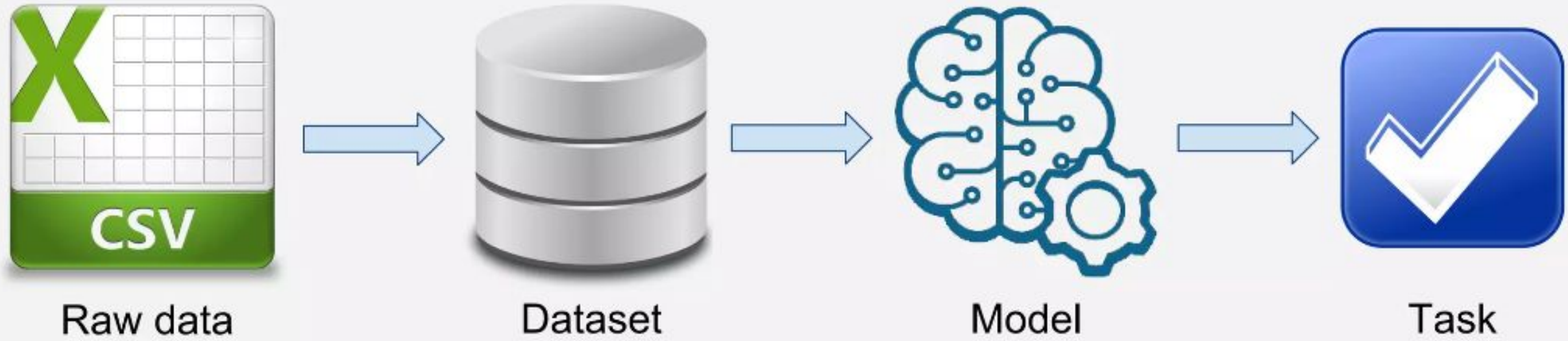
رواد مصر الرقمية

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

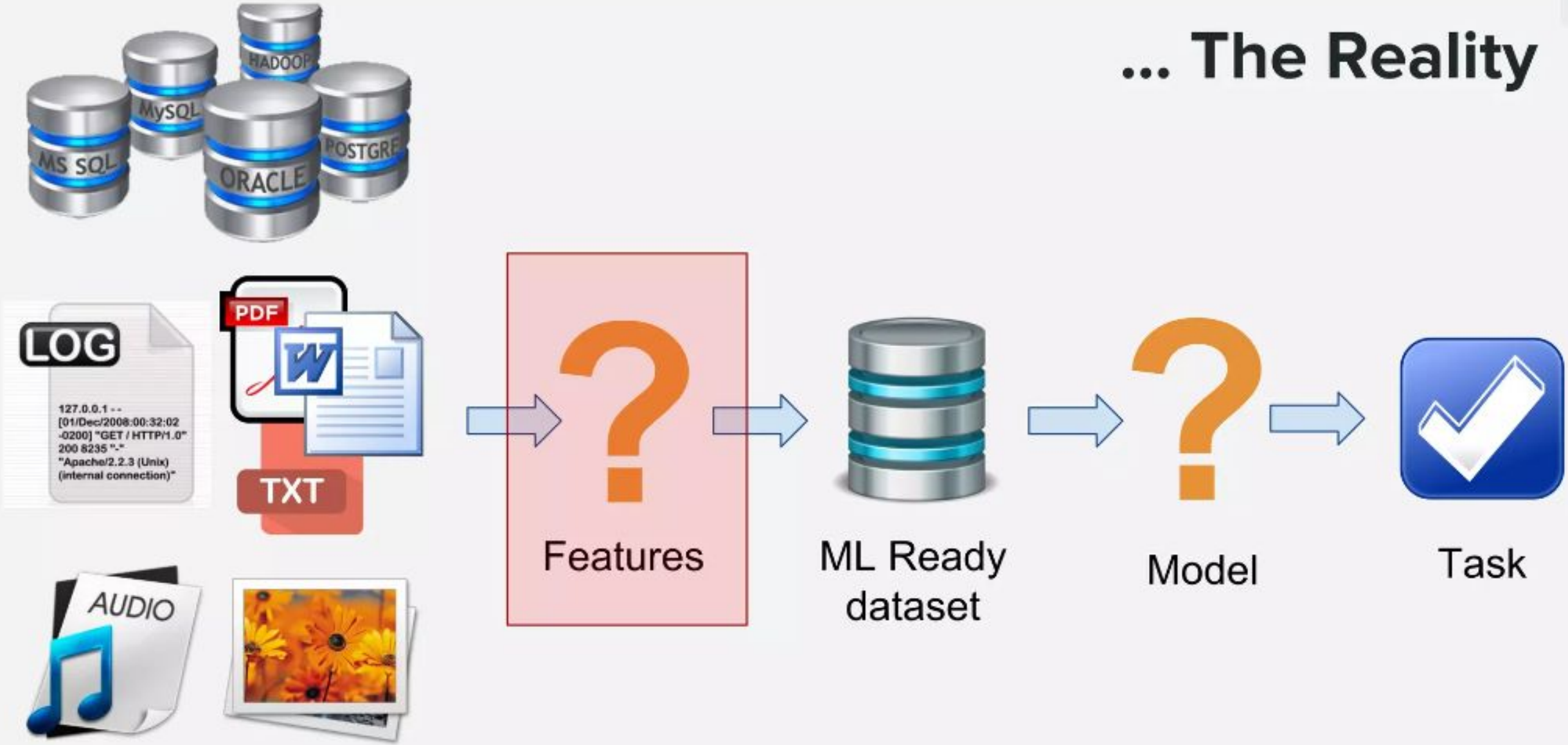
Data Preprocessing & Visualization

Session : (Pandas)

The Dream...



... The Reality


HADOOP
MySQL
MS SQL
ORACLE
POSTGRE
LOG
127.0.0.1 - -
[01/Dec/2008:00:32:02
-0200] "GET / HTTP/1.0"
200 8235 "-"
"Apache/2.2.3 (Unix)
(internal connection)"
PDF
TXT
AUDIO
Features
ML Ready
dataset
Model
Task

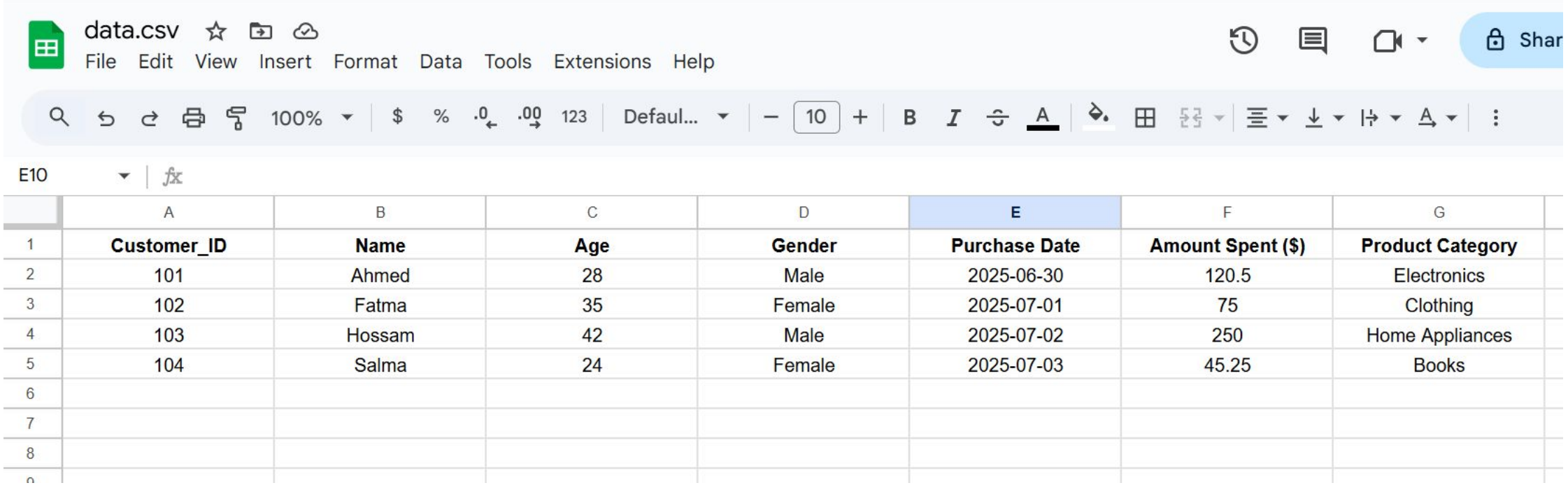
What is Data?

Data is any kind of raw information collected from the real world.

It can be numbers, text, images, or signals — anything that helps us describe or understand a situation.

Why is Data Important?

On its own, data is just raw numbers or facts.
But when we **organize, analyze, and interpret** it — we can extract **valuable insights**, solve problems, and make better decisions.

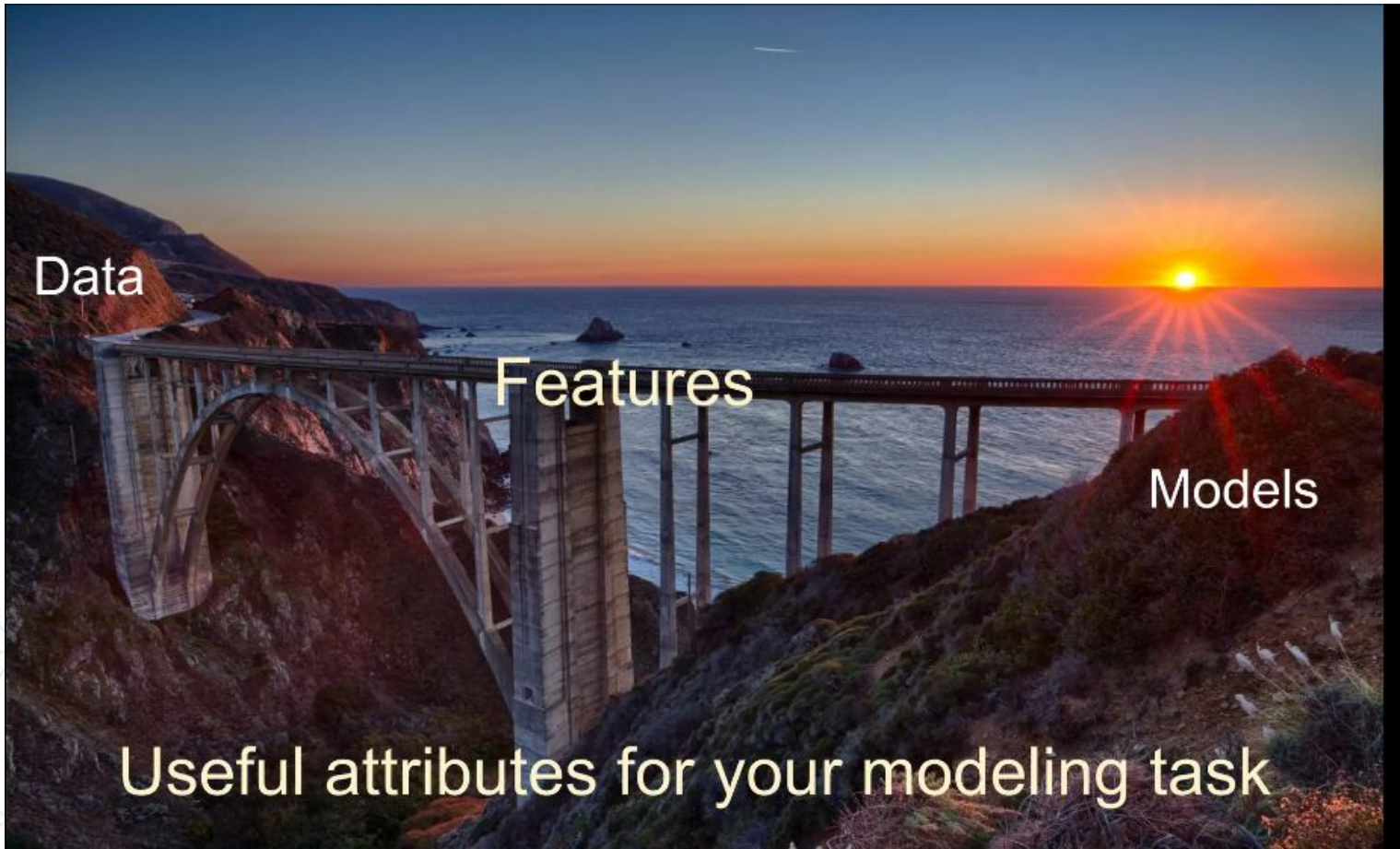


	A	B	C	D	E	F	G
1	**Customer_ID**	**Name**	**Age**	**Gender**	**Purchase Date**	**Amount Spent ($)**	**Product Category**
2	101	Ahmed	28	Male	2025-06-30	120.5	Electronics
3	102	Fatma	35	Female	2025-07-01	75	Clothing
4	103	Hossam	42	Male	2025-07-02	250	Home Appliances
5	104	Salma	24	Female	2025-07-03	45.25	Books
6							
7							
8							

“At the end of the day, some machine learning projects succeed and some fail. What makes the difference?

Easily the most important factor is the features used.”

— Prof. Pedro Domingos



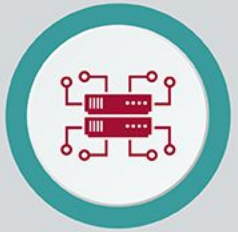
Data Wrangling vs. Data Cleaning

Data cleaning and wrangling takes up 80% of the data science workflow

DATA CLEANING

Removing inaccurate and inconsistent data

DATA WRANGLING



Transforming raw data into a more usable form

Quality over quantity

Harvard Business School Online

🧹 Data Cleaning:

"Think of it like cleaning your room before organizing it.
Data Cleaning is all about making sure the data is:

- **Correct**
- **Consistent**
- **Free of errors**

Typical tasks include:

- Handling **missing values**
- Fixing **incorrect entries**
- Standardizing formats (e.g., 'Yes', 'yes', 'Y' → all become 'Yes')

The goal is to make your dataset trustworthy."

Data Wrangling:

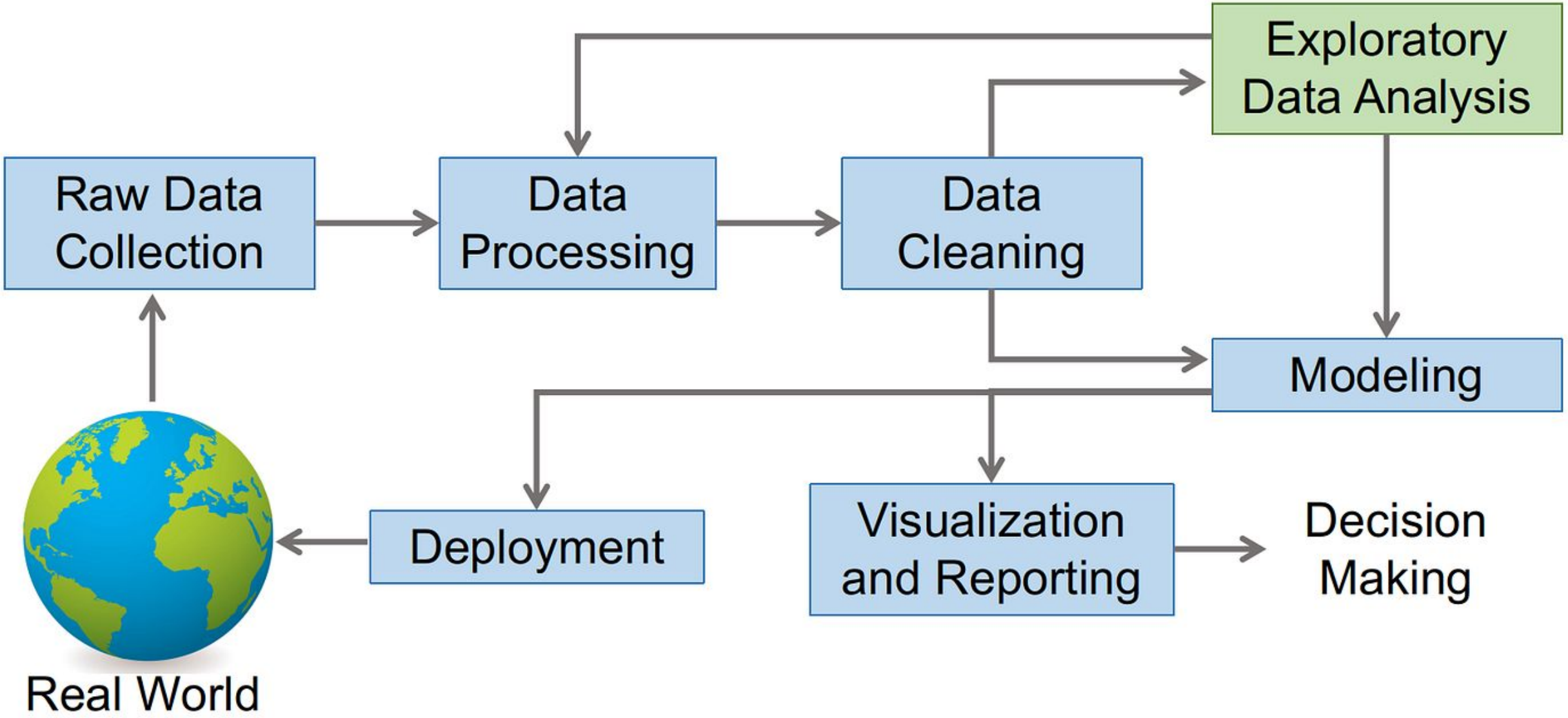
"Now that your data is clean, you move on to preparing it for analysis or modeling — this is Wrangling.
Data Wrangling is about **transforming**, **reshaping**, and **organizing** your data.

It includes tasks like:

- Converting data types (e.g., string to datetime)
- Merging datasets
- Renaming or reordering columns
- Creating new columns or dropping irrelevant ones
- Reshaping data using techniques

In short:
Cleaning = Fixing the data
Wrangling = Reshaping the data

Data Science Process



Data Preprocessing

Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- Work with Categorical data
- Detect and Handle Outliers
- Split data to Train and Test Sets
- Deal with Imbalanced classes
- Feature Scaling

Feature Engineering and Extraction

- Domain knowledge features (age , wh, **BMI**)
- Date and Time features
- String operations
- Web Data
- Geospatial features

Eng: Baraa slides

First at all ... a closer look at your data

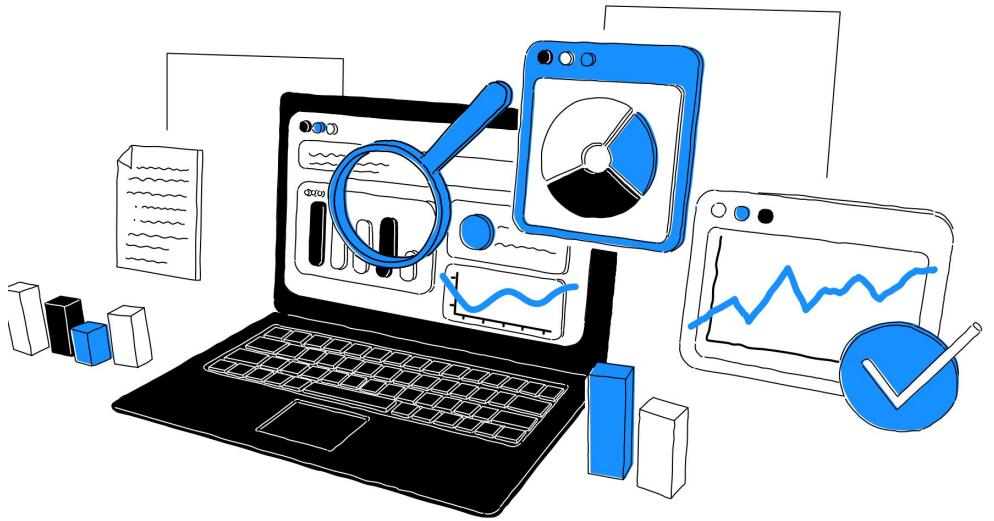
- What does the data model look like?
- What is the features distribution?
- What are the features with missing or inconsistent values?
- What are the most predictive features?

- Conduct a Exploratory Data Analysis (EDA)

رواد مصر الرقمية

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

Exploratory Data Analysis (EDA)



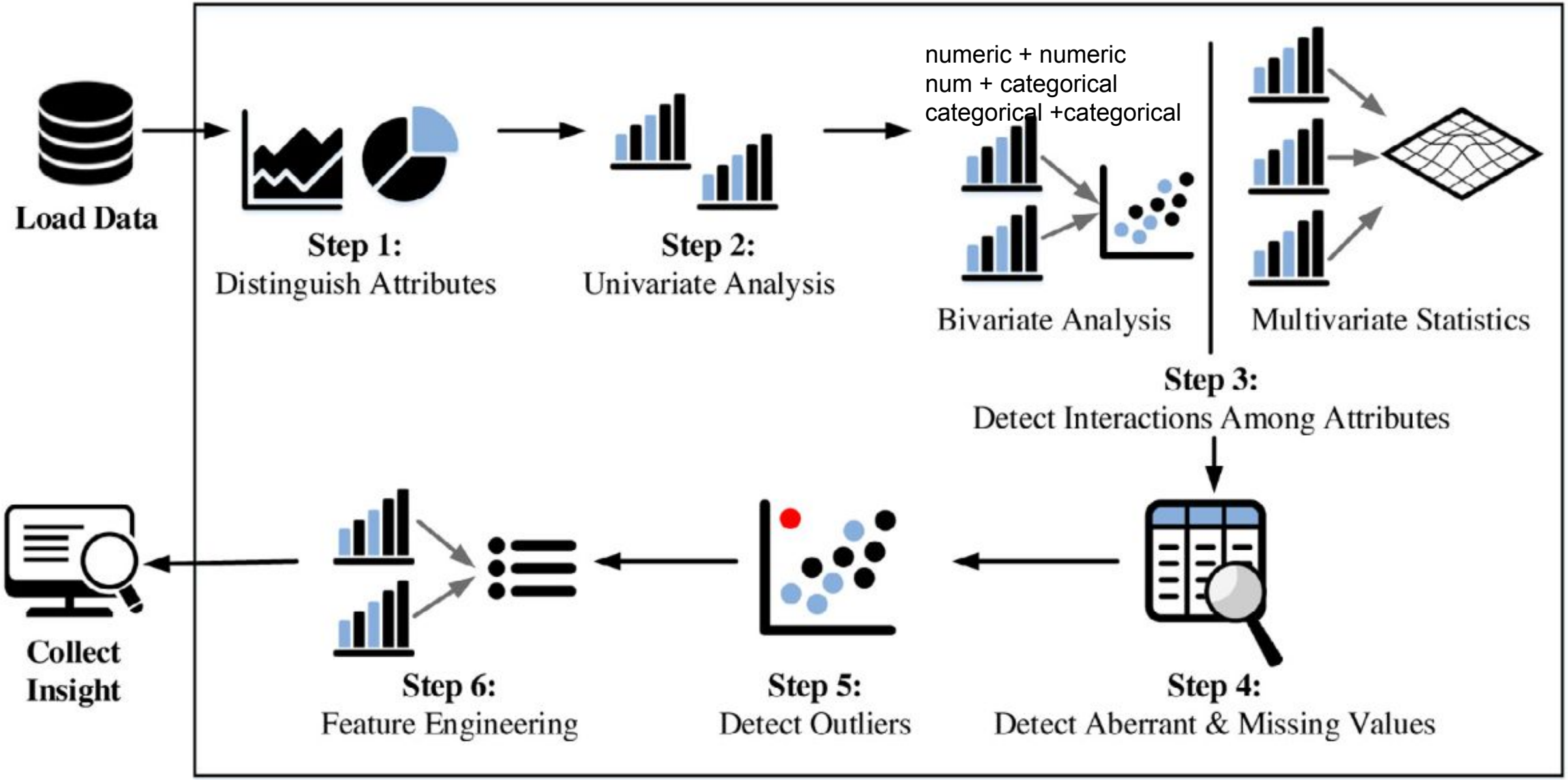
Exploratory Data Analysis (EDA)

is a crucial step in the machine learning process that involves visually and statistically analyzing the data

to gain insights, understand its characteristics, and identify patterns or trends,

And to make informed decisions about data preprocessing, feature engineering, and model selection

Exploratory Data Analysis overview



Why EDA is important in the machine learning process?

In general, EDA is a critical step in the machine learning process as it helps in :

- Understanding the data
- Assessing data quality
- Guiding feature engineering
- Informing model selection
- Effectively communicating results

Common steps for EDA:

1. Data Collection:
    - Load and import the dataset into a DataFrame for analysis.
    - Using Python libraries such as Pandas, NumPy

2. Data Inspection:
    - Get an overview of the data, check for missing values, data types, and basic statistics.
    - Using DataFrame methods like info(), head(), tail(), and describe()

Common steps for EDA:

3. Data Cleaning and wrangling:
    - Imputing missing values, removing or correcting outliers, and addressing any data quality issues that may impact the accuracy of the analysis.
    - Using Pandas and NumPy libraries:
        - For example, you can use dropna(), fillna(), or interpolate() to handle missing values, and use visualizations to detect and handle outliers.

Common steps for EDA:

4. Data Visualization:
    - Create visualizations for data exploration to explore the data and identify patterns, trends, and relationships that help in gaining insights from the data.
    - Using Matplotlib, Seaborn, or Plotly libraries:
        - You can create various types of plots such as bar charts, histograms, scatter plots, box plots, heatmaps, and more to visualize the data and identify patterns, trends, and relationships.

Common steps for EDA:

5. Data Descriptive Statistics:
    - Calculate descriptive statistics for the data and gain a quantitative understanding of its distribution and characteristics such as mean, median, mode, standard deviation, and other relevant measures.
    - Using Pandas methods like mean(), median(), mode(), std(), and var()

Common steps for EDA:

6. Feature Analysis:
    - Analyze individual features (columns) in the dataset to understand their distributions, relationships with the target variable (if applicable), and identify any potential interactions or correlations.
    - Using libraries like SciPy or Statsmodels for statistical tests and Matplotlib or Seaborn for visualizations.

Common steps for EDA:

7. Feature Engineering
    - Improving the performance of the machine learning models using domain knowledge and insights gained from EDA to perform feature engineering, which may involve:
        - Selecting relevant features
        - Creating new features,
        - Transforming existing features
    - It involves transforming the data in a way that makes it more suitable for modeling, by extracting relevant information, reducing noise, and improving the representation of the underlying patterns in the data.

EDA use case

on clothing reviews

https://www.kaggle.com/code/wonduk/eda-lstm-classification-on-clothing-reviews

More EDA use case
https://www.kaggle.com/wonduk/code

Agenda:

1	Introduction to Pandas
2	Data Manipulation with Pandas
3	Practical Examples

رواد مصر الرقمية

وزارة الاتصالات
وتكنولوجيا المعلومات

Introduction to Pandas

Introduction to Pandas

What is Pandas?

Pandas is an open-source data analysis and data manipulation library built on top of Numpy. It provides high-level data structures and functions designed to make data analysis fast and easy in Python.

Pandas is widely used in data science, finance, statistics, and many other fields for tasks like data cleaning, transformation, and analysis.

Introduction to Pandas

Key Features of Pandas

- **Series and DataFrames**: Pandas introduces two primary data structures: Series (one-dimensional) and DataFrame (two-dimensional), which are used to store and manipulate data.
- **Data Alignment**: Pandas automatically aligns data by its labels, making it easy to perform arithmetic operations on data sets with different indexes.
- **Handling Missing Data**: Pandas provides methods for detecting, filling, and dropping missing data, which is essential for cleaning datasets.
- **Powerful Data Grouping and Aggregation**: Pandas supports grouping data, aggregating results, and applying custom functions using the groupby() functionality.
- **Flexible Data Reading/Writing**: Pandas can read from and write to various file formats, including CSV, Excel, SQL databases, and more.

Pandas

We will use pandas to:

- Read in data from Excel.
- Manipulate data in spreadsheet.
- Visualize data (we will also use another Python package called ggplot to do this).
- Filter and aggregate data from spreadsheet using SQL

Data Manipulation with Pandas

Data manipulation is a core aspect of data analysis. Pandas provides various tools and functions to manipulate data efficiently.

Creating Pandas Data Structures

Creating a Pandas Series

```
import pandas as pd

# Creating a Series from a list
series = pd.Series([10, 20, 30, 40, 50])
print(series)
```

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

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

A Series is a one-dimensional labeled array capable of holding any data type. It is similar to a column in a spreadsheet.

Missing Data

```
pd.read_csv?
```

```
nrows : int, default None
    Number of rows of file to read. Useful for reading pieces of large files
na_values : scalar, str, list-like, or dict, default None
    Additional strings to recognize as NA/NaN. If dict passed, specific
    per-column NA values.  By default the following values are interpreted as
    NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan',
    '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan',
    'null'.
keep_default_na : bool, default True
    If na_values are specified and keep_default_na is False the default NaN
    values are overridden, otherwise they're appended to.
```

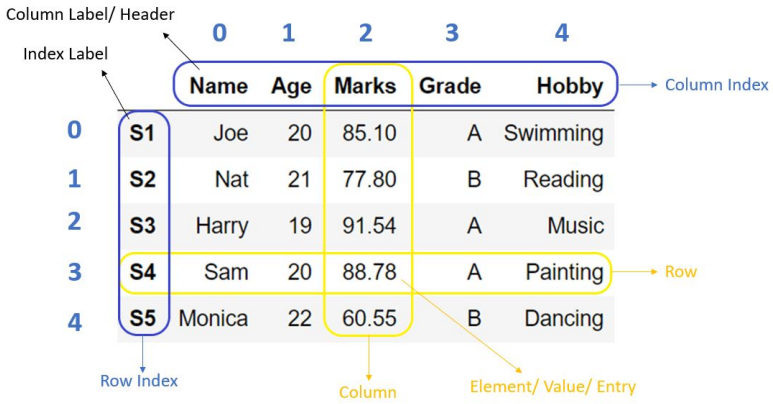
Creating Pandas Data Structures

Creating a DataFrame from a Dictionary

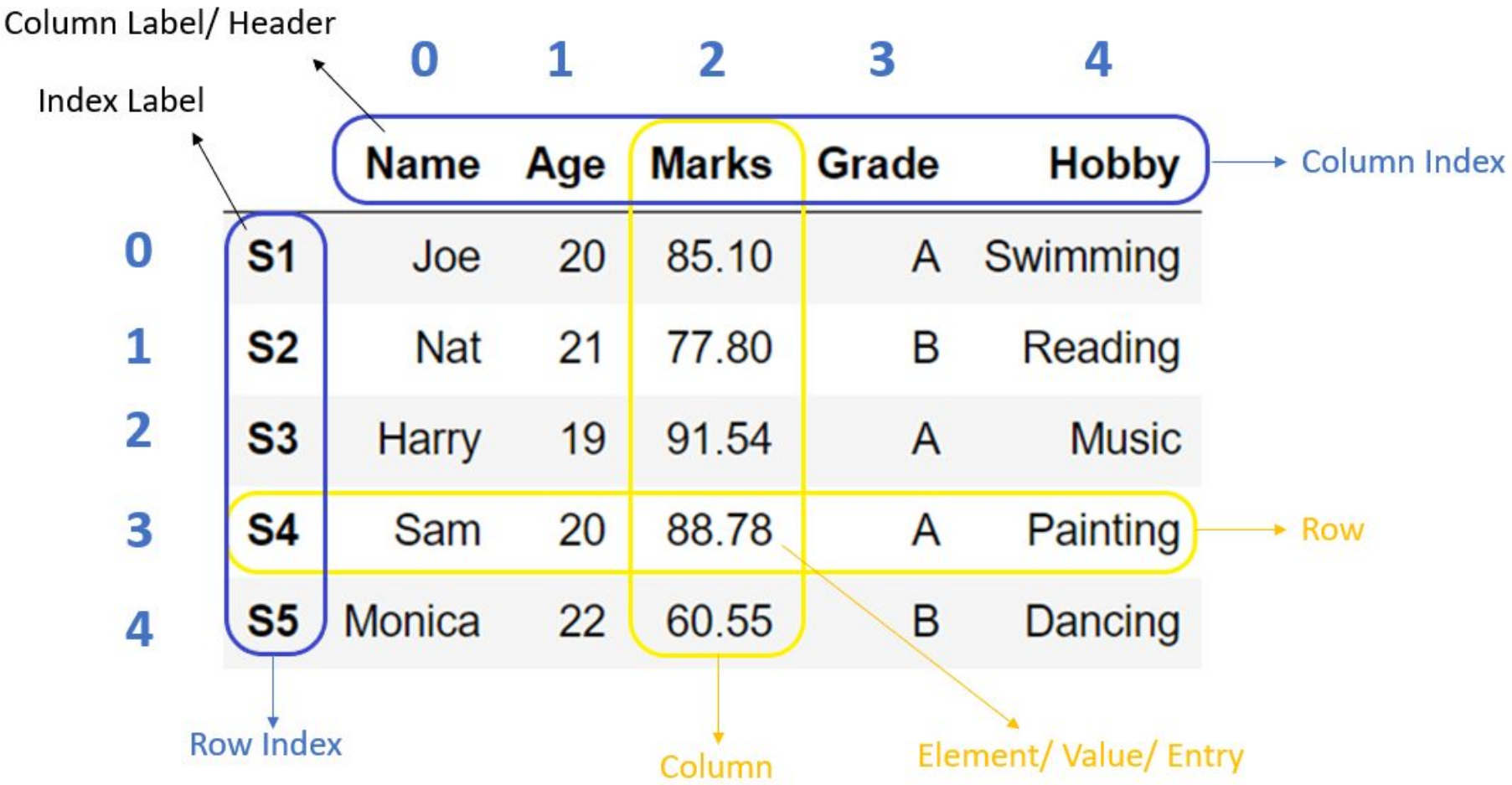
import pandas as pd

```
[2]  data = {'Name': ['Alice', 'Bob', 'Charlie'],
             'Age': [25, 30, 35],
             'City': ['New York', 'Los Angeles', 'Chicago']}
     df = pd.DataFrame(data)
     print(df)
```

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```



A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It is similar to a table in a database or a data frame in R.


Column Label/ Header
Index Label
0 1 2 3 4
Name Age Marks Grade Hobby
Column Index
0 S1 Joe 20 85.10 A Swimming
1 S2 Nat 21 77.80 B Reading
2 S3 Harry 19 91.54 A Music
3 S4 Sam 20 88.78 A Painting
4 S5 Monica 22 60.55 B Dancing
Row
Row Index
Column
Element/ Value/ Entry

Viewing and Inspecting Dat

Viewing the First Few Rows

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

The head() method returns the first five rows of the DataFrame. This is useful for quickly examining the top of a dataset.

Viewing and Inspecting Data

اخدت نظرة اولية على :

- عدد الاعمدة
- عدد ال rows
- datatype
- count of missing data

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

Getting DataFrame Information

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    3 non-null      object
 1   Age     3 non-null      int64
 2   City    3 non-null      object
dtypes: int64(1), object(2)
memory usage: 200.0+ bytes
```

The info() method provides a summary of the DataFrame, including the number of non-null entries and the data type of each column.

Viewing and Inspecting Dat

Describing Statistical Information

```
df.describe()
```

	Age
count	3.0
mean	30.0
std	5.0
min	25.0
25%	27.5
50%	30.0
75%	32.5
max	35.0

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

The describe() method generates descriptive statistics, such as mean, standard deviation, min, and max, for numerical columns.

"Whenever you open a new dataset, your **first 5 minutes** should include these five commands:

```
df.shape
df.columns
df.info()
df.describe()
df.head()
```

These will give you a **quick understanding** of the data's structure, size, types, and any missing values — before you do **any cleaning or analysis**."

Eng: Baraa slides

Selecting and Filtering Data

Selecting Columns

```
print(df['Name'])
```

```
0      Alice
1        Bob
2    Charlie
Name: Name, dtype: object
```

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

You can select a single column from a DataFrame using the column name. This returns a Series object.

Selecting and Filtering Data

Filtering Rows Based on Condition

```
[7]  filtered_df = df[df['Age'] > 30]
     print(filtered_df)
```

```
      Name  Age     City
2  Charlie   35  Chicago
```

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

You can filter rows based on a condition. Here, only rows where the 'Age' column is greater than 30 are selected.

Selecting and Filtering Data

Selecting Specific Rows and Columns

```
selected_data = df.loc[0:1, ['Name', 'City']]
print(selected_data)
```

```
    Name         City
0  Alice     New York
1    Bob  Los Angeles
```

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

The loc[] method is used for label-based indexing to select specific rows and columns. Here, rows with indices 0 and 1 and columns 'Name' and 'City' are selected.

loc[row , col]

go to code

- Use `[]` for selecting columns
- Use `.loc[row_lables, column_labels]` for label-based indexing
- Use `.iloc[row_positions, column_positions]` for positional index

[37]:
```
df
```

[37]:

	A	B	C
a	1	True	0.496714
b	2	True	-0.138264
c	3	False	0.647689

[21]:
```
df.loc['a':'b', ['A', 'C']]
```

[21]:

	A	C
a	1	0.496714
b	2	-0.138264

[43]:
```
df.iloc[0:2, [0,2]]
```

[43]:

	A	C
a	1	0.496714
b	2	-0.138264

Modifying Data

Adding a New Column

```
df['Salary'] = [70000, 80000, 90000]
print(df)
```

```
      Name  Age         City  Salary
0    Alice   25     New York   70000
1      Bob   30  Los Angeles   80000
2  Charlie   35      Chicago   90000
```

A new column 'Salary' is added to the DataFrame with specified values. This is a common operation to add calculated fields or new data.

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

Modifying Data

Updating Column Values

```
df['Age'] = df['Age'] + 1
print(df)
```

```
      Name  Age         City  Salary
0    Alice   26     New York   70000
1      Bob   31  Los Angeles   80000
2  Charlie   36      Chicago   90000
```

```
df.head()
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

This updates the values in the 'Age' column by adding 1 to each value. Pandas allows vectorized operations, making this efficient.

Modifying Data

Dropping Columns

```
df = df.drop('City', axis=1)
print(df)
```

```
      Name  Age  Salary
0    Alice   26   70000
1      Bob   31   80000
2  Charlie   36   90000
```

The drop() method removes the 'City' column from the DataFrame. The axis=1 argument specifies that a column is being dropped.

axis{0 or ‘index’, 1 or ‘columns’}, default 0
Whether to drop labels from the index (0 or ‘index’) or columns (1 or ‘columns’).

```
print(df)
```

```
      Name  Age         City  Salary
0    Alice   26     New York   70000
1      Bob   31  Los Angeles   80000
2  Charlie   36      Chicago   90000
```

Try: axis 0

Handling Missing Data

Detecting Missing Values

```
df_with_nan = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})
print(df_with_nan.isnull())
```

```
       A      B
0  False  False
1  False   True
2   True  False
```

The isnull() method detects missing values in the DataFrame and returns a DataFrame of the same shape with True or False values.

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

[99]:
```
df_with_nan= pd.DataFrame({'A':[1,2,None],'B':[None,None,6]})
df_with_nan
```

[99]:

	A	B
0	1.0	NaN
1	2.0	NaN
2	NaN	6.0

[101]:
```
df_with_nan.isnull()
```

[101]:

	A	B
0	False	True
1	False	True
2	True	False

[109]:
```
df_with_nan.isna()
```

[109]:

	A	B
0	False	True
1	False	True
2	True	False

[103]:
```
df_with_nan.isnull().sum()
```

[103]:
```
A    1
B    2
dtype: int64
```

[105]:
```
#total sum of null
df_with_nan.isnull().sum().sum()
```

[105]:
```
3
```

Handling Missing Data

Filling Missing Values

```
# Create DataFrame from dict
student_df = pd.DataFrame(student_dict , index=['a', 'b', 'c','d'] )
student_df
```

[99]:

	Name	Age	Marks
a	Joe	20.0	NaN
b	Ali	NaN	NaN
c	Baraa	40.0	95.0
d	Tala	35.0	80.0

[101]:

```
student_df.fillna(0)
```

[101]:

	Name	Age	Marks
a	Joe	20.0	0.0
b	Ali	0.0	0.0
c	Baraa	40.0	95.0
d	Tala	35.0	80.0

The fillna() method replaces missing values with a specified value, such as 0. This is useful for cleaning datasets before analysis.

```
[133]: df_with_nan= pd.DataFrame({'A':[1,2,None],'B':[None,None,6]})
       df_with_nan
```

[133]:

	A	B
0	1.0	NaN
1	2.0	NaN
2	NaN	6.0

```
[137]: df_with_nan['A'] = df_with_nan['A'].fillna(  df_with_nan['A'].mean())
```

```
[139]: df_with_nan
```

[139]:

	A	B
0	1.0	NaN
1	2.0	NaN
2	1.5	6.0

```
[131]: df_with_nan['B'] = df_with_nan['B'].fillna(  df_with_nan['B'].mean()  )
       df_with_nan
```

[131]:

	A	B
0	1.0	6.0
1	2.0	6.0
2	1.5	6.0

```python
# Python dict object
student_dict = {'Name': ['Joe', 'Ali','Baraa',"Tala"], 'Age': [20, None,40,35], 'Marks': [None, None,95,80]}

# Create DataFrame from dict
student_df = pd.DataFrame(student_dict , index=['a', 'b', 'c','d'] )
student_df
```

[99]:

	Name	Age	Marks
a	Joe	20.0	NaN
b	Ali	NaN	NaN
c	Baraa	40.0	95.0
d	Tala	35.0	80.0

[105]:
```python
student_df["Marks"].mean()
```

[105]:
```
87.5
```

[111]:
```python
student_df["Marks"].fillna( student_df["Marks"].mean()  )
```

[111]:
```
a    87.5
b    87.5
c    95.0
d    80.0
Name: Marks, dtype: float64
```

رواد مصر الرقمية

```
df_with_nan = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})
print(df_with_nan.isnull())
```

```
       A      B
0  False  False
1  False   True
2   True  False
```

Handling Missing Data

Dropping Rows with Missing Valu

```
cleaned_df = df_with_nan.dropna()
print(cleaned_df)
```

```
     A    B
0  1.0  4.0
```

The dropna() method removes rows with any missing values. This can be useful when working with datasets where missing values cannot be tolerated.

Try axis 1

Grouping and Aggregation

Applying Custom Functions with apply()

```
def double_salary(x):
    return x * 2

df['Double Salary'] = df['Salary'].apply(double_salary)
print(df)
```

```
  Department  Salary  Double Salary
0         HR   50000         100000
1         IT   60000         120000
2         HR   45000          90000
3         IT   80000         160000
```

The apply() method is used to apply a custom function to each element in a column. Here, it doubles the salary for each employee.

Using Apply

Let's see how we can use apply with a custom function on a single column:

```
def Get_Gender_Bool(gender):

    if gender == "male":
        return 1
    else:
        return 0
```

```
df_age_sex["Bool_Male"] = df_age_sex.Sex.apply(Get_Gender_Bool)

df_age_sex
```

	Age	Sex	Bool_Male
0	34.5	male	1
1	47.0	female	0
2	62.0	male	1
3	27.0	male	1
4	22.0	female	0

Calls Get_Gender_Bool once for each row.

[304]:
```
def getResult(x):
    if x == "A":
        return "Excellent"
    else:
        return "Good"
```

[298]:
```
df["Result"] = df["Grade"].apply( lambda x : "Excellent" if x == "A" else "Good"  )
```

[306]:
```
df["Result1"] = df["Grade"].apply( getResult )
```

[308]:
```
df
```

[308]:

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Grade Num	Result	Result1
0	Jake	32.0	1	19.5	20.0	1	10.0	33.0	A	A	Excellent	Excellent
1	?	32.0	1	20.0	16.0	1	14.0	32.0	A	A	Excellent	Excellent
2	Susan	30.0	1	19.0	19.0	1	10.5	33.0	A-	A-	Good	Good
3	Sol	31.0	1	22.0	13.0	1	13.0	34.0	A	A	Excellent	Excellent
4	Chris	30.0	1	19.0	17.0	1	12.5	33.5	A	A	Excellent	Excellent
5	Tarik	31.0	1	19.0	19.0	1	8.0	24.0	B	B	Good	Good

Using Apply

Last one:

```
def Get_Title(name):

    parsed_name = name.split(" ")
    title = parsed_name[1]

    return title
```

```
df_name_age["Title"] = df_name_age.Name.apply(Get_Title)
df_name_age
```

	Name	Age	Title
0	Kelly, Mr. James	34.5	Mr.
1	Wilkes, Mrs. James (Ellen Needs)	47.0	Mrs.
2	Myles, Mr. Thomas Francis	62.0	Mr.
3	Wirz, Mr. Albert	27.0	Mr.
4	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	22.0	Mrs.

Grouping and Aggregation

```
[15] df = pd.DataFrame({'Department': ['HR', 'IT', 'HR', 'IT'],
                        'Salary': [50000, 60000, 45000, 80000]})
     grouped = df.groupby('Department').mean()
     print(grouped)
```

```
            Salary
Department
HR         47500.0
IT         70000.0
```

select avg(salary)
from users
group by Department

The groupby() method groups the data by a specified column and then calculates the mean salary for each department. This is commonly used for summarizing data.

Practical Examples

Calculating the Average Salary by Department

```
average_salary_by_dept = df.groupby('Department')['Salary'].mean()
print("Average Salary by Department:\n")
average_salary_by_dept.head()
```

```
Average Salary by Department:
```

Department	Salary
Finance	87500.000000
HR	53000.000000
IT	70333.333333
Marketing	69000.000000

dtype: float64

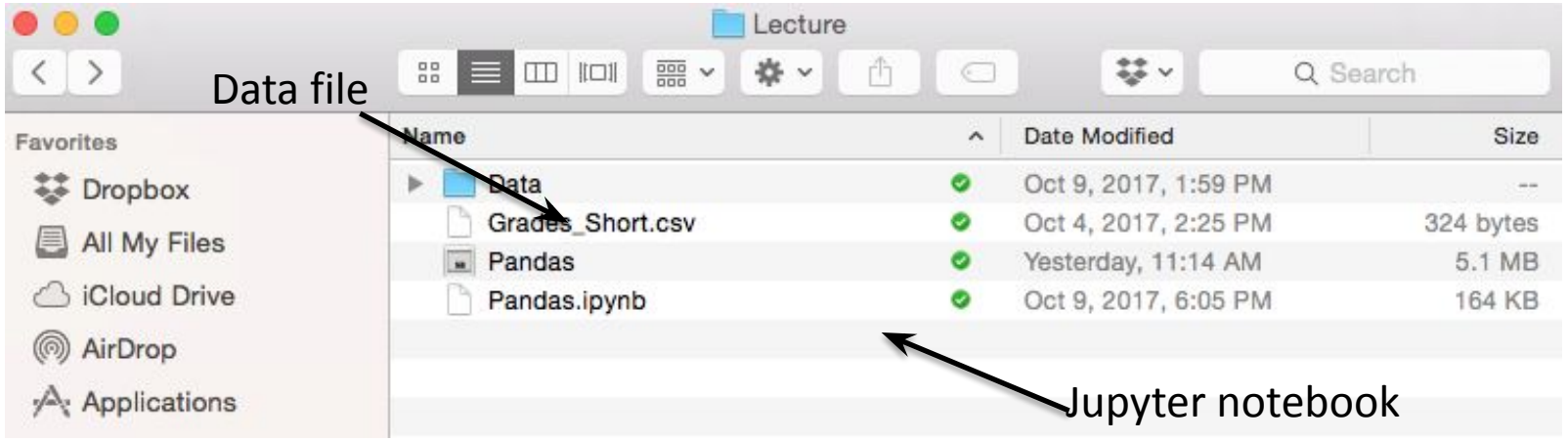
```
print("First 5 Rows:\n")
df.head()
```

```
First 5 Rows:
```

	ID	Name	Department	Age	Salary	Joining_Date
0	1	John Doe	HR	28	50000	2020-05-21
1	2	Jane Smith	IT	34	75000	2019-06-12
2	3	Bob Johnson	Finance	45	85000	2018-07-15
3	4	Alice White	IT	29	72000	2021-01-04
4	5	Charlie Brown	Marketing	32	68000	2019-11-30

The groupby() method groups the data by the 'Department' column and calculates the mean salary for each department using mean().

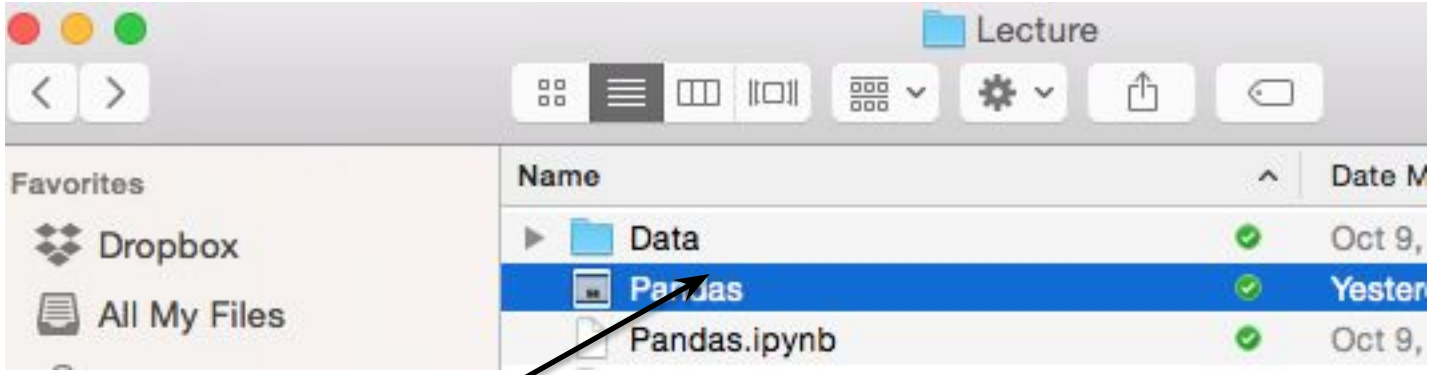
Reading in Data From Excel




```
#Relative file path
df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	62

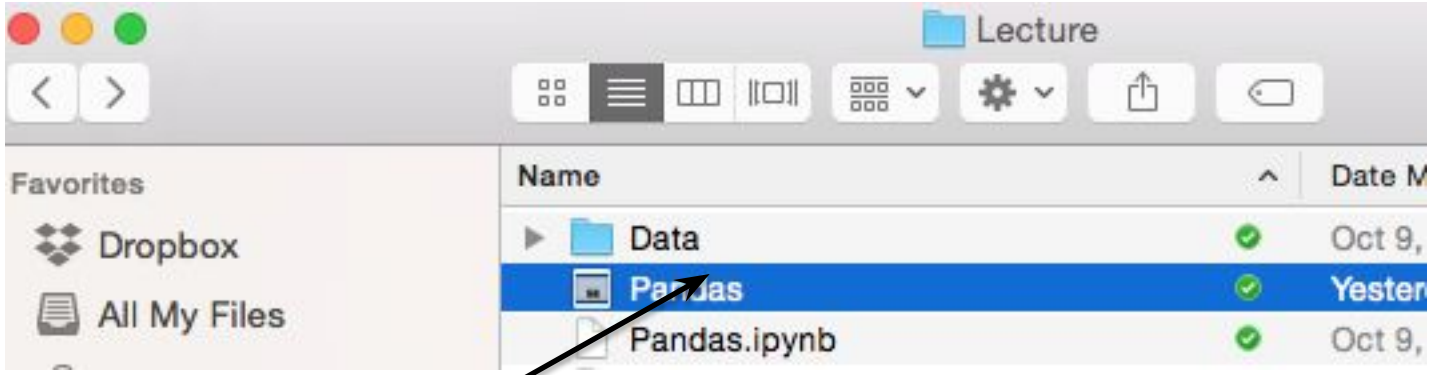
Reading in Data From Excel



Now Grades_Short.csv is in Data Folder

Jupyter notebook

Reading in Data From Excel



Now Grades_Short.csv is in Data Folder

Jupyter Notebook

"/" separates directories

```
#Relative file path
df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	ID
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	90743
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	7284
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	7625
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1237
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	62

Basic Features

```
import pandas as pd

df_grades = pd.read_csv("Grades_Short.csv")
df_grades.head(3)
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

```
#dimension of df
df_grades.shape
```

```
(7, 9)
```

Think of this as a list

```
#How each column is stored
df_grades.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2         int64
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
dtype: object
```

object = string

float64 = decimal

int64 = integer

Basic Features

column names

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-

row names = index

```
#Get column names
df_grades.columns
```

```
Index(['Name', 'Previous_Part', 'Participation1', 'Mini_Exam1', 'Mini_Exam2',
       'Participation2', 'Mini_Exam3', 'Final', 'Grade'],
      dtype='object')
```

```
#Get row names
df_grades.index
```

```
RangeIndex(start=0, stop=7, step=1)
```

Selecting a Single Column

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Get Name column
df_grades['Name']
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

- Between square brackets, the column must be given as a string
- Outputs column as a series
  - A series is a one dimensional dataframe..more on this in the slicing section

Selecting a Single Column

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Get Name column
df_grades.Name
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

- Exactly equivalent way to get Name column
  - + : don't have to type brackets or quotes
  - -: won't generalize to selecting multiple columns,, won't work if column names have spaces, can't create new columns this way

Selecting Multiple Columns

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

```
#Select multiple columns
df_grades[["Name", "Grade"]]
```

	Name	Grade
0	Jake	A
1	Joe	A
2	Susan	A-
3	Sol	A
4	Chris	A
5	Tarik	B
6	Malik	A

- List of strings, which correspond to column names.
- You can select as many column as you want.
- Column don’t have to be contiguous.

Storing Result

```
#Print the column
df_grades["Name"]
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

```
#Store the column
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

The variable name stores a series

Why store a slice?

- We might want/have to do our analysis is steps.
  - Less error prone
  - More readable

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Picking out single element

```
names[0]
```

```
'Jake'
```

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0     Jake
1      Joe
2    Susan
3      Sol
4    Chris
5    Tarik
6    Malik
Name: Name, dtype: object
```

Picking out single element

```
names[0]
```

```
'Jake'
```

Contiguous slice

non_inclusive

```
names[1:4]
```

```
1      Joe
2    Susan
3      Sol
Name: Name, dtype: object
```

Slicing a Series

Slice/index through the index, which is usually numbers

```
names= df_grades["Name"]
names
```

```
0      Jake
1       Joe
2     Susan
3       Sol
4     Chris
5     Tarik
6     Malik
Name: Name, dtype: object
```

Picking out single element

```
names[0]
```

```
'Jake'
```

Contiguous slice

```
names[1:4]
```

```
1       Joe
2     Susan
3       Sol
Name: Name, dtype: object
```

Arbitrary slice

```
names[[1,2,4]]
```

```
1       Joe
2     Susan
4     Chris
Name: Name, dtype: object
```

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- There are a few ways to pick slice a data frame, we will use the .loc method.
- Access elements through the index labels column names
    - We will see how to change both of these labels later on

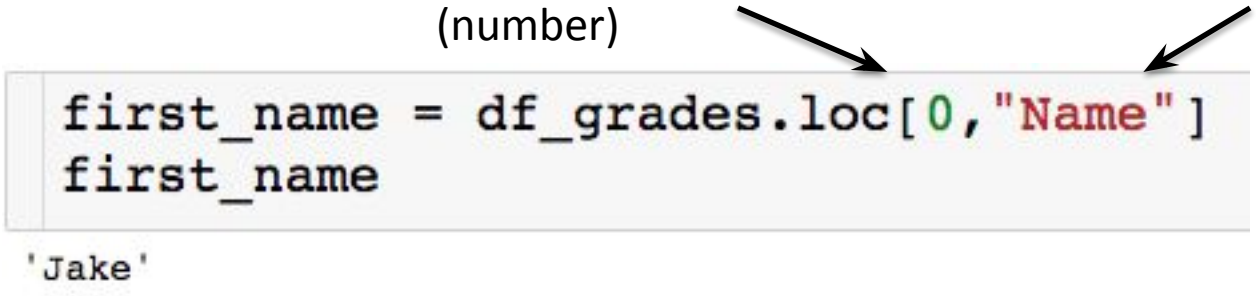
Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick a single value out.

Column name (string)

Index label (number)



Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out entire row:

```
first_row = df_grades.loc[0,:]
first_row
```

"pick out all columns"

```
Name                Jake
Previous_Part         32
Participation1         1
Mini_Exam1          19.5
Mini_Exam2            20
Participation2         1
Mini_Exam3            10
Final                 33
Grade                  A
Name: 0, dtype: object
```

first_row is a series

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out contiguous chunk:

Endpoints are inclusive!

```
slice_one = df_grades.loc[0:2,"Name":"Mini_Exam2"]
slice_one
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2
0	Jake	32.0	1	19.5	20
1	Joe	32.0	1	20.0	16
2	Susan	30.0	1	19.0	19

df_gredes.iloc[0:3 , 0:5]

Slicing a Data Frame

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

- Pick out arbitrary chunk:

```
slice_two = df_grades.loc[[0,2,3], ["Name", "Grade"]]
slice_two
```

	Name	Grade
0	Jake	A
2	Susan	A-
3	Sol	A

Built in Functions

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the average score on the final?

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the average score on the final?

```
#Print out
df_grades.Final.mean()
```

```
32.214285714285715
```

```
#Store
avg_final = df_grades.Final.mean()
avg_final
```

```
32.214285714285715
```

Built in mean() method

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

How do I compute the highest Mini Exam 1 score?

```
max_mini_1 = df_grades["Mini_Exam1"].max()
max_mini_1
```

```
22.0
```

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

summary_df is a dataframe!

```
summary_df = df_grades.describe()
summary_df
```

	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final
count	7.000000	7.0	7.000000	7.000000	7.0	7.000000	7.000000
mean	31.071429	1.0	19.785714	17.857143	1.0	11.000000	32.214286
std	0.838082	0.0	1.074598	2.734262	0.0	2.217356	3.828154
min	30.000000	1.0	19.000000	13.000000	1.0	8.000000	24.000000
25%	30.500000	1.0	19.000000	16.500000	1.0	9.500000	32.500000
50%	31.000000	1.0	19.500000	19.000000	1.0	10.500000	33.000000
75%	31.750000	1.0	20.000000	19.500000	1.0	12.750000	33.750000
max	32.000000	1.0	22.000000	21.000000	1.0	14.000000	36.000000

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

I can actually get all key stats for *numeric* columns at once with the describe() method:

```
summary_df = df_grades.describe()
summary_df[["Final", "Mini_Exam3"]]
```

	Final	Mini_Exam3
count	7.000000	7.000000
mean	32.214286	11.000000
std	3.828154	2.217356
min	24.000000	8.000000
25%	32.500000	9.500000
50%	33.000000	10.500000
75%	33.750000	12.750000
max	36.000000	14.000000

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
df_grades["Grade"].value_counts()
```

```
A     5
A-    1
B     1
Name: Grade, dtype: int64
```

value_count(): Gives a count of the number of times each unique value apears in the column. Returns a series where indices are the unique column values.

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
counts = df_grades["Grade"].value_counts()
counts
```

```
A     5
A-    1
B     1
Name: Grade, dtype: int64
```

```
counts["A"]
```

```
5
```

value_count(): Gives a count of the number of times each unique value appears in the column. Returns a series where indices are the unique column values.

Built in Functions

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A
5	Tarik	31.0	1	19.0	19	1	8.0	24.0	B
6	Malik	31.5	1	20.0	21	1	9.0	36.0	A

Other useful built in methods:

```
df_grades["Grade"].unique()
```

```
array(['A', 'A-', 'B'], dtype=object)
```

```
unique_values = df_grades["Grade"].unique()
unique_values[0]
```

```
'A'
```

```
len(unique_values)
```

```
3
```

unique(): Returns an array of all of the unique values.

Attributes vs. Methods

When do I a put a ()?

```
#Get dimensions
df_grades.shape
```

```
(7, 9)
```

```
#Get column types
df_grades.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2         int64
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
dtype: object
```

```
#Get first 5 rows
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

```
#Compute mean of finals column
df_grades["Final"].mean()
```

```
32.214285714285715
```

Attributes vs. Methods

When do I a put a ()?

dataframe attributes

```
#Get dimensions
df_grades.shape
```

```
(7, 9)
```

```
#Get column types
df_grades.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2         int64
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
dtype: object
```

dataframe methods

```
#Get first 5 rows
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

```
#Compute mean of finals column
df_grades["Final"].mean()
```

```
32.214285714285715
```

Attributes vs. Methods

When do I a put a ()?

dataframe attributes

```
#Get dimensions
df_grades.shape
```

```
(7, 9)
```

```
#Get column types
df_grades.dtypes
```

```
Name                object
Previous_Part      float64
Participation1       int64
Mini_Exam1         float64
Mini_Exam2           int64
Participation2       int64
Mini_Exam3         float64
Final              float64
Grade               object
dtype: object
```

Features of dataframe

dataframe methods

```
#Get first 5 rows
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

```
#Compute mean of finals column
df_grades["Final"].mean()
```

```
32.214285714285715
```

Require computation for output

Creating New Columns

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

Let's create a useless new column of all 1s:

```
df_grades["new_column"]=1
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1

Creating New Columns

```
import pandas as pd

df_grades = pd.read_csv("Data/Grades_Short.csv")
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

$33/36$

$32/36$

We can also create column as function of other column. The Final was worth 36 points, let's create a column for each student's percentage.

```
df_grades["Final_Percentage"]=df_grades["Final"]/36
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Final_Percentage
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	0.916667
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	0.888889
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	0.916667
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	0.944444
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	0.930556

Deleting Columns

```
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Final_Percentage	Part_perc
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	0.916667	1.0
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	0.888889	1.0
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	0.916667	1.0
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	0.944444	1.0
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	0.930556	1.0

```
#Delete single column
del df_grades["Final_Percentage"]
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Part_perc
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	1.0
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	1.0
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	1.0
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	1.0
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	1.0

Deleting Columns

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	new_column	Part_perc
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	1	1.0
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A	1	1.0
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-	1	1.0
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A	1	1.0
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A	1	1.0

```
#Delete multiple columns
df_grades.drop(["new_column","Part_perc"], axis=1, inplace = True)
df_grades.head()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A
1	Joe	32.0	1	20.0	16	1	14.0	32.0	A
2	Susan	30.0	1	19.0	19	1	10.5	33.0	A-
3	Sol	31.0	1	22.0	13	1	13.0	34.0	A
4	Chris	30.0	1	19.0	17	1	12.5	33.5	A

Missing Data

	A	B	C	D	E	F	G	H	I	J
	Name	Previous_Par	Participation	Mini_Exam1	Mini_Exam2	Participation	Mini_Exam3	Final	Grade	Temp
	Jake	32	1	19.5	20	1	10	33	A	-1
	Joe	NA	1	20	16	1	14	32	A	23
	Sol	31	1	22	13	1	13	34	A	34
	Chris	30	-1	19	not available	1	12.5	33.5	A	72

```
df_missing = pd.read_csv("Data/Missing_Data.csv")
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

Not that different columns have different indicators for missing data.

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

```
df_missing.dtypes
```

```
Name              object
Previous_Part    float64
Participation1     int64
Mini_Exam1       float64
Mini_Exam2        object
Participation2     int64
Mini_Exam3       float64
Final            float64
Grade             object
Temp               int64
dtype: object
```

na_values : *Hashable, Iterable of Hashable or dict of {Hashable : It*
optional

Additional strings to recognize as `NA`/`NaN`. If `dict` passed, specific
`NA` values. By default the following values are interpreted as `NaN`: "
"#N/A N/A", "#NA", "-1.#IND", "-1.#QNAN", "-NaN", "-nan", "1.#IND"
"1.#QNAN", "<NA>", "N/A", "NA", "NULL", "NaN", "None", "n/a", "na

We can replace the missing data with a true NaN (right now everything is just a string).

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

List of strings specifying which values are missing.

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

List of strings specifying which values are missing.

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	NaN	1	12.5	33.5	A	72

Special NaN value (from numpy package), which is not a string.

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

We know “NaN” and “not available” are missing data points, but what about -1?

Missing Data

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

We know “NaN” and “not available” are missing data points, but what about -1?

- For the Participation1 column the -1 is probably missing data.
- For the Temp column, the -1 is likely not missing data, since -1 is a valid temperature.

For each column, we can specify exactly which values correspond to missing data.

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})

df_missing
```

Curly brackets

Missing Data

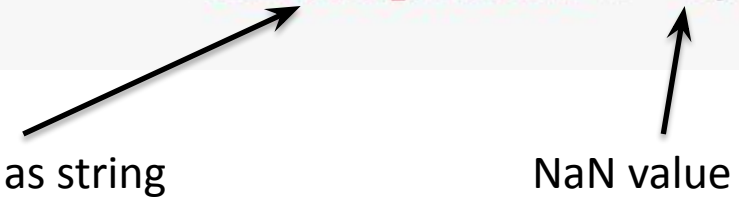
```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

Column name as string

NaN value

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```



“For column Participation1, replace all -1s with a NaN.”

Missing Data

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72

Notice that the -1 was replaced only in Participation1 column

Comparing Approaches

Approach 1:

- Does a global search and replace in all columns.

```
df_missing = pd.read_csv("Data/Missing_Data.csv", \
                         na_values=["NaN", "not available"])
df_missing
```

Approach 2:

- Allows you to specify column by column the values that should be replaced with NaN.

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values={"Mini_Exam2" : "not available",\
                                                             "Participation1": -1})
df_missing
```

Benefiting of Having NaNs

- Have common symbol for where there is missing data
    - Good for you and good for others looking at your code/data
    - These entries will be ignored if you try to compute means of columns with NaNs.
- We can easily get rid of column/rows with missing data
- We can easily replace the missing values with the mean of the column, for example.

Isnull() Method

- The isnull() method lets you check where the NaNs are:

```
df = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", -1, "not available"])
df
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

```
#Using isnull()
df.isnull()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	False	False	False	False	False	False	False	False	False	True
1	False	True	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	False	True	False	True	False	False	False	False	False

[166]:
```
df_WithNotNan=pd.read_csv("Data/Missing_Data.csv" )
df_WithNotNan
```

[166]:

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	not available	1	12.5	33.5	A	72

[168]:
```
df_WithNotNan.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 10 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Name            4 non-null      object
 1   Previous_Part   3 non-null      float64
 2   Participation1  4 non-null      int64
 3   Mini_Exam1      4 non-null      float64
 4   Mini_Exam2      4 non-null      object
 5   Participation2  4 non-null      int64
 6   Mini_Exam3      4 non-null      float64
 7   Final           4 non-null      float64
 8   Grade           4 non-null      object
 9   Temp            4 non-null      int64
dtypes: float64(4), int64(3), object(3)
memory usage: 452.0+ bytes
```

[170]:
```
df_WithNotNan.isnull().sum()
```

[170]:
```
Name              0
Previous_Part     1
Participation1    0
Mini_Exam1        0
Mini_Exam2        0
Participation2    0
Mini_Exam3        0
Final             0
Grade             0
Temp              0
dtype: int64
```

[172]:
```
df_WithNotNan["Mini_Exam2"] = pd.to_numeric(df_WithNotNan["Mini_Exam2"],errors = "coerce")
df_WithNotNan
```

[172]:

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1	19.5	20.0	1	10.0	33.0	A	-1
1	Joe	NaN	1	20.0	16.0	1	14.0	32.0	A	23
2	Sol	31.0	1	22.0	13.0	1	13.0	34.0	A	34
3	Chris	30.0	-1	19.0	NaN	1	12.5	33.5	A	72

[176]:
```
df_WithNotNan.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 10 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Name            4 non-null      object
 1   Previous_Part   3 non-null      float64
 2   Participation1  4 non-null      int64
 3   Mini_Exam1      4 non-null      float64
 4   Mini_Exam2      3 non-null      float64
 5   Participation2  4 non-null      int64
 6   Mini_Exam3      4 non-null      float64
 7   Final           4 non-null      float64
 8   Grade           4 non-null      object
 9   Temp            4 non-null      int64
```

Isnull() Method

- The isnull() method lets you check where the NaNs are:

```
#Using isnull()
df.isnull()
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	False	False	False	False	False	False	False	False	False	True
1	False	True	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False
3	False	False	True	False	True	False	False	False	False	False

```
#Remember Booleans are just 0s and 1s.
#Check how many NaNs are in each column
df.isnull().sum()
```

```
Name              0
Previous_Part     1
Participation1    1
Mini_Exam1        0
Mini_Exam2        1
Participation2    0
Mini_Exam3        0
Final             0
Grade             0
Temp              1
dtype: int64
```

Dropna() Method

```
df_missing = pd.read_csv("Data/Missing_Data.csv", na_values=["NaN", "not available",\
                                                             -1])
df_missing
```

	Name	Previous_Part	Participation1	Mini_Exam1	Mini_Exam2	Participation2	Mini_Exam3	Final	Grade	Temp
0	Jake	32.0	1.0	19.5	20.0	1	10.0	33.0	A	NaN
1	Joe	NaN	1.0	20.0	16.0	1	14.0	32.0	A	23.0
2	Sol	31.0	1.0	22.0	13.0	1	13.0	34.0	A	34.0
3	Chris	30.0	NaN	19.0	NaN	1	12.5	33.5	A	72.0

How do I get rid of all rows with NaN?

Practical Examples

Practical Examples

We'll load this CSV file into a Pandas DataFrame and perform various data manipulation tasks

```
import pandas as pd

# Load the CSV file into a Pandas DataFrame
df = pd.read_csv('employees.csv')
```

Practical Examples

Viewing the First Few Rows of the DataFrame

```
print("First 5 Rows:\n")
df.head()
```

```
First 5 Rows:
```

	ID	Name	Department	Age	Salary	Joining_Date
0	1	John Doe	HR	28	50000	2020-05-21
1	2	Jane Smith	IT	34	75000	2019-06-12
2	3	Bob Johnson	Finance	45	85000	2018-07-15
3	4	Alice White	IT	29	72000	2021-01-04
4	5	Charlie Brown	Marketing	32	68000	2019-11-30

The head() method displays the first five rows of the DataFrame, which is useful for getting a quick look at the data.

Practical Examples

Displaying Basic Information About the DataFrame

```
print("DataFrame Information:\n")
df.info()
```

```
DataFrame Information:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   ID            10 non-null     int64
 1   Name          10 non-null     object
 2   Department    10 non-null     object
 3   Age           10 non-null     int64
 4   Salary        10 non-null     int64
 5   Joining_Date  10 non-null     object
dtypes: int64(3), object(3)
memory usage: 608.0+ bytes
```

The info() method provides a summary of the DataFrame, including the number of non-null values, data types, and memory usage.

Practical Examples

Calculating Basic Statistics for Numerical Columns

```
print("Statistical Summary:\n")
df.describe()
```

```
Statistical Summary:
```

	ID	Age	Salary
count	10.00000	10.000000	10.000000
mean	5.50000	35.300000	68300.000000
std	3.02765	7.958922	13106.826042
min	1.00000	26.000000	50000.000000
25%	3.25000	29.250000	57250.000000
50%	5.50000	33.000000	69000.000000
75%	7.75000	40.250000	74250.000000
max	10.00000	50.000000	90000.000000

The describe() method generates descriptive statistics such as mean, standard deviation, and percentiles for numerical columns.

```
import pandas as pd

# We pass a dict of {column name: column values}
df = pd.DataFrame({'X':[78,85,96,80,86], 'Y':[84,None,None,None,None],'Z':[86,97,96,72,None]});
df
```

[181]:

	X	Y	Z
0	78	84.0	86.0
1	85	NaN	97.0
2	96	NaN	96.0
3	80	NaN	72.0
4	86	NaN	NaN

[183]:

```
df.describe()
```

[183]:

	X	Y	Z
count	5.0	1.0	4.000000
mean	85.0	84.0	87.750000
std	7.0	NaN	11.615363
min	78.0	84.0	72.000000
25%	80.0	84.0	82.500000
50%	85.0	84.0	91.000000
75%	86.0	84.0	96.250000

Practical Examples

Filtering Employees by IT Department

```
it_employees = df[df['Department'] == 'IT']
print("IT Department Employees:\n")
it_employees.head()
```

```
IT Department Employees:
```

	ID	Name	Department	Age	Salary	Joining_Date
1	2	Jane Smith	IT	34	75000	2019-06-12
3	4	Alice White	IT	29	72000	2021-01-04
7	8	Linda Taylor	IT	26	64000	2022-02-10

This code filters the DataFrame to show only employees who work in the IT department using a condition on the 'Department' column.

Practical Examples

Adding a New Column for Salary After a 5% Raise

```
df['Salary_After_Raise'] = df['Salary'] * 1.05
print("DataFrame with Salary After Raise:\n" )
df.head()
```

```
DataFrame with Salary After Raise:
```

	ID	Name	Department	Age	Salary	Joining_Date	Salary_After_Raise
0	1	John Doe	HR	28	50000	2020-05-21	52500.0
1	2	Jane Smith	IT	34	75000	2019-06-12	78750.0
2	3	Bob Johnson	Finance	45	85000	2018-07-15	89250.0
3	4	Alice White	IT	29	72000	2021-01-04	75600.0
4	5	Charlie Brown	Marketing	32	68000	2019-11-30	71400.0

A new column is added to the DataFrame by multiplying the 'Salary' column by 1.05 to reflect a 5% salary increase.

Practical Examples

Sorting Employees by Age in Descending Order

```
sorted_by_age = df.sort_values(by='Age', ascending=False)
print("Employees Sorted by Age (Descending):\n")
sorted_by_age.head()
```

```
Employees Sorted by Age (Descending):
```

	ID	Name	Department	Age	Salary	Joining_Date	Salary_After_Raise
6	7	Mike Davis	Finance	50	90000	2016-08-19	94500.0
2	3	Bob Johnson	Finance	45	85000	2018-07-15	89250.0
5	6	Eva Green	HR	41	54000	2017-04-22	56700.0
8	9	James Wilson	Marketing	38	70000	2020-09-08	73500.0
1	2	Jane Smith	IT	34	75000	2019-06-12	78750.0

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html

The sort_values() method sorts the DataFrame by the 'Age' column in descending order, showing the oldest employees first.

Practical Examples

Calculating the Number of Employees in Each Department

```
employee_count_by_dept = df['Department'].value_counts()
print("Number of Employees by Department:\n")
employee_count_by_dept.head()
```

```
Number of Employees by Department:
```

Department	count
HR	3
IT	3
Finance	2
Marketing	2

dtype: int64

The value_counts() method counts the number of occurrences of each unique value in the 'Department' column, showing how many employees are in each department.

Practical Examples

Finding Employees Who Joined After 2020

```
df['Joining_Date'] = pd.to_datetime(df['Joining_Date'])
recent_joins = df[df['Joining_Date'] > '2020-01-01']
print("Employees Joined After 2020:\n")
recent_joins.head()
```

```
Employees Joined After 2020:
```

	ID	Name	Department	Age	Salary	Joining_Date	Salary_After_Raise
0	1	John Doe	HR	28	50000	2020-05-21	52500.0
3	4	Alice White	IT	29	72000	2021-01-04	75600.0
7	8	Linda Taylor	IT	26	64000	2022-02-10	67200.0
8	9	James Wilson	Marketing	38	70000	2020-09-08	73500.0
9	10	Sarah Adams	HR	30	55000	2021-03-18	57750.0

This converts the 'Joining_Date' column to datetime format and filters the DataFrame to find employees who joined after January 1, 2020.

Practical Examples

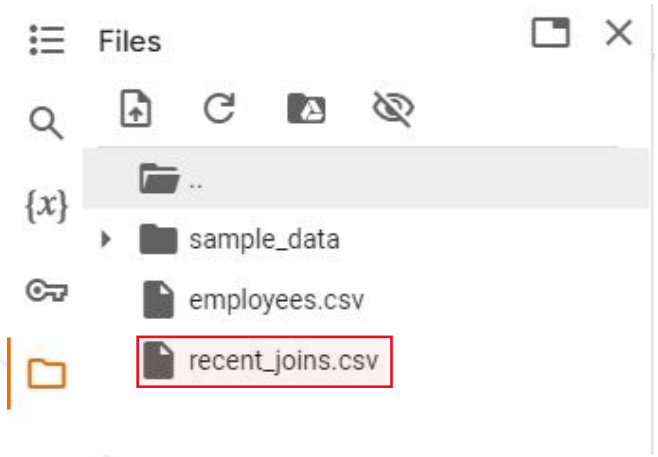
Saving the Filtered DataFrame to a New CSV File

```
recent_joins.to_csv('recent_joins.csv', index=False)
print("Filtered data saved to recent_joins.csv.")
```

```
Filtered data saved to recent_joins.csv.
```

The to_csv() method saves the filtered DataFrame (recent_joins) to a new CSV file named 'recent_joins.csv'. The index=False argument excludes the row index from the CSV file.

Summary



Data Preprocessing

Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- Detect and Handle Outliers
- Deal with Imbalanced classes
- Work with Categorical data
- Feature Scaling
- Split data to Train and Test Sets

Feature Engineering and Extraction

- Domain knowledge features (age , wh, **BMI**)
- Date and Time features
- String operations
- Web Data
- Geospatial features

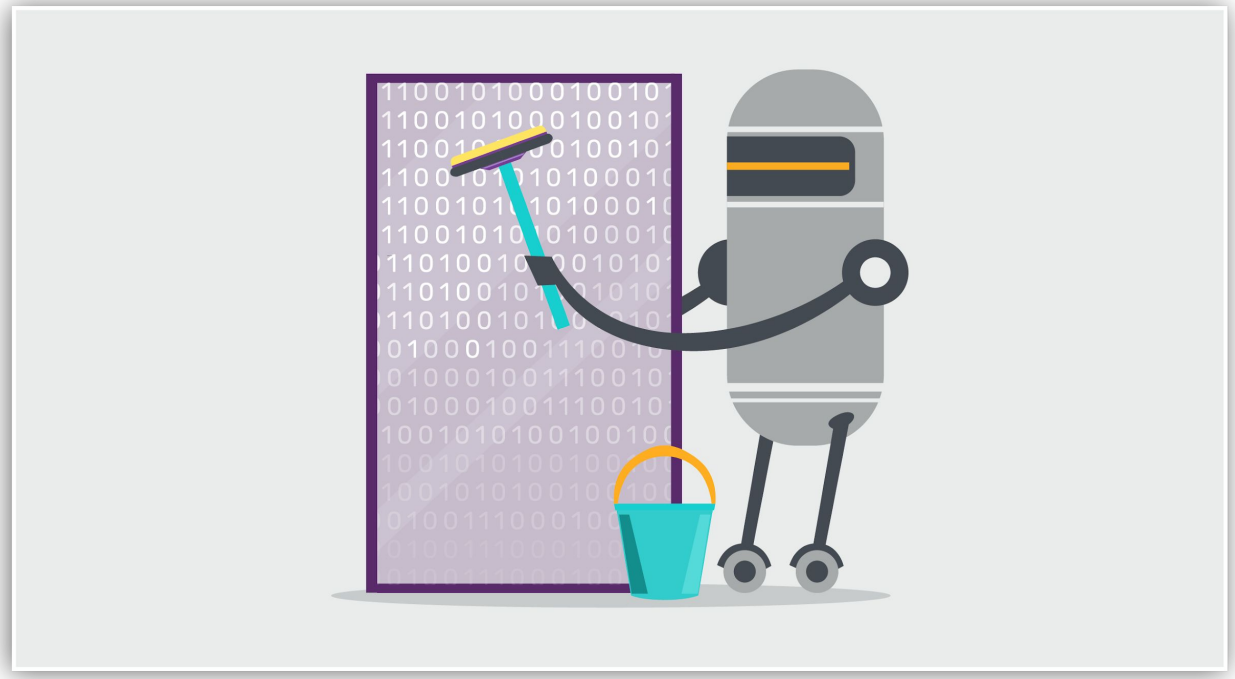
Feature Transformations

1. **Data Cleaning or Cleansing**
2. Work with Missing data
3. Detect and Handle Outliers
4. Deal with Imbalanced classes
5. Work with Categorical data
6. Feature Scaling
7. Split data to Train and Test Sets

Data Cleaning or Cleansing

Cleaning data is the process of preparing the dataset for analysis. It is very important because the accuracy of machine learning or data mining models are affected because of poor quality of data.

So, data scientists spend a large amount of their time cleaning the dataset and transform them into a format with which they can work with. In fact, data scientists spend 80% of their time cleaning the data.



Example of problems

- Columns can have missing data indicators like xx and ?.
- Height column can have value of 0.
- Weight column can have negative values.
- …

Eng: Baraa slides

Data Problems:

to identify and resolve any quality issues with your data

- **Data Constraints Problems**
- **Text and Categorical Data Problems**
- **Data Uniformity Problems**

Data Constraints Problems:

- **Data Type Constraints:**
  - **Problem:** Incorrect data types in columns.
  - **Solutions:** Convert to the correct data type.
  - **Example in Action:** A revenue_usd column that contains values like "$1,000" as strings instead of numeric data types.

- **Data Range Constraints:**
  - **Problem:** Values outside the expected range.
  - **Solutions:** Check for typos, drop rows, set to min/max, or impute.
  - **Example in Action:** A gpa column with values exceeding the valid range, such as a GPA of 5.2.

- **Uniqueness Constraints:**
  - **Problem:** Duplicates in the data.
  - **Solutions:** Keep one, merge non-exact duplicates, or infer categories.
  - **Example in Action:** Duplicate rows with the same name and phone_number columns but different height_cm values.

Text and Categorical Data Problems:

- **Membership Constraints for Categorical Data:**
  - **Problem:** Inconsistent categories.
  - **Solutions:** Drop rows, remap categories, or infer correct categories.
  - **Example in Action:** Two different entries for "New York" “NYC” , “New york”in the city column, causing inconsistency.
- **Length Violation for Text Data:**
  - **Problem:** Text length doesn't match expectations.
  - **Solutions:** Drop rows or set affected observations to missing.
  - **Example in Action:** A US phone_number column with entries that have 9 characters instead of the expected 14.
- **Text Data with Inconsistent Formatting:**
  - **Problem:** Text data not following a standard format.
  - **Solutions:** Drop rows or set affected observations to missing.
  - **Example in Action:** Inconsistent date formats like "dd-mm-yyyy" and "mm-dd-yyyy" in a date_of_birth column.

Data Uniformity Problems:

- **Unit Uniformity for Numeric Columns:**
  - **Problem:** Inconsistent units in numeric columns.
  - **Solutions:** Drop rows, standardize units, or apply domain-specific rules.
  - **Example in Action:** A temperature column in Celsius with extremely high or low values that are outside the expected range.
- **Unit Uniformity for Date Columns:**
  - **Problem:** Inconsistent date formats.
  - **Solutions:** Standardize formats, drop rows, or apply domain-specific rules.
  - **Example in Action:** A birthday column containing dates in various formats, such as "dd-mm-yyyy" and "mm-dd-yyyy."
- **Crossfield Validation for Numeric or Date Columns:**
  - **Problem:** Inconsistent or incorrect relationships between columns or between date columns.
  - **Solutions:** Drop rows or apply domain-specific validation rules.
  - **Example in Action:** Total flight bookings per class not adding up to the total recorded bookings.
  - **Example in Action:** A date_of_birth column not corresponding with the calculated age based on other date columns.

Data Cleaning (First Step in Preprocessing)

"Garbage in → Garbage out"

Why Cleaning First?

- Raw data is always messy.
- ML models are **sensitive** to errors.
- Cleaning builds the **foundation** for all later steps.
- Example: You can't cook with dirty vegetables → first wash & clean.

Data Problems Detection – Quick Checklist

1. **Inspect structure** → `shape`, `info`, `head()`
2. **Find missing values** → `isnull().sum() (we see later)`
3. **Detect duplicates** → `duplicated().sum()`
4. **Spot outliers** → `describe()`, `boxplot() (we see later)`
5. **Check category consistency** → `unique()`
6. **Catch invalid values** → logical rules (`Age < 0`, `Salary < 0`)
7. **Clean text fields** → `str.strip()`, `str.lower()`

Always **diagnose your dataset** before training

→ "Bad data in → bad model out."

→ "Garbage in → Garbage out"

Quick Data Inspection

- **`df.shape`** **`# rows & columns`**
  - Tells you (rows, columns).
  - Example: (2000, 10) → 2000 records, 10 features.
  - Helps to know if dataset is small, medium, or big.

- **`df.head()`** **`# first rows`**
  - Shows sample records.
  - Detect strange values early.
  - Example: Last row has Age = 999 → suspicious.

- **`df.info()`** **`# data types`**
  - Shows column names, data types (int, float, object, datetime).
  - Detects mismatches:
    - Age stored as “object” instead of “int”.
    - Date stored as string instead of datetime.

- **`df.describe()`** **`# summary stats`**
  - Mean, min, max, std for each numeric column.
  - Quickly spot impossible values (e.g., Height min = -10).

💡 Why Important?

- Like a **doctor’s first check-up**: weight, temperature, blood pressure.

- Before deep analysis, you need to know the “overall health” of the dataset.

Like a **doctor’s first check-up**: weight, temperature, blood pressure.

Before deep analysis, you need to know the “overall health” of the dataset.

Feature Transformations

- Data Cleaning or Cleansing
- **Work with Missing data**
- Detect and Handle Outliers
- Deal with Imbalanced classes
- Work with Categorical data
- Feature Scaling
- Split data to Train and Test Sets

Work with Missing data

```
1 df.isnull().sum()
```


- **<u>Check Missing Data</u>**
- Drop Missing Data
- Fill Missing data with Pandas
- Fill Missing data with Sklearn `SimpleImputer`
- Fill Missing data with Sklearn `KNNImputer`

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

[99]:
```
df_with_nan= pd.DataFrame({'A':[1,2,None],'B':[None,None,6]})
df_with_nan
```

[99]:

	A	B
0	1.0	NaN
1	2.0	NaN
2	NaN	6.0

[101]:
```
df_with_nan.isnull()
```

[101]:

	A	B
0	False	True
1	False	True
2	True	False

[109]:
```
df_with_nan.isna()
```

[109]:

	A	B
0	False	True
1	False	True
2	True	False

[103]:
```
df_with_nan.isnull().sum()
```

[103]:
```
A    1
B    2
dtype: int64
```

[105]:
```
#total sum of null
df_with_nan.isnull().sum().sum()
```

[105]:
```
3
```

Check missing data

In Pandas missing data is represented by two value: None/NaN

we use a function isnull() to check even if NAN is true/false

we use a function isnull() to check number of nan in each column

```
[3] import pandas as pd
    #read csv file
    data=pd.read_csv('/content/data-2010-12-01.csv')
    data.head(6)
```

	X	Y	Z	W
0	10	12.0	1.0	5.0
1	27	NaN	5.0	NaN
2	20	39.0	0.0	2.0
3	17	11.0	NaN	NaN
4	93	15.0	6.0	28.0
5	59	0.0	NaN	NaN

```
data.isnull()
```

	X	Y	Z	W
0	False	False	False	False
1	False	True	False	True
2	False	False	False	False
3	False	False	True	True
4	False	False	False	False
5	False	False	True	True

```
data.isnull().sum()
```

```
X    0
Y    1
Z    2
W    3
dtype: int64
```

Eng: Baraa slides

Drop Missing Data

1.When the data goes missing on 70–80 percent of the variable:

dropping the variable should be considered

2. When the data goes missing on 1–5 percent of the variable

dropping missing values only should be considered

dropna(axis=0, how="any", subset=None, inplace=False)

axis: possible values are {0 or ‘index’, 1 or ‘columns’}, default 0. If 0, drop rows with null values. If 1, drop columns with missing values.

how: possible values are {‘any’, ‘all’}, default ‘any’. If ‘any’, drop the row/column if any of the values is null. If ‘all’, drop the row/column if all the values are missing.

subset: specifies the rows/columns to look for null values.

inplace: a boolean value. If True, the source DataFrame is changed and None is returned

Dropping rows/columns with at least 1 null value

```
[3] import pandas as pd
    #read csv file
    data=pd.read_csv('/content/data-2010-12-01.csv')
    data.head(6)
```

	X	Y	Z	W
0	10	12.0	1.0	5.0
1	27	NaN	5.0	NaN
2	20	39.0	0.0	2.0
3	17	11.0	NaN	NaN
4	93	15.0	6.0	28.0
5	59	0.0	NaN	NaN

```
data.dropna(axis=0)
```

	X	Y	Z	W
0	10	12.0	1.0	5.0
2	20	39.0	10.0	2.0
4	93	15.0	6.0	28.0

Rows

```
data.dropna(axis=1)
```

	X
0	10
1	27
2	20
3	17
4	93
5	59

columns

Eng: Baraa slides

Work with Missing data

```python
df.dropna(axis = 0)

df.dropna(axis = 1)
```


- Check Missing Data
- **Drop Missing Data**
- Fill Missing data with Pandas
- Fill Missing data with Sklearn `SimpleImputer`
- Fill Missing data with Sklearn `KNNImputer`

Fill Missing data with Pandas

1. Filling the missing data with the **mean or median** value if it's **a numerical variable.**
2. Filling the missing data with the **median** value only if it's a **numerical variable with outliers**
3. Filling the missing data with **mode** if it'**s a categorical value.**

Work with Missing data

```python
# fill with mean
df['BuildingArea'].fillna(df['BuildingArea'].mean(), inplace=True)


# fill with median
df['YearBuilt'].fillna(df['YearBuilt'].median(), inplace=True)


# fill with mode (most frequent)
df['Car'].fillna(df['Car'].mode()[0], inplace=True)
```

- Check Missing Data
- Drop Missing Data
- **Fill Missing data with Pandas**
- Fill Missing data with Sklearn `SimpleImputer`
- Fill Missing data with Sklearn `KNNImputer`

Work with Missing data

```python
from sklearn.impute import SimpleImputer

# define the Imputer properties
imputer = SimpleImputer(strategy='mean')    # can use 'median' or 'most_frequent'

# impute
df['BuildingArea'] = imputer.fit_transform(df[['BuildingArea']])

# get the filled value
imputer.statistics_
```

- Check Missing Data
- Drop Missing Data
- Fill Missing data with Pandas
- **Fill Missing data with Sklearn `SimpleImputer`**
- Fill Missing data with Sklearn `KNNImputer`

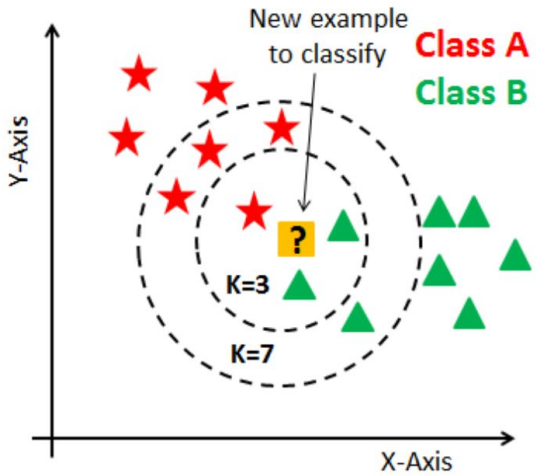
Work with Missing data

```python
import pandas as pd
from sklearn.impute import KNNImputer
df = pd.read_csv('melb_data.csv'

imputer = KNNImputer()
Num_df = df.select_dtypes(include = "number")
Cat_df = df.select_dtypes(include = "object_")
Num_df = pd.DataFrame(imputer.fit_transform(Num_df) , columns= Num_df.columns)

df = pd.concat([Num_df , Cat_df] , axis = 1)
```

- Check Missing Data
- Drop Missing Data
- Fill Missing data with Pandas
- Fill Missing data with Sklearn `SimpleImputer`
- **Fill Missing data with Sklearn `KNNImputer`**



Eng: Baraa slides

Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- **Detect and Handle Outliers**
- Deal with Imbalanced classes
- Work with Categorical data
- Feature Scaling
- Split data to Train and Test Sets

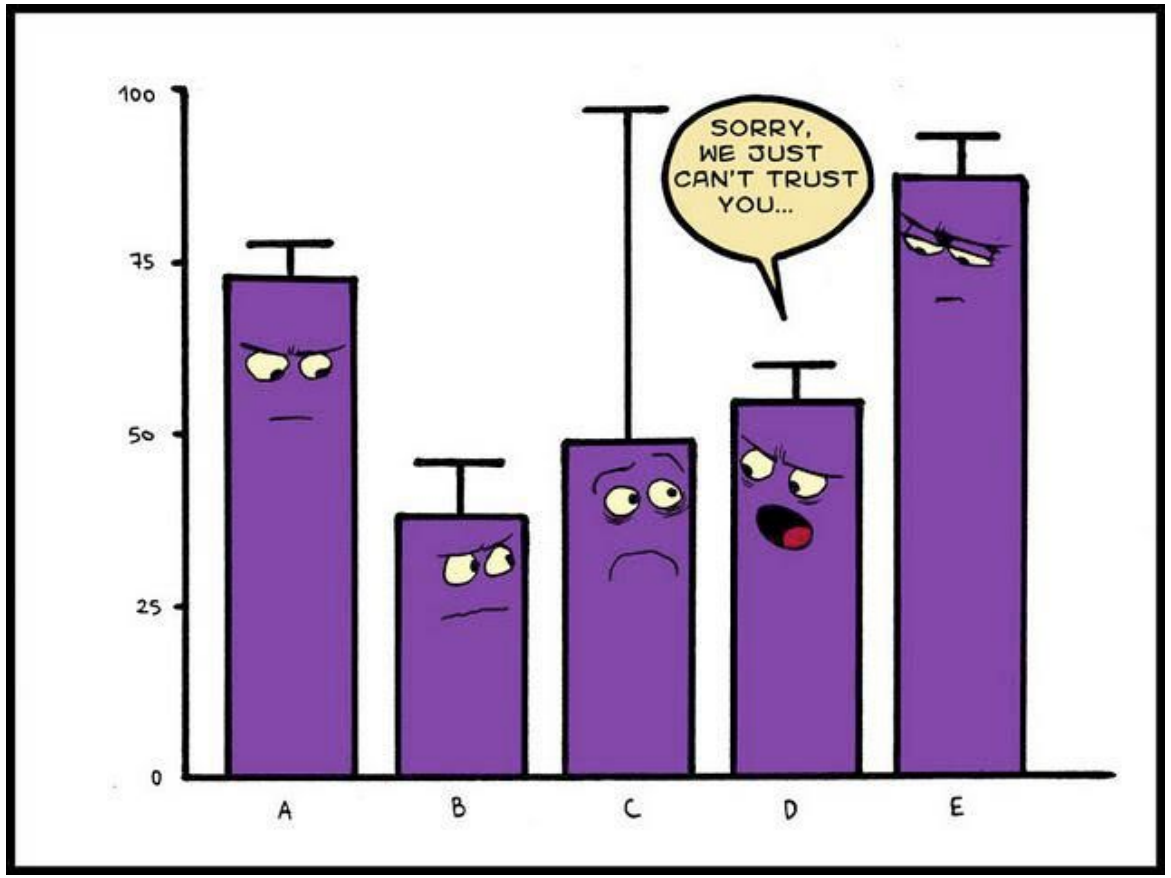
Detect and Handle Outliers

هيخربو الmean

In statistics, an outlier is a data point that differs significantly from other observations.

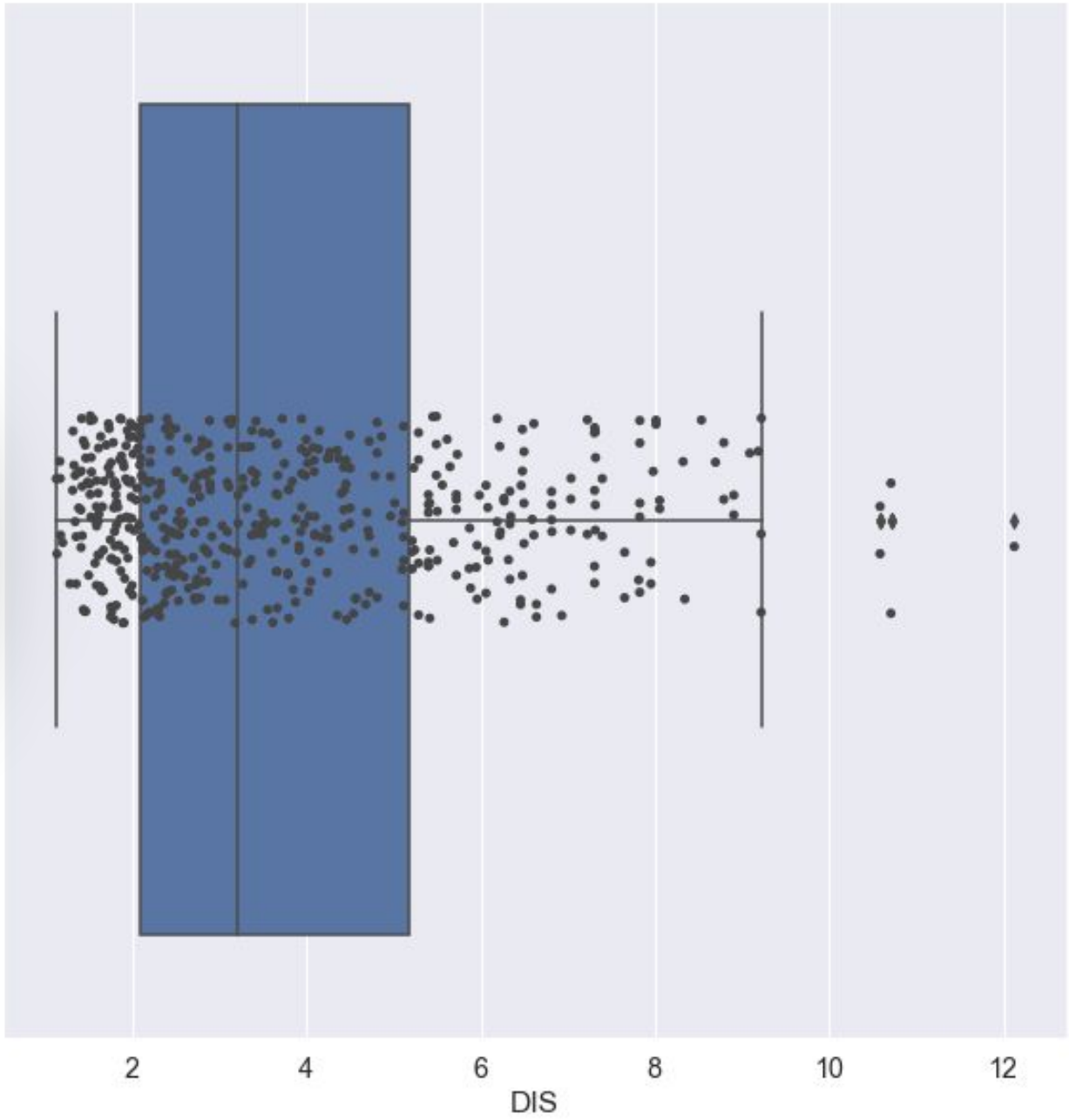
An outlier may be due to variability in the measurement or it may indicate experimental error, the latter are sometimes excluded from the data set.

An outlier can cause serious problems in statistical analyses.



Detect Outliers

```
1 sns.boxplot(x='DIS', data=df)
2 sns.stripplot(x='DIS', data=df)
```

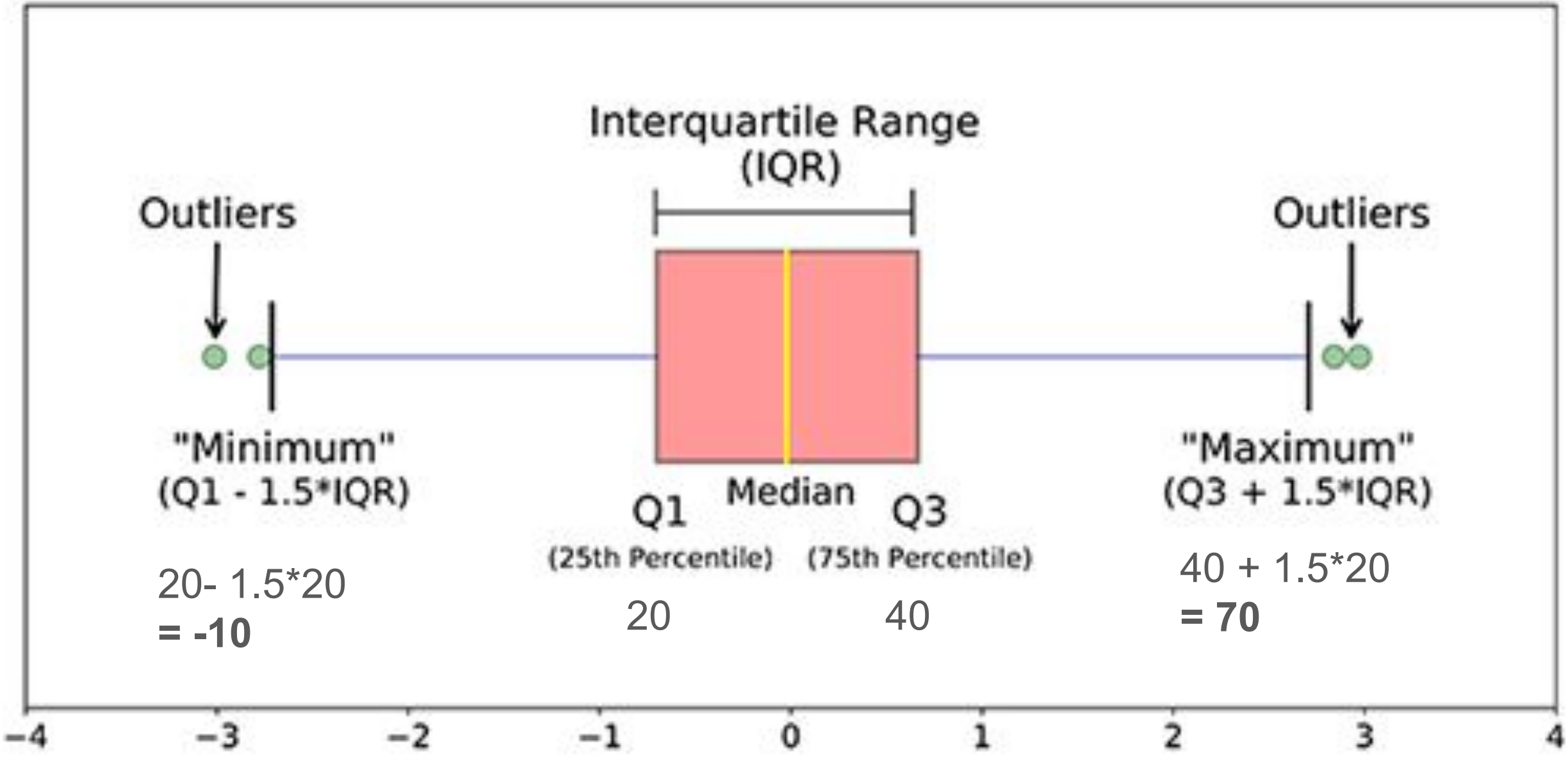


- **Detect Outliers with Visualization.**
- Detect and Handle Outliers with IQR.
- Handle Outliers with business value

Eng: Baraa slides

IQR interquartile range
range between Q1 and Q3

			1q


Interquartile Range
(IQR)
Outliers
Outliers
"Minimum"
(Q1 - 1.5*IQR)
Q1
(25th Percentile)
Median
Q3
(75th Percentile)
"Maximum"
(Q3 + 1.5*IQR)
20- 1.5*20
= -10
20
40
40 + 1.5*20
= 70
−4
−3
−2
−1
0
1
2
3
4

Detect and Handle Outliers

```python
from datasist.structdata import detect_outliers

# remove outliers
outliers_indices = detect_outliers(df, 0, df.columns)
df.drop(outliers_indices, inplace=True)


# replace outliers with median value for each column
for col in df.columns:
    outliers_indices = detect_outliers(df, 0, [col])
    col_median = df[col].median()
    df[col].iloc[outliers_indices] = col_median
```

- Detect Outliers with Visualization.
- **Detect and Handle Outliers with IQR.**
- Handle Outliers with business value

Detect and Handle Outliers with Business value

- Essentially, instead of removing outliers from the data.
- you change their values to something more representative of your data set , value related to the business of your data , that called business value.
- For example **the "Alice in Wonderland"** movie duration in this dataset is 557 Minutes which is an outlier , so we can replace this outlier value with its real duration 75 Minute

	title	year	certificate	runtime_min	genre
490	Alice in Wonderland	1951	PG-13	557	Animation, Adventure, Comedy
262	The Beatles: Get Back	2021	PG-13	468	Documentary, Biography, Music
167	Guardians of the Galaxy Vol. 2	2017	PG-13	23	Action, Adventure, Comedy
474	Willow	2022–	PG-13	46	Action, Adventure, Drama
14	Eternals	2021	PG-13	156	Action, Adventure, Fantasy
...	...	...	...	...	...
628	All About Steve	2009	PG-13	99	Comedy, Romance
638	Ghost Rider: Spirit of Vengeance	2011	PG-13	96	Action, Fantasy, Thriller
706	Kim Possible	2019 TV Movie	PG-13	86	Action, Adventure, Comedy
1355	In the Mix	2005	PG-13	95	Comedy, Crime, Drama
348	The Wolverine	2013	PG-13	0	Action, Sci-Fi

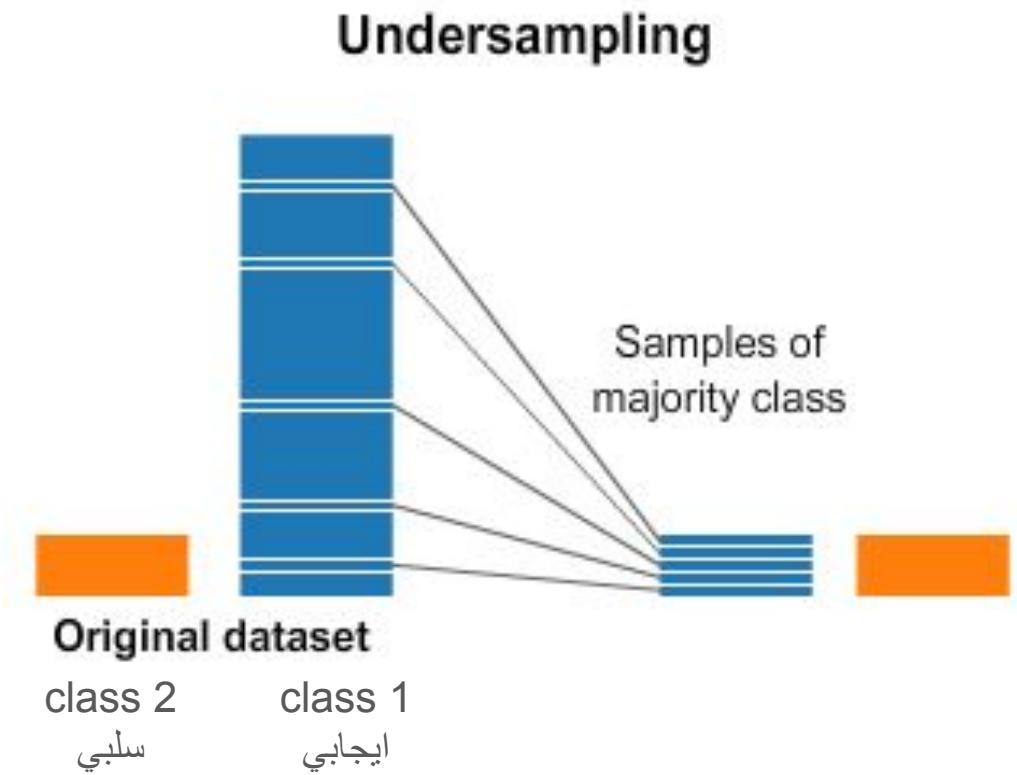
- Detect Outliers with Visualization.
- Detect and Handle Outliers with IQR.
- **Handle Outliers with business value**

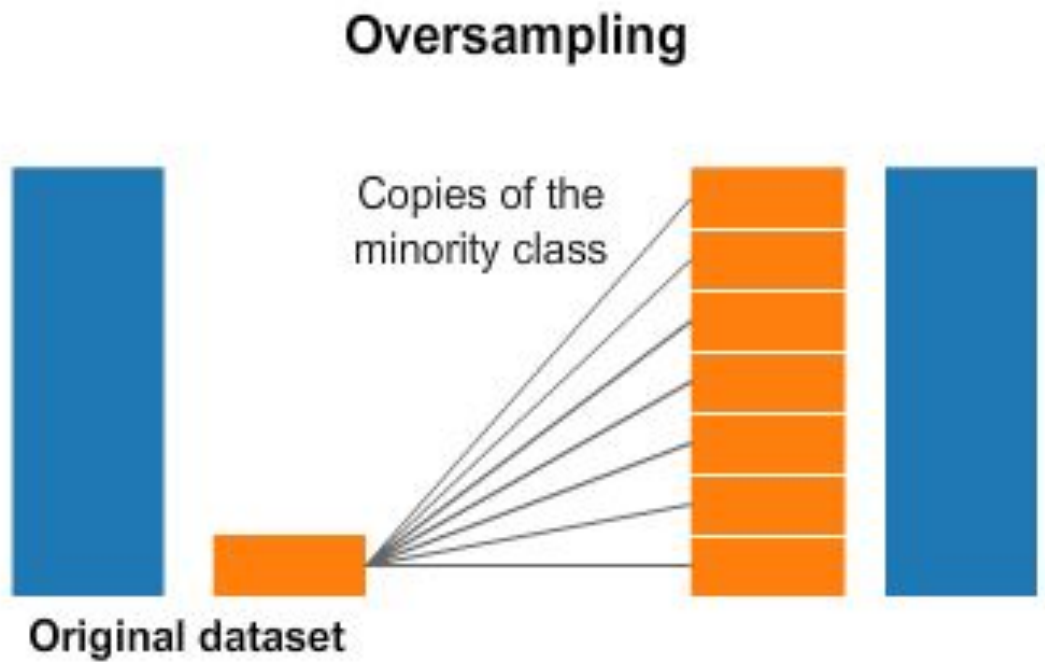
Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- Detect and Handle Outliers
- **Deal with Imbalanced classes**
- Work with Categorical data
- Feature Scaling
- Split data to Train and Test Sets

Deal with Imbalanced classes

Imbalanced classes are a common problem in machine learning classification where there are an unbalanced ratio of observations in each class. Class imbalance can be found in many different areas including medical diagnosis, spam filtering, and fraud detection.





Eng: Baraa slides

Deal with Imbalanced classes

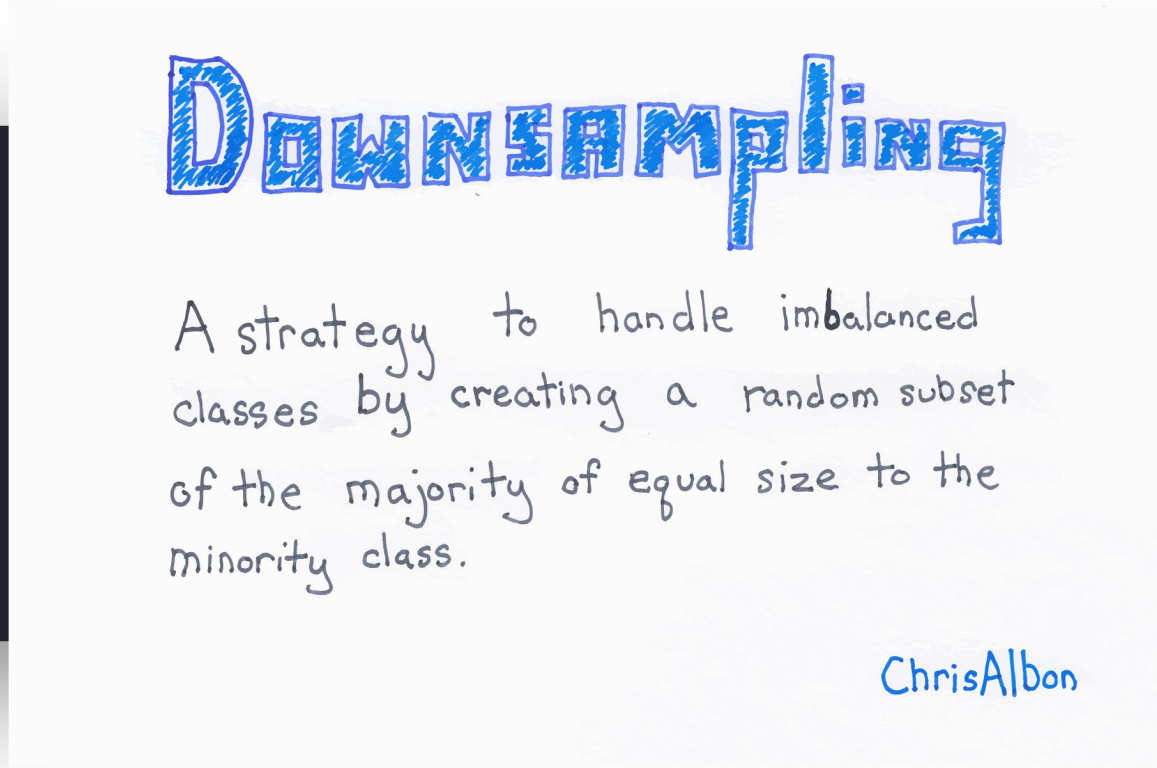
Data Leakage

```python
from imblearn.under_sampling import RandomUnderSampler

x = df.drop('Class', axis=1)
y = df['Class']

rus = RandomUnderSampler()

x_train, y_train = rus.fit_sample(x_train, y_train)
```



- **Down-sampling or Under-sampling Majority Class**
- Generate Synthetic Samples with SMOTE

Eng: Baraa slides

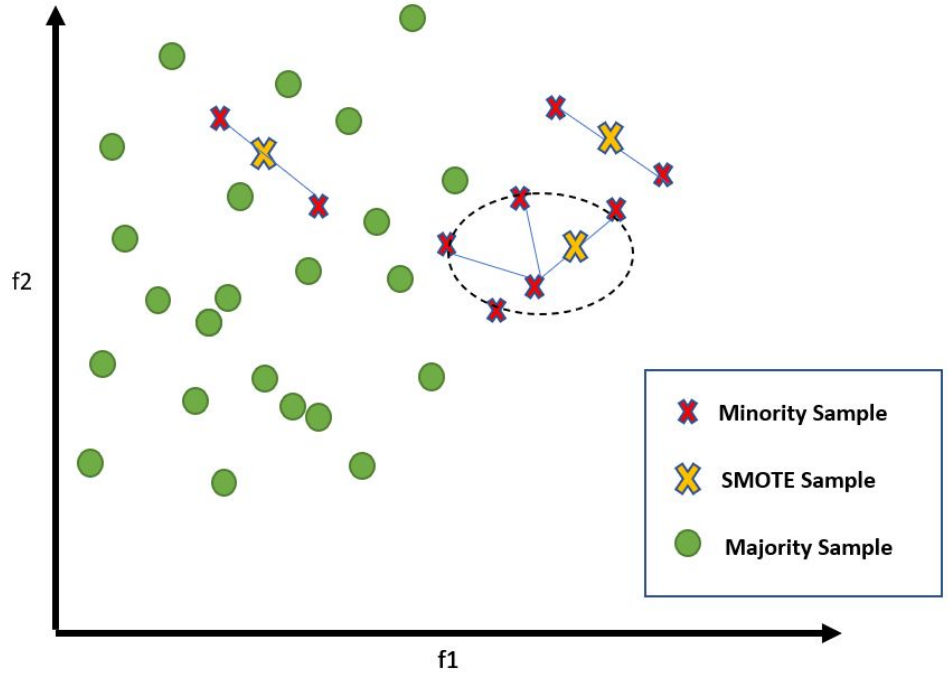
Deal with Imbalanced classes

```python
from imblearn.over_sampling import SMOTE

x = df.drop('Class', axis=1)
y = df['Class']

smote = SMOTE()

x_train, y_train = smote.fit_sample(x_train, y_train)
```



- Down-sampling or Under-sampling Majority Class
- **Generate Synthetic Samples with SMOTE**

Eng: Baraa slides

Deal with Imbalanced classes (Combination)

- Oversampling methods duplicate or create new synthetic examples in the minority class, whereas under sampling methods delete or merge examples in the majority class.
- Both types of resampling can be effective when used in isolation, although can be more effective when both types of methods are used together.

Deal with Imbalanced classes

```
# Manually Combine Over and Undersampling Methods
from imblearn.over_sampling import RandomOverSampler from
imblearn.under_sampling import RandomUnderSampler from imblearn.pipeline import import
Pipeline
#over = RandomOverSampler(sampling_strategy=0.1) over =
SMOTE(sampling_strategy=0.1) under = RandomUnderSampler(sampling_strategy=0.5)
pipeline = Pipeline(steps= [('o', over), ('u', under)]) # fit and apply the
pipeline X_resampled, y_resampled = pipeline.fit_resample(X, y)
```

Eng: Baraa slides

Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- Detect and Handle Outliers
- Deal with Imbalanced classes
- **Feature Scaling**
- Work with Categorical data
- Split data to Train and Test Sets

Feature Scaling

- It is common to scale data prior to fitting a machine learning model.
- This is because data often consists of many different input variables or features (columns) and each may have a different range of values or units of measure, such as feet, miles, kilograms, dollars, etc.
- If there are input variables that have very large values relative to the other input variables, these large values can dominate or skew some machine learning algorithms.
- The result is that the algorithms pay most of their attention to the large values and ignore the variables with smaller values.
- This includes algorithms that use a weighted sum of inputs :

1. like linear regression
2. logistic regression
3. artificial neural networks

age >> 18 to 70
salary >> 10000 to 500000

as well as algorithms that use distance measures between examples such as :

1. k-nearest neighbors
2. support vector machines.

Eng: Baraa slides

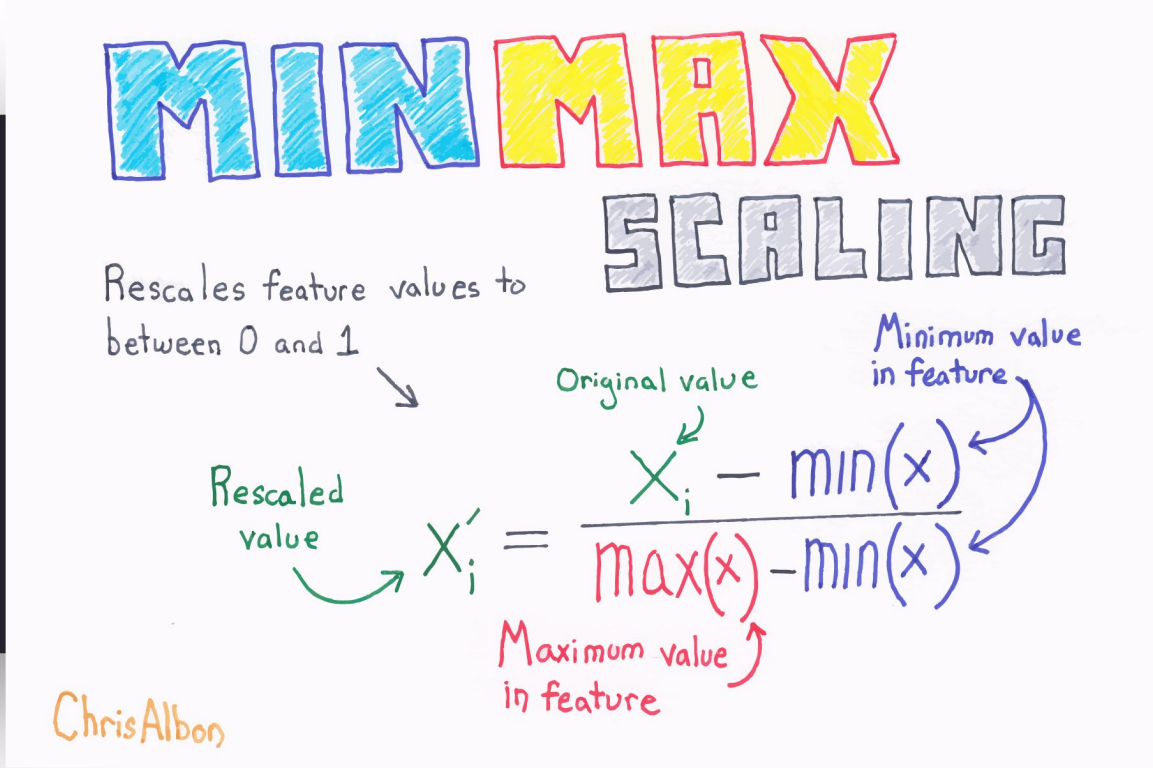
Feature Scaling

```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaler.fit(x_train)

scaled_x_train = scaler.transform(x_train)
scaled_x_test = scaler.transform(x_test)
```



- **Normalization with Sklearn `MinMaxScaler`.**
- Standardizing data with `StandardScaler`.

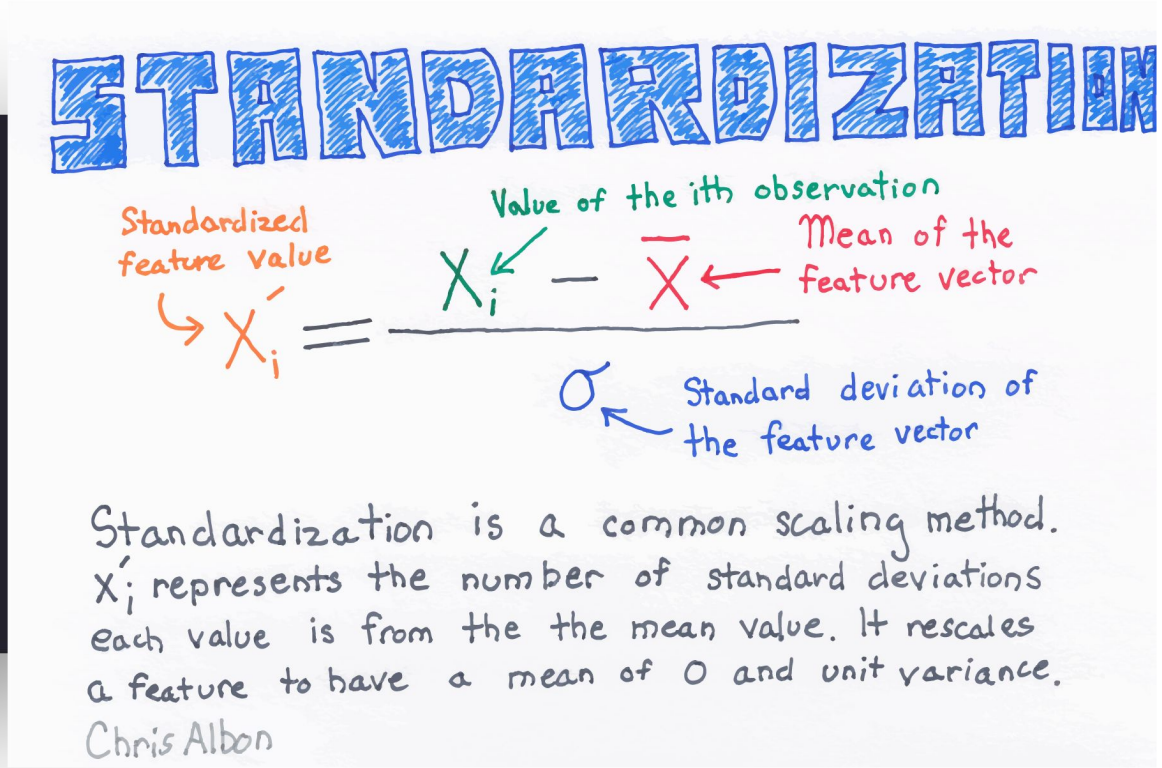
Feature Scaling

```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaler.fit(x_train)

scaled_x_train = scaler.transform(x_train)
scaled_x_test = scaler.transform(x_test)
```



- Normalization with Sklearn `MinMaxScaler`.
- **Standardizing data with `StandardScaler`.**

Eng: Baraa slides

Normalization with Sklearn MinMaxScaler.

- it is the simplest method and consists of rescaling the range of features to scale the range in [0, 1]

Standardizing data with StandardScaler.

- Feature standardization makes the values of each feature in the data have zero mean and unit variance.

Feature Scaling (Robust Scalar)

- Sometimes an input variable may have outlier values. These are values on the edge of the distribution that may have a low probability of occurrence yet are overrepresented for some reason.
- **Outliers can skew a probability distribution and make data scaling using standardization difficult as the calculated mean and standard deviation will be skewed by the presence of the outliers.**
- One approach to standardizing input variables in the presence of outliers is to ignore the outliers from the calculation of the mean and standard deviation, then use the calculated values to scale the variable. This is called robust standardization or **robust data scaling.**
- This can be achieved by calculating the median (50th percentile) and the 25th and 75th percentiles.
- The values of each variable then have their median subtracted and are divided by the interquartile range (IQR) which is the difference between the 75th and 25th percentiles

$$\text{value} = \frac{\text{value} - \text{median}}{p_{75} - p_{25}}$$

❌ **Common Mistakes (**Feature Scaling):

1. **Applying Feature Scaling on the entire dataset before splitting into train and test sets**
   → This can cause **data leakage**, as information from the test set may influence the scaling parameters (like mean and standard deviation).

2. **Forgetting to apply the same scaler (fitted on training data) to the test data**

⚠️ **Potential Mistakes** (Imbalanced Classes):

1. **Misusing Under-sampling → Loss of valuable information!**
   → Removing too many samples from the majority class can result in losing important patterns in the data.

2. **Using SMOTE before splitting the data → Data Leakage!**
   → If you apply SMOTE before splitting, synthetic samples are generated using information from the entire dataset, including the test set — which causes leakage.

Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- Detect and Handle Outliers
- Deal with Imbalanced classes
- Feature Scaling
- **Work with Categorical data**
- Split data to Train and Test Sets

- nominal > غير مرتبة >> one hot encoding, binary encoding
- ordinal >> مرتبة >> label encoding

Work with Categorical ordinal data (Label Encoder)

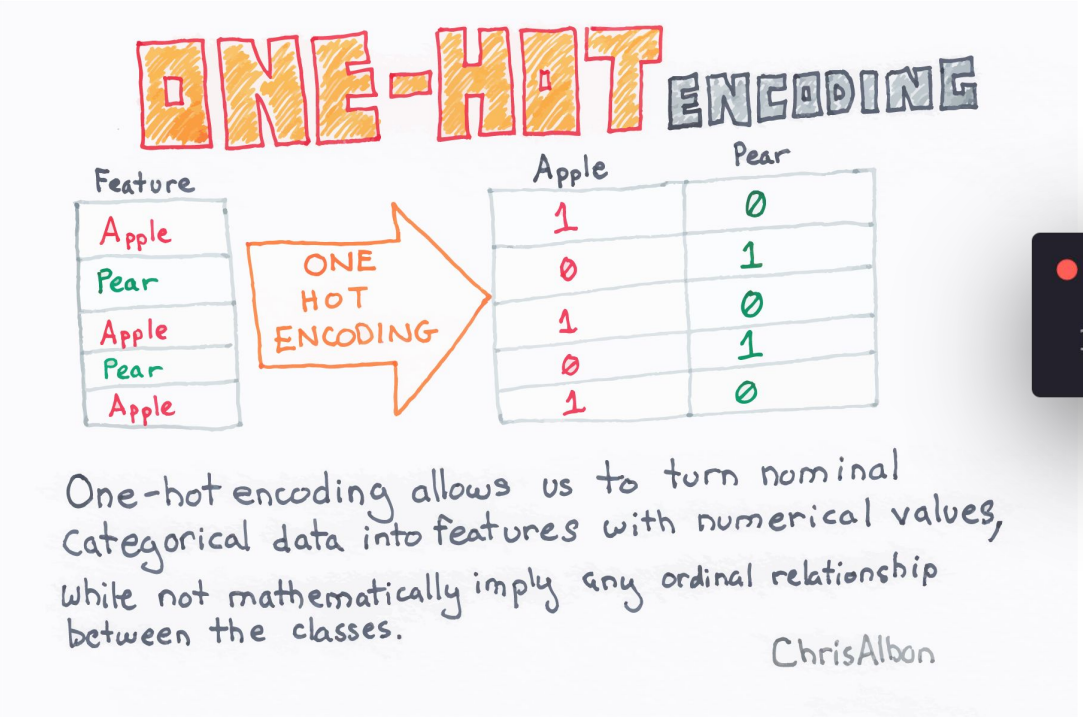
- Label encoding converts the data in machine-readable form, but it assigns a unique number(starting from 0) to each class of data , in a sequence ordinal manner.
- To use label encoding, your data should must have inherent order in its logic
- We can’t use it with nominal data (non – ordered).
- An attribute having output classes Mexico, Paris, Dubai. On Label Encoding, this column lets Mexico is replaced with 0, Paris is replaced with 1, and Dubai is replaced with 2.
- With this, it can be interpreted that Dubai has high priority than Mexico and Paris while training the model, But actually, there is no such priority relation between these cities here.

```
size_dict = {'XS':1,
             'S':2,
             'M':3,
             'L':4,
             'XL':5,
             'XXL':6}

# apply using map
df['Size'] = df['Size'].map(size_dict)
```

- **Work with Ordinal Features with pandas map method.**

Work with Categorical Nominal data (ONE-HOT Encoder / Binary Encoding)

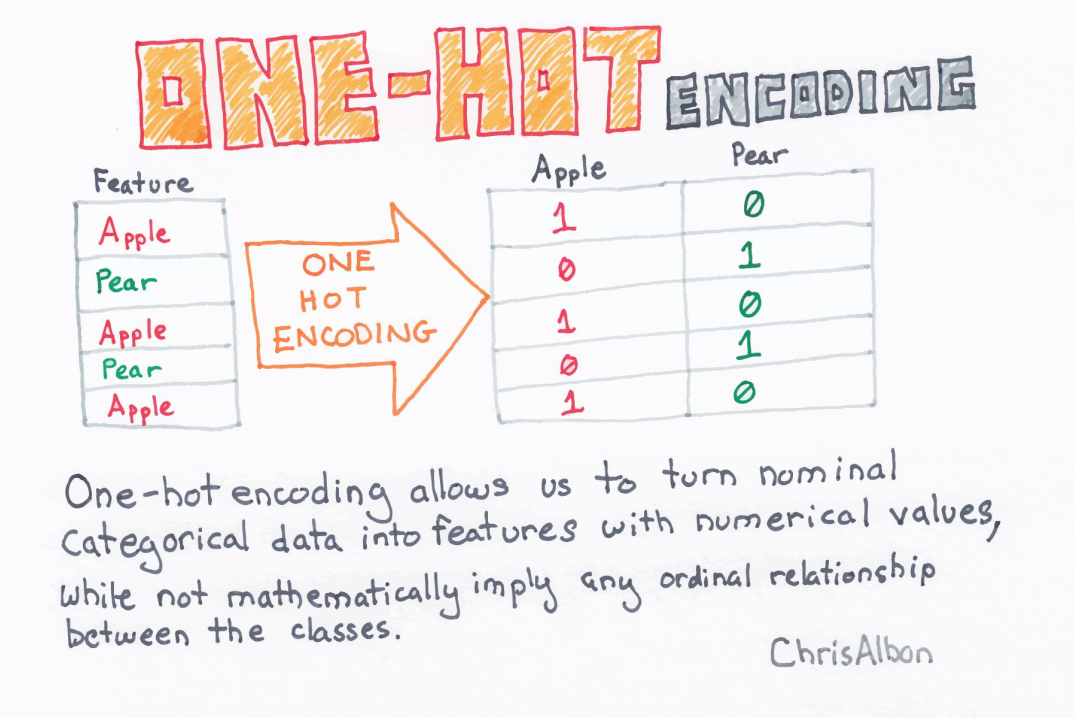



```python
df = pd.get_dummies(df, columns=['Color', 'Brand'], drop_first=True)
```

- Work with Ordinal Features with pandas `map` method.
- **Work with Nominal Features with pandas `get_dummies` method.**
  - Work with Nominal Features with sklearn `OneHotEncoder` method.

Eng: Baraa slides

Work with Categorical Nominal data (ONE-HOT Encoder / Binary Encoding)



- Work with Ordinal Features with pandas `map` method.
- Work with Nominal Features with pandas `get_dummies` method.
- **Work with Nominal Features with sklearn `OneHotEncoder` method.**

Binary Encoding for Nominal Data

- The categories are encoded in Ordinal Numeric form.
- Then those integers are converted into binary code, so for example 5 becomes 101 and 10 becomes 1010
- Then the digits from that binary string are split into separate columns.
- Each observation is encoded across the columns in its binary form.
- Example: if we have Country Feature with 16 different countries:
  - By One-Hot Encoding it will result in 16 columns.
  - By Binary Encoding, it will result only in 4 columns

```
X = pd.DataFrame(list('abcdefghijklmnop'))
X
```

	0
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

```
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(sparse=False)
countries= ohe.fit_transform(X.values.reshape(-1,1)).astype(int)
pd.DataFrame(countries)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

```
import category_encoders as ce

ce_bin = ce.BinaryEncoder()
ce_bin.fit_transform(X)
```

	0_0	0_1	0_2	0_3	0_4
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	1	0	0
4	0	0	1	0	1
5	0	0	1	1	0
6	0	0	1	1	1
7	0	1	0	0	0
8	0	1	0	0	1
9	0	1	0	1	0
10	0	1	0	1	1
11	0	1	1	0	0
12	0	1	1	0	1
13	0	1	1	1	0
14	0	1	1	1	1
15	1	0	0	0	0

10 class > one hot > 10 new clo

2^ 4 = 10

1 2 4 8 16

0 1 0 1 >>10

Eng: Baraa slides

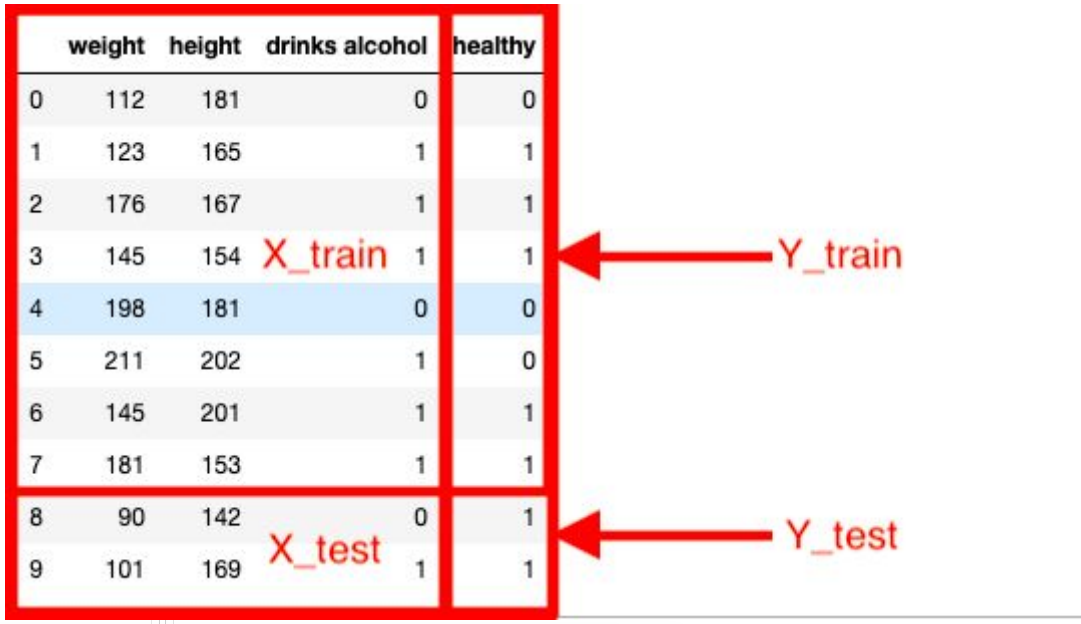
Feature Transformations

- Data Cleaning or Cleansing
- Work with Missing data
- Detect and Handle Outliers
- Deal with Imbalanced classes
- Feature Scaling
- Work with Categorical data
- **Split data to Train and Test Sets**

Split data to Train and Test Sets

When you're working on a model and want to train it, you obviously have a dataset. But after training, we have to test the model on some test dataset.

the obvious solution is to split the dataset you have into two sets, one for training and the other for testing; and you do this before you start training your model.




```python
from sklearn.model_selection import train_test_split

x = df.drop('healthy', axis=1)
y = df['healthy']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)
```

Eng: Baraa slides

first 90% form data with label “Yes”
Next 10% form data with label “No”

split 80/20
>>>80% هيتدرب على yes فقط ؟؟ انو مشافش no وهو بيتدرب
>> 20% نصهم No فبالتالي المودل مش هيقدر يكتشفهم ليش؟ لانو ما شافهم اثناء التدريب

>> shuffle = Ture

train > 90% yes, 10% no
test > 90% yes, 10% no

stratify = y

Sampling techniques in Train and Test Sets

Important in **imbalanced data.**

Parameter of train_test_split:

1. Shuffle

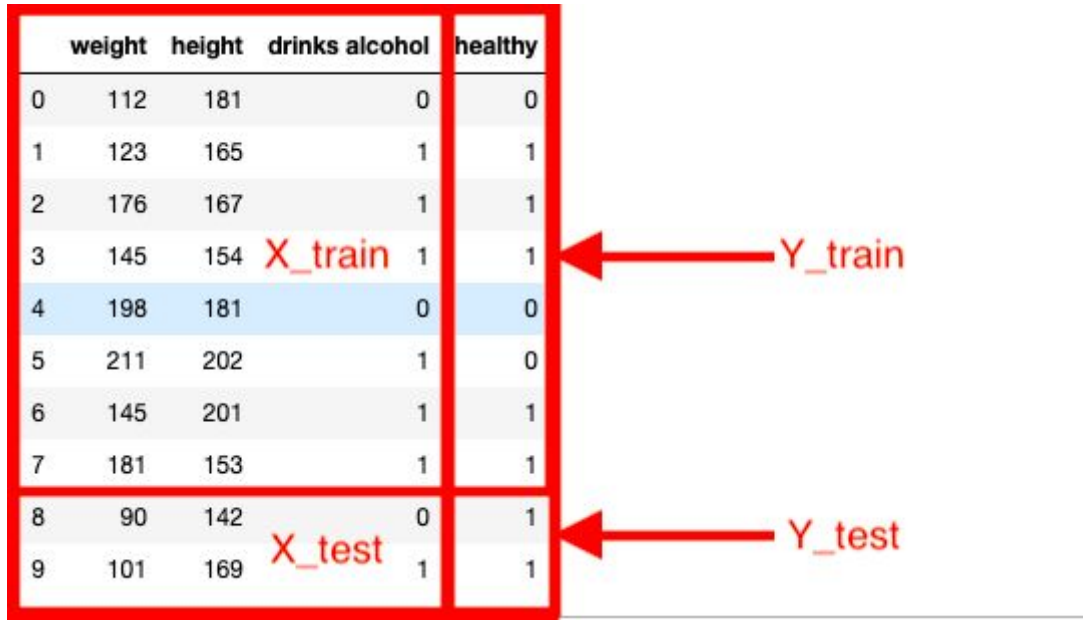
With shuffle=True, data is shuffled then the split happens.

2. Stratify

Makes a split so that a proportion of the values in the sample produced will be the same as the proportion of values provided to parameter stratify.

3. Random state (random_state)

If added you ensure that the split remains the same whenever you run the code




```
from sklearn.model_selection import train_test_split

x = df.drop('healthy', axis=1)
y = df['healthy']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2
                                                    , shuffle =True, stratify = y)
```

Eng: Baraa slides