**Pandas Questions and Answers**

**Q1** [**. Is iterating over a Pandas Dataframe a good practice? If not what are the important conditions to keep in mind before iterating?**](https://www.interviewbit.com/pandas-interview-questions/#important-conditions-to-keep-in-mind-before-iterating)

Ans Iterating Over a Pandas DataFrame: Good Practice or Not?

In general, iterating over a Pandas DataFrame is not considered good practice due to performance concerns. Pandas is designed for vectorized operations, which are much faster and more efficient than iterating row by row. However, there are specific scenarios where iteration might be necessary or unavoidable.

### Why Iteration is Usually Not Recommended:

1. Performance: Iterating over DataFrame rows using loops (for loops or iterrows()) can be slow, especially for large datasets. Pandas is optimized for vectorized operations, where operations are applied simultaneously across entire columns or rows.
2. Code Complexity: Iterative solutions are often more complex and harder to read and maintain compared to vectorized operations.

### Alternatives to Iteration:

Vectorized Operations: Use built-in Pandas functions that operate on entire columns or DataFrames. These are implemented in C and are much faster than Python loops.  
python  
Copy code  
df['C'] = df['A'] + df['B'] # Adding two columns

Apply Functions: Use apply() for row-wise or column-wise operations when you need custom logic that isn't directly supported by vectorized operations.  
python  
Copy code  
df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)

### Important Considerations Before Iterating:

1. Performance Impact: Always assess the size of your DataFrame. Iteration might be acceptable for small datasets but can be prohibitively slow for larger ones.
2. Check for Alternatives: Before resorting to iteration, check if there’s a vectorized solution or if apply() can be used.
3. Test Different Approaches: If you must iterate, compare the performance of different methods like iterrows(), itertuples(), and apply() to find the most efficient solution for your use case.

**Q2** [**. How would you iterate over rows in a DataFrame in Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#iterate-over-rows-in-a-data-frame-in-pandas)

### Iteration Methods in Pandas:

iterrows(): Yields each row as a (index, Series) pair. It's slow and not recommended for performance-sensitive tasks.  
python  
Copy code

for index, row in df.iterrows():

print(index)

print(row)

itertuples(): Yields each row as a named tuple. Faster than iterrows() because it avoids the overhead of converting each row to a Series.  
python  
Copy code

f# Iterating over rows using itertuples()

for row in df.itertuples():

print(row)

apply(): Applies a function along the axis of the DataFrame. Generally faster than iterrows() and itertuples() for many operations.  
python  
Copy code  
df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)

### Conclusion:

While iterating over a Pandas DataFrame is sometimes necessary, it should generally be avoided in favor of vectorized operations or other Pandas methods. Always consider the performance implications and look for more efficient alternatives before deciding to iterate over a DataFrame.

**Q3** [**List some statistical functions in Python Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#statistical-functions-in-python-pandas)

Here’s a list of common statistical functions in Python Pandas:

1. sum() - Sum of values.
2. mean() - Mean (average) of values.
3. median() - Median of values.
4. mode() - Mode (most frequent value).
5. std() - Standard deviation of values.
6. var() - Variance of values.
7. min() - Minimum value.
8. max() - Maximum value.
9. quantile() - Value at a given quantile.
10. corr() - Correlation between columns.
11. cov() - Covariance between columns.
12. count() - Count of non-null values.
13. describe() - Summary statistics.
14. skew() - Skewness of distribution.
15. kurt() - Kurtosis of distribution.

**Q4**  [**How to Read Text Files with Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#read-text-files-with-pandas)

Here’s a list of ways to read text files with Pandas:

1. Basic Usage:
   * Use pd.read\_csv('file.txt') for reading files with comma-separated values.
2. Specifying a Different Delimiter:
   * Use sep parameter to specify the delimiter.
   * Example: pd.read\_csv('file.txt', sep='\t') for tab-separated values.
3. No Header Row:
   * Use header=None to indicate no header row.
   * Example: pd.read\_csv('file.txt', header=None, names=['Column1', 'Column2']).
4. Reading Specific Columns:
   * Use usecols to read specific columns.
   * Example: pd.read\_csv('file.txt', usecols=['Column1', 'Column3']).
5. Handling Missing Values:
   * Use na\_values to specify custom missing value indicators.
   * Example: pd.read\_csv('file.txt', na\_values=['NA', 'Missing']).
6. Reading a Fixed-Width File:
   * Use pd.read\_fwf('file.txt') for reading fixed-width files.

**Q5**  [**How are iloc() and loc() different?**](https://www.interviewbit.com/pandas-interview-questions/#iloc-vs-loc)

The iloc[] and loc[] functions in Pandas are used for data selection and filtering in DataFrames, but they differ in how they index and retrieve data.

### Differences between iloc[] and loc[]:

1. Indexing Method:
   * iloc[]:
     + Uses integer-based indexing.
     + Selects data by the position of rows and columns (like standard Python lists).
     + Example: df.iloc[0, 1] retrieves the data in the first row and second column.
   * loc[]:
     + Uses label-based indexing.
     + Selects data by the labels or names of the rows and columns.
     + Example: df.loc['row\_label', 'column\_label'] retrieves the data in the row labeled 'row\_label' and column labeled 'column\_label'.
2. Inclusive vs. Exclusive Slicing:
   * iloc[]:
     + Slices are exclusive of the endpoint.
     + Example: df.iloc[0:3, 0:2] selects rows 0, 1, 2 and columns 0, 1.
   * loc[]:
     + Slices are inclusive of the endpoint.
     + Example: df.loc['row1':'row3', 'col1':'col2'] selects rows 'row1', 'row2', 'row3' and columns 'col1', 'col2'.

Use iloc[] for selecting data by position (integer-based).

Use loc[] for selecting data by labels (label-based).

import pandas as pd

student={"Name":["a","b","c"],"age":[21,23,22],"marks":[88,92,84]}

data=pd.DataFrame(student)

data.loc[0:1,["Name","marks"]]

data.iloc[0:2,0:3]

**Q6**  [**How will you sort a DataFrame?**](https://www.interviewbit.com/pandas-interview-questions/#sort-a-dataframe)

The function used for sorting in pandas is called DataFrame.sort\_values(). It is used to sort a DataFrame by its column or row values. The function comes with a lot of parameters, but the most important ones to consider for sort are:

* by: It is used to specify the column/row(s) which are used to determine the sorted order. It is an optional parameter.
* axis: It specifies whether the sorting is to be performed for a row or column and the value is 0 and 1 respectively.
* ascending: It specifies whether to sort the dataframe in ascending or descending order. The default value is set to ascending. If the value is set as ascending=False it will sort in descending order.

df = df.sort\_values(by='column\_name') # Sort by a single column.

df = df.sort\_values(by=['col1', 'col2']) # Sort by multiple columns.

df = df.sort\_values(by='column\_name', ascending=False) # Sort in descending order.

df = df.sort\_values(by=['col1', 'col2'], ascending=[True, False]) # Sort `col1` in ascending and `col2` in descending.

df = df.sort\_values(by='column\_name', na\_position='first') # Place NaN values at the beginning.

**Q7**[**. How would you convert continuous values into discrete values in Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#convert-continuous-values-into-discrete-values-in-pandas)

Depending on the problem, continuous values can be discretized using the cut() or qcut() function:

* cut() It bins the data based on values. We use it when we need to segment and sort data values into bins that are evenly spaced. cut() will choose the bins to be evenly spaced based on the values themselves and not the frequency of those values. For example, cut could convert ages to groups of age ranges.
* qcut() bins the data based on sample quantiles. We use it when we want to have the same number of records in each bin or simply study the data by quantiles. For example, if in a data we have 30 records, and we want to compute the quintiles, qcut() will divide the data such that we have 6 records in each bin.

**import pandas as pd**

**# Example DataFrame**

**data = {'values': [2.5, 6.7, 8.1, 3.6, 9.8, 1.2]}**

**df = pd.DataFrame(data)**

**# Define bins and labels**

**bins = [0, 3, 7, 10] # Bin edges**

**labels = ['Low', 'Medium', 'High'] # Bin labels**

**# Use pd.cut to convert continuous to discrete values**

**df['binned\_values'] = pd.cut(df['values'], bins=bins, labels=labels)**

**print(df)**

**df = pd.DataFrame({**

**'Age': [5, 12, 17, 24, 45, 64, 78]**

**})**

**def segragating(age):**

**if age<10:**

**return("Child")**

**elif age<20:**

**return("Tennage")**

**elif age<45:**

**return("adult")**

**else:**

**return("senior")**

**df["Criteria"]=df["Age"].apply(segragating)**

**Q8**  [**What is the difference between join() and merge() in Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#join-vs-merge)

Both join and merge functions are used to combine two dataframes. The major difference is that the join method combines two dataframes on the basis of their indexes whereas the merge method is more flexible and allows us to specify columns along with the index to combine the two dataframes.

These are the main differences between df.join() and df.merge():

* lookup on right table: When performing a lookup on the right table, the join() method will always use the index of df2 to perform the join operation. However, if you use the merge() method, you can choose to join based on one or more columns of df2 by default, or even the index of df2 if you specify the right\_index=True parameter.
* lookup on left table: When performing a lookup on the left table, df1.join(df2) method will use the index of df1 by default, while df1.merge(df2) method will use the column(s) of df1 for the join operation. However, you can override this behavior by specifying the on=key\_or\_keys parameter in df1.join(df2) or by setting the left\_index=True parameter in df1.merge(df2).
* left vs inner join: By default, the df1.join(df2) method performs a left join (retains all rows of df1), while the df1.merge(df2) method performs an inner join (returns only the matching rows of df1 and df2)

[**9. What is the difference(s) between merge() and concat() in Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#merge-vs-concat)

Both concat and merge functions are used to combine dataframes. There are three major key differences between these two functions.

* The Way of Combining: concat() function concatenates dataframes along rows or columns. It is nothing but stacking up of multiple dataframes whereas merge() combines dataframes based on values in shared columns thus it is more flexible compared to concat() as the combination can happen based on the given condition.
* Axis parameter: concat() function has axis parameter. Since merge() function combines dataframes on the basis of shared columns side by side it does not really need an axis parameter. The value of the axis parameter decides in what direction will the concatenation happen. For it to happen row-wise the value of the axis parameter will be ‘0’ and for it to happen side-by-side it will be ‘1’. The default value is 1.
* Join vs How: Join is a parameter of concat() function and how is a parameter of merge() function. Join can take two values outer and inner whereas how can take four values inner, outer, left, and right.

[**10. What’s the difference between interpolate() and fillna() in Pandas?**](https://www.interviewbit.com/pandas-interview-questions/#interpolate-vs-fillna)

fillna():

* Use when you want to fill NaNs with specific values, carry forward or backward the last known value, or use a statistical measure like mean, median, or mode.
* Common for categorical data or when you have a clear replacement value.

interpolate():

* Best for continuous data where you want to estimate the missing values based on the trend or pattern of the data.
* Useful for time series or when you believe that missing values can be reasonably estimated from surrounding data points.

### Methods Available:

* fillna():
  + Can use specific values, such as a constant, or methods like:
    - method='ffill': Forward fill
    - method='bfill': Backward fill
    - value: Replace with a specific value
    - inplace=True: Modify the DataFrame/Series in place
* interpolate():
  + Supports various interpolation methods such as:
    - method='linear': Default linear interpolation
    - method='polynomial': Polynomial interpolation (e.g., method='polynomial', order=2)
    - method='spline': Spline interpolation
    - method='time': Time-based interpolation (for time series data)
    - limit: Maximum number of consecutive NaNs to fill

Replaces NaN values with a specified value or method.

df['column'].fillna(0) # Replaces NaN with 0

df['column'].fillna(df['column'].mean()) # Replaces NaN with the column mean

df['column'].fillna(method='ffill') # Forward fill (propagates last valid observation)

Estimates missing values by interpolating between adjacent non-missing values.

df['column'].interpolate(method='linear') # Linear interpolation

df['column'].interpolate(method='quadratic') # Quadratic interpolation