# Goals

1. Which LoanPurpose categories have the highest and lowest default rates?

2. Compare the highest, lowest and average credit score as well as the supporting features.

3. What is the average interest rate per income group lower, middle and high class.

4. What is the corelation between the supporting features and the Credit Score.

# Importing Libraries and Excel spreadsheet

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
loan_df = pd.read_excel("C:\\Your\\File\\Path\\Loan_Information.xlsx")
```

# EDA

Starting with getting an idea of the spreadsheets size, column names and data types.

In [ ]:
```
print(loan_df.shape) #shape to get number of columns and rows.
```

```
(255347, 18)
```

followed by using the dtypes function to get a better understanding of the types of data i'll be working with. object represents a mixture of data types(or strings), int64 represents integers and float64 represents floats.

In [ ]:
```
print(loan_df.dtypes) #returns a series of the data types of all columns
```

```
LoanID             object
Age                 int64
Income              int64
LoanAmount          int64
CreditScore         int64
MonthsEmployed      int64
NumCreditLines      int64
InterestRate      float64
LoanTerm            int64
DTIRatio          float64
Education          object
EmploymentType     object
MaritalStatus      object
HasMortgage        object
HasDependents      object
LoanPurpose        object
HasCoSigner        object
Default             int64
dtype: object
```

using isna() and sum() in conjunction to get the total number of missing values from each column.

In [ ]:
```
print(loan_df.isna().sum()) #checks for any missing values per column
```

```
LoanID            0
Age               0
Income            0
LoanAmount        0
CreditScore       0
MonthsEmployed    0
NumCreditLines    0
InterestRate      0
LoanTerm          0
DTIRatio          0
Education         0
EmploymentType    0
MaritalStatus     0
HasMortgage       0
HasDependents     0
LoanPurpose       0
HasCoSigner       0
Default           0
dtype: int64
```

No missing values, so no need to drop or replace anything and I can move on to the working on getting a general summary of the dataset.

```
In [ ]: print(loan_df.columns) #gets the individual columns in the spreadsheet
```

```
Index(['LoanID', 'Age', 'Income', 'LoanAmount', 'CreditScore',
       'MonthsEmployed', 'NumCreditLines', 'InterestRate', 'LoanTerm',
       'DTIRatio', 'Education', 'EmploymentType', 'MaritalStatus',
       'HasMortgage', 'HasDependents', 'LoanPurpose', 'HasCoSigner',
       'Default'],
      dtype='object')
```

Using the describe function to return the mean, total count, standard deviation, minimum and maximum values from all numerical columns.

```
In [ ]: print(loan_df.describe()) #returns a general numeric summary per column
```

```
                 Age         Income     LoanAmount    CreditScore  \
count  255347.000000  255347.000000  255347.000000  255347.000000   
mean       43.498306   82499.304597  127578.865512     574.264346   
std        14.990258   38963.013729   70840.706142     158.903867   
min        18.000000   15000.000000    5000.000000     300.000000   
25%        31.000000   48825.500000   66156.000000     437.000000   
50%        43.000000   82466.000000  127556.000000     574.000000   
75%        56.000000  116219.000000  188985.000000     712.000000   
max        69.000000  149999.000000  249999.000000     849.000000   

       MonthsEmployed  NumCreditLines   InterestRate       LoanTerm  \
count   255347.000000   255347.000000  255347.000000  255347.000000   
mean        59.541976        2.501036      13.492773      36.025894   
std         34.643376        1.117018       6.636443      16.969330   
min          0.000000        1.000000       2.000000      12.000000   
25%         30.000000        2.000000       7.770000      24.000000   
50%         60.000000        2.000000      13.460000      36.000000   
75%         90.000000        3.000000      19.250000      48.000000   
max        119.000000        4.000000      25.000000      60.000000   

            DTIRatio        Default  
count  255347.000000  255347.000000  
mean        0.500212       0.116128  
std         0.230917       0.320379  
min         0.100000       0.000000  
25%         0.300000       0.000000  
50%         0.500000       0.000000  
75%         0.700000       0.000000  
max         0.900000       1.000000  
```

# EDA Summary

Its a spreadsheet focused on financial information. It's a big dataset it has 18 columns and 255347 rows. The columns hold some information that will be useful when answering our earlier presented questions specifically; Income, Credit Score,Employment Type, and Interest Rate. There aren't any missing values to replace, data types seem to be fine from a initial glance everything looks to be in working order. Given the vast diversity of the columns and the size of the dataset this represents a great opportunity to do some machine learning predictions with. However, for now I want to focus on answering the first 4 questions I outlined.

# Question 1

1. Which LoanPurpose categories have the highest and lowest default rates?

I need to start with understanding the different values in the LoanPurpose column. Ordinarily i'd use nunique() for the number of uniuqe values and and unique() to return the unique values but Value_counts() is much better suited for this purpose. It's a combination of nunique() and unique() it'll return the unique values alongside the number of occurrences per value.

In [ ]:
```
print(loan_df['LoanPurpose'].value_counts())
```

```
LoanPurpose
Business     51298
Home         51286
Education    51005
Other        50914
Auto         50844
Name: count, dtype: int64
```

There are 5 unique values throughout the loan purpose column. Business, Home, Education, Auto and Other. From here I'll use the groupby function to separate each unique loan purposes and their supporting features. Similar to SQL this would be the equivalent of

SELECT Default FROM Loan_df GROUP BY LoanPurpose

or more specifically, for this task

SELECT AVG(Default) FROM Loan_df GROUP BY LoanPurpose

In [ ]:
```
grouped = loan_df.groupby('LoanPurpose')
```

From here we introduce the column default and i'll make some cosmetic changes not necessary but it helps me for quicker analysis and I find that it's easier to read. Starting by getting the average number of defaults per LoanPurpose and multiplying that average by 100 so it resembles a percentage.

In [ ]:
```
default_rates = grouped['Default'].mean() * 100
print(default_rates.sort_values()) #I could have used np.sort here but it would return them sorted without the i
```

```
LoanPurpose
Home         10.234762
Other        11.788506
Education    11.838055
Auto         11.881441
Business     12.326017
Name: Default, dtype: float64
```

This answers the question, Home has the lowest percentage of defaults and Business has the highest. Lastly i'll change the answer to be an actual percentage using a for loop to iterate through the variable default_rates, .index to get the keys(columns) included inside of default_rates and zip to pair the values inside of default_rates with their corresponding keys making a key pair.

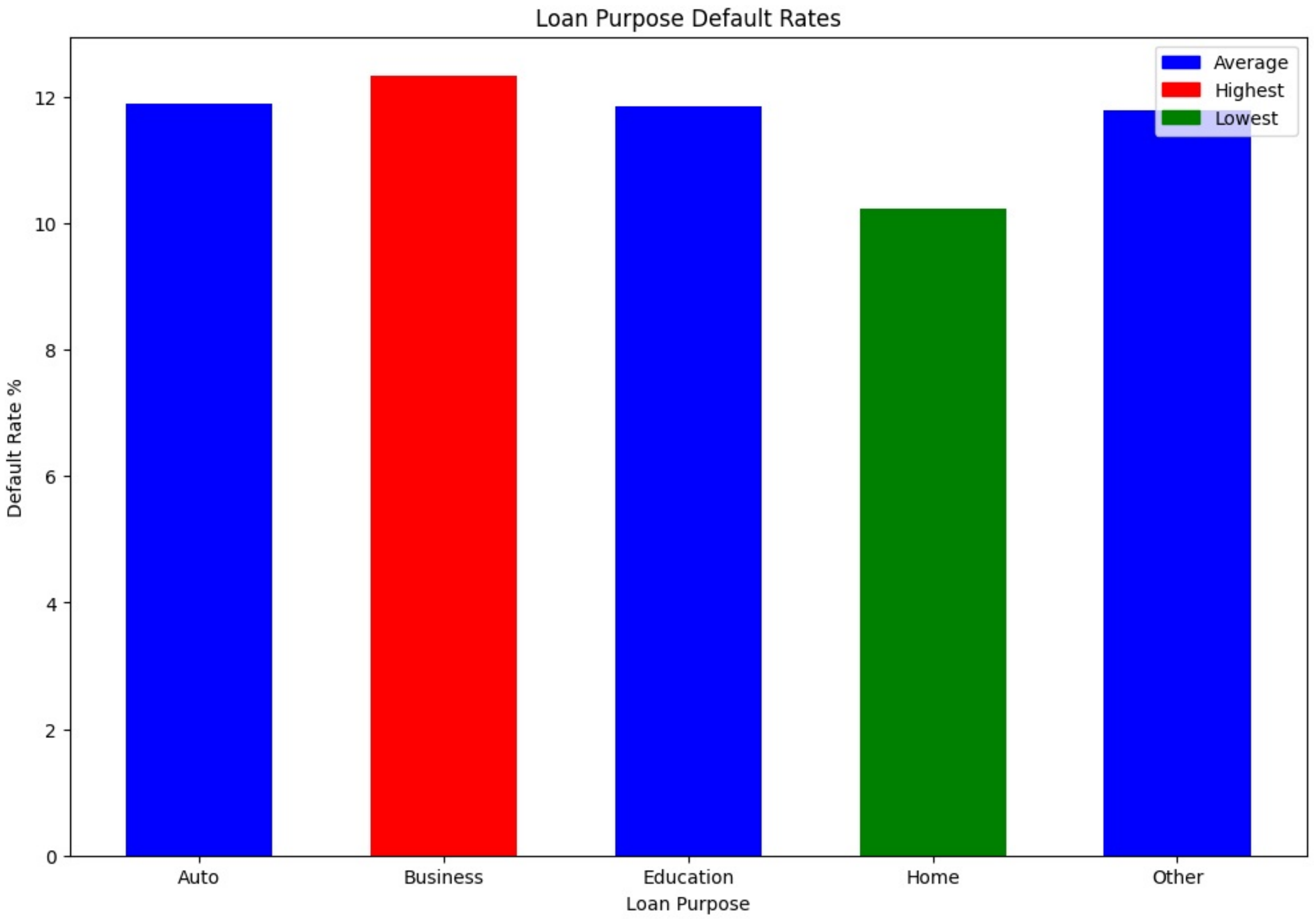
In [ ]:
```
default_rates = default_rates.round(2) #rounding so that it isn't drawn out
for purpose, rate in zip(default_rates.index, default_rates): #pairing the indexed position with the values
    print("{}: {}%".format(purpose, rate))
```

```
Auto: 11.88%
Business: 12.33%
Education: 11.84%
Home: 10.23%
Other: 11.79%
```

Lastly i'll create a bar chart to display the results, the percentages help but sometimes its easier to take a quick glance and see what your looking for.

In [ ]:
```
colors = ['blue', 'red', 'blue', 'green', 'blue'] #predefined colors(using red to highlight the highest and gre
plt.figure(figsize=(12, 8)) #adjusting chart size, since its the only one im making it slightly larger
plt.bar(default_rates.index, default_rates, width= 0.6, color=colors) # adjusting the width of each bar/assigni
plt.title('Loan Purpose Default Rates') #the name at the top of the chart

plt.xlabel('Loan Purpose') #x label ( wording on bottom of chart)
plt.ylabel('Default Rate %') #y label (wording on left side of chart)

#adding a legend
#Plt.rectangle = determining the way the legend is presented(the shape)
#((0, 0)) = the position of the icon in the legend
#1, 1 = the dimensions of the icon
#color = the color of the icon
#labels = [] = a list of the labels each icon will represent
#plt.Rectangle((0,0), 1, 1, color = 'blue')


plt.legend(handles=[plt.Rectangle((0,0),1,1, color="blue"),
                    plt.Rectangle((0,0),1,1, color="red"),
                    plt.Rectangle((0,0),1,1, color="green")],
           labels=['Average', 'Highest', 'Lowest'])
```

```
Out[ ]: <matplotlib.legend.Legend at 0x213514f4c10>
```




Although I've answered the first question I'm curious about the percent of people who do default on their payments. The first step should be to get the total number of entries in the default column(Side note I could've done this a bit smarter since I knew it didnt have any missing values I could've assumed that the total number of entries in the Default column is the same as the total amount of columns in the dataset).

```
In [ ]: print(len(loan_df['Default']))
```

```
255347
```

From here I need to create a boolean mask to get the number of people in the dataset that have defaulted at least once. Similar to in SQL we'd do

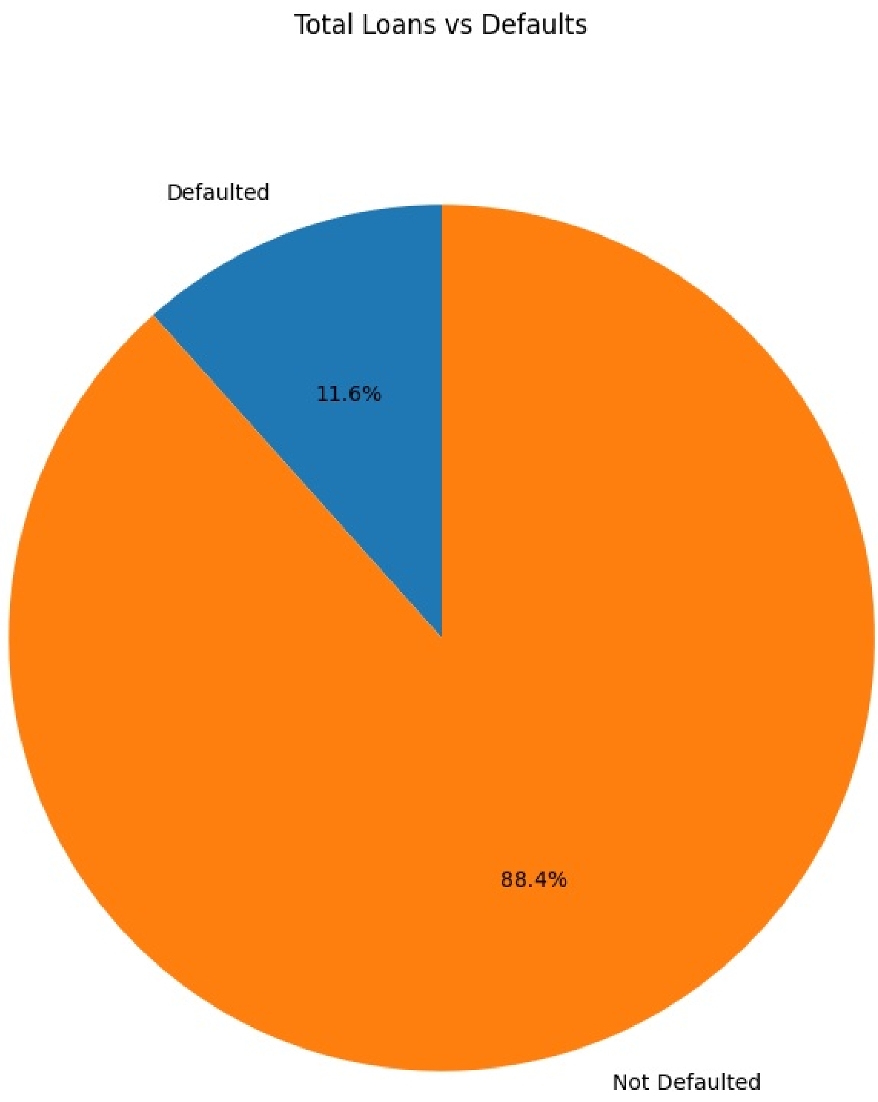
SELECT COUNT(Default) FROM Loan_df WHERE Default > 0

```
In [ ]: defaulted = loan_df[loan_df['Default'] > 0]
        print(len(defaulted))
```

```
29653
```

29,653 out of 255,347. We can put that into a pie chart for easier understanding and visualization.

```
In [ ]: labels = ['Defaulted', 'Not Defaulted'] #instead of creating a legend I attached the labels directly to each sl

        #sizes takes multiple arguments but here it will take two one for the amount of defaults vs the total amount
        sizes = [len(defaulted), 225694]

        fig, ax = plt.subplots(figsize=(8, 10))
        ax.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
        ax.axis('equal') #ensuring that the pie chart keeps its shape
        plt.title('Total Loans vs Defaults') #Title of the pie chart
        plt.show()
```




# Question 2

2. Compare the highest, lowest and average credit score as well as the supporting features.

using idxmax() and idxmin(), in collaboration with loc. or iloc. the idxmin/idxmax will return the indexed position for the maximum/minimum value(one issue I didn't anticipate, idxmax/idxmin return the first instance of the condition it didnt mess with the current analysis but something to be aware of moving forward).

In [ ]:
```
max_credit_index = loan_df['CreditScore'].idxmax()
print(max_credit_index)
```

```
254
```

The (first occurrence) value for the maximum value is 254. To get that full value I can use either loc or iloc. The main difference is that iloc takes two numerical arguments whereas loc takes one numeric argument and one categorical argument(the column name). They can both be used without the additional argument but it will return all information associated with that row. Using the idxmax() from earlier we know that the row position is 254.

In [ ]:
```
# print(loan_df.loc[254])
# print(loan_df.iloc[254])
# print(loan_df.loc[max_credit_index])

# These three will all achieve the same results but for simplicity ill do the following:
print(loan_df.loc[max_credit_index])
```

```
LoanID              U6PDHQHG5M
Age                         65
Income                   41931
LoanAmount                5769
CreditScore                849
MonthsEmployed               7
NumCreditLines               2
InterestRate               7.7
LoanTerm                    60
DTIRatio                  0.68
Education                  PhD
EmploymentType   Self-employed
MaritalStatus         Divorced
HasMortgage                 No
HasDependents               No
LoanPurpose              Other
HasCoSigner                Yes
Default                      0
Name: 254, dtype: object
```

Repeating the process for the minimum values.

In [ ]:
```
min_credit_index = loan_df['CreditScore'].idxmin()
print(loan_df.iloc[693])
```

```
LoanID              92A029ST5W
Age                         36
Income                  107236
LoanAmount               11586
CreditScore                300
MonthsEmployed              60
NumCreditLines               2
InterestRate              7.74
LoanTerm                    24
DTIRatio                  0.83
Education           Bachelor's
EmploymentType   Self-employed
MaritalStatus           Single
HasMortgage                Yes
HasDependents              Yes
LoanPurpose               Auto
HasCoSigner                 No
Default                      0
Name: 693, dtype: object
```

# Function Libary

A function library is important to keeping code modular, efficient and clean any functions used throughout this script can be found here.

In [ ]:
```
#find maximum value
def find_highest(df, column):
  x = df[column].idxmax() #maximum index
  return(df.loc[x]) #in combination with the loc

#find lowest value
def find_lowest(df, column):
  x = df[column].idxmin() #minimum index
  return(df.loc[x]) #in combination with the loc

#Function to calculate the averages for all numerical columns
def find_average(df):
    averages = {}  # Empty dictionary to store the averages
    for col in df.columns:
        if df[col].dtype in ['int64', 'float64']:  #if the column is numeric
            avg = df[col].mean()  # Calculate the average
            averages[col] = avg  # Store the average in the dictionary
    return averages

#Function to make Income resemble a financial input
def format_income(value):
    return '${:,.2f}'.format(value)
```

With the newly created find_highest, find_lowest and find_average functions I can use them to get the highest, lowest and average credit scores throughout the dataset.

In [ ]:
```
highest_creditscore = find_highest(loan_df, 'CreditScore')
print(highest_creditscore)
```

```
LoanID                U6PDHQHG5M
Age                           65
Income                     41931
LoanAmount                  5769
CreditScore                  849
MonthsEmployed                 7
NumCreditLines                 2
InterestRate                 7.7
LoanTerm                      60
DTIRatio                    0.68
Education                    PhD
EmploymentType     Self-employed
MaritalStatus           Divorced
HasMortgage                   No
HasDependents                 No
LoanPurpose                Other
HasCoSigner                  Yes
Default                        0
Name: 254, dtype: object
```

In [ ]:
```
lowest_creditscore = find_lowest(loan_df, 'CreditScore')
print(lowest_creditscore)
```

```
LoanID                92A029ST5W
Age                           36
Income                    107236
LoanAmount                 11586
CreditScore                  300
MonthsEmployed                60
NumCreditLines                 2
InterestRate                7.74
LoanTerm                      24
DTIRatio                    0.83
Education             Bachelor's
EmploymentType     Self-employed
MaritalStatus             Single
HasMortgage                  Yes
HasDependents                Yes
LoanPurpose                 Auto
HasCoSigner                   No
Default                        0
Name: 693, dtype: object
```

In [ ]:
```
averages = find_average(loan_df)
print(averages)
```

```
{'Age': 43.498306226429136, 'Income': 82499.30459727351, 'LoanAmount': 127578.86551242035, 'CreditScore': 574.26
43461642392, 'MonthsEmployed': 59.541976212761455, 'NumCreditLines': 2.501035845339871, 'InterestRate': 13.49277
3480792808, 'LoanTerm': 36.02589417537703, 'DTIRatio': 0.5002120643673119, 'Default': 0.11612824901017048}
```

Since Education is a categorical variable it doesnt have a traditional mean instead to get the average we can use the mode, the value that occurs the most frequently throughout the column. We can get that a few ways. Way number one, find which value occurs the most frequently throughout the Education column in our dataset using mode().

In [ ]:
```
#way number one
print(loan_df['Education'].mode())
```

```
0    Bachelor's
Name: Education, dtype: object
```

Way number 2 is using the value_counts() to get the total number of occurrences for each value in the column and using idxmax() to see which value occurs the most frequently, I could use max() instead of idxmax() but that would return the indexed location and i'd have to pass that through .loc[].

In [ ]:
```
#Way number 2
education_average = loan_df['Education'].value_counts().idxmax()
print(education_average)
```

```
Bachelor's
```

This returned Bachelors, its the most frequently occuring value in the education column of our dataset, now I can store the results into a separate dataset and convert that to a dataframe for easier readability and comparison.

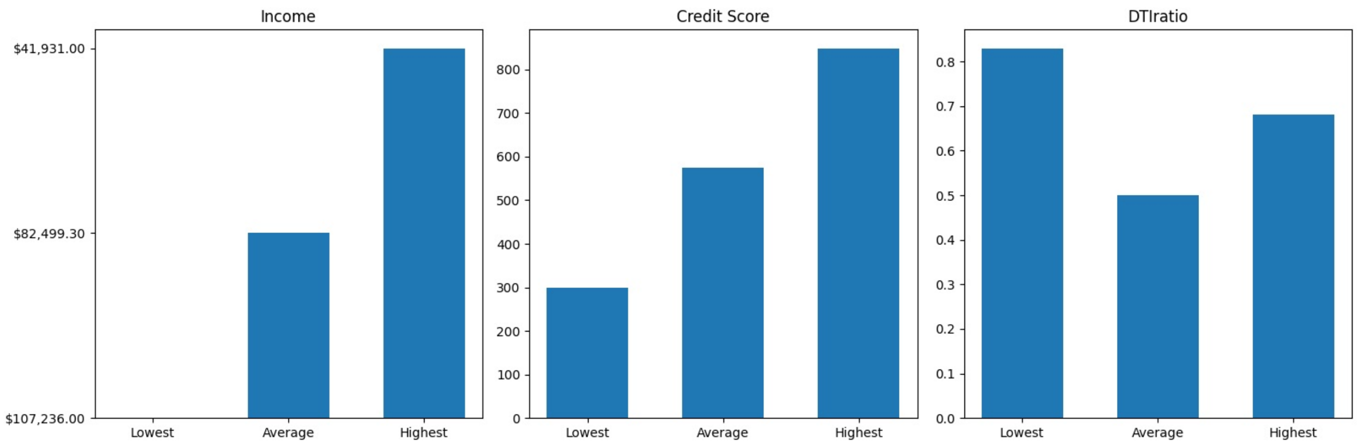
In [ ]:
```
highest_vs_lowest_creditscore = {
  'Name': ['Lowest', 'Average', 'Highest'],
  'Age': [lowest_creditscore['Age'], round(averages['Age'], 2), highest_creditscore['Age']],
  'Income': [format_income(lowest_creditscore['Income']), format_income(averages['Income']), format_income(high
  'Credit Score': [lowest_creditscore['CreditScore'], round(averages['CreditScore'], 2), highest_creditscore['C
  'Default': [lowest_creditscore['Default'], round(averages['Default'], 2), highest_creditscore['Default']],
  'DTIratio': [lowest_creditscore['DTIRatio'], round(averages['DTIRatio'], 2), highest_creditscore['DTIRatio']]
  'Education': [lowest_creditscore['Education'], education_average, highest_creditscore['Education']]
}
highest_vs_lowest_creditscore = pd.DataFrame(highest_vs_lowest_creditscore)
```

```
print(highest_vs_lowest_creditscore)
```

```
      Name   Age       Income  Credit Score  Default  DTIratio   Education
0   Lowest  36.0  $107,236.00        300.00     0.00      0.83  Bachelor's
1  Average  43.5   $82,499.30        574.26     0.12      0.50  Bachelor's
2  Highest  65.0   $41,931.00        849.00     0.00      0.68         PhD
```

From here I can start to plot my results, similar to the way I created the barplot earlier the biggest difference being the subplots line, i'll be plotting the entire dataframe instead of doing a multiline bar chart I want to emphasize the difference using four separate charts.

In [ ]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot for Age
axs[0].bar(highest_vs_lowest_creditscore['Name'], highest_vs_lowest_creditscore['Income'], width=0.6)
axs[0].set_title('Income')

# Plot for Credit Score
axs[1].bar(highest_vs_lowest_creditscore['Name'], highest_vs_lowest_creditscore['Credit Score'], width=0.6)
axs[1].set_title('Credit Score')

# Plot for DTIratio
axs[2].bar(highest_vs_lowest_creditscore['Name'], highest_vs_lowest_creditscore['DTIratio'], width=0.6)
axs[2].set_title('DTIratio')



plt.tight_layout()
plt.show()
```




Seeing it plotted out is a bit interesting, as expected Credit Score follows a familiar pattern building from left to right the DTI ratio does the opposite, it seems like the lower the credit score the higher the depth to income ratio but on average the depth to income ratio breaks even at about 0.5, and those with a higher credit score have a slightly slower DTI ratio than those with a lower credit score but are still able to maintain that credit score. Most interestingly enough, income is unlike anything I expected, compared to the highest credit score the highest makes $65,305.00 more annually. A stripling difference.

# Question 3

3. What is the average interest rate per income group lower, middle and high class.

To start I will break up the salary incomes into three parts, higher than average, average, and lower than average, seperating them into three individual parts makes the data easier to understand and classify.

In [ ]:
```
# function to separate income groups
def income_group(income):
    if income >= 100000:
        return 'Higher Than Average'
    elif income >= 50000:
        return 'Average'
    else:
        return 'Lower Than Average'
```

From here i'll create a new column called IncomeGroup to host the classifications, and use the apply() function to apply the function throughout the Income column. This way IncomeGroup becomes essentially a subset(dependent) of Income.

In [ ]:
```
#creating new column(IncomeGroup) as a subset of Income and applying the function to the entire group
loan_df['IncomeGroup'] = loan_df['Income'].apply(income_group)
```

I'll use the group by function to separate the classes in the IncomeGroup column to do some general analysis on per class.

In [ ]:
```
grouped_by_income = loan_df.groupby('IncomeGroup')
```

Since the goal is to get the general interest rate across the different classes I can use the mean() function to get the average per group, I'll also use sort_values() to sort them in descending order typically from ascending to descending.

In [ ]:
```
avg_interest_by_income = grouped_by_income['InterestRate'].mean()
print(avg_interest_by_income.sort_values(ascending= False)) #sorting from highest to lowest
```

```
IncomeGroup
Average                13.516817
Lower Than Average     13.496592
Higher Than Average    13.466032
Name: InterestRate, dtype: float64
```

They are all fairly consistent across our three classified distributions Income Groups. Interestingly enough it seems that out of the three income groups average has the highest of the three with low being the second lowest and lastly as expected High has the lowest. It's interesting to note that despite the reduction in income middle class(Average) and low class are still given a higher interest rate than those who find themselves in the Higher than average class.

# Question 4.

4. What is the corelation between the supporting features and the default.

To start I know that the dataframe has a mixture of data types including objects(strings) if I don't isolate just the numerical columns it will cause an error. Using select_dtypes and adding the np.number parameter to only get the numeric columns in the dataframe.

In [ ]:
```
numeric_df = loan_df.select_dtypes(include=[np.number])
print(numeric_df.columns)
```

```
Index(['Age', 'Income', 'LoanAmount', 'CreditScore', 'MonthsEmployed',
       'NumCreditLines', 'InterestRate', 'LoanTerm', 'DTIRatio', 'Default'],
      dtype='object')
```

Now I can use the .corr() to get the correlation between the features and the target(defaults). It's ranked from a scale of -1 to 1, 1 meaning theres a positive correlation the higher the feature the higher the target, -1 meaning the lower the feature the lower the target.

In [ ]:
```
correlation = numeric_df.corr()
print(correlation['CreditScore'])
```

```
Age              -0.000548
Income           -0.001430
LoanAmount        0.001261
CreditScore       1.000000
MonthsEmployed    0.000613
NumCreditLines    0.000016
InterestRate      0.000436
LoanTerm          0.001130
DTIRatio         -0.001039
Default          -0.034166
Name: CreditScore, dtype: float64
```

The amount of defaults seems to have the most significant negative impact on the credit score with a corelation of -0.034166, followed by loan term(0.001130)and loan amount(0.001261) contributing the most to a positive credit score.

# Conclusion

I've gone through each individual goal and worked towards answering them, i'll reiterate both questions and results below.

1. Which LoanPurpose categories have the highest and lowest default rates?

a. Starting with using the value_count() in combination with groupby() I got the total count of each unique value in the column as well as the number of times they were mentioned. After separating I pulled the defaulted mean per loan type and multiplied the results by 100 to resemble more of a percentage. From there it was just a matter of reading the numbers and making a general analysis. Business has the highest default percentage with it being 12.33%, meaning 12.33% of business loans in this dataset are defaulted on. 29,653 out of 255,347 are generally defaulted on making up 11.6% of the total dataset.

2. Compare the highest, lowest and average credit score as well as the supporting features.

a. Using idxmax(),idxmin(), loc and iloc, I found the rows with the highest and lowest credit scores. From here I was able to calculate the average for each numeric column in the dataset which proved to be a bit shocking. The highest credit score was 849, held by a 65-year-old with a PhD but a below-average income. The lowest credit score was 300, held by a 36-year-old with a Bachelor's degree and an above-average income. The average credit score was 574.26, showing a wide range around this mean. This analysis suggests that credit scores in this dataset are not strictly correlated with income or education, indicating that other factors are at play. I used a function library here to avoid some repetition and more thoroughly explain my code and decision making. Pairing my results with bar charts to better depict the results, as expected the charts moved from left to right continuously increasing for both age and credit score but showed an interesting change for depth to income ratio with the lowest being higher than both average and high potentially impacting the low credit score

3. What is the average interest rate per income group lower, middle and high class.

a. Because the dataset did not come preset with the income groups being separated I created a function to do so based on the user's income they are broken down into three groups lower than average, average and higher than average. After the creation I ran a quick script to ensure that everything was included(could've just used len instead of shape to avoid getting the number of columns as well). From here I used groupby() again to isolate the unique values in my new column and followed that with .mean to get the average interest rate per group. Interestingly enough the interest rates didnt vary much at all, all hovering around 13% but it's worthy to note that the average interest rate was higher than the two other groups by a marginal amount.

4. What is the corelation between the supporting features and the Credit Score.

a. The corelation between the supporting features and the Credit Score was a bit interesting to do, typically i've worked with the correlations but identifying them as weights in a machine learning model. The results were as expected, with the exception of income and number of credit lines. There impact was practically nonexistent, I certainly expected a much stronger correlation between the two. Defaults performed as expected having the strongest negative impact.