# Affirm Coding Challenge

**Affirm Loan Processing Service**

By Cheau-Long Ng

**Github Repo:** <https://github.com/cheaulongng/affirm_loan_processor>

**Programming Language:** Python

# Instructions to Run Program

**1. Run “process\_loans.py” script**

In terminal, change the working directory to "src" from the source codes, then run command below:

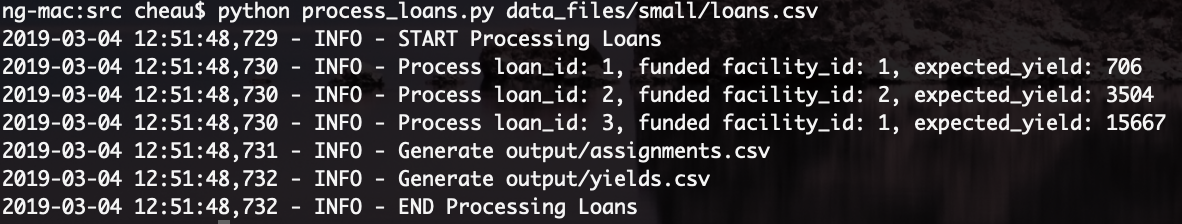
**python process\_loans.py <loan CSV file>**

**Example:**

**python process\_loans.py data\_files/loans.csv**

When the program finishes, it will generate two output files: assignments.csv, yields.csv

For example,



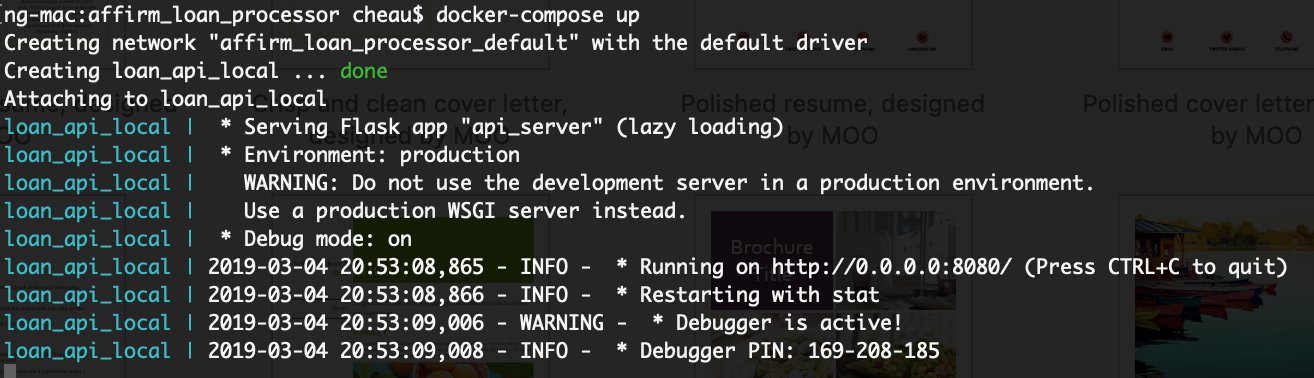
**2. Run API Server in Docker**

This is a bonus demo to illustrate processing a loan thru REST API. We can run the Loan API server in docker, this is a simple demo to process loan and get the facility yields.

To run the API thru docker-compose:

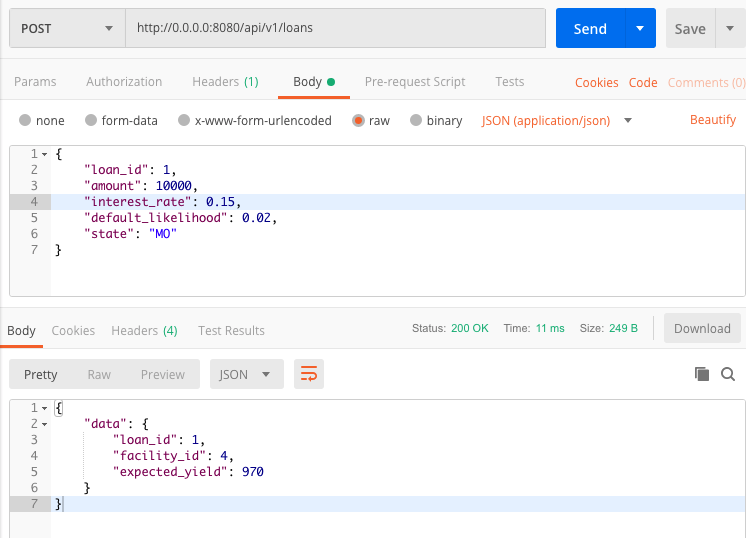
**docker-compose up**

For example,



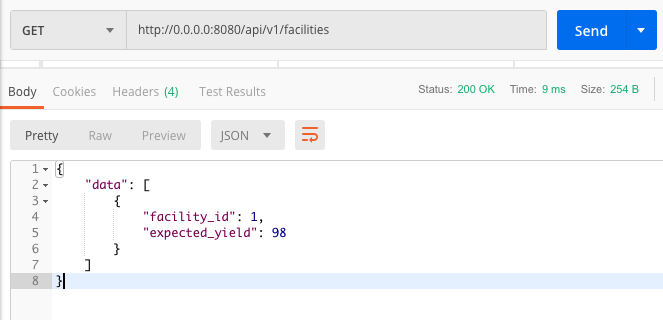
**Process Loan**

Method: POST <http://0.0.0.0:8080/api/v1/loans>

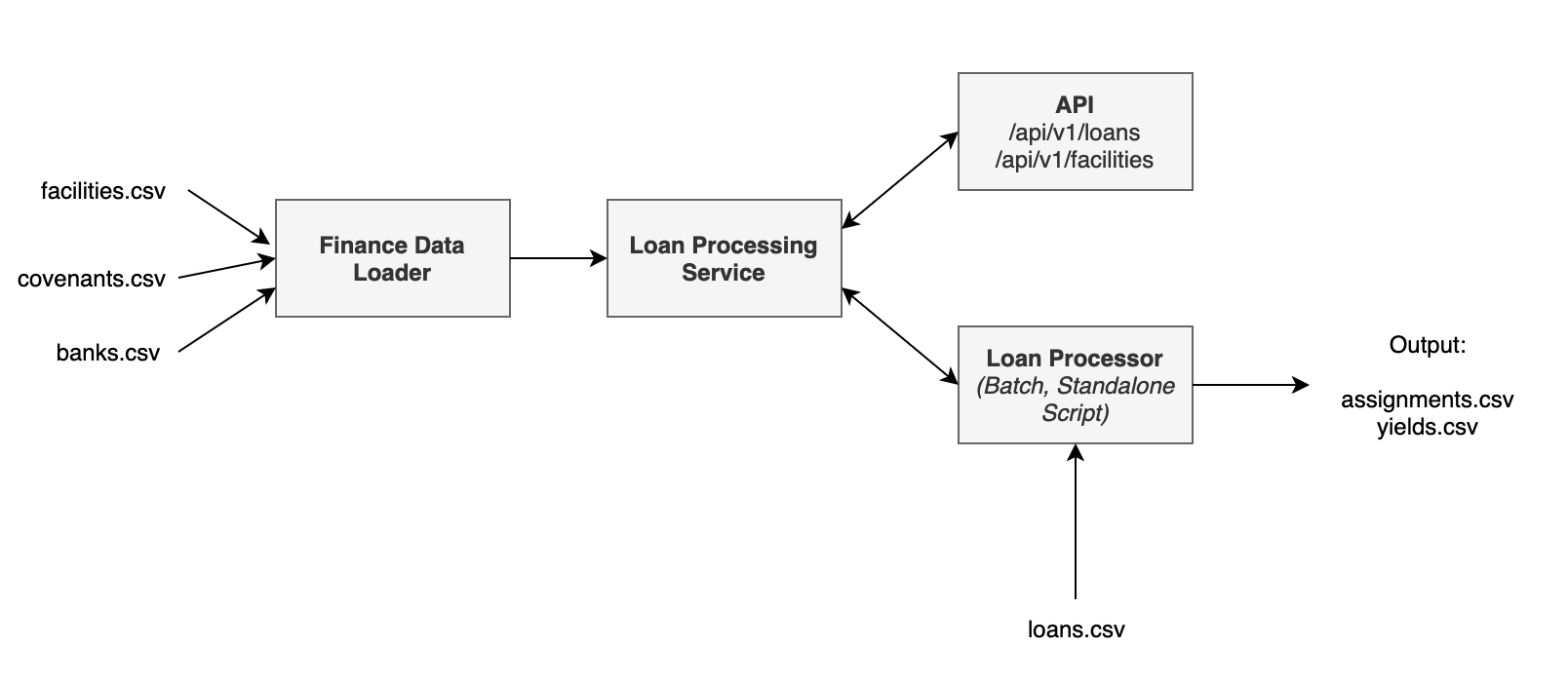


**Get Facility Yields**

Method: GET <http://0.0.0.0:8080/api/v1/facilities>



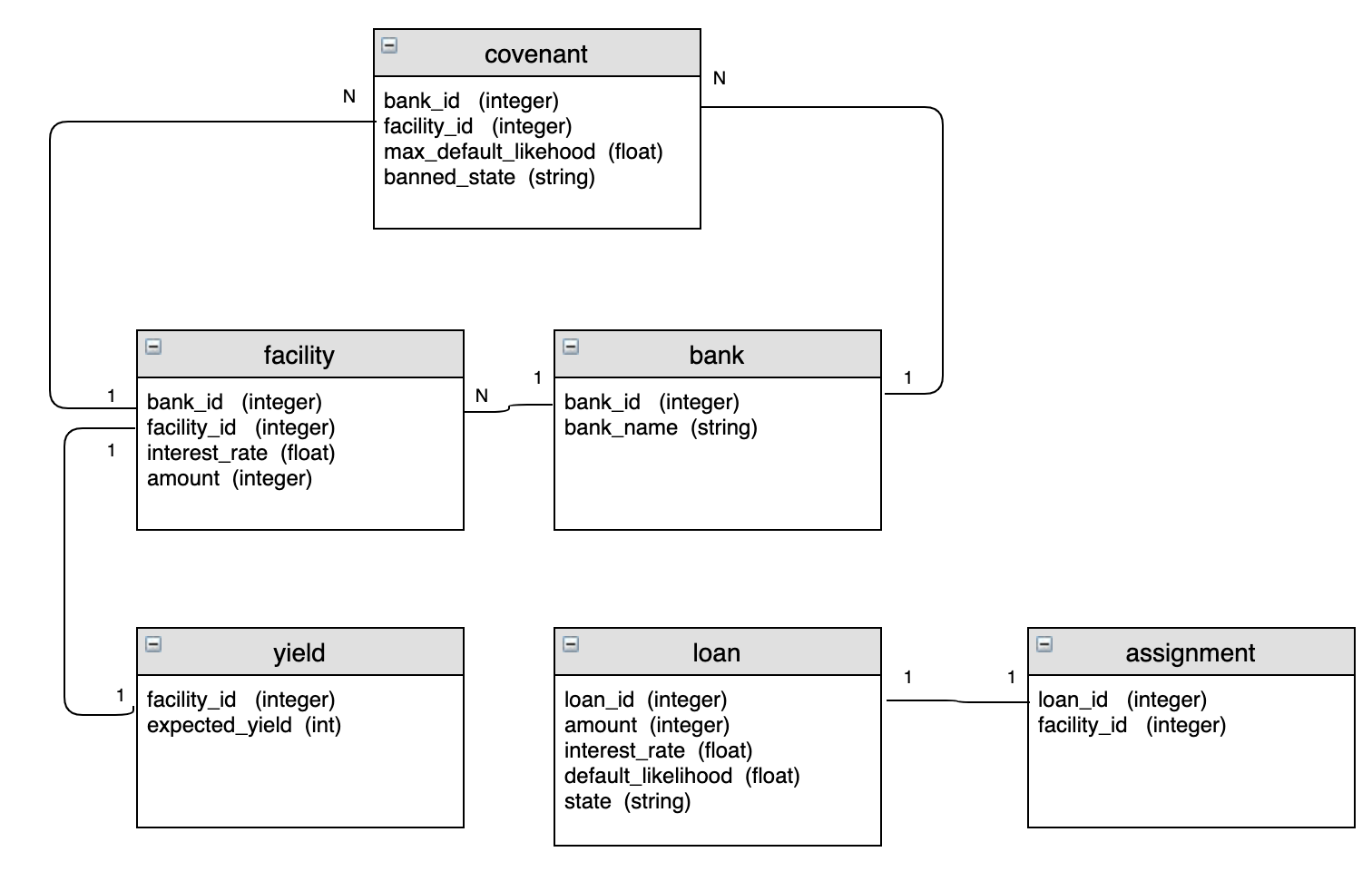
# Architecture



The solution consists of 4 major modules:

* **Finance Data Loader:** This module is used to load the required finance data (banks, facilities, covenants) from CSV files into memory.
* **Loan Processing Service:** This is the loan processing and assignment service by searching a facility that matches the criteria and rules to fund the loan.
* **Loan Processor:** A standalone loan processor by taking loan input to process the loans.
* **API:** REST API service for processing the loan and returning the facility yields

# Data Model



The data model shows the entity relationships between each model. It can be used to design a database solution to store the data. A new covenant or rule can be added to the “covenant” model. For example, minimum\_credit\_score can be added as new criteria for a loan to be qualified. This new rule can be added to the covenant model, which can then be used in the algorithm when processing the loan.

# Q & A

1. **How long did you spend working on the problem? What did you find to be the most difficult part?**

**Ans:** I spent a total of 10 hours in working on the problem. The time spent has two sections:

* **Good working solution (5 hours)**

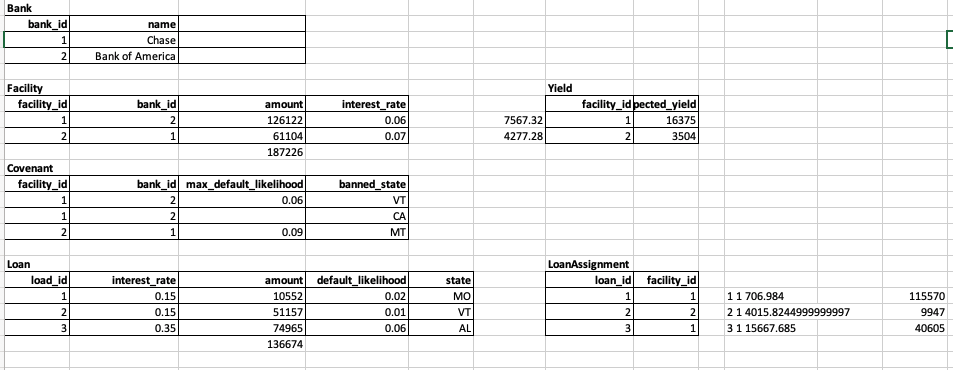
I spent about 3 hours to come out a rough framework and working program, which was able to find the facility and calculate the expected yields. However, when I validated the calculations against the expected results, I found one loan didn’t have the correct facility found. Thus I spent another 2 hours in debugging, analyzing the result sets, refactoring, testing the codes.

* **Enhanced Solution (5 hours)**

I few hours to continue refactor the codes for readability, maintainability, and reviewed any areas that could be optimized. As an inspiring fun, I prototyped a REST API to process a loan by using a common loan processing service, and dockerized the solution.

There are some challenging parts when working on the problem:

1. **Misunderstood the requirements.** I thought I understood the requirements but when I tested the program, I found out the selected facility and facility yield didn’t meet the expected result. So I had to review the requirements documentation few times to understand the rules and criteria. For example, it was unclear to me at first when a loan was funded, the facility’s amount needed to be subtracted from the funded amount. This caused the facility assignment algorithm to be incorrect. Same for the selection rules for the cheapest interest rate. At first it was difficult to look at the individual CSV file in multiple opened window. I put all the small set datasets in Excel as shown below, which was easier for me to understand the datasets and how the expected results were calculated. After some refactoring, QA, and verifying the results. A good solution came up. A lesson learned is to spend some time at the beginning to manually validate the calculations, for example in Excel, understand how it’s calculated, write the logics down before coding.



1. **Back and forth in selecting data structure and algorithms.** Initially I used array list to store banks, facilities, and covenants data. Later I discovered while in testing that the covenant’s max\_default\_likelihood actually applies to each facility even the value is missing from the row. And banned\_states is a collection to a bank or facility, then I designed Covenant data model below that banned\_states is a list. By further analyzing the data model, I decided to use hashtable (dict in python) to store the data as it’s easier to look up data thru key. For example, to find covenants, I can use key = “<bank\_id>\_<facility\_id>” for the selected facility to look up a covenant. As a result, I think the urge to optimize the codes for better performance caused more time spent in refactoring (and testing) the codes, like switching from array list to hashtable, etc.
2. **How would you modify your data model or code to account for an eventual introduction of new, as-of-yet unknown types of covenants, beyond just maximum default likelihood and state restrictions?**

**Ans:** Below is a data model class example (was implemented in the solution) can be designed to accommodate the new type of covenants. For example, min\_credit\_score can be added to this model. The loan assignment algorithm can be updated to use this new covenant when searching a facility to fund the loan.

class Covenant(object):  
 def \_\_init\_\_(self):  
 self.bank\_id = None  
 self.facility\_id = None  
 self.max\_default\_likelihood = None  
 self.banned\_states = []  
 self.min\_credit\_score = None

1. **How would you architect your solution as a production service wherein new facilities can be introduced at arbitrary points in time. Assume these facilities become available by the finance team emailing your team and describing the addition with a new set of CSVs.**

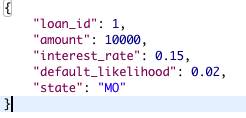
**Ans:** We can use database to store all finance data (facilities, banks, covenants, etc.). Actually, using database is an ideal solution so we can store all facts into a simple datastore instead of having multiple CSVs to manage and have version controls. When new facilities are provided thry CSV’s, we could load (ETL, or other data loading process) the new datasets into the “facilities” table. The loan processing service can always query the database to get the latest datasets. If performance becomes an issue, we can cache the data in memory (e.g. memcache) to speed up the process. The cache can be refreshed when the new datasets are added.

If database solution isn’t allowed, we can design the “finance data loader” to refresh the facilities data stored in memory (cache) when new datasets are updated, then update he “loan processing service” to pick up the new facilities.

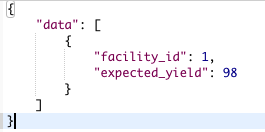
1. **Your solution most likely simulates the streaming process by directly calling a method in your code to process the loans inside of a for loop. What would a REST API look like for this same service? Stakeholders using the API will need, at a minimum, to be able to request a loan be assigned to a facility, and read the funding status of a loan, as well as query the capacities remaining in facilities.**

**Ans:** A prototype of REST API service is included in the solution to illustrate the design concept. You can run the API in docker test out the API. The API includes two endpoints:

* **POST**[**/api/v1/loans**](http://0.0.0.0:8080/api/v1/loans)
  + Pass the loan payload below to the end point, which will then call the “loan\_service.process\_load(loan)” to process the loan, and return the result for the called client to consume the result.



* **GET**[**/api/v1/facilities**](http://0.0.0.0:8080/api/v1/facilities)
  + Call this endpoint to get a list of facilities expected yields



1. **How might you improve your assignment algorithm if you were permitted to assign loans in batch rather than streaming? We are not looking for code here, but pseudo code or description of a revised algorithm appreciated.**

**Ans:** Currently the loan is processed one by one by searching for a facility that satisfies the criteria. If batch loans are also accepted, we can design the algorithms by querying all facilities that meet all the loan interest rate, amount, max\_default\_likelihood, banned\_state criteria. For example, we can find the maximum default\_likelihood value for the batch loans, then filter the facilities with covenants that have max\_default\_likelihood >= the maximum default\_likelihood value of the batch loans, then process the batch loans one by one with the filtered facilities.

def process\_loans(batch\_loans: list):  
  
 batch\_loans\_min\_amount = min(loans.amount)   
 batch\_loans\_states = combine all states from loans  
 batch\_loans\_max\_default\_likelihood = max(loans.default\_likehood)  
  
 filtered\_facilities = query all facilities.amount >= batch\_loan\_min\_amount   
 filtered\_covenants = query all covenants.banned\_states NOT IN batch\_loans\_states   
 AND covenants.max\_default\_likelihood >=

batch\_loans\_max\_default\_likelihood  
  
  
 for loan in batch\_loans:  
 process\_loan(loan)  
  
  
def process\_loan(loan):  
  
 Find facility in filtered\_facilities AND filtered\_covenants that meet filtered\_facilities AND filtered\_facilities

The other alternative for batch loans is we increase number of workers to process the batch loans in parallel, this will speed up processing the loans especially there are hundred thousand or even million of loans to process.

For example:

Worker1.process\_loans(batch\_loans1)

Worker2.process\_loans(batch\_loans2)

…

WorkerN:process\_loans(batch\_loansN)

This is even easier when we run the worker in Kubernete pods, each pod is a worker and we can easily scale the pods to process the batches, and scale down when job is finished.

1. **Discuss your solution’s runtime complexity.**

**Ans:** m is the number of loan to process, n is the number of possible facilities to search, the covenant lookup is constant, so the runtime complexity is O(m\*n) \* O(1)

for I = 1 to m (loop thru all loans)

for j = 1 to n (search facilities meeting the criteria)

look up covenant (constant)

# Analysis - Verifying Large Dataset Result

The calculated results from the large loan dataset have been verified thru the result verification in Excel. Please refer to Analysis\_large\_dataset.xslx.

The verification approach is

1. **Verifying Expected Yield:** From the loan assignment results, reverse engineering to manually calculate the expected yield for each loan, then sum the expected yield for each facility. Then verify the aggregated yield against the result from the yields.csv.
2. **Verifying State:** Check the loan state and verify the state is not in the banned state of the facility’s covenant
3. **Verifying default likelihood:** Check the loan default likelihood is smaller than the covenant’s max default likelihood.

