UNIT-IV

1) **Introduction to pandas Data Structure**

2) **Essential functionality –**

   2.1) Reindexing, Indexing ,

   2.2) selection and filtering,

   2.3) reshaping

   2.4) summarizing and computing descriptive statistics,

   2.5) handling missing data,

   2.6) filter and query methods,

   2.7) grouping,

3) **reading and writing data in text format –**

   3.1) read_csv,

   3.2) read_table.

4) **NumPy Basics-**

   4.1) creating Arrays,

5) **universal functions –**

   5.1) Basic unary and binary functions ,

6) **File Input and Output with arrays –**

   6.1) Saving and loading text files,

7) **Linear Algebra –**

   7.1) commonly used linalg functions

1. Introduction to pandas Data Structure

- ✓ Pandas and NumPy are essential Python libraries for data analysis and scientific computing.
- ✓ Pandas generally provide two data structures for manipulating data. They are:
  1) Series
  2) DataFrame

SERIES	DATA FRAME
A Series is a one-dimensional labeled array that can hold any data type. Think of it like a single column in a spreadsheet with an index.	DataFrames are two-dimensional labeled data structures with columns potentially of different types. They resemble tables with rows and columns and are the most common pandas object
For example: **pd.series(data, index)** creates a Series with data values and custom indexes	For example: **pd.DataFrame(data, columns)** constructs a DataFrame with specified column names.
Pandas Series can be created from lists, dictionaries, scalar values, etc.	Pandas Series can be created from lists, dictionaries, scalar values, etc.
import pandas as pd s = pd.series([10, 20, 30], index=['a', 'b', 'c'])	import pandas as pd df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30]})
Index : a b c Data: 10 20 30	Name age Alice 25 Bob 30

To Read a CSV file

```
import pandas as pd
df = pd.read_csv("emp.csv")
print(df)
```

```
    eno    ename            desg    sal  deptno
0  e0001    smith   sales Person  20000      10
1  e0002    Jones        Manager  45000      20
2  e0003     King        Analyst  30000      10
3  e0004  krishna  Data Engineer  40000      10
4  e0005     khan        Analyst  35000      20
```

2. Essential functionality –

2.1 Indexing and Re-Indexing

✓ **Indexing in Pandas** refers to selecting specific rows and columns from a DataFrame.

✓ **also known as Subset Selection.**

✓ The three main types of indexing in Pandas are:

2.1.1) DastaFrame[]: Known as the indexing operator, used for basic selection.

2.1.2) DataFrame.loc[]: Label-based indexing for selecting data by row/column labels.

2.1.3) DataFrame.iloc[]: Position-based indexing for selecting data by row/column integer positions.

2.1.1) DataFrame[]:

✓ Known as the indexing operator, used for basic selection.
✓ The most straightforward way to index data in Pandas is by using the [] operator.
✓ This method can be used to select individual columns or multiple columns

a) To retrieve All Rows and Specific Column/Columns using [] operator

```
[110] print(df['eno'])
```

```
0    e0001
1    e0002
2    e0003
3    e0004
4    e0005
Name: eno, dtype: object
```

```
print(df[["eno","ename"]])
```

```
    eno    ename
0  e0001    smith
1  e0002    Jones
2  e0003     King
3  e0004  krishna
4  e0005     khan
```

b) To retrieve Specific Row/Rows and All Columns using [] operator

```
# specific row and all columns
print(df[df['ename']=='smith'])
```

```
   eno  ename  age          desg    sal  deptno DESIGNATION  Dependents
0  e0001  smith   34  sales Person  20000      10         NaN           2
```

```
[118] # specific rows and all columns
      print(df[df['sal']>=40000])
```

```
   eno    ename  age           desg    sal  deptno DESIGNATION  Dependents
1  e0002    Jones   25        Manager  45000      20         NaN           1
3  e0004  krishna   35  Data Engineer  40000      10         NaN           2
```

c) To retrieve Specific Rows and Specific Columns using [] operator

```
# specific rows and specific columns
print(df.loc[df['sal'] >= 40000, ['eno', 'sal']])
```

```
   eno    sal
1  e0002  45000
3  e0004  40000
```

DataFrame.loc and DataFrame.iloc

```
[1] import pandas as pd
    df = pd.read_csv("emp.csv")
    print(df)
```

```
    eno    ename           desg    sal  deptno
0  e0001    smith   sales Person  20000      10
1  e0002    Jones        Manager  45000      20
2  e0003     King        Analyst  30000      10
3  e0004  krishna  Data Engineer  40000      10
4  e0005     khan        Analyst  35000      20
```

```
df.set_index("ename",inplace=True)
print(df)
```

```
           eno           desg    sal  deptno
ename
smith    e0001   sales Person  20000      10
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

<table>
<tr><th>loc</th><th>Iloc</th></tr>
<tr><td>Label-based indexing for selecting data by row/column</td><td>Position-based indexing for selecting data by row/column</td></tr>
<tr><th colspan="2">To retrieve Specific Row</th></tr>
<tr><td>

```
print(df.loc['krishna'])
```

```
eno              e0004
desg      Data Engineer
sal              40000
deptno              10
Name: krishna, dtype: object
```

</td><td>

```
print(df.iloc[3])
```

```
eno              e0004
desg      Data Engineer
sal              40000
deptno              10
Name: krishna, dtype: object
```

</td></tr>
<tr><th colspan="2">To retrieve Specific Rows</th></tr>
<tr><td>

```
print(df.loc[['smith','Jones']])
```

```
          eno          desg    sal  deptno
ename
smith   e0001  sales Person  20000      10
Jones   e0002       Manager  45000      20
```

</td><td>

```
print(df.iloc[0:2])
```

```
          eno          desg    sal  dept
ename
smith   e0001  sales Person  20000
Jones   e0002       Manager  45000
```

</td></tr>
<tr><th colspan="2">To retrieve Specific Column</th></tr>
<tr><td>

```
print(df.loc[:,['desg']])
```

```
                 desg
ename
smith      sales Person
Jones           Manager
King            Analyst
krishna  Data Engineer
khan            Analyst
```

</td><td>

```
print(df.iloc[:,[1]])
```

```
                 desg
ename
smith      sales Person
Jones           Manager
King            Analyst
krishna  Data Engineer
khan            Analyst
```

</td></tr>
</table>

To retrieve Specific Columns

```
[11] print(df.loc[:,['eno','sal']])
```

```
            eno    sal
ename
smith     e0001  20000
Jones     e0002  45000
King      e0003  30000
krishna   e0004  40000
khan      e0005  35000
```

```
[12] print(df.iloc[:,[0,2]])
```

```
            eno    sal
ename
smith     e0001  20000
Jones     e0002  45000
King      e0003  30000
krishna   e0004  40000
khan      e0005  35000
```

To retrieve Specific Rows and Specific Columns

```
print(df.loc[['smith','Jones'],['eno','sal']])
```

```
        eno    sal
ename
smith  e0001  20000
Jones  e0002  45000
```

```
print(df.iloc[[0,1],[0,2]])
```

```
        eno    sal
ename
smith  e0001  20000
Jones  e0002  45000
```

To retrieve All Rows and All Columns

```
print(df.loc[:,:])
```

```
           eno           desg    sal  deptno
ename
smith    e0001   sales Person  20000      10
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

```
print(df.iloc[:,:])
```

```
           eno           desg    sal  deptno
ename
smith    e0001   sales Person  20000      10
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

2.2) Selection and filtering

Filtering and selection are fundamental operations when working with data in Pandas. They allow you to extract specific subsets of data that meet certain conditions

Single Condition Filtering with Comparison Operators

```
# To get employees whose salary is greater than or equal to 30000
sal_ft =df['sal'] >= 30000
print (df[sal_ft])
```

```
           eno           desg    sal  deptno
ename
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

Combining Multiple Conditions with Logical Operators

```
# To get employees whose salary is greater than or equal to 30000 and deptno = 10
sal_ft = (df['sal'] >= 30000) & (df['deptno']==10)
print (df[sal_ft])
```

```
           eno           desg    sal  deptno
ename
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
```

Using .query() Method

```
print(df.query('sal >= 3000'))
```

ename	eno	desg	sal	deptno
smith	e0001	sales Person	20000	10
Jones	e0002	Manager	45000	20
King	e0003	Analyst	30000	10
krishna	e0004	Data Engineer	40000	10
khan	e0005	Analyst	35000	20

```
print(df.query('sal >= 3000 & deptno==10'))
```

ename	eno	desg	sal	deptno
smith	e0001	sales Person	20000	10
King	e0003	Analyst	30000	10
krishna	e0004	Data Engineer	40000	10

Filtering with String Conditions

```
res = df['desg']=='Analyst'
print(df[res])
```

ename	eno	desg	sal	deptno
King	e0003	Analyst	30000	10
khan	e0005	Analyst	35000	20

Filtering with isin() Method

```
desg_ft = df['desg'].isin(['Analyst', 'Data Engineer'])
print(df[desg_ft])
```

ename	eno	desg	sal	deptno
King	e0003	Analyst	30000	10
krishna	e0004	Data Engineer	40000	10
khan	e0005	Analyst	35000	20

2.3) reshaping

- ✓ Reshaping in Pandas is a fundamental set of operations that allows to change the structure (the number of rows and columns) of the DataFrame or Series.
- ✓ This is often necessary to prepare data for analysis, visualization, or integration with other datasets.

Reshape DataFrame in Pandas

- ✓ Below are the two methods that are used to reshape the layout of data in Pandas:
  1) Using Pandas stack() method
  2) Using unstack() method

1) Reshape the Layout of Tables in Pandas Using stack() method

- ✓ Pivots a DataFrame from a "wide" format to a "long" format by stacking columns into rows, creating a hierarchical index (MultiIndex) on the rows.
- ✓ Use **stack()** to go from wide to long when you want column labels to become part of a hierarchical row index.

Original Data Frame

```
print(df)
```

```
           eno           desg    sal  deptno
ename
smith    e0001   sales Person  20000      10
Jones    e0002        Manager  45000      20
King     e0003        Analyst  30000      10
krishna  e0004  Data Engineer  40000      10
khan     e0005        Analyst  35000      20
```

Stacked Dataframe

```
stacked_df = df.stack()
print(stacked_df)
```

```
0  ename             smith
   eno              e0001
   desg       sales Person
   sal              20000
   deptno              10
1  ename             Jones
   eno              e0002
   desg            Manager
   sal              45000
   deptno              20
2  ename              King
   eno              e0003
   desg            Analyst
   sal              30000
   deptno              10
3  ename            krishna
   eno              e0004
   desg      Data Engineer
   sal              40000
   deptno              10
4  ename              khan
   eno              e0005
   desg            Analyst
   sal              35000
   deptno              20
```

2) Reshape a Pandas DataFrame Using unstack() method

- ✓ The inverse of stack(). It pivots a DataFrame or Series with a MultiIndex (typically created by stack()) from a "long" format back to a "wide" format, with the inner level of the row index becoming the new column labels.
- ✓ Use **unstack()** to go from long (with a MultiIndex) back to wide, making an inner level of the row index the new columns.

```
unstacked_df = stacked_df.unstack()
print(unstacked_df)
```

```
     ename    eno           desg    sal deptno
0    smith  e0001   sales Person  20000     10
1    Jones  e0002        Manager  45000     20
2     King  e0003        Analyst  30000     10
3  krishna  e0004  Data Engineer  40000     10
4     khan  e0005        Analyst  35000     20
```

2.4) summarizing and computing descriptive statistics,

- ✓ Python Pandas provides a rich set of tools and methods for **summarizing and aggregating the data** within DataFrames and Series.
- ✓ These operations allows to get ***concise overviews, calculate descriptive statistics, and group data for more insightful summaries.***
- ✓ ***Two methods of computing summary statistics using Pandas are***
  1) Using describe() for Descriptive Statistics
  2) Using Individual Aggregation Functions:

1) Using describe() for Descriptive Statistics

.describe(): This is the most common and powerful function for getting a quick statistical summary of your numerical columns. It provides:

count: Number of non-missing values.
mean: Average.
std: Standard deviation.
min: Minimum value.
25% (Q1): First quartile.
50% (median or Q2): Second quartile.
75% (Q3): Third quartile.
max: Maximum value.

```
print(df['deptno'].describe())
```

```
count     5.000000
mean     14.000000
std       5.477226
min      10.000000
25%      10.000000
50%      10.000000
75%      20.000000
max      20.000000
Name: deptno, dtype: float64
```

2) Using Individual Aggregation Functions:

`.mean()`: Calculate the mean.
`.median()`: Calculate the median.
`.std()`: Calculate the standard deviation.
`.min()`: Get the minimum value.
`.max()`: Get the maximum value.
`.count()`: Count non-missing values.

```
     ename    eno           desg    sal deptno
0    smith  e0001   sales Person  20000     10
1    Jones  e0002        Manager  45000     20
2     King  e0003        Analyst  30000     10
3  krishna  e0004  Data Engineer  40000     10
4     khan  e0005        Analyst  35000     20
```

```
print("sum of salaries",df['sal'].sum())
print("Average of Salaries",df['sal'].mean())
print("Max of Salaries",df['sal'].max())
print("Min of Salaries",df['sal'].min())
print("Number of Employees",df['eno'].count())
```

```
sum of salaries 170000
Average of Salaries 34000.0
Max of Salaries 45000
Min of Salaries 20000
Number of Employees 5
```

2.5) Handling missing data,

- ✓ In Pandas, missing values, often represented as NaN (Not a Number), can cause problems during data processing and analysis.
- ✓ These gaps in data can lead to incorrect analysis and misleading conclusions.
- ✓ Pandas provides several functions and methods to
  1) ***identify, and***
  2) ***handle***
  3)

1) Identifying Missing Values:

- ✓ **isna() or isnull():** These functions detect missing values and return a DataFrame or Series of boolean values. **True** indicates a missing value (NaN), and **False** indicates a non-missing value.

To find Missing Values in Each Column using isnull() and isna()

```
df1 = pd.read_csv('stu.csv')
print(df1)
```

```
   sno           sname branch  mark1  mark2  mark3
0  100            Ivan    MCA   79.0   66.0   77.0
1  101           Korth    NaN   65.0   45.0   56.0
2  102   James Gosling    MBA    NaN    NaN   87.0
3  103     Chris Bates    MCA   66.0   54.0    NaN
4  104    Nageswar Rao    NaN   89.0    NaN    NaN
5  105  Bala Guruswamy    MCA    NaN   55.0   66.0
```

```
[35] print(df1.isnull())
```

```
     sno  sname  branch  mark1  mark2  mark3
0  False  False   False  False  False  False
1  False  False    True  False  False  False
2  False  False   False   True   True  False
3  False  False   False  False  False   True
4  False  False    True  False   True   True
5  False  False   False   True  False  False
```

```
print(df1.isna())
```

```
     sno  sname  branch  mark1  mark2  mark3
0  False  False   False  False  False  False
1  False  False    True  False  False  False
2  False  False   False   True   True  False
3  False  False   False  False  False   True
4  False  False    True  False   True   True
5  False  False   False   True  False  False
```

To find Number of Missing Values in Each Column

```
#to get number of missing values in each column
print(df1.isnull().sum())
```

```
sno       0
sname     0
branch    2
mark1     2
mark2     2
mark3     2
dtype: int64
```

2) Handling Missing Values:

2.1) Dropping Missing Values:

- ***dropna(): This function removes rows or columns that contain missing values.***
  - *axis=0 (default): Drops rows containing at least one missing value.*
  - *axis=1: Drops columns containing at least one missing value.*
  - *how='any' (default): Drop row/column if any missing values are present.*
  - *how='all': Drop row/column if all values are missing.*

```
[40] print(df1)
```

```
   sno           sname branch  mark1  mark2  mark3
0  100            Ivan    MCA   79.0   66.0   77.0
1  101           Korth    NaN   65.0   45.0   56.0
2  102   James Gosling    MBA    NaN    NaN   87.0
3  103     Chris Bates    MCA   66.0   54.0    NaN
4  104    Nageswar Rao    NaN   89.0    NaN    NaN
5  105  Bala Guruswamy    MCA    NaN   55.0   66.0
```

```
[46] #Dropping Rows with At Least One Null Value
     print(df1.dropna())
```

```
   sno sname branch  mark1  mark2  mark3
0  100  Ivan    MCA   79.0   66.0   77.0
```

```
[42] # drop rows with all NaN values
     print(df1.dropna(axis=0))
```

```
   sno sname branch  mark1  mark2  mark3
0  100  Ivan    MCA   79.0   66.0   77.0
```

```
[45] #drop columns with all NaN values
     print(df1.dropna(axis=1))
```

```
   sno           sname
0  100            Ivan
1  101           Korth
2  102   James Gosling
3  103     Chris Bates
4  104    Nageswar Rao
5  105  Bala Guruswamy
```

```
#Dropping Rows with All Null Values
print(df1.dropna(how='all'))
```

```
   sno           sname branch  mark1  mark2  mark3
0  100            Ivan    MCA   79.0   66.0   77.0
1  101           Korth    NaN   65.0   45.0   56.0
2  102   James Gosling    MBA    NaN    NaN   87.0
3  103     Chris Bates    MCA   66.0   54.0    NaN
4  104    Nageswar Rao    NaN   89.0    NaN    NaN
5  105  Bala Guruswamy    MCA    NaN   55.0   66.0
```

```
[48] #Dropping Rows/Columns with ANY Null Values
     print(df1.dropna(how='any'))
```

2.2) Filling Missing Values

2.2.1) using **fillna()**
2.2.2) using **replace()**

2.2.1) Using fillna()

2.2.1.1) fillna(value)
2.2.1.2) fillna(method="pad|ffill")
2.2.1.3) fillna(df['col'].mean()|median()|mode())

2.2.1.1) fillna(value): *Replaces all missing values with a specified scalar value.*

Original data

```
# Importing pandas and numpy
import pandas as pd
import numpy as np

# Sample DataFrame with missing values
d = {'First Score': [100, 90, np.nan, 95],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score': [np.nan, 40, 80, 98]}
df = pd.DataFrame(d)
print(df)
```

```
   First Score  Second Score  Third Score
0        100.0          30.0          NaN
1         90.0          45.0         40.0
2          NaN          56.0         80.0
3         95.0           NaN         98.0
```

Fillna(value)

```
print(df.fillna(10000))
```

```
   First Score  Second Score  Third Score
0        100.0          30.0      10000.0
1         90.0          45.0         40.0
2      10000.0          56.0         80.0
3         95.0       10000.0         98.0
```

2.2.1.2) fillna(method = 'pad|ffill'

- ✓ **fillna(method='ffill' or 'pad'):** *Forward fill - propagates the last valid observation forward to the next missing value.*
- ✓ **fillna(method='bfill' or 'backfill'):** *Backward fill - propagates the next valid observation backward to the previous missing value.*
- ✓ **ffill will not fill leading NaN values (as there's no previous valid value).**
- ✓ **bfill or pad will not fill trailing NaN values (as there's no subsequent valid value).**

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	NaN	56.0	80.0
3	95.0	NaN	98.0

Example: Fill with Previous Value (Forward Fill)

```
df.fillna(method='pad')  # Forward fill
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	NaN
1	90.0	45.0	40.0
2	90.0	56.0	80.0
3	95.0	56.0	98.0

Example: Fill with Next Value (Backward Fill)

```
df.fillna(method='bfill')  # Backward fill
```

Output

	First Score	Second Score	Third Score
0	100.0	30.0	40.0
1	90.0	45.0	40.0
2	95.0	56.0	80.0
3	95.0	NaN	98.0

2.2.1.3) ***Imputation using statistical measures: You can fill missing values with the mean, median, or mode of the respective column.***

```
print(df)
```

```
   First Score  Second Score  Third Score
0        100.0          30.0          NaN
1         90.0          45.0         40.0
2          NaN          56.0         80.0
3         95.0           NaN         98.0
```

```
[59] print(df['First Score'].fillna(df['First Score'].mean()))
```

```
0    100.0
1     90.0
2     95.0
3     95.0
Name: First Score, dtype: float64
```

```
[61] print(df['Second Score'].fillna(df['Second Score'].median()))
```

```
0    30.0
1    45.0
2    56.0
3    45.0
Name: Second Score, dtype: float64
```

```
print(df['Third Score'].fillna(df['Third Score'].mode()))
```

```
0    40.0
1    40.0
2    80.0
3    98.0
```

**

2.6) filter and query methods,

- ✓ ***Use df.query() when you want a more readable, string-based way to filter rows based on column values.***
- ✓ ***Use df.filter() when you need to select columns or rows based on the names or patterns of their labels.***

```
df.filter(items=None, like=None, regex=None, axis=None)
```

Parameters:

- **items:** A list-like of labels to keep. These should be in the `axis` specified.
- **like:** Keep labels where the string `like` is found in the label.
- **regex:** Keep labels matching the regular expression `regex`.
- **axis:** The axis to filter on.
  - `0` or `'index'` (default): Filters rows based on index labels.
  - `1` or `'columns'`: Filters columns based on column labels.

Examples:

```
#filter method
res = df2.filter(items=['eno','sal'])
print(res)
```

```
    eno    sal
0  e0001  20000
1  e0002  45000
2  e0003  30000
3  e0004  40000
4  e0005  35000
```

```
df2.set_index('ename',inplace=True)
res=df2.filter(regex='g$', axis=0)
print(res)
```

```
          eno     desg    sal  deptno
ename
King    e0003  Analyst  30000      10
```

```
res=df2.filter(like='n', axis=0)
print(res)
```

```
             eno           desg    sal  deptno
ename
Jones     e0002        Manager  45000      20
King      e0003        Analyst  30000      10
krishna   e0004  Data Engineer  40000      10
khan      e0005        Analyst  35000      20
```

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
        'Age': [25, 30, 22, 35, 28],
        'City': ['New York', 'London', 'Paris', 'London', 'New York'],
        'Score_Math': [80, 90, 75, 85, 92],
        'Score_Science': [70, 85, 80, 90, 78]}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

print("\n--- df.query() Examples (Filtering Rows by Column Values)
---")
```

```
# 1. Basic condition on a single column
query_age_over_25 = df.query('Age > 25')
print("\nRows where Age > 25:\n", query_age_over_25)

# 2. Multiple conditions using 'and'
query_london_age_over_25 = df.query('City == "London" and Age > 25')
print("\nRows where City is 'London' and Age > 25:\n",
query_london_age_over_25)

# 3. Using 'or'
query_score_math_over_90_or_age_under_23 = df.query('Score_Math > 90 or
Age < 23')
print("\nRows where Score_Math > 90 or Age < 23:\n",
query_score_math_over_90_or_age_under_23)

# 4. Using a Python variable in the query
min_score = 80
query_score_math_above_variable = df.query('Score_Math > @min_score')
print(f"\nRows where Score_Math > {min_score}:\n",
query_score_math_above_variable)

# 5. Using 'in'
cities_of_interest = ['New York', 'Paris']
query_city_in_list = df.query('City in @cities_of_interest')
print(f"\nRows where City is in {cities_of_interest}:\n",
query_city_in_list)

print("\n--- df.filter() Examples (Filtering Columns or Index Labels)
---")

# Set 'Name' as index for row filtering examples
df_indexed = df.set_index('Name')
print("\nDataFrame with 'Name' as index:\n", df_indexed)

# 1. Filtering columns using 'items'
filter_cols_items = df.filter(items=['Name', 'Age'])
print("\nColumns 'Name' and 'Age':\n", filter_cols_items)

# 2. Filtering columns using 'like'
filter_cols_like_score = df.filter(like='Score_')
print("\nColumns containing 'Score_':\n", filter_cols_like_score)

# 3. Filtering columns using 'regex'
filter_cols_regex_ends_th = df.filter(regex='th$', axis='columns')
print("\nColumns ending with 'th':\n", filter_cols_regex_ends_th)

# 4. Filtering index labels using 'items'
```

```
filter_rows_items = df_indexed.filter(items=['Alice', 'Charlie'],
axis='index')
print("\nRows with index 'Alice' and 'Charlie':\n", filter_rows_items)

# 5. Filtering index labels using 'like'
filter_rows_like_a = df_indexed.filter(like='a', axis='index')
print("\nRows with index containing 'a':\n", filter_rows_like_a)

# 6. Filtering index labels using 'regex'
filter_rows_regex_starts_with_D = df_indexed.filter(regex='^D',
axis='index')
print("\nRows with index starting with 'D':\n",
filter_rows_regex_starts_with_D)
```

2.7) Grouping

- ✓ ***groupby() as a way to split your DataFrame into smaller chunks or groups based on the unique values found in one or more specific columns.***
- ✓ ***Once groups are created , then apply calculations or transformations to each group independently and finally combine the results back into a structured format.***

- ✓ **`groupby()` follows a "Split-Apply-Combine" strategy:**

1) **Split:** The DataFrame is divided into multiple groups based on the values in the specified column(s). For example, if you group by a 'City' column, you'll get separate groups for each unique city (e.g., one group for 'London', one for 'Paris', etc.).
2) **Apply:** You then apply a function to each of these individual groups. Common operations include:
    a. **Aggregation:** Calculating summary statistics like mean(), sum(), count(), min(), max(), etc., for each group.
3) ***Combine: The results of the applied function on each group are then combined back into a new DataFrame or Series.***

```
[9] # Department number wise total salary
    res = df2.groupby('deptno')['sal'].sum()
    print(res)
```

```
deptno
10    90000
20    80000
Name: sal, dtype: int64
```

```
[10] # deptno wise minimum salary
     res = df2.groupby('deptno')['sal'].min()
     print(res)
```

```
deptno
10    20000
20    35000
Name: sal, dtype: int64
```

```
[11] #deptno wise Maximum salary
     res = df2.groupby('deptno')['sal'].max()
     print(res)
```

```
deptno
10    40000
20    45000
Name: sal, dtype: int64
```

```
# designation wise number of employees
res1= df2.groupby('desg')['eno'].count()
print(res1)
```

```
desg
Analyst          2
Data Engineer    1
Manager          1
sales Person     1
Name: eno, dtype: int64
```

3) reading and writing data in text format –

3.1) read_csv,

3.2) read_table.

3.1) read_csv :

- ✓ ***CSV files are the Comma Separated Files. It allows users to load tabular data into a DataFrame, which is a powerful structure for data manipulation and analysis.***
- ✓ ***To access data from the CSV file, we require a function read_csv() from Pandas that retrieves data in the form of the data frame***
- ✓ ***Syntax: pd.read_csv(filepath, sep=' ,' , header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)***

filename	The path to the CSV file
sep	Default(' , ')
header	✓ default 'infer' ✓ Specifies which row(s) to use as the column names ✓ 0: Use the first row as the header. ✓ 1: Use the second row as the header (0-indexed). ✓ None: The file has no header row, and Pandas will assign default integer column names.
index_col	✓ default None ✓ Specifies which column(s) to use as the row index of the DataFrame
usecols	✓ default None ✓ A list of column names or column numbers (0-indexed) to read from the file.
skiprows	✓ default None ✓ Number of rows to skip at the beginning of the file (int).
nrows	✓ default None ✓ Number of rows of the file to read


```
   ename    eno           desg    sal deptno
0   smith  e0001   sales Person  20000     10
1   Jones  e0002        Manager  45000     20
2    King  e0003        Analyst  30000     10
3  krishna  e0004  Data Engineer  40000     10
4    khan  e0005        Analyst  35000     20
```

```
df3 = pd.read_csv('emp.csv',usecols=[0,1,2],skiprows=[2,3])
print(df3)
```

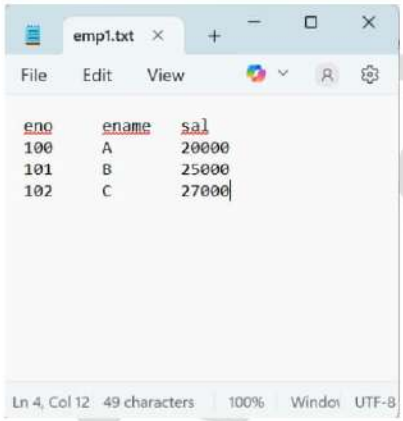
```
    eno    ename           desg
0  e0001    smith   sales Person
1  e0004  krishna  Data Engineer
2  e0005     khan        Analyst
```

3.2) read_table.

The primary purpose of pd.read_table() is to read data from a file (or a URL, file-like object, etc.) where values are separated by a delimiter other than a comma (although it can also handle comma-separated files).

- ✓ ***Syntax: pd.read_csv(filepath, sep=' ,' , header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)***

filename	The path to the CSV file
sep	Default(' \t')
header	✓ default 'infer' ✓ Specifies which row(s) to use as the column names ✓ 0: Use the first row as the header. ✓ 1: Use the second row as the header (0-indexed). ✓ None: The file has no header row, and Pandas will assign default integer column names.
index_col	✓ default None ✓ Specifies which column(s) to use as the row index of the DataFrame
usecols	✓ default None ✓ A list of column names or column numbers (0-indexed) to read from the file.
skiprows	✓ default None ✓ Number of rows to skip at the beginning of the file (int).
nrows	✓ default None ✓ Number of rows of the file to read




```
df4 = pd.read_table('emp1.txt',sep='\t')
print(df4)
```

```
   eno ename    sal
0  100     A  20000
1  101     B  25000
2  102     C  27000
```

Feature	pd.read_csv()	pd.read_table()
Primary Use	Comma-separated files (.csv)	General delimited files
Default sep	, (comma)	\t (tab)
Flexibility	Less flexible regarding separator	More flexible (explicitly set sep)
Convenience	More convenient for CSV files	Requires specifying sep for non-tab delimited

4.NUMPY

- ✓ *NumPy stands for Numerical Python.*
- ✓ *It is a Python library used for working with an array.*
- ✓ *In Python, we use the list for the array but it's slow to process.*
- ✓ *NumPy array is a powerful N-dimensional array object and is used in linear algebra, Fourier transform, and random number capabilities.*
- ✓ *It provides an array object much faster than traditional Python lists.*
- ✓ *Types of Array:*
    1. *One Dimensional Array*
    2. *Multi-Dimensional Array*

1. *One Dimensional Array:*
   - ✓ *A one-dimensional array is a type of linear array.*

One Dimensional Array

Example:

```
# importing numpy module
import numpy as np

# creating list
list = [1, 2, 3, 4]

# creating numpy array
sample_array = np.array(list)

print("List in python : ", list)

print("Numpy Array in python :",
      sample_array)
```

Output:

```
List in python :  [1, 2, 3, 4]
Numpy Array in python :  [1 2 3 4]
```

Example:

```
# importing numpy module
import numpy as np

# creating list
list_1 = [1, 2, 3, 4]
list_2 = [5, 6, 7, 8]
list_3 = [9, 10, 11, 12]

# creating numpy array
sample_array = np.array([list_1,
                         list_2,
                         list_3])

print("Numpy multi dimensional array in python\n",
      sample_array)
```

Output:

```
Numpy multi dimensional array in python
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

numpy.array(object, dtype=None, copy=True, order='K', ndmin=0)

This is the full syntax for the np.array() function in NumPy. Let's break down each parameter:

- **object (required):**
  - This is the **array-like object** you want to convert into a NumPy array. It can be a:
    - Python list or nested list
    - Python tuple or nested tuple
    - Another NumPy array
    - Any object that exposes the array interface
    - Any object whose __array__ method returns an array
- **dtype (optional):**
  - The desired data type of the resulting array. If None (the default), NumPy will try to infer the data type from the object.
  - You can explicitly specify a NumPy data type (e.g., np.int32, np.float64, np.str_).
- **copy (optional, default=True):**
  - If True, a new copy of the object is made.
  - If False, NumPy will try to return a view of the original object if possible. Be careful with this, as modifications to the view might affect the original object.
- **order (optional, default='K'):**
  - Specifies the memory layout of the array:
    - 'C': C-like row-major order.
    - 'F': Fortran-like column-major order.
    - 'A': Allow any order (NumPy might choose based on input).

- 'K': Keep order (try to match the input's order as closely as possible).

- **ndmin (optional, default=0):**
    - Specifies the minimum number of dimensions that the resulting array should have. NumPy will add leading axes (dimensions of size 1) to meet this requirement.

In simpler terms, you'll often use np.array() with just the object you want to convert and sometimes specify the dtype. The other parameters are for more advanced control over how the array is created.

Function Name	Purpose	Basic Syntax	Example
np.array()	Creates an array from an existing list, tuple, or array-like object.	np.array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)	np.array([1, 2, 3])
np.zeros()	Creates an array filled with zeros.	np.zeros(shape, dtype=float, order='C')	np.zeros((2, 3)) (creates a 2x3 array of zeros)
np.ones()	Creates an array filled with ones.	np.ones(shape, dtype=float, order='C')	np.ones(5) (creates an array of five ones)
np.empty()	Creates an array without initializing entries (can contain garbage values).	np.empty(shape, dtype=float, order='C')	np.empty((2, 2)) (creates a 2x2 uninitialized array)
np.full()	Creates an array filled with a specified scalar value.	np.full(shape, fill_value, dtype=None, order='C')	np.full((3,), 7) (creates an array of three sevens)
np.arange()	Creates an array with evenly spaced values within a given interval.	np.arange([start,] stop, [step,] dtype=None, *, like=None)	np.arange(0, 10, 2) (creates [0, 2, 4, 6, 8])
np.identity()	Creates a square identity array (similar to np.eye() for square matrices).	np.identity(n, dtype=None, *, like=None)	np.identity(4) (creates a 4x4 identity matrix)

```
[2] # Creating single Dimensional Array from List using Numpy
    import numpy as np
    l1=[45,23,67,54]
    a1= np.array(l1)
    print("Single Dimensio Array with Numpy ",a1)
```

```
Single Dimensio Array with Numpy  [45 23 67 54]
```

```
[4] # Creating Two Dimensional Array from Lists using Numpy
    l1=[1,2,3]
    l2=['hai','hello','hi']
    l3=[2.4,5.6,3.4]
    a2 = np.array([l1,l2,l3])
    print(a2)
```

```
[['1' '2' '3']
 ['hai' 'hello' 'hi']
 ['2.4' '5.6' '3.4']]
```

```
[7] # Creates an array filled with zeros.
    zarr= np.zeros((2,3))
    print(zarr)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

```
[8] # Creates an array filled with ones.
    oarr= np.ones((2,3))
    print(oarr)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
[14] #Creates an array without initializing entries (can contain garbage values).
     earr = np.empty((2,2))
     print(earr)
```

```
[[2.22e-322 1.14e-322]
 [3.31e-322 2.67e-322]]
```

```
[16] # Creates an array filled with a specified scalar value.
     res = np.full((3),7)
     print(res)
```

```
[7 7 7]
```

```
[18] # Creates an array with evenly spaced values within a given interval.
     res1 = np.arange(0,10,2)
     print(res1)
```

```
[0 2 4 6 8]
```

Universal Functions – Basic Unary Functions

Function	Description	Example
np.abs()	Element-wise absolute value.	np.abs(np.array([-1, -2, 3])) -> [1, 2, 3]
np.fabs()	Element-wise absolute value (for floating-point types).	np.fabs(np.array([-1.5, -2.8, 3.1])) -> [1.5, 2.8, 3.1]
np.sqrt()	Element-wise square root.	np.sqrt(np.array([4, 9, 16])) -> [2., 3., 4.]
np.exp()	Element-wise exponential (e^x).	np.exp(np.array([0, 1, 2])) -> [1. , 2.71828183, 7.3890561]
np.ceil()	Element-wise ceiling (smallest integer >= x).	np.ceil(np.array([1.2, 2.7, -1.5])) -> [2., 3., -1.]
np.floor()	Element-wise floor (largest integer <= x).	np.floor(np.array([1.2, 2.7, -1.5])) -> [1., 2., -2.]
np.round()	Element-wise rounding to the nearest integer.	np.round(np.array([1.4, 1.6, -1.4, -1.6])) -> [1., 2., -1., -2.]
np.rint()	Element-wise rounding to the nearest integer.	np.rint(np.array([1.4, 1.6, -1.4, -1.6])) -> [1., 2., -1., -2.]
np.sign()	Element-wise sign (+1 for positive, -1 for negative, 0 for zero).	np.sign(np.array([-2, 0, 3])) -> [-1, 0, 1]
np.cos()	Element-wise cosine.	np.cos(np.array([0, np.pi])) -> [1. , -1.]
np.sin()	Element-wise sine.	np.sin(np.array([0, np.pi/2])) -> [0. , 1.]
np.tan()	Element-wise tangent.	np.tan(np.array([0, np.pi/4])) -> [0. , 1.]
np.isnan()	Element-wise check for NaN (Not a Number).	np.isnan(np.array([1, np.nan, 3])) -> [False, True, False]
np.isfinite()	Element-wise check for finite numbers (not NaN or inf).	np.isfinite(np.array([1, np.inf, np.nan])) -> [True, False, False]
np.isinf()	Element-wise check for infinity.	np.isinf(np.array([1, np.inf, np.nan])) -> [False, True, False]

Binary functions ,

Function	Description	Example
np.add()	Element-wise addition.	np.add(np.array([1, 2]), np.array([3, 4])) -> [4, 6]
np.subtract()	Element-wise subtraction (second array from the first).	np.subtract(np.array([5, 3]), np.array([2, 1])) -> [3, 2]
np.multiply()	Element-wise multiplication.	np.multiply(np.array([2, 3]), np.array([4, 5])) -> [8, 15]
np.divide()	Element-wise division (first array by the second).	np.divide(np.array([10, 6]), np.array([2, 3])) -> [5., 2.]
np.floor_divide()	Element-wise floor division.	np.floor_divide(np.array([10, 7]), np.array([3, 3])) -> [3, 2]
np.power()	Element-wise exponentiation (first array to the power of the second).	np.power(np.array([2, 3]), np.array([3, 2])) -> [8, 9]
np.mod()	Element-wise modulo (remainder of division).	np.mod(np.array([7, 9]), np.array([3, 4])) -> [1, 1]
np.equal()	Element-wise equality comparison.	np.equal(np.array([1, 2]), np.array([1, 3])) -> [True, False]
np.not_equal()	Element-wise inequality comparison.	np.not_equal(np.array([1, 2]), np.array([1, 3])) -> [False, True]
np.less()	Element-wise less than comparison.	np.less(np.array([1, 2]), np.array([2, 1])) -> [True, False]
np.greater()	Element-wise greater than comparison.	np.greater(np.array([1, 2]), np.array([0, 3])) -> [True, False]
np.less_equal()	Element-wise less than or equal to comparison.	np.less_equal(np.array([1, 2]), np.array([2, 2])) -> [True, True]
np.greater_equal()	Element-wise greater than or equal to comparison.	np.greater_equal(np.array([1, 2]), np.array([0, 2])) -> [True, True]
np.logical_and()	Element-wise logical AND.	np.logical_and(np.array([True, False]), np.array([True, True])) -> [True, False]
np.logical_or()	Element-wise logical OR.	np.logical_or(np.array([True, False]), np.array([False, True])) -> [True, True]
np.logical_not()	Element-wise logical NOT (unary, but related).	np.logical_not(np.array([True, False])) -> [False, True]

6) File Input and Output with arrays

When working with numerical data using the NumPy library in Python, two fundamental tasks frequently arise:

Saving Arrays: The need to store NumPy arrays, whether created through computation or manipulation, to disk. This persistence allows for later use, sharing of data, and the preservation of work across different sessions or systems.

Loading Arrays: The requirement to read data stored in files back into NumPy arrays for analysis and processing. This enables the utilization of previously saved datasets or data originating from external sources.

Building complete data analysis workflows with NumPy crucially requires understanding the input and output capabilities, as they allow to persist the data and load it back seamlessly for further computation and analysis.

Saving NumPy Arrays:

1) **`np.save(file, arr)`**: Saves a single NumPy array to a binary `.npy` file.
2) **`np.savez(file, *args, **kwds)`**: Saves one or more NumPy arrays to a single uncompressed `.npz` archive.
3) **`np.savez_compressed(file, *args, **kwds)`**: Saves one or more NumPy arrays to a single compressed `.npz` archive.
4) **`np.savetxt(fname, X, fmt='%.18e', delimiter=' ', ...)`**: Saves a NumPy array to a plain text file.

Loading NumPy Arrays:

1) **`np.load(file, mmap_mode=None, allow_pickle=True, ...)`**: Loads arrays from `.npy` or `.npz` files. Returns a single array for `.npy` or a dictionary-like object for `.npz`.
2) **`np.loadtxt(fname, dtype=float, delimiter=None, skiprows=0, ...)`**: Loads data from a plain text file into a NumPy array.

Saving NumPy Arrays

1) **`np.save(file, arr)`**: Saves a single NumPy array to a binary `.npy` file.

```
import numpy as np

# Create a NumPy array
my_array = np.array([[1, 2, 3], [4, 5, 6]])

# Save the array to a .npy file
np.save('my_array.npy', my_array)

print("Array saved to my_array.npy")
```

```
Array saved to my_array.npy
```

2) **`np.savez(file, *args, **kwds)`**: Saves one or more NumPy arrays to a single uncompressed `.npz` archive.

```
import numpy as np

array1 = np.array([10, 20, 30])
array2 = np.array([[0.1, 0.2], [0.3, 0.4]])

# Save multiple arrays to a .npz file
np.savez('multiple_arrays.npz', arr1=array1, arr2=array2)

print("Multiple arrays saved to multiple_arrays.npz")
```

```
Multiple arrays saved to multiple_arrays.npz
```

3) **`np.savez_compressed(file, *args, **kwds)`: Saves one or more NumPy arrays to a single compressed `.npz` archive.**

```
import numpy as np

large_array = np.random.rand(1000, 1000)

# Save a compressed archive
np.savez_compressed('compressed_array.npz', data=large_array)

print("Large array saved to compressed_array.npz (compressed)")
```

```
Large array saved to compressed_array.npz (compressed)
```

4) **`np.savetxt(fname, X, fmt='%.18e', delimiter=' ', ...)`**: Saves a NumPy array to a plain text file.

```
import numpy as np

text_array = np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]])

# Save to a text file
np.savetxt('text_array.txt', text_array, delimiter=',', fmt='%.2f', header='Column1,Column2,Column3')

print("Array saved to text_array.txt")
```

```
Array saved to text_array.txt
```

Loading NumPy Arrays:

1) **`np.load(file, mmap_mode=None, allow_pickle=True, ...)`**: Loads arrays from `.npy` or `.npz` files. Returns a single array for `.npy` or a dictionary-like object for `.npz`.

```
import numpy as np

# Load the single array from .npy file
loaded_array = np.load('my_array.npy')
print("\nLoaded array from .npy:\n", loaded_array)

# Load multiple arrays from .npz file
loaded_multiple = np.load('multiple_arrays.npz')
print("\nLoaded arrays from .npz:")
print("Array 1:", loaded_multiple['arr1'])
print("Array 2:", loaded_multiple['arr2'])
loaded_multiple.close() # It's good practice to close .npz files
```

```
Loaded array from .npy:
 [[1 2 3]
 [4 5 6]]

Loaded arrays from .npz:
Array 1: [10 20 30]
Array 2: [[0.1 0.2]
 [0.3 0.4]]
```

2) **`np.loadtxt(fname, dtype=float, delimiter=None, skiprows=0, ...)`**: Loads data from a plain text file into a NumPy array.

```
import numpy as np

# Load from the text file
loaded_text_array = np.loadtxt('text_array.txt', delimiter=',', skiprows=1) # Skip the header row
print("\nLoaded array from text file:\n", loaded_text_array)
```

```
Loaded array from text file:
 [[1.1 2.2 3.3]
 [4.4 5.5 6.6]]
```

- For **saving and loading single NumPy arrays with full precision and type information**, use `np.save()` and `np.load()`.
- For **saving and loading multiple NumPy arrays in a single file**, use `np.savez()` and `np.load()`. Consider `np.savez_compressed()` for large datasets to save disk space.
- For **saving data in a human-readable format or for interoperability with other tools**, use `np.savetxt()` and `np.loadtxt()`. Be mindful of potential loss of type and shape information.

7) Linear Algebra –

7.1) commonly used linalg functions

Transpose:
You can transpose a matrix using the .T attribute or the np.transpose() function.

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
print("Original Matrix A:\n", A)
print("Transpose of A (A.T):\n", A.T)
print("Transpose of A (np.transpose(A)):\n", np.transpose(A))
```

```
Original Matrix A:
 [[1 2]
 [3 4]]
Transpose of A (A.T):
 [[1 3]
 [2 4]]
Transpose of A (np.transpose(A)):
 [[1 3]
 [2 4]]
```

Matrix Multiplication:
You can perform matrix multiplication using the @ operator (Python 3.5+) or the np.matmul() function. For element-wise multiplication, use the * operator.

```
B = np.array([[5, 6], [7, 8]])
print("Matrix B:\n", B)
print("Matrix Multiplication (A @ B):\n", A @ B)
print("Matrix Multiplication (np.matmul(A, B)):\n", np.matmul(A, B))
print("Element-wise Multiplication (A * B):\n", A * B)
```

```
Matrix B:
 [[5 6]
 [7 8]]
Matrix Multiplication (A @ B):
 [[19 22]
 [43 50]]
Matrix Multiplication (np.matmul(A, B)):
 [[19 22]
 [43 50]]
Element-wise Multiplication (A * B):
 [[ 5 12]
 [21 32]]
```

Determinant: Calculate the determinant of a square matrix using np.linalg.det().

```
det_A = np.linalg.det(A)
print("Determinant of A:", det_A)
```

```
Determinant of A: -2.000000000000000
```

Rank: Determine the rank of a matrix using np.linalg.matrix_rank(). The rank represents the number of linearly independent rows or columns.

```
C = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
rank_C = np.linalg.matrix_rank(C)
print("Rank of C:", rank_C)
```

```
Rank of C: 2
```

Trace: Calculate the sum of the diagonal elements of a square matrix using np.trace().

```
trace_A = np.trace(A)
print("Trace of A:", trace_A)
```

```
Trace of A: 5
```

Matrix Inversion:
Calculate the inverse of a square, non-singular matrix using np.linalg.inv().

```
inv_A = np.linalg.inv(A)
print("Inverse of A:\n", inv_A)
```

```
Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

Solving Linear Systems: To solve a system of linear equations represented as Ax=b using np.linalg.solve(). Here, A is the coefficient matrix, b is the constant vector, and x is the vector of unknowns.	``` a = np.array([[2, 1], [1, 3]]) b = np.array([4, 5]) x = np.linalg.solve(a, b) print("Solution for x in 2x + y = 4 and x + 3y = 5:", x) ```	``` Solution for x in 2x + y = 4 and x ```
Eigenvalues and Eigenvectors: Find the eigenvalues and eigenvectors of a square matrix using np.linalg.eig(). The function returns a tuple: the first element is an array of eigenvalues, and the second element is a matrix where each column is the eigenvector corresponding to the eigenvalue at the same index.	``` eigenvalues, eigenvectors = np.linalg.eig(A) print("Eigenvalues of A:", eigenvalues) print("Eigenvectors of A:\n", eigenvectors) ```	``` Eigenvalues of A: [-0.37228132 5.3 Eigenvectors of A: [[-0.82456484 -0.41597356] [0.56576746 -0.90937671]] ```