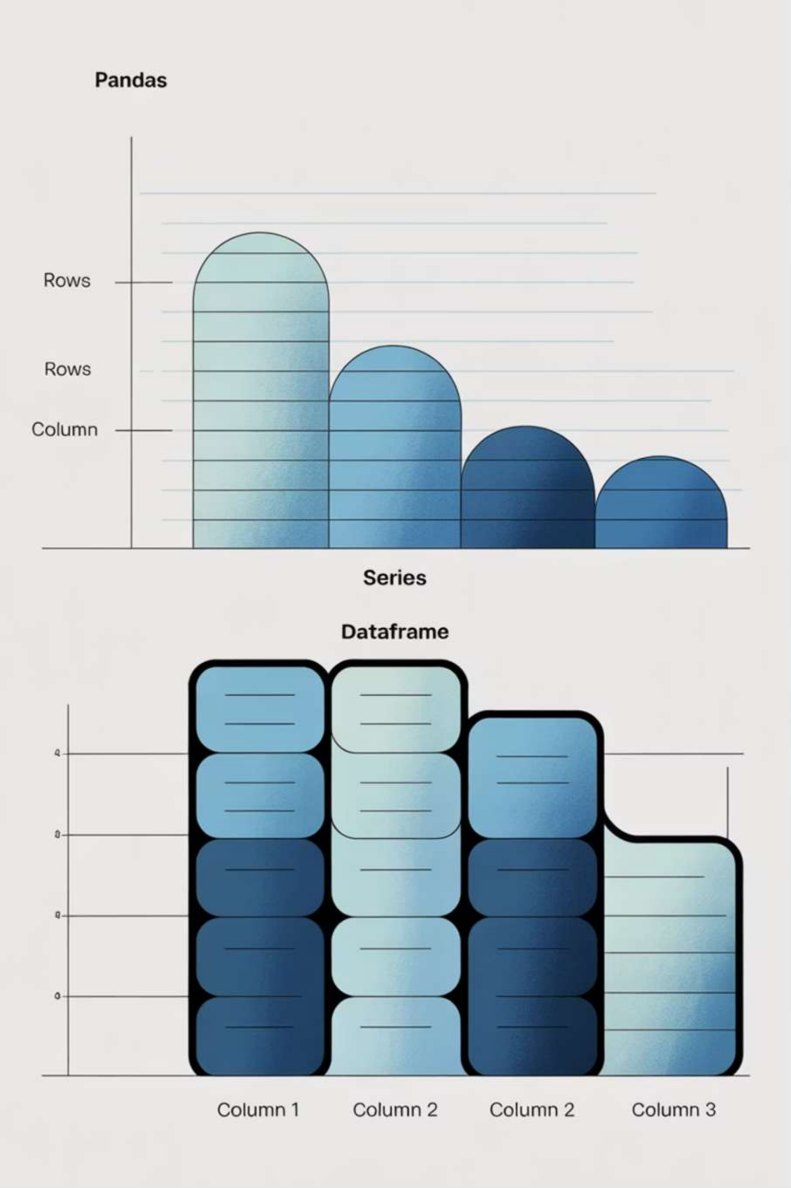
Data Manipulation with pandas: An Overview of Core Concepts

Welcome to this comprehensive guide on data manipulation using pandas, Python's premier library for working with structured data. Whether you're new to data analysis or looking to sharpen your skills, this presentation will walk you through the fundamental concepts that make pandas an essential tool in any data analyst's toolkit.



Introduction to pandas Data Structures

Fast & Flexible

pandas provides high-performance, easy-to-use data structures designed specifically for efficient data manipulation and analysis in Python.

Core Objects

The library is built around three primary data structures: Series (1D), DataFrame (2D), and Index objects, which work together to handle nearly any data format.

Designed for Tabular Data

Unlike basic Python data types, pandas objects are optimized for working with structured, tabular data—similar to what you'd find in databases, CSV files, or spreadsheets.

Chapter 1: Why pandas? The Power Behind Python Data Analysis

In the ever-evolving landscape of data science, pandas has emerged as the indisputable cornerstone of Python-based data manipulation and analysis. As of 2025, it remains the most widely adopted library among data professionals worldwide, and for good reason.

What makes pandas truly exceptional is its ability to handle real-world data challenges with remarkable efficiency. From messy CSV files with inconsistent formatting to complex time-series datasets requiring sophisticated aggregations, pandas provides an intuitive and comprehensive toolkit for transforming raw data into actionable insights.

Efficiency

Built on NumPy's optimized C backend, pandas delivers blazing-fast performance for operations on large datasets, dramatically reducing processing time compared to pure Python implementations.

Flexibility

Handle virtually any tabular data format with ease, including CSV, Excel, SQL databases, JSON, and specialized formats like Parquet and HDF5.

Integration

Seamlessly interfaces with visualization libraries like Matplotlib and Seaborn, machine learning frameworks like scikit-learn, and big data tools like Spark.

Who relies on pandas?

- Data Scientists - For exploratory data analysis and feature engineering
- Financial Analysts - For time-series analysis and risk modeling
- ML Engineers - For data preparation and transformation
- Business Analysts - For creating reports and dashboards
- Academic Researchers - For data collection and statistical analysis

Meet pandas' Core Data Structures

Series: One-Dimensional Power

A Series is pandas' implementation of a one-dimensional labeled array capable of holding any data type. Think of it as a single column from a spreadsheet with an associated index.

```
import pandas as pd

# Creating a Series from a list
s = pd.Series([10, 20, 30, 40])
print(s)

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

Key features of Series objects:

- **Custom indexes** - Replace default integer positions with meaningful labels
- **Vectorized operations** - Apply functions across all elements simultaneously
- **Automatic alignment** - Operations between Series align data by index values
- **Missing data handling** - Built-in support for NaN values and operations

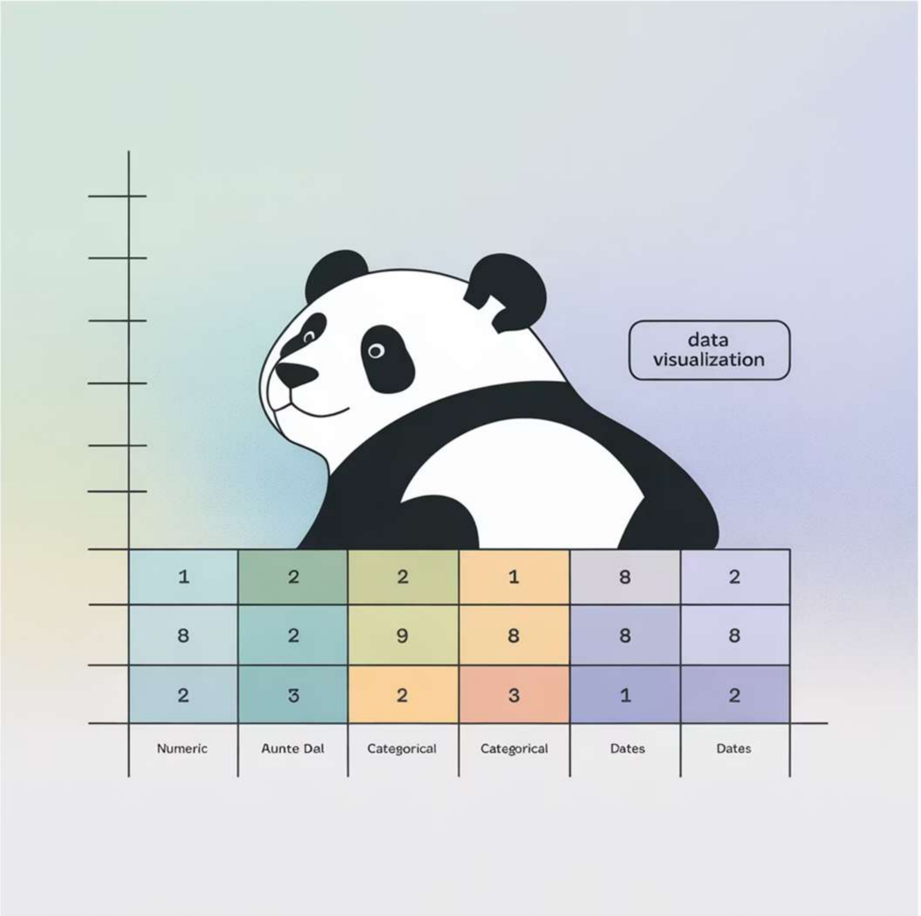
DataFrames excel at:

- **Data integration** - Merge data from multiple sources with various join operations
- **Column operations** - Add, remove, or transform columns with intuitive syntax
- **Flexible indexing** - Access data through labels, positions, or boolean conditions
- **Group operations** - Split-apply-combine functionality for aggregations
- **Time series functionality** - Specialized tools for temporal data analysis

> Both Series and DataFrame objects inherit from the `pandas.NDFrame` class, which provides common functionality and ensures consistent behavior across both structures.

DataFrame: Two-Dimensional Flexibility

A DataFrame is a 2D labeled data structure with columns that can be of different types. It's conceptually similar to a spreadsheet or SQL table.



Series: One-Dimensional Labeled Arrays

A Series is pandas' simplest data structure—essentially a one-dimensional array with axis labels. Think of it as a single column of data with an index for each row.

Key characteristics:

- Can hold any data type (integers, strings, floats, Python objects)
- Each value has a corresponding label in the index
- Supports vectorized operations
- Acts similar to both a dictionary and a NumPy array

```
# Creating a Series
import pandas as pd

# From a list
s = pd.Series([10, 20, 30, 40])

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

# From a dictionary
s = pd.Series({'a': 10, 'b': 20})
```

Import Pandas from json

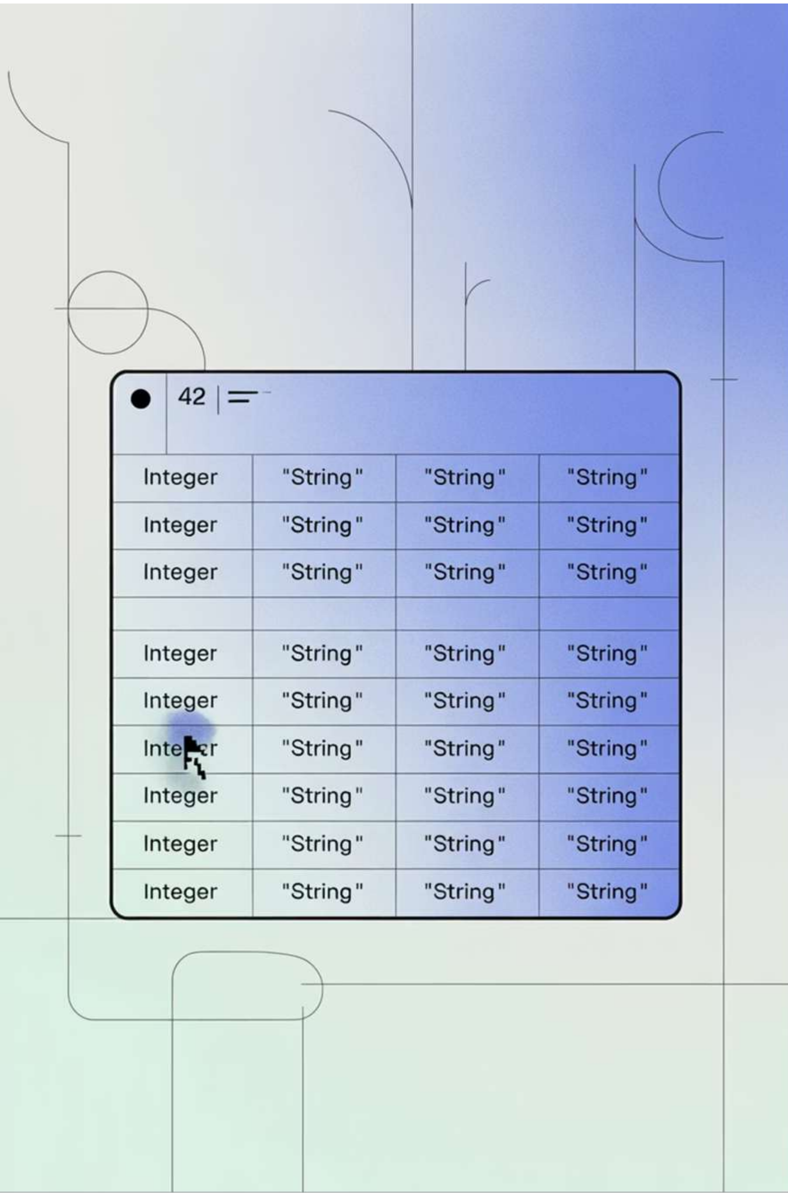
```
import pandas as pd
import json

# JSON string
json_str = '{"a": 10, "b": 20, "c": 30}'

# Convert to Python dict
data = json.loads(json_str)

# Create Series
s = pd.Series(data)
print(s)
```

Import Pandas from csv files

```
import pandas as pd
# Read CSV into DataFrame
df = pd.read_csv("weather_forecast.csv")
print(df)
# Extract a single column as Series
temperature = df["temperature"]
print(temperature)
```

Download the file

```
import requests
# Example: Weather forecast for New Delhi
url = (
    "https://api.open-meteo.com/v1/forecast?"
    "latitude=28.61&longitude=77.23&hourly=temperature_2m,r
elative_humidity_2m&format=csv"
)
response = requests.get(url)
# Save as CSV file
with open("weather_forecast.csv", "wb") as f:
    f.write(response.content)
print("Weather forecast CSV downloaded”)
```



DataFrames: Two-Dimensional Labeled Data Structures

Table-Like Structure

DataFrames are the workhorse of pandas, representing a rectangular table of data with rows and columns—similar to a spreadsheet, SQL table, or R data.frame.

Heterogeneous Data

Each column in a DataFrame can contain different data types (numeric, string, boolean, etc.), giving you flexibility when working with real-world data.

Dual Indexing

DataFrames have both row indices (vertical) and column names (horizontal), allowing for sophisticated data alignment and selection.

```
# Creating a DataFrame
df = pd.DataFrame({
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [25, 30, 35],
 'Department': ['HR', 'Engineering', 'Marketing']
})
```

Creating a Series: Simple & Labeled Data

From Lists & Arrays

```
# Basic Series from list
import pandas as pd
import numpy as np

simple_series = pd.Series([1, 3, 5, 7, 9])
array_series = pd.Series(np.array([1, 3,
5, 7, 9]))

# Both produce:
# 0    1
# 1    3
# 2    5
# 3    7
# 4    9
# dtype: int64
```

With Custom Indices

```
# Series with custom index
named_series = pd.Series(
    [10.2, 11.5, 12.7, 9.8, 8.5],
    index=['Mon', 'Tue', 'Wed', 'Thu',
'Fri']
)

# Output:
# Mon    10.2
# Tue    11.5
# Wed    12.7
# Thu     9.8
# Fri     8.5
# dtype: float64
```

From Dictionaries

```
# Series from dictionary (keys become
indices)
dict_series = pd.Series({
    'California': 39.5,
    'Texas': 29.0,
    'Florida': 21.5,
    'New York': 19.4,
    'Illinois': 12.7
})

# Output:
# California    39.5
# Texas         29.0
# Florida       21.5
# New York      19.4
# Illinois      12.7
# dtype: float64
```

Working with Series Objects

Once created, a Series offers powerful functionality for data manipulation:

Accessing Elements

Retrieve data by index position or label:

```
# By position (integer)
s[0]      # First element

# By label (if custom index)
s['Mon']  # Element with index 'Mon'

# Slicing
s[1:3] #Elements at pos 1 and 2
s['Tue':'Thu']# Elements from 'Tue'
to 'Thu'
```

Mathematical Operations

Series support vectorized operations:

```
# Arithmetic with scalar
s * 2     # Multiply all values by 2

# Operations between Series
s1 + s2   # Addition aligned by
index
```

Boolean Filtering

Filter elements based on conditions:

```
# Elements greater than 10
s[s > 10]
# Complex filtering
s[(s > 10) & (s < 20)]
```

Index Objects: The Backbone of pandas

Index objects are immutable arrays that store the axis labels for pandas objects. They serve as the foundation for data alignment and provide identity to your data.

Immutable Labels

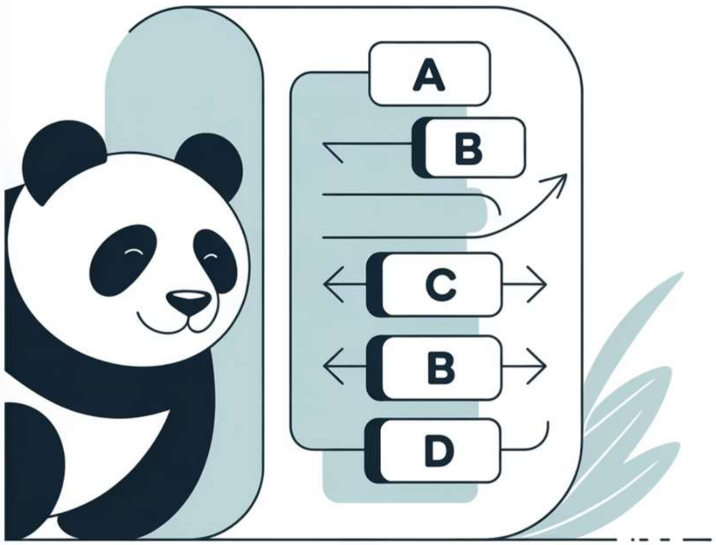
Once created, an Index cannot be modified directly, ensuring data integrity during operations.

Alignment Mechanism

Indices enable automatic alignment of data during operations, preventing errors when working with datasets of different shapes.

Duplicate Support

Unlike Python dictionaries, Index objects can contain duplicate values, offering flexibility for certain data scenarios.




```
# Working with indices
df = pd.DataFrame(data)

# View the index
print(df.index)
# RangeIndex(start=0, stop=3, step=1)

# Set a new index
df.set_index('ID', inplace=True)

# Reset index
df.reset_index()
```

Re-indexing: Conforming Data to New Labels

Re-indexing is a fundamental operation in pandas that allows you to conform a Series or DataFrame to a new set of labels. This process can insert or remove rows/columns and fill in missing values according to specified rules.

```
# Basic reindexings =
pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd’])
s_reindexed = s.reindex(['a', 'b', 'c', 'd', 'e'])# Results in: a=1, b=2, c=3, d=4, e=NaN
# Reindexing with fill method
df.reindex(new_index, fill_value=0)  # Fill missing values with zeros
```

Common use cases for reindexing include:

- Aligning multiple datasets to a common set of labels for joint analysis
- Reordering columns or rows to a specific sequence
- Adding new positions with default values
- Time series data analysis with specific date ranges

Selection and Filtering: Accessing Your Data

Label-Based Selection (.loc)

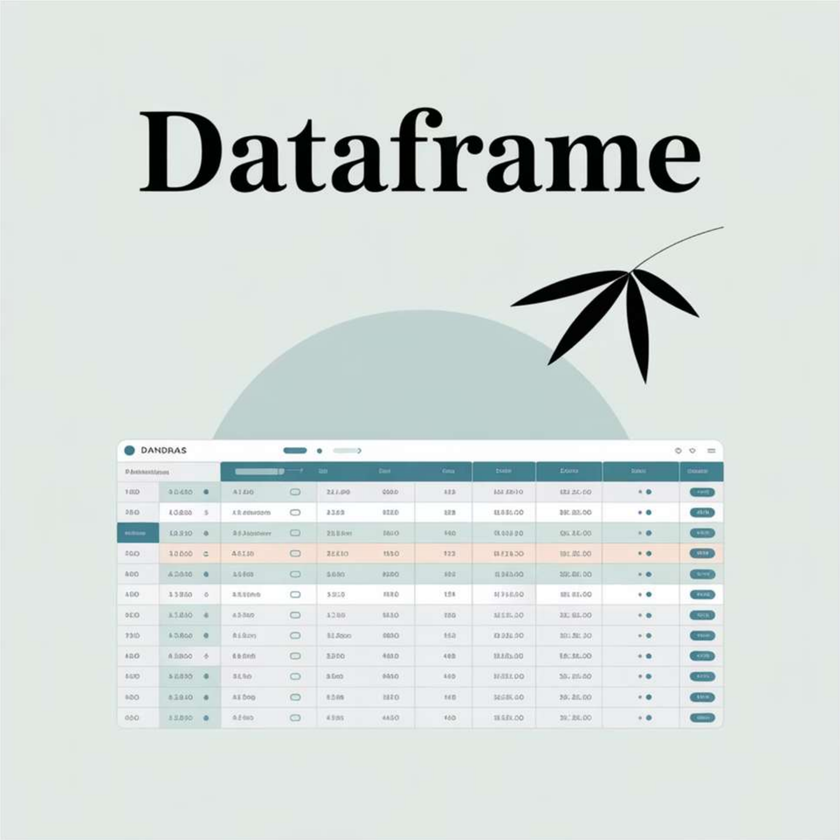
```
# Select by label
df.loc['row_label', 'column_label']

# Select multiple rows/columns
df.loc[['row1', 'row2'], ['col1', 'col2']]

# Slice with labels
df.loc['row1':'row3', :]
```

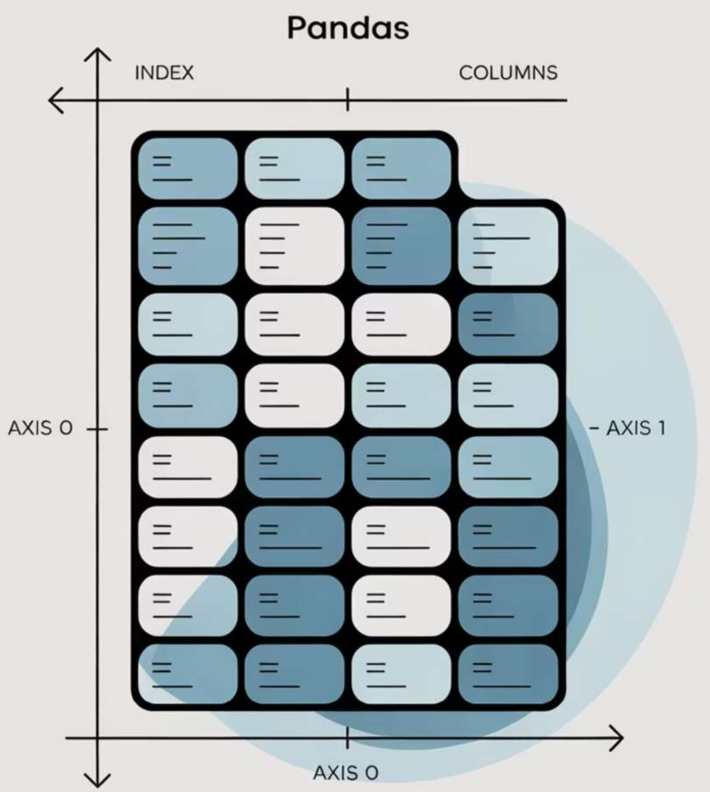
Position-Based Selection (.iloc)

```
# Select by integer position
df.iloc[0, 2]  # First row, third column

# Select multiple positions
df.iloc[[0, 2], [1, 3]]

# Slice with integers
df.iloc[0:5, :]  # First 5 rows, all columns
```

Boolean Filtering

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

# Multiple conditions
df[(df['Dept'] == 'HR') & (df['Salary'] > 50000)]

# Contains string
df[df['Name'].str.contains('Smith')]
```





Axis Indices: Row vs. Column Operations

Many pandas operations allow you to specify which axis to apply them along. Understanding axes is crucial for performing aggregations and transformations efficiently.

f

Axis=0 (Rows)

Operations applied down each column. This is the default for most pandas functions.

```
# Sum of each column
df.sum(axis=0)# Apply function to each
columndf.apply(custom_func, axis=0)
```

⟷

Axis=1 (Columns)

Operations applied across each row. Useful for row-wise calculations.

```
# Sum of each row
df.sum(axis=1)# Apply function to each rowdf.apply(custom_func,
axis=1)
```

Remember: axis=0 means "operate vertically" (down rows), while axis=1 means "operate horizontally" (across columns). This convention is consistent throughout pandas.

Summarizing Data: Extracting Insights

pandas offers powerful tools for generating descriptive statistics and summarizing your datasets, making it easy to understand the characteristics of your data at a glance.

Descriptive Statistics

The `.describe()` method generates a comprehensive statistical summary including count, mean, std, min/max, and quartiles for numeric columns.

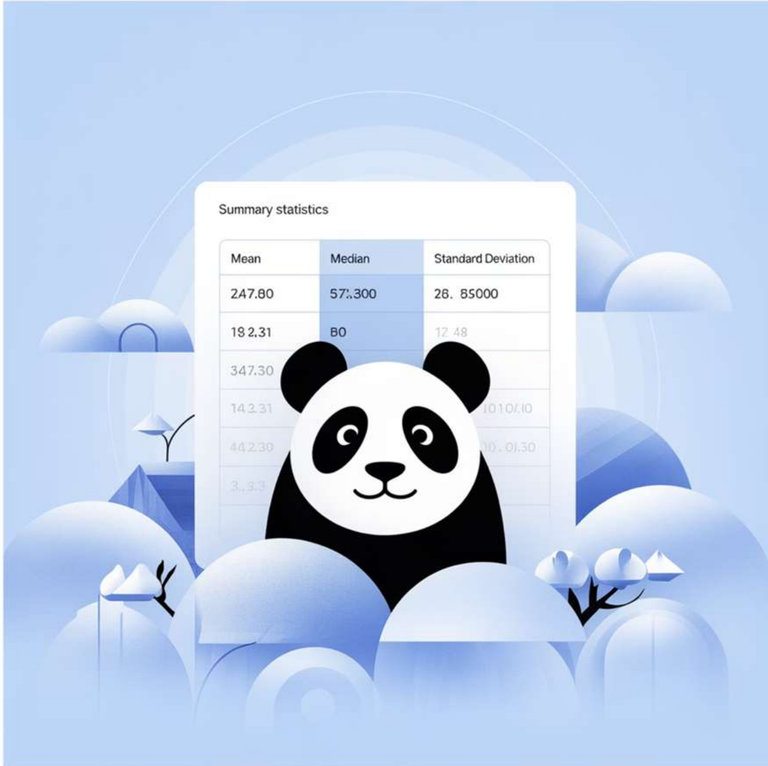
Aggregation Functions

Apply specific statistical functions like `sum()`, `mean()`, `median()`, `min()`, `max()`, and `std()` to get targeted insights.

Frequency Analysis

Use `value_counts()` to count unique values and `unique()` to identify distinct elements in your data.

```
# Basic descriptive statistics
df.describe()

# Specific aggregations
df['Salary'].mean()
df['Age'].median()
df['Department'].value_counts()

# Group by and summarize
df.groupby('Department').mean()

# Custom aggregations
df.agg({
    'Age': ['min', 'max', 'mean'],
    'Salary': ['median', 'std']
})
```



Handling Missing Data: Working with Incomplete Information

Real-world data is rarely complete. pandas provides comprehensive tools for detecting and handling missing values, represented as `NaN` (Not a Number) in your datasets.

1

Detecting Missing Values

Use `df.isnull()` or `df.isna()` to identify missing values, returning a boolean mask where `True` indicates missing data. For the opposite, use `df.notnull()` or `df.notna()`.

```
# Check for missing values
missing_mask = df.isnull()
missing_count = df.isnull().sum()
```

2

Removing Missing Values

The `dropna()` method removes rows or columns containing missing values based on specified thresholds and axis.

```
# Drop rows with any missing values
df_clean = df.dropna()
# Drop rows with all values missing
df_clean = df.dropna(how='all')
```

3

Filling Missing Values

Use `fillna()` to replace missing values with specified values, statistics, or interpolated values.

```
# Fill with a constant
df.fillna(0)
# Fill with column means
df.fillna(df.mean())
## Forward fill (propagate last valid
value)
df.fillna(method='ffill')
```

Hierarchical Indexing: Working with Multi-dimensional Data

Hierarchical indexing (MultiIndex) is a powerful pandas feature that allows you to represent higher-dimensional data in a two-dimensional DataFrame structure, enabling complex grouping and selection operations.

Multiple Levels of Indexing

A MultiIndex contains multiple levels of indices, creating a tree-like structure that allows for more complex data organization.

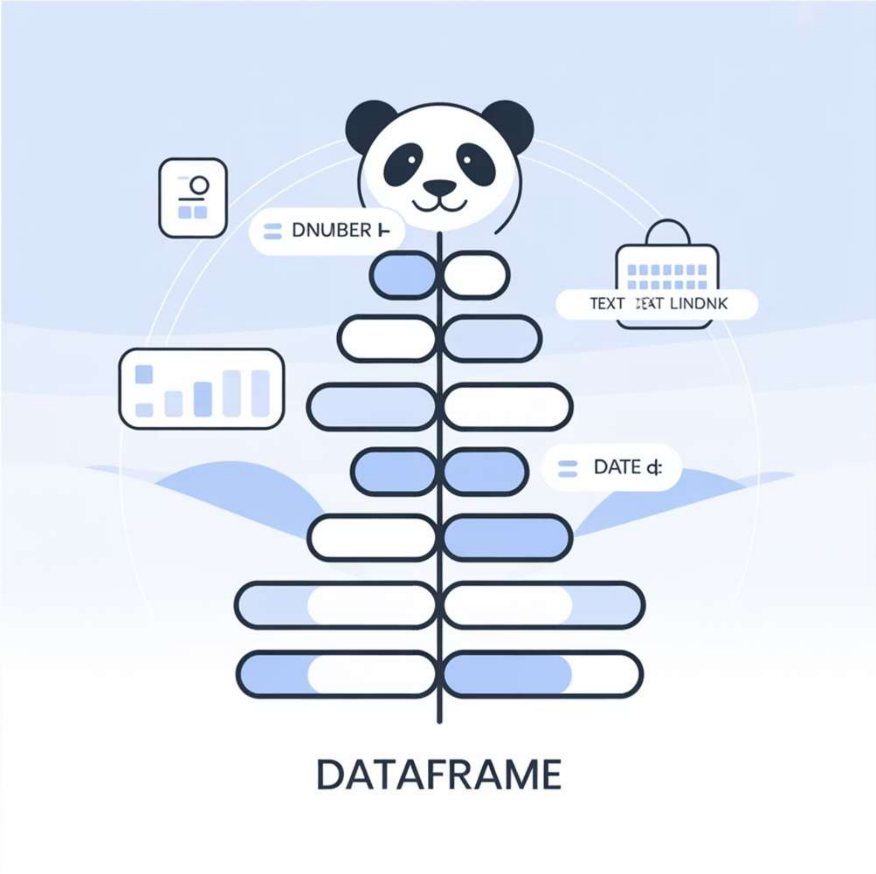
Advanced Grouping

Hierarchical indices enable sophisticated groupby operations, allowing you to aggregate data across multiple dimensions simultaneously.

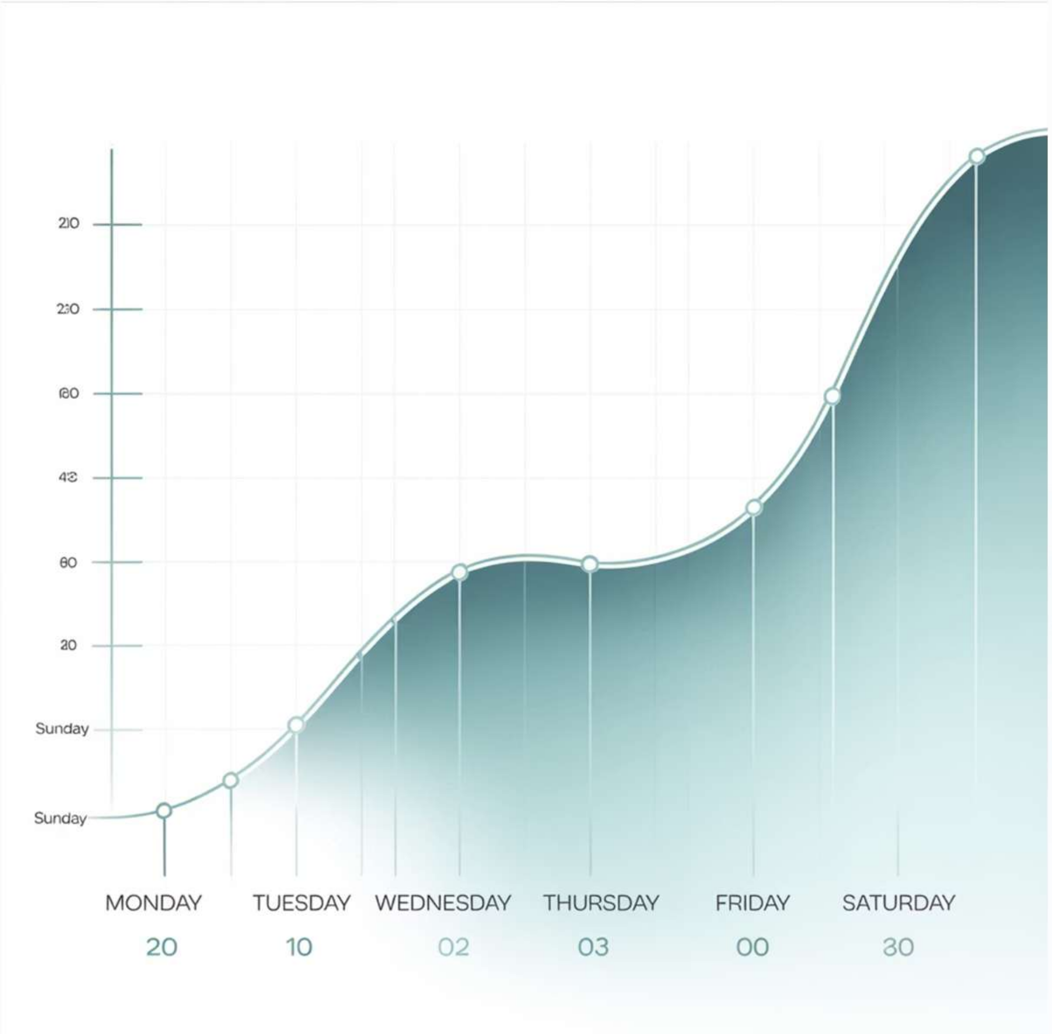
Reshaping Data

Use `stack()` and `unstack()` methods to pivot data between wide and long formats without losing information.

```
# Create MultiIndex DataFrame
arrays = [
 ['CA', 'CA', 'NY', 'NY'],
 ['San Francisco', 'Los Angeles',
 'New York', 'Buffalo']
]
index = pd.MultiIndex.from_arrays(
 arrays, names=['State', 'City'])
df = pd.DataFrame({'Population': [850000, 3900000,
 8400000, 256000]},
 index=index)

# Selection with MultiIndex
df.loc[('CA', 'San Francisco')]
```

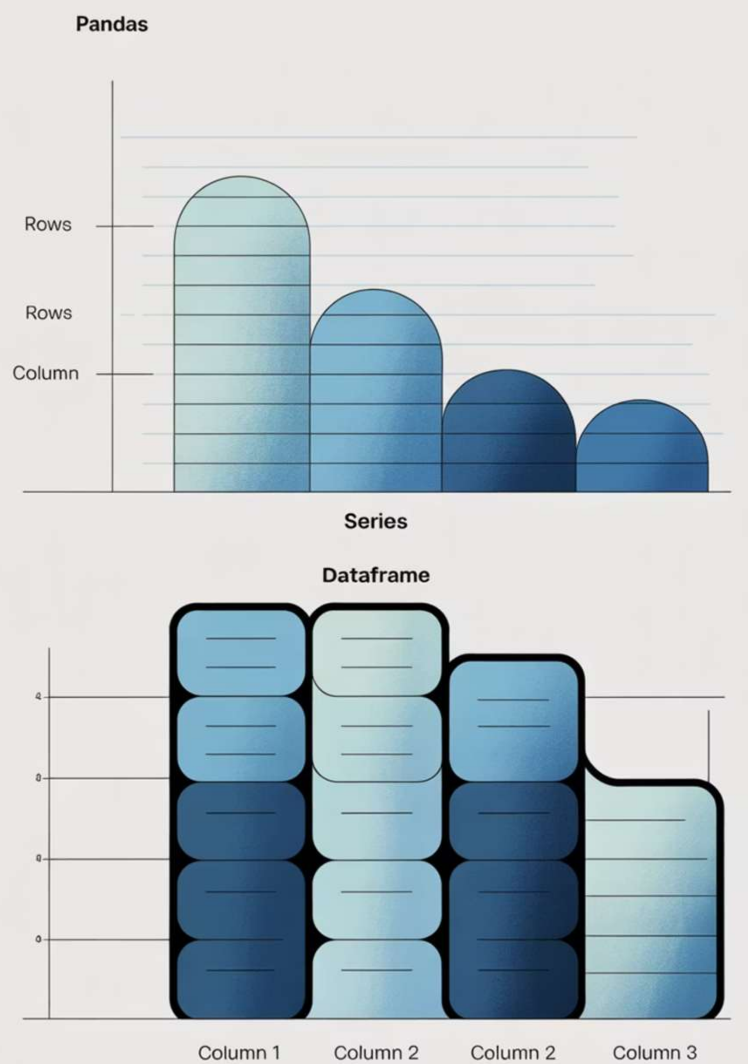


Practical Example: Temperature Analysis



Let's analyze a week's temperature data using Series operations:

```
# Daily temperatures (°F)
temps = pd.Series(
    [68, 71, 73, 69, 75, 83, 79],
    index=['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
'Sun']
)

# Basic statistics
print(f"Average: {temps.mean():.1f}°F")
print(f"Maximum: {temps.max()}°F on
{temps.idxmax()}")
print(f"Minimum: {temps.min()}°F on
{temps.idxmin()}")

# Days above 70°F
warm_days = temps[temps > 70]
print(f"Warm days: {len(warm_days)}")
print(warm_days)

# Convert to Celsius
temps_c = (temps - 32) * 5/9
print(temps_c.round(1))
```



Difference between Numpy & pandas

Feature	NumPy	pandas
Core Purpose	Numerical computing with arrays	Data analysis & manipulation (tabular, labeled data)
Main Data Structure	ndarray (N-dimensional array)	Series (1D) and DataFrame (2D)
Data Types	Primarily numbers (int, float, complex, bool)	Mixed data types (numbers, strings, dates, categories)
Indexing	Integer-based indexing (arr[0,1])	Label-based & integer indexing (df.loc["row", "col"])
Missing Data Handling	Limited (mainly NaN)	Full support (.isna(), .fillna(), .dropna())
Performance	Faster for raw numerical operations	Slightly slower but optimized for tabular data
File I/O	No direct CSV/Excel/SQL support	Direct support (read_csv, to_excel, read_sql, etc.)
Use Cases	Linear algebra, matrix ops, numerical simulations	Data cleaning, wrangling, analysis, reporting
Library Dependency	Standalone (but used under pandas)	Built on top of NumPy