How to create a Table with Matplotlib

Authored by

stats writer

December 18, 2025

The ability to generate clear and informative visualizations is paramount in data analysis, and while Matplotlib is primarily known for plotting charts and graphs, it also offers robust functionality for displaying tabular data directly within figures. Creating a table using Matplotlib is achieved through the powerful .table() method, which is essential when presenting data that requires precise row and column formatting alongside graphical representations. This method allows analysts to integrate numerical summaries or categorical data seamlessly into their visual output, providing context and clarity without needing external spreadsheet applications.

The .table() method demands specific inputs to construct the visual element correctly. Fundamentally, it requires a structured data source, typically provided as a list of lists or equivalent nested data structure, which represents the content for each cell. Additionally, clear identification of columns is achieved using a list of strings designated as column headers (or labels). Beyond mere structure, Matplotlib provides extensive options for aesthetic customization. Users can easily adjust parameters such as cell colors, font properties, and line widths, ensuring the table adheres to specific visualization standards and branding guidelines. Once the table object is initialized and configured, it is managed within the Matplotlib figure, ready to be rendered and displayed using standard commands like the .show() method.

Understanding the `matplotlib.pyplot.table()` Method

The core functionality for embedding tables within a Matplotlib figure resides in the .table() method, which typically operates on an Axes object. Proper utilization of this function requires a deep understanding of its primary arguments. The fundamental data input is handled by the `cellText` parameter, which expects a sequence of sequences (like a list of lists or a 2D NumPy array) containing the data points for each cell. This input structure dictates the number of rows and columns the resulting table will possess. A crucial aspect of making the table readable is defining the column headers, which are passed via the `colLabels` argument, requiring a list of strings matching the number of columns in the `cellText` data.

Furthermore, controlling the position of the table within the figure is achieved through the `loc` parameter. This parameter accepts various strings (e.g., 'center', 'bottom', 'top right') that specify where the table should be anchored relative to the subplot area. If `colLabels` are provided, they appear at the top of the table. If row labels are also desired, the `rowLabels` argument can be utilized. Fine-tuning the appearance is achieved using parameters like `cellColours`, which allows for distinct background colors for individual cells, and `cellLoc` and `colLoc`, which control the alignment of text within the cells and headers, respectively. Mastering these arguments is key to transforming raw data into a polished, embedded table visualization.

When integrating tables into Python visualizations using Matplotlib, two primary workflows are utilized, depending on the source and structure of the data: creating a table directly from a powerful

pandas DataFrame, which is common in analytical pipelines, or generating a table from custom, manually defined lists of values. Both methods leverage the core `ax.table()` function but differ in how the data is prepared and passed.

Method 1: Create Table from pandas DataFrame

This approach is highly efficient for data scientists who manage their structured data using the pandas library. By utilizing the `.values` attribute of the DataFrame for cell content and the `.columns` attribute for column labels, the necessary data structures for `ax.table()` are easily extracted.

#create pandas DataFrame
df = pd.DataFrame(np.random.randn(20, 2), columns=)

#create table
table = ax.table(cellText=df.values, colLabels=df.columns, loc='center')

Method 2: Create Table from Custom Values

Alternatively, if the data is small, static, or originates from a source other than pandas, the table can be constructed using standard Python list structures. This method provides direct control over the data input, although column labels must be explicitly defined if required (they are omitted in this simplified example).

#create values for table
table_data=,
,
,
,

]

#create table
table = ax.table(cellText=table_data, loc='center')

The following detailed tutorial provides practical, step-by-step examples demonstrating how to implement these two critical methods for effective table generation in Matplotlib visualizations.

Generating Tables from a pandas DataFrame

One of the most frequent use cases for displaying data tables is integrating summarized or raw

data from a pandas DataFrame directly into a Matplotlib figure. This integration is seamless because pandas objects are designed to interoperate efficiently with scientific computing libraries like Matplotlib and NumPy. When creating a table from a DataFrame, the primary challenge is not displaying the data itself, but rather configuring the Matplotlib environment--the figure and the axes--to accommodate the table visualization, often by suppressing standard plotting elements like ticks and spines that are unnecessary for a textual display.

To ensure the table is the focal point of the output, preliminary setup usually involves defining the figure (**fig**) and axes (**ax**) objects, and then explicitly hiding the axes. Hiding the axes is accomplished by setting the figure patch to invisible using `fig.patch.set_visible(False)` and turning off the axes visibility using `ax.axis('off')`. This sequence effectively removes the default borders, tick marks, and labels associated with a typical plot, resulting in a clean canvas dedicated solely to the tabular representation. We also use `ax.axis('tight')` to ensure the axes limits automatically adjust to fit the data, which is especially useful when dealing with dynamic or randomly generated content, as seen in the subsequent example.

The crucial connection between the DataFrame and the Matplotlib table is made via the `cellText` and `colLabels` parameters within the `ax.table()` call. The content for the cells is extracted using `df.values`, which returns a NumPy array representation of the DataFrame's data, perfectly suited for the `cellText` argument. Similarly, the column names are extracted using `df.columns`, providing the list of strings needed for `colLabels`. This streamlined process ensures that the column headers in the table accurately reflect the structure of the underlying DataFrame, maintaining data integrity and clarity in the visualization.

Example 1: Create Table from pandas DataFrame

The comprehensive code block below demonstrates the necessary steps to initialize the environment, generate synthetic data using NumPy and pandas, and subsequently render this data as a structured table within a Matplotlib figure. We utilize `np.random.seed(0)` to ensure that the random data generated is reproducible, allowing users to replicate the exact output shown here.

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

#make this example reproducible
np.random.seed(0)

#define figure and axes
fig, ax = plt.subplots()
```

```
#hide the axes
fig.patch.set_visible(False)
ax.axis('off')
ax.axis('tight')

#create data
df = pd.DataFrame(np.random.randn(20, 2), columns=)

#create table
table = ax.table(cellText=df.values, colLabels=df.columns, loc='center')

#display table
fig.tight_layout()
plt.show()
```

After executing this code, the `fig.tight_layout()` function is called, which automatically adjusts subplot parameters to give a tight layout, preventing labels or elements from overlapping, ensuring the large table fits neatly within the figure boundaries. The resulting image showcases a cleanly presented table derived directly from the random numbers contained within the pandas DataFrame, successfully demonstrating the Method 1 workflow.

First	Second
1.7640523459676 64	0.4001572083672233
0.9787379841057392	2.240893199201458
1.8675579901499675	-0.977277879876411
0.9500884175255894	-0.1513572082976979
-0.10321885179355784	0.41059850193837233
0.1440435711608 78	1.454273506962975
0.7610377251469934	0.12167501649282841
0.44386323274542566	0.33367432737426683
1.4940790731576061	-0.20515826376580087
0.31306770165090136	-0.8540957393017248
-2.5529898158340787	0.6536185954403606
0.8644361988595057	-0.7421650204064419
2.2697546239876076	-1.4543656745987648
0.04575851730144607	-0.1871838500258336
1.5327792143584575	1.469358769900285
0.1549474256969163	0.37816251960217356
-0.8877857476301128	-1.980796468223927
-0.3479121493261526	0.15634896910398005
1.2302906807277207	1.2023798487844113
-0.3873268174079523	-0.30230275057533557

The output confirms that the axis elements are correctly suppressed, leaving only the generated table. Note the use of `loc='center'` within the `ax.table()` call, which centrally positions the table within the defined axes area. For data visualization projects, especially those involving statistical summaries or complex model outputs, using this DataFrame-based method provides the most

robust and scalable way to integrate tabular data into Matplotlib plots.

Method 2: Creating Tables from Custom Data Structures

While integrating with pandas is standard practice in data science, situations often arise where the data to be displayed is not stored in a DataFrame. This might involve small, fixed datasets, configuration parameters, or manually entered examples, necessitating the use of custom Python list structures. In Method 2, we bypass the need for pandas entirely and pass a simple list of lists directly to the `cellText` parameter of the .table() method. This approach offers simplicity and reduced dependency for localized tasks.

The key difference here lies in data preparation. Instead of relying on DataFrame attributes, the data is explicitly defined as a nested list, where each inner list represents a row in the table. For instance, in our example, `table_data` is constructed to hold player names and their corresponding scores. When using custom values in this manner, it is important for the user to manually ensure data consistency; that is, every inner list (row) should have the same number of elements (columns), otherwise the rendering process may produce unexpected results or errors. This requirement reinforces the need for meticulous data handling when using raw list structures.

Since we are using custom values, we must decide whether to include column headers (`colLabels`). If headers are omitted, as in the example below, Matplotlib simply renders the cell content without a designated header row. The benefit of this method is its low overhead; it requires minimal imports (only Matplotlib components, potentially along with NumPy if statistical processing is needed) and executes swiftly for small datasets. This makes it ideal for quick visualizations or for educational purposes where the data structure is intentionally kept simple.

Example 2: Create Table from Custom Values

This example illustrates the direct implementation of Method 2, focusing on defining the data structure and applying basic aesthetic modifications to enhance presentation.

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#define figure and axes
fig, ax = plt.subplots()

#create values for table
table_data=,

,

,

,

]

```
#create table
table = ax.table(cellText=table_data, loc='center')

#modify table
table.set_fontsize(14)
table.scale(1,4)
ax.axis('off')

#display table
plt.show()
```

Observe the modification stage in the code. After creating the table object (assigned to the variable **table**), we immediately apply customizations. Specifically, `table.set_fontsize(14)` increases the legibility of the text within the cells, while `table.scale(1, 4)` significantly alters the dimensions of the table. The scaling function takes two arguments: the factor by which to scale the width (X-axis) and the factor by which to scale the length (Y-axis), allowing for precise control over the table's appearance relative to the figure.

Player 1	30
Player 2	20
Player 3	33
Player 4	25
Player 5	12

The resulting visualization is a vertically expanded table, centered on the axes area, demonstrating the successful application of both custom data input and post-creation scaling. This method is highly flexible when the layout and aesthetic requirements deviate significantly from Matplotlib's default table styling.

Customizing Table Aesthetics: Scaling and Sizing

One of the most powerful features for controlling the visual impact of a Matplotlib table is the `.scale(width, length)` method, which is applied directly to the table object generated by `ax.table()`. This function is essential for adapting the physical size of the table cells relative to the overall figure size and ensuring proper readability, especially when dealing with plots that have tight margins or high data density. The first parameter controls the horizontal scaling factor, and the second controls the vertical scaling factor.

For instance, in the previous example, `table.scale(1, 4)` maintained the default cell width (scaling factor of 1) but quadrupled the cell height (scaling factor of 4). This elongation is often desirable when the cell content is short but requires vertical separation for clarity, or when the table needs to occupy a large portion of the vertical axis in the subplot. It is critical to note that scaling does not change the font size; text modifications must be handled separately using methods like `table.set_fontsize()`, as shown previously.

If the requirement is to make the table significantly longer, perhaps to fill a figure that is much taller than it is wide, the vertical scaling factor can be dramatically increased. Consider the impact of changing the length factor from 4 to 10.

table.scale(1,10)

This modification stretches the table vertically, dramatically increasing the space between rows, which can be useful for emphasizing separation or when integrating the table into a specific layout where vertical space is abundant.

Player 1	30
Player 2	20
Player 3	33
Player 4	25
Player 5	12

Conclusion and Further Customization Tips

Matplotlib provides two highly flexible methods for embedding tabular data: extracting content directly from a pandas DataFrame for robust data management, or utilizing custom Python lists for simpler, manual datasets. Both approaches leverage the core `ax.table()` function, allowing for powerful customization of cell content, headers, and positioning.

Beyond simple sizing, users can delve into advanced styling by manipulating individual cell properties. The `table.get_cells()` method returns a list of all cell objects, which can then be iterated over to modify attributes such as background color (`cell.set_facecolor()`), border width (`cell.set_linewidth()`), or even adding conditional formatting based on data values. For

instance, cells containing negative values could be highlighted red, enhancing data interpretation. Effective use of these styling options transforms a basic table into a visually compelling and informative component of the overall Matplotlib figure.

Mastery of these table generation techniques ensures that data summaries and parameters are always presented with professional clarity, directly alongside their graphical counterparts. For users looking to further enhance their Matplotlib proficiency, we recommend exploring related tutorials on text management and layout control, which complement the use of embedded tables.

How to Add Text to Matplotlib Plots
How to Set the Aspect Ratio in Matplotlib
How to Change Legend Font Size in Matplotlib