Codebase Tutorial Summary

Comprehensive Analysis and Documentation

Generated On: June 12, 2025 at 11:54 PM

Document Type: Technical Documentation

Format: PDF Report

Table of Contents

Codebase Tutorial Summary

File: `main.py`

- This code begins by importing the 'Bank' class from the 'models.bank' module and the 'export_bank_data' function from the 'utils.exporter' module.

It then defines a function named 'sample_run'. This function does several things:

1. It creates an instance of the 'Bank' class.

2. It then uses this instance to create two bank accounts, one for 'Alice' and one for 'Bob', using the 'create_account' method of the 'Bank' class.

3. These accounts are credited with 1000 and 500 units respectively, using the 'deposit' method from their respective account objects.

4. After money has been deposited into these accounts, a transfer of 200 units from Alice's account to Bob's account is made using the 'transfer' method of the 'Bank' class.

5. The function then exports bank data by calling the 'export_bank_data' function.

6. Lastly, 'sample_run' returns the instance of 'Bank' class.

At the very end of the code, it checks if the script is being run directly (as opposed to being imported by another script), in which case it calls and executes the 'sample_run' function. This is a common pattern in Python scripts that are intended to be usable as both scripts and modules. When imported, the script's functionality can be used by other scripts, and when run directly, the script performs a specific task - in this case, the 'sample_run' function.

File: `__init__.py`

File: `account.py`

- This code defines an `Account` class representing a bank account.

The class defines an `__init__` method (a special method in Python classes which is automatically called when an object is instantiated) where an account owner and an initial `balance` (default value is 0) are required as input. It also initializes an empty list called `transactions` to record all transactions made on this account.

There is a `deposit` method that increases the account `balance` by a specified `amount`. It also appends a new `Transaction` object into the `transactions` list. A `Transaction` object is created with two parameters - the deposit amount and transaction type ('deposit').

The `withdraw` method subtracts a specified `amount` from the account balance. If the withdrawal `amount` is more than the current `balance`, it raises a `ValueError` exception indicating "Insufficient funds". If there are enough funds, it will also append a new `Transaction` object to the `transactions` list, indicating the withdrawal.

The `get_balance` returns the current balance of the account.

The `get_statement` function returns information about all transactions that have occurred on the account, in a dictionary format. Each `Transaction` object in the `transactions` list is converted to a dictionary by calling its `to_dict` method.

In summary, this code is a simplistic representation of a bank account, where one can deposit funds, withdraw funds (insufficient funds results in an error), check the balance, and get a statement of all transactions. It represents a simple bank ledger.

File: `bank.py`

- This code provides an object-oriented way to simulate a basic banking system. It is written in Python and consists of a class named `Bank`.

In our `Bank` class:

- An instance is initialized with an empty dictionary `self.accounts` which will store `Account` instances. It uses the convention of object names as keys.

- The `create_account` method takes `owner` (name) as a parameter. It first checks if an account with this name already exists in `self.accounts`. If it does, it raises a `ValueError`; else it creates a new `Account` instance with `owner` as a parameter and stores this instance in the `self.accounts` dictionary.

- The `get_account` method also takes `owner` as a parameter. This method is designed to return the `Account` instance for a given owner. If no account exists, it returns None.

- The `transfer` method handles transactions from one account to another. It takes `from_owner`, `to_owner`, and `amount` as parameters. It retrieves both the sender's and receiver's accounts using the `get_account` method. If either account does not exist, it raises a `ValueError`, if both exist, it withdraws the specified amount from `from_owner`'s account and deposits it into `to_owner`'s account.

This code assumes the existence of an `Account` class, implemented in the `account.py` file, following object-oriented principles, and having at least `withdraw` and `deposit` methods. The missing `Account` class would ideally contain rules for handling balance, depositing, and withdrawing the money. This also implies that there might be a directory structure given the use of the relative import dot (`from .account import Account`).

File: `transaction.py`

- This code defines a Python class called "Transaction". This class is a model that can be used to represent a financial transaction in a banking software or any other similar system.

Here's a detailed explanation of this class:

1. `datetime` is imported from Python's built-in `datetime` module. This module supplies classes for manipulating dates and times.

2. The `Transaction` class has a constructor method `__init__`. This method is automatically called when creating an object from this class. This is where you define the properties your class should have, using `self.argument`.

- The `self` keyword is a reference to instances of the class, allowing access to the attributes and methods in the class.

- `amount` holds the transaction amount.

- `transaction_type` represents the type of a transaction whether it's a deposit, withdrawal, etc.

- `timestamp` uses `datetime.now()` to capture the date and time the transaction is made. `datetime.now()` returns a datetime object containing the current local date and time.

3. The `Transaction` class also has a method `to_dict`, which turns an object of this class to a dictionary. This is typically useful for serializing the class instance (object) to a format that can be written into a file, sent over a network, or used in other parts of the program.

- The return data of method `to_dict` includes `amount`, `type`, and `timestamp`.

- `timestamp` uses `isoformat()` to return a string representing a datetime, stripping any trailing zero elements. This allows the datetime to be presented in a standardized and easily readable format.

In summary, you create an instance of the `Transaction` class by providing the `amount` and `transaction_type`. You can convert this instance to a dictionary using the `to_dict` method.

File: `__init__.py`

File: `exporter.py`

- This Python code is a simple procedure for exporting bank account data into a .json file.

First, it starts by importing the `json` library, which allows Python to work with JSON files.

The function `export_bank_data(bank, path="bank_data.json")` is defined next. This function takes in two arguments - `bank` and `path`. The `bank` parameter refers to the banking system where all the account details are stored. `path`, on the other hand, is used to define the location of the file which will be created. The default file path is set to `bank_data.json`.

Inside the function, an empty dictionary `data` is declared. This dictionary will be used to form the structure of our JSON file.

The `for` loop goes through every account in the `bank` object's dictionary of accounts, where each account is associated with its owner. For each account, it constructs a new dictionary and assigns it to the corresponding owner in the `data` dictionary. This new dictionary contains two key-value pairs: the account balance, retrieved through `acc.get_balance()`, and account transactions, retrieved through `acc.get_statement()`.

Finally, Python's `with open` statement is utilized to write our data to a file. Using `with open` ensures that the file is properly closed after it is no longer needed. `path, "w"` opens (or creates if it doesn't exist) the file at the provided location in write mode.

The line `json.dump(data, f, indent=2)` writes our `data` dictionary to that file in a JSON format. The `indent=2` parameter makes sure the JSON file is formatted in a readable way by adding 2-space indentation between each of the levels.

File: `__init__.py`