📊 Pandas Data Analysis

A Comprehensive Guide to Data Manipulation and Analysis with Python

📑 Table of Contents

1. Introduction & Library Setup

> *Pandas is a powerful Python library for data manipulation and analysis. It provides data structures like Series and DataFrame that make working with structured data intuitive and efficient.*

```
import pandas as pd import numpy as np
```

> 💡 **Tip:** Always import pandas as 'pd' and numpy as 'np' - these are standard conventions in the data science community.

2. Data Structures in Pandas

2.1 Series - One Dimensional Data

> *A Series is a one-dimensional labeled array, similar to a column in a spreadsheet or SQL table.*

```
s = pd.Series([10,11,12,13,14,15], name='Myseries',
index=['a','b','c','d','e','f'], dtype='float')
print(s)
```

```
a 10.0 b 11.0 c 12.0 d 13.0 e 14.0 f 15.0 Name: Myseries, dtype: float64
```

> 📝 **Note:** Series can be created from lists, NumPy arrays, dictionaries, and even other Series.

2.2 Series from Dictionary

```
s = pd.Series({'10':1, '11':2, '12':3, '13':4, '14':5,
'15':6}, dtype='float')
print(s)
```

2.3 Series Arithmetic with NaN Handling

> *When performing operations on Series with different indices, pandas automatically aligns data and fills missing values with NaN (Not a Number).*

```
s1 = pd.Series(12, index=[1,2,3,4,5,6])
s2 = pd.Series(12, index=[1,2,3,4])
print(s1 + s2) # No broadcasting error
```

```
1 24.0 2 24.0 3 24.0 4 24.0 5 NaN 6 NaN dtype: float64
```

2.4 DataFrame - Two Dimensional Data

> *A DataFrame is a two-dimensional labeled data structure with columns of potentially different types, resembling a spreadsheet or SQL table.*

```
data = {'col1:NO': [1,2,3,4], 'col2:Names':
['A','B','C','D']} df = pd.DataFrame(data) print(df)
```

```
col1:NO col2:Names 0 1 A 1 2 B 2 3 C 3 4 D
```

2.5 Creating DataFrame from Dictionary

```
data = {'a': [1,2,3], 'b': [4,5,6]}
var1 = pd.DataFrame(data, columns=['b']) # Select specific
column print(var1)
print("Access specific value:")
```

```
print(var1["b"][2]) # Output: 6
```

2.6 DataFrame from Lists

```
list_1 = [[1,2,3,4], [5,6,7,8], [9,10,11,12]]
df = pd.DataFrame(list_1, index=['s','p','d'])
print(df)
```

```
0 1 2 3 s 1 2 3 4 p 5 6 7 8 d 9 10 11 12
```

2.7 Combining Series into DataFrame

```
two_series = { 'op': pd.Series(['q','r','s','t']),
'ops': pd.Series(['q','r','s','t']) }
dataf = pd.DataFrame(two_series)
print(dataf)
```

3. Arithmetic Operations

> *Pandas supports vectorized operations on DataFrames, allowing efficient element-wise calculations across entire columns.*

```
data = {'A': [5,4,3,2,1], 'B': [6,5,4,7,6]}
df=pd.DataFrame(data)
# Addition
df['C'] = df['A'] + df['B']
# Subtraction
df['C'] = df['A'] - df['B']
# Multiplication
df['C'] = df['A'] * df['B']
# Division
df['C'] = df['A'] / df['B']
# Modulus
df['C'] = df['A'] % df['B']
# Exponentiation
df['C'] = df['A'] ** df['B']
```

> 💡 **Tip:** All arithmetic operations are vectorized, meaning they apply to entire columns at once without explicit loops.

3.1 Boolean Operations

```
# Create boolean column based on condition
df["python"] = df["B"] >= 6 # Check for non-null values
df["python"] = df["B"].notnull()
```

3.2 Statistical Summary

```
# Get statistical summary
df.describe()
# Get DataFrame information
df.info()
```

```
RangeIndex: 5 entries, 0 to 4 Data columns (total 4 columns): # Column Non-Null Count
Dtype --- ------ -------------- ----- 0 A 5 non-null int64 1 B 5 non-null int64 2 C 5
non-null int64 3 python 5 non-null bool dtypes: bool(1), int64(3)
```

4. Data Insertion & Deletion

4.1 Inserting Columns at Specific Position

> *The insert() method allows you to add a new column at a specific position in the DataFrame.*

```
var = pd.DataFrame({'A': [5,4,3,2,1], 'B':
[6,5,4,7,6]})
# Insert column at position 1
var.insert(1, 'python', var['A']) print(var)
```

```
A python B 0 5 5 6 1 4 4 5 2 3 3 4 3 2 2 7 4 1 1 6
```

4.2 Slicing and Creating New Columns

```
# Create column with sliced data (fills NaN for missing
indices)
var["python1"] = var['A'][:3]
print(var)
```

```
A python B python1 0 5 5 6 5.0 1 4 4 5 4.0 2 3 3 4 3.0 3 2 2 7 NaN 4 1 1 6 NaN
```

4.3 Deleting Columns

> *Two methods for deleting columns: pop() removes and returns the column, while del permanently removes it.*

```
# Method 1: pop() - removes and returns column
```

```
var1 = var.pop('B')
# Method 2: del - permanently removes column
del var['python1']
```

> ⚠ **Important:** The 'del' keyword permanently removes the column and cannot be undone. Use with caution!

5. Reading & Writing CSV Files

5.1 Creating and Writing CSV Files

```
data_set = { 's': [2,3,4,5,6,1], 'p': [2,3,5,6,8,9],
'd': [7,6,5,4,3,2] }
df = pd.DataFrame(data_set) # Write to CSV without
index
df.to_csv('test.csv', index=False) # Write with custom
headers
df.to_csv('test2.csv',index=False,header=['screen_time'
, 'study_time', 'sleeping_hrs'])
```

5.2 Reading CSV Files

```
# Basic read df = pd.read_csv("Salary_Dataset.csv")
# Read specific number of rows
df = pd.read_csv("Salary_Dataset.csv", nrows=1) # Read
specific columns
df =
pd.read_csv("Salary_Dataset.csv",usecols=['Salary'])
#Or by index: usecols=[0,3]
```

5.3 Advanced CSV Reading Options

```
# Skip specific rows df =
pd.read_csv("Salary_Dataset.csv", skiprows=[0,5]) # Set
column as index
df = pd.read_csv("Salary_Dataset.csv",
index_col=["country"]) # Use specific row as header
df = pd.read_csv("Salary_Dataset.csv", header=[3]) #
Add custom column names
```

```
df = pd.read_csv("Salary_Dataset.csv",
names=["col1","col2","col3","col4"])
```

> 📝 **Note:** When reading CSV files, specifying index_col=0 tells pandas that the first column contains the row indices.

5.4 Encoding Standards

```
# Common encoding standards: utf-8 (default) and Latin1
df = pd.read_csv("test.csv", encoding="Latin1",
index_col=0)
```

6. Data Cleaning & Missing Values

6.1 Identifying Missing Values

```
# Check for null values
data.isnull().sum()
```

```
Symbol 0 Security 0 Today-Volume 1 Avg-Volume 1 Change 1 dtype: int64
```

6.2 Handling Missing Values - Numerical Data

> *Convert to numeric and fill missing values with mean, median, or other statistical measures.*

```
# Convert to numeric (non-numeric becomes NaN)
data["Today-Volume"] = pd.to_numeric(data["Today-
Volume"], errors='coerce') # Fill NaN with mean
data["Today-Volume"].fillna(data["Today-
Volume"].mean(), inplace=True)
```

6.3 Conditional Replacement

```
# Replace negative values with NaN
data.loc[data["Change"] < 0, "Change"] = np.nan
# Fill NaN with mean
data["Change"].fillna(data["Change"].mean(),
inplace=True)
# Round to specific decimals
data["Today-Volume"] = data["Today-Volume"].round(2)
data["Change"] = data["Change"].round(4)
```

6.4 Forward and Backward Fill

Fill missing values using adjacent values - either forward fill (ffill) or backward fill (bfill).

```
# Forward fill - use previous value
data.ffill()
# Backward fill - use next value
data.bfill(axis=0)
```

6.5 Custom Fill Values

```
# Fill specific columns with specific values data.fillna({"Today-Volume":
"Java", "Change": "python"})
```

6.6 Removing Duplicates

```
# Keep first occurrence (default)
data = data.drop_duplicates(keep='first')
# Keep last occurrence
data = data.drop_duplicates(keep='last')
# Drop all duplicates
data = data.drop_duplicates(keep=False)
```

Tip: Always check the impact of your data cleaning operations using describe() and info() methods before and after.

7. Data Selection & Filtering

7.1 Selecting Single Column

```
# Using square brackets (returns Series)
single_column = df['col1'] # Using dot notation
single_column = df.col1
```

7.2 Selecting Multiple Columns

```
# Pass list of column names (returns DataFrame)
multiple_columns = df[['col1', 'col3']]
```

7.3 Using .loc[] - Label-based Selection

.loc[] allows selection by row and column labels.

```
# Select specific rows and columns
data.loc[[2,3], ["Symbol", "Security"]] # Select rows
with specific labels
columns_and_rows = df.loc[['a', 'c'], 'col1']
```

```
Symbol Security 2 MSFT Microsoft Corp. 3 AMZN Amazon.com Inc.
```

7.4 Using .iloc[] - Position-based Selection

.iloc[] allows selection by integer position.

```
# Select by position (row 0, column 1)
data.iloc[0, 1] # Returns: 'Apple Inc.'
# Select multiple columns by position
columns_by_index = df.iloc[:, [0, 2]]
# First and third columns
```

7.5 Filtering with Conditions

```
# Single condition
data.where(df["Avg-Volume"] < 40000)
# Multiple conditions with AND (&)
new_frame = df[(df["Avg-Volume"] > 50000) &
(df["Change"] > 0)]
# Multiple conditions with OR (|)
new_frame1 = df[(df["Avg-Volume"] > 50000) |
(df["Change"] > 0)]
```

7.6 Using filter() Method

```
# Filter columns containing specific string
filtered_columns = df.filter(like='prefix')
# Filter by data type
numeric_columns = df.select_dtypes(include=['number'])
```

⚠ **Important:** When filtering with multiple conditions, always use parentheses around each condition and use & (and) or | (or) operators, NOT Python's 'and' or 'or' keywords.

8. Merging & Concatenation

8.1 Merge Types (SQL-style Joins)

Pandas merge() function performs database-style joins between DataFrames.

```
df1 = pd.DataFrame({ 'key': ['A', 'B', 'C', 'D'],
'value1': [1, 2, 3, 4] })
df2 = pd.DataFrame({ 'key': ['A', 'B', 'E', 'F'],
'value2': [5, 6, 2, 6] })
```

Inner Join

```
# Returns only matching rows from both DataFrames
df_merge_inner = pd.merge(df1, df2, on="key",
how="inner")
```

```
key value1 value2 0 A 1 5 1 B 2 6
```

Left Join

```
# Returns all rows from left DataFrame, NaN for non-
matching right
df_merge_left = pd.merge(df1, df2, on="key",
how="left")
```

```
key value1 value2 0 A 1 5.0 1 B 2 6.0 2 C 3 NaN 3 D 4 NaN
```

Right Join

```
# Returns all rows from right DataFrame, NaN for non-
matching left
df_merge_right = pd.merge(df1, df2, on="key",
how="right")
```

Outer Join

```
# Returns all rows from both DataFrames, NaN where no
match
df_merge_outer = pd.merge(df1, df2, on="key",
how="outer")
```

Cross Join (Cartesian Product)

Combines every row from one DataFrame with every row from another.

```
df_merge_cross = pd.merge(df1, df2, how="cross")
```

```
key_x value1 key_y value2 0 A 1 A 5 1 A 1 B 6 2 A 1 E 2 3 A 1 F 6 ... (16 rows total)
```

8.2 Concatenation

Combine DataFrames along rows (axis=0) or columns (axis=1).

```
df_1 = pd.DataFrame({"A": [1,2,3,4], "B":
[9,10,11,12]})
df_2 = pd.DataFrame({"A": [5,6,7,8], "B":
[13,14,15,16]})
# Vertical concatenation (stack rows)
df_concat = pd.concat([df_1, df_2], axis=0)
# Horizontal concatenation (add columns)
df_concat1 = pd.concat([df_1, df_2], axis=1)
```

📝 **Note:** axis=0 means along rows (vertical), axis=1 means along columns (horizontal).

9. Grouping & Aggregation

9.1 Basic Aggregation Functions

```
df = pd.DataFrame({"A": [1,2,3,4,5], "B": [4,5,6,7,8]})
# Minimum value
df["A"].min() # Returns: 1
# Maximum value
df["A"].max() # Returns: 5
# Sum
df["A"].sum() # Returns: 15
# Count
df["A"].count() # Returns: 5
# Standard deviation
df["A"].std() # Returns: 1.5811...
# Variance
df["A"].var() # Returns: 2.5
```

9.2 GroupBy Operations

Group data by one or more columns and perform aggregation operations on each group.

```
df = pd.DataFrame({ 'Category': ['A', 'B', 'A', 'B',
'A', 'B'], 'Value': [10, 20, 30, 40, 50, 60] })
# Group by Category and sum
df.groupby('Category')['Value'].sum()
```

```
# Group by Category and calculate mean
df.groupby('Category')['Value'].mean()
# Group by Category and count
df.groupby('Category')['Value'].count()
```

```
Category A 90 B 120 Name: Value, dtype: int64 Category A 30.0 B 40.0 Name: Value,
dtype: float64
```

9.3 Multiple Aggregations

```
# Apply multiple aggregations to DataFrame
df1 = pd.DataFrame({"A": [1,2,3,4,5,6], "B":
[3,4,5,6,7,8]})
result = df1.agg(['sum', 'mean'])
```

```
A B sum 21.0 33.0 mean 3.5 5.5
```

9.4 Column-Specific Aggregations

```
# Different aggregations for different columns
result = df.agg({'Value': ['sum', 'max']})
```

```
Value sum 210 max 60
```

> 💡 **Tip:** Use groupby() for split-apply-combine operations - it's one of the most powerful features in pandas for data analysis.

10. Advanced Data Transformation

10.1 Creating Calculated Columns

```
# Add Bonus column (10% of Salary)
data["Bonus"] = data["Salary"] * 0.1
# Insert column at specific position
data.insert(3, "Bonus_Today-Volume", data["Today-
Volume"] * 0.1)
```

10.2 Transform with GroupBy

> *Apply transformations that maintain the original DataFrame shape while using group statistics.*

```
# Update Salary based on department mean (with 5%
increase)
```

```
data['Salary']=data.groupby('Department')['Salary']
.transform('mean') * 1.05
```

10.3 Apply Custom Functions

```
# Create age group using lambda function
data["Age_group"] = data["Age"].apply( lambda age:
"Junior" if age < 35 else "Senior" )
```

10.4 Pivot Tables

> *Reshape data to create summary tables similar to Excel pivot tables.*

```
# Create pivot table pivot = data.pivot_table(
values='Salary', index='Department',
columns='Age_group', aggfunc='mean' )
```

```
Age_group Junior Senior Department Engineering 57557.5 NaN Finance 69825.0 69825.0 HR
56350.0 NaN Marketing 63787.5 63787.5 Sales 74550.0 74550.0
```

10.5 Data Normalization

> *Scale data to a specific range (0 to 1) using min-max normalization.*

```
# Normalize Salary column to 0-1 range
data["Normalized Salary"] = ( (data["Salary"] -
data["Salary"].min()) / (data["Salary"].max
```