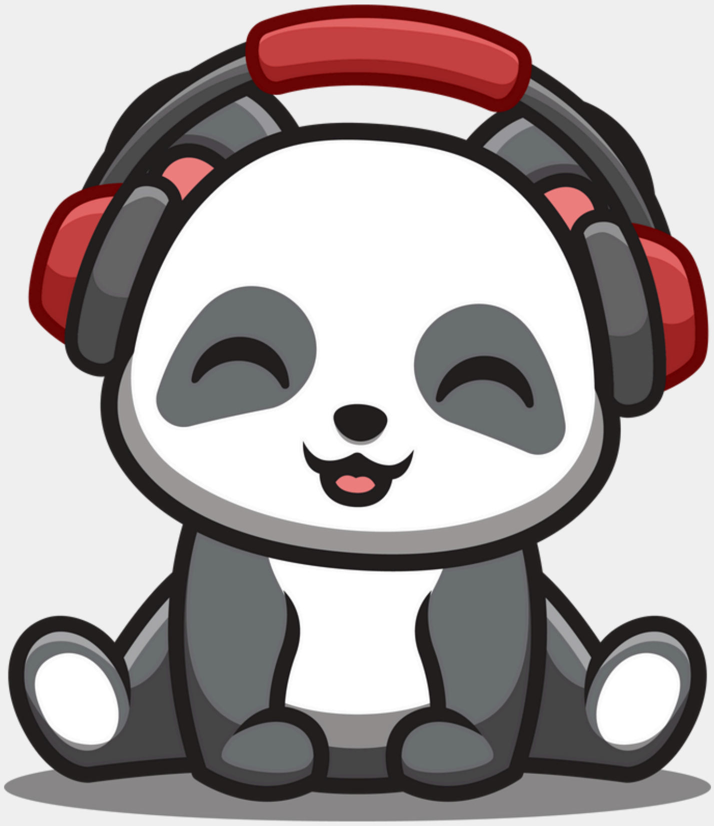
Python | Machine Learning | AI 🚀
@pycode.hubb



PANDAS FOR BEGINNERS EASY AND DETAILED GUIDE

by Pycode.Hubb

python™

Chapter 1

INTRODUCTION TO PANDAS

1. What is Pandas?
2. Use of Pandas
3. Installation and Setup
4. Overview of Pandas' Core Data Structures

What is Pandas?

Pandas is a powerful and flexible open-source data analysis and manipulation library for Python.

It is widely used in data science and analytics to work with structured data.

It provides easy-to-use data structures and data analysis tools for handling numerical tables and time series data.

Use of Pandas

Pandas is commonly used for:

- **Data Cleaning:** Handling missing data, removing duplicates, and transforming data formats.
- **Data Transformation:** Aggregating, merging, and reshaping datasets.
- **Exploratory Data Analysis (EDA):** Summarizing data and generating descriptive statistics.
- **Data Visualization:** Creating plots and graphs (with integration to libraries like Matplotlib).
- **Time Series Analysis:** Handling and analyzing time series data effectively.

Installation and Setup

To get started with Pandas, you need to install it.

If you are using Python's package manager, pip, you can install Pandas by running:

```
pip install pandas
```

Alternatively, if you are using Anaconda, Pandas comes pre-installed, or you can install it using:

```
conda install pandas
```

Pandas' Core Data Structures

Pandas offers two main data structures that are commonly used in data analysis: **Series (1D)** and **DataFrame (2D)**.

Both structures are integral to data manipulation and analysis in Pandas.

Series:

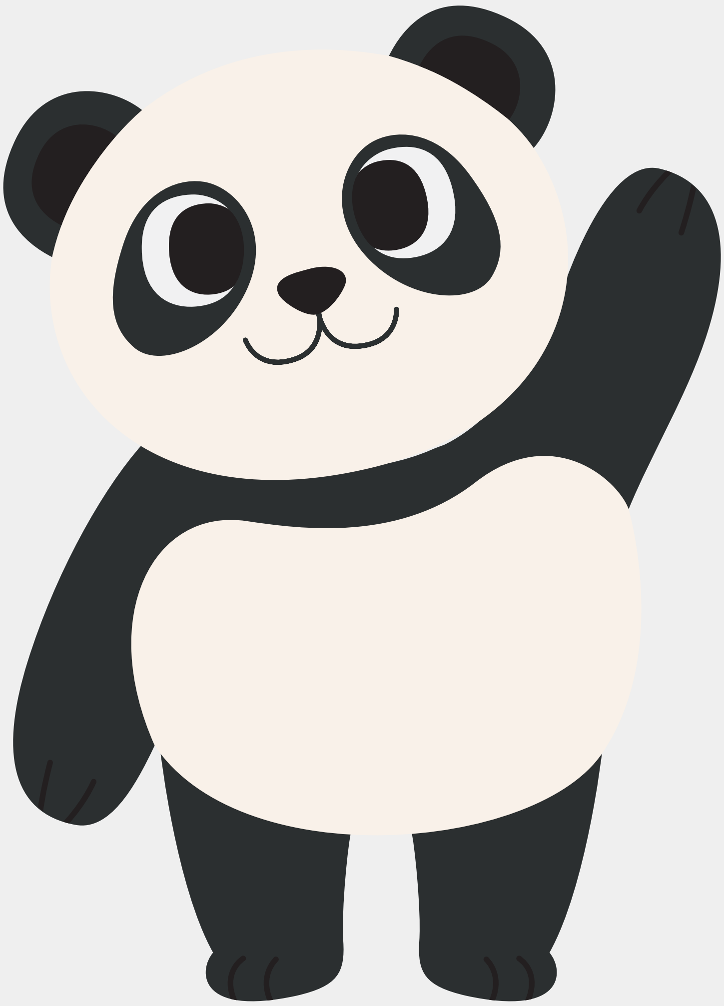
- A one-dimensional array-like object that can hold any data type (integers, strings, floats, etc.).
- It is similar to a column in a spreadsheet or a database table.
- Each element in a Series is associated with an index label.

```
import pandas as pd

s = pd.Series([1, 2, 3, 4], index=['a', 'b',
'c', 'd'])
print(s)
```

DataFrame:

- A two-dimensional table with labeled axes (rows and columns).
- It is akin to a spreadsheet or SQL table.
- Each column in a DataFrame can hold data of different types (integers, strings, floats, etc.).

```python
import pandas as pd

df = pd.DataFrame({
     'A': [1, 2, 3],
     'B': [4, 5, 6]
}, index=['row1', 'row2', 'row3'])
print(df)
```



Chapter 2

SERIES: THE 1D DATA STRUCTURE

Creating a Series

A Series in Pandas is a one-dimensional array-like structure that can hold data of any type—integers, strings, floats, etc.

Each element in a Series has a unique label called an index.

You can create a Series using different types of data, such as lists, dictionaries, or even scalar values.

Example 1: Creating a Series from a List

```
import pandas as pd

data = [10, 20, 30, 40]
s = pd.Series(data)
print(s)
```

This creates a Series with default integer index labels (0, 1, 2, 3).

Example 2: Creating a Series with Custom Index

```
import pandas as pd

data = [10, 20, 30, 40]
s = pd.Series(data, index=['a', 'b', 'c',
'd'])
print(s)
```

Here, the Series is created with custom index labels ('a', 'b', 'c', 'd').

Example 3: Creating a Series from a Dictionary

```
import pandas as pd

data = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
s = pd.Series(data)
print(s)
```

Accessing Data in a Series

You can access the data in a Series using the index labels or position-based indexing (similar to accessing elements in a list or dictionary).

Example 1: Accessing Data Using Index Labels

```
import pandas as pd

s = pd.Series([10, 20, 30, 40], index=['a',
'b', 'c', 'd'])
print(s['b'])
```

This will print the value associated with index 'b', which is 20.

Example 2: Accessing Data Using Position-Based Indexing

```
import pandas as pd

s = pd.Series([10, 20, 30, 40], index=['a',
'b', 'c', 'd'])
print(s.iloc[2])
```

This will print the value at the 2nd position in the Series, which is 30.

Example 3: Accessing Multiple Elements

```python
import pandas as pd

s = pd.Series([10, 20, 30, 40], index=['a',
'b', 'c', 'd'])
print(s[['a', 'd']])
```

This will print the values associated with indexes 'a' and 'd', which are 10 and 40 respectively.

Operations on Series

Pandas allows you to perform various operations on Series, such as arithmetic operations, aggregation, and element-wise operations.

***Example 1:* Arithmetic Operations** You can add, subtract, multiply, or divide Series by a scalar or another Series.

```python
import pandas as pd

s1 = pd.Series([10, 20, 30, 40])
s2 = pd.Series([1, 2, 3, 4])

# Adding two Series
result = s1 + s2
print(result)
```

This will add corresponding elements in s1 and s2, resulting in a new Series.

***Example 2:* Aggregation Operations** You can use functions like sum(), mean(), min(), and max() to perform aggregation on Series.

```python
import pandas as pd

s = pd.Series([10, 20, 30, 40])

# Sum of all elements
print(s.sum())

# Mean of all elements
print(s.mean())
```

This will print the sum (100) and mean (25.0) of the Series.

***Example 3:* Element-Wise Operations** You can apply functions to each element in a Series using the apply() method.

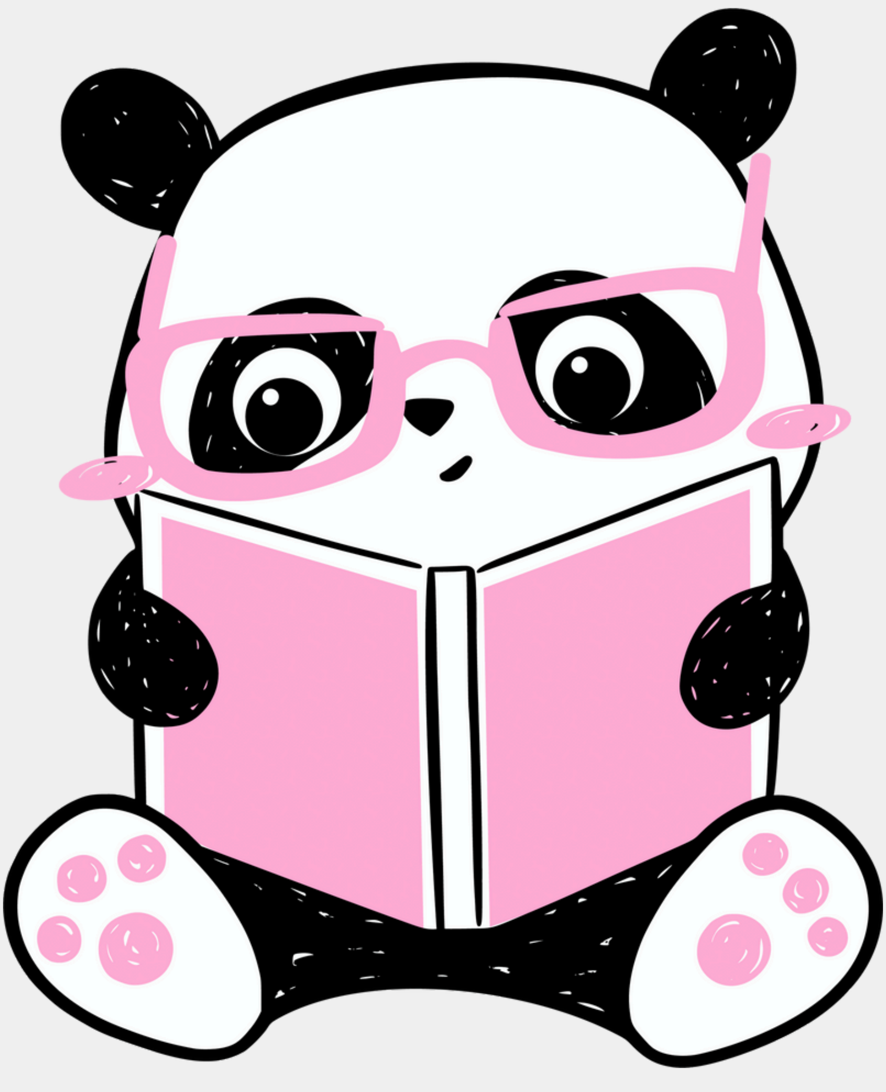
```python
import pandas as pd

s = pd.Series([1, 2, 3, 4])

# Square each element in the Series
result = s.apply(lambda x: x**2)
print(result)
```

This will return a Series with each element squared.

Series Represented By:

<class 'pandas.core.series.Series'>

Which indicates that the object is an instance of the Series class from the **pandas.core.series** module in the Pandas library.



Chapter 3

DATAFRAME: THE 2D DATA STRUCTURE

1. Creating a DataFrame
2. Accessing Data in a DataFrame
3. Adding and Removing Columns
4. DataFrame Attributes and Methods
5. Applications of DataFrame
6. Commonly used DataFrame functions

Creating a DataFrame

A DataFrame in Pandas is a two-dimensional table, similar to a spreadsheet or a SQL table.

It consists of rows and columns, where each column can hold different types of data (e.g., integers, strings, floats).

You can create a DataFrame from various data sources such as dictionaries, lists, or even external files like CSVs.

Example 1: Creating a DataFrame from a Dictionary

```
import pandas as pd

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

This creates a DataFrame where the dictionary keys become column names, and the values become the rows.

Accessing Data in a DataFrame

You can access data in a DataFrame using various methods, such as accessing specific columns, rows, or even specific elements.

Example 1: Accessing a Single Column

```python
import pandas as pd

df = pd.DataFrame({
     'Name': ['Alice', 'Bob', 'Charlie'],
     'Age': [25, 30, 35]
})

# Accessing the 'Name' column
print(df['Name'])
```

This will print the entire 'Name' column as a Series.

Example 2: Accessing Multiple Columns

```
import pandas as pd

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

# Accessing the 'Name' column
print(df['Name', 'City'])
```

This will print the 'Name' and 'City' columns as a new DataFrame.

Example 3: Accessing Rows Using .loc and .iloc

- .loc[]: Accesses rows by label/index name.
- .iloc[]: Accesses rows by position.

```python
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

# Accessing the row with label 1
print(df.loc[1])

# Accessing the first row by position
print(df.iloc[0])
```

This will print the row data for the corresponding label or position.

Example 4: Accessing Specific Elements

```python
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

# Accessing the element in the first row and
'Age' column
print(df.loc[0, 'Age'])

# Accessing the element by position
print(df.iloc[1, 0])
```

This will print specific elements from the DataFrame.

Adding and Removing Columns

Pandas makes it easy to add or remove columns in a DataFrame.

Example 1: Adding a New Column

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

# Adding a new column 'City'
df['City'] = ['New York', 'Los Angeles',
'Chicago']
print(df)
```

This will add a new column 'City' to the existing DataFrame.

Example 2: Removing a Column

```
import pandas as pd

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

# Removing the 'City' column
df = df.drop(columns=['City'])
print(df)
```

This will remove the 'City' column from the DataFrame.

DataFrame Attributes and Methods

Pandas DataFrames come with several attributes and methods that help you understand and manipulate the data.

Example 1: Common Attributes

- **.shape:** Returns the dimensions of the DataFrame (rows, columns).
- **.columns:** Returns the column labels.
- **.index:** Returns the row labels.

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

# Getting the shape of the DataFrame
print(df.shape)

# Getting the column names
print(df.columns)

# Getting the row index
print(df.index)
```

Example 2: Common Methods

- **.head(n):** Returns the first n rows (default is 5).
- **.tail(n):** Returns the last n rows (default is 5).
- **.describe():** Provides summary statistics for numerical columns.
- **.info():** Provides a concise summary of the DataFrame.

```python
import pandas as pd

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

# Displaying the first 2 rows
print(df.head(2))

# Getting summary statistics
print(df.describe())

# Getting a concise summary
df.info()
```

Applications of DataFrame

- **Work with Data Sets:** Load and view data.
- **Analyze Data:** Perform sorting, filtering, and aggregation.
- **Clean Data:** Drop or handle missing values.
- **Process Data:** Transform and manipulate data.
- **Integrate Data:** Merge or join multiple data sources.
- **Export Data:** Save to Excel, CSV, JSON, or binary formats.
- **Math & Stats:** Perform calculations and statistical operations.
- **Group Data:** Aggregate data using group by.

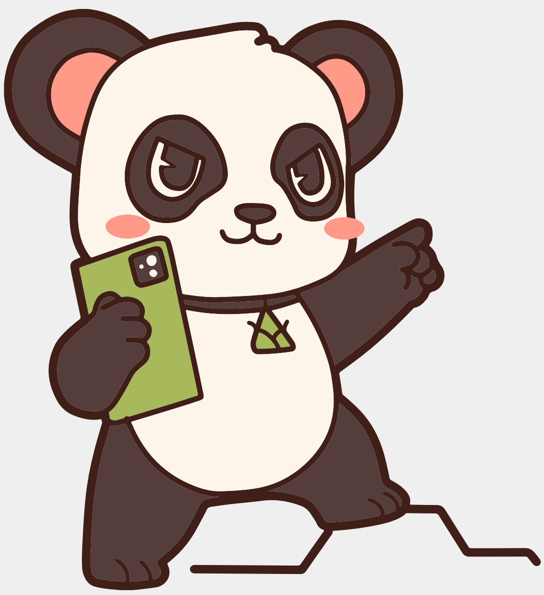
Commonly used DataFrame functions

- **Loading and Saving Data:**
  - pd.read_csv()
  - DataFrame.to_csv()
- **Viewing Data:**
  - DataFrame.head()
  - DataFrame.info()
- **Selecting Data:**
  - DataFrame.loc[]
  - DataFrame.iloc[]

- **Filtering and Sorting:**
    - DataFrame.query()
    - DataFrame.sort_values()
- **Data Cleaning:**
    - DataFrame.dropna()
    - DataFrame.fillna()
- **Data Processing:**
    - DataFrame.apply()
- **Grouping and Aggregation:**
    - DataFrame.groupby()
    - DataFrame.agg()
- **Merging and Joining:**
    - pd.merge()

Represented By:

<class 'pandas.core.frame.DataFrame'>



Chapter 4

BASIC DATA OPERATIONS

1. Indexing and Slicing
2. Filtering Data
3. Handling Missing Data
4. Sorting Data

Indexing and Slicing

Indexing and slicing help you access and manipulate parts of your data.

Indexing: Refers to accessing a specific item in a list or series using its position.

For Example:

```
numbers = [10, 20, 30]
print(numbers[0])  # Output: 10
```

Slicing: Allows you to access a range of items.

For Example:

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])  # Output: [20, 30, 40]
```

And in Pandas we can do like this:

```python
import pandas as pd

df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8]
})
print(df.loc[1])
# Output: A    2
#         B    6
#         Name: 1, dtype: int64

print(df.iloc[1, 1])  # Output: 6
```

Here, **df.loc[1]** gives the whole row at **index 1**, showing the values for columns **'A'** and **'B'**. **df.iloc[1, 1]** retrieves the value at the **2nd row** and **2nd column** directly. So, the output **'6'** is from **df.iloc[1, 1]**.

Filtering Data

Filtering helps you select specific data based on conditions.

Basic Filtering: Select rows where a condition is true.

For Example:

```
import pandas as pd

df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8]
})
print(df.loc[1])
# Output: A    2
#         B    6
#         Name: 1, dtype: int64

print(df.iloc[1, 1])  # Output: 6
```

Here, **df.loc[1]** shows the entire row at **index 1**. **df.iloc[1, 1]** gives the specific value at the **2nd row** and **2nd column**. So, **'6'** is the value from **df.iloc[1, 1]**.

Multiple Conditions: Combine conditions using **& (and)** or | **(or)**.

For Example:

```
print(df[(df['Age'] > 25) & (df['Age'] < 35)])

# Output:
# Name   Age
# 1 Bob    30
```

Here, The code filters the DataFrame to show rows where **'Age'** is between **25** and **35**.
The output shows only the row for **'Bob'** who is **30** years old.

Handling Missing Data

Handling Missing Data involves dealing with data entries that are missing or incomplete.

Identifying Missing Data: Check for missing values.

For Example:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'Name': ['Alice', 'Bob', np.nan],
    'Age': [25, np.nan, 35]
})
print(df.isnull())  # Output:
                    #     Name     Age
                    # 0  False   False
                    # 1  False    True
                    # 2   True   False
```

Here, **df.isnull()** checks for missing values in the DataFrame.

The output shows **True** where there are **NaN values** and **False** where data is **present**.

Filling Missing Data: Replace missing values with a specific value.

For Example:

```
print(df.fillna({'Name': 'Unknown', 'Age':
0}))

# Output:
#        Name   Age
# 0     Alice  25.0
# 1       Bob   0.0
# 2   Unknown  35.0
```

Here, **df.fillna()** replaces missing values with specified values.

In this case, it fills missing **'Name'** with **'Unknown'** and missing **'Age'** with **0**.

Dropping Missing Data: Remove rows with missing values.

For Example:

```
print(df.dropna())

# Output:
#        Name   Age
# 0     Alice  25.0
# 2   Unknown  35.0
```

Here, **df.dropna()** removes rows with any missing values.

The output shows **only the rows** with **complete data**.

Sorting Data

Sorting Data helps you arrange data in a specific order.

Sorting by Column: Sort a DataFrame by one column.

For Example:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

print(df.sort_values(by='Age'))

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

Here, **df.sort_values(by='Age')** sorts the DataFrame by the '**Age**' column.

The output shows the rows in ascending order of age.

Sorting in Descending Order: Use ascending=False.

For Example:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

print(df.sort_values(by='Age',
ascending=False))

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

Here, **df.sort_values(by='Age', ascending=False)** sorts the DataFrame by **'Age'** in descending order.

The output shows rows with the highest age first.

Sorting by Multiple Columns: Sort by more than one column.

For Example:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie',
'Alice'],
    'Age': [25, 30, 30, 20]
})

print(df.sort_values(by=['Age', 'Name']))

# Output:
#       Name   Age
# 3    Alice   20
# 0    Alice   25
# 1    Bob     30
# 2 Charlie   30
```

Here, **df.sort_values(by=['Age', 'Name'])** sorts the DataFrame first by **'Age'**, then by **'Name'**.

The output shows rows sorted by age, and for the same age, sorted by name.

Chapter 5

DATA MANIPULATION

1. Adding and Modifying Rows/Columns
2. Merging and Joining DataFrames
3. Concatenating DataFrames
4. Grouping and Aggregating Data

Adding and Modifying Rows/Columns

Adding and modifying rows or columns lets you update your DataFrame with new data or adjust existing data.

Adding Columns: You can add a new column to a DataFrame.

For Example:

```
import pandas as pd

df = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
})
df['City'] = ['New York', 'Los Angeles']
print(df)

# Output:
#      Name   Age    City
# 0   Alice   25     New York
# 1   Bob     30     Los Angeles
```

Here, The code imports pandas, creates a DataFrame with '**Name**' and '**Age**', then adds a '**City**' column. It prints the table with **Alice** and **Bob's** info, including their **age** and **city**.

Adding Rows: You can add a new row to a DataFrame.

For Example:

```
new_row = pd.DataFrame({'Name': ['Charlie'],
'Age': [35], 'City': ['Chicago']})

df = pd.concat([df, new_row],
ignore_index=True)

print(df)

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

Here, The code creates a new DataFrame for **Charlie**, then **concatenates** it with the **existing DataFrame**. The **updated table** now **includes Alice, Bob, and Charlie's information**, showing their name, age, and city.

Modifying Columns: You can update an existing column's values.

For Example:

```
df['Age'] = df['Age'] + 1
print(df)

# Output:
#       Name     Age   City
# 0     Alice    26    New York
# 1     Bob      31    Los Angeles
# 2     Charlie 36    Chicago
```

Here, The code updates the **'Age'** column by **adding 1** to each value. The new table shows **Alice, Bob, and Charlie** with their updated ages.

Merging and Joining DataFrames

Merging and **joining** combine data from multiple DataFrames based on common columns or indices.

Merging DataFrames: Combine DataFrames based on a common column.

For Example:

```
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'ID': [1, 2, 4],
    'Age': [25, 30, 40]
})

merged_df = pd.merge(df1, df2, on='ID',
how='inner')
print(merged_df)

# Output:
#     ID    Name   Age
# 0    1    Alice 25
# 1    2    Bob    30
```

Types of Joins:

- **inner:** Only includes rows with matching keys in both DataFrames.
- **left:** Includes all rows from the left DataFrame and matching rows from the right DataFrame.
- **right:** Includes all rows from the right DataFrame and matching rows from the left DataFrame.
- **outer:** Includes all rows from both DataFrames, with NaNs for missing matches.

Concatenating DataFrames

Concatenating combines DataFrames either vertically (adding rows) or horizontally (adding columns).

Concatenating Vertically: Stack DataFrames on top of each other.

For Example:

```
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
})

df2 = pd.DataFrame({
    'Name': ['Charlie', 'David'],
    'Age': [35, 40]
})

concatenated_df = pd.concat([df1, df2],
ignore_index=True)

print(concatenated_df)

# Output:
#        Name  Age
# 0     Alice   25
# 1       Bob   30
# 2   Charlie   35
# 3     David   40
```

Concatenating Horizontally: Combine DataFrames side by side.

For Example:

```python
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
})

df2 = pd.DataFrame({
    'City': ['New York', 'Los Angeles']
})

concatenated_df = pd.concat([df1, df2],
axis=1)

print(concatenated_df)

# Output:
#    Name   Age  City
# 0  Alice  25   New York
# 1  Bob    30   Los Angeles
```

Grouping and Aggregating Data

Grouping and **aggregating** allow you to summarize and analyze data by dividing it into groups based on a specific column and then performing calculations.

Grouping Data: Group rows that have the same values in specified columns.

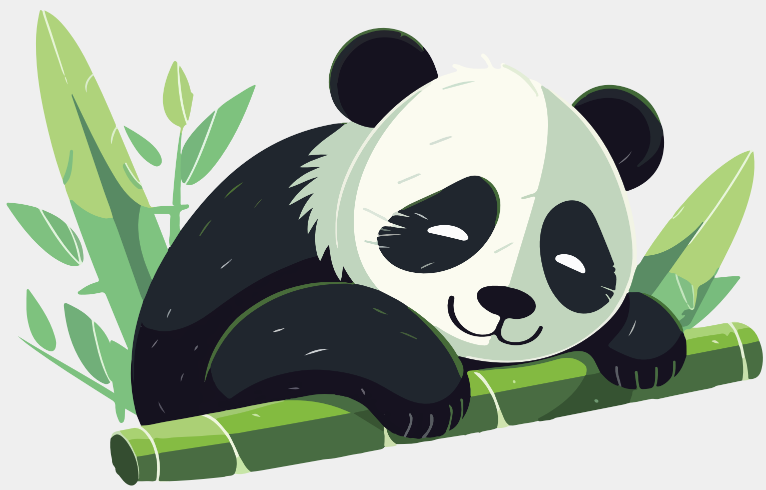
For Example:

```
df = pd.DataFrame({
'Department': ['HR', 'Finance', 'HR',
'Finance', 'IT'],
'Employee': ['Alice', 'Bob', 'Charlie',
'David', 'Eve'],
'Salary': [50000, 60000, 55000, 62000, 70000]
})

grouped_df = df.groupby('Department').mean()
print(grouped_df)

# Output:
#              Salary
# Department
# Finance      61000.0
# HR           52500.0
# IT           70000.0
```

Aggregating Data: Perform calculations on each group.

For Example:

```
grouped_df =
df.groupby('Department').agg({'Salary':
['mean', 'sum']})

print(grouped_df)

# Output:
#               Salary
#                 mean     sum
# Department
# Finance     61000.0  122000
# HR          52500.0  105000
# IT          70000.0   70000
```



Chapter 6

DATA CLEANING

Identifying and Handling Missing Data

Missing data is a common issue in datasets. Missing values can cause errors or inaccurate results during analysis, so it's important to identify and handle them.

How to Identify Missing Data: You can identify missing data using the **.isnull()** method, which returns **True** for missing values, or **.info()** to see an overview.

For Example:

```
import pandas as pd

# Sample data
data = {'Name': ['Alice', 'Bob', 'Charlie',
None],
        'Age': [25, None, 30, 22]}
df = pd.DataFrame(data)

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

Handling Missing Data:

- **Remove missing data:** Use **.dropna()** to remove rows with missing values.
- **Fill missing data:** Use **.fillna()** to replace missing values with a specific value.

For Example:

```
# Dropping rows with missing values
df_cleaned = df.dropna()

# Filling missing values with a default value
df_filled = df.fillna('Unknown')
```

Removing Duplicates

Duplicates in data can lead to incorrect conclusions, so they need to be removed. You can remove duplicate rows using **.drop_duplicates()**.

For Example:

```
# Sample data with duplicates
data = {'Name': ['Alice', 'Bob', 'Alice',
'Charlie'],
        'Age': [25, 22, 25, 30]}
df = pd.DataFrame(data)

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

This ensures only unique rows remain in the dataset.

Data Type Conversion

Sometimes, data in a column might not have the correct type (e.g., a column of numbers may be stored as strings). You can convert data types using the .astype() method.

For Example:

```
# Sample data with wrong data types
data = {'Name': ['Alice', 'Bob'],
        'Age': ['25', '30']}

# Age is stored as strings
df = pd.DataFrame(data)

# Converting the 'Age' column to integers
df['Age'] = df['Age'].astype(int)
print(df.dtypes)
```

This ensures that the data is stored in the correct format for analysis.

Renaming Columns and Indexes

To make your data easier to work with, you might want to rename columns or index labels. This can be done using the **.rename()** method.

For Example:

```
# Sample data
data = {'name': ['Alice', 'Bob'], 'age': [25,
30]}
df = pd.DataFrame(data)

# Renaming columns
df_renamed = df.rename(columns={'name':
'Name', 'age': 'Age'})
print(df_renamed)
```

You can rename columns and indexes to make the dataset more understandable.

Chapter 7

DATA VISUALIZATION WITH PANDAS

1. Plotting Data with Pandas
2. Basic Plot Types (Line, Bar, Histogram, etc.)
3. Customizing Plots

python™

Plotting Data with Pandas

Pandas has built-in functionality to create simple visualizations, making it easy to plot data directly from DataFrames. The **.plot()** method is the main function used to create plots.

For Example:

```python
import pandas as pd
import matplotlib.pyplot as plt

# Sample data
data = {'Year': [2018, 2019, 2020, 2021],
        'Sales': [200, 300, 400, 350]}
df = pd.DataFrame(data)

# Plotting a line graph
df.plot(x='Year', y='Sales', kind='line')
plt.show()
```

This generates a basic line plot of the sales over the years.

Basic Plot Types(Line, Bar, Histogram)

Pandas supports several types of plots. You can specify the type of plot using the kind argument in the **.plot()** function.

Line Plot: Useful for showing trends over time.

For Example:

```
df.plot(x='Year', y='Sales', kind='line')
plt.show()
```

Bar Plot: Useful for comparing categories.

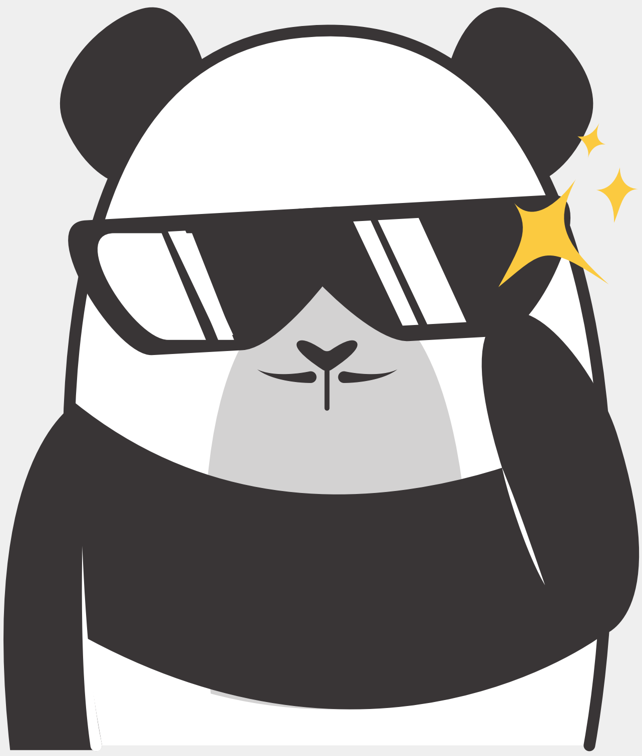
For Example:

```
df.plot(x='Year', y='Sales', kind='bar')
plt.show()
```

Histogram: Useful for showing the distribution of a variable.

For Example:

```
# Sample data for histogram
data = {'Age': [22, 25, 25, 30, 22, 35, 30,
22]}
df = pd.DataFrame(data)

# Plotting a histogram
df.plot(y='Age', kind='hist', bins=5)
plt.show()
```



Customizing Plots

You can customize plots to make them more informative by adding titles, labels, colors, and adjusting the figure size.

For Example:

```
# Customizing the line plot
df.plot(x='Year', y='Sales', kind='line',
color='green', figsize=(8, 6))

# Adding labels and title
plt.title('Yearly Sales')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.show()
```

You can customize almost every aspect of the plot to make it more readable and visually appealing.

Chapter 8

WORKING WITH DATES AND TIMES

1. DateTime in Pandas
2. Converting Strings to DateTime
3. DateTime Operations

Customizing Plots

You can customize plots to make them more informative by adding titles, labels, colors, and adjusting the figure size.

For Example:

```
# Customizing the line plot
df.plot(x='Year', y='Sales', kind='line',
color='green', figsize=(8, 6))

# Adding labels and title
plt.title('Yearly Sales')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.show()
```

You can customize almost every aspect of the plot to make it more readable and visually appealing.

DateTime in Pandas

Pandas provides powerful tools to work with dates and times using the datetime module.

Dates and times are stored in a special data type called datetime64 in pandas, which allows for efficient time-based operations.

For Example:

```
import pandas as pd

# Sample data
data = {'Date': ['2023-01-01', '2023-02-01',
'2023-03-01']}
df = pd.DataFrame(data)

# Converting the 'Date' column to datetime
df['Date'] = pd.to_datetime(df['Date'])
print(df)
```

Here, with the help of pandas we convert the string dates into datetime objects that you can work with more easily.

DateTime Operations

Pandas allows you to perform various operations on datetime data, such as extracting specific parts (e.g., year, month) or performing calculations (e.g., adding/subtracting days).

For Example 1: Extracting Year, Month, Day

```
# Sample data
data = {'Date': ['2023-01-01', '2023-02-01']}
df = pd.DataFrame(data)
df['Date'] = pd.to_datetime(df['Date'])

# Extracting year, month, and day
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
print(df)
```

Here, the code converts the **'Date'** column to **DateTime format** and then **extracts the year, month, and day** into new columns.

Then, the updated DataFrame shows the original date along with the separated year, month, and day.

Example 2: Adding/Subtracting Days *You can use **pd.DateOffset** to add or subtract time from a date.*

```
# Adding 5 days to the date
df['Date_Added'] = df['Date'] +
pd.DateOffset(days=5)
print(df)

# Subtracting 7 days from the date
df['Date_Subtracted'] = df['Date'] -
pd.DateOffset(days=7)
print(df)
```

Here, the code adds 5 days to the **'Date'** column and stores the result in a new **'Date_Added'** column.

It also **subtracts 7 days** from the date, saving it in a **'Date_Subtracted'** column.

The DataFrame now displays the original date along with the adjusted dates.

Chapter 9

INPUT/OUTPUT OPERATIONS

1. Reading Data from CSV, Excel, and other formats
2. Writing Data to Files
3. Working with Large Datasets

Reading Data from CSV, Excel, and Other Formats

Pandas makes it easy to load data from various formats like CSV and Excel into DataFrames using simple functions.

Now we will check how to read CSV, Excel files with the help of Pandas.

Reading CSV Files: CSV (Comma-Separated Values) is one of the most common formats for storing data.

You can read a CSV file using pd.read_csv().

For Example :

```python
import pandas as pd

# Reading data from a CSV file
df = pd.read_csv('data.csv')

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

Here, the code reads data from a CSV file into a DataFrame using pandas and then displays the first 5 rows of the data using **df.head()**.

- **Reading Excel Files:** You can also read data from Excel files using pd.read_excel(). You may need to specify the sheet name.

For Example :

```
# Reading data from an Excel file
df = pd.read_excel('data.xlsx',
sheet_name='Sheet1')

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

Here, the code reads data from an Excel file, specifically from **'Sheet1'**, into a **DataFrame** using pandas.

It then displays the first 5 rows of the data using **df.head()**.

Writing Data to Files

Once you've worked on your data, you can save it back to various formats like CSV, Excel, or JSON using Pandas' to_* methods.

Writing Data to a CSV File: You can save your DataFrame as a CSV file using **df.to_csv()**.

For Example :

```
# Writing data to a CSV file
df.to_csv('output.csv', index=False)
```

- **Writing Data to an Excel File:** Similarly, you can write data to an Excel file using **df.to_excel().**

For Example :

```
# Writing data to an Excel file
df.to_excel('output.xlsx', index=False)
```

You can also save the data in other formats, such as JSON, by using the `to_json()` method.

Working with Large Datasets

When working with large datasets, it’s important to handle the data efficiently to avoid memory issues.

Some ways to handle large datasets include reading the data in chunks and optimizing memory usage.

- **Reading Data in Chunks:** If the dataset is too large to fit into memory, you can load it in smaller chunks using the chunksize parameter in **read_csv()**.

For Example :

```
# Reading large CSV in chunks
chunk_size = 1000
for chunk in pd.read_csv('large_data.csv',
chunksize=chunk_size):
    print(chunk.head())
```

Here, the code reads a large CSV file in chunks of 1000 rows at a time using pandas. And then it prints the first 5 rows of each chunk.

- **Optimizing Memory Usage:** You can reduce memory usage by specifying data types when reading the file.

For Example :

```
# Reducing memory usage by specifying data
types
df = pd.read_csv('large_data.csv', dtype=
{'column_name': 'int32'})
```

Here, the code reads a CSV file while specifying data types for columns to reduce memory usage. For example, it sets '**column_name**' to use the '**int32**' data type.

Chapter 9

ADVANCED TOPICS

1. Pivot Tables
2. Reshaping Data (Melt and Pivot)
3. MultiIndex DataFrames

Pivot Tables

Pivot tables are a powerful tool to summarize and analyze data.

They allow you to aggregate data based on different criteria, similar to pivot tables in Excel.

Creating a Pivot Table: Use the pd.pivot_table() function to create a pivot table from your DataFrame.

For Example :

```
import pandas as pd

# Sample data
data = {'Date': ['2023-01-01', '2023-01-01',
'2023-02-01'],
        'Category': ['A', 'B', 'A'],
        'Sales': [200, 150, 300]}
df = pd.DataFrame(data)

# Creating a pivot table
pivot_table = pd.pivot_table(df,
values='Sales', index='Date',
columns='Category', aggfunc='sum')
print(pivot_table)
```

In this example, the pivot table summarizes the total sales for each category on each date.

Reshaping Data (Melt and Pivot)

Reshaping data helps in transforming the structure of your DataFrame to make it more suitable for analysis.

Melt: pd.melt() is used to unpivot a DataFrame from wide format to long format. It helps in converting columns into rows.

For Example :

```
# Sample data
data = {'Date': ['2023-01-01', '2023-02-01'],
        'Sales_A': [200, 300],
        'Sales_B': [150, 250]}
df = pd.DataFrame(data)

# Melting the DataFrame
melted_df = pd.melt(df, id_vars=['Date'],
value_vars=['Sales_A', 'Sales_B'],
var_name='Category', value_name='Sales')
print(melted_df)
```

In this example, the code creates a DataFrame with sales data, then melts it to reshape the DataFrame.

It converts 'Sales_A' and 'Sales_B' columns into a single 'Category' column with corresponding 'Sales' values, while keeping 'Date' as the identifier.

Pivot: pd.pivot() is used to reshape data from long format to wide format, creating a DataFrame with hierarchical indexing.

For Example :

```
# Sample data
melted_data = {'Date': ['2023-01-01', '2023-
01-01', '2023-02-01', '2023-02-01'],
               'Category': ['Sales_A',
'Sales_B', 'Sales_A', 'Sales_B'],
               'Sales': [200, 150, 300, 250]}
melted_df = pd.DataFrame(melted_data)

# Pivoting the DataFrame
pivoted_df = melted_df.pivot(index='Date',
columns='Category', values='Sales')
print(pivoted_df)
```

In this example, the code pivots a melted DataFrame to reshape it, converting '**Category**' values into columns and using '**Date**' as the index.

The result shows '**Sales_A**' and '**Sales_B**' as separate columns with their respective sales figures.

MultiIndex DataFrames

A MultiIndex DataFrame allows you to have multiple levels of indexing on rows and columns. This is useful for working with hierarchical data.

Creating a MultiIndex DataFrame: Use the pd.MultiIndex.from_tuples() function to create a MultiIndex, and then apply it to your DataFrame.

For Example :

```
# Sample data with multi-level index
arrays = [['2023-01', '2023-01', '2023-02',
'2023-02'],
          ['A', 'B', 'A', 'B']]
index = pd.MultiIndex.from_arrays(arrays,
names=('Month', 'Category'))

data = {'Sales': [200, 150, 300, 250]}
df = pd.DataFrame(data, index=index)
print(df)
```

In this example, the DataFrame features a hierarchical index with 'Month' and 'Category' as its levels. The code constructs this DataFrame and displays sales data organized by these multi-level indices, providing sales figures for each month-category pair.