Design of the exercise:

Unfortunately I didn't finish the exercise, since I didn't have enough time.
There were a few methods to implement in debits_processor part of the system to finish it.

I will explain how I decided to design the system

I decided to split the system to 3 parts:

1. credits_processor
   A FastAPI server (can automatically validate input arguments validity)
   Exposes perform_advance api.
   For every request it
   a. Performs a transaction for credit
   b. Stores credit in the Credits table
   c. Stores row in the TransactionsCredits table
2. debits_processor
   I decided that this will be a scheduled routine, once a day.
   Since download_report might be a very heavy function (5 days worth of data). We don't want to run it many times.
   It will start everyday at the same time and will do the following:
   a. Create debits for all successful credits (there is a Debits table)
   b. Work with the report to identify:
      i. Failed transactions of debits, create new debits at the end of the plan as instructed.
      ii. Timed out transactions of debits, if we don't see 5 days transaction of debit, we create a new debit in the of the plan.
   c. Call the perform_transaction method of the processor to issue transactions for debits that were not processed yet and with due time before the debits_processor started to run.
3. DataBase.
   I implemented the database as a list of classes that are stored in memory of credits_processor. debit_processor sent get and post requests to interact with the data.
   Of course this is not the best solution, just to save time. Ideally, I want it to work with RDS DB. In my design there are no UPDATE queries, so the system can work with a wider choice of available databases and be more efficient.
   Also I used UUID for the same reason, this will allow us to write concurrently without conflicts.

Here are the tables:

```
3 usages
class Credit(BaseModel):
    id: str = str(uuid4())
    acc_id: str
    amount: float
    creation_datetime: str
```

```
3 usages
class TransactionsCredits(BaseModel):
    transaction_id: str
    credit_id: str
    creation_datetime: str
```

```
class TransactionsDebits(BaseModel):
    transaction_id: str
    debit_id: str
    creation_datetime: str
```

```
6 usages
class Debit(BaseModel):
    id: str = str(uuid4())
    credit_id: str
    amount: float
    due_date: str
```

```
3 usages
class ProcessedDebit(BaseModel):
    id: str
```

Future work:

1. To allow higher scale, we can forward the performe_advance request to a queue (e.g Kafka). Then we will introduce a new component that will consume the requests from the queue and store them in DB. We can use many consumers in parallel for increased performance.
2. Should wrap invocation of processor functions with try, accept, retry blocks to handle errors. The same with all DB operations.
3. Logging.
4. Implement debit_processor with Spark for parallelism, efficiency and distributive computing for performance. Pandas DataFrame are not scalable and not distributed.
5. Testing, writing more scenarios to ensure the system doesn't break the logic, mock the processor blackbox.
6. In order to check my code, I implemented in credit_processor methods that should not be there, such as:
    a. Download_report
    b. Update_report

    There are some more… this is only for "mocking", so I can run the code and see that it works, those methods shouldn't be there in the final version.