Codebase Tutorial Summary

Comprehensive Analysis and Documentation

Generated On: June 13, 2025 at 01:03 PM

Document Type: Technical Documentation

Format: PDF Report

Table of Contents

Codebase Tutorial Summary

File: `main.py`

- This script is a simple bank simulation.

1. We have two main modules here: `models.bank` and `utils.exporter`. `models.bank` contains the Bank class that we use to perform our various banking operations, while `utils.exporter` is a utility for exporting bank data.

2. The function `sample_run()` creates a new instance of the `Bank` class named `bank`.

3. Using the `bank` object, it creates two new bank accounts. One for a user named Alice and another for a user named Bob, using the `create_account` method of `bank`.

4. The script then simulates Alice depositing $5000 into her account and Bob depositing $1000 into his account using the `get_account(username).deposit(amount)` method.

5. Following this, it simulates a bank transfer from Alice to Bob of $1000 using the `transfer(from_account, to_account, amount)` method.

6. The resulting state of the bank, after these operations - i.e., the creation of accounts and transactions - is then exported, likely for record-keeping or similar purposes, using the `export_bank_data(bank)` method from utils.exporter module.

7. The `sample_run()` function returns the complete `bank` object, presumably for future manipulation or use.

8. Lastly, the script uses a Python construct `if __name__ == "__main__":`, which means that if the script is being run directly (as opposed to being imported as a module into another script), it will execute the `sample_run()` function.

In general, this script is a simple demonstration of creating accounts, performing transactions, and managing bank operations using an object-oriented approach in Python.

File: `__init__.py`

File: `account.py`

- This Python code defines a simplified Bank Account system.

To begin, it imports a `Transaction` class from a module named `transaction`.

Then, a `Class` named `Account` is defined with several methods:

1. The `__init__` constructor method: This method is used when an object is created from the `Account` class and it allows the class to initialize the attributes of the class. It takes the owner's name and the initial balance as parameters. Default value for a balance is 0 if not specified. It also initializes an empty list to hold the transactions.

2. `deposit` method: This method is for depositing a specific amount into the account. It adds the deposit amount to the current balance and also records this transaction by appending it to the transactions list.

3. `withdraw` method: This method is for withdrawing a certain amount from the account. However, before the withdrawal is made, there's a check to ensure that the requested amount is not greater than the available balance (`raise ValueError("Insufficient funds")`). If the amount is less than or equal to the balance, then it subtracts the amount from balance and keeps a record of this interaction by appending it to the transactions list.

4. `get_balance` method: This method simply returns the current balance of the account.

5. `get_statement` method: This method is for getting the account statement, which is a list of all the transactions conducted from the account. It returns the list of all transactions as dictionaries.

Each `Transaction` object has an amount and type (either deposit or withdrawal).

File: `bank.py`

- This Python script is defining a class called 'Bank' which has methods to manage Bank accounts.

First, we import `Account` from a local module named `account`.

Instantiating `Bank` defines an instance variable `self.accounts` to an empty dictionary {}.

`create_account(self, owner)`: This function creates an account for an owner. It checks if the owner already exists within the `self.accounts` dictionary. If the owner's account already exists, it raises a `ValueError`. If the owner's account does not exist, it creates a new `Account` with "owner" as its argument.

`get_account(self, owner)`: If given the name of an account owner, this function will return the `Account` associated with that owner from the `self.accounts` dictionary. If the given owner does not exist, it will return `None`.

`transfer(self, from_owner, to_owner, amount)`: This function represents the action of transferring a certain amount of money from one account to another. It first gets the `Account` objects associated with `from_owner` and `to_owner` using the `get_account` function. If either of this function call returns `None`, it means one of the accounts does not exist, so it raises a `ValueError`. If both accounts exist, it proceeds to withdraw the specified amount from the `from_owner` account, and deposit the same amount to the `to_owner` account.

File: `transaction.py`

- This piece of code is in Python and it illustrates a basic concept of object-oriented programming: classes and objects.

We first import the `datetime` module from `datetime`, which will allow us to work with Dates and Times.

The class named 'Transaction' models a financial transaction. It contains a constructor method `__init__()` that initializes several instance variables: `self.amount`, `self.transaction_type`, and `self.timestamp`.

- `self.amount` is the transaction amount.

- `self.transaction_type` is the type of transaction. For instance, it could be 'debit' or 'credit'.

- `self.timestamp` stores the date and time when the transaction is created. It automatically captures the current date and time using `datetime.now()` function which is part of the `datetime` module we imported.

Also, notice the use of `self` here. `self` is a convention and is used as a reference to the current instance of the class, which is used to access variables that belongs to the class.

This class contains a method `to_dict(self)` that converts the instance variables to a Python dictionary. The keys of the dictionary are the string representations of the respective instance variable names (`'amount'`, `'type'`, `'timestamp'`). Their associated values are the values stored in the instance variables (`self.amount`, `self.transaction_type`,`self.timestamp.isoformat()`).

- `self.timestamp.isoformat()` turns the datetime object into a string, which is more amenable to storage or transmission.

In summary, an instance of this class represents a single transaction, which includes the transaction amount, the transaction type, and the time of the transaction. The `to_dict()` method provides a method to extract this data out of the object in a standard way, which can be particularly useful when you want to store or transmit the data.

File: `__init__.py`

File: `exporter.py`

- This Python code is about extracting banking information and saving it into a JSON file. Let's go through it step by step.

First, the built-in `json` module is imported. This module provides methods that would allow us to encode and decode information into JSON, which is a popular data interchange format.

Next, a function `export_bank_data` is defined, which accepts two parameters: `bank` and `path`. The `bank` would be the source bank object from where we are extracting the account information, while `path` is the filepath where we would be storing our data. The path is optional, if not specified, it defaults to "bank_data.json".

The function initiates an empty dictionary `data`. It then loops over all accounts in the `bank`. The `bank.accounts.items()` returns a list of tuples, where each tuple contains the owner name and the corresponding account object (`owner, acc`).

For each account, it collects the following data:

- The balance using `acc.get_balance()` method.

- The list of transactions using `acc.get_statement()` method.

Both pieces of data are stored inside a temporary dictionary, where the key is the account `owner`, and the value is another dictionary containing "balance" and "transactions".

After gathering all the data, the function opens the file at the specified `path` in 'write' mode ('w'). `json.dump` is then used to write the `data` dictionary into this file. The built-in `open()` function and `with` statement ensure that the file is properly closed after operations are finished, this helps to prevent memory leaks.

The `indent` parameter in `json.dump()` is used to pretty-print the JSON data with specified indent level. It makes the JSON output human-readable.

In sum, this script exports key account information (names, balances, and transactions) from a `bank` object into a human-readable JSON file.

File: `__init__.py`