```
# This Python 3 environment comes with many helpful analytics
libraries installed
# It is defined by the kaggle/python Docker image:
https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/"
directory
# For example, running this (by clicking run or pressing Shift+Enter)
will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/)
that gets preserved as output when you create a version using "Save &
Run All"
# You can also write temporary files to /kaggle/temp/, but they won't
be saved outside of the current session
```

```
/kaggle/input/data26/Data.csv
```

```
df=pd.read_csv("/kaggle/input/data26/Data.csv",sep=';')
df.head()
```

```
   Unnamed: 0  Origination Amount  31.05.2019  30.06.2019  31.07.2019
\
0  31.05.2019         10018746.17  1443069.08  3332200.33  1328138.75

1  30.06.2019         10868379.04        0.00  1392751.60  3011884.91

2  31.07.2019         10733932.61        0.00        0.00  1537650.24

3  31.08.2019         12558727.02        0.00        0.00        0.00

4  30.09.2019         14505071.44        0.00        0.00        0.00


   31.08.2019  30.09.2019  31.10.2019  30.11.2019  31.12.2019  ...  \
0   928085.74   736418.27   539403.31   427557.86   324459.32  ...
1  1237868.70   970929.28   892351.83   668767.02   505612.59  ...
2  2953335.55  1208316.08   879375.19   711016.84   658251.40  ...
3  1617681.94  4082016.00  1387474.94  1247623.59   886293.35  ...
4        0.00  1992242.84  3930445.60  1394620.78  1227905.58  ...

   31.03.2020  30.04.2020  31.05.2020  30.06.2020  31.07.2020
```

```
31.08.2020  \
0   116684.68    92699.67    63399.66    53265.12    37121.13
29787.10
1   255222.42   198833.96   161996.73   138461.91    92346.68
79641.30
2   302575.54   258652.52   191798.05   170027.54   127574.33
110301.21
3   417223.56   336686.08   253556.20   200066.59   151859.74
109973.00
4   628429.48   589692.85   457299.31   323764.87   288152.28
239872.99

   30.09.2020  31.10.2020  30.11.2020  31.12.2020
0    24524.90    18085.94    16581.01    11442.97
1    63457.44    52373.85    43374.70    37404.87
2    89766.69    64746.84    61408.92    50312.70
3    90228.14    70661.50    53102.83    47069.84
4   192246.98   171550.69   142575.97   116853.05

[5 rows x 22 columns]
```

```
data=df.to_numpy()
```

```
data=df.to_numpy()
for row in data[1:]:
    # Extract the second column value (assumed to be a divisor)
    divisor = row[1]

    # Sum the non-null values from the third column onward
    total_sum = sum([value for value in row[2:] if value is not None])

    # Divide the sum by the value in the second column (divisor)
    if divisor != 0:
        result = total_sum / divisor
    else:
        result = 0  # Handle division by zero if needed

    # Print or store the result
    print(f"Result for row starting with {row[0]}: {result:.2f}")
```

```
Result for row starting with 30.06.2019: 0.97
Result for row starting with 31.07.2019: 0.96
Result for row starting with 31.08.2019: 0.97
Result for row starting with 30.09.2019: 0.93
Result for row starting with 31.10.2019: 0.93
Result for row starting with 30.11.2019: 0.92
Result for row starting with 31.12.2019: 0.88
Result for row starting with 31.01.2020: 0.85
Result for row starting with 29.02.2020: 0.87
Result for row starting with 31.03.2020: 0.78
```

```
Result for row starting with 30.04.2020: 0.76
Result for row starting with 31.05.2020: 0.78
Result for row starting with 30.06.2020: 0.76
Result for row starting with 31.07.2020: 0.65
Result for row starting with 31.08.2020: 0.61
Result for row starting with 30.09.2020: 0.52
Result for row starting with 31.10.2020: 0.50
Result for row starting with 30.11.2020: 0.43
Result for row starting with 31.12.2020: 0.14
```

```
import pandas as pd

# Convert DataFrame to numpy array
data = df.to_numpy()

# Initialize an empty list to store results and track issues
results = []
issue_rows = []

# Loop through each row, starting from the second row
for row in data[1:]:
    # Check that there are enough columns in the row
    if len(row) < 2:
        issue_rows.append({"row": row, "issue": "Insufficient
columns"})
        continue

    # Extract the first column (ID) and second column (divisor)
    row_id = row[0]
    divisor = row[1]

    # Check if the divisor is zero or missing
    if divisor == 0 or divisor is None:
        issue_rows.append({"row": row_id, "issue": "Divisor is zero or
missing"})
        result = None
    else:
        # Calculate the sum of non-null values from the third column
onward
        total_sum = sum([value for value in row[2:] if value is not
None])
        result = total_sum / divisor  # Calculate result

    # Append the result along with row ID for tracking
    results.append({"row_id": row_id, "result": result})

# Convert results to a DataFrame for easier manipulation
results_df = pd.DataFrame(results)

# Add results back to the original DataFrame as a new column, if
```

```
needed
df["Result"] = results_df["result"]

# Display issues, if any
if issue_rows:
    print("Rows with issues:", issue_rows)



df.head()
```

```
   Unnamed: 0  Origination Amount  31.05.2019  30.06.2019  31.07.2019
\
0  31.05.2019         10018746.17  1443069.08  3332200.33  1328138.75

1  30.06.2019         10868379.04        0.00  1392751.60  3011884.91

2  31.07.2019         10733932.61        0.00        0.00  1537650.24

3  31.08.2019         12558727.02        0.00        0.00        0.00

4  30.09.2019         14505071.44        0.00        0.00        0.00


   31.08.2019  30.09.2019  31.10.2019  30.11.2019  31.12.2019  ...  \
0   928085.74   736418.27   539403.31   427557.86   324459.32  ...
1  1237868.70   970929.28   892351.83   668767.02   505612.59  ...
2  2953335.55  1208316.08   879375.19   711016.84   658251.40  ...
3  1617681.94  4082016.00  1387474.94  1247623.59   886293.35  ...
4        0.00  1992242.84  3930445.60  1394620.78  1227905.58  ...

   30.04.2020  31.05.2020  30.06.2020  31.07.2020  31.08.2020
30.09.2020  \
0    92699.67    63399.66    53265.12    37121.13    29787.10
24524.90
1   198833.96   161996.73   138461.91    92346.68    79641.30
63457.44
2   258652.52   191798.05   170027.54   127574.33   110301.21
89766.69
3   336686.08   253556.20   200066.59   151859.74   109973.00
90228.14
4   589692.85   457299.31   323764.87   288152.28   239872.99
192246.98

   31.10.2020  30.11.2020  31.12.2020    Result
0    18085.94    16581.01    11442.97  0.970903
1    52373.85    43374.70    37404.87  0.959725
2    64746.84    61408.92    50312.70  0.972781
3    70661.50    53102.83    47069.84  0.926431
4   171550.69   142575.97   116853.05  0.934743
```

```
[5 rows x 23 columns]

# Convert the DataFrame to a NumPy array for easier indexing
data = df.to_numpy()

# Step 1: Compute Historical Repayment Percentages
historical_percentages = []
for row in data:
    origination_amount = row[1]  # The second column is the
origination amount
    repayment_percentages = [
        repayment / origination_amount for repayment in row[2:] if
origination_amount != 0
    ]
    historical_percentages.append(repayment_percentages)

historical_percentages = np.array(historical_percentages)



# Step 2: Compute Expected Repayment Percentages
expected_percentages = []
for row_idx, row in enumerate(data):
    origination_amount = row[1]
    monthly_expected = []

    # For the first two months, use existing data if available; for
December 2020, adjust p2
    p1 = row[2] / origination_amount if origination_amount != 0 else 0
    p2 = row[3] / origination_amount if origination_amount != 0 else 2
* p1 if row_idx == len(data) - 1 else 0

    monthly_expected.append(p1)
    monthly_expected.append(p2)

    # Calculate for months 3 to 30 using provided formula
    for i in range(3, 30):
        previous_sum = sum(monthly_expected[:i - 1])
        p_i = max(p2 * np.log(1 + (1 / (30 - (i - 1)))) * (1 -
previous_sum), 0)
        monthly_expected.append(p_i)

    expected_percentages.append(monthly_expected)

expected_percentages = np.array(expected_percentages)

# Step 3: Forecasted Cash Flows
forecasted_cash_flows = data[:, 1].reshape(-1, 1) *
expected_percentages
```

```python
annual_discount_rate = 0.025
monthly_discount_rate = annual_discount_rate / 12

# Calculate present value of forecasted cash flows for each vintage
present_value_forecasted_cash_flows = []
for t in range(forecasted_cash_flows.shape[1]):
    discount_factor = 1 / ((1 + monthly_discount_rate) ** (t + 1))
    pv_for_month = forecasted_cash_flows[:, t] * discount_factor
    present_value_forecasted_cash_flows.append(pv_for_month)

# Sum across all vintages and months to get the total portfolio value
portfolio_value = np.sum(present_value_forecasted_cash_flows)




# Step 5: Compare to Client's Estimate
client_estimate = 84993122.67
absolute_difference = abs(portfolio_value - client_estimate)
relative_difference = (absolute_difference / client_estimate) * 100

# Check if within acceptable threshold
acceptable_threshold = 500000
is_within_threshold = absolute_difference <= acceptable_threshold

print(f"Computed Portfolio Value: CHF {portfolio_value:,.2f}")
print(f"Client's Estimate: CHF {client_estimate:,.2f}")
print(f"Absolute Difference: CHF {absolute_difference:,.2f}")
print(f"Relative Difference: {relative_difference:.2f}%")
print(f"Difference within acceptable threshold:
{is_within_threshold}")
```

```
Computed Portfolio Value: CHF 11,824,935.18
Client's Estimate: CHF 84,993,122.67
Absolute Difference: CHF 73,168,187.49
Relative Difference: 86.09%
Difference within acceptable threshold: False
```