KARNAVATI UNIVERSITY

Data Science

By:
Dr. Shikha Deep

DATA OPERATIONS

Contents

- Reading
- Selecting
- Filtering
- Manipulating
- Sorting
- Grouping
- Rearranging
- Ranking

Data Operations

Data Operations refers to the **set of actions** we perform to **understand, analyze, and transform** the data once it is collected and cleaned.

1. Reading

Reading data means **loading a dataset from an external source** into your Python environment using the pandas library.

We mostly use:

pd.read_csv() – for CSV files

pd.read_excel() – for Excel files

pd.read_json() – for JSON files

pd.read_html() – to extract tables from websites

General Codes:

Code 1: Reading a CSV file

```
import pandas as pd

df = pd.read_csv("students.csv")
print(df.head())
```

NOTE:

Loads the CSV file named students.csv into a DataFrame and displays the first 5 rows using .head().

Code 2: Reading an Excel file

```
df_excel = pd.read_excel("sales_data.xlsx")
print(df_excel.info())
```

NOTE:

Reads an Excel file and shows structure: columns, datatypes, missing values.

Code 3: Reading a Dataset from URL

```
url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"
df_iris = pd.read_csv(url)
print(df_iris.head())
```

NOTE:

Fetches a public CSV dataset directly from a URL.

Code 4: Reading a JSON file

```
df_json = pd.read_json("data.json")
print(df_json.head())
```

NOTE:

Reads a JSON file. JSON is commonly used in APIs.

How to View Data

.head(),

.tail()

.info()

.shape

Task to do

Q. What is Custom delimiter ?

Q. Give the Code ?

2. Selecting

Code 1 : Selecting Columns using df[]

```
df["Student"]                    # Single column
df[["Student", "CGPA"]]          # Multiple columns
```

NOTE:

df['col'] gives a **Series**
df[['col1', 'col2']] gives a **DataFrame**

Code 2 : Selecting by Labels with .loc[]

Syntax: df.loc[row_label, column_label]

```
df.loc[2]                       # Entire row with index 2

df.loc[0:2, "Student"]          # Student names from row 0 to 2

df.loc[1:3, ["Student", "CGPA"]]
 # Rows 1 to 3 with two  columns (include 1,2,3)
```

Code 3 : Selecting by Index Numbers with .iloc[]

Syntax: df.iloc[row_index, column_index]

```
df.iloc[2]                  # Row at position 2
df.iloc[1:4, [0, 2]]        # Rows 1 to 3, columns 0 and 2
```

or

```
print(df.iloc[0:2, 0:2])   # Select first two rows and first two columns
df.iloc[:, 1]               # All rows, column 1
```

Task	Code	Returns
Select one column	df['Name']	Series
Select multiple columns	df[['Name', 'City']]	DataFrame
Select row by label	df.loc[1]	Series
Select row and specific columns	df.loc[0, ['Name', 'City']]	Series
Select rows/cols by label	df.loc[[0, 2], ['Name', 'Age']]	DataFrame
Select row by position	df.iloc[1]	Series
Select value by position	df.iloc[0, 1]	Scalar value
Select range of rows and columns	df.iloc[0:2, 0:2]	DataFrame

3. Data Filtering

Example :

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 28],
        'City': ['Delhi', 'Mumbai', 'Chennai', 'Delhi']}
df = pd.DataFrame(data)
print(df)
```

OUTPUT :

	Name	Age	City
0	Alice	25	Delhi
1	Bob	30	Mumbai
2	Charlie	35	Chennai
3	David	28	Delhi

1. Filter rows based on a single condition

CODE:

```
# Filter where Age > 30
print(df[df['Age'] > 30])
```

OUTPUT:

```
      Name     Age    City
2    Charlie   35      Chennai
```

2. Filter rows based on multiple conditions

Use & (and), | (or), ~ (not) with parentheses ()

CODE:

```
# Age > 25 AND City is Delhi
print(df[(df['Age'] > 25) & (df['City'] == 'Delhi')])
```

OUTPUT:

```
  Name  Age   City
3  David   28  Delhi
```

Why use .isin()?

- In Pandas, **.isin()** is used to filter rows where a column's value is **in a list or set of multiple values**.

Eg: `df[df['City'].isin(['Delhi', 'Mumbai', 'Chennai'])]`

Without .isin():

- We need to write **multiple OR conditions**, which gets messy:

Eg: `df[(df['City'] == 'Delhi') | (df['City'] == 'Mumbai') | (df['City'] == 'Chennai')]`

Key Benefits of .isin():

Advantage	Explanation
◆ Easy syntax	More readable than writing multiple == and `
◆ Works with lists/sets	You can filter using many values at once
◆ Works with NOT using ~	Like ~df['City'].isin(['Delhi']) to exclude Delhi

3. Filter using string methods

CODE:

```
# Filter where Name starts with 'A'
print(df[df['Name'].str.startswith('A')])
```

OUTPUT:

```
    Name  Age   City
0  Alice   25  Delhi
```

Quiz

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
        'Age': [25, 30, 35, 28, 22],
        'City': ['Delhi', 'Mumbai', 'Chennai', 'Delhi',
'Kolkata']}
df = pd.DataFrame(data)
```

Q1. Filter all rows where the Age is greater than 28

Write the filtering line of code.

Q2. Filter all rows where the City is either Delhi or Kolkata

Use .isin() method.

Q3. Filter all rows where Name starts with the letter ‘C’

Q4. Filter all rows where Age is NOT 30 or 35

Q5. Filter all rows where City is not Delhi, using the ~ operator.

4. Data Manipulation

- **Manipulating data** means **modifying the contents** of a DataFrame — like changing values, adding/removing columns, renaming, replacing, or applying calculations.

This includes:

- Adding new columns
- Updating existing values
- Replacing missing or incorrect data
- Changing data types
- Applying operations to transform data

Common Manipulation Operations:

S.No.	Operation	Example
1	Add new column	Add Total or Grade column
2	Modify values	Update names, marks, etc.
3	Apply formula	Convert marks to percentage
4	Replace values	Replace 'NaN' or wrong data
5	Rename columns	Make headers cleaner or standardized

Example 1: Create a Sample DataFrame

CODE:

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Marks': [85, 90, 78],
        'City': ['Delhi', 'Mumbai', 'Chennai']}

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

OUTPUT:

	Name	Marks	City
0	Alice	85	Delhi
1	Bob	90	Mumbai
2	Charlie	78	Chennai

Code1: Add a new column (Let's add a column Grade based on Marks)

CODE:

```
df['Grade'] = ['B', 'A', 'C']
print(df)
```

OUTPUT:

	Name	Marks	City	Grade
0	Alice	85	Delhi	B
1	Bob	90	Mumbai	A
2	Charlie	78	Chennai	C

Code 2: Update values in a column (Let's say Bob improved his marks to 95)

CODE:

```
df.loc[df['Name'] == 'Bob', 'Marks'] = 95
print(df)
```

OUTPUT:

	Name	Marks	City	Grade
0	Alice	85	Delhi	B
1	Bob	95	Mumbai	A
2	Charlie	78	Chennai	C

Code 3: Apply a function to transform data (Let's add 5 bonus marks to each student)

CODE:

```
df['Final_Marks'] = df['Marks'].apply(lambda x: x + 5)
```

OUTPUT:

	Name	Marks	City	Grade	Final_Marks
0	Alice	85	Delhi	B	90
1	Bob	95	Mumbai	A	100
2	Charlie	78	Chennai	C	83

Or

CODE:

```
def add_bonus(x):
    return x + 5

df['Final_Marks'] = df['Marks'].apply(add_bonus)
```

NOTE:

(1) Lambda is a Short, inline function.

(2) No need to define a separate named function.

Code 4: Replace values (Let's change City = 'Chennai' to 'Madras')

CODE:

```
df['City'] = df['City'].replace('Chennai', 'Madras')
print(df)
```

OUTPUT:

```
      Name  Marks    City     Grade  Final_Marks
0    Alice     85     Delhi       B           90
1      Bob     95    Mumbai       A          100
2  Charlie     78    Madras       C           83
```

Code 5: Change Data Type (Change Marks from int to float)

CODE:

```
df['Marks'] = df['Marks'].astype(float)
print(df.dtypes)
```

OUTPUT:

```
Name          object
Marks        float64
City          object
Grade         object
Final_Marks    int64
dtype: object
```

Summary

Task	Code
Add new column	df['NewCol'] = [...]
Update cell value	df.loc[row, 'Col'] = new_value
Apply function to column	df['Col'].apply(func)
Replace values	df['Col'].replace(old, new)
Change data type	df['Col'].astype(new_type)

Special Cases

Ex1: Change a specific value in between (e.g., add 5 to only the second row)

- Modify a specific cell (or row/column) using .loc[] or .iloc[].

CODE: Add 5 to the Marks of the second student (index 1):

```
df.loc[1, 'Marks'] = df.loc[1, 'Marks'] + 5
```

OR

```
df.iloc[1, 1] = df.iloc[1, 1] + 5
```

(It's the 2nd row and 2nd column — index 1 and 1):

Ex 2: Check marks in float

```
print(df)
```

OUTPUT:

```
      Name  Marks    City     Grade
0    Alice   85.0     Delhi       B
1      Bob   100.0   Mumbai     A
2  Charlie   88.0    Madras     C
```

5. Data Sorting :

Sorting in Pandas means arranging your DataFrame based on the values of one or more columns.

It includes:

- Sort by column values (like Marks, Name, etc.)
- Sort by index
- Sort in ascending or descending order
- Sort multiple columns (e.g., by Subject, then Marks)

CODE:

```
df.sort_values(by='column_name', ascending=True/False)
```

Ex: Create a Data Set

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
        'Marks': [85, 95, 78, 88, 95],
        'Subject': ['Math', 'Physics', 'Chemistry', 'Math',
'Physics']}
df = pd.DataFrame(data)
print(df)
```

	Name	Marks	Subject
0	Alice	85	Math
1	Bob	95	Physics
2	Charlie	78	Chemistry
3	David	88	Math
4	Eva	95	Physics

Code1. Sort by a single column (ascending)

Sort students by Marks in ascending order:

CODE:

```
df_sorted = df.sort_values(by='Marks')
print(df_sorted)
```

OUTPUT:

	Name	Marks	Subject
2	Charlie	78	Chemistry
0	Alice	85	Math
3	David	88	Math
1	Bob	95	Physics
4	Eva	95	Physics

Code 2. Sort by a single column (descending)

```
df.sort_values(by='Marks', ascending=False)
```

Code 2. Sort by multiple columns

Sort first by Subject, then by Marks in descending order:

CODE:

```
df.sort_values(by=['Subject', 'Marks'], ascending=[True, False])
```

OUTPUT:

	Name	Marks	Subject
2	Charlie	78	Chemistry
0	Alice	85	Math
3	David	88	Math
1	Bob	95	Physics
4	Eva	95	Physics

Code 3. Sort by index

CODE:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Marks': [85, 95, 78]
}
df = pd.DataFrame(data, index=[2, 0, 1])
print("Original DataFrame:")
print(df)
```

OUTPUT:

	Name	Marks
2	Alice	85
0	Bob	95
1	Charlie	78

Now shuffle the data

CODE:

```
df_sorted = df.sort_index()
print("Sorted by index:")
print(df_sorted)
```

OUTPUT:

```
      Name  Marks
0      Bob     95
1  Charlie     78
2    Alice     85
```

General Syntax

```
df.sort_index(ascending=True)   # Ascending (default)
df.sort_index(ascending=False)  # Descending
```

6. Data Grouping :

Grouping means **splitting** your data into groups based on some category and then applying a **function** (like sum, mean, count) to each group.

Syntax of groupby ():

df.groupby('column_name')

With function:

df.groupby('column_name').sum()

df.groupby('column_name').mean()

df.groupby('column_name').count()

Example :

Name	Subject	Marks
Alice	Math	85
Bob	Math	90
Alice	Science	88
Bob	Science	84
Carol	Math	92

CODE:

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Alice', 'Bob', 'Carol'],
    'Subject': ['Math', 'Math', 'Science', 'Science', 'Math'],
    'Marks': [85, 90, 88, 84, 92]
}

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

OUTPUT:

	Name	Subject	Marks
0	Alice	Math	85
1	Bob	Math	90
2	Alice	Science	88
3	Bob	Science	84
4	Carol	Math	92

Code 1. Total Marks by Student

```
df.groupby('Name')['Marks'].sum()
```

OUTPUT:

```
Name
Alice    173
Bob      174
Carol     92
Name: Marks, dtype: int64
```

Code 2. Average Marks per Subject

```
df.groupby('Subject')['Marks'].mean()
```

OUTPUT:

```
Subject
Math       89.0
Science    86.0
Name: Marks, dtype: float64
```

Multiple Grouping Columns

CODE 1 :

```
df.groupby(['Name', 'Subject'])['Marks'].mean()
```

OUTPUT:

```
Name    Subject
Alice    Math       85
         Science    88
Bob      Math       90
         Science    84
Carol    Math       92
Name: Marks, dtype: int64
```

Code 2 : For Duplicate values

```
data = {
    'Name': ['Alice', 'Alice', 'Bob', 'Bob', 'Carol'],
    'Subject': ['Math', 'Math', 'Math', 'Science', 'Math'],
    'Marks': [85, 95, 90, 84, 92]
}
df = pd.DataFrame(data)
df.groupby(['Name', 'Subject'])['Marks'].mean()
```

Or

OUTPUT:

```
Name    Subject
Alice   Math       90.0   # (85+95)/2
Bob     Math       90.0
        Science    84.0
Carol   Math       92.0
Name: Marks, dtype: float64
```

Code 3. Use Unstack : Convert row index → columns

```
grouped = df.groupby(['Name', 'Subject'])['Marks'].mean()
print(grouped)
print(grouped.unstack())
```

OUTPUT:

```
Subject   Math  Science
Name
Alice     90.0      NaN
Bob       90.0     84.0
Carol     92.0      NaN
```

Code 4. Use of reset index

Again back to original data

```
print(grouped.reset_index())
```

7. Rearranging

Rearranging in Pandas — this includes changing the layout, column orders, or reshaping your DataFrame.

It means **changing the order** of:

- Columns
- Rows
- Or even reshaping the structure of your DataFrame.

Code1. Rearranging Columns

(Rearrange columns by simply changing their order)

```
import pandas as pd
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22],
    'City': ['Delhi', 'Mumbai', 'Chennai']
})

# Reorder columns: move 'City' first
df_rearranged = df[['City', 'Name', 'Age']]
print(df_rearranged)
```

OUTPUT:

	City	Name	Age
0	Delhi	Alice	25
1	Mumbai	Bob	30
2	Chennai	Charlie	22

Code2. Rearranging Rows by Index

Reverse the order of rows using slicing like df[::-1].

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22],
    'City': ['Delhi', 'Mumbai', 'Chennai']
})

# Reverse row order
df_reversed = df[::-1]
print(df_reversed)
```

OUTPUT:

```
    Name  Age    City
2  Charlie   22  Chennai
1      Bob   30   Mumbai
0    Alice   25    Delhi
```

Code 3. Rearranging Rows Using .reindex()

reorder rows manually

```
# Rearranging rows using custom index
df_custom = df.reindex([2, 0, 1])
print(df_custom)
```

OUTPUT:

```
      Name  Age   City
2  Charlie  22  Chennai
0    Alice  25    Delhi
1      Bob  30   Mumbai
```

Code 4. Transposing the DataFrame with .T

```
# Transpose the DataFrame
print(df.T)
```

OUTPUT:

```
         0      1        2
Name   Alice    Bob    Charlie
Age       25     30         22
City   Delhi  Mumbai  Chennai
```

8. Ranking

- In Pandas, .rank() is used to assign ranks to data values based on their size — smallest gets rank 1 by default.
- It's often used in grading, competition scores, sales analysis, etc.

Syntax:

DataFrame['column'].rank(method='average’,ascending=True)

Parameters

Parameter	Description
method	How to assign ranks to equal values: 'average', 'min', 'max', 'dense', 'first'
ascending	True for smallest rank = 1, False for highest = 1
axis	Use 0 for columns, 1 for rows (default is column-wise ranking)

Code 1. Ranking Exam Scores

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['A', 'B', 'C', 'D', 'E'],
    'Score': [85, 92, 88, 92, 70]
})

df['Rank'] = df['Score'].rank(ascending=False)
print(df)
```

OUTPUT:

	Name	Score	Rank
0	A	85	4.0
1	B	92	1.5
2	C	88	3.0
3	D	92	1.5
4	E	70	5.0

Code 2. Average, min, max, dense

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['A', 'B', 'C', 'D', 'E'],
    'Score': [90, 80, 90, 70, 60]
})
```

```
df['Rank_average'] = df['Score'].rank(method='average’,
ascending=False)

df['Rank_min'] = df['Score'].rank(method='min',
ascending=False)

df['Rank_max'] = df['Score'].rank(method='max',
ascending=False)

df['Rank_dense'] = df['Score'].rank(method='dense',
ascending=False)
print(df)
```

OUTPUT:

	Name	Score	Rank_average	Rank_min	Rank_max	Rank_dense
0	A	90	1.5	1.0	2.0	1.0
1	B	80	3.0	3.0	3.0	2.0
2	C	90	1.5	1.0	2.0	1.0
3	D	70	4.0	4.0	4.0	3.0
4	E	60	5.0	5.0	5.0	4.0

Code 3. Axis=1 (Row-wise Ranking):

```
df2 = pd.DataFrame({
    'Math': [90, 40, 70],
    'Science': [80, 60, 75],
    'English': [85, 55, 65]
}, index=['Student1', 'Student2', 'Student3’])
```

Row-wise:

```
rank_rowwise = df2.rank(axis=1, ascending=False)
print(rank_rowwise)
```

Column-wise:

```
rank_colwise = df2.rank(axis=0, ascending=False)
print(rank_colwise)
```


Thanks!