2. DataFrame (2D Data)

A **DataFrame** is a 2-dimensional table (like a spreadsheet) where you can store data in rows and columns.

```
# Create a DataFrame from a dictionary
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "City": ["New York", "Los Angeles", "Chicago"]
}

df = pd.DataFrame(data)

print(df)
```

Output:

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

How to Load Data with Pandas

You can read data from many different formats:

- **CSV**: .csv
- **Excel**: .xls, .xlsx
- **SQL databases**: MySQL, PostgreSQL
- **JSON**: .json

```
# Read a CSV file
df = pd.read_csv('your_file.csv')

# Check the first 5 rows
print(df.head())
```

Basic Operations with Pandas

1. Inspecting the Data

- **df.head():** View the first 5 rows of the DataFrame.
- **df.tail():** View the last 5 rows.
- **df.info():** Get an overview of data types and non-null counts.
- **df.describe():** Generate summary statistics for numeric columns.

```
print(df.head())      # First 5 rows
print(df.describe())  # Summary statistics
```

2. Selecting Data

- Select columns using df["ColumnName"] or df.ColumnName.
- Select rows using df.iloc[] or df.loc[].

```
# Select a column
ages = df["Age"]

# Select rows by position (first 3 rows)
first_three = df.iloc[:3]

# Select rows by label (where Age > 30)
older_than_30 = df[df["Age"] > 30]
```

Data Cleaning in Pandas

1. Handling Missing Data

- **df.isnull():** Check for missing values.
- **df.dropna():** Drop rows with missing values.
- **df.fillna():** Fill missing values with a specified value.

```
# Fill missing values with 0
df.fillna(0, inplace=True)
```

2. Renaming Columns

```
df.rename(columns={"OldName": "NewName"}, inplace=True)
```

Loading Data from CSV, Excel, JSON, APIs

One of the most common tasks in data analytics is loading data from various file formats and sources. Pandas makes this incredibly simple. Let's dive into how you can load data from **CSV**, **Excel**, **JSON**, and **APIs**.

1. Loading Data from CSV Files

CSV (Comma-Separated Values) files are one of the most common ways to store tabular data. You can easily read CSV files with the **`pd.read_csv()`** function.

```
import pandas as pd

# Load CSV file
df = pd.read_csv('your_file.csv')

# Check the first 5 rows of the DataFrame
print(df.head())
```

Additional Parameters:

- **sep:** Customize the delimiter (e.g., sep=";" for semicolon-separated files).
- **header:** Set the row to use as column names (default is the first row).
- **index_col**: Specify the column to use as the index.

```
# Load CSV with custom delimiter and set column names
df = pd.read_csv('your_file.csv', sep=";", header=0, index_col=0)
```

2. Loading Data from Excel Files

You can use the `pd.read_excel()` function to read data from `.xlsx` or `.xls` files. Make sure you have the `openpyxl` or `xlrd` package installed for `.xlsx` files.

```
pip install openpyxl
```

```
# Load Excel file (make sure to install openpyxl for .xlsx files)
df = pd.read_excel('your_file.xlsx', sheet_name='Sheet1')

# Check the first 5 rows
print(df.head())
```

Additional Parameters:

- **`sheet_name`:** Specify the sheet name or index (can be a string or an integer).
- **`header`**: Row to use for column headers.
- **`usecols`:** Specify which columns to load (use column letters like `'A:C'`).

```
# Load specific columns from Excel file
df = pd.read_excel('your_file.xlsx', usecols="A:C")
```

3. Loading Data from JSON Files

JSON (JavaScript Object Notation) files are used for storing and transmitting data structures. You can use `pd.read_json()` to load data from a `.json` file into a DataFrame.

```
# Load JSON file
df = pd.read_json('your_file.json')

# Check the first 5 rows
print(df.head())
```

Additional Parameters:

- **orient:** Specify the format of the JSON string (default is split).
- **encoding:** Set the encoding of the JSON file (e.g., utf-8).

4. Loading Data from an API (JSON format)

Many APIs return data in JSON format, and Pandas can read this directly by using Python's requests library combined with **pd.json_normalize().**

```
pip install requests
```

Here's an example using the **JSONPlaceholder API** (a free fake online REST API for testing):

```
import requests
import pandas as pd

# API URL (JSON data)
url = 'https://jsonplaceholder.typicode.com/posts'

# Fetch the data from the API
response = requests.get(url)

# Load the JSON data into a DataFrame
data = response.json()
df = pd.json_normalize(data)

# Check the first 5 rows
print(df.head())
```

Steps:

- Send a GET request to the API using **requests.get().**
- Parse the JSON response with **.json()** method.
- Normalize and load the data into a Pandas DataFrame using pd**.json_normalize().**

5. Handling Large Files with Chunking

When working with large files (CSV or Excel), it might be necessary to read the data in chunks rather than loading the entire file at once.

```
# Read CSV file in chunks
chunk_size = 1000
chunks = pd.read_csv('your_large_file.csv', chunksize=chunk_size)

# Process each chunk
for chunk in chunks:
    print(chunk.head())
    # Perform further operations on each chunk
```

DataFrames and Series – The Core Structures

In **Pandas**, **DataFrames** and **Series** are the two main data structures used for storing and manipulating data. Understanding how to work with these structures is fundamental to performing data analysis.

What is a Series?

A **Series** is a one-dimensional array-like object that holds a sequence of values along with an associated **index**. Think of a **Series** as a single column in a table or a list of data labeled with indices.

Key Features of a Series:

- It can hold any data type: integers, floats, strings, etc.
- It has an **index** (labels) for each element, similar to row names in Excel.

Creating a Series

You can create a Series from a list, a dictionary, or even a NumPy array.

```
import pandas as pd

# Create a Series from a list
data = [10, 20, 30, 40]
series = pd.Series(data)

print(series)
```

Output:

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

In this example, the numbers `10, 20, 30, 40` are stored in a **Series**, with an **index** automatically created as `0, 1, 2, 3`.

Custom Index in a Series

You can also provide a custom index to a Series.

```
# Create a Series with a custom index
data = [10, 20, 30, 40]
index = ['a', 'b', 'c', 'd']
series = pd.Series(data, index=index)

print(series)
```

Output:

```
a    10
b    20
c    30
d    40
dtype: int64
```

What is a DataFrame?

A **DataFrame** is a two-dimensional table (like a spreadsheet or SQL table) that holds data in a **tabular form** with rows and columns. A DataFrame is essentially a collection of Series sharing the same index.

Key Features of a DataFrame:

- It can store data of different types (numbers, strings, booleans) in each column.
- It has both **rows** (axis 0) and **columns** (axis 1).
- Each column is a **Series**, and the DataFrame can be thought of as a container for these Series.

Creating a DataFrame

You can create a DataFrame from various data sources like dictionaries, lists, or even existing CSV files.

```
# Create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

print(df)
```

Output:

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

Working with Series and DataFrames

Accessing Elements in a Series

- **By position**: **Use .iloc[]** for integer-location based indexing.
- **By label**: **Use .loc[]** for label-based indexing.

```
# Access element by position
print(series[0])  # Output: 10

# Access element by label
print(series['a'])  # Output: 10
```

Accessing Columns in a DataFrame

You can access a column in a DataFrame by using the column name (which is a Series).

```
# Access the 'Age' column
ages = df['Age']
print(ages)
```

Output:

```
0    25
1    30
2    35
Name: Age, dtype: int64
```

Accessing Rows in a DataFrame

You can access rows by **position** using **.iloc[]** or by **index label** using **.loc[]**.

```
# Access the first row
first_row = df.iloc[0]
print(first_row)

# Access the row with label 1 (second row)
second_row = df.loc[1]
print(second_row)
```

Modifying DataFrames and Series

Adding a New Column to a DataFrame

You can easily add a new column to a DataFrame by assigning a Series to a new column name.

```
# Add a new column 'Salary'
df['Salary'] = [50000, 60000, 70000]
print(df)
```

Output:

```
      Name  Age         City  Salary
0    Alice   25     New York   50000
1      Bob   30  Los Angeles   60000
2  Charlie   35      Chicago   70000
```

Modifying Values in a DataFrame

You can update the values in a DataFrame or Series by using indexing.

```
# Change the age of Alice (index 0)
df.loc[0, 'Age'] = 26
print(df)
```

Output:

```
      Name  Age         City  Salary
0    Alice   26     New York   50000
1      Bob   30  Los Angeles   60000
2  Charlie   35      Chicago   70000
```

Useful Operations with DataFrames and Series

1. Filtering DataFrames

You can filter rows based on conditions.

```
# Get rows where Age is greater than 30
older_than_30 = df[df['Age'] > 30]
print(older_than_30)
```

Output:

```
      Name  Age     City  Salary
2  Charlie   35  Chicago   70000
```

2. Sorting DataFrames

You can sort a DataFrame by column values.

```
# Sort by 'Age'
sorted_df = df.sort_values(by='Age')
print(sorted_df)
```

Output:

```
      Name  Age         City  Salary
0    Alice   26     New York   50000
1      Bob   30  Los Angeles   60000
2  Charlie   35      Chicago   70000
```

Indexing, Slicing & Subsetting Data

In **Pandas**, **Indexing** and **Slicing** are crucial techniques for accessing and modifying specific elements, rows, and columns in **Series** and **DataFrames**. Understanding how to perform these operations will allow you to efficiently work with large datasets.

1. Indexing a Pandas Series

A **Series** is a one-dimensional array-like object, and you can access individual elements using **indexing**.

By Position:

- **`.iloc[]`:** Integer-location based indexing. It allows you to access elements by their integer positions (like an array).

```
import pandas as pd

# Create a Series
data = [10, 20, 30, 40]
index = ['a', 'b', 'c', 'd']
series = pd.Series(data, index=index)

# Access the first element by position
print(series.iloc[0])  # Output: 10

# Access the second element by position
print(series.iloc[1])  # Output: 20
```

By Label:

- **`.loc[]`**: Label-based indexing. It allows you to access elements using the **index labels** rather than positions.

```
# Access the element by label
print(series.loc['a'])  # Output: 10
print(series.loc['b'])  # Output: 20
```

2. Slicing a Pandas Series

- You can slice a **Series** just like a list in Python. The syntax follows the **[start:stop:step]** format.

```
# Slice the Series (from index 'a' to 'c')
print(series['a':'c'])
```

Output:

```
a     10
b     20
c     30
dtype: int64
```

- **Start**: The starting index (inclusive).
- **Stop**: The ending index (inclusive).
- **Step**: The interval for selecting values (optional).

3. Indexing a Pandas DataFrame

A **DataFrame** is a two-dimensional table with rows and columns, and you can access both individual rows and columns in various ways.

Accessing Columns: Columns in a **DataFrame** can be accessed as a **Series**. You can access them using **dot notation** or **column indexing**.

```
# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

# Access a column by name (using dot notation or column indexing)
print(df['Name'])  # Output: Series with Name column values
print(df.Name)  # Output: Series with Name column values
```

Accessing Rows by Position:

iloc[]: You can use **.iloc[]** to select rows by their integer position (like row numbers).

```
# Access the first row by position
print(df.iloc[0])  # Output: First row in the DataFrame

# Access the second row by position
print(df.iloc[1])  # Output: Second row in the DataFrame
```

Accessing Rows by Label:

- **.loc[]:** You can use **.loc[]** to select rows by their **index labels**.

```
# Access the row with index 0 (first row)
print(df.loc[0])

# Access the row with index 1 (second row)
print(df.loc[1])
```

4. Slicing a Pandas DataFrame

Slicing works in a similar way to slicing in a **Series**, but you can slice by both rows and columns in a **DataFrame**.

Slicing Rows: You can slice rows of a **DataFrame** using `.iloc[]` or `.loc[]`.

```
# Slice the first two rows using .iloc[]
print(df.iloc[0:2])  # Rows at position 0 and 1

# Slice rows using .loc[] (using index labels)
print(df.loc[0:1])  # Rows with index labels 0 and 1
```

Slicing Columns: You can select a range of columns by their labels or positions.

```
# Slice columns by their positions (from index 0 to 1)
print(df.iloc[:, 0:2])  # Select the first two columns

# Slice columns by their labels (from 'Name' to 'Age')
print(df.loc[:, 'Name':'Age'])  # Select columns 'Name' and 'Age'
```

5. Subsetting Data with Conditions

You can subset **DataFrames** and **Series** based on conditions, which is useful for filtering data.

Subset a DataFrame Based on Condition: You can filter rows of a **DataFrame** by using a condition on one or more columns.

```
# Filter rows where Age > 30
print(df[df['Age'] > 30])
```

Output:

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

Subset a DataFrame with Multiple Conditions: You can combine conditions using **logical operators** such as `&` (AND), `|` (OR), and `~` (NOT).

```
# Filter rows where Age > 25 and City is 'New York'
print(df[(df['Age'] > 25) & (df['City'] == 'New York')])
```

6. Using .at[] and .iat[] for Fast Access

If you need to access a **single element** by **label** (using `.at[]`) or by **position** (using `.iat[]`), these methods are more efficient than using `.loc[]` or `.iloc[]`.

.at[]: Access a single value by **row label** and **column label**.

```
# Access a value by label (row=1, column='Age')
print(df.at[1, 'Age'])  # Output: 30
```

.iat[]: Access a single value by **row position** and **column position**.

```
# Access a value by position (row=1, column=1)
print(df.iat[1, 1])  # Output: 30
```

Data Cleaning Techniques

Data cleaning is a crucial step in data analysis. It involves addressing missing values, duplicates, outliers, and ensuring the correct data types. In this section, we'll explore common **data cleaning techniques** using **Pandas** that are essential for making your data suitable for analysis and visualization.

1. Handling Missing Values

Missing data is a common problem in real-world datasets. In Pandas, we can handle missing values using various methods:

Identifying Missing Data

- **isna():** Returns a DataFrame/Series of the same shape with **True** for missing values and **False** for non-missing values.
- **isnull():** Equivalent to isna().

```
import pandas as pd

# Sample DataFrame with missing values
data = {'Name': ['Alice', 'Bob', 'Charlie', None],
        'Age': [25, 30, None, 35]}
df = pd.DataFrame(data)

# Identifying missing data
print(df.isna())
```

Output:

```
    Name    Age
0  False  False
1  False  False
2  False   True
3   True  False
```

Removing Missing Data

- **dropna():** Removes rows or columns with missing values.

```
# Drop rows with any missing value
df_cleaned = df.dropna()
print(df_cleaned)
```

Output:

```
    Name   Age
0  Alice  25.0
1    Bob  30.0
```

- **dropna(axis=1)**: Removes columns with any missing values.

```
# Drop columns with any missing value
df_cleaned = df.dropna(axis=1)
print(df_cleaned)
```

Output:

```
     Name
0   Alice
1     Bob
2 Charlie
3    None
```

Filling Missing Data

- **fillna()**: Fills missing values with a specified value or method (e.g., forward fill, backward fill).

```
# Fill missing values with a constant value
df_filled = df.fillna({'Name': 'Unknown', 'Age': 0})
print(df_filled)
```

Output:

```
      Name   Age
0    Alice  25.0
1      Bob  30.0
2  Charlie   0.0
3  Unknown  35.0
```

- **Forward Fill (method='ffill')**: Propagates the last valid value forward.

```
# Forward fill missing values
df_filled = df.fillna(method='ffill')
print(df_filled)
```

Output:

```
      Name   Age
0    Alice  25.0
1      Bob  30.0
2  Charlie  30.0
3  Charlie  35.0
```

2. Handling Duplicates

Duplicate data is another issue in datasets, especially when collecting data from multiple sources.

Identifying Duplicates

- **duplicated()**: Returns a **Boolean** Series indicating whether a row is a duplicate of a previous row.

```
# Identify duplicates
print(df.duplicated())
```

Output:

```
0     False
1     False
2     False
3      True
dtype: bool
```

Removing Duplicates

- **drop_duplicates()**: Removes duplicate rows based on all columns (or specific columns).

```
# Remove duplicates
df_no_duplicates = df.drop_duplicates()
print(df_no_duplicates)
```

Output:

```
      Name   Age
0    Alice  25.0
1      Bob  30.0
2  Charlie  35.0
```

- **Keep Specific Duplicates**: You can also choose to keep the first or last occurrence of a duplicate using the `keep` parameter.

```
# Keep the last occurrence of the duplicate
df_no_duplicates = df.drop_duplicates(keep='last')
print(df_no_duplicates)
```

Output:

```
      Name   Age
0    Alice  25.0
1      Bob  30.0
3  Charlie  35.0
```

3. Handling Outliers

Outliers can distort statistical analyses, so it's important to identify and handle them. Outliers are values significantly higher or lower than the other values in the dataset.

Identifying Outliers Using Z-Score

The **Z-score** is a measure of how many standard deviations a data point is from the mean. A Z-score above 3 or below -3 is often considered an outlier.

```
import numpy as np

# Calculate the Z-scores
z_scores = (df['Age'] - df['Age'].mean()) / df['Age'].std()
print(z_scores)
```

Identifying Outliers Using IQR (Interquartile Range)
Outliers can also be detected using the **Interquartile Range (IQR)**, which is the difference between the 75th percentile (Q3) and the 25th percentile (Q1).

```
# Calculate IQR
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1

# Define outliers
outliers = (df['Age'] < (Q1 - 1.5 * IQR)) | (df['Age'] > (Q3 + 1.5 *
IQR))
print(outliers)
```

Output:

```
0    False
1    False
2     True
3    False
dtype: bool
```

Removing Outliers
To remove outliers using the IQR method, you can filter out rows where the values fall outside the acceptable range.

```
# Filter out outliers
df_no_outliers = df[~outliers]
print(df_no_outliers)
```

Output:

```
      Name   Age
0    Alice  25.0
1      Bob  30.0
3  Charlie  35.0
```

4. Type Conversions

Data often comes in incorrect types (e.g., numeric values stored as strings). Converting data types ensures that the columns are correctly interpreted for analysis.

Converting Data Types

You can convert columns to the correct data type using the **`astype()`** method.

```
# Convert the 'Age' column to integer
df['Age'] = df['Age'].astype(int)
print(df)
```

Output:

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

Handling Mixed Data Types

Sometimes, a column may contain both numeric and string values. You can use **`pd.to_numeric()`** to convert non-numeric values to `NaN`.

```
# Convert 'Age' column to numeric (invalid parsing values are set to
NaN)
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')
print(df)
```

Data Transformation

Data transformation is an essential step in data analysis. It involves reshaping, aggregating, and modifying data to prepare it for deeper analysis. In this section, we will cover key techniques for transforming data using **Pandas**: **Filtering, Sorting, Grouping**, along with functional transformations using **Lambda, Map, and Apply**, and data merging operations such as **Merging, Joining, and Concatenating**.

1. Filtering, Sorting, and Grouping Data

Filtering Data: You can filter data based on specific conditions using boolean indexing or `.loc[]`.

Example: Filtering rows where the "Age" column is greater than 30.

```
import pandas as pd


data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'City': ['New York', 'Chicago', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data)


# Filter rows where Age > 30
filtered_df = df[df['Age'] > 30]
print(filtered_df)
```

Output:

```
      Name  Age           City
2  Charlie   35  San Francisco
3    David   40    Los Angeles
```

Sorting Data

You can sort a DataFrame using the **`sort_values()`** method.

Sort by one or more columns (e.g., sorting by "Age").

```
# Sort by Age in ascending order
sorted_df = df.sort_values(by='Age')
print(sorted_df)
```

Output:

```
      Name  Age           City
0    Alice   25       New York
1      Bob   30        Chicago
2  Charlie   35  San Francisco
3    David   40    Los Angeles
```

Descending order (use `ascending=False`).

```
# Sort by Age in descending order
sorted_df = df.sort_values(by='Age', ascending=False)
print(sorted_df)
```

Output:

```
      Name  Age           City
3    David   40    Los Angeles
2  Charlie   35  San Francisco
1      Bob   30        Chicago
0    Alice   25       New York
```

Grouping Data

The **groupby()** method allows you to group data based on one or more columns and then apply an aggregation function (e.g., sum, mean).

Example: Grouping data by "City" and calculating the average age.

```
# Group by City and calculate mean age
grouped_df = df.groupby('City')['Age'].mean().reset_index()
print(grouped_df)
```

Output:

```
            City   Age
0        Chicago  30.0
1    Los Angeles  40.0
2  San Francisco  35.0
3       New York  25.0
```

2. Lambda, Map, and Apply Functions

Lambda Functions

A **lambda function** is a small anonymous function defined using the `lambda` keyword. It's often used for short transformations that don't require defining a full function.

Example: Adding 5 years to each value in the "Age" column using `lambda`.

```
# Add 5 to each value in the 'Age' column using lambda
df['Age'] = df['Age'].apply(lambda x: x + 5)
print(df)
```

Output:

```
      Name  Age           City
0    Alice   30       New York
1      Bob   35        Chicago
2  Charlie   40  San Francisco
3    David   45    Los Angeles
```

Map Function

map() is used to map values in a Series based on a dictionary, series, or function.

Example: Mapping each city to its abbreviation using a dictionary.

```
# Define a dictionary for mapping
city_abbr = {'New York': 'NY', 'Chicago': 'CHI', 'San Francisco':
'SF', 'Los Angeles': 'LA'}

# Map city names to abbreviations
df['City'] = df['City'].map(city_abbr)
print(df)
```

Output:

```
      Name  Age City
0    Alice   30   NY
1      Bob   35  CHI
2  Charlie   40   SF
3    David   45   LA
```

Apply Function

The **apply()** function allows applying a function along the axis (rows or columns) of a DataFrame or Series.

Example: Calculating the length of each name using `apply()`.

```
# Apply function to calculate name length
df['Name Length'] = df['Name'].apply(len)
print(df)
```

Output:

```
      Name  Age City  Name Length
0    Alice   30   NY            5
1      Bob   35  CHI            3
2  Charlie   40   SF            7
3    David   45   LA            5
```