MindSparXs
Empowering your future
Training Division of Ctronics InfoTech Pvt. Ltd.

MindSparXs Courseware

A Comprehensive Guide to Pandas

Comprehensive Guide to Pandas

1. Introduction to Pandas

Pandas is widely known as the "Python Data Analysis Library." It is a highly popular and crucial Python library for data processing, used extensively in data science, machine learning, and deep learning applications. Developed by Wes McKinney, Pandas is a free and open-source library that provides high-level data manipulation and analysis tools.

Pandas is built on the NumPy library, leveraging its efficient array processing capabilities. The core data structures in Pandas are **Series** and **DataFrames**, which are fundamental for managing and manipulating data in tabular and sequential formats.

2. Core Data Structures: Series and DataFrames

Pandas revolves around two primary data structures:

Series: A one-dimensional labeled array capable of holding any data type (integers, strings, floats, Python objects, etc.). It is similar to a dictionary where keys are labels (index) and values are the data elements, often stored as a list.

DataFrames: A two-dimensional labeled data structure with columns of potentially different types. It is like a spreadsheet or SQL table, or a dictionary of Series objects. DataFrames are invaluable for organizing and analyzing tabular data.

While NumPy focuses on N-Dimensional array processing, Pandas specializes in the management and manipulation of Series and DataFrames.

3. Installation

Pandas can be easily installed using Python's package installer, pip:

C:\> pip install pandas

Alternatively, if you are using the Anaconda distribution, Pandas is typically pre-installed along with many other essential data science libraries.

4. Creating Pandas Series

A Pandas Series is a one-dimensional array-like structure. It has an index associated with each element.

4.1. From Python Lists or Tuples

When created from a list or tuple, Series automatically gets a default numeric index starting from 0.

Values: The actual data contained in the Series.

Index: Labels for each element, defaulting to integers (0, 1, 2, ...).

Name: An optional name for the Series.

```
import pandas as pd
from pandas import Series
# Create a Series from a list
result_series = Series( [50, 90, 76, 45] )
result_series.name = 'Result'
# Assigning a nameprint(result_series)
print(result_series.values)
# Prints only the values
print(result_series.index)
# Prints the index
print(result_series.name)    # Prints the name of the series
```

4.2. Creating Series with Custom Index

You can specify a custom index when creating a Series:

```
import pandas as pd
from pandas import Series
# Create a Series with a custom index
num_series = Series( [15, 24, 33, 52, 16], index=['a','b','c','d','e'] )
num_series.name='Nums'
```

```
print(num_series)
```

Note: While changing the index for display is useful, it might complicate certain data processing operations. It's generally preferred to use default numeric indices unless a specific logical index is required.

4.3. Creating Series from Python Dictionaries

Creating a Series from a Python dictionary uses the dictionary's keys as the Series index.

```
import pandas as pd
from pandas import Series
# Python Dictionary
salary_dict = { 'Rajesh' : 50000, 'Amit': 75000, 'Pooja': 85000, 'Anita': 40000, 'Vijay': 67000 }
# Create Series from dictionary
employee_salary_series = Series(salary_dict)
employee_salary_series.name = 'EmployeeSalaries'
print(employee_salary_series)
print(employee_salary_series.index) # Keys of the dict become the index
```

5. Creating Pandas DataFrames

DataFrames are the most popular and versatile data structure in Pandas, essential for most data science projects.

5.1. Creating a DataFrame from a Dictionary of Lists/Series

A DataFrame is essentially a collection of Series. The most common way to create one is by passing a dictionary where keys are column names and values are lists or Series representing the data for each column.

```
import pandas as pd
from pandas import DataFrame
# Dictionary of lists (each list represents a column)
states_data = {    'State': ['Gujarat', 'Tamil Nadu', 'Andhra', 'Karnataka', 'Kerala'],    'Population': [ 36, 44, 67, 89, 34],    'Language': ['Gujarati', 'Tamil', 'Telugu', 'Kannada', 'Malayalam']}
# Create DataFrame
```

```
df_states = DataFrame(states_data)
print(df_states)
```

5.2. DataFrame Properties and Attributes

DataFrames have numerous useful attributes and methods for inspecting their structure and content:

shape: Returns a tuple representing the dimensions (rows, columns).

dtypes: Shows the data type of each column.

info(): Provides a concise summary of the DataFrame, including index dtype, column dtypes, non-null values, and memory usage.

columns: Returns an Index object containing the column labels.

size: Returns the total number of elements in the DataFrame (rows * columns).

index: Returns the index labels for the rows.

axes: Returns a list of the row and column indices.

ndim: Returns the number of dimensions (always 2 for DataFrame).

values: Returns the DataFrame's data as a NumPy array.

T (Transpose): Returns the transposed DataFrame (rows become columns and vice-versa).

6. Accessing Data in DataFrames

Efficiently accessing data is crucial for analysis.

6.1. Accessing Columns

You can access columns using bracket notation or dot notation (if the column name is a valid Python identifier and doesn't conflict with DataFrame methods).

Using brackets: df['ColumnName'] (preferred, handles spaces and special characters)

Using dot notation: df.ColumnName (convenient but less flexible)

```
# Assuming df_states from previous example
print(df_states['State'])
print(df_states.Population)
```

6.2. Accessing Rows

Rows can be accessed using slicing or specific indexing methods.

Slicing: df[start:stop] returns a DataFrame containing rows from the start index up to (but not including) the stop index.

Row Indexing (via .loc **or** .iloc**):** For more explicit label-based or integer-position-based indexing, .loc and .iloc are recommended, though not detailed in the original notes. The simple slicing df[1:3] works for integer-based row selection.

```
# Accessing the first two rows
print(df_states[0:2])
# Accessing rows from index 1 up to (but not including) 3
print(df_states[1:3])
```

6.3. Accessing Specific Elements

Elements can be accessed by combining row and column selection, typically using .loc or .iloc, or by first selecting a column and then an element from that Series.

```
# Accessing the 'Language' of the first state (index 0)
print(df_states.loc[0, 'Language'])
 # or df_states['Language'][0]
```

7. Manipulating DataFrames

7.1. Adding Columns

New columns can be added by assigning a list, Series, or a scalar value to a new column name.

```
# Add a 'Capital' column
df_states['Capital'] = ['Gandhinagar', 'Chennai', 'Amaravati', 'Bengaluru', 'Thiruvananthapuram']
print(df_states)
```

7.2. Reordering Columns

You can change the column sequence when creating a DataFrame or by selecting columns in a new order.

```
# Reorder columns
df_reordered = DataFrame(states_data, columns=['Language', 'State', 'Population'])
print(df_reordered)
```

7.3. Transposing a DataFrame

Use the .T attribute to transpose a DataFrame.

```
# Transpose the states DataFrameprint(df_states.T)
```

7.4. Dropping Data

The drop() function is used to remove rows or columns.

Dropping Columns: Specify the column name and axis=1.

Dropping Rows: Specify the row index and axis=0 (or omit axis as it defaults to 0).

inplace=True: Modifies the DataFrame directly without returning a new one.

```
# Drop the 'Capital' column (axis=1)
df_dropped_col = df_states.drop('Capital', axis=1)
print(df_dropped_col)
# Drop the first row (index 0)
df_dropped_row = df_states.drop(0)
print(df_dropped_row)
# Drop 'Capital' column permanently#
 df_states.drop('Capital', axis=1, inplace=True)
```

8. Statistical Functions

Pandas offers comprehensive statistical functions, many of which are built on NumPy's capabilities.

8.1. The describe() Function

This is a very useful function that provides a summary of descriptive statistics for numeric columns in a DataFrame.

count: Number of non-null observations.

mean: Average value.

std: Standard deviation (measure of data dispersion).

min: Minimum value.

25% **(Q1)**: First quartile.

50% **(Q2/Median)**: Median.

75% **(Q3)**: Third quartile.

max: Maximum value.

Example using describe() on df_states (Population column is numeric)

print(df_states.describe())

8.2. Individual Statistical Functions

Pandas also provides direct access to individual statistical methods:

Numeric Data: count(), sum(), mean(), median(), std(), min(), max().

Numeric and String Data: sum() and cumsum() can also operate on string columns (for concatenation).

String Data: mode() finds the most frequent value (most common term).

Note: The mode() function returns the most frequent value(s) and can be used for both numeric and string data.

Example: Sum of Population column

print("Sum of Population:", df_states['Population'].sum())

Example: Mean of Population

print("Mean Population:", df_states['Population'].mean())

Example: Mode of Language column

print("Mode Language:", df_states['Language'].mode())

9. Reading Data from Files

A fundamental capability of Pandas is reading data from various file formats into DataFrames.

Common File Types: .csv (Comma Separated Values - most popular), .txt (Text files, often space-separated), .xls/.xlsx (Excel files), .json (JavaScript Object Notation).

Reading Functions:

pd.read_csv('filepath.csv')

pd.read_excel('filepath.xls') or pd.read_excel('filepath.xlsx')

pd.read_json('filepath.json')

pd.read_table('filepath.txt') (older, read_csv is often used for text files too)

Viewing Data:

df.head(n): Displays the first n rows.

df.tail(n): Displays the last n rows.

Once data is loaded, you can use all the DataFrame properties and methods (shape, dtypes, columns, etc.) to explore it.

Example: Reading data from a CSV file

Make sure 'your_dataset.csv' exists in the specified path#

data = pd.read_csv('path/to/your_dataset.csv')

df_from_file = DataFrame(data)

print(df_from_file.head())

print(df_from_file.shape)

10. Additional DataFrame Operations

10.1. Selecting Specific Columns to Create a DataFrame

You can create a new DataFrame containing only a subset of columns from an existing one.

```
# Create a new DataFrame with only 'State' and 'Language'

df_subset = DataFrame(states_data, columns=['State', 'Language'])

print(df_subset)
```

10.2. Indexing and Slicing DataFrames

As mentioned in Section 6, Pandas provides flexible ways to select data:

Columns: df['ColName'] or df.ColName

Rows: df[start:stop] or using .loc/.iloc for more control.

Combined: df.loc[row_indexer, column_indexer]

10.3. Changing Column Sequence (Shuffling)

Columns can be reordered by passing a new list of column names to the DataFrame constructor or by re-indexing.

```
# Reorder columns during creation

df_shuffled = DataFrame(states_data, columns=['Language', 'State', 'Population'])

print(df_shuffled)
```