Lab1

Part1

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
#Loading the data files
heartTraindf = pd.read_csv("heart_dataset.csv")
```

In [3]:
```
# Display the first few rows of the training dataset
print("\nHeart Train Dataset Preview:")

# Convert 'famhist' column to binary values
heartTraindf['famhist'] = heartTraindf['famhist'].replace({'Present': 1, 'Absent': 0})
heartTraindf.head()
```

```
Heart Train Dataset Preview:
```

Out[3]:

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age	chd
0	134.0	13.60	3.50	27.78	1.0	60.0	25.99	57.34	49.0	1.0
1	132.0	6.20	6.47	36.21	1.0	62.0	30.77	14.14	45.0	0.0
2	142.0	4.05	3.38	16.20	0.0	NaN	20.81	2.62	38.0	0.0
3	114.0	4.08	4.59	14.60	1.0	62.0	23.11	6.72	58.0	NaN
4	114.0	NaN	3.83	19.40	1.0	49.0	24.86	2.49	NaN	NaN

1. Identify the dataset columns into nominal, categorical, continuous, etc. categories. (10pts)

Data Column Categories

- **Nominal/Categorical Data** : Nominal data consists of categories with no inherent order or ranking. Categorical data can be text or numbers, but the numbers are used as labels rather than for numerical operations.
- **Binary Data** consists of two categories or values (0 and 1), representing yes/no, true/false, or presence/absence.
- **Continuous Data** represents numerical values that can take any real number within a range. It can have decimal points and an infinite number of possible values.

In [4]:
```
# 1. Identify dataset columns into nominal, categorical, continuous, etc. categories.
# Categorize columns based on their data type

nominal_categorical_columns = ['famhist']
binary_categorical_columns = ['chd']
discrete_columns = ['sbp','ldl','age']
continuous_columns = ['tobacco', 'adiposity', 'typea', 'obesity', 'alcohol']
```

2. Find the number of null values for each column. (10pts)

In [5]:
```
null_values = heartTraindf.isnull()

# Count the number of null values for each column of Train data
null_count = null_values.sum()

print("\nNumber of Null Values for Each Column in Train Data:")

# Convert the Series to a DataFrame for better readability
null_count_df = pd.DataFrame({'Column Name': null_count.index, 'Null Count': null_count.
null_count_df
```

```
Number of Null Values for Each Column in Train Data:
```

Out[5]:

	Column Name	Null Count
0	sbp	28
1	tobacco	40
2	ldl	39
3	adiposity	40
4	famhist	45
5	typea	41
6	obesity	40
7	alcohol	40
8	age	35
9	chd	39

3. Descriptive Analysis (40pts / 4pts for each sub question)

1) Show the general descriptive statistics by using describe function

In [6]:
```
heartTraindf.describe()
```

Out[6]:

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age
count	384.000000	372.000000	373.000000	372.000000	367.000000	371.000000	372.000000	372.000000	377.000000
mean	139.216146	3.676425	4.569303	25.210753	0.449591	52.008086	25.763602	18.425134	42.453581
std	20.307368	4.568564	1.888691	7.760257	0.498132	9.822888	3.854265	25.971090	15.312649
min	101.000000	0.000000	0.980000	7.120000	0.000000	20.000000	17.890000	0.000000	15.000000
25%	124.000000	0.057500	3.240000	19.307500	0.000000	46.000000	22.835000	0.195000	30.000000
50%	136.000000	1.800000	4.220000	26.115000	0.000000	52.000000	25.675000	7.300000	45.000000
75%	148.500000	5.640000	5.470000	30.790000	1.000000	58.000000	28.167500	25.820000	57.000000
max	218.000000	27.400000	14.160000	42.490000	1.000000	73.000000	40.340000	145.290000	64.000000

2) Find the oldest person

In [7]:
```
oldest_person = heartTraindf['age'].max()
print("The Oldest Person's Age:", oldest_person)

# Filter the data with age == 64
heartTraindf[heartTraindf['age']==64]
```

```
The Oldest Person's Age: 64.0
```

Out[7]:

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age	chd
58	158.0	3.60	2.97	NaN	0.0	NaN	26.64	108.00	64.0	0.0
70	152.0	12.18	4.04	37.83	1.0	63.0	34.57	4.17	64.0	0.0
110	126.0	0.00	5.98	29.06	1.0	56.0	25.39	11.52	64.0	1.0
167	148.0	8.20	7.75	34.46	1.0	46.0	26.53	6.04	64.0	1.0
170	128.0	5.16	4.90	NaN	1.0	57.0	26.42	0.00	64.0	0.0
206	NaN	8.60	3.90	32.16	1.0	52.0	28.51	11.11	64.0	1.0
241	160.0	0.60	6.94	30.53	0.0	36.0	25.68	1.42	64.0	0.0
256	138.0	2.00	5.11	31.40	1.0	49.0	27.25	2.06	64.0	1.0
276	128.0	0.73	3.97	23.52	0.0	NaN	23.81	NaN	64.0	0.0
348	140.0	8.60	3.90	32.16	1.0	52.0	28.51	11.11	64.0	1.0
374	160.0	0.60	6.94	30.53	0.0	36.0	25.68	NaN	64.0	0.0
402	174.0	2.02	6.57	31.90	1.0	50.0	28.75	11.83	64.0	1.0

64 years old is the oldest age in this data and the table above shows the data of all the oldest people in heart-train data. There are total 12 people who are 64 years old.

3) Find the youngest person

In [8]:
```
youngest_person = heartTraindf['age'].min()
print("The Youngest Person's Age:", youngest_person)

# Filter the data with age == 15
heartTraindf[heartTraindf['age']==15]
```

```
The Youngest Person's Age: 15.0
```

Out[8]:

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age	chd
9	132.0	0.0	1.87	17.21	0.0	49.0	23.63	0.97	15.0	0.0
38	NaN	0.0	3.67	12.13	0.0	NaN	19.15	0.60	15.0	0.0

15 years old is the youngest age in this data and the table above shows the data of the youngest people in the dataframe. Total of 2 people.

4) Find the average age group

In [9]:
```
print("The Average age is : {:.2f}".format(heartTraindf['age'].mean()))
```

```
The Average age is : 42.45
```

In [10]:
```
heartTraindf.age.std()
```

Out[10]:
```
15.31264927550187
```

By calculating the summary statistic, we could get the value for **average** (42.686893) and **standard deviation** (15.129338) of age column.

5) Find median age

In [11]:
```
median_age = heartTraindf['age'].median()
print("Median Age:", median_age)
```

```
Median Age: 45.0
```

6) Find the relationship between deaths and ages(the class column is your prediction variable)

In [12]:
```
# Group the data by 'age' and calculate the mean of 'chd'
age_chd_relationship = heartTraindf.groupby('age')['chd'].mean().reset_index()
age_chd_relationship
```

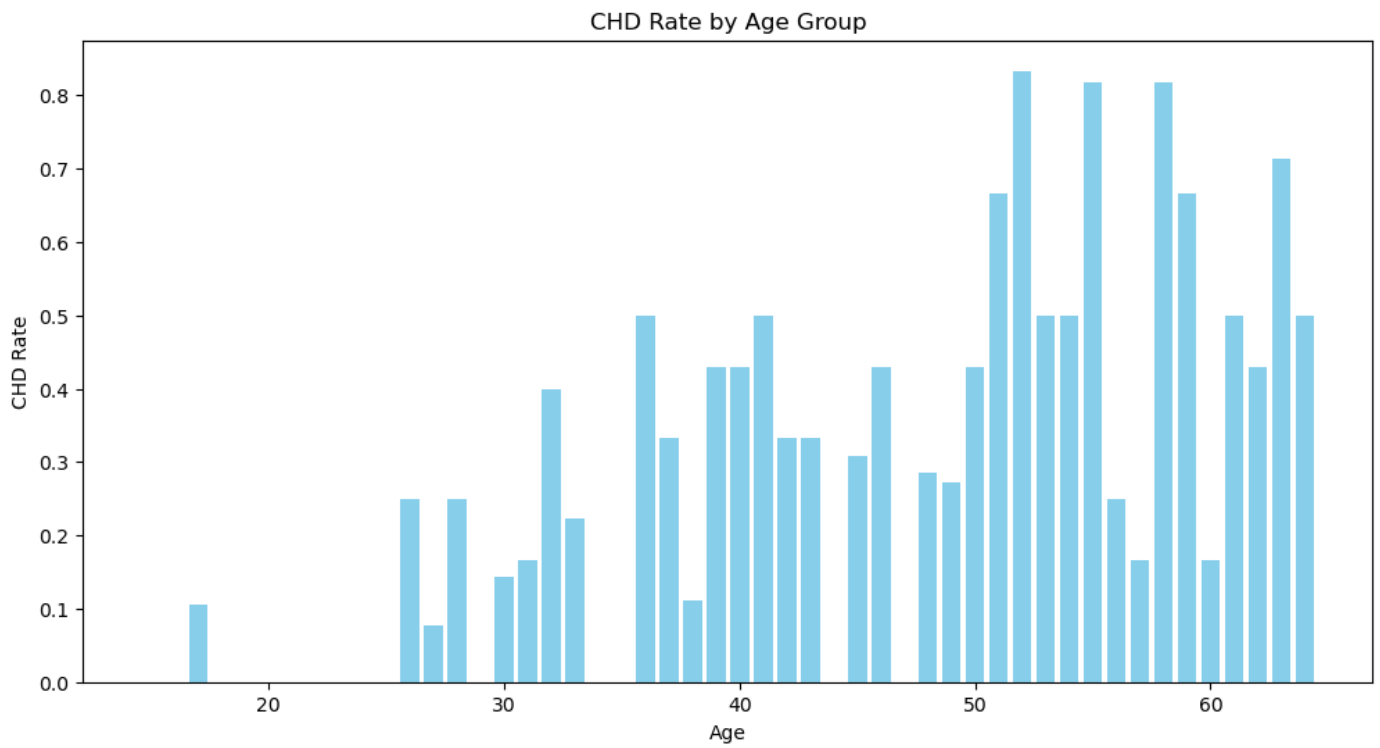
Out[12]:

	age	chd
0	15.0	0.000000
1	16.0	0.000000
2	17.0	0.105263
3	18.0	0.000000
4	19.0	0.000000
5	20.0	0.000000
6	21.0	NaN
7	23.0	0.000000
8	24.0	0.000000
9	25.0	0.000000
10	26.0	0.250000
11	27.0	0.076923
12	28.0	0.250000
13	29.0	0.000000
14	30.0	0.142857
15	31.0	0.166667
16	32.0	0.400000
17	33.0	0.222222
18	34.0	0.000000
19	35.0	0.000000
20	36.0	0.500000
21	37.0	0.333333
22	38.0	0.111111
23	39.0	0.428571

24	40.0	0.428571
25	41.0	0.500000
26	42.0	0.333333
27	43.0	0.333333
28	44.0	0.000000
29	45.0	0.307692
30	46.0	0.428571
31	47.0	0.000000
32	48.0	0.285714
33	49.0	0.272727
34	50.0	0.428571
35	51.0	0.666667
36	52.0	0.833333
37	53.0	0.500000
38	54.0	0.500000
39	55.0	0.818182
40	56.0	0.250000
41	57.0	0.166667
42	58.0	0.818182
43	59.0	0.666667
44	60.0	0.166667
45	61.0	0.500000
46	62.0	0.428571
47	63.0	0.714286
48	64.0	0.500000

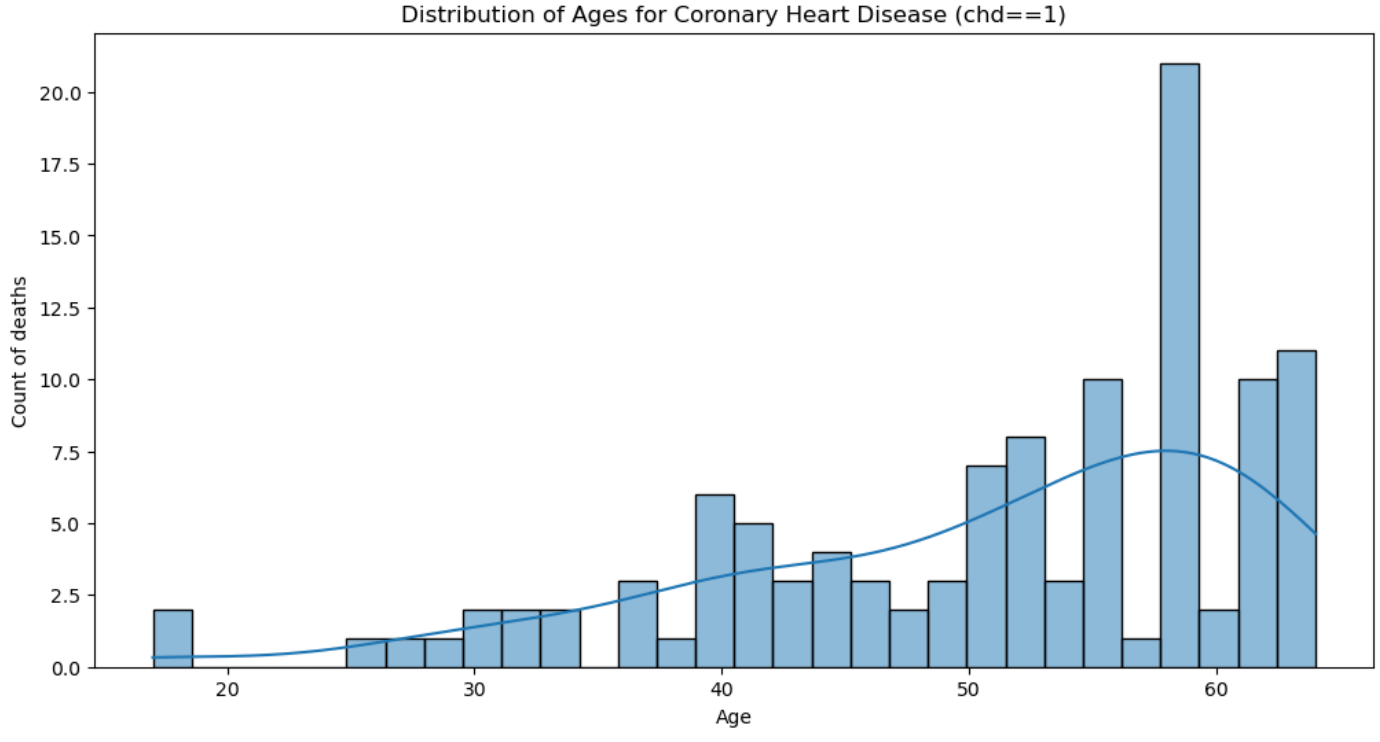
Now that we have the **age_chd_relationship**, we can create a bar chart that shows how the presence of coronary heart disease (chd) is distributed among different age groups

In [13]:
```
plt.figure(figsize=(12, 6))
plt.bar(age_chd_relationship['age'], age_chd_relationship['chd'], color='skyblue')
plt.xlabel('Age')
plt.ylabel('CHD Rate')
plt.title('CHD Rate by Age Group')
plt.show()
```



This bar graph shows that the presence of coronary heart disease is highest in higher age groups.

In [14]:
```
#another way of calculating relationship between deaths and ages is by calculating numbe

chd_1_data = heartTraindf[heartTraindf['chd'] == 1]

# Plotting the distribution of ages for chd==1
plt.figure(figsize=(12, 6))
sns.histplot(data=chd_1_data, x='age', bins=30, kde=True)
plt.title('Distribution of Ages for Coronary Heart Disease (chd==1)')
plt.xlabel('Age')
plt.ylabel('Count of deaths')
plt.show()
```



To find a relationship between deaths and ages, we first filtered data that are `chd == 1`, which indicates

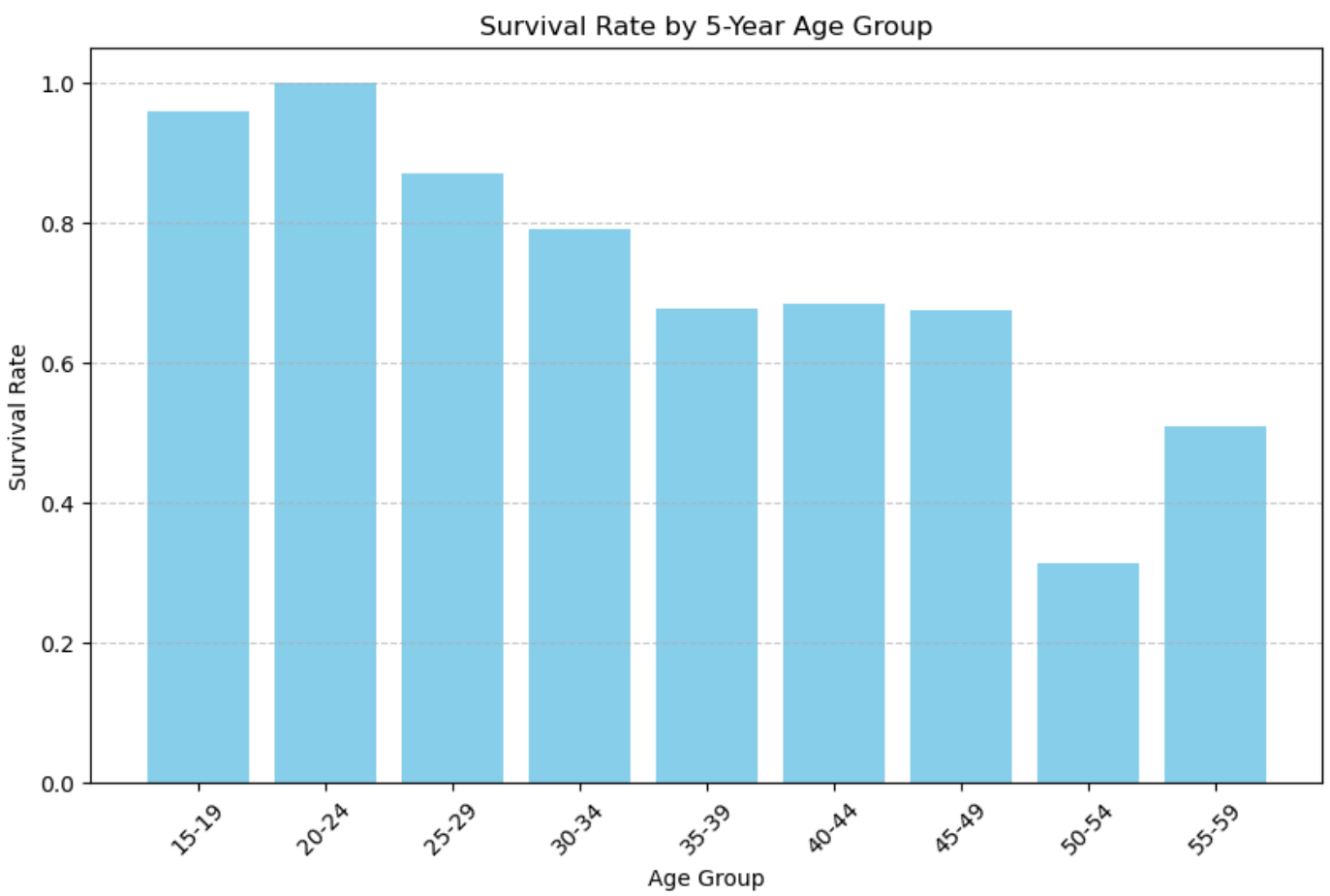
the dead. Then, we drew a histogram graph to get the insight for the distribution between number of deaths and ages. By observing the histogram above, we could infer that as the age gets older, the number of deaths also increase. The left skewed line shows the number of deaths are more focused on the age group of between 55 years old to 60 years old.

7) Find the age groups whose survival rate is the largest:

In [15]:
```
# Create a new DataFrame to preserve the original 'heartTraindf' data
heartTraindf_age_groups = heartTraindf.copy()

# Define the age bins and labels for grouping
age_bins = range(15, 65, 5)
age_labels = [f"{start}-{start+4}" for start in age_bins[:-1]]

# Add the 'age_group' column to the new DataFrame
heartTraindf_age_groups['age_group'] = pd.cut(heartTraindf_age_groups['age'], bins=age_b

# Calculate the survival rate for each age group
survival_rate_by_age_group = 1 - heartTraindf_age_groups.groupby('age_group')['chd'].mea

# Find the age group with the largest survival rate
largest_survival_age_group = survival_rate_by_age_group.idxmax()
print("Age group with the largest survival rate:", largest_survival_age_group)
```

```
Age group with the largest survival rate: 20-24
```

In [16]:
```
# creating a bar plot to visualize the survival rates by 5-year age groups

# Plot survival rates by 5-year age group
plt.figure(figsize=(10, 6))
plt.bar(survival_rate_by_age_group.index, survival_rate_by_age_group.values, color='skyb
plt.xlabel("Age Group")
plt.ylabel("Survival Rate")
plt.title("Survival Rate by 5-Year Age Group")
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



The above bar plot shows survival rates across 5-year age groups and we can see that survival rate is largest for age group 20-24.

8) relationship between 'famhist' (Family History) and 'chd' (Coronary Heart Disease)

In [17]:
```
# Calculate the mean of 'chd' for different values in 'famhist' (Family History)
relationship_famhist = heartTraindf.groupby('famhist')['chd'].mean().reset_index()

# Display the relationships
print("\nRelationship between 'famhist' and 'chd':\n")
print(relationship_famhist)
```

```
Relationship between 'famhist' and 'chd':

   famhist       chd
0      0.0  0.243094
1      1.0  0.443709
```

The relationship between 'famhist' (Family History) and 'chd' (Coronary Heart Disease) can provide insights into how family history might influence the likelihood of developing coronary heart disease. In the provided data:

- When 'famhist' is 'Absent,' the mean 'chd' value is approximately 0.251.
- When 'famhist' is 'Present,' the mean 'chd' value is approximately 0.445.

From this relationship:

1. Higher Risk with Family History: The mean 'chd' value is higher (approximately 0.445) when 'famhist' is 'Present.' This suggests that individuals with a family history of coronary heart disease are more likely to

develop the condition compared to those without a family history.

2. Lower Risk without Family History: When 'famhist' is 'Absent,' the mean 'chd' value is lower (approximately 0.251). This indicates that individuals without a family history of coronary heart disease have a lower risk of developing the condition.

Based on this relationship, we can make the following predictions:

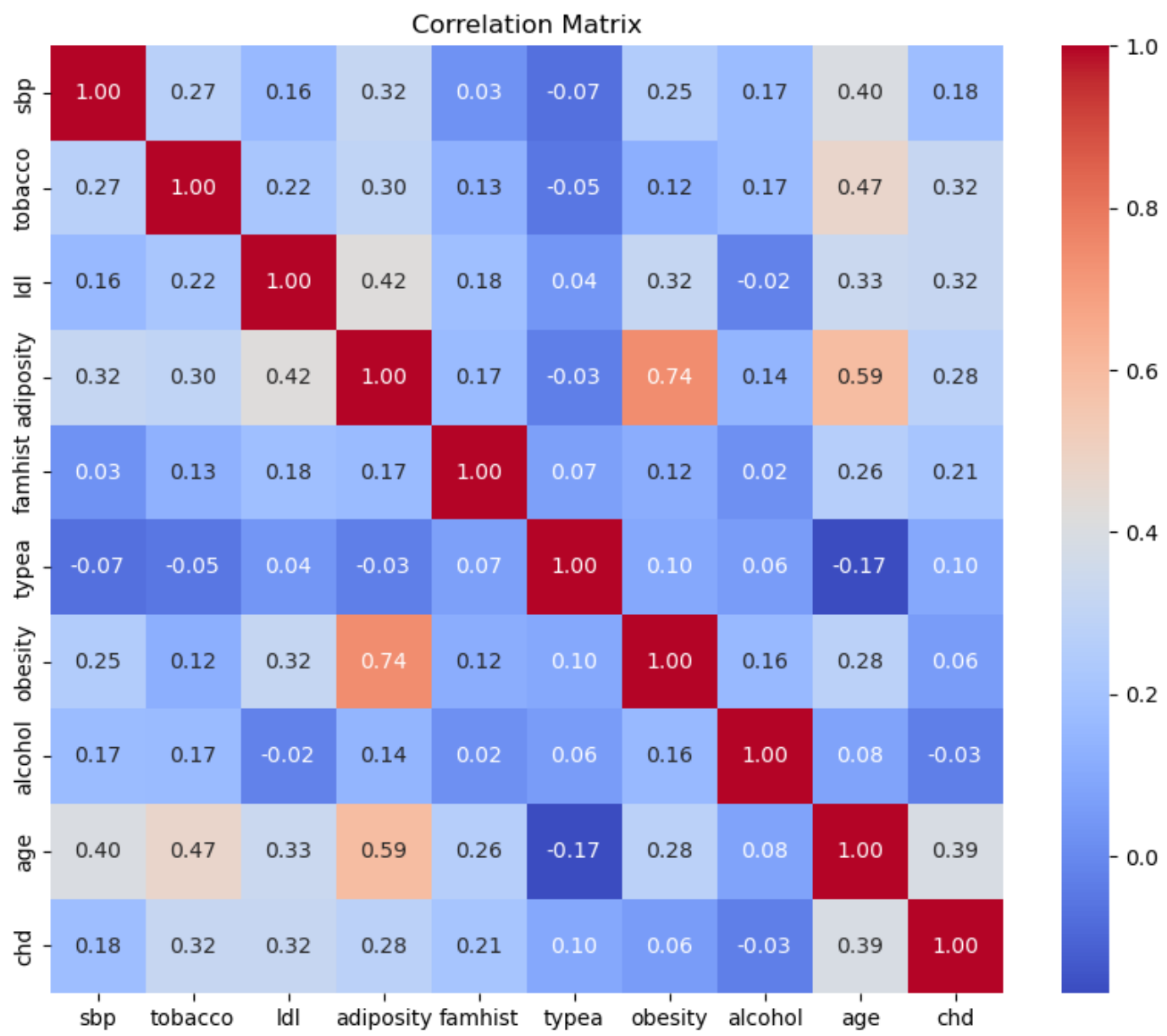
- Family history may be a significant risk factor for coronary heart disease.
- Individuals with a family history of the disease may need to be more vigilant in terms of prevention and monitoring.
- Lifestyle and healthcare interventions might be more critical for individuals with a family history of CHD to reduce their risk.

These insights can be valuable for risk assessment, healthcare decision-making, and designing preventive strategies for individuals with and without a family history of coronary heart disease.

9) Get more visuals on data distributions:

- i. Plot Correlation Matrix

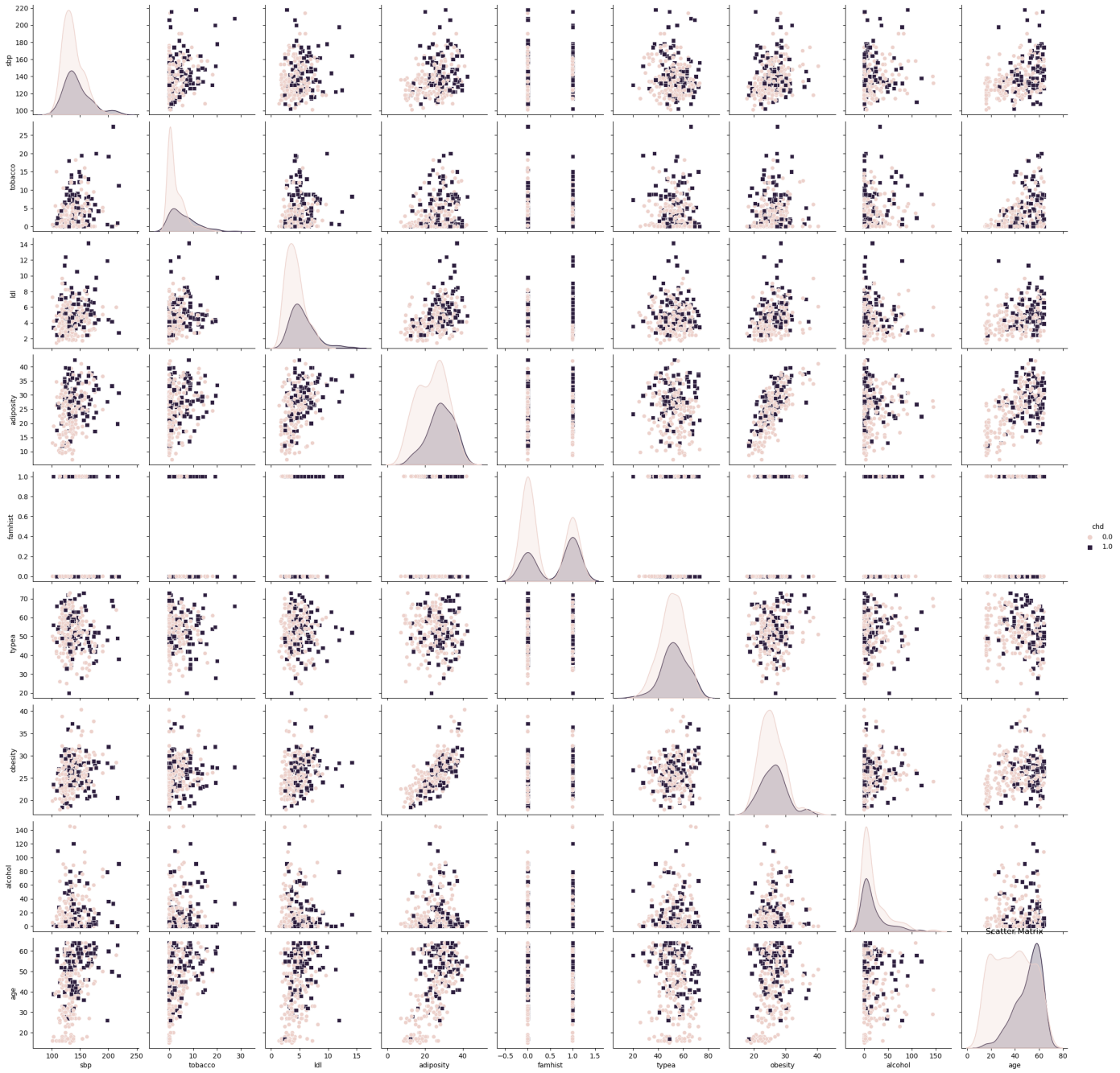
In [18]:
```
# Select only numeric columns for correlation analysis
numeric_columns = heartTraindf.select_dtypes(include=[np.number])

# Calculate the correlation matrix
correlation_matrix = numeric_columns.corr()

# Plot the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Matrix")
plt.show()
```



Using the numeric columns of our dataset, we calculated the correlation matrix (correlation_matrix) to examine the relationships between the variables. Now to visualize this correlation, we generated a heatmap using Seaborn's heatmap function, displaying the correlation matrix as a color-coded grid. The values are annotated on the heatmap in a 'coolwarm' color map format to visualize the strength and direction of relationships among the numeric variables. Warmer colors (like red) represent positive correlations, while cooler colors (like blue or green) represent negative correlations here.

- ii. Plot Scatter Matrix

In [19]:
```
sns.pairplot(heartTraindf, hue='chd', markers=["o", "s"])
plt.title("Scatter Matrix")
plt.show()
```



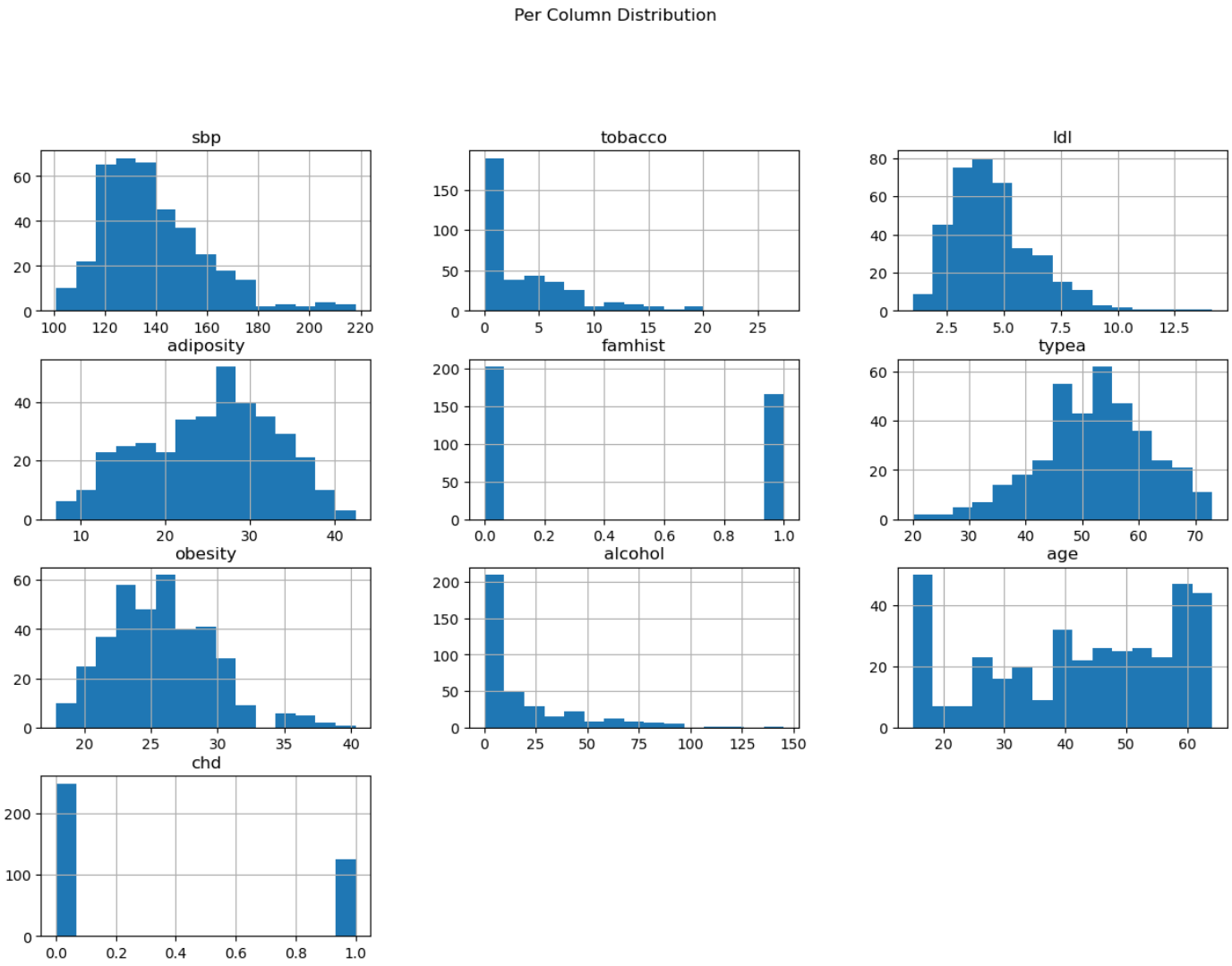
Here, we are using pairplot(). The sns.pairplot(heartTraindf, hue='chd', markers=["o", "s"]) command generates a grid of plots, consisting of histograms along the diagonal and scatter plots elsewhere. And then we color-code the scatter plots based on the 'chd' variable, distinguishing between its categories (chd = 0 & 1) using markers: circles ('o') and squares ('s').

Analysis: Histograms: The diagonal histograms illustrate the distribution of each individual variable. These histograms represent the frequency or density of values within each feature. Scatter Plots: The off-diagonal scatter plots visualize relationships between pairs of variables. The two distinct colors (orange and blue) in the scatter plots denote different 'chd' categories, allowing observation of how these categories relate across different variable pairings. This visualization offers insights into both the **distributions of individual features and potential relationships between variables**, specifically regarding **the influence of the 'chd' variable on different pairs of features**.

- iii. Plot Per Column Distribution

In [20]:
```
# Select only numeric columns for correlation analysis
numeric_columns = heartTraindf.select_dtypes(include=[np.number])
```

```
heartTraindf[numeric_columns.columns].hist(bins=15, figsize=(15, 10))
plt.suptitle("Per Column Distribution", y=1.02)
plt.show()
```



Variable Distributions: The histograms display the distributions of various numerical features within the dataset.

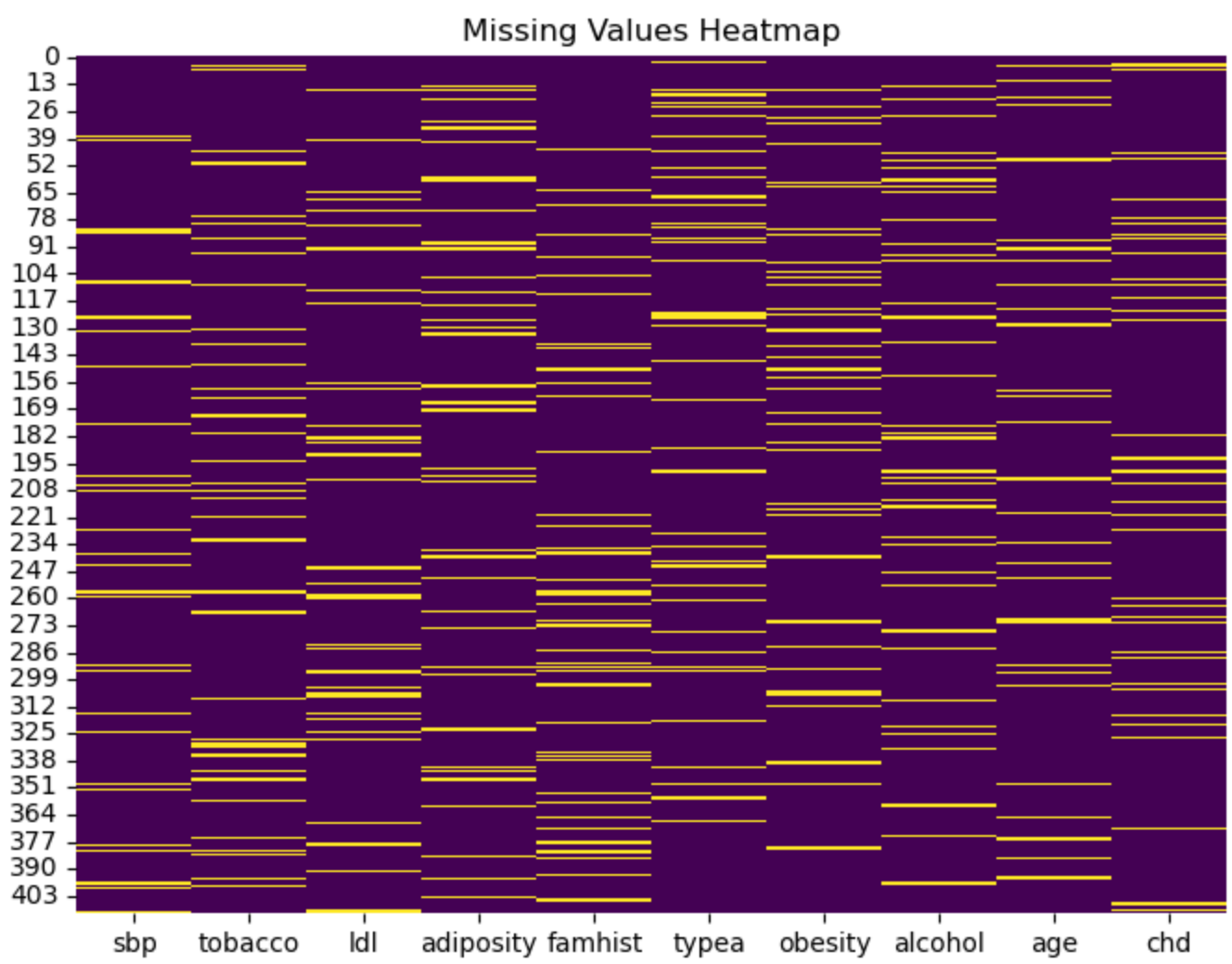
Observations: The histograms illustrate different characteristics:

- 'tobacco', 'alcohol', and 'adiposity' exhibit varied distributions, while 'sbp' and 'ldl' showcase a somewhat symmetric pattern.
- 'obesity', 'typea' and 'age' display distributions with peaks and varying spread across their ranges.
- 'chd' and 'famhist' likely represent categorical variables encoded as numeric values, as their histograms exhibit distinct bars.

These visualizations provide insight into the distribution patterns of the dataset's numeric variables, assisting in identifying potential trends or irregularities in the data.

- iiii. Plot a heat map for missing values

In [21]:
```
plt.figure(figsize=(8, 6))
sns.heatmap(heartTraindf.isnull(), cbar=False, cmap='viridis')
plt.title("Missing Values Heatmap")
plt.show()
```



The code generates a heatmap visualizing missing values in our dataset. The entire heatmap is uniformly purple, indicating an absence of missing or null values across all features.

The uniform purple coloring in the heatmap signifies a lack of missing values within the dataset. This visualization reassures that the **dataset is complete**, without any significant gaps or null entries, ensuring data integrity for subsequent analyses or modeling.

10) Check for missing values and What additional techniques to handle null values, excluding the drop na feature?

As we saw in the above parts, there are no missing values in our dataset, so handling null values is not a concern here. However, we can consider dealing with handling missing values (without using the 'dropna' feature) by following methods:

Handling null data commonly involves **imputation**. Basic methods include using **statistical measures (mean, median, or mode)** for filling missing entries. Advanced techniques like **K-Nearest Neighbors (KNN)** or **predictive modeling** can better estimate and replace missing values, preserving more data for robust analysis.

When considering imputation methods, the **K-Nearest Neighbors (KNN) algorithm** is a valuable approach. KNN imputation involves locating the 'k' nearest observations based on similarity to the missing data entry. These neighboring observations help predict and substitute the missing value. This method provides a more contextual and data-driven way of handling missing values, especially in datasets where the information is spatially or contextually correlated.

Part 2

1. Basic Matrix Multiplication (20 pts)

In [22]:
```python
import numpy as np

def matrix_multiply(A, B):
    """
    Multiplies two matrices A and B without using the `dot` function.

    Parameters:
    A (numpy.ndarray): First matrix.
    B (numpy.ndarray): Second matrix.

    Returns:
    numpy.ndarray: The product of matrices A and B.
    """
    # Get the dimensions of the input matrices
    rows_A, cols_A = A.shape
    rows_B, cols_B = B.shape

    # Initialize the result matrix with zeros
    result = np.zeros((rows_A, cols_B))

    # Perform the matrix multiplication manually
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result

# Example usage:
A = np.array([[12, 21], [2, 8]])
B = np.array([[13, 7], [7, 8]])

print(matrix_multiply(A, B))
```

```
[[303. 252.]
 [ 82.  78.]]
```

2. Comupte the Determinant (10pts)

In [23]:
```python
import numpy as np

def compute_determinant(A):
    """
    Computes the determinant of a square matrix A.

    Parameters:
    A (numpy.ndarray): Square matrix.

    Returns:
    float: Determinant of the matrix A.
    """
    return np.linalg.det(A)

# Example usage:
A = np.array([[11, 13], [15, 17]])
print(compute_determinant(A))
```

```
-8.000000000000012
```

3. Solve a System of Linear Equations (10pts)

In [24]:
```python
import numpy as np

def solve_linear_system(A, b):
    """
    Solves the system of linear equations Ax = b.

    Parameters:
    A (numpy.ndarray): Coefficient matrix.
    b (numpy.ndarray): Constant vector.

    Returns:
    numpy.ndarray: Solution vector x.
    """
    return np.linalg.solve(A, b)

# Example usage:
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
print(solve_linear_system(A, b))
```

```
[2. 3.]
```