PYTHON INTERVIEW Q&A

EDITION 2025

Developed by Swapnjeet S (Data Tutorials)

1. **What is Pandas? How is it used in Data Analysis?**
    - **Answer**: Pandas is an open-source Python library that provides flexible and efficient tools for data manipulation and analysis. It offers data structures like Series and DataFrame for handling structured data.
2. **What are the key data structures in Pandas?**
    - **Answer**: The two primary data structures in Pandas are:
        - Series: A one-dimensional labeled array.
        - DataFrame: A two-dimensional, size-mutable, and labeled data structure, similar to a table.
3. **How do you create a Pandas DataFrame?**
    - **Answer**: You can create a DataFrame from:
        - A dictionary: pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
        - A list of lists: pd.DataFrame([[1, 2], [3, 4]], columns=['A', 'B'])
        - A CSV/Excel file: pd.read_csv('file.csv')
4. **How do you handle missing data in Pandas?**
    - **Answer**:
        - Drop rows with missing values: df.dropna()
        - Fill missing values: df.fillna(value)
        - Check for missing values: df.isnull() or df.isna()

---

5. **What are the different ways to select data in a Pandas DataFrame?**
    - **Answer**:
        - Using column labels: df['column_name']
        - Using row indices: df.iloc[0]
        - Using both labels and indices: df.loc[row_label, col_label]
6. **How can you filter rows in a DataFrame based on conditions?**
    - **Answer**:

```
# Filter rows where column A > 10

filtered_df = df[df['A'] > 10]
```

7. **What is the difference between loc and iloc in Pandas?**
    - **Answer**:
        - loc is label-based indexing (e.g., df.loc['row_label', 'col_label']).
        - iloc is position-based indexing (e.g., df.iloc[0, 1]).
8. **How do you merge, join, or concatenate DataFrames?**
    - **Answer**:
        - Merge: pd.merge(df1, df2, on='common_column')
        - Join: df1.join(df2, how='inner')
        - Concatenate: pd.concat([df1, df2], axis=0) (rows) or axis=1 (columns)
9. **How do you group data in Pandas?**
    - **Answer**:

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

result = grouped['target_column'].sum()
```

10. **Explain the use of the apply function in Pandas.**
    - **Answer**: The apply function applies a function along an axis of the DataFrame (row-wise or column-wise).

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

---

11. **How do you handle large datasets in Pandas?**
    - **Answer**:
        - Use chunks: pd.read_csv('file.csv', chunksize=1000)
        - Optimize data types: Convert columns to appropriate data types (e.g., category for low-cardinality columns).
        - Use dask for parallelized operations.
12. **What is the difference between pivot and pivot_table in Pandas?**
    - **Answer**:
        - pivot: Reshape data based on unique column values but does not handle duplicate entries.
        - pivot_table: Allows aggregations like mean, sum, etc., and handles duplicates.

13. **How do you handle time-series data in Pandas?**
    - **Answer**:
        - Convert to datetime: pd.to_datetime(df['date_column'])
        - Set as index: df.set_index('date_column', inplace=True)
        - Resample: df.resample('M').mean()
14. **What is a MultiIndex in Pandas, and how do you use it?**
    - **Answer**: MultiIndex is an advanced index object supporting multiple levels.

df.set_index(['col1', 'col2'], inplace=True)

15. **How do you perform vectorized operations in Pandas?**
    - **Answer**: Vectorized operations operate directly on entire columns without looping.

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

16. **How do you detect and remove duplicate rows in Pandas?**
    - **Answer**:
        - Detect: df.duplicated()
        - Remove: df.drop_duplicates()

Scenario-Based Questions

17. **Given a DataFrame with sales data, how would you calculate the cumulative sales by month?**
    - **Answer**:

df['cumulative_sales'] = df['sales'].cumsum()

18. **How would you find the top 5 products with the highest sales in a DataFrame?**
    - **Answer**:

top_products = df.nlargest(5, 'sales')

19. **How do you calculate correlation between numerical columns in Pandas?**
    - **Answer**:

correlation_matrix = df.corr()

20. **How do you convert a DataFrame into a dictionary?**
    - **Answer**:

data_dict = df.to_dict()

21. **What are the main differences between Pandas and NumPy?**

- **Answer**:
  - **Data Type**: Pandas handles structured/tabular data, while NumPy deals with homogeneous arrays.
  - **Indexing**: Pandas provides labeled indexing with Series and DataFrame; NumPy uses integer-based indexing.
  - **Functionality**: Pandas includes tools for data cleaning, reshaping, and time-series analysis. NumPy focuses on numerical computation.

---

22. **Explain Pandas DataFrame broadcasting and its importance.**
    - **Answer**:
      - Broadcasting allows arithmetic operations between DataFrames and Series, aligning data on indices and filling missing values with NaN.
      - Example:

```
df['new_col'] = df['existing_col'] + 10  # Broadcasting a scalar
```

---

23. **What is the difference between .apply(), .map(), and .applymap()?**
    - **Answer**:
      - map(): Used for element-wise operations on Series.
      - apply(): Used for row-wise or column-wise operations on DataFrame.
      - applymap(): Element-wise operations on an entire DataFrame.

---

24. **What are Pandas vectorized operations, and why are they faster than loops?**
    - **Answer**:
      - Vectorized operations apply a function on an entire array or column at once, leveraging low-level optimizations in C/Cython, making them faster than Python loops.
      - Example:

```
df['new_col'] = df['col1'] + df['col2']  # Vectorized addition
```

---

25. **Explain the concept of data alignment in Pandas.**
    - **Answer**:
      - Data alignment ensures that operations between Series or DataFrames align by their labels (indices) automatically. Missing values are filled with NaN.

- Example:

```
s1 = pd.Series([1, 2], index=['A', 'B'])

s2 = pd.Series([3, 4], index=['B', 'C'])

result = s1 + s2  # Result aligns by index
```

---

26. **How does Pandas handle categorical data, and why is it useful?**
    - **Answer**:
        - Categorical data is represented using the Categorical dtype, which stores data as categories with associated codes. It reduces memory usage and speeds up operations for columns with low cardinality.
        - Example:

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

---

27. **What are the performance considerations for Pandas operations on large datasets?**
    - **Answer**:
        - Use appropriate data types (e.g., int8 for small integers, category for categorical data).
        - Avoid row-wise operations; prefer vectorized operations.
        - Use chunksize while reading large files to process in smaller parts.
        - Utilize parallel libraries like dask or modin for distributed processing.

---

28. **What is the role of the transform function in Pandas, and how is it different from apply?**
    - **Answer**:
        - transform applies a function element-wise but retains the same shape as the input. It is commonly used in conjunction with groupby for broadcasting results back to the original DataFrame.
        - Example:

```
df['mean_sales'] = df.groupby('category')['sales'].transform('mean')
```

---

29. **How can you optimize memory usage in Pandas?**
    - **Answer**:
        - Use df.info(memory_usage='deep') to analyze memory usage.
        - Downcast numeric columns: pd.to_numeric(col, downcast='float').

- Convert string columns to category if they have repetitive values.
- Use sparse data types for columns with many zeros.

---

30. **What is the difference between reshaping methods melt and pivot in Pandas?**
    - **Answer**:
        - melt: Converts wide-format data to long-format data (unpivoting).
        - pivot: Converts long-format data to wide-format data (pivoting).
        - Example:

```
melted = pd.melt(df, id_vars=['id'], value_vars=['A', 'B'])
pivoted = melted.pivot(index='id', columns='variable', values='value')
```

---

31. **What are Pandas Window Functions, and how do they work?**
    - **Answer**:
        - Window functions operate over a sliding window of rows, enabling calculations like rolling averages or cumulative sums.
        - Example:

```
df['rolling_mean'] = df['sales'].rolling(window=3).mean()
```

---

32. **Explain the purpose of pd.cut() and pd.qcut().**
    - **Answer**:
        - pd.cut(): Bins continuous data into equal intervals.
        - pd.qcut(): Bins data into equal-sized quantiles.
        - Example:

```
df['bins'] = pd.cut(df['values'], bins=[0, 10, 20])
```

---

33. **What is a Pandas Index, and what are its types?**
    - **Answer**:
        - Pandas Index is the axis labeling system for Series and DataFrame. Types include:
            - RangeIndex: Default integer index.
            - MultiIndex: Multi-level hierarchical index.
            - DatetimeIndex: Index for time-series data.

- CategoricalIndex: Index for categorical data.

---

34. **What are Pandas eval() and query() methods? How do they improve performance?**
    - **Answer**:
        - eval(): Evaluates expressions for arithmetic operations.
        - query(): Filters data using a string expression.
        - These methods are faster as they avoid Python's interpreter and use Numexpr for computations.

---

35. **How do you check if two DataFrames are identical?**
    - **Answer**:

```
result = df1.equals(df2)
```

---

36. **How would you handle unstructured or nested data (e.g., JSON) in Pandas?**
    - **Answer**:
        - Load JSON data using pd.json_normalize() to flatten nested structures into columns.
        - Example:

```
normalized_df = pd.json_normalize(json_data)
```

37. **What are some best practices for improving Pandas performance on extremely large datasets?**

- **Answer**:
    - **Use proper data types**: Convert columns to smaller data types (int8, float32, category).
    - **Indexing**: Set appropriate indexes for faster lookup.
    - **Vectorized operations**: Avoid Python loops and use vectorized methods.
    - **Lazy loading**: Use chunksize while reading large files.
    - **External tools**: Use libraries like Dask or Vaex for distributed computations.

---

38. **Explain the difference between merge and join. When should each be used?**

- **Answer**:
    - merge: Combines DataFrames based on a common key. It is more flexible and supports all SQL-style joins.

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

- join: Combines DataFrames based on the index by default or on specified columns.

python

Copy code

df1.join(df2, how='left')

- Use merge when you need explicit control over join keys and operations, and join for simpler index-based merges.

---

39. **How do you perform hierarchical indexing, and what are its benefits?**

- **Answer**:
  - **Hierarchical indexing** (or MultiIndex) allows multiple levels of indexing for rows and columns, enabling complex data aggregation and slicing.
  - Example:

df.set_index(['col1', 'col2'], inplace=True)

df.loc[('value1', 'value2')]

  - **Benefits**:
    - Simplifies representation of multi-dimensional data.
    - Enhances grouping and aggregation capabilities.

---

40. **What is the difference between rolling, expanding, and ewm in Pandas?**

- **Answer**:
  - rolling: Applies a moving window function (e.g., moving average).

df['rolling_mean'] = df['col'].rolling(window=3).mean()

  - expanding: Expands the window to include all preceding data.

df['cumulative_mean'] = df['col'].expanding().mean()

  - ewm (Exponentially Weighted Moving): Weighs recent observations more heavily.

df['ewm_mean'] = df['col'].ewm(span=3).mean()

---

41. **What is the pd.Grouper class, and how is it useful?**

- **Answer**:
  - pd.Grouper is used to group data by specific intervals or criteria, especially for time-series data.

- Example:

```
df.groupby(pd.Grouper(key='date', freq='M'))['sales'].sum()
```

- It simplifies grouping operations without creating additional columns.

---

42. **How do you efficiently work with sparse data in Pandas?**

- **Answer**:
  - Convert dense columns to sparse types using pd.SparseDtype:

```
df['col'] = df['col'].astype(pd.SparseDtype("float", fill_value=0))
```

  - Sparse data structures store only non-default values, reducing memory usage significantly for datasets with many zeros.

---

43. **Explain chaining operations in Pandas. How can it be made more readable?**

- **Answer**:
  - Chaining refers to applying multiple operations sequentially:

```
result = (df.query("col > 10")
    .assign(new_col=lambda x: x['col'] * 2)
    .sort_values('new_col'))
```

  - **Best practices for readability**:
    - Use parentheses.
    - Avoid in-place operations.
    - Use .pipe() for custom functions.

---

44. **How does Pandas handle duplicates in a MultiIndex DataFrame?**

- **Answer**:
  - duplicated identifies duplicate entries at any level:

```
df.duplicated(subset=['level1', 'level2'])
```

  - drop_duplicates removes duplicate rows:

```
df.drop_duplicates(subset=['level1', 'level2'])
```

---

45. **What is the difference between combine_first and fillna in Pandas?**

- **Answer**:

- fillna: Replaces missing values with a specified value or method.

```
df['col'].fillna(0)
```

- combine_first: Combines two DataFrames or Series, using values from the second where the first has NaN.

```
df1.combine_first(df2)
```

---

46. **Explain the concept of "index alignment" in Pandas arithmetic operations.**

- **Answer**:
  - Index alignment ensures that arithmetic operations align Series or DataFrame indices before computation.
  - Example:

```
s1 = pd.Series([1, 2], index=['A', 'B'])
s2 = pd.Series([3, 4], index=['B', 'C'])
result = s1 + s2  # Indexes are aligned: {'A': NaN, 'B': 6, 'C': NaN}
```

---

47. **How can you debug or trace errors in chained Pandas operations?**

- **Answer**:
  - Break the chain into smaller steps for inspection.
  - Use intermediate assignments:

```
step1 = df.query("col > 10")
step2 = step1.assign(new_col=step1['col'] * 2)
```

  - Use the debug() method (available in IDEs or debuggers).

---

48. **What is pd.IndexSlice, and how is it used in MultiIndex slicing?**

- **Answer**:
  - pd.IndexSlice allows slicing MultiIndex DataFrames concisely.
  - Example:

```
df.loc[pd.IndexSlice[:, 'sub_level'], :]
```

---

49. **How do you manage column-wise operations in wide-format data?**

- **Answer**:
  - Use df.apply() for column-wise operations:

df.apply(lambda x: x.max() - x.min(), axis=0)

- Use vectorized functions for faster operations:

df['new_col'] = df['col1'] * df['col2']

---

50. **How would you create a custom aggregation function for a grouped DataFrame?**

- **Answer**:

```
def custom_agg(series):
    return series.mean() + series.std()

grouped = df.groupby('col').agg(custom_agg)
```