BANKING SYSTEM

Python Bank Account Management System

```
import pickle
import numpy as np
from datetime import datetime
import os
import getpass

class BankAccount:
    def __init__(self, account_holder_name, account_type, initial_balance=0, password=None):
        self.account_holder_name = account_holder_name
        self.account_number = self._generate_account_number()
        self.account_type = account_type
        self.balance = initial_balance
        self.transaction_history = []
        self.password = password  # For bonus login functionality

        # Record initial deposit if any
        if initial_balance > 0:
            self._record_transaction("deposit", initial_balance)

    def _generate_account_number(self):
        """Generate a unique account number"""
        timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
        return f"ACC{timestamp}"

    def _record_transaction(self, transaction_type, amount, other_account=None):
        """Record a transaction in the history"""
        transaction = {
            "date": datetime.now(),
            "type": transaction_type,
            "amount": amount,
            "balance_after": self.balance
        }

        if other_account:
            transaction["other_account"] = other_account

        self.transaction_history.append(transaction)

    def deposit(self, amount):
        """Deposit money into the account"""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")

        self.balance += amount
        self._record_transaction("deposit", amount)
        return True

    def withdraw(self, amount):
        """Withdraw money from the account"""
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")

        if amount > self.balance:
            raise ValueError("Insufficient funds")

        self.balance -= amount
        self._record_transaction("withdrawal", amount)
        return True

    def transfer(self, amount, target_account):
        """Transfer money to another account"""
        if amount <= 0:
            raise ValueError("Transfer amount must be positive")
```

```
            if amount > self.balance:
                raise ValueError("Insufficient funds for transfer")

            # Withdraw from this account
            self.balance -= amount
            self._record_transaction("transfer_out", amount, target_account.account_number)

            # Deposit to target account
            target_account.balance += amount
            target_account._record_transaction("transfer_in", amount, self.account_number)

            return True

        def get_account_details(self):
            """Return account details"""
            return {
                "account_holder_name": self.account_holder_name,
                "account_number": self.account_number,
                "account_type": self.account_type,
                "balance": self.balance
            }

        def get_transaction_history(self):
            """Return transaction history"""
            return self.transaction_history

        def get_summary_statistics(self):
            """Generate summary statistics using NumPy"""
            if not self.transaction_history:
                return {
                    "total_deposits": 0,
                    "total_withdrawals": 0,
                    "average_deposit": 0,
                    "average_withdrawal": 0,
                    "total_transactions": 0
                }

            deposits = [t["amount"] for t in self.transaction_history if t["type"] in ["deposit", "transfer_
            withdrawals = [t["amount"] for t in self.transaction_history if t["type"] in ["withdrawal", "tra

            return {
                "total_deposits": np.sum(deposits) if deposits else 0,
                "total_withdrawals": np.sum(withdrawals) if withdrawals else 0,
                "average_deposit": np.mean(deposits) if deposits else 0,
                "average_withdrawal": np.mean(withdrawals) if withdrawals else 0,
                "total_transactions": len(self.transaction_history)
            }


    class BankingSystem:
        def __init__(self, data_file="bank_data.pkl"):
            self.data_file = data_file
            self.accounts = self._load_data()

        def _load_data(self):
            """Load account data from file"""
            if os.path.exists(self.data_file):
                try:
                    with open(self.data_file, 'rb') as file:
                        return pickle.load(file)
                except (pickle.PickleError, EOFError):
                    return {}
            return {}

        def _save_data(self):
            """Save account data to file"""
            with open(self.data_file, 'wb') as file:
                pickle.dump(self.accounts, file)

        def create_account(self, account_holder_name, account_type, initial_balance=0, password=None):
            """Create a new bank account"""
            # Input validation
```

```
            if not account_holder_name or not account_type:
                raise ValueError("Account holder name and type are required")

            if initial_balance < 0:
                raise ValueError("Initial balance cannot be negative")

            # Create account
            account = BankAccount(account_holder_name, account_type, initial_balance, password)
            self.accounts[account.account_number] = account

            # Save to file
            self._save_data()

            return account.account_number

        def get_account(self, account_number, password=None):
            """Retrieve an account by number with optional password check"""
            if account_number not in self.accounts:
                raise ValueError("Account not found")

            account = self.accounts[account_number]

            # Password check for bonus feature
            if account.password and account.password != password:
                raise ValueError("Invalid password")

            return account

        def deposit(self, account_number, amount):
            """Deposit money into an account"""
            account = self.get_account(account_number)
            result = account.deposit(amount)
            self._save_data()
            return result

        def withdraw(self, account_number, amount):
            """Withdraw money from an account"""
            account = self.get_account(account_number)
            result = account.withdraw(amount)
            self._save_data()
            return result

        def transfer(self, from_account_number, to_account_number, amount):
            """Transfer money between accounts"""
            if from_account_number not in self.accounts:
                raise ValueError("Source account not found")

            if to_account_number not in self.accounts:
                raise ValueError("Target account not found")

            from_account = self.accounts[from_account_number]
            to_account = self.accounts[to_account_number]

            result = from_account.transfer(amount, to_account)
            self._save_data()
            return result

        def get_account_details(self, account_number):
            """Get account details"""
            account = self.get_account(account_number)
            return account.get_account_details()

        def get_transaction_history(self, account_number):
            """Get transaction history for an account"""
            account = self.get_account(account_number)
            return account.get_transaction_history()

        def get_summary_statistics(self, account_number):
            """Get summary statistics for an account"""
            account = self.get_account(account_number)
            return account.get_summary_statistics()
    def display_menu():
```

```
    """Display the main menu"""
    print("\n" + "="*50)
    print("          BANKING SYSTEM MENU")
    print("="*50)
    print("1. Open a new account")
    print("2. View account details")
    print("3. Deposit money")
    print("4. Withdraw money")
    print("5. Transfer money")
    print("6. View transaction history")
    print("7. View account summary statistics")
    print("8. Exit")
    print("="*50)


def main():
    banking_system = BankingSystem()

    print("Welcome to the Banking System!")

    while True:
        display_menu()
        choice = input("Enter your choice (1-8): ").strip()

        if choice == '1':
            # Open a new account
            try:
                name = input("Enter account holder's name: ").strip()
                account_type = input("Enter account type (savings/current): ").strip().lower()

                if account_type not in ['savings', 'current']:
                    print("Error: Account type must be 'savings' or 'current'")
                    continue

                initial_balance = float(input("Enter initial balance: "))

                # Bonus: Password protection
                set_password = input("Set a password for this account? (y/n): ").strip().lower()
                password = None
                if set_password == 'y':
                    password = getpass.getpass("Enter password: ")
                    confirm_password = getpass.getpass("Confirm password: ")
                    if password != confirm_password:
                        print("Error: Passwords do not match")
                        continue

                account_number = banking_system.create_account(
                    name, account_type, initial_balance, password
                )
                print(f"Account created successfully! Your account number is: {account_number}")

            except ValueError as e:
                print(f"Error: {e}")
            except Exception as e:
                print(f"An error occurred: {e}")

        elif choice in ['2', '3', '4', '5', '6', '7']:
            # Operations requiring an account number
            account_number = input("Enter your account number: ").strip()

            # Bonus: Password check if account has password
            try:
                account = banking_system.get_account(account_number)
                password = None
                if account.password:
                    password = getpass.getpass("Enter account password: ")
                    account = banking_system.get_account(account_number, password)
            except ValueError as e:
                print(f"Error: {e}")
                continue

            if choice == '2':
```

```
            # View account details
            try:
                details = banking_system.get_account_details(account_number)
                print("\nAccount Details:")
                print(f"Holder Name: {details['account_holder_name']}")
                print(f"Account Number: {details['account_number']}")
                print(f"Account Type: {details['account_type']}")
                print(f"Current Balance: ${details['balance']:.2f}")
            except ValueError as e:
                print(f"Error: {e}")

        elif choice == '3':
            # Deposit money
            try:
                amount = float(input("Enter deposit amount: "))
                banking_system.deposit(account_number, amount)
                print(f"Deposit successful! New balance: ${banking_system.get_account_details(accoun
            except ValueError as e:
                print(f"Error: {e}")

        elif choice == '4':
            # Withdraw money
            try:
                amount = float(input("Enter withdrawal amount: "))
                banking_system.withdraw(account_number, amount)
                print(f"Withdrawal successful! New balance: ${banking_system.get_account_details(acc
            except ValueError as e:
                print(f"Error: {e}")

        elif choice == '5':
            # Transfer money
            try:
                target_account = input("Enter target account number: ").strip()
                amount = float(input("Enter transfer amount: "))
                banking_system.transfer(account_number, target_account, amount)
                print(f"Transfer successful! New balance: ${banking_system.get_account_details(accou
            except ValueError as e:
                print(f"Error: {e}")

        elif choice == '6':
            # View transaction history
            try:
                history = banking_system.get_transaction_history(account_number)
                if not history:
                    print("No transactions found.")
                else:
                    print("\nTransaction History:")
                    print("-" * 80)
                    print(f"{'Date':<20} {'Type':<15} {'Amount':<15} {'Balance After':<15}")
                    print("-" * 80)
                    for transaction in history:
                        date_str = transaction['date'].strftime("%Y-%m-%d %H:%M:%S")
                        trans_type = transaction['type']
                        amount = f"${transaction['amount']:.2f}"
                        balance = f"${transaction['balance_after']:.2f}"
                        print(f"{date_str:<20} {trans_type:<15} {amount:<15} {balance:<15}")
            except ValueError as e:
                print(f"Error: {e}")

        elif choice == '7':
            # View account summary statistics
            try:
                stats = banking_system.get_summary_statistics(account_number)
                print("\nAccount Summary Statistics:")
                print(f"Total Deposits: ${stats['total_deposits']:.2f}")
                print(f"Total Withdrawals: ${stats['total_withdrawals']:.2f}")
                print(f"Average Deposit: ${stats['average_deposit']:.2f}")
                print(f"Average Withdrawal: ${stats['average_withdrawal']:.2f}")
                print(f"Total Transactions: {stats['total_transactions']}")
            except ValueError as e:
                print(f"Error: {e}")
    elif choice == '8':
```

```
            # Exit
            print("Thank you for using the Banking System. Goodbye!")
            break

        else:
            print("Invalid choice. Please enter a number between 1 and 8.")


# Bonus: Lambda functions for quick calculations
calculate_interest = lambda principal, rate, time: principal * rate * time / 100
calculate_future_value = lambda principal, rate, time: principal * (1 + rate/100) ** time

if __name__ == "__main__":
    main()
```