DIGITAL WALLET SIMULATOR - TECHNICAL REQUIREMENTS DOCUMENT

Executive Summary

This project simulates modern digital payment systems similar to UPI (Unified Payments Interface) platforms. Students will implement comprehensive account management, secure money transfers, transaction tracking, and expense categorization features to understand digital payments and practice Python programming concepts.

1. Executive Summary

This project simulates modern digital payment systems similar to UPI (Unified Payments Interface) platforms. Students will implement comprehensive account management, secure money transfers, transaction tracking, and expense categorization features. The purpose is to practice Python programming concepts using a mock digital wallet system that runs entirely in memory.

Note: All data is mock data. No persistence to files or databases will be implemented in this version.

2. Project Scope

2.1 In Scope

- User account creation and management
- Secure PIN-based authentication
- Money transfer functionality (UPI-style)
- QR code payment simulation (string-based)
- Transaction history tracking
- Balance management and tracking
- Expense categorization system
- Basic reporting and analytics

2.2 Out of Scope

- Real banking integration
- Actual monetary transactions
- External payment gateway integration
- File or database persistence (mock data only)
- Mobile or web application development (console-based only)

3. Technical Architecture Overview

3.1 Programming Language Requirements

- Primary Language: Python 3.8+
- Dependencies: None beyond Python standard library

3.2 System Architecture

- Data Storage: In-memory Python dictionary (mock only)
- All accounts and transactions are stored in a variable called `data`.
- Data is lost once the program ends.
- User Interface: Command-line interface (CLI)

4. Python Technical Concepts Implementation

4.1 Data Types Implementation

Data Type	Usage	Implementation Area
String	Usernames, PINs, transaction IDs	Accounts, authentication
Integer	PIN digits, menu selections	Input validation
Float	Account balances, amounts	Financial calculations
Boolean	Account status, session state	Validation and control flow
Dictionary	Accounts, transactions	Core data structure
List	Transaction history, categories	Iteration, grouping
Tuple	Constants, immutable settings	Configuration

4.2 Input Function Requirements

username = input("Enter your username: ")

pin_code = input("Enter 4-digit PIN: ")

transfer_amount = float(input("Enter transfer amount: "))

recipient = input("Enter recipient username: ")

category = input("Select expense category: ")

Validation:

- PIN must be exactly 4 digits
- Amount must be positive
- Username must exist in `data["accounts"]`

4.3 Data Structures

- Lists
- Store transactions per user
- Maintain categories
- Dictionaries
- Accounts keyed by username
- Each account stores balance, pin, and transactions
- Sets (optional)
- Track unique transaction IDs

4.4 Core Functions (to implement)

def create_account(data):

"""Create new user account"""

def login(data):

"""Authenticate user with username + PIN"""

def deposit(data, username):

"""Add money to account"""

def withdraw(data, username):

"""Withdraw money from account"""

def transfer(data, sender):

"""Send money to another account"""

def qr_generate(data, username):

"""Generate QR payload (string)"""

def qr_pay(data, payer):

"""Pay using QR payload"""

def show_transactions(data, username):

"""Display all transactions"""

def report_category_spend(data, username):

"""Report spending by category"""

def report_monthly_summary(data, username):

"""Monthly inflow/outflow report"""

def report_top_payees(data, username):

"""Top payees ranking"""

4.6 Control Flow

- For loops: Iterate over transactions, categories
- While loops: Main menu loop, PIN retry loop
- If-elif-else: Menu navigation, transaction validation
- Break/Continue: Exit menus, skip invalid transactions

5. Functional Requirements

5.1 Account Management

- FR-001: Create accounts with unique usernames
- FR-002: Require 4-digit PIN for login
- FR-003: Allow PIN changes
- FR-004: Show balance

5.2 Transfers

- FR-005: Allow transfers between accounts
- FR-006: Validate sufficient funds
- FR-007: Record both sender and receiver transactions

5.3 QR Payments

- FR-008: Generate mock QR payload string
- FR-009: Parse QR payload and process payment

5.4 Transactions

- FR-010: Maintain in-memory transaction history
- FR-011: Allow expense categorization
- FR-012: Generate reports (categories, monthly, top payees)

5.5 Security

- FR-013: Lock user after 3 failed PIN attempts

6. Non-Functional Requirements

- NFR-001: Response time ≤ 2 seconds
- NFR-002: CLI interface should be easy to navigate
- NFR-003: Error messages must be clear (e.g., “Insufficient Balance”)

7. Data Model Design

7.1 User Account (Mock)

user_account = {

"pin": "string", 4-digit PIN

"balance": 0.0, float

"transactions": [] list of transaction dicts

}

7.2 Transaction (Mock)

transaction = {

"id": "string",

"timestamp": "string",

"type": "string", (deposit / withdraw / transfer_in / transfer_out / qr_in / qr_out)

"amount": "float",

"note": "string",

"category": "string or None",

"counterparty": "string"

}

}

}

8. Error Handling

- Input validation errors → show retry prompts

- Authentication errors → track failed attempts, lock after 3 tries

- Transaction errors → insufficient funds, invalid usernames

9. Testing

- Unit test each function (`deposit`, `withdraw`, `transfer`, etc.)

- Test menu navigation flows

- Test invalid inputs and edge cases (zero balance, wrong PIN, etc.)