Python Pandas: From Beginner to Advanced

Welcome to a comprehensive guide to Python's powerful data analysis library - Pandas. This presentation will take you from the basics to advanced techniques, with practical code examples and clear explanations at every step.

Pandas combines the flexibility of Python with the power of specialized data structures, making it the go-to tool for data scientists, analysts, and developers working with structured data. Whether you're a complete beginner or looking to enhance your skills, this guide will provide you with the knowledge to manipulate, analyze, and visualize data effectively.

V **by Vikas**

Getting Started with Pandas

Installation

Install using pip:

```
pip install pandas
pip install numpy
pip install matplotlib
```

Along with NumPy, Matplotlib and other dependencies.

Importing

Standard convention:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

This allows you to access all functionality with the pd prefix.

Basic Structures

Series: 1D labeled array

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

DataFrame: 2D labeled data structure

```
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
```

Pandas offers intuitive ways to create data structures from scratch or import external data. Loading data from CSV files is as simple as

```
df = pd.read_csv('filename.csv')
```

. Once loaded, you can examine your data with methods like

```
df.head()
```

to show the first few rows,

```
df.info()
```

for structure information, and

```
df.describe()
```

for statistical summaries.

Data Selection and Indexing

Column Selection

Access columns using either:

```
df['column_name']  # Dictionary-like
df.column_name    # Attribute-like
```

Row Selection

Select rows by position or label:

```
df.loc['row_label']    # Label-based
df.iloc[0]           # Integer-based
```

Boolean Indexing

Filter data with conditions:

```
df[df['age'] > 30]        # Single condition
df[(df['age'] > 30) & (df['income'] > 50000)]
```

Pandas provides flexible methods to slice and dice your data. You can select specific subsets by index (

```
df.set_index('column')
```

), reset indices back to integers (

```
df.reset_index()
```

), or chain operations for complex selections. The combination of position-based, label-based, and boolean indexing makes data selection intuitive yet powerful.

Data Cleaning and Preprocessing

Missing Values

Identify missing values:

```
df.isnull().sum()
```

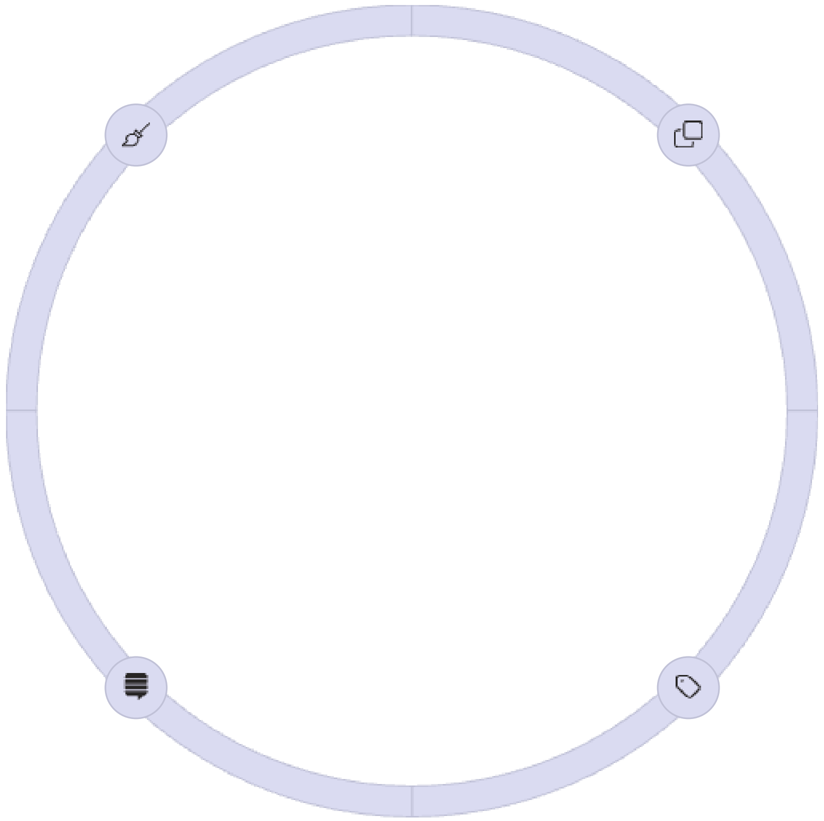
Handle missing data:

```
df.dropna()      # Remove rows
df.fillna(value)  # Replace with value
```

Data Types

Convert types:

```
df['col'].astype('int')
pd.to_numeric(df['col'])
pd.to_datetime(df['date'])
```



Duplicates

Find and remove duplicates:

```
df.duplicated()       # Identify
df.drop_duplicates()  # Remove
```

Column Renaming

Change column names:

```
df.rename(columns={'old': 'new'})
```

Clean data is essential for accurate analysis. Pandas provides comprehensive tools to identify and handle missing values, duplicates, and inconsistent formats. Remember that

```
dropna()
```

and

```
fillna()
```

can be customized with parameters like

```
subset
```

to target specific columns or

```
method
```

Data Manipulation and Transformation

1 Sorting

Order your data easily with sort methods:

```
df.sort_values('column', ascending=False)
df.sort_index()  # Sort by index
```

2 Grouping & Aggregation

Split-apply-combine operations:

```
df.groupby('category').mean()
df.groupby(['cat1', 'cat2']).agg({
    'col1': 'sum',
    'col2': ['min', 'max']
})
```

3 Merging & Joining

Combine datasets like SQL joins:

```
pd.merge(df1, df2, on='key', how='inner')
df1.join(df2, how='left')
```

4 Reshaping

Transform data layout:

```
df.pivot(index='date', columns='category', values='amount')
pd.melt(df, id_vars=['date'], value_vars=['A', 'B'])
```

5 Concatenation

Append DataFrames:

```
pd.concat([df1, df2])  # Stack vertically
pd.concat([df1, df2], axis=1)  # Combine horizontally
```

These transformation capabilities are what make Pandas truly powerful. The

```
groupby()
```

method in particular follows SQL-like semantics but with the flexibility of Python. When working with multiple DataFrames, the various join operations (

```
inner
```

,

```
outer
```

,

Advanced Data Analysis

Custom Functions

Apply functions across rows, columns or elements:

```
df.apply(lambda x: x*2)        # By row/column
df.applymap(lambda x: str(x))  # Element-wise
```

Time Series

Handle date-based data:

```
df.resample('M').mean()    # Monthly averages
df['col'].shift(periods=1)  # Lag values
```

Window Functions

Calculate moving statistics:

```
df.rolling(window=3).mean()    # Rolling average
df.expanding().sum()           # Cumulative sum
```

String Operations

Manipulate text data:

```
df['text'].str.contains('pattern')
df['text'].str.extract(r'(\d+)')
```

Advanced analysis techniques unlock deeper insights from your data. The

```
apply()
```

method lets you run arbitrary Python functions across your data, while specialized accessors like

```
.str
```

,

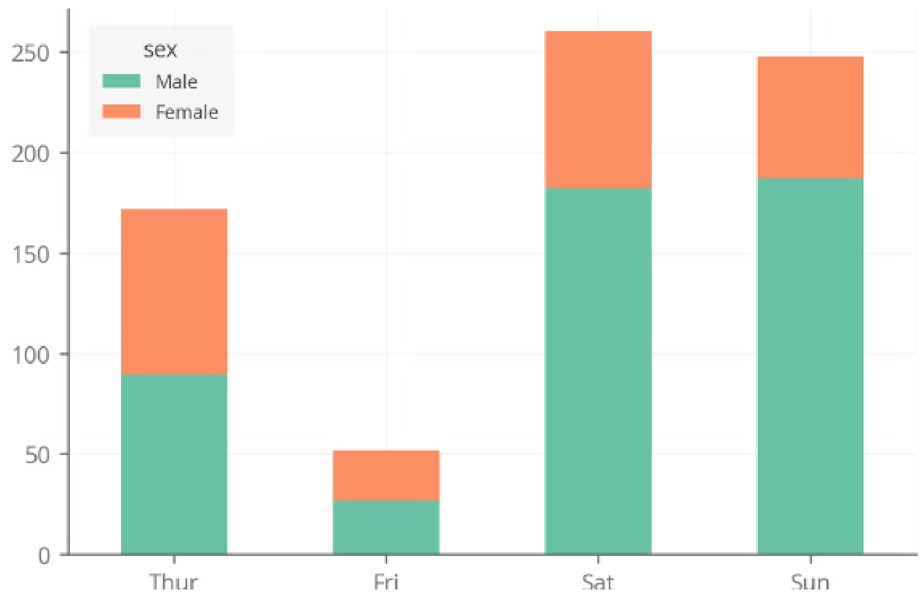
```
.dt
```

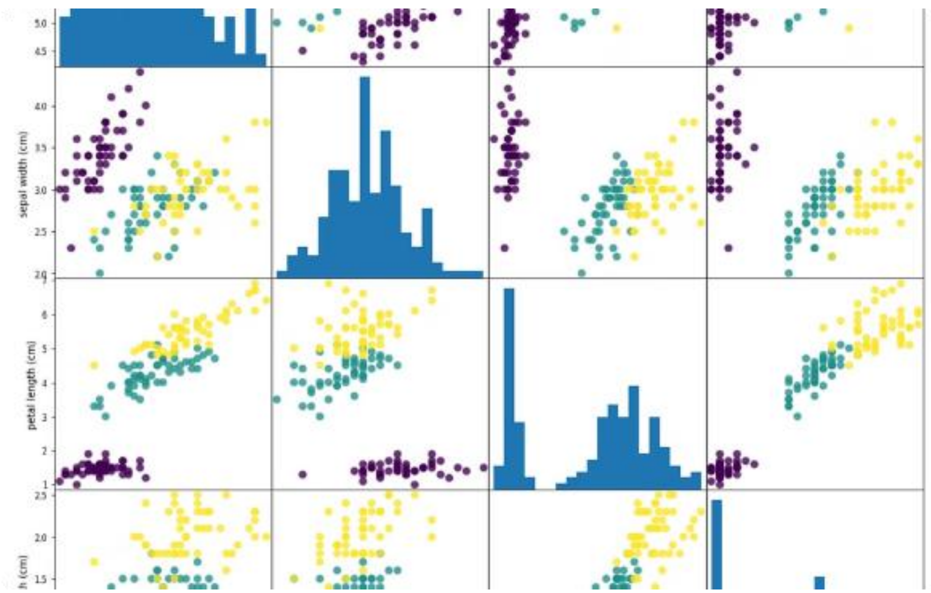
, and

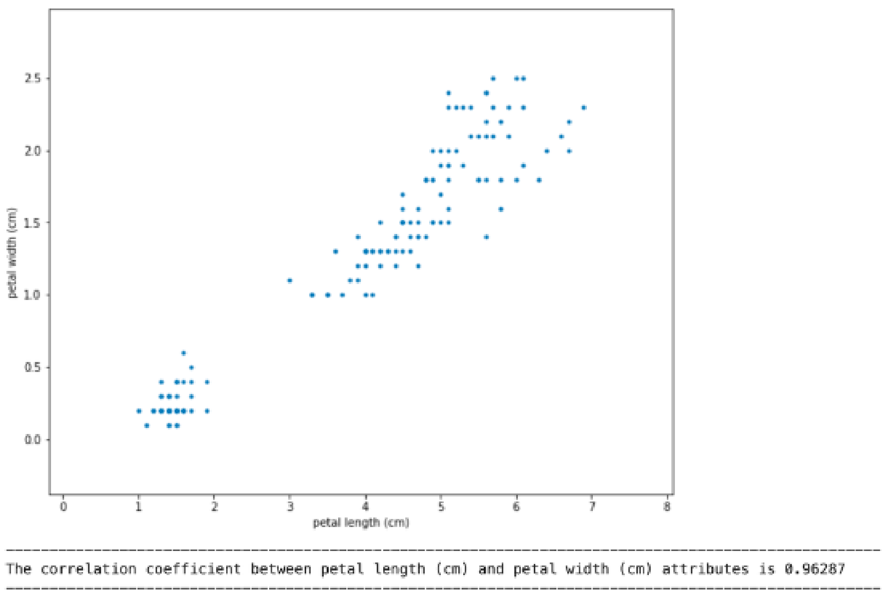
```
.cat
```

enable operations on strings, dates, and categorical data respectively. Time series functionality is particularly robust, with resampling, shifting, and seasonal decomposition capabilities.

Data Visualization with Pandas







Basic Plotting

Quick visualizations with built-in plotting:

```
df.plot()          # Line plot
df.plot.area()     # Area plot

# Bar chart
df.plot.bar(stacked=True)
```

Statistical Plots

Visualize distributions and summary statistics:

```
df.hist(bins=20)  # Histogram
df.plot.box()      # Box plot
df.plot.density()  # KDE plot
```

Relationships

Understand correlations and patterns:

```
df.plot.scatter(x='col1', y='col2')
pd.plotting.scatter_matrix(df)
df.corr().plot.heatmap()
```

Pandas integrates seamlessly with Matplotlib, allowing you to create informative visualizations with minimal code. The

```
plot
```

accessor provides a simple interface for common chart types. For more customization, you can access the underlying Matplotlib figure with

```
fig, ax = plt.subplots()
```

followed by

```
df.plot(ax=ax)
```

Pandas Supported File Types

CSV

Read comma-separated values into a DataFrame.

DataFrame.

```
import pandas as pd
df = pd.read_csv('file.csv')
```

Excel

Read and write Excel files (xls, xlsx).

```
import pandas as pd
df = pd.read_excel('file.xlsx',
sheet_name='Sheet1')
```

JSON

Read JSON strings into a DataFrame.

```
import pandas as pd
df = pd.read_json('data.json')
```

SQL

Execute SQL queries to read data from databases.

```
import pandas as pd
import sqlite3
conn = sqlite3.connect('database.db')
df = pd.read_sql_query("SELECT * FROM table", conn)
```

HTML

Parse HTML tables into a DataFrame.

```
import pandas as pd
df = pd.read_html('table.html')[0]
```

Pandas provides versatile tools for reading data from various file formats, simplifying data import and manipulation for analysis.

Exporting DataFrames with Pandas

Pandas also provides tools for exporting DataFrames to various file formats, making it easy to share and store your data.

CSV

Write a DataFrame to a comma-separated values file.

```
df.to_csv('file.csv', index=False)
```

Excel

Write a DataFrame to an Excel file (xls, xlsx).

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

JSON

Write a DataFrame to a JSON string.

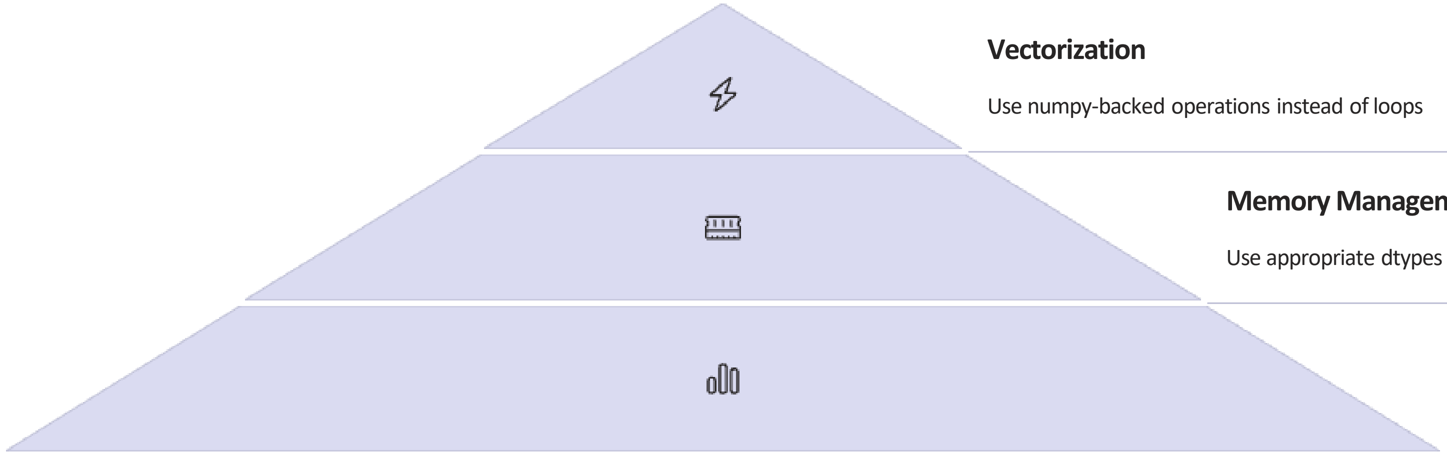
```
df.to_json('data.json')
```

SQL

Write a DataFrame to a SQL database.

```
import sqlite3
nconn = sqlite3.connect('database.db')
ndf.to_sql('table', conn, if_exists='replace', index=False)
```

Best Practices and Performance Optimization



Vectorization

Use numpy-backed operations instead of loops

Memory Management

Use appropriate dtypes and chunk processing

Efficient I/O

Optimize data reading and writing

For large datasets, performance matters. Always prefer vectorized operations (like

```
df['col'].str.contains('pattern')
```

) over iterating through rows. When reading large files, use

```
pd.read_csv('file.csv', chunksize=10000)
```

to process data in chunks, and consider

```
dtype
```

specifications to control memory usage.

Convert appropriate columns to categorical type with

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

when you have text columns with repeated values. This can dramatically reduce memory usage. Finally, use

```
df.info(memory_usage='deep')
```

to understand your DataFrame's memory footprint and identify opportunities for optimization.

Faster Alternatives to Pandas

While Pandas is a versatile library, other options offer significant performance improvements, especially for large datasets.

Polars

A DataFrame library implemented in Rust using Apache Arrow as its memory model. Its syntax is very similar to Pandas.

- Extremely fast, leveraging multi-core processing.
- Lazy evaluation for query optimization.
- Designed for large datasets.

datatable

`datatable` is a Python package dedicated to fast data manipulation. Its syntax is very similar to Pandas.

- Focus on speed and memory efficiency.
- Expressive syntax inspired by R's `data.table`.
- Optimized for large data, both in-memory and out-of-memory.

Dask

A flexible library for parallel computing in Python. Its syntax is very similar to Pandas.

- Enables parallel computations on larger-than-memory datasets.
- Integrates well with Pandas and other data science tools.
- Useful for distributed computing across multiple machines.

Consider these libraries when performance is critical for your data analysis workflows.