How to Bin Variables in Python Using numpy.digitize()

Authored by

stats writer

December 24, 2025

Binning variables, also known as discretization, is a fundamental process in data preprocessing, especially when working with statistical models that perform better with categorical inputs. In Python, the most robust and efficient way to achieve this is by utilizing the powerful capabilities of the NumPy library, specifically the numpy.digitize() function. This tool is designed to classify numerical data points efficiently.

The core mechanism of numpy.digitize() involves taking an input array of numerical values and mapping them onto predefined bins or intervals. While the bins are not required to be of equal size, the function excels at assigning each element an index that corresponds to the specific bin it falls into. This transformation is pivotal for converting challenging continuous data--which often poses challenges for modeling linearity assumptions--into simpler, manageable discrete categories for streamlined statistical analysis or machine learning applications.

The Importance of Discretization in Data Science

Data analysis frequently requires that raw, quantitative variables be transformed into structured, qualitative categories. This process, known as binning or discretization, allows analysts to handle outliers more gracefully, reduce the impact of minor measurement errors, and prepare data for algorithms that prefer or require categorical features. When utilizing Python for these tasks, the NumPy library provides the performance and functionality necessary to execute binning operations swiftly across large datasets.

The primary tool for this operation within NumPy is the `numpy.digitize()` function. Unlike other binning methods that might prioritize uniform interval width, `digitize()` is particularly effective for defining custom, non-uniform bin boundaries based on specific domain knowledge or statistical thresholds. Understanding how to set these boundaries is key to successful implementation.

Mastering the `numpy.digitize()` Syntax

The syntax for numpy.digitize() is clean and focuses on two mandatory arguments and one critical optional parameter. This function returns an array of indices, where each index specifies the bin to which the corresponding input element belongs.

numpy.digitize(x, bins, right=False)

Here is a detailed breakdown of the required arguments:

x: This is the input array or list containing the numerical data points that need to be categorized. These values are the subjects of the binning operation.
bins: This is an array or list defining the boundaries of the bins. Importantly, these boundaries must be sorted (either increasing or decreasing). For example, if you define `bins=`, you establish

three bins: values less than 10, values between 10 and 20, and values greater than 20.
right: This optional boolean parameter dictates how boundary conditions are handled. By default, `right=False`, meaning the intervals are generally closed on the left and open on the right (e.g.,). This detail is crucial for precise categorical mapping.

The subsequent examples will clearly illustrate how these parameters interact and how to properly structure your bin definitions to achieve specific categorization goals.

Example 1: Creating Simple Binary Bins (Default Behavior)

In many analytical scenarios, the goal is simply to divide a dataset into two distinct groups based on a single threshold, such as classifying customers as "low spend" or "high spend," or defining passing scores. This creates a binary categorization. Using numpy.digitize() for this purpose requires only one boundary value in the `bins` array.

For instance, if we set the boundary at 20, the function automatically creates two resulting index categories, corresponding to two distinct bins. Remember that by default, `right=False`, which means the lower boundary is inclusive for the subsequent bin.

The resulting bins are mapped as follows:

Index 0: Assigned if the value (x) is strictly less than 20 (x < 20).
Index 1: Assigned if the value (x) is greater than or equal to 20 (x ≥ 20).

We apply the function to our sample data using the boundary :

import numpy as np

Create the sample data set
data =

# Place values into two bins using 20 as the sole boundary
np.digitize(data, bins=)

array()

As expected, all values up to 19 are assigned the index 0, and the value 20 itself (and everything above it) is assigned the index 1. This demonstrates the strict adherence to the default boundary logic: bins are defined by boundaries, and the index returned corresponds to the bin number (0, 1, 2, ...).

Example 2: Establishing Multi-Level Categories (Default `right=False`)

When more granular categorization is needed, we define multiple boundaries in the `bins` array. If we specify N boundaries, numpy.digitize() will produce N+1 bins. For instance, defining two boundaries (10 and 20) yields three distinct bins, indexed 0, 1, and 2.

This example uses `bins=`, which establishes the following intervals under the default `right=False` setting (left side inclusive, right side exclusive):

Index 0: Values less than the first boundary (x < 10).
Index 1: Values between the first and second boundaries, including the lower boundary (10 ≤ x < 20).
Index 2: Values greater than or equal to the second boundary (x ≥ 20).

We utilize the same principle with a slightly modified dataset to observe the boundary assignments closely. Notice how the value 20 is handled; since it is the upper bound of Bin 1, and the default setting excludes the right edge, 20 must fall into the next bin (Index 2).

import numpy as np

Define the data array
data =

# Place values into three bins using boundaries 10 and 20
np.digitize(data, bins=)

array()

The output confirms that 12 and 14 fall into index 1 (since 10 ≤ x < 20), while 20 is correctly categorized into index 2, starting the highest bin. This behavior is standard for array binning where the returned index `i` satisfies `bins <= x < bins`.

Example 3: Handling Boundary Conditions with `right=True`

The parameter `right` is essential when precise control over boundary inclusion is required. By setting `right=`**`True`**, we fundamentally alter the interval logic. Instead of the left boundary being inclusive, the right boundary of each interval becomes inclusive, while the left boundary is exclusive. This setting is particularly useful in scenarios where the higher endpoint of a range should belong to that bin, such as grading systems where the maximum score for a grade level is critical.

When `bins=` and `right=True`, the resulting bin definitions change to include the upper limit:

Index 0: Values less than or equal to the first boundary ($x \leq 10$).
Index 1: Values greater than the first boundary and less than or equal to the second ($10 < x \leq 20$).
Index 2: Values strictly greater than the highest boundary ($x > 20$).

Observing the result on our test array, note the critical change in how the value 20 is handled. Unlike the previous example (where 20 was index 2), here, 20 falls squarely within the second bin (Index 1) because the interval is defined as (10, 20].

import numpy as np

Define the data array
data =

# Place values into bins, ensuring the right edge is inclusive
np.digitize(data, bins=, right=True)

array()

The shift in the index assignment for 20 (from 2 down to 1) perfectly illustrates the impact of the `right` parameter. It is vital to determine the required boundary inclusion based on the nature of the continuous data being analyzed before executing the numpy.digitize() function.

Example 4: Implementing Fine-Grained Segmentation

Often, researchers require a more complex categorization scheme, dividing the continuous data into quartiles, deciles, or custom segmentation defined by three or more boundaries. If we define three boundaries, `numpy.digitize()` generates four resulting bins.

In this scenario, we aim to segment the data into four groups using the boundaries 10, 20, and 30. Keeping the default `right=False` ensures the bin definitions follow the standard left-inclusive logic:

Index 0: $x < 10$
Index 1: $10 \leq x < 20$
Index 2: $20 \leq x < 30$
Index 3: $x \geq 30$ (The final bin captures all values greater than or equal to the highest boundary.)

This fine-grained segmentation allows for better differentiation between data points that might otherwise be lumped together in broader categories. We utilize the function below to observe how the boundary 30 effectively splits the highest category.

import numpy as np

Define the data array
data =

# Place values into four bins using three boundaries: 10, 20, and 30
np.digitize(data, bins=)

array()

The results show that values 20, 22, and 24 fall into index 2, while values 31 and 34, which are greater than the final boundary of 30, are correctly assigned to index 3. This successful segmentation highlights the utility of `numpy.digitize()` in creating specialized categorical outcomes from numerical inputs.

Advanced Application: Counting Bin Frequencies with `numpy.bincount()`

Once variables have been successfully discretized using numpy.digitize(), a common subsequent analytical step is determining the distribution of data across these newly created discrete categories. NumPy offers a highly optimized function specifically for this task: numpy.bincount(). This function is designed to count the frequency of non-negative integers (which are exactly what `digitize()` returns).

The numpy.bincount() function takes the output index array from `digitize()` and returns a new array where the value at index `i` is the count of how many times the integer `i` appeared in the input index array. This provides an immediate and efficient summary of the categorical distribution.

We will reuse the data from Example 2 (three bins defined by boundaries 10 and 20) and apply `bincount()` to the resulting index array:

import numpy as np

Define the data array
data =

# Step 1: Place values into bins (Index array generated)
bin_data = np.digitize(data, bins=)

View the intermediate binned data (indices)
bin_data

array()

Step 2: Count the frequency of each bin index using bincount

np.bincount(bin_data)

array()

The resulting numpy.bincount() output, `array()`, is interpreted sequentially based on the index:

The first element (4) corresponds to Index 0 (x < 10), containing **4** data values.
The second element (2) corresponds to Index 1 (10 ≤ x < 20), containing **2** data values.
The third element (5) corresponds to Index 2 (x ≥ 20), containing **5** data values.

This combination of `numpy.digitize()` and `numpy.bincount()` provides a complete workflow for converting continuous data into structured, countable discrete categories for immediate frequency analysis.

Conclusion: Optimizing Data Analysis Through Binning

The process of binning is indispensable in preprocessing pipelines, enabling analysts and data scientists to move seamlessly between continuous measurements and meaningful categorical definitions. The numpy.digitize() function provides a powerful, high-performance solution within Python for achieving this transformation. Its clarity, combined with the precise control offered by the `bins` array and the `right` parameter, makes it the preferred method for numerical discretization.

By mastering the application of custom bin boundaries and understanding the subtle but important effects of boundary inclusion (`right=True` vs. `right=False`), users can ensure that their data classification is accurate and aligned with the requirements of downstream statistical modeling or machine learning algorithms. Furthermore, integrating the output with functions like numpy.bincount() allows for immediate frequency assessment, completing the typical workflow of data segmentation and summarization.

This tutorial demonstrates that effective variable binning in Python is not only efficient but also highly customizable, providing the flexibility needed to handle diverse analytical challenges. Continue exploring NumPy documentation to uncover additional features that enhance data manipulation capabilities.