Q1. How do you load a CSV file into a Pandas Dataframe?

To load a CSV (Comma Separated Values) file into a Pandas DataFrame, you can use the read_csv() function provided by Pandas. Here's the general syntax:

```
import pandas as pd

# Load CSV file into a DataFrame

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

Q2. How do you check the data type of a column in a Pandas DataFrame?

To check the data type of a column in a Pandas DataFrame, you can use the dtypes attribute or the dtype method. Here's an example:

```
import pandas as pd

# Create a sample DataFrame

data = {'Name': ['John', 'Jane', 'Mike'],

        'Age': [25, 30, 35],

        'Salary': [50000.0, 60000.0, 70000.0]}

df = pd.DataFrame(data)

# Check the data types using dtypes attribute

print(df.dtypes)

# Check the data type of a specific column using dtype method

print(df['Age'].dtype)
```

Output:

go

Copy code

```
Name      object

Age        int64

Salary    float64

dtype: object

int64
```

In the example above, the dtypes attribute returns the data types of all columns in the DataFrame, while the dtype method is used to check the data type of a specific column (in this case, the 'Age' column). The output shows the data types of each column in the DataFrame.

Q3. How do you select rows from a Pandas DataFrame based on a condition

To select rows from a Pandas DataFrame based on a condition, you can use boolean indexing. Here's an example:

```
import pandas as pd

# Create a DataFrame

data = {'Name': ['John', 'Emma', 'Peter', 'Emily', 'Daniel'],

        'Age': [25, 30, 20, 35, 28],

        'Country': ['USA', 'Canada', 'UK', 'USA', 'Australia']}

df = pd.DataFrame(data)

# Select rows where Age is greater than 25

condition = df['Age'] > 25

selected_rows = df[condition]

print(selected_rows)
```

Output:

markdown

Copy code

```
    Name  Age    Country

1   Emma   30     Canada

3  Emily   35        USA

4  Daniel   28  Australia
```

In this example, we created a DataFrame with three columns: 'Name', 'Age', and 'Country'. We then defined a condition using boolean comparison (df['Age'] > 25) to check if the 'Age' column is greater than 25. This condition creates a boolean Series with True for rows where the condition is satisfied and False for rows where it is not.

Finally, we used the boolean Series to index the original DataFrame (df[condition]), which returns a new DataFrame containing only the rows where the condition is True. In this case, it selects the rows where the 'Age' is greater than 25.

You can use various comparison operators (e.g., ==, !=, <, >, <=, >=) and combine multiple conditions using logical operators (& for AND, | for OR) to select rows based on different criteria.

Q4. How do you rename columns in a Pandas DataFrame?

To rename columns in a Pandas DataFrame, you can use the rename() method. The rename() method allows you to specify new names for one or more columns.

Here's an example of how to rename columns in a Pandas DataFrame:

```
import pandas as pd

# Create a sample DataFrame

data = {'A': [1, 2, 3],

        'B': [4, 5, 6],

        'C': [7, 8, 9]}

df = pd.DataFrame(data)

# Rename columns

df = df.rename(columns={'A': 'Column1', 'B': 'Column2', 'C': 'Column3'})

# Print the updated DataFrame

print(df)
```

Output:

Copy code

```
   Column1  Column2  Column3

0        1        4        7

1        2        5        8

2        3        6        9
```

In the rename() method, you pass a dictionary where the keys are the current column names, and the values are the new column names you want to assign. In the example above, columns 'A', 'B', and 'C' are renamed to 'Column1', 'Column2', and 'Column3', respectively. The rename() method returns a new DataFrame with the updated column names. If you want to modify the original DataFrame in-place, you can set the inplace parameter to True:

```
df.rename(columns={'A': 'Column1', 'B': 'Column2', 'C': 'Column3'},
inplace=True)
```

This will modify the original DataFrame df with the new column names.

Q5. How do you drop columns in a Pandas DataFrame?

In Pandas, you can drop columns from a DataFrame using the drop() function. Here's how you can do it:

```
import pandas as pd

# Create a DataFrame

data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}

df = pd.DataFrame(data)

# Display the original DataFrame

print("Original DataFrame:")

print(df)

# Drop a single column

df = df.drop('B', axis=1)

# Display the DataFrame after dropping the column

print("DataFrame after dropping 'B':")

print(df)

# Drop multiple columns

columns_to_drop = ['A', 'C']

df = df.drop(columns_to_drop, axis=1)

# Display the DataFrame after dropping multiple columns

print("DataFrame after dropping 'A' and 'C':")

print(df)
```

Output:

less

Copy code

```
Original DataFrame:
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
DataFrame after dropping 'B':
   A  C
0  1  7
1  2  8
2  3  9
DataFrame after dropping 'A' and 'C':
Empty DataFrame
Columns: []
Index: [0, 1, 2]
```

In the drop() function, the axis=1 argument is used to specify that we want to drop columns. By default, axis=0 is used to drop rows. The function returns a new DataFrame with the specified columns dropped, leaving the original DataFrame unchanged. If you want to modify the DataFrame in-place without creating a new DataFrame, you can set the inplace=True argument in the drop() function.

Q6. How do you find the unique values in a column of a Pandas DataFrame?

To find the unique values in a column of a Pandas DataFrame, you can use the unique() function. Here's an example of how to do it:

```
import pandas as pd

# Assuming you have a DataFrame called 'df' and a column named 'column_name'

unique_values = df['column_name'].unique()
```

```
# Print the unique values

print(unique_values)
```

In the above code, df['column_name'] refers to the specific column you want to find the unique values for. The unique() function returns an array-like object containing all the unique values in that column. You can then print or further manipulate the unique values as needed.

Q7. How do you find the number of missing values in each column of a Pandas DataFrame

To find the number of missing values in each column of a Pandas DataFrame, you can use the isnull() function to identify the missing values and then sum them up column-wise.

Here's an example of how you can do it:

```
import pandas as pd

# Create a sample DataFrame

df = pd.DataFrame({'A': [1, 2, None, 4],

                   'B': [5, None, None, 8],

                   'C': [9, 10, 11, 12]})

# Count the missing values in each column

missing_values = df.isnull().sum()

print(missing_values)
```

Output:

css

Copy code

```
A    1

B    2

C    0

dtype: int64
```

In the above example, the isnull() function returns a DataFrame of the same shape as df, with True values where the corresponding element is missing and False otherwise. Then, the sum() function is used to sum up the True values (which are treated as 1) column-wise, resulting in a Series that shows the number of missing values in each column.

Q 8 How do you fill missing values in a Pandas DataFrame with a specific value?

To fill missing values in a Pandas DataFrame with a specific value, you can use the fillna() method. This method allows you to replace NaN or missing values with a given value or strategy of your choice.

Here's an example of how you can fill missing values with a specific value in a Pandas DataFrame:

```
import pandas as pd

# Create a sample DataFrame

data = {'A': [1, 2, None, 4, None],

        'B': [5, None, 7, 8, 9],

        'C': [None, 11, 12, None, 14]}

df = pd.DataFrame(data)

# Fill missing values with a specific value

df_filled = df.fillna(0)  # Replace NaN with 0

print(df_filled)
```

Output:

css

Copy code

```
     A    B     C
0  1.0  5.0   0.0
1  2.0  0.0  11.0
2  0.0  7.0  12.0
3  4.0  8.0   0.0
4  0.0  9.0  14.0
```

Q9. How do you concatenate two Pandas DataFrames?

To concatenate two Pandas DataFrames, you can use the concat function provided by Pandas. The concat function allows you to concatenate DataFrames vertically (along rows) or horizontally (along columns). Here's an example of how you can use it:

```
import pandas as pd

# Create two sample DataFrames

df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]})

# Concatenate vertically (along rows)

result = pd.concat([df1, df2])

print(result)
```

Output:

css

Copy code

```
   A   B
0  1   4
1  2   5
2  3   6
0  7  10
1  8  11
2  9  12
```

In the example above, concat is used to vertically concatenate df1 and df2, resulting in a new DataFrame result that contains the rows from both df1 and df2.

You can also specify the axis parameter to concatenate the DataFrames horizontally (along columns). Here's an example:

```
import pandas as pd

# Create two sample DataFrames

df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
```

```
df2 = pd.DataFrame({'C': [7, 8, 9], 'D': [10, 11, 12]})

# Concatenate horizontally (along columns)

result = pd.concat([df1, df2], axis=1)

print(result)
```

Output:

css

Copy code

```
   A  B  C   D
0  1  4  7  10
1  2  5  8  11
2  3  6  9  12
```

Q10. How do you merge two Pandas DataFrames on a specific column

To merge two Pandas DataFrames on a specific column, you can use the merge() function. The merge() function allows you to combine DataFrames based on one or more common columns. Here's an example of how to merge two DataFrames on a specific column:

```
import pandas as pd

# Create two sample DataFrames

df1 = pd.DataFrame({'ID': [1, 2, 3, 4],
                    'Name': ['John', 'Alice', 'Bob', 'Jane']})

df2 = pd.DataFrame({'ID': [1, 3, 4, 5],
                    'Age': [25, 30, 35, 40]})

# Merge the DataFrames based on the 'ID' column

merged_df = pd.merge(df1, df2, on='ID')

print(merged_df)
```

Output:

Copy code

```
   ID  Name  Age
```

```
0   1  John   25
1   3   Bob   30
2   4  Jane   35
```

Q11. How do you group data in a Pandas DataFrame by a specific column and apply an aggregation function?

To group data in a Pandas DataFrame by a specific column and apply an aggregation function, you can use the groupby() function followed by the desired aggregation function.

Here's the general syntax:

```
df.groupby('column_name').agg(aggregation_function)
```

Q12. How do you pivot a Pandas DataFrame?

To pivot a Pandas DataFrame, you can use the pivot() function. The pivot() function allows you to reshape your DataFrame by converting unique values from one column into multiple columns, and reorganizing the data accordingly. Here's how you can pivot a DataFrame:

```
import pandas as pd
# Create a sample DataFrame
data = {
    'City': ['New York', 'New York', 'Los Angeles', 'Los Angeles', 'Chicago', 'Chicago'],
    'Year': [2019, 2020, 2019, 2020, 2019, 2020],
    'Temperature': [25, 30, 28, 33, 20, 22]
}
df = pd.DataFrame(data)

# Pivoting the DataFrame
pivot_df = df.pivot(index='City', columns='Year', values='Temperature')
print(pivot_df)
```

Output:

sql

Copy code

```
Year        2019  2020
City
Chicago       20    22
Los Angeles   28    33
New York      25    30
```

Q13. How do you change the data type of a column in a Pandas DataFrame?

To change the data type of a column in a Pandas DataFrame, you can use the astype() method. This method allows you to specify the desired data type for the column. Here's an example:

```
import pandas as pd
# Create a sample DataFrame
data = {'col1': [1, 2, 3], 'col2': [4.5, 5.5, 6.5], 'col3': ['7', '8', '9']}
df = pd.DataFrame(data)
# Display the original DataFrame
print("Original DataFrame:")
print(df)
# Change the data type of 'col1' from int to float
df['col1'] = df['col1'].astype(float)
# Change the data type of 'col3' from string to int
df['col3'] = df['col3'].astype(int)
# Display the modified DataFrame
print("\nModified DataFrame:")
print(df)
```

Output:

```
Original DataFrame:
   col1  col2 col3
0     1   4.5    7
1     2   5.5    8
2     3   6.5    9
Modified DataFrame:
   col1  col2  col3
0   1.0   4.5     7
1   2.0   5.5     8
2   3.0   6.5     9
```

Q14. How do you sort a Pandas DataFrame by a specific column?

To sort a Pandas DataFrame by a specific column, you can use the sort_values() function. Here's how you can do it:

```
import pandas as pd
# Create a sample DataFrame
data = {'Name': ['John', 'Emily', 'Sam', 'Jessica'],
        'Age': [25, 30, 18, 21],
        'City': ['New York', 'London', 'Paris', 'Berlin']}
df = pd.DataFrame(data)
# Sort the DataFrame by the 'Age' column in ascending order
df_sorted = df.sort_values('Age')
print(df_sorted)
```

output:

```
     Name  Age      City
2     Sam   18     Paris
3  Jessica   21    Berlin
0    John   25  New York
1   Emily   30    London
```

By default, sort_values() sorts the DataFrame in ascending order based on the specified column. If you want to sort in descending order, you can pass the ascending=False parameter:

```
df_sorted_desc = df.sort_values('Age', ascending=False)
```

Q14. How do you sort a Pandas DataFrame by a specific column?

To sort a Pandas DataFrame by a specific column, you can use the sort_values() function. Here's how you can do it:

```
import pandas as pd

# Create a sample DataFrame

data = {'Name': ['John', 'Emily', 'Sam', 'Jessica'],

        'Age': [25, 30, 18, 21],

        'City': ['New York', 'London', 'Paris', 'Berlin']}

df = pd.DataFrame(data)

# Sort the DataFrame by the 'Age' column in ascending order

df_sorted = df.sort_values('Age')

print(df_sorted)
```

output:

```
     Name  Age      City

2     Sam   18     Paris

3  Jessica   21    Berlin

0    John   25  New York

1   Emily   30    London
```

By default, sort_values() sorts the DataFrame in ascending order based on the specified column. If you want to sort in descending order, you can pass the ascending=False parameter:

```
df_sorted_desc = df.sort_values('Age', ascending=False)
```

Q15. How do you create a copy of a Pandas DataFrame?

To create a copy of a Pandas DataFrame, you can use the copy() method. Here's how you can do it:

```
import pandas as pd
# Creating a DataFrame
data = {'Name': ['John', 'Emma', 'Mike'],
        'Age': [25, 28, 30],
        'City': ['New York', 'London', 'Paris']}
df_original = pd.DataFrame(data)
# Creating a copy of the DataFrame
df_copy = df_original.copy()
# Modifying the copy
df_copy['Age'] = [26, 29, 31]
# Verifying the original DataFrame is not affected
print(df_original)
print(df_copy)
```

Output:

```
   Name  Age      City
0  John   25  New York
1  Emma   28    London
2  Mike   30     Paris
   Name  Age      City
0  John   26  New York
1  Emma   29    London
2  Mike   31     Paris
```

In the above example, df_original is the original DataFrame, and df_copy is the copy. Modifying df_copy does not affect the original DataFrame df_original.

Q16. How do you filter rows of a Pandas DataFrame by multiple conditions?

To filter rows of a Pandas DataFrame by multiple conditions, you can use logical operators like & (and) and | (or) to combine multiple conditions. Here's an example of how you can filter rows based on multiple conditions:

```
import pandas as pd
# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'Dave'],
        'Age': [25, 30, 35, 40],
        'City': ['New York', 'London', 'Paris', 'Tokyo'],
        'Salary': [50000, 60000, 70000, 80000]}
df = pd.DataFrame(data)
# Filter rows based on multiple conditions
filtered_df = df[(df['Age'] > 30) & (df['Salary'] > 60000)]
print(filtered_df)
```

Output:

markdown

Copy code

```
      Name  Age   City  Salary
2  Charlie   35  Paris   70000
3     Dave   40  Tokyo   80000
```

Q17. How do you calculate the mean of a column in a Pandas DataFrame?

To calculate the mean of a column in a Pandas DataFrame, you can use the mean() function. Here's an example:

```
import pandas as pd
# Create a sample DataFrame
data = {'Name': ['John', 'Jane', 'Mike', 'Sara'],
        'Age': [25, 30, 35, 40],
        'Salary': [50000, 60000, 70000, 80000]}
df = pd.DataFrame(data)
# Calculate the mean of the 'Salary' column
mean_salary = df['Salary'].mean()
print(mean_salary)
```

Q18. How do you calculate the standard deviation of a column in a Pandas DataFrame?

To calculate the standard deviation of a column in a Pandas DataFrame, you can use the std() method. Here's an example:

python

Copy code

```
import pandas as pd

# Create a sample DataFrame

data = {'Column1': [1, 2, 3, 4, 5],

        'Column2': [6, 7, 8, 9, 10]}

df = pd.DataFrame(data)

# Calculate the standard deviation of 'Column1'

std_dev = df['Column1'].std()

print("Standard Deviation:", std_dev)
```

To calculate the correlation between two columns in a Pandas DataFrame, you can use the corr() function. Here's an example of how to do it:

```
import pandas as pd

# Create a DataFrame

data = {'Column1': [1, 2, 3, 4, 5],

        'Column2': [2, 4, 6, 8, 10]}

df = pd.DataFrame(data)

# Calculate the correlation

correlation = df['Column1'].corr(df['Column2'])

print("Correlation:", correlation)
```

Q20. How do you select specific columns in a DataFrame using their labels?

To select specific columns in a DataFrame using their labels, you can use the indexing operator ([]) or the loc accessor. Here's how you can do it:

Using the indexing operator ([]):

```
import pandas as pd
```

```
# Create a DataFrame

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})

# Select specific columns using labels

selected_columns = df[['A', 'C']]

print(selected_columns)
```

21. How do you select specific rows in a DataFrame using their indexes?

To select specific rows in a DataFrame using their indexes, you can use the loc or iloc accessor in pandas. The loc accessor is used for label-based indexing, while the iloc accessor is used for integer-based indexing.

Here's how you can use loc to select specific rows by their indexes:

```
import pandas as pd

# Assuming you have a DataFrame called 'df'

# Select a single row by index label

selected_row = df.loc[3]

# Select multiple rows by index labels

selected_rows = df.loc[[1, 3, 5]]

# Select a range of rows by index labels

selected_range = df.loc[2:5]  # inclusive range

# Select rows based on a condition

condition = df['column_name'] > 10

selected_rows_condition = df.loc[condition]
```

22. How do you sort a DataFrame by a specific column?

To sort a DataFrame by a specific column, you can use the sort_values() function in pandas. Here's an example of how to do it:

```
import pandas as pd

# Create a sample DataFrame

data = {'Name': ['John', 'Mike', 'Sarah', 'Jessica'],

        'Age': [25, 32, 18, 41],
        'City': ['New York', 'London', 'Paris', 'Tokyo']}

df = pd.DataFrame(data)

# Sort the DataFrame by the 'Age' column in ascending order

sorted_df = df.sort_values('Age')

# Print the sorted DataFrame

print(sorted_df)
```

Q23. How do you create a new column in a DataFrame based on the values of another column?

To create a new column in a DataFrame based on the values of another column, you can use the following steps:

Access the column you want to base the new column on. Let's say the column you want to use is called "original_column".

Use the column's values to perform the desired computation or transformation.

Assign the computed values to a new column in the DataFrame.

Here's an example in Python using the pandas library:

```
import pandas as pd

# Create a sample DataFrame

data = {'original_column': [1, 2, 3, 4, 5]}

df = pd.DataFrame(data)

# Create a new column based on the values of 'original_column'

df['new_column'] = df['original_column'] * 2

# Display the updated DataFrame

print(df)
```

Q24. How do you remove duplicates from a DataFrame?

To remove duplicates from a DataFrame, you can use the drop_duplicates() method in pandas. This method returns a new DataFrame with the duplicates removed. Here's an example of how to use it:

```
import pandas as pd

# Create a sample DataFrame with duplicates
```

```
data = {'col1': [1, 1, 2, 2, 3, 3],
        'col2': ['a', 'a', 'b', 'b', 'c', 'c']}
df = pd.DataFrame(data)
# Remove duplicates
df_no_duplicates = df.drop_duplicates()
# Print the resulting DataFrame
print(df_no_duplicates)
```

Q25. What is the difference between .loc and .iloc in Pandas?

In Pandas, both .loc and .iloc are attribute accessors used for indexing and slicing data in a DataFrame. However, they differ in the way they handle the indexing.

.loc is primarily label-based and is used to access data based on label or boolean indexing. It accepts a label or a boolean array as an input to select rows and columns. When using .loc, the indexing is inclusive of both the start and end points. For example:

python

Copy code

```
df.loc[row_label, column_label]
df.loc[row_label]
df.loc[boolean_array]
```

Here, row_label and column_label can be single labels, lists, or slices, while boolean_array is a boolean array of the same length as the DataFrame's index.

On the other hand, .iloc is primarily integer-based and is used to access data based on integer positions or integer array indexing. It accepts integer values or integer arrays as inputs to select rows and columns. The indexing with .iloc is exclusive of the end point, similar to Python's slicing convention. For example:

python

Copy code

```
df.iloc[row_index, column_index]
df.iloc[row_index]
df.iloc[boolean_array]
```

Here, row_index and column_index can be single integers, lists, or slices, and boolean_array is a boolean array of the same length as the DataFrame's index.

To summarize, the main difference between .loc and .iloc is the way they handle indexing. .loc uses label-based indexing, while .iloc uses integer-based indexing.