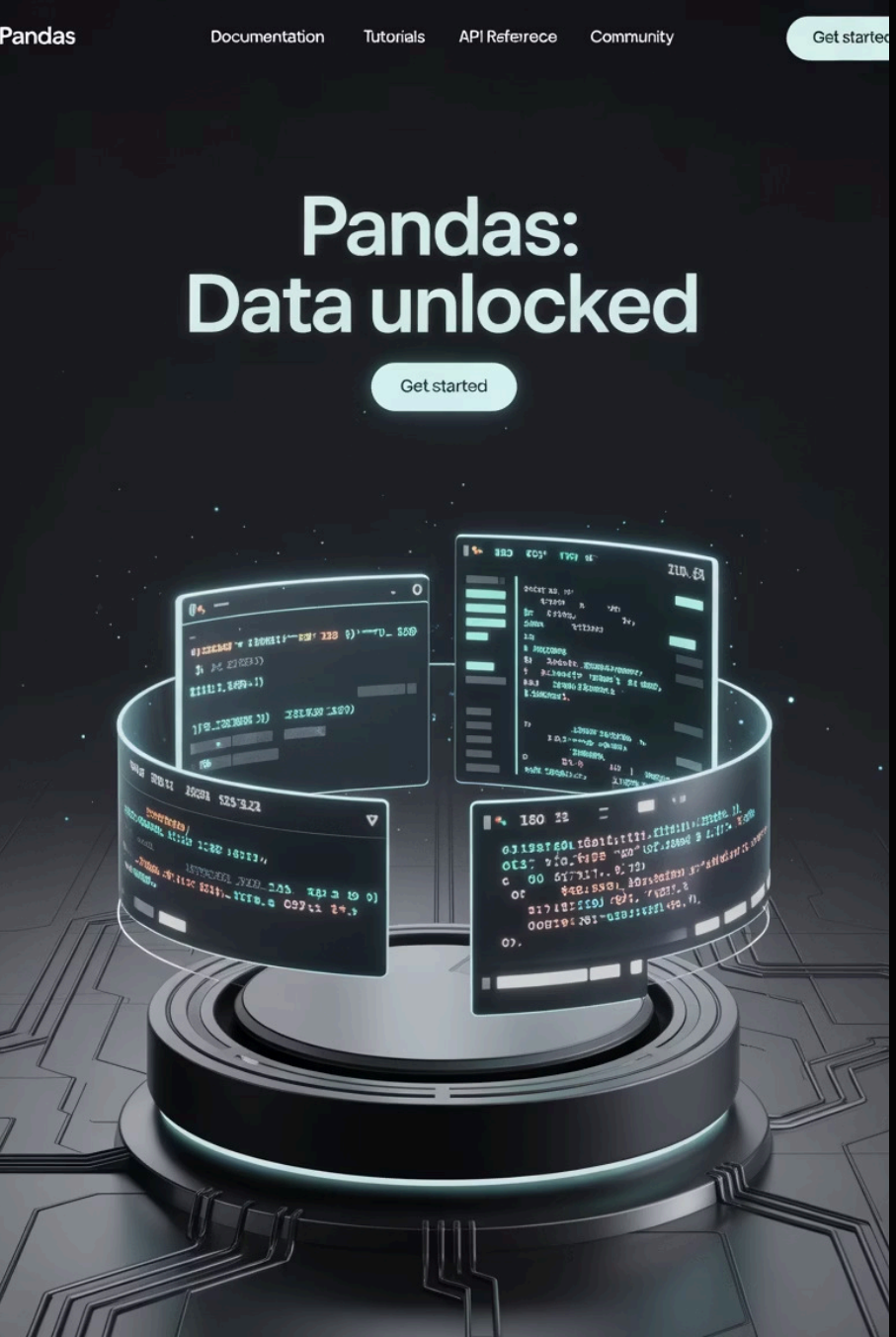


Getting Started with Pandas: Essential Data Analysis for Python

Chandrashekar Babu <https://www.chandrashekar.info> | <training@chandrashekar.info>

Welcome to this comprehensive guide on getting started with pandas, the powerful data manipulation library for Python. Whether you're analysing financial data, scientific measurements, or social statistics, pandas provides the tools you need to efficiently work with structured data. This presentation will walk you through the fundamentals of pandas, from basic operations to advanced analytics techniques, giving you the skills to transform, analyse, and visualise your data effectively.

Agenda

1. ## Introduction to Pandas

   History, core components, and key benefits

2. ## Working with DataFrames

   Loading, creating, and manipulating dataframes from various sources

3. ## Data Selection & Manipulation

   Indexing, filtering, and transforming data

4. ## Time Series Analysis

   Working with timestamps and time-based data

5. ## Analytics & Visualisation

   Statistical methods and plotting capabilities

Throughout this presentation, we'll provide practical examples with real-world applications, making these concepts immediately applicable to your data science projects.

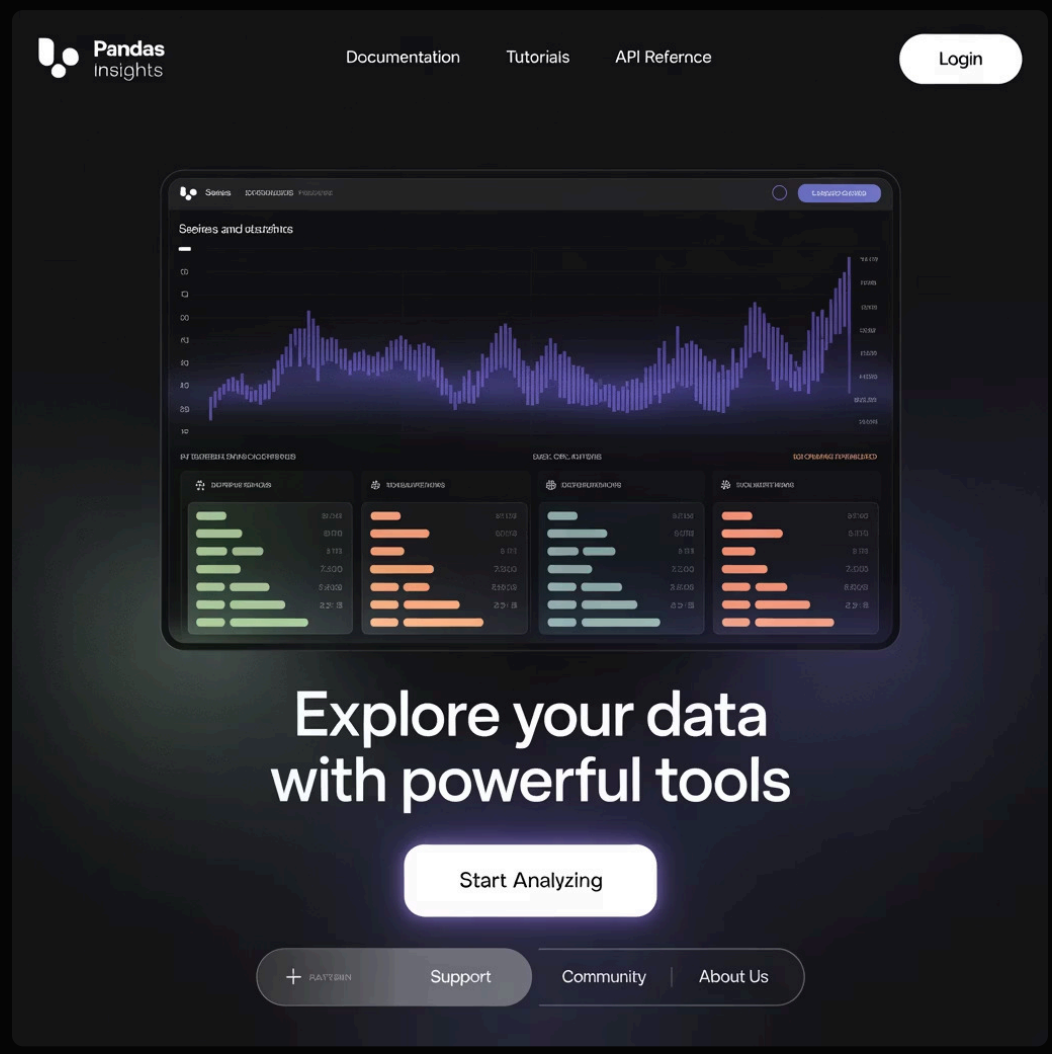
Introduction to Pandas

Pandas was created by Wes McKinney in 2008 while working at AQR Capital Management. The name "pandas" is derived from "panel data", an econometrics term for multidimensional structured datasets.

It has since become one of the most essential libraries in the Python data science ecosystem, working seamlessly with NumPy, Matplotlib, and scikit-learn to form the backbone of data analysis in Python.

Core Components:

- **Series**: One-dimensional labelled array capable of holding any data type
- **DataFrame**: Two-dimensional labelled data structure with columns of potentially different types
- **Panel**: (Deprecated in newer versions) Three-dimensional labelled array



Pandas is built on top of NumPy and makes heavy use of its array structure. However, it adds powerful indexing, data alignment capabilities, and tools for reading and writing data between in-memory data structures and different file formats.

The library is particularly well-suited for:

- Tabular data with heterogeneously-typed columns
- Time series data
- Arbitrary matrix data with row and column labels
- Observational/statistical data sets

Loading DataFrames from Various Sources

CSV Files

```
import pandas as pd

# Basic loading
df = pd.read_csv('data.csv')

# With options
df = pd.read_csv('data.csv',
                 sep=',',
                 header=0,
                 index_col=0,
                 parse_dates=['Date'])
```

SQL Databases

```
from sqlalchemy import
create_engine

# Create connection
engine =
create_engine('sqlite:///database.db')

# Query directly
df = pd.read_sql_query("SELECT *
FROM table", engine)

# Read entire table
df = pd.read_sql_table("table_name",
engine)
```

JSON Data

```
# From file
df = pd.read_json('data.json')

# From URL
import requests
response =
requests.get('https://api.example.com
/data')
df = pd.read_json(response.text)

# Normalize nested JSON
df =
pd.json_normalize(response.json()
['results'])
```

Pandas provides a consistent interface for loading data across different formats, making it easy to work with diverse data sources. The library handles many of the complexities of parsing, type inference, and memory management behind the scenes.

Creating DataFrames from Custom Sources

From Custom Log Parsers

```
import pandas as pd
import re

# Define pattern for log parsing
pattern = r'(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) \[(\w+)\] (.*)'

# Parse logs into dictionary
data = []
with open('application.log', 'r') as file:
    for line in file:
        match = re.match(pattern, line)
        if match:
            timestamp, level, message = match.groups()
            data.append({
                'timestamp': timestamp,
                'level': level,
                'message': message
            })

# Create DataFrame
log_df = pd.DataFrame(data)
log_df['timestamp'] = pd.to_datetime(log_df['timestamp'])
```

From Python Objects

```
# From dictionary
data = {
    'Name': ['John', 'Anna', 'Peter', 'Linda'],
    'Age': [28, 34, 29, 42],
    'City': ['London', 'Manchester', 'Liverpool', 'Leeds']
}
df = pd.DataFrame(data)

# From list of dictionaries
records = [
    {'Name': 'John', 'Age': 28, 'City': 'London'},
    {'Name': 'Anna', 'Age': 34, 'City': 'Manchester'},
]
df = pd.DataFrame.from_records(records)

# From NumPy array
import numpy as np
array = np.random.randn(4, 3)
df = pd.DataFrame(array, columns=['A', 'B', 'C'])
```

Handling Large Datasets with Chunks

When working with datasets that are too large to fit into memory, pandas provides functionality to process files in chunks. This is particularly useful for CSV files containing millions of rows.

```
import pandas as pd

# Define chunk size
chunk_size = 100000

# Create a reader object
reader = pd.read_csv('large_file.csv', chunksize=chunk_size)

# Process each chunk
total_rows = 0
age_sum = 0

for chunk in reader:
    # Perform operations on each chunk
    total_rows += len(chunk)
    age_sum += chunk['Age'].sum()

    # Filter and save specific records
    seniors = chunk[chunk['Age'] > 65]
    seniors.to_csv('seniors.csv', mode='a', header=False, index=False)

# Calculate final result
average_age = age_sum / total_rows
print(f"Average age across {total_rows} records: {average_age:.2f}")
```

Using this approach, you can process datasets that are many times larger than your available RAM. The key is to ensure that your operations are designed to work incrementally, aggregating results as you go rather than requiring the entire dataset to be in memory at once.

Working with Excel Spreadsheets

Pandas provides robust support for Excel files through the **openpyxl** and **xlrd** engines. This allows for reading from and writing to Excel workbooks with multiple sheets and complex formatting.

Reading Excel Files

```
import pandas as pd

# Read a specific sheet
df = pd.read_excel('financial_data.xlsx',
                   sheet_name='2023_Q1')

# Read multiple sheets (returns dict of DataFrames)
all_sheets = pd.read_excel('financial_data.xlsx',
                           sheet_name=None)

# Read specific columns
df = pd.read_excel('financial_data.xlsx',
                   usecols="A:C,F")

# Skip rows and set header
df = pd.read_excel('financial_data.xlsx',
                   skiprows=3,
                   header=1)
```

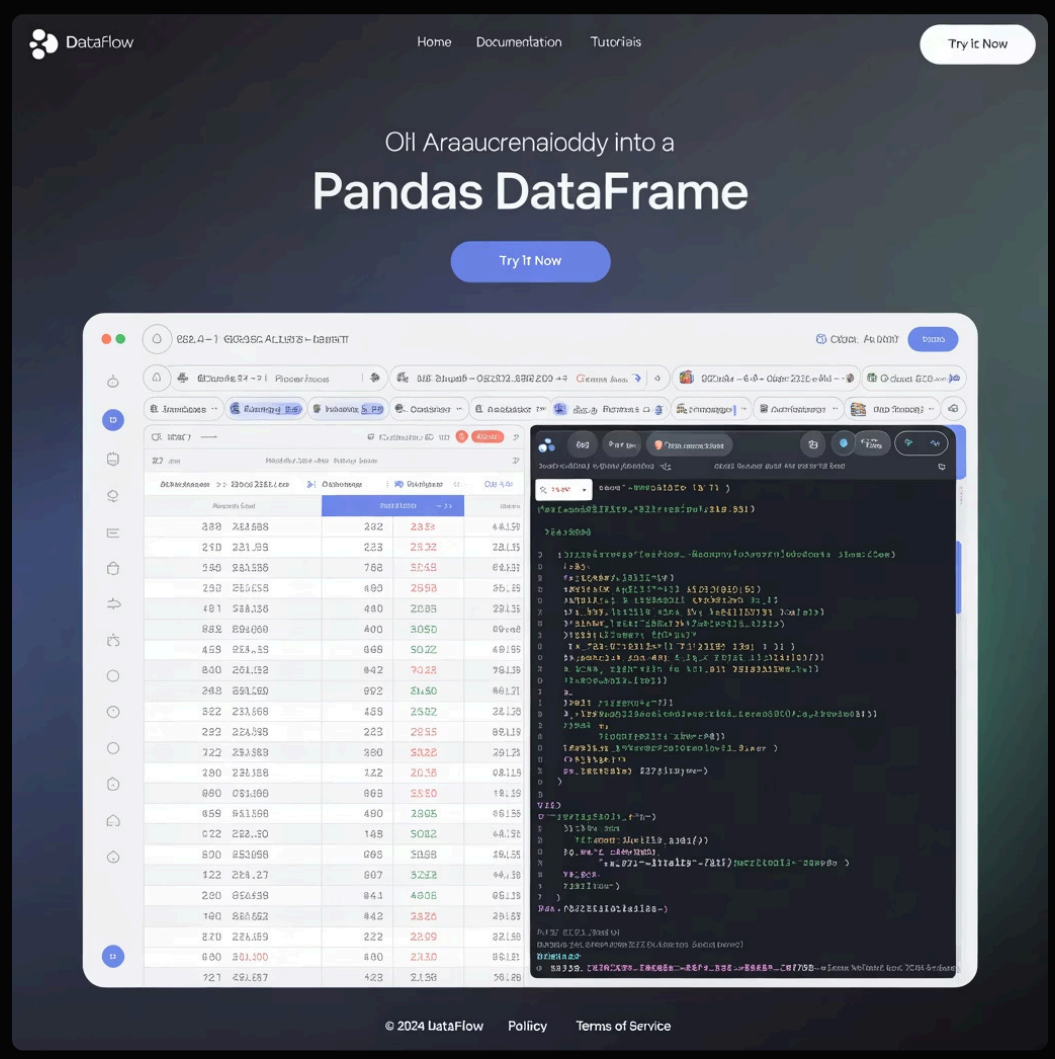
Writing to Excel

```
# Write to a new Excel file
df.to_excel('output.xlsx',
            sheet_name='Summary',
            index=False)

# Write multiple DataFrames to different sheets
with pd.ExcelWriter('output.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Q1', index=False)
    df2.to_excel(writer, sheet_name='Q2', index=False)
    df3.to_excel(writer, sheet_name='Summary', index=False)
```

Converting Excel to CSV

```
# Read Excel and save as CSV
for sheet_name, df in pd.read_excel('data.xlsx',
                                    sheet_name=None).items():
    df.to_csv(f"{sheet_name}.csv", index=False)
```



Pandas Series and DataFrame Fundamentals

1 Series: The Building Block

A Series is a one-dimensional array-like object containing an array of data and an associated array of data labels, called its index.

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
s2 = pd.Series({'a': 1, 'b': 2, 'c': 3})
```

2 DataFrame: The Data Table

A DataFrame is a 2-dimensional labelled data structure with columns of potentially different types.

```
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': pd.date_range('20230101', periods=4),
    'C': pd.Series(range(4), dtype='float64'),
    'D': pd.Categorical(["test", "train", "test", "train"]),
    'E': 'foo'
})
```

3 Basic Operations

Common methods and attributes for exploration:

- **df.head()** and **df.tail()**: View first/last n rows
- **df.info()**: Summary including dtypes and non-null values
- **df.describe()**: Statistical summary of numerical columns
- **df.shape**: Dimensions of the DataFrame (rows, columns)

4 Mathematical Operations

Pandas supports vectorised operations:

```
# Element-wise operations
df['A'] * 2
df['A'] + df['C']

# Aggregations
df.sum()
df.mean(axis=1) # Row-wise mean

# Apply custom functions
df.apply(lambda x: x.max() - x.min())
```

Indexing and Selecting Data

Label-based Selection with .loc

```
# Select rows by label
df.loc['2023-01-01']

# Select rows and columns by label
df.loc['2023-01-01':'2023-01-03', ['A', 'B']]

# Boolean indexing with .loc
df.loc[df['A'] > 0, ['B', 'C']]
```

Position-based Selection with .iloc

```
# Select by integer position
df.iloc[0]  # First row
df.iloc[0:3, 1:3]  # First 3 rows, 2nd & 3rd columns

# Select specific positions
df.iloc[[0, 2, 4], [0, 2]]
```

Boolean Indexing

```
# Filter rows based on condition
df[df['A'] > 0]

# Multiple conditions
df[(df['A'] > 0) & (df['B'] < 0)]

# Using query method
df.query('A > 0 and B < 0')

# isin for membership tests
df[df['A'].isin([1, 3, 5])]
```

Advanced Selection Techniques

```
# Select by callable
df.loc[lambda df: df['A'] > 0]

# Using where for conditional replacement
df.where(df > 0, -df)

# Using masks
mask = (df['A'] > 0) & (df['B'] < 0)
df.loc[mask, 'C'] = 0
```

Data Munging Techniques

Handling Missing Data

```
# Check for missing values
df.isna().sum()

# Fill missing values
df.fillna(0)  # With a constant
df.fillna(method='ffill')  # Forward fill
df.fillna({'A': 0, 'B': 1})  # Different by column

# Drop rows with any missing values
df.dropna()

# Drop rows where all columns are missing
df.dropna(how='all')
```

Data Transformation

```
# Apply function to each element
df.applymap(lambda x: x**2 if isinstance(x, (int, float)) else x)

# Apply function to each column/row
df.apply(np.cumsum)  # Cumulative sum
df.apply(lambda x: x.max() - x.min(), axis=1)

# Map values
df['category'] = df['category'].map({'A': 'Alpha', 'B': 'Beta'})

# Replace values
df.replace([1, 2], [10, 20])
```

Reshaping Data

```
# Pivot: reshape from long to wide format
pivot_df = df.pivot(index='date', columns='category',
values='value')

# Melt: reshape from wide to long format
melted_df = pd.melt(df, id_vars=['id'], value_vars=['A', 'B'])

# Stack and unstack
stacked = df.stack()  # Wide to long
unstacked = stacked.unstack()  # Long to wide
```

Combining Datasets

```
# Concatenate DataFrames
combined = pd.concat([df1, df2], axis=0) # Vertically
combined = pd.concat([df1, df2], axis=1) # Horizontally

# Merge (like SQL join)
merged = pd.merge(df1, df2, on='key')
merged = pd.merge(df1, df2, left_on='key1', right_on='key2')

# Join on index
joined = df1.join(df2, how='left')
```

Managing Timestamps and Time Series Data

Creating DateTime Objects

```
# Create a date range
dates = pd.date_range('20230101', periods=6)

# Convert string to datetime
df['date'] = pd.to_datetime(df['date_string'])

# Handle various formats
df['date'] = pd.to_datetime(df['date_string'],
                            format='%d/%m/%Y')

# Handle errors
df['date'] = pd.to_datetime(df['date_string'],
                            errors='coerce')  # Invalid -> NaT
```

DateTime Components

```
# Extract components
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['weekday'] = df['date'].dt.day_name()
df['quarter'] = df['date'].dt.quarter
```

Time Series Operations

```
# Resampling (change frequency)
daily = df.resample('D', on='date').mean()  # Daily
monthly = df.resample('M', on='date').sum()  # Monthly

# Shift values
df['previous_day'] = df['value'].shift(1)
df['next_day'] = df['value'].shift(-1)

# Rolling windows
df['7d_avg'] = df['value'].rolling(window=7).mean()
df['30d_std'] = df['value'].rolling(window=30).std()

# Time-based indexing
df.loc['2023-01':'2023-03']  # All dates in Q1 2023
df.loc['2023-01-15':'2023-02-15']  # Date range
```

Time series functionality is one of pandas' most powerful features, making it particularly suitable for financial analysis, IoT data processing, and any application involving temporal patterns and seasonality.

Basic Analytics Using Pandas

Statistical Functions

- **df.describe()**: Summary statistics for numerical columns
- **df.corr()**: Correlation between columns
- **df.cov()**: Covariance between columns
- **df.kurt()**: Kurtosis over requested axis
- **df.skew()**: Skewness over requested axis
- **df.quantile([0.25, 0.5, 0.75])**: Specific quantiles

Grouping and Aggregation

```
# Simple groupby with single aggregation
df.groupby('category').mean()

# Multiple aggregations
df.groupby('category').agg(['min', 'max', 'mean'])

# Different aggregations per column
df.groupby('category').agg({
    'A': 'sum',
    'B': 'mean',
    'C': ['min', 'max', 'count']
})

# Custom aggregation function
df.groupby('category').apply(lambda x: x.max() - x.min())
```

Advanced Analytics

```
# Pivot tables (like Excel)
pivot = pd.pivot_table(df,
                       values='value',
                       index=['date'],
                       columns=['category'],
                       aggfunc='sum',
                       fill_value=0)

# Cross-tabulation
pd.crosstab(df['category'], df['region'])

# Cumulative statistics
df['cumsum'] = df.groupby('category')['value'].cumsum()
df['cummax'] = df.groupby('category')['value'].cummax()

# Rolling operations by group
df['rolling_mean'] = df.groupby('category')['value'] \
                       .rolling(window=7).mean().reset_index(level=0,
drop=True)
```

Plotting and Visualisation with Pandas

Pandas provides a high-level interface to matplotlib, allowing you to create various plots directly from DataFrames and Series. This makes it easy to quickly visualise your data during analysis.

Basic Plotting

```
# Line plot (good for time series)
df.plot(figsize=(10, 6))
df['A'].plot()

# Bar plot
df.plot.bar()
df.plot.barh()  # Horizontal bars

# Histogram
df['A'].plot.hist(bins=20)

# Box plot
df.plot.box()

# Scatter plot
df.plot.scatter(x='A', y='B', alpha=0.5)
```

Advanced Visualisations

```
# Area plot
df.plot.area(stacked=True)

# Pie chart
df.plot.pie(subplots=True, figsize=(10,
6))

# Hexbin plot (for dense scatter
plots)
df.plot.hexbin(x='A', y='B',
gridsize=20)

# Density plot (KDE)
df['A'].plot.kde()

# Andrews Curves
from pandas.plotting import
andrews_curves
andrews_curves(df, 'category')
```

Customising Plots

```
# Styling options
df.plot(kind='bar',
 title='Sales by Region',
 color=['r', 'g', 'b', 'y'],
 rot=45, # Rotate x-labels
 grid=True,
 alpha=0.7)

# Multiple subplots
fig, axes = plt.subplots(nrows=2,
ncols=2, figsize=(12, 10))
df.plot(ax=axes[0,0])
df.plot.bar(ax=axes[0,1])
df.plot.box(ax=axes[1,0])
df.plot.hist(ax=axes[1,1])
plt.tight_layout()
```

Real-World Example: Customer Analysis

Let's put together several pandas techniques to analyze a customer dataset. This example demonstrates how you might apply pandas in a typical data science workflow.

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

# Load data
customers = pd.read_csv('customer_data.csv')

# Data cleaning
customers['registration_date'] = pd.to_datetime(customers['registration_date'])
customers['last_purchase'] = pd.to_datetime(customers['last_purchase'])
customers.dropna(subset=['customer_id', 'registration_date'], inplace=True)
customers['email'] = customers['email'].str.lower()

# Feature engineering
customers['account_age_days'] = (pd.Timestamp.now() - customers['registration_date']).dt.days
customers['days_since_purchase'] = (pd.Timestamp.now() - customers['last_purchase']).dt.days
customers['purchase_frequency'] = customers['total_purchases'] / customers['account_age_days'] * 30 # Monthly

# Segment customers
def get_segment(row):
 if row['total_spent'] > 1000 and row['purchase_frequency'] > 2:
   return 'High Value'
 elif row['days_since_purchase'] > 180:
   return 'Dormant'
 elif row['account_age_days'] < 30:
   return 'New'
 else:
   return 'Regular'

customers['segment'] = customers.apply(get_segment, axis=1)

# Analyze segments
segment_analysis = customers.groupby('segment').agg({
 'customer_id': 'count',
 'total_spent': ['mean', 'sum'],
 'purchase_frequency': 'mean',
 'days_since_purchase': 'mean'
})

# Visualize
segment_counts = customers['segment'].value_counts()
segment_counts.plot.pie(figsize=(10, 6), autopct='%1.1f%%', title='Customer Segments')
plt.show()

# Time series of new customers
monthly_registrations = customers.resample('M', on='registration_date')['customer_id'].count()
monthly_registrations.plot(figsize=(12, 6), title='New Customer Registrations by Month')
plt.show()
```

Key Takeaways and Next Steps

1

What You've Learned

- Loading data from various sources including CSV, Excel, SQL, and custom formats
- Handling large datasets with chunking techniques
- Manipulating and transforming data with pandas' powerful indexing capabilities
- Working with time series data and timestamps
- Performing statistical analysis and visualising results

2

Next Steps

- Integrate pandas with scikit-learn for machine learning pipelines
- Explore advanced time series functionality with pandas-datareader
- Optimize memory usage with categorical data types and sparse arrays
- Enhance visualisations with interactive libraries like Plotly and Bokeh
- Scale up with Dask or Modin for distributed pandas operations

3

Resources

- Official pandas documentation: **pandas.pydata.org/docs**
- Python for Data Analysis by Wes McKinney (creator of pandas)
- Community Q&A: **Stack Overflow - pandas**
- Tutorial notebooks: **GitHub - pandas tutorials**
- Cheat sheet: **pandas Cheat Sheet**

Pandas is a vast library with many capabilities. This presentation has covered the essentials, but there's always more to learn. As you work with your own datasets, you'll discover additional functionality that will make your data analysis workflows even more efficient. Happy analysing!