How to Group a Dataframe by Value Ranges Using Pandas

Authored by

stats writer

November 20, 2025

Grouping a DataFrame by a range of values is a fundamental operation in data analysis, often referred to as binning or discretization. This technique is essential for converting continuous numerical data into discrete categories, which simplifies visualization and facilitates more manageable statistical analysis. The primary tool for achieving this in Pandas is the **cut() function**. By leveraging cut(), analysts can define custom intervals or bins from a column of numeric data. Once these bins are established, they create a categorical structure that can then be seamlessly used with the powerful groupby() method. This combined approach allows for robust aggregation across specified value ranges, producing powerful insights that might be obscured in the raw, continuous data.

The process of grouping data by defined ranges requires a two-step approach within the Pandas ecosystem. First, we must categorize the numerical data into structured intervals (bins). Second, we apply the **groupby()** operation using these newly created categories as the grouping key. This synergy enables us to calculate descriptive statistics--such as sums, means, or counts--for data points that fall within specific numerical boundaries. Understanding the syntax for integrating the **cut() function** directly into the **groupby()** method is crucial for concise and efficient data manipulation, allowing analysts to transition smoothly from data preparation to meaningful statistical analysis without creating intermediate columns.

Defining Data Ranges Using pd.cut()

The Pandas **cut() function** is specifically designed for segmenting and sorting data values into discrete intervals. It requires the data series to be binned and a list of explicit bin edges. These edges define the boundaries of the resulting categories. For instance, if analyzing income data, one might define bins like to segment individuals into distinct income brackets. The power of **cut()** lies in its ability to transform a continuous variable into an ordinal or nominal variable, preparing it perfectly for subsequent group-wise analysis. This transformation is key when managing data where the magnitude itself matters less than the category to which that magnitude belongs, such as grading systems or risk tiers.

When using **cut()**, the output is a categorical series that retains the structure of the input DataFrame indices but replaces the original numerical values with interval labels. These labels represent the bins into which the original data points fall. By default, Pandas creates right-inclusive bins (e.g., (A, B]), meaning the bin includes values strictly greater than A and less than or equal to B. This default behavior can be customized using the `right=False` parameter if left-inclusive bins are required. However, for most standard data analysis tasks, the right-inclusive interval ensures clear demarcation between adjacent bins, preventing ambiguous assignment of boundary values.

Core Syntax for Range-Based Aggregation

To perform range-based aggregation, we embed the output of the **pd.cut()** function directly into the groupby() method. This integrated syntax is highly efficient and streamlined. The operation creates temporary categories defined by the bin edges and immediately uses these categories to partition the data for calculation. The standard practice for grouping a column by defined ranges before performing an aggregation is demonstrated in the following code block:

df.groupby(pd.cut(df,)).sum()

In this powerful one-liner, the code instructs Pandas to first discretize the values in **my_column** based on the provided list of cut points: 0, 25, 50, 75, and 100. The resulting categorical series then acts as the primary grouping key for the **groupby()** method. Finally, the **sum()** function performs the actual data aggregation, calculating the total sum for every column within the DataFrame that contains numerical data, effectively summarizing the data according to the established value ranges.

Interpreting Interval Notation and Bins

The list of boundary values provided to **pd.cut()** defines a sequence of contiguous, non-overlapping intervals. For the specific boundary sequence , Pandas generates four distinct ranges. It is crucial to correctly interpret the interval notation, where the parenthesis `(` denotes an exclusive boundary (value not included) and the square bracket `]` denotes an inclusive boundary (value included), following the default `right=True` setting:

(0, 25]: Represents values strictly greater than 0 up to and including 25.
(25, 50]: Represents values strictly greater than 25 up to and including 50.
(50, 75]: Represents values strictly greater than 50 up to and including 75.
(75, 100]: Represents values strictly greater than 75 up to and including 100.

These intervals become the new index labels for the resulting grouped DataFrame after aggregation. It is essential to understand this interval notation to accurately interpret the output, especially when boundary values are present in the original data. If a data point falls exactly on a boundary (e.g., 50), it will be assigned to the interval where that boundary is inclusive, which, in this default setup, is the higher bin, (25, 50]. This careful assignment ensures that every data point within the specified range is assigned to one and only one group, ensuring complete and accurate grouping.

Practical Demonstration: Setting Up Retail Data

To solidify the theoretical concepts, let us apply this methodology to a concrete, real-world scenario. Suppose we are working with a DataFrame containing information about retail locations, specifically detailing the **store_size** (a continuous numerical variable) and corresponding **sales** figures. Our business objective is to determine the aggregate sales performance based on different size categories (Small, Medium, Large) defined by specific size ranges. This requires grouping the data by predefined ranges of the **store_size** column, providing immediate insight into performance segmentation.

We begin by setting up the sample data using Pandas. This initial step ensures we have a clear, reproducible basis for demonstrating the range-based grouping syntax. The data includes ten stores with sizes ranging from 14 units to 98 units, making it an ideal candidate for binning into four equal size categories (quartiles) between 0 and 100.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'store_size': ,
'sales': })

#view DataFrame
print(df)

store_size sales
0 14 15
1 25 18
2 26 24
3 29 25
4 45 20
5 58 35
6 67 34
7 81 49
8 90 44
9 98 49

Example 1: Grouping and Aggregating All Columns

We apply the core range grouping syntax using the explicit boundaries . Following the bin creation, we calculate the sum of values across all numerical columns (**store_size** and **sales**) for each defined size category. This operation simultaneously groups the data and performs the aggregation, resulting in a summary table indexed by the size ranges.

The output table clearly shows the aggregate results, indexed by the categorical interval labels generated by **pd.cut()**. For instance, the bin (0, 25] includes stores whose sizes are 14 and 25. The aggregated results show that the sum of their sizes is **39**, and the sum of their sales is **33**. This immediate summarization provides a clear comparison of cumulative metrics across different segments of store size.

#group by ranges of store_size and calculate sum of all columns
df.groupby(pd.cut(df,)).sum()

store_size sales
store_size
(0, 25] 39 33
(25, 50] 100 69
(50, 75] 125 69
(75, 100] 269 142

Reviewing the aggregated metrics:

The smallest stores (0 to 25 units) generated total sales of **33**.
Mid-sized stores (25 to 50 units) generated total sales of **69**.
The largest stores (75 to 100 units) exhibit the highest collective sales volume at **142**, suggesting a strong correlation between store size and overall sales output within this dataset.

Example 2: Targeted Aggregation of a Single Column

While the previous example provided totals for all numerical columns, often the requirement is to focus the aggregation on a specific metric. If our sole interest is analyzing sales performance across the store size ranges, we can refine the syntax to perform the **sum()** operation exclusively on the **sales** column. This modification is implemented by chaining the column selection bracket notation after the **groupby()** method but before the final aggregation function.

This targeted approach yields a cleaner Series object indexed by the store size ranges, focusing solely on the total sales for each bin. This structure is often preferred when preparing data for

visualization or when the analysis requires isolation of a single dependent variable, enhancing clarity and reducing computational overhead compared to aggregating all available columns.

#group by ranges of store_size and calculate sum of sales
df.groupby(pd.cut(df,)).sum()

store_size
(0, 25] 33
(25, 50] 69
(50, 75] 69
(75, 100] 142
Name: sales, dtype: int64

Automating Bin Creation with NumPy arange()

Manually listing all cut points, such as , can be inefficient and susceptible to errors, particularly when dealing with variables spanning large ranges or requiring a high number of evenly spaced bins. A powerful solution involves leveraging the capabilities of the NumPy library, specifically the **NumPy arange() function**. This function allows us to automatically generate the required evenly spaced numerical values to define our bin edges, significantly simplifying the code while maintaining precision.

The arange() function operates by taking a starting value, a stopping value, and a step size. For our retail example, defining bins from 0 to 100 with a step of 25 is accomplished by calling `np.arange(0, 101, 25)`. It is essential to remember the array stops *before* the specified stop value; thus, we use 101 to ensure that 100 is included as the final boundary point. This automation is vital in pipelines where the data range might change based on external inputs, ensuring that bin creation scales dynamically.

By substituting the manual list of bin edges with the output of **np.arange()**, we achieve the same grouping result with cleaner, more programmatic syntax. This method minimizes hardcoding and improves the maintainability and generality of the analysis script. Notice that the integration is seamless, as the **pd.cut()** function accepts a NumPy array for its bin argument.

import numpy as np

#group by ranges of store_size and calculate sum of sales

```
df.groupby(pd.cut(df, np.arange(0, 101, 25))).sum()
```

```
store_size
(0, 25] 33
(25, 50] 69
(50, 75] 69
(75, 100] 142
Name: sales, dtype: int64
```

Summary of Best Practices

Grouping a DataFrame by a range of values is an indispensable technique for converting continuous data into meaningful, analyzable segments. The core methodology involves using **pd.cut()** to define bins and integrating its categorical output directly into the **groupby()** method. This approach bypasses the need for creating temporary columns, leading to highly efficient and readable code.

For analysts seeking maximum efficiency and automation, understanding how to utilize NumPy's array generation functions, like **arange()**, is critical for automating the creation of evenly spaced bin boundaries. Always be mindful of the default interval notation (right-inclusive) used by **pd.cut()** to ensure accurate interpretation of the resulting groups. By mastering these functions, data professionals can efficiently perform complex range-based aggregations, transforming raw numerical data into actionable segmented metrics and statistical summaries.

Note: You can find the complete documentation for the NumPy arange() function, along with other array creation routines, in the official NumPy documentation.