Lecture 8

Learning outcomes:

- *Data Analysis and Manipulation:*
  - *Introduction to pandas for tabular data*
  - *Cleaning and processing experimental or simulation data*

What is pandas?

- Pandas is an open-source python library that provides data structures and is designed to handle and analyze tabular data in Python.
- Pandas allows you to easily manage and manipulate data in Python. Pandas provides two main data structures: Series and DataFrames:

- A series in Pandas is a one-dimensional labeled data structure. It is similar to a 1D array in NumPy, but has an index that allows access to the values by label. A series can contain any kind of data: integers, strings, Python objects.etc.
- A DataFrame in Pandas is a two-dimensional labeled data structure. It is similar to a 2D array in NumPy, but has an index that allows access to the values per label, per row, and column.

What is pandas used for?

- Pandas is used throughout the data analysis workflow. With pandas, you can:

➢ Import datasets from databases, spreadsheets, comma-separated values (CSV) files, and more.

➢ Clean datasets, for example, by dealing with missing values.

➢ Tidy datasets by reshaping their structure into a suitable format for analysis.

➢ Aggregate data by calculating summary statistics such as the mean of columns, correlation between them, and more.

➢ Visualize datasets and uncover insights.

- Pandas also contains functionality for time series analysis and analyzing text data.

A series in Pandas

❑ A Pandas series has two distinct parts:

- Index (index): An array of tags associated with the data.
- Value (value): An array of data.

	apples
0	3
1	2
2	0
3	1

- A series can be created using the Series class of the library with a list of elements as an argument. For example:

```
import pandas as pd
serie1 = pd.Series([1, 2, 3, 4, 5])
serie1
```

- **Output**:

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

- This will create a series with elements 1, 2, 3, 4 and 5. In addition, since we have not included information about the indexes, an automatic index is generated starting at 0:

```
serie2 = pd.Series([1, 2, 3, 4, 5], index = ["a", "b", "c", "d", "e"])
serie2
```

✓**Output**:

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

- The above series has an index composed of letters.

❖Both series examples store the same values, but the way they are accessed may vary according to the index.

In a series, its elements can be accessed by index or by position. Below are some operations that can be performed using the above series:

```
# Access the third element
print(serie2["c"]) # By index
print(serie2[2]) # By position

# Change the value of the second element
serie2["b"] = 7
print(serie2)

# Add 10 to all elements
serie2 += 10
print(serie2)

# Calculate the sum of the elements
sum_all = serie2.sum()
print(sum_all)
```

✓ **Output**:

```
3
3
a    1
b    7
c    3
d    4
e    5
dtype: int64
a    11
b    17
c    13
d    14
e    15
dtype: int64
70
```

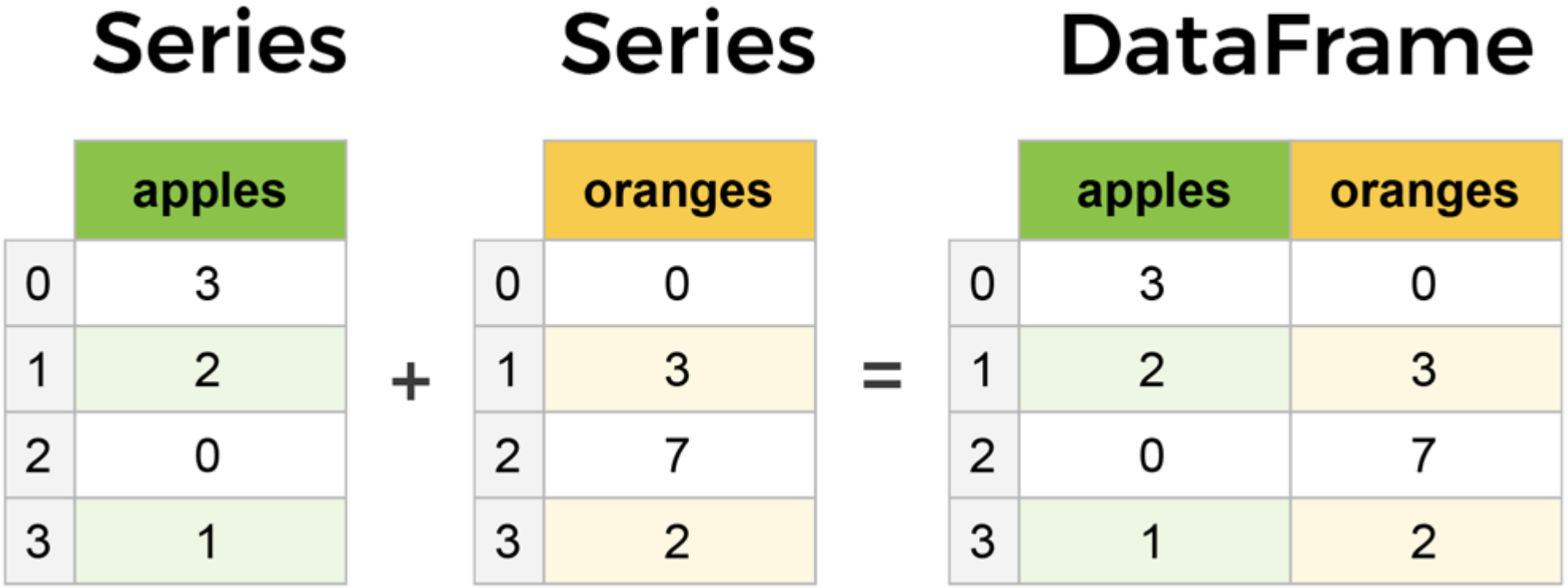
A DataFrame in Pandas

❑ A DataFrame in Pandas has several differentiated parts:

- Data (data): An array of values that can be of different types per column.
- Row index (row index): An array of labels associated to the rows.
- Column index (column index): An array of labels associated to the columns.

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

- A DataFrame can be seen as a set of series joined in a tabular structure, with an index per row in common and a column index specific to each series.



- A DataFrame can be created using the DataFrame class. For example:

```
dataframe = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
dataframe
```

✓**Output**:

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

✓This will create a DataFrame with three rows and three columns for each row. As was the case with series, a DataFrame will generate automatic indexes for rows and columns if they are not passed as arguments in the constructor of the class.

- If we wanted to create a new DataFrame with concrete indexes for rows and columns, it would be programmed as follows:

```
data = {
    "col A": [1, 2, 3],
    "col B": [4, 5, 6],
    "col C": [7, 8, 9]
}
dataframe = pd.DataFrame(data, index = ["a", "b", "c"])
dataframe
```

✓**Output**:

	col A	col B	col C
a	1	4	7
b	2	5	8
c	3	6	9

✓In this way, a custom index is provided for the columns (labeling the rows within a dictionary) and for the rows (with the index argument, as was the case with the series).

- In a DataFrame its elements can be accessed by index or by position. Below are some operations that can be performed using the above DataFrame:

```
# Access all the data in a column
print(dataframe["col A"]) # By index
print(dataframe.loc[:,"col A"]) # By index
print(dataframe.iloc[:,0]) # By position

# Access all the data in a row
print(dataframe.loc["a"]) # By index
print(dataframe.iloc[0]) # By position

# Access to a specific element (row, column)
print(dataframe.loc["a", "col A"]) # By index
print(dataframe.iloc[0, 0]) # By position
```

✓**Output**:

a 1
b 2
c 3
Name: col A, dtype: int64
a 1
b 2
c 3
Name: col A, dtype: int64
a 1
b 2
c 3
Name: col A, dtype: int64TI

✓**Output**:

```
col A    1
col B    4
col C    7
Name: a, dtype: int64
col A    1
col B    4
col C    7
Name: a, dtype: int64
1
1
```

```
# Create a new column
dataframe["col D"] = [10, 11, 12]
print(dataframe)

# Create a new row
dataframe.loc["d"] = [13, 14, 15, 16]
print(dataframe)

# Multiply by 10 the elements of a column
dataframe["col A"] *= 10
print(dataframe)

# Calculate the sum of all elements
sum_all = dataframe.sum()
print(sum_all)
```

✓ **Output**:

```
col A  col B  col C  col D
a      1      4      7     10
b      2      5      8     11
c      3      6      9     12
   col A  col B  col C  col D
a      1      4      7     10
b      2      5      8     11
c      3      6      9     12
d     13     14     15     16
   col A  col B  col C  col D
a     10      4      7     10
b     20      5      8     11
c     30      6      9     12
d    130     14     15     16
col A    190
col B     29
col C     39
col D     49
dtype: int64
```

Functions in Python Pandas

- Pandas provide a large number of predefined functions that can be applied on the data structures seen above. Some of the most used in data analysis are:

```
import pandas as pd

s1 = pd.Series([1, 2, 3])
s2 = pd.Series([4, 5, 6])
d1 = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
d2 = pd.DataFrame([[7, 8, 9], [10, 11, 12]])

# Arithmetic Operations
print("Sum of series:", s1.add(s2))
print("Sum of DataFrames:", d1.add(d2))
```

```
# Statistical Operations
# They can be applied in the same way to DataFrames
print("Mean:", s1.mean())
print("Median:", s1.median())
print("Number of elements:", s1.count())
print("Standard deviation:", s1.std())
print("Variance:", s1.var())
print("Maximum value:", s1.max())
print("Minimum value:", s1.min())
print("Correlation:", s1.corr(s2))
print("Statistic summary:", s1.describe())
```

✓**Output**:

Sum of series: 0 5
1 7
2 9
dtype: int64
Sum of DataFrames: 0 1 2
0 8 10 12
1 14 16 18
Mean: 2.0
Mediaa: 2.0
Number of elements: 3
Standard derviation: 1.0

Variance: 1.0
Maximum value: 3
Minimum value: 1
Correlation: 1.0
Statistic summary: count   3.0
mean   2.0
std   1.0
min   1.0
25%   1.5
50%   2.0
75%   2.5
max   3.0
dtype: float64

Pandas allows you to use custom python functions (including lambda)

- In addition to the Pandas predefined functions, we can also define and apply others to the data structures. To do this, we have to program the function to receive a value (or a column or row in the case of a DataFrame) and return another modified one, and reference it with apply.

- In addition, this function allows using lambda expressions for the anonymous declaration of functions.

- The following shows how to apply functions to series:

```
import pandas as pd
s = pd.Series([1, 2, 3, 4])

# Explicit definition of the function
def squared(x):
    return x ** 2
s1 = s.apply(squared)
print(s1)

# Anonymous definition of the function
s2 = s.apply(lambda x: x ** 2)
print(s2)
```

✓**Output**:

```
0     1
1     4
2     9
3    16
dtype: int64
0     1
1     4
2     9
3    16
dtype: int64
```

- The following shows how to apply functions to a DataFrame, which can be done by row, by column or by elements, similar to series:

```
df = pd.DataFrame({
    "A": [1, 2, 3],
    "B": [4, 5, 6]
})

# Apply function along a column
df["A"] = df["A"].apply(lambda x: x ** 2)
print(df)

# Apply function along a row
df.loc[0] = df.loc[0].apply(lambda x: x ** 2)
print(df)

# Apply function to all elements
df = df.applymap(lambda x: x ** 2)
print(df)
```

✓**Output**:

```
   A  B
0  1  4
1  4  5
2  9  6
   A   B
0  1  16
1  4   5
2  9   6
    A    B
0   1  256
1  16   25
2  81   36
```

- apply is more flexible than other vectorized Pandas functions, but can be slower, especially when applied to large data sets. It is always important to explore the Pandas or NumPy built-in functions first, as they are usually more efficient than the ones we could implement ourselves.

- Also, this function can return results in different ways, depending on the function applied and how it is configured.

Importing data in pandas

- To begin working with pandas, import the pandas Python package as shown below. When importing pandas, the most common alias for pandas is pd.

✓import pandas as pd

- pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, …), each of them with the prefix read_*.

Importing CSV files

- Use read_csv() with the path to the CSV file to read a comma-separated values file (see our tutorial on importing data with read_csv() for more detail).

- df = pd.read_csv(“dataset.csv")

- This read operation loads the CSV file diabetes.csv to generate a pandas Dataframe object df. Throughout this tutorial, you'll see how to manipulate such DataFrame objects.

Importing text files

- Reading text files is similar to CSV files. The only nuance is that you need to specify a separator with the sep argument, as shown below. The separator argument refers to the symbol used to separate rows in a DataFrame. Comma (sep = ","), whitespace(sep = "\s"), tab (sep = "\t"), and colon(sep = ":") are the commonly used separators. Here \s represents a single white space character.

- df = pd.read_csv(" dataset.txt", sep="\s")

Importing Excel files (single sheet)

- Reading excel files (both XLS and XLSX) is as easy as the read_excel() function, using the file path as an input.

- df = pd.read_excel(dataset.xlsx’)

- You can also specify other arguments, such as header for to specify which row becomes the DataFrame's header. It has a default value of 0, which denotes the first row as headers or column names. You can also specify column names as a list in the names argument. The index_col (default is None) argument can be used if the file contains a row index.

Note: In a pandas DataFrame or Series, the index is an identifier that points to the location of a row or column in a pandas DataFrame. In a nutshell, the index labels the row or column of a DataFrame and lets you access a specific row or column by using its index (you will see this later on). A DataFrame’s row index can be a range (e.g., 0 to 303), a time series (dates or timestamps), a unique identifier (e.g., employee_ID in an employees table), or other types of data. For columns, it's usually a string (denoting the column name).

Importing Excel files (multiple sheets)

- Reading Excel files with multiple sheets is not that different. You just need to specify one additional argument, sheet_name, where you can either pass a string for the sheet name or an integer for the sheet position (note that Python uses 0-indexing, where the first sheet can be accessed with sheet_name = 0)

Extracting the second sheet since Python uses 0-indexing

- df = pd.read_excel(dataset _multi.xlsx', sheet_name=1)

Importing JSON file

- Similar to the read_csv() function, you can use read_json() for JSON file types with the JSON file name as the argument (for more detail read this tutorial on importing JSON and HTML data into pandas). The below code reads a JSON file from disk and creates a DataFrame object df.

- df = pd.read_json(" dataset.json")

Writing Tabular Data

- Writing in Excel file:

```
# Reading the data from a CSV file named 'dataset.csv' into a pandas DataFrame
data = pd.read_csv('dataset.csv')

# Specifying the path for the new Excel file to be created
excel_file_path = 'newDataset.xlsx'

# Writing the DataFrame to an Excel file with the specified path, excluding the index column
data.to_excel(excel_file_path, index=False)

# Displaying a message indicating that the data has been successfully written to the Excel file
print(f'Data written to Excel file: {excel_file_path}’)
```

✓ **Output**:

```
Data written to Excel file: newDataset.xlsx
```

- Writing in CSV file:

```
# Reading the data from a CSV file named 'dataset.csv' into a pandas DataFrame
data = pd.read_csv('dataset.csv')

# Specifying the path for the new CSV file to be created
csv_file_path = 'newDataset.csv'

# Writing the DataFrame to a CSV file with the specified path, excluding the index column
data.to_csv(csv_file_path, index=False)

# Displaying a message indicating that the data has been successfully written to the CSV file
print(f'Data written to CSV file: {csv_file_path}’)
```

✓**Output**:

```
Data written to CSV file: newDataset.csv
```

Outputting data in pandas

- Just as pandas can import data from various file types, it also allows you to export data into various formats. This happens especially when data is transformed using pandas and needs to be saved locally on your machine. Below is how to output pandas DataFrames into various formats.

❖ Outputting a DataFrame into a CSV file

- A pandas DataFrame (here we are using df) is saved as a CSV file using the .to_csv() method. The arguments include the filename with path and index – where index = True implies writing the DataFrame's index.
- df.to_csv("dataset_out.csv", index=False)

❖Outputting a DataFrame into a JSON file

- Export DataFrame object into a JSON file by calling the .to_json() method.
- df.to_json(" dataset_out.json")

- Note: A JSON file stores a tabular object like a DataFrame as a key-value pair. Thus you would observe repeating column headers in a JSON file.

❖Outputting a DataFrame into a text file

- As with writing DataFrames to CSV files, you can call .to_csv(). The only differences are that the output file format is in .txt, and you need to specify a separator using the sep argument.
- df.to_csv('dataset_out.txt', header=df.columns, index=None, sep=' ‘)

❖Outputting a DataFrame into an Excel file

- Call .to_excel() from the DataFrame object to save it as a “.xls” or “.xlsx” file.

- df.to_excel("dataset_out.xlsx", index=False)

Viewing and understanding DataFrames using pandas

- After reading tabular data as a DataFrame, you would need to have a glimpse of the data.
- You can either view a small sample of the dataset or a summary of the data in the form of summary statistics.

❖ **How to view data using .head() and .tail()**

- You can view the first few or last few rows of a DataFrame using the .head() or .tail() methods, respectively. You can specify the number of rows through the n argument (the default value is 5).

✓df.head()

- First 10 rows of the DataFrame

✓df.tail(n = 10)

❖Understanding data using .describe()

- The .describe() method prints the summary statistics of all numeric columns, such as count, mean, standard deviation, range, and quartiles of numeric columns.

✓df.describe()

- You can also modify the quartiles using the percentiles argument. Here, for example, we're looking at the 30%, 50%, and 70% percentiles of the numeric columns in DataFrame df.

✓df.describe(percentiles=[0.3, 0.5, 0.7])

- You can also isolate specific data types in your summary output by using the include argument. Here, for example, we're only summarizing the columns with the integer data type.

✓df.describe(include=[int])

- Similarly, you might want to exclude certain data types using exclude argument.
- Get summary statistics of non-integer columns only

✓df.describe(exclude=[int])

- Often, practitioners find it easy to view such statistics by transposing them with the .T attribute.

Transpose summary statistics with .T

✓df.describe().T

Understanding data using .info()

- The .info() method is a quick way to look at the data types, missing values, and data size of a DataFrame. Here, we're setting the show_counts argument to True, which gives a few over the total non-missing values in each column. We're also setting memory_usage to True, which shows the total memory usage of the DataFrame elements. When verbose is set to True, it prints the full summary from .info().

✓df.info(show_counts=True, memory_usage=True, verbose=True)

Understanding your data using .shape

- The number of rows and columns of a DataFrame can be identified using the .shape attribute of the DataFrame. It returns a tuple (row, column) and can be indexed to get only rows, and only columns count as output.

✓df.shape # Get the number of rows and columns

✓df.shape[0] # Get the number of rows only

✓df.shape[1] # Get the number of columns only

- Get all columns and column names

Calling the .columns attribute of a DataFrame object returns the column names in the form of an Index object. As a reminder, a pandas index is the address/label of the row or column.

✓df.columns

Renaming columns

- A common data cleaning task is renaming columns. With the .rename() method, you can use columns as an argument to rename specific columns. The below code shows the dictionary for mapping old and new column names.

✓df.rename(columns = {'col D':'Name'}, inplace = True)

✓df.head()

- You can also directly assign column names as a list to the DataFrame.

✓df.columns = ['1', '2', '3', '4’ ,' 5', '6 ']

✓df.head()

Ensuring Data Integrity by Checking Data Types

When loading data into a Pandas DataFrame, it is important to verify the data types match what you expect. This avoids potential errors down the line.

- Use the .dtypes attribute to print the data types of each column:

✓df.dtypes

- For example, if a column contains numeric data but is loaded as objects/strings, calculations may produce unexpected results. Cast to the appropriate dtype using .astype():

✓df[‘col name'] = df[‘col name'].astype(float)

Checking for missing values in pandas with .isnull()

- The sample DataFrame does not have any missing values. Let's introduce a few to make things interesting. The .copy() method makes a copy of the original DataFrame. This is done to ensure that any changes to the copy don’t reflect in the original DataFrame. Using .loc (to be discussed later), you can set rows two to five of the income column to NaN values, which denote missing values.

✓df2 = df.copy()

✓df2.loc[2:5, ' income '] = None

✓df2.head(7)

- You can check whether each element in a DataFrame is missing using the .isnull() method.

✓df2.isnull().head(7)

- Given it's often more useful to know how much missing data you have, you can combine .isnull() with .sum() to count the number of nulls in each column.

✓df2.isnull().sum()

- You can also do a double sum to get the total number of nulls in the DataFrame.

✓df2.isnull().sum().sum()

Cleaning data using pandas

- Data cleaning is one of the most common tasks in data science. pandas lets you preprocess data for any use, including but not limited to training machine learning and deep learning models. Let’s use the DataFrame df2 from earlier, having four missing values, to illustrate a few data cleaning use cases. As a reminder, here's how you can see how many missing values are in a DataFrame.

✓df2.isnull().sum()

Dealing with missing data technique #1: Dropping missing values

- One way to deal with missing data is to drop it. This is particularly useful in cases where you have plenty of data and losing a small portion won't impact the downstream analysis. You can use a .dropna() method as shown below. Here, we are saving the results from .dropna() into a DataFrame df3.

✓df3 = df2.copy()

✓df3 = df3.dropna()

✓df3.shape

(6, 6) # this is 6 rows less than df2

❖Dropping missing data in pandas

- The axis argument lets you specify whether you are dropping rows, or columns, with missing values. The default axis removes the rows containing NaNs. Use axis = 1 to remove the columns with one or more NaN values. Also, notice how we are using the argument inplace=True which lets you skip saving the output of .dropna() into a new DataFrame.

✓df3 = df2.copy()

✓df3.dropna(inplace=True, axis=1)

✓df3.head()

- You can also drop both rows and columns with missing values by setting the how argument to 'all'

✓df3 = df2.copy()

✓df3.dropna(inplace=True, how='all')

Dealing with missing data technique #2: Replacing missing values

- Instead of dropping, replacing missing values with a summary statistic or a specific value (depending on the use case) maybe the best way to go. For example, if there is one missing row from a temperature column denoting temperatures throughout the days of the week, replacing that missing value with the average temperature of that week may be more effective than dropping values completely. You can replace the missing data with the row, or column mean using the code below.

```
df3 = df2.copy()
# Get the mean of income
mean_value = df3['5'].mean()
# Fill missing values using .fillna()
df3 = df3.fillna(mean_value)
df3.head()
```

Dealing with Duplicate Data

- Let's add some duplicates to the original data to learn how to eliminate duplicates in a DataFrame. Here, we are using the .concat() method to concatenate the rows of the df2 DataFrame to the df2 DataFrame, adding perfect duplicates of every row in df2.

✓df3 = pd.concat([df2, df2])

✓df3.shape

(20, 6)

- You can remove all duplicate rows (default) from the DataFrame using .drop_duplicates() method.

✓df3 = df3.drop_duplicates()

✓df3.shape

(10, 6)

Data analysis in pandas

- The main value proposition of pandas lies in its quick data analysis functionality. In this section, we'll focus on a set of analysis techniques you can use in pandas.

❖Summary operators (mean, mode, median)

- As you saw earlier, you can get the mean of each column value using the .mean() method.

✓df.mean()

- A mode can be computed similarly using the .mode() method.

✓df.mode()

- Similarly, the median of each column is computed with the .median() method

✓df.median()

❖Create new columns based on existing columns

- pandas provides fast and efficient computation by combining two or more columns like scalar variables. The below code divides each value in the column 1 with the corresponding value in the 2 column to compute a new column named new.

✓df2['New'] = df2['1']/df2['2’]

df2.head()

❖Counting using .value_counts()

- Often times you'll work with categorical values, and you'll want to count the number of observations each category has in a column. Category values can be counted using the .value_counts() methods. Here, for example, we are counting the number of observations where the Football player (1) and the number of observations where the Not a football player (0).

✓df['6'].value_counts()

- Adding the normalize argument returns proportions instead of absolute counts.

✓df['6'].value_counts(normalize=True)

Aggregating data with .groupby() in pandas

- pandas lets you aggregate values by grouping them by specific column values. You can do that by combining the .groupby() method with a summary method of your choice. The below code displays the mean of each of the numeric columns grouped by Name.

✓df.groupby(‘4').mean()

- .groupby() enables grouping by more than one column by passing a list of column names, as shown below.

✓df.groupby(['4', '7']).mean()

- Any summary method can be used alongside .groupby(), including .min(), .max(), .mean(), .median(), .sum(), .mode(), and more.