Create a Distribution Plot in Matplotlib

Authored by

Mohammed looti

November 16, 2025

<hr>

The effective visualization of data's underlying statistical structure is absolutely essential in any professional data visualization or statistical analysis workflow. Central to this process are distribution plots, which provide an immediate, visual summary of the frequency or probability associated with various values within a dataset. These indispensable plots are critical for revealing key insights regarding the data's shape, its measure of central tendency, and the degree of its spread, or variability. Within the Python data science ecosystem, practitioners predominantly rely on two robust and flexible libraries: Matplotlib and Seaborn, which offer powerful frameworks for generating these necessary visualizations.

This comprehensive tutorial is meticulously structured to guide you through the systematic generation of sophisticated distribution plots, leveraging both the foundational capabilities inherent to Matplotlib and the high-level, statistical feature set provided by Seaborn. We will begin by exploring the creation of classic frequency plots, known universally as histograms, and subsequently demonstrate how to significantly enhance these visualizations by seamlessly integrating smooth probability density curves. This integrated approach offers a more nuanced and statistically profound understanding of the data's inherent structure. By the end of this guide, you will possess the requisite technical knowledge to confidently select the most appropriate methodology for your specific data visualization demands and accurately interpret the resulting plots to formulate sound analytical conclusions.

<h3>Understanding the Core Concept of Data Distribution</h3>

A distribution plot functions as the graphical cornerstone for representing both the scatter and the concentration of numerical data points. Its primary and most vital role is to visually illustrate the range of values a variable takes on and to quantify the relative frequency with which these values occur. This powerful visual representation is invaluable for the rapid identification of underlying data patterns, for comprehensively assessing the dataset's overall variability, and for efficiently detecting potential outliers.

The most common types of distribution plots include histograms, which rigorously display frequency counts of observations partitioned into defined ranges called "bins," and Kernel Density Estimates (KDEs), which provide a smoothed, continuous estimation of the data's probability density function.

For professional data scientists and analysts, establishing a foundational grasp of the data distribution is a non-negotiable prerequisite before initiating any form of complex analytical modeling. This initial visualization step is absolutely critical for making informed decisions regarding necessary data transformations, the appropriate selection of subsequent statistical models, and the formulation of rigorous hypothesis tests. A quick, diagnostic inspection of a distribution plot can immediately reveal whether the data exhibits properties such as perfect symmetry (e.g., a normal distribution), pronounced skewness, or multimodality. These insights are fundamental because they directly dictate the subsequent analytical paths and are essential for guaranteeing the statistical validity and overall reliability of all inferences drawn from the analyzed data.

<h3>Preparing Reproducible Sample Data using NumPy</h3>

Before we can effectively proceed with implementing and customizing our visualization techniques, it is essential to establish a reliable and consistent sample dataset upon which all operations will be performed. For all numerical operations, array manipulation, and sophisticated data generation tasks within Python, the NumPy library is universally recognized as the indispensable standard tool. For the purposes of this demonstration, we will construct a synthetic array comprising exactly 1000 data points that statistically adhere to a normal distribution. This specific distribution pattern, often referred to as the Gaussian distribution, is frequently observed across countless natural phenomena and serves as the baseline assumption for the vast majority of parametric statistical models used in data science.

The following code snippet efficiently leverages the specialized array generation capabilities of NumPy to construct our synthetic yet statistically robust sample data. To ensure that the results are completely reproducible--meaning that executing the code yields the identical dataset every single time, which is critical for sharing analysis--we explicitly set a random seed using <code style="background-color: #f0f0f0">np.random.seed()</code>. The core data

generation is expertly handled by the <code style="background-color: #f0f0f0">np.random.normal()</code> function, which requires three critical parameters: the total size of the resulting array (1000), the desired mean value (<code style="background-color: #f0f0f0">loc</code>, set to 10), and the standard deviation (<code style="background-color: #f0f0f0">scale</code>, set to 2) of the final distribution.

<pre style="background-color: #ececec;font-size: 15px">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()
</pre>

Upon the successful execution of this script, the designated variable, <code style="background-color: #f0f0f0">data</code>, will rigorously hold 1000 continuous numerical values. These values are statistically clustered around the defined mean of 10, with the vast majority (approximately 95%) naturally falling within the range of 6 and 14, which precisely mirrors the statistical characteristics inherent to a normal distribution having a standard deviation of 2. This meticulously prepared array provides the ideal, statistically sound foundation required for effectively demonstrating how to generate, interpret, and subsequently customize sophisticated distribution plots utilizing Python's leading visualization tools, Matplotlib and Seaborn.

<h3>Method 1: Creating a Basic Histogram with Matplotlib</h3>

Matplotlib remains the foundational, comprehensive plotting library in the Python stack, giving users unparalleled, exceptionally granular control over virtually

every aesthetic and structural element of any generated plot. For the specific task of generating a basic frequency plot, or histogram, the primary function of choice is <code style="background-color: #f0f0f0">plt.hist()</code>, which is efficiently accessed via the essential `matplotlib.pyplot` module. This function is highly regarded for its straightforward implementation and extensive customizability, positioning it as the preferred option for both rapid, initial exploratory visualizations and the meticulous, detailed adjustments needed for finalized, publication-quality output.

To visualize the statistical distribution of our newly created NumPy array using Matplotlib, the process is remarkably simple and direct: we pass our <code style="background-color: #f0f0f0">data</code> variable directly as the primary argument to the <code style="background-color: #f0f0f0">plt.hist()</code> function. Beyond the raw data input, we retain the flexibility to specify numerous aesthetic and structural parameters. These customizations are vital for enhancing the plot's immediate visual appeal and are crucial for ensuring that the resulting visualization accurately and effectively communicates the essential characteristics, shape, and structure of the underlying data distribution to the viewer.

<pre style="background-color: #ececec;font-size: 15px">import matplotlib.pyplot as plt

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

The code execution above utilizes several pivotal parameters that meticulously control both the final visual appearance and the structural organization of the histogram. Specifically, the mandatory <code style="background-color: #f0f0f0">data</code> parameter accepts the core numerical array whose frequency distribution is being visualized. The <code style="background-color: #f0f0f0">color</code> argument dictates the internal fill color of the bars ('lightgreen'), while <code style="background-color: #f0f0f0">ec</code> (edge color) sets the color of the borders around each bar ('black'), which is vital for visually separating the individual bins. Most significantly, the <code style="background-color: #f0f0f0">bins</code> argument determines the specific number of equal-width intervals into which the entire data range is subdivided. Setting <code style="background-color: #f0f0f0">bins=15</code> instructs Matplotlib to partition the data space into 15 segments, significantly influencing the visual granularity and interpretation of the

resulting plot.

The resulting histogram provides a clear and immediate visual representation of the data's frequency distribution. The horizontal x-axis rigorously spans the range of numerical values present in the input NumPy array, while the vertical y-axis precisely quantifies the frequency or total count of data points that successfully fall within the boundaries of each specific bin. This arrangement allows the viewer to instantly and intuitively identify which ranges of numerical values are the most common (indicated by the highest bars) and which are the least common (indicated by the lowest bars), thereby offering a rapid and intuitive understanding of the data's central tendencies and overall spread.

 <img class=" wp-image-33255 aligncenter" src="https://stats.arabpsychology.com/wp-content/uploads/2023/07/displot1-1.jpg" width="502" height="368" title="displot1-1">

Mastering the nuanced adjustment of the <code style="background-color: #f0f0f0">bins</code> argument is perhaps the single most crucial step for generating an effective and truthful histogram. Using too large a number of bins can produce an overly detailed plot with many narrow bars, potentially revealing fine-grained anomalies, but often resulting in a visually jagged, noisy appearance dominated by random sampling fluctuations. Conversely, employing too few bins yields a coarser, excessively generalized overview that might inadvertently obscure essential, meaningful features or structures within the distribution. Effective visualization necessitates careful experimentation to determine the optimal number of bins that successfully strikes a balanced trade-off between revealing necessary statistical detail and maintaining visual clarity specific to the dataset under rigorous examination.

<h3>Method 2: Enhancing Histograms with Seaborn and KDE</h3>

While Matplotlib furnishes the essential foundational capabilities for all Python plotting, Seaborn is strategically engineered as a powerful, high-level statistical visualization library built directly upon the Matplotlib framework. Seaborn significantly streamlines the process of generating highly attractive and statistically informative graphics. Its dedicated <code style="background-color: #f0f0f0">displot()</code> function is exceptionally versatile for visualizing univariate distributions. Crucially, <code style="background-color: #f0f0f0">displot()</code> provides a seamless, integrated mechanism to combine the discrete representation of a histogram with a smooth Kernel Density Estimate (KDE)

curve, thereby offering a significantly richer and more robust analytical perspective on the data.

The integration of the KDE curve provides a continuous, estimated representation of the data's underlying probability density function. This powerful statistical technique effectively circumvents certain inherent limitations associated with traditional histograms, most notably their high sensitivity and dependency on the arbitrary choice of bin width. By intelligently overlaying the continuous KDE on top of the discrete histogram bars, analysts benefit immensely from a dual perspective: they receive the raw, observed frequency counts provided by the histogram combined with a statistically smooth approximation of the true underlying probability distribution. This synergy makes it considerably easier and more reliable to accurately discern the natural, continuous shape of the data without visual noise.

<pre style="background-color: #ececec;font-size: 15px">import seaborn as sns

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

Within the robust <code style="background-color: #f0f0f0">sns.displot()</code> function call, three parameters hold particular significance for achieving this enhanced visualization. The <code style="background-color: #f0f0f0">data</code> parameter, as in Matplotlib, provides the essential dataset to be analyzed and plotted. The highly critical <code style="background-color: #f0f0f0">kde=True</code> parameter serves as the command to calculate and then seamlessly overlay the smooth Kernel Density Estimate curve onto the histogram plot. If this parameter were intentionally omitted or explicitly set to <code style="background-color: #f0f0f0">False</code>, the function would automatically revert to displaying only the basic frequency histogram. Finally, the <code style="background-color: #f0f0f0">bins</code> parameter functions identically to its counterpart in Matplotlib, meticulously controlling the level of granularity for the histogram component of the combined visualization.

 <img loading="lazy" class=" wp-image-33256 aligncenter" src="https://stats.arabpsychology.com/wp-content/uploads/2023/07/displot2.jpg" width="497" height="495" title="displot2">

The resulting visualization represents a highly effective and harmonious combination of the discrete, categorical nature of the histogram bars and the fluid, continuous line of the KDE. This integrated approach is exceptionally effective for succinctly summarizing the overall shape of the distribution, enabling quick and reliable identification of key features such as peaks (modes), troughs (valleys), and general trends, all without the inherent visual distraction often caused by the high sensitivity to individual bin choices. Fundamentally, the KDE curve operates as a sophisticated non-parametric estimator of the underlying probability density function of the random variable, thereby offering continuous and profound statistical insight into the data's true likelihood profile.

<h3>Choosing the Optimal Visualization Tool: Matplotlib vs. Seaborn</h3>

While both Matplotlib and Seaborn are undeniably powerful and highly effective libraries for generating distribution plots, they are fundamentally designed to cater to distinct requirements and analytical preferences. A clear and comprehensive understanding of the core strengths, design philosophies, and inherent limitations of each library is absolutely essential for optimizing one's data science workflow and maximizing the clarity and overall impact of the resulting data visualizations. This deliberate choice between the two is a key determinant of efficient and effective data analysis.

Matplotlib's primary and undeniable advantage lies in its capacity to offer unparalleled, granular control over virtually every constituent element of a plot. If the requirement involves the meticulous fine-tuning of individual axis tick marks, the precise placement and sizing of plot components within a canvas, or the complex construction of highly specialized, multi-panel layouts, Matplotlib furnishes the necessary low-level access via its exhaustive API. It is critically important to remember that Matplotlib serves as the fundamental underlying graphical engine that powers a vast ecosystem of Python plotting libraries, including Seaborn itself. However, this superior degree of low-level control often necessitates writing a significantly greater volume of code, even for generating relatively simple plots, which can consequently increase overall development time and verbosity for routine tasks.

Conversely, Seaborn is expertly designed to excel at generating aesthetically pleasing, publication-quality, and statistically informative plots with the absolute minimum amount of code required. It intelligently applies sensible and attractive defaults for color palettes, general plot styles, and statistical estimation methods, making it the ideal tool for rapid Exploratory

Data Analysis (EDA). Seaborn's high-level functions, such as <code style="background-color: #f0f0f0">displot()</code>, frequently integrate multiple visualization types--like the histogram and KDE--into a single, concise function call. This powerful integration drastically simplifies the creation of complex visualizations and significantly reduces the amount of repetitive boilerplate code necessary for producing high-quality statistical graphics.

In summation, analysts should opt for Matplotlib when the project demands maximum, intricate customization, when building deeply complex, multi-layered plots from their base components, or when the plots must be seamlessly integrated into specialized application environments where precise control is paramount. Conversely, one should choose Seaborn when the priority is ease of implementation, speed, attractive visual defaults, and robust statistical plotting functionalities, especially during the critical phase of exploratory analysis where the goal is to quickly and reliably grasp complex statistical relationships and distributions. It is common practice for these libraries to be used synergistically, with Seaborn handling the initial generation of high-level statistical plots, while Matplotlib is subsequently utilized for final, detailed modifications and stylistic refinements.

<h3>Best Practices for Generating Effective Distribution Plots</h3>

The professional process of generating truly effective distribution plots involves far more than merely executing a single line of code; it requires a thoughtful, strategic consideration of several key factors to ensure that the resulting visualization accurately, clearly, and compellingly conveys insightful information about the underlying dataset. Adherence to established visualization best practices will fundamentally enhance the overall quality, interpretability, and communicative power of your final plots, transforming raw data into actionable knowledge.

Optimal Bin Selection: The strategic choice of the number of <code style="background-color: #f0f0f0">bins</code> in a histogram is perhaps the single most critical decision impacting visual fidelity. Employing too few bins risks obscuring the true, unique shape of the distribution, potentially making distinct modes indistinguishable and leading to under-interpretation. Conversely, selecting too many bins can cause the plot to appear excessively noisy, inadvertently highlighting random fluctuations and obscuring generalized, meaningful patterns. It is highly recommended to experiment systematically with various bin numbers or, when leveraging Seaborn, to rely on its powerful statistical algorithms to automatically determine an optimal and statistically defensible bin width.

Clear Axis Labels and Descriptive Titles: It is imperative to always label both the x and y axes explicitly and clearly, and to furnish the plot with a concise yet highly descriptive title. This essential contextual information is non-negotiable for any individual attempting to accurately interpret the visualization. For example, the x-axis must clearly identify the specific variable values being measured, and the y-axis must unambiguously state whether the scale represents raw frequency count, relative frequency, or estimated probability density.

Judicious Color Choices: Colors must be selected judiciously and purposefully, serving an analytical function rather than just aesthetic decoration. The chosen colors should be visually appealing, distinct, and, critically, must effectively differentiate multiple distributions if they are being compared on the same plot. Analysts should actively avoid the use of overly saturated or clashing colors, and it is a professional best practice to consider colorblind-friendly palettes when anticipating a general or diverse audience.
&li>

Handling and Contextualizing Outliers: Distribution plots serve as an excellent initial diagnostic tool for identifying potentially influential outliers. Depending entirely on the specific objectives of the analysis, one might choose to explicitly visualize these outliers to understand their overall impact on the distribution's shape, or alternatively, preprocess the data to mitigate their influence before plotting, ensuring they don't unduly skew the fundamental visual interpretation of the main data cluster.

Interpretation within Context: The final and most crucial step is to always interpret the plot within the specific context of the data and the core problem being addressed. While a symmetric, bell-shaped normal distribution might be expected in certain statistical scenarios, the presence of a significantly skewed or multimodal distribution could hold immense scientific or business significance in others, often serving as a powerful prompt for deeper, specialized investigation.

<p>
By rigorously adhering to these established visualization best practices, data professionals can consistently create distribution plots that are not only highly engaging and aesthetically optimized but also profoundly informative and readily understandable, thereby effectively communicating the

data's narrative and robustly supporting sound analytical conclusions across any domain.

<h3>Additional Resources for Advanced Data Visualization</h3>

The mastery of distribution plots, while fundamental, represents only a single, foundational stage in the expansive and evolving journey of modern data visualization. To further expand your professional skill set in generating insightful, high-quality, and engaging statistical charts using Python, we strongly encourage exploring these related documentation and advanced tutorial resources:

Official Matplotlib Documentation: Offers comprehensive, in-depth guides covering all standard plot types and detailing advanced customization options for expert users seeking granular control.
&li>

Official Seaborn Documentation: Provides rich, comprehensive resources specifically focused on advanced statistical plotting, leveraging its beautiful default aesthetics and powerful statistical integration capabilities.
&li>

Creating Scatter Plots in Matplotlib: Essential learning material for visualizing relationships, patterns, and correlations between any two continuous variables within a dataset, crucial for bivariate analysis.
&li>

Generating Box Plots for Outlier Detection: Detailed instruction on how to accurately understand data spread, identify quartiles (Q1, Median, Q3), and effectively detect potential outliers using standardized box and whisker plots.
&li>

Visualizing Time Series Data with Line Plots: Expert techniques dedicated to plotting ordered data points over sequential time intervals to accurately reveal critical temporal trends, seasonality, and long-term patterns in time-dependent data.
&li>

These curated resources are designed to help you construct a comprehensive and powerful toolkit for effective data visualization in Python, ultimately enabling you to convey compelling, accurate, and robust analytical stories with the data you analyze.
</p>