Pandas: Python Library for Datasets

educative

pandas is a powerful Python library used for data manipulation and analysis, providing tools to work with structured data seamlessly. It's widely used for data cleaning, transformation, and analysis tasks.

Installation

```
pip install pandas
```

Getting Started

Importing pandas

To start using pandas, import it as follows:

```
import pandas as pd
```

Basic Data Structures

pandas primarily use two data structures:

- Series
- DataFrame

Series and DataFrame

A **series** is a one-dimensional labeled array, and a **DataFrame** is a two-dimensional labeled data structure.

Creating a series:

```
# Create a series from a list
s = pd.Series([1, 2, 3, 4])
```

Creating a DataFrame:

```
# Create a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
```

Overview of pandas Data Types

pandas supports various data types like `int64`, `float64`, `category`, `datetime64`, etc., allowing efficient storage and computation.

The category data type in pandas is designed to handle categorical data, which can take on a limited and usually fixed number of possible values.

```
# Example of creating a categorical column
data = pd.DataFrame({
'Color': pd.Categorical(['red', 'green', 'blue', 'green', 'red', 'blue']),
'Size': pd.Categorical(['small', 'medium', 'large', 'medium', 'small',
'large'],
categories=['small', 'medium', 'large'], ordered=True)})

# Output
#    Color  Size
# 0  red    small
# 1  green  medium
# 2  blue   large
# 3  green  medium
# 4  red    small
# 5  blue   large
```

Converting between data types

```
# Convert 'Age' column to float
df['Age'] = df['Age'].astype(float)
```

Handling categorical data

Categorical data can be stored more efficiently and can improve performance in analysis.

```
# Convert a column to the categorical type
df['Category'] = df['Category'].astype('category')
```

DataFrame Basics

Creating DataFrames

DataFrames can be created from various data sources, making them versatile for different data types.

From dictionaries

Create a DataFrame using a dictionary where keys are column names.

```
# Creating DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'], 'Age': [25,
30, 22, 35, 28], 'City': ['New York', 'Los Angeles', 'Chicago', 'Houston',
'San Francisco']}

df = pd.DataFrame(data)

# Output
#     Name    Age  City
# 0   Alice   25   New York
# 1   Bob     30   Los Angeles
# 2   Charlie 22   Chicago
# 3   David   35   Houston
# 4   Eve     28   San Francisco
```

From lists

Create a DataFrame from a list of lists, specifying column names.

```
data = [['Alice', 25, 'New York'], ['Bob', 30, 'Los Angeles'], ['Charlie',
22, 'Chicago'], ['David', 35, 'Houston'], ['Eve', 28, 'San Francisco']]

df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])

# Output
#     Name    Age  City
# 0   Alice   25   New York
# 1   Bob     30   Los Angeles
# 2   Charlie 22   Chicago
# 3   David   35   Houston
# 4   Eve     28   San Francisco
```

From CSV/Excel files

Read data from CSV or Excel files into a DataFrame.

```
df = pd.read_csv('data.csv')   # For CSV files
df = pd.read_excel('data.xlsx') # For Excel files
```

Inspecting DataFrames

After creating a DataFrame, inspecting it helps one understand its structure and content.

`head(), tail()`

View the first or last few rows of the DataFrame.

```
df.head()  # Displays the first 5 rows
df.tail()  # Displays the last 5 rows
```

`info(), describe()`

Get a summary of the DataFrame's structure and basic statistics.

```
df.info()      # Summary of data types and non-null values

# Output
# <class 'pandas.core.frame.DataFrame'>
# RangeIndex: 5 entries, 0 to 4
# Data columns (total 3 columns):
#  #   Column  Non-Null Count  Dtype
# ---  ------  --------------  -----
#  0   Name    5 non-null      object
#  1   Age     5 non-null      int64
#  2   City    5 non-null      object
# dtypes: int64(1), object(2)
# memory usage: 252.0+ bytes
# None

df.describe()  # Summary statistics for numerical columns

# Output
#             Age
# count   5.000000
# mean   28.000000
# std     4.949747
# min    22.000000
# 25%    25.000000
# 50%    28.000000
# 75%    30.000000
# max    35.000000
```

`shape, columns, index`

Check the size, column names, and index of the DataFrame.

```
df.shape    # Tuple of (rows, columns)

# Output
# (5, 3)

df.columns  # List of column names

# Output
# Index(['Name', 'Age', 'City'], dtype='object')

df.index    # Index labels of the DataFrame

# Output
# RangeIndex(start=0, stop=5, step=1)
```

Indexing and Selecting Data

Selecting columns

Access columns using either dot notation or bracket notation.

```
df['Name']  # Select the 'Name' column
df.Name     # Another way to select the 'Name' column

# Output
# 0    Alice
# 1    Bob
# 2    Charlie
# 3    David
# 4    Eve
# Name: Name, dtype: object
```

Selecting rows

Rows can be selected using slicing or specific methods like `loc` and `iloc`.

Label-based indexing

Use `loc[]` to select data by labels (index names).

```
df.loc[0]       # Select the first row

# Output
# Name     Alice
# Age      25
# City     New York
# Name: 0, dtype: object

df.loc[:, 'Age'] # Select the 'Age' column for all rows

# Output
# 0    25
# 1    30
# 2    22
# 3    35
# 4    28
# Name: Age, dtype: int64
```

Position-based indexing

Use `iloc[]` for selection by position (integer-based indexing).

```
df.iloc[0]      # Select the first row

# Output
# Name     Alice
# Age      25
# City     New York
# Name: 0, dtype: object

df.iloc[:, 1]   # Select the second column

# Output
# 0    25
# 1    30
# 2    22
# 3    35
# 4    28
# Name: Age, dtype: int64
```

Boolean indexing

Filter data by applying conditions directly on the DataFrame.

```
df[df['Age'] > 25]  # Select rows where 'Age' is greater than 25

# Output
#     Name   Age  City
# 1   Bob    30   Los Angeles
# 3   David  35   Houston
# 4   Eve    28   San Francisco
```

Setting and resetting the index

Set a column as the index or reset it to default integer indexing.

```
df.set_index('Name', inplace=True)  # Set the 'Name' column as index

# If inplace=True: The original DataFrame is modified directly and no
new DataFrame is created.

# If inplace=False: A new DataFrame is returned with the 'Name' column
set as the index and the original DataFrame 'df' remains unchanged.

# Output
# Name      Age   City
# Alice     25    New York
# Bob       30    Los Angeles
# Charlie   22    Chicago
# David     35    Houston
# Eve       28    San Francisco

df.reset_index(inplace=True)        # Reset to default indexing

# Output
#     Name      Age   City
# 0   Alice     25    New York
# 1   Bob       30    Los Angeles
# 2   Charlie   22    Chicago
# 3   David     35    Houston
# 4   Eve       28    San Francisco
```

Creating MultiIndex (hierarchical indexing)

`MultiIndex` allows for more complex data structures where you can index along multiple dimensions (levels), such as grouping by categories and subcategories.

```
# Define two arrays for the two levels of the index
arrays = [
    ['A', 'A', 'B', 'B'],    # First level of the index
    ['one', 'two', 'one', 'two']  # Second level of the index
]

# Create a MultiIndex from the arrays
index = pd.MultiIndex.from_arrays(arrays, names=['Group', 'Subgroup'])

# Create a DataFrame with the MultiIndex
data = {'Value': [10, 20, 30, 40]}  # Sample data to populate the DataFrame
df = pd.DataFrame(data, index=index)

# Output:
# Group  Subgroup   Value
# A      one        10
#        two        20
# B      one        30
#        two        40
```

Accessing data with `MultiIndex`

Select data within a `MultiIndex` DataFrame using tuples.

```
df.loc['A', 'one']  # Access data for Group 'A' and Subgroup 'one'

# Output
# Value    10
# Name: (A, one), dtype: int64
```

Operations on `MultiIndex` DataFrames

Perform operations like aggregation on `MultiIndex` DataFrames.

```
df.groupby(level='Group').sum()  # Sum data at the 'Group' level

# Ouput
# Group  Value
# A      30
# B      70
```

Data Manipulation and Cleaning

Adding/Removing Columns

Easily modify the structure of your DataFrame by adding or removing columns.

Adding new columns

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

```
df['Salary'] = [50000, 60000, 70000, 10000, 20000]  # Add 'Salary' column

# Output
#     Name     Age  City           Salary
# 0   Alice    25   New York       50000
# 1   Bob      30   Los Angeles    60000
# 2   Charlie  22   Chicago        70000
# 3   David    35   Houston        10000
# 4   Eve      28   San Francisco  20000
```

Dropping columns

Remove unwanted columns from DataFrame.

```
df.drop('Salary', axis=1, inplace=True)  # Drop the 'Salary' column

# axis=1 specifies that you want to drop a column (axis=0 would be for
dropping rows).

# Output
#     Name     Age  City
# 0   Alice    25   New York
# 1   Bob      30   Los Angeles
# 2   Charlie  22   Chicago
# 3   David    35   Houston
# 4   Eve      28   San Francisco
```

Renaming columns and index

Change the names of columns or the index labels.

```
df.rename(columns={'Name': 'FullName'}, index={0: 'Row1', 1: 'Row2', 2:
'Row3', 3: 'Row4', 4: 'Row5'}, inplace=True)

# Output
#       FullName     Age  City
# Row1  Alice     25   New York
# Row2  Bob       30   Los Angeles
# Row3  Charlie   22   Chicago
# Row4  David     35   Houston
# Row5  Eve       28   San Francisco
```

Handling Missing Data

Missing data is common in datasets. pandas provide ways to detect, drop, fill, or replace missing values.

Detecting missing data

Identify missing values using `isnull()`.

```
df['Age'].isnull()  # Check for missing values in 'Age' column

# Output
# Row1    False
# Row2    False
# Row3    False
# Row4    False
# Row5    False
# Name: Age, dtype: bool
```

Dropping missing data

Remove rows or columns that contain missing values.

```
df.dropna(inplace=True)  # Drop rows with missing values

# Output
#       FullName  Age  City
# Row1  Alice     25   New York
# Row2  Bob       30   Los Angeles
# Row3  Charlie   22   Chicago
# Row4  David     35   Houston
# Row5  Eve       28   San Francisco
```

Filling missing data

Fill in missing data with respective values.

```
df.fillna({
    'FullName': 'Unknown',    # Fill missing FullName with 'Unknown'
    'Age': 0,                 # Fill missing Age with 0
    'City': 'Unknown'         # Fill missing City with 'Unknown'
}, inplace=True)

# Output
#       FullName  Age  City
# Row1  Alice     25   New York
# Row2  Bob       30   Los Angeles
# Row3  Charlie   22   Chicago
# Row4  David     35   Houston
# Row5  Eve       28   San Francisco
```

Replacing values

Replace specific values with others.

```
df.replace({'Bob': 'Alice'}, inplace=True) # Replace 'Bob' with 'Alice'
df.replace({30: 25}, inplace=True)     # Replace '30' with '25'
df.replace({'Los Angeles': 'New York'}, inplace=True)      # Replace 'New York'' with 'Los Angeles'

# Output
#       FullName  Age  City
# Row1  Alice     25   New York
# Row2  Alice     25   New York
# Row3  Charlie   22   Chicago
# Row4  David     35   Houston
# Row5  Eve       28   San Francisco
```

Handling Duplicates

Detect and manage duplicate data to ensure data integrity.

Detecting duplicates

Find duplicate rows in DataFrame.

```
df.duplicated()  # Check for duplicate rows

# Output
# Row1    False
# Row2    True
# Row3    False
# Row4    False
# Row5    False
# dtype: bool

df.duplicated(subset=df.columns) # Check for duplicate rows based on the
values in the specified columns

# Output
# Row1    False
# Row2    True
# Row3    False
# Row4    False
# Row5    False
# dtype: bool
```

Removing duplicates

Remove duplicate rows from DataFrame.

```
df.drop_duplicates(inplace=True)

# Output
#       FullName  Age  City
# Row1  Alice     25   New York
# Row3  Charlie   22   Chicago
# Row4  David     35   Houston
# Row5  Eve       28   San Francisco
```

String Operations

pandas provide powerful string manipulation capabilities to clean and transform text data.

String methods

Apply string methods directly to DataFrame columns.

```
df['FullName'] = df['FullName'].str.upper()  # Convert names to uppercase

# Output
#       FullName  Age  City
# Row1  ALICE     25   New York
# Row3  CHARLIE   22   Chicago
# Row4  DAVID     35   Houston
# Row5  EVE       28   San Francisco

df['FullName'].str.contains(r'A', regex=True)  # Filter rows where 'Full-
Name' contains the letter 'A'

# Output
# Row1      True
# Row3      True
# Row4      True
# Row5     False
# Name: FullName, dtype: bool

df['City'].str.replace(r'^New', 'Old')  # Replace 'New' with 'Old' in the
'City' column
```

```
# Output
# Row1   Old York
# Row3   Chicago
# Row4   Houston
# Row5   San Francisco
# Name: City, dtype: object

df['FullName'].str.extract(r'(^\w)', expand=False)  # Extract the first letter of 'FullName'

# expand=False: Returns a Series for a single group or a DataFrame for multiple groups in str.extract(). if expand=True, it always returns a DataFrame.

# Output
# Row1  A
# Row3  C
# Row4  D
# Row5  E
# Name: FullName, dtype: object
```

Splitting and replacing strings

Split or replace parts of strings in a column.

```
df['FirstName'] = df['FullName'].str.split().str[0]  # Extract first names

# Output
#       FullName  Age   City          FirstName
# Row1  ALICE     25    New York      ALICE
# Row3  CHARLIE   22    Chicago       CHARLIE
# Row4  DAVID     35    Houston       DAVID
# Row5  EVE       28    San Francisco EVE
```

The `apply()` function for element-wise operations

Apply a function to each element of DataFrame.

```
df['Age'] = df['Age'].apply(lambda x: x + 5)  # Add 5 to each age

# Output
#       FullName  Age  City
# Row1  Alice     30   New York
# Row3  Charlie   27   Chicago
# Row4  David     40   Houston
# Row5  Eve       33   San Francisco
```

Data Operations

Basic Operations

pandas allow easy arithmetic and statistical operations on DataFrames.

Arithmetic operations

Perform element-wise operations directly on DataFrames.

```
df['New_Age'] = df['Age'] + 5



# Output
#       FullName  Age  City          New_Age
# Row1  ALICE     30   New York      35
# Row3  CHARLIE   27   Chicago       32
# Row4  DAVID     40   Houston       45
# Row5  EVE       33   San Francisco 38
```

Statistical operations

Compute summary statistics like mean, median, etc.

```
df['Age'].mean()

# Output
# 32.5
```

Grouping Data

Group data based on a column and perform aggregate functions.

Aggregation functions

Group by a column and calculate aggregate statistics.

```
df.groupby('Age').sum()

# Output
# Age  FullName  City          New_Age
# 27   CHARLIE   Chicago       32
# 30   ALICE     New York      35
# 33   EVE       San Francisco 38
# 40   DAVID     Houston       45
```

Merging and Joining

Combine DataFrames using merge or join operations.

Concatenating

Concatenate two DataFrames vertically or horizontally.

```
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
df_concat = pd.concat([df1, df2])

# Output
#    A  B
# 0  1  3
# 1  2  4
# 0  5  7
# 1  6  8
```

Data Transformation

Reshaping Data

Reshape DataFrames to get the desired layout.

Pivot tables

Create pivot tables to summarize data.

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'Alice'],
        'Subject': ['Math', 'Math', 'Math', 'Science'],
        'Score': [90, 85, 88, 95]}

df = pd.DataFrame(data)

df_pivot = df.pivot_table(values='Score', index='Name', columns='Subject',
aggfunc='mean')


# Output
# Subject  Math  Science
# Name
# Alice    90.0  95.0
# Bob      85.0  NaN
# Charlie  88.0  NaN
```

Melting data

Convert DataFrame from wide format to long format.

```
data = {'Name': ['Alice', 'Bob'],
        'Math_Score': [90, 85],
        'Science_Score': [95, 89]}
df = pd.DataFrame(data)

df_melted = pd.melt(df, id_vars=['Name'], value_vars=['Math_Score',
'Science_Score'],var_name='Subject', value_name='Score')


# Output
#    Name    Subject        Score
# 0  Alice   Math_Score     90
# 1  Bob     Math_Score     85
# 2  Alice   Science_Score  95
# 3  Bob     Science_Score  89
```

Sorting Data

Sort DataFrames by index or values.

Sorting by index

Sort data by the index labels.

```
data = {'Name': ['Charlie', 'Alice', 'Bob'],
        'Age': [23, 25, 22]}
df = pd.DataFrame(data)
df_sorted_index = df.set_index('Name').sort_index()


# Output
# Name      Age
# Alice     25
# Bob       22
# Charlie   23
```

Sorting by values

Sort data by column values.

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Score': [88, 92, 85]}
df = pd.DataFrame(data)

df_sorted_values = df.sort_values(by='Score', ascending=False)


# Output
#     Name     Score
# 1   Bob      92
# 0   Alice    88
# 2   Charlie  85
```

Binning Data

Break continuous data into separate groups or categories.

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [23, 30, 25]}
df = pd.DataFrame(data)

df['Age_Binned'] = pd.cut(df['Age'], bins=[20, 25, 30], labels=['20-25',
'25-30'])


# Output
#     Name     Age   Age_Binned
# 0   Alice    23    20-25
# 1   Bob      30    25-30
# 2   Charlie  25    20-25
```

Input/Output Operations

Reading Data

pandas make reading data from various formats easy, enabling efficient data loading and manipulation.

From SQL

Read data from an SQL database into a DataFrame using a connection string.

```
import sqlite3

conn = sqlite3.connect('database.db')
df = pd.read_sql_query("SELECT * FROM employees", conn)
```

From JSON

Read data from a JSON file into a DataFrame.

```
df = pd.read_json('data.json')
```

Writing Data

pandas allow you to export DataFrames to various formats, making data storage and sharing straightforward.

```
df = pd.read_json('data.json')
```

To CSV

Write a DataFrame to a CSV file.

```
df.to_csv('output.csv', index=False)  # index=False controls whether the row index of the DataFrame is included in the output CSV file.
```

To Excel

Write a DataFrame to an Excel file.

```
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

To SQL

Write a DataFrame to an SQL database table.

```
df.to_sql('employees', conn, if_exists='replace', index=False)
```

To JSON

Write a DataFrame to a JSON file.

```
df.to_json('output.json')
```

Time Series Data

Date Functionality

`pandas` provide robust functionality for handling and manipulating dates.

Converting to the `datetime` format

Convert a column or series to the `datetime` format for time series analysis.

```
data = { 'Date': ['2024-08-01', '2024-08-02', '2024-08-03'] }
df = pd.DataFrame(data)
df['Date'] = pd.to_datetime(df['Date'])

# Output:
#      Date
# 0   2024-08-01
# 1   2024-08-02
# 2   2024-08-03
```

Generating date ranges

Create a range of dates, useful for constructing time series data.

```
dates = pd.date_range(start='2023-01-01', periods=5, freq='D')

# Output:
# DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
# '2023-01-05'], dtype='datetime64[ns]', freq='D')
```

Resampling data

Aggregate data over a specified time-frequency, like converting daily data to monthly.

```
data = {
    'Date': ['2024-07-10', '2024-07-25', '2024-08-05', '2024-08-20'],
    'Sales': [200, 150, 300, 250]
}
df = pd.DataFrame(data)
df['Date'] = pd.to_datetime(df['Date'])

# Output
#    Date         Sales
# 0  2024-07-10   200
# 1  2024-07-25   150
# 2  2024-08-05   300
# 3  2024-08-20   250

df.set_index('Date').resample('ME').sum()  # the ME stands for "Month End"

# Output
# Date          Sales
# 2024-07-31    350
# 2024-08-31    550
```

Shifting data

Shift data backward or forward in time, often used for calculating differences or lagged features.

```
df['Previous Day'] = df['Value'].shift(1)

# Output:
#    Value  Previous Day
# 0  1      NaN
# 1  2      1.0
# 2  3      2.0
```

Time zone handling

Convert time series data to different time zones.

```
df['Date'] = df['Date'].dt.tz_localize('UTC').dt.tz_convert('US/Eastern')

# Output
#   Date
# 0 2024-07-31 20:00:00-04:00
# 1 2024-08-01 20:00:00-04:00
# 2 2024-08-02 20:00:00-04:00
```

Visualization with pandas

Basic Plotting

pandas provide easy-to-use methods for quick visualizations directly from DataFrames.

Plotting methods

You can quickly generate basic plots using the `plot()` method available on DataFrames and Series.

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [24, 27, 22, 32],
        'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']}
df = pd.DataFrame(data)
df.plot()
```

Customizing plots

Customize plots by adjusting parameters like title, labels, and figure size.

```
df.plot(title='Sample Plot', xlabel='Name', ylabel='Age', figsize=(8, 4))
```

Integrating with `Matplotlib` and `Seaborn` for Advanced Plots

`pandas` plots can be further enhanced by integrating with Matplotlib and seaborn for advanced visualizations.

Matplotlib integration

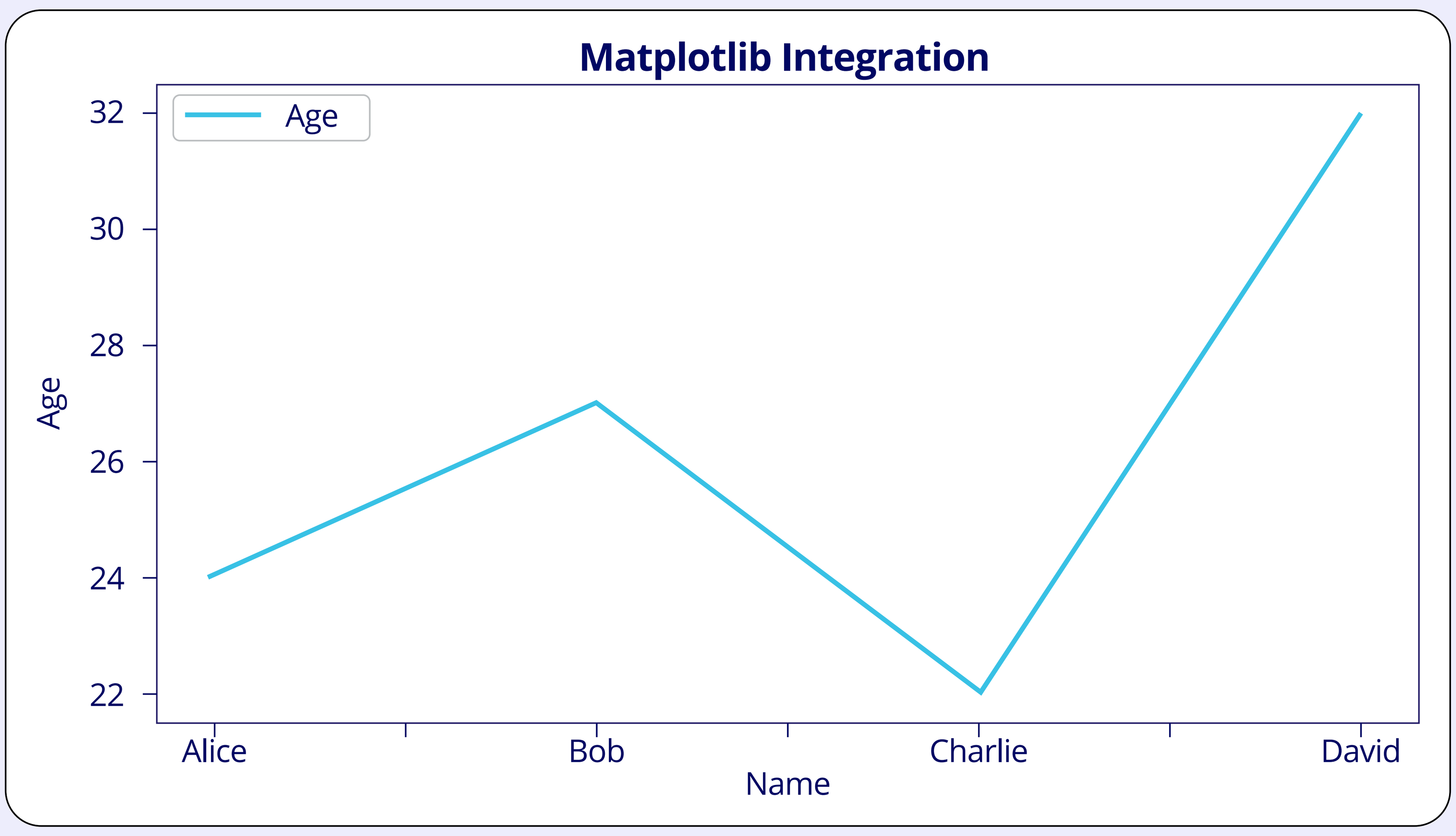
Use `Matplotlib` to further customize or save plots.

```
import matplotlib.pyplot as plt


# Sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [24, 27, 22, 32],
        'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']}

df = pd.DataFrame(data)

# Adjusting the plot with 'Name' as x-label and 'Age' as y-label
df.plot(x='Name', y='Age', kind='line')
plt.title('Matplotlib Enhanced Plot')
plt.xlabel('Name')
plt.ylabel('Age')
plt.show()
```



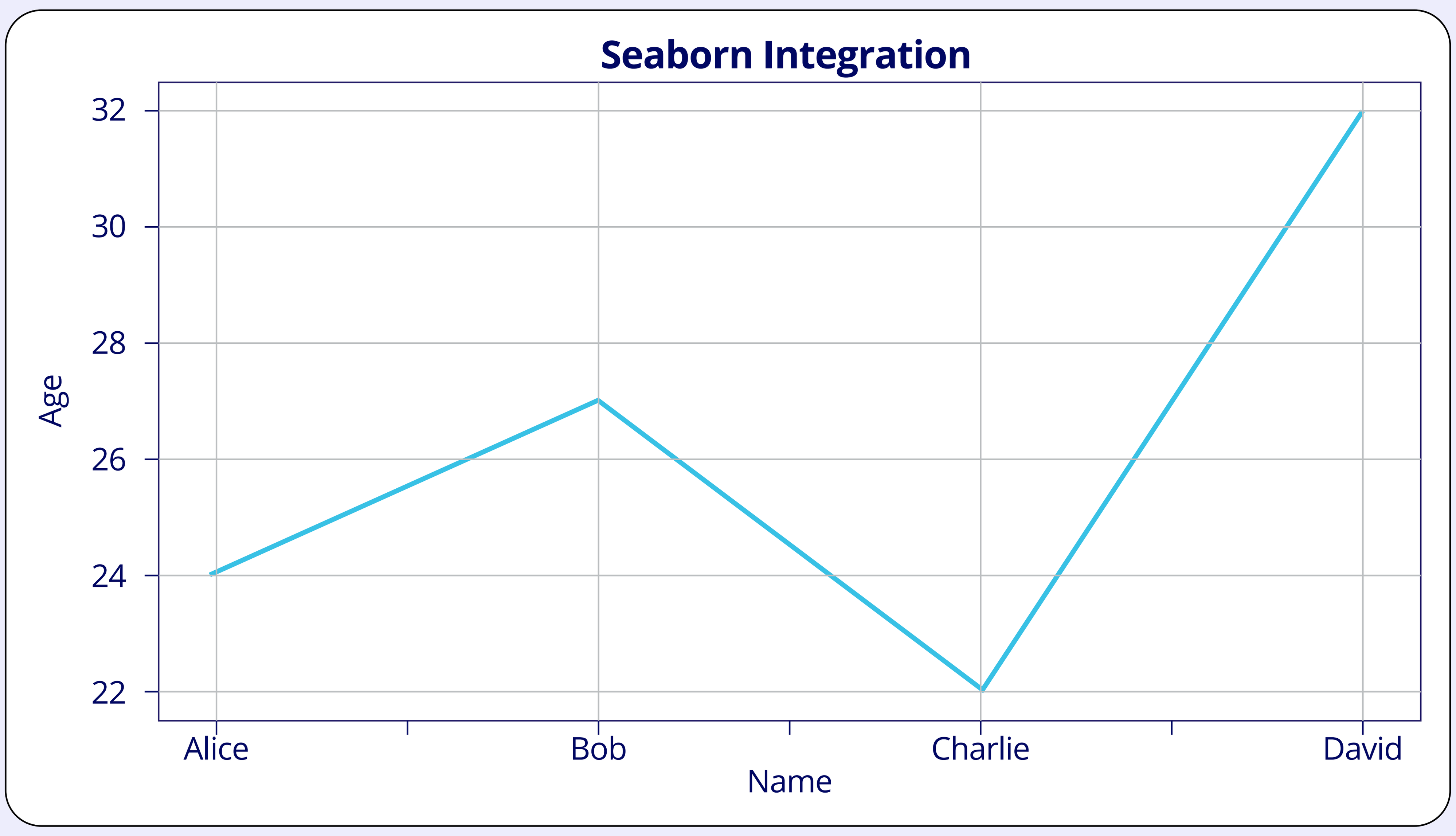
Seaborn integration

Leverage `Seaborn` for statistical plots and more advanced visual styles.

```
import seaborn as sns

# Setting the style and creating the plot
sns.set(style="whitegrid")
sns.lineplot(data=df, x='Name', y='Age')

# Display the plot
plt.title('Seaborn Integration')
plt.show()
```



Advanced Topics

Working with Large Datasets

Handling large datasets efficiently is crucial for performance.

Chunking with read_csv()

Process large CSV files in smaller chunks to save memory.

```
chunk_size = 1000
chunks = pd.read_csv('large_data.csv', chunksize=chunk_size)
for chunk in chunks:
    process(chunk)  # Replace with your processing function

# Assuming 'large_data.csv' has 10,000 rows
# Each chunk will have 1,000 rows, processed iteratively
```

Using Dask for parallel computing

Dask can parallelize pandas operations for larger-than-memory datasets.

```
import dask.dataframe as dd

ddf = dd.read_csv('large_data.csv')
result = ddf.groupby('column_name').mean().compute()

# Output is similar to pandas, but processed in parallel result:
# column_name  mean_value
# 0            15.4
# 1            22.1
```

Enhancing Performance

Optimizing code can significantly improve execution speed.

Vectorization

Apply operations to entire arrays or DataFrames instead of looping through elements.

```
df['new_column'] = df['column1'] + df['column2']

# Example DataFrame:
#     column1  column2
# 0   1        2
# 1   3        4
# 2   5        6

# After vectorization:
#     column1   column2   new_column
# 0   1         2         3
# 1   3         4         7
# 2   5         6         11
```

Avoiding loops

Replace loops with pandas built-in functions to increase speed.

```
df['new_column'] = df['column1'].apply(lambda x: x * 2)

# Original 'column1': [1, 2, 3]
# new_column after applying lambda: [2, 4, 6]
```

Working with the Categorical Data

Categorical data improves memory usage and speeds up operations.

```
df['category_column'] = df['category_column'].astype('category')

# Original DataFrame:
#      category_column
# 0    cat
# 1    dog
# 2    bird

# After conversion:
#      category_column
# 0    cat
# 1    dog
# 2    bird
# dtype: category
# Categories (3, object): ['bird', 'cat', 'dog']
```