

How to Display Percentages on Your Pandas Histogram Y-Axis

Authored by
stats writer

November 21, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Display Percentages on Your Pandas Histogram Y-Axis*.
PSYCHOLOGICAL SCALES. Retrieved from <https://scales.arabpsychology.com/?p=98828>

To display the percentage on the Y-axis of a Pandas histogram, a two-step process is required: normalization of the bin heights using the `weights` argument, and applying the `PercentFormatter` function from the Matplotlib library. This combined approach ensures that the vertical axis accurately displays the relative frequency of each bin as a percentage of the total dataset.

The Need for Normalized Histograms

When generating a histogram, the default behavior in both Pandas and Matplotlib is to plot the absolute frequency or count of observations falling into each bin on the Y-axis. While useful for understanding raw distribution sizes, this count-based approach can obscure the true relative frequency, making direct comparisons between datasets of differing sizes challenging. The ability to display the relative frequency, or the percentage of total observations, is critical for standardizing visualizations and providing immediate context regarding data distribution.

To convert these raw counts into true proportions or percentages, we must normalize the data. Normalization ensures that the area under all the bars sums up to 1 (representing 100% of the data). Although Pandas offers a `density=True` argument in its plotting functions, for granular control and precise percentage labeling on the Y-axis--specifically integrating the percent symbol--we rely on specialized components from the underlying visualization library, Matplotlib. This method involves manually calculating the weights corresponding to the relative frequency of each bin.

By shifting the focus from absolute counts to relative percentages, our visualization becomes immediately more interpretable for audiences interested in distribution characteristics rather than total sample size. This transition requires a specific configuration of plot parameters, primarily leveraging the `weights` argument during the plotting phase and subsequently applying a specialized formatting tool to label the axis correctly. This technique ensures mathematical accuracy in the height calculation and visual clarity in the axis labeling.

Understanding the Core Mechanism: Weights and Formatters

The solution relies on two key components working in tandem. First, to tell the Matplotlib plotting function to calculate relative frequencies instead of counts, we utilize the `weights` parameter in the `plt.hist()` function. The `weights` array must be constructed such that the sum of the weights equals 1. For a DataFrame column, the appropriate weight for every single observation is calculated using NumPy as `np.ones(len(df)) / len(df)`. This ensures that when the function plots the data, the height of each bin accurately represents the proportion of total observations contained within that specific bin.

Second, even after setting the weights correctly, the Y-axis tick labels will still display decimal

values (e.g., 0.10, 0.25) representing the proportion. To visually format these proportions as percentages (e.g., 10%, 25%), we must import and apply the `PercentFormatter` utility from `matplotlib.ticker`. This utility is responsible purely for presentation and does not affect the underlying data calculation performed by the `weights` argument. It transforms the numerical tick labels into readable percentage strings by multiplying them by 100.

The `PercentFormatter` is applied directly to the Y-axis object, which is accessed via `plt.gca().yaxis`. The argument passed to the formatter, typically `PercentFormatter(1)`, specifies the scaling factor. Because the weights were calculated correctly to sum to 1, the scale factor of 1 tells the formatter that the input (the Y-axis values) are already normalized proportions that simply need to be displayed as a percentage. This essential two-step process--calculation via `weights` and formatting via `PercentFormatter`--yields the desired percentage display.

Essential Python Setup and Syntax

The following syntax block demonstrates the fundamental setup required to generate a relative frequency histogram using the combined power of Pandas, NumPy, and Matplotlib. Note the required imports for numerical operations and specialized axis formatting:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter

#create histogram, using percentages instead of counts
plt.hist(df, weights=np.ones(len(df)) / len(df))

#apply percentage format to y-axis
plt.gca().yaxis.set_major_formatter(PercentFormatter(1))
plt.show()
```

This code snippet is the blueprint for achieving percentage-based visualization. The critical line is the calculation of `weights`, where we divide an array of ones (matching the length of the data) by the total number of data points. This ensures that each observation contributes equally to the total sum, effectively normalizing the distribution so that the area of all bars sums to 1. Subsequently, `plt.gca()` retrieves the current axes object, allowing us to manipulate the Y-axis properties and apply the specific formatting provided by `PercentFormatter`.

For users accustomed to the simplified plotting methods offered by Pandas directly (e.g., `df.hist()`), this approach requires slightly more verbose Matplotlib integration. However, this level of control is necessary because it grants access to the specific axis formatting tools that

Pandas wrappers often simplify or abstract away, preventing the straightforward addition of percentage labels.

Practical Example: Preparing the Dataset

To illustrate this technique practically, we will generate a sample dataset mimicking statistics for basketball players. This dataset, stored in a Pandas DataFrame, will contain three columns: points, assists, and rebounds. We utilize NumPy's random functions to ensure the data follows a normal distribution, making the resulting histogram meaningful for distribution analysis.

It is standard practice when working with random data generation in examples to set a seed. Using `np.random.seed(1)` ensures that the generated dataset is reproducible across different executions, meaning anyone running the code will see the exact same distribution and results. The points column, which we will analyze, is centered around a mean (loc) of 20 with a standard deviation (scale) of 2, creating 300 data points in total for our sample population.

The code below sets up the data structure and displays the first few rows to confirm the creation of the synthetic statistical data, preparing us for the visualization steps that follow.

```
import pandas as pd
import numpy as np

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

#create DataFrame
df = pd.DataFrame({'points': np.random.normal(loc=20, scale=2, size=300),
'assists': np.random.normal(loc=14, scale=3, size=300),
'rebounds': np.random.normal(loc=12, scale=1, size=300)})

#view head of DataFrame
print(df.head())

points assists rebounds
0 23.248691 20.197350 10.927036
1 18.776487 9.586529 12.495159
2 18.943656 11.509484 11.047938
3 17.854063 11.358267 11.481854
4 21.730815 13.162707 10.538596
```

Default Histogram Behavior

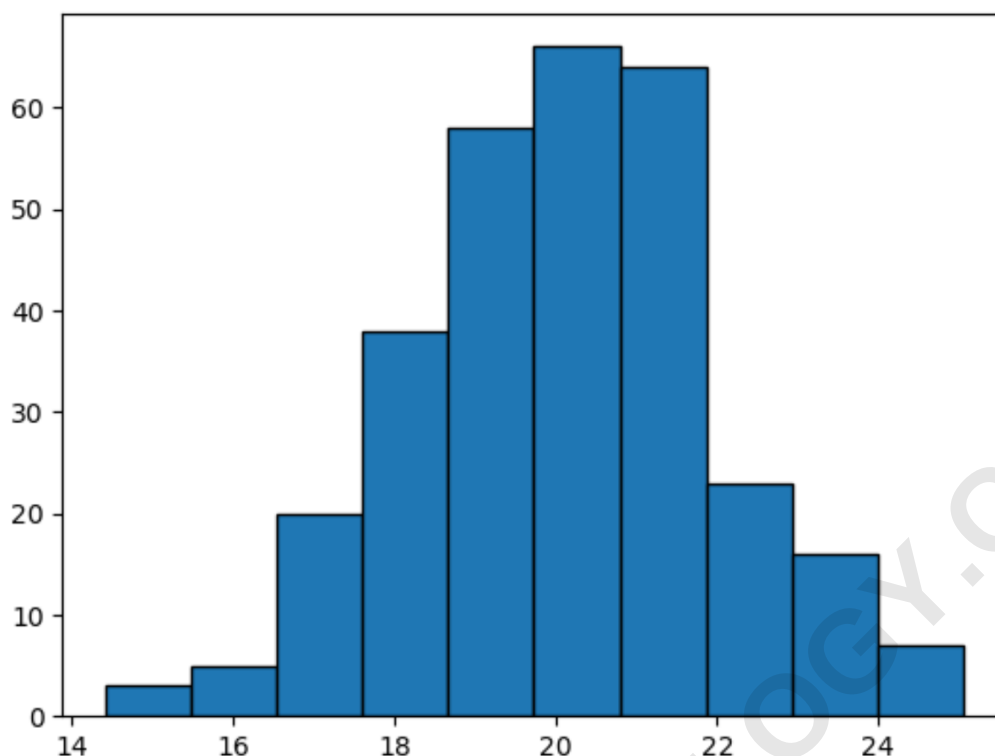
Before implementing the percentage formatting, it is helpful to visualize the standard output when plotting the distribution of the `points` column. By default, when we call `plt.hist()` without specifying the `weights` argument or setting `density=True`, the vertical Y-axis automatically scales to display the raw count (frequency) of data points falling into each bar's bin. Since our DataFrame has 300 entries, the maximum height of any bar will correspond to the bin holding the highest number of observations, giving us the absolute count of players in that scoring range.

The following code generates this baseline visualization. We explicitly add `edgecolor='black'` simply to enhance the visual separation between the bars, a common practice in producing clean histograms.

```
import matplotlib.pyplot as plt
```

```
#create histogram for points column  
plt.hist(df, edgecolor='black')
```

As observed in the resulting image below, the Y-axis labels run from 0 up to a maximum count of approximately 50, confirming that we are plotting absolute frequencies. While this is informative, it requires mental translation if we want to determine what percentage of players scored between specific point ranges.



Implementing the Percentage Display using PercentFormatter

To successfully transform the visualization from raw counts to relative percentages, we must apply the two-part strategy introduced earlier. First, we calculate the normalization weights using `NumPy`. Since we have 300 data points, the proportional weight for every single point is $1/300$. We pass this resulting array into the `weights` parameter of the `plt.hist()` function. This normalization ensures the height of the bars now represents proportions (values between 0 and 1).

Second, we apply the specialized formatting utility. We import `PercentFormatter` from `matplotlib.ticker` and apply it to the current figure's Y-axis using `plt.gca().yaxis.set_major_formatter()`. The resulting visualization clearly indicates the relative distribution of the data, showing, for example, that the most frequent bin accounts for approximately 15% to 20% of all observations.

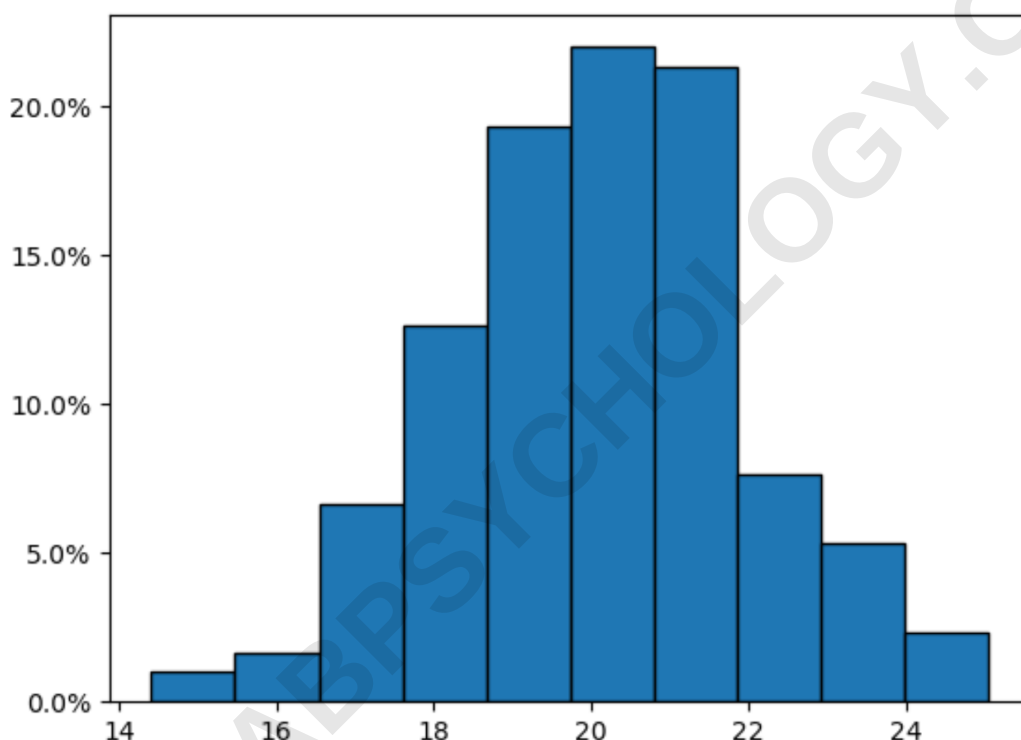
The code below executes this transformation, resulting in a histogram where the Y-axis is clearly labeled with percentages, greatly enhancing the immediate readability and statistical utility of the chart.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import PercentFormatter
```

```
#create histogram, using percentages instead of counts
plt.hist(df, weights=np.ones(len(df)) / len(df), edgecolor='black')

#apply percentage format to y-axis
plt.gca().yaxis.set_major_formatter(PercentFormatter(1))
plt.show()
```

Upon reviewing the generated image, it is evident that the vertical axis now displays values such as 5.0%, 10.0%, 15.0%, etc. This confirms the successful implementation of both the weighting calculation and the axis formatting.



Refining the Visualization: Removing Decimal Places

In many reporting or presentation contexts, displaying percentages with high precision (one or more decimal places) can be distracting or unnecessary, especially when the proportions themselves are relatively coarse. The `PercentFormatter` object offers a straightforward way to control the displayed precision using the `decimals` argument, ensuring a cleaner visual output.

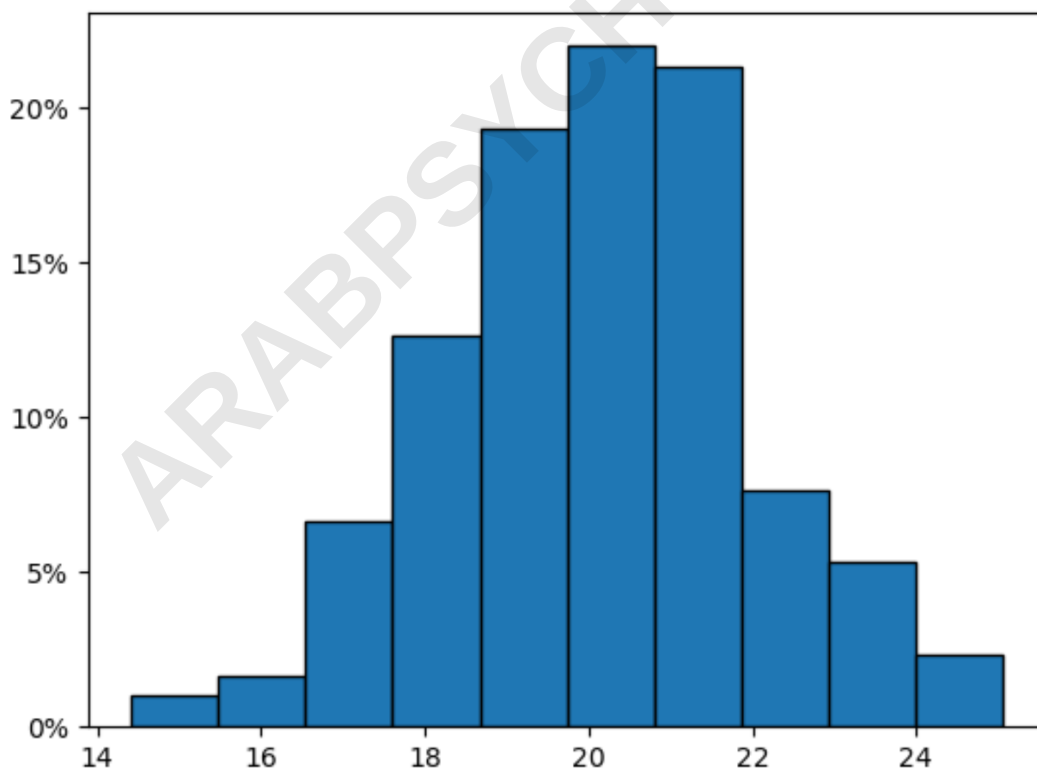
To ensure cleaner tick labels that only show whole numbers (integers), we modify the call to the formatter by setting `decimals=0`. This adjustment affects only the visual presentation of the labels; it does not change the underlying proportional calculation of the histogram bins, which remains

precise. Using this parameter allows for customization based on audience requirements.

The refined code below demonstrates this implementation, providing a visually cleaner histogram suitable for simplified reports where high precision is not mandatory.

```
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.ticker import PercentFormatter  
  
#create histogram, using percentages instead of counts  
plt.hist(df, weights=np.ones(len(df)) / len(df), edgecolor='black')  
  
#apply percentage format to y-axis  
plt.gca().yaxis.set_major_formatter(PercentFormatter(1, decimals=0))  
plt.show()
```

After running this final version of the code, we can confirm via the image that the Y-axis tick labels are now displayed as clean integers (e.g., 0%, 5%, 10%, 15%, 20%), demonstrating precise control over the visual output of our relative frequency histogram.



Summary of Key Steps

Generating a percentage-based histogram involves overriding the default count visualization by implementing proportional weights. This process is complex because it requires tight integration between the data handling capabilities of Pandas and the specialized plotting utilities of Matplotlib. For ease of reference, the crucial steps are summarized below:

Data Normalization: Calculate proportional weights for every data point using NumPy's `np.ones(len(df)) / len(df)`.

Plotting: Pass the calculated weights array to the `weights` parameter in `plt.hist()`.

Formatting Import: Import the `PercentFormatter` class from `matplotlib.ticker`.

Axis Application: Use `plt.gca().yaxis.set_major_formatter()` to apply the formatter to the Y-axis.

Precision Control: Optionally use the `decimals=0` argument within `PercentFormatter` to control the numerical precision of the resulting percentage labels.

By mastering these techniques, data analysts can produce histograms that are not only statistically accurate but also highly effective and readable for communicating frequency distributions across varying sample sizes.