

Data Science

Exploration & Cleaning

Zeham Management Technologies BootCamp by SDAIA

July 22nd, 2024



SDAIA

الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority

Introduction to Data Analysis

Let's start together...





Agenda



Data Analysis Workflow



Data Types Handling



Data Exploration



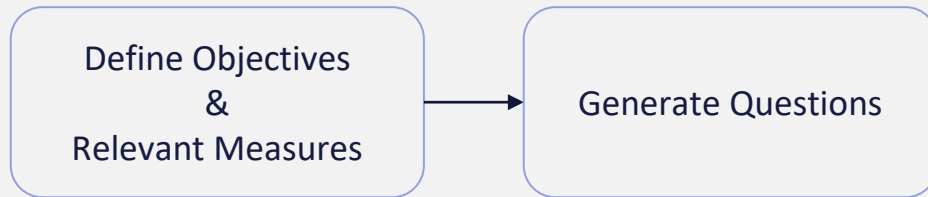
Data Analysis Workflow

Data Analysis Flow

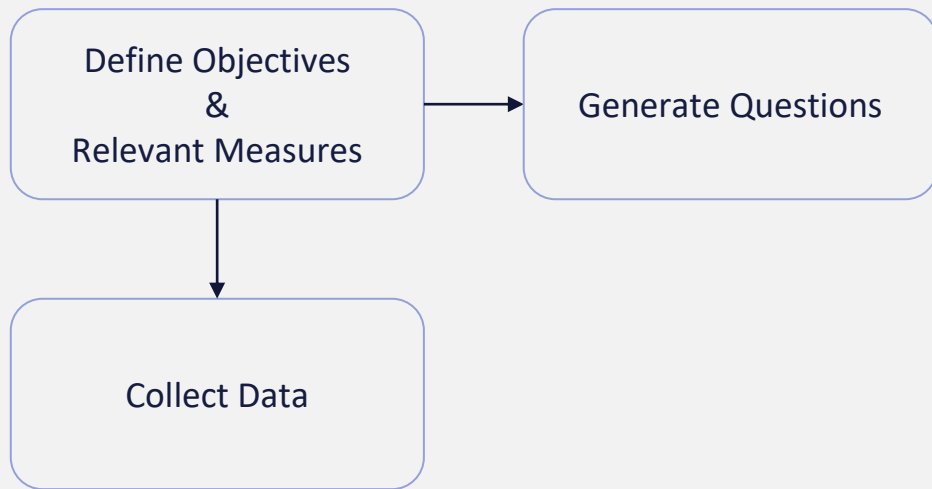
Define Objectives
&
Relevant Measures



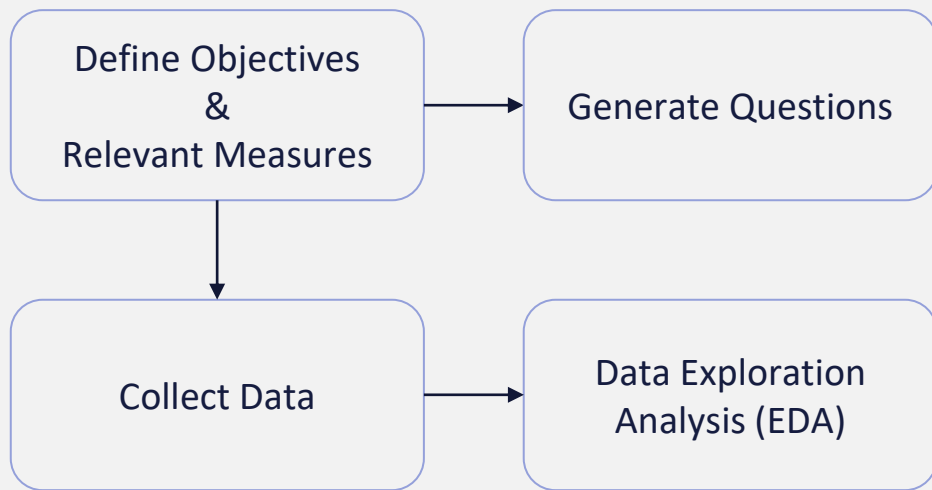
▶ Data Analysis Flow



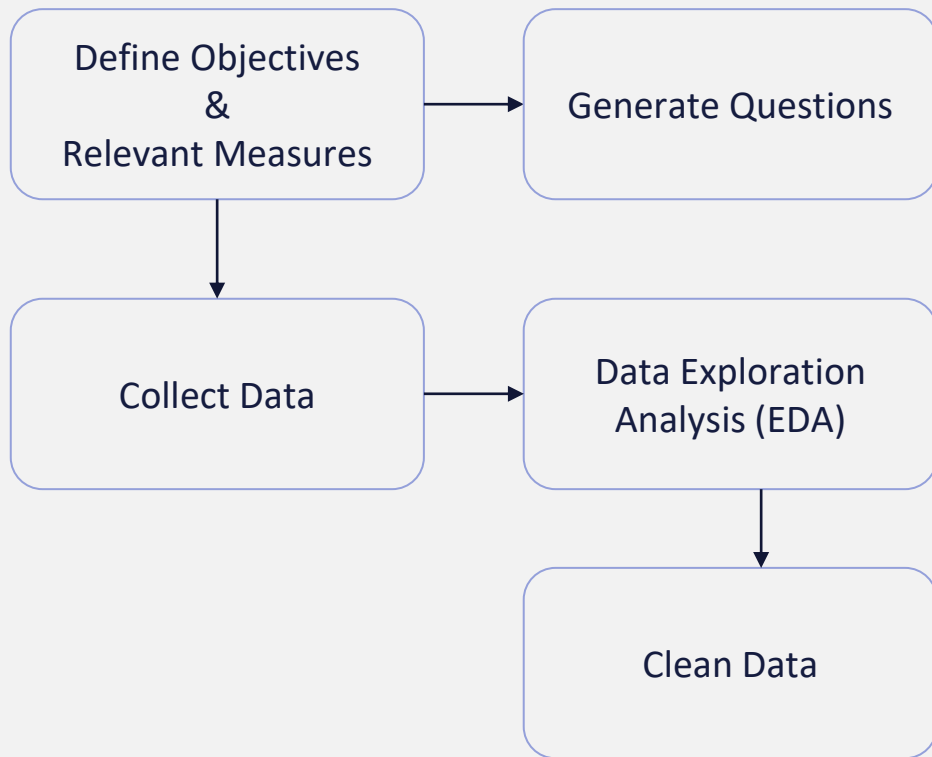
► Data Analysis Flow



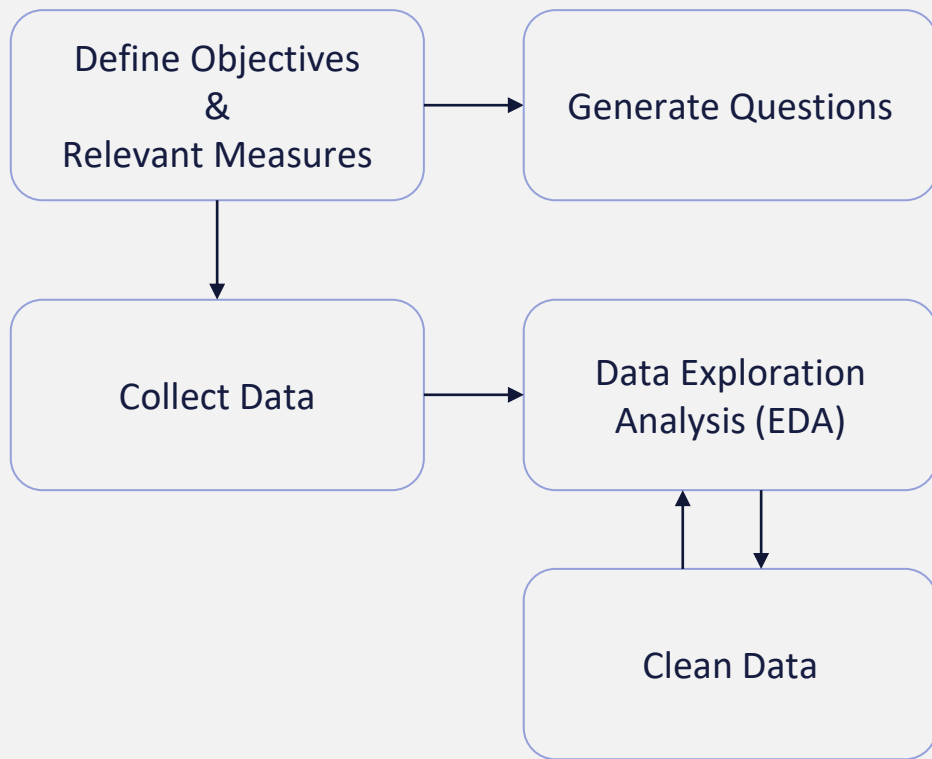
► Data Analysis Flow



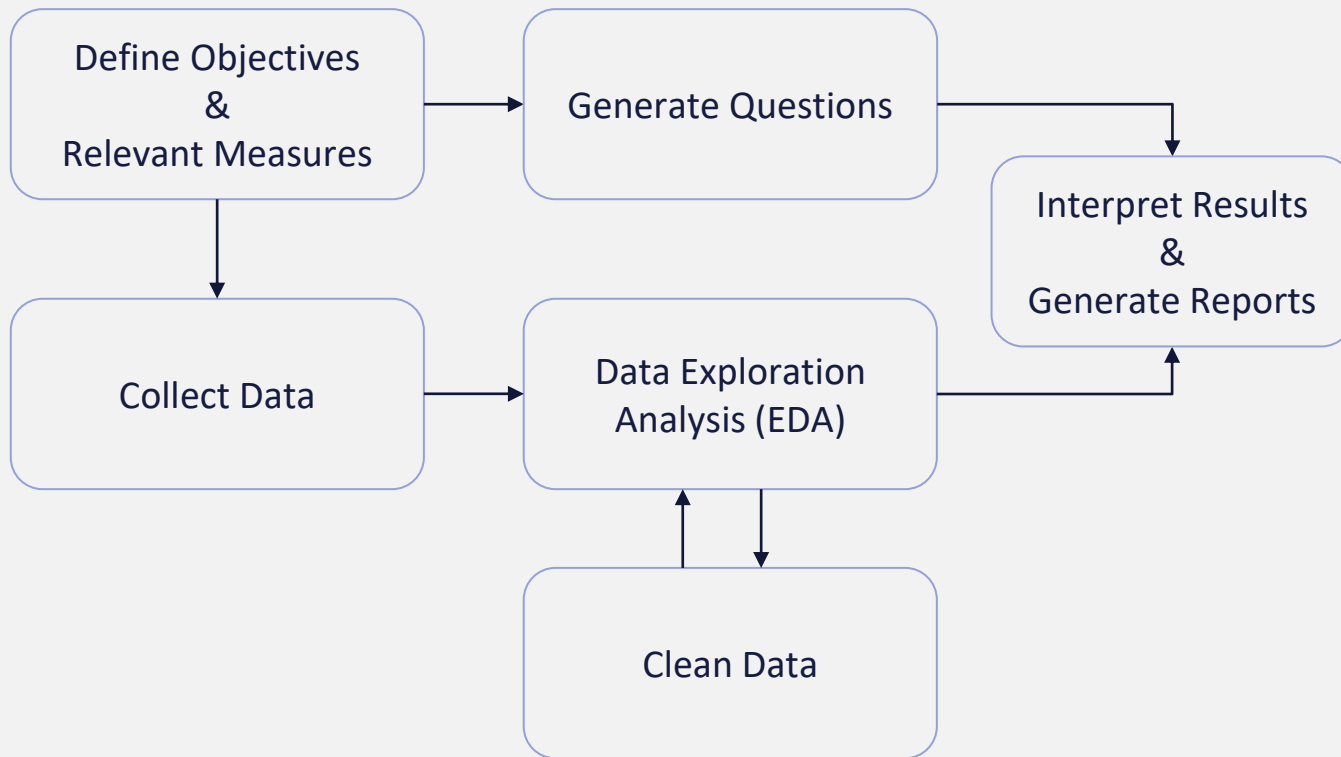
► Data Analysis Flow



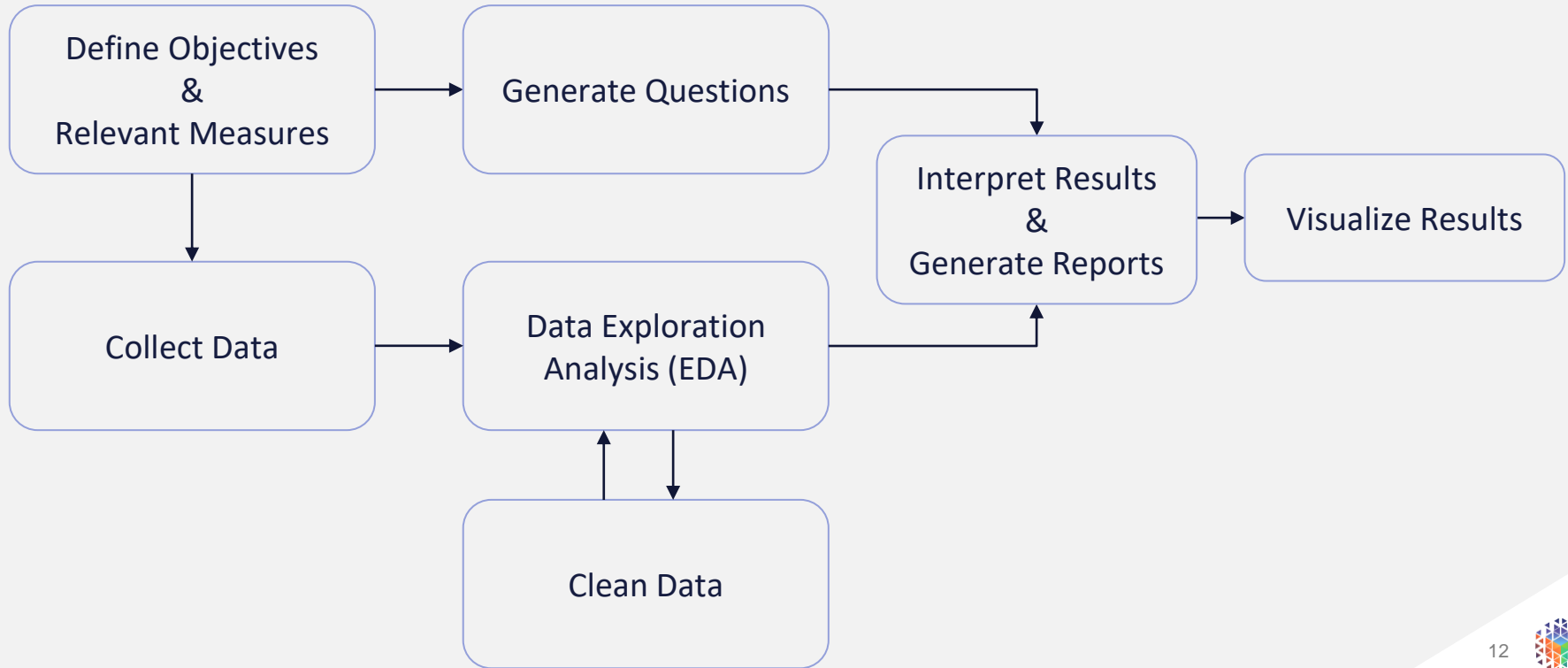
► Data Analysis Flow



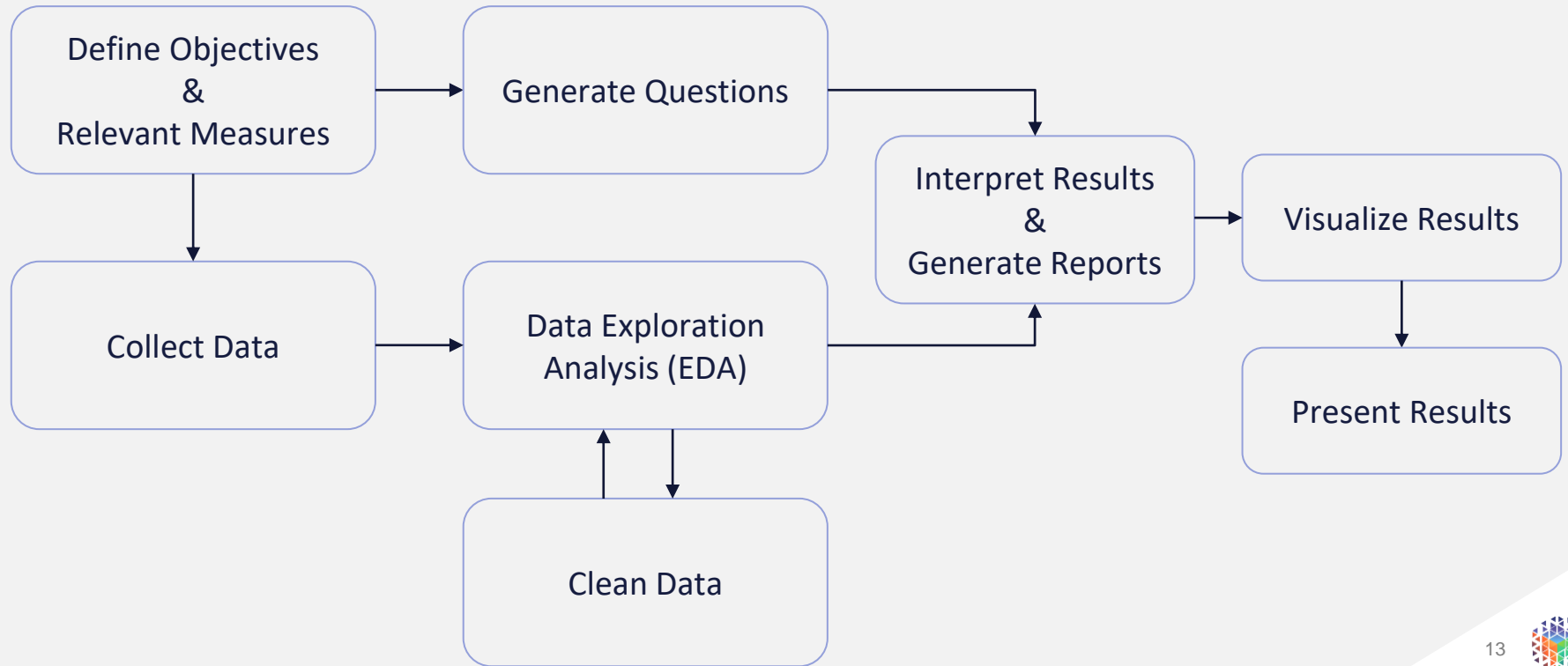
► Data Analysis Flow



► Data Analysis Flow



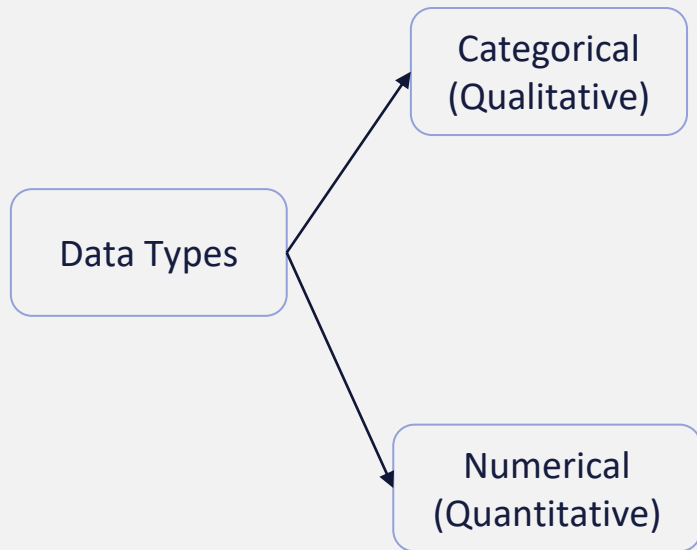
▶ Data Analysis Flow



Data Types

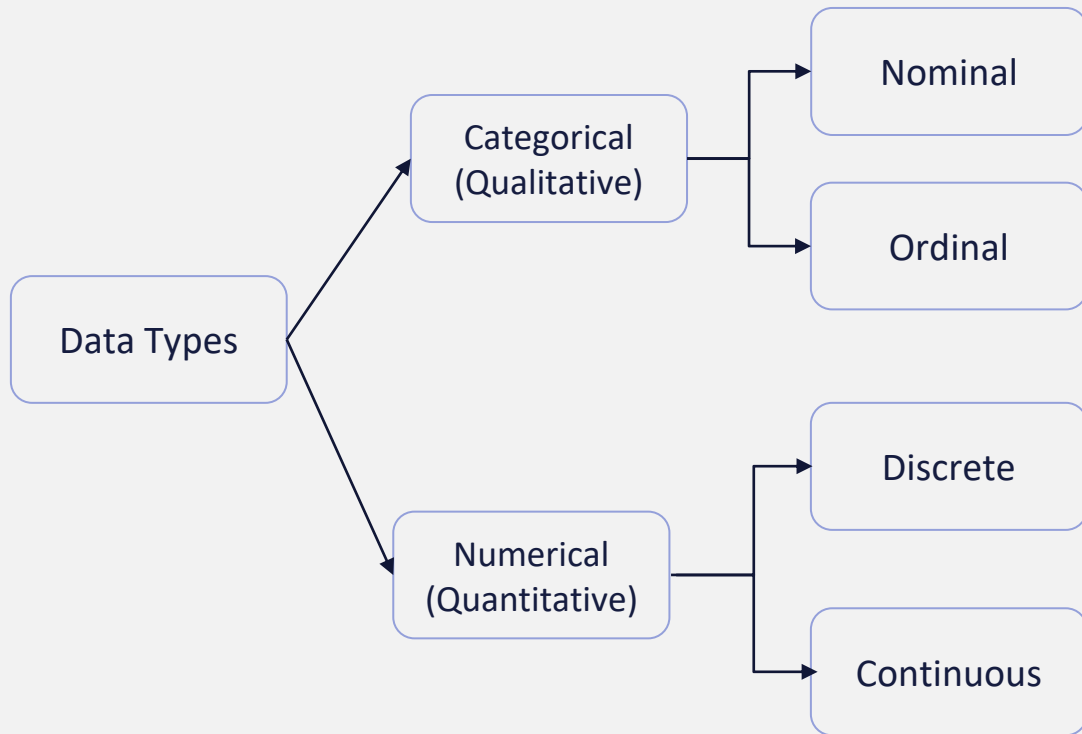
Data Types

Data types Can be broadly be categorized into two groups



Data Types

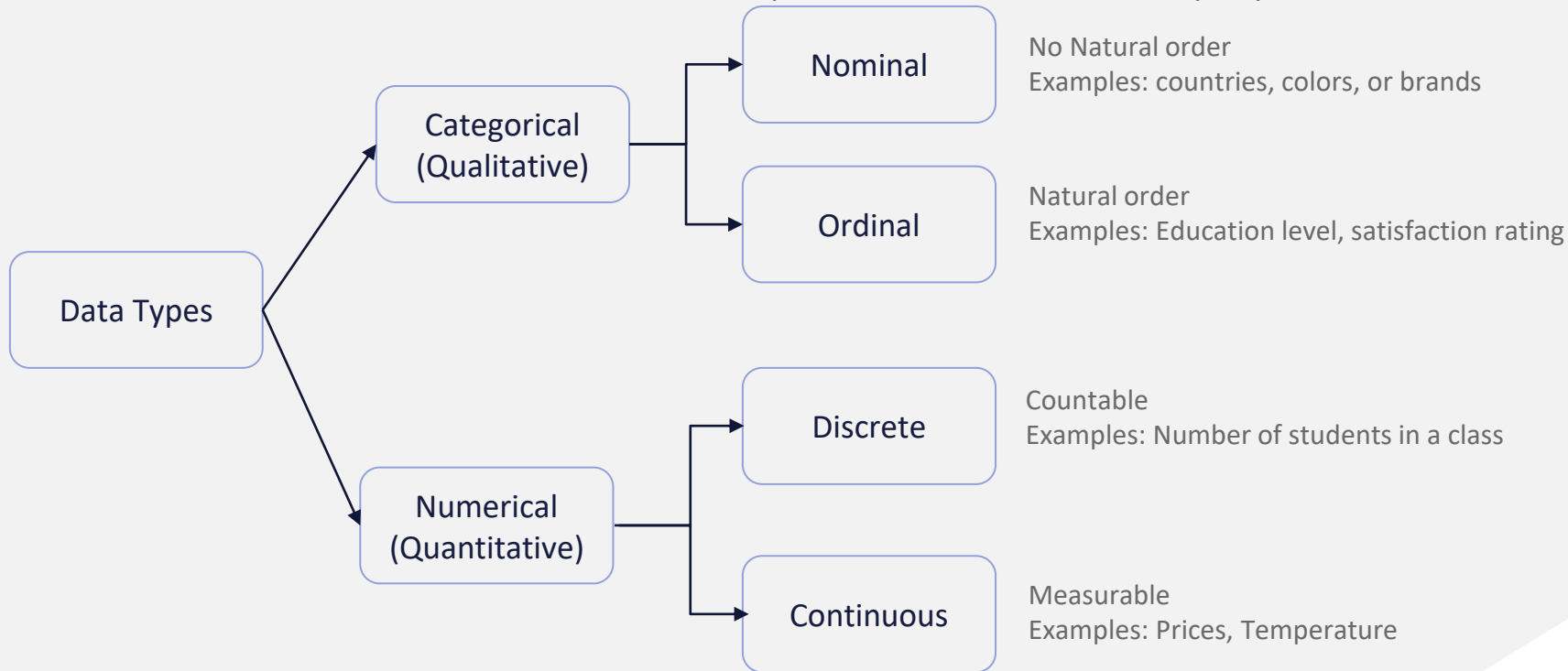
Data types Can be broadly be categorized into two groups and each of these groups can further be divided



Data Types

Data types Can be broadly be categorized into two groups and each of these groups can further be divided

Note that there are other divisions that will be explained later from a statistical perspective



► Special Data Types

There are some data types that need particular care while handling it, The following are two such types:

- **Time Series Data:** Clarify that time series data is not just any data that includes times or dates but specifically refers to data points collected or recorded at regular time intervals. This could be financial data, weather data, or something like the number of daily visits to a website.
- **Dates:** Point out that while dates can be part of time series data, they also stand alone in many datasets as significant markers or features. Handling dates properly can unlock valuable insights, such as seasonal trends or age calculations.

How to handle each of these will be explained further down the road for now it's worth noting that they need special care due to there unique nature

Table 1: Example of Time Series Data

Date	Week	Sales
1/4/2010	1	96,200
1/11/2010	2	98,491
1/18/2010	3	96,595
1/25/2010	4	93,511
...
12/5/2011	101	80,407
12/12/2011	102	81,479
12/19/2011	103	83,093
12/26/2011	104	72,752



Data Exploration

Data Exploration

Data Exploration involves using visual and quantitative methods to understand a dataset.

Methods and Techniques:

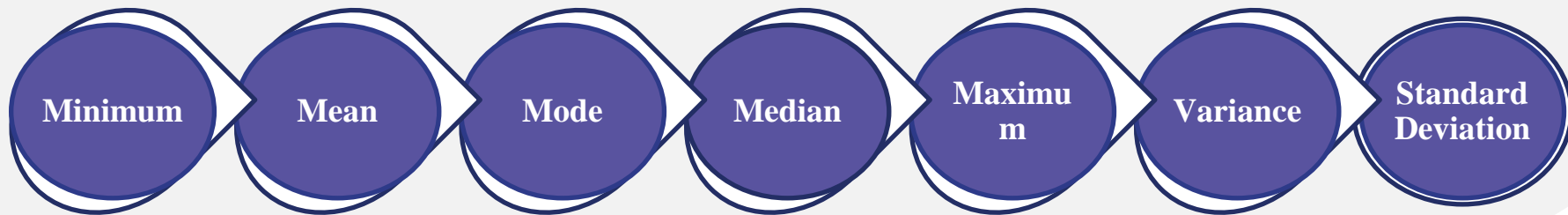
- **Summary Statistics:** Calculating mean, median, mode, etc., to understand data distribution.
- **Visualization:** Using histograms, box plots, scatter plots, etc., to visually explore data.
- **Correlation Analysis:** Identifying relationships between variables.
- **Dimensionality Reduction:** Reducing the number of variables with techniques like PCA (will be explained later) to study feature importance.



► Data Summarization

Summary statistics are numerical measures that capture key characteristics of a dataset, simplifying large amounts of data into smaller, informative summaries. They are crucial for understanding the distribution and central tendencies of the data.

Here are the primary summary statistics (More details can be found in the descriptive statistics presentation):



Correlation Analysis

Correlation analysis involves quantifying the degree to which two variables are related. It helps in understanding how one variable may change in relation to another.

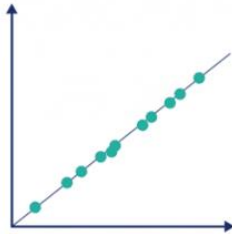
Key points:

- **Correlation Coefficient:** From -1 (strong negative) to 1 (strong positive), with 0 indicating no linear relationship.
- **Methods:** Common methods include Pearson correlation (measures linear correlation) and Spearman's rank correlation (for non-linear relationships).
- **Usage:** Used to identify potential associations or to suggest hypotheses for further study.

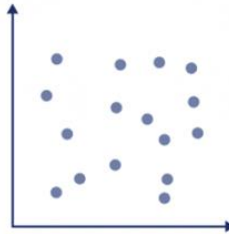


Correlation Analysis

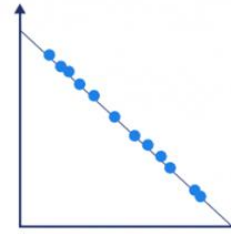
Perfect positive correlation



Zero correlation



Perfect negative correlation



► Data Exploration using pandas (dataset info)

First, we can use the `.info()` method to get a concise summary of the Data Frame. This method provides essential information about the dataset, including the number of non-null entries, column data types, and memory usage. This helps us understand the basic structure and completeness of the data, making it easier to identify any missing values or data types that may need to be adjusted for further analysis.



► Data Exploration using pandas (dataset info)

```
1 import pandas as pd
2 df = pd.read_csv('Traffic.csv')
3 df.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5952 entries, 0 to 5951
Data columns (total 9 columns):
#   Column              Non-Null Count  Dtype
---  -
0   Time                5952 non-null   object
1   Date                5952 non-null   int64
2   Day of the week     5952 non-null   object
3   CarCount            5952 non-null   int64
4   BikeCount           5952 non-null   int64
5   BusCount            5952 non-null   int64
6   TruckCount          5952 non-null   int64
7   Total               5952 non-null   int64
8   Traffic Situation   5952 non-null   object
dtypes: int64(6), object(3)
memory usage: 418.6+ KB
```




▶ Data Exploration using pandas (display data)

To explore data, you will usually start with viewing the data. In pandas, to view the data there are multiple methods to use:

- **.head():** This method shows the first 5 rows by default and to change the number of rows displayed you can pass the desired number.
- **.tail():** This method shows the last 5 rows instead and accepts the desired number of rows.
- **.sample():** This method shows a random row as a sample from the dataset, you can change it to display more samples by passing the desired number of samples as argument.

As you can see in the example we used `.head()` to read the dataset and get an idea.

```
1 import pandas as pd
2 df = pd.read_csv('Traffic.csv')
3 df.head()
```



	Time	Date	Day of the week	CarCount	BikeCount	BusCount	TruckCount	Total	Traffic Situation
0	12:00:00 AM	10	Tuesday	13	2	2	24	41	normal
1	12:15:00 AM	10	Tuesday	14	1	1	36	52	normal
2	12:30:00 AM	10	Tuesday	10	2	2	32	46	normal
3	12:45:00 AM	10	Tuesday	10	2	2	36	50	normal
4	1:00:00 AM	10	Tuesday	11	2	1	34	48	normal





Data Exploration using pandas (understanding the dataset)

By looking at the data we can see that it have 9 columns with counts of the cars at a specific time along with the traffic situation at the time of capturing the data.

This dataset appears to be a **time series** data as you can see it is recorded at regular time intervals which is every 15 minutes.

	Time	Date	Day of the week	CarCount	BikeCount	BusCount	TruckCount	Total	Traffic Situation
0	12:00:00 AM	10	Tuesday	13	2	2	24	41	normal
1	12:15:00 AM	10	Tuesday	14	1	1	36	52	normal
2	12:30:00 AM	10	Tuesday	10	2	2	32	46	normal
3	12:45:00 AM	10	Tuesday	10	2	2	36	50	normal
4	1:00:00 AM	10	Tuesday	11	2	1	34	48	normal



► Data Exploration using pandas (data summary)

This is not enough to deeply understand the data. We can use also `.describe()` method to get statistical summary. This method provides summary statistics of the numerical columns in the dataset, including **count**, **mean**, **standard deviation**, **min**, **max**, and the **25th**, **50th (median)**, and **75th percentiles**.

Percentiles are statistical measures that indicate the relative position of a value within a dataset. For example, the 25th percentile is the value below which 25% of the data falls, while the 50th percentile (median) is the midpoint of the data. They provide insight into the distribution and spread of the data.




	Date	CarCount	BikeCount	BusCount	TruckCount	Total
count	5952.000000	5952.000000	5952.000000	5952.000000	5952.000000	5952.000000
mean	16.000000	65.440692	12.161458	12.912970	18.646337	109.161458
std	8.945023	44.749335	11.537944	12.497736	10.973139	55.996312
min	1.000000	5.000000	0.000000	0.000000	0.000000	21.000000
25%	8.000000	18.750000	3.000000	2.000000	10.000000	54.000000
50%	16.000000	62.000000	9.000000	10.000000	18.000000	104.000000
75%	24.000000	103.000000	19.000000	20.000000	27.000000	153.000000
max	31.000000	180.000000	70.000000	50.000000	60.000000	279.000000



► Data Exploration using pandas (data correlation)

To understand the relationships between numerical variables in the dataset, the correlation matrix reveals how variables like CarCount, BikeCount, BusCount, TruckCount, and Total are related, with values close to 1 or -1 indicating strong correlations.

```
1 numerical_columns = df[['CarCount', 'BikeCount', 'BusCount', 'TruckCount', 'Total']]
2
3 correlation_matrix = numerical_columns.corr()
4
5 print(correlation_matrix)
```



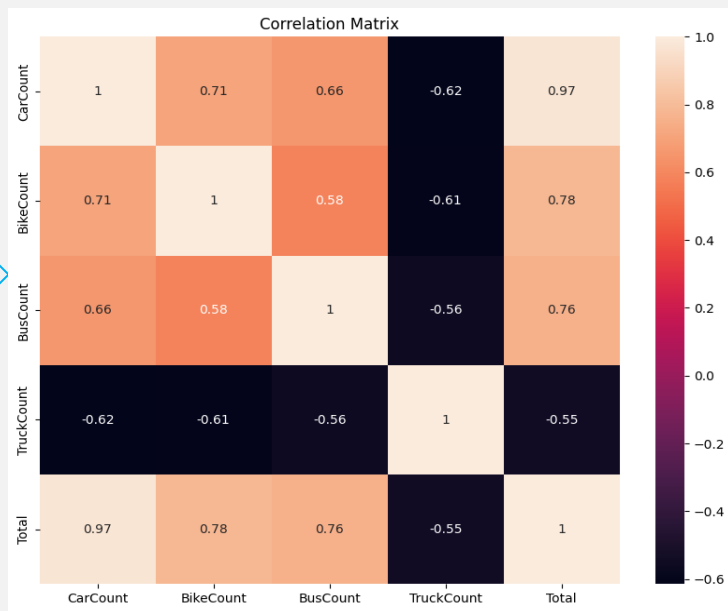
	CarCount	BikeCount	BusCount	TruckCount	Total
CarCount	1.000000	0.708243	0.658417	-0.615043	0.971507
BikeCount	0.708243	1.000000	0.577671	-0.607720	0.781879
BusCount	0.658417	0.577671	1.000000	-0.558372	0.758970
TruckCount	-0.615043	-0.607720	-0.558372	1.000000	-0.545390
Total	0.971507	0.781879	0.758970	-0.545390	1.000000



► Data Exploration using pandas (data correlation) cont.

To represent the correlation matrix, we can use a heatmap to visually display the strength and direction of relationships between numerical variables, with colors indicating the magnitude of the correlations.

```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3
4 plt.figure(figsize=(10, 8))
5 sns.heatmap(correlation_matrix, annot=True)
6 plt.title('Correlation Matrix')
7 plt.show()
```



Practice

Dataset Path:

2- Data Collection with Python:

`/LAB/Datasets/Traffic.csv`

Notebook Path:

2- Data Collection with Python

`/LAB/exploratory_data_analysis.ipynb`



Data Science

Data selection and indexing



SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority



Agenda



Introduction to data selection and indexing



Basic data selection techniques



Advanced indexing techniques



Setting and resetting index



Introduction to Data Selection and Indexing

► Understanding Data Selection

Data selection is the process of choosing a subset of a larger dataset, based on certain criteria, for further analysis.

Why is data selection important ?

- ❑ It reduces processing time and improves efficiency.
- ❑ It enhances the accuracy of models by focusing on relevant data.



Basic Data Selection Techniques

Basic Data Selection Techniques

For simple data selection in pandas, we use:

- **Column selection:** Get a single column with `df['column_name']` or more than one with `df[['column1', 'column2']]`.
- **Row selection using indices:** Get rows by their number using `df.iloc[index]` or by name with `df.loc[label]`.
- **Combining selections:** Choose specific rows and columns together, for example, `df.loc[rows, 'column']`. These steps are the foundation for working with data in Python.





Basic Data Selection Techniques (Column Selection)

An example to column selection:

Single Column Selection

```
1 df['City']
```

```
0    Abudhabi
1    Abudhabi
2    Abudhabi
3    Abudhabi
4    Abudhabi
...
66634    Riyadh
66635    Riyadh
66636    Riyadh
66637    Riyadh
66638    Riyadh
Name: City, Length: 66639, dtype: object
```

Multiple Column Selection

```
1 df[['City', 'JamsCount', 'JamsDelay']]
```

	City	JamsCount	JamsDelay
0	Abudhabi	4	15.6
1	Abudhabi	7	20.5
2	Abudhabi	8	25.0
3	Abudhabi	11	30.6
4	Abudhabi	20	62.1
...
66634	Riyadh	33	127.8
66635	Riyadh	27	87.0
66636	Riyadh	17	49.8
66637	Riyadh	16	61.3
66638	Riyadh	15	39.5

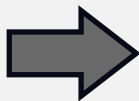




Basic Data Selection Techniques (Row Selection)

First method of **Row selection** is to slice a specific row by its integer location:

```
1 df.iloc[86]
```



```
City                Abudhabi
Datetime            2023-07-10 22:01:00
TrafficIndexLive    6
JamsCount           9
JamsDelay           27.5
JamsLength          2.5
TrafficIndexWeekAgo 7
TravelTimeHistoric  58.912254
TravelTimeLive      57.438891
Name: 86, dtype: object
```





Basic Data Selection Techniques (Row Selection)

The second method of **Row selection** is to slice multiple rows by their integer locations:

```
1 df.iloc[86:89]
```



	City	Datetime	TrafficIndexLive	JamsCount	JamsDelay	JamsLength	TrafficIndexWeekAgo	TravelTimeHistoric	TravelTimeLive
86	Abudhabi	2023-07-10 22:01:00	6	9	27.5	2.5	7	58.912254	57.438891
87	Abudhabi	2023-07-10 23:01:30	4	7	35.7	3.3	4	58.009904	55.515934
88	Abudhabi	2023-07-11 00:01:30	3	1	0.8	0.6	3	55.952296	53.245858





Basic Data Selection Techniques (Row Selection)

The third method of **Row selection** is to locate using Label:

```
1 df.set_index(['City', 'Datetime'], inplace=True)
2
3 df.loc[('Abudhabi', '2023-07-07 08:01:30')]
```



```
TrafficIndexLive      6.000000
JamsCount              4.000000
JamsDelay             15.600000
JamsLength             0.700000
TrafficIndexWeekAgo   13.000000
TravelTimeHistoric     59.611918
TravelTimeLive        54.803617
Name: (Abudhabi, 2023-07-07 08:01:30), dtype: float64
```



Basic Data Selection Techniques (Combining selections)

An example to combining selections:

```
1 df.loc[86:88, ['City', 'JamsCount', 'JamsDelay']]
```



	City	JamsCount	JamsDelay
86	Abudhabi	9	27.5
87	Abudhabi	7	35.7
88	Abudhabi	1	0.8



Advanced Indexing Techniques

▶ Advanced Indexing Techniques

Advanced indexing includes:


- **Conditional selection:** Filter rows by setting conditions, like `df[df['column'] > value]`.
- **Using the `.query()` method:** A simpler way to write conditions, like `df.query('column > value')`.
- **Multi-level selection:** Useful for complicated data, allowing you to choose by several conditions at once.



▶ Advanced Indexing Techniques (Conditional Selection)

An example to conditional selections:

```
1 df[df['JamsCount'] > 1200]
```



	City	Datetime	TrafficIndexLive	JamsCount	JamsDelay	JamsLength	TrafficIndexWeekAgo	TravelTimeHistoric	TravelTimeLive
36472	Dubai	2023-10-26 17:46:00	115	1292	9400.3	1173.9	56	73.680174	120.189364
36473	Dubai	2023-10-26 18:31:00	138	1359	9989.4	1171.1	70	77.218820	133.372646



Advanced Indexing Techniques (Query Method)

An example to query method:

```
1 df.query('JamsCount > 1200')
```



	City	Datetime	TrafficIndexLive	JamsCount	JamsDelay	JamsLength	TrafficIndexWeekAgo	TravelTimeHistoric	TravelTimeLive
36472	Dubai	2023-10-26 17:46:00	115	1292	9400.3	1173.9	56	73.680174	120.189364
36473	Dubai	2023-10-26 18:31:00	138	1359	9989.4	1171.1	70	77.218820	133.372646

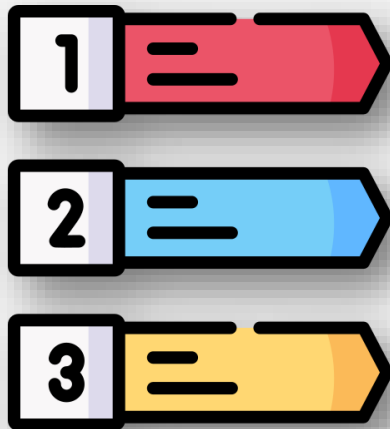


► Understanding Indexing

Indexing is the organization of data according to a specific schema or plan to facilitate quicker searches and retrieval.

Why is indexing important ?

- ❑ Speeds up data retrieval.
- ❑ Essential for performing efficient queries.



Setting and Resetting Index



Setting and Resetting Index

Indexes help organize your data:

- **Setting a column as an index:** Make a column into an index for faster searches and data combinations, like `df.set_index('column_name')`.
- **Resetting to the default index:** Change back to the default number sequence, useful when reorganizing the index, like `df.reset_index()`.



► Setting and Resetting Index (Setting a column as an index)

An example to setting column as an index:

```
1 df.set_index('Datetime', inplace=True)
2 df.head()
```



	City	TrafficIndexLive	JamsCount	JamsDelay	JamsLength	TrafficIndexWeekAgo	TravelTimeHistoric	TravelTimeLive
Datetime								
2023-07-07 08:01:30	Abudhabi	6	4	15.6	0.7	13	59.611918	54.803617
2023-07-07 09:01:30	Abudhabi	7	7	20.5	1.7	8	60.221387	56.118629
2023-07-07 10:46:30	Abudhabi	7	8	25.0	2.8	6	59.161978	55.518834
2023-07-07 11:16:30	Abudhabi	8	11	30.6	5.5	6	59.738138	56.413917
2023-07-07 12:01:30	Abudhabi	8	20	62.1	6.5	5	58.958314	56.059246



▶ Setting and Resetting Index (Setting a column as an index)

An example to setting multiple column as an index:

```
1 df.set_index(['Datetime', 'JamsLength'], inplace=True)
2 df.head()
```



			City	TrafficIndexLive	JamsCount	JamsDelay	TrafficIndexWeekAgo	TravelTimeHistoric	TravelTimeLive
Datetime	JamsLength								
2023-07-07 08:01:30	0.7	Abudhabi	6	4	15.6	13	59.611918	54.803617	
2023-07-07 09:01:30	1.7	Abudhabi	7	7	20.5	8	60.221387	56.118629	
2023-07-07 10:46:30	2.8	Abudhabi	7	8	25.0	6	59.161978	55.518834	
2023-07-07 11:16:30	5.5	Abudhabi	8	11	30.6	6	59.738138	56.413917	
2023-07-07 12:01:30	6.5	Abudhabi	8	20	62.1	5	58.958314	56.059246	



▶ Setting and Resetting Index (Resetting to the default index)

An example to setting multiple column as an index:

```
1 df.reset_index(inplace=True)
2 df.head()
```



	index	Datetime	JamsLength	City	TrafficIndexLive	JamsCount	JamsDelay	TrafficIndexWeekAgo	TravelTimeHistoric	TravelTimeLive
0	0	2023-07-07 08:01:30	0.7	Abudhabi	6	4	15.6	13	59.611918	54.803617
1	1	2023-07-07 09:01:30	1.7	Abudhabi	7	7	20.5	8	60.221387	56.118629
2	2	2023-07-07 10:46:30	2.8	Abudhabi	7	8	25.0	6	59.161978	55.518834
3	3	2023-07-07 11:16:30	5.5	Abudhabi	8	11	30.6	6	59.738138	56.413917
4	4	2023-07-07 12:01:30	6.5	Abudhabi	8	20	62.1	5	58.958314	56.059246



Practice

DataSet Path :

2- Data Collection with Python

`/LAB/Datasets/Traffic.csv`

Notebook Path :

2- Data Collection with Python

`/LAB/selection_indexing.ipynb`



Data Science

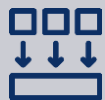
Grouping and Aggregation



SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority



Agenda



Grouping and Aggregation



Grouping with pandas



Aggregation with pandas



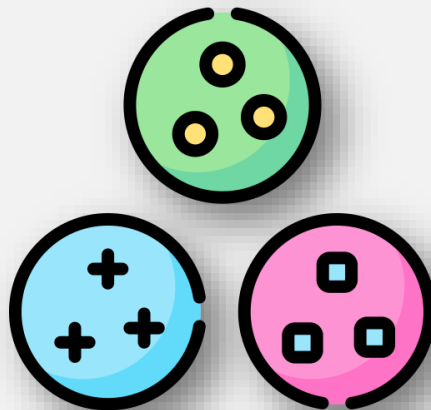
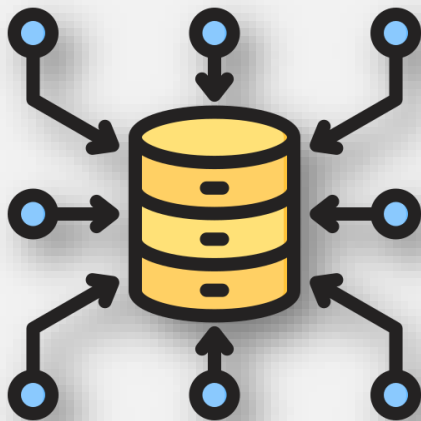
Using Grouping with Aggregation in pandas



Grouping and Aggregation

► Grouping and Aggregation

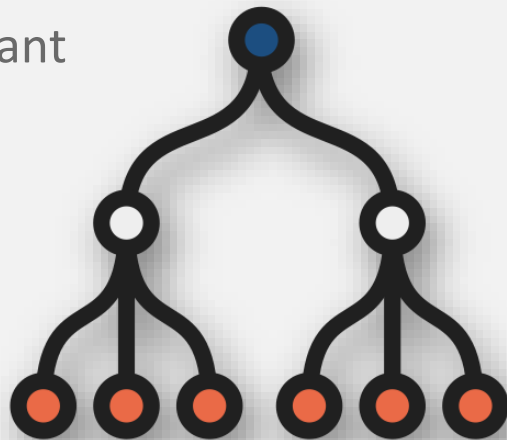
Grouping and aggregation are methods used in data analysis to organize and summarize large amounts of data. When we group data, we collect data points together based on shared features.



► Grouping and Aggregation (Benefits)

These methods help us manage large datasets by breaking them down into smaller, more manageable pieces and then summarizing them to find important information.

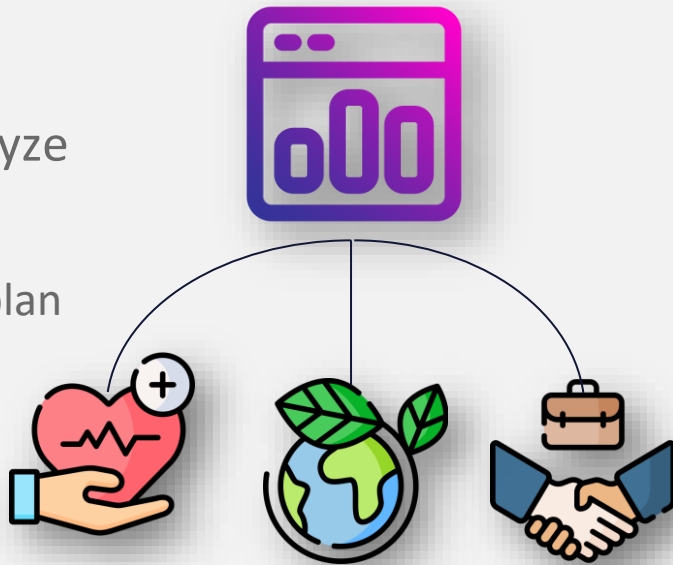
- Makes large data sets simpler to handle.
- Improves the accuracy of our analysis.
- Helps compare different groups of data easily.



► Grouping and Aggregation (Examples)

These methods are used in various fields to analyze data and make decisions based on that data.

- **Business:** Calculating average sales by region to plan better marketing.
- **Healthcare:** Summing up patient data to find common treatment results.
- **Environment:** Organizing decades of climate data to see temperature changes.



Grouping with pandas

▶ Grouping with pandas

In pandas, grouping involves organizing data into categories based on some criteria. This lets us analyze subsets of data separately.



▶ Grouping with pandas

When using pandas, we decide on the key criteria for grouping based on the questions we're trying to answer with our data.

- Choose columns that align with the objectives of your analysis.
- Ensure the groups are meaningful and statistically valid.
- Consider the size and distribution of groups to avoid skewed results.



► Grouping with pandas (Multi-level)

Multi-level grouping is about organizing data into groups within groups. This approach helps us see the data at several levels and get more detailed insights. For example, you might take more that column to analyze the company's sales. Grouping by the 'state', 'city', and 'store_type' might helps you to understand how geographic location and store characteristics influence sales performance.



```
1 grouped_data = sales_data.groupby(['State', 'City', 'Store_Type'])
```



► Grouping with pandas (Examples)

The DataFrame is grouped by the 'Product Category' using the `groupby()` method. This method collects all rows with the same category into a group. The `sum()` function is then applied to the 'Total Sales' column of these groups to get the total sales for each category.

```
1 df = pd.read_csv('Retail_Sales.csv')
2 grouped_sales = df.groupby('Product Category')['Total Sales'].sum()
3 print(grouped_sales)
```

```
Product Category
Clothing          350
Electronics       1050
Home & Kitchen     320
Name: Total Sales, dtype: int64
```



Aggregation with pandas

► Aggregation with pandas

Aggregation involves combining data from several items or groups to form a single summary result.

This method helps us understand big patterns by looking at smaller, summarized data pieces.



► Aggregation with pandas

There are several ways to summarize data depending on what you need to know. Here are some of the most common types.

- **Sum:** Adding up values to get a total.
- **Average (Mean):** Calculating the central value.
- **Maximum:** Finding the highest value in a set.
- **Minimum:** Finding the lowest value in a set.
- **Count:** Counting the number of items in a set.



► Aggregation with Pandas Example (dataset)

In this example, we will use a dataset containing sales information for a chain of stores. The dataset includes the following columns:

- **Store:** The store number (1, 2, or 3).
- **Product:** The product sold (Apple, Banana, Orange, Grapes, Melon).
- **Sales:** The sales amount.
- **Date:** The date of the sale.



	Store	Product	Sales	Date
0	3	Melon	93	2024-01-01
1	1	Grapes	133	2024-01-02
2	3	Grapes	193	2024-01-03
3	2	Melon	124	2024-01-04
4	3	Apple	58	2024-01-05



► Aggregation with Pandas Example (sum)

Sum: Adding up values to get a total.



```
1 df['Sales'].max()  
2 # Output: 199
```



► Aggregation with Pandas Example (Average)

Average (Mean): Calculating the central value.



```
1 df['Sales'].mean()  
2 # Output: 124.223
```



► Aggregation with Pandas Example (Maximum)

Maximum: Finding the highest value in a set.



```
1 df['Sales'].max()  
2 # Output: 199
```



► Aggregation with Pandas Example (Minimum)

Minimum: Finding the lowest value in a set.



```
1 df['Sales'].min()  
2 # Output: 50
```



► Aggregation with Pandas Example (Count)

Count: Counting the number of items in a set.



```
1 df['Sales'].count()  
2 # Output: 1000
```



Using Grouping with Aggregation in Pandas

▶ Using Grouping with Aggregation in Pandas Example

When you want to apply these aggregations for each group in a column (for example, per product or per store), you use **.groupby()** along with **.agg()**:

```
1 grouped_data = df.groupby('Product').agg(  
2     Total_Sales=('Sales', 'sum'),  
3     Average_Sales=('Sales', 'mean'),  
4     Maximum_Sales=('Sales', 'max'),  
5     Minimum_Sales=('Sales', 'min'),  
6     Count=('Sales', 'count')  
7 )  
8 grouped_data
```

	Total_Sales	Average_Sales	Maximum_Sales	Minimum_Sales	Count
Product					
Apple	28833	129.878378	199	50	222
Banana	23047	123.245989	199	50	187
Grapes	23322	120.216495	199	50	194
Melon	26603	124.313084	199	50	214
Orange	22418	122.502732	199	50	183



Using Grouping with Aggregation in Pandas Example (Multiple Columns)

In this example we grouped by two columns and then we applied multiple aggregations on each one:

```
1 grouped_data = df.groupby(['Product', 'Store']).agg(  
2     Total_Sales=('Sales', 'sum'),  
3     Average_Sales=('Sales', 'mean'),  
4     Maximum_Sales=('Sales', 'max'),  
5     Minimum_Sales=('Sales', 'min'),  
6     Count=('Sales', 'count')  
7 )  
8 grouped_data
```

		Total_Sales	Average_Sales	Maximum_Sales	Minimum_Sales	Count
Product	Store					
Apple	1	7595	130.948276	199	52	58
	2	10891	134.456790	199	52	81
	3	10347	124.662651	197	50	83
Banana	1	8605	122.928571	198	50	70
	2	5845	116.900000	189	50	50
	3	8597	128.313433	199	51	67
Grapes	1	7806	120.092308	194	54	65
	2	8447	118.971831	193	50	71
	3	7069	121.879310	199	50	58
Melon	1	8582	128.089552	199	50	67
	2	8936	133.373134	195	53	67
	3	9085	113.562500	199	50	80
Orange	1	7280	121.333333	199	50	60
	2	7850	124.603175	198	50	63
	3	7288	121.466667	194	50	60



Practice

DataSet Path :

2- Data Collection with Python

/LAB/Datasets/Traffic.csv

Notebook Path :

2- Data Collection with Python

/LAB/grouping_aggregation.ipynb



Data Science

Merging and Joining data



SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority



Agenda



Data Merging



Joins



Data Merging

► Data Merging and Joining

Merging and joining are techniques used to combine data from different sources into a single DataFrame. These methods can help us to create a new dataset that is easier to analyze.



Data Merging and Joining Benefits

Combining data allows us to view relationships and insights that aren't visible from separate datasets. It's crucial for completeness in analysis, ensuring no information is overlooked.

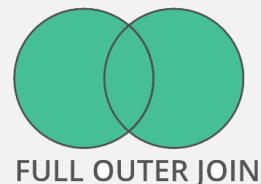
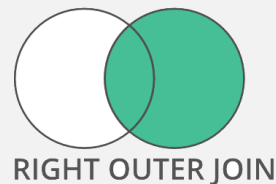
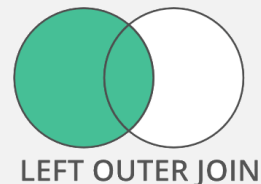
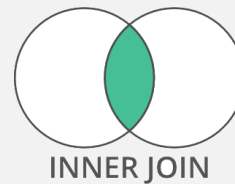
- Enables comprehensive analysis.
- Reduces the risk of overlooking critical insights.
- Enhances the richness of data by combining features from different sources.



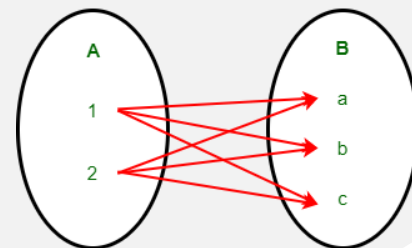
▶ Data Merging and Joining Concepts

Understanding the key terms like 'keys', 'join types', and 'indices' is crucial before performing data merges.

- **Keys:** Attributes used to identify the same entity across different datasets.
- **Join Types:** Various methods (inner, outer, left, right) determine how rows are combined based on the keys.
- **Indices:** Often used as keys, especially when they uniquely identify data rows.



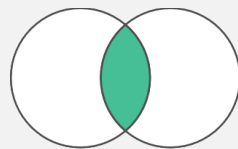
CROSS JOIN



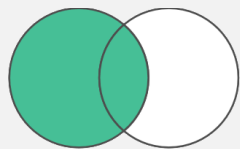
Joins

Types of Joins in Pandas (Inner Join)

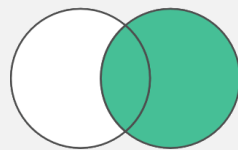
An inner join combines rows from both datasets wherever there are matching keys. It's the best option when you only want to keep rows that match in both datasets.



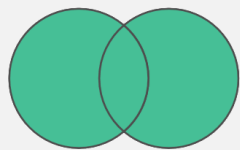
INNER JOIN



LEFT OUTER JOIN

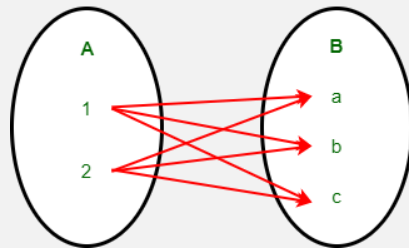


RIGHT OUTER JOIN



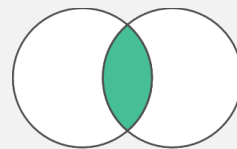
FULL OUTER JOIN

CROSS JOIN

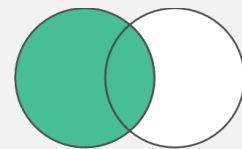


Types of Joins in Pandas (Left and Right Joins)

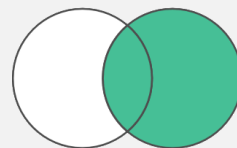
A left join keeps all rows from the left dataset and matches from the right. A right join does the opposite. Useful for ensuring no data is lost from the primary dataset.



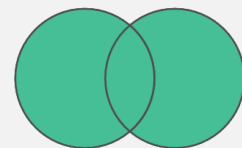
INNER JOIN



LEFT OUTER JOIN

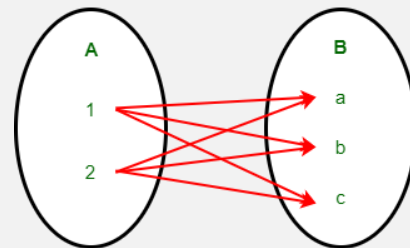


RIGHT OUTER JOIN



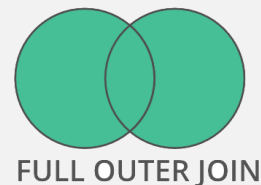
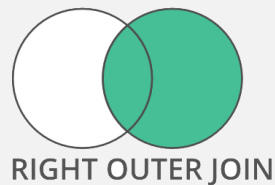
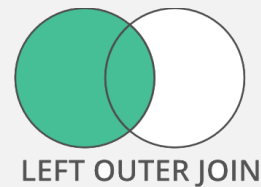
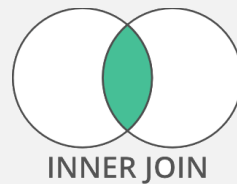
FULL OUTER JOIN

CROSS JOIN

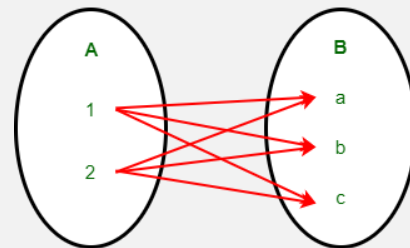


Types of Joins in Pandas (Outer Join)

An outer join returns all rows from both datasets, filling in missing values with NaNs where no match exists. Useful for retaining all available data.

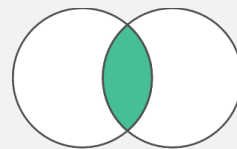


CROSS JOIN

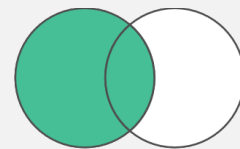


Types of Joins in Pandas (Cross Joins)

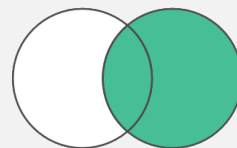
A cross join combines each row of one dataset with every row of another dataset. This type of join does not require a key to merge on and results in a Cartesian product of the datasets.



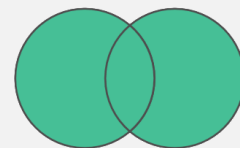
INNER JOIN



LEFT OUTER JOIN

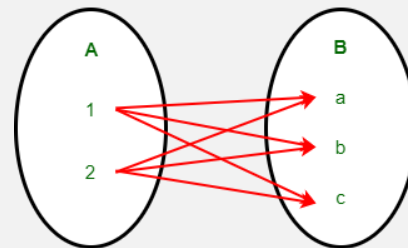


RIGHT OUTER JOIN



FULL OUTER JOIN

CROSS JOIN



▶ Joins Examples (Datasets)

A left join keeps all rows from the left dataset and matches from the right. A right join does the opposite. Useful for ensuring no data is lost from the primary dataset.

First DataFrame:

	key	value_df1
0	A	1
1	B	2
2	C	3

Second DataFrame:

	key	value_df2
0	B	4
1	C	5
2	D	6



Joins Examples (Inner Join)

Example on Inner Join:



```
1 inner_join_df = pd.merge(df1, df2, on='key', how='inner')  
2 print(inner_join_df)
```

	key	value_df1	value_df2
0	B	2	4
1	C	3	5



Joins Examples (Left Join)

Example on Left Join:



```
1 left_join_df = pd.merge(df1, df2, on='key', how='left')
2 print(left_join_df)
```

	key	value_df1	value_df2
0	A	1	NaN
1	B	2	4.0
2	C	3	5.0



Joins Examples (Right Join)

Example on Right Join:



```
1 right_join_df = pd.merge(df1, df2, on='key', how='right')  
2 print(right_join_df)
```

	key	value_df1	value_df2
0	B	2.0	4
1	C	3.0	5
2	D	NaN	6



Joins Examples (Outer Join)

Example on Outer Join:



```
1 outer_join_df = pd.merge(df1, df2, on='key', how='outer')  
2 print(outer_join_df)
```

	key	value_df1	value_df2
0	A	1.0	NaN
1	B	2.0	4.0
2	C	3.0	5.0
3	D	NaN	6.0



Joins Examples (Cross Join)

Example on Cross Join:



```
1 cross_join_df = pd.merge(df1, df2, how="cross")
2 print(cross_join_df)
```

	key_x	value_df1	key_y	value_df2
0	A	1	B	4
1	A	1	C	5
2	A	1	D	6
3	B	2	B	4
4	B	2	C	5
5	B	2	D	6
6	C	3	B	4
7	C	3	C	5
8	C	3	D	6



Practice

Notebook Path:

2- Data Collection with Python:

`/LAB/merge_joining.ipynb`



THANK YOU



SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority