

Matplotlib Library

Aayush Bajaj

July 10, 2026

Contents

1 the architecture: figure, axes, artist	1
1.1 the two apis	2
2 anatomy of a figure	3
3 subplots and layout	3
4 the plot families	3
5 styling	4
6 3d and animation, briefly	4
7 backends	4
8 saving figures	5
9 see also	5

matplotlib has a reputation for being clunky, and the reputation is earned exactly by people fighting the *wrong* api. there are two: a stateful **pyplot** layer that mimics matlab (“draw on whatever was touched last”), and an object-oriented core in which every visible thing is an object you hold a reference to. the second is the real library; the first is sugar for one-liners. write **fig, ax = plt.subplots()** and stay in object land, and most of the clunk evaporates. all code on this page executed for real; figures were written to **/tmp** and are described rather than embedded, with true file sizes.

official docs:
<https://matplotlib.org/stable/>
— the ‘anatomy of a figure’
poster there is the single best
page of documentation in
scientific python

1 the architecture: figure, axes, artist

three nouns carry the whole library:

- **figure** — the top-level canvas: owns everything, knows its size in inches and its dpi. one window / one file = one figure.
- **axes** — a plotting region with its own coordinate system: two (or three) **axis** objects, tick machinery, labels, title, and the data artists drawn inside it. a figure holds any number of axes — “subplot” is just an axes on a grid.
- **artist** — the universal base class. lines, patches, text, images, ticks, legends, the axes themselves, the figure itself: *everything rendered is an artist*, arranged in one containment tree, and every property (colour, alpha, zorder, transform)

*axes is the worst name in the
library: one axes, two axes, and
an axes contains axis objects.
everyone stumbles once*

is settable on any of them.

rendering is a tree walk: the figure asks its children to draw, they ask theirs. customising a plot = locating the right artist in the tree and setting its properties. that single sentence replaces about half of stack overflow.

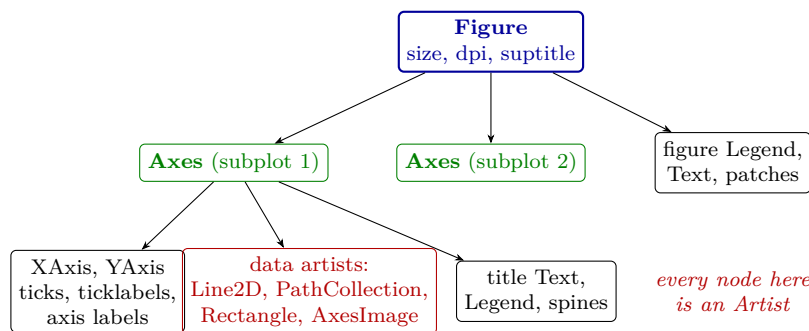


Figure 1: the containment hierarchy: a Figure owns Axes; each Axes owns its axis machinery and the data artists; every node in the tree is an Artist.

1.1 the two apis

`plt.plot(...)` secretly means “fetch the *current* axes (creating a figure if none), call `.plot` on it” — global mutable state, fine in a repl, hostile in functions. the OO form makes the state explicit:

```

1  import matplotlib
2  matplotlib.use("Agg")                # non-interactive backend
3  import matplotlib.pyplot as plt
4  import numpy as np
5
6  x = np.linspace(0, 2 * np.pi, 200)
7  fig, ax = plt.subplots(figsize=(6, 4)) # Figure + one Axes
8  line, = ax.plot(x, np.sin(x), label=r"$\sin x$")
9  ax.set(xlabel="x", ylabel="y", title="one axes, one artist per thing")
10 ax.legend()
11
12 # walk the artist hierarchy: everything on screen is an Artist
13 print("figure children :", [type(c).__name__ for c in fig.get_children()])
14 print("axes has         :", len(ax.lines), "Line2D,",
15       len(ax.texts), "Text,", len(ax.get_xticklabels()), "xticklabels")
16 print("the line artist :", line)
17 print("its parent axes :", line.axes is ax, "| its figure:", line.figure is
18       ↪ fig)
19
20 fig.savefig("/tmp/mpl-sine.png", dpi=110)
21 import os
22 print("saved /tmp/mpl-sine.png:", os.path.getsize("/tmp/mpl-sine.png"),
23       ↪ "bytes")

```

```

figure children : ['Rectangle', 'Axes']
axes has         : 1 Line2D, 0 Text, 9 xticklabels
the line artist : Line2D($\sin x$)
its parent axes : True | its figure: True
saved /tmp/mpl-sine.png: 28122 bytes

```

the resulting png (28 kB): a single blue sine period on a white ground, legend box top-right — nothing exotic, but every piece of it is now a named object we could restyle after the fact. note the figure’s children: a background **Rectangle** and the **Axes** — the tree is real, not a metaphor.

2 anatomy of a figure

the checklist to run when a plot looks wrong, outermost in:

- **figure:** `figsize` (inches!) and `dpi` — pixels are their product; a “too small fonts” problem is usually a “figure too large” problem.
- **axes position:** the rectangle the axes occupies inside the figure (layout engines below).
- **spines:** the four box edges; hide or move them (`ax.spines["top"].set_visible(False)`).
- **axis objects:** locators decide *where* ticks go, formatters decide *what they say* — the right fix for ugly ticks is a `Locator` or `Formatter`, never manual strings.
- **data artists:** what you drew.
- **text:** title, axis labels, annotations — all `Text` artists, all `mathtext`-capable (`$...$`).

3 subplots and layout

`plt.subplots(rows, cols)` returns the figure and an array of axes — the only subplot api worth remembering. the historical pain of overlapping labels is solved by layout engines: `layout="constrained"` (preferred, a real constraint solver) or `layout="tight"` (older heuristic). both negotiate space so titles, ticklabels and colourbars stop colliding.

4 the plot families

choosing the mark is the actual design decision; the code is uniform. one executed tour:

```
1 import matplotlib
2 matplotlib.use("Agg")
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import os
6
7 rng = np.random.default_rng(0)
8 fig, axs = plt.subplots(2, 3, figsize=(10, 6), layout="constrained")
9
10 x = np.linspace(0, 4, 100)
11 axs[0, 0].plot(x, np.exp(-x) * np.cos(6 * x)); axs[0, 0].set_title("line:
12 ↪ ordered x")
13 axs[0, 1].scatter(rng.normal(size=200), rng.normal(size=200), s=8)
14 axs[0, 1].set_title("scatter: paired points")
15 axs[0, 2].bar(["a", "b", "c", "d"], [3, 7, 2, 5]); axs[0, 2].set_title("bar:
16 ↪ categories")
17 axs[1, 0].hist(rng.standard_normal(2000), bins=40); axs[1, 0].set_title("hist:
18 ↪ distribution")
19 img = rng.random((20, 20))
20 axs[1, 1].imshow(img, cmap="viridis"); axs[1, 1].set_title("imshow: a matrix")
21 X, Y = np.meshgrid(np.linspace(-2, 2, 80), np.linspace(-2, 2, 80))
22 axs[1, 2].contour(X, Y, X**2 + 0.5 * Y**2, levels=8)
23 axs[1, 2].set_title("contour: a scalar field")
24
25 fig.savefig("/tmp/mpl-families.png", dpi=110)
```

```
23 print("6 axes in one figure ->", os.path.getsize("/tmp/mpl-families.png"),
    ↪ "bytes")
```

6 axes in one figure -> 106589 bytes

the saved figure (107 kB) is a 2×3 grid, constrained layout keeping all six titles clear of neighbouring ticklabels: a damped cosine line, a gaussian point cloud, four bars, a 40-bin histogram, a viridis heatmap, and eight elliptical contour rings. when to reach for which:

mark	when
<code>plot</code>	<i>y</i> against <i>ordered x</i> — trends, trajectories, functions
<code>scatter</code>	paired observations, no ordering implied; size/colour as extra channels
<code>bar</code>	a handful of <i>categories</i> (if the x-axis is numeric and dense, you wanted a line)
<code>hist</code>	one variable's distribution (bin count is a real modelling choice)
<code>imshow</code>	matrices and images — anything indexed by two integers
<code>contour</code> / <code>contourf</code>	smooth scalar fields $f(x, y)$ — level sets, decision boundaries

5 styling

two mechanisms, layered:

- **rcParams** — a 322-key dictionary of defaults (font sizes, line widths, colour cycle, dpi ...). set globally (`plt.rcParams["font.size"] = 11`), or temporarily with `plt.rc_context({...})`.
- **style sheets** — named rcParams bundles: `plt.style.use("ggplot")`, 29 shipped (measured below), stackable, and you can ship your own `.mplstyle` file per project — the correct way to make every figure in a thesis agree.

6 3d and animation, briefly

- **3d**: `fig.add_subplot(projection="3d")` gives an `Axes3D` with `plot_surface`, `scatter`, `plot_wireframe`. honest advice: matplotlib's 3d is a projection of artists, not a scene graph — occlusion sorting is approximate (artists are z-sorted per collection, not per fragment), so intersecting surfaces render wrongly. fine for a quick surface, use a real 3d engine for more.
- **animation**: `FuncAnimation(fig, update, frames=...)` calls your `update(frame)` to mutate artists in place (`line.set_ydata(...)`), then writes via ffmpeg or pillow. the artist tree pays off again: animation is just property-setting on schedule.

7 backends

a backend is the renderer behind the artist tree: **Agg** rasterises to png (the batch/server default — this page used it), **pdf/svg/ps** write vector files, and gui backends (macosx, qt, tk) couple a renderer to a window and event loop. code is backend-agnostic; you select with `matplotlib.use(...)` before pyplot import, or the `MPLBACKEND` environment variable. if a script “hangs in ssh”, it is waiting on a gui event loop — force **Agg**.

8 saving figures

```
1 import matplotlib
2 matplotlib.use("Agg")
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import os
6
7 x = np.linspace(0, 2 * np.pi, 200)
8 fig, ax = plt.subplots(figsize=(6, 4))
9 ax.plot(x, np.sin(x))
10 ax.set(xlabel="x", ylabel=r"$\sin x$", title="same figure, five files")
11
12 for name, kw in [("100dpi.png", {"dpi": 100}),
13                 ("300dpi.png", {"dpi": 300}),
14                 ("tight.png", {"dpi": 100, "bbox_inches": "tight"}),
15                 ("vector.pdf", {}),
16                 ("vector.svg", {})] :
17     path = "/tmp/mpl-" + name
18     fig.savefig(path, **kw)
19     print(f"{name:>12}: {os.path.getsize(path):7d} bytes")
20
21 print("styles available:", len(plt.style.available),
22       "e.g.", plt.style.available[:3])
23 print("rcParams entries:", len(plt.rcParams))
```

```
100dpi.png:    23785 bytes
300dpi.png:    88269 bytes
tight.png:    23457 bytes
vector.pdf:   13625 bytes
vector.svg:   24729 bytes
```

```
styles available: 29 e.g. ['Solarize_Light2', '_classic_test_patch', '_mpl-
gallery']
```

```
rcParams entries: 322
```

the numbers teach the policy:

- **dpi** only matters for raster: tripling it roughly 3.7×-ed the png. print wants 300; screens are fine at 100–150.
- **bbox_inches="tight"** crops the saved file to the artists' actual extent — the fix for amputated axis labels.
- **vector formats** (pdf, svg) are resolution-independent and here *smaller* than the raster — for line art they are strictly better. use pdf for L^AT_EX documents, svg for the web; reserve png for images and plots with millions of points (where vector files balloon).
- the **pgf backend** (`fig.savefig("plot.pgf")`) emits pgf/tikz commands that L^AT_EX compiles itself, so figure fonts and math are typeset by the *document's* fonts — the same aesthetic this wiki gets by drawing its diagrams in tikz directly. for matplotlib-generated plots destined for a typeset pdf, pgf is the highest-fidelity route.

9 see also

- **numpy** — every `ax.plot` argument is one of its arrays

- `pandas` — `df.plot()` drives this machinery with labels attached
- mastery of `matplotlib` — exercises