**pd.merge how parameter options and explaination**

The how parameter in the pd.merge() function in Pandas specifies how to determine which keys are to be included in the resulting DataFrame. The how parameter has several options that control the type of set operation that will be performed during the merge. Here are the common options for the how parameter and their explanations:

1. **how='inner'**: This option performs an inner join between the two DataFrames. It returns only the rows where the merge key exists in both DataFrames. If a key is present in one DataFrame but not the other, the resulting DataFrame will not include that row.
2. **how='outer'**: This option performs a full outer join between the two DataFrames. It returns all rows from both DataFrames and fills in missing values with NaN where data is not available in one of the DataFrames.
3. **how='left'**: This option performs a left join between the two DataFrames. It returns all rows from the left DataFrame and the matched rows from the right DataFrame. If there is no match, the result will contain NaN values for the columns from the right DataFrame.
4. **how='right'**: This option performs a right join between the two DataFrames. It returns all rows from the right DataFrame and the matched rows from the left DataFrame. If there is no match, the result will contain NaN values for the columns from the left DataFrame.
5. **how='cross'**: This option performs a cross join or Cartesian product between the two DataFrames. It combines all rows from the left DataFrame with all rows from the right DataFrame, resulting in a DataFrame with a length equal to the product of the lengths of the two DataFrames.

By specifying the how parameter in the pd.merge() function, you can control how the merge operation is performed and which rows are included in the resulting DataFrame based on the keys present in the DataFrames being merged.

**.merge parameters explaination**

**When using the .merge() method in pandas to combine DataFrames, there are several parameters you can use to customize the merge operation. Here are some of the key parameters:**

1. **left**:
   * The left DataFrame to be merged.
2. **right**:
   * The right DataFrame to be merged.
3. **how**:
   * Specifies the type of join to perform. Options include:
     + 'left': Use keys from the left DataFrame only.
     + 'right': Use keys from the right DataFrame only.
     + 'inner': Use intersection of keys from both DataFrames.
     + 'outer': Use union of keys from both DataFrames.
4. **on**:
   * Column or index level names to join on. Must be found in both DataFrames.
5. **left\_on** and **right\_on**:
   * Columns or index levels from the left and right DataFrames to use as keys for the merge.
6. **left\_index** and **right\_index**:
   * Use the index from the left or right DataFrame as the join key.
7. **suffixes**:
   * A tuple of string suffixes to apply to overlapping column names in the left and right DataFrames.
8. **indicator**:
   * Adds a special column to the merged DataFrame called "\_merge" that indicates the source of each row.
9. **validate**:
   * Checks if the merge is a valid merge. Options include 'one\_to\_one', 'one\_to\_many', 'many\_to\_one', or 'many\_to\_many'.

These parameters give you flexibility in how you merge DataFrames, allowing you to specify the type of join, the columns to join on, whether to use indices for joining, and more. By understanding and using these parameters effectively, you can perform complex data merges and manipulations in pandas with ease.

**how to use .groupby**

To use the .groupby() method in pandas, you typically follow these steps:

1. **Import Pandas**: Make sure you have pandas imported in your Python script or Jupyter notebook.
2. import pandas as pd
3. **Load Your Data**: Load your data into a pandas DataFrame.
4. df = pd.read\_csv('your\_data.csv')
5. **Group Your Data**: Use the .groupby() method on your DataFrame to create a GroupBy object. You can specify the column or columns you want to group by.
6. grouped = df.groupby('column\_name')
7. **Aggregate Data**: Once you have grouped your data, you can perform aggregate functions on the groups. For example, you can calculate the mean, sum, count, etc.
8. mean\_values = grouped.mean()
9. sum\_values = grouped.sum()
10. count\_values = grouped.size()
11. **Accessing Groups**: You can access individual groups using the get\_group() method.
12. group = grouped.get\_group('group\_name')
13. **Iterating Over Groups**: You can iterate over the groups using a for loop.
14. for name, group in grouped:
15. print(name)
16. print(group)
17. **Resetting Index**: You can reset the index of the resulting DataFrame using the reset\_index() method.
18. grouped\_mean = grouped.mean().reset\_index()

By following these steps, you can effectively use the .groupby() method in pandas to group and analyze your data based on specific criteria.

***Found count for each timepoint for each mouse. Trying to isolate only info for duplicate mouse.***

**debug duplicate\_mouse\_count = unique\_mousetime.value\_counts() duplicate\_mouse\_count.head(10) If duplicate\_mouse\_count>1 == True: print(duplicate\_mouse\_count['Mouse ID'])**

It seems like you are trying to find and display the mouse IDs with duplicate time points in your dataset. However, the code snippet you provided is not entirely correct. Let me help you correct it:

duplicate\_mouse\_count = unique\_mousetime.value\_counts()

duplicate\_mouse\_count = duplicate\_mouse\_count[duplicate\_mouse\_count > 1]

if not duplicate\_mouse\_count.empty:

print("Mouse IDs with duplicate time points:")

print(duplicate\_mouse\_count)

This code will find and display the mouse IDs that have duplicate time points in your dataset.

***Trying to filter the original df to only show the identified duplicate’s values for optional step. Issue with the syntax I’m using for .query***

**debug unique\_mousetime.query('Mouse ID' != 'g989')**

In your code snippet, the condition 'Mouse ID' != 'g989' is not correctly specified within the .query() method. You should use the correct syntax to compare the 'Mouse ID' column values with 'g989'. Here is the corrected code:

unique\_mousetime\_filtered = unique\_mousetime.query("`Mouse ID` != 'g989'")

In this corrected code:

* We use backticks (`) around the column name 'Mouse ID' to handle spaces in the column name.
* We compare the 'Mouse ID' column values with 'g989' to filter out the rows where the 'Mouse ID' is 'g989'.

Please try running this corrected code to filter out the rows with 'Mouse ID' equal to 'g989' from your DataFrame.

**Trouble with syntax for reset index for more than 1 column**  
**reset index to multiple columns**

To reset the index to multiple columns in a DataFrame, you can pass a list of column names to the reset\_index() method. Here's how you can reset the index to multiple columns:

# Assuming df is your DataFrame

df.reset\_index(level=['column1', 'column2'], inplace=True)

In this code snippet:

* Replace 'column1' and 'column2' with the names of the columns you want to set as the index.
* The inplace=True parameter ensures that the DataFrame is modified in place without creating a new DataFrame.

After running this code, the index of the DataFrame will be reset to the specified columns.

**Originally did filtered\_df = combined\_metadata.query("`Mouse ID` != 'g989'") but search into methods to drop values from a DF led me to alt of filtered\_df = combined\_metadata[combined\_metadata['Mouse ID'] != 'g989']. Wanted to know difference between the two (namely if the query version actually dropped the value or if it had ghost ver of it. Because when I did query of new DF done with .query, still found value but nothing showed up as values for that mouse**

**Both filtered\_df = combined\_metadata.query("Mouse ID != 'g989'") and filtered\_df = combined\_metadata[combined\_metadata['Mouse ID'] != 'g989'] achieve the same result of filtering out rows where the 'Mouse ID' is 'g989'. However, there are some differences in how these two methods work:**

1. **query() method:**
   * The query() method in pandas allows you to filter a DataFrame using a SQL-like syntax with string expressions.
   * It provides a more concise and readable way to write filtering conditions compared to traditional indexing.
   * It can be more flexible when dealing with complex filtering conditions involving multiple columns and logical operators.

Example:

filtered\_df = combined\_metadata.query("`Mouse ID` != 'g989'")

1. **Boolean indexing:**
   * Using boolean indexing with square brackets is a common and standard way to filter DataFrames in pandas.
   * It directly accesses the DataFrame columns using column names and applies the filtering condition.
   * It is more explicit and familiar to users who are used to working with pandas DataFrames.

Example:

filtered\_df = combined\_metadata[combined\_metadata['Mouse ID'] != 'g989']

Both methods are valid and widely used in pandas for filtering DataFrames. You can choose the one that you find more comfortable and readable based on your preference and the complexity of your filtering conditions.

**If you are observing that the count of unique mouse IDs remains the same after filtering out the rows where 'Mouse ID' is 'g989', it could be due to the presence of duplicate mouse IDs in the original DataFrame. When filtering out rows based on a specific condition, the count of unique mouse IDs may not change if there are duplicate entries for the same mouse ID.**

To ensure that the count of unique mouse IDs reflects the filtering operation correctly, you can follow these steps:

1. Check for and remove any duplicate rows based on 'Mouse ID' and 'Timepoint' before filtering.
2. Filter out the rows where 'Mouse ID' is 'g989'.
3. Verify the count of unique mouse IDs after filtering.

Here is an example of how you can perform these steps:

import pandas as pd

# Assuming combined\_metadata is your DataFrame

# Drop duplicates based on 'Mouse ID' and 'Timepoint'

combined\_metadata.drop\_duplicates(subset=['Mouse ID', 'Timepoint'], inplace=True)

# Filter out rows where 'Mouse ID' is 'g989'

filtered\_df = combined\_metadata[combined\_metadata['Mouse ID'] != 'g989']

# Check the count of unique mouse IDs after filtering

unique\_mouse\_ids = filtered\_df['Mouse ID'].nunique()

print("Number of unique mouse IDs after filtering:", unique\_mouse\_ids)

By removing duplicates and then filtering out the rows where 'Mouse ID' is 'g989', you should see the count of unique mouse IDs change accordingly in the filtered DataFrame.

**how to make Mouse ID = 'g989' not show up when query or filtering**

To exclude rows where 'Mouse ID' is 'g989' when querying or filtering a DataFrame, you can use the following approaches:

1. Using the query() method:

filtered\_df = combined\_metadata.query("`Mouse ID` != 'g989'")

1. Using boolean indexing with square brackets:

filtered\_df = combined\_metadata[combined\_metadata['Mouse ID'] != 'g989']

Both of these methods will filter out rows where 'Mouse ID' is 'g989' from the DataFrame. After applying either of these filters, 'Mouse ID' with the value 'g989' should not show up in the resulting DataFrame filtered\_df.

**above g989 still shows up when query in filtered\_df but with blank values**

If 'Mouse ID' with the value 'g989' is showing up in the filtered\_df DataFrame with blank values after filtering, it might be due to the presence of missing or NaN values in other columns for that specific 'Mouse ID'. To exclude rows with 'Mouse ID' as 'g989' completely from the DataFrame, including those with missing values, you can drop those rows entirely. Here's how you can do it:

# Drop rows where 'Mouse ID' is 'g989' including rows with missing values

filtered\_df = combined\_metadata[combined\_metadata['Mouse ID'] != 'g989'].dropna(subset=['Mouse ID'])

In this code snippet:

* The filter combined\_metadata['Mouse ID'] != 'g989' excludes rows where 'Mouse ID' is 'g989'.
* The dropna(subset=['Mouse ID']) method is used to drop any rows with missing values (NaN) in the 'Mouse ID' column to ensure that rows with 'Mouse ID' as 'g989' are completely removed from the DataFrame.

After applying this combined filter and dropping missing values, the resulting filtered\_df DataFrame should not contain any rows with 'Mouse ID' as 'g989', including those with blank values.

**In the code snippet you provided, there seems to be a small syntax error in the filtering condition. The correct way to filter out rows where** **'Mouse ID' is 'g989' is to use parentheses to group the condition properly. Here is the corrected code snippet:**

# Drop rows where 'Mouse ID' is 'g989' including rows with missing values

filtered\_df = combined\_metadata[(combined\_metadata['Mouse ID'] != 'g989')].dropna(subset=['Mouse ID'])

By using parentheses around the condition (combined\_metadata['Mouse ID'] != 'g989'), you ensure that the comparison is applied correctly before dropping any missing values in the 'Mouse ID' column. This should filter out rows where 'Mouse ID' is 'g989' and then remove any rows with missing 'Mouse ID' values.

After applying this corrected filter and dropping missing values, the resulting filtered\_df DataFrame should accurately reflect the total number of unique mice, excluding 'Mouse ID' as 'g989'.

**… I had forgotten to write mice2 in one place, hence incorrect values :/ my code was fine :/**

**Forgot how to calculate sem (std error) for values**

**how to calculate sem of groupedby object**

To calculate the Standard Error of the Mean (SEM) for a grouped DataFrame in pandas, you can use the sem() function after grouping the data. Here's an example of how you can calculate the SEM for a grouped DataFrame in pandas:

import pandas as pd

# Create a sample DataFrame

data = {

'group': ['A', 'A', 'B', 'B', 'B', 'C'],

'value': [10, 15, 20, 25, 30, 35]

}

df = pd.DataFrame(data)

# Group the DataFrame by 'group'

grouped\_df = df.groupby('group')

# Calculate the SEM for each group

sem\_values = grouped\_df['value'].sem()

print("SEM for each group:")

print(sem\_values)

In this example:

1. We create a sample DataFrame with two columns: 'group' and 'value'.
2. We group the DataFrame by the 'group' column using groupby().
3. We then calculate the SEM for the 'value' column within each group using the sem() function.

The sem() function calculates the standard error of the mean for each group in the grouped DataFrame. The output will be a Series with the SEM values for each group.

**Use aggregation method to produce summary stats I already did**  
**In pandas, you can use the agg() method to perform aggregation for summary statistics on a DataFrame. The agg() method allows you to apply multiple aggregation functions to different columns of the DataFrame simultaneously.**

**Here is an example of how you can use the agg() method to calculate multiple summary statistics for different columns in a DataFrame:**

import pandas as pd

# Create a sample DataFrame

data = {

'A': [1, 2, 3, 4, 5],

'B': [10, 20, 30, 40, 50],

'C': [100, 200, 300, 400, 500]

}

df = pd.DataFrame(data)

# Calculate multiple summary statistics using agg()

summary\_stats = df.agg({

'A': ['mean', 'median', 'std'], # Mean, Median, Standard Deviation for column 'A'

'B': ['var', 'sem'], # Variance, Standard Error of the Mean for column 'B'

'C': ['min', 'max'] # Minimum and Maximum for column 'C'

})

print("Summary Statistics:")

print(summary\_stats)

In this example:

1. We create a sample DataFrame with columns 'A', 'B', and 'C'.
2. We use the agg() method on the DataFrame df to calculate multiple summary statistics for different columns.
3. We specify the columns and the aggregation functions we want to apply in a dictionary format within the agg() method.

The agg() method allows you to calculate various summary statistics such as mean, median, standard deviation, variance, standard error of the mean, minimum, maximum, etc., for different columns in a DataFrame in a single operation.

Also found: <https://stackoverflow.com/questions/55388610/how-to-calculate-aggregated-summary-statistics-in-pandas-dataframe>

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.aggregate.html#pandas-dataframe-aggregate>

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.agg.html>

**issues with syntax of agg method**

**debug filtered\_df.groupby(['Drug Regimen','Tumor Volume (mm3)']).agg(['mean','median','var','std','sem'])**

It looks like you are trying to calculate summary statistics for the 'Tumor Volume (mm3)' column grouped by 'Drug Regimen'. The code you provided seems to have a small issue. The correct way to calculate these statistics would be as follows:

summary\_stats = filtered\_df.groupby('Drug Regimen')['Tumor Volume (mm3)'].agg(['mean', 'median', 'var', 'std', 'sem'])

This code will group the DataFrame by 'Drug Regimen', then calculate the mean, median, variance, standard deviation, and standard error of the mean for the 'Tumor Volume (mm3)' column.

**Sources for plotting help**

**Bar:**

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.core.groupby.DataFrameGroupBy.plot.html#pandas.core.groupby.DataFrameGroupBy.plot>

<https://pandas.pydata.org/pandas-docs/stable/getting_started/intro_tutorials/04_plotting.html>

**Pie:**

[same as above]

**parameters for pandas plot(kind='pie')**

When using the DataFrame.plot() method in Pandas with kind='pie' to create a pie chart, you can pass several parameters to customize the appearance of the chart. Here are some common parameters you can use:

1. figsize: Tuple specifying the width and height of the figure in inches.
2. title: A string to set the title of the plot.
3. autopct: A string or function used to label the wedges with their numeric value. The default is None, which means no labels.
4. labels: A list or array to label the wedges with a specific list of strings.
5. colors: A list or array of colors for the wedges.
6. explode: A list or array to specify the fraction of the radius with which to offset each wedge.
7. startangle: The angle by which the start of the pie is rotated, counterclockwise from the x-axis.
8. shadow: Boolean value to draw a shadow beneath the pie.
9. legend: Boolean value to display a legend.

These are some of the parameters you can use with kind='pie' in Pandas. You can experiment with these parameters to customize your pie chart as needed.

**above how to add percentage to pie chart**

To add percentage labels to a pie chart created using Pandas DataFrame.plot() with kind='pie', you can use the autopct parameter. The autopct parameter allows you to specify a string or a function to format the percentage labels that appear on each wedge of the pie chart.

Here's an example of how you can add percentage labels to a pie chart using Pandas:

import pandas as pd

# Create a DataFrame with some data

data = {'Flavor': ['Apple', 'Cherry', 'Blueberry', 'Pumpkin'],

'Votes': [350, 200, 150, 100]}

df = pd.DataFrame(data)

# Create a pie chart with percentage labels

df.plot(kind='pie', y='Votes', labels=df['Flavor'], autopct='%1.1f%%')

In this example:

* y='Votes' specifies the column in the DataFrame to use for the pie chart values.
* labels=df['Flavor'] sets the labels for each wedge of the pie chart.
* autopct='%1.1f%%' formats the percentage labels to show one decimal place.

By specifying autopct='%1.1f%%', the percentage labels will display with one decimal place. You can adjust the formatting string in autopct to change the appearance of the percentage labels on the pie chart.