How to Easily Create Distribution Plots with Matplotlib

Authored by

stats writer

November 21, 2025

Creating a distribution plot in Matplotlib is a fundamental skill for any data analyst using Python. These visualizations are essential for understanding the underlying distribution of a dataset, helping to identify central tendency, variance, skewness, and the presence of outliers. While Matplotlib provides the robust `hist` function for generating traditional histograms, integrating libraries like Seaborn offers powerful extensions, such as automatic density curve overlays, simplifying the visualization workflow significantly.

This comprehensive guide details the steps required to generate high-quality distribution plots using both core Python libraries. We will cover the necessary imports, parameter customization (including bins, colors, and edges), and the critical differences between a standard histogram and a plot enhanced with a Kernel Density Estimation (KDE) curve. Mastering these techniques ensures that you can effectively communicate the shape and characteristics of your data to stakeholders or colleagues.

Understanding Distribution Visualization in Python

The goal of a distribution plot is to provide a graphical summary of the frequency or probability of different values within a dataset. In Python, this is typically achieved using a histogram, which groups data into specified ranges (bins) and displays the count of observations falling into each range. Understanding how to control these visual elements is paramount for accurate data representation.

When visualizing distributions, we primarily rely on two immensely popular libraries: Matplotlib, which serves as the foundational plotting tool, and Seaborn, which is built upon Matplotlib and specializes in statistical data visualization. While Matplotlib requires explicit setup for features like density curves, Seaborn streamlines this process, allowing for rapid generation of sophisticated statistical graphics.

We will explore two distinct, yet complementary, methodologies for generating these plots. The first method leverages the direct control offered by Matplotlib's core functions, suitable for detailed customization. The second method utilizes Seaborn's higher-level interface, which is ideal for quickly adding statistical overlays like density estimates.

Method 1: Generating Histograms Using Matplotlib

The most direct way to create a frequency distribution plot in Python is by utilizing the built-in `hist()` function provided by the `matplotlib.pyplot` module. This approach gives the user granular control over every aspect of the histogram's appearance, including bar colors, edge definitions, and the critical setting of the number of bins. To begin, ensure the library is imported, typically aliased as `plt`.

```
import matplotlib.pyplot as plt

plt.hist(data, color='lightgreen', ec='black', bins=15)
```

This basic syntax immediately generates a histogram based on the input data array. The power of this function lies in its parameter flexibility, allowing the analyst to define the visual characteristics necessary for optimal data storytelling. Understanding the function parameters is key to creating meaningful and interpretable plots.

Key Parameters for Matplotlib's `hist()` Function

The appearance and effectiveness of a histogram are heavily dependent on how its core parameters are defined. The three most commonly used parameters in the Matplotlib `hist()` function provide essential control over the visualization:

`color`: This argument dictates the fill color of the bars within the histogram. By setting `color='lightgreen'`, we are ensuring the interior of the bars is visually distinct, making the plot easier to read against a typical white background. Color choice is crucial for accessibility and brand consistency in professional reports.

`ec` (Edge Color): This parameter controls the color used for the outlines of the bars. Setting `ec='black'` enhances the definition of each bin, especially when the bars are closely packed or when a large number of bins is used. Clear edge delineation prevents bars from visually blending together, improving the perception of frequency changes.

`bins`: Perhaps the most critical parameter, `bins` specifies the number of intervals or ranges into which the data will be divided. A smaller number of bins results in a coarse view of the distribution, potentially masking important details. Conversely, too many bins can make the plot appear noisy and highly variable. Choosing the appropriate number of bins--often based on rules like Freedman-Diaconis or Scott's rule, though 15 is a common starting point--is vital for accurately representing the underlying data structure.

It is important to remember that the interpretation of the resulting plot, including its shape and characteristics, can change significantly depending on the value chosen for `bins`. A larger value for the `bins` argument results in more bars, providing finer detail but also increasing the potential for visual noise.

Method 2: Enhanced Distribution Plots with Seaborn's `displot()`

While Matplotlib excels at foundational plotting, the Seaborn library offers high-level interfaces designed specifically for statistical visualization. Seaborn's `displot()` function is a versatile tool

that can generate histograms, density plots, and empirical cumulative distribution function (ECDF) plots easily. For distribution visualization, `displot()` often provides a more insightful view by automatically incorporating advanced statistical elements.

```
import seaborn as sns
sns.displot(data, kde=True, bins=15)
```

The primary advantage of using Seaborn is its integration of statistical estimation methods directly onto the histogram framework. This allows users to visualize not just the discrete frequency counts (the histogram bars), but also a smooth estimate of the probability density function using a Kernel Density Estimation (KDE) curve.

The Role of Kernel Density Estimation (KDE) in Seaborn

The key differentiator in the Seaborn approach is the `kde=True` parameter. This flag instructs the `displot()` function to calculate and overlay a density curve on the frequency histogram. The KDE curve acts as a non-parametric way to estimate the probability density function of a random variable, effectively smoothing out the inherent blockiness of the histogram bars.

The benefit of using a Kernel Density Estimation (KDE) curve is substantial: it summarizes the shape of the underlying distribution using a single continuous curve, making it easier to identify the mode(s), assess symmetry, and compare the distribution against theoretical models. While the histogram shows empirical counts, the KDE curve provides a theoretical approximation of where the population values are likely to fall.

The `bins` parameter functions identically to its role in Matplotlib, controlling the resolution of the histogram bars. However, even if the histogram is highly binned, the KDE curve remains continuous and smooth, offering a stable visual representation of the overall distribution shape. Analysts should refer to the official Seaborn documentation for the `displot()` function.

Setting Up the Sample Data (NumPy)

To provide practical illustrations of both visualization methods, we must first generate a representative dataset. We will use the NumPy library, which is the cornerstone of numerical computing in Python, to create an array of 1,000 values drawn from a standard normal distribution. This synthetic data will serve as the input for all subsequent plotting examples.

```
import numpy as np

#make this example reproducible.
np.random.seed(1)
```

#create numpy array with 1000 values that follow normal dist with mean=10 and sd=2
data = np.random.normal(size=1000, loc=10, scale=2)

#view first five values
data

array()

By setting a random seed (`np.random.seed(1)`), we ensure that the results are reproducible across different environments. The `np.random.normal` function generates 1,000 observations centered around a mean (`loc=10`) with a standard deviation (`scale=2`). This generates a classic bell-shaped distribution, which is highly useful for demonstrating visualization techniques.

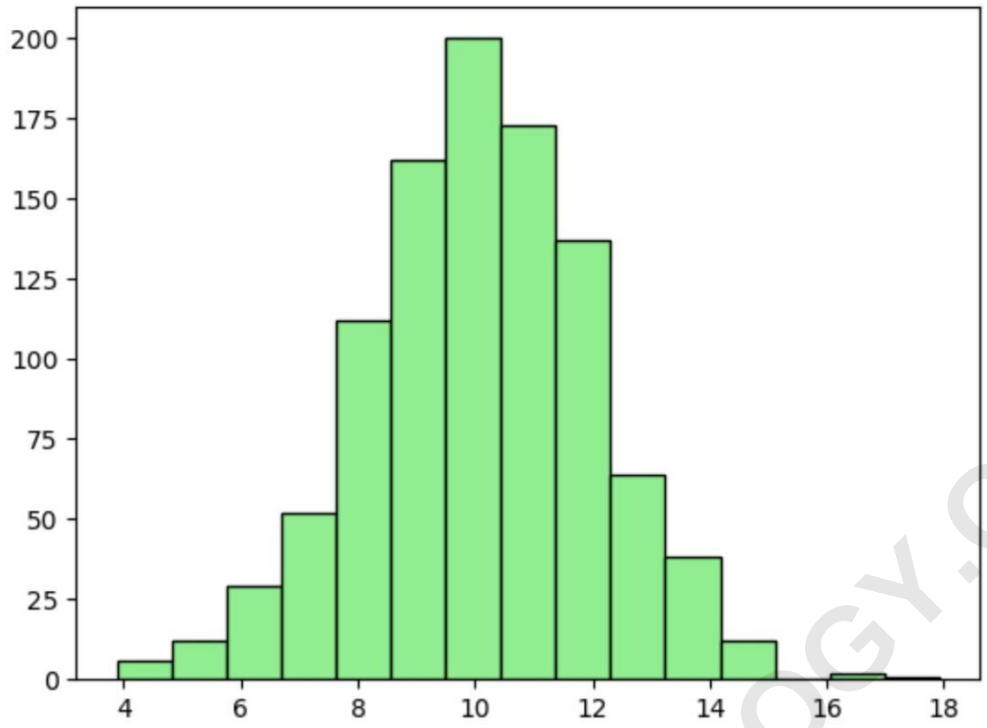
Detailed Matplotlib Example and Interpretation

Using the data array created by NumPy, we can now execute the Matplotlib code to generate a foundational histogram. This example specifically focuses on defining the visual boundaries of the bars for maximum clarity.

import matplotlib.pyplot as plt

#create histogram
plt.hist(data, color='lightgreen', ec='black', bins=15)

The resulting plot, visualized below, clearly illustrates the frequency distribution.



In this visualization, the horizontal axis (x-axis) represents the range of values present in the NumPy array, extending approximately from 4 to 16. The vertical axis (y-axis) displays the frequency, or count, of observations that fall into each corresponding bin interval. We can observe that the highest frequency occurs near the central value of 10, confirming the parameters used during the data generation step.

It is instructive to consider the impact of the **`bins`** parameter here. If we had chosen 5 bins, the plot would appear blockier, aggregating a wider range of values. Conversely, choosing 50 bins would result in many narrow bars, potentially highlighting small, non-significant fluctuations. The choice of 15 bins provides a good balance between detail and smoothness for a normal distribution of this size.

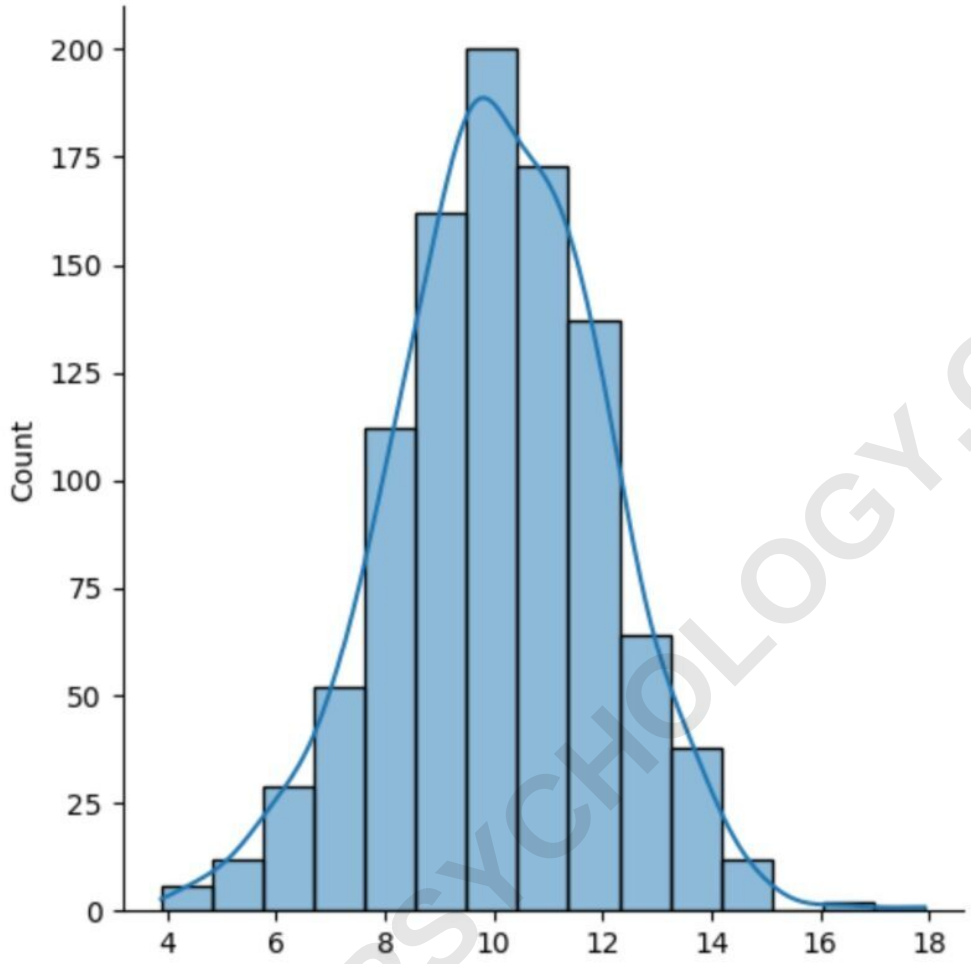
Detailed Seaborn Example and Interpretation

To demonstrate the added statistical depth provided by Seaborn, we utilize the `displot()` function. This function automatically handles many visualization details and, crucially, allows for the effortless overlay of the density curve.

import seaborn as sns

#create histogram with density curve overlaid
sns.displot(data, kde=True, bins=15)

Executing this code produces a visualization that combines the traditional histogram with a continuous line representing the estimated density.



The result is a highly informative plot featuring both the discrete frequency counts (the histogram bars) and the continuous probability density function estimate (the KDE curve). Note that the y-axis in a KDE plot often represents density rather than raw counts, allowing the curve's area to integrate to one, a critical property for probability functions.

This combined visualization is powerful because the KDE curve offers a robust summary of the distribution's shape, even if the histogram itself exhibits slight irregularities due to sampling variability. Analysts frequently rely on the smoothness of the KDE curve to quickly assess if the data approximates a known distribution type, such as a normal or uniform distribution, offering superior visual confirmation compared to a bar plot alone.

Further Exploration in Python Data Visualization

Visualizing data distributions is the first step in understanding dataset characteristics. Whether you

choose the foundational control of Matplotlib or the statistical elegance of Seaborn, mastering these visualization methods is crucial for robust data analysis.

To further enhance your Python visualization skills, consider exploring tutorials on creating other essential chart types, such as scatter plots for correlation analysis, box plots for quartile summaries, and violin plots for comparing multiple distributions simultaneously. These skills collectively form the foundation of effective exploratory data analysis (EDA).

The following tutorials explain how to create other common charts in Python: