University of Sulaimani
College of Science
Computer Department
4th Stage

Data Science Management

Exploratory Data Analysis (EDA)

Pandas - Python Data Analysis Library

Theoretical and practical lectures

Class 5

Assist. Prof. Dr. Miran Taha Abdullah
2025-2026

Class Agenda

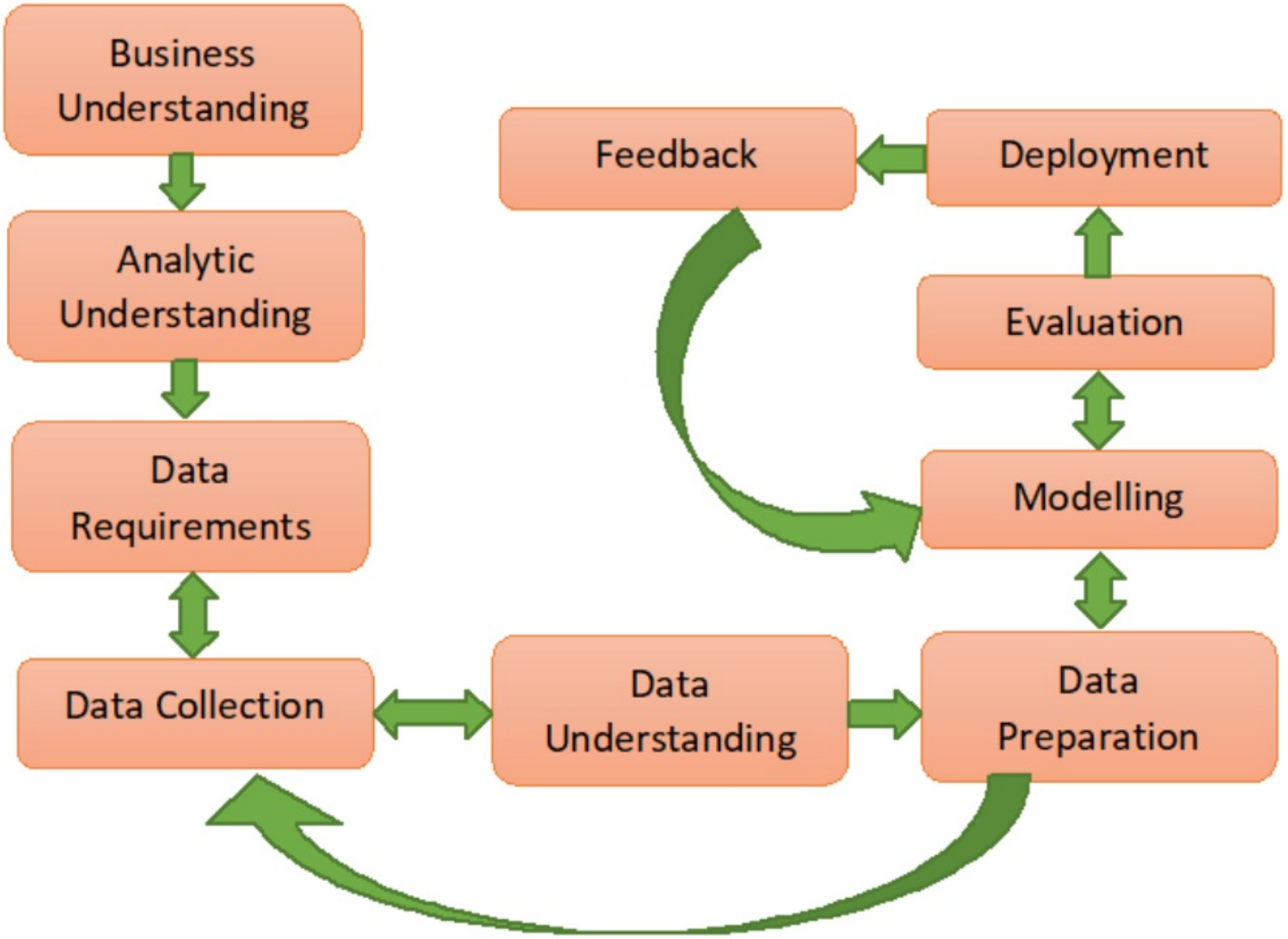
- Review of the previous lecture (Diagram)
- Introduction to Pandas (EDA)
- Main Pandas methods
- Pandas for Data Analysis
- Implementations using Python

Objectives

1. To understand the functionalities of the Pandas library for data manipulation and analysis.

2. To apply Pandas in cleaning and organizing datasets for effective data science workflows.

- **Review of the previous lecture (Diagram)**



Important Modules

1. **NumPy**: Supports large, multi-dimensional arrays and matrices, with a collection of mathematical functions for array operations.
2. **SciPy**: Provides advanced scientific computing tools like optimization, integration, and signal processing.
3. **Statsmodels**: Focuses on statistical modeling and hypothesis testing, offering tools for regression and time series analysis.
4. **Pandas**: Ideal for data manipulation and analysis, particularly for tabular data using DataFrames.
5. Matplotlib: A library for creating static, interactive, and animated visualizations in Python.
6. Keras: A high-level neural networks API for building and training deep learning models, often used with TensorFlow.
7. Scikit-Learn: Provides tools for machine learning tasks, including classification, regression, clustering, and dimensionality reduction.
8. PyTorch: A deep learning framework known for its dynamic computation graph and strong GPU support.
9. Scrapy: A robust framework for web scraping and extracting data from websites.
10. BeautifulSoup: A library for parsing HTML and XML documents, commonly used for web scraping.
11. TensorFlow: An open-source platform for machine learning, particularly deep learning and neural networks.

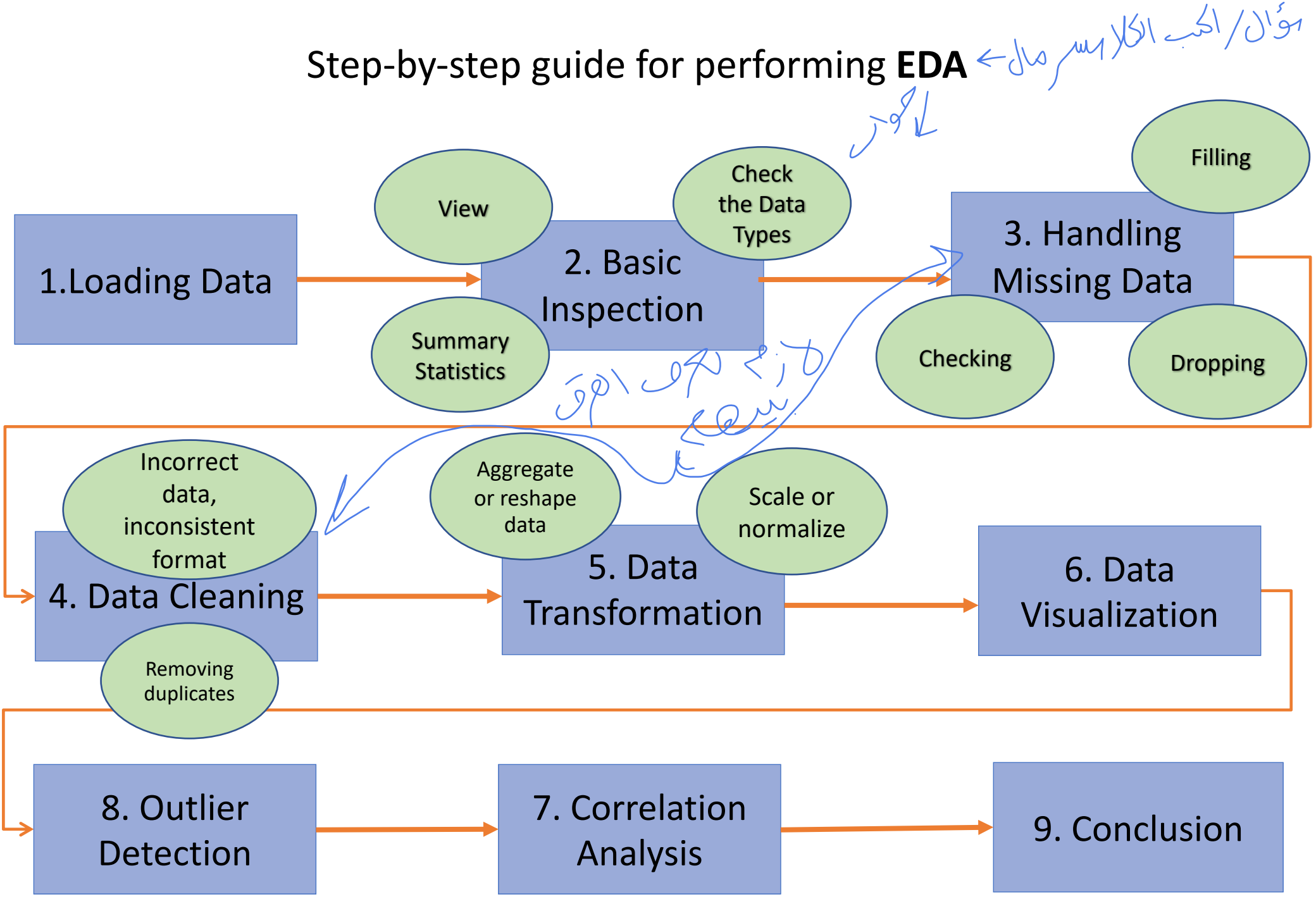
Exploratory Data Analysis (EDA) with Pandas

- **Exploratory data analysis (EDA)** is used by data scientists to analyze and investigate data sets and summarize their main characteristics, often employing data visualization methods.

The outcomes of Exploratory Data Analysis (EDA) are:

- Better understanding of the given dataset which helps clean up the data.
- It gives you a clear picture of the features and the relationships between the data.
- Providing guidelines for essential variables and non-essential variables.
- Handling missing values or human error.
- Identifying outliers.
- The EDA process would maximize insights into a dataset.
- It is also helpful when applying machine learning algorithms later on.

Step-by-step guide for performing **EDA**

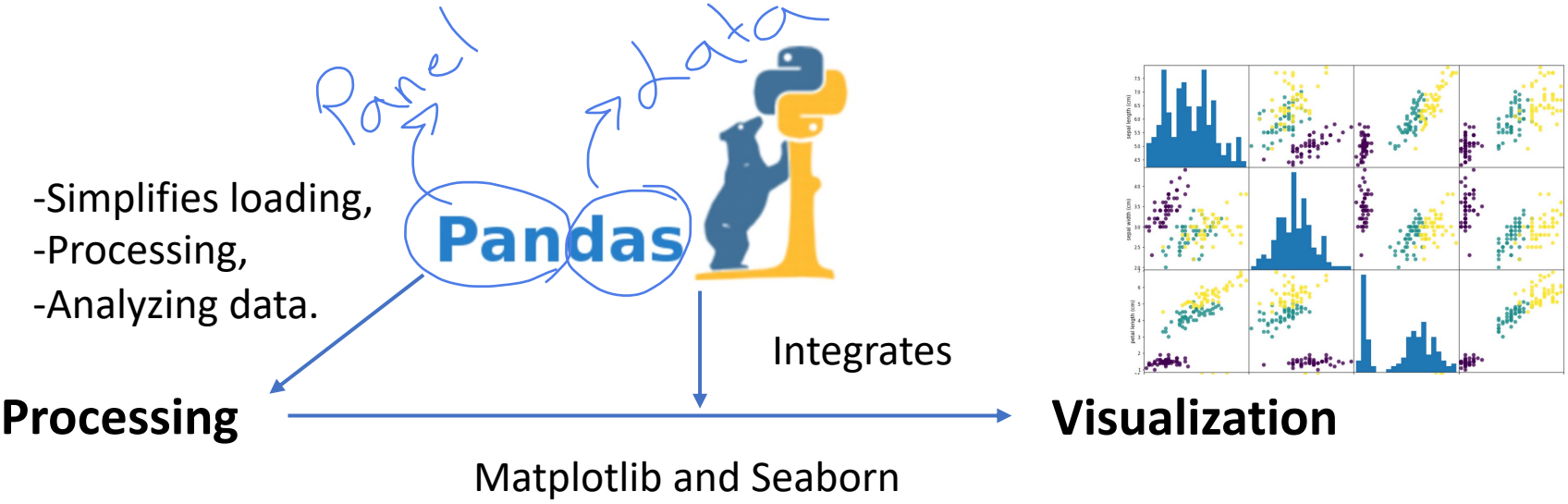


• Introduction to Pandas

Definition: Pandas is a powerful Python library used for data manipulation and analysis, particularly for structured data in tabular formats (e.g., .csv, .tsv, .xlsx).

Primary Features:

- Provides DataFrame and Series objects for easy data handling.
- Offers tools for cleaning, transforming, and analyzing data.
- Allows easy manipulation of large datasets.
- Integrates with other Python libraries for visualization and machine learning.



Advantages of Pandas:

1. **Efficient Data Handling**: Easily handles large datasets with high performance.
2. **Flexible Data Structures**: Provides DataFrame and Series for convenient data manipulation.
3. **SQL-like Operations**: Supports operations like filtering, grouping, and joining, making data analysis simpler.
4. **Integration with Visualization Libraries**: Works well with Matplotlib and Seaborn for graphical data representation.
5. **Wide Range of File Formats Supported**: Supports various file formats like .csv, .xlsx, .json, .sql, and more.

Key Features of Pandas

Data Structures:

- **Series**: A one-dimensional labeled array (like a list or array).
- **DataFrame**: A two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

Functions for Data Handling:

- **Loading data** (e.g., pd.read_csv(), pd.read_excel()).
- **Inspecting data** (head(), tail(), info(), describe()).

Operations on Data:

- **Indexing and slicing.**
- **Filtering, sorting, and grouping.**
- **Handling missing data.**

Pandas is Data Frame

	A	B	C	D	E	F
1	Country	Salesperson	Order Date	OrderID	Units	Order Amount
2	USA	Fuller	1/01/2011	10392	13	1,440.00
3	UK	Gloucester	2/01/2011	10397	17	716.72
4	UK	Bromley	2/01/2011	10771	18	344.00
5	USA	Finchley	3/01/2011	10393	16	2,556.95
6	USA	Finchley	3/01/2011	10394	10	442.00
7	UK	Gillingham	3/01/2011	10395	9	2,122.92
8	USA	Finchley	6/01/2011	10396	7	1,903.80
9	USA	Callahan	8/01/2011	10399	17	1,765.60
10	USA	Fuller	8/01/2011	10404	7	1,591.25
11	USA	Fuller	9/01/2011	10398	11	2,505.60
12	USA	Coghill	9/01/2011	10403	18	855.01
13	USA	Finchley	10/01/2011	10401	7	3,868.60
14	USA	Callahan	10/01/2011	10402	11	2,713.50
15	UK	Rayleigh	13/01/2011	10406	15	1,830.78
16	USA	Callahan	14/01/2011	10408	10	1,622.40
17	USA	Farnham	14/01/2011	10409	19	319.20
18	USA	Farnham	15/01/2011	10410	16	802.00

max std filter min count

Data Frame:
is a 2 dimensional data structure,
Like;
a 2 dimensional array,
or a table with rows and columns.

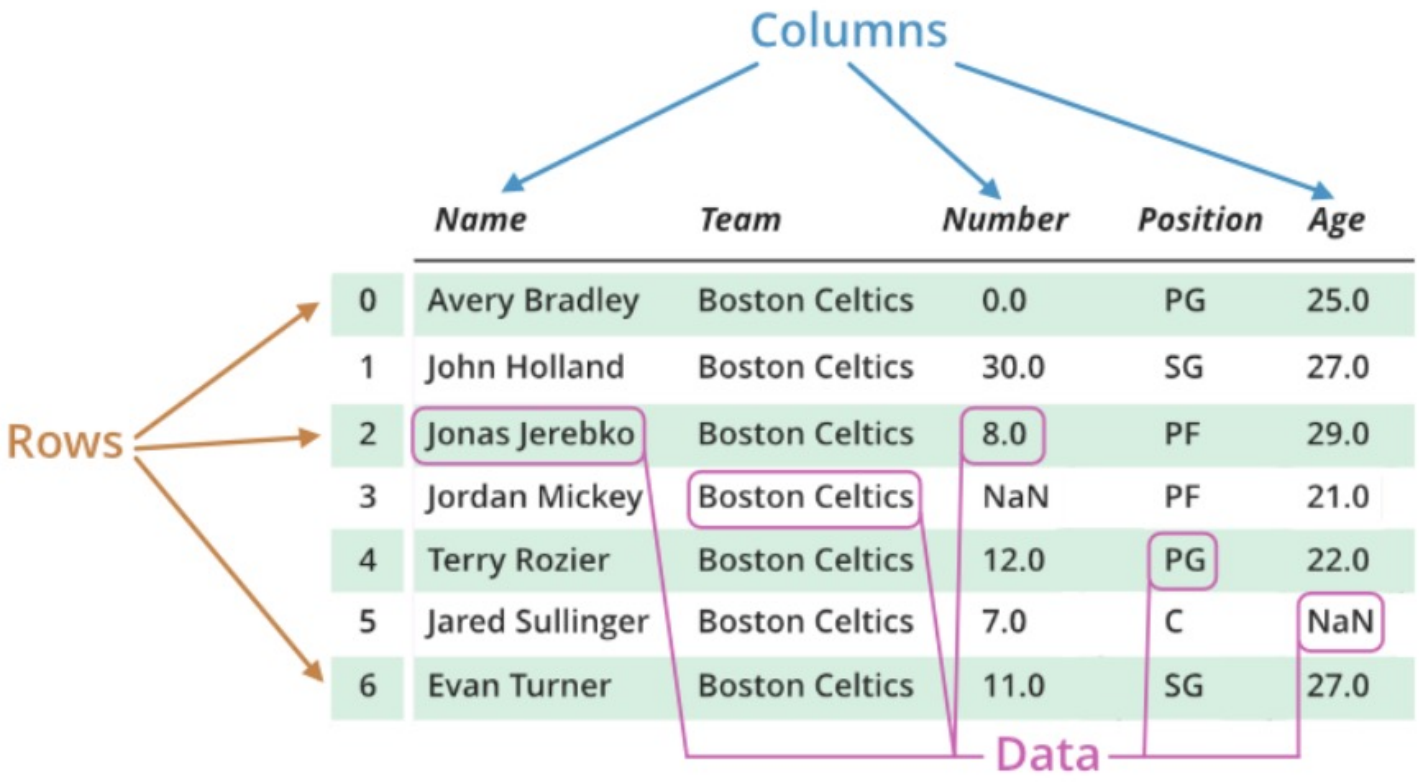
What Can Pandas Do? Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?

* What is average value? * Max value? * Min value?

Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labelled axes (rows and columns).

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the **data**, **rows**, and **columns**.



	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

How to install pandas

- You can run following command to install pandas.
  - `pip install pandas`
  - OR
  - `conda install pandas`
- After installation you can check version of pandas by running the following command at python script mode.
  - `Import pandas`
  - `pandas.__version__`

Data Structures

- There are 2 data structures in pandas.
  - Series.(One D array)
  - DataFrame

Series

	apples
0	3
1	2
2	0
3	1

+

Series

	oranges
0	0
1	3
2	7
3	2

=

DataFrame

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

Series in Pandas: It is a one-dimensional array holding data of any type.

Series Data Structure

- Series is a one –dimensional array.
- It contain homogeneous data.
- Size is fixed.

10	20	30	40	50	60	70

The Pandas Series can be defined as a one-dimensional array that is capable of storing **various data types**. We can easily convert the list, tuple, and dictionary into series using "**series**' method.

Series Data Structure Cont.

- Series is One Dimensional array to hold different types of values.
  - `pandas.Series( data, index, dtype, copy)`
- **data:** It can be any list, dictionary, or scalar value.
- **index:** The value of the index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default np.arrange(n) will be used.
- **dtype:** It refers to the data type of series.
- **copy:** It is used for copying the data.

pandas.Series(data=None, index=None, dtype=None, name=None, copy=False)

Creating a Series

- We can create Series by using various inputs:
  - Array
  - Dictonary
  - Scalar value

Create a Series in two ways:

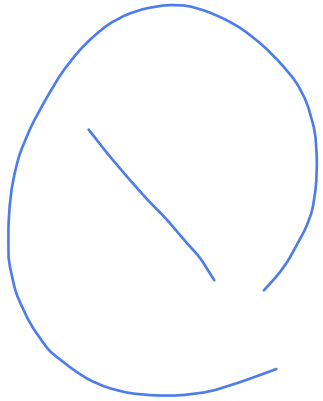
1. Create an empty Series.
2. Create a Series using inputs.

Using Array

- `import pandas as pd`
- `import numpy as np`
- `info = np.array(['R','a','y','s','T','e','c', 'h'])`
- `a = pd.Series(info)`
- `print(a)`

Using Dictionary and scalar Value

- **Using dictionary**
  - `import pandas as pd`
  - `info = {1:'Rays',2:'Tech'}`
  - `a = pd.Series(info)`
  - `print(a)`
- **Using Scalar Value**
  - ```
    import pandas as pd
    a = pd.Series(4,index=[1,2,3,4,5])
    print(a)
    ```
    ```



Create and Load data

Series - DataFrame

```
import pandas as pd

a = [1, 7, 2]

speed = pd.Series(a)

print(speed)
```

```
0    1
1    7
2    2
```

Print(speed[0]) → ?

print(speed.index)
print(speed.values)

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)
```

	calories	duration
0	420	50
1	380	40
2	390	45

Print(df[0]) → ?

DataFrame

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)
```

Locate Row

```
print(df.loc[0])

print(df.loc[[0, 1]])
```

```
   calories  duration
0       420        50
1       380        40
2       390        45
```

Create Labels (Rename Index)

With the index argument, you can name your own labels.

```
import pandas as pd

a = [1, 7, 2]

result = pd.Series(a)

print(result)
```

Create Index

```
import pandas as pd

a = [1, 7, 2]

result = pd.Series(a, index = ["x", "y", "z"])

print(result)

print(result["y"])
```

Assign Labels

With the index argument, you can name your own indexes.

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)

print(df.loc["day2"])
```

```
      calories  duration
day1       420        50
day2       380        40
day3       390        45
```

Difference Between loc[] vs iloc[] in pandas DataFrame

- loc[] is used to select rows and columns by Names/Labels
- iloc[] is used to select rows and columns by **Integer Index/Position**. **zero** based index position.

Example

	Courses	Fee	Duration	Discount
0	Spark	20000	30day	1000
1	PySpark	25000	40days	2300
2	Hadoop	26000	35days	1200
3	Python	22000	40days	2500
4	pandas	24000	60days	2000

	Courses	Fee	Duration	Discount
r1	Spark	20000	30day	1000
r2	PySpark	25000	40days	2300
r3	Hadoop	26000	35days	1200
r4	Python	22000	40days	2500
r5	pandas	24000	60days	2000

```
import pandas as pd
list = {
   'Courses':["Spark","PySpark","Hadoop","Python","pandas"],
   'Fee' :[20000,25000,26000,22000,24000],
   'Duration':['30day','40days','35days','40days','60days'],
   'Discount':[1000,2300,1200,2500,2000]
   }
df=pd.DataFrame(list)
print(df)
index_list =['r1','r2','r3','r4','r5']
df=pd.DataFrame(list, index = index_list)
print(df)
```

	loc[] - By Label	iloc[] - By Index
Select Single Row	df.loc[‘r2’]	df.iloc[1]
Select Single Column	df.loc[:, “Courses”]	df.iloc[:, 0]
Select Multiple Rows	df.loc[[‘r2’,’r3’]]	df.iloc[[1,2]]
Select Multiple Columns	df.loc[:, [“Courses”,”Fee”]]	df.iloc[:, [0,1]]
Select Rows Range	df.loc[‘r1’:’r4’]	df.iloc[0:4]
Select Columns Range	df.loc[:,’Fee’:’Discount’]	df.iloc[:,1:4]
Select Alternate Rows	df.loc[‘r1’:’r4’:1]	df.iloc[0:4:1]
Select Alternate Columns	df.loc[:,’Fee’:’Discount’:1]	df.iloc[:,1:4:1]
Using Condition	df.loc[df['Fee'] >= 24000]	df.iloc[list(df['Fee'] >= 24000)]

Select Single Value
Select Multiple Values
Select Range of Values
Select Alternate Rows & Columns
Using Conditions

Read details:
https://sparkbyexamples.com/pandas/pandas-difference-between-loc-vs-iloc-in-dataframe/

```
print(df.loc['r2'])
print(df.iloc[1])
```

```
print(df.loc[:, "Courses"])
print(df.iloc[:, 0])
```

```
print(df.loc[:, ["Courses","Fee","Discount"]])
print(df.iloc[:, [0,1,3]])
```

```
print(df.loc[['r2','r3']])
print(df.iloc[[1,2]])
```

```
print(df.iloc[list(df['Fee'] >= 24000)])
print(df.loc[df['Fee'] >= 24000])
```

```
print(df.loc[:,'Fee':'Discount'])
print(df.iloc[:,1:4])
```

	Courses	Fee	Duration	Discount
r1	Spark	20000	30day	1000
r2	PySpark	25000	40days	2300
r3	Hadoop	26000	35days	1200
r4	Python	22000	40days	2500
r5	pandas	24000	60days	2000

Importing Data:

- pd.read_csv(filename) : It read the data from CSV file.
- pd.read_table(filename) : It is used to read the data from delimited text file.
- pd.read_excel(filename) : It read the data from an Excel file.
- pd.read_sql(query,connection _object) : It read the data from a SQL table/database.
- pd.read_json(json _string) : It read the data from a JSON formatted string, URL or file.

Load Files Into a DataFrame

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df)
```

If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.to_string())
```

use **to_string()** to print the entire DataFrame.

Why does Pandas only display the first 5 and the last 5 rows of a large DataFrame by default, and how can you modify this behavior to view more rows?

Pandas Read JSON

you have a JSON in a string, you can read or load this into pandas DataFrame using read_json() function.

```
import pandas as pd
# Read json from String
json_str = '{"Courses":{"r1":"Spark"},"Fee":{"r1":"25000"},"Duration":{"r1":"50 Days"}}'
df = pd.read_json(json_str)
print(df)
```

```
import pandas as pd
df = pd.read_json('data.json')
print(df.to_string())
```

JSON stands for **J**ava**S**cript **O**bject **N**otation

Examples:

```
data_array = [
    ["ID", "Name", "Age", "City", "Salary"],
    [1, "Alice", 28, "New York", 70000],
    [2, "Bob", 34, "Los Angeles", 80000],
    [3, "Charlie", 25, "Chicago", 60000],
    [4, "David", 42, "Houston", 90000],
    [5, "Eve", 30, "Phoenix", 75000]
]
```

print(data_array) → what will be the result ?

Convert array to a DataFrame

```
df = pd.DataFrame(data_array[1:], columns=data_array[0])
```

Save as CSV

```
df.to_csv("your_data.csv", index=False)
```

Read the CSV

```
data = pd.read_csv("your_data.csv")
```

print(data)

Inspection methods

Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the **head() method**. The head() method returns the headers and a specified number of rows, starting from the top. here is also a **tail() method** for viewing the *last* rows of the DataFrame.

df.head()

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head())
```

df.tail()

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.tail())
```

df.head(7)

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
5	60	102	127	300.5
6	60	110	136	374.0

df.tail(7)

	Duration	Pulse	Maxpulse	Calories
162	45	95	130	270.0
163	45	100	140	280.9
164	60	105	140	290.8
165	60	110	145	300.4
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

Information on the Data

df.info() # Provides summary info about the DataFrame (data types and missing values)

Example

```
import pandas as pd

df = pd.read_csv('data.csv’)

print(df.info())

print(df.head(3))

print(df.tail(3))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 5.4 KB
None
   Duration  Pulse  Maxpulse  Calories
0        60    110       130     409.1
1        60    117       145     479.0
2        60    103       135     340.0
     Duration  Pulse  Maxpulse  Calories
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4
```

info() method also tells us how many Non-Null values there are present in each column

Object Attribute
(Check the Data Types and Null Values)

```
import pandas as pd

df = pd.read_csv('data.csv')

print("type::", df.dtypes)

print("shape::",df.shape)

print("size::", df.size)

print("index::", df.index)

print("dimension::", df.ndim)

print("Array values::" , df.values)
```

```
type:: Duration      int64
Pulse         int64
Maxpulse      int64
Calories    float64
dtype: object
shape:: (169, 4)
size:: 676
index:: RangeIndex(start=0, stop=169, step=1)
dim:: 2
values:: [[  60.   110.   130.   409.1]
 [  60.   117.   145.   479. ]
 [  60.   103.   135.   340. ]
 [  45.   109.   175.   282.4]
 [  45.   117.   148.   406. ]
 [  60.   102.   127.   300.5]
 [  60.   110.   136.   374. ]
```

Summary Statistics on the Data

Quickly summarize numeric columns (mean, standard deviation, min, max, etc.)

- df.describe() # Provides statistical summary for numeric columns

```
import pandas as pd

data = {
    'Height': [150, 160, 165, 170, 175],
    'Weight': [60, 70, 65, 80, 85],
    'Age': [23, 34, 29, 41, 25]
}

df = pd.DataFrame(data)

summary = df.describe()

print(summary)
```

	Height	Weight	Age
count	5.000000	5.0	5.0
mean	164.000000	72.0	30.4
std	9.054891	8.0	7.1
min	150.000000	60.0	23.0
25%	160.000000	65.0	25.0
50%	165.000000	70.0	29.0
75%	170.000000	80.0	34.0
max	175.000000	85.0	41.0

Commands and Samples

1. View the first or last rows

df.head() Show first 5 rows (default)
df.head(10) Show first 10 rows
df.tail() Show last 5 rows (default)
df.tail(10) Show last 10 rows

2. View general information about the dataset

df.info() Overview: number of rows, columns, non-null count, data types
df.shape Returns (rows, columns)
df.columns List of column names
df.dtypes Data type of each column

3. Summary statistics

df.describe() Summary of numerical columns: count, mean, std, min, 25%, 50%, 75%, max
df.describe(include='all') Summary including categorical columns

Commands and Samples

4. Access specific rows or columns

df['ColumnName']	Access a single column
df[['Col1','Col2']]	Access multiple columns
df.iloc[0]	Access first row by index
df.iloc[0:5]	Access first 5 rows by index
df.loc[0]	Access first row by label
df.loc[:, ['Col1','Col2']]	Access all rows but specific columns

5. Check for missing or null values

df.isnull()	Returns True/False for each cell
df.isnull().sum()	Count of missing values per column
df.notnull()	Opposite of isnull

6. Check duplicates

df.duplicated()	Returns True/False if row is duplicate
df.duplicated().sum()	Count of duplicate rows

Commands and Samples

7. Check unique values

df['ColumnName'].unique()	Array of unique values
df['ColumnName'].nunique()	Number of unique values
df['ColumnName'].value_counts()	Frequency of each unique value

8. Sorting data

df.sort_values('ColumnName')	Sort by column ascending
df.sort_values('ColumnName', ascending=False)	Descending

9. Sample of data

df.sample(5)	Random 5 rows

10. Transpose data

df.T	Swap rows and columns

df_transposed = df.T

Cleaning Data

Data Transformation

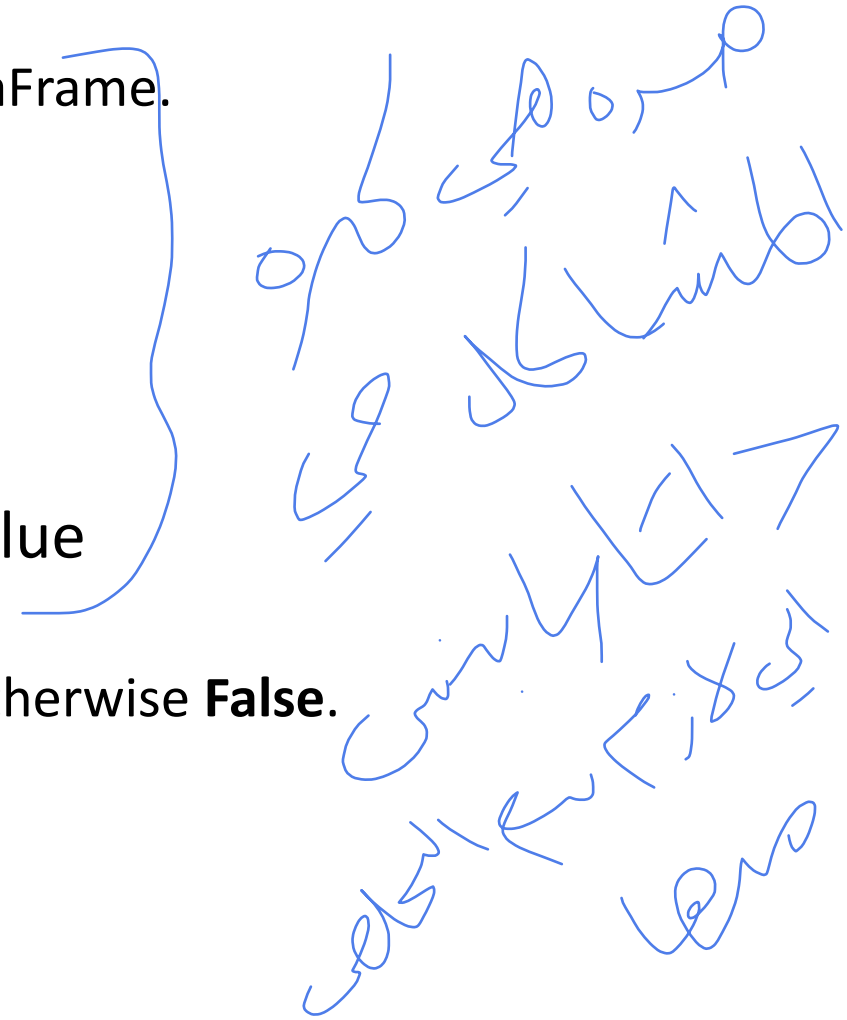
Handling missing values

What are Null Values?

Null values represent **missing data** in a DataFrame.

Types of null values:

- NaN → Missing numeric values
- NaT → Missing datetime values
- None → Generic Python missing value



df.isnull() → Returns **True** if a cell is null, otherwise **False**.

df.isnull().sum()

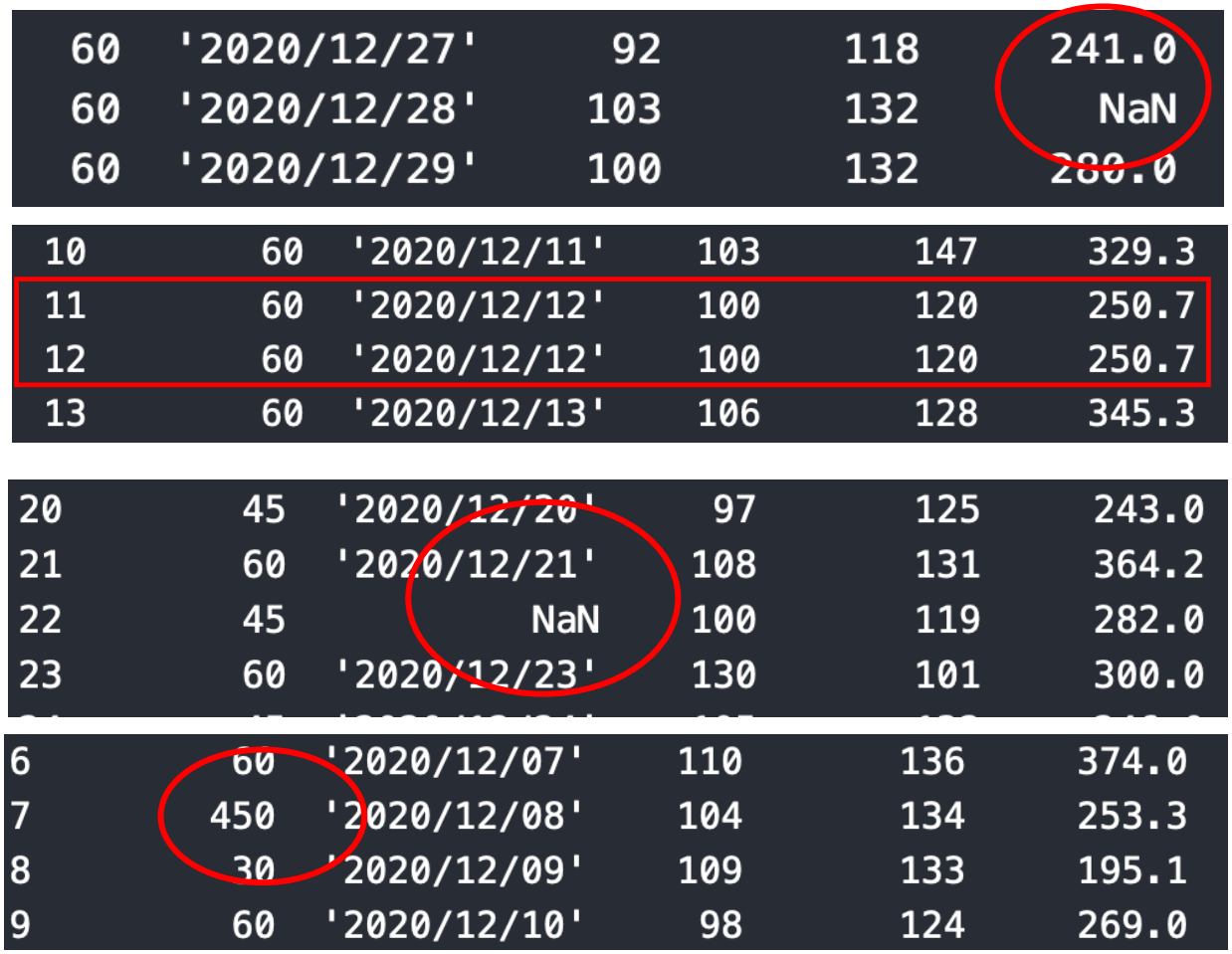
df[df.isnull().any(axis=1)]

notnull() → opposite of isnull(), shows **only non-missing values**.

Data Cleaning

Data cleaning means fixing bad data in your data set.

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates



60	'2020/12/27'	92		118	241.0
60	'2020/12/28'	103		132	NaN
60	'2020/12/29'	100		132	280.0

10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3

20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0

6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0

Pandas - Cleaning Empty Cells

- ***Empty cells can potentially give you a wrong result when you analyze data.***

1- Remove Rows

One way to deal with empty cells is to **remove rows** that contain **empty cells**. This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.

dropna() removes rows or columns with NaN values.

Use axis=0 (default) to drop rows with null values.

Use axis=1 to drop columns with null values.

```
import pandas as pd
df = pd.read_csv('data.csv')
new_df = df.dropna()
Print(new_df)
```

Delete null values from all columns/rows

- In Pandas, when performing operations that modify the original DataFrame (such as dropping rows or columns, or filling missing values), you can use the inplace=True argument to apply the changes directly to the original DataFrame without needing to reassign it to a new variable.
- **inplace=True** modifies the original DataFrame and doesn't return a new object.
- If **inplace=False** (or if the argument is not provided), the operation will return a new DataFrame, and you need to assign the result to a variable.

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'City': ['NY', 'LA', 'SF', 'TX']}
df = pd.DataFrame(data)
df.drop('City', axis=1, inplace=True)
print(df)
```

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
3    David   40
```

2- Remove and replace Empty Values

```
import pandas as pd
df = pd.read_csv('data.csv')
df.dropna(inplace = True)
print(df.to_string())
```

Removing null values from all columns, the rows will be deleted

```
import pandas as pd

# Sample DataFrame with space values

data = {'Name': ['Alice', 'Bob', ' ', 'David'],

        'City': ['NY', ' ', 'LA', 'TX']}

df = pd.DataFrame(data)

df_filled = df.replace(' ', 'Unknown')

print(df_filled)
```

	Name	City
0	Alice	NY
1	Bob	Unknown
2	Unknown	LA
3	David	TX

3- Replace Only For Specified Columns

```
import pandas as pd
df = pd.read_csv('data.csv')
df["Calories"].fillna(130, inplace = True)
```

Specify column, fill row's value

```
import pandas as pd

import numpy as np

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

        'Age': [25, np.nan, 35, np.nan],

        'City': ['NY', np.nan, 'LA', 'TX']}

df = pd.DataFrame(data)

df_filled = df.fillna({'Age': 0, 'City': 'Unknown'})

print(df_filled)
```

	Name	Age	City
0	Alice	25.0	NY
1	Bob	NaN	NaN
2	Charlie	35.0	LA
3	David	NaN	TX

	Name	Age	City
0	Alice	25.0	NY
1	Bob	0.0	Unknown
2	Charlie	35.0	LA
3	David	0.0	TX

4- Replace Using Mean, Median, or Mode

Pandas uses the mean() median() and mode() methods to calculate the respective values for a specified column:

Calculate the MEAN, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mean()

print( df["Calories"].fillna(x, inplace = True))
```

Calculate the MEDIAN, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].median()

df["Calories"].fillna(x, inplace = True)
```

Calculate the MODE, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mode()[0]

df["Calories"].fillna(x, inplace = True)
```

mode() is a pandas method that returns the **mode** of the specified column. The mode is the most frequently occurring value in the column.

Since mode() returns a **Series** (in case there are multiple modes), we use **[0]** to select the first mode (the most frequent value).

Pandas - Cleaning Data of Wrong Format

Convert all cells in the 'Date' column into standard date.

```
import pandas as pd
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

21	60	'2020/12/21'	108
22	45	NaN	100
23	60	'2020/12/23'	130
24	45	'2020/12/24'	105
25	60	'2020/12/25'	102
26	60	20201226	100
27	60	'2020/12/27'	92

pd.to_datetime() converts the "Date" column into a proper datetime format.

21	60	'2020/12/21'	108
22	45	NaT	100
23	60	'2020/12/23'	130
24	45	'2020/12/24'	105
25	60	'2020/12/25'	102
26	60	'2020/12/26'	100

Removing Rows (NaT)

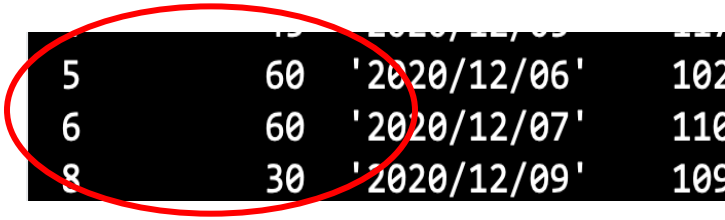
```
df.dropna(subset=['Date'], inplace = True)
```

Pandas - Fixing Wrong Data

6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)
```

Removing Rows



5	60	'2020/12/06'	102	
6	60	'2020/12/07'	110	
8	30	'2020/12/09'	109	

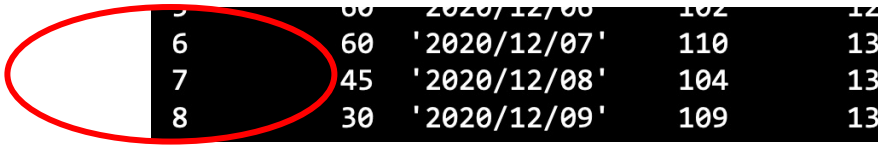
```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
```

Replacing Values

set some boundaries for legal values, and replace any values that are outside of the boundaries.

```
df.loc[7, 'Duration'] = 45
```

Replacing Values



6	60	'2020/12/07'	110	13
7	45	'2020/12/08'	104	13
8	30	'2020/12/09'	109	13

Pandas - Removing Duplicates

To discover duplicates, we can use the duplicated() method.

The duplicated() method returns a Boolean values for each row.

Returns True for every row that is a duplicate, otherwise False.

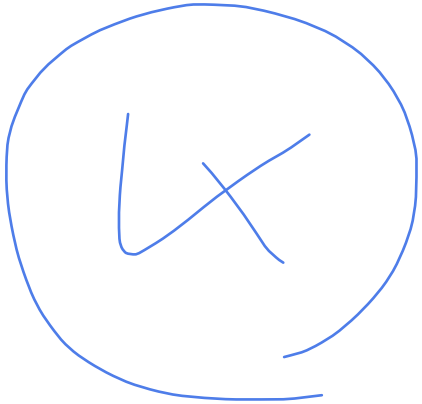
```
print(df.duplicated())
```
Find Duplicate values

```
df1= df.drop_duplicates()
```
Remove all Duplicate values

```
Print(df1)
```

```
df2.drop_duplicates(inplace = True)
```
Remove all duplicates in original data frame

```
Print(df2)
```



Data Visualization

- **Data visualization** plays a crucial role in **Exploratory Data Analysis (EDA)** as it helps data scientists to **better understand** and **interpret** the underlying patterns and relationships in the data. It is used to support data exploration, cleaning, and preparation tasks.

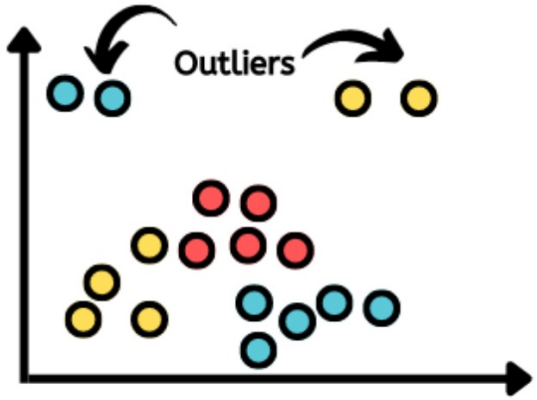
- **Why data visualization is important in the EDA process:**

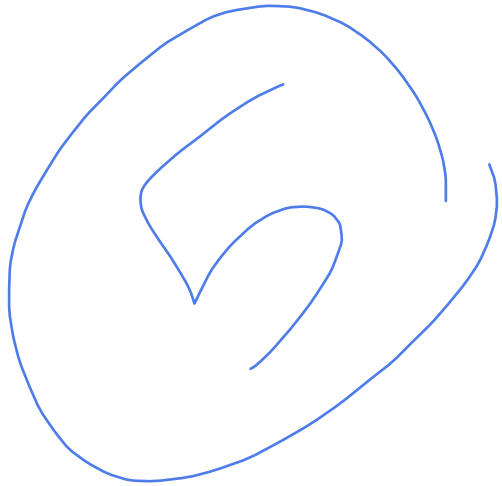
1-Understanding the structure and relationships in data.

2-Detecting issues such as missing values and outliers.

3-Guiding decisions about data preprocessing, modeling, and analysis.

4-Providing insights that improve the accuracy and effectiveness of analyses.





Oulier Detection

Oulier Detection (Interquartile Range)

```
import pandas as pd
data = {'Age': [25, 30, 35, 40, 100, 45, 50, 60, 65, 150]}
df = pd.DataFrame(data)

#Calculate Q1 (25th percentile) and Q3 (75th percentile)
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1

#Define the lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

#Identify outliers
outliers = df[(df['Age'] < lower_bound) | (df['Age'] > upper_bound)]

print(outliers)
```

Oulier Detection (Using Z-Score)

```
import pandas as pd
from scipy import stats
data = {'Age': [25, 30, 35, 40, 100, 45, 50, 60, 65, 150]}
df = pd.DataFrame(data)

# Calculate Z-scores
z_scores = stats.zscore(df['Age’])

outliers = df[abs(z_scores) > 3]
print(outliers)
```

Oulier Detection (Using Percentiles for Thresholding)

```
import pandas as pd

data = {'Age': [25, 30, 35, 40, 100, 45, 50, 60, 65, 150]}
df = pd.DataFrame(data)

# Calculate the 95th percentile
percentile_95 = df['Age'].quantile(0.95)

# Identify outliers (values above the 95th percentile)
outliers = df[df['Age'] > percentile_95]


print(outliers)
```

Data Correlations

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.corr())
```

```
              Duration     Pulse  Maxpulse  Calories
Duration      1.000000 -0.155408  0.009403  0.922721
Pulse        -0.155408  1.000000  0.786535  0.025120
Maxpulse      0.009403  0.786535  1.000000  0.203814
Calories      0.922721  0.025120  0.203814  1.000000
```

Perfect Correlation:

We can see that "Duration" and "Duration" got the number 1.000000, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:

"Duration" and "Calories" got a 0.922721 correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

Bad Correlation:

"Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.

Students should provide all the Pandas commands

- pandas– where()
- pandas – replace()
- pandas – fillna()
- pandas – dropna()
- pandas – append()
- pandas – sort_values()
- pandas – sort_index()
- pandas – drop()
- pandas – mean()
- pandas – drop_duplicates()
- pandas – corr()
- pandas – mean()
- Padnas – to_datetime()
- pandas – head()
- ……

Students should be ready to answer

- Know **all commands for viewing and inspecting data**.
- Be able to **detect and handle missing data, duplicates, and outliers**.
- Be able to **transform and engineer features** for model input.
- Understand **data distributions, skewness, and correlation**.
- Be ready to **visualize data** for insights.
- Be capable of **preparing datasets** for machine learning models.

What You Learned

End of Class 5