Advanced python programming

Name:P.Sriram

Reg.NO:22MID0290

LAB18

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

# Initializing a Series from a list
data = [1, 2.3, 'a', 4, 5]
series_from_list = pd.Series(data)
print(series_from_list)
```

```
0      1
1    2.3
2      a
3      4
4      5
dtype: object
```

In [2]:
```
# Creating a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [23, 25, 22, 24],
    'Score': [85, 90, 78, 92]
}

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

```
      Name  Age  Score
0    Alice   23     85
1      Bob   25     90
2  Charlie   22     78
3    David   24     92
```

In [3]:
```
#alignment
s1 = pd.Series([1, 2, 3], index=["a", "b", "c"])
s2 = pd.Series([4, 5, 6], index=["b", "c", "d"])
print(s1 * s2)
```

```
a     NaN
b     8.0
c    15.0
d     NaN
dtype: float64
```

In [4]:
```
series_a = pd.Series([1, 2, 3])
series_b = pd.Series([4, 5, 6])
```

```
sum_series = series_a + series_b
print(sum_series)
```

```
0    5
1    7
2    9
dtype: int64
```

In [5]:
```
# Creating a MultiIndex Series
arrays = [
    ['A', 'A', 'B', 'B'],
    ['Math', 'Science', 'Math', 'Science']
]
index = pd.MultiIndex.from_arrays(arrays, names=('Alphabet', 'Subject'))

multi_s = pd.Series([90, 85, 88, 92], index=index)
print(multi_s)
```

```
Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64
```

In [6]:
```
# Creating a MultiIndex Series
import pandas as pd

arrays = [
    ['A', 'A', 'B', 'B'],    # <-- fixed quotes
    ['Math', 'Science', 'Math', 'Science']
]

index = pd.MultiIndex.from_arrays(arrays, names=('Alphabet',  'Subject'))

multi_s = pd.Series([90, 85, 88, 92], index=index)
print(multi_s)
```

```
Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64
```

In [7]:
```
import pandas as pd

tuples = [('A', 'Math'), ('A', 'Science'), ('B', 'Math'), ('B', 'Science')]
index = pd.MultiIndex.from_tuples(tuples, names=('Alphabet', 'Subject'))

multi_s = pd.Series([90, 85, 88, 92], index=index)
print(multi_s)
```

```
Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64
```

In [8]:
```
index = pd.MultiIndex.from_product(
    [['A', 'B'], ['Math', 'Science']],
```

```
    names=('Alphabet', 'Subject')
)

multi_s = pd.Series([90, 85, 88, 92], index=index)
print(multi_s)
```

```
Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64
```

In [9]:
```
df = pd.DataFrame({
    'Alphabet': ['A', 'A', 'B', 'B'],
    'Subject': ['Math', 'Science', 'Math', 'Science']
})
index = pd.MultiIndex.from_frame(df, names=('Alphabet', 'Subject'))

multi_s = pd.Series([90, 85, 88, 92], index=index)
print(multi_s)
```

```
Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64
```

In [10]:
```
import pandas as pd
import numpy as np

# ------------------------------------------------------------------------------------------
# 1. Creating a MultiIndex Series in Different Ways
# ------------------------------------------------------------------------------------------

# From arrays
arrays = [
    ["A", "A", "B", "B"],
    ["Math", "Science", "Math", "Science"]
]
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
multi_s = pd.Series([90, 85, 88, 92], index=index)
print("MultiIndex Series from arrays:\n", multi_s, "\n")

# From tuples
tuples = [
    ("A", "Math"), ("A", "Science"),
    ("B", "Math"), ("B", "Science")
]
index2 = pd.MultiIndex.from_tuples(tuples, names=("Alphabet", "Subject"))
multi_s2 = pd.Series([70, 75, 80, 82], index=index2)
print("MultiIndex Series from tuples:\n", multi_s2, "\n")

# From product (Cartesian product of iterables)
iterables = [["A", "B"], ["Math", "Science"]]
index3 = pd.MultiIndex.from_product(iterables, names=("Alphabet", "Subject"))
multi_s3 = pd.Series(np.random.randint(60, 100, size=4), index=index3)
print("MultiIndex Series from product:\n", multi_s3, "\n")

# ------------------------------------------------------------------------------------------
```

```
# 2. Accessing and Indexing
# ------------------------------------------------------------------------------

print("Access all subjects for 'A':\n", multi_s.loc["A"], "\n")
print("Access specific element (B, Science):\n", multi_s.loc[("B", "Science")], "

# ------------------------------------------------------------------------------
# 3. Slicing in MultiIndex
# ------------------------------------------------------------------------------

print("Slicing from A to B:\n", multi_s.loc["A":"B"], "\n")
print("Partial slice for all Math:\n", multi_s.loc[:, "Math"], "\n")

# ------------------------------------------------------------------------------
# 4. Swapping and Reordering Levels
# ------------------------------------------------------------------------------

print("Swapping levels:\n", multi_s.swaplevel(), "\n")
print("Reordering levels:\n", multi_s3.reorder_levels(["Subject", "Alphabet"]), "

# ------------------------------------------------------------------------------
# 5. Passing List of Arrays directly to Series / DataFrame
# ------------------------------------------------------------------------------

multi_s_auto = pd.Series(
    np.random.randn(4),
    index=pd.MultiIndex.from_arrays([["A", "A", "B", "B"], ["X", "Y", "X", "Y"]])
)
print("MultiIndex Series constructed automatically:\n", multi_s_auto, "\n")

df_auto = pd.DataFrame(
    np.random.randn(4, 2),
    index=pd.MultiIndex.from_arrays([["Group1", "Group1", "Group2", "Group2"],
                                     ["One", "Two", "One", "Two"]]),
    columns=["Score1", "Score2"]
)
print("DataFrame with MultiIndex automatically:\n", df_auto, "\n")

# ------------------------------------------------------------------------------
# 6. Data Alignment and Reindexing
# ------------------------------------------------------------------------------

df = pd.DataFrame({
    "Math": [85, 90, 95, 80],
    "Science": [82, 88, 92, 84]
}, index=pd.MultiIndex.from_arrays([["A", "A", "B", "B"], ["one", "two", "one", "
print("Original DataFrame:\n", df, "\n")

# Group by first level and compute mean
mean_by_group = df.groupby(level=0).mean()
print("Mean by group:\n", mean_by_group, "\n")

# Reindexing with MultiIndex
aligned = mean_by_group.reindex(df.index, level=0)
print("Reindexed to align with original index:\n", aligned, "\n")

# ------------------------------------------------------------------------------
# 7. Using xs() for Cross-Section
# ------------------------------------------------------------------------------
```

```
print("Cross-section for level 'two':\n", df.xs("two", level=1), "\n")

# ------------------------------------------------------------------------------------------
# 8. Sorting and Removing Unused Levels
# ------------------------------------------------------------------------------------------

unsorted = multi_s_auto.sample(frac=1)   # shuffle randomly
print("Unsorted MultiIndex Series:\n", unsorted, "\n")
print("Sorted by index:\n", unsorted.sort_index(), "\n")

sub_df = df_auto[["Score1"]]  # remove one column
print("Unused levels before removing:\n", sub_df.columns.levels, "\n")
print("After remove_unused_levels:\n", sub_df.columns.remove_unused_levels().leve
```

```
MultiIndex Series from arrays:
 Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64

MultiIndex Series from tuples:
 Alphabet  Subject
A         Math       70
          Science    75
B         Math       80
          Science    82
dtype: int64

MultiIndex Series from product:
 Alphabet  Subject
A         Math       82
          Science    60
B         Math       87
          Science    74
dtype: int32

Access all subjects for 'A':
 Subject
Math       90
Science    85
dtype: int64

Access specific element (B, Science):
 92

Slicing from A to B:
 Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64

Partial slice for all Math:
 Alphabet
A    90
B    88
dtype: int64

Swapping levels:
 Subject  Alphabet
Math     A           90
Science  A           85
Math     B           88
Science  B           92
dtype: int64

Reordering levels:
 Subject  Alphabet
Math     A           82
Science  A           60
Math     B           87
```

```
Science  B           74
dtype: int32

MultiIndex Series constructed automatically:
 A  X   -0.095928
    Y    0.238070
B   X    0.768439
    Y    1.745636
dtype: float64

DataFrame with MultiIndex automatically:
               Score1    Score2
Group1 One -1.511754  -0.082332
       Two -0.086140  -1.692028
Group2 One  0.996158   0.377584
       Two -1.211479   0.551397

Original DataFrame:
        Math  Science
A one    85       82
  two    90       88
B one    95       92
  two    80       84

Mean by group:
    Math  Science
A  87.5     85.0
B  87.5     88.0

Reindexed to align with original index:
        Math  Science
A one  87.5     85.0
  two  87.5     85.0
B one  87.5     88.0
  two  87.5     88.0

Cross-section for level 'two':
    Math  Science
A    90       88
B    80       84

Unsorted MultiIndex Series:
 A  X   -0.095928
B   Y    1.745636
A   Y    0.238070
B   X    0.768439
dtype: float64

Sorted by index:
 A  X   -0.095928
    Y    0.238070
B   X    0.768439
    Y    1.745636
dtype: float64
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[10], line 104
    101 print("Sorted by index:\n", unsorted.sort_index(), "\n")
    103 sub_df = df_auto[["Score1"]]  # remove one column
--> 104 print("Unused levels before removing:\n", sub_df.columns.levels, "\n")
    105 print("After remove_unused_levels:\n", sub_df.columns.remove_unused_levels
().levels, "\n")

AttributeError: 'Index' object has no attribute 'levels'
```

In [11]:
```python
# Program 1: Creating MultiIndex Series in different ways
import pandas as pd
import numpy as np

# From arrays
arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
multi_s = pd.Series([90, 85, 88, 92], index=index)
print("MultiIndex Series from arrays:\n", multi_s, "\n")

# From tuples
tuples = [("A", "Math"), ("A", "Science"), ("B", "Math"), ("B", "Science")]
index2 = pd.MultiIndex.from_tuples(tuples, names=("Alphabet", "Subject"))
multi_s2 = pd.Series([70, 75, 80, 82], index=index2)
print("MultiIndex Series from tuples:\n", multi_s2, "\n")

# From product
iterables = [["A", "B"], ["Math", "Science"]]
index3 = pd.MultiIndex.from_product(iterables, names=("Alphabet", "Subject"))
multi_s3 = pd.Series(np.random.randint(60, 100, size=4), index=index3)
print("MultiIndex Series from product:\n", multi_s3, "\n")
```

```
MultiIndex Series from arrays:
 Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64

MultiIndex Series from tuples:
 Alphabet  Subject
A         Math       70
          Science    75
B         Math       80
          Science    82
dtype: int64

MultiIndex Series from product:
 Alphabet  Subject
A         Math       85
          Science    83
B         Math       99
          Science    80
dtype: int32
```

In [12]:
```python
import pandas as pd
import numpy as np
```

```
# ------------------------------------------------------------
# Example 1: Creating a MultiIndex directly from arrays
# ------------------------------------------------------------
arrays = [
    ["A", "A", "B", "B"],
    ["Math", "Science", "Math", "Science"]
]
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))

df1 = pd.DataFrame(
    np.random.randint(50, 100, size=(4, 2)),
    index=index,
    columns=["Score1", "Score2"]
)
print("Example 1: MultiIndex from arrays\n", df1, "\n")

# ------------------------------------------------------------
# Example 2: Creating a MultiIndex from tuples
# ------------------------------------------------------------
tuples = [
    ("A", "Math"),
    ("A", "Science"),
    ("B", "Math"),
    ("B", "Science")
]
index2 = pd.MultiIndex.from_tuples(tuples, names=("Alphabet", "Subject"))

df2 = pd.DataFrame(
    np.random.randn(4, 2),
    index=index2,
    columns=["Value1", "Value2"]
)
print("Example 2: MultiIndex from tuples\n", df2, "\n")

# ------------------------------------------------------------
# Example 3: Creating a MultiIndex from product
# ------------------------------------------------------------
index3 = pd.MultiIndex.from_product(
    [["Group1", "Group2"], ["Math", "Science"]],
    names=("Group", "Subject")
)

df3 = pd.DataFrame(
    np.random.randint(1, 10, size=(4, 2)),
    index=index3,
    columns=["Col1", "Col2"]
)
print("Example 3: MultiIndex from product\n", df3, "\n")

# ------------------------------------------------------------
# Example 4: Creating MultiIndex directly from DataFrame
# ------------------------------------------------------------
data = {
    "Group": ["A", "A", "B", "B"],
    "Subject": ["Math", "Science", "Math", "Science"],
    "Score": [88, 92, 85, 90]
}

df4 = pd.DataFrame(data)
```

```
df4 = df4.set_index(["Group", "Subject"])  # setting multiple columns as index
print("Example 4: MultiIndex created from DataFrame columns\n", df4, "\n")
```

```
Example 1: MultiIndex from  arrays
                   Score1  Score2
Alphabet Subject
A        Math          59      88
         Science       59      91
B        Math          91      55
         Science       73      60

Example 2: MultiIndex from tuples
                     Value1    Value2
Alphabet Subject
A        Math      0.587060  0.262616
         Science  -0.642557 -0.753775
B        Math      2.248573 -0.103884
         Science  -0.641433  0.480017

Example 3: MultiIndex from product
                 Col1  Col2
Group  Subject
Group1 Math         5     2
       Science      9     3
Group2 Math         8     6
       Science      3     3

Example 4: MultiIndex created from DataFrame columns
                Score
Group Subject
A     Math         88
      Science      92
B     Math         85
      Science      90
```

In [13]:
```
#Program 2: Accessing and Indexing
# Program 2: Accessing and indexing in MultiIndex Series
import pandas as pd

arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
multi_s = pd.Series([90, 85, 88, 92], index=index)

print("Access all subjects for 'A':\n", multi_s.loc["A"], "\n")
print("Access specific element (B, Science):\n", multi_s.loc[("B", "Science")], "
```

```
Access all subjects for 'A':
 Subject
Math       90
Science    85
dtype: int64

Access specific element (B, Science):
 92
```

In [14]:
```
# Program 3: Slicing in MultiIndex Series
import pandas as pd

arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
```

```
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
multi_s = pd.Series([90, 85, 88, 92], index=index)

print("Slicing from A to B:\n", multi_s.loc["A":"B"], "\n")
print("Partial slice for all Math:\n", multi_s.loc[:, "Math"], "\n")
```

```
Slicing from A to B:
 Alphabet  Subject
A         Math       90
          Science    85
B         Math       88
          Science    92
dtype: int64

Partial slice for all Math:
 Alphabet
A    90
B    88
dtype: int64
```

In [15]:
```
# Program 4: Swapping and reordering levels
import pandas as pd

arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
multi_s = pd.Series([90, 85, 88, 92], index=index)

print("Swapping levels:\n", multi_s.swaplevel(), "\n")
print("Reordering levels:\n", multi_s.reorder_levels(["Subject", "Alphabet"]), "\
```

```
Swapping levels:
 Subject  Alphabet
Math     A           90
Science  A           85
Math     B           88
Science  B           92
dtype: int64

Reordering levels:
 Subject  Alphabet
Math     A           90
Science  A           85
Math     B           88
Science  B           92
dtype: int64
```

In [16]:
```
# Program 5: Passing arrays directly to create MultiIndex in Series/DataFrame
import pandas as pd
import numpy as np

# Series with automatic MultiIndex
multi_s_auto = pd.Series(
    np.random.randn(4),
    index=pd.MultiIndex.from_arrays([["A", "A", "B", "B"], ["X", "Y", "X", "Y"]])
)
print("MultiIndex Series constructed automatically:\n", multi_s_auto, "\n")

# DataFrame with automatic MultiIndex
df_auto = pd.DataFrame(
```

```
    np.random.randn(4, 2),
    index=pd.MultiIndex.from_arrays([["Group1", "Group1", "Group2", "Group2"],
                                     ["One", "Two", "One", "Two"]]),
    columns=["Score1", "Score2"]
)
print("DataFrame with MultiIndex automatically:\n", df_auto, "\n")
```

```
MultiIndex Series constructed automatically:
A  X   -0.489304
   Y   -0.216374
B  X   -1.083211
   Y    1.783962
dtype: float64

DataFrame with MultiIndex automatically:
               Score1    Score2
Group1 One -2.422487  1.880864
       Two  0.613409  1.285681
Group2 One  1.069546  0.299296
       Two -0.151640  1.133076
```

In [17]:
```
# Program 6: Data alignment and reindexing with MultiIndex
import pandas as pd

df = pd.DataFrame({
    "Math": [85, 90, 95, 80],
    "Science": [82, 88, 92, 84]
}, index=pd.MultiIndex.from_arrays([["A", "A", "B", "B"], ["one", "two", "one", "
print("Original DataFrame:\n", df, "\n")

# Group by first level and compute mean
mean_by_group = df.groupby(level=0).mean()
print("Mean by group:\n", mean_by_group, "\n")

# Reindexing with MultiIndex
aligned = mean_by_group.reindex(df.index, level=0)
print("Reindexed to align with original index:\n", aligned, "\n")
```

```
Original DataFrame:
        Math  Science
A one     85       82
  two     90       88
B one     95       92
  two     80       84

Mean by group:
    Math  Science
A  87.5     85.0
B  87.5     88.0

Reindexed to align with original index:
        Math  Science
A one  87.5     85.0
  two  87.5     85.0
B one  87.5     88.0
  two  87.5     88.0
```

In [18]:
```
# Program 8: Sorting MultiIndex and removing unused levels
import pandas as pd
```

```
import numpy as np

# Unsorted MultiIndex Series
multi_s = pd.Series(
    np.random.randn(4),
    index=pd.MultiIndex.from_arrays([["B", "A", "B", "A"], ["X", "Y", "Y", "X"]])
)
print("Unsorted MultiIndex Series:\n", multi_s, "\n")
print("Sorted by index:\n", multi_s.sort_index(), "\n")

# Removing unused levels
df = pd.DataFrame(
    np.random.randn(4, 2),
    index=pd.MultiIndex.from_arrays([["Group1", "Group1", "Group2", "Group2"], ["
    columns=pd.MultiIndex.from_arrays([["Score1", "Score2"], ["X", "Y"]])
)
print("Before removing unused levels:\n", df.columns.levels, "\n")
sub_df = df[["Score1"]] # drop Score2
print("After removing unused levels:\n", sub_df.columns.remove_unused_levels().le
```

```
Unsorted MultiIndex Series:
 B  X   -0.766789
A  Y   -0.023753
B  Y   -0.355786
A  X    1.519860
dtype: float64

Sorted by index:
 A  X    1.519860
   Y   -0.023753
B  X   -0.766789
   Y   -0.355786
dtype: float64

Before removing unused levels:
 [['Score1', 'Score2'], ['X', 'Y']]

After removing unused levels:
 [['Score1'], ['X']]
```

In [ ]:

Program 1: Creating DataFrames (different ways)

In [19]:
```
#Program 1: Creating DataFrames (different ways)
import pandas as pd
import numpy as np

# ------------------------------------------------------------------
# 1. From dictionary of lists
# ------------------------------------------------------------------
data1 = {"Name": ["Alice", "Bob", "Charlie"],
         "Age": [24, 27, 22],
         "Score": [85, 90, 88]}
df1 = pd.DataFrame(data1)
```

```
print("DataFrame from dictionary of lists:\n", df1, "\n")

# ------------------------------------------------------------------
# 2. From dictionary of Series
# ------------------------------------------------------------------
data2 = {"Math": pd.Series([90, 80, 85], index=["Alice", "Bob", "Charlie"]),
         "Science": pd.Series([88, 92, 84], index=["Alice", "Bob", "Charlie"])}
df2 = pd.DataFrame(data2)
print("DataFrame from dictionary of Series:\n", df2, "\n")

# ------------------------------------------------------------------
# 3. From NumPy array
# ------------------------------------------------------------------
df3 = pd.DataFrame(np.arange(9).reshape(3, 3),
                   columns=["Col1", "Col2", "Col3"])
print("DataFrame from NumPy array:\n", df3, "\n")
```

```
DataFrame from dictionary of lists:
       Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88

DataFrame from dictionary of Series:
          Math  Science
Alice       90       88
Bob         80       92
Charlie     85       84

DataFrame from NumPy array:
    Col1  Col2  Col3
0     0     1     2
1     3     4     5
2     6     7     8
```

In [20]:

```
#Program 2: Accessing rows and columns
import pandas as pd

data = {"Name": ["Alice", "Bob", "Charlie", "David"],
        "Age": [24, 27, 22, 30],
        "Score": [85, 90, 88, 95]}
df = pd.DataFrame(data)

print("Original DataFrame:\n", df, "\n")

# Access single column
print("Accessing single column (Score):\n", df["Score"], "\n")

# Access multiple columns
print("Accessing multiple columns:\n", df[["Name", "Age"]], "\n")

# Access row by index label
print("Access row using loc:\n", df.loc[2], "\n")

# Access row by integer location
print("Access row using iloc:\n", df.iloc[1], "\n")
```

```
Original DataFrame:
       Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88
3    David   30     95

Accessing single column (Score):
 0    85
1    90
2    88
3    95
Name: Score, dtype: int64

Accessing multiple columns:
       Name  Age
0    Alice   24
1      Bob   27
2  Charlie   22
3    David   30

Access row using loc:
 Name      Charlie
Age            22
Score          88
Name: 2, dtype: object

Access row using iloc:
 Name      Bob
Age        27
Score      90
Name: 1, dtype: object
```

In [21]:

```python
#Program 3: Indexing and Slicing
import pandas as pd

data = {"Name": ["Alice", "Bob", "Charlie", "David", "Eva"],
        "Age": [24, 27, 22, 30, 28],
        "Score": [85, 90, 88, 95, 89]}
df = pd.DataFrame(data)

print("Original DataFrame:\n", df, "\n")

# Slicing rows
print("First three rows:\n", df[:3], "\n")

# Slicing specific rows using loc
print("Rows 1 to 3:\n", df.loc[1:3], "\n")

# Slicing specific columns
print("Columns Name and Score:\n", df.loc[:, ["Name", "Score"]], "\n")

# Conditional selection
print("Rows where Score > 88:\n", df[df["Score"] > 88], "\n")
```

```
Original DataFrame:
      Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88
3    David   30     95
4      Eva   28     89

First three rows:
      Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88

Rows 1 to 3:
      Name  Age  Score
1      Bob   27     90
2  Charlie   22     88
3    David   30     95

Columns Name and Score:
      Name  Score
0    Alice     85
1      Bob     90
2  Charlie     88
3    David     95
4      Eva     89

Rows where Score > 88:
     Name  Age  Score
1     Bob   27     90
3   David   30     95
4     Eva   28     89
```

In [22]:
```python
#Program 4: Adding, Updating, and Deleting Data
import pandas as pd

df = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie"],
                   "Age": [24, 27, 22]})
 print("Original DataFrame:\n", df, "\n")

# Adding new column
df["Score"] = [85, 90, 88]
print("After adding Score column:\n", df, "\n")

# Updating values
df.at[1, "Age"] = 28
print("After updating Age of Bob:\n", df, "\n")

# Deleting column
df = df.drop("Score", axis=1)
print("After deleting Score column:\n", df, "\n")

# Deleting row
df = df.drop(2, axis=0)
print("After deleting row with index 2:\n", df, "\n")
```

```
Original DataFrame:
       Name  Age
0    Alice   24
1      Bob   27
2  Charlie   22

After adding Score column:
       Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88

After updating Age of Bob:
       Name  Age  Score
0    Alice   24     85
1      Bob   28     90
2  Charlie   22     88

After deleting Score column:
       Name  Age
0    Alice   24
1      Bob   28
2  Charlie   22

After deleting row with index 2:
     Name  Age
0  Alice   24
1    Bob   28
```

In [23]:
```python
#Program 5: Handling Missing Data
import pandas as pd
import numpy as np

df = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie"],
                   "Age": [24, np.nan, 22],
                   "Score": [85, 90, np.nan]})
print("Original DataFrame with NaN values:\n", df, "\n")

# Detect missing values
print("Detect missing values:\n", df.isnull(), "\n")

# Fill missing values
print("Fill missing values:\n", df.fillna(0), "\n")

# Drop rows with missing values
print("Drop rows with missing values:\n", df.dropna(), "\n")
```

```
Original DataFrame with NaN values:
       Name    Age  Score
0    Alice  24.0   85.0
1      Bob   NaN   90.0
2  Charlie  22.0    NaN

Detect missing values:
     Name    Age  Score
0  False  False  False
1  False   True  False
2  False  False   True

Fill missing values:
       Name    Age  Score
0    Alice  24.0   85.0
1      Bob   0.0   90.0
2  Charlie  22.0    0.0

Drop rows with missing values:
     Name    Age  Score
0  Alice  24.0   85.0
```

In [24]:
```python
#Program 6: Data Alignment and Reindexing
import pandas as pd

df1 = pd.DataFrame({"Score": [85, 90, 88]}, index=["Alice", "Bob", "Charlie"])
df2 = pd.DataFrame({"Score": [92, 80]}, index=["Bob", "David"])

print("DataFrame 1:\n", df1, "\n")
print("DataFrame 2:\n", df2, "\n")

# Automatic alignment in arithmetic
print("Adding df1 and df2:\n", df1 + df2, "\n")

# Reindexing
df3 = df1.reindex(["Alice", "Bob", "Charlie", "David"], fill_value=0)
print("Reindexed DataFrame:\n", df3, "\n")
```

```
DataFrame 1:
          Score
Alice        85
Bob          90
Charlie      88

DataFrame 2:
        Score
Bob        92
David      80

Adding df1 and df2:
          Score
Alice       NaN
Bob       182.0
Charlie     NaN
David       NaN

Reindexed DataFrame:
          Score
Alice        85
Bob          90
Charlie      88
David         0
```

In [25]:
```python
#Program 7: Sorting and Grouping
import pandas as pd

data = {"Name": ["Alice", "Bob", "Charlie", "David", "Eva"],
        "Age": [24, 27, 22, 30, 28],
        "Score": [85, 90, 88, 95, 89]}
df = pd.DataFrame(data)

print("Original DataFrame:\n", df, "\n")

# Sorting by column
print("Sorted by Score:\n", df.sort_values(by="Score"), "\n")

# Grouping data
grouped = df.groupby("Age")["Score"].mean()
print("Average score grouped by Age:\n", grouped, "\n")
```

```
Original DataFrame:
      Name  Age  Score
0    Alice   24     85
1      Bob   27     90
2  Charlie   22     88
3    David   30     95
4      Eva   28     89

Sorted by Score:
      Name  Age  Score
0    Alice   24     85
2  Charlie   22     88
4      Eva   28     89
1      Bob   27     90
3    David   30     95

Average score grouped by Age:
 Age
22    88.0
24    85.0
27    90.0
28    89.0
30    95.0
Name: Score, dtype: float64
```

In [26]:
```python
#Grouping Example
#A. Series
import pandas as pd

# Salary series
salary = pd.Series([50000, 55000, 60000, 62000],
                   index=["Alice", "Bob", "Charlie", "David"])
# Department for each person
department = pd.Series(["HR", "HR", "IT", "IT"],
                       index=["Alice", "Bob", "Charlie", "David"])

# Grouping by department (no aggregation yet)
grouped_series = salary.groupby(department)
print("Grouped Series (just groups, no aggregation):\n", grouped_series, "\n")

# You can access a group
print("HR group in Series:\n", grouped_series.get_group("HR"), "\n")
```

```
Grouped Series (just groups, no aggregation):
 <pandas.core.groupby.generic.SeriesGroupBy object at 0x000001E9DB9E98E0>

HR group in Series:
 Alice    50000
Bob      55000
dtype: int64
```

In [27]:
```python
#Aggregation Example
#A. Series
# Aggregation on grouped Series
# Compute mean salary per department
aggregated_series = grouped_series.mean()
print("Aggregated Series (mean salary per department):\n", aggregated_series, "\n
```

```
# Other aggregations
print("Sum per department:\n", grouped_series.sum(), "\n")
print("Max per department:\n", grouped_series.max(), "\n")
```

```
Aggregated Series (mean salary per department):
 HR    52500.0
IT    61000.0
dtype: float64

Sum per department:
 HR    105000
IT    122000
dtype: int64

Max per department:
 HR    55000
IT    62000
dtype: int64
```

In [28]:
```
#Example: Grouping and Aggregating for Series / MultiLevel Series
import pandas as pd

# Single Series
salary = pd.Series([50000, 55000, 60000, 62000],
                   index=["Alice", "Bob", "Charlie", "David"])
department = pd.Series(["HR", "HR", "IT", "IT"],
                       index=["Alice", "Bob", "Charlie", "David"])

# Grouping salaries by department and calculating mean
grouped_series = salary.groupby(department).mean()
print("Mean salary by department (Series):\n", grouped_series, "\n")

# MultiLevel Series
arrays = [
    ["HR", "HR", "IT", "IT"],
    ["Alice", "Bob", "Charlie", "David"]
]
index = pd.MultiIndex.from_arrays(arrays, names=("Dept", "Employee"))
multi_s = pd.Series([50000, 55000, 60000, 62000], index=index)

# Group by first level (Dept)
grouped_multi = multi_s.groupby(level=0).mean()
print("Mean salary by department (MultiLevel Series):\n", grouped_multi, "\n")
```

```
Mean salary by department (Series):
 HR    52500.0
IT    61000.0
dtype: float64

Mean salary by department (MultiLevel Series):
 Dept
HR    52500.0
IT    61000.0
dtype: float64
```

In [29]:
```
#Grouping (DataFrame)
import pandas as pd

# DataFrame
```

```
df = pd.DataFrame({
    "Department": ["HR", "HR", "IT", "IT"],
    "Employee": ["Alice", "Bob", "Charlie", "David"],
    "Salary": [50000, 55000, 60000, 62000]
})

# Grouping by Department
grouped_df = df.groupby("Department")
print("Grouped DataFrame (just groups, no aggregation):\n", grouped_df, "\n")

# Accessing one group
print("HR group in DataFrame:\n", grouped_df.get_group("HR"), "\n")
```

```
Grouped DataFrame (just groups, no aggregation):
 <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001E9DA890E60>

HR group in DataFrame:
   Department Employee  Salary
0         HR    Alice   50000
1         HR      Bob   55000
```

In [30]:
```
# Aggregating DataFrame
# Aggregation on grouped DataFrame
# Compute mean salary per department
aggregated_df = grouped_df["Salary"].mean()
print("Aggregated DataFrame (mean salary per department):\n", aggregated_df, "\n"

# Multiple aggregations on DataFrame
multi_agg_df = grouped_df.agg({
    "Salary": ["mean", "sum", "max"]
})
print("Aggregated DataFrame (multiple stats):\n", multi_agg_df, "\n")
```

```
Aggregated DataFrame (mean salary per department):
 Department
HR    52500.0
IT    61000.0
Name: Salary, dtype: float64

Aggregated DataFrame (multiple stats):
              Salary
               mean     sum    max
Department
HR          52500.0  105000  55000
IT          61000.0  122000  62000
```

In [31]:
```
import pandas as pd

# ---------------------------------------------------------------
# Sample DataFrame
# ---------------------------------------------------------------
df = pd.DataFrame({
    "Department": ["HR", "HR", "IT", "IT", "HR", "IT"],
    "Employee": ["Alice", "Bob", "Charlie", "David", "Eva", "Frank"],
    "Salary": [50000, 55000, 60000, 62000, 58000, 61000],
    "Bonus": [5000, 6000, 7000, 8000, 5500, 7500]
})

print("Original DataFrame:\n", df, "\n")
```

```
# ------------------------------------------------------------
# Grouping by Department (no aggregation yet)
# ------------------------------------------------------------
grouped = df.groupby("Department")
print("Groups formed:\n", grouped.groups, "\n")

# Access a specific group
print("HR group:\n", grouped.get_group("HR"), "\n")

# ------------------------------------------------------------
# Aggregating after grouping
# ------------------------------------------------------------
# 1. Single aggregation function (mean salary)
mean_salary = grouped["Salary"].mean()
print("Mean Salary per Department:\n", mean_salary, "\n")

# 2. Multiple aggregation functions
agg_stats = grouped.agg({
    "Salary": ["mean", "max", "min"],
    "Bonus": ["sum", "mean"]
})
print("Aggregated stats per Department:\n", agg_stats, "\n")

# ------------------------------------------------------------
# Optional: filtering groups (e.g., mean salary > 55000)
# ------------------------------------------------------------
high_salary_dept = grouped.filter(lambda x: x["Salary"].mean() > 55000)
print("Departments with mean salary > 55000:\n", high_salary_dept, "\n")
```

```
Original DataFrame:
   Department Employee  Salary  Bonus
0          HR    Alice   50000   5000
1          HR      Bob   55000   6000
2          IT  Charlie   60000   7000
3          IT    David   62000   8000
4          HR      Eva   58000   5500
5          IT    Frank   61000   7500

Groups formed:
 {'HR': [0, 1, 4], 'IT': [2, 3, 5]}

HR group:
   Department Employee  Salary  Bonus
0          HR    Alice   50000   5000
1          HR      Bob   55000   6000
4          HR      Eva   58000   5500

Mean Salary per Department:
 Department
HR    54333.333333
IT    61000.000000
Name: Salary, dtype: float64

Aggregated stats per Department:
                    Salary                 Bonus
                      mean    max    min    sum    mean
Department
HR           54333.333333  58000  50000  16500  5500.0
IT           61000.000000  62000  60000  22500  7500.0

Departments with mean salary > 55000:
   Department Employee  Salary  Bonus
2          IT  Charlie   60000   7000
3          IT    David   62000   8000
5          IT    Frank   61000   7500
```

Difference between grouping/ aggregating in series and dataframes

In grouping, a Series can be grouped only by another Series or index level, while a DataFrame can be grouped by one or more columns.

Series aggregation operates on a single column and returns a Series, whereas DataFrame aggregation can operate on multiple columns and return a DataFrame.

Series is simpler and suitable for single-column data, while DataFrame is more powerful for handling complex, multi-column datasets.

In [32]:
```
#series
import pandas as pd
salary = pd.Series([50000, 55000, 60000, 62000], index=["Alice","Bob","Charlie","
```

```
department = pd.Series(["HR","HR","IT","IT"], index=["Alice","Bob","Charlie","Dav

grouped_series = salary.groupby(department)
```

In [33]:
```
#data frames
df = pd.DataFrame({
    "Department":["HR","HR","IT","IT"],
    "Employee":["Alice","Bob","Charlie","David"],
    "Salary":[50000,55000,60000,62000]
})

grouped_df = df.groupby("Department")
```

In [34]:
```
import pandas as pd

# ---------------------------------------------------------------
# 1. Series Example
# ---------------------------------------------------------------
salary_series = pd.Series([50000, 55000, 60000, 62000],
                          index=["Alice", "Bob", "Charlie", "David"])
department_series = pd.Series(["HR", "HR", "IT", "IT"],
                              index=["Alice", "Bob", "Charlie", "David"])

# Grouping (creates groups, no aggregation yet)
grouped_series = salary_series.groupby(department_series)
print("Grouped Series (no aggregation):")
for dept, group in grouped_series:
    print(f"{dept}: {group.values}")
print()

# Aggregating (mean per department)
aggregated_series = grouped_series.mean()
print("Aggregated Series (mean salary per department):")
print(aggregated_series)
print("\n" + "="*50 + "\n")

# ---------------------------------------------------------------
# 2. DataFrame Example
# ---------------------------------------------------------------
df = pd.DataFrame({
    "Department": ["HR", "HR", "IT", "IT"],
    "Employee": ["Alice", "Bob", "Charlie", "David"],
    "Salary": [50000, 55000, 60000, 62000],
    "Bonus": [5000, 6000, 7000, 8000]
})

# Grouping by Department (creates groups)
grouped_df = df.groupby("Department")
print("Grouped DataFrame (no aggregation):")
for dept, group in grouped_df:
    print(f"{dept} group:\n{group}\n")

# Aggregating (mean and sum for numeric columns)
aggregated_df = grouped_df.agg({
    "Salary": ["mean", "sum"],
    "Bonus": ["mean", "sum"]
})
print("Aggregated DataFrame (Salary and Bonus stats per Department):")
print(aggregated_df)
```

```
Grouped Series (no aggregation):
HR: [50000 55000]
IT: [60000 62000]

Aggregated Series (mean salary per department):
HR    52500.0
IT    61000.0
dtype: float64

================================================

Grouped DataFrame (no aggregation):
HR group:
  Department Employee  Salary  Bonus
0         HR    Alice   50000   5000
1         HR      Bob   55000   6000

IT group:
  Department Employee  Salary  Bonus
2         IT  Charlie   60000   7000
3         IT    David   62000   8000

Aggregated DataFrame (Salary and Bonus stats per Department):
              Salary            Bonus
                mean     sum    mean    sum
Department
HR           52500.0  105000  5500.0  11000
IT           61000.0  122000  7500.0  15000
```

In [35]:
```python
#merging DataFRames
import pandas as pd

# DataFrame 1
df1 = pd.DataFrame({
    "Employee": ["Alice", "Bob", "Charlie", "David"],
    "Department": ["HR", "IT", "HR", "IT"]
})

# DataFrame 2
df2 = pd.DataFrame({
    "Employee": ["Alice", "Bob", "Charlie", "Eva"],
    "Salary": [50000, 60000, 55000, 58000]
})

# Inner merge (only common employees)
inner_merge = pd.merge(df1, df2, on="Employee", how="inner")
print("Inner Merge:\n", inner_merge, "\n")

# Outer merge (all employees)
outer_merge = pd.merge(df1, df2, on="Employee", how="outer")
print("Outer Merge:\n", outer_merge, "\n")
```

```
Inner Merge:
   Employee Department  Salary
0    Alice          HR   50000
1      Bob          IT   60000
2  Charlie          HR   55000

Outer Merge:
   Employee Department    Salary
0    Alice          HR  50000.0
1      Bob          IT  60000.0
2  Charlie          HR  55000.0
3    David          IT      NaN
4      Eva         NaN  58000.0
```

In [36]:
```
#Merging on different key columns

#You can merge two DataFrames using different column names in each DataFrame by u

#Example:

import pandas as pd

df1 = pd.DataFrame({
    "EmpID": [1, 2, 3],
    "Name": ["Alice", "Bob", "Charlie"]
})

df2 = pd.DataFrame({
    "EmployeeID": [2, 3, 4],
    "Salary": [60000, 55000, 58000]
})

# Merge using different column names
merged_df = pd.merge(df1, df2, left_on="EmpID", right_on="EmployeeID", how="inner
print("Merge on different keys:\n", merged_df)
```

```
Merge on different keys:
    EmpID     Name  EmployeeID  Salary
0      2      Bob           2   60000
1      3  Charlie           3   55000
```

In [37]:
```
#Handling overlapping column names

#If both DataFrames have columns with the same name other than the key, Pandas au

#You can customize suffixes using the suffixes parameter.

df1 = pd.DataFrame({
    "Employee": ["Alice", "Bob"],
    "Salary": [50000, 60000]
})

df2 = pd.DataFrame({
    "Employee": ["Bob", "Charlie"],
    "Salary": [65000, 55000]
})

# Merge with custom suffixes
```

```
merged_df = pd.merge(df1, df2, on="Employee", how="outer", suffixes=("_Old", "_Ne
print("Merge with custom suffixes:\n", merged_df)
```

```
Merge with custom suffixes:
   Employee  Salary_Old  Salary_New
0     Alice     50000.0         NaN
1       Bob     60000.0     65000.0
2   Charlie         NaN     55000.0
```

In [38]:
```
#Merging using indexes

#Instead of merging on columns, you can merge using the row index with left_index

df1 = pd.DataFrame({"Salary": [50000, 60000]}, index=["Alice", "Bob"])
df2 = pd.DataFrame({"Department": ["HR", "IT"]}, index=["Alice", "Bob"])

# Merge using index
merged_index_df = pd.merge(df1, df2, left_index=True, right_index=True)
print("Merge using indexes:\n", merged_index_df)
```

```
Merge using indexes:
        Salary Department
Alice    50000         HR
Bob      60000         IT
```

Summary table

df.pivot_table(values='NumericColumn', index='RowCategory', columns='ColumnCategory', aggfunc='mean') values → numeric column to summarize

index → row labels

columns → column labels (optional)

aggfunc → aggregation function (mean, sum, count, etc.)

In [39]:
```
#Summary Tables
df = pd.DataFrame({
    "Department": ["HR","HR","IT","IT"],
    "Team": ["A","B","A","B"],
    "Salary": [50000,55000,60000,62000]
})
summary = df.pivot_table(values="Salary", index="Department", columns="Team", agg
print(summary)
```

```
Team               A        B
Department
HR           50000.0  55000.0
IT           60000.0  62000.0
```

In [40]:
```
# summary table pivot table
import pandas as pd

df = pd.DataFrame({
    "Department": ["HR", "HR", "IT", "IT", "HR", "IT"],
    "Team": ["A", "B", "A", "B", "A", "B"],
    "Salary": [50000, 55000, 60000, 62000, 58000, 61000]
})
```

```
# Pivot table: average salary by Department and Team
summary_table = df.pivot_table(values="Salary", index="Department", columns="Team
print("Summary Table (Average Salary):\n", summary_table)
```

```
Summary Table (Average Salary):
 Team              A        B
Department
HR          54000.0  55000.0
IT          60000.0  61500.0
```

In [41]:
```
#Pivot Table with Multiple Aggregation Functions

#You can apply more than one aggregation function at once using a list:

summary_table_multi = df.pivot_table(
    values="Salary",
    index="Department",
    columns="Team",
    aggfunc=["mean", "sum", "max"]
)
print("Pivot Table with Multiple Aggregations:\n", summary_table_multi)
```

```
Pivot Table with Multiple Aggregations:
                mean               sum             max
Team               A        B       A       B      A      B
Department
HR          54000.0  55000.0  108000   55000  58000  55000
IT          60000.0  61500.0   60000  123000  60000  62000
```

In [42]:
```
#Handling Missing Data

#Pivot tables automatically fill missing combinations with NaN.

#You can replace missing values with fill_value:

summary_table_fill = df.pivot_table(
    values="Salary",
    index="Department",
    columns="Team",
    aggfunc="mean",
    fill_value=0
)
print("Pivot Table with Missing Values Filled:\n", summary_table_fill)
```

```
Pivot Table with Missing Values Filled:
Team              A        B
Department
HR          54000.0  55000.0
IT          60000.0  61500.0
```

In [43]:
```
#Grouping vs Pivot Tables

#Pivot tables are essentially groupby + aggregation + reshape in one step.

#They are easier to read when summarizing across two categorical variables.

# Equivalent using groupby
grouped = df.groupby(["Department", "Team"])["Salary"].mean().unstack()
print("Equivalent using groupby + unstack:\n", grouped)
```

```
Equivalent using groupby + unstack:
Team              A        B
Department
HR          54000.0  55000.0
IT          60000.0  61500.0
```

In [ ]: