Part 1: Programming Solution

Task 1: Loan Data Automation

Loading Data from Different File Formats:

Excel Files:

- To import data from Excel files, the pandas library provides a simple function, `pd.read_excel('file_path.xlsx')`

PDF Files:

- While pandas doesn't have a built-in function for PDFs, there's a reliable solution: the **tabula-py** library, specifically designed for extracting tabular data from PDFs. A community thread on Stack Overflow recommends this tool

Learn more about tabula-py here: https://pypi.org/project/tabula-py/

In [ ]:
```
# First we install tabula using pip
!pip install tabula-py
```

```
Requirement already satisfied: tabula-py in /usr/local/lib/python3.10/dist-packages (2.9.0)
Requirement already satisfied: pandas>=0.25.3 in /usr/local/lib/python3.10/dist-packages (from tabula-py) (2.0.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from tabula-py) (1.25.2)
Requirement already satisfied: distro in /usr/lib/python3/dist-packages (from tabula-py) (1.7.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25.3->tabula-p
y) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25.3->tabula-py) (2023.
4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25.3->tabula-py) (202
4.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=0.25.3
->tabula-py) (1.16.0)
```

Importing necessary libraries

In [ ]:

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

Loading in the Excel file

In [ ]:

```
# loading the excel file using pandas method .read_excel
loans_xlsx = pd.read_excel("/content/drive/MyDrive/AFS/APEX Loan Data.xlsx")
loans_xlsx.head()
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Te
0	2284	1	0	0	0	0	3902	1666.0	109	
1	2287	2	0	0	1	0	1500	1800.0	103	
2	2288	1	1	2	0	0	2889	0.0	45	
3	2296	1	0	0	0	0	2755	0.0	65	
4	2297	1	0	0	1	0	2500	20000.0	103	

In [ ]:

```
# checking the datatypes of the dataframe
loans_xlsx.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 247 entries, 0 to 246
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            247 non-null    int64  
 1   Gender             247 non-null    int64  
 2   Married            247 non-null    int64  
 3   Dependents         247 non-null    int64  
 4   Graduate           247 non-null    int64  
 5   Self_Employed      247 non-null    int64  
 6   ApplicantIncome    247 non-null    int64  
 7   CoapplicantIncome  247 non-null    float64
 8   LoanAmount         247 non-null    int64  
 9   Loan_Amount_Term   247 non-null    int64  
 10  Credit_History     247 non-null    int64  
 11  Property_Area      247 non-null    int64  
 12  Loan_Status        247 non-null    object 
dtypes: float64(1), int64(11), object(1)
memory usage: 25.2+ KB
```

In [ ]:

```
# Converting `CoapplicantIncome` to a integer data type
loans_xlsx['CoapplicantIncome'] = loans_xlsx['CoapplicantIncome'].astype(int)
```

In [ ]:

```
# Confirming changes
loans_xlsx.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 247 entries, 0 to 246
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            247 non-null    int64
 1   Gender             247 non-null    int64
 2   Married            247 non-null    int64
 3   Dependents         247 non-null    int64
 4   Graduate           247 non-null    int64
 5   Self_Employed      247 non-null    int64
 6   ApplicantIncome    247 non-null    int64
 7   CoapplicantIncome  247 non-null    int64
 8   LoanAmount         247 non-null    int64
 9   Loan_Amount_Term   247 non-null    int64
 10  Credit_History     247 non-null    int64
 11  Property_Area      247 non-null    int64
 12  Loan_Status        247 non-null    object
dtypes: int64(12), object(1)
memory usage: 25.2+ KB
```

Loading in the PDF tables

In [ ]:

```
# loading the pdf tables using tabula-py
loans_df_list = tabula.read_pdf("/content/drive/MyDrive/AFS/APEX_Loans_Database_Table.pdf", pages='all')
```

In [ ]:

```
# Tabula loads all the tables in the pdf file as a list of dataframes
# printing the type and number of objects in the loans_df_list list
print(len(loans_df_list))
print(type(loans_df_list))
```

```
14
<class 'list'>
```

The number of tables in the list (14) matches the number of pages in the original PDF

Cleaning PDF dataframes

In [ ]:
```
# Checking the first dataframe in the list
loans_df_list[0].head()
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Te
0	1002	1	0	0	1	0	5849	0	128	
1	1003	1	1	1	1	0	4583	1508	128	
2	1005	1	1	0	1	1	3000	0	66	
3	1006	1	1	0	0	0	2583	2358	120	
4	1008	1	0	0	1	0	6000	0	141	

In [ ]:
```
# Checking the second dataframe in the list
loans_df_list[1].head()
```

Out[ ]:

	1086	1	0	0.1	0.2	0.3	1442	0.4	35	360	1.1	1.2	N
0	1087	2	0	2	1	0	3750	2083	120	360	1	2	Y
1	1091	1	1	1	1	0	4166	3369	201	360	0	1	N
2	1095	1	0	0	1	0	3167	0	74	360	1	1	N
3	1097	1	0	1	1	1	4692	0	106	360	1	3	N
4	1098	1	1	0	1	0	3500	1667	114	360	1	2	Y

Issue:

- The first dataframe has correct column headers, but later dataframes mistakenly use the first data row as headers.
- This prevents direct concatenation using `pd.concat` due to mismatched headers.

Solution:

1. **Isolate Correct Header DataFrame:**

   - Create an empty list, `processed_dfs`, to store corrected dataframes.
   - Add the first dataframe (with correct headers) to `processed_dfs` as it is.

2. **Fix Remaining DataFrames:**

   - Iterate through the rest of the dataframes:
     - **Extract Misplaced Headers:** Create a new dataframe using the first row (which contains misplaced headers) as its first data row.
     - **Add Correct Headers:** Assign appropriate column headers from the first dataframe to this new dataframe.
     - **Concatenate with Corrected Data:** Combine this header-fixed dataframe with the remaining rows of the original dataframe using `dataframe_list[i].iloc[1:]`.
     - **Rename Columns:** Apply the correct headers from the first dataframe to ensure consistency.
     - **Append to Processed DataFrames:** Add the corrected dataframe to the `processed_dfs` list.

3. **Concatenate Corrected DataFrames:**

   - Merge all dataframes in `processed_dfs` using `pd.concat` without header-related errors, as they now share a consistent header structure.

In [ ]:

```
# function to resolve the issue
def concatenate_pdfs(dataframe_list):
  """Concatenates a list of DataFrames, handling potential header inconsistencies.

  This function addresses cases where initial DataFrames may have incorrect
  headers (e.g., using the first data row as headers). It ensures consistent
  headers before concatenation.

  Args:
    dataframe_list: A list of DataFrames to be concatenated.

  Returns:
    A single DataFrame containing the concatenated data, with consistent headers.

  Raises:
    ValueError: If the number of column headers and new names provided for renaming
               are not equal.
  """
```

```
    # Create an empty list to store the processed DataFrames
    processed_dfs = []

    # Process the first DataFrame as-is
    processed_dfs.append(dataframe_list[0])

    # Process the remaining DataFrames
    for i in range(1, len(dataframe_list)):
        # Extract column headers and create desired names
        column_headers = dataframe_list[i].columns
        new_column_names = dataframe_list[0].columns.to_list()  # Replace with your desired names

        # Check if the number of headers and new names match
        if len(column_headers) != len(new_column_names):
            raise ValueError("Number of column headers and new names must be equal")

        # Create a new DataFrame with headers as the first row
        new_df = pd.DataFrame([column_headers], columns=column_headers)

        # Insert the new row and rename columns
        processed_df = pd.concat([new_df, dataframe_list[i].iloc[1:]], ignore_index=True).rename(columns=dict(zip(column_headers
        processed_dfs.append(processed_df)


    # Concatenate the processed DataFrames into a single DataFrame
    loans_df = pd.concat(processed_dfs, ignore_index=True)
    return loans_df
```

In [ ]:
```
# Running the function
loans_pdf = concatenate_pdfs(loans_df_list)
```

In [ ]:
```
# checking the datatypes of the combined dataframe
loans_pdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 385 entries, 0 to 384
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype 
---  ------             --------------  ----- 
 0   Loan_ID            385 non-null    object
 1   Gender             385 non-null    object
 2   Married            385 non-null    object
 3   Dependents         385 non-null    object
 4   Graduate           385 non-null    object
 5   Self_Employed      385 non-null    object
 6   ApplicantIncome    385 non-null    object
 7   CoapplicantIncome  385 non-null    object
 8   LoanAmount         385 non-null    object
 9   Loan_Amount_Term   385 non-null    object
 10  Credit_History     385 non-null    object
 11  Property_Area      385 non-null    object
 12  Loan_Status        385 non-null    object
dtypes: object(13)
memory usage: 39.2+ KB
```

While `loans_pdf.info()` output shows all columns as 'object', we expect some to be numeric. We should convert these columns to appropriate numeric data types.

In [ ]:
```
# Viewing a section of the dataframe such that we see rows that were at the end and beginning of the tables in the pdf
loans_pdf.loc[27:70]
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amour
27	1073	1	1	2	0	0	4226	1040	110	
28	1086	1	0	0.1	0.2	0.3	1442	0.4	35	
29	1091	1	1	1	1	0	4166	3369	201	
30	1095	1	0	0	1	0	3167	0	74	
31	1097	1	0	1	1	1	4692	0	106	
32	1098	1	1	0	1	0	3500	1667	114	
33	1100	1	0	3	1	0	12500	3000	320	
34	1106	1	1	0	1	0	2275	2067	128	
35	1109	1	1	0	1	0	1828	1330	100	
36	1112	2	1	0	1	0	3667	1459	144	
37	1114	1	0	0	1	0	4166	7210	184	
38	1116	1	0	0	0	0	3748	1668	110	
39	1119	1	0	0	1	0	3600	0	80	
40	1120	1	0	0	1	0	1800	1213	47	
41	1123	1	1	0	1	0	2400	0	75	
42	1131	1	1	0	1	0	3941	2336	134	
43	1136	1	1	0	0	1	4695	0	96	
44	1137	2	0	0	1	0	3410	0	88	
45	1138	1	1	1	1	0	5649	0	44	
46	1144	1	1	0	1	0	5821	0	144	
47	1146	2	1	0	1	0	2645	3440	120	
48	1151	2	0	0	1	0	4000	2275	144	

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amour
49	1155	2	1	0	0	0	1928	1644	100	
50	1157	2	0	0	1	0	3086	0	120	
51	1164	2	0	0	1	0	4230	0	112	
52	1179	1	1	2	1	0	4616	0	134	
53	1186	2	1	1	1	1	11500	0	286	
54	1194	1	1	2	1	0	2708	1167	97	
55	1195	1	1	0	1	0	2132	1591	96	
56	1197	1	1.1	0	1.2	0.1	3366	2200	135	
57	1199	1	1	2	0	0	3357	2859	144	
58	1205	1	1	0	1	0	2500	3796	120	
59	1206	1	1	3	1	0	3029	0	99	
60	1207	1	1	0	0	1	2609	3449	165	
61	1213	1	1	1	1	0	4945	0	128	
62	1222	2	0	0	1	0	4166	0	116	
63	1225	1	1	0	1	0	5726	4595	258	
64	1228	1	0	0	0	0	3200	2254	126	
65	1233	1	1	1	1	0	10750	0	312	
66	1238	1	1	3	0	1	7100	0	125	
67	1241	2	0	0	1	0	4300	0	136	
68	1243	1	1	0	1	0	3208	3066	172	
69	1245	1	1	2	0	1	1875	1875	97	
70	1248	1	0	0	1	0	3500	0	81	

Issue:

- There is an issue with Loan IDs 1086 and 1197. Their integer values (`Married` and others) appear as floats with a single decimal place (e.g., "1.1" instead of "1").
- Manually verifying the PDF confirms these are indeed integers, and this conversion error seems to affect all first rows of the PDF tables.

Solution:

1. **Data Type Conversion:** We'll convert all columns except the last one to numeric data types. The last column will be converted to a categorical type.
2. **Float to Integer Correction:** The `floor` function from the `math` module rounds down a number to the nearest integer.

In [ ]:
```
# We select the dataframe except the last column
int_cols = loans_pdf.iloc[:, :-1]

# Then we apply pd.to_numeric() to the integer columns
loans_pdf[int_cols.columns] = int_cols.apply(lambda x: pd.to_numeric(x))

# And convert the loan status column to categorical
loans_pdf["Loan_Status"] = loans_pdf["Loan_Status"].astype('category')
```

In [ ]:
```
# Checking the datatypes again
loans_pdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 385 entries, 0 to 384
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            385 non-null    int64
 1   Gender             385 non-null    int64
 2   Married            385 non-null    float64
 3   Dependents         385 non-null    float64
 4   Graduate           385 non-null    float64
 5   Self_Employed      385 non-null    float64
 6   ApplicantIncome    385 non-null    int64
 7   CoapplicantIncome  385 non-null    float64
 8   LoanAmount         385 non-null    int64
 9   Loan_Amount_Term   385 non-null    int64
 10  Credit_History     385 non-null    float64
 11  Property_Area      385 non-null    float64
 12  Loan_Status        385 non-null    category
dtypes: category(1), float64(7), int64(5)
memory usage: 36.7 KB
```

In [ ]:
```
# Checking the dataframe values
loans_pdf.head()
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Te
0	1002	1	0.0	0.0	1.0	0.0	5849	0.0	128	
1	1003	1	1.0	1.0	1.0	0.0	4583	1508.0	128	
2	1005	1	1.0	0.0	1.0	1.0	3000	0.0	66	
3	1006	1	1.0	0.0	0.0	0.0	2583	2358.0	120	
4	1008	1	0.0	0.0	1.0	0.0	6000	0.0	141	

In [ ]:
```
# Now we convert the numeric columns to integers by first flooring all the numeric columns and then converting their
# datatypes to integers using math.floor
```

```python
# Selecting numeric columns
numeric_cols = loans_pdf.select_dtypes(include='number').columns

# Apply math.floor() to the numeric columns
loans_pdf[numeric_cols] = loans_pdf[numeric_cols].apply(lambda x: x.map(math.floor))

# Converting loans_pdf numeric cplumns to integers
loans_pdf[numeric_cols] = loans_pdf[numeric_cols].astype(int)
```

In [ ]:
```python
# Checking the datatypes to confirm changes
loans_pdf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 385 entries, 0 to 384
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype   
---  ------             --------------  -----   
 0   Loan_ID            385 non-null    int64   
 1   Gender             385 non-null    int64   
 2   Married            385 non-null    int64   
 3   Dependents         385 non-null    int64   
 4   Graduate           385 non-null    int64   
 5   Self_Employed      385 non-null    int64   
 6   ApplicantIncome    385 non-null    int64   
 7   CoapplicantIncome  385 non-null    int64   
 8   LoanAmount         385 non-null    int64   
 9   Loan_Amount_Term   385 non-null    int64   
 10  Credit_History     385 non-null    int64   
 11  Property_Area      385 non-null    int64   
 12  Loan_Status        385 non-null    category
dtypes: category(1), int64(12)
memory usage: 36.7 KB
```

In [ ]:
```python
# Validating changes
loans_pdf.head()
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Te
0	1002	1	0	0	1	0	5849	0	128	
1	1003	1	1	1	1	0	4583	1508	128	
2	1005	1	1	0	1	1	3000	0	66	
3	1006	1	1	0	0	0	2583	2358	120	
4	1008	1	0	0	1	0	6000	0	141	

With both the PDF and Excel data loaded as DataFrames, we can now combine them using `pd.concat.`

Combining the two data sources

In [ ]:

```
# concatenating the dataframes and saving them in to a 'loans' dataframe
loans = pd.concat([loans_xlsx, loans_pdf], ignore_index=True)
```

In [ ]:

```
print(f"There are {loans_xlsx.shape[0]} rows in the pdf")
print(f"There are {loans_pdf.shape[0]} rows in the Excel file")
print(f"There are {loans.shape[0]} rows in the combined dataset")

# Checking if the concatenation worked
loans_pdf.shape[0] + loans_xlsx.shape[0] == loans.shape[0]

# Assert concatenation integrity
assert loans_pdf.shape[0] + loans_xlsx.shape[0] == loans.shape[0], "Concatenation failed: Row count mismatch"
```

```
There are 247 rows in the pdf
There are 385 rows in the Excel file
There are 632 rows in the combined dataset
```

Task 2: Descriptive Analytics

Data quality checks

In [ ]:
```python
# Visual assessment: checking the first 5 records
loans.head()
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Te
0	2284	1	0	0	0	0	3902	1666	109	
1	2287	2	0	0	1	0	1500	1800	103	
2	2288	1	1	2	0	0	2889	0	45	
3	2296	1	0	0	0	0	2755	0	65	
4	2297	1	0	0	1	0	2500	20000	103	

In [ ]:
```python
# number of rows in the dataframe
loans.shape[0]
```

Out[ ]: 632

In [ ]:
```python
# number of columns in the dataframe
loans.shape[1]
```

Out[ ]: 13

In [ ]:
```python
# what are these column names?
loans.columns
```

Out[ ]:
```
Index(['Loan_ID', 'Gender', 'Married', 'Dependents', 'Graduate',
       'Self_Employed', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',
       'Loan_Amount_Term', 'Credit_History', 'Property_Area', 'Loan_Status'],
      dtype='object')
```

In [ ]:
```python
# General info on the data
loans.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 632 entries, 0 to 631
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype 
---  ------             --------------  ----- 
 0   Loan_ID            632 non-null    int64 
 1   Gender             632 non-null    int64 
 2   Married            632 non-null    int64 
 3   Dependents         632 non-null    int64 
 4   Graduate           632 non-null    int64 
 5   Self_Employed      632 non-null    int64 
 6   ApplicantIncome    632 non-null    int64 
 7   CoapplicantIncome  632 non-null    int64 
 8   LoanAmount         632 non-null    int64 
 9   Loan_Amount_Term   632 non-null    int64 
 10  Credit_History     632 non-null    int64 
 11  Property_Area      632 non-null    int64 
 12  Loan_Status        632 non-null    object
dtypes: int64(12), object(1)
memory usage: 64.3+ KB
```

In [ ]:

```
# set Loan_ID as the dataframe index
loans = loans.set_index("Loan_ID")
loans.head()
```

Out[ ]:

Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
2284	1	0	0	0	0	3902	1666	109	333
2287	2	0	0	1	0	1500	1800	103	333
2288	1	1	2	0	0	2889	0	45	180
2296	1	0	0	0	0	2755	0	65	300
2297	1	0	0	1	0	2500	20000	103	333

Data cleaning

Correction of duplicates

In [ ]:
```
# checking for duplicates in the Loan_ID column
loans.reset_index()[loans.reset_index().duplicated(subset="Loan_ID",keep=False)].sort_values(by='Loan_ID')
```

Out[ ]:

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_
11	1900	1	1	1	1	0	2750	1842	115	
513	1900	1	1	1	1	0	2750	1842	115	
12	1903	1	1	0	1	0	3993	3274	207	
514	1903	1	1	0	1	0	3993	3274	207	
13	1904	1	1	0	1	0	3103	1300	80	
515	1904	1	1	0	1	0	3103	1300	80	
14	1907	1	1	0	1	0	14583	0	436	
516	1907	1	1	0	1	0	14583	0	436	
15	1908	2	1	0	0	0	4100	0	124	
517	1908	2	1	0	0	0	4100	0	124	
518	1910	1	0	1	0	1	4053	2426	158	
16	1910	1	0	1	0	1	4053	2426	158	
519	1914	1	1	0	1	0	3927	800	112	
17	1914	1	1	0	1	0	3927	800	112	
520	1915	1	1	2	1	0	2301	985	78	
18	1915	1	1	2	1	0	2301	985	78	
19	1917	2	0	0	1	0	1811	1666	54	
521	1917	2	0	0	1	0	1811	1666	54	
522	1922	1	1	0	1	0	20667	0	128	
20	1922	1	1	0	1	0	20667	0	333	
21	1924	1	0	0	1	0	3158	3053	89	
523	1924	1	0	0	1	0	3158	3053	89	

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_
22	1925	2	0	0	1	1	2600	1717	99	
524	1925	2	0	0	1	1	2600	1717	99	
23	1926	1	1	0	1	0	3704	2000	120	
525	1926	1	1	0	1	0	3704	2000	120	
526	1931	2	0	0	1	0	4124	0	115	
24	1931	2	0	0	1	0	4124	0	115	
527	1935	1	0	0	1	0	9508	0	187	
25	1935	1	0	0	1	0	9508	0	187	
528	1938	1	1	2	1	0	4400	0	127	
27	1938	1	1	2	1	0	4400	0	127	
28	1940	1	1	2	1	0	3153	1560	134	
529	1940	1	1	2	1	0	3153	1560	134	
530	1945	2	0	0	1	0	5417	0	143	
29	1945	2	0	0	1	0	5417	0	143	
531	1947	1	1	0	1	0	2383	3334	172	
30	1947	1	1	0	1	0	2383	3334	172	
532	1949	1	1	3	1	0	4416	1250	110	
31	1949	1	1	3	1	0	4416	1250	110	
533	1953	1	1	1	1	0	6875	0	200	
32	1953	1	1	1	1	0	6875	0	200	
534	1954	2	1	1	1	0	4666	0	135	
33	1954	2	1	1	1	0	4666	0	135	

	Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_
535	1955	2	0	0	1	0	5000	2541	151	
34	1955	2	0	0	1	0	5000	2541	151	
536	1963	1	1	1	1	0	2014	2925	113	
35	1963	1	1	1	1	0	2014	2925	113	
36	1964	1	1	0	0	0	1800	2934	93	
537	1964	1	1	0	0	0	1800	2934	93	
538	1972	1	1	0	0	0	2875	1750	105	
37	1972	1	1	0	0	0	2875	1750	105	
539	1974	2	0	0	1	0	5000	0	132	
38	1974	2	0	0	1	0	5000	0	132	
540	1977	1	1	1	1	0	1625	1803	96	
39	1977	1	1	1	1	0	1625	1803	96	
541	1978	1	0	0	1	0	4000	2500	140	
40	1978	1	0	0	1	0	4000	2500	140	
41	1990	1	0	0	0	0	2000	0	333	
542	1990	1	0	0	0	0	2000	0	128	

Duplicate rows do exists, so we remove them

In [ ]:

```
# Saving the number rows before dropping duplicates
x = loans.shape[0]

# reseting the index so that Loan_ID can be used as the subset parameter in .drop_duplicates
loans = loans.reset_index().drop_duplicates(subset=['Loan_ID'])
```

```python
# Saving the number of rows after dropping duplicates
y = loans.shape[0]

print(f"{x - y} rows were dropped as duplicates")
```

```
30 rows were dropped as duplicates
```

In [ ]:

```python
# setting Loan_ID back to index
loans = loans.set_index("Loan_ID")
loans.head()
```

Out[ ]:

	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
Loan_ID									
2284	1	0	0	0	0	3902	1666	109	333
2287	2	0	0	1	0	1500	1800	103	333
2288	1	1	2	0	0	2889	0	45	180
2296	1	0	0	0	0	2755	0	65	300
2297	1	0	0	1	0	2500	20000	103	333

Missing values

In [ ]:

```python
# checking the number of nulls per column
loans.isna().sum()
```

Out[ ]:
```
Gender               0
Married              0
Dependents           0
Graduate             0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status          0
dtype: int64
```

Now, we assign meaningful labels to the categories in our data will improve interpretation when performing Exploratory Data Analysis (EDA).

In [ ]:
```python
# Mapping categorical variables to actual values
loans['Gender'] = loans['Gender'].map({1: 'Male', 2: 'Female'})
loans['Married'] = loans['Married'].map({0: 'Single', 1: 'Married'})
loans['Graduate'] = loans['Graduate'].map({0: 'No', 1: 'Yes'})
loans['Self_Employed'] = loans['Self_Employed'].map({0: 'No', 1: 'Yes'})
loans['Credit_History'] = loans['Credit_History'].map({0: 'No', 1: 'Yes'})
loans['Property_Area'] = loans['Property_Area'].map({1: 'Urban', 2: 'Semiurban', 3: 'Rural'})
```

In [ ]:
```python
# Checking the transformed dataframe
loans.head()
```

Out[ ]:

Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
2284	Male	Single	0	No	No	3902	1666	109	333
2287	Female	Single	0	Yes	No	1500	1800	103	333
2288	Male	Married	2	No	No	2889	0	45	180
2296	Male	Single	0	No	No	2755	0	65	300
2297	Male	Single	0	Yes	No	2500	20000	103	333

Data type conversion

Several columns, including "Gender," "Married," "Dependents," "Graduate," "Self_Employed," "Credit_History," "Property_Area," and "Loan_Status," contain categorical data. To minimize memory consumption, it's recommended to convert them to a categorical data type.

In [ ]:
```
cat_columns = ['Gender', 'Married', 'Dependents', 'Graduate', 'Self_Employed', 'Credit_History', 'Property_Area', 'Loan_Status
loans[cat_columns] = loans[cat_columns].apply(lambda x: x.astype('category'))
```

In [ ]:
```
# checking after transformation
loans.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 602 entries, 2284 to 2281
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype   
---  ------             --------------  -----   
 0   Gender             602 non-null    category
 1   Married            602 non-null    category
 2   Dependents         602 non-null    category
 3   Graduate           602 non-null    category
 4   Self_Employed      602 non-null    category
 5   ApplicantIncome    602 non-null    int64   
 6   CoapplicantIncome  602 non-null    int64   
 7   LoanAmount         602 non-null    int64   
 8   Loan_Amount_Term   602 non-null    int64   
 9   Credit_History     602 non-null    category
 10  Property_Area      602 non-null    category
 11  Loan_Status        602 non-null    category
dtypes: category(8), int64(4)
memory usage: 29.3 KB
```

In [ ]:

```
# Using a list comprehension to print the unique values in each categorical column to check for errors
print("\n".join([f"\nValue counts for column '{col}':\n{loans[col].unique()}" for col in loans.select_dtypes(include=['categor
```

```
Value counts for column 'Gender':
['Male', 'Female']
Categories (2, object): ['Female', 'Male']

Value counts for column 'Married':
['Single', 'Married']
Categories (2, object): ['Married', 'Single']

Value counts for column 'Dependents':
[0, 2, 1, 3]
Categories (4, int64): [0, 1, 2, 3]

Value counts for column 'Graduate':
['No', 'Yes']
Categories (2, object): ['No', 'Yes']

Value counts for column 'Self_Employed':
['No', 'Yes']
Categories (2, object): ['No', 'Yes']

Value counts for column 'Credit_History':
['Yes', 'No']
Categories (2, object): ['No', 'Yes']

Value counts for column 'Property_Area':
['Rural', 'Semiurban', 'Urban']
Categories (3, object): ['Rural', 'Semiurban', 'Urban']

Value counts for column 'Loan_Status':
['Y', 'N']
Categories (2, object): ['N', 'Y']
```

There does not seem to be any unusual values

In [ ]:
```
# checking transformed dataframe
loans.head()
```

Out[ ]:

Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
2284	Male	Single	0	No	No	3902	1666	109	333
2287	Female	Single	0	Yes	No	1500	1800	103	333
2288	Male	Married	2	No	No	2889	0	45	180
2296	Male	Single	0	No	No	2755	0	65	300
2297	Male	Single	0	Yes	No	2500	20000	103	333

Visualising outliers

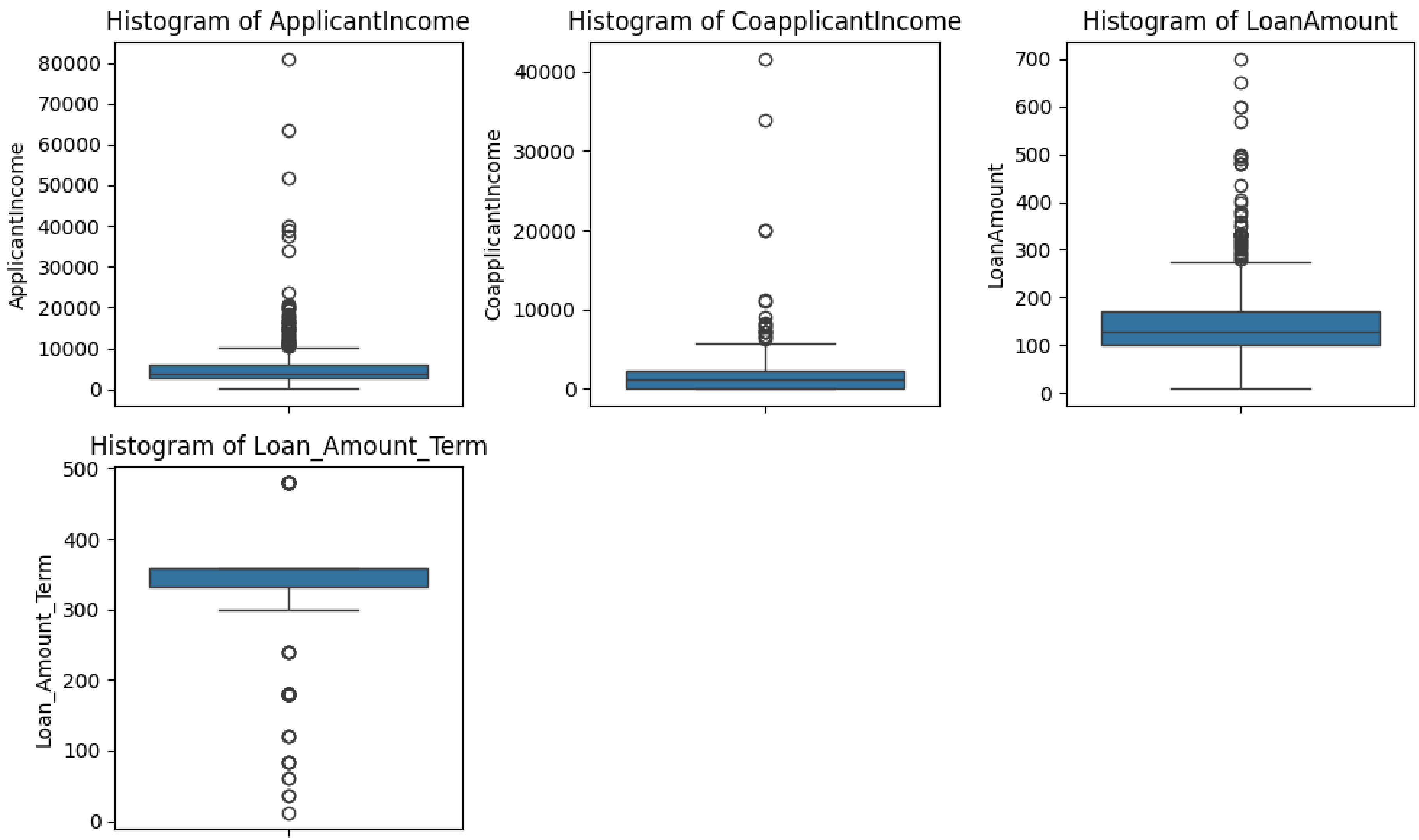
In [ ]:

```
def visualize_outliers(df):
    """
    Visualizes outliers in numeric columns of a DataFrame using boxplots.

    Args:
        df (pandas.DataFrame): The input DataFrame.

    Returns:
        None
    """
    numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns

    # Create a grid of subplots based on the number of numeric columns
    nrows = (len(numeric_cols) + 2) // 3  # Number of rows in the plot grid
    ncols = 3  # Number of columns in the plot grid
    fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(10, 3 * nrows))
    axes = axes.flatten()

    # Plot each numeric column in a separate subplot
    for i, col in enumerate(numeric_cols):
        ax = axes[i]
        sns.boxplot(data=df[col], ax=ax)
```

```python
        ax.set_title(f"Histogram of {col}")
        ax.tick_params(axis='x', rotation=45)

    # Hide unused subplots
    for j in range(i + 1, len(axes)):
        axes[j].set_visible(False)

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

In [ ]:

```python
visualize_outliers(loans)
```



Applicant Income Distribution:

The median applicant income sits at #3,813, indicating that half of the applicants earn less than this amount. However, the visualization reveals a wider range, with some applicants earning between #10,000 and #80,000. This suggests a potential income disparity among applicants.

Loan Amount Distribution:

The average loan amount is #149, as indicated by the mean value. However, the chart highlights the presence of outliers exceeding #300. These outliers seem to be valid data points and might represent applicants requiring larger loans. Since they are not errors, it's reasonable to keep them for further analysis.

Descriptive statistics

In [ ]:
```
# Descriptive stats of loans
loans.describe()
```

Out[ ]:

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
count	602.000000	602.000000	602.000000	602.000000
mean	5422.194352	1612.981728	148.935216	333.039867
std	6159.336898	2948.005981	87.674053	64.223080
min	150.000000	0.000000	9.000000	12.000000
25%	2877.500000	0.000000	100.000000	333.000000
50%	3813.500000	1128.500000	128.000000	360.000000
75%	5795.000000	2295.250000	170.000000	360.000000
max	81000.000000	41667.000000	700.000000	480.000000

Key points

- **Categorical Columns:**
  - There are around 632 entries for each categorical feature (Gender, Married, etc.).
  - There are more male than female applicants
  - Most applicants are married (mean of 0.65 for Married), have no dependents (median of 0 for Dependents), and are graduates (mean of 0.78 for Graduate).
  - Self-employment is uncommon (mean of 0.13 for Self_Employed).
- **Numerical Columns (Income and Loan):**

- Median income is lower than the mean, suggesting a positive skew (more applicants on the lower end). The median is around $3815, whilethemeanis5386$.
- Loan amounts also show a positive skew, with a median around $128andameanof148$. There are outliers exceeding $300 for loan amounts.
- Coapplicant income seems to follow a similar pattern to applicant income with a lower mean ($1599) compared to the median.
- Loan terms range from 12 to 480 months, with a mean of 334 months respectively.

- **Credit History and Property Area:**
  - The majority of applicants credit history (mean of 0.78).
  - Property area values range from 1 to 3, with a slight skew towards higher values (mean of 1.96). This means most applicants live in semi-urban areas

Exploratory Data Analysis

Univariate Analysis: Categorical columns

In [ ]:
```
# viewing only categorical
loans.select_dtypes(include='category').head()
```

Out[ ]:

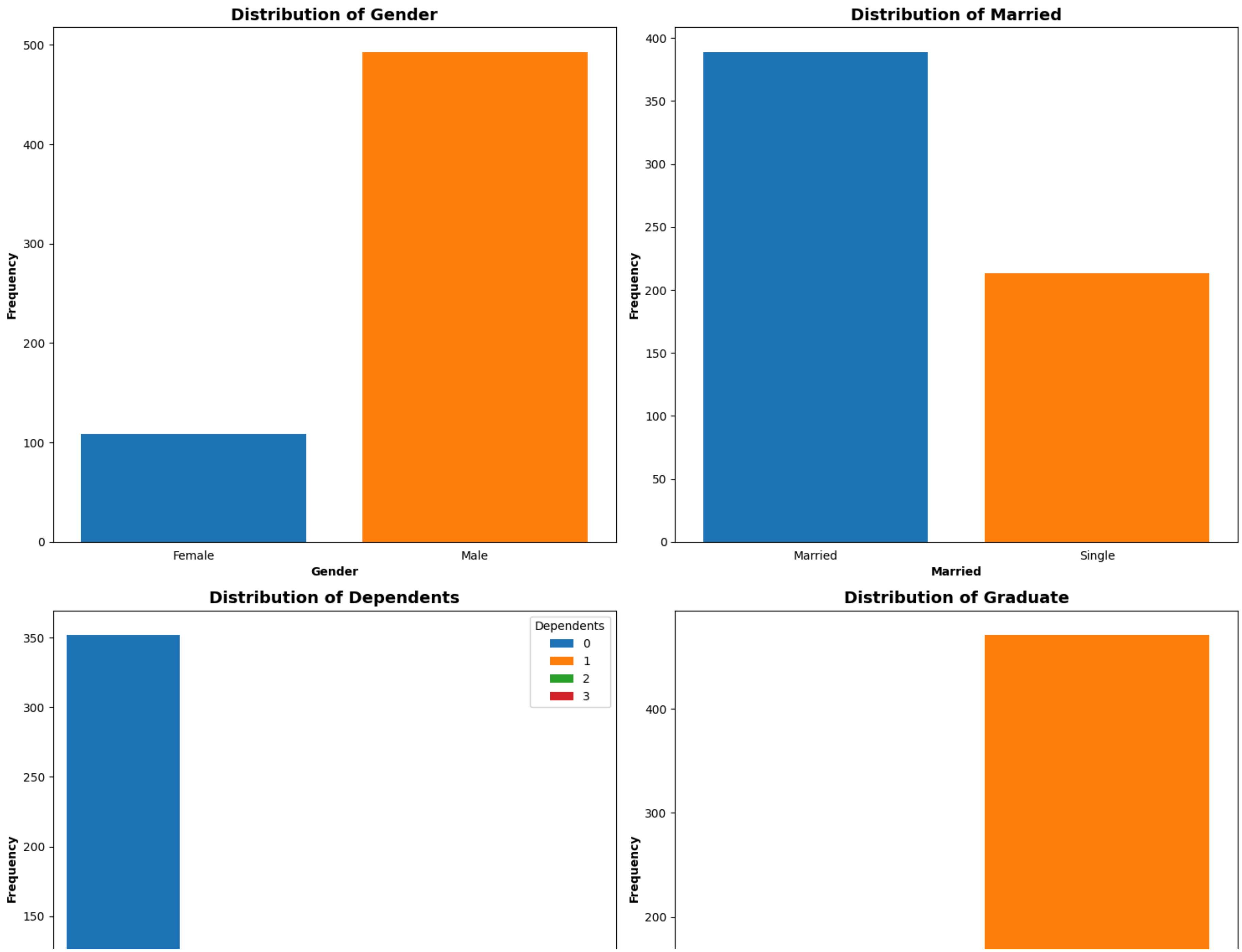
Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	Credit_History	Property_Area	Loan_Status
2284	Male	Single	0	No	No	Yes	Rural	Y
2287	Female	Single	0	Yes	No	No	Semiurban	N
2288	Male	Married	2	No	No	No	Urban	N
2296	Male	Single	0	No	No	Yes	Rural	N
2297	Male	Single	0	Yes	No	Yes	Semiurban	Y

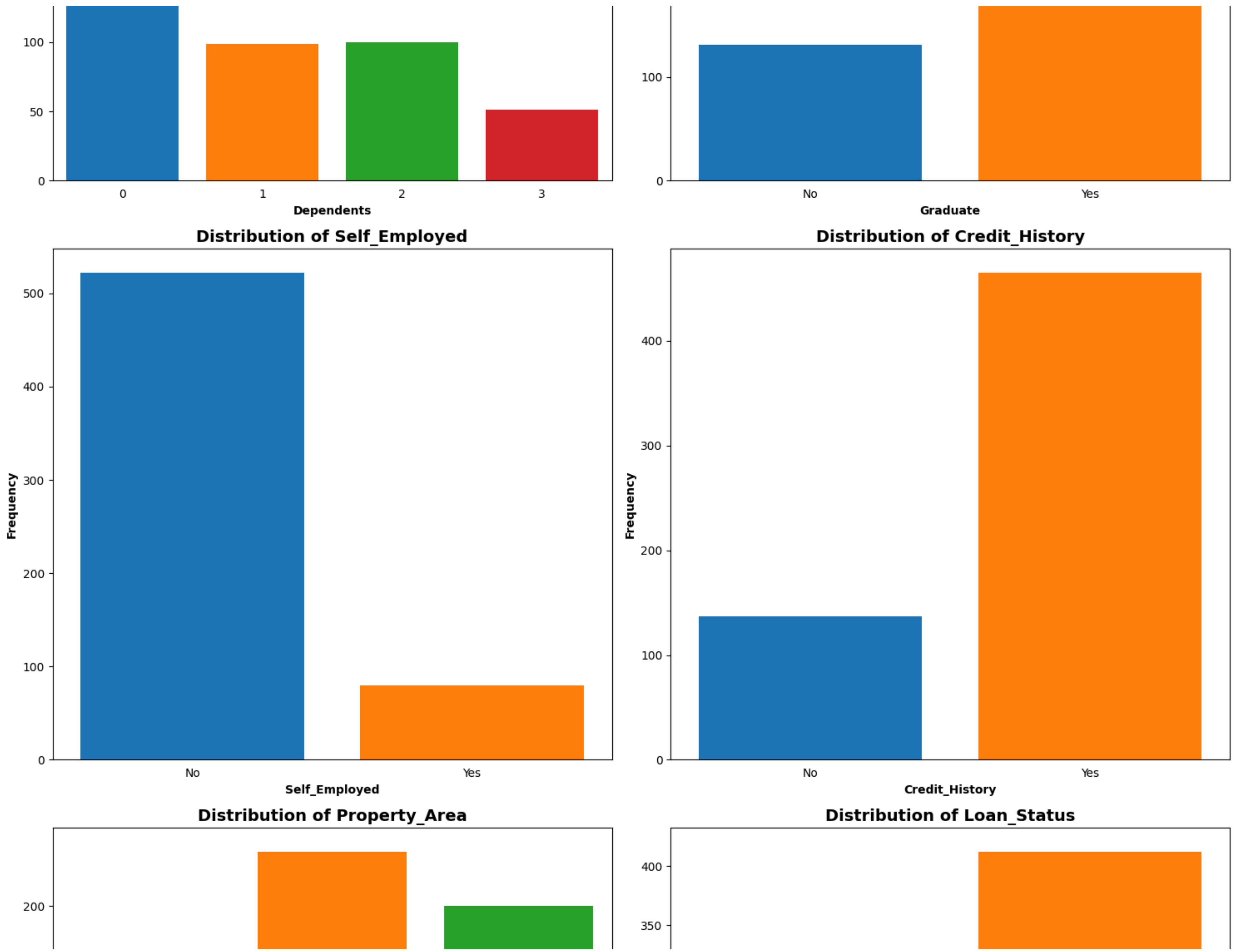
In [ ]:
```
def plot_categorical_distributions(df):
  """
  Plots count plots for all categorical columns in a pandas DataFrame.
```

```
    Args:
        df (pandas.DataFrame): The DataFrame containing the data.
    """
    categorical_cols = df.select_dtypes(include='category').columns  # Select categorical columns
    rows = int(np.ceil(len(categorical_cols) / 2))  # Calculate number of rows for subplots
    cols = 2  # Number of columns for subplots (adjust as needed)
    fig, axes = plt.subplots(rows, cols, figsize=(15, rows * 7))  # Create subplots

    for i, col in enumerate(categorical_cols):
      row = int(i / cols)
      col_index = i % cols
      ax = axes[row, col_index]
      sns.countplot(data=df, x=col, hue=col, saturation=1, ax=ax)  # Plot on current subplot
      ax.set_title(f'Distribution of {col}', fontdict={'fontsize': 14, 'fontweight':'bold'})
      ax.set_xlabel(col, fontweight= 'bold')
      ax.set_ylabel('Frequency', fontweight= 'bold')
      ax.tick_params(bottom=False)  # Remove x-axis ticks for readability

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

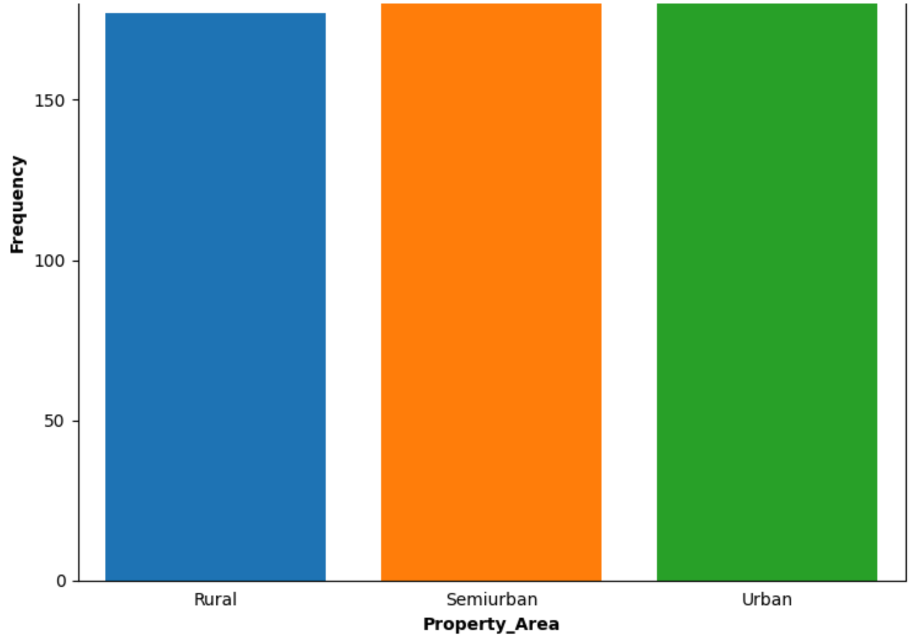
In [ ]:

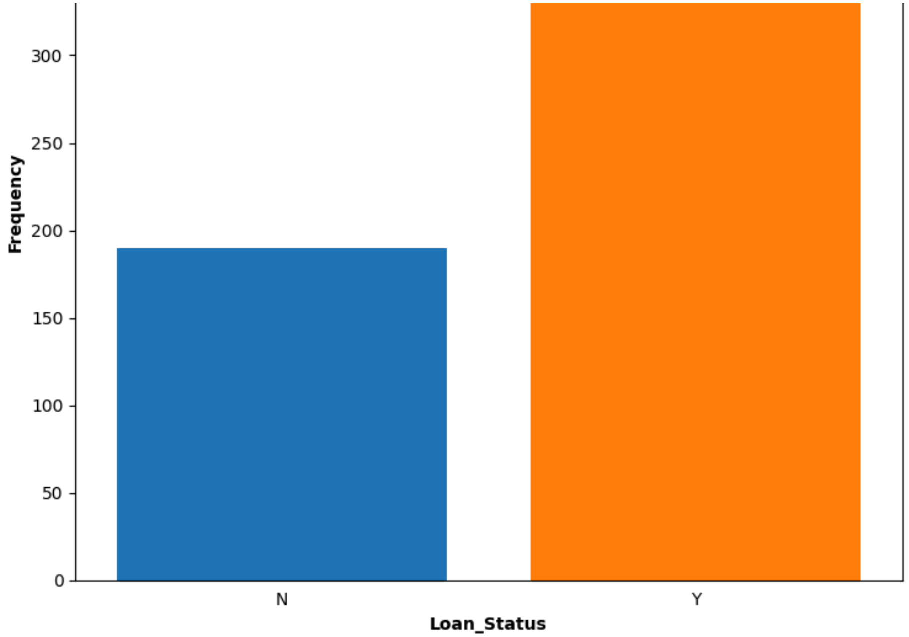
```
plot_categorical_distributions(loans)
```


Distribution of Gender
Frequency
Female
Male
Gender
Distribution of Married
Married
Single
Married
Distribution of Dependents
Dependents
0
1
2
3
Distribution of Graduate



Dependents
Graduate
Distribution of Self_Employed
Distribution of Credit_History
Frequency
Self_Employed
Credit_History
Distribution of Property_Area
Distribution of Loan_Status
No
Yes





Key Insights

- **Gender**: The dataset is heavily male-dominated, with 493 males compared to just 109 females.
- **Marital Status**: The majority of the dependents are married, with 389 married individuals compared to only 213 single individuals.
- **Dependents**: Most dependents have no dependents, at 352 individuals respectively.
- **Graduate**: Only 131 individuals are graduates, while the majority, 471 are non-graduates.
- **Self-Employed**: The majority of the applicants, 522 individuals, are not self-employed, while only 80 are self-employed.
- **Credit History**: The majority of the applicants, 465 individuals, have a credit history, while 137 have no credit history.
- **Loan Status**: The majority of the applicants, 190 individuals, have a loan status of 'N', while 412 have a loan status of 'Y'.

Univariate Analysis: Continuous variables

In [ ]:
```
loans.select_dtypes(include=['float64', 'int64']).head()
```

Out[ ]:

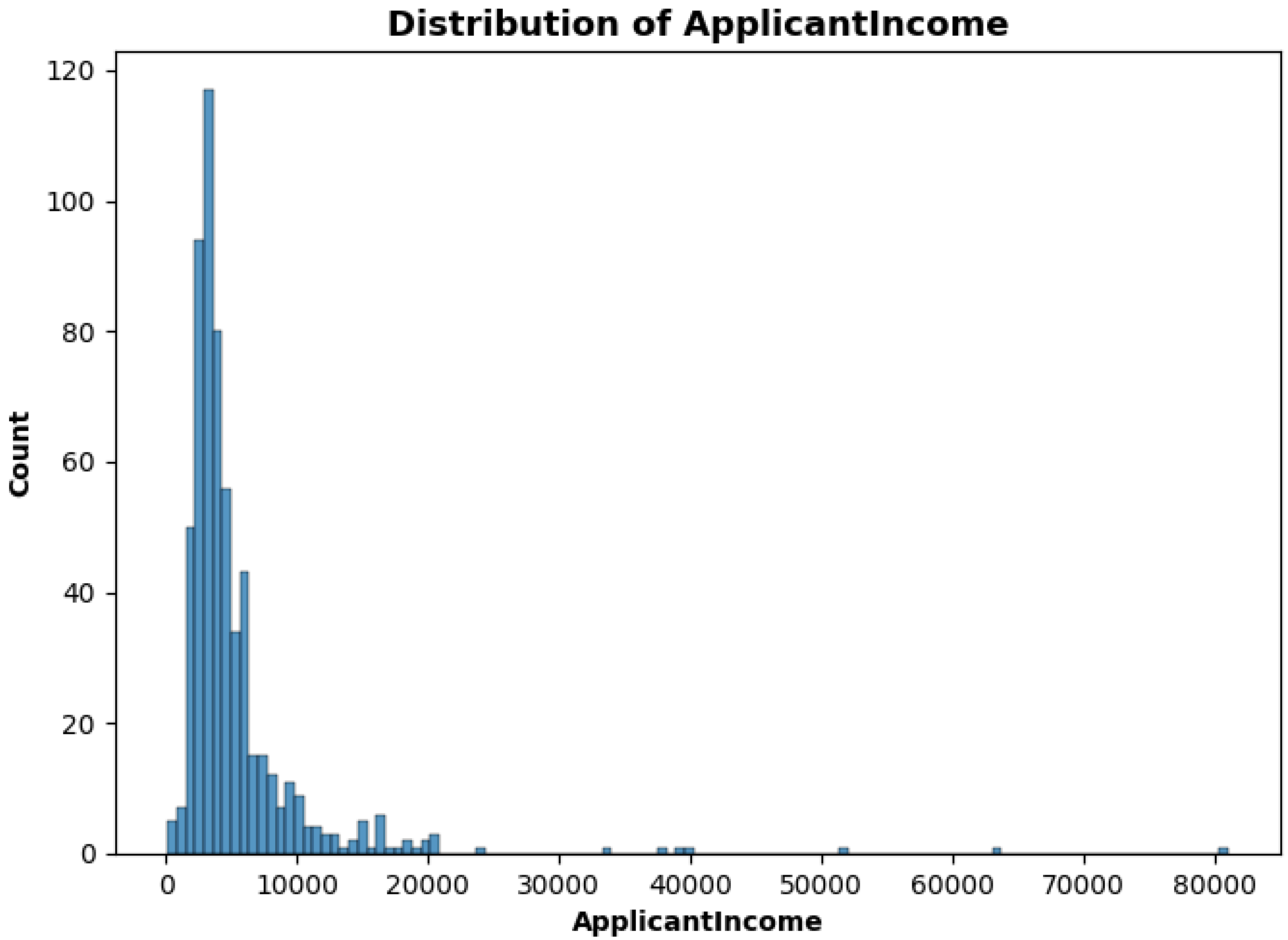
	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
Loan_ID				
2284	3902	1666	109	333
2287	1500	1800	103	333
2288	2889	0	45	180
2296	2755	0	65	300
2297	2500	20000	103	333

In [ ]:

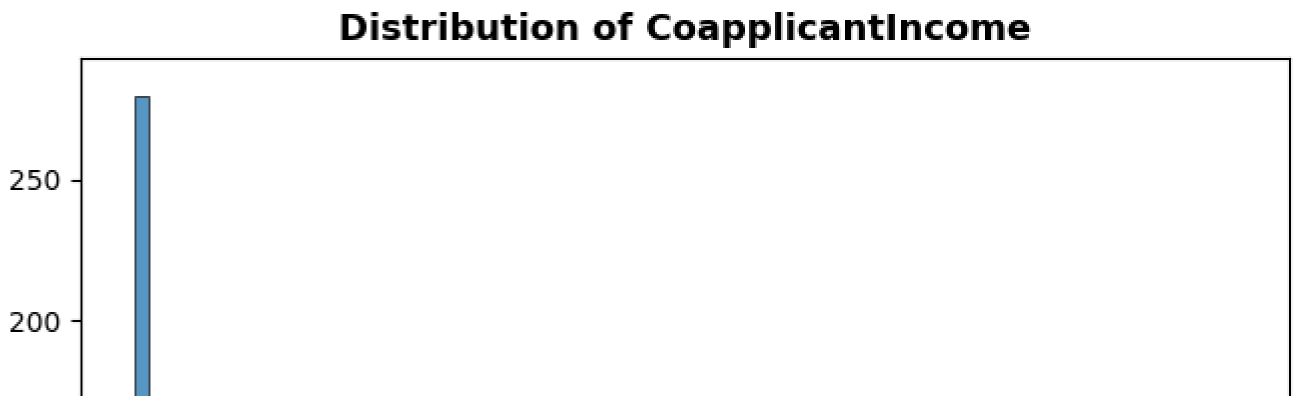
```
def plot_continuous_histograms(df):
    """
    Plots histograms for all continuous columns in a DataFrame.

    Args:
        df (pandas.DataFrame): The input DataFrame.

    Returns:
        None
    """
    numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns
    nrows = len(numeric_cols)
    ncols = 1
    fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(7, 5 * nrows), sharey=False)

    if nrows == 1:
        axes = [axes]  # Make axes a list for consistent indexing

    for i, col in enumerate(numeric_cols):
        ax = axes[i]
        sns.histplot(data=df, x=col, ax=ax)
        ax.set_title(f'Distribution of {col}', fontdict={'fontweight': 'bold', 'fontsize': 13})
        ax.set_xlabel(col, fontweight='bold')
        ax.set_ylabel('Count', fontweight='bold')
```

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

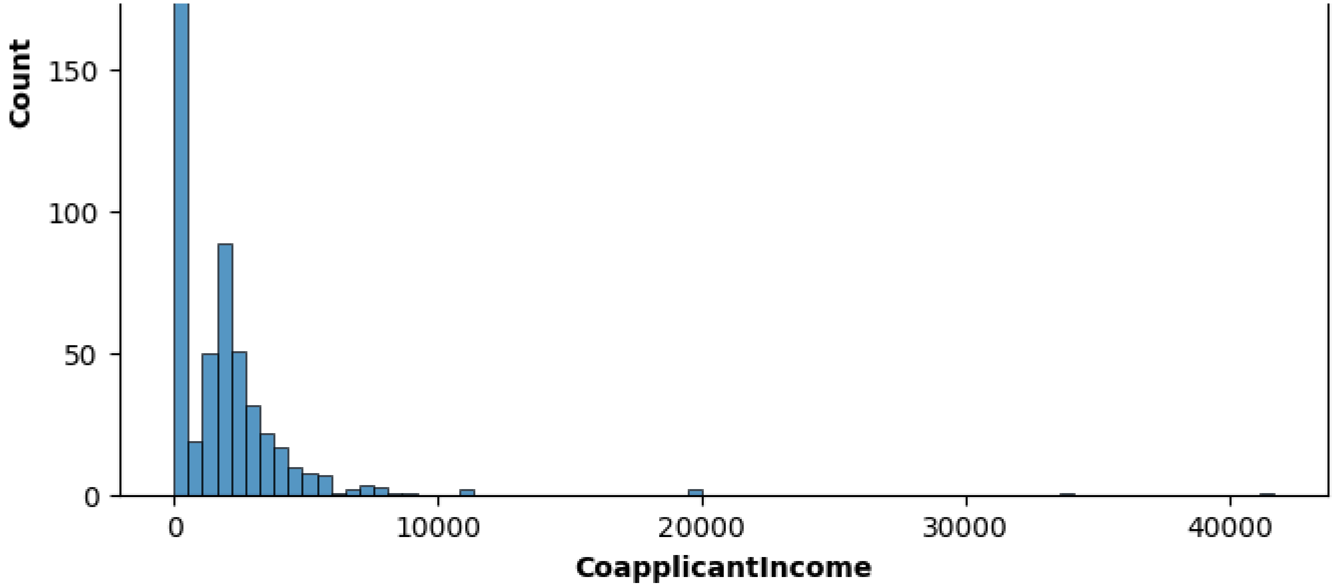
In [ ]:
```
plot_continuous_histograms(loans)
```

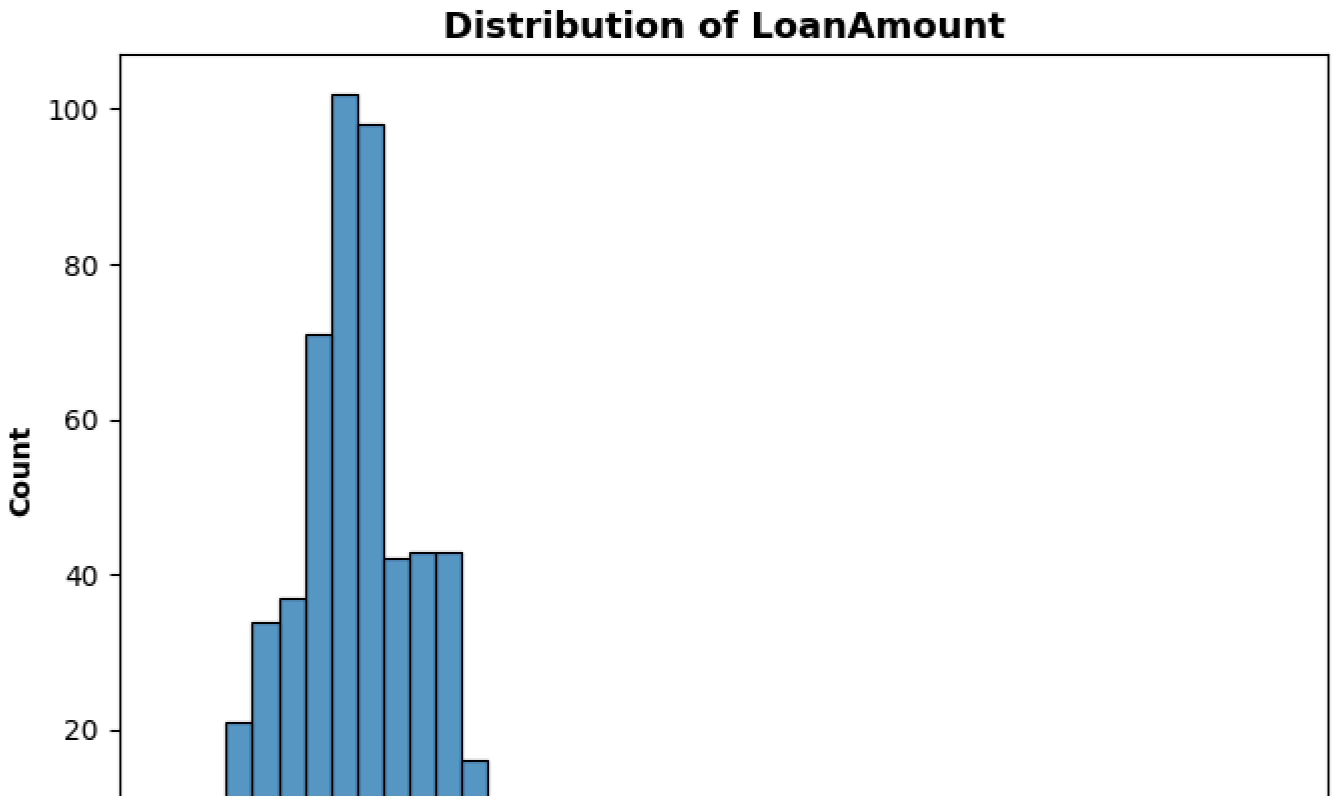


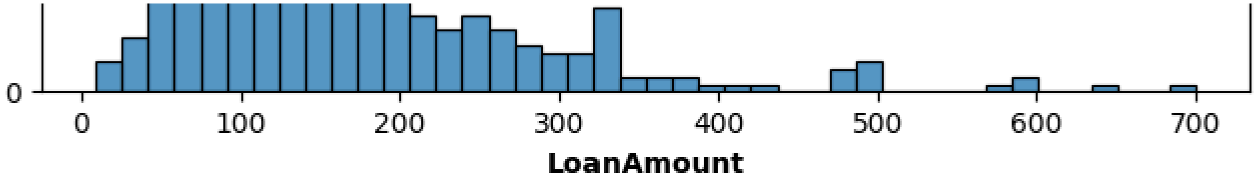
Distribution of ApplicantIncome
120
100
80
60
40
20
0
Count
0
10000
20000
30000
40000
50000
60000
70000
80000
ApplicantIncome

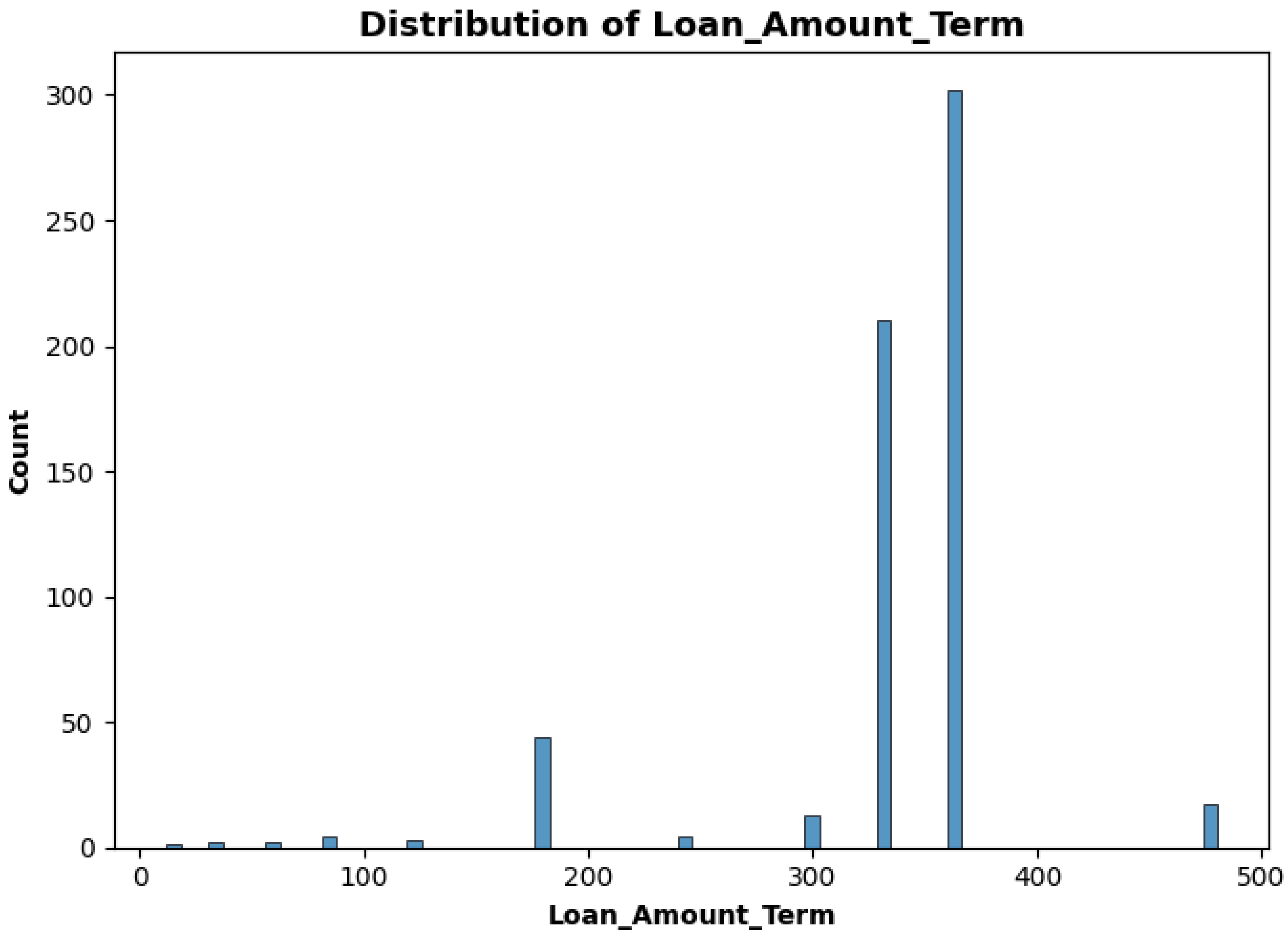


Distribution of CoapplicantIncome
250
200


Count
150
100
50
0
0
10000
20000
30000
40000
CoapplicantIncome

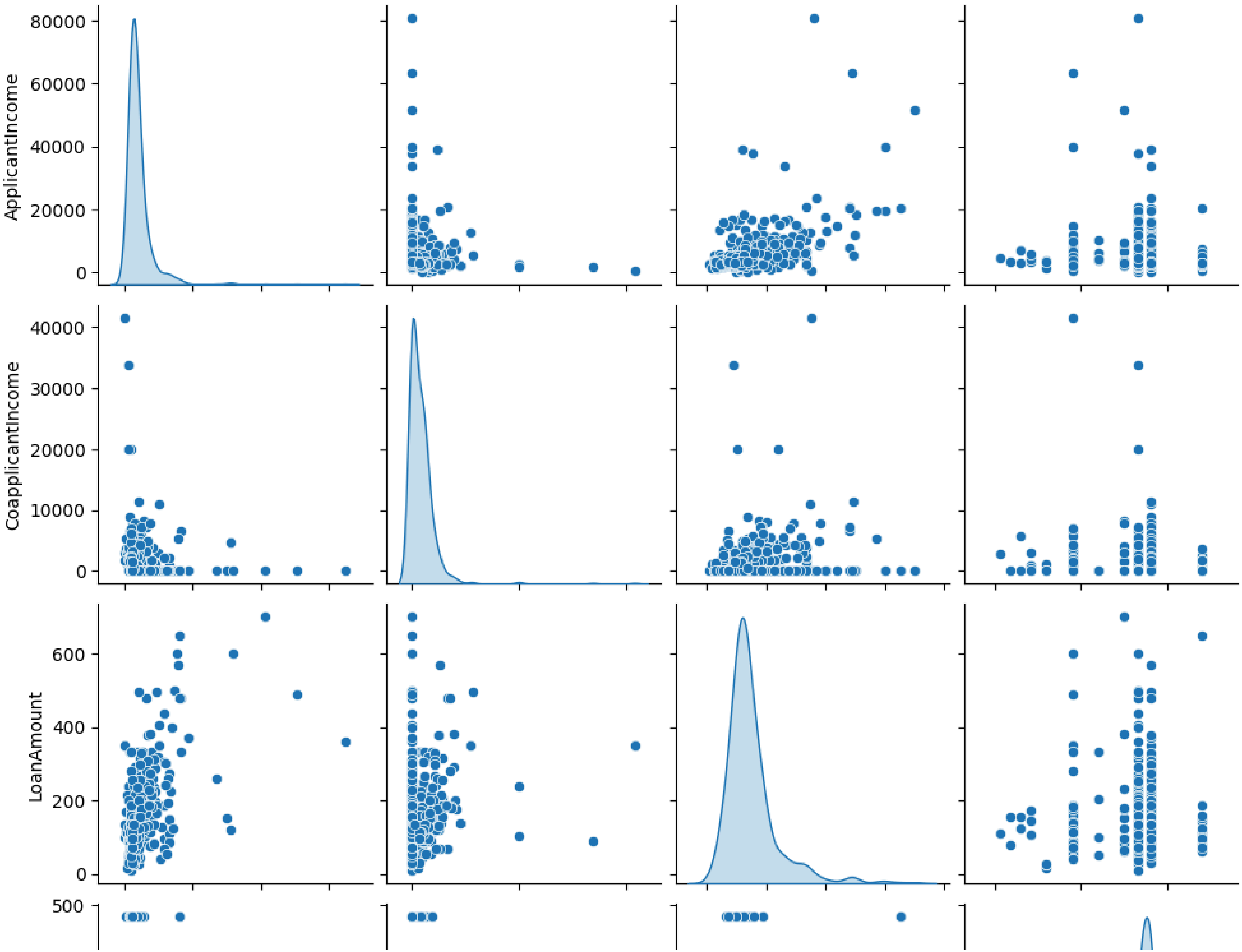

Distribution of LoanAmount
Count
100
80
60
40
20



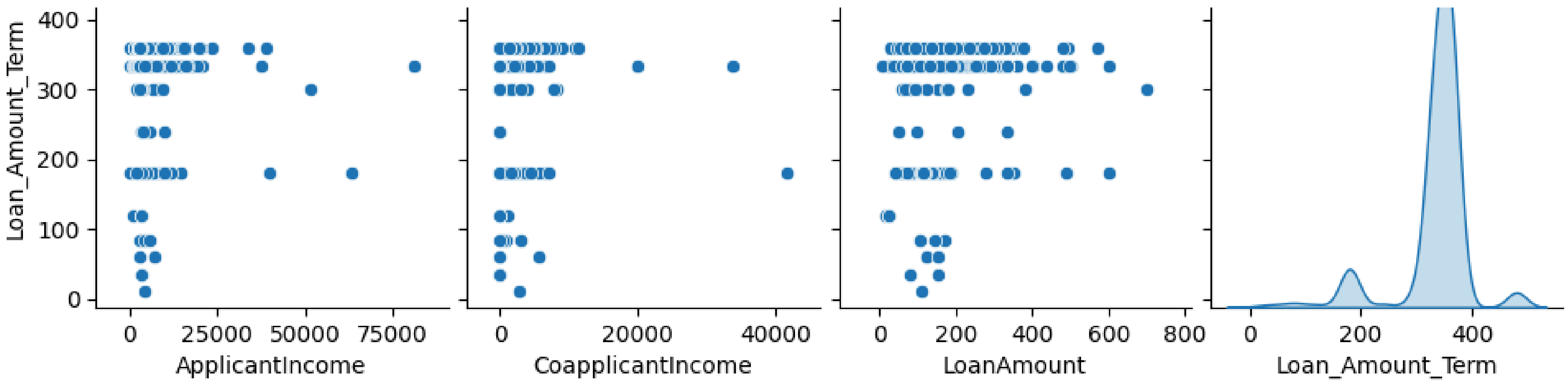


Bivariate Analysis

In [ ]:
```
sns.pairplot(loans, diag_kind='kde')
plt.show()
```



ApplicantIncome
CoapplicantIncome
LoanAmount
80000
60000
40000
20000
0
40000
30000
20000
10000
0
600
400
200
0
500



There seem to be a positive correlation between loan amount and an applicant's income.

Required analyses

In [ ]:
```
# Descriptive analysis
print("\nTotal amount loaned by AFS: ", loans['LoanAmount'].sum())
print("Average amount loaned: ", round(loans['LoanAmount'].mean(),2))
print("Average loan term: ", loans['Loan_Amount_Term'].mean())
```

```
Total amount loaned by AFS:  89659
Average amount loaned:  148.94
Average loan term:  333.03986710963454
```

In [ ]:
```
# Applicant counts by approval status and gender, (Approved=Y , Rejected=N)
print("Applicant counts by approval status and gender:")
loans.groupby(['Loan_Status', 'Gender'], observed=True).size().unstack(fill_value=0)
```

```
Applicant counts by approval status and gender:
```

Out[ ]:

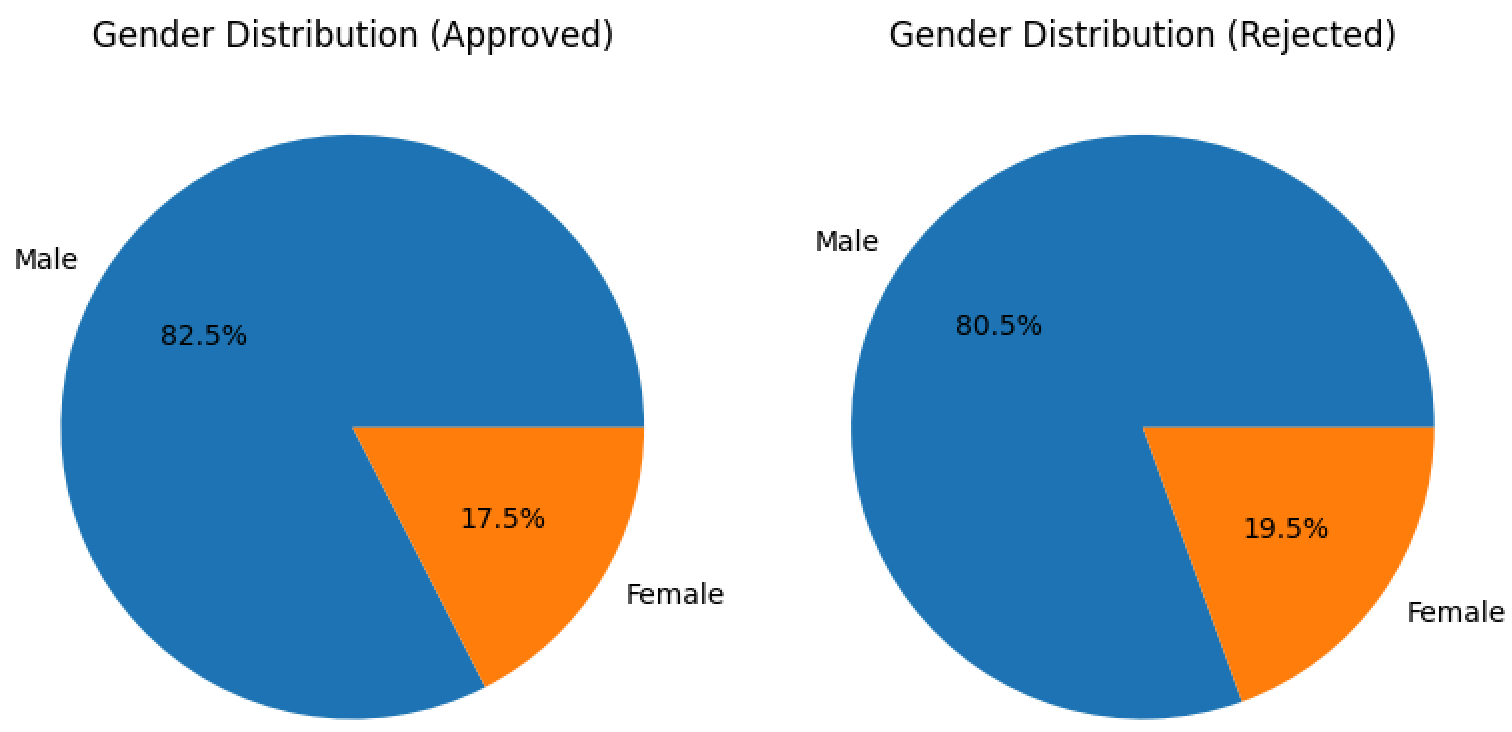
Gender	Female	Male
Loan_Status		
N	37	153
Y	72	340

In [ ]:

```
# Gender distribution (Approved and Rejected)
gender_approved = loans[loans['Loan_Status'] == 'Y']['Gender'].value_counts()
gender_rejected = loans[loans['Loan_Status'] == 'N']['Gender'].value_counts()
```

In [ ]:

```
# Distribution of gender for approved loan applicants
plt.figure(figsize=(8, 6))
plt.subplot(1, 2, 1)
plt.pie(gender_approved, labels=gender_approved.index, autopct='%1.1f%%')
plt.title('Gender Distribution (Approved)')

plt.subplot(1, 2, 2)
plt.pie(gender_rejected, labels=gender_rejected.index, autopct='%1.1f%%')
plt.title('Gender Distribution (Rejected)')
plt.tight_layout()

plt.show()
```



In [ ]:
```
# Applicant counts by approval status and gender
print("Applicant counts by approval status and gender:")
loans.groupby(['Loan_Status', 'Gender'], observed=True).size().unstack(fill_value=0)
```

```
Applicant counts by approval status and gender:
```

Out[ ]:

Gender	Female	Male
Loan_Status		
N	37	153
Y	72	340

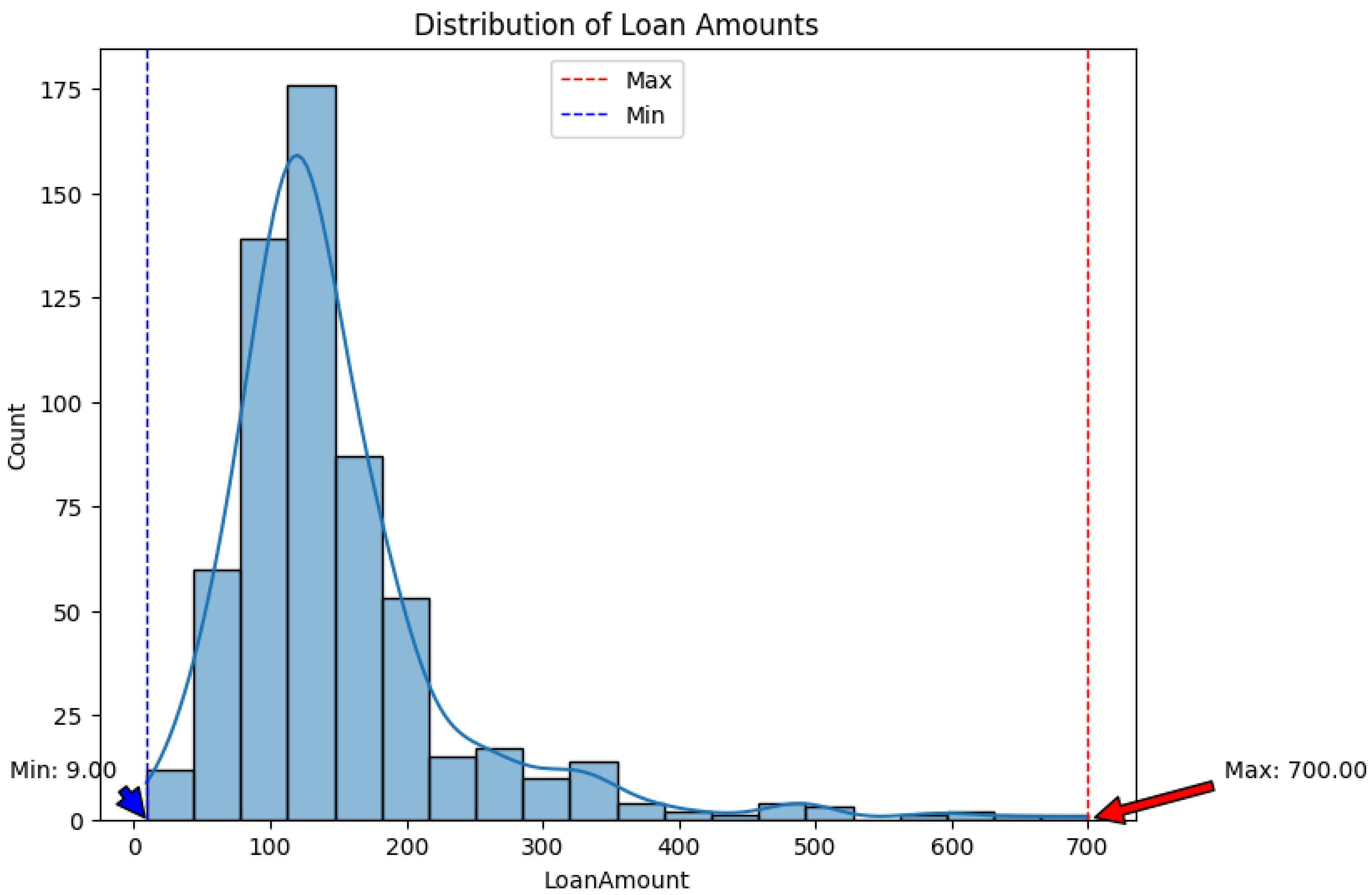
In [ ]:
```
# Maximum and minimum loan amounts
max_loan = loans['LoanAmount'].max()
```

```python
min_loan = loans['LoanAmount'].min()

print(f"Maximum loan amount: {max_loan}")
print(f"Minimum loan amount: {min_loan}")
```

```
Maximum loan amount: 700
Minimum loan amount: 9
```

In [ ]:

```python
plt.figure(figsize=(8, 6))
sns.histplot(loans['LoanAmount'], bins=20, kde=True)
plt.title('Distribution of Loan Amounts')

# Annotate maximum and minimum values
plt.annotate('Max: {:.2f}'.format(max_loan), xy=(max_loan, 0), xytext=(max_loan + 100, 10),
             arrowprops=dict(facecolor='red', shrink=0.05))
plt.annotate('Min: {:.2f}'.format(min_loan), xy=(min_loan, 0), xytext=(min_loan - 100, 10),
             arrowprops=dict(facecolor='blue', shrink=0.05))

# Draw vertical lines for max and min
plt.axvline(x=max_loan, color='red', linestyle='dashed', linewidth=1, label='Max')
plt.axvline(x=min_loan, color='blue', linestyle='dashed', linewidth=1, label='Min')
plt.legend()

plt.show()
```

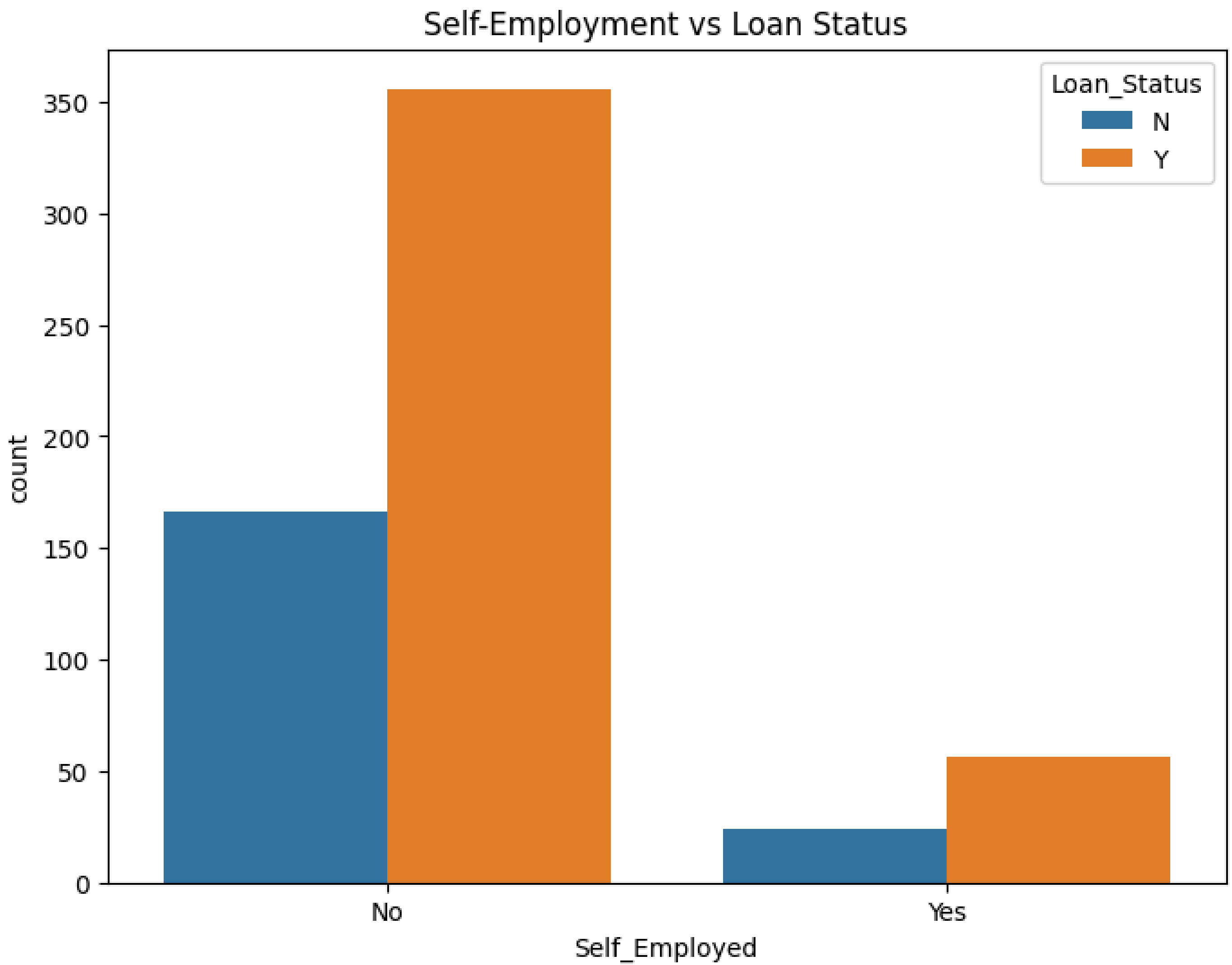


In [ ]:

```
# Percentage of self-employed with approved loans
self_employed_approved = loans[(loans['Self_Employed'] == 'Yes') & (loans['Loan_Status'] == 'Y')].shape[0]
total_approved = loans[loans['Loan_Status'] == 'Y'].shape[0]
self_employed_approved_percentage = (self_employed_approved / total_approved) * 100

print(f"Percentage of self-employed applicants with approved loans: {self_employed_approved_percentage:.2f}%")
```

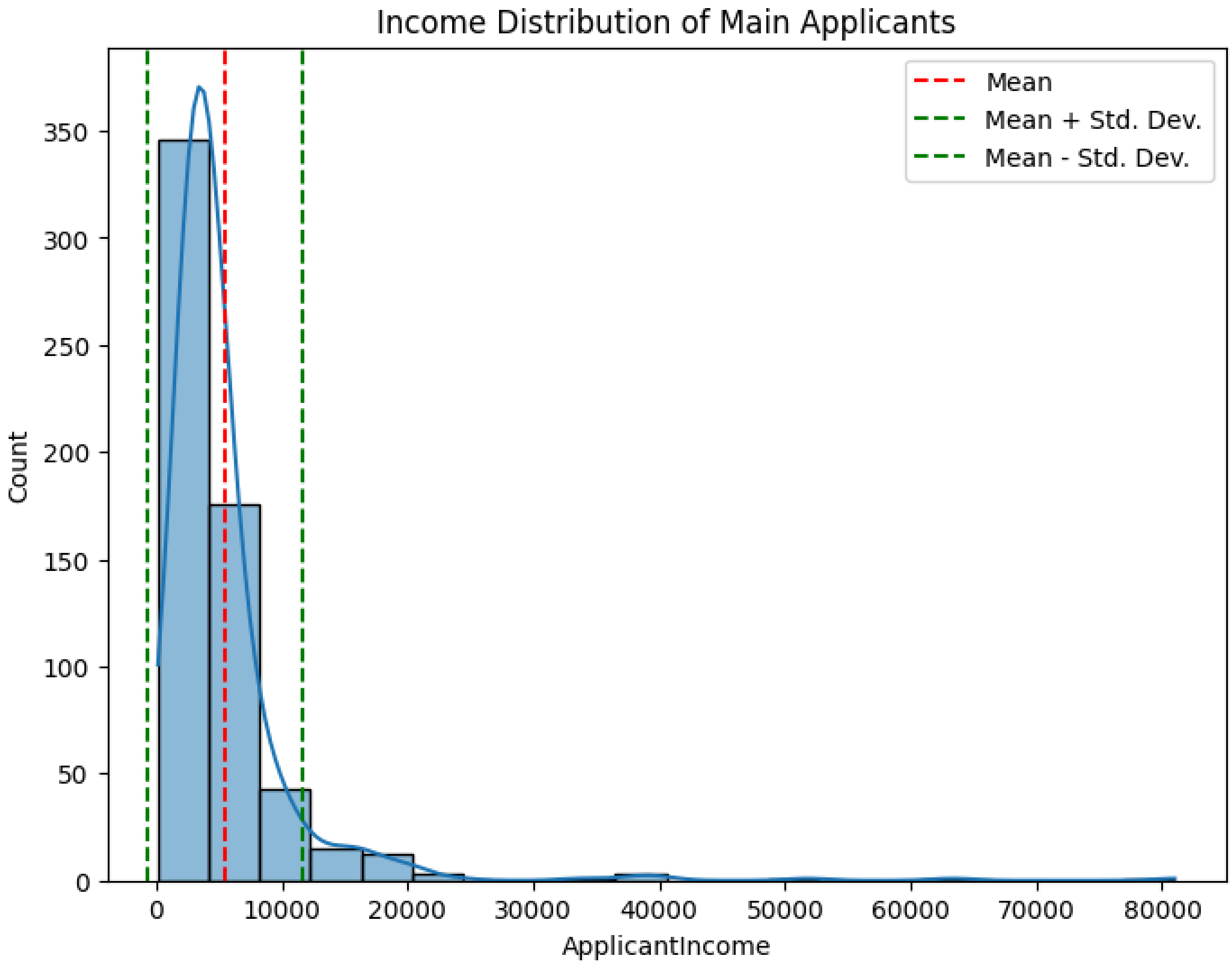
```
plt.figure(figsize=(8, 6))
sns.countplot(x='Self_Employed', hue='Loan_Status', data=loans)
plt.title('Self-Employment vs Loan Status')
plt.show()
```

```
Percentage of self-employed applicants with approved loans: 13.59%
```




```
In [ ]: # Income distribution of  applicants
        plt.figure(figsize=(8, 6))
```

```
sns.histplot(loans['ApplicantIncome'], bins=20, kde=True)
plt.title('Income Distribution of Main Applicants')
plt.axvline(loans['ApplicantIncome'].mean(), color='r', linestyle='--', label='Mean')
plt.axvline(loans['ApplicantIncome'].mean() + loans['ApplicantIncome'].std(), color='g', linestyle='--', label='Mean + Std. De
plt.axvline(loans['ApplicantIncome'].mean() - loans['ApplicantIncome'].std(), color='g', linestyle='--', label='Mean - Std. De
plt.legend()
plt.show()

print(f"\nMean income of main applicants: {loans['ApplicantIncome'].mean():.2f}")
print(f"Standard deviation of income: {loans['ApplicantIncome'].std():.2f}")
```




```
Mean income of main applicants: 5422.19
Standard deviation of income: 6159.34
```

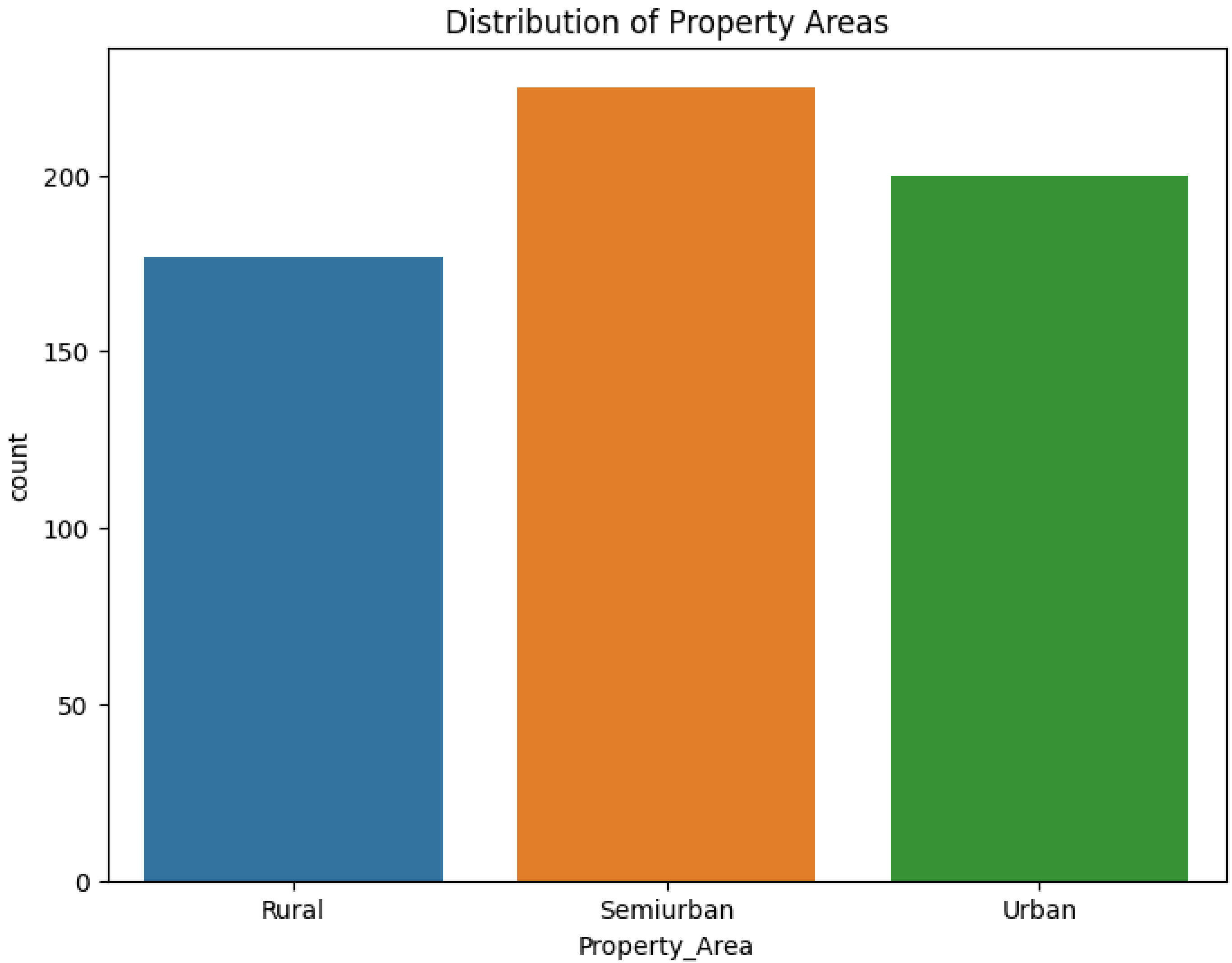
In [ ]:
```
# Top 10 applicants by loan amount
loans.sort_values(by='LoanAmount', ascending=False).head(10)
```

Out[ ]:

Loan_ID	Gender	Married	Dependents	Graduate	Self_Employed	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term
1585	Male	Married	3	Yes	No	51763	0	700	300
1469	Male	Single	0	Yes	Yes	20166	0	650	480
1536	Male	Married	3	Yes	No	39999	0	600	180
2813	Female	Married	1	Yes	Yes	19484	0	600	333
2191	Male	Married	0	Yes	No	19730	5266	570	360
2547	Male	Married	1	Yes	No	18333	0	500	333
2959	Female	Married	1	Yes	No	12000	0	496	333
1610	Male	Married	3	Yes	No	5516	11300	495	360
2101	Male	Married	0	Yes	No	63337	0	490	180
2693	Male	Married	2	Yes	Yes	7948	7166	480	333

In [ ]:

```
# Property distribution of loan applicants
plt.figure(figsize=(8, 6))
sns.countplot(x='Property_Area', data=loans, hue='Property_Area')
plt.title('Distribution of Property Areas')
plt.show()
```


Distribution of Property Areas
count
0
50
100
150
200
Rural
Semiurban
Urban
Property_Area