Calculating Relative Frequency with Python: A Step-by-Step Guide

Authored by

Mohammed looti

November 7, 2025

In the critical fields of statistics and data analysis, a foundational skill is mastering the distribution of observations within any given dataset. The metric that provides this vital context is relative frequency. This measure effectively quantifies the proportion of times a specific observation or event occurs compared to the total number of observations recorded. By transforming raw counts into normalized proportions, relative frequency offers a clear picture of the underlying probability distribution, moving beyond simple counts to provide meaningful comparisons.

Calculating relative frequencies manually is often impractical and prone to error, especially when processing large volumes of data. Fortunately, the Python programming language, a staple in modern statistical computation, provides robust and elegant solutions for data manipulation. The function detailed in this guide offers a clean, efficient, and reusable method for deriving these frequencies from any input list or iterable structure, thus streamlining the essential first steps of exploratory data analysis (EDA).

The Fundamental Concept of Relative Frequency

At its core, relative frequency is defined by a simple ratio: the frequency of a particular event divided by the total count of all trials or observations. If we consider a dataset containing **N** total items, and a specific value **V** appears **F** times, the relative frequency is calculated directly as the ratio **F/N**. It is essential to remember that the resulting output will always be a value between 0 and 1, representing the proportional share of that observation.

This process of normalization is what makes relative frequency so powerful; it allows for straightforward comparisons across datasets, regardless of their total size. Unlike absolute frequency (raw counts), relative frequency provides immediate insight into the empirical probability associated with each value. For instance, a calculated relative frequency of **0.25** immediately tells us that the corresponding value accounts for 25% of all observations in the sample.

A crucial property of all relative frequency distributions is that the sum of the calculated relative frequencies for every unique value in a dataset must precisely equal 1 (or 100%). This property serves as a vital internal check to confirm the accuracy of the calculation and ensures that the entirety of the data distribution has been comprehensively accounted for.

Developing the Relative Frequency Function in Python

To calculate relative frequencies effectively using Python, we can define a flexible function designed to accept an input list or array of data. This function harnesses efficient, built-in Python features to quickly identify unique data points and execute the necessary normalization calculations.

Our implementation relies on the efficient structure of list comprehension, which enables the

generation of a new list (in our case, a list of tuples) by iterating concisely over an existing sequence. We begin by converting the input data list, `x`, into a set using `set(x)`. This preliminary step is key, as it guarantees that the subsequent iteration and calculation are performed only once for each unique value present in the data, drastically improving performance.

For every unique value identified, the function first determines its absolute count using the `x.count(value)` method. This count is then divided by the total number of items in the original list, obtained via `len(x)`, thereby normalizing the count. The result is structured as a tuple containing the unique value and its corresponding calculated relative frequency, which the function returns as a complete list of distributions.

def rel_freq(x):
freqs =
return freqs

The subsequent practical examples demonstrate the versatility and power of this custom Python function when applied to different types of data structures commonly encountered in statistical analysis.

Practical Application 1: Analyzing Discrete Numerical Data

Our initial application involves a straightforward numerical dataset. This scenario is highly representative of analyzing discrete variables, such as scores, rankings, or simple counts. We define a standard list of integers and then apply our custom `rel_freq` function to quickly visualize the statistical distribution of these numbers.

The output generated is a list of tuples. The first element of each tuple identifies the unique data point, while the second element represents its calculated relative frequency. It is worth noting that standard floating-point arithmetic in Python often produces decimals with high precision, exceeding typical manual rounding expectations.

 # Define the numerical data list
data =

# Calculate relative frequencies for each value
rel_freq(data)

Interpreting this output is highly intuitive, allowing us to immediately translate the decimal proportions into meaningful insights regarding the data's composition:

The value "1" has a relative frequency of **0.42857**, signifying that it accounts for approximately 42.86% of all observations.
The value "2" and the value "3" each share a relative frequency of **0.142857**, representing about 14.29% of the total observations individually.
The value "4" has a relative frequency of **0.28571**, thus accounting for roughly 28.57% of the data points.

Importantly, summing these proportions (0.42857... + 0.142857... + 0.142857... + 0.28571...) yields exactly 1.0. This outcome confirms that the function correctly normalized the frequencies across the entire sample space, providing a valid statistical distribution.

Practical Application 2: Analyzing Categorical Data

Relative frequency calculations are not limited solely to numerical data; they are exceptionally useful when analyzing qualitative or categorical data, such as types, labels, or textual categories. This technique allows for rapid assessment of market share, popularity, or the proportional distribution of distinct categories within a given sample.

In this example, we define a list composed of character labels. Our function treats each unique character string as a distinct category, calculating its occurrence relative to the total number of characters in the list. This is a common requirement when processing survey responses or text-based data.

 # Define the categorical data list
data =

# Calculate relative frequencies for each category
rel_freq(data)

The clear and concise output immediately displays the proportional representation of each character category:

The category "a" has a relative frequency of **0.4**, meaning it constitutes 40% of the observations.
The category "b" also shows a relative frequency of **0.4**, accounting for another 40%.
The category "c" has a relative frequency of **0.2**, accounting for the remaining 20%.

As expected, the sum of these relative frequencies (0.4 + 0.4 + 0.2) equals 1.0, validating the proportional distribution across all defined categories within the sample space.

Practical Application 3: Integrating with pandas DataFrames

In practical data science environments, data is rarely confined to simple lists. Instead, it is typically managed in organized, tabular structures, most often utilizing the pandas DataFrame. While pandas provides its own highly optimized method for frequency calculation--specifically the `.value_counts(normalize=True)` Series method--our generic Python function can still be applied effectively by first extracting the targeted column data into a standard Python iterable.

This example illustrates the necessary steps to initialize a small DataFrame and then calculate the relative frequencies for the values within a single column, designated 'A'. It is crucial to explicitly cast the pandas Series object `data` into a standard list format before passing it as the argument `x` to our `rel_freq` function. This ensures compatibility with the function's expected input type.

import pandas as pd

# Define the DataFrame structure
data = pd.DataFrame({'A': ,
'B': ,
'C': })

# Calculate relative frequencies of values in column 'A'
rel_freq(list(data))

Analyzing the output specific to column 'A' allows us to quickly derive its distribution:

The value "25" has a relative frequency of **0.2** (20%).
The value "19" has a relative frequency of **0.2** (20%).
The value "14" has a relative frequency of **0.2** (20%).
The value "15" has a relative frequency of **0.4** (40%).

This final example underscores the flexibility of our custom Python function, confirming its ability to interface successfully with widely used data libraries like pandas, provided the data is appropriately prepared into a standard list format for processing.

Additional Resources

To further enhance your understanding and exploration of relative frequency concepts and specialized applications, refer to the following resources:

Relative Frequency Calculator
Relative Frequency Histogram: Definition + Example

How to Calculate Relative Frequency in Excel