Data Mainpulation by Mrittika Megaraj

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

In [2]:
```
arr = np.array([1,2,3,4])
print(arr)
```

```
[1 2 3 4]
```

Advanced Indexing and Selection

Multi-level indexing with hierarchical indexing

- Creating DataFrames with multiple levels of indexes to work with multi-dimensional data.

In [3]:
```
import pandas as pd

data = {
    'A': [1, 2, 3, 4, 5, 6],
    'B': [7, 8, 9, 10, 11, 12],
    'C': [13, 14, 15, 16, 17, 18]
}
index = pd.MultiIndex.from_tuples([('X', 2020), ('X', 2021), ('Y', 2020), (
df = pd.DataFrame(data, index=index)

print(df['A']['X'])
```

```
Year
2020    1
2021    2
Name: A, dtype: int64
```

Indexing and slicing with loc[] and iloc[]:

- Accessing DataFrame elements using labeled and integer-based indexing.

In [4]:
```
# Using loc[] for labeled indexing
print(df.loc[('X', 2020), 'A'])

# Using iloc[] for integer-based indexing
print(df.iloc[0, 1])
```

```
1
7
```

Boolean indexing and filtering:

- Selecting data from a DataFrame based on specified conditions.

In [5]:
```
# Boolean indexing to filter rows with 'B' values greater than 9
filtered_df = df[df['B'] > 9]
print(filtered_df)
```

```
           A   B   C
City Year
Y    2021  4  10  16
Z    2020  5  11  17
     2021  6  12  18
```

Combining DataFrames

Merging and joining DataFrames with merge() and join():

- Combining DataFrames based on common columns.
- inner Join: Only common records will be displayed from the both the table and their matching values
- left : All records from the first table will be displayed and matching records from the right table and if there is no matching data null values will be displayed
- right : All records from the second table will be displayed and matching records from the right table and if there is no matching data null values will be displayed
- full : All records from the both tables will be displayed and matchign data and if matching data is not available null will be displayed

In [6]:
```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Age': [25, 30, 22]})

# Merging based on 'ID'
merged_df = pd.merge(df1, df2, on='ID', how='inner')
print(merged_df)
```

```
   ID     Name  Age
0   2      Bob   25
1   3  Charlie   30
```

Concatenating DataFrames using concat():

- Combining DataFrames along a specified axis (rows or columns).

In [7]:
```
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

# Concatenating along rows
concatenated_df = pd.concat([df1, df2])
print(concatenated_df)
```

```
   A  B
0  1  3
1  2  4
0  5  7
1  6  8
```

Data Manipulation

Filtering and subsetting data based on conditions

- Extracting specific subsets of data using conditional statements.

In [8]:
```
d = {'id':[1,2,3],'name':['A','B','C'],'Age':[22,26,38],'City':['Kolhapur',
df = pd.DataFrame(d)

# Filter rows where column 'Age' is greater than 25
filtered_data = df[df['Age'] > 25]
print(filtered_data)
```

```
   id name  Age    City
1   2    B   26    Pune
2   3    C   38  Mumbai
```

Sorting and ranking data:

- Ordering data based on column values and assigning ranks to data elements.

In [9]:
```
# Sorting DataFrame based on 'Age' in descending order
sorted_df = df.sort_values(by='Age', ascending=False)

# Ranking 'Age' within the DataFrame
df['Rank'] = df['Age'].rank(ascending=False)

print(df)
```

```
   id name  Age      City  Rank
0   1    A   22  Kolhapur   3.0
1   2    B   26      Pune   2.0
2   3    C   38    Mumbai   1.0
```

Aggregating and summarizing data using groupby()

- Grouping data based on one or more columns and applying aggregation functions.

In [10]:
```
# Grouping data by 'City' and calculating mean 'Age'
grouped_df = df.groupby('City')['Age'].mean()
print(grouped_df)
```

```
City
Kolhapur    22.0
Mumbai      38.0
Pune        26.0
Name: Age, dtype: float64
```

Pivoting and melting data for reshaping:

- Changing the layout of the DataFrame to perform analysis efficiently.

In [ ]:
```
data = {
    'Id': [1,2,3,4,5],
    'Name': ['A', 'B', 'C', 'D', 'E'],
    'City': ['Kolhapur', 'Pune', 'Sangli', 'Satara', 'Mumbai'],
    'Age': [21, 23, 56, 34, 68],
    'Salary': [23000, 45000, 35000, 78000, 56000],
    'Year': [2021, 2018, 2023, 2015, 2019]
}

df = pd.DataFrame(data)

# Pivoting DataFrame to show 'Age' for each 'City'
pivoted_df = df.pivot(index='City', columns='Year', values='Age')

print(pivoted_df)
```

In [ ]:
```
# Melting DataFrame to convert columns into rows
melted_df = pd.melt(df, id_vars=['City', 'Year'], value_vars=['Age', 'Salar

print(melted_df)
```

Advanced Data Manipulation

Multi-level indexing and hierarchical data

- Creating DataFrames with multiple levels of indexes to handle complex datasets.

In [12]:
```
import pandas as pd

# Creating a DataFrame with multi-level index
data = {
    'A': [1, 2, 3, 4, 5, 6],
    'B': [7, 8, 9, 10, 11, 12],
    'C': [13, 14, 15, 16, 17, 18]
}
index = pd.MultiIndex.from_tuples([('X', 2020), ('X', 2021), ('Y', 2020), (
df = pd.DataFrame(data, index=index)

print(df['A']['X'])
```

```
Year
2020    1
2021    2
Name: A, dtype: int64
```

Pivot tables and cross-tabulations:

- Transforming data and summarizing it using pivot tables and cross-tabulations.

In [13]:
```
# Creating a DataFrame
data = {
    'City': ['A', 'A', 'B', 'B', 'A', 'B'],
    'Year': [2020, 2021, 2020, 2021, 2020, 2021],
    'Sales': [100, 150, 120, 200, 80, 250]
}
df = pd.DataFrame(data)

# Creating a pivot table to summarize 'Sales' based on 'City' and 'Year'
pivot_table = df.pivot_table(values='Sales', index='City', columns='Year',
print(pivot_table)
```

```
Year  2020  2021
City
A      180   150
B      120   450
```

Handling text data and regular expressions:

- Dealing with text data and applying regular expressions for pattern matching and extraction.

In [14]:
```
# Creating a DataFrame with text data
data = {
    'Text': ['apple', 'orange', 'banana', 'grape', 'peach']
}
df = pd.DataFrame(data)

# Using str.contains() to filter rows with text containing 'a'
filtered_df = df[df['Text'].str.contains('a')]
print(filtered_df)
```

```
     Text
0   apple
1  orange
2  banana
3   grape
4   peach
```

Working with JSON and other data:

- Reading, manipulating, and analyzing data in JSON format and other formats like XML, HTML, etc.

In [15]:
```
# Reading JSON data into a DataFrame
import json

json_data = '{"name": "John", "age": 30, "city": "New York"}'
df = pd.DataFrame(json.loads(json_data), index=[0])
print(df)
```

```
   name  age      city
0  John   30  New York
```

Data Aggregation and Grouping

Grouping data using groupby()

- Splitting data into groups based on one or more categorical variables.

In [16]:
```
# Creating a DataFrame
data = {
    'City': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Sales': [100, 120, 80, 150, 200, 250]
}
df = pd.DataFrame(data)

# Grouping data by 'City'
grouped_df = df.groupby('City')

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

```
('A',   City  Sales
0    A    100
2    A     80
4    A    200)
('B',   City  Sales
1    B    120
3    B    150
5    B    250)
```

Applying aggregation functions to groups:

- Calculating summary statistics for each group.

In [17]:
```
# Calculating the total sales for each city
grouped_df = df.groupby('City')['Sales'].sum()
print(grouped_df)
```

```
City
A    380
B    520
Name: Sales, dtype: int64
```

Performing multi-level aggregation:

- Aggregating data at multiple levels of grouping.

In [18]:
```
data = {
    'Id': [1,2,3,4,5,6],
    'City': ['Kolhapur', 'Pune', 'Sangli', 'Mumbai', 'Satara', 'Kolhapur'],
    'Year': [2012, 2016, 2013, 2015, 2017, 2018],
    'Sales': [23000, 43000, 30000, 40000, 65000, 34000]
}

df = pd.DataFrame(data)

# Grouping data by 'City' and 'Year', and calculating the total sales for e
grouped_df = df.groupby(['City', 'Year'])['Sales'].sum()
print(grouped_df)
```

```
City      Year
Kolhapur  2012    23000
          2018    34000
Mumbai    2015    40000
Pune      2016    43000
Sangli    2013    30000
Satara    2017    65000
Name: Sales, dtype: int64
```

Grouping data with groupby() and split-apply-combine operations

- Applying transformations to groups and combining the results.

In [19]:
```
# Applying multiple aggregation functions to 'Sales' for each city
grouped_df = df.groupby('City')['Sales'].agg(['sum', 'mean', 'max'])
print(grouped_df)
```

```
            sum     mean    max
City
Kolhapur  57000  28500.0  34000
Mumbai    40000  40000.0  40000
Pune      43000  43000.0  43000
Sangli    30000  30000.0  30000
Satara    65000  65000.0  65000
```

Aggregation functions (e.g., mean, sum, count, min, max):

- Using various aggregation functions to calculate statistics on grouped data.

In [20]:
```
# Calculating the average and total sales for each city
grouped_df = df.groupby('City')['Sales'].agg(['mean', 'sum'])
print(grouped_df)
```

```
              mean    sum
City
Kolhapur  28500.0  57000
Mumbai    40000.0  40000
Pune      43000.0  43000
Sangli    30000.0  30000
Satara    65000.0  65000
```

Applying Functions to DataFrames

.apply for series and DataFrames

- Use this method to apply a custom function to a series or to the entire dataframe.
- when you use this on series, Each element of the original column will be passed to the function.
- when you use this for the entire dataframe, based on the axis (1 - row, 0 -column), the entire row or the entire column will be passed to the function.

In [24]:
```
data = {'A': [1, 2, 3, 1, 2], 'B': [4, 5, 6, 4, 5]}
df = pd.DataFrame(data)

# Define a function to calculate the average of a row
def average_row(row):
    return row.mean()

# Apply the function row-wise.
df['Row_Average'] = df.apply(average_row, axis=1)
df
```

Out[24]:

	A	B	Row_Average
0	1	4	2.5
1	2	5	3.5
2	3	6	4.5
3	1	4	2.5
4	2	5	3.5

.map for series

- It's particularly useful for transforming one column based on values from another.

In [25]:
```
# Mapping values in a column based on a dictionary
mapping_dict = {1: 'one', 2: 'two', 3: 'three'}
df['Encoded_A'] = df['A'].map(mapping_dict)
df
```

Out[25]:

	A	B	Row_Average	Encoded_A
0	1	4	2.5	one
1	2	5	3.5	two
2	3	6	4.5	three
3	1	4	2.5	one
4	2	5	3.5	two

.applymap for entire dataframe

- When you want to apply a function to each element in the entire DataFrame, you can use .applymap().

In [26]:
```
# Sample DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Define a function to add 10 to a value
def add_10(x):
    return x + 10

# Apply the function to the entire DataFrame
df = df.applymap(add_10)
df
```

Out[26]:

	A	B
0	11	14
1	12	15
2	13	16

In [ ]:
```

```