Pandas

Notes

Pandas Full Notes

Introduction

Pandas is an open-source Python library for data analysis and manipulation. It provides data structures like Series and DataFrame for handling structured data.

Installation

```
pip install pandas
```

Key Data Structures

1. Series

A one-dimensional labeled array capable of holding any data type (integer, float, string, etc.).

Created using:

```
import pandas as pd
s = pd.Series([1, 2, 3, 4])
```

2. DataFrame

A two-dimensional labeled data structure with columns of potentially different types.

Created using:

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

Data Input/Output

1. CSV

Reading:

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

Writing:

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

2. Excel

Reading:

```
df = pd.read_excel('file.xlsx')
```

Writing:

```
df.to_excel('output.xlsx', index=False)
```

3. JSON

Reading:

```
df = pd.read_json('file.json')
```

Writing:

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

4. Parquet

Reading:

```
df = pd.read_parquet('file.parquet')
```

Writing:

```
df.to_parquet('output.parquet')
```

Data Exploration

View top rows:

```
df.head()
```

View bottom rows:

df.tail()

Summary statistics:

df.describe()

Column names:

df.columns

Data types:

df.dtypes

Check for missing values:

df.isnull().sum()

Data Selection

1. Select a column:

df['column_name']

2. Select multiple columns:

```
df[['col1', 'col2']]
```

3. Select rows by index:

```
df.iloc[0:5]
```

4. Select rows by condition:

```
df[df['column_name'] > 10]
```

Data Manipulation

1. Add a new column:

```
df['new_col'] = df['col1'] + df['col2']
```

2. Drop columns:

```
df = df.drop(['col1', 'col2'], axis=1)
```

3. Rename columns:

```
df = df.rename(columns={'old_name': 'new_name'})
```

4. Replace values:

```
df['column_name'] = df['column_name'].replace(old_value, new_value)
```

Grouping and Aggregation

1. Group data:

```
grouped = df.groupby('column_name')
```

2. Aggregate data:

```
grouped['col1'].sum()
```

3. Multi-level grouping:

```
df.groupby(['col1', 'col2']).mean()
```

Sorting

1. Sort by a column:

```
df.sort_values('column_name', ascending=True)
```

2. Sort by multiple columns:

```
df.sort_values(['col1', 'col2'], ascending=[True, False])
```

Missing Data Handling

1. Fill missing values:

```
df['column_name'].fillna(value, inplace=True)
```

2. Drop rows with missing values:

```
df.dropna(inplace=True)
```

Merging and Joining

1. Concatenate DataFrames:

```
pd.concat([df1, df2], axis=0)
```

2. Merge DataFrames:

```
pd.merge(df1, df2, on='key_column')
```

3. Join DataFrames:

```
df1.join(df2, on='key_column')
```

Pivot Tables

1. Create pivot table:

```
df.pivot_table(values='col1', index='col2', columns='col3',
aggfunc='mean')
```

Apply Functions

1. Apply custom function:

```
df['column_name'].apply(lambda x: x * 2)
```

2. Apply row-wise or column-wise:

```
df.apply(lambda row: row['col1'] + row['col2'], axis=1)
```

Time Series

1. Convert column to datetime:

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

2. Extract specific date parts:

```
df['year'] = df['date_column'].dt.year
```

Visualization with Pandas

1. Line plot:

```
df.plot(x='col1', y='col2', kind='line')
```

2. Bar plot:

```
df.plot(kind='bar')
```

3. Histogram:

```
df['col1'].plot(kind='hist')
```

Performance Optimization

1. Use efficient data types:

```
df['int_column'] = df['int_column'].astype('int32')
```

2. Use vectorized operations:

```
df['new_col'] = df['col1'] + df['col2']
```

100 One-Liner Pandas Codes

1. Import pandas:

```
import pandas as pd
```

2. Create a Series:

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

3. Create a DataFrame from a dictionary:

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

4. Read a CSV file:

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

5. Write a DataFrame to a CSV file:

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

6. View the first 5 rows:

```
df.head()
```

7. View the last 5 rows:

```
df.tail()
```

8. Get DataFrame info:

```
df.info()
```

9. Get summary statistics:

```
df.describe()
```

10. Rename a column:

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

11. Select a column:

```
df['col']
```

12. Select multiple columns:

```
df[['col1', 'col2']]
```

13. Add a new column:

```
df['new_col'] = df['col1'] + df['col2']
```

14. Drop a column:

```
df.drop('col', axis=1, inplace=True)
```

15. Check for null values:

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

16. Fill null values:

```
df.fillna(0, inplace=True)
```

17. Drop rows with null values:

```
df.dropna(inplace=True)
```

18. Filter rows by condition:

```
df[df['col'] > 10]
```

19. Sort by a column:

```
df.sort_values('col', ascending=False)
```

20. Get unique values in a column:

```
df['col'].unique()
```

21. Count unique values:

```
df['col'].nunique()
```

22. Group by a column:

```
df.groupby('col').mean()
```

23. Reset the index:

```
df.reset_index(drop=True, inplace=True)
```

24. Set a column as index:

```
df.set_index('col', inplace=True)
```

25. Concatenate DataFrames:

```
pd.concat([df1, df2])
```

26. Merge DataFrames:

```
pd.merge(df1, df2, on='key')
```

27. Get column data types:

```
df.dtypes
```

28. Change data type of a column:

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

29. Apply a function to a column:

```
df['col'] = df['col'].apply(lambda x: x * 2)
```

30. Apply a function to rows:

```
df.apply(lambda row: row.sum(), axis=1)
```

31. Get the index of a DataFrame:

```
df.index
```

32. Get column names:

```
df.columns
```

33. Transpose the DataFrame:

```
df.T
```

34. Drop duplicates:

```
df.drop_duplicates(inplace=True)
```

35. Check if a column has duplicates:

```
df.duplicated().any()
```

36. Pivot a DataFrame:

```
df.pivot(index='col1', columns='col2', values='col3')
```

37. Melt a DataFrame:

```
df.melt(id_vars=['col1'], value_vars=['col2'])
```

38. Create a crosstab:

```
pd.crosstab(df['col1'], df['col2'])
```

39. Calculate cumulative sum:

```
df['col'].cumsum()
```

40. Calculate cumulative product:

```
df['col'].cumprod()
```

41. Get rolling mean:

```
df['col'].rolling(3).mean()
```

42. Get column max value:

```
df['col'].max()
```

43. Get column min value:

```
df['col'].min()
```

44. Get column mean:

```
df['col'].mean()
```

45. Get column median:

```
df['col'].median()
```

46. Get column standard deviation:

```
df['col'].std()
```

47. Count non-NA values in a column:

```
df['col'].count()
```

48. Drop rows with specific condition:

```
df = df[df['col'] != 0]
```

49. Sample n rows:

```
df.sample(5)
```

50. Generate random DataFrame:

```
pd.DataFrame(np.random.rand(3, 3))
```

51. Calculate correlation:

```
df.corr()
```

52. Calculate covariance:

```
df.cov()
```

53. Add prefix to column names:

```
df.add_prefix('prefix_')
```

54. Add suffix to column names:

```
df.add_suffix('_suffix')
```

55. Get row with maximum value in column:

```
df.loc[df['col'].idxmax()]
```

56. Get row with minimum value in column:

```
df.loc[df['col'].idxmin()]
```

57. Create a DataFrame from a list:

```
pd.DataFrame([1, 2, 3], columns=['col'])
```

58. Filter rows by multiple conditions:

```
df[(df['col1'] > 0) & (df['col2'] < 10)]
```

59. Create a datetime column:

```
df['date'] = pd.to_datetime(df['date'])
```

60. Extract year from datetime:

```
df['year'] = df['date'].dt.year
```

61. Extract month from datetime:

```
df['month'] = df['date'].dt.month
```

62. Extract day from datetime:

```
df['day'] = df['date'].dt.day
```

63. Get the difference between dates:

```
df['days_diff'] = (df['date2'] - df['date1']).dt.days
```

64. Check for missing data:

```
df.isna()
```

65. Replace missing values:

```
df['col'].replace(np.nan, 0, inplace=True)
```

66. Get data types of all columns:

```
df.dtypes
```

67. Calculate column-wise sum:

```
df.sum(axis=0)
```

68. Calculate row-wise sum:

```
df.sum(axis=1)
```

69. Read JSON file:

```
df = pd.read_json('file.json')
```

70. Write JSON file:

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

71. Append a new row:

```
df = df.append({'col1': 1, 'col2': 2}, ignore_index=True)
```

72. Extract text from a string column:

```
df['new_col'] = df['col'].str.extract(r'(\d+)')
```

73. Check for string pattern:

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

74. Replace strings in a column:

```
df['col'] = df['col'].str.replace('old', 'new')
```

75. Strip whitespace from strings:

```
df['col'] = df['col'].str.strip()
```

76. Capitalize strings:

```
df['col'] = df['col'].str.capitalize()
```

77. Lowercase strings:

```
df['col'] = df['col'].str.lower()
```

78. Uppercase strings:

```
df['col'] = df['col'].str.upper()
```

79. Convert column to category:

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

80. Rename index:

```
df.rename_axis('index_name', inplace=True)
```

81. Explode a list column:

```
df.explode('col')
```

82. Check memory usage:

```
df.memory_usage(deep=True)
```

83. Get column value counts:

```
df['col'].value_counts()
```

84. Select rows by index range:

```
df.iloc[2:5]
```

85. Select specific cells:

```
df.at[0, 'col']
```

86. Convert column to numeric:

```
pd.to_numeric(df['col'], errors='coerce')
```

87. Create a DataFrame of random integers:

```
pd.DataFrame(np.random.randint(0, 100, size=(5, 5)),
columns=list('ABCDE'))
```

88. Resample time-series data:

```
df.resample('M').mean()
```

89. Perform string split:

```
df['col'].str.split(',')
```

90. Filter DataFrame by regex:

```
df[df['col'].str.contains(r'\d+')]
```

91. Create a range of dates:

```
pd.date_range(start='2023-01-01', periods=10, freq='D')
```

92. Get the shape of the DataFrame:

```
df.shape
```

93. Filter DataFrame by index values:

```
df.loc[df.index > 5]
```

94. Drop a row by index:

```
df.drop(index=3, inplace=True)
```

95. Rename DataFrame index:

```
df.index = ['row1', 'row2']
```

96. Filter numeric columns:

```
df.select_dtypes(include='number')
```

97. Filter categorical columns:

```
df.select_dtypes(include='category')
```

98. Convert DataFrame to NumPy array:

```
df.to_numpy()
```

99. Create a MultiIndex DataFrame:

```
pd.MultiIndex.from_tuples([('A', 1), ('B', 2)])
```

100. Plot a DataFrame: python df.plot(kind='line')

The most important points about Pandas, covering various concepts and operations:

1. Pandas Library: Pandas is a powerful open-source library for data manipulation and analysis in Python, built on top of NumPy.

2. Series: A one-dimensional array-like object in Pandas that can hold any data type (integers, strings, floats, etc.).

3. DataFrame: A two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

4. Creating DataFrames: DataFrames can be created from dictionaries, lists, NumPy arrays, and external data sources (like CSV or SQL).

5. Indexing: DataFrames and Series can be indexed using labels or integers for both rows and columns.

6. Selecting Columns: You can select columns from a DataFrame using df['column_name'] or df.column_name.

7. Selecting Rows: Rows can be selected using iloc (by integer location) or loc (by label).

8. DataFrame Shape: The shape of a DataFrame can be accessed using the .shape attribute, returning the number of rows and columns.

9. Accessing Data Types: Data types of columns can be accessed using df.dtypes.

10. Handling Missing Data: Pandas provides methods like .isna(), .fillna(), and .dropna() to detect and handle missing data.

11. Renaming Columns: Columns can be renamed using the .rename() method.

12. Filtering Data: Use boolean indexing or conditions to filter data (e.g., df[df['column'] > 10]).

13. Sorting: Sorting data can be done by using df.sort_values() and df.sort_index().

14. Aggregation: Grouping data for aggregation can be done using df.groupby() for operations like sum, mean, count, etc.

15. Descriptive Statistics: Pandas provides df.describe() to generate summary statistics like count, mean, std, min, and max.

16. Handling Duplicates: Use .drop_duplicates() to remove duplicate rows based on specific columns.

17. Merging DataFrames: Combine multiple DataFrames using pd.merge() or df.merge() based on a common column.

18. Concatenation: You can concatenate DataFrames vertically or horizontally using pd.concat().

19. Pivot Tables: Pivot data using df.pivot() or df.pivot_table() to restructure the DataFrame for analysis.

20. Reshaping: Pandas provides melt() and stack() functions to reshape the data for better manipulation.

21. Datetime Handling: Pandas has powerful tools for working

with time-series data, using pd.to_datetime() to convert strings to dates.

22. Time Resampling: Resample time-series data for different frequencies (e.g., df.resample('M').mean() for monthly data).

23. String Methods: Pandas has a suite of string manipulation methods (.str) for operations like .split(), .replace(), and .lower().

24. Apply Function: The .apply() method allows you to apply a function along a DataFrame or Series axis (rows or columns).

25. Vectorization: Pandas operations are highly optimized for vectorized operations, making them faster than iterating over rows.

26. MultiIndex: Pandas supports hierarchical indexing, allowing for multi-level row and column indexing with pd.MultiIndex.

27. Missing Data Imputation: Use methods like .fillna() or

.interpolate() to replace missing values with specific values or estimates.

28. Dropping Columns/Rows: Columns or rows can be dropped using .drop() by specifying axis=0 (rows) or axis=1 (columns).

29. Sorting by Multiple Columns: You can sort DataFrame by multiple columns using df.sort_values(['col1', 'col2']).

30. Efficient Memory Usage: Use df.memory_usage() to monitor memory usage, and astype() to optimize data types for better performance.

31. Applymap: The .applymap() method allows applying a function element-wise to the entire DataFrame.

32. Column-Wise Operations: Operations like sum, mean, min, max can be done column-wise using the .sum(), .mean() functions, etc.

33. Row-Wise Operations: You can perform operations row-wise by specifying axis=1 in functions like .sum(axis=1).

34. Indexing with Conditions: You can filter data by creating conditional expressions directly in DataFrame indexing (e.g., df[df['col'] > 5]).

35. Window Functions: Pandas offers moving window statistics such as rolling mean using .rolling().

36. DataFrame to NumPy: You can convert a DataFrame or Series to a NumPy array using .to_numpy().

37. Handling Categorical Data: Pandas supports categorical data types, which help reduce memory usage and improve performance with pd.Categorical.

38. Using Apply for Row Operations: The .apply() function can be used to apply a function to each row for row-wise operations.

39. DataFrame Indexing: You can access and manipulate the index of a DataFrame directly using df.index.

40. Column Selection by Data Type: Select columns by their data type using df.select_dtypes(include='float').

41. Using Loc for Label-based Indexing: The .loc[] method allows selecting rows and columns by their labels.

42. Using Iloc for Integer Position-based Indexing: The .iloc[] method is used for selecting rows and columns by integer positions.

43. Saving to Excel: You can save DataFrames to Excel files using .to_excel().

44. Reading Excel Files: Pandas allows reading Excel files with pd.read_excel().

45. Handling JSON: Use pd.read_json() to read JSON data and df.to_json() to save data in JSON format.

46. Handling CSV: Use pd.read_csv() for reading CSV files and df.to_csv() for saving DataFrames to CSV format.

47. Filter DataFrame Columns: Select columns based on data type or condition using df.select_dtypes().

48. Copying Data: When making a copy of a DataFrame, use

.copy() to avoid modifying the original DataFrame.

49. Broadcasting: You can perform operations like arithmetic or applying functions directly to columns or rows without loops, leveraging Pandas broadcasting.

50. Pivoting and Unstacking: Use df.pivot() or .unstack() to reorganize your DataFrame for better representation or summarization of the data.

History of Pandas :

The history of Pandas traces its development from the need for a powerful, flexible tool for data manipulation and analysis in Python. Here's an overview of its key milestones:

1. Origin and Initial Development (2008)

Creator: Pandas was created by Wes McKinney, a former employee of AQR Capital Management.

Problem: While working as a quantitative analyst, Wes McKinney needed a flexible tool for working with time-series data and financial datasets, which were difficult to manage using existing tools like NumPy or R.

Inspiration: McKinney was inspired by tools such as R and Matlab which provided rich data manipulation functionality.

He aimed to create something similar for Python, a language that was gaining popularity for scientific computing.

Early Development: The first version of Pandas was developed in 2008 as an internal tool for handling financial data at AQR.

2. Release to the Public (2009)

Open Source: Wes McKinney released Pandas to the public in 2009 under the BSD open-source license. This allowed other developers to contribute and enhance the tool.

Initial Features: The initial versions of Pandas included support for data structures like Series (for one-dimensional data) and DataFrame (for two-dimensional data). These structures were designed to make it easier to work with labeled data, which was a challenge in many existing Python libraries at the time.

3. Growth and Adoption (2010-2013)

Increased Popularity: As the demand for data analysis and manipulation tools in Python grew, Pandas quickly gained adoption in both the academic and business worlds. Its ability to handle missing data, perform time-series operations, and interface with various data sources (CSV, Excel, SQL, etc.) made it a go-to library for data scientists.

Key Contributions: In these early years, various features were added, including improved time series support, better I/O capabilities, and more efficient methods for handling large datasets.

Community Contributions: The open-source nature of Pandas allowed the community to contribute code, fix bugs, and suggest improvements. Many new features were added by contributors from the data science and machine learning communities.

4. Continued Enhancement and Performance Improvements (2014-2017)

Performance Focus: As Pandas grew, performance became a key focus. The library introduced more efficient ways to handle large datasets and optimized internal algorithms for better memory usage and speed.

Enhanced Compatibility: Over this period, support for different file formats (like Parquet and HDF5) was added, making it easier to work with big data.

Pandas in Jupyter: With the rise of Jupyter Notebooks (formerly IPython notebooks), Pandas became increasingly popular as a tool for exploratory data analysis in an interactive environment. The ability to view DataFrames as tables within Jupyter made data manipulation and visualization easier.

5. Modern Pandas (2018-Present)

Version 1.0 (2020): Pandas 1.0 was released in 2020, marking a significant milestone with many new features, bug fixes, and improved performance. The 1.0 release included:

New data types and extension arrays (e.g., support for nullable integers and Boolean data types).

Enhanced support for timezones in datetime operations.

Improvements to GroupBy and merge operations.

Ongoing Improvements: The Pandas team and contributors continue to add new features, address bugs, and enhance performance. Notable features include:

DataFrame.query(): Allowing users to query DataFrames using a string expression.

Improved pd.merge(): Making join operations more efficient.

Performance enhancements: With each release, Pandas continues to optimize its internal implementation for handling larger datasets with lower memory overhead.

6. Integration with Modern Data Science and Machine

Learning Workflows

Ecosystem Integration: Pandas became an essential part of the Python data science stack, alongside libraries like NumPy, SciPy, matplotlib, Seaborn, scikit-learn, and TensorFlow. It is often used for data wrangling and preprocessing before analysis or feeding data into machine learning models.

Collaboration with PyArrow and Dask: In recent years, Pandas has also integrated well with libraries such as PyArrow (for handling Apache Arrow data structures) and Dask (for parallelizing operations on larger-than-memory datasets), allowing it to scale with big data.

7. Pandas in the Future

Better Support for Big Data: Although Pandas is extremely powerful, its memory usage has always been a limitation when dealing with very large datasets. The introduction of Dask DataFrame and Modin as distributed alternatives has helped scale operations for big data, while Pandas continues to work on memory optimizations.

Enhanced API and Features: Future versions of Pandas will likely continue to improve the library's performance, introduce new functionalities (like better support for category types and new I/O capabilities), and further integrate with other popular tools in the Python data ecosystem.

Key Contributions and Evolutionary Impact

Tool for Data Science: Pandas has become the cornerstone of data science and is a required tool for anyone working with data in Python. Its ease of use, flexibility, and powerful data manipulation capabilities have made it widely adopted by data analysts, data scientists, and researchers.

Influence on Other Libraries: Many libraries in the Python data ecosystem (like Koalas, Dask, and Modin) have drawn inspiration from Pandas for their API design, ensuring that its features are replicated and extended for distributed computing or bigger datasets.

Pandas has gone from an internal tool for financial data analysis to one of the most important libraries in the Python ecosystem for data manipulation and analysis. Its development is ongoing, and it continues to evolve with the needs of the data science community.

Advantages of Pandas:

1. Easy to Use and Intuitive:

Pandas provides simple, easy-to-understand data structures like DataFrame and Series, making it easy to load, manipulate, and analyze data. The syntax is intuitive, and its learning curve is relatively shallow for those familiar with

Python.

2. Data Cleaning and Preprocessing:

It has powerful tools for data cleaning, handling missing data, duplicate removal, and filtering data. Operations like fillna(), dropna(), drop_duplicates(), and replace() are highly efficient and intuitive.

3. Flexible Data Structures:

Pandas' primary data structures, Series (1D) and DataFrame (2D), are flexible and capable of holding a wide range of data types (integers, strings, dates, etc.). This makes it very versatile for data wrangling.

4. Powerful Data Aggregation:

Pandas excels at aggregating and summarizing data through methods like groupby(), pivot_table(), and agg(). It allows easy manipulation of large datasets by grouping, transforming, and applying functions.

5. Efficient I/O Handling:

Pandas provides fast and efficient methods to read and write data from/to a wide variety of formats, including CSV, Excel, SQL databases, JSON, Parquet, and HDFS, among others.

6. Time Series Handling:

One of the standout features of Pandas is its support for time-series data. It offers robust features for resampling, shifting, windowing, and working with time zones, making it ideal for applications like financial analysis.

7. Advanced Indexing and Selection:

Pandas offers powerful, flexible indexing and selection capabilities, such as loc[], iloc[], at[], iat[], allowing easy access and manipulation of data based on labels, integer positions, and conditional filters.

8. Integration with Other Libraries:

Pandas integrates seamlessly with other libraries in the Python ecosystem, like NumPy, Matplotlib, Seaborn, scikit-

and TensorFlow, providing a comprehensive workflow for data analysis, visualization, and machine learning.

9. Large Community and Documentation:

Pandas has an extensive community and rich documentation, with many tutorials, guides, and resources available for learning. The large user base ensures that any issues or bugs can be quickly addressed.

10. Compatibility with Python Ecosystem:

Being built on top of NumPy, Pandas provides seamless compatibility with numerical and scientific computation tools. It also works well in the broader Python data science stack.

Disadvantages of Pandas:

1. Memory Consumption:

Pandas can be memory-intensive, particularly when working with large datasets. It loads the entire dataset into memory, which can lead to out-of-memory errors when dealing with very large datasets that don't fit into the available RAM.

2. Performance with Large Datasets:

While efficient for small to medium-sized datasets, Pandas can struggle with very large datasets, especially those that exceed memory capacity. Operations on huge DataFrames can be slow, and multi-core or distributed processing is not natively supported (though libraries like Dask and Modin can help).

3. Complexity with Multi-Indexing:

Multi-indexing (having more than one index level) is a powerful feature but can become complex and difficult to manage when trying to perform certain operations or transformations, leading to confusion and bugs in large data manipulation tasks.

4. Lack of Built-in Parallelism:

Pandas doesn't have built-in support for parallelism or multi-threading. For parallel data processing, users need to use additional tools like Dask, Joblib, or multiprocessing, which complicates the workflow for large-scale data analysis.

5. Not Designed for Distributed Systems:

Unlike libraries such as Dask or Spark, which are specifically designed for distributed systems and can handle datasets that do not fit in memory, Pandas is meant for single-machine use. It requires all data to fit into the machine's memory.

6. Limited Support for Complex Data Types:

While Pandas excels with structured data, it does not handle more complex or nested data types like graphs or geospatial data as efficiently as specialized libraries such as NetworkX or GeoPandas.

7. Slow for Certain Operations:

While Pandas is fast for many tasks, operations like merging and joining large DataFrames can be slow, especially when

the datasets are large and require complex joins. This issue is particularly evident when merging on non-unique keys.

8. Difficult to Debug for Large Operations:

Because of its implicit and chained operations, debugging complex Pandas code, especially with large datasets, can be tricky. Errors can often appear as a result of operations on multiple chained methods, making it harder to pinpoint where the issue lies.

9. Handling Sparse Data:

While Pandas does support sparse data through special data structures, it is not as efficient in handling sparse matrices as SciPy, which is specifically designed for sparse data.

10. Inconsistent Behavior with Certain Data Types:

Certain operations may behave inconsistently when applied to different data types. For example, handling string data, date types, and categorical data might require additional effort to ensure smooth processing, and not all edge cases are well-documented.

Pandas is an incredibly powerful and widely used library for data manipulation and analysis in Python. While it provides a wealth of features and excellent functionality for most data science tasks, its limitations with memory, performance on very large datasets, and lack of built-in parallelism and distribution support are key drawbacks. Users working with larger datasets or needing high-performance computing may need to look into additional tools like Dask, Modin, or PySpark to overcome these limitations.

5 Use Cases of Pandas:

1. Data Cleaning and Preprocessing:

Scenario: You have a dataset with missing values, duplicate entries, and inconsistent formats.

Pandas Solution: Pandas provides tools like fillna(), dropna(), and drop_duplicates() for cleaning and preprocessing data. You can also use replace() to handle outliers or incorrect values and astype() to convert data types, ensuring that the dataset is ready for analysis.

Example:

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')
df.dropna(inplace=True) # Remove rows with missing values
df['column_name'] = df['column_name'].replace('old_value',
'new_value') # Replace values
```

2. Data Aggregation and Grouping:

Scenario: You want to summarize a dataset by grouping it by a specific column (e.g., category) and calculating statistics such as mean, sum, or count.

Pandas Solution: The groupby() method in Pandas allows you to split the data into groups based on a column and then apply aggregation functions like sum(), mean(), count(), or custom aggregation.

Example:

```
df = pd.read_csv('sales_data.csv')
grouped = df.groupby('region').agg({'sales': 'sum', 'profits':
'mean'})
print(grouped)
```

3. Time Series Analysis:

Scenario: You have time-series data, such as daily stock prices, and you want to resample the data to find monthly

or yearly trends.

Pandas Solution: Pandas has built-in support for time series operations like resampling, shifting, and rolling windows. You can easily convert date columns to datetime format and perform operations like resample(), shift(), and rolling() to analyze trends over time.

Example:

```
df = pd.read_csv('stock_data.csv', parse_dates=['date'])
df.set_index('date', inplace=True)
monthly_data = df.resample('M').mean() # Resample to monthly data
print(monthly_data)
```

4. Merging and Joining Datasets:

Scenario: You have two datasets (e.g., customer information and order history) and need to combine them based on a common key (e.g., customer ID).

Pandas Solution: Pandas provides efficient merging functions like merge() and join() to combine datasets based on common columns. This enables you to link data from different sources easily.

Example:

```
customer_df = pd.read_csv('customers.csv')
orders_df = pd.read_csv('orders.csv')
merged_df = pd.merge(customer_df, orders_df,
on='customer_id', how='inner')
print(merged_df)
```

5. Data Visualization Preparation:

Scenario: You need to prepare your data for visualization, such as calculating averages, totals, or creating new features before plotting charts.

Pandas Solution: You can use Pandas to manipulate and aggregate data, creating summary statistics or new derived features, which can then be visualized using libraries like Matplotlib or Seaborn.

Example:

```
import matplotlib.pyplot as plt
df = pd.read_csv('sales_data.csv')
sales_by_category = df.groupby('category')['sales'].sum()
sales_by_category.plot(kind='bar')
plt.show()
```

Each of these use cases demonstrates how Pandas can help streamline and optimize data processing, analysis, and preparation for further tasks such as reporting or machine learning.

50 top Pandas interview questions that are commonly asked:

Basic Questions:

1. What is Pandas, and why is it used in data analysis?

2. What are the main data structures in Pandas?

3. How do you create a DataFrame in Pandas?

4. What is the difference between a Pandas Series and a DataFrame?

5. How do you import a CSV file into a Pandas DataFrame?

6. Explain the concept of indexing in Pandas.

7. How do you access specific rows or columns in a DataFrame?

8. What is the purpose of the loc[] and iloc[] indexers in Pandas?

9. How do you filter data in a Pandas DataFrame based on certain conditions?

10. How can you check for missing values in a DataFrame?

Intermediate Questions:

11. How do you handle missing data in Pandas?

12. What is the purpose of the dropna() and fillna() functions?

13. How can you change the data type of a column in a DataFrame?

14. What is the purpose of the apply() function in Pandas?

15. How do you rename columns in a Pandas DataFrame?

16. What is a MultiIndex in Pandas?

17. How can you sort a DataFrame based on one or more columns?

18. Explain the difference between merge(), join(), and concat() in Pandas.

19. What is the purpose of the groupby() function in Pandas?

20. What is the difference between agg() and apply()?

Advanced Questions:

21. How do you handle categorical data in Pandas?

22. Explain the pivot_table() function in Pandas.

23. How can you handle duplicate rows in a DataFrame?

24. What is the difference between pivot() and melt() in Pandas?

25. Explain how to merge multiple DataFrames on different columns.

26. What is the purpose of the shift() function in Pandas?

27. How can you work with time-series data in Pandas?

28. What is resampling, and how is it done in Pandas?

29. How do you convert a string column to a datetime column in Pandas?

30. How can you find the correlation between two columns in a Pandas DataFrame?

Data Transformation Questions:

31. What is the difference between map() and apply() in Pandas?

32. Explain how to use the transform() function in Pandas.

33. How can you combine two or more columns into a single column in Pandas?

34. What is vectorized operation, and how is it used in Pandas?

35. How can you create a new column based on conditions in Pandas?

36. Explain the concept of "broadcasting" in Pandas.

37. How can you use Pandas to deal with outliers in your data?

38. How do you work with missing data when performing statistical operations in Pandas?

39. How can you filter out specific rows from a DataFrame based on string values?

40. How do you handle infinite values in Pandas?

Performance and Optimization Questions:

41. How can you improve the performance of Pandas operations on large datasets?

42. What is the purpose of the inplace=True parameter in many Pandas functions?

43. How can you handle large datasets that don't fit in memory in Pandas?

44. What are some memory-efficient ways to work with large datasets in Pandas?

45. What is the difference between Pandas and Dask when working with large data?

Time Series and Date/Time Questions:

46. How can you extract the year, month, and day from a datetime column in Pandas?

47. How do you resample time-series data in Pandas?

48. What are time-shifts in Pandas, and how are they used?

49. How do you handle time zone conversions in Pandas?

50. How do you calculate rolling averages or moving windows in time-series data?

Basic Questions:

1. What is Pandas, and why is it used in data analysis?
Pandas is a Python library used for data manipulation and analysis. It provides data structures like Series and DataFrame for efficiently handling and analyzing structured data.

2. What are the main data structures in Pandas?
Pandas has two main data structures: Series (1D labeled

array) and DataFrame (2D labeled data structure).

3. How do you create a DataFrame in Pandas?
You can create a DataFrame using a dictionary, list of dictionaries, or from other data formats like CSV, Excel, or SQL databases.
Example:

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

4. What is the difference between a Pandas Series and a DataFrame?
A Series is a one-dimensional labeled array, while a DataFrame is a two-dimensional labeled data structure with rows and columns.

5. How do you import a CSV file into a Pandas DataFrame?
Use the read_csv() function.
Example:

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

6. Explain the concept of indexing in Pandas.
Indexing allows accessing data in rows and columns using labels or integer-based positions.

7. How do you access specific rows or columns in a DataFrame?
Use column names for accessing columns, and loc[] or iloc[] for accessing rows.
Example:

```
df['column_name'] # Access column
df.loc[0] # Access row by label
df.iloc[0] # Access row by position
```

8. What is the purpose of the loc[] and iloc[] indexers in Pandas?

loc[]: Label-based indexing.

iloc[]: Integer-based indexing.

9. How do you filter data in a Pandas DataFrame based on certain conditions?
Use boolean indexing.
Example:

```
df_filtered = df[df['column'] > 10]
```

10. How can you check for missing values in a DataFrame?
Use the isna() or isnull() functions.

Example:

```
df.isna().sum()
```

Intermediate Questions:

11. How do you handle missing data in Pandas?
Use fillna() to fill missing values and dropna() to remove them.

12. What is the purpose of the dropna() and fillna() functions?

dropna(): Removes rows or columns with missing data.

fillna(): Replaces missing data with a specified value or method.

13. How can you change the data type of a column in a DataFrame?
Use the astype() function.

Example:

```
df['column'] = df['column'].astype('int')
```

14. What is the purpose of the apply() function in Pandas?
It applies a function to each element, row, or column of a DataFrame or Series.

15. How do you rename columns in a Pandas DataFrame?
Use the rename() function or directly modify the columns attribute.
Example:

```
df.rename(columns={'old_name': 'new_name'}, inplace=True)
```

16. What is a MultiIndex in Pandas?
A MultiIndex is a hierarchical index that allows multiple levels of indexing in rows or columns.

17. How can you sort a DataFrame based on one or more columns?
Use the sort_values() function.
Example:

df.sort_values(by='column', ascending=True, inplace=True)

18. Explain the difference between merge(), join(), and concat() in Pandas.

merge(): Combines DataFrames based on keys.

join(): Combines DataFrames based on the index.

concat(): Combines DataFrames along rows or columns.

19. What is the purpose of the groupby() function in Pandas?
It is used for grouping data based on column values and applying aggregate functions.

20. What is the difference between agg() and apply()?

agg(): Used for aggregation operations like sum, mean.

apply(): Used for custom functions applied to rows/columns.

Advanced Questions:

21. How do you handle categorical data in Pandas?
Use the astype('category') method or pd.get_dummies() for one-hot encoding.

22. Explain the pivot_table() function in Pandas.
It is used for summarizing data with an aggregation function and reshaping the DataFrame.

23. How can you handle duplicate rows in a DataFrame?
Use drop_duplicates().
Example:

```
df.drop_duplicates(inplace=True)
```

24. What is the difference between pivot() and melt() in Pandas?

pivot(): Reshapes data into a wider format.

melt(): Reshapes data into a longer format.

25. Explain how to merge multiple DataFrames on different columns.
Use the merge() function with the on parameter for multiple column names.
Example:

```
pd.merge(df1, df2, on=['col1', 'col2'])
```

26. What is the purpose of the shift() function in Pandas?
Shifts data by a specified number of periods along an axis.

27. How can you work with time-series data in Pandas?
Use pd.to_datetime() for conversion and various time-series functions like resample().

28. What is resampling, and how is it done in Pandas?
Resampling is changing the frequency of time-series data using resample().

29. How do you convert a string column to a datetime column in Pandas?
Use the pd.to_datetime() function.
Example:

```
df['date'] = pd.to_datetime(df['date'])
```

30. How can you find the correlation between two columns in a Pandas DataFrame?
Use the corr() function.
Example:

```
df['col1'].corr(df['col2'])
```

Data Transformation Questions:

31. What is the difference between map() and apply() in Pandas?

map(): Element-wise operations on Series.

apply(): Works on rows/columns of DataFrame.

32. Explain how to use the transform() function in Pandas.
transform() applies a function element-wise and maintains the DataFrame's shape.

33. How can you combine two or more columns into a single column in Pandas?
Use string concatenation or the + operator.
Example:

```
df['new_col'] = df['col1'] + ' ' + df['col2']
```

34. What is vectorized operation, and how is it used in Pandas?
Operations applied to entire arrays for performance optimization.

35. How can you create a new column based on conditions in Pandas?
Use the np.where() or .apply().
Example:

```
df['new_col'] = np.where(df['col'] > 10, 'High', 'Low')
```

Amar Sharma

AI Engineer

Follow me on LinkedIn for more informative content 📝