Machine Learning

Pandas Basics

Dept. SW and Communication Engineering

Prof. Giseop Noh (kafa46@hongik.ac.kr)

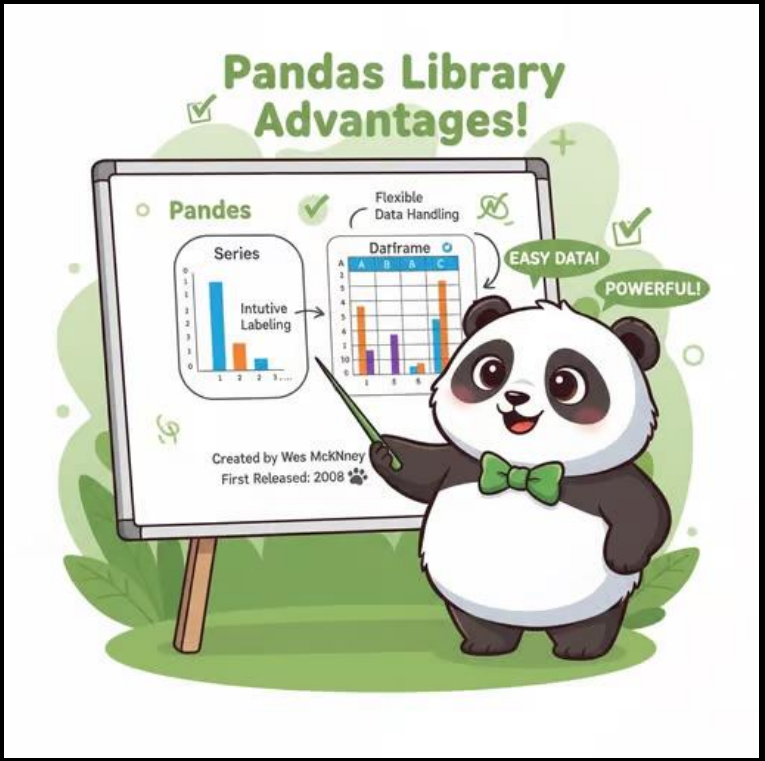
Lecture Goals

- **Understand the role and advantages of Pandas in data analysis and machine learning workflows**
- **Learn how to create and manipulate Series and DataFrame objects**
- **Perform indexing, selection, and conditional filtering on data**
- **Handle missing values using dropna() and fillna() techniques**
- **Apply basic statistical operations, grouping, and merging to explore datasets**
- **Build practical skills for data preprocessing prior to machine learning tasks**

Introduction to Pandas

Why Pandas?

- **Real-world datasets are rarely clean:**
  - Missing values, mixed data types, irregular structures
- **Python lists & NumPy arrays are great for numerical computing**
  - but not ideal for labeled, heterogeneous, tabular data
- **Pandas fills this gap with powerful, flexible data structures**



What is Pandas?

- **Developed by Wes McKinney (2008)**
- **High-level library for structured data manipulation**
- **Built on NumPy → integrates with Matplotlib, scikit-learn, etc.**
- **Supports the entire EDA (Exploratory Data Analysis) pipeline**

Core Data Structures

Series

- 1D labeled array
- Like a column in a spreadsheet
- Official Doc: https://pandas.pydata.org/docs/reference/api/pandas.Series.html

DataFrame

- 2D labeled data structure
- Like a spreadsheet or relational table
- Official Doc: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

Why Pandas Matters in ML

- **Importing and inspecting raw data**
- **Handling missing values**
- **Feature engineering & transformation**
- **Preparing model-ready inputs efficiently**

Series

Series — One-Dimensional Labeled Array

- **A Series is a 1D labeled array in Pandas.**
- **Holds any data type (int, float, string, object).**
- **Conceptually like a column**
  in a spreadsheet or field in a database.
- **Each value is paired with an index**
  (default or custom).

Creating a Series

Creating a Series from a List

```
import pandas as pd

# Create a Series with explicit index labels
data = pd.Series([10, 20, 30], index=["A", "B", "C"])
print(data)
```

Execution Result

```
A    10
B    20
C    30
dtype: int64
```

- Values: [10, 20, 30]
- Index: "A", "B", "C"
- Data type: int64

Accessing Index and Values

Accessing Index and Values

```
import pandas as pd
# Create a Series with explicit index labels
data = pd.Series([10, 20, 30], index=["A", "B", "C"])
print(data.index)
print(data.values)
```

Execution Result

```
Index(['A', 'B', 'C'], dtype='object')
[10 20 30]
```

- data.index

  → returns the labels of the Series ('A', 'B', 'C').

- data.values

  → returns the underlying NumPy array ([10, 20, 30]).

- Separation of labels and values makes Series more flexible than NumPy arrays.

  👉 Enables intuitive data alignment, indexing, and manipulation.

DataFrame

DataFrame — Two-Dimensional Labeled Data Structure

- **A DataFrame is a 2D labeled data structure with columns of potentially different types.**
- **Conceptually like a spreadsheet or relational database table.**
  - Rows = records
  - Columns = variables
- **Constructed from:**
  - Python dictionaries
  - Lists of dictionaries
  - External data files (e.g., CSV, TSV, Excel)

Creating a DataFrame from a Dict

Creating a Series from a List

```
import pandas as pd
data_dict = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age":  [25, 30, 35],
    "Score": [85, 90, 95]
}
df_from_dict = pd.DataFrame(data_dict)
print(df_from_dict)
```

Execution Result

	Name	Age	Score
0	Alice	25	85
1	Bob	30	90
2	Charlie	35	95

- Keys → column labels ("Name", "Age", "Score")
- Values → data sequences
- Default row index = integers (0, 1, 2)
- Columns can hold different data types

Creating from a List of Dictionaries

Creating from a List of Dictionaries

```
import pandas as pd
data_list = [
    {"Name": "Alice", "Age": 25, "Score": 85},
    {"Name": "Bob", "Age": 30, "Score": 90},
    {"Name": "Charlie", "Age": 35, "Score": 95}
]
df_from_list = pd.DataFrame(data_list)
print(df_from_list)
```

Execution Result

	Name	Age	Score
0	Alice	25	85
1	Bob	30	90
2	Charlie	35	95

- Each dict represents a row.
- Keys = column names, Values = row entries.
- Useful when loading data from JSON-like structures or APIs.
- Pandas automatically aligns columns based on the dict keys.

Creating a DataFrame from a CSV File

Importing Data from a CSV File

```python
import pandas as pd

# Load CSV file into a DataFrame
df_from_csv = pd.read_csv("05_random_students.csv")

# Display the first 5 rows
print(df_from_csv.head())
```

Execution Result

```
      Name  Age  Score
0    Alice   25     85
1      Bob   30     90
2  Charlie   35     95
```

- pd.read_csv("sample_students.csv") reads the file and constructs a DataFrame.
- The first row is used as column headers ("Name", "Age", "Score").
- Each subsequent line becomes a row in the DataFrame.
- Pandas automatically assigns a default integer index (0, 1, 2, ...).

Column should be used as the index

■ If a column used as the index, the parameter index_col can be specified

```
import pandas as pd

# Load CSV file into a DataFrame
df_csv_indexed = pd.read_csv("sample_students.csv", index_col="Name")
print(df_csv_indexed)
```

■ **Execution Result**

Name	Age	Score
Alice	25	85
Bob	30	90
Charlie	35	95

- Here, the "Name" column is used as the row index, which can be useful when the column uniquely identifies each record.

Using .head(), .info(), and .describe()

- **Once a DataFrame is created,**
  - Often necessary to quickly inspect its structure and contents
  - Pandas provides several convenient methods for this purpose.
- **.head(n)**
  - Displays the first n rows (default is 5).
  - Useful for quickly checking the beginning of a dataset.
- **.info( )**
  - Prints a concise summary including column names, non-null counts, and data types.
- **.describe( )**
  - Generates descriptive statistics (count, mean, std, min, quartiles, max) for numerical columns.

Generate & Save Example Dataset

- Generate 1,000 rows dataset for demonstration

```python
import pandas as pd
import numpy as np

# Generate random dataset
np.random.seed(42)
n = 1000

data = {
    "StudentID": range(1, n+1),
    "Age": np.random.randint(18, 30, size=n),
    "MathScore": np.random.randint(50, 100, size=n),
    "EnglishScore": np.random.randint(50, 100, size=n),
    "ScienceScore": np.random.randint(50, 100, size=n)
}

df = pd.DataFrame(data)

# Save to CSV
df.to_csv("05_random_students.csv", index=False)
```

Loading the CSV File

Loading the CSV File using Pandas

```
import pandas as pd

# Load the dataset
df = pd.read_csv("students_scores.csv")

# Show the first 5 rows
print(df.head())

print(df.info())

print(df. describe())
```

	StudentID	Age	MathScore	EnglishScore	ScienceScore
0	1	24	88	61	58
1	2	28	51	92	73
2	3	28	89	74	92
3	4	23	83	87	72
4	5	27	94	66	87

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   StudentID     1000 non-null   int64
 1   Age           1000 non-null   int64
 2   MathScore     1000 non-null   int64
 3   EnglishScore  1000 non-null   int64
 4   ScienceScore  1000 non-null   int64
dtypes: int64(5)
memory usage: 39.2 KB
```

	StudentID	Age	MathScore	EnglishScore	ScienceScore
count	1000.00000	1000.000000	1000.000000	1000.000000	1000.000000
mean	500.50000	23.944000	74.393000	74.348000	74.189000
std	288.81944	3.430000	14.367000	14.565000	14.442000
min	1.00000	18.000000	50.000000	50.000000	50.000000
25%	250.75000	21.000000	62.000000	62.000000	62.000000
50%	500.50000	24.000000	75.000000	74.000000	74.000000
75%	750.25000	27.000000	87.000000	87.000000	87.000000
max	1000.00000	29.000000	99.000000	99.000000	99.000000

Summary & Interpretation

- **DataFrame is the core structure of Pandas**
  - Represents data in a tabular format with labeled rows and columns

- **We explored three creation methods**
  - From a Python dictionary
  - From a list of dictionaries
  - By importing a CSV file

- **We learned inspection methods**
  - .head() → Quick preview of data
  - .info() → Structure, data types, missing values
  - .describe() → Basic statistics for EDA

Indexing

Index in Series

Index in Series

- A Series = values + index labels
- The index uniquely identifies each element in the Series
- Default index: integers starting from 0
- Users can define custom labels (e.g., strings, dates)
- Flexible access
  - series[0] → access by position
  - series['A'] → access by label

Index in DataFrame

Index in DataFrame

- A DataFrame extends indexing into two dimensions:
  - **Row index** → Identifies each record (default: 0, 1, 2, …)
  - **Column index** → Identifies each variable (column) and is typically defined by the dataset's header or assigned during creation.

Dual indexing provides powerful and flexible data access

- Row access by label → df.loc[0]
- Column access by name → df['Age']
- Row + Column → df.loc[0, 'Age']

Label-based Indexing with .loc

```
import pandas as pd


data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Score": [85, 90, 95]
}
df = pd.DataFrame(data, index=["S1", "S2", "S3"])


# Access by label
print(df)
print("*****" * 5)
# Entire row with label "S1"
print(df.loc["S1"])
print("*****" * 5)
# Value in row "S2", column "Age"
print(df.loc["S2", "Age"])
print("*****" * 5)
# Subset of rows and one column
print(df.loc[["S1", "S3"], "Name"])
```

The .loc accessor selects rows and columns using their **explicit labels**.
- Rows are identified by their index labels.
- Columns are identified by their names.

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

```
    Name  Age  Score
S1   Alice   25     85
S2     Bob   30     90
S3 Charlie   35     95
*************************
Name     Alice
Age         25
Score       85
Name: S1, dtype: object
*************************
30
*************************
S1       Alice
S3     Charlie
Name: Name, dtype: object
```

Position-based Indexing with .iloc

```python
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Score": [85, 90, 95]
}
df = pd.DataFrame(data, index=["S1", "S2", "S3"])

# Access by label
print(df)
print("*****" * 5)
print(df.iloc[0])        # First row
print("*****" * 5)
print(df.iloc[1, 1])     # Second row, second column
print("*****" * 5)
print(df.iloc[[0, 2], 0]) # First and third rows,
                          # first column
```

.loc uses labels to access rows and columns,
the .iloc accessor is purely **integer position-based**.

It works like Python list slicing, where both rows and columns are referenced by zero-based indices.

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

```
   Name  Age  Score
S1    Alice   25     85
S2      Bob   30     90
S3  Charlie   35     95
*************************
Name      Alice
Age          25
Score        85
Name: S1, dtype: object
*************************
30
*************************
S1       Alice
S3     Charlie
Name: Name, dtype: object
```

Label-based Indexing (.loc) vs. Position-based Indexing (.iloc)

Basis of Selection

- .loc: Uses labels (row index names and column names).
- .iloc: Uses integer positions (0-based index, like Python lists).

Inclusion Rules in Slicing

- .loc: Both the start and end labels are inclusive.

```
df.loc["S1":"S3"]  # includes rows S1, S2, and S3
```

- .iloc: The end position is exclusive, following Python slicing rules.

```
df.iloc[0:3]  # includes rows at positions 0, 1, and 2
```

Typical Use Cases

- .loc
  - More intuitive when datasets have meaningful labels.
  - Commonly used when accessing data by identifiers such as student IDs, dates, or names.
- .iloc
  - Useful when working with purely positional logic.
  - Preferred in loops, algorithmic selection, or when labels are not known in advance.

Boolean Indexing (Conditional Selection)

Boolean Indexing (Conditional Selection)

- Allows selecting rows based on conditions applied to columns
  - Instead of labels or positions, create a Boolean mask → a Series of True/False
  - Only rows where the mask is True are returned

Boolean Indexing (Conditional Selection) - codes

```python
import pandas as pd


data = {
    "Name": ["Alice", "Bob", "Charlie", "Dana"],
    "Age": [25, 30, 35, 28],
    "Score": [85, 90, 65, 72]
}
df = pd.DataFrame(data)


# Select rows where Age is greater than 28
print(df[df["Age"] > 28])


# Select rows where Score is at least 80
print(df[df["Score"] >= 80])


# Combine conditions with & (and), | (or)
print(df[(df["Age"] > 25) & (df["Score"] > 70)])
```

```
      Name  Age  Score
2  Charlie   35     65
3     Dana   28     72

    Name  Age  Score
0  Alice   25     85
1    Bob   30     90

    Name  Age  Score
1    Bob   30     90
3   Dana   28     72
```

Resetting and Setting Index

- In Pandas, the index is flexible and can be reset or reassigned to better reflect the structure of the dataset.

- Two useful methods are:
  - reset_index(): Resets the existing index to the default integer index. The old index can be preserved as a column if needed.
  - set_index(): Assigns one of the columns to become the new index.

Resetting and Setting Index - codes

```python
import pandas as pd


data = {
    "StudentID": [101, 102, 103],
    "Name": ["Alice", "Bob", "Charlie"],
    "Score": [85, 90, 95]
}
df = pd.DataFrame(data)


# Set "StudentID" as the index
df_indexed = df.set_index("StudentID")
print("DataFrame with StudentID as index:")
print(df_indexed)


# Reset index back to default integers
df_reset = df_indexed.reset_index()
print("\nDataFrame after reset_index():")
print(df_reset)
```

```
DataFrame with StudentID as index:
           Name  Score
StudentID
101       Alice     85
102         Bob     90
103     Charlie     95

DataFrame after reset_index():
   StudentID     Name  Score
0        101    Alice     85
1        102      Bob     90
2        103  Charlie     95
```

Selection

Selecting a Single Column

- **A single column can be selected by using its column label.**
  - The result is a Series.

```
import pandas as pd


data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Score": [85, 90, 95]
}
df = pd.DataFrame(data)


# Select a single column
print(df["Name"])
```

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

Selecting Multiple Columns

- Multiple columns can be selected by passing a list of column labels.
  - The result is a DataFrame.

```
import pandas as pd


data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Score": [85, 90, 95]
}

print(df[["Name", "Score"]])
```

```
      Name  Score
0    Alice     85
1      Bob     90
2  Charlie     95
```

Slicing

- Slicing can be used for row ranges or column ranges.

```python
import pandas as pd


data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Score": [85, 90, 95]
}
df = pd.DataFrame(data)


# Slice rows
print(df[0:2])


# Slice rows and columns together
print(df.loc[0:2, ["Name", "Age"]])
```

```
 Name  Age  Score
0  Alice   25     85
1    Bob   30     90

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

Handling Missing Data

Using dropna() and fillna()

- Real-world datasets often contain missing values, which must be addressed before applying machine learning models.
  - Pandas provides simple methods to handle missing data.

- **dropna()**
  - Removes rows or columns that contain missing values.

- **fillna()**
  - Replaces missing values with a specified constant or a computed value (e.g., mean, median).

Example

```python
import pandas as pd
import numpy as np
data = {
    "Name": ["Alice", "Bob", "Charlie", "Dana"],
    "Age": [25, np.nan, 35, 28],
    "Score": [85, 90, np.nan, 72]
}
df = pd.DataFrame(data)
print(df)
print("-----" * 5)

# Remove rows with any missing values
df_drop = df.dropna()
print(df_drop)
print("-----" * 5)

# Replace missing values with 0
df_fill_zero = df.fillna(0)
print(df_fill_zero)
print("-----" * 5)

# Replace missing values with the mean of the column
df_fill_mean = df.fillna(df.mean(numeric_only=True))
print(df_fill_mean)
```

```
      Name   Age  Score
0    Alice  25.0   85.0
1      Bob   NaN   90.0
2  Charlie  35.0    NaN
3     Dana  28.0   72.0
-------------------------
      Name   Age  Score
0    Alice  25.0   85.0
3     Dana  28.0   72.0
-------------------------
      Name   Age  Score
0    Alice  25.0   85.0
1      Bob   0.0   90.0
2  Charlie  35.0    0.0
3     Dana  28.0   72.0
-------------------------
      Name        Age      Score
0    Alice  25.000000  85.000000
1      Bob  29.333333  90.000000
2  Charlie  35.000000  82.333333
3     Dana  28.000000  72.000000
```

Basic Operations

Summary Statistics – mean, max, min

```python
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Charlie", "Dana"],
    "Age": [25, 30, 35, 28],
    "Score": [85, 90, 95, 72]
}

df = pd.DataFrame(data)

# Summary statistics
print(df["Score"].mean())   # Mean
print(df["Score"].max())    # Maximum
print(df["Score"].min())    # Minimum
print(df.describe())        # Comprehensive summary
```

```
85.5
95
72
             Age       Score
count   4.000000    4.000000
mean   29.500000   85.500000
std     4.203173    9.574271
min    25.000000   72.000000
25%    27.250000   81.750000
50%    29.000000   87.500000
75%    31.250000   91.250000
max    35.000000   95.000000
```

Using groupby

```
import pandas as pd


data = {
    "Department": ["Math", "Math", "CS", "CS", "Physics"],
    "Student": ["Alice", "Bob", "Charlie", "Dana", "Evan"],
    "Score": [85, 90, 95, 72, 88]
}
df = pd.DataFrame(data)


# Average score by department
print(df)
print("*****" * 5)
print(df.groupby("Department")["Score"].mean())
```

```
  Department  Student  Score
0       Math    Alice     85
1       Math      Bob     90
2         CS  Charlie     95
3         CS     Dana     72
4    Physics     Evan     88
**********
Department
CS         83.5
Math       87.5
Physics    88.0
Name: Score, dtype: float64
```

Using merge

```python
import pandas as pd

students = pd.DataFrame({
    "StudentID": [1, 2, 3],
    "Name": ["Alice", "Bob", "Charlie"]
})

scores = pd.DataFrame({
    "StudentID": [1, 2, 3],
    "Score": [85, 90, 95]
})

print(students)
print("*****" * 5)
print(scores)
print("*****" * 5)

# Merge on StudentID
df_merged = pd.merge(students, scores, on="StudentID")
print(df_merged)
```

```
 StudentID      Name
0          1    Alice
1          2      Bob
2          3  Charlie
************************
   StudentID  Score
0          1     85
1          2     90
2          3     95
************************
   StudentID      Name  Score
0          1    Alice     85
1          2      Bob     90
2          3  Charlie     95
```

Homeworks

- https://www.deepshark.org/courses/machine_learning_1/w/05_pandas_basics#homeworks

Thank you!