How to Easily Reorder Items in Your Matplotlib Legend

Authored by

stats writer

December 3, 2025

Matplotlib is a fundamental library for data visualization in Python. While generating complex plots, the accompanying legend is crucial for interpreting the data series. By default, Matplotlib orders legend entries based on the sequence in which the corresponding plot elements (lines, bars, etc.) were added to the axes. However, this default sequence may not always align with logical presentation requirements or visual hierarchy. Achieving a customized order is essential for creating publication-quality graphics that guide the reader effectively through the visualization.

The `plt.legend()` function provides comprehensive control over the appearance and placement of these explanatory components. To override the default sorting, we must explicitly pass the desired order of labels and their associated graphical representations to this function. This process involves retrieving the underlying graphical elements (known as handles) and their corresponding textual identifiers (labels), manipulating their sequence, and then reconstructing the legend with the revised structure. This method grants precise control, moving beyond simple keyword argument adjustments to achieve sophisticated visualization results.

Understanding the core components of a Matplotlib plot--the figure, the axes, the artists, and the legend--is key to successful manipulation. The most robust approach for reordering requires interacting directly with the Current Axes (gca()) object, which holds the record of all plotted elements. By accessing this record, we can extract the necessary information--the handles and labels--before applying our custom sort index. This technique is universally applicable whether you are dealing with line plots, scatter plots, or complex multi-series visualizations.

The standard procedure to change the order of items in a Matplotlib legend involves capturing the plot elements and then passing them back to `plt.legend()` in a new sequence defined by a custom index array. This powerful technique overrides the automatic ordering generated by the plotting commands. The key steps are obtaining the handles and labels associated with the plotted series, defining the desired numerical index order, and finally, calling `plt.legend()` using list comprehension to apply this custom sequence.

The Mechanism of Reordering: Understanding Handles and Labels

Before we implement the reordering, it is critical to grasp how Matplotlib manages the components of a legend. Every element plotted that is intended to appear in the legend (e.g., a line, a set of points) is represented by an "artist" object, referred to as a **handle**. Simultaneously, each handle is associated with a descriptive string, or **label**, which is defined when the plotting function (like `plt.plot()`) is called using the `label` keyword argument. Matplotlib stores these pairs internally, and to customize the order, we must retrieve these pairs explicitly.

We use the `plt.gca().get_legend_handles_labels()` method to extract these lists. The `plt.gca()` function stands for "Get Current Axes," returning the active Axes object where the data is being

drawn. Calling `get_legend_handles_labels()` on this object returns two synchronized lists: one containing the handles (the physical representation of the lines/markers) and one containing the labels (the text descriptions). By sorting these two lists simultaneously based on a custom index, we can dictate the final presentation order.

The following code block outlines the essential logic required to capture these components and apply a custom ordering index. This pattern is the foundation for advanced legend manipulation in Matplotlib.

#get handles and labels
handles, labels = plt.gca().get_legend_handles_labels()

#specify order of items in legend
order =

#add legend to plot
plt.legend(for idx in order], for idx in order])

Step-by-Step Guide to Implementing Custom Order

To successfully reorder your legend entries, follow these distinct steps. First, ensure all your data series are plotted and that the `label` keyword argument has been used for each series intended for the legend. If labels are missing, Matplotlib cannot automatically associate a description with the handle, and the process will become complicated.

Second, execute the command to retrieve the handles and labels. It is vital to understand that the order in which `get_legend_handles_labels()` returns these lists is the default order--the sequence in which the plot commands were initially executed. For instance, if you plotted 'Points', then 'Assists', and then 'Rebounds', the labels list will be ``.

Third, define the custom `order` list. This list must contain indices referencing the positions in the original handles/labels lists. If the original order is indexed 0, 1, 2, and you want the new order to be 1, 2, 0, then `order = `. The length of the `order` list must exactly match the number of handles and labels retrieved.

Finally, call `plt.legend()`, passing the restructured lists. We use list comprehensions (` for idx in order]`) to iterate through the custom `order` index list and pull the corresponding handle or label from the original lists. This dynamic selection creates two new, reordered lists--the handles and the labels--which are then passed as positional arguments to the `plt.legend()` function, thereby applying the new visual sequence to the plot.

Practical Example: Setting up the Initial Visualization

To illustrate this technique, we will create a multi-series line chart using the popular Pandas library for data handling, followed by plotting the data with Matplotlib. This setup generates the default legend order that we intend to modify later. The dataset simulates performance metrics (Points, Assists, Rebounds) over several observations.

In the following code block, notice that the lines are added to the plot in a specific sequence: 'Points' first, 'Assists' second, and 'Rebounds' third. This sequence dictates the default order of the legend entries, which often reflects the chronological addition to the plot rather than the desired interpretive structure (e.g., ordering by statistical importance or alphabetical criteria).

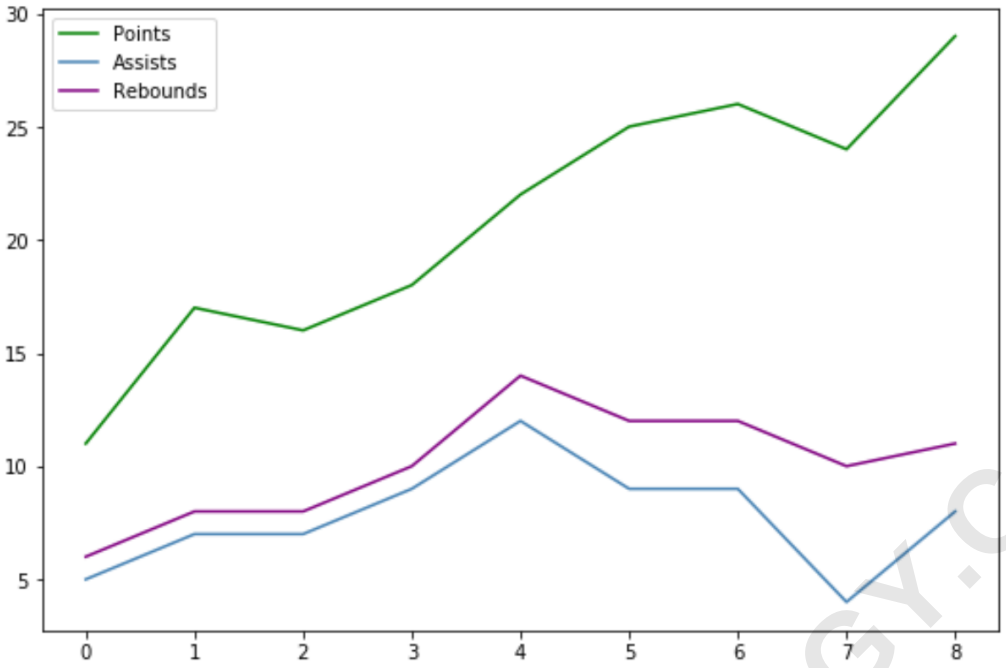
```
import pandas as pd
import matplotlib.pyplot as plt

#create data
df = pd.DataFrame({'points': ,
'assists': ,
'rebounds': })

#add lines to plot
plt.plot(df, label='Points', color='green')
plt.plot(df, label='Assists', color='steelblue')
plt.plot(df, label='Rebounds', color='purple')

#add legend
plt.legend()
```

Executing this code produces a plot where the legend items reflect the order of execution. If we observe the resulting visualization, which is displayed below, we can confirm the default ordering of entries: Points, Assists, and then Rebounds.



Analyzing the Default Legend Output

As demonstrated in the initial visualization, the items in the legend are placed in the precise order that the lines were added to the plot. This default behavior of `plt.legend()` is usually sufficient but becomes problematic when the plotting logic does not match the desired visual hierarchy. For instance, if 'Rebounds' is considered the primary metric, it should ideally appear first in the legend, regardless of when its line was added to the axes.

In our current example, the original indices are: **0** ('Points'), **1** ('Assists'), and **2** ('Rebounds'). To achieve a more meaningful presentation, suppose we decide that the metrics should be ordered based on their typical numerical value magnitude (Assists being the lowest, Rebounds intermediate, and Points highest) or perhaps simply in an order that aids visual comparison across the plot. We need a way to map the existing indices to a new sequence.

This is where the power of programmatic reordering shines. Instead of physically changing the plotting code--which can be cumbersome, especially in complex plotting functions--we utilize the handles and labels mechanism to post-process the legend appearance without altering the underlying figure artists. This separation of concerns allows for cleaner, more maintainable code.

Implementing the Custom Order

To implement a custom order, we retrieve the handles and labels and then define a specific index mapping that dictates the new presentation. We aim to change the sequence from to .

The original index positions are:

Position 0: "Points"
Position 1: "Assists"
Position 2: "Rebounds"

To achieve the desired order , we need to select the element at index 1 first, then the element at index 2, and finally the element at index 0. Therefore, our custom index list, `order`, must be set to ``.

The following syntax incorporates the retrieval of handles and labels and applies this index mapping using list comprehension, yielding the customized legend display.

import pandas as pd
import matplotlib.pyplot as plt

#create data
df = pd.DataFrame({'points': ,
'assists': ,
'rebounds': })

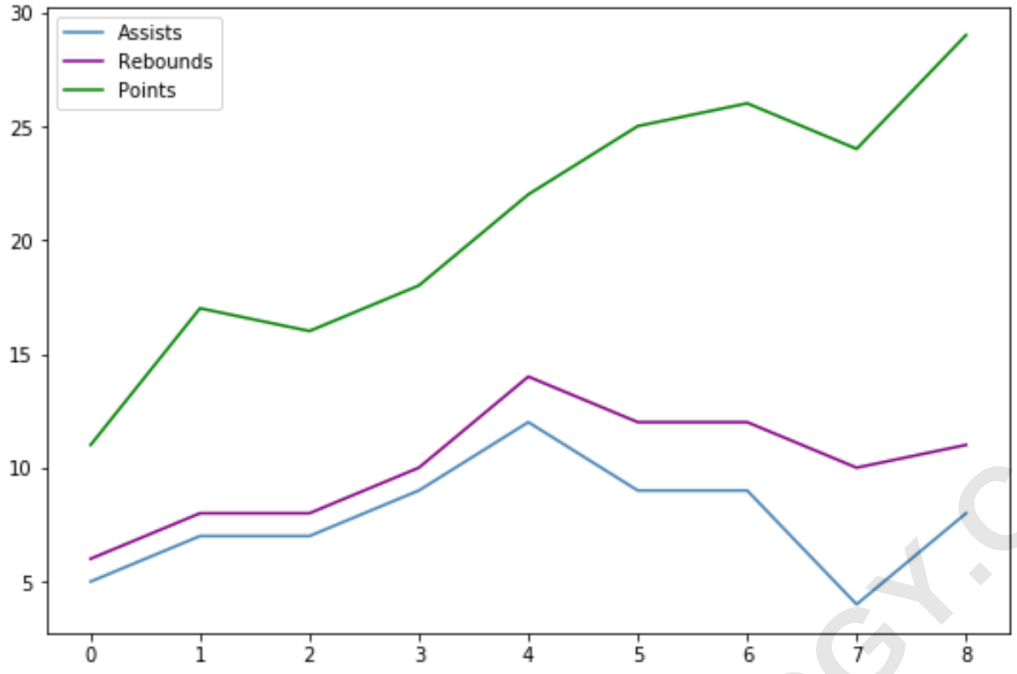
#add lines to plot
plt.plot(df, label='Points', color='green')
plt.plot(df, label='Assists', color='steelblue')
plt.plot(df, label='Rebounds', color='purple')

#get handles and labels
handles, labels = plt.gca().get_legend_handles_labels()

#specify order of items in legend
order =

#add legend to plot
plt.legend(for idx in order], for idx in order])

The resulting plot now features a legend where the items appear in the newly defined order: Assists, followed by Rebounds, and finally Points. This successful manipulation confirms that we can control the legend sequence independently of the plot creation sequence.



Deconstructing the Custom Index

It is crucial to fully understand how the `order = ` index array functions in conjunction with the original handles and labels. This array acts as a map, instructing Matplotlib which element to select next from the retrieved lists.

We specified the mapping as:

The first item in the new legend is determined by the first element of `order`, which is **1**. This selects the label that was originally at index position 1: **"Assists"**.
The second item in the new legend is determined by the second element of `order`, which is **2**. This selects the label that was originally at index position 2: **"Rebounds"**.
The third item in the new legend is determined by the third element of `order`, which is **0**. This selects the label that was originally at index position 0: **"Points"**.

This method ensures that the graphical element (handle) corresponding to 'Assists' is paired precisely with the label 'Assists', maintaining the integrity of the data representation while granting complete control over the display sequence.

Advanced Customization Techniques for Matplotlib Legends

While reordering is a key customization, the `plt.legend()` function offers numerous keyword arguments to further enhance the visual appeal and clarity of the legend. Once you have established the correct order using the handle/label method, you can pass additional parameters to

the final `plt.legend()` call.

For instance, the `loc` keyword argument allows precise placement, such as `loc='upper right'` or `loc=(1.05, 1.0)` for external positioning. Other useful parameters include `ncol` to specify the number of columns (useful for horizontal legends), `frameon` to toggle the background box, and `fontsize` to adjust text size. Combining custom ordering with these graphical parameters results in highly polished, professional visualizations.

Furthermore, for highly complex plots or plots generated using object-oriented methods (using `ax.plot()` instead of `plt.plot()`), the same principle applies. You would call `ax.get_legend_handles_labels()` on the specific Axes object (`ax`) rather than using `plt.gca()`, reinforcing the flexibility and robustness of this manipulation strategy across different Matplotlib workflows. Mastering the handles and labels approach is essential for any advanced visualization task.