Justin Min
01/19/2026

Loan Portfolio Risk Manager Project

Project Overview

In this simulation, I assumed the role of a Credit Analyst at a finance firm. The objective was to perform an end-of-year risk review for a newly acquired portfolio of consumer loans originated in 2025. To perform this due diligence, I engineered an automated Extract-Transform-Load (ETL) pipeline using Python (Pandas) to standardize messy vendor data and utilized SQL to aggregate risk metrics. The final output was a standardized risk assessment report generated in Excel to support the Investment Team's review of the 2025 vintage.

Scenario

The Investment Team requires a performance audit of the 2025 consumer loans. While the dataset was complete, the "loan tape" provided by the seller contained formatting inconsistencies that made it inefficient:

- Inconsistent Date Formats: The "Date_Issued" column contained a mix of text-based entries (e.g., "Jan-2025") and standard numerical dates (e.g., "2025/02/01").
- Lack of Standardization: Management needed a unified view of 2025 to assess default rates, but the data was fragmented and individualized.

The goal was to automate the cleaning process and generate a final "2025 Performance Scorecard."

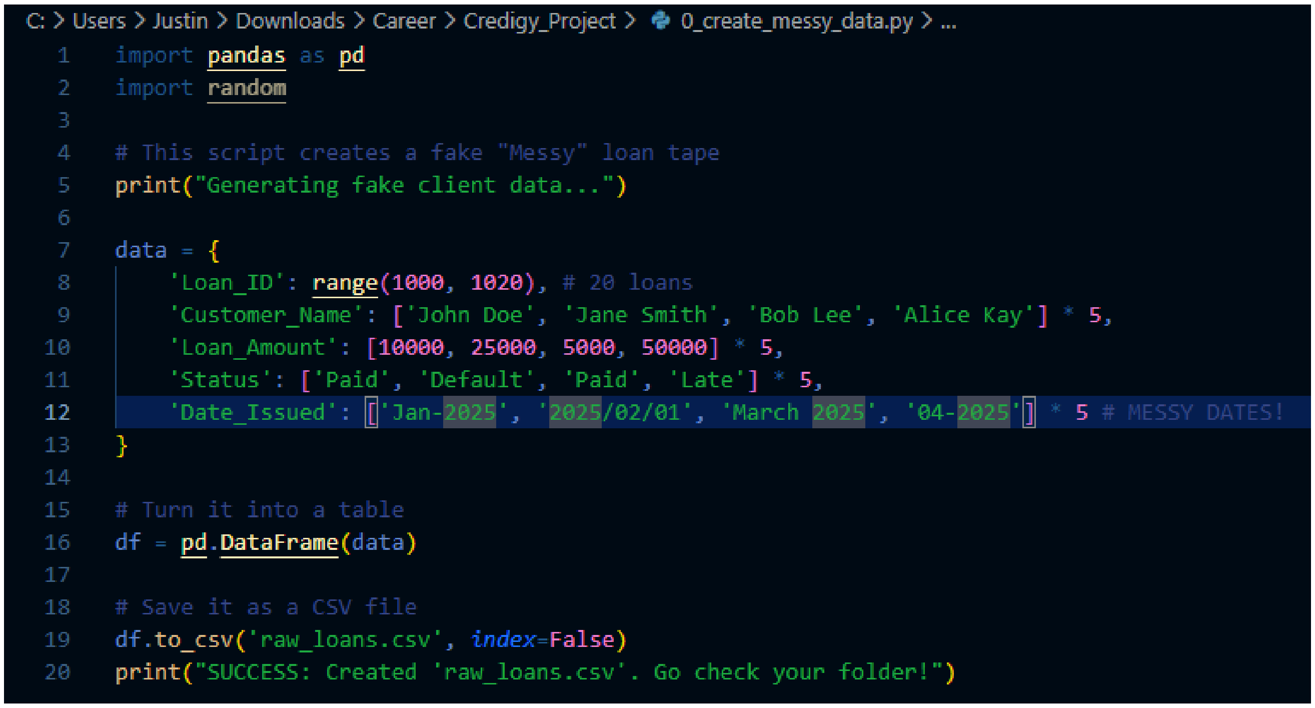
Task 1: The Setup

Before I began analyzing the loans in the messy data, I created a folder on my desktop to consolidate everything and keep it organized. Every company likes things done a certain way, and more likely than not the data will come formatted in a way that is not ideal. I started by transforming my local machine into a dedicated analytics workstation using Visual Studio Code (VS Code). I created a virtual Python environment and downloaded a standard "analyst toolkit": Pandas for heavy-lifting data manipulation and SQLite3 to mimic a database server.

The objective of this simulation is to organize and translate messy loan data into a consolidated final scorecard. To replicate receiving a disorganized file from a seller, I decided to create my own problems. I wrote a Python script (0_create_messy_data.py) specifically designed with formatting inconsistencies. I programmed the script to generate a synthetic loan tape where date formats were intentionally broken, randomly switching between text strings like "Jan-2025" and standard dates like "2025/02/01." This aspect was twofold; to make the data and to simulate a disorganized data predicament for my pipeline to solve.

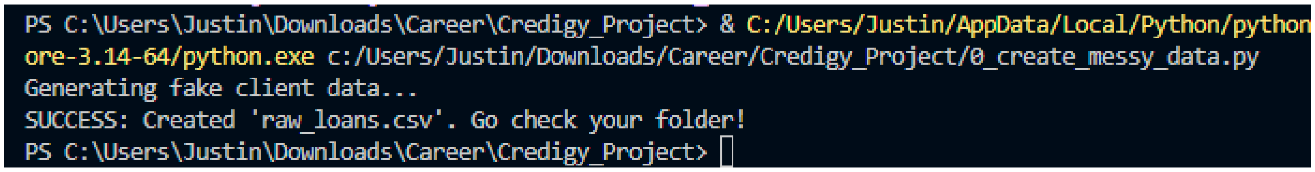
Action 1: Setting Up the Environment

1. I first created a folder labeled "**Credigy_Project**" and downloaded Visual Studio Code
2. In the VS Code terminal, I installed **Pandas**, an open-sourced Python library used for data analysis and manipulating tabular data (*Figure 1.1*)
3. I then created a new file named **0_create_messy_data_py** and entered a script that created a fake messy loan tape (*Figure 1.2*)
4. After running the script, a new file labeled **raw_laons.csv** was downloaded into the **Credigy_Project** folder with the messy data we will be using for this project (*Figure 1.3*)



```
C: > Users > Justin > Downloads > Career > Credigy_Project > 0_create_messy_data.py > ...
import pandas as pd
import random

# This script creates a fake "Messy" loan tape
print("Generating fake client data...")

data = {
    'Loan_ID': range(1000, 1020), # 20 loans
    'Customer_Name': ['John Doe', 'Jane Smith', 'Bob Lee', 'Alice Kay'] * 5,
    'Loan_Amount': [10000, 25000, 5000, 50000] * 5,
    'Status': ['Paid', 'Default', 'Paid', 'Late'] * 5,
    'Date_Issued': ['Jan-2025', '2025/02/01', 'March 2025', '04-2025'] * 5 # MESSY DATES!
}

# Turn it into a table
df = pd.DataFrame(data)

# Save it as a CSV file
df.to_csv('raw_loans.csv', index=False)
print("SUCCESS: Created 'raw_loans.csv'. Go check your folder!")
```

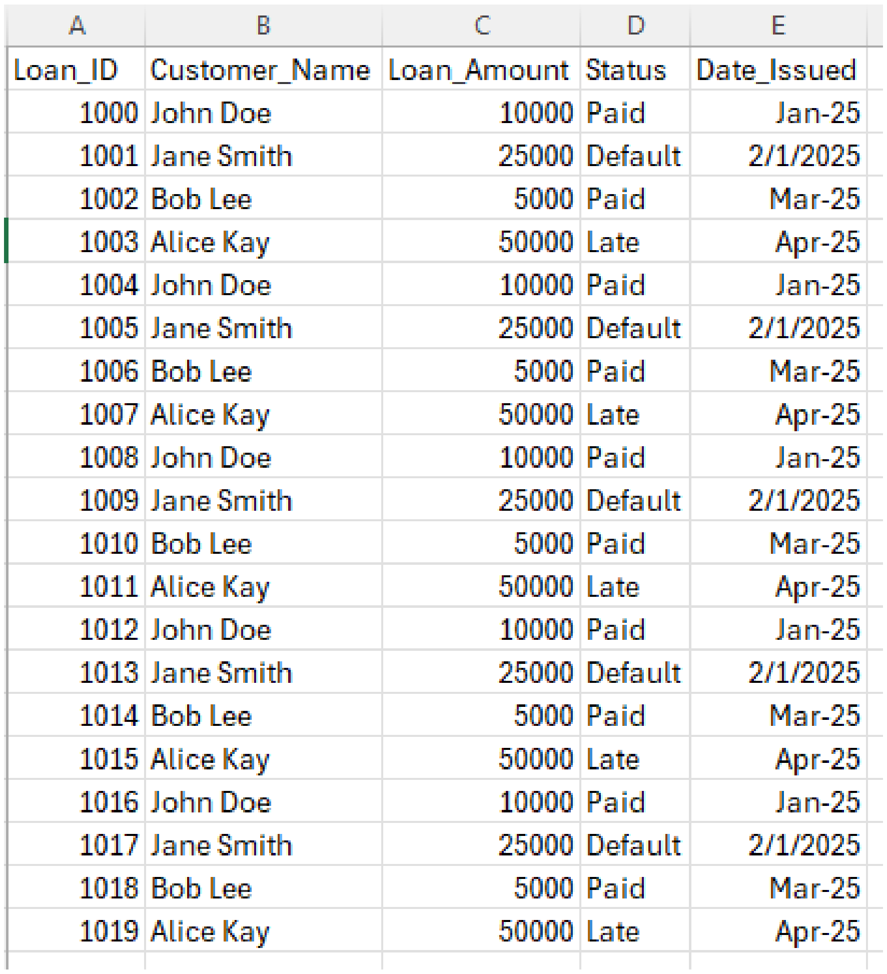
Figure 1.1



```
PS C:\Users\Justin\Downloads\Career\Credigy_Project> & C:/Users/Justin/AppData/Local/Python/python
ore-3.14-64/python.exe c:/Users/Justin/Downloads/Career/Credigy_Project/0_create_messy_data.py
Generating fake client data...
SUCCESS: Created 'raw_loans.csv'. Go check your folder!
PS C:\Users\Justin\Downloads\Career\Credigy_Project>
```

Figure 1.2

Scroll to Continue



A	B	C	D	E
Loan_ID	Customer_Name	Loan_Amount	Status	Date_Issued
1000	John Doe	10000	Paid	Jan-25
1001	Jane Smith	25000	Default	2/1/2025
1002	Bob Lee	5000	Paid	Mar-25
1003	Alice Kay	50000	Late	Apr-25
1004	John Doe	10000	Paid	Jan-25
1005	Jane Smith	25000	Default	2/1/2025
1006	Bob Lee	5000	Paid	Mar-25
1007	Alice Kay	50000	Late	Apr-25
1008	John Doe	10000	Paid	Jan-25
1009	Jane Smith	25000	Default	2/1/2025
1010	Bob Lee	5000	Paid	Mar-25
1011	Alice Kay	50000	Late	Apr-25
1012	John Doe	10000	Paid	Jan-25
1013	Jane Smith	25000	Default	2/1/2025
1014	Bob Lee	5000	Paid	Mar-25
1015	Alice Kay	50000	Late	Apr-25
1016	John Doe	10000	Paid	Jan-25
1017	Jane Smith	25000	Default	2/1/2025
1018	Bob Lee	5000	Paid	Mar-25
1019	Alice Kay	50000	Late	Apr-25

Figure 1.3

Task 2: Building the Pipeline

With my environment built, I assessed the vendor file (raw_loans.csv) I had created to confirm the code ran successfully and the (intentionally) messy data was in fact, messy. In financial due diligence, it is essential to audit before trusting any data.

The data contained conflicting standards. The critical Date_Issued column determining the vintage of a loan was effectively unusable. If I were to have ran a Pivot Table on this with some dates being text strings like "Jan-2025" and others "2025/02/01", Excel would have treated them as two completely different years and ruining the vintage curve. Alternatively, scouring through and replacing these inconsistent cells would be time consuming and run the risk of human error, making it a less-than-ideal solution. Therefore, I went with a programmatic solution that was repeatable and auditable to clean this data.

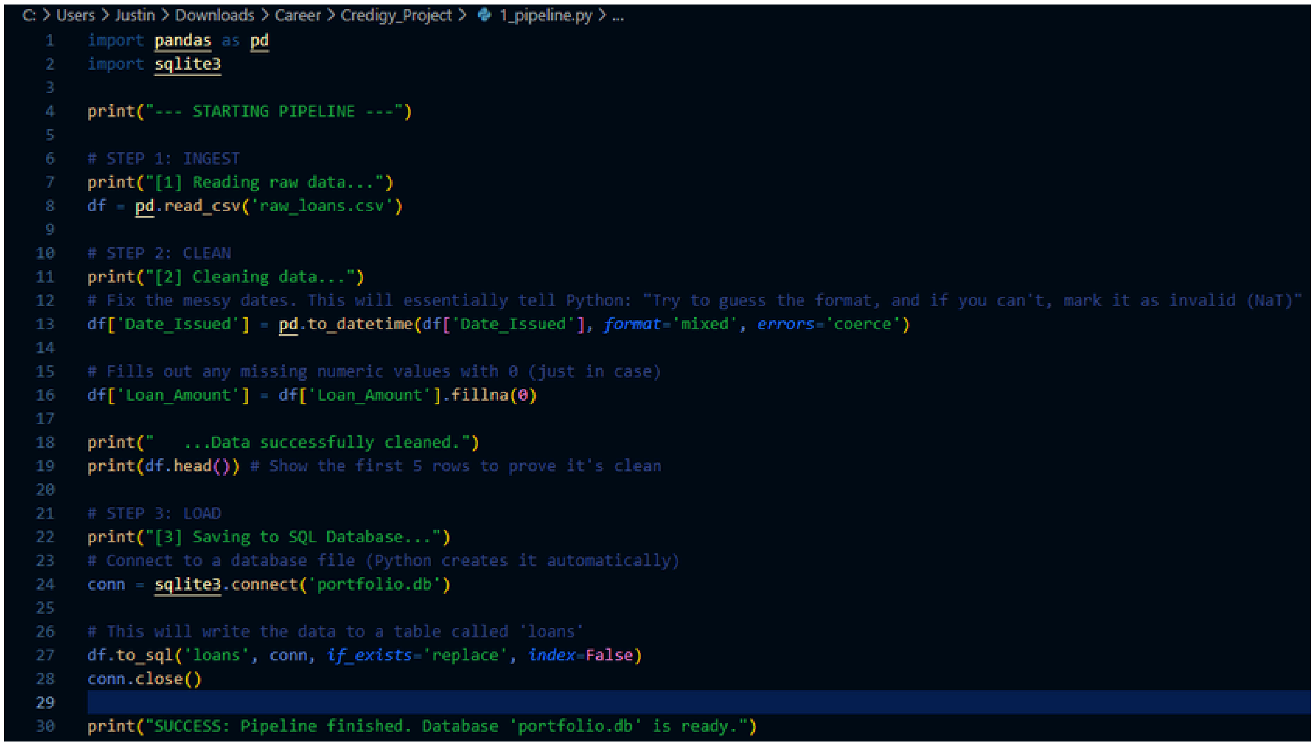
To solve the data quality issues while keeping the original file untouched, I engineered an automated ETL (Extract, Transform, Load) pipeline using Python. This not only provided a solution for this file, but could be reused and recycled for other vintage analyses as well.

I utilized the Pandas library to handle the heavy lifting. The core of my solution was the to_datetime function with the errors='coerce' parameter. This command forces Python to look at every single date entry, figure out what format it is in, and convert it into a standard YYYY-MM-DD ISO format.
The coerce command is to keep things running smooth; if it finds a piece of data that is not relevant (like "Not a Date"), it marks it as NaT (Not a Time) rather than crashing the entire script. This ensures the pipeline is robust enough to handle dirty real-world data. Finally, rather than dumping the clean data back into a CSV, I piped it directly into a local SQLite database (portfolio.db). This allows for scalable SQL querying later, simulating a true enterprise data warehouse architecture.

Action 2: Building the Pipeline

1. I created another new file titled **1_pipeline.py**
2. Next, I implemented a code that accomplished several tasks (*Figure 2.1)*:
    a. Ingested the raw data (**raw_loan.csv**)
    b. Standardized the messy data
    c. Saved the data into a **SQL database (portfolio.db)**
3. After running the **1_pipeline.py** script, the SQL database was created as **portfolio.db**

This process essentially read the original messy .csv file, took a snapshot for memory (RAM), and used digital “white-out”, then finally printing a brand new, clean copy and filed it in the cabinet **portfolio.db**.


C: > Users > Justin > Downloads > Career > Credigy_Project > 1_pipeline.py > ...

```python
import pandas as pd
import sqlite3

print("--- STARTING PIPELINE ---")

# STEP 1: INGEST
print("[1] Reading raw data...")
df = pd.read_csv('raw_loans.csv')

# STEP 2: CLEAN
print("[2] Cleaning data...")
# Fix the messy dates. This will essentially tell Python: "Try to guess the format, and if you can't, mark it as invalid (NaT)"
df['Date_Issued'] = pd.to_datetime(df['Date_Issued'], format='mixed', errors='coerce')

# Fills out any missing numeric values with 0 (just in case)
df['Loan_Amount'] = df['Loan_Amount'].fillna(0)

print("   ...Data successfully cleaned.")
print(df.head()) # Show the first 5 rows to prove it's clean

# STEP 3: LOAD
print("[3] Saving to SQL Database...")
# Connect to a database file (Python creates it automatically)
conn = sqlite3.connect('portfolio.db')

# This will write the data to a table called 'loans'
df.to_sql('loans', conn, if_exists='replace', index=False)
conn.close()

print("SUCCESS: Pipeline finished. Database 'portfolio.db' is ready.")
```

Figure 2.1

Task 3: The Vintage Analysis

Now with the cleaned data stored in my local **portfolio.db** database, I shifted from data engineering to analyst. I needed to answer the Investment Team's core question: *How is the 2025 Vintage performing?*

Instead of relying on Excel formulas, I wrote another Python script (**2_analysis.py**) to execute a SQL query against the database. This allows the analysis to be repeatable; if we receive new data next week, I can simply re-run the script without rebuilding a spreadsheet.

I designed a SQL query to aggregate the individual loans into a vintage-level summary. The query calculated three Key Performance Indicators (KPIs):

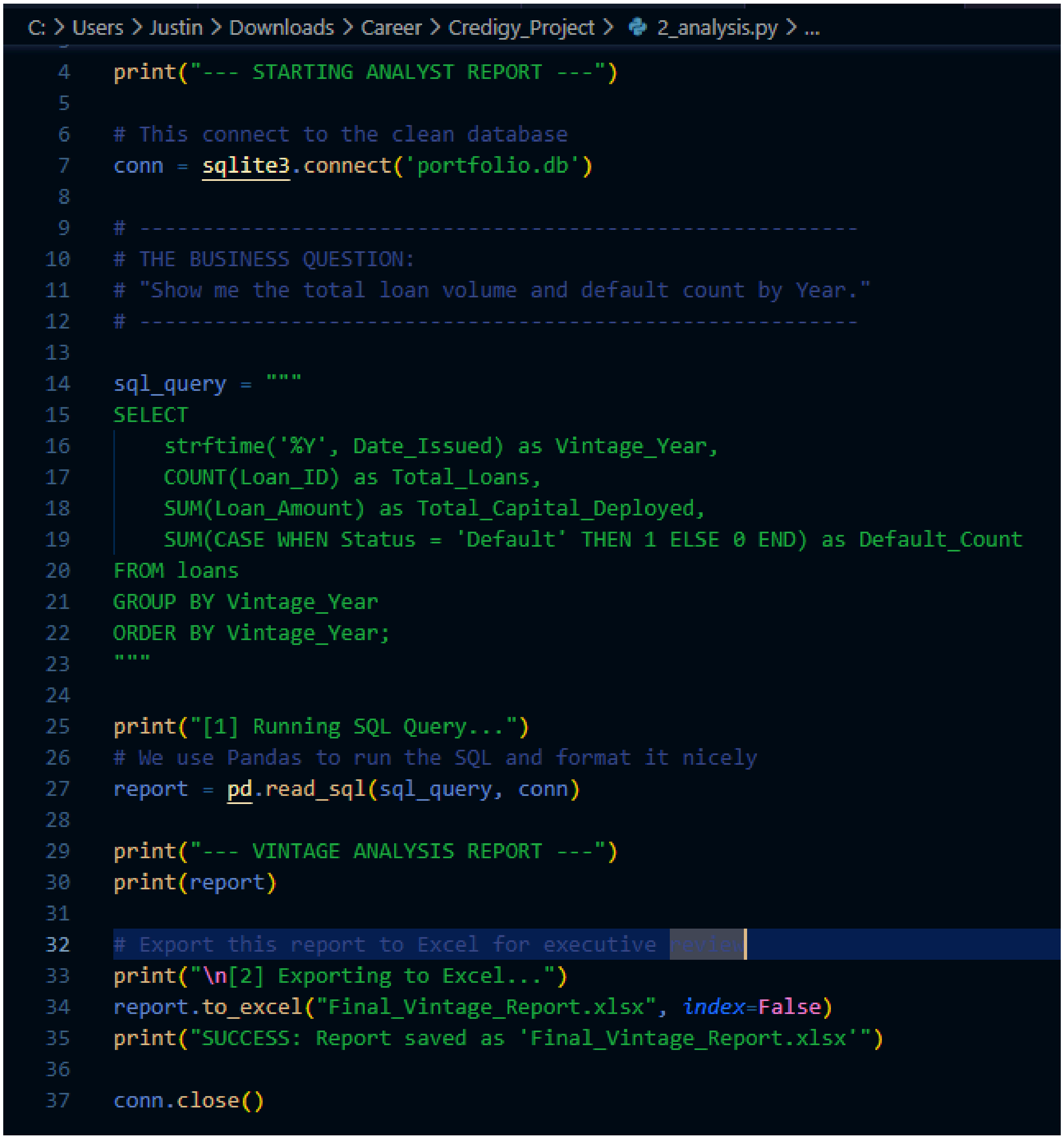
1. Total Loans: The volume of originations in 2025.
2. Capital Deployed: The sum of Loan_Amount to quantify financial exposure.
3. Default Count: A conditional count (CASE WHEN Status = 'Default') to identify immediate credit failures.

Action 3: The Analysis

- I created another file in VS Code named **2_analysis.py**
- Input a script that:
  - Connected to the portfolio.db (Figure 3.1)
  - Pulled the total loan volume and default count by Year (Figure 3.1)
  - Used Pandas to run the SQL and format the data (Figure 3.1)
  - After running the script, exported the final report as a new Excel file called **Final_Vintage_Report.xlsx**

The **Final_Vintage_Report.xlsx** shows the completed vintage analysis (*Figure 3.3*).

Scroll to Continue

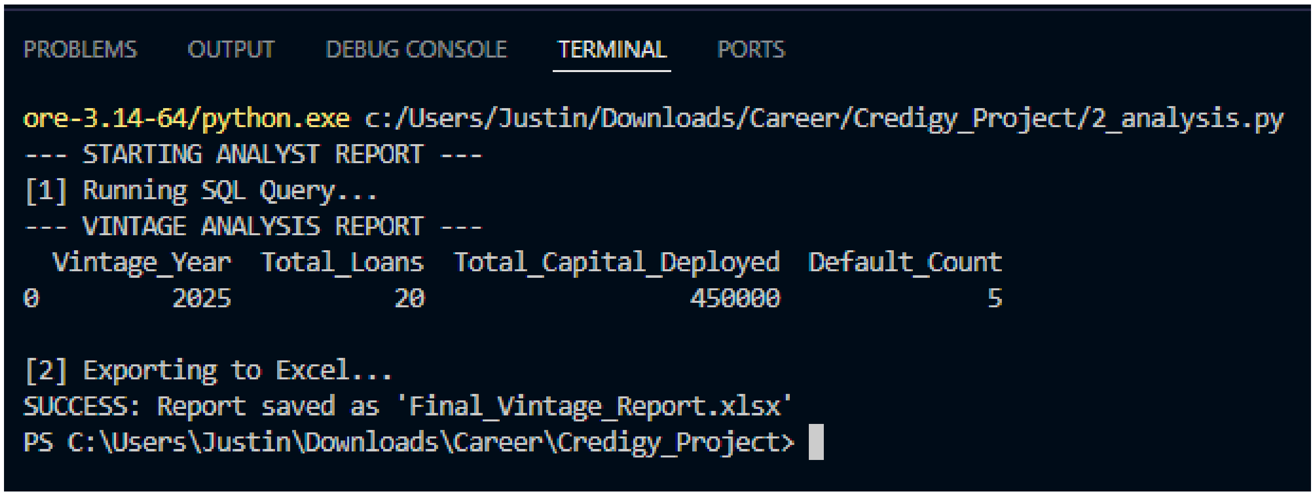


C: > Users > Justin > Downloads > Career > Credigy_Project > 2_analysis.py > ...

```python
print("--- STARTING ANALYST REPORT ---")

# This connect to the clean database
conn = sqlite3.connect('portfolio.db')

# ----------------------------------------------------------
# THE BUSINESS QUESTION:
# "Show me the total loan volume and default count by Year."
# ----------------------------------------------------------

sql_query = """
SELECT
    strftime('%Y', Date_Issued) as Vintage_Year,
    COUNT(Loan_ID) as Total_Loans,
    SUM(Loan_Amount) as Total_Capital_Deployed,
    SUM(CASE WHEN Status = 'Default' THEN 1 ELSE 0 END) as Default_Count
FROM loans
GROUP BY Vintage_Year
ORDER BY Vintage_Year;
"""

print("[1] Running SQL Query...")
# We use Pandas to run the SQL and format it nicely
report = pd.read_sql(sql_query, conn)

print("--- VINTAGE ANALYSIS REPORT ---")
print(report)

# Export this report to Excel for executive review
print("\n[2] Exporting to Excel...")
report.to_excel("Final_Vintage_Report.xlsx", index=False)
print("SUCCESS: Report saved as 'Final_Vintage_Report.xlsx'")

conn.close()
```

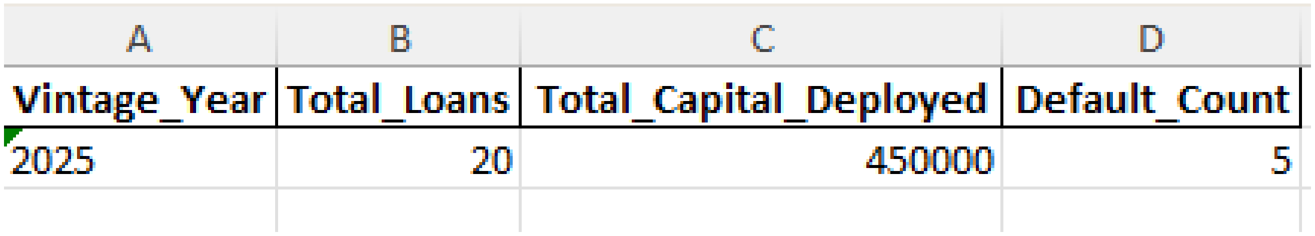
Figure 3.1

Scroll to Continue



```
ore-3.14-64/python.exe c:/Users/Justin/Downloads/Career/Credigy_Project/2_analysis.py
--- STARTING ANALYST REPORT ---
[1] Running SQL Query...
--- VINTAGE ANALYSIS REPORT ---
  Vintage_Year  Total_Loans  Total_Capital_Deployed  Default_Count
0         2025           20                  450000              5

[2] Exporting to Excel...
SUCCESS: Report saved as 'Final_Vintage_Report.xlsx'
PS C:\Users\Justin\Downloads\Career\Credigy_Project>
```

Figure 3.2



A	B	C	D
Vintage_Year	Total_Loans	Total_Capital_Deployed	Default_Count
2025	20	450000	5

Figure 3.3

Scroll to Continue

Conclusion

The immediate value of this project was the risk alert: identifying a critical 25% default rate in the 2025 vintage, which served as a data-driven basis for the Investment Team to pause further acquisitions. However, the operational victory was the creation of a repeatable data pipeline.

By shifting the due diligence process from manual Excel manipulation to an automated Python/SQL workflow, I eliminated the risk of human error in date standardization. This architecture transforms the "Vintage Analysis" from a one-off task that takes hours into a scalable process that takes seconds, regardless of whether the input file has 20 loans or 20,000.

Reflection

This end-to-end pipeline project was a pivotal exercise in understanding the "Analyst Stack." It bridged the gap between theoretical financial data concepts and the messy reality of financial due diligence.

Prior to this project, I viewed data analysis primarily through the lens of Excel spreadsheets. Building the ETL pipeline in Python shifted my perspective on Scalability. I learned that while Excel is excellent for viewing data, it is fragile for processing it. Writing the pandas.to_datetime logic taught me the power of reproducibility. I captured my logic in code so it can be audited and reused, whereas manually finding and replacing each cell in Excel leaves no trail.

The most challenging aspect was not the final analysis, but the initial ingestion. Dealing with the mixed date formats (strings vs. objects) highlighted the concept of “Garbage In, Garbage Out”. I learned that a robust risk model is useless if the underlying data pipeline breaks when it encounters a typo. This experience reinforced the importance of defensive programming (using errors='coerce'), a mindset I will carry into future Governance, Risk, and Compliance (GRC) roles.

Finally, persisting the data into a SQLite database demystified the concept of Enterprise Data Warehousing. I now understand why banks store data in "invisible" backend databases rather than user-facing files: it creates a single source of truth that separates the raw storage from the analytical reporting.

This project gave me the confidence to navigate the intersection of technical data engineering and strategic financial analysis.