"Pandas Notes"

"1.Introduction to Pandas"

What is Pandas?

```
* Pandas is a Python library used for working with data sets.
* It has functions for analysing, cleaning,exploring,and manipulating data.
* The name "Pandas" has a reference to both "Panel Data" and "Python Data Analysis" and was
created by Wes McKinney in 2008.
```

why is Pandas?

```
* Pandas allows us to analyze big data and make some conclusions based on stastical thoeries.
* Pandas can clean some messy data sets and make them readble and relevent.
* Relevant data is very important in data science
```

What can Pandas do?

```
Pandas gives you answers about the data like:

* Is there correlation between two or more columns?
* what is average value?
* MAx value?
* Min value?
```

- Pandas are also able to delete rows that are not relevant or contains wrong values,like empty or NULL values.This is called "cleaning the data."

Installation:

pip install pandas

2.Key Data Structures

* Series :

```
* Defination :-> A one -dimentional labeled array capble of holding any data type (e.g
integers,strings,floats).
```

Creating A Series:

In [13]:
```
import pandas as pd

s=pd.Series([1,2,3,"vasu",5],index=['a','b','c','d','e'])
print(s)
```

```
a       1
b       2
c       3
d    vasu
e       5
dtype: object
```

* DataFrame:

```
* Defination : A two-dimentional labeled data structure with columns of potentially different
types,similar to a spreadsheet or SQL table.
```

Creating a DataFrame:

In [278…
```
data={
    'Name':["vasu","sai","ravi","mani","vamsi"],
    'Age':[21,21,20,20,21],
    'Gender':["Male","Female","Male","Male","Female"],
    'Location':["Andhra","Mumbai","Banglore","Hyderabad","Chennai"],
    "Salary":[10000,200000,200000,20000,29900]
}
```

```
data
```

Out[278…
```
{'Name': ['vasu', 'sai', 'ravi', 'mani', 'vamsi'],
 'Age': [21, 21, 20, 20, 21],
 'Gender': ['Male', 'Female', 'Male', 'Male', 'Female'],
 'Location': ['Andhra', 'Mumbai', 'Banglore', 'Hyderabad', 'Chennai'],
 'Salary': [10000, 200000, 200000, 20000, 29900]}
```

In [280…
```
df=pd.DataFrame(data)
```

In [234…
```
df
```

Out[234…

	Name	Age	Gender	Location	Salary
0	vasu	21	Male	Andhra	10000
1	sai	21	Female	Mumbai	200000
2	ravi	20	Male	Banglore	200000
3	mani	20	Male	Hyderabad	20000
4	vamsi	21	Female	Chennai	29900

3.DataFrame Manipulations

* Accessing Data:

* By Column:

In [19]:
```
print(df["Name"])
```

```
0     vasu
1      sai
2     ravi
3     mani
4    vamsi
Name: Name, dtype: object
```

* By Row:

In [21]:
```
print(df.loc[0]) #First Row
```

```
Name            vasu
Age               21
Gender          Male
Location      Andhra
Salary         10000
Name: 0, dtype: object
```

Filtering Data:

In [23]:
```
filtered_df=df[df["Age"]>20]
print(filtered_df)
```

```
    Name  Age  Gender Location  Salary
0   vasu   21    Male   Andhra   10000
1    sai   21  Female   Mumbai  200000
4  vamsi   21  Female  Chennai   29900
```

In [25]:
```
filtered_df=df[df["Age"]==20]
print(filtered_df)
```

```
   Name  Age Gender   Location  Salary
2  ravi   20   Male   Banglore  200000
3  mani   20   Male  Hyderabad   20000
```

In [27]:
```
filtered_df=df[df["Age"]>=20]
print(filtered_df)
```

```
    Name  Age  Gender   Location  Salary
0   vasu   21    Male     Andhra   10000
1    sai   21  Female     Mumbai  200000
2   ravi   20    Male   Banglore  200000
3   mani   20    Male  Hyderabad   20000
4  vamsi   21  Female    Chennai   29900
```

Adding New Columns:

In [282…
```
df['Department']=["Data Scientist","Data Analyst","Powerbi Analyst","SQL Developer","Python developer"]
print(df)
```

```
    Name  Age  Gender   Location  Salary        Department
0   vasu   21    Male    Andhra    10000    Data Scientist
1    sai   21  Female    Mumbai   200000      Data Analyst
2   ravi   20    Male  Banglore   200000   Powerbi Analyst
3   mani   20    Male Hyderabad    20000     SQL Developer
4  vamsi   21  Female    Chennai    29900  Python developer
```

4.Handling Missing Data:

* Identifying Missing Values:

In [31]:
```
df.loc[1,"Name"]=None # Set Bob's age to None
# print(df)
# print(df.isnull())  # Output: DataFrame of True/False for null values
df
```

Out[31]:

	Name	Age	Gender	Location	Salary	Department
0	vasu	21	Male	Andhra	10000	Data Scientist
1	None	21	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	20	Male	Hyderabad	20000	SQL Developer
4	vamsi	21	Female	Chennai	29900	Python developer

In [33]:
```
df
```

Out[33]:

	Name	Age	Gender	Location	Salary	Department
0	vasu	21	Male	Andhra	10000	Data Scientist
1	None	21	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	20	Male	Hyderabad	20000	SQL Developer
4	vamsi	21	Female	Chennai	29900	Python developer

* Filling Missing Values:

In [53]:
```
df["Name"].fillna(df['Name'].mode(),inplace=True) #We have to give inplace=True for older versions
df
```

Out[53]:

	Name	Age	Gender	Location	Salary	Department
0	vasu	21	Male	Andhra	10000	Data Scientist
1	ravi	21	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	20	Male	Hyderabad	20000	SQL Developer
4	vamsi	21	Female	Chennai	29900	Python developer

In [55]:
```
df.loc[2,"Age"]=None
df
```

Out[55]:

	Name	Age	Gender	Location	Salary	Department
0	vasu	21.0	Male	Andhra	10000	Data Scientist
1	ravi	21.0	Female	Mumbai	200000	Data Analyst
2	ravi	NaN	Male	Banglore	200000	Powerbi Analyst
3	mani	20.0	Male	Hyderabad	20000	SQL Developer
4	vamsi	21.0	Female	Chennai	29900	Python developer

In [108…
```
df["Age"].fillna(df['Age'].mean()) #We have to give inplace=True for older versions
df
```

Out[108…

	Name	Age	Gender	Location	Salary	Department
0	vasu	23	Male	Andhra	10000	Data Scientist
1	ravi	23	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	23	Male	Hyderabad	20000	SQL Developer
4	vamsi	23	Female	Chennai	29900	Python developer

In [120…

```
df.loc[1:2,"Age"]=None
# df["Age"].fillna(df["Age"].mode())
# df
```

In [122…

```
df.loc[2,"Age"]=20
```

In [124…

```
df
```

Out[124…

	Name	Age	Gender	Location	Salary	Department
0	vasu	23.0	Male	Andhra	10000	Data Scientist
1	ravi	NaN	Female	Mumbai	200000	Data Analyst
2	ravi	20.0	Male	Banglore	200000	Powerbi Analyst
3	mani	23.0	Male	Hyderabad	20000	SQL Developer
4	vamsi	23.0	Female	Chennai	29900	Python developer

In [153…

```
df["Age"].fillna(df["Age"].median())
df
```

Out[153…

	Name	Age	Gender	Location	Salary	Department
0	vasu	23.0	Male	Andhra	10000	Data Scientist
1	ravi	NaN	Female	Mumbai	200000	Data Analyst
2	ravi	NaN	Male	Banglore	200000	Powerbi Analyst
3	mani	23.0	Male	Hyderabad	20000	SQL Developer
4	vamsi	23.0	Female	Chennai	29900	Python developer

In [155…

```
df
```

Out[155…

	Name	Age	Gender	Location	Salary	Department
0	vasu	23.0	Male	Andhra	10000	Data Scientist
1	ravi	NaN	Female	Mumbai	200000	Data Analyst
2	ravi	NaN	Male	Banglore	200000	Powerbi Analyst
3	mani	23.0	Male	Hyderabad	20000	SQL Developer
4	vamsi	23.0	Female	Chennai	29900	Python developer

*Droping Missing Values

In [187…

```
df.loc[1:2,"Age"]=None

df
```

Out[187…

	Name	Age	Gender	Location	Salary	Department
0	vasu	23.0	Male	Andhra	10000	Data Scientist
1	ravi	NaN	Female	Mumbai	200000	Data Analyst
2	ravi	NaN	Male	Banglore	200000	Powerbi Analyst
3	mani	23.0	Male	Hyderabad	20000	SQL Developer
4	vamsi	23.0	Female	Chennai	29900	Python developer

In [149…

```
df.dropna()# Drop rows with NaN values
```

Out[149…

	Name	Age	Gender	Location	Salary	Department
0	vasu	23.0	Male	Andhra	10000	Data Scientist
3	mani	23.0	Male	Hyderabad	20000	SQL Developer
4	vamsi	23.0	Female	Chennai	29900	Python developer

5.Data Adding Multiple to records

In [189…
```
df.loc[1:2,"Age"]=21
```

In [238…
```
df
```

Out[238…

	Name	Age	Gender	Location	Salary	Department
0	vasu	21	Male	Andhra	10000	Data Scientist
1	sai	21	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	20	Male	Hyderabad	20000	SQL Developer
4	vamsi	21	Female	Chennai	29900	Python developer

In [290…
```
df.loc[len(df)] = ["parasuram", 27, "Male", "Banglore", 10000, "Data Scientist"]
```

In [292…
```
df
```

Out[292…

	Name	Age	Gender	Location	Salary	Department
0	vasu	21	Male	Andhra	10000	Data Scientist
1	sai	21	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	20	Male	Hyderabad	20000	SQL Developer
4	vamsi	21	Female	Chennai	29900	Python developer
5	ashwinth	27	Male	Banglore	10000	Data Scientist
6	ashwinth	27	Male	Banglore	10000	Data Scientist
7	ravi	27	Male	Banglore	10000	Data Scientist
8	parasuram	27	Male	Banglore	10000	Data Scientist

In [288…
```
# New data to be added as a DataFrame
new_data = pd.DataFrame([["ravi", 27, "Male", "Banglore", 10000, "Data Scientist"]],
                        columns=["Name", "Age", "Gender", "Location", "Salary", "Department"])

# Concatenating the new row to the existing DataFrame
df = pd.concat([df, new_data], ignore_index=True)

df
```

Out[288…

	Name	Age	Gender	Location	Salary	Department
0	vasu	21	Male	Andhra	10000	Data Scientist
1	sai	21	Female	Mumbai	200000	Data Analyst
2	ravi	20	Male	Banglore	200000	Powerbi Analyst
3	mani	20	Male	Hyderabad	20000	SQL Developer
4	vamsi	21	Female	Chennai	29900	Python developer
5	ashwinth	27	Male	Banglore	10000	Data Scientist
6	ashwinth	27	Male	Banglore	10000	Data Scientist
7	ravi	27	Male	Banglore	10000	Data Scientist

6.Data Aggregation and Grouping

*Group By:

In [5]:
```
import pandas as pd

# Example DataFrame
data = {
    "Name": ["John", "Sara", "Mike", "Ashley", "Tom", "Vamsi"],
```

```
    "Age": [25, 30, 35, 32, 28, 12],  # 'saivamsi' is a string
    "Gender": ["Male", "Female", "Male", "Female", "Male", "Female"],
    "Location": ["New York", "Los Angeles", "Chicago", "New York", "Los Angeles", "Chicago"],
    "Salary": [50000, 60000, 70000, 65000, 62000, 58000],
    "Department": ["HR", "Finance", "Engineering", "Finance", "Data Scientist", "Marketing"]
}

df = pd.DataFrame(data)

# Check data types
print(df.dtypes)
```

```
Name          object
Age            int64
Gender        object
Location      object
Salary         int64
Department    object
dtype: object
```

2. Identify and Handle Non-Numeric Data

    Use pd.to_numeric() to convert columns to numeric types, coercing errors (i.e., non-convertible values) to NaN.

In [7]:
```
# Convert 'Age' and 'Salary' to numeric, coercing errors to NaN
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')
df['Salary'] = pd.to_numeric(df['Salary'], errors='coerce')
```

In [9]:
```
df["Age"].fillna(df["Age"].mean())
df
```

Out[9]:

	Name	Age	Gender	Location	Salary	Department
0	John	25	Male	New York	50000	HR
1	Sara	30	Female	Los Angeles	60000	Finance
2	Mike	35	Male	Chicago	70000	Engineering
3	Ashley	32	Female	New York	65000	Finance
4	Tom	28	Male	Los Angeles	62000	Data Scientist
5	Vamsi	12	Female	Chicago	58000	Marketing

In [56]:
```
# Identify rows where 'Age' conversion failed
invalid_age = df[df['Age'].isna()]
print("Rows with invalid 'Age':")
print(invalid_age)
```

```
Rows with invalid 'Age':
Empty DataFrame
Columns: [Name, Age, Gender, Location, Salary, Department]
Index: []
```

In [58]:
```
# Drop rows with NaN in 'Age'
df_clean = df.dropna(subset=['Age'])
```

In [60]:
```
# Fill NaN in 'Age' with the mean age
mean_age = df['Age'].mean()
df['Age'].fillna(mean_age)
```

Out[60]:
```
0    25
1    30
2    35
3    32
4    28
5    12
Name: Age, dtype: int64
```

In [62]:
```
# Drop rows with NaN in 'Age'
df_clean = df.dropna(subset=['Age'])

print("Cleaned DataFrame:")
print(df_clean)
```

```
Cleaned DataFrame:
     Name  Age  Gender     Location  Salary      Department
0    John   25    Male     New York   50000              HR
1    Sara   30  Female  Los Angeles   60000         Finance
2    Mike   35    Male      Chicago   70000     Engineering
3  Ashley   32  Female     New York   65000         Finance
4     Tom   28    Male  Los Angeles   62000  Data Scientist
5   Vamsi   12  Female      Chicago   58000       Marketing
```

In [64]:
```
# Group by 'Gender' and calculate mean of 'Age' and 'Salary'
grouped_data = df_clean.groupby('Gender')[['Age', 'Salary']].mean()

print("Grouped Data (Mean Age and Salary by Gender):")
print(grouped_data)
```

```
Grouped Data (Mean Age and Salary by Gender):
              Age        Salary
Gender
Female  24.666667  61000.000000
Male    29.333333  60666.666667
```

In [96]:
```
grouped_data = df_clean.groupby('Gender')[["Age","Salary"]].mean()
print(grouped_data)
```

```
              Age        Salary
Gender
Female  24.666667  61000.000000
Male    29.333333  60666.666667
```

In [68]:
```
grouped_data = df_clean.groupby(['Gender', 'Location'])[["Age","Salary"]].mean()
print(grouped_data)
```

```
                     Age   Salary
Gender Location
Female Chicago      12.0  58000.0
       Los Angeles  30.0  60000.0
       New York     32.0  65000.0
Male   Chicago      35.0  70000.0
       Los Angeles  28.0  62000.0
       New York     25.0  50000.0
```

In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

In [42]:
```
df.head(10)
```

Out[42]:

	Name	Age	Gender	Location	Salary	Department
0	John	25	Male	New York	50000	HR
1	Sara	30	Female	Los Angeles	60000	Finance
2	Mike	35	Male	Chicago	70000	Engineering
3	Ashley	32	Female	New York	65000	Finance
4	Tom	28	Male	Los Angeles	62000	Data Scientist
5	Vamsi	12	Female	Chicago	58000	Marketing

Aggregation Data:

In [30]:
```
aggregation=df.agg({'Age':'mean','Salary':'sum'})
print(aggregation)
```

```
Age           27.0
Salary    365000.0
dtype: float64
```

In [34]:
```
aggregation=df.agg({'Age':'min','Salary':'max'})
print(aggregation)
```

```
Age          12
Salary    70000
dtype: int64
```

In [36]:
```
aggregation=df.agg({'Age':'min','Salary':'min'})
print(aggregation)
```

```
Age          12
Salary    50000
dtype: int64
```

In [46]:
```
aggregation=df.agg({'Age':'median','Salary':'median'})
```

```
print(aggregation)
```

```
Age          29.0
Salary    61000.0
dtype: float64
```

In [76]:
```
import pandas as pd

# Sample DataFrame
data = {
    'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago', 'Los Angeles'],
    'Population': [8000000, 4000000, 2700000, 8100000, 2750000, 4100000],
    'Area': [468.9, 503, 234, 468.9, 234, 503]
}

df = pd.DataFrame(data)
df
```

Out[76]:

	City	Population	Area
0	New York	8000000	468.9
1	Los Angeles	4000000	503.0
2	Chicago	2700000	234.0
3	New York	8100000	468.9
4	Chicago	2750000	234.0
5	Los Angeles	4100000	503.0

In [82]:
```
# Group by 'City' and aggregate
grouped = df.groupby('City').agg({
    'Population': 'sum',   # Total population per city
    'Area': 'mean'         # Average area per city
})

print(grouped)
```

```
             Population   Area
City
Chicago         5450000  234.0
Los Angeles     8100000  503.0
New York       16100000  468.9
```

In [108…
```
grouped = df.groupby('City').agg({
    'Population': ['sum', 'mean'],  # Apply sum and mean to 'Population'
    'Area': ['min', 'max']           # Apply min and max to 'Area'
})

print(grouped)
```

```
            Population              Area
                   sum       mean    min    max
City
Chicago        5450000  2725000.0  234.0  234.0
Los Angeles    8100000  4050000.0  503.0  503.0
New York      16100000  8050000.0  468.9  468.9
```

In [110…
```
# Define custom function: return the range (max - min)
def range_func(x):
    return x.max() - x.min()

grouped = df.groupby('City').agg({
    'Population': 'sum',
    'Area': range_func  # Apply custom function to calculate range of Area
})

print(grouped)
```

```
             Population  Area
City
Chicago         5450000   0.0
Los Angeles     8100000   0.0
New York       16100000   0.0
```

Explanation:

- Custom Function (range_func): Returns the range (difference between max and min values) of "Area".
- Since each city has only one unique value for "Area", the range is 0.0.

In [ ]:
```
4. Aggregating Across All Columns
```

In [119…

```
# Aggregate across all columns
grouped = df.groupby('City').sum()

print(grouped)
```

```
             Population    Area
City
Chicago         5450000   468.0
Los Angeles     8100000  1006.0
New York       16100000   937.8
```

In [ ]:

```
5. Combining Aggregation with Filtering
```

In [132…

```
# Filter before aggregation
filtered_df = df[df['Population'] > 5000000]

grouped = filtered_df.groupby('City').agg({
    'Population': 'sum',
    'Area': 'mean'
})

grouped
```

Out[132…

	Population	Area
City		
New York	16100000	468.9

Explanation:

- The data is first filtered to include only rows where "Population" is greater than 5 million, and then the aggregation is performed.

In [ ]:

```
6. Applying Multiple Aggregation Functions to All Columns
```

In [126…

```
grouped=df.groupby('City').agg(['mean','sum','count'])
grouped
```

Out[126…

	Population			Area		
	mean	**sum**	**count**	**mean**	**sum**	**count**
City						
Chicago	2725000.0	5450000	2	234.0	468.0	2
Los Angeles	4050000.0	8100000	2	503.0	1006.0	2
New York	8050000.0	16100000	2	468.9	937.8	2

In [ ]:

```
7. Named Aggregation (for Clarity)
```

In [134…

```
grouped=df.groupby('City').agg(
    total_population=('Population','sum'),
    avg_area=('Area','mean')
)

grouped
```

Out[134…

	total_population	avg_area
City		
Chicago	5450000	234.0
Los Angeles	8100000	503.0
New York	16100000	468.9

In [152…

```
grouped=df.groupby('City').agg({
    'Population':'sum',
    'Area':'mean'
}).reset_index()
grouped_sorted=grouped.sort_values(by='Population',ascending=False)
grouped_sorted
```

Out[152...

	City	Population	Area
2	New York	16100000	468.9
1	Los Angeles	8100000	503.0
0	Chicago	5450000	234.0

```
Explanation:
```

- After grouping and aggregating, we reset the index and then sort the result based on the total population in descending orde

```
*   Summary of Key Functions:
```

- sum(): Sums the values in each group.
- mean(): Calculates the average value for each group.
- count(): Counts the number of non-null values for each group.
- `min() / max(): Finds the minimum / maximum value in each group.`
- `agg(): Applies multiple aggregation functions to one or more columns.`
- `Custom Functions: Allows the use of custom aggregation logic (e.g., lambda functions).`

6.Merging and Joining DataFrames

* Concatenation:

In [159...

```
df
```

Out[159...

	City	Population	Area
0	New York	8000000	468.9
1	Los Angeles	4000000	503.0
2	Chicago	2700000	234.0
3	New York	8100000	468.9
4	Chicago	2750000	234.0
5	Los Angeles	4100000	503.0

In [163...

```
df2=pd.DataFrame({'City':['India'],'Population':[140000000],'Area':[1000.23]})

concatenated=pd.concat([df,df2],ignore_index=True)
concatenated
```

Out[163...

	City	Population	Area
0	New York	8000000	468.90
1	Los Angeles	4000000	503.00
2	Chicago	2700000	234.00
3	New York	8100000	468.90
4	Chicago	2750000	234.00
5	Los Angeles	4100000	503.00
6	India	140000000	1000.23

*Merging DataFrames Using merge()

```
* The merge() function is used to combine two DataFrames based on one more common columns pr
indices.By defualt ,'merge()' performs an inner join,meaning it only incudes rows that have
matching keys in both DataFrames
```

In [184...

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({
    'EmployeeID': [101, 102, 103],
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Department': ['HR', 'IT', 'Finance']
})

df2 = pd.DataFrame({
```

```
    'EmployeeID': [101, 102, 104],
    'Salary': [70000, 80000, 65000],
    'Location': ['New York', 'Chicago', 'Los Angeles']
})

# Merge on 'EmployeeID'
merged_df = pd.merge(df1, df2, on='EmployeeID')

merged_df
```

Out[184...

	EmployeeID	Name	Department	Salary	Location
0	101	Alice	HR	70000	New York
1	102	Bob	IT	80000	Chicago

```
Explanation:
```

- Inner Join: The merge() only keeps rows where the EmployeeID is present in both DataFrames (101 and 102).

2. Specifying Join Type (how)

- You can specify different types of joins using the how parameter in merge():
- how='inner' (default): Keeps only rows with keys present in both DataFrames.
- how='left': Keeps all rows from the left DataFrame, with matching rows from the right.
- how='right': Keeps all rows from the right DataFrame, with matching rows from the left.
- how='outer': Keeps all rows from both DataFrames, filling missing values with NaN.

In [193...

```
merged_df=pd.merge(df1,df2,on='EmployeeID',how='left')
merged_df
```

Out[193...

	EmployeeID	Name	Department	Salary	Location
0	101	Alice	HR	70000.0	New York
1	102	Bob	IT	80000.0	Chicago
2	103	Charlie	Finance	NaN	NaN

```
Explanation:
```

- Left Join: Keeps all rows from df1 (the left DataFrame), even if there's no match in df2. Missing values in df2 are filled with NaN.

In [196...

```
merged_df=pd.merge(df1,df2,on='EmployeeID',how='right')
merged_df
```

Out[196...

	EmployeeID	Name	Department	Salary	Location
0	101	Alice	HR	70000	New York
1	102	Bob	IT	80000	Chicago
2	104	NaN	NaN	65000	Los Angeles

- how='right': Keeps all rows from the right DataFrame, with matching rows from the left.

In [199...

```
merged_df=pd.merge(df1,df2,on='EmployeeID',how='inner')
merged_df
```

Out[199...

	EmployeeID	Name	Department	Salary	Location
0	101	Alice	HR	70000	New York
1	102	Bob	IT	80000	Chicago

```
*how='inner' (default): Keeps only rows with keys present in both DataFrames.
```

In [202...

```
merged_df=pd.merge(df1,df2,on='EmployeeID',how='outer')
merged_df
```

Out[202…

	EmployeeID	Name	Department	Salary	Location
0	101	Alice	HR	70000.0	New York
1	102	Bob	IT	80000.0	Chicago
2	103	Charlie	Finance	NaN	NaN
3	104	NaN	NaN	65000.0	Los Angeles

    * how='outer': Keeps all rows from both DataFrames, filling missing values with NaN.

3. Merging on Multiple Columns

    You can merge DataFrames based on multiple columns by passing a list of column names to the on
    parameter.

In [206…

```
# Sample DataFrames with multiple keys
df1 = pd.DataFrame({
    'EmployeeID': [101, 102, 103],
    'Department': ['HR', 'IT', 'Finance'],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'EmployeeID': [101, 101, 103],
    'Department': ['HR', 'HR', 'Finance'],
    'Salary': [70000, 75000, 65000]
})

# Merge on multiple columns
merged_df = pd.merge(df1, df2, on=['EmployeeID', 'Department'])

merged_df
```

Out[206…

	EmployeeID	Department	Name	Salary
0	101	HR	Alice	70000
1	101	HR	Alice	75000
2	103	Finance	Charlie	65000

4. Merging DataFrames with Different Column Names:

    If the column names are different in the two DataFrames, you can use the left_on and right_on
    parameters to specify the corresponding columns.

    Example:

In [209…

```
# Sample DataFrames with different column names
df1 = pd.DataFrame({
    'ID': [101, 102, 103],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'EmployeeID': [101, 102, 104],
    'Salary': [70000, 80000, 65000]
})

# Merge with different column names
merged_df = pd.merge(df1, df2, left_on='ID', right_on='EmployeeID')

merged_df
```

Out[209…

	ID	Name	EmployeeID	Salary
0	101	Alice	101	70000
1	102	Bob	102	80000

5. Merging on Index

    You can merge DataFrames based on the index using the left_index and right_index parameters.

In [217…

```
# Sample DataFrames
df1 = pd.DataFrame({
```

```
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Department': ['HR', 'IT', 'Finance']
}, index=[101, 102, 103])

df2 = pd.DataFrame({
    'Salary': [70000, 80000, 65000],
    'Location': ['New York', 'Chicago', 'Los Angeles']
}, index=[101, 102, 104])

# Merge on index
merged_df = pd.merge(df1, df2, left_index=True, right_index=True)

merged_df
```

Out[217…

	Name	Department	Salary	Location
101	Alice	HR	70000	New York
102	Bob	IT	80000	Chicago

6. Joining DataFrames Using join()

In [232…

```
# Sample DataFrames
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Department': ['HR', 'IT', 'Finance']
}, index=[101, 102, 103])

df2 = pd.DataFrame({
    'Salary': [70000, 80000, 65000],
    'Location': ['New York', 'Chicago', 'Los Angeles']
}, index=[101, 102, 104])

# Left join using join()
joined_df = df2.join(df1,how='outer')

joined_df
```

Out[232…

	Salary	Location	Name	Department
101	70000.0	New York	Alice	HR
102	80000.0	Chicago	Bob	IT
103	NaN	NaN	Charlie	Finance
104	65000.0	Los Angeles	NaN	NaN

7. Concatenating DataFrames Using concat()

The concat() function is used to concatenate DataFrames either vertically (stack rows) or horizontally (add columns).

In [240…

```
# Sample DataFrames
df1 = pd.DataFrame({
    'EmployeeID': [101, 102],
    'Name': ['Alice', 'Bob'],
    'Department': ['HR', 'IT']
})

df2 = pd.DataFrame({
    'EmployeeID': [103, 104],
    'Name': ['Charlie', 'David'],
    'Department': ['Finance', 'Marketing']
})

# Concatenate vertically
concat_df = pd.concat([df1, df2],ignore_index=True)

concat_df
```

Out[240…

	EmployeeID	Name	Department
0	101	Alice	HR
1	102	Bob	IT
2	103	Charlie	Finance
3	104	David	Marketing

Horizontal Concatenation (Adding Columns):

In [245…

```
# Concatenate horizontally
concat_df = pd.concat([df1, df2], axis=1)

concat_df
```

Out[245…

	EmployeeID	Name	Department	EmployeeID	Name	Department
0	101	Alice	HR	103	Charlie	Finance
1	102	Bob	IT	104	David	Marketing

Data Visualization:

* Basic Plotting:

```
* Using Matplot with Pandas for Visualisation.
```

In [251…

```
df
```

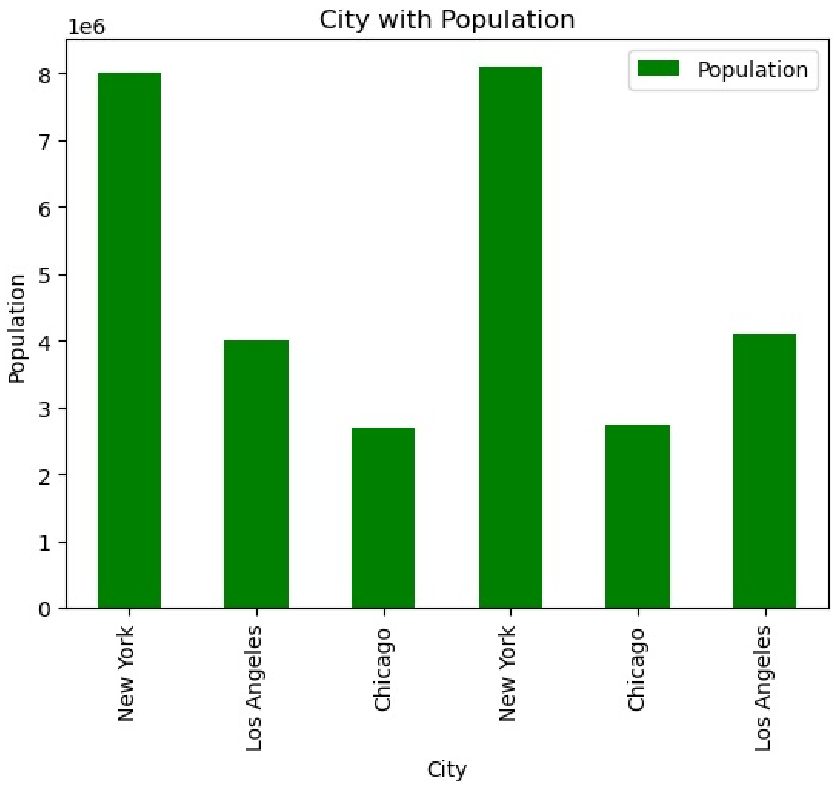
Out[251…

	City	Population	Area
0	New York	8000000	468.9
1	Los Angeles	4000000	503.0
2	Chicago	2700000	234.0
3	New York	8100000	468.9
4	Chicago	2750000	234.0
5	Los Angeles	4100000	503.0

In [277…

```
import matplotlib.pyplot as plt

df.plot(x='City',y='Population',kind='bar',color='green')
plt.title('City with Population')
plt.ylabel('Population')
plt.show()
```



Time Series Data:

* Creating Time Series:

```
You can create a time series DataFrame using the pd.date_range() function to generate a range
of dates and assign data to those dates.
```

In [289…

```
dates=pd.date_range('2024-01-01',periods=5)
time_series=pd.Series([100,200,300,400,500],index=dates)
```

```
time_series
```

Out[289…

```
2024-01-01    100
2024-01-02    200
2024-01-03    300
2024-01-04    400
2024-01-05    500
Freq: D, dtype: int64
```

In [298…

```
import pandas as pd

# Creating a date range
dates = pd.date_range(start='2023-01-01', periods=10, freq='D') #d=Dates,M=Months,Year=Y

# Creating a DataFrame with time series data
data = {
    'Sales': [100, 150, 200, 180, 220, 240, 260, 300, 320, 340],
    'Profit':[100, 150, 200, 180, 220, 240, 260, 300, 320, 340]
}
df = pd.DataFrame(data, index=dates)

print(df)
```

```
            Sales  Profit
2023-01-01    100     100
2023-01-02    150     150
2023-01-03    200     200
2023-01-04    180     180
2023-01-05    220     220
2023-01-06    240     240
2023-01-07    260     260
2023-01-08    300     300
2023-01-09    320     320
2023-01-10    340     340
```

2. Indexing by Time

```
Pandas treats DatetimeIndex differently, allowing you to perform operations that depend on
time.
```

In [300…

```
# Selecting data by date
print(df['2023-01-05':'2023-01-07'])  # Filter by date range
```

```
            Sales  Profit
2023-01-05    220     220
2023-01-06    240     240
2023-01-07    260     260
```

```
3. Resampling Data
```

You can resample time series data to different frequencies (e.g., daily to monthly, weekly to quarterly, etc.). The resample() function allows you to aggregate data at different intervals.

In [304…

```
# Resample data to weekly frequency and calculate the mean for each week
weekly_data = df.resample('W').mean()
print(weekly_data)
```

```
                 Sales      Profit
2023-01-01  100.000000  100.000000
2023-01-08  221.428571  221.428571
2023-01-15  330.000000  330.000000
```

In [307…

```
resampled=df.resample('2D').sum()
resampled
```

Out[307…

	Sales	Profit
2023-01-01	250	250
2023-01-03	380	380
2023-01-05	460	460
2023-01-07	560	560
2023-01-09	660	660

9.File I/O with Pandas

```
File I/O (Input/Output) in pandas refers to the operations of reading data from files and
```

writing data back to files. Pandas supports various file formats for reading and writing data, including CSV, Excel, JSON, SQL, HTML, and more. Here is how you can handle file I/O in pandas.

1. Reading from CSV Files

* CSV (Comma-Separated Values) is a popular data storage format. Pandas provides the read_csv() function to load data from CSV files.

In [319…
```
import pandas as pd

#Reading dat from the CSv file

data=pd.read_csv('data.csv')
data.head()
```

Out[319…

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0

2. Writing to CSV Files

we can write a Pandas DataFrame to a csv file using 'to_csv()'

In [321…
```
df.to_csv('csv_file.csv',index=False)
```

It will created one csv file with the name of 'csv_file.csv' in our homepage of jupytor

3. Reading from Excel Files

Pandas also supports reading from Excel files using the read_excel() function. It works with both .xls and .xlsx files

In [330…
```
df1=pd.read_excel('dat2.xlsx')  #sheet_name='Sheet1') if we have multple sheets in perticulat worksheet
```

In [334…
```
df1.head()
```

Out[334…

	Duration	Pulse	Maxpulse	Calories	date
0	60.0	110.0	130	409.1	2024-08-09
1	60.0	117.0	145	479.0	2024-08-10
2	60.0	103.0	NaN	340.0	2024-08-11
3	45.0	234.0	175	282.4	NaN
4	45.0	117.0	148	406.0	2024-08-13

4. Writing to Excel Files

You can write pandas DataFrames to Excel files using to_excel().

In [337…
```
data.to_excel('excel.xlsx',index=False)
```

In [344…
```
df2=pd.read_excel('excel.xlsx')
df2.head()
```

Out[344…

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0

5. Reading from JSON Files

JSON (JavaScript Object Notation) is a common data format, and pandas provides the read_json() function to load data from JSON files.

In [84]:
```
data={
    'Name':["vasu","sai","ravi","vishnu","vamsi"],
    'age':[23,21,22,34,23],
    'Salary':[1000,100000,203004,30450,3044]
}
df=pd.DataFrame(data)
df
```

Out[84]:

	Name	age	Salary
0	vasu	23	1000
1	sai	21	100000
2	ravi	22	203004
3	vishnu	34	30450
4	vamsi	23	3044

In [357…
```
df.to_json('json.json',orient='records')
```

In [364…
```
df=pd.read_json('json.json')
df
```

Out[364…

	Name	age	Salary
0	vasu	23	1000
1	sai	21	100000
2	ravi	22	203004
3	vishnu	34	30450
4	vamsi	23	3044

6. Writing to JSON Files

In [ ]:
```
we can write DataFrems to JSON foarmat using to_json()
```

In [367…
```
# Writing DataFrame to a JSON file
df.to_json('output.json', orient='records')
```

In [369…
```
df=pd.read_json('output.json')
df
```

Out[369…

	Name	age	Salary
0	vasu	23	1000
1	sai	21	100000
2	ravi	22	203004
3	vishnu	34	30450
4	vamsi	23	3044

7. Reading from SQL Databases

Pandas supports reading from Sql databases using read_sql() or read_sql_query() when using SQLAlchemy or a database connector.

In [30]:
```
import mysql.connector as conn
connection=conn.connect(
    host="localhost",
    user="root",
    password="Gsrinu@789",
    database="customers"
)
```

In [32]:
```
if connection.is_connected():
```

```
    print("connected")
```

connected

In [46]:
```
import mysql.connector as conn
mydb=conn.connect(
    host="localhost",
    user="root",
    password="Gsrinu@789"
)
mycursor=mydb.cursor()
mycursor.execute("Show databases")
for i in mycursor:
    print(i)
```

```
('customers',)
('employee1',)
('hms_db',)
('hosp_mang_sys',)
('hospital_management_db',)
('hospital_mang_sys',)
('information_schema',)
('inventory_db',)
('joins',)
('library',)
('machinelearning',)
('mani',)
('my',)
('mysql',)
('performance_schema',)
('srinu',)
('studentmanagement',)
('sys',)
('vasu',)
('vasu1',)
('vasu11',)
('vasu122',)
```

In [48]:
```
import mysql.connector as conn
mydb=conn.connect(
    host="localhost",
    user="root",
    password="Gsrinu@789",
    database="vasu122"
)
mycursor=mydb.cursor()
mycursor.execute("Show tables")
for i in mycursor:
    print(i)
```

```
('country',)
('customer',)
('customers',)
('orders',)
('shipper',)
('shippers',)
('srinu',)
```

In [70]:
```
import mysql.connector as conn
mydb=conn.connect(
    host="localhost",
    user="root",
    password="Gsrinu@789",
    database="vasu122"
)
mycursor=mydb.cursor()
mycursor.execute("select*from orders")
myresult=mycursor.fetchall()

for i in myresult:
    print(i)
```

```
(1222, 3, datetime.date(1993, 3, 2), None, None)
(1234, 4, datetime.date(1993, 3, 9), 'vasu', 1000)
(10677, 2, datetime.date(1995, 5, 5), 'vasu', 1000)
(32217, 22, datetime.date(1895, 5, 5), 'vasu', 1000)
```

9. Reading from HTML Files

In [88]:
```
df.to_html('html.html')
```

In [98]:
```
data_html=pd.read_html('html.html')
```

In [100…
```
data_html
```

Out[100…
```
[   Unnamed: 0    Name  age  Salary
 0           0    vasu   23    1000
 1           1     sai   21  100000
 2           2    ravi   22  203004
 3           3  vishnu   34   30450
 4           4   vamsi   23    3044]
```

In [104…
```
df=data_html[0]
df
```

Out[104…

	Unnamed: 0	Name	age	Salary
0	0	vasu	23	1000
1	1	sai	21	100000
2	2	ravi	22	203004
3	3	vishnu	34	30450
4	4	vamsi	23	3044

10. Reading from and Writing to Parquet Files

```
Parquet is a columnar storage format commonly used with big data systems.

You can use read_parquet() and to_parquet()
```

In [108…
```
df.to_parquet('data.parquet')
```

In [110…
```
df=pd.read_parquet('data.parquet')
df
```

Out[110…

	Unnamed: 0	Name	age	Salary
0	0	vasu	23	1000
1	1	sai	21	100000
2	2	ravi	22	203004
3	3	vishnu	34	30450
4	4	vamsi	23	3044

In [ ]:
```

```

In [118…
```
data={
    'Name':['vasu','ravi','sai','mani','vamsi'],
    'Age':[21,22,21,23,21],
    'Role':['Data Scientist','Data Analyst','Machine Learning Engineer','Devops Engineer','fronted developer'],
    'Salary':[10000,200000,1000000,3002300,293993],
    'Location':['Hyderabad','Banglore','Chennai','Delhi','Mumbai']
}
```

In [261…
```
df=pd.DataFrame(data)
df
```

Out[261…

	Name	Age	Role	Salary	Location
0	vasu	21	Data Scientist	10000	Hyderabad
1	ravi	22	Data Analyst	200000	Banglore
2	sai	21	Machine Learning Engineer	1000000	Chennai
3	mani	23	Devops Engineer	3002300	Delhi
4	vamsi	21	fronted developer	293993	Mumbai

10. Applying Functions

* Using apply():

```
1. apply() Function
The apply() function allows you to apply a function along either axis (rows or columns) of a
DataFrame.

Axis 0 (Columns): The function will be applied to each column.
Axis 1 (Rows): The function will be applied to each row.
```

In [167…

```
df['Salary After Hike']=df['Salary'].apply(lambda x:x*10.8)
#Aplly function to each element
df
```

Out[167…

	Name	Age	Role	Salary	Location	Salary After Hike
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0
1	ravi	22	Data Analyst	200000	Banglore	2160000.0
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4

Example(Row-wise applications)

In [169…

```
# Apply a function to each row
df['Salary_Age_Ratio'] = df.apply(lambda row: row['Salary'] / row['Age'], axis=1)

df
```

Out[169…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667

2. applymap() Function

    The applymap() function is used to apply a function element-wise to all cells in the
    DataFrame. This is typically used for transformations on every element of the DataFrame.

In [161…

```
# Sample DataFrame with mixed data types
df = pd.DataFrame({'A': [1, 2, 3], 'B': [10, 20, 30]})

# Multiply every element by 10
df_multiplied = df.applymap(lambda x: x * 10)

df_multiplied
```

```
C:\Users\gadam\AppData\Local\Temp\ipykernel_55700\2379198818.py:5: FutureWarning: DataFrame.applymap has been de
precated. Use DataFrame.map instead.
  df_multiplied = df.applymap(lambda x: x * 10)
```

Out[161…

	A	B
0	10	100
1	20	200
2	30	300

In [171…

```
df
```

Out[171…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667

3. map() Function

    The map() function is used to substitute each value in a Series with another value using a
    dictionary, a function, or a Series. It is most commonly used with a single pandas Series.

In [176…

```
df['Salary_adjusted']=df['Salary'].map(lambda x:x*10.3)
df
```

Out[176…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio	Salary_adjusted
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476	103000.0
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091	2060000.0
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619	10300000.0
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609	30923690.0
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667	3028127.9

In [187…

```
# Replace values in a column using a dictionary
mapping = {'vasu': 'A', 'ravi': 'B', 'sai': 'C'}
df['Name_Code'] = df['Name'].map(mapping)

df
```

Out[187…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio	Salary_adjusted	Name_Code
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476	103000.0	A
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091	2060000.0	B
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619	10300000.0	C
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609	30923690.0	NaN
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667	3028127.9	NaN

4. Applying Custom Functions

    You can also define your own functions and pass them to apply() or applymap().

In [191…

```
def age_category(Age):
    if Age<30:
        return 'Young'
    elif 30<=Age <40:
        return 'Middile Aged'
    else:
        return 'Senior'

df['Age_Category']=df['Age'].apply(age_category)
df
```

Out[191…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio	Salary_adjusted	Name_Code	Age_Category
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476	103000.0	A	Young
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091	2060000.0	B	Young
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619	10300000.0	C	Young
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609	30923690.0	NaN	Young
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667	3028127.9	NaN	Young

11. String Manipulation

- String Operations:

In [203…

```
df
```

Out[203…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio	Salary_adjusted	Name_Code	Age_Category
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476	103000.0	A	Young
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091	2060000.0	B	Young
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619	10300000.0	C	Young
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609	30923690.0	NaN	Young
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667	3028127.9	NaN	Young

In [207…

```
df['Name Upper']=df['Name'].str.upper()
#converts lowaercase to uppercase
df
```

Out[207…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio	Salary_adjusted	Name_Code	Age_Category	Name Upper
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476	103000.0	A	Young	VASU
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091	2060000.0	B	Young	RAVI
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619	10300000.0	C	Young	SAI
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609	30923690.0	NaN	Young	MANI
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667	3028127.9	NaN	Young	VAMSI

In [209…

```
# Convert to lowercase
df['Name_lower'] = df['Name'].str.lower()
```

In [211…

```
df
```

Out[211…

	Name	Age	Role	Salary	Location	Salary After Hike	Salary_Age_Ratio	Salary_adjusted	Name_Code	Age_Category	Name Upper
0	vasu	21	Data Scientist	10000	Hyderabad	108000.0	476.190476	103000.0	A	Young	VASU
1	ravi	22	Data Analyst	200000	Banglore	2160000.0	9090.909091	2060000.0	B	Young	RAVI
2	sai	21	Machine Learning Engineer	1000000	Chennai	10800000.0	47619.047619	10300000.0	C	Young	SAI
3	mani	23	Devops Engineer	3002300	Delhi	32424840.0	130534.782609	30923690.0	NaN	Young	MANI
4	vamsi	21	fronted developer	293993	Mumbai	3175124.4	13999.666667	3028127.9	NaN	Young	VAMSI

2. Removing Whitespace

```
You can remove leading and trailing whitespace using str.strip(), .lstrip() for leading
spaces, and str.rstrip() for trailing spaces.
```

In [215…

```
df = pd.DataFrame({'Name': [' Alice ', ' Bob ', ' Charlie ']})

# Remove leading and trailing spaces
df['Name_stripped'] = df['Name'].str.strip()

print(df)
```

```
        Name Name_stripped
0     Alice          Alice
1       Bob            Bob
2   Charlie        Charlie
```

3. Substring Extraction

You can extract substrings from a pandas column using str.slice(), str[:n], or str[-n:].

In [221…
```python
# Extract first 3 characters
df['First_3'] = df['Name'].str[:3]

# Extract last 3 characters
df['Last_3'] = df['Name'].str[-3:]

print(df)
```

```
      Name Name_stripped First_3 Last_3
0    Alice         Alice      Al     ce
1      Bob           Bob      Bo     ob
2  Charlie       Charlie      Ch     ie
```

4. String Replacement

You can replace specific substrings using str.replace().

In [224…
```python
df = pd.DataFrame({'Name': ['Mr. Alice', 'Mr. Bob', 'Ms. Charlie']})

# Replace 'Mr.' with 'Dr.'
df['Name_replaced'] = df['Name'].str.replace('Mr.', 'Dr.')

print(df)
```

```
          Name Name_replaced
0    Mr. Alice     Dr. Alice
1      Mr. Bob       Dr. Bob
2  Ms. Charlie   Ms. Charlie
```

5. String Splitting

You can split strings into multiple columns using str.split(). For example, splitting a full name into first and last names.

In [239…
```python
df = pd.DataFrame({'FullName': ['Alice Johnson', 'Bob Brown', 'Charlie Davis']})

# Split the full name into first and last names
df[['FirstName', 'LastName']] = df['FullName'].str.split(' ', expand=True)

df
```

Out[239…

	FullName	FirstName	LastName
0	Alice Johnson	Alice	Johnson
1	Bob Brown	Bob	Brown
2	Charlie Davis	Charlie	Davis

6. Finding and Matching Substrings

You can find substrings or check if a string contains a specific pattern using str.contains(), str.startswith(), or str.endswith().

In [244…
```python
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie']})

# Check if the name contains 'li'
df['Contains_li'] = df['Name'].str.contains('li')

# Check if the name starts with 'A'
df['Starts_with_A'] = df['Name'].str.startswith('A')

# Check if the name ends with 'e'
df['Ends_with_e'] = df['Name'].str.endswith('e')

df
```

Out[244…

	Name	Contains_li	Starts_with_A	Ends_with_e
0	Alice	True	True	True
1	Bob	False	False	False
2	Charlie	True	False	True

7. String Length

You can get the length of strings in a pandas column using str.len().

In [247…
```
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Charlie']})

# Get the length of each string
df['Name_length'] = df['Name'].str.len()

df
```

Out[247…

	Name	Name_length
0	Alice	5
1	Bob	3
2	Charlie	7

8. Concatenating Strings

    You can concatenate strings in a pandas column using + or str.cat().

In [252…
```
df = pd.DataFrame({'First': ['Alice', 'Bob', 'Charlie'],
                   'Last': ['Johnson', 'Brown', 'Davis']})

# Concatenate first and last names
df['FullName'] = df['First'] + ' ' + df['Last']

df
```

Out[252…

	First	Last	FullName
0	Alice	Johnson	Alice Johnson
1	Bob	Brown	Bob Brown
2	Charlie	Davis	Charlie Davis

9. Extracting Using Regular Expressions

    You can extract patterns from a string using regular expressions (str.extract()).

In [259…
```
df = pd.DataFrame({'Email': ['alice@example.com', 'bob@example.org', 'charlie@example.net']})

# Extract domain names
df['Domain'] = df['Email'].str.extract(r'@([A-Za-z_]+)\.')
df
```

Out[259…

	Email	Domain
0	alice@example.com	example
1	bob@example.org	example
2	charlie@example.net	example

In [263…
```
df
```

Out[263…

	Name	Age	Role	Salary	Location
0	vasu	21	Data Scientist	10000	Hyderabad
1	ravi	22	Data Analyst	200000	Banglore
2	sai	21	Machine Learning Engineer	1000000	Chennai
3	mani	23	Devops Engineer	3002300	Delhi
4	vamsi	21	fronted developer	293993	Mumbai

12. Categorical Data

Creating Categorical Data:

    In pandas, categorical data refers to a variable that can take on one of a limited, fixed
    number of possible values (categories). Examples include gender, country names, or product
    categories. Categorical data can be useful in improving performance (both in terms of memory
    and speed) and can be used for more efficient data analysis.

In [267…
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   Name      5 non-null      object
 1   Age       5 non-null      int64 
 2   Role      5 non-null      object
 3   Salary    5 non-null      int64 
 4   Location  5 non-null      object
dtypes: int64(2), object(3)
memory usage: 332.0+ bytes
```

In [273…

```
df['Location']=df['Location'].astype('category')
df['Location'].cat.codes
```

Out[273…

```
0    3
1    0
2    1
3    2
4    4
dtype: int8
```

In [21]:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Department': ['HR', 'Finance', 'IT', 'HR']
})

# Convert the 'Department' column to categorical
df['Department'] = pd.Categorical(df['Department'])

df
```

Out[21]:

	Name	Department
0	Alice	HR
1	Bob	Finance
2	Charlie	IT
3	David	HR

In [23]:

```
print(df.dtypes)
```

```
Name            object
Department    category
dtype: object
```

2. Checking Unique Categories

    You can check the unique categories of a categorical column using .cat.categories.

In [25]:

```
# Check unique categories
print(df['Department'].cat.categories)
```

```
Index(['Finance', 'HR', 'IT'], dtype='object')
```

3.Changing the Categories

    We can rename ,add,or remove categories using the .cat.catgories

In [31]:

```
import pandas as pd

# Sample DataFrame with categorical data
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Department': pd.Categorical(['HR', 'Finance', 'IT', 'HR'])
})

# Rename categories using .rename_categories()
df['Department'] = df['Department'].cat.rename_categories(['HR Dept', 'Finance Dept', 'IT Dept'])

print(df)
```

```
      Name    Department
0    Alice  Finance Dept
1      Bob       HR Dept
2  Charlie       IT Dept
3    David  Finance Dept
```

4.Reordering the Categories

Sometimes, you may want to impose a specific order on the categories, especially if they have a natural order (e.g., 'Small', 'Medium', 'Large'). You can do this by passing the ordered=True argument and providing the order.

In [35]:
```
# Create a categorical column with an order
sizes = pd.Categorical(['Medium', 'Large', 'Small', 'Small'],
                       categories=['Small', 'Medium', 'Large'],
                       ordered=True)

df = pd.DataFrame({'Size': sizes})

print(df)
```

```
     Size
0  Medium
1   Large
2   Small
3   Small
```

5. Sorting Categorical Data

Categorical data can be sorted according to the order of categories (if defined). This is useful when the categories are ordered.

In [40]:
```
# Sorting by 'Size' based on the category order
df = df.sort_values(by='Size')
df
```

Out[40]:

	Size
2	Small
3	Small
0	Medium
1	Large

6. Replacing Categories

You can use .cat.rename_categories() to rename or map categories in a categorical column.

In [45]:
```
# Rename the categories
df['Size'] = df['Size'].cat.rename_categories({'Small': 'S', 'Medium': 'M', 'Large': 'L'})

df
```

Out[45]:

	Size
2	S
3	S
0	M
1	L

7. Using .cat.codes

You can get the integer codes of the categorical values using .cat.codes, which can be useful for machine learning models or when you need numeric representations of categorical values

In [52]:
```
# Get integer codes for the categorical column
df['Size_code'] = df['Size'].cat.codes

df
```

Out[52]:

	Size	Size_code
2	S	0
3	S	0
0	M	1
1	L	2

In [13]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

13. Advanced Indexing and Selection

In [59]:
```
arrays=[['A','A','B','B'],['one','two','one','two']]
index=pd.MultiIndex.from_arrays(arrays,names=('first','second'))
df_multi=pd.DataFrame({'data':[1,2,3,4]},index=index)
df_multi
```

Out[59]:

first	second	data
A	one	1
	two	2
B	one	3
	two	4

- Selecting Data with MultiIndex:

In [62]:
```
print(df_multi.loc['A'])  # Select all data for index 'A'
```

```
        data
second
one        1
two        2
```

1. Selecting Data with loc and iloc

```
loc is label-based, which means you have to specify the names of the rows and columns.
iloc is integer-location-based, meaning you can select data by row and column indices.
```

In [67]:
```
import pandas as pd

# Create a sample DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'Salary': [50000, 60000, 70000, 80000]
})

# Select rows and columns using labels
df.loc[0:2, ['Name', 'Salary']]  # Select rows 0 to 2 and columns 'Name' and 'Salary
```

Out[67]:

	Name	Salary
0	Alice	50000
1	Bob	60000
2	Charlie	70000

In [71]:
```
# Select rows and columns using integer-location based index
df.iloc[0:3, 0:2]  # Select first 3 rows and first 2 columns
```

Out[71]:

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35

2. Boolean Indexing

```
Boolean indexing is a powerful technique where you use boolean conditions to filter data from
```

your DataFrame.

In [76]:
```
# Filter rows where Age is greater than 30
filtered_df = df[df['Age'] > 30]
filtered_df
```

Out[76]:

	Name	Age	Salary
2	Charlie	35	70000
3	David	40	80000

we can combine multiple conditions using bitwise operators (&, |, ~).

In [79]:
```
# Filter rows where Age is greater than 30 and Salary is less than 80000
filtered_df = df[(df['Age'] > 30) & (df['Salary'] < 80000)]
filtered_df
```

Out[79]:

	Name	Age	Salary
2	Charlie	35	70000

3. Using query() for Filtering

query() is a method that allows you to filter the DataFrame using a query string, making the syntax more concise for complex conditions.

In [82]:
```
filtered_df=df.query('Age > 30 and Salary < 80000')
filtered_df
```

Out[82]:

	Name	Age	Salary
2	Charlie	35	70000

4. Using isin() for Filtering

The isin() method is used to filter rows where a column's value belongs to a list of specified values.

In [85]:
```
filtered_df=df[df['Name'].isin(['Alice','David'])]
filtered_df
```

Out[85]:

	Name	Age	Salary
0	Alice	25	50000
3	David	40	80000

5. Setting Values with Conditional Indexing

You can set values in a DataFrame using conditional indexing.

In [89]:
```
# Set Salary to 0 where Age is greater than 30

df.loc[df['Age']>30 ,'Salary']=0
df
```

Out[89]:

	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	0
3	David	40	0

6. MultiIndex for Advanced Indexing

Pandas supports multi-level indexing, where you can index your data by more than one key (multi-index or hierarchical index).

In [94]:
```
# Create a sample DataFrame with a MultiIndex
arrays = [
    ['California', 'California', 'Texas', 'Texas'],
```

```
    ['Los Angeles', 'San Francisco', 'Houston', 'Austin']
]
index = pd.MultiIndex.from_arrays(arrays, names=('State', 'City'))
df_multi = pd.DataFrame({'Population': [4000000, 880000, 2300000, 960000]}, index=index)

df_multi
```

Out[94]:

State	City	Population
California	**Los Angeles**	4000000
	San Francisco	880000
Texas	**Houston**	2300000
	Austin	960000

In [98]:
```
# Select data for Texas
df_multi.loc['Texas']
```

Out[98]:

City	Population
Houston	2300000
Austin	960000

In [106…
```
# Select data for Houston
df_multi.loc[('Texas', 'Houston')]
```

Out[106…
```
Population    2300000
Name: (Texas, Houston), dtype: int64
```

14. Reshaping Data

- Using melt():

In [132…
```
data={
    'Name':['vasu','ravi','sai','mani','vamsi','Ashwinth'],
    'Age':[21,22,21,23,21,27],
    'Role':['Data Scientist','Data Analyst','Machine Learning Engineer','Devops Engineer','fronted developer','I
    'Salary':[10000,200000,1000000,3002300,293993,100000],
    'Location':['Hyderabad','Banglore','Chennai','Delhi','Mumbai','Banglore']
}
```

In [134…
```
df=pd.DataFrame(data)
df
```

Out[134…

	Name	Age	Role	Salary	Location
0	vasu	21	Data Scientist	10000	Hyderabad
1	ravi	22	Data Analyst	200000	Banglore
2	sai	21	Machine Learning Engineer	1000000	Chennai
3	mani	23	Devops Engineer	3002300	Delhi
4	vamsi	21	fronted developer	293993	Mumbai
5	Ashwinth	27	Data Scientist	100000	Banglore

In [136…
```
df_melted=df.melt(id_vars=['Name'],value_vars=['Salary','Age'],
                  var_name='Variable',value_name='Value')
df_melted
```

Out[136…

	Name	Variable	Value
0	vasu	Salary	10000
1	ravi	Salary	200000
2	sai	Salary	1000000
3	mani	Salary	3002300
4	vamsi	Salary	293993
5	Ashwinth	Salary	100000
6	vasu	Age	21
7	ravi	Age	22
8	sai	Age	21
9	mani	Age	23
10	vamsi	Age	21
11	Ashwinth	Age	27

15.Common Methods and Functions

* Basic Descriptive Statistics:

In [138…

```
df
```

Out[138…

	Name	Age	Role	Salary	Location
0	vasu	21	Data Scientist	10000	Hyderabad
1	ravi	22	Data Analyst	200000	Banglore
2	sai	21	Machine Learning Engineer	1000000	Chennai
3	mani	23	Devops Engineer	3002300	Delhi
4	vamsi	21	fronted developer	293993	Mumbai
5	Ashwinth	27	Data Scientist	100000	Banglore

In [140…

```
df.describe()
```

Out[140…

	Age	Salary
count	6.000000	6.000000e+00
mean	22.500000	7.677155e+05
std	2.345208	1.150132e+06
min	21.000000	1.000000e+04
25%	21.000000	1.250000e+05
50%	21.500000	2.469965e+05
75%	22.750000	8.234982e+05
max	27.000000	3.002300e+06

• Getting Unique Values:

In [144…

```
unique_roles=df['Role'].unique()
print(unique_roles)
```

```
['Data Scientist' 'Data Analyst' 'Machine Learning Engineer'
 'Devops Engineer' 'fronted developer']
```

* Counting Values

In [160…

```
city_counts=df['Location'].value_counts()
city_counts
```

Out[160…

```
Location
Banglore     2
Hyderabad    1
Chennai      1
Delhi        1
Mumbai       1
Name: count, dtype: int64
```

In [162…

```
df.count()
```

Out[162…

```
Name        6
Age         6
Role        6
Salary      6
Location    6
dtype: int64
```

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js