Learning to Create Frequency Tables with Python

Authored by

Mohammed looti

November 8, 2025

A **frequency table** is an indispensable tool in descriptive statistics, serving to organize raw, unstructured data by clearly displaying the count of occurrences (the frequency) for different values or categories within a given dataset. This foundational organizational structure is crucial for initiating **exploratory data analysis** (EDA), as it immediately offers essential insights into the data's underlying distribution, identifies the central tendency, and helps pinpoint potential outliers or anomalies. By effectively summarizing complex data, frequency tables transform large datasets into manageable, actionable summaries that guide subsequent analytical steps.

For any data analysis project, understanding the precise distribution of variable values is paramount. It allows data scientists and analysts to accurately characterize their variables--determining whether they follow a normal distribution, exhibit skewness, or contain categories with sparse observations. When executing these analyses in **Python**, the industry-standard **pandas** library provides an exceptionally efficient and intuitive framework for generating these statistical tables. These methods are equally effective whether you are handling a simple one-dimensional array of data or conducting complex analyses across multivariate datasets.

This comprehensive tutorial focuses on the practical application of core pandas functions specifically designed to construct both one-way (univariate) and two-way (bivariate) frequency tables. We will thoroughly explore two primary functions: the highly focused **value_counts()** method, used primarily for analyzing individual Series objects, and the more robust and versatile **crosstab()** function, which is essential for general DataFrame frequency calculation and advanced contingency analysis.

Calculating Univariate Frequencies for a pandas Series

When the objective is to analyze a single column or a one-dimensional array of data--referred to in the pandas ecosystem as a **Series**--the most direct and idiomatic method for calculating frequencies is by employing the built-in **value_counts()** function. This method is specifically engineered for Series objects and rapidly generates a new Series whose index consists of the unique values found in the input data, and whose values contain the corresponding counts. By default, the output is conveniently sorted in descending order based on the frequency count, allowing analysts to instantly identify the most common occurrences or the data's mode with minimal effort.

To practically demonstrate this functionality, we begin by importing the necessary **pandas** library and defining a small sample Series containing several repeated integer values. Applying the **value_counts()** method directly to this dataset immediately produces the complete frequency distribution for every unique element. Observe in the output how the results are automatically ordered from the highest frequency count to the lowest, providing immediate insight into the distribution of the sample data:

import pandas as pd

#define Series
data = pd.Series()

#find frequencies of each value
data.value_counts()

3 4
1 3
4 2
5 1
2 1

Although sorting the results by frequency is the default and often preferred behavior, there are instances where analysts need the output ordered based on the original data value itself, rather than by the count. This is particularly relevant when the unique values represent naturally ordered categories, such as chronological dates or specific age groups. To achieve this alternative ordering, the function's behavior can be easily modified by setting the argument **sort=False**. This adjustment ensures that the categories in the resulting output maintain their natural numerical or alphabetical order, irrespective of how frequently they appear in the source data.

data.value_counts(sort=False)

1 3
2 1
3 4
4 2
5 1

The interpretation of these results is essential for accurately reading the generated **frequency table**: the index column identifies the specific category or unique value found in the source data, and the adjacent number provides the count of its total occurrences. Specifically, analyzing the counts obtained from the original Series reveals the following breakdown:

The value "1" occurs **3** times in the Series.
The value "2" occurs **1** time in the Series.
The value "3" occurs **4** times in the Series.
The value "4" occurs **2** times in the Series.
The value "5" occurs **1** time in the Series.

Generating Univariate Frequencies in a pandas DataFrame

When working within a multi-column dataset, known as a **DataFrame**, although the `value_counts()` method can certainly be applied to individual columns (which are Series objects), the pandas function **crosstab()** provides a more structured and powerful mechanism. This function is particularly useful as a consistent method, especially when analysts anticipate transitioning from simple univariate analysis to complex bivariate or contingency table creation. While its primary role is generating contingency tables, `crosstab()` is perfectly capable of calculating one-way frequencies by simply specifying only one variable of interest.

The fundamental syntax for the **crosstab()** function requires the definition of both the index and the column structure for the resulting table. For constructing a univariate frequency table, the variable of interest (the specific column chosen from the **DataFrame**) is designated as the **index**. A static placeholder name is then used for the columns to label the resulting count. The general functional structure is `crosstab(index, columns)`, where the `index` parameter receives the Series data we intend to count, and the `columns` parameter is conventionally set to a descriptive string like 'count' to clearly label the output frequencies.

To illustrate this, let us define a sample **DataFrame** containing hypothetical student data, including their 'Grade', 'Age', and 'Gender'. We will then utilize the `crosstab()` function to determine the frequency distribution of the categorical 'Grade' column across all ten records in the dataset. This demonstrates how easily this function adapts to standard DataFrame operations:

#create data
df = pd.DataFrame({'Grade': ,
'Age': ,
'Gender': })

#view data
df

Grade Age Gender
0 A 18 M
1 A 18 M
2 A 18 F
3 B 19 F
4 B 19 F
5 B 20 M
6 B 18 M
7 C 18 F
8 D 19 M

9 D 19 F

#find frequency of each letter grade
pd.crosstab(index=df, columns='count')

col_0 count
Grade
A 3
B 4
C 1
D 2

The resulting output is a newly constructed **DataFrame** that clearly maps the number of students who achieved each letter grade. In this structure, the 'Grade' categories become the index labels, while the corresponding frequency counts populate the 'count' column. Based on this precise **frequency table**, we can derive the following specific interpretations regarding the academic performance distribution:

3 students received an 'A' in the class.
4 students received a 'B' in the class.
1 student received a 'C' in the class.
2 students received a 'D' in the class.

The versatility of **crosstab()** means we can use this exact same syntax to swiftly calculate the frequency counts for any other column in the DataFrame. For instance, to examine the demographic distribution of student ages, we simply modify the column passed to the **index** parameter. This demonstrates the function's core power in handling diverse univariate counting tasks across a complex dataset:

pd.crosstab(index=df, columns='count')

col_0 count
Age
18 5
19 4
20 1

This final table effectively summarizes the age demographics of the student population, confirming the precise counts for each age group within the provided dataset. Specifically, this output indicates the following clear breakdown by age:

5 students are 18 years old.
4 students are 19 years old.
1 student is 20 years old.

Calculating Relative Frequencies and Proportions

While raw counts provide the absolute number of observations for each category, calculating **relative frequencies** is often a more powerful technique for interpretation, especially when the goal is to compare distributions across samples of different sizes. Relative frequency is achieved by converting the raw counts into proportions, expressing the percentage contribution of each category relative to the total population. These proportions always sum up precisely to 1 (or 100% if multiplied by 100). This normalization step is vital, representing the transition from simple descriptive counting to more sophisticated inferential statistical analysis.

Within the **pandas** environment, transforming a raw frequency count table into a table of relative frequencies is remarkably straightforward. This conversion is typically achieved by dividing the entire count table by its grand total sum. This process takes advantage of pandas' robust vectorized operations, which apply the division efficiently across every cell in the resulting table. Using the age frequency table generated in the previous section as our starting point, we first store the raw `crosstab` result in a variable, and then execute the necessary calculation:

#define crosstab
tab = pd.crosstab(index=df, columns='count')

#find proportions
tab/tab.sum()

col_0 count
Age
18 0.5
19 0.4
20 0.1

The resulting output now clearly represents the fractional proportion of the dataset contained within each age category. For instance, the calculated value of 0.5 for age 18 signifies that 50% of the students recorded in the **DataFrame** are 18 years old. This proportional view offers significantly more interpretive value than raw counts alone, making it indispensable when preparing data for advanced statistical hypothesis testing or complex predictive modeling, where reliable distributional assumptions are foundational.

Accurate interpretation of the relative frequency table requires recognizing that the values

represent probabilities or percentages within the context of the dataset. The normalized results clearly illustrate the demographic breakdown based on age, providing percentages rather than simple counts, which aids in standardized comparisons across different studies:

50% of students are 18 years old (0.5 proportion).
40% of students are 19 years old (0.4 proportion).
10% of students are 20 years old (0.1 proportion).

Creating Contingency Tables (Bivariate Frequency Analysis)

The true strength and versatility of the **crosstab()** function shine brightest in its ability to generate two-way **frequency table**s, which are formally known as **contingency tables**. These tables are fundamental to **bivariate analysis** because they simultaneously display the joint frequency distribution of two distinct categorical variables. This powerful capability allows analysts to meticulously examine potential relationships, dependencies, and associations that exist between the variables--moving beyond individual distributions to explore how variables interact.

To successfully create a two-way frequency table, the process is straightforward: we pass two different **Series** (columns) from the DataFrame to the **index** and **columns** arguments, respectively. Returning to our student dataset, we can use `crosstab()` to explore the relationship between 'Age' and 'Grade'. This allows us to investigate whether a student's age is associated with their academic performance, thereby constructing a matrix that details joint counts for every combination:

pd.crosstab(index=df, columns=df)

Grade A B C D
Age
18 3 1 1 0
19 0 2 0 2
20 0 1 0 0

This resultant contingency table delivers a substantially richer perspective on the data compared to analyzing two separate one-way tables. The rows in this structure represent the categories of the primary variable ('Age'), while the columns represent the categories of the secondary variable ('Grade'). Critically, the intersection of any row and column contains the count of observations that share both specified attributes. For example, by locating the cell where the 'Age' is 18 and the 'Grade' is 'A', we immediately determine the joint count is 3, meaning three students fall into this precise category.

Interpreting the findings from a two-way table demands careful scrutiny of these specific cell

counts, as they pinpoint distinct characteristics within the overall population. This methodology is often key to uncovering patterns and dependencies that remain concealed when only individual variables are considered:

There are **3** students who are 18 years old and received an 'A' in the class.
There is **1** student who is 18 years old and received a 'B' in the class.
There is **1** student who is 18 years old and received a 'C' in the class.
There are **0** students who are 18 years old and received a 'D' in the class.

Contingency tables serve as the essential foundational data for calculating advanced statistical measures of association, such as the **Chi-squared test**. Such tests help determine whether the observed relationship between the two variables is statistically significant or merely due to random chance. Furthermore, the official documentation for the **crosstab()** function details various advanced parameters, including options for calculating marginal row and column sums, and methods for normalizing the results based on rows, columns, or the total count. These advanced features enable analysts to perform sophisticated proportional analyses that extend far beyond basic frequency counting.