# Mastering Pandas in Python


**Outline**

**1. Introduction:**

- What is Pandas?
- Purpose of Pandas.
- Why Pandas is used ?

**2. Starting with Pandas:**

- Installing Pandas.
- Importing Pandas.

**3. Data Structures in Pandas:**

- Series: Definition, Creation, and Examples.
- DataFrame: Definition, Creation, and Example.

**4. Loading Data:**

- From CSV, Excel, JSON, HTML etc..

**6. Attributes of DataFrame:**

- Shape: Definition, Example.
- info: Definition, Example.
- columns: Definition, Example.
- dtypes: Definition, Example.
- describe: Definition, Example.
- head: Definition, Example.
- tail: Definition, Example.

**7. Accessing the data:**

- To access a column
- df.iloc()
- df.loc()
- To extract single value:-
    1. df.at()
    2. df.iat()

**8. Descriptive Statistics:**

- Categorical data
    1. unique()
    2. nunique()
    3. value_counts()
    4. normalize = True
    5. mode()
- Numerical data
    1. mean()
    2. median()
    3. max()
    4. min()
    5. var()
    6. std()
    7. quantile()
    8. skew()
    9. corr()

---

**Introduction**

**What is Pandas?** Pandas is a powerful open source Python library designed for data manipulation and analysis. It provides flexible data structures, such as Series and DataFrame, making data analysis tasks simpler and more efficient.

**Purpose of Pandas**

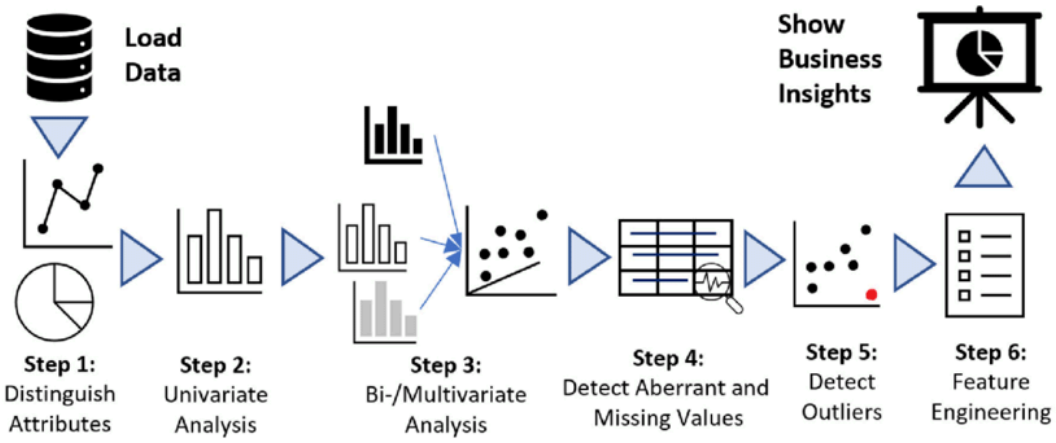
1. Data Manipulation
2. Data Cleaning
3. Data Visualization

**Why Pandas is Used?** Pandas is a Python library used for working with structured data, such as tables and spreadsheets. It makes it easier to organize, clean, analyze, and visualize data efficiently.

1. Simplifies Data Handling.
2. Powerful Tools for Analysis.
3. Flexible Data Structures.

4. Easier Exploration and Visualization.
5. Saves Time and Effort.

---




**Starting with Pandas**

**Installing Pandas**

- Install Pandas using pip

!pip install pandas

**Importing Pandas**

In [1]:
```
import pandas as pd
```

---

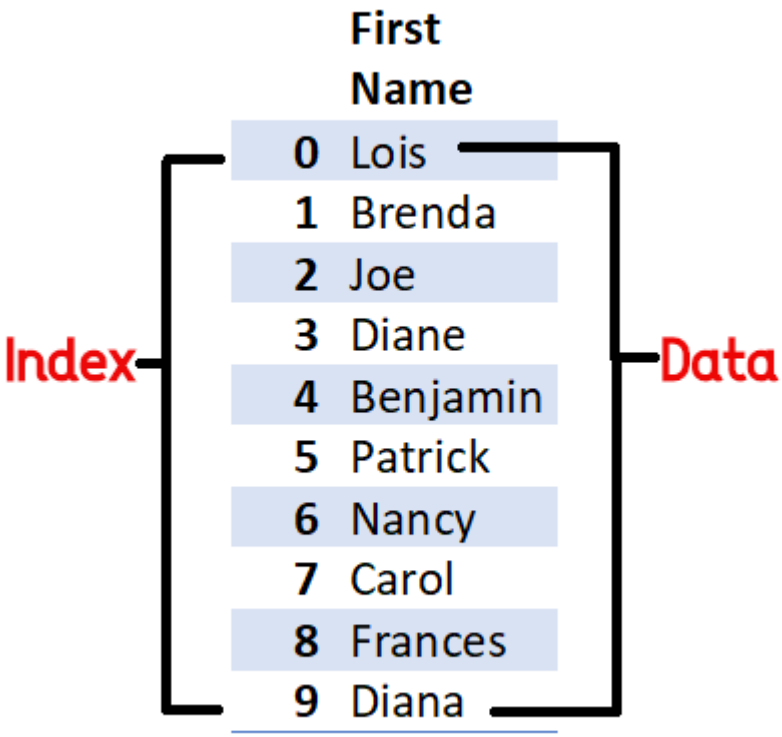
**Data Structures in Pandas**

1. **Series:**

A Series is a one-dimensional (1-D data), capable of holding any data type (e.g., integers, floats, strings, etc.). It is a single column or row of data.

**Features of Series:**

- Homogeneous data (all elements have the same type).
- Has an index for labeling each element.




In [8]:
```
# List

import pandas as pd

# Creating a Series
data = [10, 20, 30, 40]
series = pd.Series(data)
print(series)
```

```
0    10
1    20
2    30
3    40
dtype: int64
```

In [9]:
```
# tuple

data = (10, 20, 30, 40)
series = pd.Series(data)
print(series)
```

```
0    10
1    20
2    30
3    40
dtype: int64
```

In [11]:
```
# From numpy array

import numpy as np
data = np.random.randint(1,20,4)
data_1 = pd.Series(data)
data_1
```

Out[11]:
```
0     8
1     9
2     7
3    10
dtype: int32
```

In [12]:
```
#  Using index for labeling each element

data = [10, 20, 30, 40]
series = pd.Series(data, index = ['a','b','c','d'])
print(series)
```

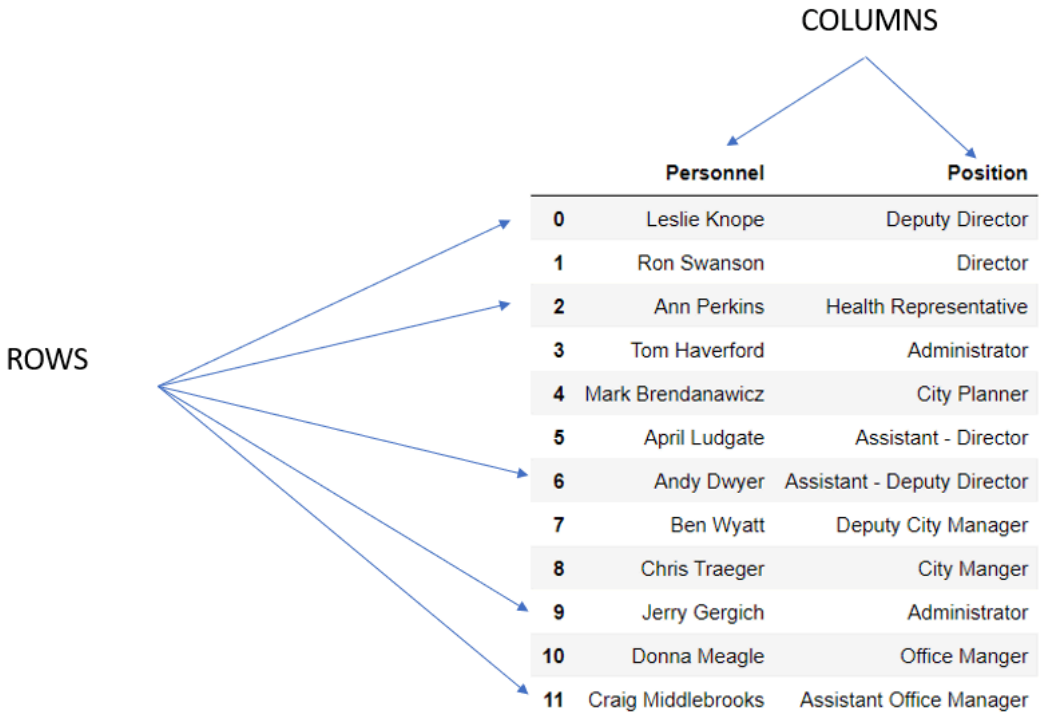
```
a    10
b    20
c    30
d    40
dtype: int64
```

2. **DataFrame:**

A DataFrame is a two-dimensional (2-D data) data structure, it is rows and columns data.
A DataFrame is a combination of Series data.

**Features of DataFrame:**

- Heterogeneous data (different types in different columns).
- Supports a variety of operations like filtering, grouping, and merging.




| | Personnel | Position |
|---|---|---|
| 0 | Leslie Knope | Deputy Director |
| 1 | Ron Swanson | Director |
| 2 | Ann Perkins | Health Representative |
| 3 | Tom Haverford | Administrator |
| 4 | Mark Brendanawicz | City Planner |
| 5 | April Ludgate | Assistant - Director |
| 6 | Andy Dwyer | Assistant - Deputy Director |
| 7 | Ben Wyatt | Deputy City Manager |
| 8 | Chris Traeger | City Manger |
| 9 | Jerry Gergich | Administrator |
| 10 | Donna Meagle | Office Manger |
| 11 | Craig Middlebrooks | Assistant Office Manager |

In [15]:
```
# From a dictionary

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'Salary': [50000, 60000, 70000]}
df = pd.DataFrame(data)
df
```

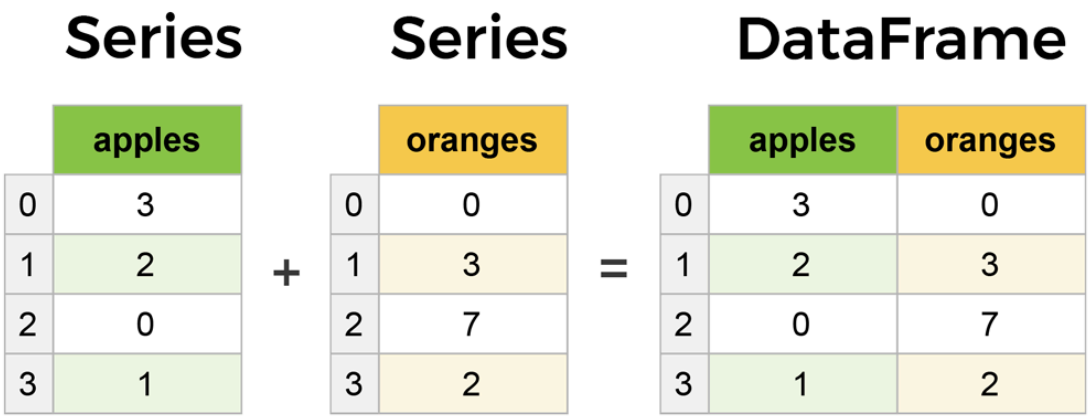
Out[15]:

| | Name | Age | Salary |
|---|---|---|---|
| **0** | Alice | 25 | 50000 |
| **1** | Bob | 30 | 60000 |
| **2** | Charlie | 35 | 70000 |

In [7]:
```
# List of list

df = pd.DataFrame([['TS','Hyd','Revanth'],
                   ['AP','Amaravathi','CBN'],
                   ['KA','Bng','SR']],columns=['State','Capital','CM'])
df
```

Out[7]:

| | State | Capital | CM |
|---|---|---|---|
| **0** | TS | Hyd | Revanth |
| **1** | AP | Amaravathi | CBN |
| **2** | KA | Bng | SR |




**A DataFrame is a combination of Series data.**

---

**Loading Data**

1. .csv --> Comma seperated values.
2. .html --> hyper text markup language.
3. .xls or .xlsx --> excel

**How to handel csv files:**

- **syntax**: pd.read_csv(r'copy as path')
- **syntax**: pd.read_html(r'copy as path')
- **syntax**: pd.read_excel(r'copy as path')

In [20]:
```
df = pd.read_csv(r"D:\IND 2\Downloads\Datasets\Datasets\loandata.csv")
```

---

### Attributes of DataFrame

1. **shape**: It gives no of rows and columns.
2. **info()**: It gives the infromation about the quality of data.
3. **columns**: It gives the column names of the DataFrame.
4. **dtypes**: It gives the column datatypes.
5. **describe()**: It gives the descriptive statistics of the numerical columns only.
6. **head()**: It gives first five rows.
7. **tail()**: It gives last five rows.

In [21]:
```
df.shape
```

Out[21]:
```
(614, 13)
```

In [22]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Loan_ID            614 non-null    object 
 1   Gender             601 non-null    object 
 2   Married            611 non-null    object 
 3   Dependents         599 non-null    object 
 4   Education          614 non-null    object 
 5   Self_Employed      582 non-null    object 
 6   ApplicantIncome    614 non-null    int64  
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object 
 12  Loan_Status        614 non-null    object 
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

In [23]:
```
df.columns
```

Out[23]:
```
Index(['Loan_ID', 'Gender', 'Married', 'Dependents', 'Education',
       'Self_Employed', 'ApplicantIncome', 'CoapplicantIncome', 'LoanAmount',
       'Loan_Amount_Term', 'Credit_History', 'Property_Area', 'Loan_Status'],
      dtype='object')
```

In [24]:
```
df.dtypes
```

Out[24]:
```
Loan_ID              object
Gender               object
Married              object
Dependents           object
Education            object
Self_Employed        object
ApplicantIncome       int64
CoapplicantIncome   float64
LoanAmount          float64
Loan_Amount_Term    float64
Credit_History      float64
Property_Area        object
Loan_Status          object
dtype: object
```

In [25]:
```
df.describe()
```

Out[25]:

| | ApplicantIncome | CoapplicantIncome | LoanAmount | Loan_Amount_Term | Credit_H |
|---|---|---|---|---|---|
| **count** | 614.000000 | 614.000000 | 592.000000 | 600.00000 | 564.0 |
| **mean** | 5403.459283 | 1621.245798 | 146.412162 | 342.00000 | 0.8 |
| **std** | 6109.041673 | 2926.248369 | 85.587325 | 65.12041 | 0.3 |
| **min** | 150.000000 | 0.000000 | 9.000000 | 12.00000 | 0.0 |
| **25%** | 2877.500000 | 0.000000 | 100.000000 | 360.00000 | 1.0 |
| **50%** | 3812.500000 | 1188.500000 | 128.000000 | 360.00000 | 1.0 |
| **75%** | 5795.000000 | 2297.250000 | 168.000000 | 360.00000 | 1.0 |
| **max** | 81000.000000 | 41667.000000 | 700.000000 | 480.00000 | 1.0 |

In [26]:
```
df.head()
```

Out[26]:

| | Loan_ID | Gender | Married | Dependents | Education | Self_Employed | ApplicantIncome |
|---|---|---|---|---|---|---|---|
| **0** | LP001722 | Male | Yes | 0 | Graduate | No | 150 |
| **1** | LP002502 | Female | Yes | 2 | Not Graduate | NaN | 210 |
| **2** | LP002949 | Female | No | 3+ | Graduate | NaN | 416 |
| **3** | LP002603 | Female | No | 0 | Graduate | No | 645 |
| **4** | LP001644 | NaN | Yes | 0 | Graduate | Yes | 674 |

In [27]:
```
df.tail()
```

Out[27]:

| | Loan_ID | Gender | Married | Dependents | Education | Self_Employed | ApplicantInco |
|---|---|---|---|---|---|---|---|
| **609** | LP001640 | Male | Yes | 0 | Graduate | Yes | 39 |
| **610** | LP001536 | Male | Yes | 3+ | Graduate | No | 39 |
| **611** | LP001585 | NaN | Yes | 3+ | Graduate | No | 51 |
| **612** | LP002101 | Male | Yes | 0 | Graduate | NaN | 63 |
| **613** | LP002317 | Male | Yes | 3+ | Graduate | No | 81 |

---

### Accessing The Data

1. **To access a single column**:-

- **syntax**: df.column_name
- **To get two or more columns**:-
- **syntax**: df[['column_name','column_name']]

In [29]:
```
df['Education']
```

Out[29]:
```
0          Graduate
1      Not Graduate
2          Graduate
3          Graduate
4          Graduate
           ...
609        Graduate
610        Graduate
611        Graduate
612        Graduate
613        Graduate
Name: Education, Length: 614, dtype: object
```

In [31]:
```
type(df['Education'])
```

Out[31]:
```
pandas.core.series.Series
```

In [30]:
```
df[['Gender','Property_Area']]
```

Out[30]:

| | Gender | Property_Area |
|---|---|---|
| **0** | Male | Rural |
| **1** | Female | Semiurban |
| **2** | Female | Urban |
| **3** | Female | Rural |
| **4** | NaN | Rural |
| **...** | ... | ... |
| **609** | Male | Semiurban |
| **610** | Male | Semiurban |
| **611** | NaN | Urban |
| **612** | Male | Urban |
| **613** | Male | Rural |

614 rows × 2 columns

In [32]:
```
type(df[['Gender','Property_Area']])
```

Out[32]:
```
pandas.core.frame.DataFrame
```

2. **df.iloc()**: To access the data using index numbers (or) Indexed based accessing the data.

- **syntax**: df.iloc[rows,columns]
- i --> integer or index.
- loc --> location of that particular column name.

In [34]:
```
df.iloc[:,[1,4]]
```

Out[34]:

| | Gender | Education |
|---|---|---|
| **0** | Male | Graduate |
| **1** | Female | Not Graduate |
| **2** | Female | Graduate |
| **3** | Female | Graduate |
| **4** | NaN | Graduate |
| **...** | ... | ... |
| **609** | Male | Graduate |
| **610** | Male | Graduate |
| **611** | NaN | Graduate |
| **612** | Male | Graduate |
| **613** | Male | Graduate |

614 rows × 2 columns

In [38]:
```
df.iloc[[9],:]
```

Out[38]:

| | Loan_ID | Gender | Married | Dependents | Education | Self_Employed | ApplicantIncome |
|---|---|---|---|---|---|---|---|
| **9** | LP001267 | Female | Yes | 2 | Graduate | No | 1378 |

In [39]:
```
df.iloc[:,[1]]
```

Out[39]:

| | Gender |
|---|---|
| **0** | Male |
| **1** | Female |
| **2** | Female |
| **3** | Female |
| **4** | NaN |
| **...** | ... |
| **609** | Male |
| **610** | Male |
| **611** | NaN |
| **612** | Male |
| **613** | Male |

614 rows × 1 columns

3. **df.loc()**: To extract the data using label names.

- **syntax**: df.loc[row_names,column_names]
- loc --> location of that particular column name.

In [41]:
```
df.loc[:,['Dependents']]
```

Out[41]:

| | Dependents |
|---|---|
| **0** | 0 |
| **1** | 2 |
| **2** | 3+ |
| **3** | 0 |
| **4** | 0 |
| **...** | ... |
| **609** | 0 |
| **610** | 3+ |
| **611** | 3+ |
| **612** | 0 |
| **613** | 3+ |

614 rows × 1 columns

In [42]:
```
df.loc[4:8,['Self_Employed','LoanAmount']]
```

Out[42]:

| | Self_Employed | LoanAmount |
|---|---|---|
| **4** | Yes | 168.0 |
| **5** | Yes | 110.0 |
| **6** | No | 112.0 |
| **7** | No | 216.0 |
| **8** | No | 17.0 |

In [44]:
```
df.loc[4:8,'Self_Employed':'LoanAmount']
```

Out[44]:

| | Self_Employed | ApplicantIncome | CoapplicantIncome | LoanAmount |
|---|---|---|---|---|
| **4** | Yes | 674 | 5296.0 | 168.0 |
| **5** | Yes | 1000 | 3022.0 | 110.0 |
| **6** | No | 1025 | 2773.0 | 112.0 |
| **7** | No | 1025 | 5500.0 | 216.0 |
| **8** | No | 1299 | 1086.0 | 17.0 |

4. **To extract single value**:

- 1. **df.at()**: Access the single value using label name.
- **syntax**: df.at[row_names,column_names]
- 2. **df.iat()**: Access the single value using index number.
- **syntax**: df.iat[row_number,column_number]

In [45]:
```
df.at[6,'CoapplicantIncome']
```

Out[45]:
```
2773.0
```

In [46]:
```
df.iat[6,7]
```

Out[46]:
```
2773.0
```

---

**Descriptive Statistics**

- **Categorical data**: To extract categorical columns from dataset.
- **syntax**: df.select_dtypes('object') or df.select_dtypes(include = 'object')

---

1. **unique()**:- What are the possible values/different values in the given columns.
2. **nunique()**:- How many possible values/different values in the given columns.
3. **value_counts()**:- Counts of the unique values in the given columns.
4. **normalize = True**:- To get value_counts in percentage.
5. **Mode()**:- Most repeated or frequently occured values in the column.

In [47]:
```
cat_data = df.select_dtypes(include='object')
```

In [48]:
```
cat_data
```

Out[48]:

| | Loan_ID | Gender | Married | Dependents | Education | Self_Employed | Property_Area |
|---|---|---|---|---|---|---|---|
| **0** | LP001722 | Male | Yes | 0 | Graduate | No | Rura |
| **1** | LP002502 | Female | Yes | 2 | Not Graduate | NaN | Semiurbar |
| **2** | LP002949 | Female | No | 3+ | Graduate | NaN | Urbar |
| **3** | LP002603 | Female | No | 0 | Graduate | No | Rura |
| **4** | LP001644 | NaN | Yes | 0 | Graduate | Yes | Rura |
| **...** | ... | ... | ... | ... | ... | ... | .. |
| **609** | LP001640 | Male | Yes | 0 | Graduate | Yes | Semiurbar |
| **610** | LP001536 | Male | Yes | 3+ | Graduate | No | Semiurbar |
| **611** | LP001585 | NaN | Yes | 3+ | Graduate | No | Urbar |
| **612** | LP002101 | Male | Yes | 0 | Graduate | NaN | Urbar |
| **613** | LP002317 | Male | Yes | 3+ | Graduate | No | Rura |

614 rows × 8 columns

In [49]:
```
cat_data['Loan_Status']
```

Out[49]:
```
0      N
1      Y
2      N
3      Y
4      Y
      ..
609    Y
610    Y
611    Y
612    Y
613    N
Name: Loan_Status, Length: 614, dtype: object
```

In [50]:
```
cat_data['Property_Area'].unique()
```

Out[50]:
```
array(['Rural', 'Semiurban', 'Urban'], dtype=object)
```

In [51]:
```
cat_data['Property_Area'].nunique()
```

Out[51]:
```
3
```

In [52]:
```
cat_data['Property_Area'].value_counts()
```

Out[52]:
```
Property_Area
Semiurban    233
Urban        202
Rural        179
Name: count, dtype: int64
```

In [53]:
```
cat_data['Property_Area'].value_counts(normalize=True)
```

Out[53]:
```
Property_Area
Semiurban    0.379479
Urban        0.328990
Rural        0.291531
Name: proportion, dtype: float64
```

In [54]:
```
cat_data['Property_Area'].mode()
```

Out[54]:
```
0    Semiurban
Name: Property_Area, dtype: object
```

- **Numerical data**: To extract numerical columns from dataset.
- **syntax**: df.select_dtypes(include = ['int','float'])

---

1. **mean()**:- The sum of all data points divided by the number of points.
2. **median()**:- It is used to find middle value of the data points after sorting
3. **min()**: The smallest value in the dataset.
4. **max()**: The largest value in the dataset.
5. **var()**: The average of the squared differences from the mean. It measures the data's spread.
6. **std()**: The square root of variance. It measures the dispersion of data.
7. **quantile()**: Points that divide the data into intervals with equal probabilities. Common quantiles include:

- 0.25 (1st quartile)
- 0.5 (Median or 2nd quartile)
- 0.75 (3rd quartile)

8. **skew()**: A measure of the asymmetry of the data distribution.

- Positive skew: Tail is longer on the right.
- Negative skew: Tail is longer on the left.

9. **corr()**: Measures the relationship between two variables.

- 1: Perfect positive relationship.
- -1: Perfect negative relationship.
- 0: No relationship.

10. **kurt()**: Measures the "tailedness" of the distribution.

- High kurtosis: Heavy tails, more extreme values.
- Low kurtosis: Light tails, fewer extreme values.

In [56]:
```
num_data = df.select_dtypes(include=['int','float'])
```

In [57]:
```
num_data
```

Out[57]:

| | ApplicantIncome | CoapplicantIncome | LoanAmount | Loan_Amount_Term | Credit_His |
|---|---|---|---|---|---|
| **0** | 150 | 1800.0 | 135.0 | 360.0 | |
| **1** | 210 | 2917.0 | 98.0 | 360.0 | |
| **2** | 416 | 41667.0 | 350.0 | 180.0 | |
| **3** | 645 | 3683.0 | 113.0 | 480.0 | |
| **4** | 674 | 5296.0 | 168.0 | 360.0 | |
| **...** | ... | ... | ... | ... | |
| **609** | 39147 | 4750.0 | 120.0 | 360.0 | |
| **610** | 39999 | 0.0 | 600.0 | 180.0 | |
| **611** | 51763 | 0.0 | 700.0 | 300.0 | |
| **612** | 63337 | 0.0 | 490.0 | 180.0 | |
| **613** | 81000 | 0.0 | 360.0 | 360.0 | |

614 rows × 5 columns

In [58]:
```
df['LoanAmount'].mean()
```

Out[58]: 146.41216216216216

In [59]:
```
df['LoanAmount'].median()
```

Out[59]: 128.0

In [60]:
```
df['LoanAmount'].min()
```

Out[60]: 9.0

In [61]:
```
df['LoanAmount'].max()
```

Out[61]: 700.0

In [62]:
```
df['LoanAmount'].var()
```

Out[62]: 7325.190241002426

In [63]:
```
df['LoanAmount'].std()
```

Out[63]: 85.58732523570546

In [64]:
```
df['LoanAmount'].quantile(0.25)
```

Out[64]: 100.0

In [65]:
```
df['LoanAmount'].skew()
```

Out[65]: 2.6775516792560583

In [67]:
```
df.corr(numeric_only=True) # by default pearson
```

Out[67]:

| | ApplicantIncome | CoapplicantIncome | LoanAmount | Loan_Amount_T |
|---|---|---|---|---|
| **ApplicantIncome** | 1.000000 | -0.116605 | 0.570909 | -0.045 |
| **CoapplicantIncome** | -0.116605 | 1.000000 | 0.188619 | -0.059 |
| **LoanAmount** | 0.570909 | 0.188619 | 1.000000 | 0.039 |
| **Loan_Amount_Term** | -0.045306 | -0.059878 | 0.039447 | 1.000 |
| **Credit_History** | -0.014715 | -0.002056 | -0.008433 | 0.001 |

In [68]:
```
df.corr(numeric_only=True,method = 'spearman')
```

Out[68]:

| | ApplicantIncome | CoapplicantIncome | LoanAmount | Loan_Amount_T |
|---|---|---|---|---|
| **ApplicantIncome** | 1.000000 | -0.320056 | 0.515397 | -0.040 |
| **CoapplicantIncome** | -0.320056 | 1.000000 | 0.240719 | -0.013 |
| **LoanAmount** | 0.515397 | 0.240719 | 1.000000 | 0.041 |
| **Loan_Amount_Term** | -0.040111 | -0.013747 | 0.041486 | 1.000 |
| **Credit_History** | 0.043241 | -0.007458 | -0.002361 | 0.015 |

In [69]:
```
df['LoanAmount'].kurt()
```

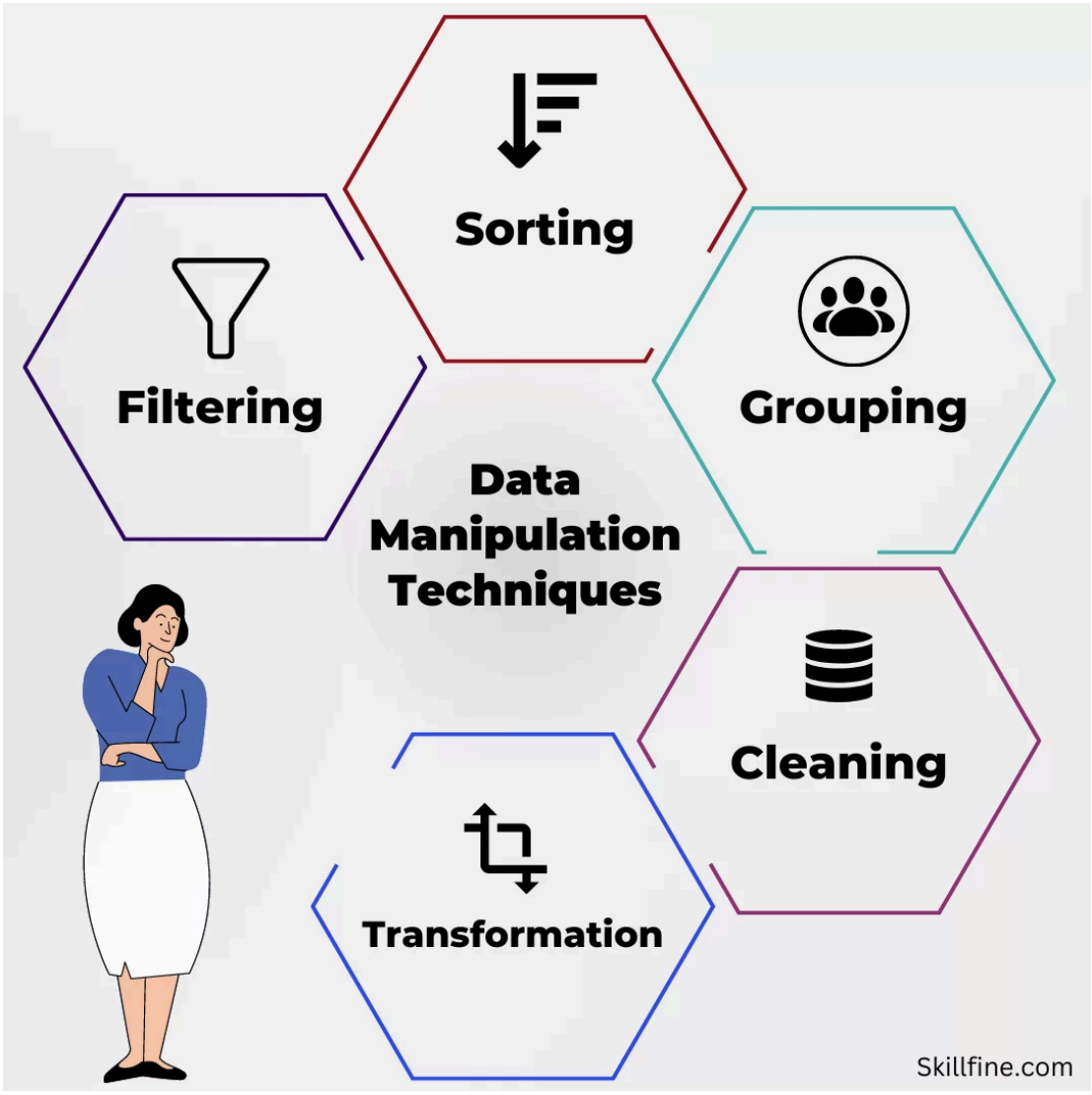
Out[69]:
```
10.401533490294154
```

---

# Data Manipulation

**Why Do We Need Data Manipulation?**

- **Data manipulation is essential because raw data is often messy, inconsistent, or not structured properly. By performing data manipulation, we can clean, organize, and transform the data, allowing for effective analysis.**




**Outline**

**1. Adding a Column and Row:**

**2. Dropping a Column:**

**3. Dropping a Row:**

**4. Renaming Columns:**

**5. Filtering Data Using between():**

**6. Filtering Data Using where():**

**7. Finding the Largest Values Using nlargest():**

**8. Finding the Smallest Values Using nsmallest():**

**9. Sorting Data Using sort_values():**

**10. Resetting Index Using reset_index():**

**11. Joins in Pandas:**

### 12. Merge in Pandas:

### 13. Using apply() for Functions:

### 14. Using map() for Element-wise Mapping:

### 15. Using crosstab() for Frequency Tables:

### 16. Aggregating Data Using agg():

### 17. Creating Summary Tables Using pivot_table():

### 18. Grouping Data Using groupby():

In [4]:
```
import pandas as pd
```

### Adding a Column:

In [13]:
```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)
df
```

Out[13]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **1** | Bob | 30 |
| **2** | Charlie | 35 |

In [14]:
```
df['Gender'] = ['Female', 'Male', 'Male']
df
```

Out[14]:

| | Name | Age | Gender |
|---|---|---|---|
| **0** | Alice | 25 | Female |
| **1** | Bob | 30 | Male |
| **2** | Charlie | 35 | Male |

### Adding a Row:

In [15]:
```
new_row = {'Name': 'David', 'Age': 40, 'Gender': 'Male'}
df.loc[len(df)] = new_row
df
```

Out[15]:

| | Name | Age | Gender |
|---|---|---|---|
| **0** | Alice | 25 | Female |
| **1** | Bob | 30 | Male |
| **2** | Charlie | 35 | Male |
| **3** | David | 40 | Male |

**Dropping a Column:**

- Use `drop()` to remove a column.
- axis = 1 --> columns
- axis = 0 --> rows

In [16]:
```
df = df.drop(columns=['Age'])
df
```

Out[16]:

| | Name | Gender |
|---|---|---|
| **0** | Alice | Female |
| **1** | Bob | Male |
| **2** | Charlie | Male |
| **3** | David | Male |

In [17]:
```
df.drop(['Gender'], axis=1, inplace=True) # inplace = True to change the origina
```

In [18]:
```
df
```

Out[18]:

| | Name |
|---|---|
| **0** | Alice |
| **1** | Bob |
| **2** | Charlie |
| **3** | David |

**Dropping a Row:**

- Remove a row using `drop()`.

In [26]:
```
data = {'Name': ['Alice', 'Bob', 'Charlie','John','Alex'],
        'Age': [25, 30, 35, 40, 45]}
df = pd.DataFrame(data)
df
```

Out[26]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **1** | Bob | 30 |
| **2** | Charlie | 35 |
| **3** | John | 40 |
| **4** | Alex | 45 |

In [27]:
```
df = df.drop(index=1)
df
```

Out[27]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **2** | Charlie | 35 |
| **3** | John | 40 |
| **4** | Alex | 45 |

In [28]:
```
df.drop([0, 2], axis=0, inplace=True)
df
```

Out[28]:

| | Name | Age |
|---|---|---|
| **3** | John | 40 |
| **4** | Alex | 45 |

**Resetting Index Using reset_index():**

In [29]:
```
df.reset_index(drop = True, inplace=True) # After dropping rows, reset the index
```

In [30]:
```
df
```

Out[30]:

| | Name | Age |
|---|---|---|
| **0** | John | 40 |
| **1** | Alex | 45 |

**Renaming Columns:**

- Use `rename()` to rename column names.

In [31]:
```
df.rename(columns={'Name': 'Full_Name'}, inplace=True)
df
```

Out[31]:

| | Full_Name | Age |
|---|---|---|
| **0** | John | 40 |
| **1** | Alex | 45 |

**Filtering Data Using between():**

- `between()` helps filter values within a range.

In [32]:
```
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie'],
                   'Age': [25, 30, 45]})
filtered_df = df[df['Age'].between(20, 40)]
filtered_df
```

Out[32]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **1** | Bob | 30 |

**Filtering Data Using where():**

- `where()` works like `if-else` for filtering.

In [33]:
```
df['Age'] = df['Age'].where(df['Age'] > 30)
df
```

Out[33]:

| | Name | Age |
|---|---|---|
| **0** | Alice | NaN |
| **1** | Bob | NaN |
| **2** | Charlie | 45.0 |

**Finding the Largest Values Using nlargest():**

- Find the top N largest values.

In [36]:
```
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie'],
                   'Age': [25, 30, 45]})
df
```

Out[36]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **1** | Bob | 30 |
| **2** | Charlie | 45 |

In [37]:
```
df.nlargest(2, 'Age')
```

Out[37]:

| | Name | Age |
|---|---|---|
| **2** | Charlie | 45 |
| **1** | Bob | 30 |

**Finding the Smallest Values Using nsmallest():**

- Find the top N smallest values.

In [40]:
```
df.nsmallest(2, 'Age')
```

Out[40]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **1** | Bob | 30 |

**Sorting Data Using sort_values():**

- Sort data based on column values.

In [41]:
```
df.sort_values(by='Age', ascending=True, inplace=True)
```

In [42]:
```
df
```

Out[42]:

| | Name | Age |
|---|---|---|
| **0** | Alice | 25 |
| **1** | Bob | 30 |
| **2** | Charlie | 45 |

In [44]:
```
df.sort_values(by='Name', ascending=False)
```

Out[44]:

| | Name | Age |
|---|---|---|
| **2** | Charlie | 45 |
| **1** | Bob | 30 |
| **0** | Alice | 25 |

**Sorting the index:**

- The `sort_index()` method sorts a DataFrame based on the index

In [51]:
```
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
df = pd.DataFrame(data, index=[2, 0, 1])
print("Before Sorting:")
print(df)
df_sorted = df.sort_index()
print("\nAfter Sorting:")
print(df_sorted)
```

```
Before Sorting:
      Name  Age
2    Alice   25
0      Bob   30
1  Charlie   35

After Sorting:
      Name  Age
0      Bob   30
1  Charlie   35
2    Alice   25
```

**Joins in Pandas**

- `join()` is used to merge two DataFrames on their index.

In [62]:
```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [1, 2, 3], 'Age': [25, 30, 35]})

df1.set_index("ID", inplace=True)
df2.set_index("ID", inplace=True)

result = df1.join(df2)
result
```

Out[62]:

| ID | Name | Age |
|---|---|---|
| 1 | Alice | 25 |
| 2 | Bob | 30 |
| 3 | Charlie | 35 |

**Merge in Pandas:**

- `merge()` is used to combine two DataFrames based on a common column.
- Supports different join types: inner, left, right, outer.

In [64]:
```
df_merged = pd.merge(df1, df2, on='ID', how='inner')
df_merged
```

Out[64]:

| ID | Name | Age |
|---|---|---|
| 1 | Alice | 25 |
| 2 | Bob | 30 |
| 3 | Charlie | 35 |

**Using apply() for Functions:**

- `apply()` allows applying a function to each row or column.

In [65]:
```
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]})
def age_category(age):
    return "Young" if age < 30 else "Old"
df['Category'] = df['Age'].apply(age_category)
df
```

Out[65]:

| | Name | Age | Category |
|---|---|---|---|
| **0** | Alice | 25 | Young |
| **1** | Bob | 30 | Old |
| **2** | Charlie | 35 | Old |

**Using map() for Element-wise Mapping:**

- `map()` is used for element-wise transformations in a Series.

In [67]:
```
gender_map = {'M': 'Male', 'F': 'Female'}
df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Gender': ['F', 'M']})
df['Gender'] = df['Gender'].map(gender_map)
df
```

Out[67]:

| | Name | Gender |
|---|---|---|
| **0** | Alice | Female |
| **1** | Bob | Male |

**Using crosstab() for Frequency Tables:**

- `crosstab()` creates frequency tables between two categorical variables.

In [70]:
```
df = pd.DataFrame({'Gender': ['Male', 'Female', 'Female', 'Male'],
                   'Survived': ['Yes', 'No', 'Yes', 'Yes']})
result = pd.crosstab(df['Gender'], df['Survived'])
result
```

Out[70]:

| Survived | No | Yes |
|---|---|---|
| **Gender** | | |
| **Female** | 1 | 1 |
| **Male** | 0 | 2 |

**Aggregating Data Using agg():**

- `agg()` allows applying multiple aggregation functions on columns.

In [73]:
```
df = pd.DataFrame({'Age': [25, 30, 35, 40], 'Salary': [50000, 60000, 70000, 8000
df
```

Out[73]:

| | Age | Salary |
|---|---|---|
| **0** | 25 | 50000 |
| **1** | 30 | 60000 |
| **2** | 35 | 70000 |
| **3** | 40 | 80000 |

In [75]:
```
df['Salary'].agg(['sum','max'])
```

Out[75]:
```
sum    260000
max     80000
Name: Salary, dtype: int64
```

**Creating Summary Tables Using pivot_table():**

- `pivot_table()` summarizes data with aggregations.

In [76]:
```
df = pd.DataFrame({'Department': ['HR', 'IT', 'IT', 'HR'],
                   'Salary': [50000, 60000, 70000, 55000]})
pivot = df.pivot_table(values='Salary', index='Department', aggfunc='mean')
pivot
```

Out[76]:

| | Salary |
|---|---|
| **Department** | |
| **HR** | 52500.0 |
| **IT** | 65000.0 |

**Grouping Data Using groupby():**

- `groupby()` is used to group and aggregate data based on column values.

In [77]:
```
df = pd.DataFrame({'Department': ['HR', 'IT', 'IT', 'HR'],
                   'Salary': [50000, 60000, 70000, 55000]})
df_grouped = df.groupby('Department')['Salary'].mean()
df_grouped
```

Out[77]:
```
Department
HR    52500.0
IT    65000.0
Name: Salary, dtype: float64
```

---

# Data Cleaning

**Why Do We Need Data Cleaning?**

- **To handle missing values, outliers, duplicates, and inconsistent data, we need to perform data cleaning.**


**Outline**

**1. Missing values:**

**2. Outliers:**

**3. Duplicates:**

**4. Inconsistent data**

**Handling Missing Values:**

- Missing values occur when data is incomplete or not recorded properly. Handling them is crucial to avoid incorrect analysis.

In [81]:
```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', None],
        'Age': [25, None, 35, 40],
        'Salary': [50000, 60000, None, 80000]}
df = pd.DataFrame(data)
df.isnull().sum() # Use to find no of missing values.
```

Out[81]:
```
Name      1
Age       1
Salary    1
dtype: int64
```

**Filling Missing Values:**

In [85]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [91]:
```
# Fill missing numerical values with mean or median
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].mean(), inplace=True)

# Fill missing categorical values with mode (most frequent value)
df['Name'].fillna(df['Name'].mode()[0], inplace=True)
```

In [92]:
```
df.isna().sum() # They are no missing values now
```

Out[92]:
```
Name      0
Age       0
Salary    0
dtype: int64
```

**Handling Outliers:**

- Outliers are extreme values that differ significantly from the rest of the data. They can distort results and affect model performance.

**Detecting Outliers Using IQR (Interquartile Range):**

In [95]:
```
import numpy as np

df = pd.DataFrame({'Age': [22, 25, 30, 35, 40, 100]})  # 100 is an outlier
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

outliers = df[(df['Age'] < lower_bound) | (df['Age'] > upper_bound)]
print("Outliers:\n", outliers)
```

```
Outliers:
    Age
5  100
```

**Removing Outliers:**

In [96]:
```
df_cleaned = df[(df['Age'] >= lower_bound) & (df['Age'] <= upper_bound)]
print(df_cleaned)
```

```
   Age
0   22
1   25
2   30
3   35
4   40
```

**Handling Duplicate Data:**

- Duplicate records are repeated entries that can lead to incorrect results. They should be removed to maintain data integrity.

In [101…
```
df = pd.DataFrame({'ID': [1, 2, 3, 3],
                   'Name': ['Alice', 'Bob', 'Charlie', 'Charlie']})

print(df.duplicated())
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

In [100…
```
print(df.duplicated().sum())
```

1

**Removing Duplicates:**

In [108…
```
df_1 = df.drop_duplicates()
```

**Fixing Inconsistent Data:**

- Inconsistent data includes incorrect formats, extra spaces, or mismatched capitalization that can cause errors in analysis.

**Standardizing Text Data:**

In [110…
```
# Sample Data
df = pd.DataFrame({'Category': ['electronics', 'ELECTRONICS', '  Electronics  ']
df
```

Out[110…

| | Category |
|---|---|
| **0** | electronics |
| **1** | ELECTRONICS |
| **2** | Electronics |

In [112…
```
df['Category'] = df['Category'].str.lower().str.strip()
df
```

Out[112…

| | Category |
|---|---|
| **0** | electronics |
| **1** | electronics |
| **2** | electronics |

**Converting Data Types:**

In [114…
```
data = {
    'Product': ['Laptop', 'Mobile', 'Tablet', 'Headphones'],
    'Price': ['50000', '20000', '15000', '3000'],
    'Date': ['2024-02-01', '2024-02-05', '2024/02/10', '01-02-2024']}
df = pd.DataFrame(data)
df
```

Out[114…

| | Product | Price | Date |
|---|---|---|---|
| **0** | Laptop | 50000 | 2024-02-01 |
| **1** | Mobile | 20000 | 2024-02-05 |
| **2** | Tablet | 15000 | 2024/02/10 |
| **3** | Headphones | 3000 | 01-02-2024 |

In [115…
```
df.dtypes
```

Out[115…
```
Product    object
Price      object
Date       object
dtype: object
```

In [117…
```
df['Price'] = df['Price'].astype(int)
df['Date'] = pd.to_datetime(df['Date'],format='mixed')
```

In [119…
```
df.dtypes
```

Out[119…
```
Product            object
Price               int32
Date       datetime64[ns]
dtype: object
```

In [120…
```
df
```

Out[120…

| | Product | Price | Date |
|---|---|---|---|
| **0** | Laptop | 50000 | 2024-02-01 |
| **1** | Mobile | 20000 | 2024-02-05 |
| **2** | Tablet | 15000 | 2024-02-10 |
| **3** | Headphones | 3000 | 2024-01-02 |

---

# Data Visualization

**Why Do We Need Data Visualization?**

- **Data visualization helps in understanding complex data easily by representing it through graphs and charts, making insights more accessible and actionable.**


**Univariate Analysis (Single Variable):**

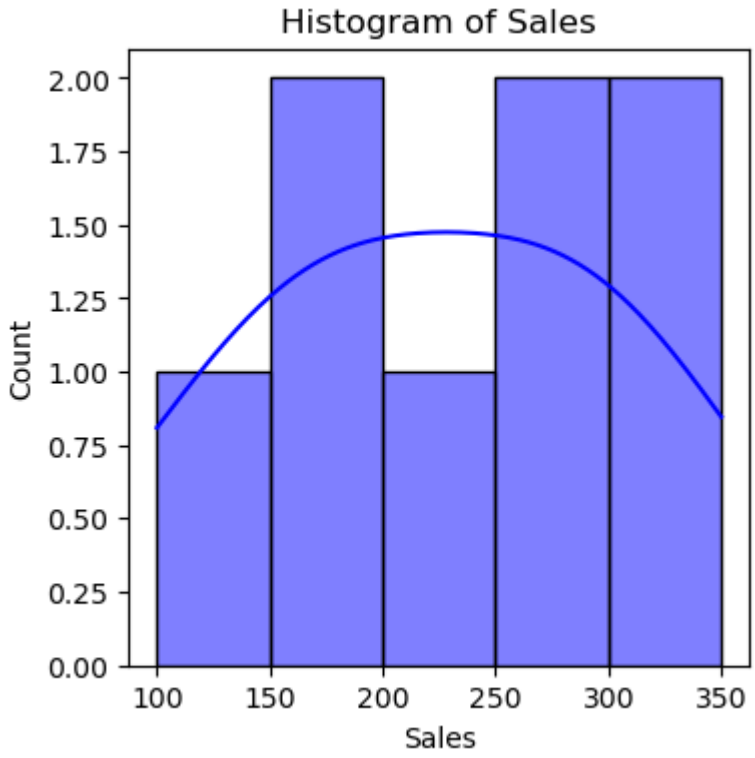
- Univariate analysis examines one variable at a time to understand its distribution and characteristics.

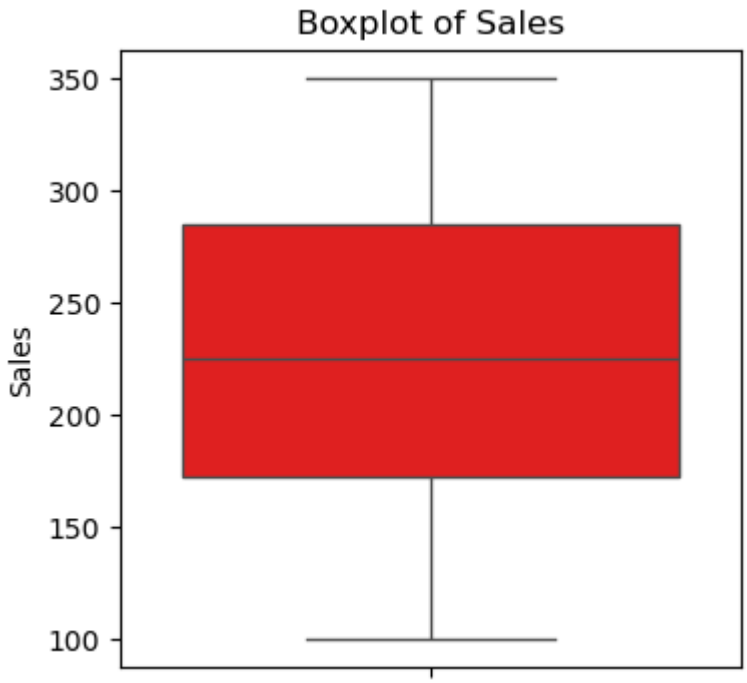
- **Suitable Plots for Univariate Analysis:**

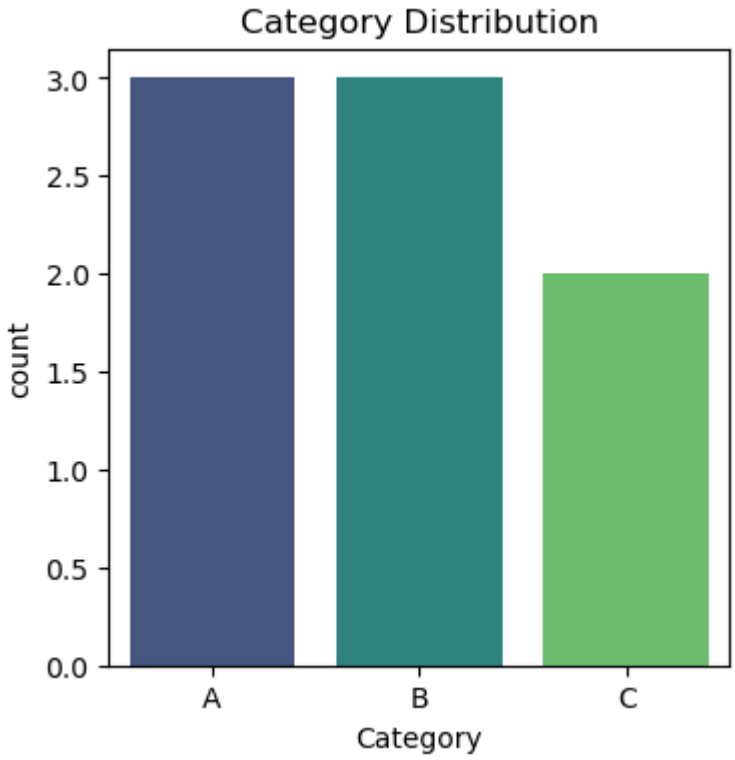
1. **Histogram** → To check frequency distribution of numerical data
2. **Box Plot** → To detect outliers and spread of data
3. **Density Plot** → For probability distribution
4. **Bar Plot** → For categorical data frequency distribution
5. **Count Plot** → For categorical data distribution
6. **Pie Chart** → To visualize proportions

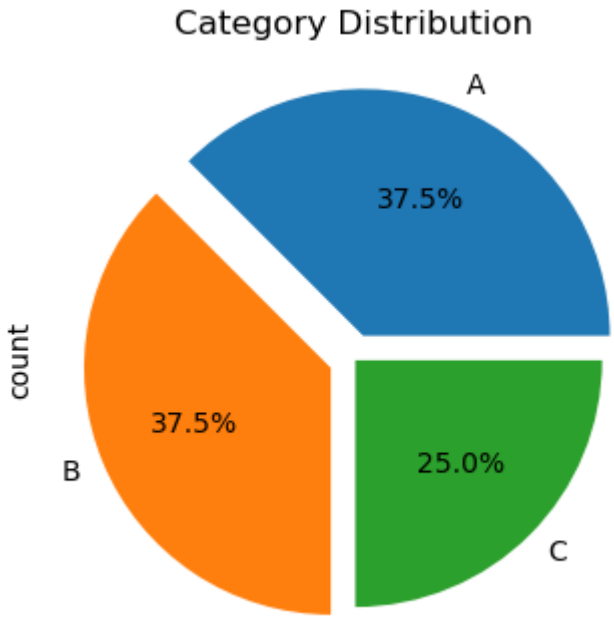
In [135…

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

data = {'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'B'],
        'Sales': [100, 200, 150, 300, 250, 180, 350, 280]}

df = pd.DataFrame(data)

# Histogram
plt.figure(figsize=(4, 4))
sns.histplot(df['Sales'], bins=5, kde=True, color='blue')
plt.title('Histogram of Sales')
plt.show()

# Box Plot
plt.figure(figsize=(4, 4))
sns.boxplot(y=df['Sales'], color='red')
plt.title('Boxplot of Sales')
plt.show()

# Bar Plot (for categorical data)
plt.figure(figsize=(4, 4))
sns.countplot(x=df['Category'], palette='viridis')
plt.title('Category Distribution')
plt.show()

# pie chart (for categorical data)
plt.figure(figsize=(4, 4))
df['Category'].value_counts().plot(kind='pie',autopct='%.1f%%',explode=[0.1,0.1,
plt.title('Category Distribution')
plt.show()
```


Histogram of Sales
Count
Sales
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
100
150
200
250
300
350


Boxplot of Sales
Sales
100
150
200
250
300
350




**Bivariate Analysis (Two Variables):**

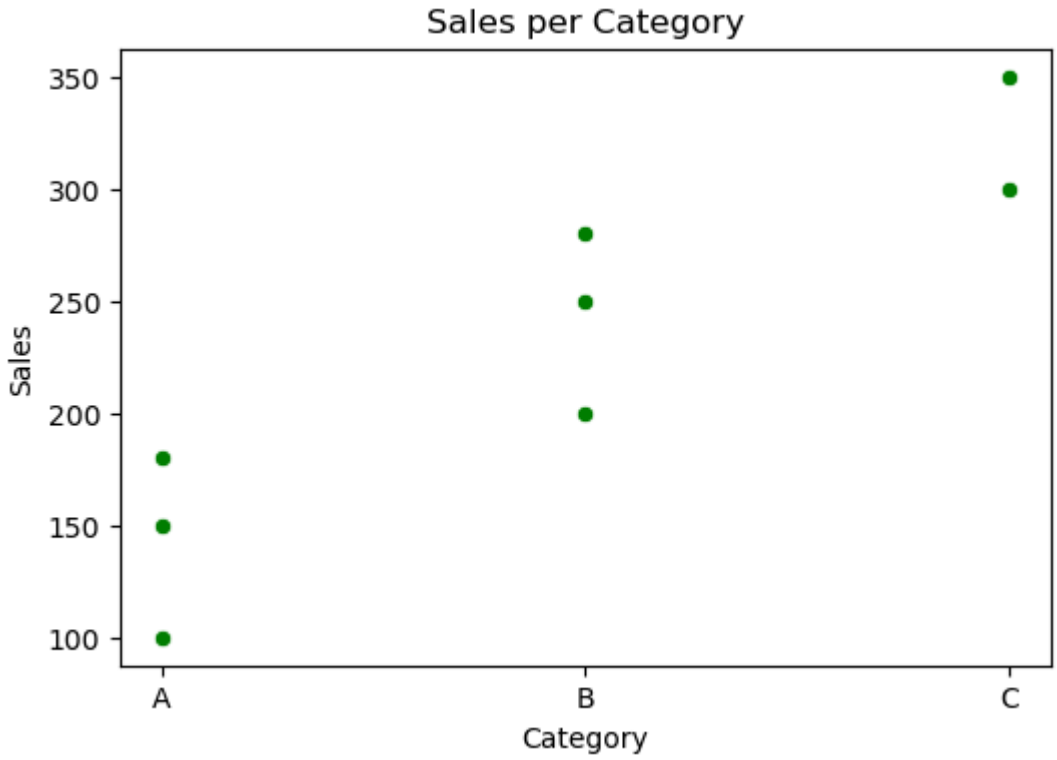
- Bivariate analysis examines relationships between two variables (numerical vs numerical, categorical vs numerical, categorical vs categorical).

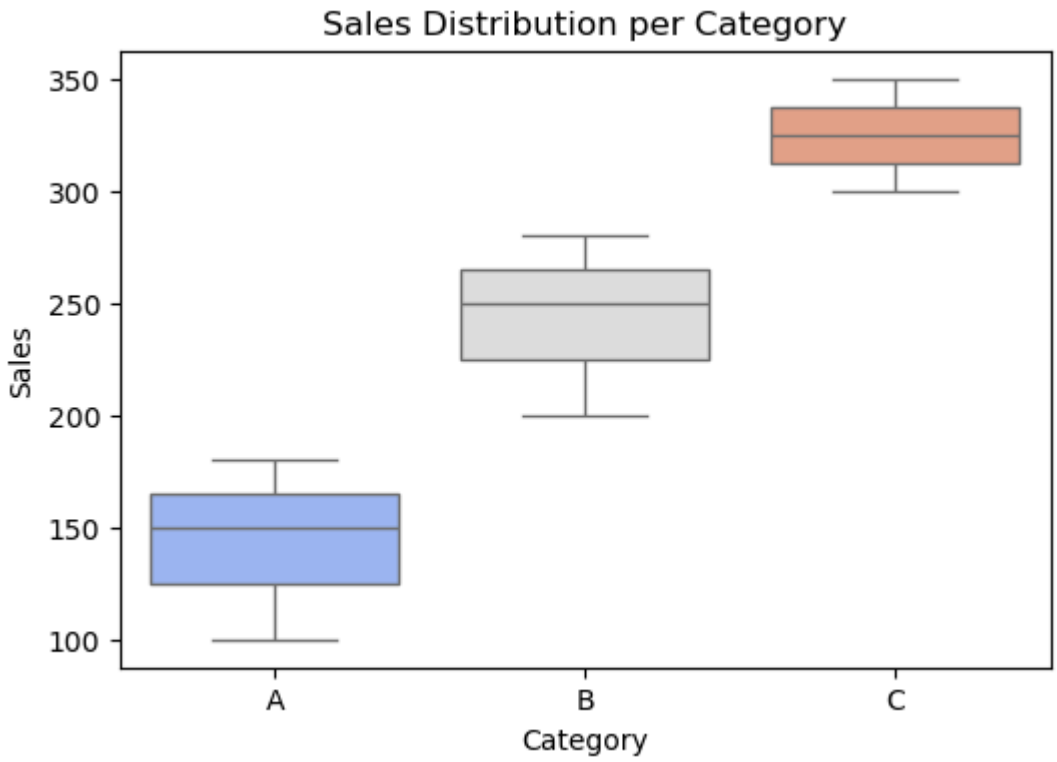
**Suitable Plots for Bivariate Analysis:**

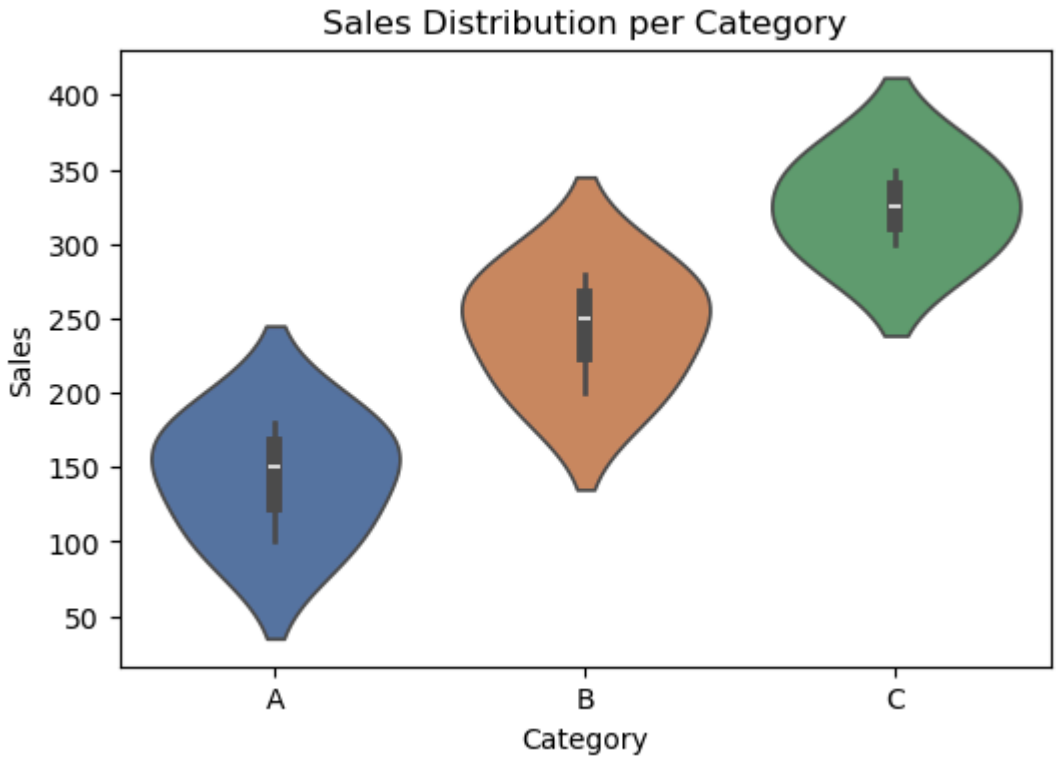
1. **Scatter Plot** → To check relationships between two numerical variables
2. **Line Plot** → To observe trends over time
3. **Box Plot** → To compare a numerical variable across categories
4. **Violin Plot** → To show distribution & density of data per category

In [138…

```
# Scatter Plot (Numerical vs Numerical)
plt.figure(figsize=(6, 4))
sns.scatterplot(x=df['Category'], y=df['Sales'], color='green')
plt.title('Sales per Category')
plt.show()

# Box Plot (Categorical vs Numerical)
plt.figure(figsize=(6, 4))
sns.boxplot(x=df['Category'], y=df['Sales'], palette='coolwarm')
plt.title('Sales Distribution per Category')
plt.show()

# Violin plot
plt.figure(figsize=(6, 4))
sns.violinplot(x=df['Category'], y=df['Sales'], palette='deep')
plt.title('Sales Distribution per Category')
plt.show()
```






**Multivariate Analysis (Three or More Variables):**

- Multivariate analysis explores relationships among three or more variables.

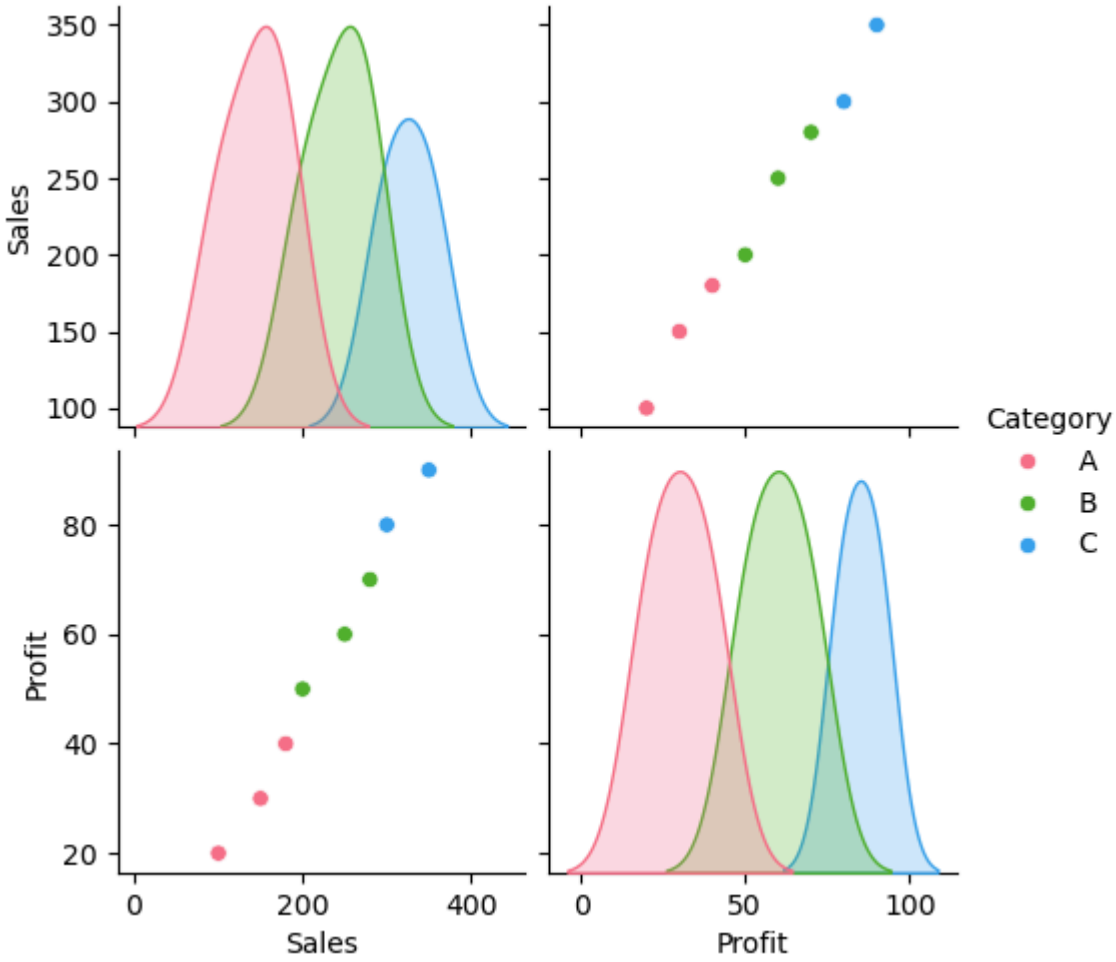
**Suitable Plots for Multivariate Analysis:**

1. **Pair Plot** → Visualizes relationships among multiple numerical variables.
2. **Heatmap** → Shows correlations between numerical features.

In [140…

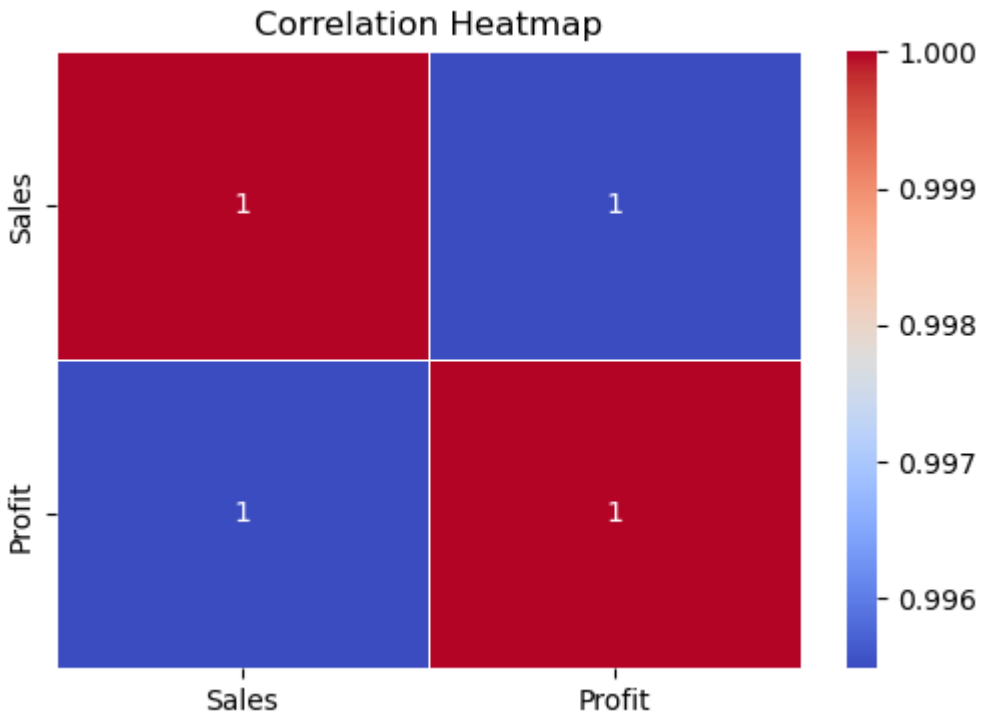
```
data = {'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'B'],
        'Sales': [100, 200, 150, 300, 250, 180, 350, 280],
```

```
        'Profit': [20, 50, 30, 80, 60, 40, 90, 70]}

df = pd.DataFrame(data)

# Pair Plot (Multiple Numeric Variables)
sns.pairplot(df, hue='Category', palette='husl')
plt.show()
```




In [142…

```
# Heatmap (Correlation Between Numeric Variables)
plt.figure(figsize=(6, 4))
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap='coolwarm', linewidths=
plt.title('Correlation Heatmap')
plt.show()
```




# Implementing Data Manipulation, Cleaning, and Visualization on a Dataset:

In [171…
```
import pandas as pd
```

In [172…
```
df = pd.read_csv(r"D:\IND 2\Downloads\Datasets\Datasets\titanic (1).csv")
```

In [173…
```
df.shape
```

Out[173… (900, 12)

- **PassengerId:** Unique ID assigned to each passenger
- **Pclass:** Ticket class (1st, 2nd, or 3rd class)
- **Name:** Full name of the passenger
- **Sex:** Gender of the passenger (male/female)
- **Age:** Age of the passenger

- **SibSp:** Number of siblings/spouses aboard the Titanic
- **Parch:** Number of parents/children aboard the Titanic
- **Ticket:** Ticket number
- **Fare:** Price of the ticket
- **Cabin:** Cabin number assigned to the passenger
- **Embarked:** Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)
- **Survived:** Survival status (0 = Did not survive, 1 = Survived)

In [174…
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 900 entries, 0 to 899
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  900 non-null    int64  
 1   Pclass       900 non-null    int64  
 2   Name         900 non-null    object 
 3   Sex          900 non-null    object 
 4   Age          725 non-null    float64
 5   SibSp        900 non-null    int64  
 6   Parch        900 non-null    int64  
 7   Ticket       900 non-null    object 
 8   Fare         900 non-null    float64
 9   Cabin        206 non-null    object 
 10  Embarked     898 non-null    object 
 11  Survived     900 non-null    int64  
dtypes: float64(2), int64(5), object(5)
memory usage: 84.5+ KB
```

In [175…
```
df.describe(include='all')
```

Out[175...

| | PassengerId | Pclass | Name | Sex | Age | SibSp | Parch |
|---|---|---|---|---|---|---|---|
| **count** | 900.000000 | 900.000000 | 900 | 900 | 725.000000 | 900.000000 | 900.000000 |
| **unique** | NaN | NaN | 891 | 2 | NaN | NaN | NaN |
| **top** | NaN | NaN | Alexander, Mr. William | male | NaN | NaN | NaN |
| **freq** | NaN | NaN | 2 | 583 | NaN | NaN | NaN |
| **mean** | 450.500000 | 2.310000 | NaN | NaN | 46.505062 | 0.523333 | 0.380000 |
| **std** | 259.951919 | 0.836135 | NaN | NaN | 318.158513 | 1.104400 | 0.804631 |
| **min** | 1.000000 | 1.000000 | NaN | NaN | 0.420000 | 0.000000 | 0.000000 |
| **25%** | 225.750000 | 2.000000 | NaN | NaN | 21.000000 | 0.000000 | 0.000000 |
| **50%** | 450.500000 | 3.000000 | NaN | NaN | 28.000000 | 0.000000 | 0.000000 |
| **75%** | 675.250000 | 3.000000 | NaN | NaN | 39.000000 | 1.000000 | 0.000000 |
| **max** | 900.000000 | 3.000000 | NaN | NaN | 7000.000000 | 8.000000 | 6.000000 |

In [176...

```
df.head()
```

Out[176...

| | PassengerId | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabi |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN |
| **1** | 2 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 |
| **2** | 3 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN |
| **3** | 4 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35.0 | 1 | 0 | 113803 | 53.1000 | C123 |
| **4** | 5 | 3 | Allen, Mr. William Henry | male | 35.0 | 0 | 0 | 373450 | 8.0500 | NaN |

In [177...

```
df.tail()
```

Out[177…

| | PassengerId | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabi |
|---|---|---|---|---|---|---|---|---|---|---|
| **895** | 896 | 1 | McCarthy, Mr. Timothy J | male | 54.0 | 0 | 0 | 17463 | 51.8625 | E4 |
| **896** | 897 | 3 | Alexander, Mr. William | male | 26.0 | 0 | 0 | 3474 | 7.8875 | Na |
| **897** | 898 | 3 | Lester, Mr. James | male | 39.0 | 0 | 0 | A/4 48871 | 24.1500 | Na |
| **898** | 899 | 2 | Slemen, Mr. Richard James | male | 35.0 | 0 | 0 | 28206 | 10.5000 | Na |
| **899** | 900 | 3 | Andersson, Miss. Ebba Iris Alfrida | female | 6.0 | 4 | 2 | 347082 | 31.2750 | Na |

In [178…
```
df.columns
```

Out[178…
```
Index(['PassengerId', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch',
       'Ticket', 'Fare', 'Cabin', 'Embarked', 'Survived'],
      dtype='object')
```

In [179…
```
df.dtypes
```

Out[179…
```
PassengerId      int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
Survived         int64
dtype: object
```

In [180…
```
df['FamilySize'] = df['SibSp'] + df['Parch']
```

In [181…
```
df.drop(columns=['PassengerId', 'Ticket', 'Name'], inplace=True)
```

In [182…
```
df
```

Out[182...

| | Pclass | Sex | Age | SibSp | Parch | Fare | Cabin | Embarked | Survived | FamilySiz |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 3 | male | 22.0 | 1 | 0 | 7.2500 | NaN | S | 0 | |
| **1** | 1 | female | 38.0 | 1 | 0 | 71.2833 | C85 | C | 1 | |
| **2** | 3 | female | 26.0 | 0 | 0 | 7.9250 | NaN | S | 1 | |
| **3** | 1 | female | 35.0 | 1 | 0 | 53.1000 | C123 | S | 1 | |
| **4** | 3 | male | 35.0 | 0 | 0 | 8.0500 | NaN | S | 0 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **895** | 1 | male | 54.0 | 0 | 0 | 51.8625 | E46 | S | 0 | |
| **896** | 3 | male | 26.0 | 0 | 0 | 7.8875 | NaN | S | 0 | |
| **897** | 3 | male | 39.0 | 0 | 0 | 24.1500 | NaN | S | 0 | |
| **898** | 2 | male | 35.0 | 0 | 0 | 10.5000 | NaN | S | 0 | |
| **899** | 3 | female | 6.0 | 4 | 2 | 31.2750 | NaN | S | 0 | |

900 rows × 10 columns

In [183...
```
df.isnull().sum()
```

Out[183...
```
Pclass          0
Sex             0
Age           175
SibSp           0
Parch           0
Fare            0
Cabin         694
Embarked        2
Survived        0
FamilySize      0
dtype: int64
```

In [184...
```
# Fill missing 'Age' with median
df['Age'].fillna(df['Age'].median(), inplace=True)

# Fill missing 'Embarked' with most frequent value
df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)

# Drop 'Cabin' column since it has too many missing values
df.drop(columns=['Cabin'], inplace=True)
```

In [185...
```
df.isna().sum()
```

Out[185…
```
Pclass        0
Sex           0
Age           0
SibSp         0
Parch         0
Fare          0
Embarked      0
Survived      0
FamilySize    0
dtype: int64
```

In [186…
```
df.duplicated().sum()
```

Out[186…
```
122
```

In [188…
```
df.drop_duplicates(inplace=True)
```

In [189…
```
df.duplicated().sum()
```

Out[189…
```
0
```

In [190…
```
df
```

Out[190…

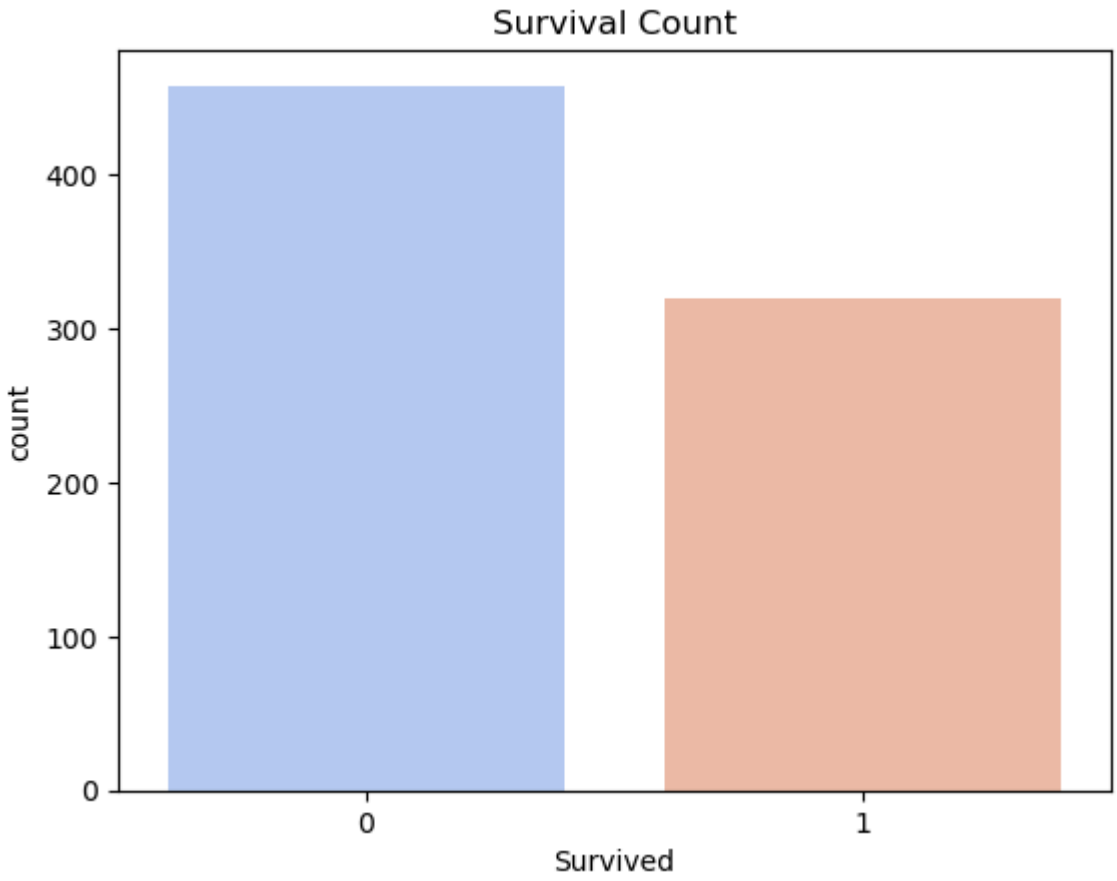
| | Pclass | Sex | Age | SibSp | Parch | Fare | Embarked | Survived | FamilySize |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 3 | male | 22.0 | 1 | 0 | 7.2500 | S | 0 | 1 |
| **1** | 1 | female | 38.0 | 1 | 0 | 71.2833 | C | 1 | 1 |
| **2** | 3 | female | 26.0 | 0 | 0 | 7.9250 | S | 1 | 0 |
| **3** | 1 | female | 35.0 | 1 | 0 | 53.1000 | S | 1 | 1 |
| **4** | 3 | male | 35.0 | 0 | 0 | 8.0500 | S | 0 | 0 |
| **…** | … | … | … | … | … | … | … | … | … |
| **887** | 1 | female | 19.0 | 0 | 0 | 30.0000 | S | 1 | 0 |
| **888** | 3 | female | 28.0 | 1 | 2 | 23.4500 | S | 0 | 3 |
| **889** | 1 | male | 26.0 | 0 | 0 | 30.0000 | C | 1 | 0 |
| **890** | 3 | male | 32.0 | 0 | 0 | 7.7500 | Q | 0 | 0 |
| **894** | 3 | male | 28.0 | 0 | 0 | 8.4583 | Q | 0 | 0 |

778 rows × 9 columns

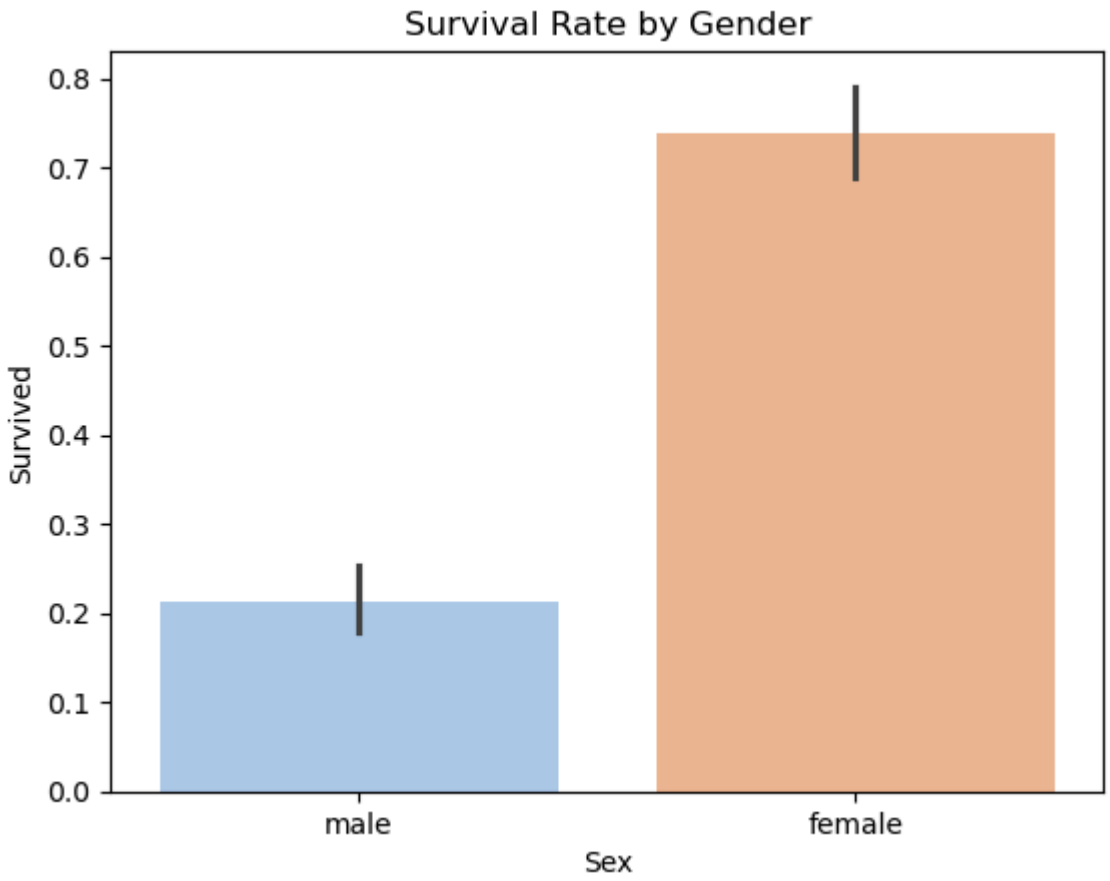
In [191…
```
df.dtypes
```

Out[191…
```
Pclass          int64
Sex            object
Age           float64
SibSp           int64
Parch           int64
Fare          float64
Embarked       object
Survived        int64
FamilySize      int64
dtype: object
```

In [195…
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.countplot(x=df['Survived'], palette='coolwarm')
plt.title("Survival Count")
plt.show()
```

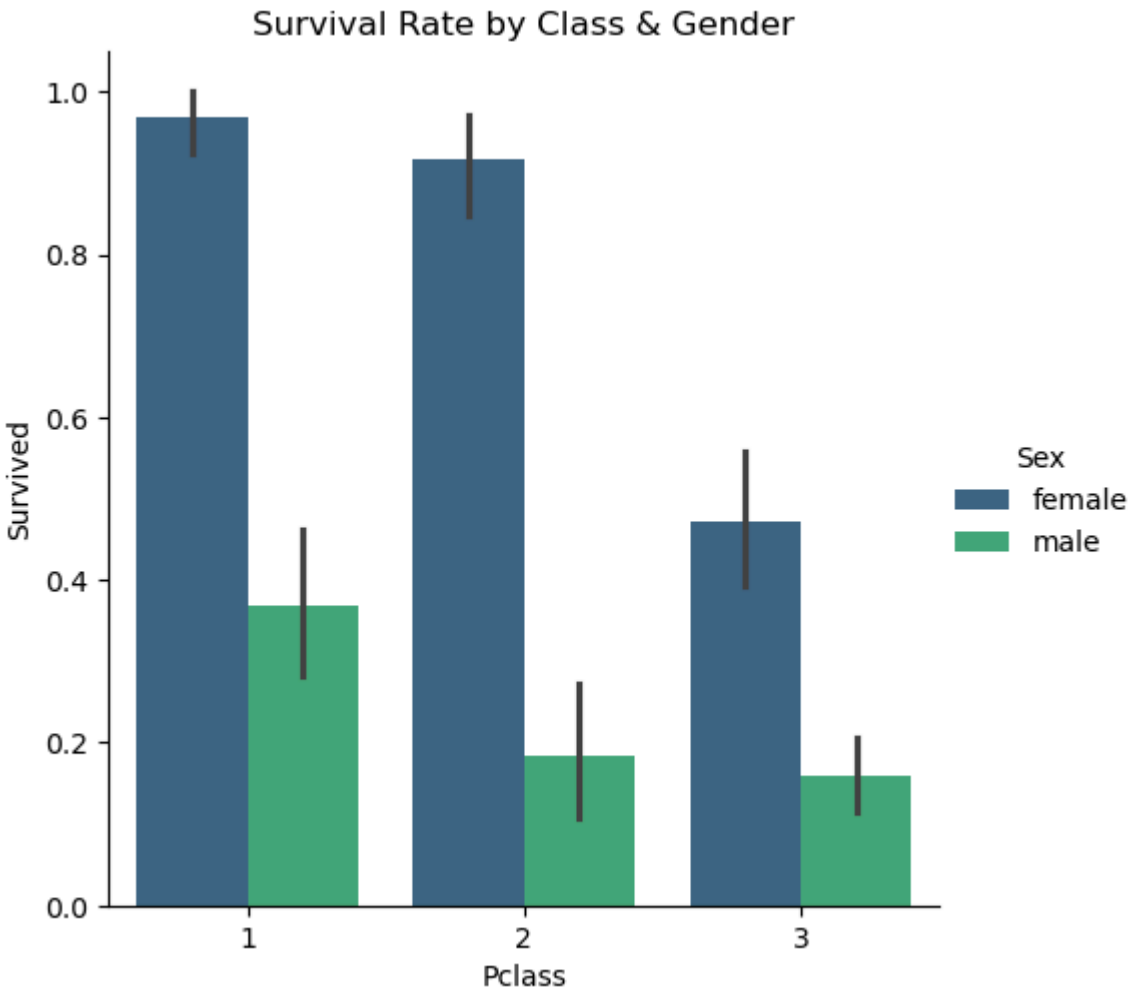



In [196…
```
sns.barplot(x=df['Sex'], y=df['Survived'], palette='pastel')
plt.title("Survival Rate by Gender")
plt.show()
```




In [218…

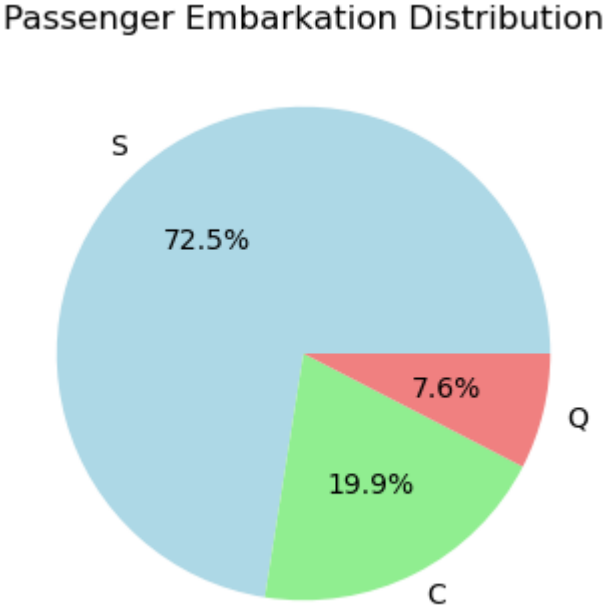
```
sns.catplot(x='Pclass', y='Survived', hue='Sex', data=df, kind='bar', palette='v
plt.title("Survival Rate by Class & Gender")
plt.show()
```




**Q1. How many passengers embarked from each location?**

In [215…

```
plt.figure(figsize=(6,4))
df['Embarked'].value_counts().plot.pie(autopct='%.1f%%', colors=['lightblue', 'l
plt.title("Passenger Embarkation Distribution")
plt.ylabel("")
plt.show()
```

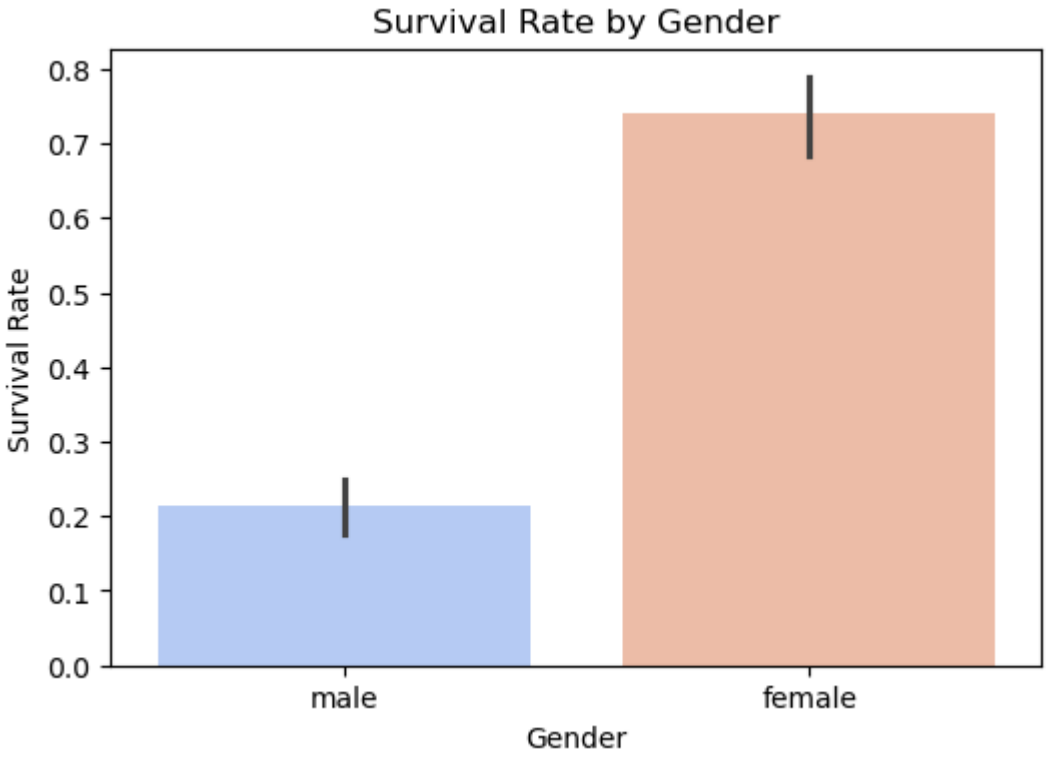


**72.5% of the passengers embarked from Southampton, 19.9% from Cherbourg, and 7.6% from Queenstown.**

**Q2. Which gender had a higher survival rate?**

In [203…

```
plt.figure(figsize=(6, 4))
sns.barplot(x=df['Sex'], y=df['Survived'], palette='coolwarm')
plt.title("Survival Rate by Gender")
plt.xlabel("Gender")
plt.ylabel("Survival Rate")
plt.show()
```

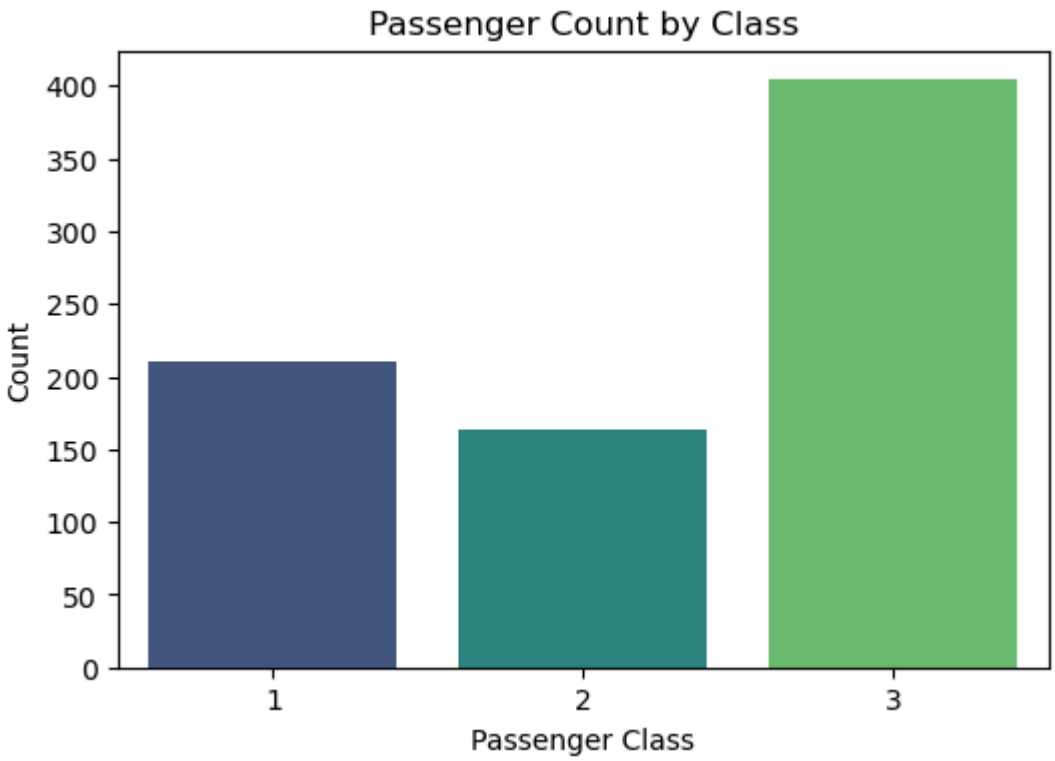



**Females had a higher survival rate than males.**

**Q3. What is the distribution of passengers across different classes?**

In [208…

```
plt.figure(figsize=(6,4))
sns.countplot(x=df['Pclass'], palette='viridis')
plt.title("Passenger Count by Class")
plt.xlabel("Passenger Class")
plt.ylabel("Count")
plt.show()
```

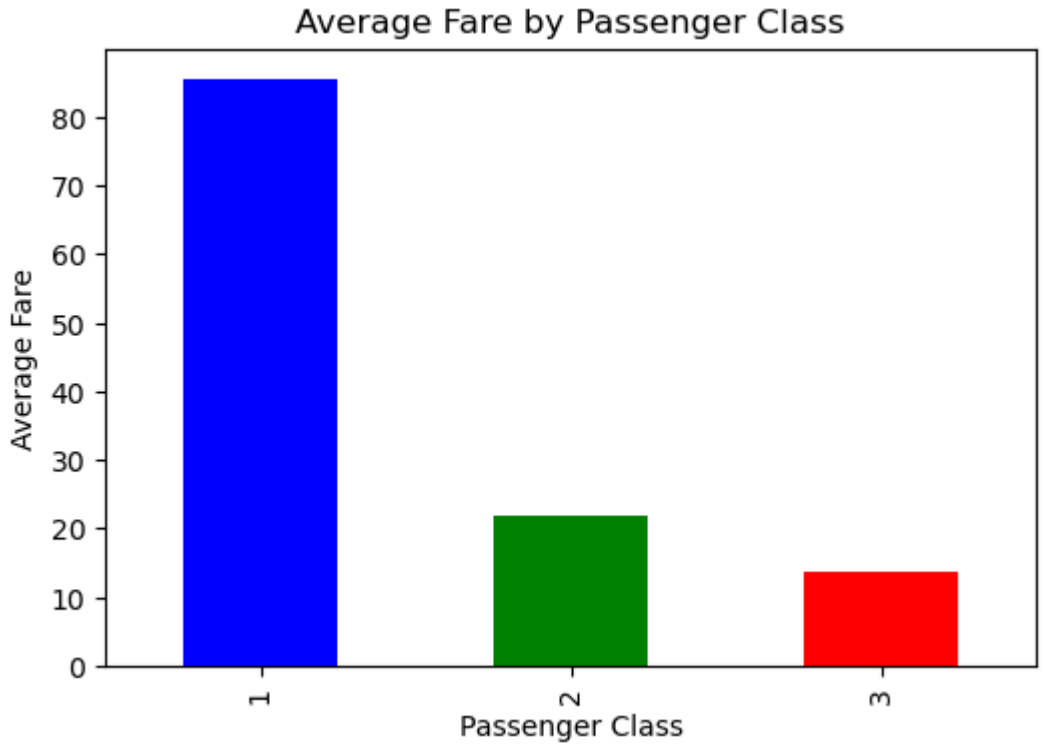



**The number of passengers in Class 3 is higher compared to Class 1 and Class 2.**

**Q4. What is the average fare paid by each class?**

In [210…

```
plt.figure(figsize=(6,4))
df.groupby('Pclass')['Fare'].mean().plot(kind='bar', color=['blue', 'green', 're
plt.title("Average Fare by Passenger Class")
plt.xlabel("Passenger Class")
plt.ylabel("Average Fare")
plt.show()
```

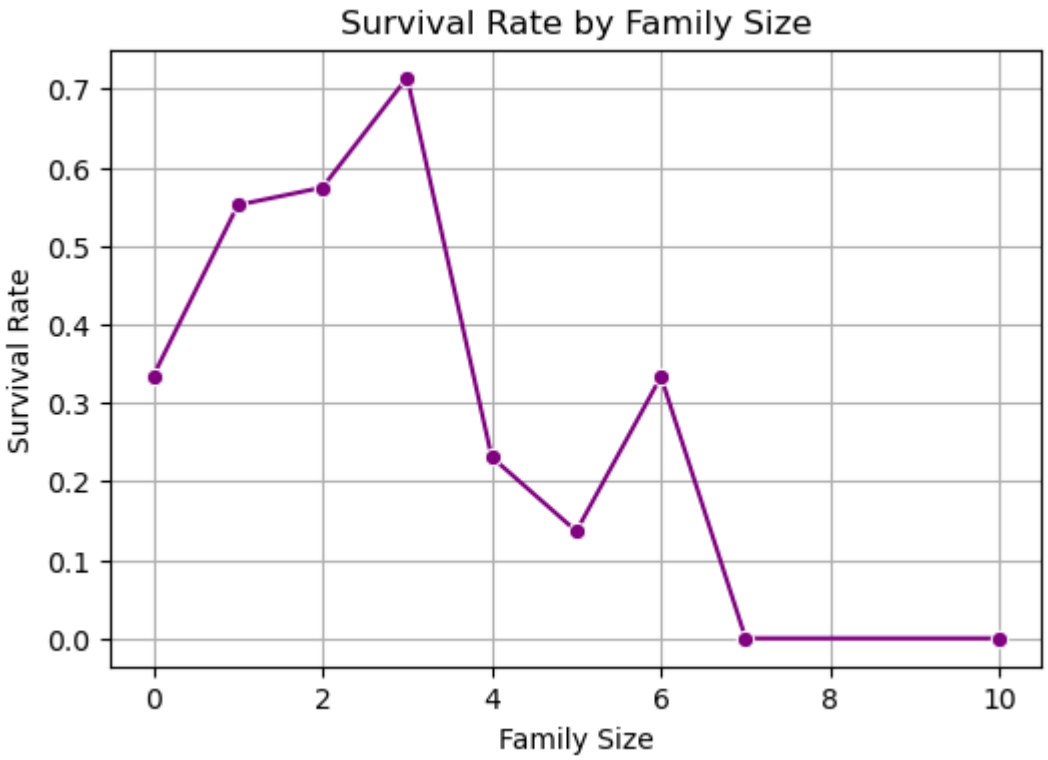


**The average fare paid by Class 1 passengers is around 85, for Class 2 it is around 20, and for Class 3 it is around 10.**

**Q5. What is the survival rate based on family size?**

In [214…

```
survival_by_family = df.groupby('FamilySize')['Survived'].mean().reset_index()
plt.figure(figsize=(6,4))
sns.lineplot(x=survival_by_family['FamilySize'], y=survival_by_family['Survived'
plt.title("Survival Rate by Family Size")
plt.xlabel("Family Size")
plt.ylabel("Survival Rate")
plt.grid()
plt.show()
```

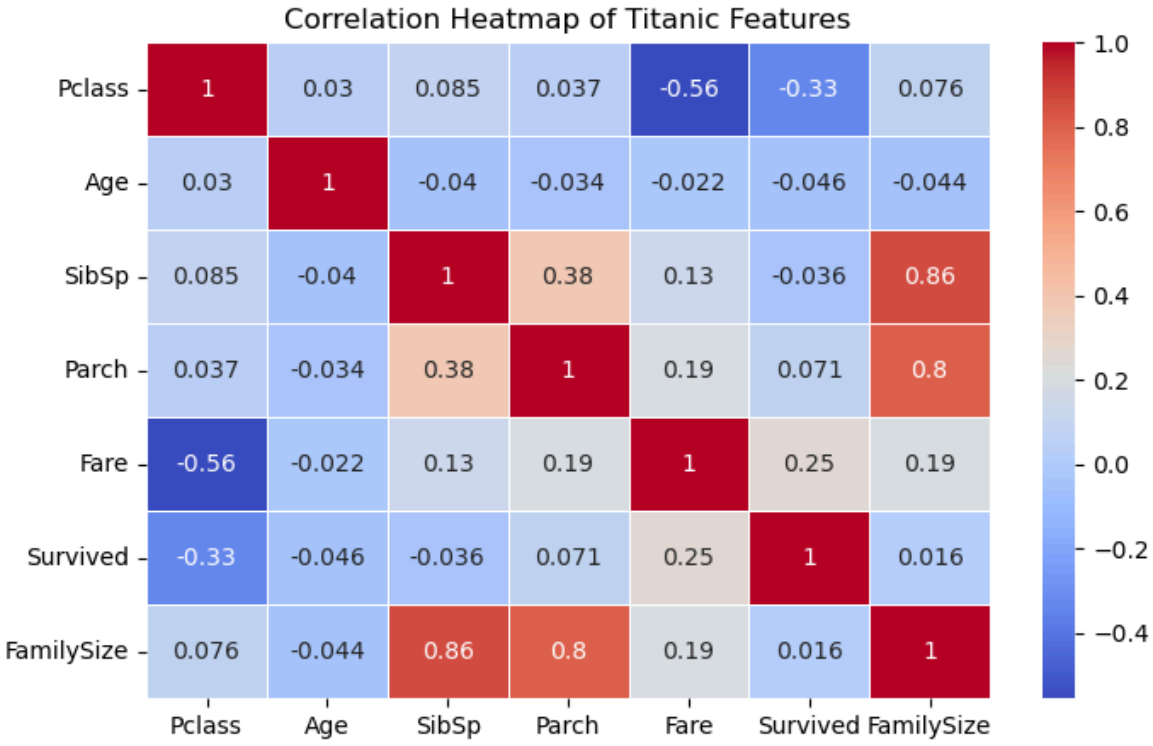



**The survival rate increases as family size increases from 1 to around 3, peaking at 0.7. However, as family size exceeds 3, the survival rate declines, reaching almost 0 for larger families of 7 or more.**

**Q6. What is the correlation between numerical features?**

In [216…

```
plt.figure(figsize=(8,5))
sns.heatmap(df.corr(numeric_only = True), annot=True, cmap='coolwarm', linewidth
plt.title("Correlation Heatmap of Titanic Features")
plt.show()
```




- **Negative correlation between Pclass and Fare (-0.56) means that passengers in higher classes paid more for their tickets.**
- **There is a moderate positive correlation (0.38) between SibSp (number of siblings/spouses) and Parch (number of parents/children).**
- **Age has very weak correlations with all other variables.**

---