4.3 Dataframe Handling using Panda and Numpy:

4.3.1 csv and excel file extract and write using Dataframe

4.3.2 Extracting specific attributes and rows from dataframe.

4.3.3 Central Tendency measures:

4.3.3.1 mean, median, mode, variance, Standard Deviation

4.3.4 Dataframe functions: head, tail, loc, iloc, value, to_numpy(), describe()

NumPy Introduction

- **NumPy** = **Numerical Python** (open-source, free).
- Provides **fast mathematical operations** on **arrays & matrices**.
- Important for **Machine Learning & Data Science** (used with Pandas, Matplotlib, TensorFlow, etc.).
- Created in **2005 by Travis Oliphant**.

What is NumPy?

- A **Python library** for working with **arrays**.
- Supports **linear algebra, matrices.**
- Uses a special array object called **ndarray**.

Why Use NumPy?

- Normal **Python lists are slow**.
- **NumPy arrays (ndarray)** are up to **50x faster**.
- Widely used in **Data Science** for storing, analyzing, and processing data quickly.

Language Used

- NumPy = Python library.
- Written partly in **Python**, but heavy calculations are in **C/C++** for speed.

Import NumPy

Once NumPy is installed, import it in your applications by adding the import keyword:

```
import numpy
```

Example:

```
import numpy
arr=numpy.array([1, 2, 3, 4, 5])
print(arr)

importnumpy as n
lits1=[1,2,3,4]
print(n.array(list1))
```

Checking NumPy Version

The version string is stored under __version__ attribute.

Example

```
import numpy as np
print(np.__version__)
```

Create a NumPyndarray Object

The array object in NumPy is called ndarray.
We can create a NumPy ndarray object by using the array() function.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

output: [1 2 3 4 5]
<class 'numpy.ndarray'>

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Example:Use a tuple to create a NumPy array:

```
import numpy as np
arr=np.array((1, 2, 3, 4, 5))
print(arr)
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example: Create a 0-D array with value 42

```
import numpy as np
arr=np.array(42)
print(arr)
```

file name : array_type.py

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

Example:Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5])
print(arr)
```

file name : array_type.py

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.
NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Example:Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
Or
arr = np.array([[1, 2, 3], ['a', 'b', 'c']])
>>>print(arr)
```

output :

```
[['1' '2' '3']
['a' 'b' 'c']
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.
These are often used to represent a 3rd order tensor.

Example:Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr=np.array([[[1, 2, 3],[4, 5, 6]],[[1, 2, 3],[4, 5, 6]]])
print(arr)
```

Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example:Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
```

```
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

NumPy **Array Indexing**

Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example:Get the first element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Example:Get the second element from the following array.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[1])
```

Example:Get third and fourth elements from the following array and add them.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[2] + arr[3])
```

Negative Indexing

Use negative indexing to access an array from the end.

Example:Print the last element from the 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

NumPy Data Types

Data Types in Python

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3

- float - used to represent real numbers. e.g. 1.2, 42.42
- boolean - used to represent True or False.
- complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

The NumPy array object has a property called dtype that returns the data type of the array:

Example:Get the data type of an array object:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

Example:Get the data type of an array containing strings:

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

4.3 Dataframe Handling using Panda and Numpy:

4.3.1 csv and excel file extract and write using Dataframe

Reading and Writing CSV Files using Pandas

CSV file (Comma Separated Values) – Stores data in plain text (like Excel but simpler).

Excel file (.xlsx) – Spreadsheet file used in MS Excel.

Example: Reading a CSV file

```
import pandas aspd
                        # Read CSV file
df = pd.read_csv("students.csv")
print(df)
```

[read_csv_pandas.py]

Example: Writing DataFrame to CSV

df.to_csv("output.csv", index=False)
(index=False means row numbers won't be saved).
[read_csv_pandas.py]

Example: Reading Excel file

Import pandas as pd
df = pd.read_excel("students.xlsx")
print(df)
[read_excel_pandas.py]

Example: Writing DataFrame to Excel

df.to_excel("output.xlsx", index=False)
[read_excel_pandas.py]

Customizing Headers

By default, the read_csv() method uses the first row of the CSV file as the column headers.
Sometimes, these headers might have odd names, and you might want to use your own headers. Y
ou can set headers either after reading the file, simply by assigning the columns field of the DataFrame instance another list, or you can set the headers while reading the CSV in the first place.

```
import pandas as pd
col_names = ['Id',
             'Survived',
             'Passenger Class',
             'Full Name']
titanic_data = pd.read_csv(titanic.csv', names=col_names)
print(titanic_data.head())
```

Removing Headers

You can also decide to remove the header completely, which would result in a DataFrame that simply has 0...n header columns, by setting the header argument to None:

```
titanic_data = pd.read_csv(r'titanic.csv', header=None)
```

Specifying Delimiters

As stated earlier, you'll eventually probably encounter a CSV file that doesn't actually use commas to separate data.
In such cases, you can use the sep argument to specify other delimiters:

```
titanic_data = pd.read_csv(r'titanic.csv', sep=';')
```

code:

```
# readcsv.py
        import pandas as pd
        #df = pd.read_csv('a1.csv',header=None)
        df = pd.read_csv(r'a1.csv')
        col_names = ['studno','studname']
        df = pd.read_csv(r'a1.csv',names=col_names)
        print(df)
        #print(df.head())

    import pandas as pd
    df = pd.read_csv('a1.csv',header=None)
    df.to_csv('a1_update.csv')
    print(df)
    '''
    to_csv is a method to an object which is a df (DataFrame);
    whilepd is Panda module.
    Hence, your code was not working and throwing this Error
    " AttributeError: module 'pandas' has no attribute 'to_csv'"
    '''
```

Writing CSV Files with to_csv()

Again, DataFrames are tabular.

Turning a DataFrame into a CSV file is as simple as turning a CSV file into a DataFrame - we call the write_csv() function on the DataFrame instance.

When writing a DataFrame to a CSV file, you can also change the column names, using the columns argument, or specify a delimiter via the sep argument.

If you don't specify either of these, you'll end up with a standard Comma-Separated Value file.

```
        import pandas as pd
        cities=pd.DataFrame([['Sacramento','California'], ['Miami', 'Florida']],
        columns=['City', 'State'])
        cities.to_csv('cities.csv')
```

Here, we've made a simple DataFrame with two cities and their respective states. Then, we've gone ahead and saved that data into a CSV file using to_csv() and providing the filename.

Code:

```
# Tocsv.py
            import pandas as pd
            df = pd.read_csv('a1.csv',header=None)
            df.to_csv('a1_update.csv')
            print(df)
```

```
'''     to_csv is a method to an object which is a df (DataFrame);
        whilepd is Panda module.
        Hence, your code was not working and throwing this Error
        " AttributeError: module 'pandas' has no attribute 'to_csv'"
```

Some Examples:

Example #1: Save csv to working directory.

```
# importing pandas as pd
import pandas as pd

# list of name, degree, score
nm = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
dict = {'name': nm, 'degree': deg, 'score': scr}
df = pd.DataFrame(dict)
# saving the dataframe
df.to_csv('file1.csv')
```

Example #2: Saving CSV without *headers* and *index*.

```
# importing pandas as pd
import pandas as pd

# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
dict = {'name': nme, 'degree': deg, 'score': scr}
df = pd.DataFrame(dict)

# saving the dataframe
df.to_csv('file2.csv', header=False, index=False)
```

Example #3: Save csv file to a specified location.

```
# importing pandas as pd
import pandas as pd

# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
```

```
dict = {'name': nme, 'degree': deg, 'score': scr}
df = pd.DataFrame(dict)

# saving the dataframe
df.to_csv(r'C:\Users\Admin\Desktop\file3.csv', index=False)
```

4.3.3 Measures of Central Tendency and Spread

What is Central Tendency?

- Central tendency means finding the **center value** of a dataset.
- It tells us the **average** or **most common value** around which all data values are spread.
- It helps us to understand: Typical value of the dataset and how much new values may fit with existing data.

The **three main measures** are:

1. **Mean** (Average)
2. **Median** (Middle value)
3. **Mode** (Most frequent value)

1. Mean (Average)

Formula:

$$\text{Mean} = \frac{\text{Sum of all values}}{\text{Number of values}}$$

Example:

Dataset: [4, 8, 6, 5, 3, 2, 8, 9, 2, 5]

$$\text{Mean} = \frac{4+8+6+5+3+2+8+9+2+5}{10} = \frac{52}{10} = 5.2$$

Python Example (mean_function.py):

```
Importnumpy
list1 = [1, 3, 4, 5, 7, 9, 2]
x = numpy.mean(list1)
print("Mean is :", x)
```

Output: Mean is: 4.428571428571429

2. Median (Middle value)

- The **middle value** after sorting the data.
- If dataset has **odd** number of values → pick middle one.

- If dataset has **even** number of values → take average of two middle ones.

Example 1 (Odd count):
[3, 5, 1, 4, 2] → Sorted = [1, 2, 3, 4, 5] → Median = **3**
Example 2 (Even count):
[1, 2, 3, 4, 5, 6] → Middle two = (3+4)/2 = **3.5**

Python Example (median_function.py):
importnumpy
print(numpy.median([1, 3, 5, 7, 9, 11, 13]))
print(numpy.median([1, 3, 5, 7, 9, 11]))
print(numpy.median([-11, 5.5, -3.4, 7.1, -9, 22]))

Output: 7, 5, -3.4

3. Mode (Most frequent value/Occurance)

- The value that appears **most often** in the dataset.
- A dataset can have:
  - No mode
  - One mode
  - More than one mode

Example:
[1, 3, 3, 3, 5, 7, 7, 9, 11] → Mode = **3**

from statistics import mode
print(mode([4, 1, 2, 2, 3, 5])) # 2
print(mode([4, 1, 2, 2, 3, 5, 4])) # 4
print(mode(["few", "few", "many", "some", "many"])) # few

Measures of Spread (Dispersion)
These show how **spread out** the data is.

1. Range

- Difference between largest and smallest value.

$$Range = \text{Maximum value} - \text{Minimum value}$$

Example:

Dataset = [2, 4, 6, 8, 10]

Range = 10 – 2 = **8**

Code:

```
data = [2, 4, 6, 8, 10]
                    # Range
range_val = max(data) - min(data)
print("Range:", range_val)
```

2. Variance (σ²)

- Variance measures how far each value is from the **mean**.
- Formula:

$$Variance = \frac{\sum(x_i - \text{mean})^2}{n}$$

where x_i = each value, and n = number of values.

Larger variance = data is more spread out.

Code:

```
importnumpy as np
data = [2, 4, 6, 8, 10]
variance_val = np.var(data)
print("Variance:", variance_val)
```

Output: Variance: 8.0

1. **Mean** of `[2, 4, 6, 8, 10]` is
   $(2 + 4 + 6 + 8 + 10)/5 = 30/5 = 6$
2. Subtract the mean and square each result:
   $(2-6)^2 = 16,\ (4-6)^2 = 4,\ (6-6)^2 = 0,\ (8-6)^2 = 4,\ (10-6)^2 = 16$
3. Add the squared differences:
   $16 + 4 + 0 + 4 + 16 = 40$
4. Divide by the number of elements (population variance):
   $40/5 = 8.0$

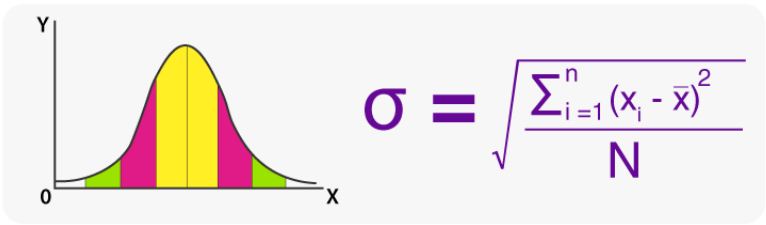
3. Standard Deviation (σ)

- Square root of variance.
- Easier to understand because it is in **same unit** as data.

```
importnumpy as np
data = [2, 4, 6, 8, 10]
            # Standard Deviation
std_val = np.std(data)
print("Standard Deviation:", std_val)
```

Formula for Standard Deviation

Standard Deviation Formula



Notations for Standard Deviation

- σ = Standard Deviation
- x_i = Terms Given in the Data
- $\bar{x}$ = Mean
- n = Total number of Terms

Example Question based on Standard Deviation Formula

Question: During a survey, 6 students were asked how many hours per day they study on an average? Their answers were as follows: 2, 6, 5, 3, 2, 3. Evaluate the standard deviation.

Solution:

Find the mean of the data:

(2+6+5+3+2+3)/6
= 3.5
Step 2: Construct the table:

x_1	$x_1 - \bar{x}$	$(x_1 - \bar{x})^2$
2	-1.5	2.25
6	2.5	6.25
5	1.5	2.25
3	-0.5	0.25
2	-1.5	2.25
3	-0.5	0.25
= 13.5		

Step 3: Now, use the Standard Deviation formula

Sample Standard Deviation =

$$s = \sqrt{\frac{\sum(X-\bar{X})^2}{n-1}}$$

$=\sqrt{(13.5/[6-1])}$

$=\sqrt{[2.7]}$

$=1.643$

Note:

- **Range**:
- Difference between the largest and smallest values.
- It takes largest and smallest value from the data set.
- It uses only the data less than 6 in a data set.
- **Variance**:
- Variance measures how far a data set is spread out.
- The average squared deviation of values from the mean.
- **StandardDeviation**
- It takes all data from the data set.
- We can use standard deviation, if the data in a data set are more than 6.
- Standard Deviation is the square root of variance.

Measure	Meaning	Example Result
Mean	Average value	[1,2,3,4,5] → 3
Median	Middle value	[1,2,3,4,5,6] → 3.5
Mode	Most frequent value	[2,2,3,4] → 2
Range	Largest - Smallest	[2,4,6,8] → 6
Variance	Spread from mean (squared)	High variance = more spread
Std. Deviation	Square root of variance	Easier to interpret

Dataframe functions: head, tail, loc, iloc, value, to_numpy(), describe()

1. head()

- Shows the **first n rows** of a DataFrame.
- By default, it shows the **first 5 rows**.
- If we give negative values for 'n', this method returns all the rows except the last n rows which is equivalent to df[:-n].

Example:

```
import pandas aspd
data = {'Name': ['A', 'B', 'C', 'D', 'E'], 'Marks': [85, 90, 78, 88, 95]}
df = pd.DataFrame(data)
print(df.head())      # First 5 rows
print(df.head(3))     # First 3 rows
```

Output:

```
Name  Marks
0A85
1B90
2   C    78
```

```
#HEAD METHOD

import pandas as pd

data ={'Language':['English','Hindi','Gujarati','Tamil','Malyalam','Marathi','Konkani']}

df= pd.DataFrame(data)
print("First 5 rows of the DataFrame is:")
print(df)

print(df.head())
print(df.head(2))
print(df.head(-4))

#TAIL METHOD

import numpy as np
import pandas as pd
df = pd.DataFrame({'animal':['snake', 'bat', 'tiger', 'lion', 'fox', 'eagle', 'shark', 'dog', 'deer']})
#print(df)
#print(df.tail())
print(df.tail(4))
```

2. tail()

- Shows the **last n rows** of a DataFrame.
- By default, it shows the **last 5 rows**.

Example:

```
print(df.tail())     # Last 5 rows
print(df.tail(2))    # Last 2 rows
```

5. to_numpy()

- The .to_numpy() method is primarily used with **pandas**DataFrames to convert them into **NumPy arrays**.
- Useful for mathematical operations.

Example:

```
import pandas as pd
import numpy as np

# Create a DataFrame with student scores
data = {'Math': [85, 90, 78, 92, 88],
   'Science': [89, 94, 75, 91, 87]
}
df = pd.DataFrame(data)
                # Display the DataFrame
print("Original DataFrame:")
print(df)
print(type(df))
                # Convert DataFrame to NumPy array
array_data = df.to_numpy()
print("Converted NumPy array:")
print(array_data)
print(type(array_data))
```

6. describe()

- Gives **summary statistics** of numerical columns.
- It provides an overview of the distribution of numerical data — useful for quick exploration of a dataset.
- Includes count, mean, min, max, std (standard deviation), and quartiles.

Statistic	Meaning
count	Number of non-null entries
mean	Average of the values
std	Standard deviation
min	Minimum value
25%	1st quartile (Q1)
50%	Median (Q2)
75%	3rd quartile (Q3)
max	Maximum value

Example:

```
import pandas as pd
data = {
   'Math': [85, 90, 78, 92, 88],
   'Science': [89, 94, 75, 91, 87]
}
df = pd.DataFrame(data)
# Describe the dataset
print(df.describe())
```

Output:

```
Marks
count5.000000
mean87.200000
std6.758466
min78.000000
25%    85.000000
50%    88.000000
75%    90.000000
max95.000000
```

What is loc?

loc() means **location by labels (names or row numbers written in the index)**.
It is used when you want to select rows/columns using **row names or labels**.

Example:

```
import pandas as pd
                    # Create a DataFrame
df = pd.DataFrame(
{
   "Name": ["Amit", "Bhavna", "Chirag", "Divya"],
   "Dept": ["IT", "HR", "IT", "Finance"],
   "Salary": [50000, 60000, 55000, 70000]
}
)
print(df)
```

Output:

```
   Name    Dept   Salary
0   Amit       IT   50000
1  Bhavna     HR   60000
2  Chirag      IT   55000
3   Divya     Finance   70000
```

Select a single row by label

```
print(df.loc[2])   # Row with index label 2
```

Output:

```
Name      Chirag
Dept        IT
Salary    55000
```

Select multiple rows

```
print(df.loc[[1, 3]])
```

Output:

```
    Name      Dept    Salary
1  Bhavna     HR   60000
3  Divya    Finance   70000
```

Select specific columns

```
print(df.loc[:, ["Name", "Salary"]])
```

Output:

```
    Name  Salary
0    Amit   50000
1  Bhavna 60000
2  Chirag   55000
3   Divya   70000
```

Update values using loc

```
df.loc[[0, 3], "Salary"] = 60000
print(df)
```

What is iloc?

iloc means **index-location** (by position).
It is used when you want to select rows/columns by **numbers (0, 1, 2 …)**.

Example: Select specific rows & columns

```
print(df.iloc[[1, 3], [0, 2]])   # rows 1 and 3, columns 0 and 2
```

Output:

```
Name  Salary
1  Bhavna   60000
3   Divya   70000
```

Update a value using iloc

```
df.iloc[1, 2] = 100   # row 1, col 2 → Salary changed to 100
print(df)
```

Slice rows and columns

```
print(df.iloc[0:3, 0:2])   # first 3 rows, first 2 cols
```

Output:

```
Name  Dept
0    Amit    IT
1  Bhavna    HR
```

2 Chirag IT

Difference Between loc and iloc

Feature	loc (Label based)	iloc (Index-position based)
Usage	Row/Column **labels (names)**	Row/Column **positions (0,1,2)**
Example	df.loc[2, "Name"] → Chirag	df.iloc[2, 0] → Chirag
Error type	KeyError if label not found	IndexError if index is out of range

- Use **loc** when you know the **row/column names**.
- Use **iloc** when you want to use **row/column numbers**.