How to Easily Customize Histogram Bin Size in Matplotlib

Authored by

stats writer

December 4, 2025

In Matplotlib, adjusting the aggregation level of your data within a histogram is essential for accurate visualization. This aggregation level is governed by the bin size, which determines how data points are grouped along the x-axis. Manipulating the bin size is achieved primarily by modifying the `bins` argument within the `plt.hist()` function. This critical parameter accepts either a single integer, specifying the total number of bins to create across the data range, or a sequence of values, defining the exact edges where the bins begin and end.

By default, Matplotlib assigns 10 bins to a histogram. However, relying solely on this default may obscure critical patterns or introduce excessive noise, depending on the volume and distribution of your dataset. Therefore, data scientists must actively manage the bin configuration. Increasing the number of bins results in narrower intervals, offering higher granularity but potentially highlighting noise. Conversely, decreasing the bin count creates wider intervals, smoothing the distribution but risking the loss of fine details regarding the data's true shape.

Beyond merely setting the bin count or edges, the `plt.hist()` function provides flexibility through the optional `range` argument. This argument is particularly useful when you need to focus the visualization on a specific segment of the data, excluding outliers or irrelevant extreme values. By specifying the minimum and maximum data values () to be included in the plot, you ensure that the bins are calculated only within that designated interval, offering a cleaner and more targeted statistical overview.

The Significance of Bin Size in Data Visualization

A histogram serves as a fundamental tool in exploratory data analysis (EDA), providing a visual estimate of the probability distribution of continuous data. The structure of this visualization depends entirely on how the data is partitioned into discrete intervals, known as bins. The selection of an appropriate bin size is not trivial; it directly impacts how the data's underlying distribution is perceived by the viewer. Too few bins can lead to excessive smoothing, masking multimodality or skewness, while too many bins can result in a choppy visualization dominated by sampling variability and noise, making it difficult to discern the true central tendency or spread.

Effective data visualization requires balancing these trade-offs. If the bins are too wide, distinct features of the distribution, such as multiple peaks (modes), might be compressed into a single bar, leading to a loss of information and potentially misleading conclusions about the data generation process. Conversely, if the bins are extremely narrow, each bar may represent very few data points, creating a jagged pattern that suggests variance where none might statistically exist, thereby hindering clear communication of the results. This makes the adjustment of the `bins` parameter in Matplotlib a crucial step in preparing publication-quality statistical graphics.

Experienced analysts approach bin selection methodically, often trying several configurations to assess how robust the observed data features are. The goal is to choose a binning strategy that

best reveals the underlying structure of the dataset--whether it is Gaussian, skewed, or multimodal--without being unduly influenced by random fluctuations. Matplotlib's flexibility in accepting three distinct methods for bin definition allows users to move beyond heuristic guessing and employ calculated approaches based on statistical theory or practical necessity. We will explore these three primary mechanisms for controlling histogram construction in detail.

Mastering the `plt.hist()` Function Parameters

The `plt.hist()` function is the primary interface for generating histograms in Matplotlib. Understanding its core arguments is paramount for customization. The function requires the data array as its first positional argument. The subsequent and most impactful argument is `bins`. When `bins` is supplied as a single integer (e.g., `bins=20`), Matplotlib automatically calculates 20 equally spaced intervals between the minimum and maximum values of the input data, distributing the observations accordingly. This is the simplest and most common method for quick visualization.

However, when greater control is needed, the `bins` argument can accept a sequence, typically a list or a NumPy array, representing the explicit boundaries of the bins (e.g., `bins=`). In this case, the number of resulting bins will be one less than the number of boundaries provided. This method is indispensable when the analyst requires non-uniform bin widths--perhaps focusing finer resolution on a dense area of the distribution while using wider bins for sparse tails. This level of manual specification ensures that the grouping aligns perfectly with domain knowledge or external classification standards.

Furthermore, while not directly controlling the bin count or width, the optional `range` parameter plays a supporting role. If the input data contains extreme outliers that unnecessarily stretch the histogram's axis, setting `range=(min_val, max_val)` allows the user to clip the data being considered for bin calculation. For instance, if data spans from 1 to 1000 but 99% of observations lie between 10 and 50, setting `range=(10, 50)` ensures that the bins are tightly focused on the relevant core distribution, resulting in a much more informative plot, although care must be taken to acknowledge that data outside this range is excluded from the count.

The following methods offer distinct approaches to precisely adjusting the bin size of histograms in Matplotlib:

Method 1: Specify Number of Bins

plt.hist(data, bins=6)

Method 2: Specify Bin Boundaries

plt.hist(data, bins=)

Method 3: Specify Bin Width

w=2
plt.hist(data, bins=np.arange(min(data), max(data) + w, w))

The examples below demonstrate the implementation and visual effect of each of these three crucial methods for bin manipulation in practice, allowing for fine-tuned control over your statistical visualizations.

Method 1: Controlling Granularity by Specifying the Number of Bins

The most straightforward way to control histogram resolution is by passing an integer to the `bins` argument of the `plt.hist()` function. This integer dictates the total number of equally sized bins that will span the range defined by the minimum and maximum values of the input data. For instance, setting `bins=10` instructs Matplotlib to divide the total data range into ten segments, with the width of each segment being calculated automatically as `(max(data) - min(data)) / 10`.

This method is ideal for initial exploratory analysis or when comparing the distribution of two datasets with similar scales, as it guarantees proportional binning relative to the data range. However, it is important to remember that the resulting bin width is dependent on the data's span. If the minimum and maximum values are far apart, even a large number of bins might still result in relatively wide bins, potentially obscuring features in a densely populated region.

When employing this method, selecting the integer value requires careful consideration. A common rule of thumb suggests that the optimal number of bins, k, should be roughly proportional to the square root of the number of observations, N ($k approx sqrt{N}$), although more rigorous statistical methods like the Freedman-Diaconis rule or Sturges' formula exist. For smaller datasets, choosing a low integer (e.g., 5 to 10) often provides a sufficient overview, whereas larger datasets necessitate higher counts to prevent oversmoothing.

The following example illustrates how specifying `bins=6` results in six uniform intervals, providing a balanced view of the small sample dataset.

Practical Application: Specifying the Number of Bins (Example 1)

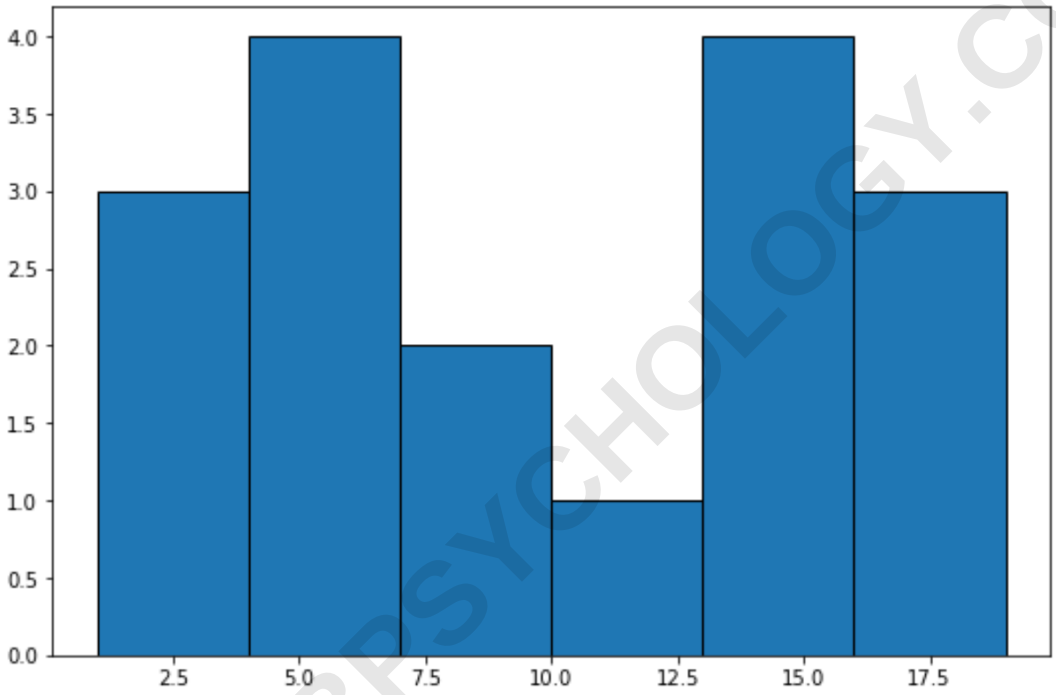
The code snippet below demonstrates how to define a small dataset and then generate a histogram using a fixed number of bins. Notice the inclusion of `edgecolor='black'`, which visually distinguishes the boundaries between the bins, a highly recommended practice for clarity.

import matplotlib.pyplot as plt

```
#define data
data =

#create histogram with specific number of bins
plt.hist(data, edgecolor='black', bins=6)
```

In this example, the data ranges from 1 to 19. By setting `bins=6`, Matplotlib automatically creates six equal intervals across this range. The visual output clearly shows how the 17 data points are distributed across these six segments.



It is crucial to note the direct relationship between the specified number of bins and the resulting bin width. Generally, the more bins you specify, the narrower each individual interval will be. While narrower bins offer finer detail, they can also lead to fewer counts per bar, potentially making the resulting distribution look less stable or representative of the true population density, especially with limited sample sizes.

Method 2: Defining Precise Data Groupings via Bin Boundaries

When statistical requirements demand precise, non-uniform, or domain-specific bin groupings, analysts must forgo the automatic calculation provided by an integer input and instead specify the exact boundaries (or edges) of each bin. This is accomplished by passing a list or NumPy array of values to the `bins` argument. For instance, if you are analyzing income data and want bins

corresponding to defined tax brackets (e.g., $0-50k, $50k-100k, etc.), specifying boundaries is the only reliable approach.

The sequence provided as the `bins` argument must contain $N+1$ values to define N bins. Each element in the sequence acts as a boundary point. Matplotlib constructs the bins such that they are left-inclusive and right-exclusive, except for the last bin, which includes both boundaries to capture all data points. This explicit control allows for highly tailored visualizations that align perfectly with external classification schemes or theoretical thresholds.

This approach is particularly powerful when dealing with datasets that exhibit highly skewed distributions or contain complex structures that standard, uniform binning fails to capture effectively. For example, if a large majority of data points are clustered near zero, while a few significant outliers exist, setting manually defined, narrow bins near zero and wider bins further out ensures optimal visual resolution across the critical range without unnecessarily stretching the plot.

In the subsequent example, we define the bin boundaries explicitly as . This creates five custom bins: . This method bypasses the automated calculation based on min/max and forces the data to be grouped exactly according to these specified intervals, ensuring reproducible and comparable bin structures regardless of minor fluctuations in the input data's overall range.

Practical Application: Defining Custom Bin Boundaries (Example 2)

The following code snippet demonstrates the implementation of Method 2, where a list of boundary values is passed to the `bins` argument. This results in five bins of uniform width (4 units each) in this specific case, starting precisely at 0 and ending at 20, regardless of whether the data points actually cover that full range.

import matplotlib.pyplot as plt

#define data
data =

#create histogram with specific bin boundaries
plt.hist(data, edgecolor='black', bins=)

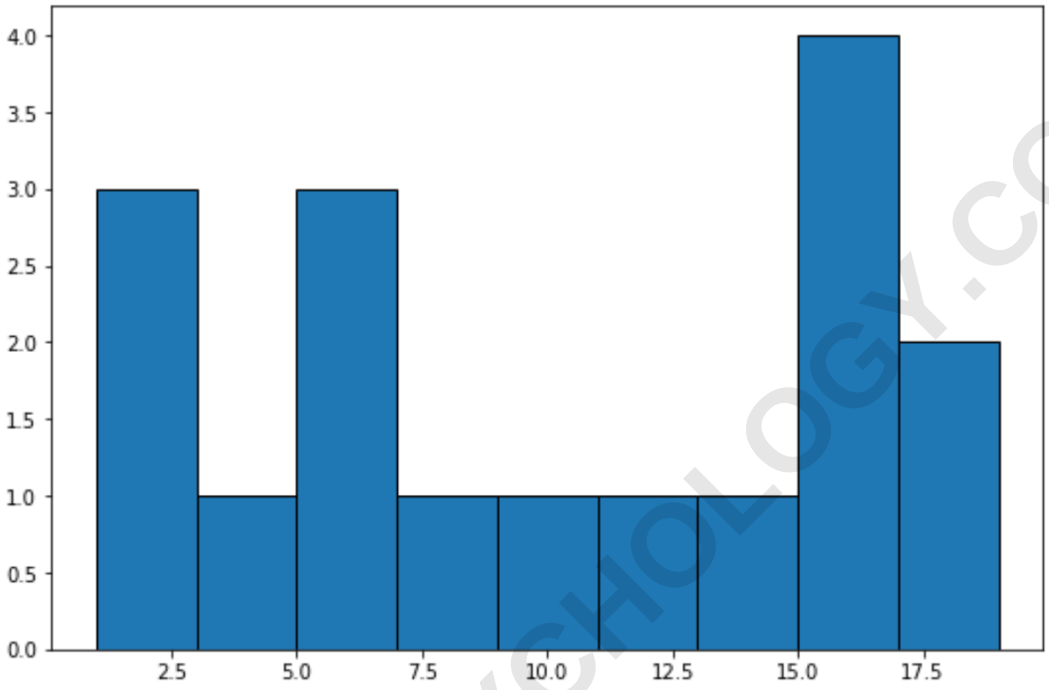
By specifying the exact bin boundaries, we override Matplotlib's default calculation. In the resulting visualization, data points like 1, 2, and 2 fall into the first bin

#specify bin width to use
w=2

#create histogram with specified bin width

plt.hist(data, edgecolor='black', bins=np.arange(min(data), max(data) + w, w))

By setting $W=2$, we generate a much larger number of bins compared to Example 1, where we used only 6 bins. The visual output now exhibits greater detail, revealing finer fluctuations in the data distribution.



When utilizing a fixed bin width, remember the inverse relationship: the smaller the bin width (W) you specify, the more narrow the bins will become, and consequently, the total number of bins will increase. While a very small width offers high detail, it can lead to sparsity (many bars with zero or one count), making the underlying distribution difficult to interpret. Conversely, a large width will over-smooth the data.

Choosing the Optimal Bin Strategy for Your Dataset

Determining the optimal bin size is perhaps the most subjective yet critical step in histogram creation. There is no universally 'correct' bin size; the best choice depends heavily on the sample size, the intrinsic variability of the data, and the specific goal of the analysis. For rapid exploration, specifying an integer number of bins (Method 1) is efficient. However, for rigorous reporting or specific analytical needs, one of the two boundary-setting methods (Methods 2 or 3) is usually preferred.

Statistical rules of thumb, such as the Freedman-Diaconis rule (which is robust against outliers) or Scott's normal reference rule, aim to minimize the integrated mean squared error of the density

estimate. While these rules often provide excellent starting points, they require external calculation (frequently facilitated by specialized libraries like NumPy or Matplotlib's automatic calculation modes, such as `bins='auto'`), and the resulting number or width must then be passed to the `bins` argument, often via a NumPy `arange` sequence.

Ultimately, the analyst should experiment with several bin configurations. If a key feature, such as bimodality, appears consistently across different, reasonable bin sizes, then that feature is likely genuine. If the visual interpretation changes drastically with minor bin size adjustments, it suggests that the dataset may be too small or too noisy to support strong conclusions based on the histogram alone, necessitating alternative visualization techniques or further statistical testing.

Conclusion: Enhancing Statistical Insights Through Bin Manipulation

The ability to precisely control the bin configuration of a histogram in Matplotlib is a fundamental skill for any data scientist. Whether you opt for the simplicity of specifying the total number of bins, the surgical precision of defining exact bin boundaries, or the methodological consistency of utilizing a fixed bin width calculated via NumPy's `arange`, these three methods ensure that your visualization accurately reflects the underlying distribution of your data.

By consciously manipulating the `bins` argument in the `plt.hist()` function, you move beyond mere plotting and engage in genuine data exploration, making informed decisions that maximize the clarity and statistical validity of your graphical output. Mastering these techniques guarantees that your histograms provide reliable insights into data structure, central tendency, and variance.

The following tutorials explain how to perform other common functions in Matplotlib: