```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
```

```
! gdown '1VgAJLIgLGLA5RP-dItewchpk7Oa0diuu'
```

```
Downloading...
From: https://drive.google.com/uc?id=1VgAJLIgLGLA5RP-dItewchpk7Oa0diuu
To: /content/Credit_score.csv
100% 27.4M/27.4M [00:00<00:00, 158MB/s]
```

```
pd.set_option('display.max_columns',None)
```

```
df = pd.read_csv('/content/Credit_score.csv')
df.head(4)
```

```
<ipython-input-288-97e8b992493f>:1: DtypeWarning: Columns (26) have mixed types. Specify dtype option on import or set low_memory=Fa
  df = pd.read_csv('/content/Credit_score.csv')
```

	ID	Customer_ID	Month	Name	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credi
0	0x1602	CUS_0xd40	January	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	1824.843333	3	
1	0x1603	CUS_0xd40	February	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	
2	0x1604	CUS_0xd40	March	Aaron Maashoh	-500	821-00-0265	Scientist	19114.12	NaN	3	
3	0x1605	CUS_0xd40	April	Aaron Maashoh	23	821-00-0265	Scientist	19114.12	NaN	3	

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 27 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   ID                        100000 non-null  object
 1   Customer_ID               100000 non-null  object
 2   Month                     100000 non-null  object
 3   Name                      90015 non-null   object
 4   Age                       100000 non-null  object
 5   SSN                       100000 non-null  object
 6   Occupation                100000 non-null  object
 7   Annual_Income             100000 non-null  object
 8   Monthly_Inhand_Salary     84998 non-null   float64
 9   Num_Bank_Accounts         100000 non-null  int64
 10  Num_Credit_Card           100000 non-null  int64
 11  Interest_Rate             100000 non-null  int64
 12  Num_of_Loan               100000 non-null  object
 13  Type_of_Loan              88592 non-null   object
 14  Delay_from_due_date       100000 non-null  int64
 15  Num_of_Delayed_Payment    92998 non-null   object
 16  Changed_Credit_Limit      100000 non-null  object
 17  Num_Credit_Inquiries      98035 non-null   float64
 18  Credit_Mix                100000 non-null  object
 19  Outstanding_Debt          100000 non-null  object
 20  Credit_Utilization_Ratio  100000 non-null  float64
 21  Credit_History_Age        90970 non-null   object
 22  Payment_of_Min_Amount     100000 non-null  object
 23  Total_EMI_per_month       100000 non-null  float64
 24  Amount_invested_monthly   95521 non-null   object
 25  Payment_Behaviour         100000 non-null  object
 26  Monthly_Balance           98800 non-null   object
dtypes: float64(4), int64(4), object(19)
memory usage: 20.6+ MB
```

Missing Value Treatment

```
# To view missing data percentages
missing_percentage = (df.isna().sum() / len(df)) * 100
print(missing_percentage)
```

```
ID                          0.000
Customer_ID                 0.000
Month                       0.000
Name                        9.985
Age                         0.000
SSN                         0.000
Occupation                  0.000
Annual_Income               0.000
Monthly_Inhand_Salary      15.002
Num_Bank_Accounts           0.000
Num_Credit_Card             0.000
Interest_Rate               0.000
Num_of_Loan                 0.000
Type_of_Loan               11.408
Delay_from_due_date         0.000
Num_of_Delayed_Payment      7.002
Changed_Credit_Limit        0.000
Num_Credit_Inquiries        1.965
Credit_Mix                  0.000
Outstanding_Debt            0.000
Credit_Utilization_Ratio    0.000
Credit_History_Age          9.030
Payment_of_Min_Amount       0.000
Total_EMI_per_month         0.000
Amount_invested_monthly     4.479
Payment_Behaviour           0.000
Monthly_Balance             1.200
dtype: float64
```

```
# Droping Name column as it has 10% missing value and it is not used for analysis, instead we could use customer ID.
df.drop('Name',axis = 1,inplace = True)
```

```
# Checking if the data is normal or not
df['Monthly_Inhand_Salary'].skew()
```

```
1.1272722698181399
```

Insight: The column 'Monthly_Inhand_Salary' is right skewed hence we will impute using median.

```
# Impute missing values in 'Monthly_Inhand_Salary' column with the mean
df['Monthly_Inhand_Salary'].fillna(df['Monthly_Inhand_Salary'].median(), inplace=True)
```

```
<ipython-input-293-f769cc0c0f7d>:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Monthly_Inhand_Salary'].fillna(df['Monthly_Inhand_Salary'].median(), inplace=True)
```

```
# Impute the missing value in Type of Loan
df['Type_of_Loan'].fillna('Missing', inplace=True)
```

```
<ipython-input-294-1f240f133ece>:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Type_of_Loan'].fillna('Missing', inplace=True)
```

```
# Changing the datatype of the following columns
df['Amount_invested_monthly'] = pd.to_numeric(df['Amount_invested_monthly'], errors='coerce')
df['Monthly_Balance'] = pd.to_numeric(df['Monthly_Balance'], errors='coerce')
```

```
# Impute the missing values
df['Amount_invested_monthly'].fillna(df['Amount_invested_monthly'].median(),inplace = True)
df['Monthly_Balance'].fillna(df['Monthly_Balance'].median(),inplace = True)
```

```
<ipython-input-296-500c2e4e00ee>:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Amount_invested_monthly'].fillna(df['Amount_invested_monthly'].median(),inplace = True)
<ipython-input-296-500c2e4e00ee>:3: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Monthly_Balance'].fillna(df['Monthly_Balance'].median(),inplace = True)
```

```
# defining a function to convert the credit_history_Age text to int no. of month
def convert_credit_history_Age(age):
  if pd.isna(age):
    return None
  year,month = map(int,re.findall(r'\d+',age))
  return year*12 + month
```

```
# Creating a column by apply the created fuction
df['Credit_History_Months'] = df['Credit_History_Age'].apply(convert_credit_history_Age)
```

```
# impute the missing values
df['Credit_History_Months'].fillna(df['Credit_History_Months'].median(),inplace = True)
```

```
<ipython-input-299-d4b4b40b7a26>:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Credit_History_Months'].fillna(df['Credit_History_Months'].median(),inplace = True)
```

```
# checking the skewness of the data
df['Num_Credit_Inquiries'].skew()
```

```
9.78624574664581
```

```
# Converting the column values in numeric values
df['Num_of_Delayed_Payment'] = pd.to_numeric(df['Num_of_Delayed_Payment'], errors='coerce')
```

```
# Filling the missing values
df['Num_Credit_Inquiries'].fillna(df['Num_Credit_Inquiries'].median(),inplace = True)
df['Num_of_Delayed_Payment'].fillna(df['Num_of_Delayed_Payment'].median(),inplace = True)
```

```
<ipython-input-302-22d708ff7ff1>:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Num_Credit_Inquiries'].fillna(df['Num_Credit_Inquiries'].median(),inplace = True)
<ipython-input-302-22d708ff7ff1>:3: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained as
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col

  df['Num_of_Delayed_Payment'].fillna(df['Num_of_Delayed_Payment'].median(),inplace = True)
```

```
# Dropping the Credit History Age column
df.drop('Credit_History_Age',axis = 1,inplace = True)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 26 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   ID                        100000 non-null  object
 1   Customer_ID               100000 non-null  object
 2   Month                     100000 non-null  object
 3   Age                       100000 non-null  object
```

```
 4   SSN                       100000 non-null  object
 5   Occupation                100000 non-null  object
 6   Annual_Income             100000 non-null  object
 7   Monthly_Inhand_Salary     100000 non-null  float64
 8   Num_Bank_Accounts         100000 non-null  int64
 9   Num_Credit_Card           100000 non-null  int64
 10  Interest_Rate             100000 non-null  int64
 11  Num_of_Loan               100000 non-null  object
 12  Type_of_Loan              100000 non-null  object
 13  Delay_from_due_date       100000 non-null  int64
 14  Num_of_Delayed_Payment    100000 non-null  float64
 15  Changed_Credit_Limit      100000 non-null  object
 16  Num_Credit_Inquiries      100000 non-null  float64
 17  Credit_Mix                100000 non-null  object
 18  Outstanding_Debt          100000 non-null  object
 19  Credit_Utilization_Ratio  100000 non-null  float64
 20  Payment_of_Min_Amount     100000 non-null  object
 21  Total_EMI_per_month       100000 non-null  float64
 22  Amount_invested_monthly   100000 non-null  float64
 23  Payment_Behaviour         100000 non-null  object
 24  Monthly_Balance           100000 non-null  float64
 25  Credit_History_Months     100000 non-null  float64
dtypes: float64(8), int64(4), object(14)
memory usage: 19.8+ MB
```

Basic EDA

```
# unique number of customers
df['Customer_ID'].nunique()
```

```
12500
```

Insight: There are 12500 number of unique customers.

```
# Top 10 customers by count of records
grouped_customer = df.groupby('Customer_ID').size().reset_index(name='Count')
top_10_customers = grouped_customer.sort_values(by='Count', ascending=False).head(10)
top_10_customers
```

	Customer_ID	Count
0	CUS_0x1000	8
8350	CUS_0x8cfe	8
8328	CUS_0x8cbe	8
8329	CUS_0x8cc1	8
8330	CUS_0x8cc5	8
8331	CUS_0x8cc8	8
8332	CUS_0x8cc9	8
8333	CUS_0x8ccd	8
8334	CUS_0x8cce	8
8335	CUS_0x8cd	8

Next steps: Generate code with top_10_customers | View recommended plots | New interactive sheet

```
# Grouped customers by count of records
grouped_customer = df.groupby('Customer_ID').size().reset_index(name='Count')
grouped_customer['Count'].value_counts()
```

Count	count
8	12500

dtype: int64

Insight: Every unique customer has 8 records in the data.

```
grouped_month = df.groupby('Month')['ID'].count().reset_index()
grouped_month
```

	Month	ID
0	April	12500
1	August	12500
2	February	12500
3	January	12500
4	July	12500
5	June	12500
6	March	12500
7	May	12500

Next steps: Generate code with grouped_month | View recommended plots | New interactive sheet

Insight: There are equal 12500 number of records in the data spresd through the 8 above mentioned months.

```
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')
```

```
grouped_age = df.groupby('Age')['ID'].count().reset_index()
grouped_age
```

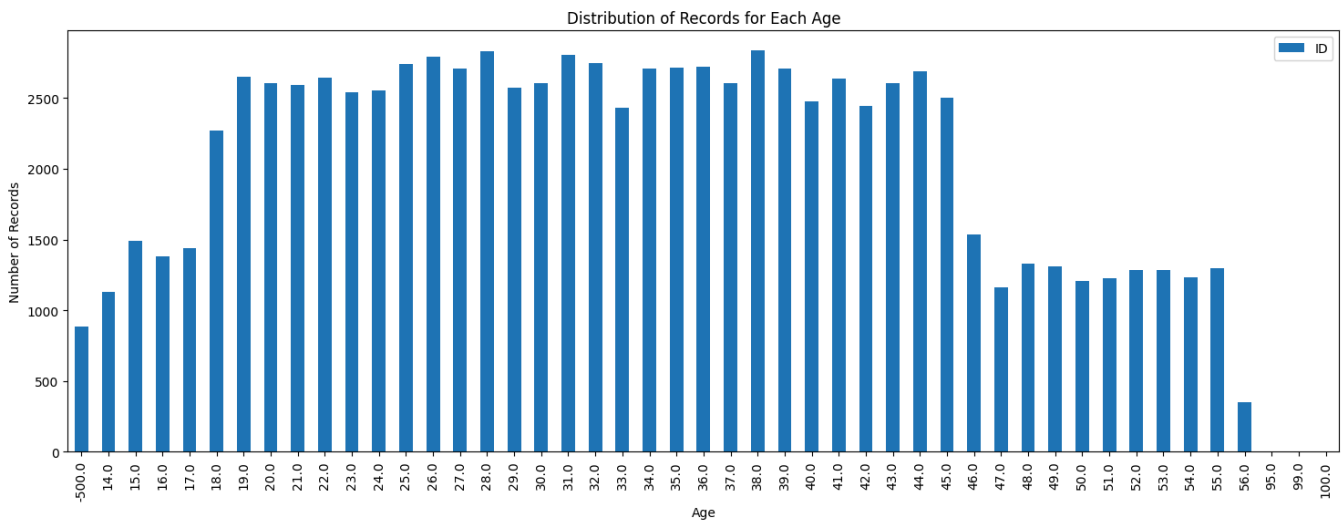
	Age	ID
0	-500.0	886
1	14.0	1129
2	15.0	1488
3	16.0	1378
4	17.0	1438
...	...	...
1656	8674.0	1
1657	8678.0	1
1658	8682.0	1
1659	8697.0	1
1660	8698.0	1

1661 rows × 2 columns

Next steps: Generate code with grouped_age | View recommended plots | New interactive sheet

```
# Distribution of records for each age under 100 Age.
age_dist = grouped_age[grouped_age['Age'] <= 100]
```

```
age_dist.plot(kind = 'bar',x = 'Age',y = 'ID',figsize = (18,6))
plt.xlabel('Age')
plt.ylabel('Number of Records')
plt.title('Distribution of Records for Each Age')
plt.show()
```



Insight:

1. The most number of records are for the age 18 to 45.
2. The starting entry for -500 could be place holder used by them for missing value or just anamoly in the dataset.

```
# Distribution of records for each age above 100 Age.
grouped_age[grouped_age['Age'] > 100].sort_values(by='ID', ascending=False)
```

	Age	ID
854	4494.0	3
594	2980.0	3
1047	5579.0	3
751	3920.0	3
783	4083.0	3
...	...	...
619	3155.0	1
617	3145.0	1
616	3132.0	1
615	3125.0	1
1660	8698.0	1

1614 rows × 2 columns

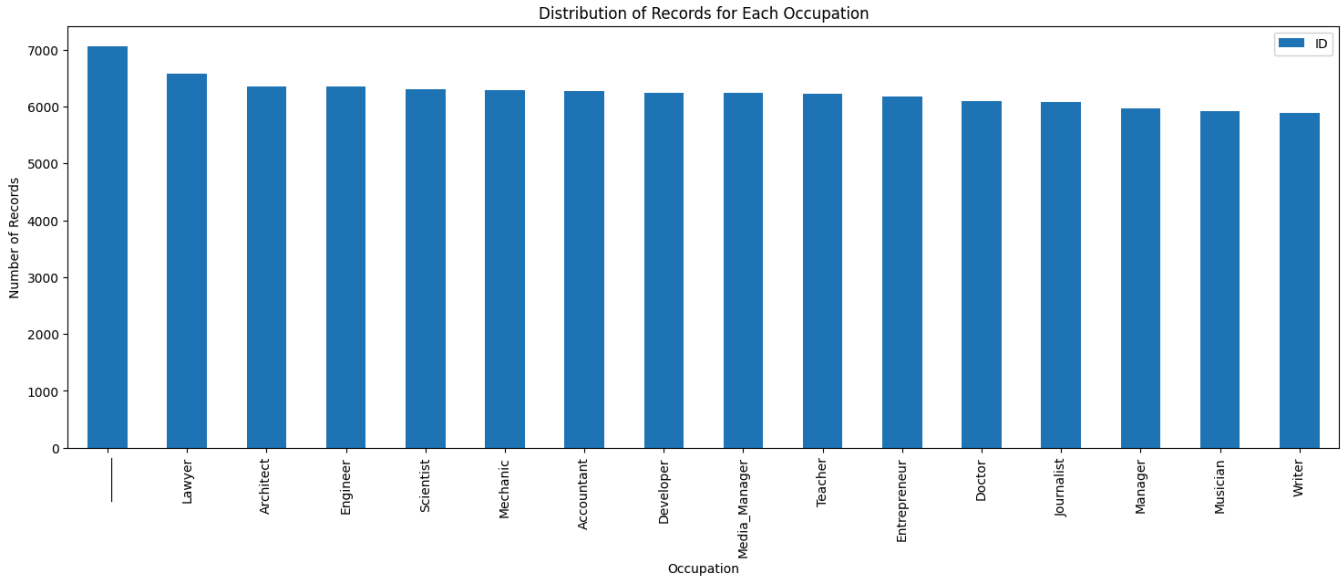
Insight: These are the anamolies of the dataset.

```
# Grouping data based on Occupation
grouped_occupation = df.groupby('Occupation')['ID'].count().reset_index().sort_values('ID', ascending=False)
grouped_occupation
```

	Occupation	ID
15	_____	7062
7	Lawyer	6575
1	Architect	6355
4	Engineer	6350
12	Scientist	6299
9	Mechanic	6291
0	Accountant	6271
2	Developer	6235
10	Media_Manager	6232
13	Teacher	6215
5	Entrepreneur	6174
3	Doctor	6087
6	Journalist	6085
8	Manager	5973
11	Musician	5911
14	Writer	5885

Next steps: Generate code with grouped_occupation View recommended plots New interactive sheet

```
# Plotting the data
grouped_occupation.plot(kind = 'bar',x = 'Occupation',y = 'ID',figsize = (18,6))
plt.xlabel('Occupation')
plt.ylabel('Number of Records')
plt.title('Distribution of Records for Each Occupation')
plt.show()
```



Insight:

1. Almost all the records are distributed similarly amongst the each occupation, meaning people from all the occupation are interested.
2. The first ____ could be a place holder or anamoly in the dataset.

```
# Grouping data based on Age and Occupation.
grouped_age_occupation = df.groupby(['Age', 'Occupation'])['ID'].count().reset_index().sort_values('ID', ascending=False)
```

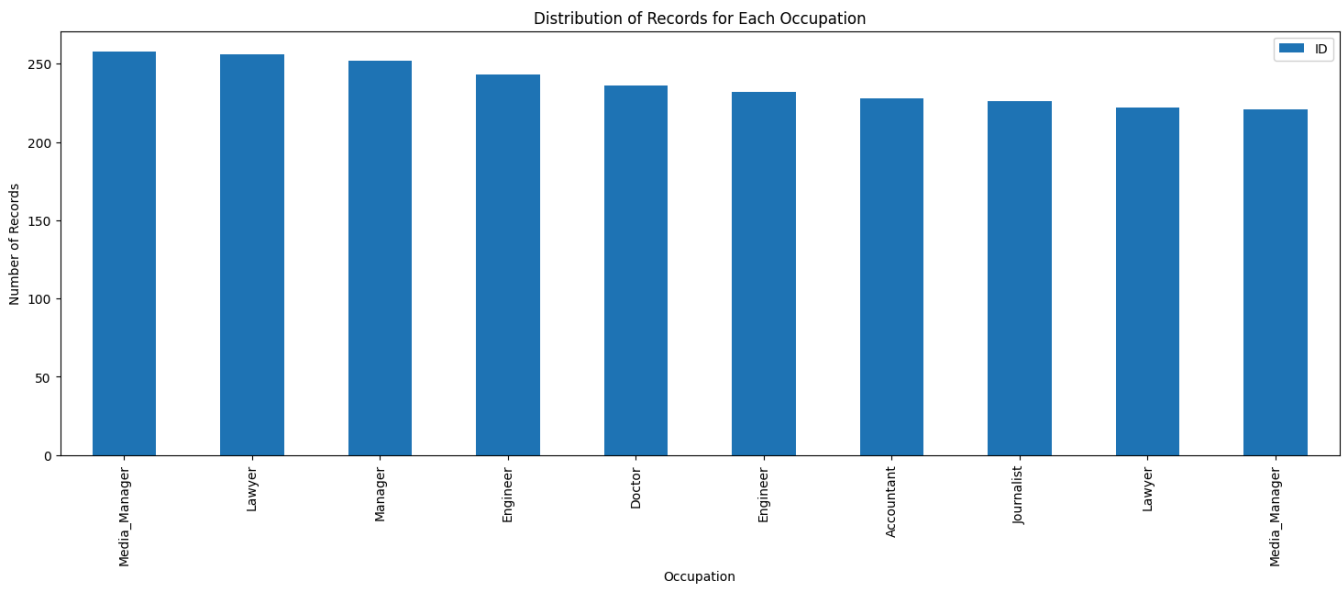
```
grouped_age_occupation
```

	Age	Occupation	ID
394	37.0	Media_Manager	258
455	41.0	Lawyer	256
296	31.0	Manager	252
116	20.0	Engineer	243
499	44.0	Doctor	236
...	...	...	...
1303	2875.0	Entrepreneur	1
1302	2871.0	Engineer	1
1301	2864.0	Developer	1
1300	2858.0	Lawyer	1
2495	8698.0	Entrepreneur	1

2496 rows × 3 columns

Next steps: Generate code with grouped_age_occupation | View recommended plots | New interactive sheet

```
# Plotiing the data
grouped_age_occupation.head(10).plot(kind = 'bar',x = 'Occupation',y = 'ID',figsize = (18,6))
plt.xlabel('Occupation')
plt.ylabel('Number of Records')
plt.title('Distribution of Records for Each Occupation')
plt.show()
```



Insight: The max records are for 37 year old Media_Managers about 258 times.

```
# Grouping on occupation and monthly salary
grouped_monthly = df.groupby('Occupation')['Monthly_Inhand_Salary'].agg(lambda x: x.mean()).reset_index().sort_values(by='Monthly_Inhanc
grouped_monthly
```

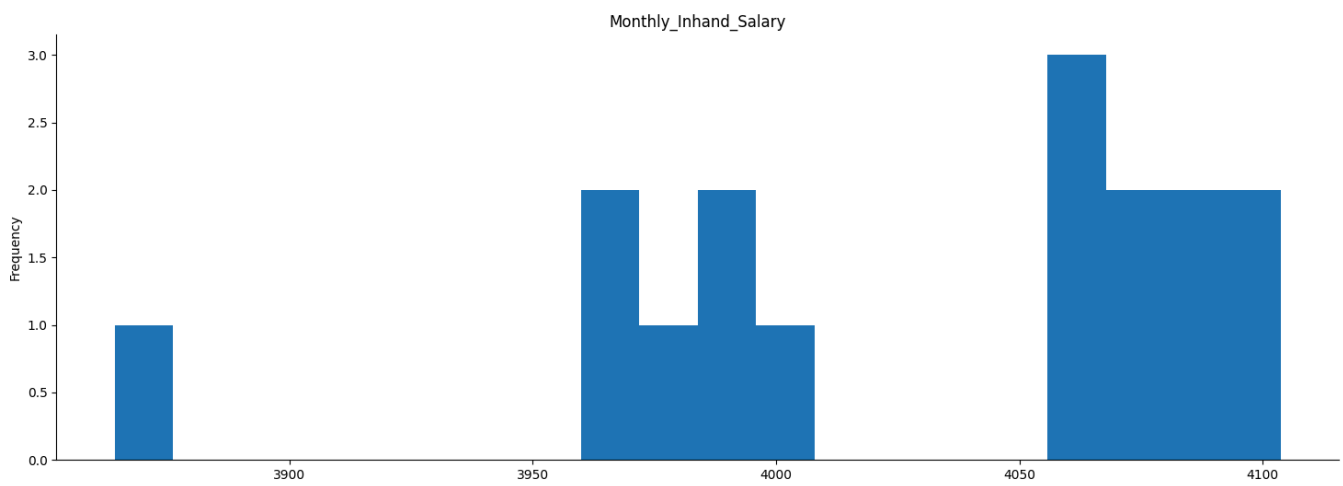
	Occupation	Monthly_Inhand_Salary
1	Architect	4103.738453
11	Musician	4103.481629
8	Manager	4090.041822
12	Scientist	4080.479366
4	Engineer	4074.729287
14	Writer	4073.920266
10	Media_Manager	4064.045506
0	Accountant	4060.158229
5	Entrepreneur	4058.989539
15	_____	4006.025818
9	Mechanic	3993.336606
2	Developer	3987.628784
7	Lawyer	3980.232725
3	Doctor	3967.510210
13	Teacher	3963.627925
6	Journalist	3864.083734

Next steps: Generate code with grouped_monthly View recommended plots New interactive sheet

Insight:

1. The Architect draws max avg monthly salary (4103.738453).
2. The Journalist draws min avg monthly salary (3864.083734).

```
# Plotting the graph
grouped_monthly['Monthly_Inhand_Salary'].plot(kind='hist', bins=20, title='Monthly_Inhand_Salary',figsize = (18,6))
plt.gca().spines[['top', 'right',]].set_visible(False)
```



Insight: Max monthly salary is between 4050 and 4100.

```
grouped_monthly.describe()
```

	Monthly_Inhand_Salary
count	16.000000
mean	4029.501869
std	66.259256
min	3864.083734
25%	3985.779769
50%	4059.573884
75%	4076.166807
max	4103.738453

Checking the median of the data.

```
df['Annual_Income'] = pd.to_numeric(df['Annual_Income'], errors='coerce')
```

```
grouped_annual = df.groupby('Occupation')['Annual_Income'].mean().reset_index().sort_values(by = 'Annual_Income',ascending = False)
grouped_annual
```

	Occupation	Annual_Income
7	Lawyer	196902.095402
11	Musician	193646.143008
3	Doctor	193396.271264
5	Entrepreneur	190136.314500
0	Accountant	188485.516996
10	Media_Manager	188309.698113
15	_______	181639.665282
12	Scientist	176848.311358
2	Developer	176408.779327
4	Engineer	173357.385394
9	Mechanic	172979.871216
1	Architect	168287.777660
8	Manager	166528.152301
6	Journalist	164462.879393
14	Writer	162184.188665
13	Teacher	162108.368540

Next steps: Generate code with grouped_annual View recommended plots New interactive sheet

Insight:

1. The Lawyer makes the most salary annually (196902.095402).
2. The Teacher makes the least salary annually (162108.368540).

```
# Grouping the data on the basis of cust_Id aand getting the max of Num_Bank_Accounts
grouped_customer_num_bank_accounts = df.groupby(['Customer_ID'])['Num_Bank_Accounts'].agg(lambda x: x.mode()[0]).reset_index().sort_valu
grouped_customer_num_bank_accounts
```

	Customer_ID	Num_Bank_Accounts
1566	CUS_0x294b	10
9917	CUS_0xa411	10
11109	CUS_0xb5c9	10
8836	CUS_0x9438	10
4683	CUS_0x56ef	10
...	...	...
7652	CUS_0x82c2	0
4849	CUS_0x5993	-1
3330	CUS_0x43bc	-1
10204	CUS_0xa878	-1
4161	CUS_0x4f2a	-1

12500 rows × 2 columns

Next steps: Generate code with grouped_customer_num_bank_accounts View recommended plots New interactive sheet

Insight:

1. Customer CUS_0x294b has 10 number of banks, which is the highest.
2. Some of the count is negative which could be the anamolies in the data or just a place holder for null enteries.

```
grouped_customer_num_credit_cards = df.groupby(['Customer_ID'])['Num_Credit_Card'].agg(lambda x: x.mode()[0]).reset_index().sort_values(
grouped_customer_num_credit_cards
```

	Customer_ID	Num_Credit_Card
805	CUS_0x1d6f	11
8932	CUS_0x958b	11
7784	CUS_0x84df	11
9545	CUS_0x9ea6	11
2136	CUS_0x3187	11
...	...	...
1491	CUS_0x281d	1
10405	CUS_0xaba4	1
11665	CUS_0xbdbf	1
1135	CUS_0x22be	0
7259	CUS_0x7ce5	0

12500 rows × 2 columns

Next steps: Generate code with grouped_customer_num_credit_cards View recommended plots New interactive sheet

Insight:

1. Customer CUS_0x1d6f has 11 number of credit cards, which is the highest.
2. Customer CUS_0x7ce5 has 0 number of credit cards, which is the least.

```
df['Num_of_Loan'] = pd.to_numeric(df['Num_of_Loan'], errors='coerce')
```

```
grouped_customer_num_of_loan = df.groupby(['Customer_ID'])['Num_of_Loan'].agg(lambda x: x.mode()[0]).reset_index().sort_values('Num_of_L
grouped_customer_num_of_loan
```

	Customer_ID	Num_of_Loan
3999	CUS_0x4d1c	9.0
5875	CUS_0x682b	9.0
540	CUS_0x18e2	9.0
4997	CUS_0x5bba	9.0
7149	CUS_0x7b3	9.0
...	...	...
4203	CUS_0x4fc9	0.0
4209	CUS_0x4feb	0.0
4222	CUS_0x5029	0.0
4225	CUS_0x5036	0.0
6250	CUS_0x6da9	0.0

12500 rows × 2 columns

Next steps: Generate code with grouped_customer_num_of_loan View recommended plots New interactive sheet

Insight:

1. The customer CUS_0x4d1c has highest number of loans 9.
2. The customer CUS_0x6da9 has least number of loans 0.

```
grouped_customer_delayed_payments = df.groupby(['Customer_ID'])['Num_of_Delayed_Payment'].agg(lambda x: x.mode()[0]).reset_index().sort_
grouped_customer_delayed_payments
```

	Customer_ID	Num_of_Delayed_Payment
10450	CUS_0xac39	28.0
2024	CUS_0x2fd6	27.0
10629	CUS_0xaea7	26.0
0	CUS_0x1000	25.0
12326	CUS_0xd61	25.0
...	...	...
4913	CUS_0x5a90	-2.0
1536	CUS_0x28d0	-2.0
88	CUS_0x117d	-2.0
8107	CUS_0x8974	-2.0
9739	CUS_0xa1a0	-2.0

12500 rows × 2 columns

Next steps: Generate code with grouped_customer_delayed_payments View recommended plots New interactive sheet

Insight: The customer CUS_0xac39 has 28 delayed number, highest number of dealyed payment.

```
# Grouping the values on customer_Id level for suming the Num_Credit_Inquiries.
grouped_customer_num_credit_inquiry = df.groupby(['Customer_ID'])['Num_Credit_Inquiries'].agg(lambda x: x.mode()[0]).reset_index().sort_
grouped_customer_num_credit_inquiry
```

	Customer_ID	Num_Credit_Inquiries
2618	CUS_0x392a	17.0
2637	CUS_0x3992	17.0
1887	CUS_0x2dc0	17.0
9743	CUS_0xa1ab	17.0
10084	CUS_0xa6a7	17.0
...	...	...
2512	CUS_0x37c9	0.0
461	CUS_0x17b0	0.0
1050	CUS_0x2174	0.0
11688	CUS_0xbe20	0.0
2198	CUS_0x32ac	0.0

12500 rows × 2 columns

Next steps: Generate code with grouped_customer_num_credit_inquiry View recommended plots New interactive sheet

Insight: The customer CUS_0x392a has highest number of inquiries 17.

```
# Converting the column values to numerics.
df['Outstanding_Debt'] = pd.to_numeric(df['Outstanding_Debt'],errors = 'coerce')
```

```
# Grouping the values on customer_Id level for suming the outstanding amount.
grouped_customer_outstanding_debt = df.groupby(['Customer_ID'])['Outstanding_Debt'].agg(lambda x: x.mode()[0]).reset_index().sort_value
grouped_customer_outstanding_debt
```

	Customer_ID	Outstanding_Debt
12189	CUS_0xc5ee	4998.07
6938	CUS_0x7837	4997.10
1180	CUS_0x2366	4997.05
3605	CUS_0x4791	4992.25
8829	CUS_0x9423	4990.91
...	...	...
10363	CUS_0xab0e	0.77
635	CUS_0x1a5c	0.56
12125	CUS_0xc4f4	0.54
2530	CUS_0x3801	0.34
3937	CUS_0x4c5	0.23

12500 rows × 2 columns

Next steps: Generate code with grouped_customer_outstanding_debt View recommended plots New interactive sheet

Insight:

1. The customer CUS_0xc5ee has 4998.07 outstanding debt, which is the highest debt.
2. The customer CUS_0x4c5 has 0.23 outstanding debt, which is the lowest debt.

```
# Grouping the values on customer_Id level for suming the Credit_Utilization_Ratio  .
grouped_customer_outstanding_debt = df.groupby(['Customer_ID'])['Credit_Utilization_Ratio'].median().reset_index().sort_values('Credit_U
grouped_customer_outstanding_debt
```

	Customer_ID	Credit_Utilization_Ratio
5765	CUS_0x6698	43.774821
5617	CUS_0x6479	42.696945
2380	CUS_0x35ac	42.092632
7017	CUS_0x7968	41.989372
9642	CUS_0xa014	41.765097
...	...	...
5543	CUS_0x636d	24.840293
12101	CUS_0xc4a2	24.830454
8992	CUS_0x9684	24.514389
8028	CUS_0x883e	24.391950
12197	CUS_0xc600	23.698861

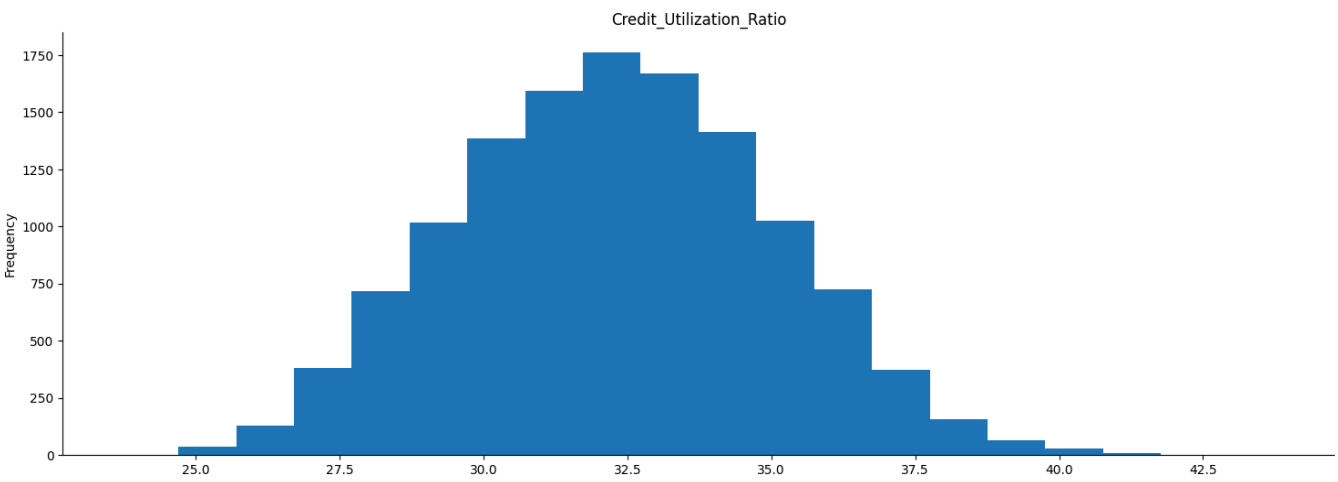
12500 rows × 2 columns

Next steps: Generate code with grouped_customer_outstanding_debt View recommended plots New interactive sheet

Insight:

1. The customer CUS_0x6698 has avg credit utilization of 43.774821 which is the highest.
2. The customer CUS_0xc600 has avg credit utilization of 23.698861 which is the lowest.

```
# Plotting the graph
grouped_customer_outstanding_debt['Credit_Utilization_Ratio'].plot(kind='hist', bins=20, title='Credit_Utilization_Ratio',figsize = (18,
plt.gca().spines[['top', 'right',]].set_visible(False)
```



Insight: The mean of the credit utilization is about 40%.

```
df['Type_of_Loan'].nunique()
```

```
6261
```

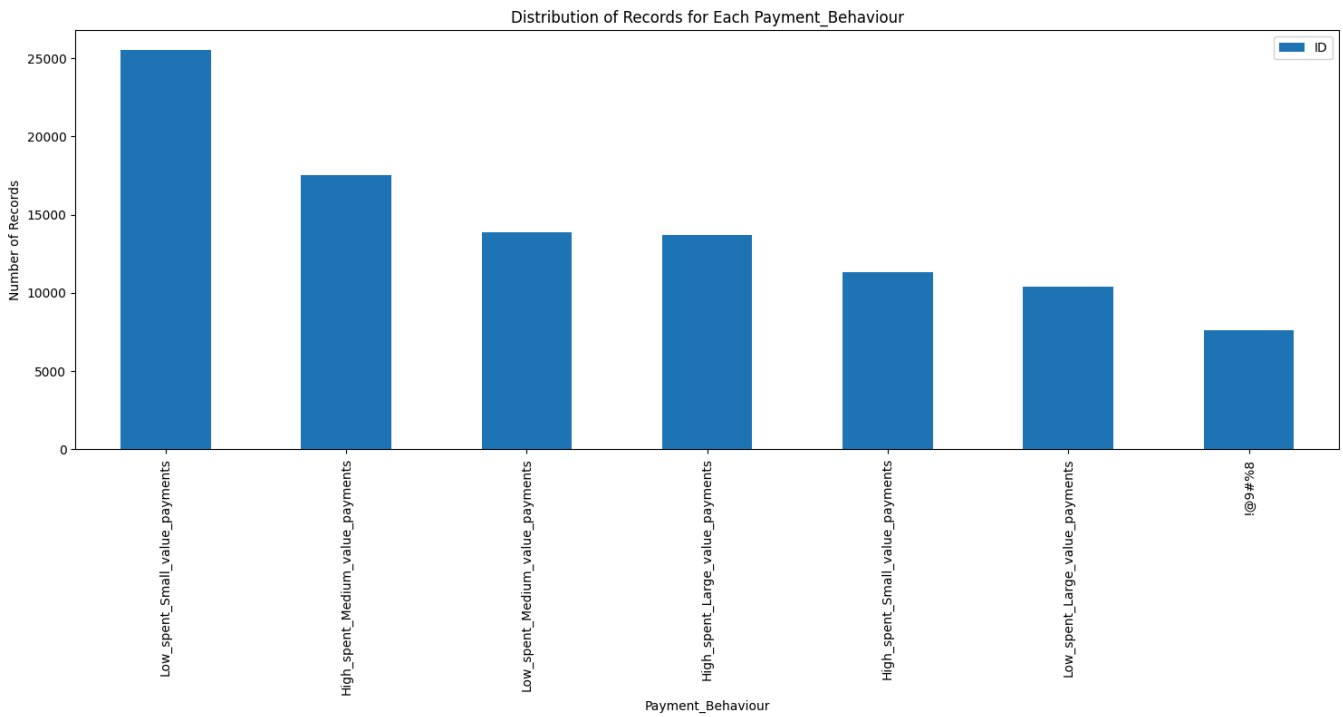
Insight: There are 6261 types of loan that the customers have.

```
# Grouping the payment behaviour for counting the total number of records.
grouped_Payment_behaviour = df.groupby(['Payment_Behaviour'])['ID'].count().reset_index().sort_values('ID', ascending=False)
grouped_Payment_behaviour
```

	Payment_Behaviour	ID
6	Low_spent_Small_value_payments	25513
2	High_spent_Medium_value_payments	17540
5	Low_spent_Medium_value_payments	13861
1	High_spent_Large_value_payments	13721
3	High_spent_Small_value_payments	11340
4	Low_spent_Large_value_payments	10425
0	!@9#%8	7600

Next steps: Generate code with grouped_Payment_behaviour | View recommended plots | New interactive sheet

```
# Plotting the graph
grouped_Payment_behaviour.plot(kind = 'bar',x = 'Payment_Behaviour',y = 'ID',figsize = (18,6))
plt.xlabel('Payment_Behaviour')
plt.ylabel('Number of Records')
plt.title('Distribution of Records for Each Payment_Behaviour')
plt.show()
```



Insight:

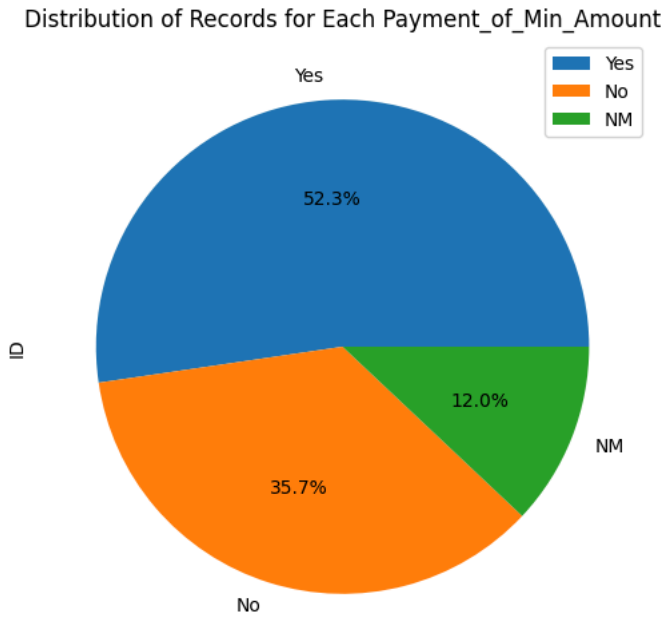
1. Low_spent_Small_value_payments accounts for the highest num. of types of payment behaviour.
2. !@9#%8 presents the anamolies in the dataset or just the placeholder used for null values.

```
# Grouping the Payment_of_Min_Amountto count total records in each category.
grouped_payment_of_min_amount = df.groupby(['Payment_of_Min_Amount'])['ID'].count().reset_index().sort_values('ID', ascending=False)
grouped_payment_of_min_amount
```

	Payment_of_Min_Amount	ID
2	Yes	52326
1	No	35667
0	NM	12007

Next steps: Generate code with grouped_payment_of_min_amount | View recommended plots | New interactive sheet

```
# plotting a graph.
grouped_payment_of_min_amount.plot(kind='pie', y='ID', figsize=(6, 6), autopct='%1.1f%%', labels=['Yes', 'No', 'NM'])
plt.title('Distribution of Records for Each Payment_of_Min_Amount')
plt.show()
```



Insight:

1. 52.3 % of the records have customer who pays the minimum amount.
2. 35.7 % of the records have customer who doesn't pay minimum amount.

```
# Grouping the customer ID to sum the Total_EMI_per_month   .
grouped_total_EMI_per_month  = df.groupby(['Customer_ID'])['Total_EMI_per_month'].agg(lambda x : x.mode()[0]).reset_index().sort_values(
grouped_total_EMI_per_month
```

	Customer_ID	Total_EMI_per_month
9761	CUS_0xa1ec	1779.103254
10551	CUS_0xadb2	1701.955013
2962	CUS_0x3ebe	1679.017067
4561	CUS_0x552f	1645.529388
7321	CUS_0x7dcd	1642.825388
...	...	...
4051	CUS_0x4ddb	0.000000
10314	CUS_0xaa3d	0.000000
8122	CUS_0x89a	0.000000
10311	CUS_0xaa39	0.000000
6250	CUS_0x6da9	0.000000

12500 rows × 2 columns

Next steps: Generate code with grouped_total_EMI_per_month | View recommended plots | New interactive sheet

Insight:

1. The customer CUS_0xa1ec has the highest Total EMI per month (1779.103254).
2. The customer CUS_0x6da9 has the lowest Total EMI per month (0).

Credit Scoring

Defining function: It will check 8 rules :

1. Credit Utilization Ratio < 40%
2. Num of Delayed Payments < 3
3. Diverse Loan Types
   len(Diverse Loan Types) < 2 or Not Specified
4. Credit Mix Bad or _
5. Num Credit Inquiries < 6
6. Debt-to-Income Ratio < 40%
7. Payment of Min Amount
   Yes or NM
8. Credit History Age > 36 months

It check if more than half times the total rules are failing then it gives the score 0 else 1.

```
def calculate_credit_score(row):
    fail_count = 0  # Initialize a counter for failed rules
    total_rules = 8  # Total number of rules being checked

    # Rule 1: Credit Utilization Ratio < 40%
    if row['Credit_Utilization_Ratio'] >= 40:
        fail_count += 1  # Increment fail count

    # Rule 2: Num of Delayed Payments < 3
    if row['Num_of_Delayed_Payment'] >= 3:
        fail_count += 1  # Increment fail count

    # Rule 3: Diverse Loan Types
    loan_types = row['Type_of_Loan'].split(', ')
    unique_loans = set(loan_types)  # Set to count unique loan types
    if 'Not Specified' in unique_loans or len(unique_loans) < 2:
        fail_count += 1  # Increment fail count

    # Rule 4: Credit Mix
    if row['Credit_Mix'] == 'Bad' or row['Credit_Mix'] == '_':
        fail_count += 1  # Increment fail count

    # Rule 5: Num Credit Inquiries < 6
    if row['Num_Credit_Inquiries'] >= 6:
        fail_count += 1  # Increment fail count

    # Rule 6: Debt-to-Income Ratio < 40%
    debt_to_income_ratio = row['Outstanding_Debt'] / row['Annual_Income'] * 100
    if debt_to_income_ratio >= 40:
        fail_count += 1  # Increment fail count

    # Rule 7: Payment of Min Amount
    if row['Payment_of_Min_Amount'] == 'Yes':
        fail_count += 1  # Increment fail count
    elif row['Payment_of_Min_Amount'] == 'NM':
        fail_count += 1  # Treat NM as a penalty

    # Rule 8: Credit History Age > 36 months
    if row['Credit_History_Months'] < 36:
        fail_count += 1  # Increment fail count

    # Check if the number of failed rules is greater than half the total rules
    if fail_count > total_rules / 2:
        return 0  # More than half failed, return 0
    else:
        return 1  # Less than or equal to half failed, return 1

# Apply the function to the DataFrame
df['Credit_Score'] = df.apply(calculate_credit_score, axis=1)
```

```
df.head()
```

	ID	Customer_ID	Month	Age	SSN	Occupation	Annual_Income	Monthly_Inhand_Salary	Num_Bank_Accounts	Num_Credit_Card
0	0x1602	CUS_0xd40	January	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	4
1	0x1603	CUS_0xd40	February	23.0	821-00-0265	Scientist	19114.12	3093.745000	3	4
2	0x1604	CUS_0xd40	March	-500.0	821-00-0265	Scientist	19114.12	3093.745000	3	4
3	0x1605	CUS_0xd40	April	23.0	821-00-0265	Scientist	19114.12	3093.745000	3	4
4	0x1606	CUS_0xd40	May	23.0	821-00-0265	Scientist	19114.12	1824.843333	3	4

```
df['Credit_Score'].value_counts()
```

Credit_Score	count
1	84920
0	15080

dtype: int64

Insight:

1. According to credit score nearly 85K rows here(i.e. the customer maybe duplicate) are credit-worthy.
2. According to credit score nearly 15K rows here(i.e. the customer maybe duplicate) are not credit-worthy.

```
Start coding or generate with AI.
```