Learn How to Draw Rectangles in Matplotlib with Examples

Authored by

Mohammed looti

November 6, 2025

Drawing geometric shapes is a fundamental task in Matplotlib, essential for tasks ranging from highlighting specific regions in charts to creating bounding boxes in computer vision applications. To effectively draw a rectangle, we utilize the powerful **patches** module within Matplotlib. This module provides primitive shapes that can be added directly to an Axes object.

Specifically, the primary function used for this purpose is matplotlib.patches.Rectangle. This function requires precise specification of location, dimension, and orientation to correctly render the desired shape on your visualization canvas. Understanding the core arguments of this function is the first step toward mastering geometric annotations in Python plotting.

Understanding the matplotlib.patches.Rectangle Function

The `matplotlib.patches.Rectangle` function is the backbone for creating rectangular elements in your plots. It is highly flexible, allowing control over position, size, and rotation. When initializing a new rectangle object, you must define its anchor point and dimensions. This object is then treated as an **Artist** in Matplotlib and must be explicitly added to an existing **Axes** object using methods like `ax.add_patch()`.

The basic syntax for defining a rectangle is straightforward but requires attention to the input parameters:

matplotlib.patches.Rectangle(xy, width, height, angle=0.0, **kwargs)

Here is a detailed breakdown of the required and optional arguments that define the rectangle's properties:

xy: This is a tuple representing the **(x, y) coordinates** for the anchor point of the rectangle. By default, this usually corresponds to the bottom-left corner of the rectangle, unless rotation is applied.

width: A numeric value specifying the horizontal length of the rectangle along the x-axis.

height: A numeric value specifying the vertical length of the rectangle along the y-axis.

angle: An optional float value specifying the rotation in degrees counter-clockwise about the **xy** anchor point. The default value of 0.0 results in an unrotated, axis-aligned rectangle.

Additionally, the function accepts numerous keyword arguments (`**kwargs`) for styling, which we will explore in a subsequent section. These styling parameters allow you to control the color, line thickness, fill, and other visual attributes, transforming a simple box into a sophisticated annotation tool for data visualization.

Practical Application: Drawing a Simple Rectangle on a Plot

The most common use case for the `Rectangle` patch is highlighting a specific data range or region of interest (ROI) on a standard 2D plot. This requires importing the necessary components from Matplotlib and defining an Axes object to which the patch will be attached. The following example demonstrates the fundamental workflow, creating a simple line plot and then overlaying a rectangle defined by its bottom-left corner (1, 1), a width of 2, and a height of 6.

To successfully execute this, we first initialize the figure and axes using `plt.subplots()`. We then define our context--a simple diagonal line--before using `ax.add_patch()` to integrate the newly created `Rectangle` instance. This method is crucial, as simply defining the patch object does not automatically render it; it must be explicitly added to the drawing canvas associated with the axes.

The coordinates provided (1, 1) serve as the absolute starting point within the plot's coordinate system. This approach ensures that the rectangle scales and moves correctly with the rest of the plot elements, making it an integral part of the visualization.

import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

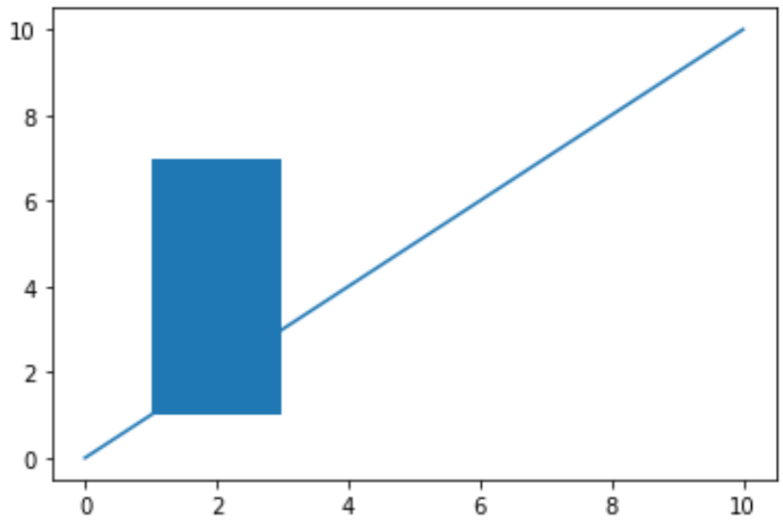
#define Matplotlib figure and axis
fig, ax = plt.subplots()

#create simple line plot
ax.plot(,)

#add rectangle to plot
ax.add_patch(Rectangle((1, 1), 2, 6))

#display plot
plt.show()

Once executed, the code generates a plot featuring a diagonal line from (0, 0) to (10, 10), with a standard, unfilled rectangle drawn starting at (1, 1). This visual confirmation confirms the successful placement and scaling of the patch relative to the data coordinates.



Enhancing Visuals: Customizing Rectangle Appearance and Style

While the default rectangle is functional, most visualizations require specific styling to match the aesthetic or to ensure the annotation stands out clearly. Matplotlib allows extensive customization through keyword arguments passed directly to the matplotlib.patches.Rectangle function. These arguments control aspects such as edge color, face color (fill), line width, and transparency.

In the following example, we demonstrate how to apply several key styling properties. We use `edgecolor` to define the color of the boundary line, `facecolor` to set the internal fill color, `fill=True` to ensure the shape is colored internally, and `lw` (linewidth) to control the thickness of the border. Choosing contrasting colors, such as blue fill and pink edges in this case, ensures high visibility against the background plot elements.

These styling parameters are critical for creating professional-grade plots. For instance, you might set `facecolor` to a semi-transparent value (using the `alpha` parameter, not shown here) if you need to highlight an area without completely obscuring the underlying data points. Experimenting with different combinations of color and line attributes allows you to tailor the rectangle's appearance precisely to your data visualization needs.

import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

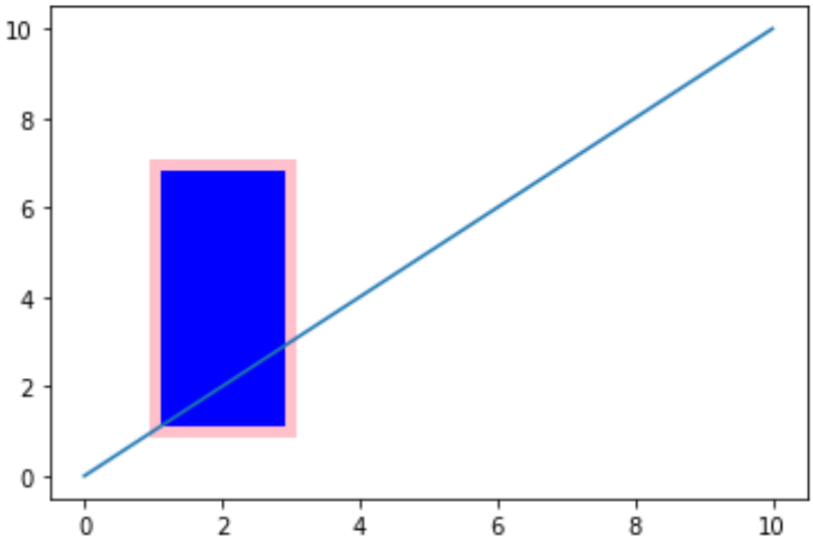
#define Matplotlib figure and axis
fig, ax = plt.subplots()

#create simple line plot
ax.plot(,)

```
#add rectangle to plot with custom styling
ax.add_patch(Rectangle((1, 1), 2, 6,
edgecolor = 'pink',
facecolor = 'blue',
fill=True,
lw=5))

#display plot
plt.show()
```

The result is a visually distinct rectangle, clearly demarcating the specified region. A complete list of all available styling properties, including options for hatching, dash patterns, and advanced color mapping, can be found in the official Matplotlib documentation for patches.



You can find a comprehensive list of all styling properties that can be applied to a rectangle patch here.

Advanced Usage: Annotating Images with Rectangles

Beyond standard data plots, rectangles are crucial for tasks involving image processing, such as defining a region of interest (ROI) on a displayed image. When drawing a rectangle on an image, the underlying **coordinate system** changes: the coordinates now represent pixel values, where (0, 0) is typically the top-left corner of the image, and the axes are scaled according to the image dimensions.

To demonstrate this, we first load an image (in this case, 'stinkbug.png') using a library like PIL

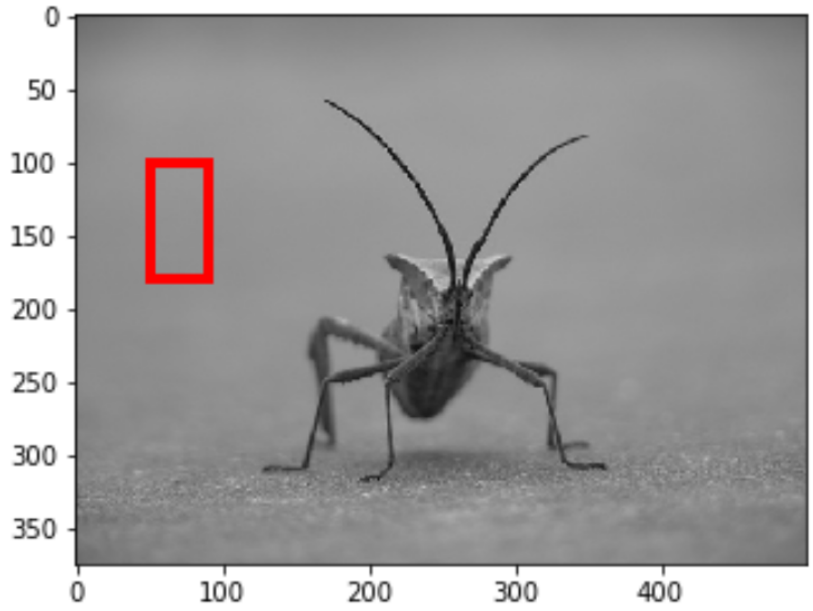
(Pillow) and display it using `plt.imshow()`. Unlike adding patches to a standard plot where we use `ax.add_patch()`, when working directly with `plt.imshow()`, we must retrieve the current Axes object using `plt.gca()` (Get Current Axes) before adding the patch.

In the code below, we define a rectangle starting at pixel coordinates (50, 100), with a width of 40 and a height of 80. Crucially, notice the styling used: `facecolor='none'` is set to prevent the rectangle from obscuring the image content, relying only on the `edgecolor` to outline the bounding box. This technique is standard practice when annotating images to preserve visibility of the underlying visual data.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from PIL import Image

#display the image (Note: Ensure 'stinkbug.png' is available locally)
plt.imshow(Image.open('stinkbug.png'))

#add rectangle to the current Axes
plt.gca().add_patch(Rectangle((50,100),40,80,
edgecolor='red',
facecolor='none',
lw=4))
```



Note: To replicate this example, you must first obtain the image file ('stinkbug.png') and save it in

the same directory as your Python script. This image is commonly used in Matplotlib tutorials for image processing examples.

Manipulating Orientation: Rotating Rectangles

One of the most powerful features of the matplotlib.patches.Rectangle function is the ability to easily rotate the shape using the optional `angle` parameter. This parameter accepts a float value representing the number of degrees by which the rectangle should be rotated counter-clockwise. The rotation occurs around the specified anchor point (**xy**), which usually serves as the bottom-left corner of the unrotated rectangle.

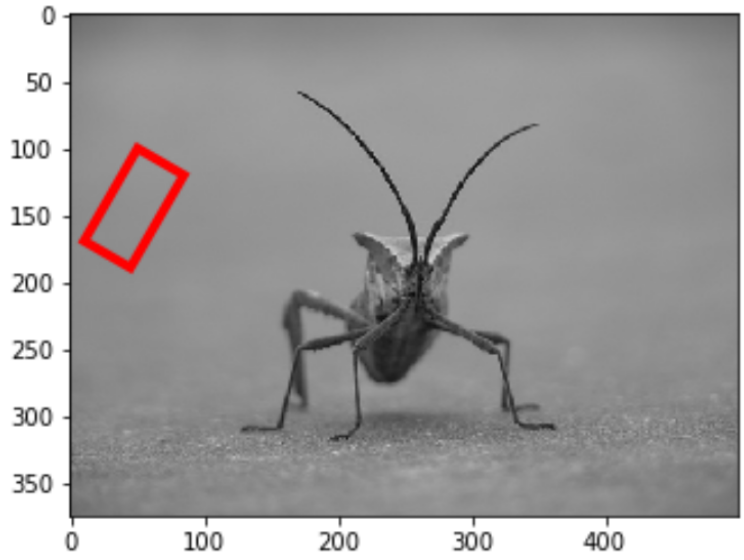
Understanding the reference point for rotation is crucial. If the anchor point is (50, 100), and you set an `angle` of 30, the entire rectangle pivots 30 degrees counter-clockwise around that exact pixel location. This means the appearance and effective boundaries of the rectangle change dramatically, even though its width and height remain mathematically the same.

The following code snippet takes the previous image annotation example and introduces a 30-degree rotation. This demonstrates how rotating the patch can be used to align the bounding box with objects that are not axis-aligned in the image, often improving the accuracy of visual annotations.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from PIL import Image

#display the image
plt.imshow(Image.open('stinkbug.png'))

#add rotated rectangle
plt.gca().add_patch(Rectangle((50,100),40,80,
angle=30,
edgecolor='red',
facecolor='none',
lw=4))
```



By changing the `angle` parameter, you can achieve any desired orientation, providing flexibility for complex visualizations and precise object localization tasks in computer graphics.

Summary of Key Parameters and Best Practices

Working with Matplotlib patches, particularly the `Rectangle` class, offers a robust method for adding structured annotations to your visualizations. To ensure clean, effective code, always adhere to a few best practices. First, remember that patches are **Artist** objects and must be explicitly added to an **Axes** object using `ax.add_patch()` or `plt.gca().add_patch()`. Simply instantiating the patch object is not enough to render it.

Second, always consider the **coordinate system** context. For standard scatter or line plots, coordinates relate to your data values. For image plots, coordinates typically correspond to pixel locations. Misinterpreting the coordinate system is the most common source of error when placing patches.

Finally, utilize the extensive styling options (like `edgecolor`, `facecolor`, `alpha`, and `lw`) to make your annotations informative and visually appealing. Using named colors or hex codes allows for high precision in design. Mastering the placement, sizing, and styling of the Rectangle patch is fundamental for creating sophisticated and publication-ready graphics in Matplotlib.

Related Reading: For those interested in drawing other geometric primitives, explore how to utilize Matplotlib's patch collection for circles and ellipses.

Related: How to Plot Circles in Matplotlib (With Examples)