Solution

Coding Test on 22nd Sept 2025 - FIL Fresher Training

Architecture Overview

money.py
Money type with validations

accounts.py
Account hierarchy

exceptions.py
Custom typed exceptions

strategies.py
Interest calculations

transactions.py
Immutable transactions

bank.py
Main orchestrator

mixins.py
Cross-cutting concerns

utils.py
Helper functions

Core Components Detail

money.py

- **Decimal + currency** combination with operator overloads
- Built-in validations for currency operations

transactions.py

- **Immutable & hashable** Transaction objects
- Enables safe set operations and deduplication

mixins.py

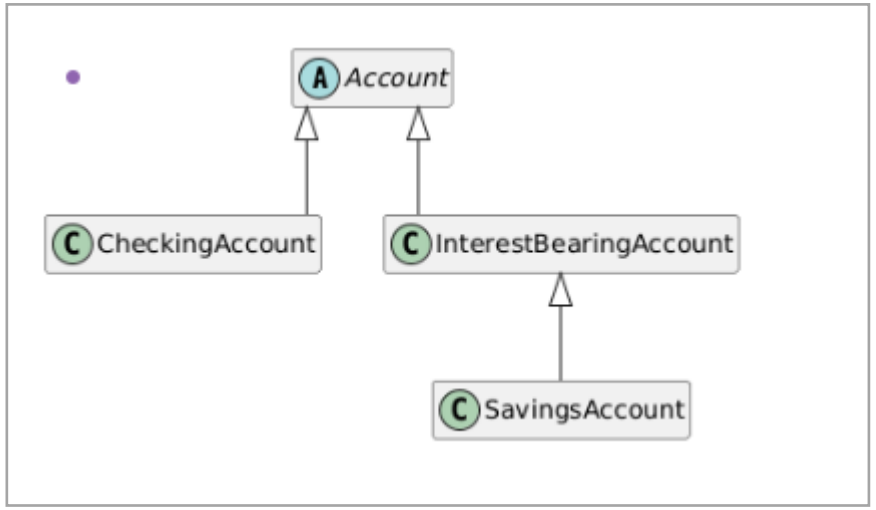
- **JSONSerializable:** Data persistence capability
- **Auditable:** Transaction tracking & logging

OOP Mapping to Requirements

Encapsulation

- **Private balance field** with property access
- Read-only properties ensure data integrity

Inheritance Hierarchy



Multiple Inheritance

- SavingsAccount + JSONSerializable + Auditable
- Clean separation of concerns

Exception Handling Strategy

Specific Exception Types

- **Currency mismatch:** Prevents invalid operations
- **Overdraft violation:** Business rule enforcement
- **Unknown account:** Data integrity protection

Best Practices

```
# Preserve exception context raise CustomException("Clear message")
from e # Raise at invariant boundaries if balance < withdrawal_amount:
raise InsufficientFundsException()
```

Benefits

- Clear error messages for debugging
- Context preservation with **raise ... from e**

Magic Methods & Testing

Magic Methods Implementation

- **Money arithmetic:** __add__, __sub__, rich comparisons
- **Account.__len__:** Returns transaction count
- **Transaction.__hash__:** Enables set usage for dedup

Testing Strategy

- **Unit tests per module:** Comprehensive coverage
- **Deterministic fixtures:** Fixed dates & amounts
- **Behavior-driven assertions:** Test OOP semantics

```
# Example: Testing polymorphic behavior assert
isinstance(savings.apply_interest(), Money) assert
checking.account_type == "CHECKING"
```

Complexity & Extensibility

Pluggable Architecture

- **Strategy Pattern:** Multiple interest calculation methods
- **Mixin Pattern:** Decoupled cross-cutting concerns
- **Fee strategies:** Extensible business rule engine

Key Takeaways

- Clean separation of concerns through proper OOP design
- Extensible architecture supporting business growth
- Robust error handling and testing methodology
- Real-world application of OOP principles

Complexity & Extensibility

Pluggable Architecture

- **Strategy Pattern:** Multiple interest calculation methods
- **Mixin Pattern:** Decoupled cross-cutting concerns
- **Fee strategies:** Extensible business rule engine

Key Takeaways

- Clean separation of concerns through proper OOP design
- Extensible architecture supporting business growth
- Robust error handling and testing methodology
- Real-world application of OOP principles

What We'll Learn Today (Quantize)

Function Overview

The `quantize_2` function rounds decimal numbers to exactly **two decimal places** using precise mathematical rules.

Key Benefits:

- **Financial Accuracy:** Perfect for currency calculations ($10.58)
- **Consistent Rounding:** Uses standard rounding rules (0.005 rounds up)
- **No Floating-Point Errors:** Essential for financial calculations

Why This Matters: In financial software, even tiny rounding errors can compound into significant problems. This function eliminates those risks.

Breaking Down the Code - Function Definition

```
def quantize_2(amount: Decimal) -> Decimal:
```

Understanding Each Part:

- `def quantize_2(...)`: Defines our function named **quantize_2**
- `amount: Decimal`: Type hint - expects a **Decimal** object (not regular float)
- `-> Decimal`: Return type hint - function returns a **Decimal** object

Why Decimal Instead of Float?

The **Decimal** type provides high-precision calculations, avoiding the binary representation errors that can occur with standard float numbers.

Breaking Down the Code - The Logic

```
return amount.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
```

Understanding the Components:

- `amount.quantize(...)`: Core method that rounds the number
- `Decimal("0.01")`: Template for rounding - specifies **2 decimal places**
- `rounding=ROUND_HALF_UP`: Traditional rounding method

How ROUND_HALF_UP Works:

- Numbers ending in 5 or greater → **Round UP**
- Numbers less than 5 → **Round DOWN**

Why Not Use Python's Built-in round()?

```
return amount.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
```

The Problem with Float Precision:

```
price = 10.99 tax = 0.075 total = price * (1 + tax) # 10.99 * 1.075 print(total)
```

```
Output: 11.814249999999999
```

The Issue: Standard float numbers can have tiny, hidden inaccuracies due to binary storage representation. This is dangerous for financial calculations!

Solution:

Using `Decimal` with `quantize()` ensures precise results that match financial expectations.

Practical Examples

```
from decimal import Decimal, ROUND_HALF_UP
def quantize_2(amount: Decimal) -> Decimal:
return amount.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)

# Test scenarios:
price1 = Decimal("12.345")
price2 = Decimal("12.3449")
price3 = Decimal("12.3")
```

Results:

Input	Output	Explanation
12.345	12.35	Rounds up (5 or greater)
12.3449	12.34	Rounds down (less than 5)
12.3	12.30	Adds trailing zero for consistency

Key Takeaways

What You've Learned:

- **Precision Matters:** Financial calculations require exact decimal handling
- **Decimal > Float:** Use Decimal type for money calculations
- **Quantize Method:** Powerful tool for consistent decimal place formatting
- **ROUND_HALF_UP:** Standard rounding rule for financial applications

Best Practice:

Always use `Decimal` with `quantize()` for any financial calculations in your applications. This prevents costly rounding errors and ensures accuracy.

Money Class Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

📋 Requirements

1. Decimal Precision & Currency

Represents money using `decimal.Decimal` for precision with currency support (e.g., "INR")

2. Same-Currency Arithmetic

Supports `+` and `-` operations only between same currency Money objects

3. Comparison Operations

Implements `==, <, <=, >, >=` comparisons between Money objects

4. String Representation

Provides meaningful `__str__` and `__repr__` methods

5. Currency Validation

Raises `CurrencyMismatchError` for cross-currency operations

⚙️ Implementation

`@dataclass + Decimal fields`

```
amount: Decimal
currency: str = "INR"
```

Uses dataclass with Decimal type and __post_init__ for validation

`__add__ & __sub__ methods`

```
def __add__(self, other):
self._check_currency(other)
return Money(self.amount + other.amount)
```

Currency validation before arithmetic operations

`_check_currency method`

```
def _check_currency(self, other):
if self.currency != other.currency:
raise CurrencyMismatchError(...)
```

Private method ensures currency consistency

`@total_ordering + __eq__ & __lt__`

```
@total_ordering
def __eq__(self, other):
def __lt__(self, other):
```

Decorator provides all comparison methods from eq and lt

`__str__ & __repr__ methods`

```
def __str__(self): return f"{self.currency}
{self.amount}"
def __repr__(self): return f"Money(amount=
{self.amount}...)"
```

User-friendly and developer-friendly string formats

Transaction Class Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

📋 Requirements

Immutable Record:
Must represent a balance change that cannot be modified once created

Required Fields:
- amount (Money type)
- timestamp
- description

Hashable:
Must be usable as dictionary key or in sets

Printable:
Must have readable string representation

⚙️ Implementation

`frozen=True`
Makes all fields read-only

`frozen=True`
Auto-generates __hash__()

`@dataclass`
Auto-generates __repr__()

`slots=True`
Reduces memory overhead

Account Class Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

📋 Requirements

🏛️ Abstract Class

Must be abstract for specialization

📝 Fields Required

- id (string)
- owner (string)
- encapsulated balance
- ledger of transactions

⚙️ Core Methods

- deposit(money: Money)
- withdraw(money: Money)
- transfer(to: Account, money: Money)

⚙️ Implementation

✅ Abstract Implementation

```
class Account(ABC):
  @abstractmethod
  def account_type(self) -> str:
```

✅ Encapsulated Fields

```
self._id = id
self._owner = owner
self._balance: Money
self._ledger: List[Transaction]
```

✅ Business Logic

```
def deposit(self, money: Money):
def withdraw(self, money: Money):
def transfer(self, to: "Account", money:
Money):
```

Account Class Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

📋 Requirements

🔒 Encapsulation

- Read-only balance property
- No direct balance mutation
- Protected internal fields

🎯 Special Methods

- __len__ (transaction count)
- __repr__ (string representation)
- account_type() abstract method

⚙️ Implementation

✅ Property Access

```
@property
def balance(self) -> Money:
  return self._balance
```

✅ Error Handling

```
raise InsufficientFundsError
raise CurrencyMismatchError
raise InvalidOperationError
```

🎓 Key Learning Points

Abstraction

ABC ensures subclasses implement account_type(). Base class provides common functionality.

Encapsulation

Private fields (_balance, _ledger) with controlled access via properties and methods.

Transaction Atomicity

Transfer method includes rollback mechanism to maintain data consistency.

Validation & Error Handling

Comprehensive validation with specific exceptions for different error scenarios.

Account Types - Inheritance Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

1 InterestBearingAccount → Abstract Class

Requirement: Create an abstract class with `apply_interest()` method to be called during month-end

⬇️

Solution Implementation:

```
class InterestBearingAccount(Account, ABC): @abstractmethod def apply_interest(self) -> None: ...
```

- ✅ Inherits from Account and ABC (Abstract Base Class)
- ✅ Defines abstract method `apply_interest()` that must be implemented by subclasses

Account Types - Inheritance Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

2 SavingsAccount → Interest Rate Implementation

Requirement: Inherit from InterestBearingAccount with interest_rate (3-7% p.a., monthly pro-rated)

⬇️

Solution Implementation:

```
class SavingsAccount(InterestBearingAccount, JSONSerializable, Auditable): def __init__(self, id, owner, opening_balance, interest_rate):
self.interest_rate = Decimal(str(interest_rate)) def apply_interest(self) -> None: monthly_rate = (self.interest_rate / Decimal("12")) /
Decimal("100") inc = Money(self.balance.amount * monthly_rate, self.balance.currency)
```

- ✅ Inherits from `InterestBearingAccount`
- ✅ Stores interest_rate as `Decimal` for precision
- ✅ Implements `monthly pro-rated interest` calculation
- ✅ Overrides the abstract `apply_interest()` method

Account Types - Inheritance Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

3 CheckingAccount → Overdraft Implementation

Requirement: Inherit from Account with overdraft_limit allowing negative balances up to the limit

⬇️

Solution Implementation:

```
class CheckingAccount(Account): def __init__(self, id, owner, opening_balance, overdraft_limit): self.overdraft_limit = overdraft_limit def withdraw(self, money: Money) -> None: projected = self.balance.amount - money.amount if projected < -self.overdraft_limit.amount: raise InsufficientFundsError("Overdraft limit exceeded")
```

- ✅ Inherits directly from `Account`
- ✅ Stores `overdraft_limit` as Money object
- ✅ Overrides `withdraw()` method to check overdraft
- ✅ Allows negative balance up to the limit

Account Types - Inheritance Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

4 Method Overriding & super() Usage

Requirement: Demonstrate method overriding and super() usage

⬇️

Solution Implementation:

```
# Method Overriding Examples: - SavingsAccount.apply_interest() → overrides abstract method - CheckingAccount.withdraw() → overrides base withdraw logic - Both classes override account_type() → return specific types # super() Usage: super().__init__(id, owner, opening_balance) # Calls parent constructor
```

- ✅ Method overriding demonstrated in multiple places
- ✅ super() used to call parent constructors
- ✅ Polymorphic behavior through overridden methods

Account Types - Inheritance Implementation

Requirement Analysis & Solution Mapping - FIL Fresher Training

5 Multiple Inheritance with Mixin

Requirement: Add a mixin (e.g., JSONSerializable) to show multiple inheritance

⬇️

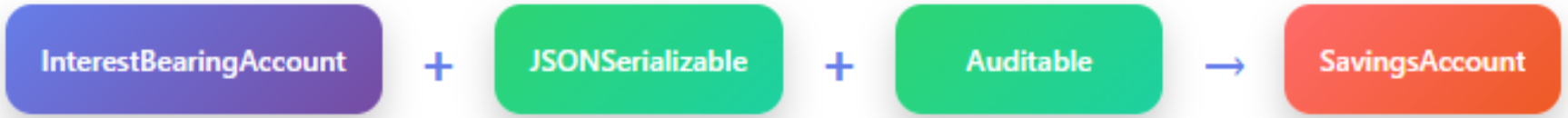
Solution Implementation:

```
class SavingsAccount(InterestBearingAccount, JSONSerializable, Auditable): def __init__(self, ...): super().__init__(id, owner, opening_balance) Auditable.__init__(self) # Explicit mixin initialization
```

- ✅ Multiple inheritance from 3 classes
- ✅ JSONSerializable mixin added
- ✅ Auditable mixin also included
- ✅ Proper initialization of multiple parent classes

Multiple Inheritance in Python

Understanding SavingsAccount Class Design



What You'll Learn

- ✓ What is multiple inheritance?
- ✓ Understanding parent classes and mixins
- ✓ Proper initialization patterns
- ✓ Method Resolution Order (MRO)
- ✓ Real-world application examples

What is Multiple Inheritance?

Understanding SavingsAccount Class Design

Multiple inheritance allows a class to inherit features from **multiple parent classes** simultaneously.

```
class SavingsAccount(InterestBearingAccount, JSONSerializable, Auditable): # This class inherits from
THREE different parents pass
```

Key Benefits

- ✓ **Code Reuse:** Combine functionality from multiple sources
- ✓ **Modularity:** Each parent provides specific, focused features
- ✓ **Flexibility:** Mix and match capabilities as needed

Breaking Down the Parents

Understanding SavingsAccount Class Design

1. Primary Parent: InterestBearingAccount

- ✓ **Main functionality:** Core account operations
- ✓ **Relationship:** SavingsAccount "*IS A*" InterestBearingAccount
- ✓ **Provides:** Balance management, interest calculation

2. Mixin Classes

JSONSerializable

- ✓ Adds JSON export capability
- ✓ Provides `.to_json()` method
- ✓ Self-contained functionality

Auditable

- ✓ Adds logging capabilities
- ✓ Tracks account activities
- ✓ Requires initialization setup

```
# Each parent contributes different capabilities: account = SavingsAccount("123", "John", 1000) # From
InterestBearingAccount: account.calculate_interest() account.get_balance() # From JSONSerializable:
json_data = account.to_json() # From Auditable: account.log_transaction("Deposit", 500)
```

The Critical __init__ Pattern

Understanding SavingsAccount Class Design

```
class SavingsAccount(InterestBearingAccount, JSONSerializable, Auditable): def __init__(self, id, owner,
opening_balance): # Step 1: Initialize primary parent via MRO super().__init__(id, owner, opening_balance)
# Step 2: Explicitly initialize mixins that need setup Auditable.__init__(self) # Note: JSONSerializable
doesn't need initialization
```

Why This Pattern?

- ✓ **super().__init__():** Follows Method Resolution Order (MRO) - handles the main parent
- ✓ **Direct calls:** Guarantees mixin initialization even if MRO chain breaks
- ✓ **Explicit is better:** Clear, predictable, and robust
- ✓ **Skip if not needed:** JSONSerializable has no setup requirements

Method Resolution Order (MRO)

Python follows a specific order when looking for methods: `SavingsAccount → InterestBearingAccount → JSONSerializable → Auditable → object`

You can check this with `SavingsAccount.__mro__`

🚗 Car Building Analogy

Understanding SavingsAccount Class Design

Think of multiple inheritance like building a custom car from different kits:

InterestBearingAccount
Base Car Kit
Chassis, engine, wheels
(Main assembly line)

Auditable
Black Box Kit
Flight recorder
(Needs installation & activation)

JSONSerializable
GPS Module Kit
Navigation system
(Just bolt it on)

```
def __init__(self, id, owner, opening_balance): # Build the base car (main assembly line)
super().__init__(id, owner, opening_balance) # Install and activate the black box
Auditable.__init__(self) # GPS module is ready to use (no setup needed)
```

Key Takeaways

- ✓ **Multiple inheritance** combines focused functionalities
- ✓ **Primary parent** provides core "IS A" relationship
- ✓ **Mixins** add specific, reusable capabilities
- ✓ **Initialization** must be handled carefully and explicitly
- ✓ **This pattern** is common in professional Python codebases

Bank Orchestrator Implementation

Requirement vs Solution Mapping - FIL Fresher Training

📋 Requirements

1. Factory Pattern for Account Creation

Creates accounts with factory class method `Bank.from_config` or `Bank.create_*`

2. Account Lookup by ID

Looks up accounts by id with proper error handling

💡 Implementation

✅ Factory Method Implemented

`@classmethod create_default()` provides factory pattern

```
@classmethod
def create_default(cls) -> "Bank":
return cls()
```

✅ Account Lookup with Exception Handling

`get(account_id)` method with proper exception handling

```
def get(self, account_id: str) -> Account:
try:
return self._accounts[account_id]
except KeyError as e:
raise AccountNotFoundError(account_id) from e
```

Bank Orchestrator Implementation

Requirement vs Solution Mapping - FIL Fresher Training

📋 Requirements

3. Month-End Processing

Processes month-end using duck-typed strategy:
`strategy.apply_month_end(account)` called for each account

4. Total Assets Calculation

Provides `total_assets()` returning a Money sum across all accounts

💡 Implementation

✅ Duck-Typed Strategy Pattern

`monthly_process(strategy)` with hasattr check

```
def monthly_process(self, strategy) -> None:
for acct in self._accounts.values():
if hasattr(strategy, "apply_month_end"):
strategy.apply_month_end(acct)
```

✅ Money Sum Aggregation

`total_assets()` sums all account balances

```
def total_assets(self) -> Money:
total = sum((a.balance.amount for a in self._accounts.values()),
start=Money(0, self._currency).amount)
return Money(total, self._currency)
```

🎯 Key Design Patterns Demonstrated:

- **Factory Pattern:** create_default() method for object creation
- **Duck Typing:** hasattr() check for strategy compatibility
- **Exception Chaining:** Proper error handling with custom exceptions
- **Aggregation:** Sum pattern for calculating total assets
- **Encapsulation:** Private _accounts dictionary with public interface

Strategy Pattern Implementation

Abstraction & Duck Typing in Python

📋 REQUIREMENT

6. Strategies (Abstraction & Duck Typing)

Define an `InterestStrategy` ABC with `apply_month_end(account)`.

Provide at least one concrete strategy (e.g., `SimpleInterestStrategy`) that calls `apply_interest()` where applicable.

What we need to create:

- Abstract base class for interest strategies
- Define interface with `apply_month_end()` method
- Concrete implementation that handles different account types
- Use duck typing to check if account supports interest

💡 SOLUTION

```
from __future__ import annotations
from abc import ABC, abstractmethod
from .accounts import Account, InterestBearingAccount

class InterestStrategy(ABC):
@abstractmethod
def apply_month_end(self, account: Account) -> None:
...

class SimpleInterestStrategy(InterestStrategy):
        def apply_month_end(self, account: Account) -> None:
                if isinstance(account,InterestBearingAccount):
                        account.apply_interest()
```

How the solution works:

- **ABC**: Defines the contract all strategies must follow
- **Abstract Method**: Forces concrete classes to implement `apply_month_end`
- **Duck Typing**: Checks if account "looks like" an interest-bearing account
- **Type Safety**: Uses isinstance() for runtime type checking

Python Mixins Implementation

Requirement Analysis & Solution Mapping

📋 Requirements

1. JSONSerializable Mixin

Must provide:

- `to_json()` method
- `from_json()` class method
- Use only `public API` (no private attributes)

2. Auditable Mixin

Must provide:

- `created_at` timestamp
- `updated_at` timestamp
- Auto-update on `balance change`

⚙️ Implementation

JSONSerializable Class

```
class JSONSerializable:
    def to_json(self) -> str:
        data = {}
        # Collect public attrs only
        for k, v in self.__dict__.items():
            if not k.startswith("_"):
                data[k] = v if isinstance(v,
                    (int, float, str, bool,
                     type(None))) else str(v)
        return json.dumps(data, default=str)

    @classmethod
    def from_json(cls, data: str):
        obj = json.loads(data)
        return cls(**obj)
```

Auditable Class

```
class Auditable:
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.created_at = datetime.utcnow()
        self.updated_at = self.created_at

    def _touch(self):
        # Updates timestamp
        self.updated_at = datetime.utcnow()
```

Python Mixins Implementation

Requirement Analysis & Solution Mapping

🎯 Key Implementation Details

- **Public API Only:** Code checks `not k.startswith("_")` to exclude private attributes
- **Type Safety:** Uses TypeVar and proper type hints for better code quality
- **Timestamp Management:** `created_at` set once, `updated_at` via `_touch()`
- **Mixin Pattern:** Uses `super().__init__()` for proper inheritance chain
- **JSON Handling:** Converts non-primitive types to strings for serialization
- **Extensibility:** Both mixins can be combined with any base class

Exception Design & Best Practices

Requirements vs Implementation Mapping

📋 Requirements

Custom Exception Classes

Create specific exceptions: BankError, CurrencyMismatchError, InsufficientFundsError, AccountNotFoundError, InvalidOperationError

Specific Exception Handling

Use specific exceptions and avoid bare except: clauses

Meaningful Error Messages

Use raise with meaningful messages and consider exception chaining

Proper Exception Structure

Use try/except/else/finally at Bank or I/O boundaries

✅ Implementation

```
# Base exception class for all bank operations
class BankError(Exception):
"""Base class for all bank-related exceptions."""
        pass

# Specific exception classes inheriting from BankError

class CurrencyMismatchError(BankError):
        pass
class InsufficientFundsError(BankError):
    pass

class AccountNotFoundError(BankError):
        pass

class InvalidOperationError(BankError):
        pass
```

Exception Design & Best Practices

Requirements vs Implementation Mapping

Key Design Benefits

- **Inheritance Hierarchy:** All custom exceptions inherit from `BankError`, allowing catch-all handling when needed
- **Specific Error Types:** Each exception represents a `specific business scenario` for targeted error handling
- **Clean Architecture:** Exceptions are `domain-specific` and provide clear error categorization
- **Extensibility:** Easy to add new exception types while maintaining the `established hierarchy`

Magic/Dunder Methods Implementation

Banking System - Requirements vs Solution Mapping

Money Class

📋 **Requirements:**

- __add__
- __sub__
- __eq__
- ordering methods
- __repr__

✅ **Implementation:**

✓ `__add__`	Currency-safe addition
✓ `__sub__`	Currency-safe subtraction
✓ `__eq__`	Equality comparison
✓ `__lt__`	@total_ordering provides all
✓ `__repr__`	Developer representation

Account Class

📋 **Requirements:**

- __len__
- __repr__

✅ **Implementation:**

✓ `__len__`	Returns ledger size
✓ `__repr__`	Shows account details

Note: Abstract base class implemented in SavingsAccount and CheckingAccount

Transaction Class

📋 **Requirements:**

- __hash__
- __repr__

✅ **Implementation:**

✓ `__hash__`	Frozen dataclass provides
✓ `__repr__`	Dataclass auto-generates

Smart Implementation: Using frozen=True dataclass automatically provides both methods

🎯 Key Implementation Highlights

🔒 Type Safety

All magic methods include proper type checking and return NotImplemented when appropriate

💱 Currency Validation

Money operations validate currency compatibility before performing calculations

🎨 Smart Decorators

@total_ordering reduces boilerplate by generating all comparison methods from __eq__ and __lt__

📊 Dataclass Benefits

Frozen dataclass automatically provides __hash__ and __repr__ for Transaction class

Appendix