IT Talent Training Course

Aug. 2025.

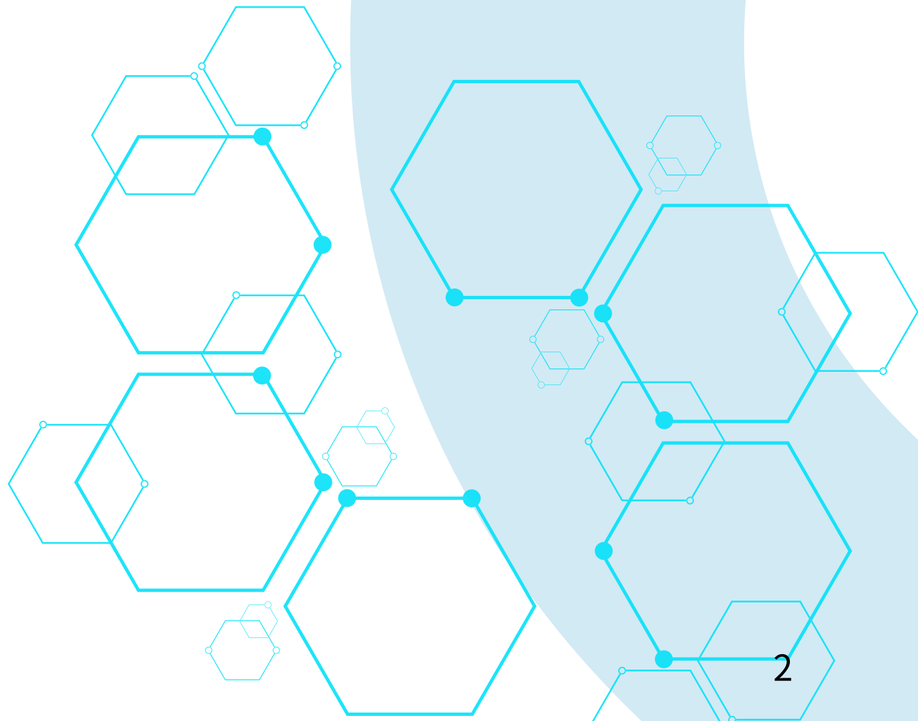
# A.I. PROGRAMMING WITH PYTORCH


**Instructor :**
Daesung Kim

**2nd Day - Part 01**

# INDEX

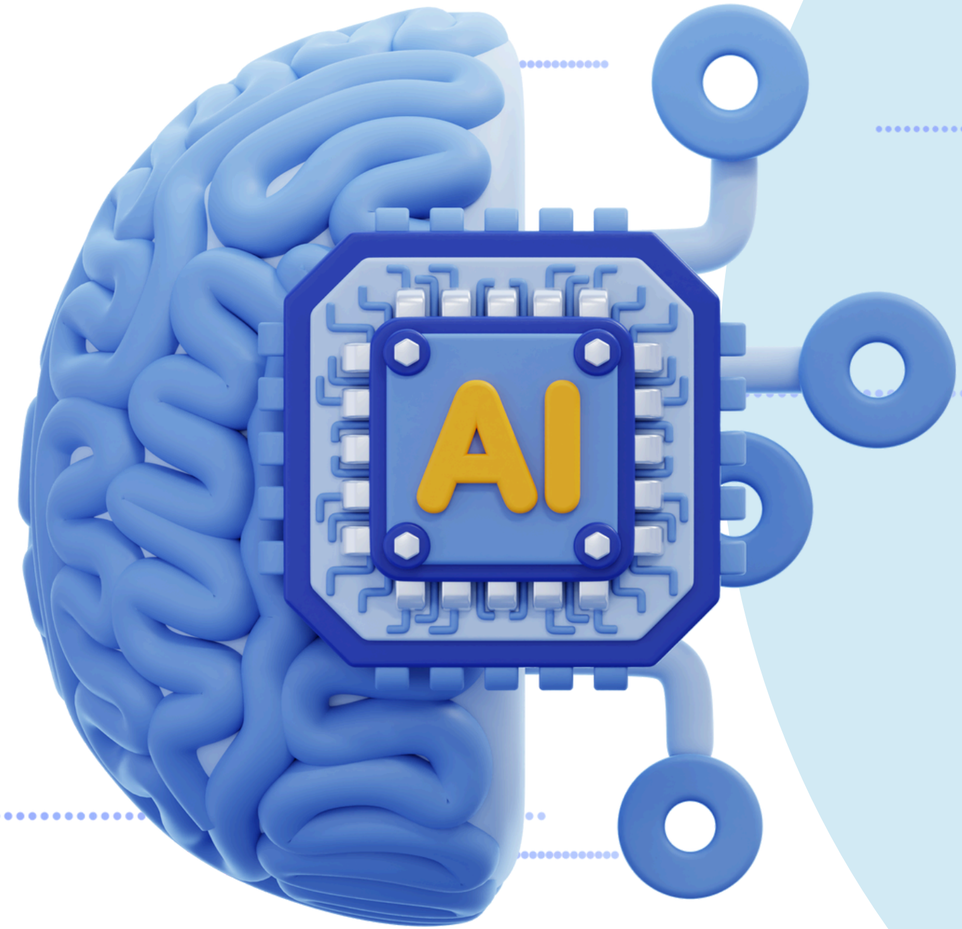




# 01
# GOOGLE COLAB

**Keywords**

Cloud Computing, Web Browser, Google Account



# WHY USE GOOGLE COLAB? - 1

This is a link to get started with Colab provided by Google. For detailed instructions on how to use and utilize it, please refer to the start page provided in the link.
https://colab.research.google.com/?hl=en

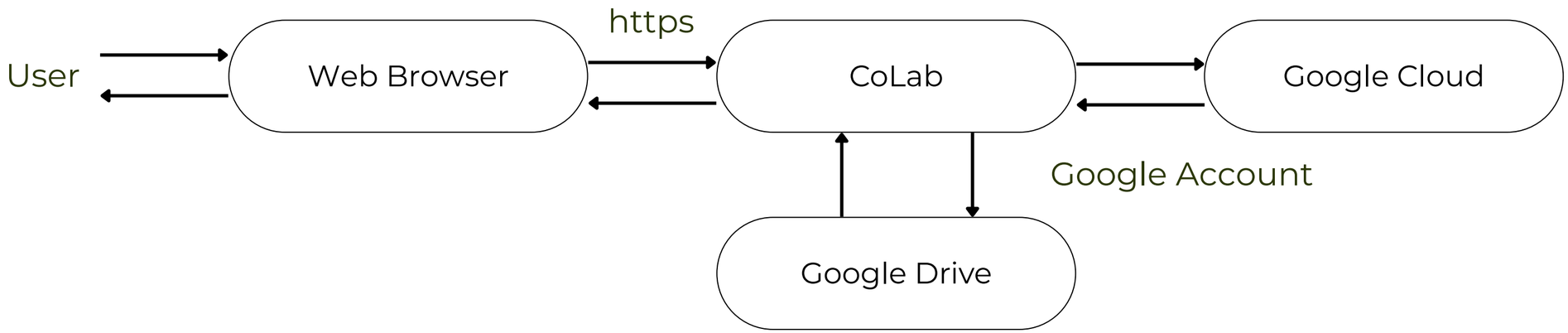
Colab is a cloud-based coding environment provided by Google. You can write and run Python code anywhere with just a web browser without a complicated installation process. In particular, using Colab is very important for our course. Here's why:

- **Zero-Setup:** Regardless of your personal computer specifications or operating system (Windows, macOS, etc.), all students can focus on practicing in the same environment. You can completely be free from the headache of environment setup issues such as library conflicts.

- **Free GPU:** Deep learning requires a huge amount of calculation. The learning process that can take several days with a general computer can be shortened to a few minutes or hours by utilizing the **GPU (Graphics Processing Unit)** provided by Colab for free. This is a key factor that enables in-depth practice within a limited time.

- **Easy sharing and collaboration:** Colab notebooks are automatically saved to Google Drive, and you can easily share them with others and work together with just one link. This is very useful when collaborating with team members during a project.

# WHY USE GOOGLE COLAB? - 2



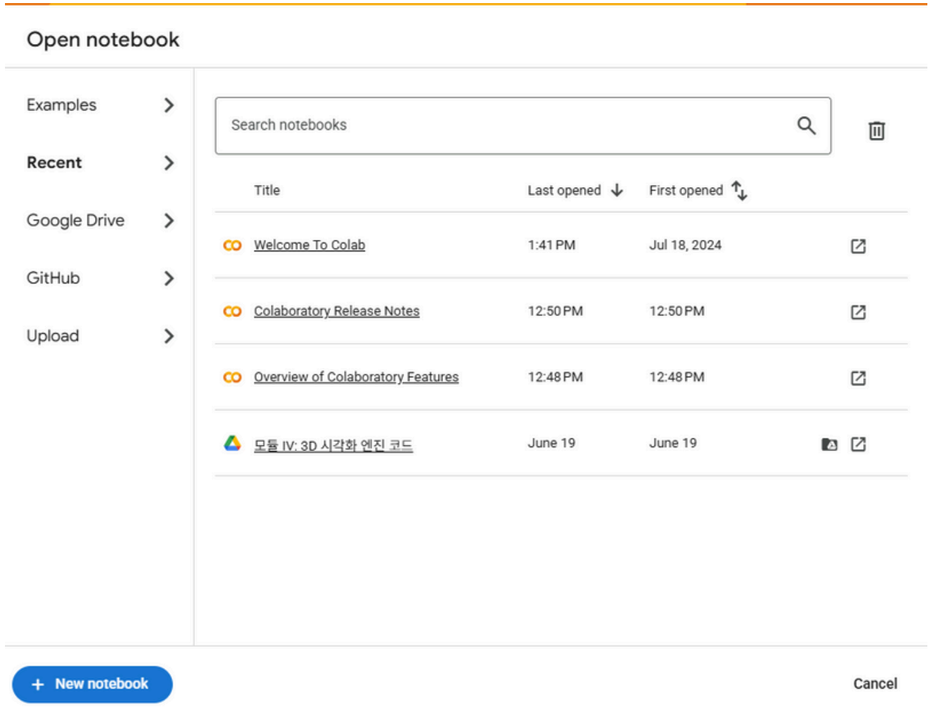

**Google Colab Disadvantages**

- There is a limitation on the session time.
- When the session time expires, the data you were working on is lost.
- Data containing personal information cannot be used

# COLAB INTERFACE - 1

## 1. Create a new notebook

1. Open a web browser and go to Google Colab.
2. Log in with your Google account.
3. In the pop-up window that appears on your screen, You can create a new notebook by clicking New Note or by selecting New Note from the File menu in the top left.



# COLAB INTERFACE - 2

## 2. Code Cell & Text Cell

Colab notebooks are made up of a combination of 'cells'.

- **Code Cell:** A space to write and run Python code. You can run the code by pressing the '▶' (Run) button on the left side of the cell or by pressing the keyboard shortcut Shift + Enter.

- **Text Cell:** A space to record descriptions, titles, notes, etc. about the code. You can easily add bullets, bold text, links, etc. using a simple grammar called 'Markdown'.

When you hover your mouse over a cell, the + Code and + Text buttons will appear, allowing you to add the type of cell you want.

# COLAB INTERFACE - 3

## [Practice] Running the first code and writing text

- The first cell in the newly created notebook is a code cell. Enter the following code and press Shift + Enter to run it. You will see the execution result appear right below the cell.

```
# The first time you run the code, it may take a while to connect to the Colab virtual machine.
print("Hello, LAOS AI Course!")

a = 10
b = 20
print(f"{a} + {b} = {a+b}")
```

- Hover over the code cell you just ran and click the + Text button to add a text cell. Type the following:

```
# Day 2 Practice

## This code is an example of adding two variables.
* Declare variables called `a` and `b`.
* Use `f-string` to neatly output the result.
```

- Pressing Shift + Enter in a text cell will convert it to clean text with Markdown syntax applied.

# LEARN ESSENTIAL SKILLS FOR PRACTICE - 1

## 1. Using data files: Google Drive integration

| | |
|---|---|
| **File System** | A system for organizing files and folders on a computer and accessing them through paths. |
| **Runtime** | It stands for 'execution environment'. In Colab, it refers to a virtual computer environment that runs the code we wrote, and we can assign CPU or GPU to this runtime. |

- In machine learning projects, you need to load data. In Colab, you can upload files directly from your computer or connect files from Google Drive.

- In this process, we use Google Drive integration as the default.

- This is because files uploaded directly will **disappear when the runtime is initialized**, but Google Drive files will be preserved permanently.

# LEARN ESSENTIAL SKILLS FOR PRACTICE - 1

## [Practice]

- Enter the following code into a new code cell and run it.

```
from google.colab import drive
drive.mount('/content/drive')
```

- When you run the code, you will be prompted to enter the authentication link and code.
- Click the link to sign in with your Google account, copy the authentication code that appears, paste it into the input field in Colab, and press Enter.
- If you see the message "Mounted at /content/drive", you have successfully connected.
- Now, if you click on the folder icon in the left sidebar, you will see a folder called drive, and within that, you will have access to all the files and folders in your Google Drive.

# LEARN ESSENTIAL SKILLS FOR PRACTICE - 1

## 2. Setting up hardware accelerator (GPU)

1. From the top menu, click **Runtime -> Change Runtime Type**.
2. In the pop-up window, **select GPU from the Hardware Accelerator** drop-down menu and click Save.
3. The runtime will restart and connect to the new GPU environment.
4. Run the following code to check if the GPU is properly allocated and what type of GPU it is. Let's check it using PyTorch, the core library of our course.

```
import torch

# The torch.cuda.is_available() function returns True/False whether PyTorch can use the GPU.
if torch.cuda.is_available():
    # If the GPU is available, print the name of GPU 0.
    print(f"Success! GPU is ready. Device name: {torch.cuda.get_device_name(0)}")
    # Store the 'cuda' device in a variable for future operations.
    device = torch.device("cuda")
else:
    print("Failed. GPU is not available. Use CPU.")
    device = torch.device("cpu")
```

# LEARN ESSENTIAL SKILLS FOR PRACTICE - 1

## 3. Install the required libraries yourself

Colab comes with most libraries installed, but sometimes you need a special library. In that case, you can install it yourself using the package manager called pip.

## [Practice]

You can execute system commands, not Python code, by prefixing a command with ! in a code cell. For example, let's install the streamlit library in advance, which we'll use later when creating a web UI.

```
# The !pip install command installs a new library.
!pip install streamlit

# Check if the installation was successful.
!streamlit --version
```

# SUMMARY

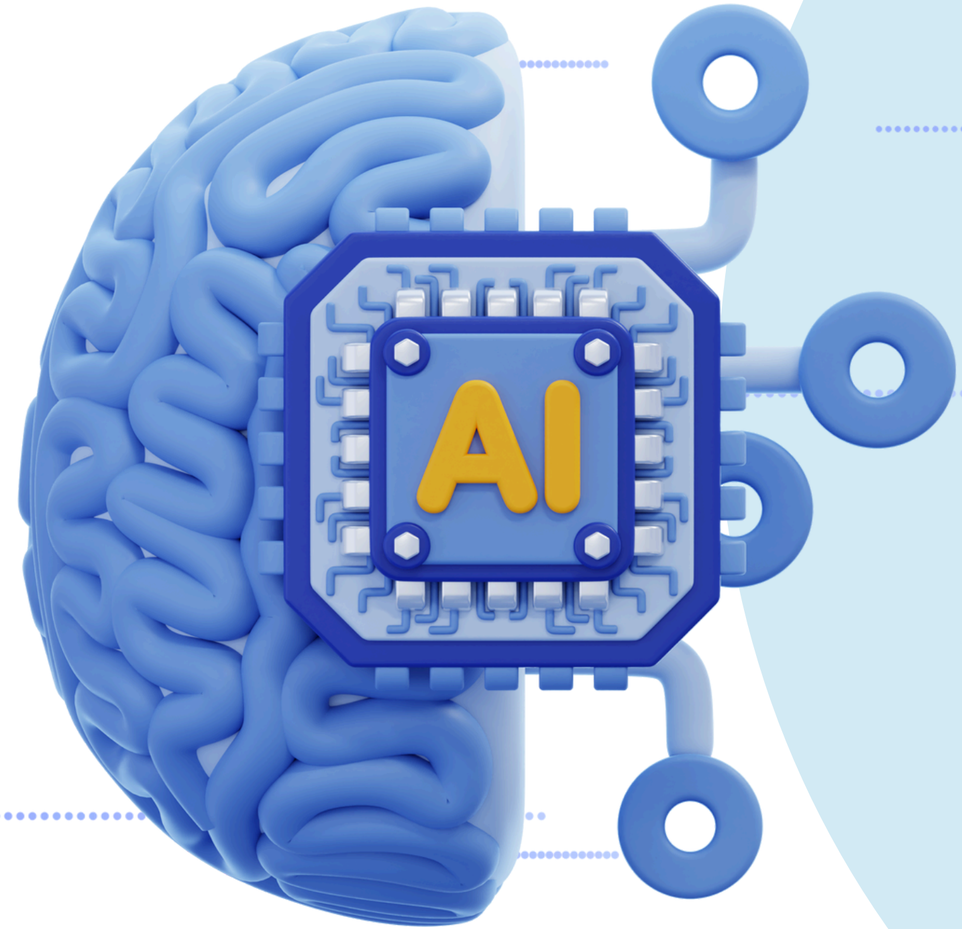
## Summary

- Colab is a cloud development environment without installation, providing free GPU and pre-installed libraries.
- Notebooks consist of code cells and text cells, and are executed cell by cell (Shift+Enter), and variables and libraries are shared across all cells.
- There are three ways to manage files:
  a. Local file upload: For temporary testing. The file disappears when the session ends.
  b. Google Drive mount: Most important. Used for permanent data storage and management.
  c. Download notebook: Save your work results to my computer.
- You can enable GPU in the Change Runtime Type menu to speed up deep learning.
- Libraries that are not installed by default can be installed directly with the !pip install [library name] command.

**Glossary of terms:**

- Colaboratory (Colab): A cloud-based Jupyter Notebook service provided by Google.
- Notebook: A Colab work file unit with the .ipynb extension.
- Cell: A basic block that makes up a notebook. It has code cells and text cells.
- Runtime: A virtual computer environment where code is executed. CPU or GPU is allocated depending on the runtime type.
- Mount: A process of connecting external storage such as Google Drive to Colab's file system so that it can be used as a local folder.
- GPU (Graphics Processing Unit): A hardware specialized for large-scale parallel operations that dramatically improves deep learning learning speed.

# 02
# PANDAS LIBRARY

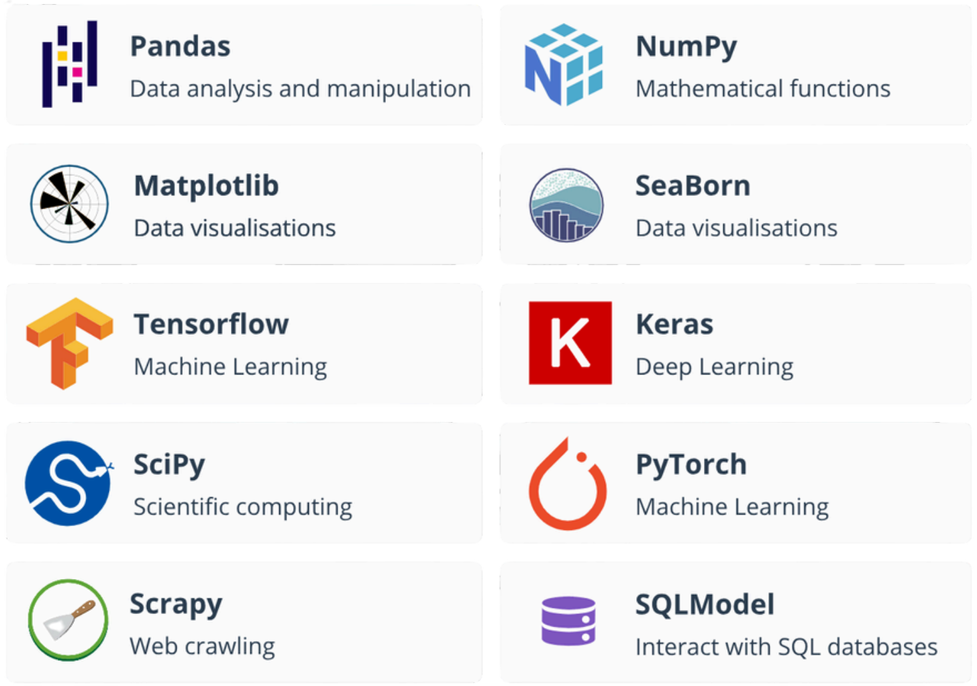



**Keywords**

Structured Data, DataFrame, Series, Missing Values, Type Conversion, Indexing, Slicing, Filtering, Grouping, Aggregation

# DATA SCIENCE LIBRARIES

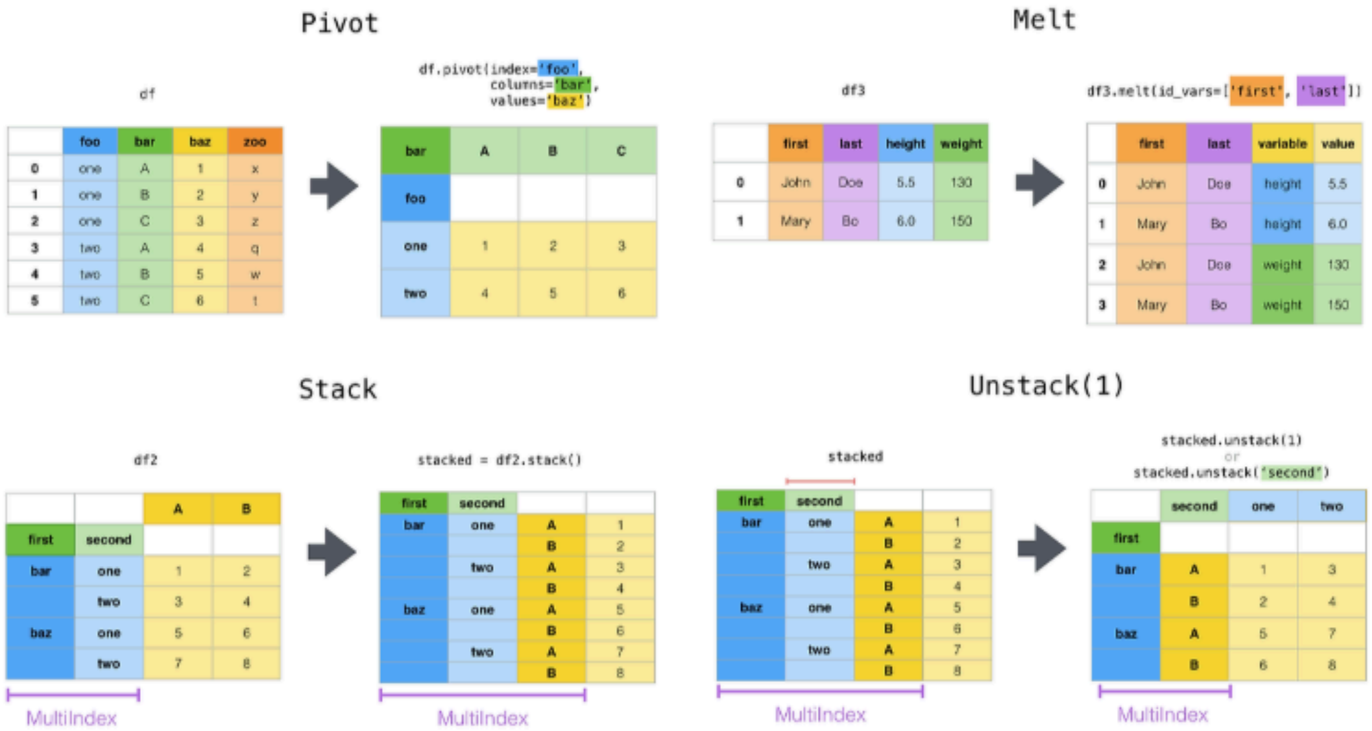
The performance of AI models is largely **dependent on the quality of the data.** No matter how much data there is and how sophisticated the model is, if the data is messy or difficult to analyze, it is difficult to expect good results. Pandas is an essential Python library for **loading, refining, and processing** such structured data into the desired form.





# ADVANTAGE OF PANDAS

- **Ease of use:** Similar to Excel in data manipulation, so even beginners to Python can use it easily.
- **Flexibility:** The data structure (Series, DataFrame) is very flexible, so it can handle various types of data.
- **Powerful functions:** It provides all the functions needed to load, save, filter, aggregate, and transform data.
- **Large-scale data processing:** It can efficiently process large amounts of data, and perform complex data analysis tasks with simple code.



# DATA STRUCTURES: SERIES AND DATAFRAME - 1

## 1. Series

Series has a structure similar to a one-dimensional array. Each data is given an index, and data can be accessed through this index.

**Characteristics:**

- **One-dimensional data:** Represents one column or one row of data.
- **Index:** Each value is given a unique label (index). By default, an integer index starting from 0 is given, but a user-defined index can also be used.
- **Single data type:** It is common for all elements in a Series to have the same data type.

Series

| | apples |
|---|---|
| 0 | 3 |
| 1 | 2 |
| 2 | 0 |
| 3 | 1 |

Series

| | oranges |
|---|---|
| 0 | 0 |
| 1 | 3 |
| 2 | 7 |
| 3 | 2 |

# DATA STRUCTURES: SERIES AND DATAFRAME - 2

```
import pandas as pd

# 1. Create Series with list (default integer index)
s1 = pd.Series([10, 20, 30, 40, 50])
print("--- Series 1 ---")
print(s1)
print(f"Data type: {s1.dtype}")
print("-" * 20)

# 2. Create Series with list and user-defined index
s2 = pd.Series([100, 200, 300], index=['a', 'b', 'c'])
print("--- Series 2 ---")
print(s2)
print("-" * 20)

# 3. Create Series with dictionary (dictionary keys become index)
data = {'Seoul': 970, 'Busan': 340, 'Incheon': 290}
s3 = pd.Series(data)
print("--- Series 3 ---")
print(s3)
print("-" * 20)
```

# DATA STRUCTURES: SERIES AND DATAFRAME - 3

## 2. DataFrame

DataFrame is the most core data structure in Pandas, **representing two-dimensional data** in the form of a table. You can think of it as a DataFrame made up of multiple Series. It is similar to a table in a relational database or an Excel spreadsheet.

**Characteristics:**

- **Two-dimensional data:** Consists of rows and columns.
- **Column Names:** Each column has a unique name, allowing you to intuitively distinguish data.
- **Row Index:** Each row is also given a unique index.
- **Various data types:** Each column can have different data types (e.g. one column is a number, another column is a string).

DataFrame

| | apples | oranges |
|---|---|---|
| 0 | 3 | 0 |
| 1 | 2 | 3 |
| 2 | 0 | 7 |
| 3 | 1 | 2 |

# DATA STRUCTURES: SERIES AND DATAFRAME - 4

```
# 1. Create a DataFrame using a dictionary
data = {
    'name': ['Alice', 'Bob', 'Charlie', 'David'],
    'age': [25, 30, 35, 40],
    'city': ['Seoul', 'Busan', 'Seoul', 'Jeju']     }
df1 = pd.DataFrame(data)
print("--- DataFrame 1 ---")
print(df1)
print("-" * 20)

# 2. Create a DataFrame from a list of lists (requires specifying column names)
data_list = [
    ['Apple', 100, 1.5],
    ['Banana', 150, 0.5],
    ['Cherry', 200, 2.0]     ]
df2 = pd.DataFrame(data_list, columns = ['fruit', 'quantity', 'price'])
print("--- DataFrame 2 ---")
print(df2)
print("-" * 20)

# 3. Create a DataFrame from a NumPy array
import numpy as np
np_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
df3 = pd.DataFrame(np_array, columns = ['Col1', 'Col2', 'Col3'])
print("--- DataFrame 3 ---")
print(df3)
print("-" * 20)
```

# DATA INPUT/OUTPUT - 1

Pandas provides powerful functions for loading and storing data in a variety of formats. We will learn how to handle the most commonly used CSV (Comma Separated Values) files and Excel files.

## 1. CSV file read( pd.read_csv() )

- A CSV file is a text file in which each value is separated by a comma (,). It is one of the most widely used data formats.

```
# Create a virtual CSV file in Colab environment (for practice)
csv_content = """Name,Age,Score
Alice,25,85
Bob,30,92
Charlie,22,78
David,28,65
Eve,35,95
"""

with open('sample.csv', 'w') as f:
    f.write(csv_content)

# Read CSV file
df_csv = pd.read_csv('sample.csv')
print("--- Read sample.csv file ---")
print(df_csv)
print("-" * 20)
```

**Main options:**

- **sep:** Delimiter (default is comma). For tab-separated files, specify **sep='\t'**.
- **header:** Position of header (column name) (default is 0, first row). For files without header, specify header=None.
- **index_col:** Specify the column to use as the index.
- **encoding:** File encoding (e.g. 'utf-8', 'euc-kr').

# DATA INPUT/OUTPUT - 2

## 2. Excel file read( pd.read_excel() )

```
# Create a virtual Excel file in Colab environment (openpyxl library required for practice)
!pip install openpyxl
from pandas import ExcelWriter

excel_content = {
    'Sheet1': {
        'Product': ['Laptop', 'Mouse', 'Keyboard'],
        'Price': [1200, 25, 75]
    },
    'Sheet2': {
        'City': ['Hanoi', 'Vientiane', 'Phnom Penh'],
        'Population': [800, 70, 200]
    }
}

with ExcelWriter('sample.xlsx') as writer:
    pd.DataFrame(excel_content['Sheet1']).to_excel(writer, sheet_name='Sheet1', index=False)
    pd.DataFrame(excel_content['Sheet2']).to_excel(writer, sheet_name='Sheet2', index=False)
```

# DATA INPUT/OUTPUT - 3

```
# Read Excel file (default is the first sheet)
df_excel_sheet1 = pd.read_excel('sample.xlsx')
print("--- sample.xlsx (Sheet1) file read result ---")
print(df_excel_sheet1)
print("-" * 20)

# Read specific sheet
df_excel_sheet2 = pd.read_excel('sample.xlsx', sheet_name='Sheet2')
print("--- sample.xlsx (Sheet2) file read result ---")
print(df_excel_sheet2)
print("-" * 20)
```

# DATA INPUT/OUTPUT - 4

## 2. Save as dataframe file( to_csv(), to_excel() )

```
# Create DataFrame
df_save = pd.DataFrame({
    'Item': ['Pen', 'Notebook', 'Eraser'],
    'Quantity': [10, 5, 20],
    'UnitPrice': [1.2, 5.0, 0.5]
})

# Save to CSV file (excluding index)
df_save.to_csv('output.csv', index=False)
print("output.csv file was created successfully.")

# Save to Excel file (excluding index)
df_save.to_excel('output.xlsx', index=False)
print("output.xlsx file was created successfully.")
```

# DATA CLEANING - 1

Real-world data is often incomplete or inconsistent. Data cleansing is the process of resolving these issues and putting the data into a form suitable for analysis and modeling.

## 1. Handling missing values

Missing values are values that are missing in the data (e.g. NaN, None). Pandas provides a variety of ways to check for and handle missing values.

**[ Code ] Day02 - 01_Handling_missing_value.py**

# DATA CLEANING - 2

## 2. Data Type Conversion

Often, data is not stored in the correct type. For example, a value that should be recognized as a number may be stored as a string, or a date may be stored as plain text. Pandas can convert data types using functions such as astype() or pd.to_datetime().

**[ Code ] Day02 - 02_Data type_conversion.py**

# DATA SELECTION AND MANIPULATION - 1

Once data cleansing is complete, you need to learn how to efficiently select the data you need and process it into a new form.

## 1. Data selection : loc, iloc

The most powerful way to access the data you want in a DataFrame is using the loc and iloc indexers.

- **loc:** Performs label-based indexing. Selects data using row indices and column names.
- **iloc:** Performs integer-location based indexing. Selects data using integer locations starting from 0.

## [ Code ] Day02 - 03_Data_Selection.py

# DATA SELECTION AND MANIPULATION - 2

## 2. Data filtering

- Extracting only data that satisfies certain conditions is called filtering. In Pandas, filtering can be done very intuitively using Boolean masking.

**[ Code ] Day02 - 04_Data_filtering.py**

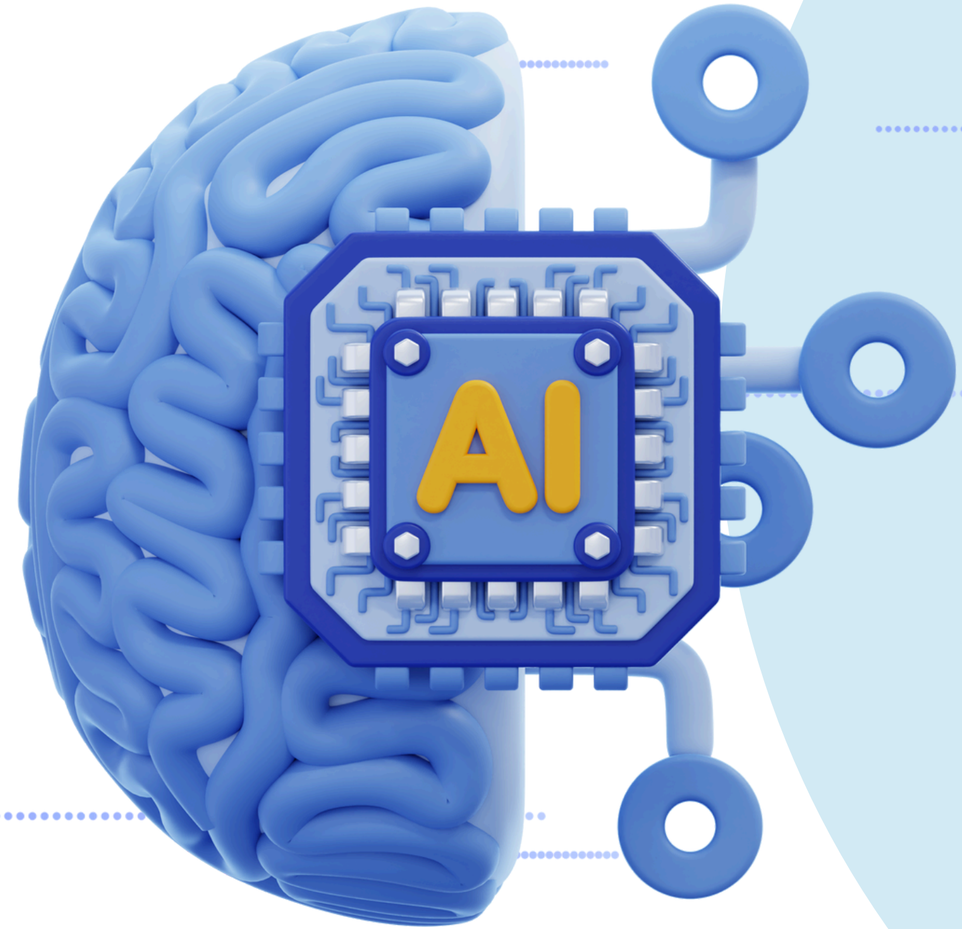
# DATA SELECTION AND MANIPULATION - 3

### 3. Grouping and Aggregation

- The groupby() method allows you to group data based on the values of a specific column and apply various aggregate functions such as sum, average, and count to each group. This is very useful for finding statistics by specific properties of the data.

**[ Code ] Day02 - 05_Grouping_Aggregation.py**

# 03
# INTEGRATED PRACTICE EXAMPLE



# MACHINE LEARNING WORKFLOW

Using real data provided by data platforms such as Kaggle, you can apply the functions of Pandas learned above more realistically. Here, we will simulate the data cleaning and processing process using a simple virtual 'used car price dataset' as an example.

**Goals:**

1. Create and load a virtual used car dataset.
2. Check for missing values and process them appropriately.
3. Convert the 'Year' column to an integer.
4. Calculate the average price by 'Brand'.
5. Filter only vehicles with mileage less than 100,000 km.

**[ Code ] Day02 - 06_Integraded_Practice_Example.py**

# SUMMARY

## Summary

- Pandas is an essential library for data cleaning and processing.
- The main data structures are one-dimensional Series and two-dimensional DataFrame.
- You can load data with pd.read_csv(), pd.read_excel() and save it with to_csv(), to_excel().
- Use isnull(), dropna(), and fillna() to efficiently handle missing values.
- Convert data types with astype(), pd.to_datetime(), etc.
- Select data using loc (label-based) and iloc (integer location-based).
- Filter data based on specific conditions using Boolean masking.
- Group data and calculate statistics using groupby() and aggregate functions (sum(), mean(), agg()).

**Glossary of terms:**

- Pandas: A library for data analysis and manipulation in Python.
- Series: A one-dimensional data structure in Pandas.
- DataFrame: A two-dimensional tabular data structure in Pandas.
- Missing Value: A value that is not in the data.
- loc: An indexer that selects data based on labels in a Pandas DataFrame.
- iloc: An indexer that selects data based on integer positions in a Pandas DataFrame.
- Filtering: A process of extracting data that satisfies a specific condition.
- Grouping: A process of grouping data based on the values of a specific column.
- Aggregation: A process of applying statistical functions to grouped data to summarize it.
- Vectorized Operation: A method of performing operations on the entire array without loops. (A concept learned in NumPy, but also important in Pandas)

IT Talent Training Course

Aug. 2025.

# A.I. PROGRAMMING WITH PYTORCH


**Instructor :**
Daesung Kim

**2nd Day - Part 02**

# INDEX



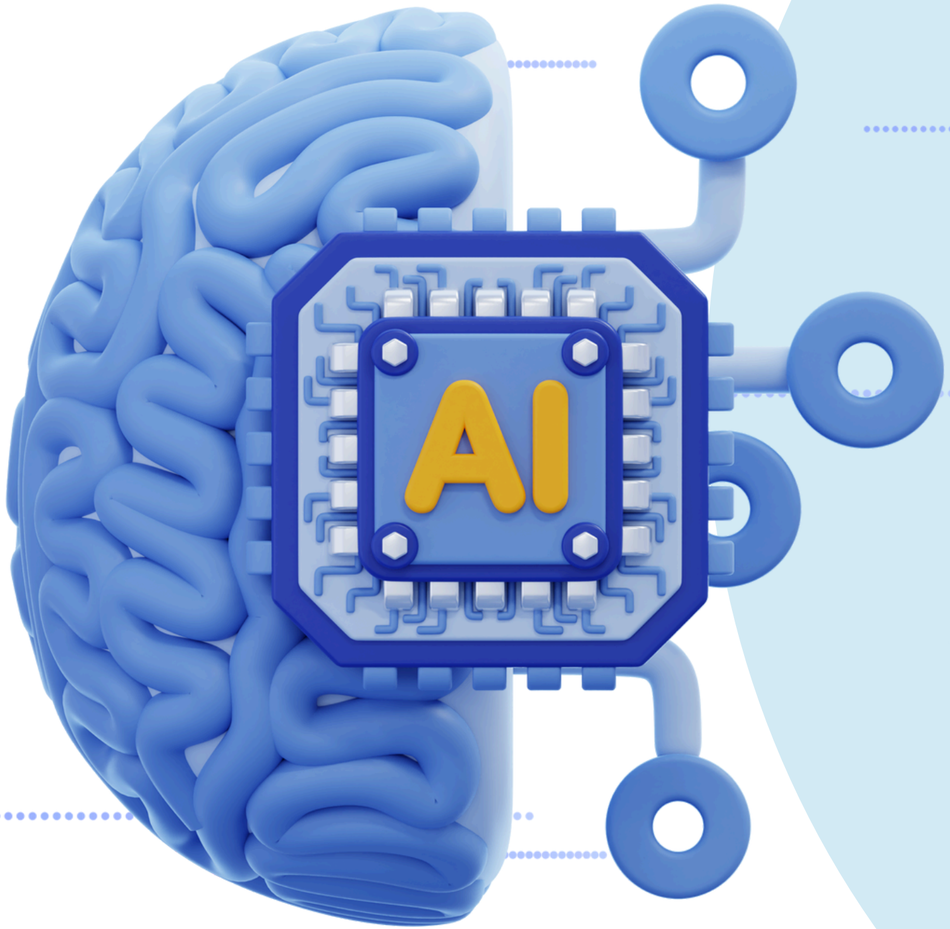
# WHY IS DATA VISUALIZATION IMPORTANT?

In the morning, we learned how to import data using Pandas and cleanly refine it into the desired form. Now we have a well-organized data table in front of us. However, it is very difficult to see hidden patterns or anomalies in the data at a glance by looking at a table full of numbers.

**Data visualization** is a necessary skill at this point. Visualization is not simply the task of making data into a pretty graph. It is the most intuitive and powerful way to communicate with data.

- **The core of Exploratory Data Analysis (EDA):** It is the process of visually exploring what variables are in the data, what distribution each variable has, and what relationships are hidden between the variables, thereby increasing understanding of the data itself.
- **Discovering Insights:** You can find meaningful patterns, trends, and unexpected outliers in complex data to gain key ideas for business decisions or modeling strategies.
- **Persuasive Communication:** When conveying analysis results to others, a well-made graph delivers a much clearer and more persuasive message than a long explanation.
- **Diagnose model performance:** After training a deep learning model, you can visually analyze the parts where the model is good at predicting and the parts where it is particularly difficult to find ways to improve the model.

# 01
# Matplotlib

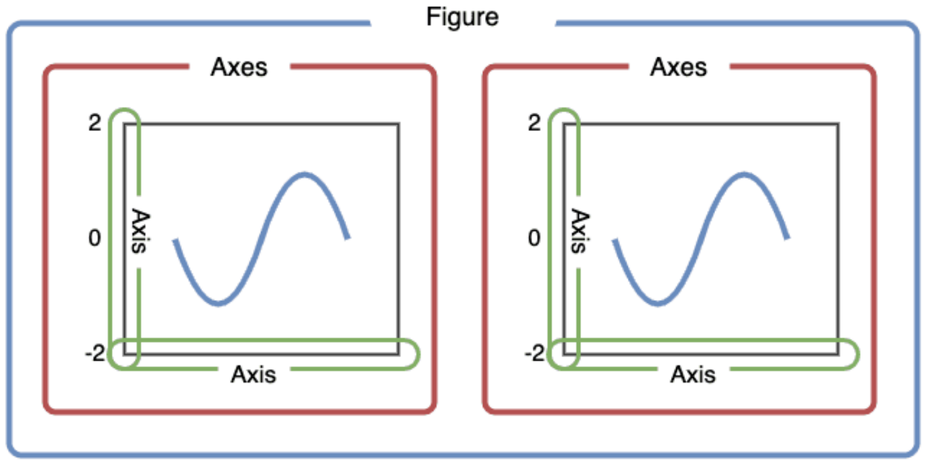


# MATPLOTLIB OVERVIEW

Matplotlib is the most basic library for visualizing data in Python. Almost all Python visualization tools are based on or heavily influenced by Matplotlib. Therefore, if you understand how Matplotlib works, you can easily handle other libraries.

Matplotlib works by placing a "picture (Axes)" on a "drawing paper (Figure)" and drawing various graphs (Plots) within that picture.

- **Figure:** The entire drawing paper or window to draw a picture. You can set the size or the space to hold multiple pictures.
- **Axes:** Individual pictures (subplots) within a Figure. You can place multiple Axes within a Figure. It includes the components of the graph, such as the x-axis, y-axis, and title.
- **Plot:** The actual graph, such as lines, bars, and points, drawn on the Axes.



# BASIC GRAPH - 1

First, import the libraries needed for the practice. For the most part, matplotlib.pyplot is abbreviated as plt, pandas as pd, and seaborn as sns.

```
# Import required libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Font settings in Matplotlib
# Use specific English font (e.g. Arial)
# Assuming that Arial font is installed in Colab. If not, install it.
plt.rc('font', family='Arial')
plt.rcParams['axes.unicode_minus'] = False # Prevent minus sign from being broken

print("English font setting complete. After running this code, press Runtime -> Restart Runtime.")
```

# BASIC GRAPH - 2

## 1. Line plot : plt.plot()

Suitable for showing changes in data over time (time series data) or relationships between continuous values.

```
# Example data
years = [2018, 2019, 2020, 2021, 2022]
sales = [100, 120, 110, 150, 180]

# 1. Prepare a drawing paper (Figure).
plt.figure(figsize=(8, 5)) # Set the drawing paper size with figsize. (Width, height inches)

# 2. Draw a graph (Plot) on the figure (Axes).
plt.plot(years, sales, marker='o', linestyle='--')

# 3. Decorate the graph. plt.title('Sales by year') # Title
plt.xlabel('Year') # X-axis name
plt.ylabel('Sales (100 million won)') # Y-axis name
plt.grid(True) # Add grid pattern

# 4. Show the graph on the screen.
plt.show()
```

# BASIC GRAPH - 3

## 2. Bar plot : plt.bar()

Mainly used when comparing values (size, count, etc.) of categorical data.

```
# Sample Data
products = ['A', 'B', 'C', 'D']
units_sold = [350, 500, 200, 420]

plt.figure(figsize=(8, 5))
plt.bar(products, units_sold, color='skyblue')

plt.title('Sales by Product')
plt.xlabel('Products')
plt.ylabel('Sold Count')

plt.show()
```

# BASIC GRAPH - 4

## 3. Scatter plot : plt.scatter()

Used to determine the relationship (correlation) between two numeric variables.

```
# Example data (randomly generated)
study_hours = np.random.rand(50) * 10
test_scores = study_hours * 8 + np.random.randn(50) * 10

plt.figure(figsize=(8, 5))
plt.scatter(study_hours, test_scores, alpha=0.6) # alpha is the transparency of the points

plt.title('Relationship between study hours and test scores')
plt.xlabel('Study hours')
plt.ylabel('Test scores')

plt.show()
```

# BASIC GRAPH - 5

## 4. Histogram : plt.hist()

Used to check what distribution a numeric variable has (what range of values is the data mostly concentrated in)

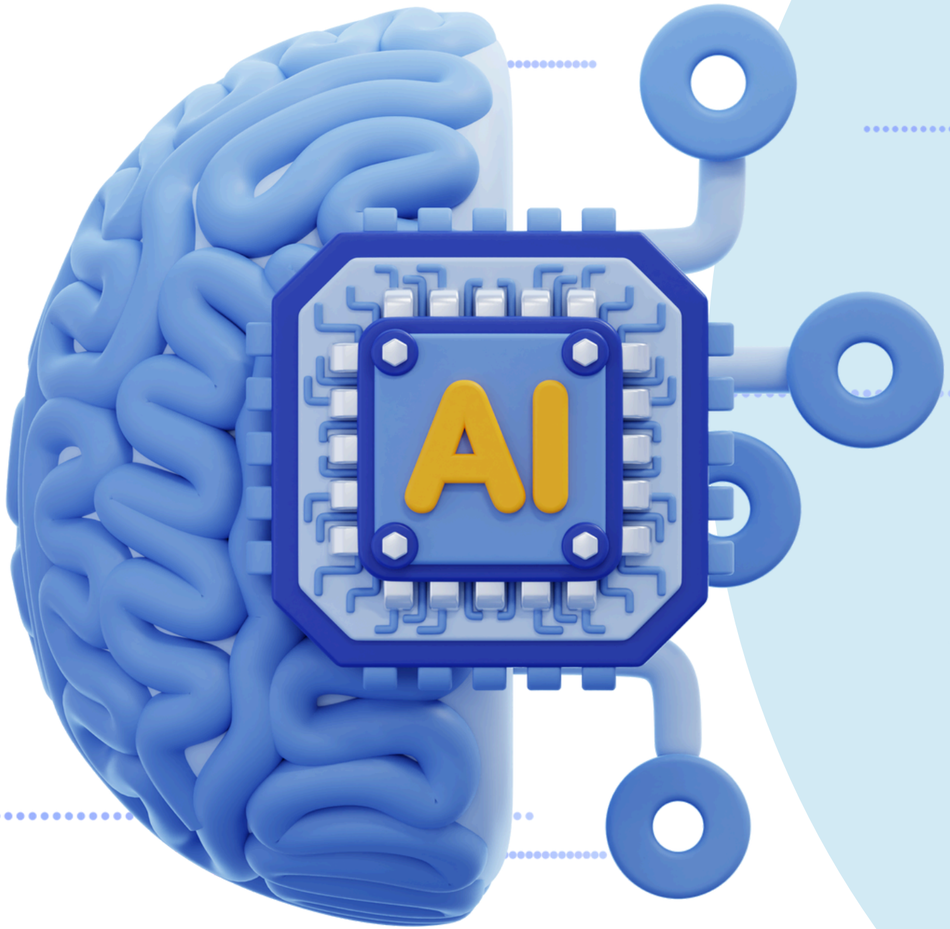
```
# Example data (random data following normal distribution)
data = np.random.randn(1000)

plt.figure(figsize=(8, 5))
plt.hist(data, bins=30, color='coral') # bins means the number of bars.

plt.title('Check the data distribution')
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```

# DRAW MULTIPLE GRAPHS AT ONCE

When you want to compare multiple graphs, you can use subplots to place multiple figures (axes) on a single drawing sheet.

```
# Create a subplot with 1 row and 2 columns. fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Draw a line graph on the first figure (axes[0])
axes[0].plot(years, sales, marker='o', color='b')
axes[0].set_title('Sales by year')
axes[0].set_xlabel('Year')
axes[0].set_ylabel('Sales (100 million won)')

# Draw a bar graph on the second figure (axes[1])
axes[1].bar(products, units_sold, color='g')
axes[1].set_title('Sales by product')
axes[1].set_xlabel('Product')
axes[1].set_ylabel('Sold count')

# Set the overall title
fig.suptitle('2024 'Summary of first half performance', fontsize=16)

# Automatic layout adjustment
plt.tight_layout(rect=[0, 0.03, 1, 0.95]) # Adjust so that it does not overlap with suptitle

plt.show()
```

# 02
# Seaborn

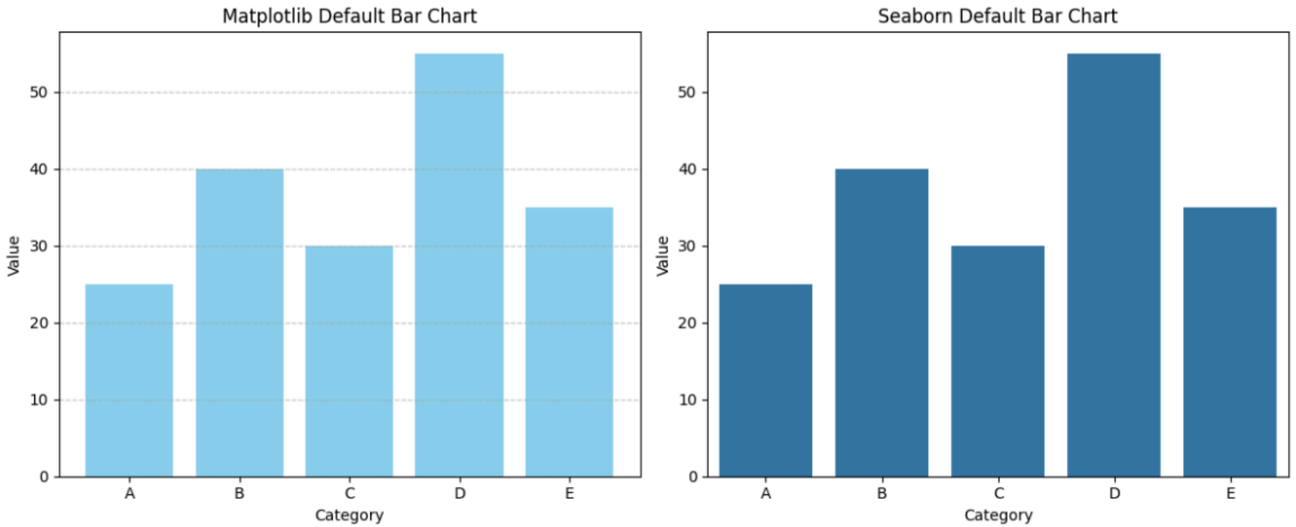


# SEABORN OVERVIEW

Matplotlib is very flexible and powerful, but it is a hassle to set up many settings yourself to create a good-looking graph. Seaborn is a high-level library that is built on Matplotlib, but allows you to draw more beautiful and statistically significant graphs with less code.

**Advantages of Seaborn:**

- **Beautiful default style:** The default themes and color palettes are nice to look at.
- **Built-in statistical information:** There are many functions that automatically calculate and display statistical information such as the mean and confidence interval when drawing a graph.
- **Perfect compatibility with Pandas DataFrame:** You can specify data using the column names of Pandas DataFrame directly, which is very convenient.



# BASIC STATISTICS GRAPH - 1

Let's use Kaggle's "Tips" dataset as an example. This dataset contains information about tips paid by customers who dine at restaurants.

```
# Load the 'tips' dataset built into Seaborn
tips = sns.load_dataset("tips")

# Check the first 5 rows of the dataset.
print(tips.head())
```

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

# BASIC STATISTICS GRAPH - 2

## 1. Relationship visualization

When you want to see the relationship between two numeric variables, use scatterplot. Pass a DataFrame to the data argument, and the column names as strings to the x and y arguments.

```
plt.figure(figsize=(8, 6))
sns.scatterplot(data=tips, x='total_bill', y='tip')
plt.title('Relationship between total meal amount and tip')
plt.show()
```

By adding the hue argument to the scatterplot, you can express three-dimensional information by coloring the points differently depending on a specific categorical variable.

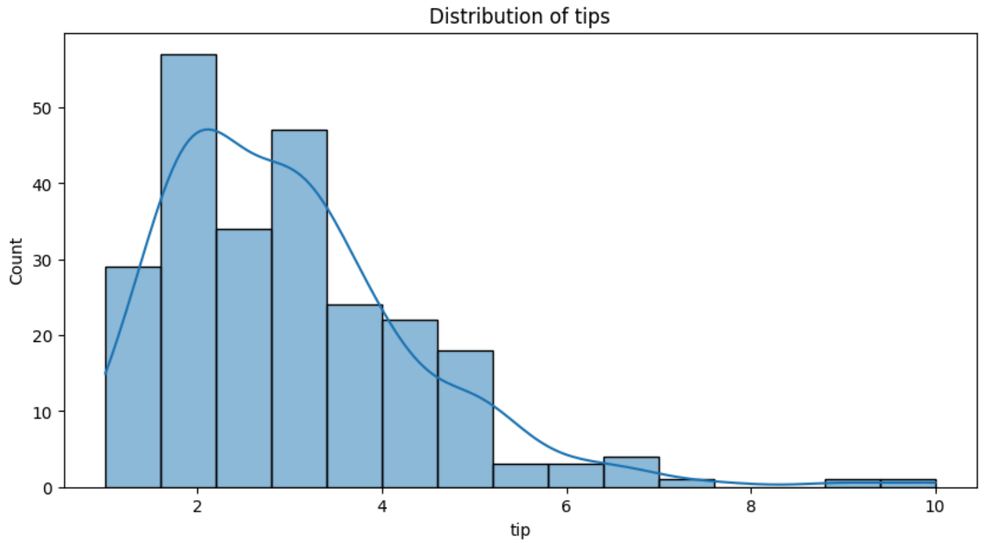
```
plt.figure(figsize=(8, 6))
# Add hue='smoker' to display dots in different colors depending on whether or not they smoke
# Add style='time' to display dots in different shapes depending on the time
sns.scatterplot(data=tips, x='total_bill', y='tip', hue='smoker', style='time', s=100) # s is the size of the dots
plt.title('Relationship between total meal amount and tip (by smoking and time)')
plt.show()
```

# BASIC STATISTICS GRAPH - 3

## 2. Distribution visualization

When looking at the distribution of a single numeric variable, use histplot. If you give the kde=True option, you can draw a Kernel Density Estimate curve to get a smoother view of the distribution.

```
plt.figure(figsize=(10, 5))
sns.histplot(data=tips, x='tip', kde=True, bins=15)
plt.title('Distribution of tips')
plt.show()
```



# BASIC STATISTICS GRAPH - 4

When you want to compare the distribution of numeric variables according to categorical variables, boxplot or violinplot are very useful.

- **Box Plot:** Shows the quartiles (25%, 50% (median), 75%) of the data in the form of boxes. It is good for understanding the overall distribution of the data and outliers.
- **Violin Plot:** This is a combination of Box Plot and KDE Plot. It provides richer information by showing the density of how the data is distributed at each value.

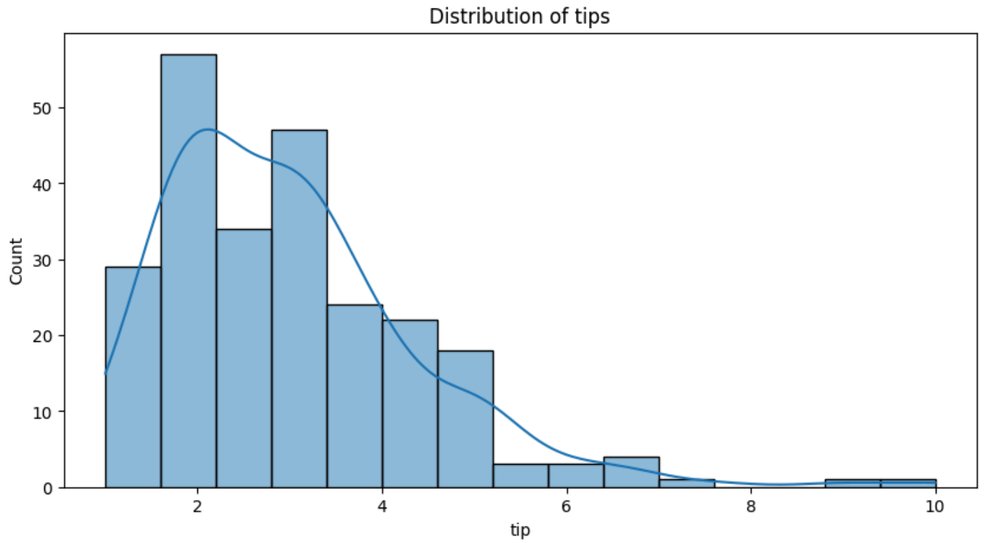
```
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Box Plot: Distribution of tips by day of the week
sns.boxplot(data=tips, x='day', y='tip', ax=axes[0])
axes[0].set_title('Distribution of tips by day of the week (Box Plot)')

# Violin Plot: Distribution of tips by day of the week
sns.violinplot(data=tips, x='day', y='tip', ax=axes[1])
axes[1].set_title('Distribution of tips by day of the week (Violin Plot)')

plt.show()
```

# BASIC STATISTICS GRAPH - 5

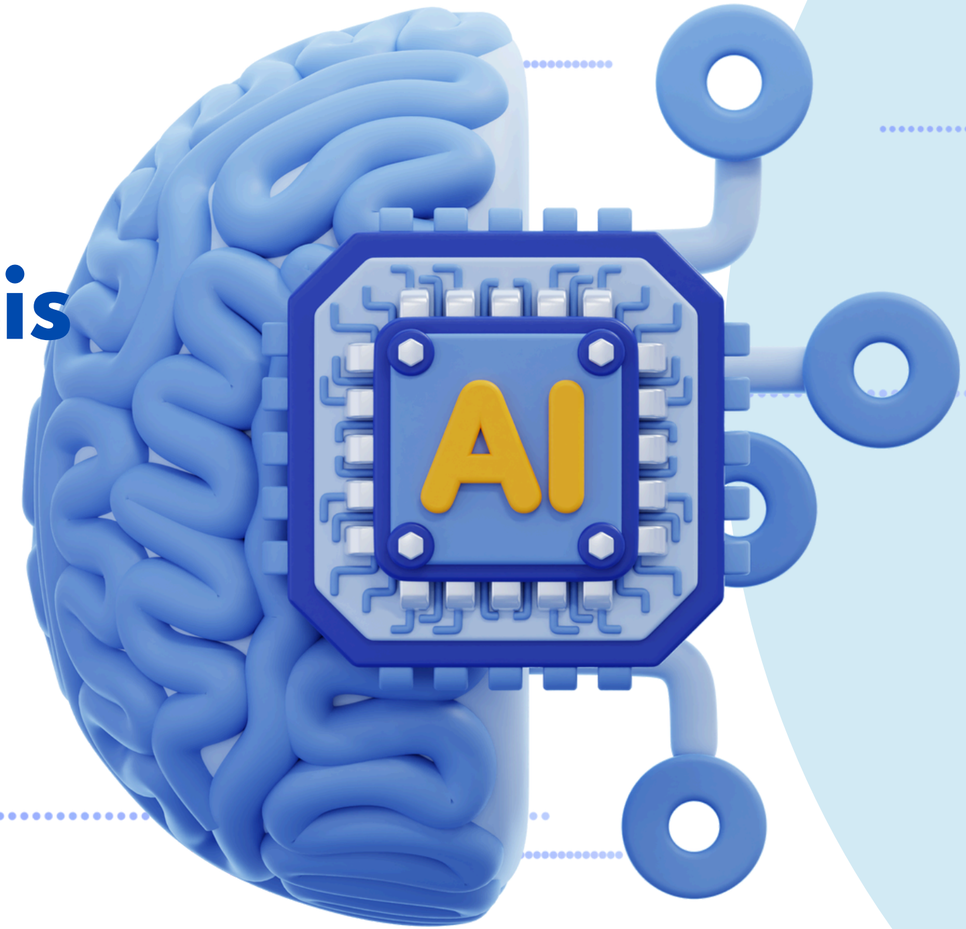
## 3. Categorical data visualization

Use countplot to count how many of each category there are in a categorical variable.

```
plt.figure(figsize=(8, 5))
sns.countplot(data=tips, x='day', order=['Thur', 'Fri', 'Sat', 'Sun']) # Order by order
plt.title('Visits by day of the week')
plt.show()
```



# 03
# Exploratory Data Analysis (EDA)



# EDA OVERVIEW

Now that you've learned the features of Matplotlib and Seaborn, let's combine them to perform full-scale exploratory data analysis (EDA). There is no set answer to the analysis process, but it usually proceeds as follows:

- Loading data and checking basic information (reviewing what we learned in the morning)
- Univariate Analysis: Check the distribution of one individual variable (column).
- Bivariate Analysis: Check the relationship between two variables.
- Multivariate Analysis: Check the relationship between three or more variables.

Let's follow this process with the **'Kaggle Used Car Price Dataset'** we covered this morning.

```
# Create a virtual used car dataframe for practice.
# In practice, load the file like pd.read_csv('used_car_data.csv'). data = {
'Year': np.random.randint(2010, 2023, 200);
'Kilometers_Driven': np.random.randint(10000, 150000, 200);
'Fuel_Type': np.random.choice(['Petrol', 'Diesel', 'CNG'], 200, p=[0.6, 0.35, 0.05]),
'Transmission': np.random.choice(['Manual', 'Automatic'], 200, p=[0.7, 0.3]),
'Price': np.random.uniform(2, 25, 200)
}
car_data = pd.DataFrame(data)
car_data['Price'] = (2023 - car_data['Year']) * -2 + car_data['Kilometers_Driven'] / 5000 + np.random.randn(200) * 2
car_data['Price'] = np.maximum(1, car_data['Price']) # Make sure the price is not negative

print(car_data.head())
print(car_data.info())
print(car_data.describe())
```

# UNIVARIATE ANALYSIS

The first step in analysis is to look at what each variable looks like.

- **Numeric variables:** Use histplot or kdeplot to check the distribution.
- **Categorical variables:** Use countplot to check the frequency of each category.

```
fig, axes = plt.subplots(2, 2, figsize=(15, 12))
fig.suptitle('Used Car Data Univariate Analysis', fontsize=16)

# Price Distribution
sns.histplot(data=car_data, x='Price', kde=True, ax=axes[0, 0])
axes[0, 0].set_title('Price Distribution')

# Kilometers_Driven Distribution
sns.histplot(data=car_data, x='Kilometers_Driven', kde=True, ax=axes[0, 1])
axes[0, 1].set_title('Driving Distance Distribution')

# Fuel Type Frequency
sns.countplot(data=car_data, x='Fuel_Type', ax=axes[1, 0])
axes[1, 0].set_title('Number of Vehicles by Fuel Type')

# Transmission Type Frequency
sns.countplot(data=car_data, x='Transmission', ax=axes[1, 1])
axes[1, 1].set_title('Number of Vehicles by Transmission Type')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

**Analysis Results:**

- Price shows a right-skewed distribution with a lot of data concentrated in the lower price ranges.
- Kilometers_Driven shows a relatively even distribution.
- We can see that the fuel type is overwhelmingly gasoline, and the transmission is more often manual.

# BIVARIATE ANALYSIS

Now let's pair the variables and look at the relationships.

- **Number vs. number:** Look at correlations with scatterplots or regplots (including regression lines).
- **Category vs. number:** Compare distributions across groups with boxplots, violinplots, and stripplots.
- **Category vs. category:** Look at counts by group with crosstabs and heatmaps.

```
fig, axes = plt.subplots(1, 3, figsize=(20, 6))

# Relationship between Year and Price
sns.regplot(data=car_data, x='Year', y='Price', ax=axes[0], line_kws={"color": "red"})
axes[0].set_title('Relationship between Year and Price')

# Relationship between Kilometers_Driven and Price
sns.scatterplot(data=car_data, x='Kilometers_Driven', y='Price', ax=axes[1])
axes[1].set_title('Relationship between Kilometers and Price')

# Price distribution by transmission type
sns.boxplot(data=car_data, x='Transmission', y='Price', ax=axes[2])
axes[2].set_title('Price distribution by transmission type')

plt.tight_layout()
plt.show()
```

**Analysis Results:**

- The newer the model year (the higher the number), the higher the price. (Positive correlation)
- The higher the mileage, the lower the price. (Negative correlation)
- Automatic transmission vehicles have a higher average price than manual transmission vehicles.

# MULTIVARIATE ANALYSIS - 1

One of the most effective ways to see the relationships between multiple variables at a glance is a Correlation Heatmap and a Pair Plot.

## 1. Correlation Heatmap

This is a map that calculates the correlation coefficient (a value between -1 and 1) between numeric variables and expresses it in colors. The closer it is to 1, the stronger the positive correlation, and the closer it is to -1, the stronger the negative correlation.

```
# Select only numeric variables to compute the correlation matrix.
corr = car_data.corr(numeric_only=True)

plt.figure(figsize=(8, 6))
# annot=True: display numbers in each cell, fmt='.2f': display up to two decimal places
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt='.2f', linewidths=.5)
plt.title('Heatmap of correlations between variables')
plt.show()
```

The heatmap shows at a glance that Price is positively correlated with Year and negatively correlated with Kilometers_Driven.

# MULTIVARIATE ANALYSIS - 2

## 2. Pair Plot

A powerful function that draws all variable pair relationships at once for numeric variables in a data frame. The diagonal shows the distribution (histogram or kde) of each variable, and the remaining space shows a scatterplot of variable pairs.

```
# Be careful with pairplots, they can be computationally intensive.
# The hue argument allows you to color-code specific categorical variables.
sns.pairplot(car_data, hue='Transmission', diag_kind='kde')
plt.show()
```

**Analysis Results:**

- With pairplot, you can see the relationship between all pairs of numeric variables and the distribution of each variable at a glance. The hue option is also very useful because you can see how the distribution of points changes depending on the type of transmission.

# SUMMARY

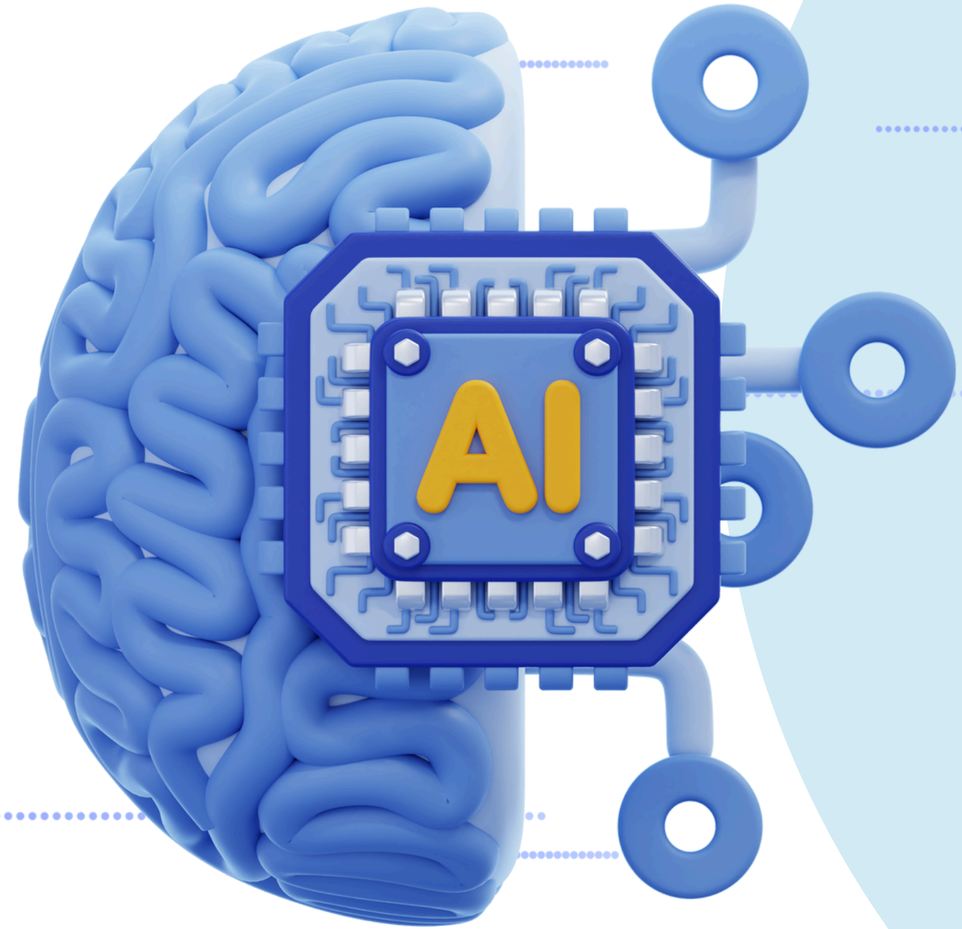
## Summary

- Seaborn is a high-level library based on Matplotlib that allows you to draw beautiful and statistical graphs with less code.
- It is highly compatible with Pandas DataFrame and can be used by directly passing DataFrame and column names to data, x, and y arguments.
- Exploratory Data Analysis (EDA) deepens your understanding of data through the process of univariate, bivariate, and multivariate analysis.
- You can effectively perform EDA by utilizing various functions of Seaborn such as histplot, boxplot, scatterplot, heatmap, and pairplot.

**Glossary of terms:**

- Seaborn: A Python data visualization library based on Matplotlib. Specialized in statistical graphs.
- Box Plot: A graph that visualizes the distribution of data by visualizing quartiles, medians, and outliers of data.
- Violin Plot: A graph that combines Box Plot with Kernel Density Estimation (KDE) to show the distribution of data.
- Correlation Heatmap: A table that expresses correlations between variables in colors. It is easy to see the relationship between variables at a glance.
- Pair Plot: A graph that shows scatter plots and the distribution of each variable for all pairs of numeric variables in a data frame at once. Useful for viewing the entire data.

# 04
# Integrated Practice Example



# MINI PROJECT: ANALYZING TITANIC SURVIVOR DATA - 1

**Project Goal**

- We will analyze the passenger data of the Titanic to answer the question, "What factors influenced the survival of the passengers?" As data detectives, we will use the powerful magnifying glass of visualization to find clues that differentiate between survival and death.

**Introduction to the Dataset**

We will use the 'titanic' dataset built into the Seaborn library. The main columns are:

- survived: Survival status (0 = death, 1 = survival) - our target variable!
- pclass: cabin class (1 = 1st class, 2 = 2nd class, 3 = 3rd class)
- sex: gender ('male', 'female')
- age: age
- sibsp: number of siblings or spouses traveling together
- parch: number of parents or children traveling together
- fare: fare
- embarked: port of embarkment (C = Cherbourg, Q = Queenstown, S = Southampton)
- who: adult male/female/child classification ('man', 'woman', 'child')
- alone: whether traveling alone (True/False)

# MINI PROJECT: ANALYZING TITANIC SURVIVOR DATA - 2

### Analysis Steps (Step-by-Step Guide)

Follow the steps below to fill out your own analysis notebook!

### Step 1: Loading Data and Basic Exploration

- First, load the data and use head(), info(), and describe() to get an overview of what information is in it.

```
# Load titanic dataset from Seaborn
import seaborn as sns
import matplotlib.pyplot as plt

# Set Korean font (can be omitted if executed in previous cell)
plt.rc('font', family='NanumBarunGothic')

titanic = sns.load_dataset('titanic')

# Check the beginning of data
print("--- Data sample ---")
print(titanic.head())

# Check basic data information (missing values, etc.)
print("\n--- Data information ---")
titanic.info()
```

# MINI PROJECT: ANALYZING TITANIC SURVIVOR DATA - 3

**Step 2: Finding Clues through Questions and Visualizations**

Choose and draw the most appropriate graph to answer the questions below.

- **Question 1. What is the ratio of total survivors to deaths?**
  - Hint: Count the values in the survived column.
  - Recommended graph: sns.countplot()
- **Question 2. Did sex affect survival? (e.g. Is it true that "women and children first"?)**
  - Hint: Compare the distribution of survived based on the sex column.
  - Recommended graph: Add hue='survived' option to sns.countplot()
- **Question 3. Did the survival rate differ depending on the room class (pclass)?**
  - Hint: Compare the number of survivors and deaths by room class.
  - Recommended graph: Add x='pclass', hue='survived' options to sns.countplot()
- **Question 4. What is the age distribution, and how did age affect survival?**
  - Hint: Compare the age distributions of the survivor group and the deceased group separately.
  - Recommended graph: Add hue='survived' option to sns.kdeplot() or sns.histplot()
- **Question 5. Did people who paid more fares live longer?**
  - Hint: Room class and fares are closely related. Compare the fare distributions by whether or not they survived.
  - Recommended graph: Add x='survived', y='fare' options to sns.boxplot() or sns.violinplot()

# MINI PROJECT: ANALYZING TITANIC SURVIVOR DATA - 4

**Step 3: Draw a comprehensive conclusion**

Based on the graphs drawn above, summarize your own conclusion in 2-3 sentences about what factors you think had the greatest impact on the Titanic's survival.

**(Example)**

- The analysis results show that gender and cabin class had a decisive impact on survival. In particular, the survival rate for women was overwhelmingly higher than for men, and the survival rate for first-class passengers was much higher than for third-class passengers. This suggests that women and passengers in higher cabins may have been rescued first in an emergency situation.