How to create frequency table based on multiple columns in pandas

Authored by

stats writer

November 21, 2025

Generating a Frequency table is a fundamental step in exploratory data analysis, providing critical insights into the distribution and occurrence of categorical variables within a dataset. When working with complex, real-world data, analysts often need to understand the joint frequency of observations across multiple dimensions simultaneously. Leveraging the powerful capabilities of the pandas library in Python simplifies this process dramatically, allowing for the quick calculation and structured presentation of these combined counts. While functions like `pd.crosstab` are often utilized for generating cross-tabulations, the `value_counts()` method, applied to a subset of columns, offers a highly efficient and intuitive way to generate a frequency series based on the unique combinations found across those chosen columns.

This comprehensive guide details the process of constructing such multi-column frequency distributions using `value_counts()`. We will explore the initial syntax, delve into practical examples using sample sports data, and demonstrate advanced techniques for transforming the output into a cleanly formatted DataFrame, ready for further statistical analysis or visualization. Understanding how to precisely count unique pairings of values--such as team and position, or state and product category--is essential for any data professional seeking to extract meaningful patterns from high-dimensional datasets. This methodology ensures that every unique intersection of the specified column values is accurately counted and presented in a hierarchical structure.

1. Understanding Frequency Tables in Data Analysis

A Frequency table, or frequency distribution, is an organized tabulation that summarizes the number of times each distinct value occurs within a dataset. In simple univariate analysis, this table reveals the distribution of a single variable, indicating common occurrences and outliers. However, the real complexity and utility emerge when we analyze the frequency across multiple variables simultaneously. This multivariate approach helps identify correlations, dependencies, or structural imbalances within the data, which is often crucial for making informed business or research decisions.

When we seek a frequency count based on multiple columns, we are essentially asking: "How many times does this specific combination of values (e.g., 'Team A' AND 'Position G') appear in the dataset?" pandas handles this by treating the combination of values from the specified columns as a single, composite key. The resulting frequency count then represents the total occurrence of that unique key combination. This is far more informative than running separate frequency counts for each column individually, especially in scenarios involving survey results, transactional data, or demographic studies where the interaction between features is paramount.

The primary benefit of using this technique is obtaining an immediate, quantifiable measure of co-occurrence. For instance, in analyzing sales data, counting the frequency of 'Region' and 'Product Type' allows an analyst to quickly determine which product types are most popular in specific

geographic regions. This insight is foundational for optimizing inventory, targeting marketing campaigns, and understanding market saturation. Furthermore, because pandas integrates seamlessly with Python's vast analytical ecosystem, the generated frequency series can easily be converted into visual formats, such as bar charts or heatmaps, to communicate findings effectively.

To implement this analysis within Python, specifically using the pandas library, the most straightforward approach involves utilizing the robust `value_counts()` method applied directly to the DataFrame. The syntax requires passing a list of column names to the function, indicating which dimensions should be included in the joint frequency calculation. The following basic syntax provides a clear visualization of how this function operates on a selected subset of columns:

df.value_counts()

The subsequent sections will demonstrate how to apply this precise syntax using a concrete example, illustrating the steps from initial data loading to generating the final, formatted frequency table. This step-by-step approach ensures clarity regarding input, process, and output interpretation.

2. Prerequisites and Initial Data Setup

Before executing the frequency counting operation, it is essential to ensure the necessary Python libraries are imported and the data is correctly loaded into a pandas DataFrame. The pandas library is the cornerstone of data manipulation in Python, providing the structural foundation required for efficient columnar data operations. Importing it conventionally as `pd` is standard practice, making subsequent function calls concise and readable. The setup phase is crucial, as errors here can prevent all downstream analytical steps from executing correctly.

For demonstration purposes, we will construct a sample dataset representing basketball player statistics. This dataset contains three columns: `team` (categorical, detailing the team affiliation), `position` (categorical, indicating the player's primary role), and `points` (numerical, showing scores). This structure is ideal for illustrating multi-column frequency analysis because both `team` and `position` contain repeated, discrete values whose joint occurrences we wish to quantify.

The following code snippet imports pandas, initializes the demonstration DataFrame, and displays the resulting data structure. Pay close attention to how the data is organized, particularly the combinations of 'A'/'G', 'B'/'F', etc., which will be the basis for our frequency analysis:

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team' : ,

'position' : ,
'points': })

#view DataFrame
print(df)

team position points
0 A G 24
1 A G 33
2 A G 20
3 A F 15
4 B G 16
5 B G 16
6 B F 29
7 B F 25

3. Generating a Multi-Column Frequency Series using value_counts()

The primary method for obtaining the joint frequencies of multiple columns in pandas is through the `value_counts()` function. When applied to a subset of columns (passed as a list), this function automatically calculates the count of every unique tuple formed by the values in those columns. It returns the result as a pandas Series, where the index is a MultiIndex representing the unique combinations, and the values are the corresponding counts.

To determine the frequency distribution across our `team` and `position` columns, we invoke `value_counts()` and pass as the argument. This operation internally groups the DataFrame rows by the unique pairings of these two variables and aggregates the size of each group. The resulting Series is inherently sorted in descending order by count, providing an immediate view of the most frequent combinations, which is often the most critical information required in initial data exploration.

Executing this command produces a clear, hierarchical output that directly answers our question regarding the co-occurrence of team and position. Observe the structure of the output below; the results show the frequency of each combination, indexed by the two columns we specified:

#count frequency of values in team and position columns
df.value_counts()

team position
A G 3
B F 2

G 2
A F 1
dtype: int64

4. Interpreting the Results: Analyzing the MultiIndex Output

The output generated by `value_counts()` when using multiple columns is a pandas Series indexed by a MultiIndex. Understanding this structure is key to interpreting the frequency data correctly. The MultiIndex allows for multiple levels of indexing, visually grouping similar entries. In the example above, `team` forms the outer index level, and `position` forms the inner index level. The integer values on the right represent the frequency count for that specific unique combination.

By examining the structured output, we can draw precise conclusions about the composition of our dataset. The structure clearly indicates the counts for every observed combination. For instance, the combination `(A, G)` appears 3 times, while the combination `(A, F)` appears only once. This hierarchical representation is highly efficient for displaying aggregated data, especially when dealing with many categorical variables. The descending order by count ensures that the most prominent patterns--those combinations that occur most frequently--are presented first, optimizing the time required for initial data assessment.

From the results of our basketball player DataFrame, the joint frequencies reveal specific details regarding team structure. We can summarize the findings directly from the output:

There are **3** total occurrences where the team is A and the position is G (Guard).
There are **2** total occurrences where the team is B and the position is F (Forward).
There are **2** total occurrences where the team is B and the position is G (Guard).
There is **1** total occurrence where the team is A and the position is F (Forward).

5. Converting the Frequency Series into a Clean DataFrame

While the MultiIndex Series output is highly functional for internal pandas operations, it can sometimes be less intuitive for external reporting, visualization tools, or further non-hierarchical data manipulation. For scenarios requiring a standard, flat tabular format, it is often necessary to convert this Series back into a regular pandas DataFrame. This conversion flattens the MultiIndex, promoting the index levels (our original columns) back into standard columns.

The transformation is easily achieved by chaining the `reset_index()` method immediately after the `value_counts()` call. The `reset_index()` function takes the current index (which is the MultiIndex structure containing `team` and `position`) and converts those levels into columns, assigning a default numerical index to the resulting DataFrame. This process standardizes the data

structure, making it more accessible for subsequent steps, such as filtering or calculating relative frequencies.

The resulting DataFrame will contain the two original frequency columns (`team` and `position`) and a third column containing the counts. Note that pandas automatically assigns the column name to the count column when using `reset_index()` on a Series, which is a detail we will address in the next section for better clarity.

#count frequency of values in team and position columns and return DataFrame
df.value_counts().reset_index()

team position 0
0 A G 3
1 B F 2
2 B G 2
3 A F 1

6. Refining the Output: Renaming the Count Column

The final step in preparing the DataFrame for presentation or external use is ensuring all columns have meaningful and descriptive names. As demonstrated previously, the count column generated by `reset_index()` receives the generic name `0`, which provides no context regarding the data it holds. Renaming this column to something explicit, like `count`, `frequency`, or `occurrence`, significantly enhances the readability and self-documentation of the resulting table.

To rename a column in pandas, we chain the `rename()` function onto the existing sequence of operations. This function accepts a dictionary argument, where the keys are the current column names (in this case, `0`) and the values are the desired new column names (e.g., `count`). By executing this step, we finalize the structure and labeling of our DataFrame, making it instantly understandable to any analyst or stakeholder.

The complete pipeline--from counting values to resetting the index and renaming the count column--represents the best practice for generating a clean, multidimensional frequency table in pandas. This final output is structurally sound and semantically clear, ready for deployment in reports or further aggregation tasks.

#get frequency of values in team and position column and rename count column
df.value_counts().reset_index().rename(columns={0:'count'})

team position count
0 A G 3

1 B F 2
2 B G 2
3 A F 1

7. Advanced Applications and Conclusion

The end result is a polished DataFrame that accurately contains the frequency of each unique combination of values in the `team` and `position` columns. This structured output is highly versatile. Once the frequency table is created, advanced data manipulation techniques can be applied. For example, one could easily calculate the percentage of total observations for each combination by dividing the `count` column by the total number of rows in the original DataFrame, providing a relative frequency distribution. Moreover, the DataFrame can be sorted by any column--perhaps alphabetically by team or position, or ascendingly by count--using the `sort_values()` method to further refine the presentation.

Furthermore, while this guide focused specifically on `value_counts()`, analysts often choose `pandas.crosstab()` for two-way frequency analyses, especially when generating contingency tables that require marginal subtotals or normalizing frequencies (percentages). However, `value_counts()` remains the most straightforward and fastest method for simply obtaining raw counts across multiple columns in a flattened Series structure, particularly beneficial when dealing with more than two categorical variables.

In summary, mastering the combination of `value_counts()`, `reset_index()`, and `rename()` provides a robust, efficient, and clean workflow for generating essential frequency tables from multi-dimensional data using pandas. This skill is foundational for exploratory data analysis, enabling data professionals to quickly summarize complex datasets and identify key distributional patterns.