Learning Matplotlib: A Guide to Creating Subplots with fig.add_subplot

Authored by

Mohammed looti

November 1, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning Matplotlib: A Guide to Creating Subplots with fig.add_subplot*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=7945

The ability to display multiple plots simultaneously within a single visualization space is fundamental to data analysis. In the Matplotlib library, this is achieved through the concept of subplots. While there are several ways to manage these graphical components, the `fig.add_subplot()` method offers explicit control over the placement of each axes object within a predefined grid system. This approach is particularly useful when customizing complex layouts or integrating new axes into an existing Figure object. We begin by examining the core syntax required to initialize these plotting areas using the highly popular pyplot module.

The fundamental syntax for generating individual subplots within a Matplotlib Figure is demonstrated below. This method requires us to first define the parent Figure, and then sequentially call `add_subplot()`, specifying the exact position of the new axes object using a three-digit integer code. This method is essential for users who prioritize granular control over their visualization structure.

import matplotlib.pyplot as plt

#define figure
fig = plt.figure()

#add first subplot in layout that has 3 rows and 2 columns
fig.add_subplot(321)

#add fifth subplot in layout that has 3 rows and 2 columns
fig.add_subplot(325)

...

This syntax is powerful because it allows developers to precisely map where each plot will reside within the overall layout. The following detailed examples illustrate how to leverage this function effectively, ranging from simple, uniform grids to more complex, asymmetrical arrangements.

Decoding the fig.add_subplot Arguments

The key to mastering `fig.add_subplot()` lies in understanding the required positional argument, typically represented as a three-digit integer (e.g., 321). This argument is not arbitrary; it compactly encodes the geometry of the subplot grid and the specific location the new axes object should occupy. When interpreting this structure as **RCN**, where R, C, and N are integers, Matplotlib processes these digits sequentially to construct the visualization space.

Specifically, the first digit (R) denotes the total number of rows in the grid, the second digit (C) specifies the total number of columns, and the third digit (N) indicates the index position of the

subplot, counting from 1 in the top-left corner and moving row-wise. For instance, the argument `234` defines a grid structure of 2 rows and 3 columns (totaling 6 potential plot positions), and places the current subplot in the 4th position. This indexing system is crucial for accurately defining the plot location and is 1-based, making it distinct from the 0-based indexing common in many other Python programming contexts.

While the three-digit integer format is common for brevity, it is important to note that the function also accepts three separate integer arguments: `fig.add_subplot(rows, columns, index)`. Both notations achieve the exact same result, but the single integer format (RCN) is often preferred for its conciseness in scripting. Using the object-oriented approach provided by `add_subplot()` returns an Axes object, allowing immediate method chaining, such as setting the title or axis labels, which simplifies the subsequent customization steps.

Example 1: Creating a Uniform Subplot Grid

Creating a uniform layout, where all subplots share the same dimensions and fit neatly into an organized grid, is the most common use case for `fig.add_subplot()`. This example demonstrates how to generate a total of six individual plots arranged in a 3x2 matrix (three rows and two columns). By iterating through the positions from 1 to 6 within the 32x context, we ensure every available slot in the defined grid is populated with its own dedicated plotting area, resulting in a perfectly balanced visualization.

The code snippet below defines the Figure object and then consecutively calls `add_subplot()` for each position. Note the use of method chaining: immediately following the creation of the axes object, `.set_title()` is called to label each subplot clearly with its corresponding positional code (e.g., 321, 322, etc.). This practice confirms that the user correctly understands the row-first indexing pattern inherent in Matplotlib's subplot numbering system, which is essential for accurate placement.

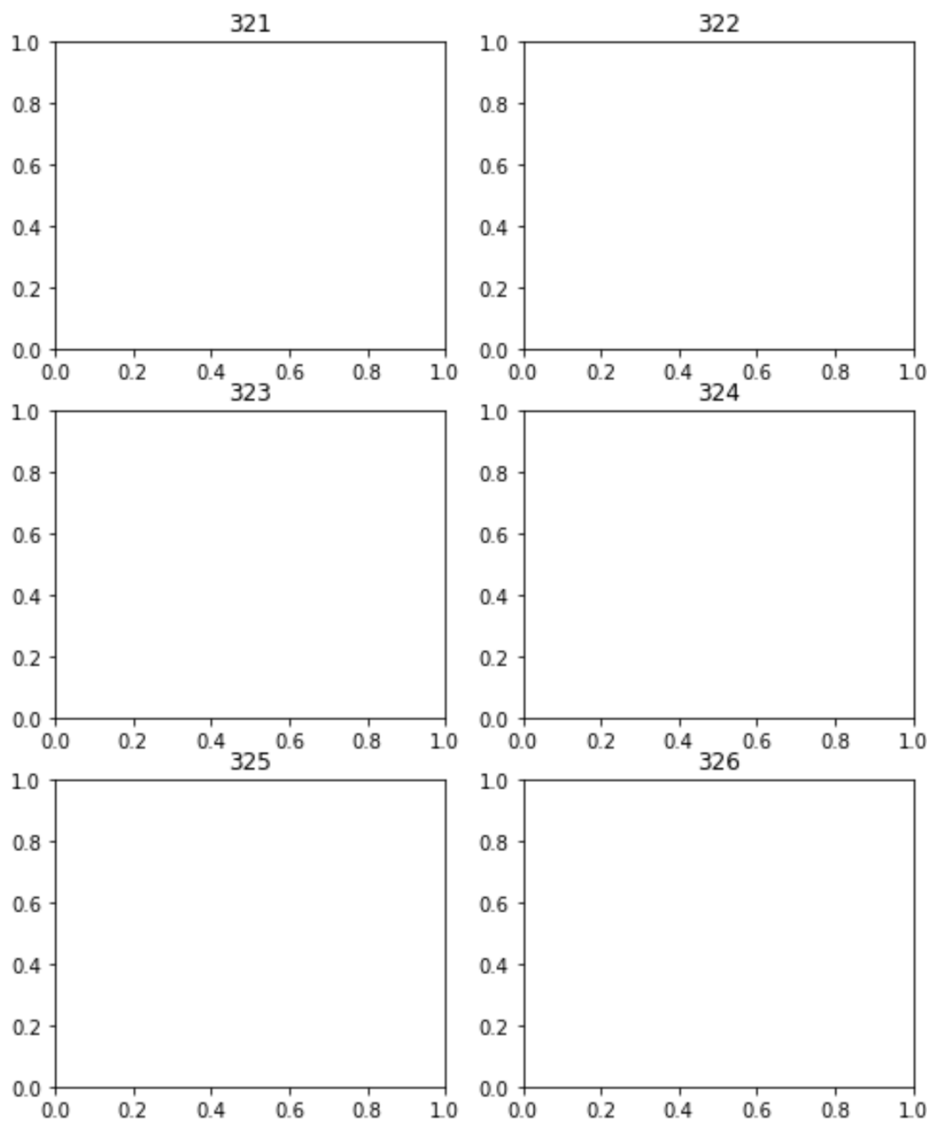
```
import matplotlib.pyplot as plt

#define figure
fig = plt.figure()

#add subplots
fig.add_subplot(321).set_title('321')
fig.add_subplot(322).set_title('322')
fig.add_subplot(323).set_title('323')
fig.add_subplot(324).set_title('324')
fig.add_subplot(325).set_title('325')
fig.add_subplot(326).set_title('326')
```

```
#display plots
plt.show()
```

Upon execution, the resulting visualization confirms the 3x2 arrangement. This straightforward structure is ideal for comparative analysis, where multiple datasets or different views of the same data need to be presented side-by-side or stacked vertically. Achieving this clean, balanced look is a fundamental step in using Matplotlib effectively for scientific visualization.



As clearly depicted in the output image, all six subplots are displayed consistently within the specified layout of 3 rows and 2 columns. This uniformity ensures visual balance and ease of interpretation when analyzing the generated plots, making it a highly readable display for complex data series.

Example 2: Implementing Asymmetrical Subplot Arrangements

The true flexibility of `fig.add_subplot()` shines when creating complex, asymmetrical layouts where certain plots need to span multiple rows or columns--a crucial technique for highlighting a primary visualization alongside several smaller, supplementary charts. This method involves defining the subplot based on one grid structure, and then defining subsequent subplots based on entirely different grid parameters, effectively making the later subplot occupy a larger space by referencing a grid structure that grants it greater dimensions.

In this advanced example, we aim to construct a visualization composed of four plots, where three are small and stacked vertically, and the fourth is large, occupying the entire right side of the canvas. This is achieved by planning the layout based on two distinct grid concepts applied sequentially to the same Figure object:

The first three plots are positioned within a **3 row by 2 column grid** (32x).
The fourth plot is defined using a **1 row by 2 column grid** (12x), allowing it to naturally span across the entire width of the right column, effectively merging the space originally intended for three vertical plots.

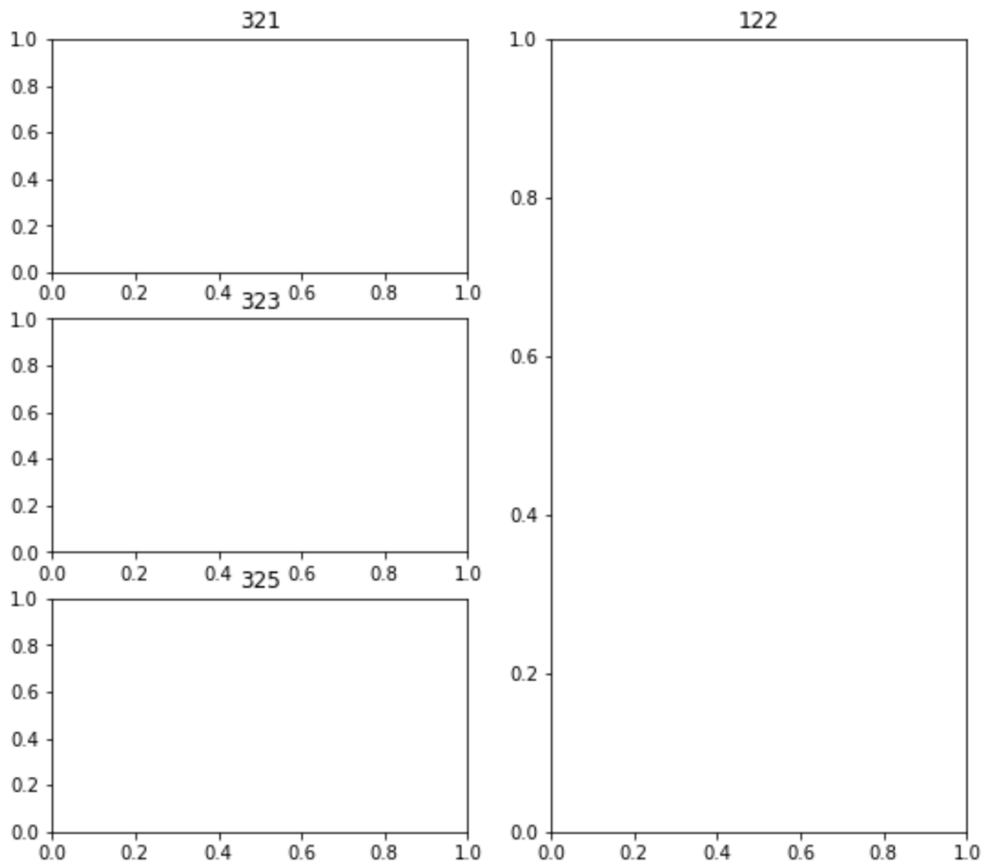
This grid manipulation allows for highly customized visualizations. The key is understanding that `add_subplot(RCN)` defines the **potential** grid size for the entire figure at that moment for that specific subplot, and if the dimensions conflict with existing plots, Matplotlib handles the drawing boundaries gracefully. Below is the code implementation detailing this procedure:

```
import matplotlib.pyplot as plt

#define figure
fig = plt.figure()

#add subplots
fig.add_subplot(321).set_title('321')
fig.add_subplot(323).set_title('323')
fig.add_subplot(325).set_title('325')
fig.add_subplot(122).set_title('122')

#display plots
plt.show()
```



The final output clearly demonstrates the power of mixing different grid definitions. The three plots on the left occupy the odd-numbered positions (1, 3, 5) of the initial 3x2 grid. The final plot, defined as 122, occupies the second position of a theoretical 1x2 grid, forcing it to consume the remaining, vertically stacked space on the right side of the Figure. This technique is often preferred over using complex layout managers like `GridSpec` when the layout asymmetry is relatively simple and involves clear mergers.

Comparing fig.add_subplot() and plt.subplots()

When working with Matplotlib, users frequently encounter two primary methods for creating plotting areas: `fig.add_subplot()` and `plt.subplots()`. While both achieve the goal of generating axes objects, they cater to different programming styles and use cases. Understanding the distinction is crucial for efficient coding, especially in large-scale data visualization projects. The `fig.add_subplot()` method is intrinsically tied to the object-oriented API, requiring an existing Figure instance, and is ideal for incremental layout building or complex, non-uniform subplots, as shown in Example 2.

Conversely, `plt.subplots()` is a convenience function that utilizes the functional approach, returning both the Figure object and a NumPy array of Axes objects simultaneously. It is vastly

superior for creating simple, uniform N x M grids where all subplots are the same size. For instance, creating a 2x2 grid is a single line operation: `fig, axes = plt.subplots(2, 2)`. This simplicity makes `plt.subplots()` the preferred choice for quick visualizations and standard data reporting, significantly reducing the boilerplate code required.

In summary, the choice depends heavily on the complexity required. We can break down the decision criteria into the following key points:

Use `fig.add_subplot(RCN)` when you need **fine-grained control** over the position of individual plots, need to mix different grid definitions (asymmetrical layouts), or are adding plots dynamically to an already existing Figure object where the Figure was not initially created using `plt.subplots()`.
Use `plt.subplots(R, C)` when you require a **simple, standard grid layout** where all axes objects are uniform in size and placement. This method manages the creation of both the Figure and the Subplots automatically.

Experienced developers often mix these techniques, initiating the Figure and main Axes using `plt.subplots()` for the core visualization, and then using `fig.add_subplot()` or `fig.add_axes()` to insert specialized inset plots or complex annotations later in the process, maximizing both efficiency and flexibility.

Summary and Further Learning

The `fig.add_subplot()` method provides robust, explicit control over the creation and placement of axes objects within a Matplotlib Figure. By utilizing the RCN integer argument, users can easily define complex grid structures, allowing for both uniform and highly customized, asymmetrical plot arrangements essential for effective data storytelling. Mastery of this function is a cornerstone of advanced Matplotlib usage, complementing the faster functional methods like `plt.subplots()`.

We highly recommend practicing both the uniform and non-uniform examples provided to solidify the understanding of the RCN indexing system. Paying close attention to how the grid definitions interact when plots overlap or span multiple theoretical cells is key to successfully designing complex statistical graphics and ensuring maximum visual impact.

Additional Resources

For those looking to expand their knowledge of data visualization techniques in Matplotlib, the following tutorials offer guidance on other common operations and advanced features: