

# How to Easily Create Subplots in Matplotlib with fig.add\_subplot()

Authored by  
**stats writer**

December 2, 2025

## RECOMMENDED CITATION

stats writer (2025). *How to Easily Create Subplots in Matplotlib with fig.add\_subplot()*. PSYCHOLOGICAL SCALES. Retrieved from <https://scales.arabpsychology.com/?p=103770>

The **Matplotlib** library is the cornerstone of **Python** data visualization, offering unparalleled flexibility for creating static, interactive, and animated plots. Central to its architecture is the concept of managing multiple visualization panels within a single display area. This crucial task is handled efficiently by the `fig.add_subplot()` function, a specialized method belonging to the **Figure object**.

Utilizing `fig.add_subplot()` allows developers and data scientists to systematically arrange several independent graphical representations, known as **subplots**, into a unified composition. This function is essential when intricate comparisons or detailed views of related datasets are required, providing a structured approach to generating complex visual layouts that are both informative and aesthetically organized. Understanding its mechanism is foundational for mastering multi-panel plotting in Matplotlib.

The function requires mandatory positional arguments, specifically detailing the geometry of the desired grid layout--the total number of rows, the total number of columns, and the specific index position where the new subplot should be placed. Importantly, `fig.add_subplot()` does not return a Figure object, but rather an **Axes object**. This returned Axes object represents the actual plotting area, which is then used by subsequent Matplotlib commands to draw data, apply customizations, set titles, and define axis labels, thus bridging the gap between layout definition and graphical execution.

## Understanding the Matplotlib Hierarchy: Figure and Axes

To effectively use `fig.add_subplot()`, one must first grasp the core hierarchy established by **Matplotlib**: the relationship between the Figure and the Axes. The **Figure** serves as the top-level container, analogous to a canvas or an entire window in which plotting occurs. It holds everything related to the plot, including the background, titles, and, most critically, one or more Axes objects.

The **Axes object**, despite its plural-sounding name, represents a single plot area--a single graphical panel. It is the core of the visualization where data points are drawn, containing the x-axis, y-axis, ticks, labels, and the plotted data itself. When we call `fig.add_subplot()`, we are explicitly instructing the Figure (`fig`) to create and insert a new Axes object into the layout at a specified location.

This separation of concerns is fundamental to Matplotlib's object-oriented interface. The Figure manages the overall presentation and saving of the image, while the Axes manages the specifics of the data visualization within its boundaries. Consequently, any operations related to drawing data, such as calling `plot()`, `scatter()`, or `hist()`, are methods applied directly to the returned Axes object, ensuring modular and clean code structure when handling complex multi-panel visualizations.

## Deconstructing the `fig.add_subplot` Parameters

The primary power of the `fig.add_subplot()` function lies in its concise parameter structure, often used in a three-digit integer format (RCS). While the function supports various calling conventions, the most common and compact method involves passing a single three-digit integer specifying the grid geometry and the subplot position. Let us thoroughly examine these three crucial components: rows, columns, and subplot index.

The first digit, R, defines the total number of **rows** in the grid layout. This dictates the vertical dimension of the visualization space. The second digit, C, defines the total number of **columns**, setting the horizontal partitioning of the Figure. Together, R and C establish a grid framework (R x C) that determines the maximum number of potential **subplots** that can be created. For instance, passing `321` indicates a 3-row, 2-column grid, allowing for up to six possible subplot locations.

The third digit, S, specifies the **index position** (or plot number) within that grid where the new Axes object should be placed. Indexing starts at 1 (not 0) and proceeds row-wise, from left to right. In a 3x2 grid, position 1 is top-left, 2 is top-right, 3 is middle-left, and so on, up to 6 (bottom-right). This indexing system is vital for accurately mapping where each individual plot will render on the overall **Figure object**, enabling precise control over the visual composition.

## Basic Syntax and Initialization

Before creating any subplots, we must first import the necessary modules and instantiate the top-level Figure container. It is conventional practice in **Python** data analysis to import the `pyplot` interface of Matplotlib under the alias `plt`. Once imported, the Figure is initialized, serving as the canvas upon which all subsequent Axes objects will be drawn using `fig.add_subplot()`.

The following syntax demonstrates the minimal setup required to define a figure and begin adding subplots. Notice how the three-digit parameter (e.g., `321` or `325`) defines a consistent grid structure (3 rows, 2 columns) and then selects the specific cell index within that grid for placement. Even if we only place two plots, the overall structure of 3 rows and 2 columns is defined by the call, reserving space for the other four potential plots.

Understanding this basic invocation is key to scaling up to more complex layouts. We are defining the grid once and then specifying the position index repeatedly for each new plot we wish to introduce. This object-oriented approach ensures that the layout remains coherent, even when plots are added non-sequentially or sparsely across the grid.

You can use the following basic syntax to create subplots in **Matplotlib**:

```
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)

...
```

The following examples show how to use this syntax in practice, demonstrating layouts ranging from perfectly symmetrical grids to complex, asymmetric compositions.

### Example 1: Creating Subplots with an Even Layout

The most straightforward application of `fig.add_subplot()` is generating a complete, uniform grid where every available cell is occupied by an **Axes object**. This is typically used when presenting a series of related comparisons, such as time-series data for multiple different variables or different statistical transformations of the same underlying data. A 3x2 grid is a common, balanced choice for displaying six distinct plots.

In the code below, we define the grid geometry as 3 rows and 2 columns (32x), and then systematically iterate through the index positions 1 through 6. Each call returns a unique Axes object, upon which we immediately apply `.set_title()` to label the position clearly. This method is concise and guarantees that all six plotting areas are correctly sized and positioned relative to each other within the main Figure container.

When running this code, observe that the resulting image perfectly aligns the six subplots, maximizing the visual space. The use of `plt.show()` is crucial, as it instructs Matplotlib to render the defined Figure and all its contained Axes objects to the screen. This ensures the visualization is displayed accurately according to the geometric constraints we defined using the three-digit notation.

The following code shows how to create six subplots in a layout that has 3 rows and 2 columns:

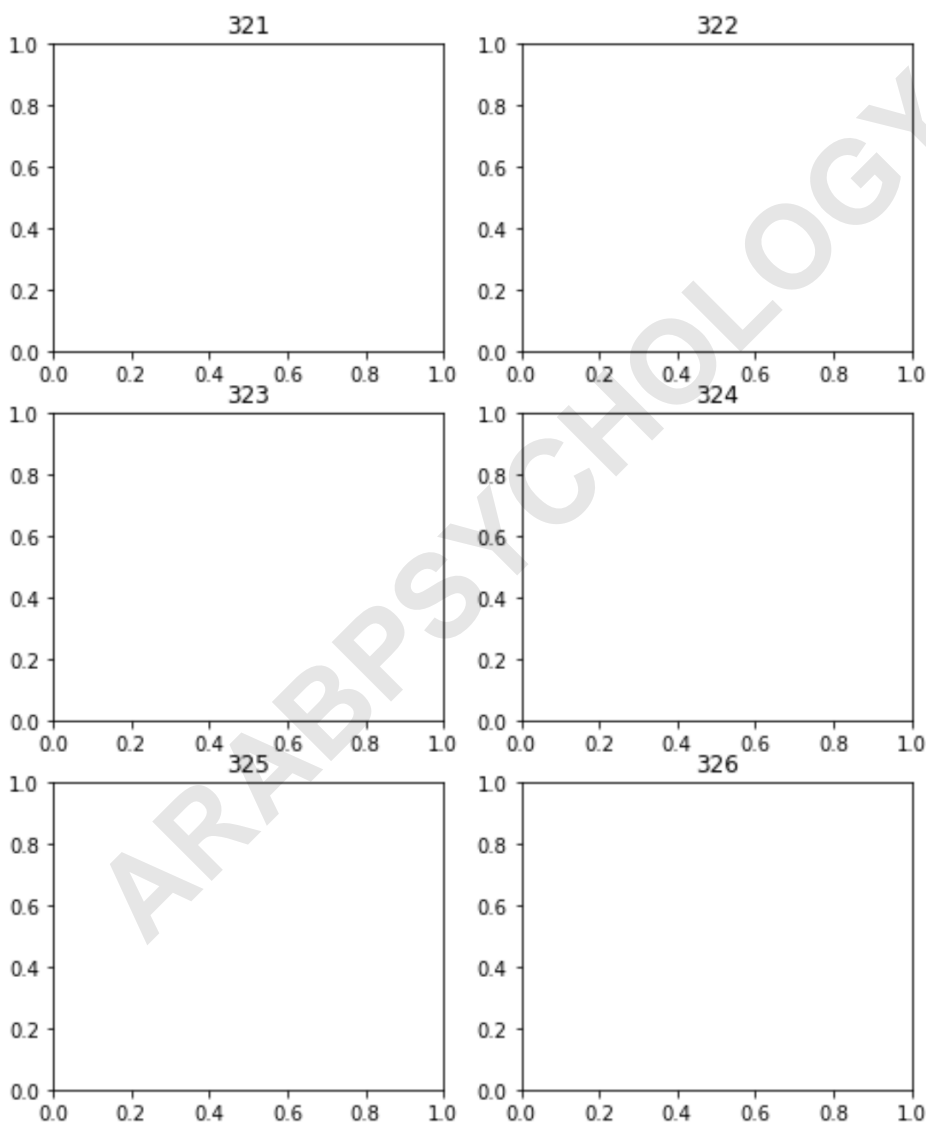
```
import matplotlib.pyplot as plt
```

```
#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()
```



Notice that the result is six distinct **subplots** displayed in a layout that has 3 rows and 2 columns, occupying every slot available in the defined grid geometry.

## Example 2: Creating Subplots with an Uneven Layout

One of the most powerful features of `fig.add_subplot()` is its ability to create complex, non-uniform layouts by spanning one Axes object across multiple cells of the underlying grid. This is achieved by changing the R, C, or S parameters, or by utilizing the more expressive signature that defines the row and column span explicitly. However, the three-digit integer method can still create uneven visual effects by defining different base grids.

In this example, we aim for an asymmetrical layout where three plots occupy a 3x2 grid (the left side), and a final, larger plot occupies the entire right side. To achieve this, we first define the three smaller plots using the 3x2 geometry (321, 323, 325). Crucially, for the final, larger plot, we define a different, smaller grid geometry of 1 row and 2 columns, and place the plot in the second position (122).

By defining the final plot using `122`, we essentially create a temporary 1x2 grid spanning the whole figure width, placing the plot in the right half. Because Matplotlib is intelligent about managing overlap, it places the `122` plot cleanly alongside the already existing 3x2 structure, effectively demonstrating how we can mix and match grid definitions to achieve visually distinct plotting areas within the same **Figure object**.

The following code shows how to create four subplots in the following manner, demonstrating an advanced, uneven configuration:

Three of the plots are created in a grid with 3 rows and 2 columns, occupying the left column slots (1, 3, and 5).

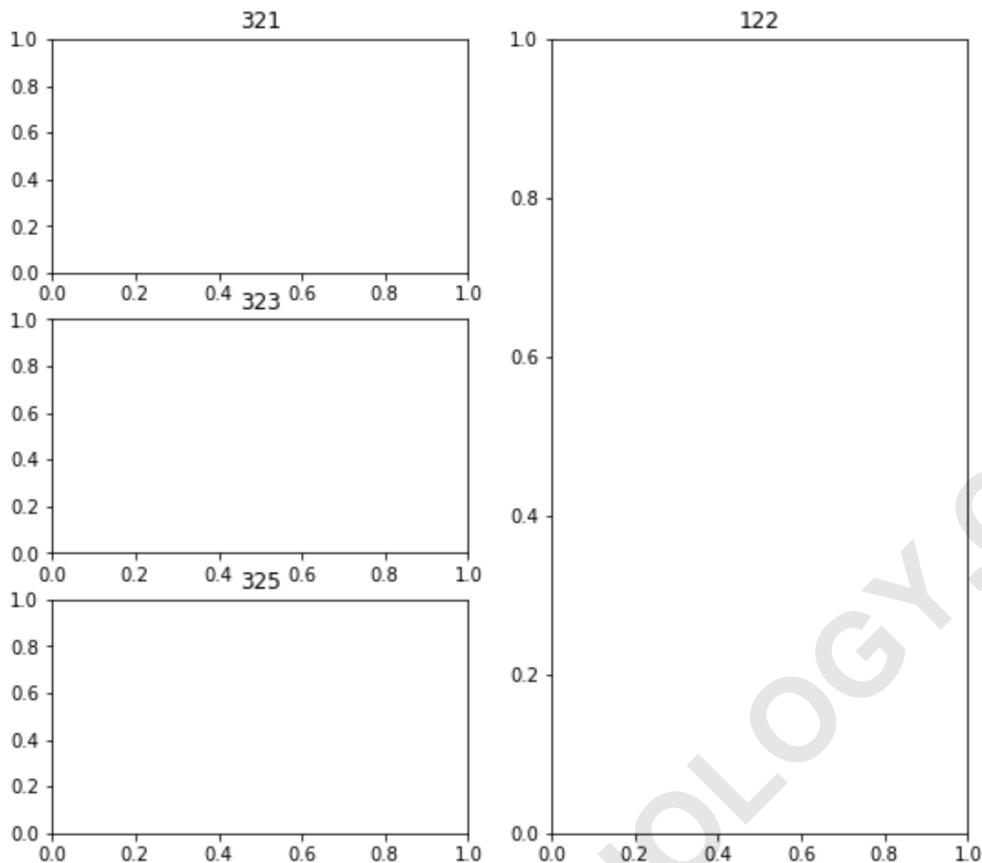
The fourth plot is created using a new grid definition of 1 row and 2 columns, occupying the right half of the figure.

```
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 end result clearly shows three smaller **subplots** displayed vertically in the 3x2 grid, while the final, larger subplot occupies the entire right side, defined by the 1x2 grid structure. This confirms the flexibility of defining multiple distinct grid geometries within a single Matplotlib Figure.

## Advanced Considerations: The Subplot Indexing System

Understanding the indexing system used by `fig.add_subplot(RCS)` is paramount, especially when creating sparse or complex grid arrangements. Unlike zero-based indexing common in **Python** arrays, Matplotlib's subplot index (S) is one-based. This system is designed to be intuitive for visual layout, where position 1 is always the top-left corner of the defined grid.

Indexing proceeds horizontally across the rows before moving down to the next row. For a 4x3 grid (4 rows, 3 columns), the index sequence would run 1, 2, 3 (Row 1); 4, 5, 6 (Row 2); 7, 8, 9 (Row 3); and 10, 11, 12 (Row 4). If you attempt to use an index number greater than the total number of cells ( $R * C$ ), Matplotlib will typically raise an error, highlighting the importance of matching the index (S) to the total grid size defined by R and C.

It is important to note that the index parameter (S) can also be used to specify spanning behavior, although this is more elegantly handled using the GridSpec method for extremely complex layouts.

When using the three-digit integer approach, careful planning of the R, C, and S values prevents plots from overlapping unexpectedly or creating unintended whitespace, ensuring optimal utilization of the canvas space provided by the **Figure object**.

## Alternative Methods: When to Use `subplots()` vs. `add_subplot()`

While `fig.add_subplot()` offers granular, object-oriented control for adding individual subplots sequentially, Matplotlib also provides the convenience function `plt.subplots()`. Choosing between these two methods depends heavily on the complexity and regularity of the desired visualization layout.

The `plt.subplots(R, C)` function is generally preferred when creating a perfectly uniform, standard grid of R rows and C columns. It automatically creates both the Figure and all necessary Axes objects simultaneously, returning them as a tuple: `(fig, ax)`, where `ax` is typically an array of **Axes object** references. This method is concise and excellent for quick, standard visualizations but offers less flexibility in varying the size or placement of individual plots dynamically.

In contrast, `fig.add_subplot()` is necessary when dealing with uneven grids, mixing plot types, or integrating plots iteratively within an existing Figure structure. Because `fig.add_subplot()` is a method of an existing Figure, it provides the low-level control needed to define specific geometries (like the 321 and 122 combination shown in Example 2) or use advanced features like `GridSpec`, making it indispensable for highly customized or publication-quality figures.

## Conclusion and Best Practices for Multi-Panel Plotting

Mastering `fig.add_subplot()` is a critical step in becoming proficient with complex data visualization in **Matplotlib**. This function provides the necessary precision to manage the spatial arrangement of multiple Axes objects within a single Figure container. Its structure, utilizing the three-digit RCS notation (Rows, Columns, Position), offers a direct and highly legible way to define the grid geometry and plot placement.

To ensure maintainable and readable code when generating multi-panel figures, always adhere to the object-oriented approach: define the Figure first, then use `fig.add_subplot()` to create and retrieve the individual **Axes object** for each panel. Remember that all plotting commands, such as data plotting, setting limits, and adding legends, must be applied to the specific Axes object returned by the function call.

For layouts where plots span multiple cells or where the arrangement is highly non-uniform, consider utilizing Matplotlib's `GridSpec` alongside `fig.add_subplot()` for enhanced explicit control, especially in production environments where visual consistency is mandatory. By carefully managing the Figure and Axes hierarchy, developers can create sophisticated and insightful

visualizations that effectively communicate complex analytical results.

ARABPSYCHOLOGY.COM