Pandas

Contents

Pandas is a powerful open-source Python library for **data manipulation and analysis**. It was initially created in **2008** by **Wes McKinney** while working at AQR Capital to address the need for a flexible and high-performance tool for financial data analysis. Since then, Pandas has become one of the most widely used libraries in **data science** and **machine learning**.

Relation to Python

- Built entirely in **Python**, with heavy reliance on **NumPy** for fast numerical operations.
- Provides an intuitive way to handle **structured data** (tables, spreadsheets, time series).
- Integrates seamlessly with other Python libraries like **Matplotlib**, **SciPy**, **Scikit-learn**, **TensorFlow**, and **PyTorch**.

Importance for AI & Machine Learning

- 🖌️ **Data Cleaning**: Preprocessing raw datasets before training AI models.
- 📊 **Feature Engineering**: Creating, transforming, and scaling features used in ML algorithms.
- ⏱️ **Time Series Analysis**: Useful for forecasting models.
- 🔗 **Integration**: Works as a bridge between raw data (CSV, SQL, JSON) and ML frameworks.

Core Components of Pandas

- **Series** → One-dimensional labeled array (like a column).
- **DataFrame** → Two-dimensional labeled data structure (like an Excel table).

Why Pandas?

- 🚀 Easy-to-use data structures.
- 📊 Tools for reading/writing data (CSV, Excel, SQL, JSON).
- 🔍 Rich indexing and selection.
- 🧮 Powerful groupby and aggregation.
- ⚡ Handles missing data gracefully.

📌 Official Website: https://pandas.pydata.org/

```python
import pandas as pd

# Check installed version
print("Pandas version:", pd.__version__)
```

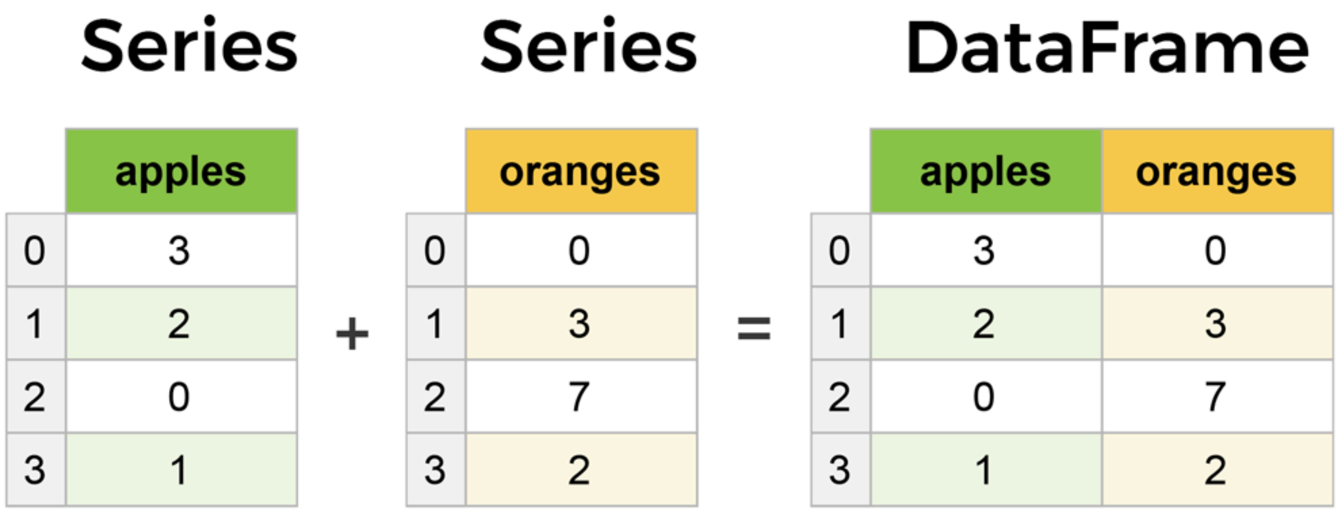
```
Pandas version: 2.2.3
```

Pandas Series vs DataFrame

Pandas has two **core data structures**:

- **Series** → One-dimensional labeled array.
- **DataFrame** → Two-dimensional table of data (like an Excel sheet).

Visual Representation:



From the image:

- A **Series** is like one column (e.g., apples or oranges).
- A **DataFrame** is like combining multiple Series side by side.

Example: Series

A **Series** is a one-dimensional labeled array. It has an **index** and **values**.

Example Code:

```
apples = pd.Series([3, 2, 0, 1], name="apples")
print("Series - Apples:")
print(apples)

print("\nAccess element at index 1:", apples[1])
```

```
Series - Apples:
0    3
1    2
2    0
3    1
Name: apples, dtype: int64

Access element at index 1: 2
```

Example: DataFrame

A **DataFrame** is a two-dimensional labeled data structure with rows and columns.

You can build it by combining multiple Series together.

Example Code:

```
apples = pd.Series([3, 2, 0, 1], name="apples")
oranges = pd.Series([0, 3, 7, 2], name="oranges")

df = pd.DataFrame({"apples": apples, "oranges": oranges})
print("DataFrame:")
print(df)
```

```
DataFrame:
   apples  oranges
0       3        0
1       2        3
2       0        7
3       1        2
```

Data Indexing and Selection

In Pandas, **indexing and selection** are very powerful tools that let you access rows and columns efficiently.

Two Main Methods:

- `.loc[]` → **Label-based indexing**
  - Use when you know the **row/column labels**.
  - Safer when working with **non-integer indexes** (e.g., names, dates).
- `.iloc[]` → **Integer position-based indexing**
  - Use when you want to access by **row/column position**.
  - Useful in loops or when labels are not known.

Comparison Table:

Method	Input Type	Example	Output
`loc`	Labels (row/col)	`df.loc[1, "name"]`	'Bob'
`iloc`	Integers (position)	`df.iloc[2, 1]`	35
Column	Column name	`df["age"]`	Series
Slice	Row slice	`df[1:3]`	Rows 1–2

📌 **Rule of Thumb:**

- Use `loc` when working with **labels** (more explicit).
- Use `iloc` when working with **positions** (like Python lists).

```python
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    "name": ["Alice", "Bob", "Charlie"],
    "age": [25, 30, 35],
    "city": ["NY", "LA", "Chicago"]
})

print("DataFrame:")
print(df)

# Access examples
print("\nAccess with loc (row index=1, column 'name'):", df.loc[1, "name"])
print("Access with iloc (row position=2, col position=1):", df.iloc[2, 1])

print("\nAge column:")
df["age"]

print("\nRows 1 to 2 (slice):")
df[1:3]
```

```
DataFrame:
      name  age     city
0    Alice   25       NY
1      Bob   30       LA
2  Charlie   35  Chicago

Access with loc (row index=1, column 'name'): Bob
Access with iloc (row position=2, col position=1): 35

Age column:

Rows 1 to 2 (slice):
```

	name	age	city
1	Bob	30	LA
2	Charlie	35	Chicago

Operations on Data

Pandas makes it easy to **add new columns**, **apply functions**, and **compute statistics** directly on DataFrames.

Example Use Cases:

- Add calculated columns.
- Create boolean flags (e.g., `is_adult`).
- Apply custom functions to each value.

```
# Add 5 to each age
df["age_plus_5"] = df["age"] + 5

# Create boolean column (True/False)
df["is_adult"] = df["age"] >= 18

# Apply function: string length of names
df["name_length"] = df["name"].apply(len)
```

```
# Add new calculated columns
df["age_plus_5"] = df["age"] + 5
df["is_adult"] = df["age"] >= 18
df["name_length"] = df["name"].apply(len)

print("Updated DataFrame with new columns:")
df
```

```
Updated DataFrame with new columns:
```

	name	age	city	age_plus_5	is_adult	name_length
0	Alice	25	NY	30	True	5
1	Bob	30	LA	35	True	3
2	Charlie	35	Chicago	40	True	7

In this notebook, we will use **Pandas** to:

- Read CSV and JSON files
- Analyze data
- Clean data: handle empty cells, wrong formats, invalid values, duplicates

```
import pandas as pd
import numpy as np

print("Pandas version:", pd.__version__)
```

```
Pandas version: 2.2.3
```

Create Example Bank Dataset

We will simulate a small **bank customers dataset**.

```python
data = {
    "CustomerID": [1, 2, 3, 4, 5, 5],
    "Name": ["Alice", "Bob", "Charlie", "David", "Eva", "Eva"],
    "Age": [25, 30, None, 40, "thirty", 28],
    "Balance": [1000.50, 2500.00, 3000.75, None, 1500.20, 1500.20],
    "JoinDate": ["2022-01-10", "2021-05-12", "wrong_date", "2020-07-15", "2022-03-20"
}

df = pd.DataFrame(data)
df.to_csv("bank_data.csv", index=False)
df.to_json("bank_data.json", orient="records", lines=True)

print("Bank dataset created: bank_data.csv & bank_data.json")
df
```

```
Bank dataset created: bank_data.csv & bank_data.json
```

	CustomerID	Name	Age	Balance	JoinDate
0	1	Alice	25	1000.50	2022-01-10
1	2	Bob	30	2500.00	2021-05-12
2	3	Charlie	None	3000.75	wrong_date
3	4	David	40	NaN	2020-07-15
4	5	Eva	thirty	1500.20	2022-03-20
5	5	Eva	28	1500.20	2022-03-20

Reading Data: CSV and JSON

```python
# Read CSV
df_csv = pd.read_csv("bank_data.csv")
print("CSV Data:")
display(df_csv)

# Read JSON
df_json = pd.read_json("bank_data.json", lines=True)
print("JSON Data:")
display(df_json)
```

```
CSV Data:
```

	CustomerID	Name	Age	Balance	JoinDate
0	1	Alice	25	1000.50	2022-01-10
1	2	Bob	30	2500.00	2021-05-12
2	3	Charlie	NaN	3000.75	wrong_date
3	4	David	40	NaN	2020-07-15
4	5	Eva	thirty	1500.20	2022-03-20
5	5	Eva	28	1500.20	2022-03-20

```
JSON Data:
```

	CustomerID	Name	Age	Balance	JoinDate
0	1	Alice	25	1000.50	2022-01-10
1	2	Bob	30	2500.00	2021-05-12
2	3	Charlie	None	3000.75	wrong_date
3	4	David	40	NaN	2020-07-15
4	5	Eva	thirty	1500.20	2022-03-20
5	5	Eva	28	1500.20	2022-03-20

Analyzing Data

- `.head()` → preview first rows

- `.info()` → summary of dataset
- `.describe()` → statistical summary

```
print(df.head())
print("\nDataset Info:")
print(df.info())
print("\nStatistical Summary:")
print(df.describe(include="all"))
```

```
   CustomerID     Name     Age  Balance    JoinDate
0           1    Alice      25  1000.50  2022-01-10
1           2      Bob      30  2500.00  2021-05-12
2           3  Charlie    None  3000.75  wrong_date
3           4    David      40      NaN  2020-07-15
4           5      Eva  thirty  1500.20  2022-03-20

Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   CustomerID  6 non-null      int64  
 1   Name        6 non-null      object 
 2   Age         5 non-null      object 
 3   Balance     5 non-null      float64
 4   JoinDate    6 non-null      object 
dtypes: float64(1), int64(1), object(3)
memory usage: 372.0+ bytes
None

Statistical Summary:
        CustomerID Name   Age      Balance    JoinDate
count     6.000000    6   5.0     5.000000           6
unique         NaN    5   5.0          NaN           5
top            NaN  Eva  25.0          NaN  2022-03-20
freq           NaN    2   1.0          NaN           2
mean      3.333333  NaN   NaN  1900.330000         NaN
std       1.632993  NaN   NaN   821.649309         NaN
min       1.000000  NaN   NaN  1000.500000         NaN
25%       2.250000  NaN   NaN  1500.200000         NaN
50%       3.500000  NaN   NaN  1500.200000         NaN
75%       4.750000  NaN   NaN  2500.000000         NaN
max       5.000000  NaN   NaN  3000.750000         NaN
```

Cleaning Data

1. Cleaning Empty Cells

```
print("Rows with missing values:")
print(df[df.isnull().any(axis=1)])

# Make a copy
df_filled = df.copy()

# Convert Age & Balance to numeric (force errors to NaN)
df_filled["Age"] = pd.to_numeric(df_filled["Age"], errors="coerce")
df_filled["Balance"] = pd.to_numeric(df_filled["Balance"], errors="coerce")

# Fill missing values
df_filled["Age"] = df_filled["Age"].fillna(df_filled["Age"].median())
df_filled["Balance"] = df_filled["Balance"].fillna(0)

df_filled
```

```
Rows with missing values:
   CustomerID     Name   Age  Balance    JoinDate
2           3  Charlie  None  3000.75  wrong_date
3           4    David    40      NaN  2020-07-15
```

	CustomerID	Name	Age	Balance	JoinDate
0	1	Alice	25.0	1000.50	2022-01-10
1	2	Bob	30.0	2500.00	2021-05-12
2	3	Charlie	29.0	3000.75	wrong_date
3	4	David	40.0	0.00	2020-07-15
4	5	Eva	29.0	1500.20	2022-03-20
5	5	Eva	28.0	1500.20	2022-03-20

Reading Data with Pandas

Pandas provides easy methods to load data from different formats like **CSV** and **JSON**.

Why important?

- Real-world datasets are often stored in CSV/JSON.
- Pandas provides a **fast, unified interface** to load them.
- Makes it easy to start analyzing immediately.

Supported formats: CSV, JSON, Excel, SQL, Parquet, etc.

Reading CSV File

We will use the sample dataset: `datasets/sample_data/sample_bank_data.csv`

```
import pandas as pd

df_csv = pd.read_csv("datasets/sample_data/sample_bank_data.csv")
print(df_csv.head())
```

```
import pandas as pd

# Read CSV
df_csv = pd.read_csv("datasets/sample_data/sample_bank_data.csv")
print("CSV Data Preview:")
df_csv.head()
```

```
CSV Data Preview:
```

	account_id	name	age	gender	city	account_type	balance	loan_amou
0	MY1000	Aisyah Bin Ali	56.0	Male	Kuala Lumpur	Savings	47747.78	53970
1	MY1001	Siti Bin Hafiz	69.0	Male	Kota Kinabalu	Fixed Deposit	36921.06	0
2	MY1002	Aisyah Bin Fatimah	NaN	Female	Kuching	Savings	27762.27	34961
3	MY1003	Farid Bin Ahmad	32.0	Male	Penang	Savings	30624.87	35021
4	MY1004	Aisyah Bin Siti	60.0	Male	Kuching	Current	21038.04	73881

Reading JSON File

We will use the sample dataset located at: `datasets/sample_data/sample_bank_data.json`

```
df_json = pd.read_json("datasets/sample_data/sample_bank_data.json")
print(df_json.head())
```

Notes:

- `pd.read_json()` can read JSON files or JSON strings directly.
- Pandas automatically converts JSON into a structured **DataFrame**.
- Useful for APIs and web data that often come in JSON format.
- If the JSON has nested objects, use `json_normalize()` to flatten it.

```
import pandas as pd

# Read line-delimited JSON file
df_json = pd.read_json("datasets/sample_data/sample_bank_data.json", lines=True)
print("JSON Data Preview:")
df_json.head()
```

```
JSON Data Preview:
```

	account_id	name	age	gender	city	account_type	balance	loan_amou
0	MY1000	Aisyah Bin Ali	56.0	Male	Kuala Lumpur	Savings	47747.78	53970
1	MY1001	Siti Bin Hafiz	69.0	Male	Kota Kinabalu	Fixed Deposit	36921.06	0
2	MY1002	Aisyah Bin Fatimah	NaN	Female	Kuching	Savings	27762.27	34961
3	MY1003	Farid Bin Ahmad	32.0	Male	Penang	Savings	30624.87	35021
4	MY1004	Aisyah Bin Siti	60.0	Male	Kuching	Current	21038.04	73881

Analyzing Data

Once the dataset is loaded, Pandas provides quick methods to understand it:

- `.head()` → Preview first rows.
- `.info()` → Column names, data types, memory usage.
- `.describe()` → Statistical summary (mean, std, min, max, quartiles).
- `.shape` → Number of rows & columns.

```python
# Analyze CSV data
print("CSV Info:")
print(df_csv.info())

print("\nCSV Description:")
print(df_csv.describe())

print("\nShape (rows, columns):", df_csv.shape)
```

```
CSV Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 510 entries, 0 to 509
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   account_id        510 non-null    object 
 1   name              510 non-null    object 
 2   age               484 non-null    float64
 3   gender            510 non-null    object 
 4   city              485 non-null    object 
 5   account_type      510 non-null    object 
 6   balance           485 non-null    float64
 7   loan_amount       510 non-null    float64
 8   transaction_count  510 non-null    int64  
 9   last_transaction   510 non-null    object 
dtypes: float64(3), int64(1), object(6)
memory usage: 40.0+ KB
None

CSV Description:
              age       balance   loan_amount  transaction_count
count  484.000000    485.000000    510.000000         510.000000
mean    44.351240  24716.251134  22284.779196          99.141176
std     14.992496  14642.001337  31368.151813          57.625780
min     18.000000    331.140000      0.000000           1.000000
25%     32.000000  12083.270000      0.000000          48.000000
50%     45.000000  24771.570000      0.000000          99.500000
75%     57.000000  37438.160000  41234.527500         149.000000
max     69.000000  49985.910000  99688.990000         199.000000

Shape (rows, columns): (510, 10)
```

```python
# Analyze JSON data
print("JSON Info:")
print(df_json.info())

print("\nJSON Description:")
print(df_json.describe())

print("\nShape (rows, columns):", df_json.shape)
```

```
JSON Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 510 entries, 0 to 509
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   account_id         510 non-null    object 
 1   name               510 non-null    object 
 2   age                484 non-null    float64
 3   gender             510 non-null    object 
 4   city               485 non-null    object 
 5   account_type       510 non-null    object 
 6   balance            485 non-null    float64
 7   loan_amount        510 non-null    float64
 8   transaction_count  510 non-null    int64  
 9   last_transaction   510 non-null    int64  
dtypes: float64(3), int64(2), object(5)
memory usage: 40.0+ KB
None

JSON Description:
              age       balance   loan_amount  transaction_count  \
count  484.000000    485.000000    510.000000         510.000000   
mean    44.351240  24716.251134  22284.779196          99.141176   
std     14.992496  14642.001337  31368.151813          57.625780   
min     18.000000    331.140000      0.000000           1.000000   
25%     32.000000  12083.270000      0.000000          48.000000   
50%     45.000000  24771.570000      0.000000          99.500000   
75%     57.000000  37438.160000  41234.527500         149.000000   
max     69.000000  49985.910000  99688.990000         199.000000   

       last_transaction  
count      5.100000e+02  
mean       1.687963e+12  
std        8.906938e+09  
min        1.672704e+12  
25%        1.680761e+12  
50%        1.687824e+12  
75%        1.694995e+12  
max        1.703981e+12  

Shape (rows, columns): (510, 10)
```

When working with real-world datasets, you often encounter problems like:

- Empty cells (missing values)
- Wrong formats (dates stored as text)
- Wrong or unrealistic data (negative age, negative balance)
- Duplicate rows

If we don't clean the data properly, any **statistical analysis or machine learning model** will produce misleading results. For example: a missing value in `age` will distort the average age, which can bias the model.

```python
import pandas as pd

# Load dataset
df = pd.read_csv("datasets/sample_data/sample_bank_data.csv")
print(" Original Data Preview:")
df.head()
```

```
 Original Data Preview:
```

	account_id	name	age	gender	city	account_type	balance	loan_amou
0	MY1000	Aisyah Bin Ali	56.0	Male	Kuala Lumpur	Savings	47747.78	53970
1	MY1001	Siti Bin Hafiz	69.0	Male	Kota Kinabalu	Fixed Deposit	36921.06	0
2	MY1002	Aisyah Bin Fatimah	NaN	Female	Kuching	Savings	27762.27	34961
3	MY1003	Farid Bin Ahmad	32.0	Male	Penang	Savings	30624.87	35021
4	MY1004	Aisyah Bin Siti	60.0	Male	Kuching	Current	21038.04	73881

Cleaning Empty Cells

- `dropna()` → removes rows/columns with missing values.
- `fillna()` → fills missing values with a given value (e.g., `0`, `mean`, `median`).

Why it matters?

- Models cannot handle `NaN` values.

- Choice of replacement depends on context (e.g., missing `balance` → `0`, missing `age` → median).

```
df_clean = df.copy()

# Convert 'age' to numeric (invalid values → NaN)
df_clean['age'] = pd.to_numeric(df_clean['age'], errors='coerce')

# Fill missing ages with the median
df_clean['age'] = df_clean['age'].fillna(df_clean['age'].median())

# Fill missing balances with 0
df_clean['balance'] = df_clean['balance'].fillna(0)

print(" After filling missing values:")
df_clean.head()
```

```
 After filling missing values:
```

	account_id	name	age	gender	city	account_type	balance	loan_amou
0	MY1000	Aisyah Bin Ali	56.0	Male	Kuala Lumpur	Savings	47747.78	53970.
1	MY1001	Siti Bin Hafiz	69.0	Male	Kota Kinabalu	Fixed Deposit	36921.06	0.
2	MY1002	Aisyah Bin Fatimah	45.0	Female	Kuching	Savings	27762.27	34961.
3	MY1003	Farid Bin Ahmad	32.0	Male	Penang	Savings	30624.87	35021.
4	MY1004	Aisyah Bin Siti	60.0	Male	Kuching	Current	21038.04	73881.

Cleaning Wrong Format

Sometimes columns like **dates** are stored as plain text. We can convert them using `pd.to_datetime()`.

Why it matters?

- If dates remain as text, we cannot sort or calculate differences correctly.

```
df_clean['last_transaction'] = pd.to_datetime(df_clean['last_transaction'], errors='co
print(" After converting last_transaction to datetime:")
print(df_clean.dtypes)
```

```
 After converting last_transaction to datetime:
account_id                    object
name                          object
age                          float64
gender                        object
city                          object
account_type                  object
balance                      float64
loan_amount                  float64
transaction_count              int64
last_transaction      datetime64[ns]
dtype: object
```

Cleaning Wrong Data

Examples:

- Ages less than 0 or greater than 120 → unrealistic.
- Negative balances → invalid.
- Negative transaction counts → invalid.

Why it matters?

- Invalid values distort averages and distributions.
- Models may learn misleading patterns (e.g., negative balance = unrealistic customer).

```
# Keep only realistic ages
df_clean = df_clean[(df_clean['age'] >= 0) & (df_clean['age'] <= 120)]

# Fix negative balances by setting them to 0
df_clean.loc[df_clean['balance'] < 0, 'balance'] = 0

# Ensure transaction_count is non-negative
df_clean.loc[df_clean['transaction_count'] < 0, 'transaction_count'] = 0

print(" After cleaning invalid values:")
print(df_clean.head())
```

```
 After cleaning invalid values:
  account_id                name   age  gender           city   account_type  \
0     MY1000       Aisyah Bin Ali  56.0    Male   Kuala Lumpur        Savings
1     MY1001       Siti Bin Hafiz  69.0    Male  Kota Kinabalu  Fixed Deposit
2     MY1002   Aisyah Bin Fatimah  45.0  Female        Kuching        Savings
3     MY1003      Farid Bin Ahmad  32.0    Male         Penang        Savings
4     MY1004      Aisyah Bin Siti  60.0    Male        Kuching        Current

    balance  loan_amount  transaction_count last_transaction
0  47747.78     53970.86                 55       2023-12-14
1  36921.06         0.00                100       2023-04-25
2  27762.27     34961.43                173       2023-04-14
3  30624.87     35021.84                136       2023-04-28
4  21038.04     73881.37                119       2023-01-05
```

Removing Duplicates

- `drop_duplicates()` removes duplicated rows.

Why it matters?

- Duplicates bias statistical analysis (e.g., balance counted twice).
- For machine learning, duplicates cause models to overfit and learn repeated patterns.

```
df_clean = df_clean.drop_duplicates()
print(" After removing duplicates:")
print(df_clean.head())
```

```
After removing duplicates:
  account_id                name   age  gender           city   account_type  \
0     MY1000       Aisyah Bin Ali  56.0    Male   Kuala Lumpur        Savings
1     MY1001       Siti Bin Hafiz  69.0    Male  Kota Kinabalu  Fixed Deposit
2     MY1002   Aisyah Bin Fatimah  45.0  Female        Kuching        Savings
3     MY1003      Farid Bin Ahmad  32.0    Male         Penang        Savings
4     MY1004      Aisyah Bin Siti  60.0    Male        Kuching        Current

    balance  loan_amount  transaction_count last_transaction
0  47747.78     53970.86                 55       2023-12-14
1  36921.06         0.00                100       2023-04-25
2  27762.27     34961.43                173       2023-04-14
3  30624.87     35021.84                136       2023-04-28
4  21038.04     73881.37                119       2023-01-05
```

Summary Table of Cleaning Steps

Step	Function(s)	Effect on Data	Importance for Models
Empty cells	`dropna()`, `fillna()`	Handle missing values	Prevents errors & bias
Wrong format	`to_datetime()`	Convert text → dates	Enables time analysis
Wrong/unrealistic data	`loc[]` with conditions	Fix invalid values	Prevents misleading patterns
Duplicates	`drop_duplicates()`	Remove repeated rows	Reduces bias & overfitting