-text-file-and-reading-them>
15. *Python File Open*. (2019). W3schools.com. <https://www.w3schools.com/python/python_file_open.asp>
16. *Guide to Text Files in Pseudocode - PseudoEditor*. (n.d.). Pseudoeditor.com. Retrieved March 19, 2022, from <https://pseudoeditor.com/guides/reading-and-writing-files>
17. *JavaScript for Loop*. (2019). W3schools.com. <https://www.w3schools.com/js/js_loop_for.asp>
18. *Python File Write*. (2019). W3schools.com. <https://www.w3schools.com/python/python_file_write.asp>
19. *Python Classes and Objects*. (2019, October 15). GeeksforGeeks. <https://www.geeksforgeeks.org/python-classes-and-objects/#:~:text=Class%20creates%20a%20user%2Ddefined>