The provided Python code implements a banking system using object-oriented programming (OOP) principles, incorporating **inheritance**, **polymorphism**, and **Pydantic** for data validation. Below is a detailed description of the code's structure, components, and functionality:

Overview

The code defines a banking system with a base class BankAccount and two derived classes: SavingsAccount and CheckingAccount. It uses Pydantic to enforce data validation for account attributes and demonstrates polymorphism through overridden methods. The system supports basic banking operations like depositing, withdrawing, and checking balances, with specific behaviors for savings and checking accounts.

Key Components

1. **Pydantic Model: AccountData**
    - **Purpose**: Ensures data validation for account attributes.
    - **Attributes**:
        - account_holder: A string (enforced by constr) with a minimum length of 1, maximum length of 100, and restricted to letters and spaces (via regex ^[A-Za-z\s]+$).
        - account_number: A positive integer (enforced by PositiveInt).
        - balance: A float with a default value of 0.0.
        - interest_rate: An optional positive float (for savings accounts).
        - overdraft_limit: An optional positive float (for checking accounts).
    - **Role**: Validates input data before creating account objects, ensuring type safety and constraints (e.g., positive account numbers, valid names).
2. **Base Class: BankAccount**
    - **Purpose**: Serves as the parent class for all account types, defining common attributes and methods.
    - **Attributes**:
        - _account_holder: Stores the account holder's name (string).
        - _account_number: Stores the account number (integer).
        - _balance: Stores the account balance (float).
    - **Methods**:
        - __init__(self, account_data: AccountData): Initializes the account with validated data from the AccountData model.
        - deposit(self, amount: float) -> str: Adds a positive amount to the balance, using Pydantic's PositiveFloat for validation. Returns a formatted string with the deposit result.
        - withdraw(self, amount: float) -> str: Subtracts a positive amount from the balance if sufficient funds are available. Returns a formatted string with the withdrawal result or an error message.
        - get_balance(self) -> str: Returns the current balance as a formatted string.
        - get_account_info(self) -> str: Returns account holder and account number details as a string.

    - **Note**: Methods use string formatting for consistent output (e.g., Deposited $200.00. New balance: $1200.00).
3. **Derived Class: SavingsAccount**
    - **Purpose**: Extends BankAccount to represent a savings account with an interest rate and a withdrawal fee.
    - **Attributes**:
        - Inherits all attributes from BankAccount.
        - _interest_rate: A float (default 0.02 or 2%) for calculating interest, set via AccountData or default value.
    - **Methods**:
        - __init__(self, account_data: AccountData): Initializes the savings account, calling the parent constructor and setting the interest rate.
        - withdraw(self, amount: float) -> str (Polymorphic): Overrides the base withdraw method to include a $5 withdrawal fee. Validates the amount and checks if the balance covers both the amount and fee.
        - apply_interest(self) -> str: Calculates and adds interest to the balance based on the interest rate. Returns a formatted string with the result.
    - **Polymorphism**: The withdraw method behaves differently from the base class, applying a fee specific to savings accounts.
4. **Derived Class: CheckingAccount**
    - **Purpose**: Extends BankAccount to represent a checking account with an overdraft limit.
    - **Attributes**:
        - Inherits all attributes from BankAccount.
        - _overdraft_limit: A float (default 100) allowing withdrawals beyond the balance up to this limit, set via AccountData or default value.
    - **Methods**:
        - __init__(self, account_data: AccountData): Initializes the checking account, calling the parent constructor and setting the overdraft limit.
        - withdraw(self, amount: float) -> str (Polymorphic): Overrides the base withdraw method to allow withdrawals up to the balance plus the overdraft limit.
    - **Polymorphism**: The withdraw method allows overdrafts, unlike the base or savings account implementations.
5. **Main Function**
    - **Purpose**: Demonstrates the functionality of the banking system.
    - **Execution**:
        - Creates two AccountData instances with validated data:
            - Savings account: "John Doe", account number 123456, initial balance $1000, interest rate 0.02.
            - Checking account: "Jane Smith", account number 456789, initial balance $500, overdraft limit $100.
        - Instantiates SavingsAccount and CheckingAccount objects.
        - Iterates through a list of accounts to demonstrate polymorphism by calling deposit, withdraw, and get_balance methods.
        - For savings accounts, also calls apply_interest.

        - Handles potential validation errors from Pydantic using a try-except block.
    - **Output**: Prints account information, transaction results, and balances in a formatted manner.

OOP Principles Demonstrated

- **Inheritance**:
    - SavingsAccount and CheckingAccount inherit from BankAccount, reusing its attributes and methods.
    - The super().__init__ call ensures proper initialization of the parent class.
- **Polymorphism**:
    - The withdraw method is overridden in both derived classes to provide specific behavior:
        - SavingsAccount: Adds a $5 fee to withdrawals.
        - CheckingAccount: Allows overdrafts up to the limit.
    - The main function calls withdraw on a list of accounts, and each account type executes its own version of the method.
- **Encapsulation**:
    - Attributes are prefixed with _ to indicate they are protected (convention in Python).
    - Pydantic ensures data validation before attributes are set.

Pydantic Integration

- **Validation**:
    - Ensures account_holder is a string with letters and spaces only.
    - Ensures account_number is a positive integer.
    - Validates balance, interest_rate, and overdraft_limit as appropriate numeric types.
- **Error Handling**: The try-except block in main catches validation errors (e.g., invalid account holder or negative account number).
- **Type Safety**: Pydantic's PositiveFloat and PositiveInt enforce positive values for monetary amounts and account numbers.

Example Output

Running the main function produces output like this:

text
CollapseWrap
Copy

```
Account Holder: John Doe, Account Number: 123456
Deposited $200.00. New balance: $1200.00
Withdrew $100.00 (+$5.00 fee). New balance: $1095.00
Account balance: $1095.00
```

```
Applied interest $21.90. New balance: $1116.90

Account Holder: Jane Smith, Account Number: 456789
Deposited $200.00. New balance: $700.00
Withdrew $100.00. New balance: $600.00
Account balance: $600.00
```

Key Features

- **Data Validation**: Pydantic ensures robust input validation (e.g., positive integers for account numbers, valid strings for names).
- **Polymorphic Behavior**: Different account types handle withdrawals differently, showcasing OOP flexibility.
- **Error Messages**: User-friendly messages for invalid inputs or insufficient funds.
- **Extensibility**: New account types can be added by extending BankAccount with custom behavior.

This code provides a clean, maintainable, and type-safe implementation of a banking system, leveraging Pydantic for validation and OOP principles for structure and flexibility.

Inheritance in object-oriented programming (OOP) is a mechanism where a new class (called a derived or child class) inherits attributes and methods from an existing class (called a base or parent class). This allows the derived class to reuse, extend, or modify the behavior and properties of the parent class, promoting code reusability and hierarchical organization.

Explanation with Context from the Banking Code

In the provided banking system code, inheritance is demonstrated through the SavingsAccount and CheckingAccount classes, which inherit from the BankAccount base class. Here's a detailed breakdown of how inheritance is implemented:

Key Components of Inheritance in the Code

1. **Base Class: BankAccount**
   - **Purpose**: Defines common attributes and methods for all types of bank accounts.
   - **Attributes**:
     - _account_holder (string): The name of the account holder.
     - _account_number (integer): The account number.
     - _balance (float): The current balance.
   - **Methods**:
     - __init__(self, account_data: AccountData): Initializes the account with validated data.
     - deposit(self, amount: float) -> str: Adds a positive amount to the balance.

        - withdraw(self, amount: float) -> str: Deducts an amount if sufficient funds are available.
        - get_balance(self) -> str: Returns the current balance.
        - get_account_info(self) -> str: Returns account holder and number details.
    - **Role**: Acts as the parent class, providing shared functionality for all account types.
2. **Derived Class: SavingsAccount**
    - **Inheritance**: Defined as class SavingsAccount(BankAccount), indicating it inherits from BankAccount.
    - **Attributes**:
        - Inherits _account_holder, _account_number, and _balance from BankAccount.
        - Adds _interest_rate (float): Specific to savings accounts for interest calculations.
    - **Methods**:
        - __init__(self, account_data: AccountData): Calls the parent's __init__ using super().__init__(account_data) to initialize inherited attributes, then sets _interest_rate.
        - withdraw(self, amount: float) -> str: Overrides the parent's withdraw method to include a $5 fee.
        - apply_interest(self) -> str: A new method specific to savings accounts, not present in the parent class.
    - **Role**: Extends BankAccount with savings-specific features (interest rate, withdrawal fee).
3. **Derived Class: CheckingAccount**
    - **Inheritance**: Defined as class CheckingAccount(BankAccount), indicating it inherits from BankAccount.
    - **Attributes**:
        - Inherits _account_holder, _account_number, and _balance from BankAccount.
        - Adds _overdraft_limit (float): Specific to checking accounts for overdraft allowance.
    - **Methods**:
        - __init__(self, account_data: AccountData): Calls the parent's __init__ using super().__init__(account_data) to initialize inherited attributes, then sets _overdraft_limit.
        - withdraw(self, amount: float) -> str: Overrides the parent's withdraw method to allow withdrawals up to the balance plus the overdraft limit.
    - **Role**: Extends BankAccount with checking-specific features (overdraft limit).

How Inheritance Works in the Code

- **Reusability**: Both SavingsAccount and CheckingAccount inherit the deposit, get_balance, and get_account_info methods from BankAccount without redefining them. This reduces code duplication.

- **Extension**: Each derived class adds its own attributes (_interest_rate for SavingsAccount, _overdraft_limit for CheckingAccount) and methods (apply_interest for SavingsAccount).
- **Method Overriding**: The withdraw method is overridden in both derived classes to provide specific behavior, which also demonstrates **polymorphism**.
- **Super Call**: The super().__init__(account_data) call in the derived classes ensures that the parent class's initialization logic is executed, setting up shared attributes.

Example from the Code

In the main function:

python
CollapseWrapRun
Copy

```
savings = SavingsAccount(AccountData(account_holder="John Doe",
account_number=123456, balance=1000, interest_rate=0.02))
checking = CheckingAccount(AccountData(account_holder="Jane Smith",
account_number=456789, balance=500, overdraft_limit=100))
```

- Both savings and checking objects inherit the _account_holder, _account_number, and _balance attributes from BankAccount.
- They reuse the deposit and get_balance methods from BankAccount but use their own versions of withdraw.
- SavingsAccount adds the apply_interest method, which is unique to it.

Output Demonstrating Inheritance

When the main function runs:

text
CollapseWrap
Copy

```
Account Holder: John Doe, Account Number: 123456
Deposited $200.00. New balance: $1200.00
Withdrew $100.00 (+$5.00 fee). New balance: $1095.00
Account balance: $1095.00
Applied interest $21.90. New balance: $1116.90

Account Holder: Jane Smith, Account Number: 456789
Deposited $200.00. New balance: $700.00
Withdrew $100.00. New balance: $600.00
```

```
Account balance: $600.00
```

- The deposit and get_balance calls use the inherited BankAccount methods.
- The withdraw calls use the overridden methods specific to each class.
- The apply_interest call is only available for SavingsAccount.

Key Benefits of Inheritance

- **Code Reusability**: Common functionality (e.g., deposit, get_balance) is defined once in BankAccount and reused by derived classes.
- **Extensibility**: New account types (e.g., BusinessAccount) can inherit from BankAccount and add specific features.
- **Maintainability**: Changes to shared logic in BankAccount automatically apply to all derived classes.
- **Hierarchical Organization**: The code models a natural hierarchy where all accounts share basic features but have specialized behaviors.

Why Inheritance Matters in the Code

Inheritance allows the banking system to:

- Share common account functionality across different account types.
- Customize behavior for specific account types (e.g., fees for savings, overdrafts for checking).
- Maintain a clean, organized structure that's easy to extend or modify.

In summary, inheritance in the code enables SavingsAccount and CheckingAccount to build upon the foundation of BankAccount, reusing shared functionality while adding or modifying behavior to suit their specific needs.