> *To import data into dataframe without headers, include parameter "header=None" in pd.read_csv()*

```
auto = pd.read_csv(path,header=None)
```

Maximum of a dataframe column

```
maxs = auto[0].max()
```

Frequency of values in a df column

```
freq = auto[0].value_counts()
```

Sum of a df column

```
sales = auto[0].sum()
```

Mean, Median and Mode of data

> *Can use numpy to find mean and median of a df column*

```
salesmean = np.mean(auto[0])
salesmed = np.median(auto[0])
salesfreq = np.bincount(auto[0]) # gives the frequency of the data
salesmode = salesfreq.argmax()
```

Quartiles of the data

```
firstquart = np.quantile(auto[0],0.25)
thirdquart = np.quantile(auto[0],0.75)
```

Boxplot

```
import matplotlib.pyplot as plt
plt.boxplot(auto[0])
```

Shape

```
auto.shape()
```

Nulls

```
auto[0].isnull().sum()
```

Histogram

```
fig = plt.figure(figsize =(10, 7))
plt.hist(auto[0],bins=10,color=colors,label=labels)
plt.legend()
plt.title("Hist Plot")
plt.show()
```

Boxplot

```
plt.boxplot(auto[0])
```

Piechart

```
plt.pie(auto[0])
```

Scatterplot

```
plt.scatter(x1, y1, color='blue', label='Group 1')
plt.scatter(x2, y2, color='red', label='Group 2')
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')
```

Swarmplot

> *A swarm plot is a type of data visualization used in statistics and data analysis to display the distribution of a dataset. It is similar to a scatter plot but is specifically designed for categorical data. hue is a parameter used to add an additional dimension to a plot by coloring data points based on a categorical or numerical variable.*

```
import seaborn as sb
sb.swarmplot(x='Sepal Width',y='Sepal Length',hue='Category',data=iris)
plt.title('Scatter plots based on the feature sepal width and sepal length')
plt.show()
```

Density Plot

> *Density plots (also known as kernel density plots or KDE plots) are a type of data visualization used to represent the distribution of a continuous variable. They are particularly useful for understanding the underlying probability density function of the data.*

```
sb.kdeplot(iris['Sepal Width'],hue='Category')
plt.title('Density plot for sepal width')
```

Violin Plot

> *A violin plot is a data visualization tool that combines the features of a box plot and a density plot. It is used to visualize the distribution of a numeric variable across different categories or groups. Violin plots are excellent for comparing the distributions of a numeric variable across different categories or groups.*

```
sb.violinplot(x='Category',y='Petal Length',hue='Category',data=iris)
```

Description of data

.describe() method in Pandas only calculates summary statistics for numerical columns in a DataFrame. However, you can customize its behavior to include categorical (non-numeric) columns as well.

```
# Default .describe()
summary = df.describe()
summary = df.describe(include='all') # includes categorical columns as well
```

Creating scatter plots of all possible combinations of numerical attributes in the given dataset

```
df.pairplot(df[column])
```

Box Plot

```
sb.boxplot(data=df)
```

Cat Plot

The catplot() function in Seaborn is a versatile and high-level interface for creating categorical plots. It allows you to easily generate various types of visualizations for categorical data, such as bar plots, box plots, violin plots, swarm plots, and more.

```
sb.catplot(data=df,kind="")
```

Correlation Matrix

```
cm = df.corr()
```

Heatmap

```
sb.heatmap(cm, annot=True, cmap='coolwarm', fmt=".2f")
```

Imputation

Numerical

1. Mean Imputation

```
df[col].fillna(df[col].mean(), inplace=True)
```

2. Median

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

3. Regression

> *Assuming Linearity*

```
from sklearn.linear_model import LinearRegression

# Split into observed and missing data
observed = df[~df['y'].isnull()]
missing = df[df['y'].isnull()]

# Train a linear regression model on observed data
model = LinearRegression()
model.fit(observed[['X']], observed['y'])

# Predict missing values
predicted = model.predict(missing[['X']])

# Fill in missing values
df_imputed = df.copy()
df_imputed.loc[df['y'].isnull(), 'y'] = predicted
```

> *For Non linear Data*

```
from sklearn.tree import DecisionTreeRegressor

# Split into observed and missing data
observed = df[~df['y'].isnull()]
missing = df[df['y'].isnull()]

# Train a decision tree regressor on observed data
model = DecisionTreeRegressor(random_state=42)
model.fit(observed[['X']], observed['y'])

# Predict missing values
predicted = model.predict(missing[['X']])

# Fill in missing values
df_imputed = df.copy()
df_imputed.loc[df['y'].isnull(), 'y'] = predicted
```

Categorical

1. Mode

```
df['col'] = df['col'].fillna(df['col'].mode())
```

2. Logistic Regression

```
from sklearn.linear_model import LogisticRegression

# Split into observed and missing data
observed = df.dropna()
missing = df[df['col'].isnull()]

# Encode categorical features
encoder = OneHotEncoder()
X_observed = encoder.fit_transform(observed[['X', 'Y']])
y_observed = observed['col']

# Train a classifier (e.g., logistic regression)
model = LogisticRegression()
model.fit(X_observed, y_observed)

# Predict missing values
X_missing = encoder.transform(missing[['X', 'Y']])
predicted = model.predict(X_missing)

# Fill in missing values
df.loc[df['col'].isnull(), 'col'] = predicted
```

3. KNN

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=5)
features = rbnb[['Beds number', 'Maximum allowed guests', 'Bedrooms number']]
imputed_values = imputer.fit_transform(features)
rbnb['Bedrooms number'] = imputed_values[:, 2]
```

Binning

1. Equal Width

```
df["col_binned"] = pd.cut(df["col"], bins=binsnumber)
```

2. Equal Frequency

```
df["col_binned"] = pd.qcut(df["col"], q=numofquantiles)
```

Encoding

1. Label Encoder

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['col']=le.fit_transform(df['col'])
```

2. One hot encoder

```
rbnb = pd.get_dummies(rbnb, columns=['Season'])
```

Normalization

1. Min-Max Normalization

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data
df['col_transformed'] = scaler.fit_transform(df[['col']])
```

2. Z - Score Normalization

```
from sklearn.preprocessing import StandardScaler

# Initialize StandardScaler
scaler = StandardScaler()

# Fit and transform the data
df['col_transformed'] = scaler.fit_transform(df[['col']])
```

3. Normalization by Decimal Scaling

```
import numpy as np

# Find the maximum absolute value
max_abs_value = np.max(np.abs(df['Values']))

# Calculate j (number of digits in the maximum absolute value)
j = len(str(int(max_abs_value)))

# Normalize by decimal scaling
df['col_transformed'] = df['col'] / (10 ** j)
```