WORKING ON PANDAS SERIES

In [1]:
```
import pandas as pd               # Import pandas library as pd

# Initializing a Series from a list
data = [1, 2.3, 'a', 4, 5]        # Create list with mixed values
series_from_list = pd.Series(data) # Convert list into pandas Series
print(series_from_list)            # Display the created Series
```

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

In [3]:
```
# Creating a DataFrame
data = {                                # Define dictionary for DataFrame
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],  # List of student names
    'Age': [23, 25, 22, 24],            # List of corresponding ages
    'Score': [85, 90, 78, 92]           # List of corresponding scores
}

df = pd.DataFrame(data)                 # Create DataFrame using dictionary
print(df)                               # Print the DataFrame contents
```

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

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

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

In [7]:
```
series_a = pd.Series([1, 2, 3])   # Create first Series with numbers
series_b = pd.Series([4, 5, 6])   # Create second Series with numbers
sum_series = series_a + series_b  # Add corresponding elements of Series
print(sum_series)                 # Print the summed Series result
```

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

In [9]:
```
# Creating a MultiIndex Series
arrays = [                               # Define nested lists for index
    ['A', 'A', 'B', 'B'],                # First level: Alphabet labels
    ['Math', 'Science', 'Math', 'Science'] # Second level: Subject labels
]
index = pd.MultiIndex.from_arrays(arrays, names=('Alphabet', 'Subject'))
# Create MultiIndex from arrays with names

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex values

print(multi_s)                           # Print the MultiIndex Series
```

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

In [11]:
```
# Creating a MultiIndex Series
import pandas as pd                      # Import pandas library as pd

arrays = [                               # Define nested lists for index
    ['A', 'A', 'B', 'B'],                # First level: Alphabet labels
    ['Math', 'Science', 'Math', 'Science'] # Second level: Subject labels
]

index = pd.MultiIndex.from_arrays(arrays, names=('Alphabet', 'Subject'))
# Create MultiIndex object with two levels

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex and values

print(multi_s)                           # Display MultiIndex Series output
```

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

In [13]:
```
import pandas as pd                               # Import pandas library as pd

tuples = [('A', 'Math'), ('A', 'Science'), ('B', 'Math'), ('B', 'Science')]
```

```
# Define list of tuple pairs for index

index = pd.MultiIndex.from_tuples(tuples, names=('Alphabet', 'Subject'))
# Create MultiIndex object from tuple list

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex and values

print(multi_s)
```

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

In [15]:

```
index = pd.MultiIndex.from_product(             # Create MultiIndex using product
    [['A', 'B'], ['Math', 'Science']],          # Cartesian product of lists
    names=('Alphabet', 'Subject')               # Assign level names
)

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex values

print(multi_s)
```

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

In [17]:

```
df = pd.DataFrame({                           # Create DataFrame with two columns
    'Alphabet': ['A', 'A', 'B', 'B'],         # Column for Alphabet labels
    'Subject': ['Math', 'Science', 'Math', 'Science'] # Column for Subject labels
})
index = pd.MultiIndex.from_frame(df, names=('Alphabet', 'Subject'))
# Create MultiIndex directly from DataFrame

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex values

print(multi_s)
```

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

In [19]:

```
import pandas as pd                         # Import pandas library as pd
import numpy as np                          # Import NumPy library as np

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

# From arrays
arrays = [                                  # Nested lists for MultiIndex
    ["A", "A", "B", "B"],                   # First level: Alphabet
    ["Math", "Science", "Math", "Science"]# Second level: Subject
]
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
# Create MultiIndex from arrays
multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex
print("MultiIndex Series from arrays:\n", multi_s, "\n")
# Display Series

# From tuples
tuples = [                                 # List of tuple pairs
    ("A", "Math"), ("A", "Science"),       # Tuples for level values
    ("B", "Math"), ("B", "Science")
]
index2 = pd.MultiIndex.from_tuples(tuples, names=("Alphabet", "Subject"))
# Create MultiIndex from tuples
multi_s2 = pd.Series([70, 75, 80, 82], index=index2)
# Series with tuple-based MultiIndex
print("MultiIndex Series from tuples:\n", multi_s2, "\n")
# Display Series

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

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

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

```
# Access single element by tuple

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

print("Slicing from A to B:\n", multi_s.loc["A":"B"], "\n")
# Slice from first to second level
print("Partial slice for all Math:\n", multi_s.loc[:, "Math"], "\n")
# Slice across first level for "Math"

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

print("Swapping levels:\n", multi_s.swaplevel(), "\n")
# Swap two MultiIndex levels
print("Reordering levels:\n", multi_s3.reorder_levels(["Subject", "Alphabet"]), "\n")
# Reorder levels in custom order

# -----------------------------------------------------------
# 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"]])
)
# Create Series with automatic MultiIndex
print("MultiIndex Series constructed automatically:\n", multi_s_auto, "\n")
# Display Series

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"]
)
# Create DataFrame with MultiIndex rows
print("DataFrame with MultiIndex automatically:\n", df_auto, "\n")
# Display DataFrame

# -----------------------------------------------------------
# 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", "two"]]))
# DataFrame with MultiIndex rows
print("Original DataFrame:\n", df, "\n")
# Display DataFrame

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

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

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

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

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

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

sub_df = df_auto[["Score1"]]  # Remove one column
print("Unused levels before removing:\n", sub_df.columns.levels, "\n")
# Show levels before removal
print("After remove_unused_levels:\n", sub_df.columns.remove_unused_levels().levels, "\n")
# Remove unused levels from MultiIndex
```

```
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       76
          Science    95
B         Math       75
          Science    94
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           76
Science  A           95
Math     B           75
Science  B           94
dtype: int32

MultiIndex Series constructed automatically:
 A  X   -0.855940
   Y   -1.316288
B  X    2.047078
   Y   -0.204769
dtype: float64

DataFrame with MultiIndex automatically:
               Score1    Score2
Group1 One  1.651829  0.254939
       Two  0.028699  0.519734
Group2 One -0.622525  0.525064
       Two  0.013233  1.163239

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:
 B  Y   -0.204769
A  X   -0.855940
   Y   -1.316288
B  X    2.047078
dtype: float64

Sorted by index:
 A  X   -0.855940
   Y   -1.316288
B  X    2.047078
   Y   -0.204769
dtype: float64
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[19], line 133
    130 # Display Series sorted by MultiIndex
    132 sub_df = df_auto[["Score1"]]  # Remove one column
--> 133 print("Unused levels before removing:\n", sub_df.columns.levels, "\n")
    134 # Show levels before removal
    135 print("After remove_unused_levels:\n", sub_df.columns.remove_unused_levels().levels, "\n")

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

error solved

In [25]:

```
# Program 1: Creating MultiIndex Series in different ways
import pandas as pd                         # Import pandas library as pd
import numpy as np                          # Import NumPy library as np

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

# From tuples
tuples = [("A", "Math"), ("A", "Science"), ("B", "Math"), ("B", "Science")]
# List of tuples for MultiIndex
index2 = pd.MultiIndex.from_tuples(tuples, names=("Alphabet", "Subject"))
# Create MultiIndex from tuples
multi_s2 = pd.Series([70, 75, 80, 82], index=index2)
# Create Series with tuple-based MultiIndex
print("MultiIndex Series from tuples:\n", multi_s2, "\n")
# Print Series

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

```
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       64
          Science    73
B         Math       86
          Science    86
dtype: int32
```

In [27]:

```
import pandas as pd                          # Import pandas library as pd
import numpy as np                           # Import NumPy library as np

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

```
# Create MultiIndex from arrays

df1 = pd.DataFrame(
    np.random.randint(50, 100, size=(4, 2)), # Random integers 50-99
    index=index,                              # Set MultiIndex as row index
    columns=["Score1", "Score2"]             # Column names
)
print("Example 1: MultiIndex from arrays\n", df1, "\n")
# Display DataFrame

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

df2 = pd.DataFrame(
    np.random.randn(4, 2),                 # Random float numbers
    index=index2,                           # Set MultiIndex as row index
    columns=["Value1", "Value2"]           # Column names
)
print("Example 2: MultiIndex from tuples\n", df2, "\n")
# Display DataFrame

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

df3 = pd.DataFrame(
    np.random.randint(1, 10, size=(4, 2)),  # Random integers 1-9
    index=index3,                            # MultiIndex rows
    columns=["Col1", "Col2"]                 # Column names
)
print("Example 3: MultiIndex from product\n", df3, "\n")
# Display DataFrame

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

df4 = pd.DataFrame(data)                    # Create DataFrame
df4 = df4.set_index(["Group", "Subject"])  # Set multiple columns as index
print("Example 4: MultiIndex created from DataFrame columns\n", df4, "\n")
# Display final DataFrame with MultiIndex
```

```
Example 1: MultiIndex from arrays
                   Score1  Score2
Alphabet Subject
A        Math          76      94
         Science       71      86
B        Math          94      73
         Science       77      63

Example 2: MultiIndex from tuples
                     Value1    Value2
Alphabet Subject
A        Math     0.964726 -0.216546
         Science  0.339988 -0.910130
B        Math     0.874872 -1.662070
         Science -0.682999  1.052067

Example 3: MultiIndex from product
                 Col1  Col2
Group  Subject
Group1 Math         6     2
       Science      2     7
Group2 Math         4     2
       Science      6     4

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

In [29]:

```
# Program 2: Accessing and indexing in MultiIndex Series
import pandas as pd                      # Import pandas library as pd

arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
# Nested lists for MultiIndex levels
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
# Create MultiIndex from arrays

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex
```

```
print("Access all subjects for 'A':\n", multi_s.loc["A"], "\n")
# Access all entries for first level "A"

print("Access specific element (B, Science):\n", multi_s.loc[("B", "Science")], "\n")
# Access single element using tuple key
```

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

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

In [31]:

```
# Program 3: Slicing in MultiIndex Series
import pandas as pd                     # Import pandas library as pd

arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
# Nested lists for MultiIndex levels
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
# Create MultiIndex from arrays

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex

print("Slicing from A to B:\n", multi_s.loc["A":"B"], "\n")
# Slice Series from first to last index

print("Partial slice for all Math:\n", multi_s.loc[:, "Math"], "\n")
# Slice across first level for second level "Math"
```

```
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 [33]:

```
# Program 4: Swapping and reordering levels
import pandas as pd                     # Import pandas library as pd

arrays = [["A", "A", "B", "B"], ["Math", "Science", "Math", "Science"]]
# Nested lists for MultiIndex levels
index = pd.MultiIndex.from_arrays(arrays, names=("Alphabet", "Subject"))
# Create MultiIndex from arrays

multi_s = pd.Series([90, 85, 88, 92], index=index)
# Create Series with MultiIndex

print("Swapping levels:\n", multi_s.swaplevel(), "\n")
# Swap first and second MultiIndex levels

print("Reordering levels:\n", multi_s.reorder_levels(["Subject", "Alphabet"]), "\n")
# Reorder levels in custom order
```

```
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 [ ]:

MULTIINDEX IN SERIES

In [1]:

```
# Program 5: Passing arrays directly to create MultiIndex in Series/DataFrame
import pandas as pd                      # Import pandas library as pd
import numpy as np                       # Import NumPy library as np

# Series with automatic MultiIndex
multi_s_auto = pd.Series(
    np.random.randn(4),                  # Generate 4 random float numbers
    index=pd.MultiIndex.from_arrays([["A", "A", "B", "B"], ["X", "Y", "X", "Y"]])
    # Create MultiIndex directly from arrays
)
print("MultiIndex Series constructed automatically:\n", multi_s_auto, "\n")
# Print automatically created MultiIndex Series

# DataFrame with automatic MultiIndex
df_auto = pd.DataFrame(
    np.random.randn(4, 2),               # Generate 4x2 random float numbers
    index=pd.MultiIndex.from_arrays([["Group1", "Group1", "Group2", "Group2"],
                                     ["One", "Two", "One", "Two"]]),
    # Create MultiIndex for DataFrame rows
    columns=["Score1", "Score2"]         # Assign column names
)
print("DataFrame with MultiIndex automatically:\n", df_auto, "\n")
# Print DataFrame with automatic MultiIndex
```

```
MultiIndex Series constructed automatically:
 A  X    0.051284
   Y   -0.597297
B  X    1.196862
   Y    1.390003
dtype: float64

DataFrame with MultiIndex automatically:
               Score1    Score2
Group1 One -0.821329 -1.071760
       Two -0.508296 -1.055866
Group2 One  0.923310  0.204589
       Two -0.803608  0.008728
```

In [3]:

```
# Program 6: Data alignment and reindexing with MultiIndex
import pandas as pd                      # Import pandas library as pd

df = pd.DataFrame({
    "Math": [85, 90, 95, 80],           # Math scores for each row
    "Science": [82, 88, 92, 84]         # Science scores for each row
}, index=pd.MultiIndex.from_arrays([["A", "A", "B", "B"], ["one", "two", "one", "two"]]))
# Create DataFrame with MultiIndex rows

print("Original DataFrame:\n", df, "\n")
# Print original DataFrame

# Group by first level and compute mean
mean_by_group = df.groupby(level=0).mean()
# Compute mean values by first index level

print("Mean by group:\n", mean_by_group, "\n")
# Display group mean

# Reindexing with MultiIndex
aligned = mean_by_group.reindex(df.index, level=0)
# Align group means to original MultiIndex rows

print("Reindexed to align with original index:\n", aligned, "\n")
# Display reindexed DataFrame
```

```
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 [5]:

```
# Program 8: Sorting MultiIndex and removing unused levels
import pandas as pd                      # Import pandas library as pd
import numpy as np                       # Import NumPy library as np

# Unsorted MultiIndex Series
multi_s = pd.Series(
    np.random.randn(4),                  # Generate 4 random float numbers
    index=pd.MultiIndex.from_arrays([["B", "A", "B", "A"], ["X", "Y", "Y", "X"]])
    # Create MultiIndex with unsorted values
)
```

```
print("Unsorted MultiIndex Series:\n", multi_s, "\n")
# Display unsorted MultiIndex Series

print("Sorted by index:\n", multi_s.sort_index(), "\n")
# Display Series sorted by MultiIndex

# Removing unused levels
df = pd.DataFrame(
    np.random.randn(4, 2),               # Generate 4x2 random float numbers
    index=pd.MultiIndex.from_arrays([["Group1", "Group1", "Group2", "Group2"], ["One", "Two", "One", "Two"]]),
    # MultiIndex for rows
    columns=pd.MultiIndex.from_arrays([["Score1", "Score2"], ["X", "Y"]])
    # MultiIndex for columns
)
print("Before removing unused levels:\n", df.columns.levels, "\n")
# Display column levels before removing

sub_df = df[["Score1"]]                   # Drop column Score2
print("After removing unused levels:\n", sub_df.columns.remove_unused_levels().levels, "\n")
# Remove and display unused column levels
```

```
Unsorted MultiIndex Series:
 B  X    1.094073
A  Y    1.477834
B  Y   -0.483658
A  X    0.175345
dtype: float64

Sorted by index:
 A  X    0.175345
   Y    1.477834
B  X    1.094073
   Y   -0.483658
dtype: float64

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

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

CREATING DATAFRAMES DIFFERENT WAYS

In [9]:

```
# Program 1: Creating DataFrames (different ways)
import pandas as pd                      # Import pandas library as pd
import numpy as np                       # Import NumPy library as np

# ------------------------------------------
# 1. From dictionary of lists
# ------------------------------------------
data1 = {"Name": ["Alice", "Bob", "Charlie"],  # Dictionary with lists
         "Age": [24, 27, 22],                 # Age list
         "Score": [85, 90, 88]}               # Score list
df1 = pd.DataFrame(data1)                     # Create DataFrame from dictionary
print("DataFrame from dictionary of lists:\n", df1, "\n")
# Print DataFrame

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

# ------------------------------------------
# 3. From NumPy array
# ------------------------------------------
df3 = pd.DataFrame(np.arange(9).reshape(3, 3),  # Create 3x3 array
                   columns=["Col1", "Col2", "Col3"])
                   # Assign column names
print("DataFrame from NumPy array:\n", df3, "\n")
# Print DataFrame
```

```
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 [11]:

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

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

print("Original DataFrame:\n", df, "\n")
# Print original DataFrame

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

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

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

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

```
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 [13]:

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

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

print("Original DataFrame:\n", df, "\n")
# Print original DataFrame

# Slicing rows
print("First three rows:\n", df[:3], "\n")
# Slice first three rows using standard indexing

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

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

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

```
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 [15]:

```
# Program 4: Adding, Updating, and Deleting Data
import pandas as pd                      # Import pandas library as pd

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

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

# Updating values
df.at[1, "Age"] = 28                     # Update Age for index 1 (Bob)
print("After updating Age of Bob:\n", df, "\n")
# Display DataFrame after updating value

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

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

```
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 [17]:

```
# Program 5: Handling Missing Data
import pandas as pd                      # Import pandas library as pd
import numpy as np                       # Import NumPy library as np

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

# Detect missing values
print("Detect missing values:\n", df.isnull(), "\n")
# Check and display missing values as boolean

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

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

```
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 [19]:

```
# Program 6: Data Alignment and Reindexing
import pandas as pd                     # Import pandas library as pd

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

print("DataFrame 1:\n", df1, "\n")
# Display first DataFrame
print("DataFrame 2:\n", df2, "\n")
# Display second DataFrame

# Automatic alignment in arithmetic
print("Adding df1 and df2:\n", df1 + df2, "\n")
# Add DataFrames; aligns by index automatically

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

```
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 [21]:

```
# Program 7: Sorting and Grouping
import pandas as pd                     # Import pandas library as pd

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

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

```python
# Display original DataFrame

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

# Grouping data
grouped = df.groupby("Age")["Score"].mean()
# Group by Age and compute mean Score

print("Average score grouped by Age:\n", grouped, "\n")
# Display grouped average scores
```

```
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 [23]:

```python
# Grouping Example
# A. Series
import pandas as pd                      # Import pandas library as pd

# Salary series
salary = pd.Series([50000, 55000, 60000, 62000],
                   # Series of salaries
                   index=["Alice", "Bob", "Charlie", "David"])
                   # Index represents employee names

# Department for each person
department = pd.Series(["HR", "HR", "IT", "IT"],
                       # Series representing department
                       index=["Alice", "Bob", "Charlie", "David"])
                       # Index matches salary Series

# Grouping by department (no aggregation yet)
grouped_series = salary.groupby(department)
# Group salaries by department without aggregation

print("Grouped Series (just groups, no aggregation):\n", grouped_series, "\n")
# Print grouped object (no calculation yet)

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

```
Grouped Series (just groups, no aggregation):
 <pandas.core.groupby.generic.SeriesGroupBy object at 0x000001E304532720>

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

In [25]:

```python
# Aggregation Example
# A. Series
# Aggregation on grouped Series
aggregated_series = grouped_series.mean()
# Compute mean salary per department

print("Aggregated Series (mean salary per department):\n", aggregated_series, "\n")
# Display mean salary for each department

# Other aggregations
print("Sum per department:\n", grouped_series.sum(), "\n")
# Compute total salary per department

print("Max per department:\n", grouped_series.max(), "\n")
# Compute maximum salary per department
```

```
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 [27]:

```python
# Example: Grouping and Aggregating for Series / MultiLevel Series
import pandas as pd                       # Import pandas library as pd

# Single Series
salary = pd.Series([50000, 55000, 60000, 62000],
                   # Salary values
                   index=["Alice", "Bob", "Charlie", "David"])
                   # Employee names as index
department = pd.Series(["HR", "HR", "IT", "IT"],
                       # Department for each employee
                       index=["Alice", "Bob", "Charlie", "David"])
                       # Align with salary index

# Grouping salaries by department and calculating mean
grouped_series = salary.groupby(department).mean()
# Group salary by department and compute mean

print("Mean salary by department (Series):\n", grouped_series, "\n")
# Display mean salary per department

# MultiLevel Series
arrays = [
    ["HR", "HR", "IT", "IT"],              # Departments for MultiIndex
    ["Alice", "Bob", "Charlie", "David"]  # Employees for MultiIndex
]
index = pd.MultiIndex.from_arrays(arrays, names=("Dept", "Employee"))
# Create MultiIndex for Series

multi_s = pd.Series([50000, 55000, 60000, 62000], index=index)
# MultiLevel Series with salary values

# Group by first level (Dept)
grouped_multi = multi_s.groupby(level=0).mean()
# Compute mean salary by department

print("Mean salary by department (MultiLevel Series):\n", grouped_multi, "\n")
# Display grouped mean salary
```

```
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]:

```python
# Grouping (DataFrame)
import pandas as pd                       # Import pandas library as pd

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

# Grouping by Department
grouped_df = df.groupby("Department")
# Group DataFrame by Department column

print("Grouped DataFrame (just groups, no aggregation):\n", grouped_df, "\n")
# Print grouped object (no aggregation yet)

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

```
Grouped DataFrame (just groups, no aggregation):
 <pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001E3030BC440>

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

In [31]:

```
# Aggregating DataFrame
# Aggregation on grouped DataFrame
aggregated_df = grouped_df["Salary"].mean()
# Compute mean salary for each department

print("Aggregated DataFrame (mean salary per department):\n", aggregated_df, "\n")
# Display mean salary per department

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

```
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 [33]:

```
import pandas as pd                      # Import pandas library as pd

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

# ------------------------------------------
# Grouping by Department (no aggregation yet)
# ------------------------------------------
grouped = df.groupby("Department")
# Group DataFrame by Department column

print("Groups formed:\n", grouped.groups, "\n")
# Display indices of each group

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

# ------------------------------------------
# Aggregating after grouping
# ------------------------------------------
# 1. Single aggregation function (mean salary)
mean_salary = grouped["Salary"].mean()
# Compute mean Salary per department

print("Mean Salary per Department:\n", mean_salary, "\n")
# Display mean Salary per department

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

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

print("Departments with mean salary > 55000:\n", high_salary_dept, "\n")
# Display groups meeting the Salary condition
```

```
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 [37]:

```python
# Series
import pandas as pd                      # Import pandas library as pd

salary = pd.Series([50000, 55000, 60000, 62000],
                   # Create Series for salaries
                   index=["Alice","Bob","Charlie","David"])
                   # Index represents employee names

department = pd.Series(["HR","HR","IT","IT"],
                       # Series representing department
                       index=["Alice","Bob","Charlie","David"])
                       # Aligns with salary Series index

grouped_series = salary.groupby(department)
# Group salary Series by department
```

In [39]:

```python
# DataFrame
df = pd.DataFrame({
    "Department":["HR","HR","IT","IT"],
    # Department column
    "Employee":["Alice","Bob","Charlie","David"],
    # Employee names column
    "Salary":[50000,55000,60000,62000]
    # Salary column
})
# Create DataFrame with multiple columns

grouped_df = df.groupby("Department")
# Group DataFrame by Department column
```

In [41]:

```python
import pandas as pd                      # Import pandas library as pd

# ------------------------------------------
# 1. Series Example
# ------------------------------------------
salary_series = pd.Series([50000, 55000, 60000, 62000],
                          # Create Series for salaries
                          index=["Alice", "Bob", "Charlie", "David"])
                          # Index represents employee names

department_series = pd.Series(["HR", "HR", "IT", "IT"],
                              # Series for department of each employee
                              index=["Alice", "Bob", "Charlie", "David"])
                              # Aligns with salary Series

# Grouping (creates groups, no aggregation yet)
grouped_series = salary_series.groupby(department_series)
# Group salary Series by department

print("Grouped Series (no aggregation):")
# Display grouped Series information
for dept, group in grouped_series:
    # Iterate over each department group
    print(f"{dept}: {group.values}")
    # Print department name and salary values
print()

# Aggregating (mean per department)
aggregated_series = grouped_series.mean()
```

```
# Compute mean salary per department

print("Aggregated Series (mean salary per department):")
# Display aggregated mean salary
print(aggregated_series)
# Print mean salary Series
print("\n" + "="*50 + "\n")
# Separator for readability

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

# Grouping by Department (creates groups)
grouped_df = df.groupby("Department")
# Group DataFrame by Department column

print("Grouped DataFrame (no aggregation):")
# Display grouped DataFrame
for dept, group in grouped_df:
    # Iterate over each department group
    print(f"{dept} group:\n{group}\n")
    # Print group name and its rows

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

```
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 [43]:

```
# Merging DataFrames
import pandas as pd                      # Import pandas library as pd

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

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

# Inner merge (only common employees)
inner_merge = pd.merge(df1, df2, on="Employee", how="inner")
# Merge on Employee, keep common rows only
```

```
print("Inner Merge:\n", inner_merge, "\n")
# Display result of inner merge

# Outer merge (all employees)
outer_merge = pd.merge(df1, df2, on="Employee", how="outer")
# Merge on Employee, keep all rows

print("Outer Merge:\n", outer_merge, "\n")
# Display result of outer merge
```

```
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 [45]:

```
# Merging on different key columns
# Can merge using left_on/right_on

import pandas as pd                      # Import pandas library as pd

df1 = pd.DataFrame({
    "EmpID": [1, 2, 3],
    # Employee ID column
    "Name": ["Alice", "Bob", "Charlie"]
    # Employee names column
})
# Create first DataFrame

df2 = pd.DataFrame({
    "EmployeeID": [2, 3, 4],
    # Employee ID column (different name)
    "Salary": [60000, 55000, 58000]
    # Salary column
})
# Create second DataFrame

# Merge using different column names
merged_df = pd.merge(df1, df2, left_on="EmpID", right_on="EmployeeID", how="inner")
# Merge DataFrames on different key columns

print("Merge on different keys:\n", merged_df)
# Display merged DataFrame result
```

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

In [47]:

```
# Handling overlapping column names
# Use suffixes to avoid column conflicts

# If both DataFrames have same column names
# Pandas adds default _x and _y suffixes

# You can customize suffixes using parameter

import pandas as pd                      # Import pandas library as pd

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

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

# Merge with custom suffixes
merged_df = pd.merge(df1, df2, on="Employee", how="outer", suffixes=("_Old", "_New"))
# Merge DataFrames, add custom suffixes

print("Merge with custom suffixes:\n", merged_df)
# Display merged DataFrame result
```

```
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 [49]:

```
# Merging using indexes
# Merge DataFrames based on row index

import pandas as pd                      # Import pandas library as pd
```

```python
df1 = pd.DataFrame({"Salary": [50000, 60000]}, index=["Alice", "Bob"])
# Create first DataFrame with index

df2 = pd.DataFrame({"Department": ["HR", "IT"]}, index=["Alice", "Bob"])
# Create second DataFrame with index

# Merge using index
merged_index_df = pd.merge(df1, df2, left_index=True, right_index=True)
# Merge DataFrames using their indexes

print("Merge using indexes:\n", merged_index_df)
# Display merged DataFrame result
```

```
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 [51]:

```python
# Summary Tables using pivot_table
import pandas as pd                       # Import pandas library as pd

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

summary = df.pivot_table(values="Salary", index="Department", columns="Team", aggfunc="mean")
# Create pivot table with mean salaries

print(summary)
# Display pivot table result
```

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

In [53]:

```python
# Summary table using pivot_table
import pandas as pd                       # Import pandas library as pd

df = pd.DataFrame({
    "Department": ["HR", "HR", "IT", "IT", "HR", "IT"],
    # Department column
    "Team": ["A", "B", "A", "B", "A", "B"],
    # Team column
    "Salary": [50000, 55000, 60000, 62000, 58000, 61000]
    # Salary column
})
# Create DataFrame with department, team, salary

# Pivot table: average salary by Department and Team
summary_table = df.pivot_table(values="Salary", index="Department", columns="Team", aggfunc="mean")
# Compute mean salary by department and team

print("Summary Table (Average Salary):\n", summary_table)
# Display resulting pivot table
```

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

In [55]:

```python
# Pivot Table with Multiple Aggregation Functions
# Apply several functions at once

summary_table_multi = df.pivot_table(
    values="Salary",
    # Column to aggregate
    index="Department",
    # Rows: group by department
    columns="Team",
    # Columns: group by team
    aggfunc=["mean", "sum", "max"]
    # Apply multiple aggregation functions
)
# Create pivot table with multiple stats

print("Pivot Table with Multiple Aggregations:\n", summary_table_multi)
# Display pivot table with aggregated values
```

```
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 [57]:

```python
# Handling Missing Data in Pivot Table
# Fill missing combinations with specific value

summary_table_fill = df.pivot_table(
    values="Salary",
    # Column to aggregate
    index="Department",
    # Rows grouped by department
    columns="Team",
    # Columns grouped by team
    aggfunc="mean",
    # Use mean as aggregation function
    fill_value=0
    # Replace missing values with 0
)
# Create pivot table with filled missing values

print("Pivot Table with Missing Values Filled:\n", summary_table_fill)
# Display pivot table with no NaNs
```

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

In [59]:

```
#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 [ ]: