basic

May 11, 2022

1 Basic Usage

The tutorials covers some basic patterns and best practices to get started with Matplotlib

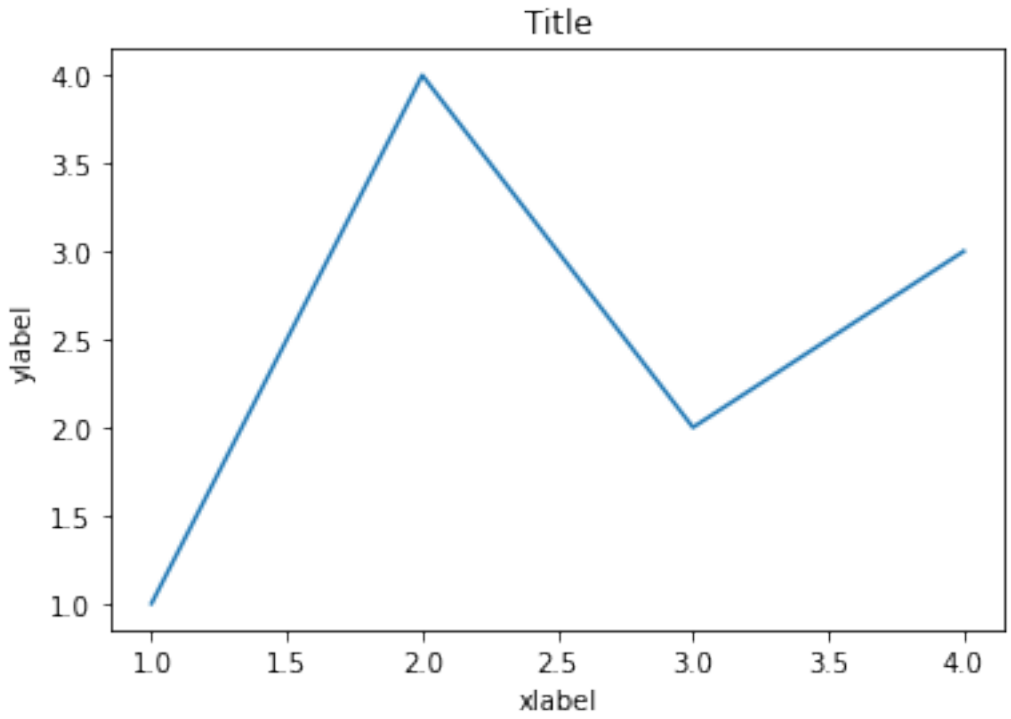
[ ]:
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

1.1 A simple example

Matplotlib graphs the data on **Figures** (e.g., windows, Jupyter widgets, etc.), each of which can contain one or more **Axes**, an area where points can be specified in terms of x-y coordinates (or theta-r in a polar plot, x-y-z in a 3D plot, etc). The simplest way of creating a Figure with an Axes is using `pyplot.subplots`. We can then use `Axes.plot` to draw some data on the Axes

[ ]:
```
fig, ax = plt.subplots()
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
ax.set_xlabel("xlabel")
ax.set_ylabel("ylabel")
ax.set_title("Title")
```

[ ]:
```
Text(0.5, 1.0, 'Title')
```



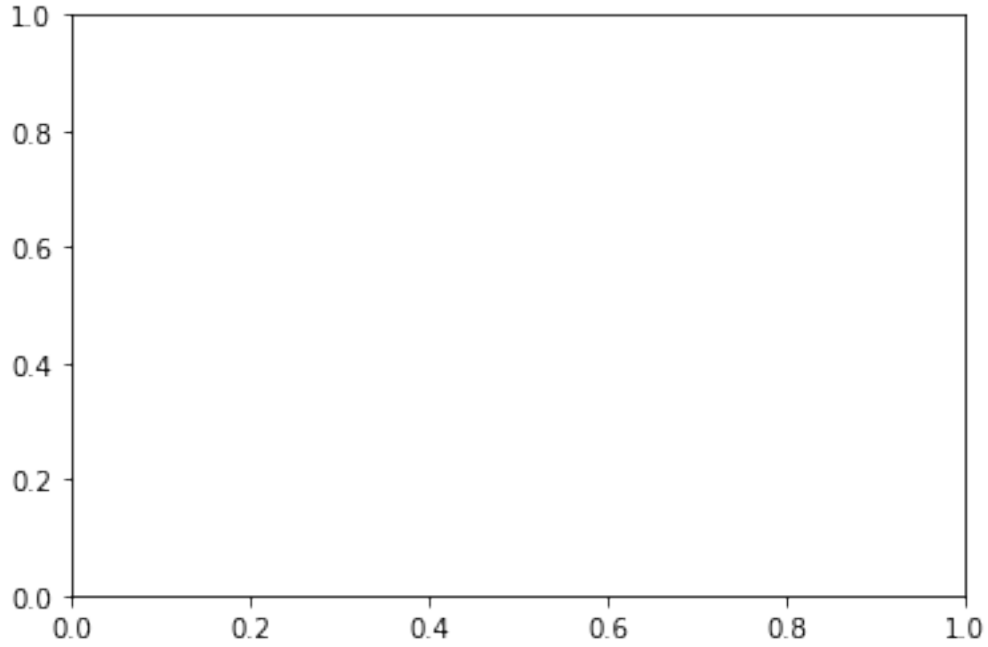
1.2 Figure

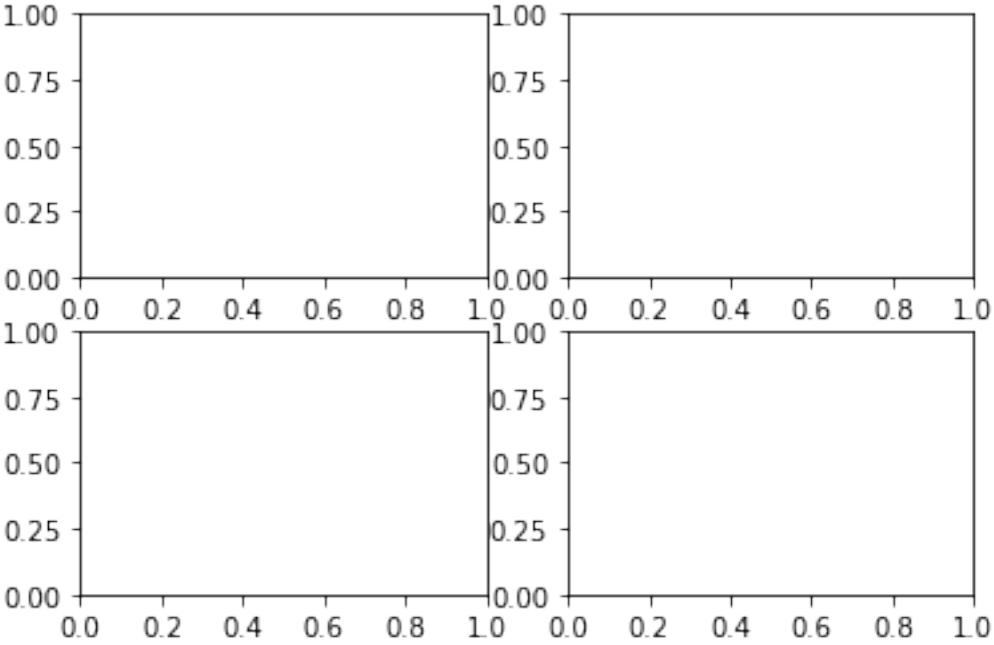
The **whole** figure. The Figure keeps track of all the child **Axes**, a group of 'special' Artists (titles, figure legends, colorbars, etc), and even nested subfigures.

The easiest way to create a new Figure is with pyplot

[ ]:
```
fig = plt.figure()
fig, ax = plt.subplots()
fig, axs = plt.subplots(2, 2)
```

```
<Figure size 432x288 with 0 Axes>
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.00
0.75
0.50
0.25
0.00
0.0
0.2
0.4
0.6
0.8
1.0

1.3 Types of inputs to plotting functions

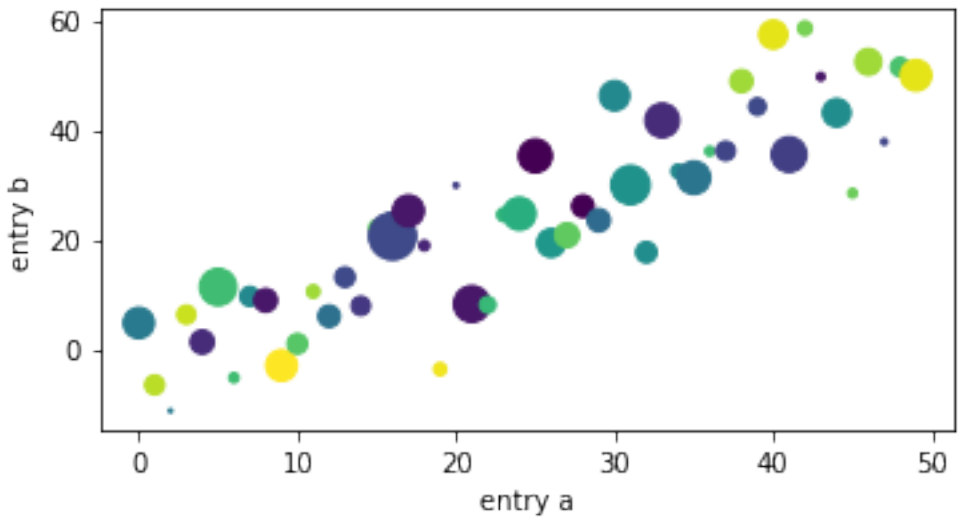
Plotting functions expect `numpy.array` or `numpy.ma.masked_array` as input, or objects that can be passed to `numpy.asarray`. Classes that are similar to arrays ('array-like') such as `pandas` data objects and `numpy.matrix` may not work as intended. Common convention is to convert these to `numpy.array` objects prior to plotting. For example, to convert a `numpy.matrix`.

[ ]:
```
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

Most methods will also parse an addressable object like a dict, a `numpy.recarray`, or a `pandas.DataFrame`. Matplotlib allows you provide the data keyword argument and generate plots passing the strings corresponding to the x and y variables.

[ ]:
```
np.random.seed(19680801)
data = {
    'a': np.arange(50),
    'c': np.random.randint(0, 50, 50),
    'd': np.random.randn(50)
}

data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.scatter('a', 'b', c='c', s='d', data=data)
ax.set_xlabel("entry a")
ax.set_ylabel("entry b")
```

[ ]:
```
Text(0, 0.5, 'entry b')
```



1.4 Coding styles

1.4.1 The object-oriented and the pyplot interfaces

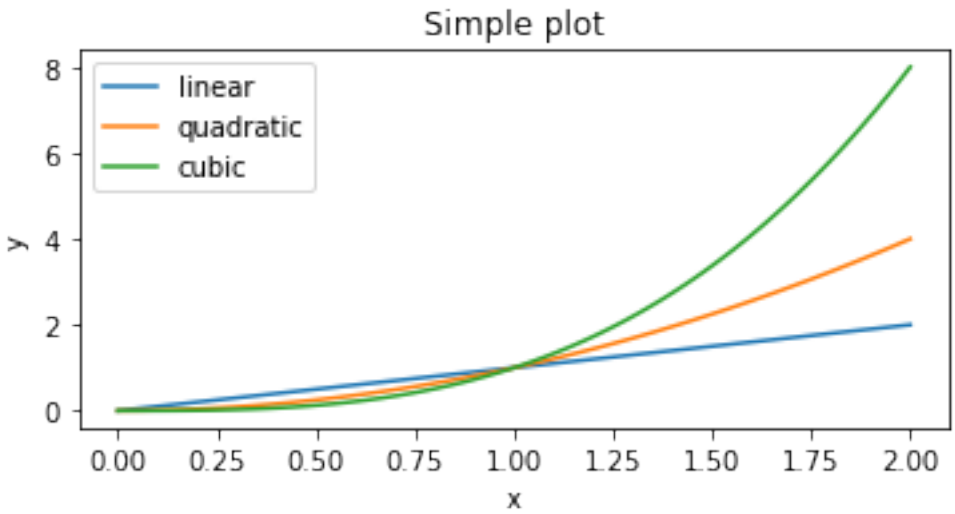
As noted above, there are essentially two ways to use Matplotlib

- Explicitly create Figures and Axes, and call methods on them (the "object-oriented (OO) style").
- Rely on pyplot to automatically create and manage the Figures and Axes, and use pyplot functions for plotting.

So one can use the OO-style

[ ]:
```
## Object-Oriented Style for plotting using Matplotlib.pyplot subpackage

x = np.linspace(0, 2, 100)

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
ax.plot(x, x, label='linear')
ax.plot(x, x**2, label='quadratic')
ax.plot(x, x**3, label='cubic')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Simple plot')
ax.legend()
```

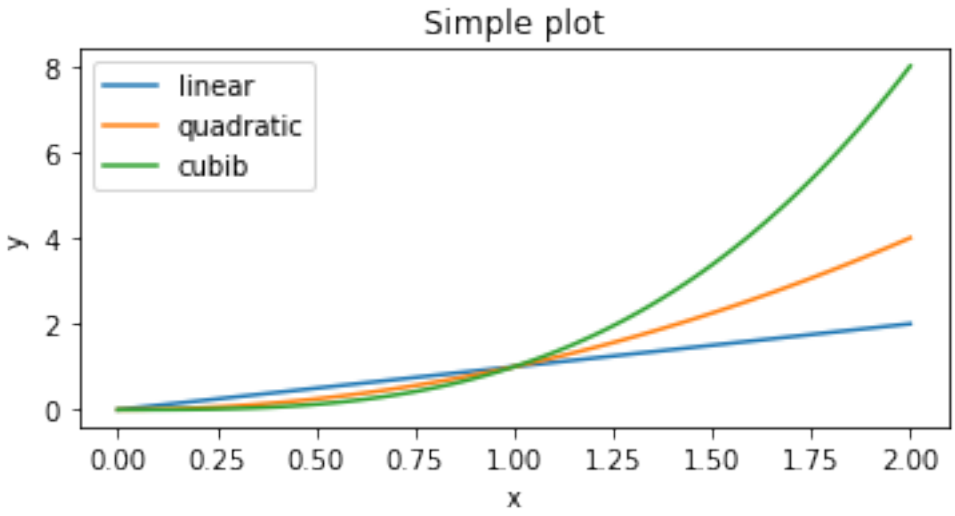
[ ]:
```
<matplotlib.legend.Legend at 0x7fd4e9138f90>
```



or the pyplot-style

[ ]:
```
## pyplot style

x = np.linspace(0, 2, 100)

plt.figure(figsize=(5, 2.7), layout='constrained')
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubib')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Simple plot')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7fd4eb949390>
```



1.5 Making a helper functions

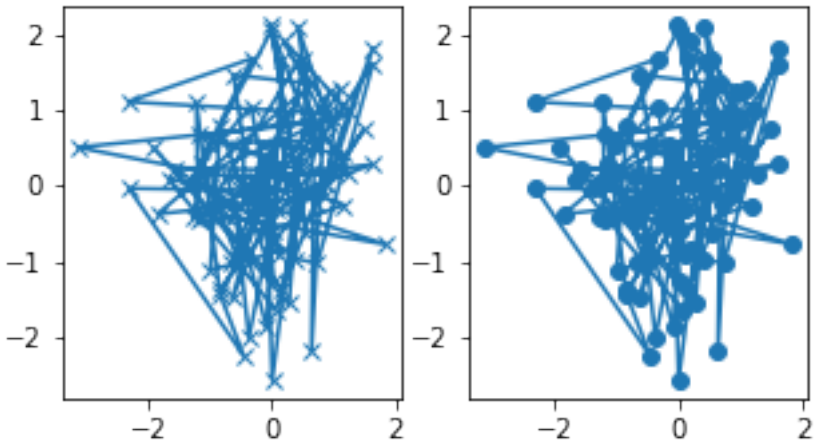
If you need to make the same plots over and over again with different data sets, or want to easily wrap Matplotlib methods, use the recommended signature function below.

Which you would then use twice to populate two subplots

[ ]:
```
def my_plotter(ax, data1, data2, param_dict):
    """A helper function to make a graph"""
    out = ax.plot(data1, data2, **param_dict)
    return out

data1, data2, data3, data4 = np.random.randn(4, 100)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(5, 2.7))
```

```
my_plotter(ax1, data1, data2, {'marker': 'x'})
my_plotter(ax2, data1, data2, {'marker': 'o'})
```

[ ]: [<matplotlib.lines.Line2D at 0x7fd4e8fde510>]

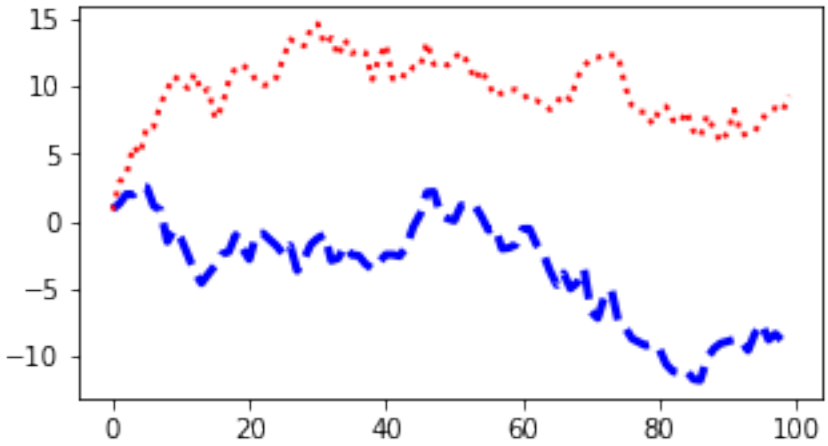


1.6 Styling Artists

Most plotting methods have styling options for the Artists, accessible either when a plotting method is called, or from a "setter" on the Artist. In the plot below we manually set the *color*, *linewidth*, and *linestyle* of the Artists created by **plot**, and we set the linestyle of the second line after the fact with **set_linestyle**.

[ ]:
```
fig, ax = plt.subplots(figsize=(5, 2.7))
x = np.arange(len(data1))
ax.plot(x, np.cumsum(data1), color='blue', linewidth=3, linestyle='--')
ax.plot(x, np.cumsum(data2), color='red', linewidth=2, linestyle=':')
```

[ ]: [<matplotlib.lines.Line2D at 0x7fd4e8f6f5d0>]

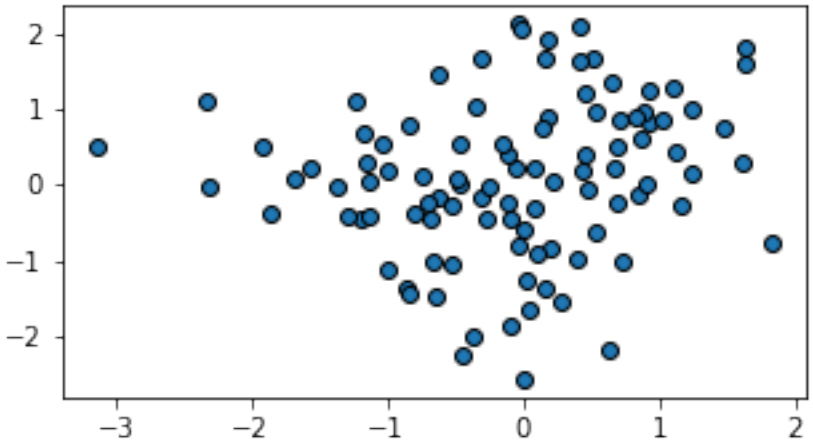


1.7 Colors

Matplotlib has a very flexible array of colors that are accepted for most Artists. Some Artists will take multiple colors. i.e. for a **scatter** plot, the edge of the markers can be different colors from the interior.

[ ]:
```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.scatter(data1, data2, facecolor='C0', edgecolor='k')
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x7fd4e8f26650>
```



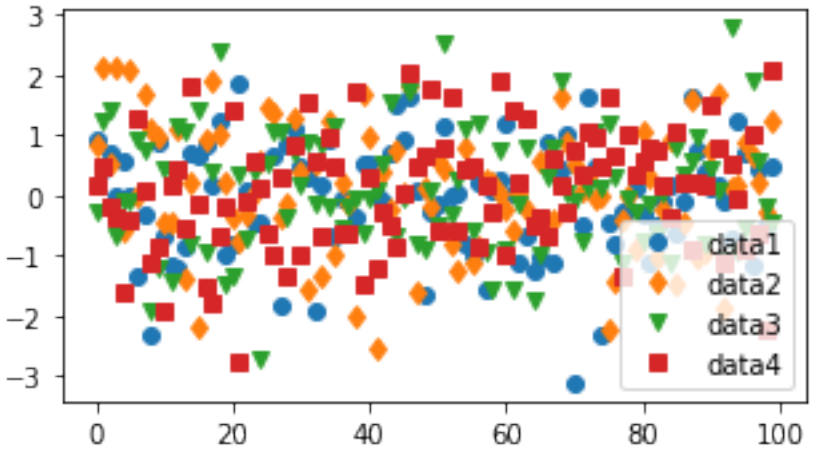
1.8 Linewidths, linestyles, and markersizes

Line widths are typically in typographic points (1 pt = 1/72 inch) and available for Artists that have stroked lines. Similarly, stroked lines can have a linestyle.

Marker size depends on the method being used. **plot** specifies markersize in points, and is generally the "diameter" or width of the marker. **scatter** specifies markersize as approximately proportional to the visual area of the marker. There is an array of markerstyles available as string codes (see **markers**), or users can define their own **MarkerStyle**.

[ ]:
```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(data1, 'o', label='data1')
ax.plot(data2, 'd', label='data2')
ax.plot(data3, 'v', label='data3')
ax.plot(data4, 's', label='data4')
ax.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7fd4e8e4ce10>
```



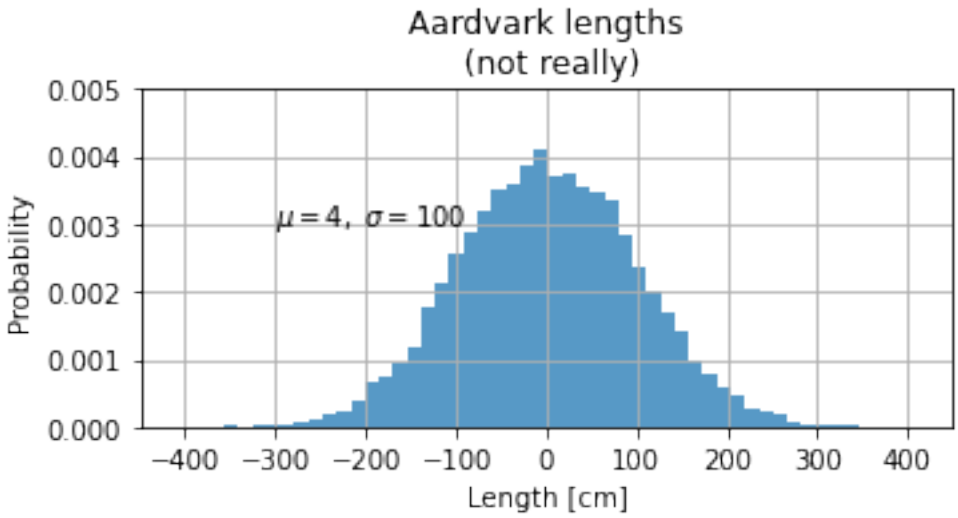
1.9 Labelling plots

1.9.1 Axes labels and text

`set_xlabel`, `set_ylabel`, and `set_title` are used to add text in the indicated locations. Text can also be directly added to plots using `text`

[ ]:
```
mu, sigma = 4, 100
x = mu + sigma * np.random.randn(10000)
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
n, bins, patches = ax.hist(x, bins=50, density=1, facecolor='C0', alpha=0.75)
ax.set_xlabel('Length [cm]')
```

```
ax.set_ylabel('Probability')
ax.set_title('Aardvark lengths\n (not really)')
ax.text(-300, .003, r'$\mu=4, \ \sigma=100$')
ax.axis([-450, 450, 0, 0.005])
ax.grid(True)
```

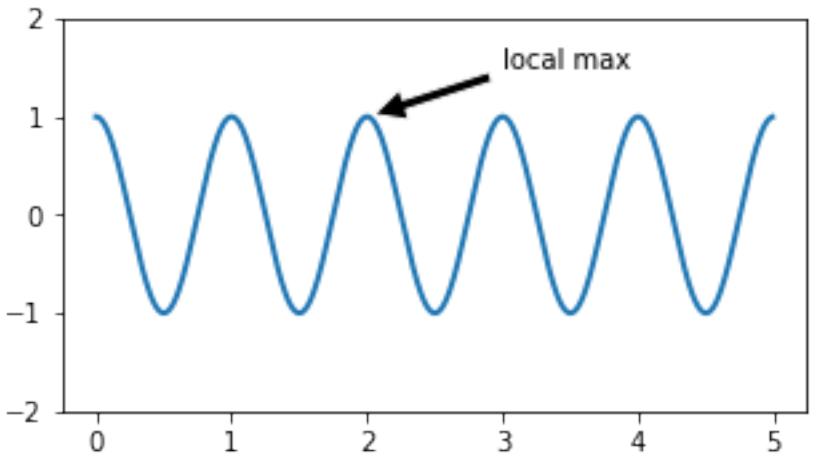


1.9.2 Annotations

We can also annotate points on a plot, often by connecting an arrow pointing to *xy*, to a piece of text at *xytext*.

[ ]:
```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, linewidth=2)
ax.annotate(
    'local max', xy=(2, 1), xytext=(3, 1.5),
    arrowprops=dict(facecolor='black', edgecolor='white', shrink=0.05)
)
ax.set_ylim(-2, 2)
```

[ ]:
```
(-2.0, 2.0)
```



1.10 Acis scales and ticks

Each Axes has two (or three) **Axis** objects representing the x- and y-axis. These control the *scale* of the Axis, the tick *locators* and the tick *formatters*. Additional Axes can be attached to display further Axis objects.

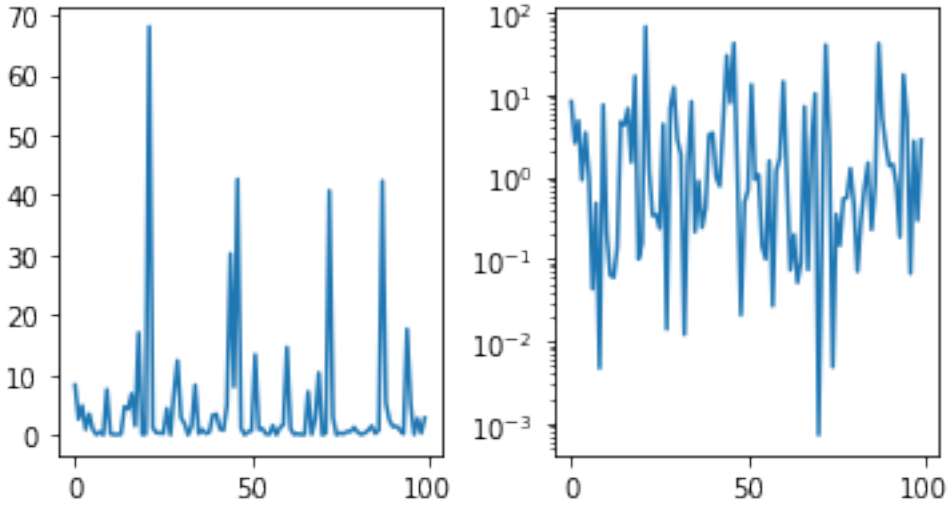
1.10.1 Scales

In addition to the linear scale, Matplotlib supplies non-linear scales, such as a log-scale. Since log-scales are used so much there are also direct methods like **loglog**, **semilogx**, and **semilogy**. There are a number of scales. Here we set the scale manually

The scale sets the mapping from data values to spacing along the Axis. This happens in both directions, and gets combined into a *transform*, which is the way that Matplotlib maps from data coordinates to Axes, Figure, or screen coordinates.

[ ]:
```
fig, axs = plt.subplots(1, 2, figsize=(5, 2.7), layout='constrained')
xdata = np.arange(len(data1))
data = 10 ** data1
axs[0].plot(xdata, data)
axs[1].set_yscale('log')
axs[1].plot(xdata, data)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7fd4e8b87390>]
```



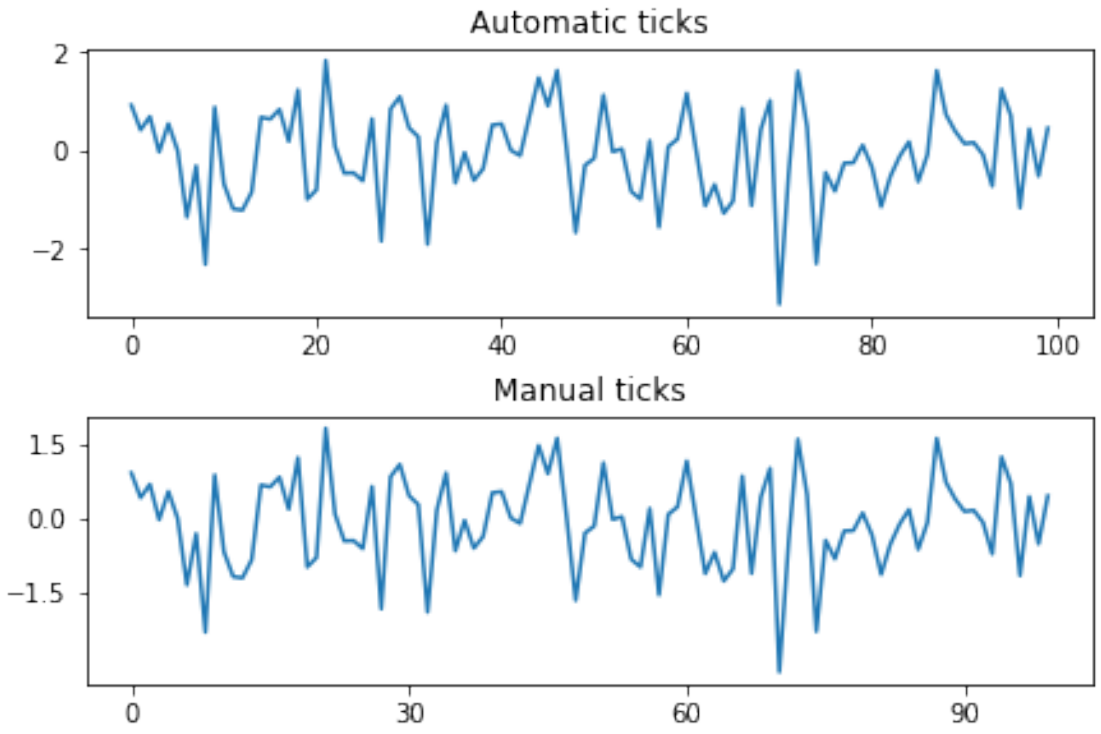
1.10.2 Ticks locators and formatters

Each Axis has a tick *locator* and *formatter* that choose where along the Axis objects to put tick marks. A simple interface to this is **set_xticks**.

Different scales can have different locators and formatters; for instance the log-scale above uses **LogLocator** and **LogFormatter**

[ ]:
```
fig, axs = plt.subplots(2, 1, layout='constrained')
axs[0].plot(xdata, data1)
axs[0].set_title('Automatic ticks')

axs[1].plot(xdata, data1)
axs[1].set_xticks(np.arange(0, 100, 30), ['0', '30', '60', '90'])
axs[1].set_yticks([-1.5, 0, 1.5])
axs[1].set_title('Manual ticks')
```

[ ]:
```
Text(0.5, 1.0, 'Manual ticks')
```



1.11 Plotting dates and strings

Matplotlib can handle plotting arrays of dates and arrays of strings, as well as floating point numbers. These get special locators and formatters as appropriate. For dates

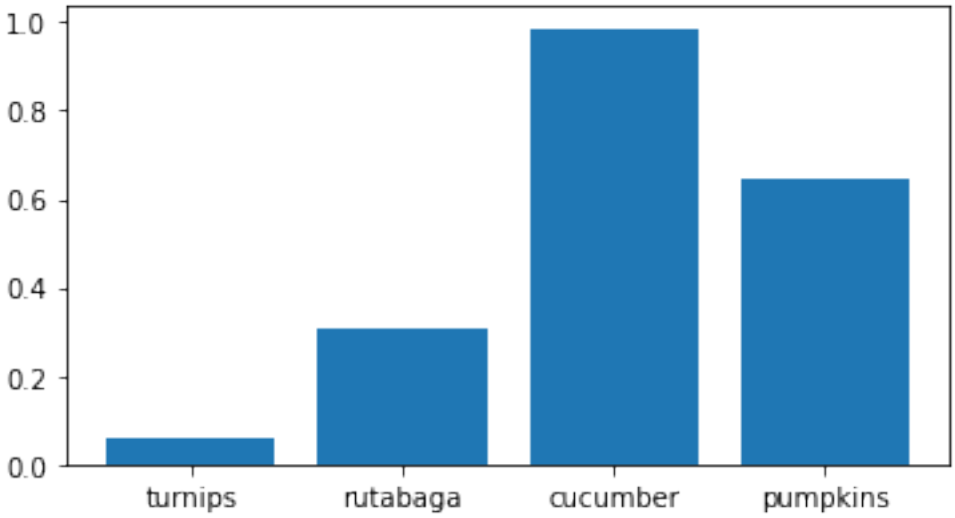
[ ]:
```
# fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
# dates = np.arange(
#     np.datetime64('2011-11-15'),
#     np.datetime64('2011-12-25'),
#     np.timedelta64(1, 'h')
# )

# data = np.cumsum(np.random.randn(len(dates)))
# ax.plot(dates, data)
# cdf = mpl.dates.ConciseDateFormatter(ax.xasix.get_major_locator())
# ax.xaxis.set_major_locator(cdf)
```

For strings, we get categorical plotting.

One caveat about categorical plotting is that some methods of parsing text files return a list of strings, even if the strings all represent numbers or dates. If you pass 1000 strings, Matplotlib will think you meant 1000 categories and will add 1000 ticks to your plot!

[ ]:
```
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
categories = ['turnips', 'rutabaga', 'cucumber', 'pumpkins']

ax.bar(categories, np.random.rand(len(categories)))
```

[ ]:
```
<BarContainer object of 4 artists>
```



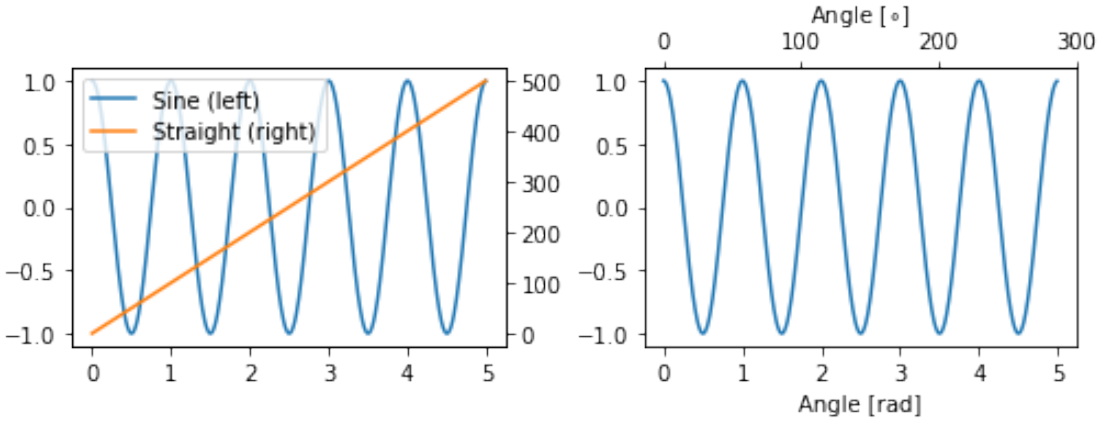
1.12 Additional Axis objects

Plotting data of different magnitude in one chart may require an additional y-axis. Such an Axis can be created by using **twinx** to add a new Axes with an invisible x-axis and a y-axis positioned at the right (analogously for **twiny**).

Similarly, you can add a **secondary_xaxis** or **secondary_yaxis** having a different scale than the main Axis to represent the data in different scales or units. See Secondary Axis for further examples.

[ ]:
```
fig, (ax1, ax3) = plt.subplots(1, 2, figsize=(7, 2.7), layout='constrained')

l1, = ax1.plot(t, s)
ax2 = ax1.twinx()
l2, = ax2.plot(t, range(len(t)), 'C1')
ax2.legend([l1, l2], ['Sine (left)', 'Straight (right)'])

ax3.plot(t, s)
ax3.set_xlabel('Angle [rad]')
ax4 = ax3.secondary_xaxis('top', functions=(np.rad2deg, np.deg2rad))
ax4.set_xlabel(r'Angle [$\circ$]')
```

[ ]: Text(0.5, 0, 'Angle [$\\circ$]')

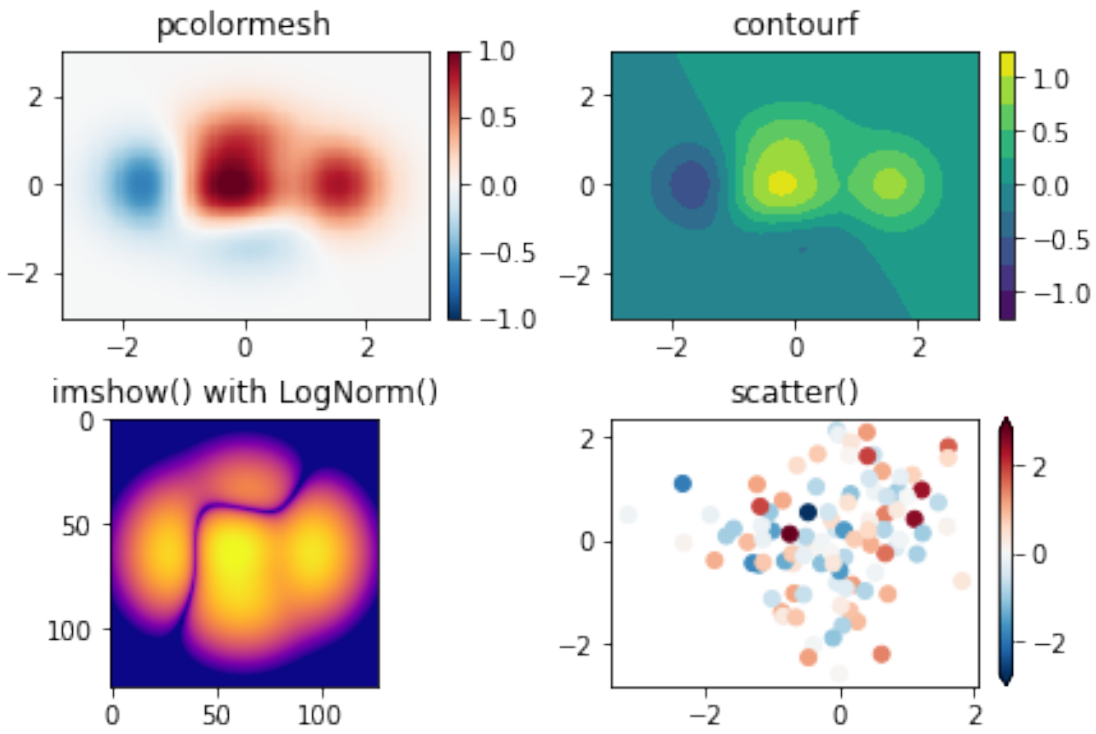


1.13 Color mapped data

Often we want to have a third dimension in a plot represented by a colors in a colormap. Matplotlib has a number of plot types that do this

[ ]:
```
X, Y = np.meshgrid(np.linspace(-3, 3, 128), np.linspace(-3, 3, 128))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

fig, axs = plt.subplots(2, 2, layout='constrained')

pc = axs[0, 0].pcolormesh(X, Y, Z, vmin=-1, vmax=1, cmap="RdBu_r")
fig.colorbar(pc, ax=axs[0, 0])
axs[0, 0].set_title('pcolormesh')

co = axs[0, 1].contourf(X, Y, Z, levels=np.linspace(-1.25, 1.25, 11))
fig.colorbar(co, ax=axs[0, 1])
axs[0, 1].set_title('contourf')

pc = axs[1, 0].imshow(Z**2 * 100, cmap='plasma', norm=mpl.colors.LogNorm(vmin=0.
↪01, vmax=100))
axs[1, 0].set_title('imshow() with LogNorm()')

pc = axs[1, 1].scatter(data1, data2, c=data3, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[1, 1], extend='both')
axs[1, 1].set_title('scatter()')
```

[ ]: Text(0.5, 1.0, 'scatter()')



Created in Deepnote