Programming Level-up

An Introduction to Matplotlib

Jay Morgan

Outline

What is Matplotlib?

In summary:

- Matplotlib is one of the defacto plotting libraries for Python. While there are many others and certainly some that are built for specific plot types, Matplotlib continues to pervade scientific plotting.
- You can create basic plots (such as line or scatter plots) to more complicated plots that include interactivity.

Installing and importing Matplotlib

Matplotlib can be installed via conda:

```
1 conda install matplotlib
```

or with pip:

```
2 pip install matplotlib
```

Remember! You can install packages in ipython REPL/juypter notebook by inserting a '!' to the beginning of a shell command.

Basic plotting

First, we will import the matplotlib module. The plotting function is located within the `pyplot` package within matplotlib. The use of this package is so common that 99% of Python users will alias this import as `plt`:

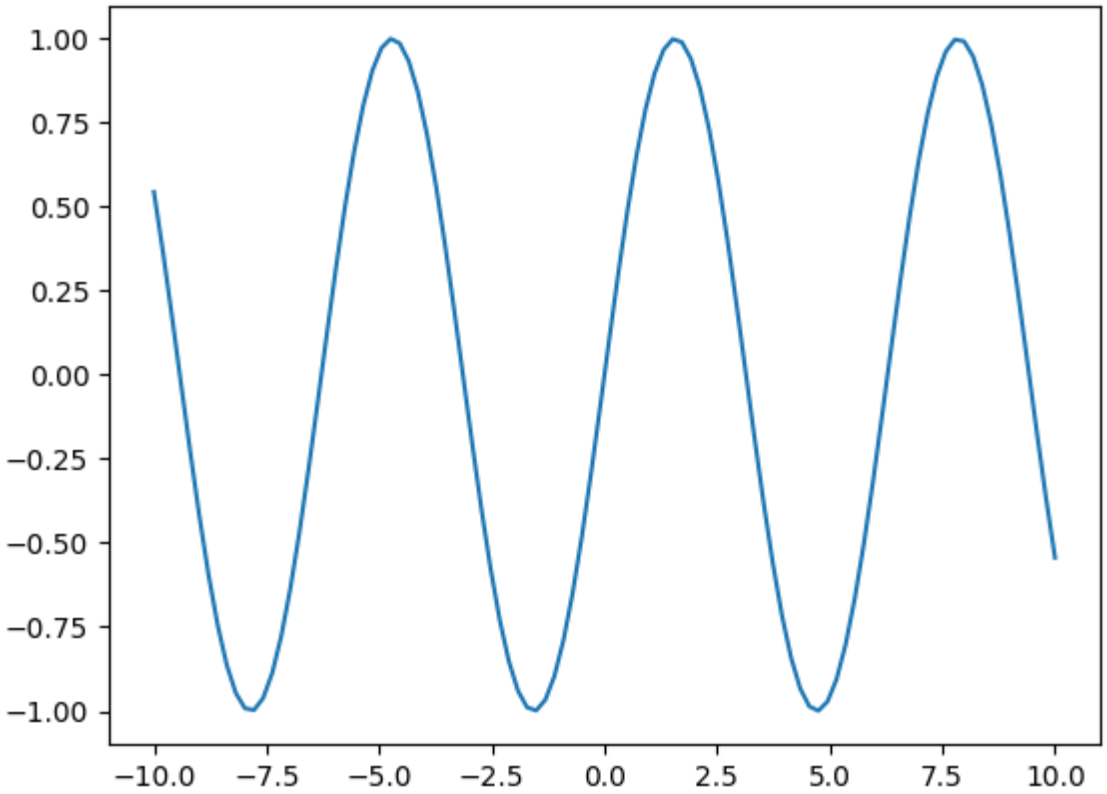
```
3 import matplotlib.pyplot as plt
```

With this package now imported, we can now use the `plot` function. To begin with, let's just plot a simple line chart. In this case, the `plot` function takes an `x` and `y` argument, where `x` denotes the values along the x-axis and `y` are the values along the y-axis.

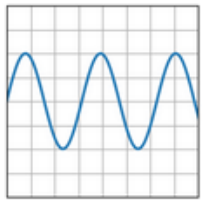
```
4 x = np.linspace(-10, 10, 100)
5 y = np.sin(x)
6 plt.plot(x, y)
```

In this example, we have created two vectors. The first `x`, creates a vector of 100 values from -10 to 10. `y` is the sin function applied to `x`. Finally, in the third line, we plot the sin wave using these two vectors.

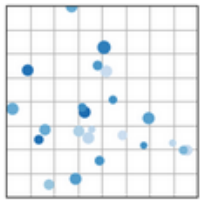
Basic plotting



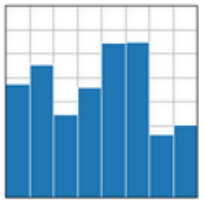
Different types of Plots



plot(x, y)



scatter(x, y)



bar(x, height) / barh(y, width)

There are many different types of plots that one can make using matplotlib. These include the most popular:

- Line plots
- Scatter plots
- Bar plots
- Histograms
- Box plots
- Image plots

We're going to take a look at how we create each type of plot, examining what type of inputs they require.

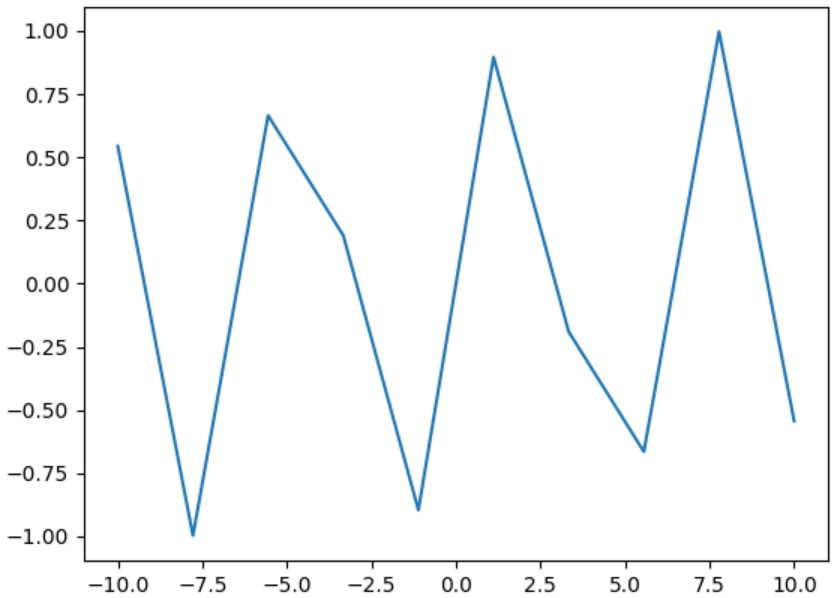
Line plots

We've already seen one example of a line plot. This plot draws a line between each x,y point. For instance in the previous example, we created a sin wave by 'sampling' such wave using 100 samples from -10 to 10. Let's see what happens when we sample only 10 points:

```
7  x = np.linspace(-10, 10, 10)
8  y = np.sin(x)
9  plt.plot(x, y)
```

Line plots

We see the results are a less than ideal representation of a sin wave as `plot` will simply draw a straight line from each point.



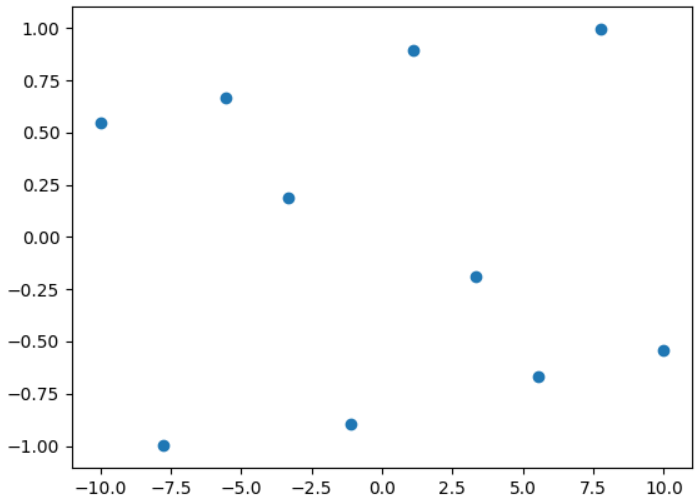
Scatter plots

If we want to see where each sample of the sin wave is, we could use instead the scatter plot, which will (by default) place a small circle at every x,y value. To create a scatter plot, we use `scatter` instead of the `plot` function. The arguments to this function are the same, however.

```
10  x = np.linspace(-10, 10, 10)
11  y = np.sin(x)
12  plt.scatter(x, y)
```

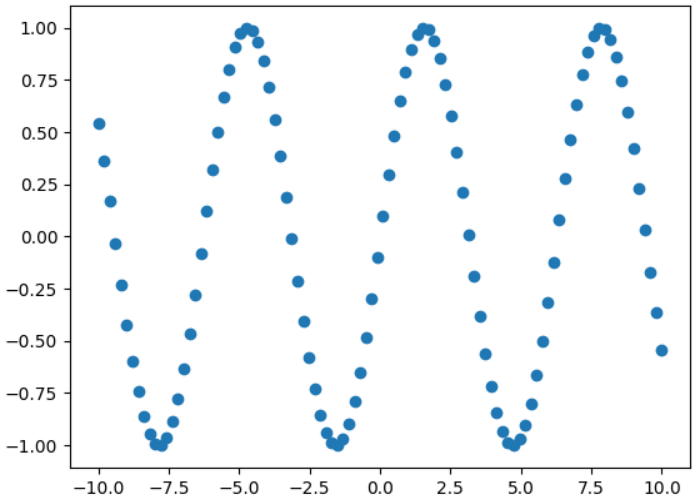
Scatter plots

Now we can see the position of each individual sample from the sin wave. If we, once again, sample 100 points from this curve, we will see better results.



Scatter plots

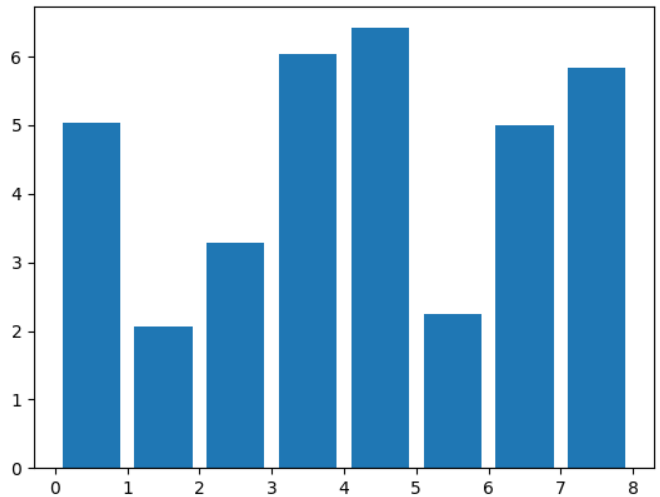

```
13  x = np.linspace(-10, 10, 100)
14  y = np.sin(x)
15  plt.scatter(x, y)
```



Bar plots

Bar plots are a simple plot that again takes an `x` and a `y`, where `x` is the numerical position of the bar's centre, and `y` is the height of the bar.

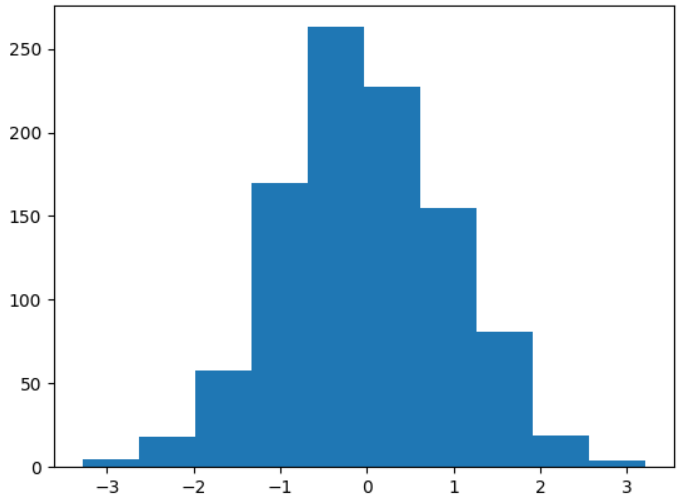
```
16  x = np.arange(0, 8)
17  y = np.random.uniform(2, 7, len(x))
18  plt.bar(x, y)
```



Histograms

Histograms allow us to visualise the distribution of values. In matplotlib, we can create a histogram of a vector by using the `hist` function that takes only the vector as its argument.

```
19 x = np.random.randn(1000)
20 plt.hist(x)
```



Box plots

Box plots also allow us to visualise the distribution, but the distribution of values within a group. In this example we're visualising the distribution of 3 groups. Using the `boxplot` function, we pass a matrix.

```
21 x = np.random.randn(10, 3)
22 plt.boxplot(x)
```

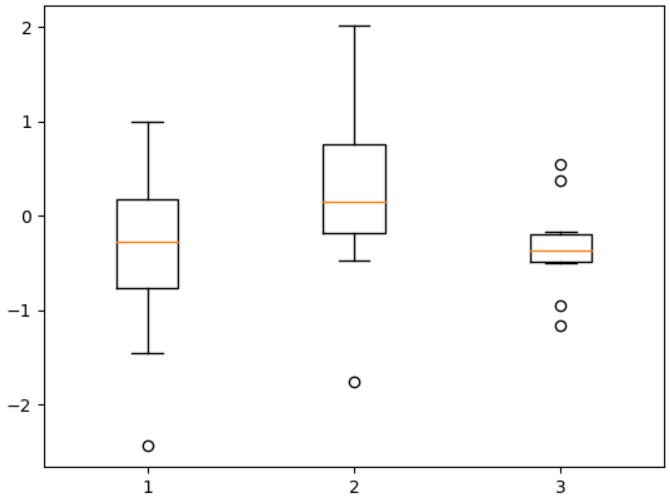


Image plots

In matplotlib, we can plot an 'image' – that is a 2D matrix – using the `imshow` function. For example:

```
23 fig = plt.figure()
24 x = np.random.randn(10, 10)
25 plt.imshow(x)
```

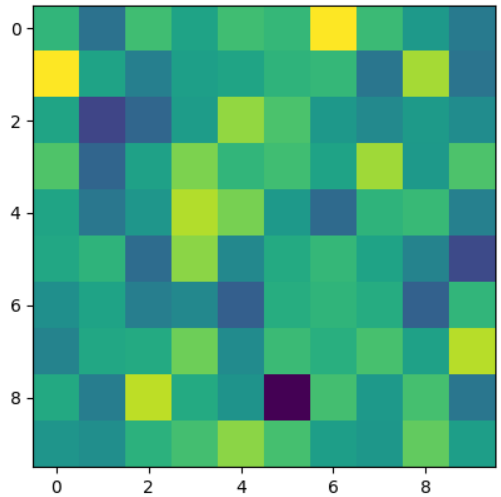
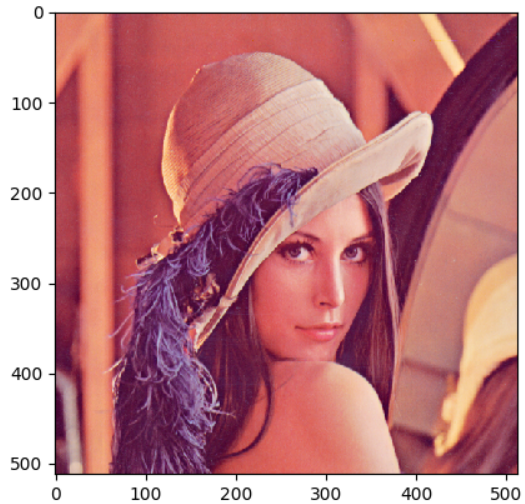


Image plots

Of course, given the name, we can then use `imshow` to plot an image as well, as long as we have the image loaded as a 2D array of values.

```
26 import PIL  # using the PIL module to read an image
27 img = np.array(PIL.Image.open("images/Lenna.png"))
28 plt.imshow(img)
```



Different types of Plots

There are many more different types of plots you can make using matplotlib. You can find a comprehensive list at:

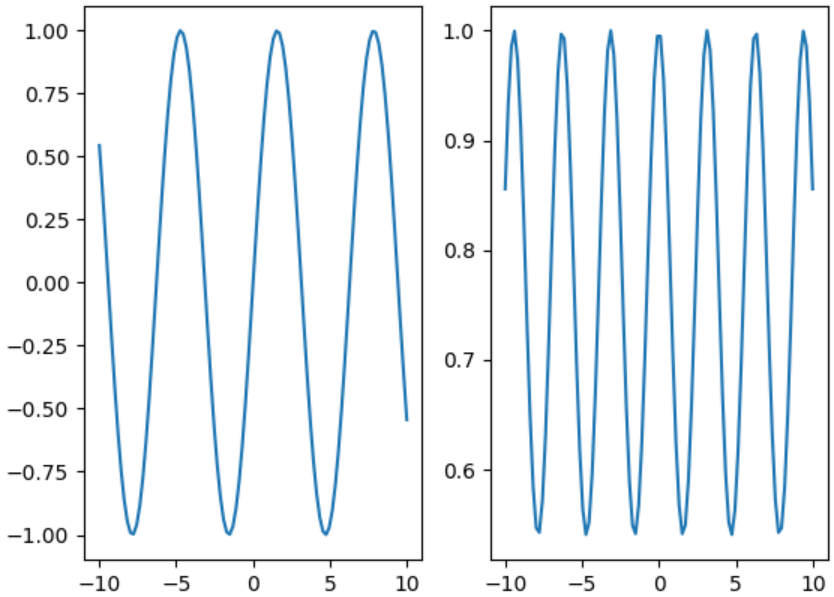
`https://matplotlib.org/stable/plot_types/index.html`

Subplots

What if we wanted to create many plots side-by-side? For this we can use the `subplots` function. This function takes the number of rows, and number of columns to create. It returns two values, the first is the figure (entire figure), and the second value is a list of sub figures. Using this list, we can place a plot of each of them.

```
29 x = np.linspace(-10, 10, 100)
30 y = np.sin(x)
31 z = np.cos(y)
32
33 fig, ax = plt.subplots(1, 2)
34 # ax is a list of sub figures
35 ax[0].plot(x, y)
36 ax[1].plot(x, z)
```

Subplots

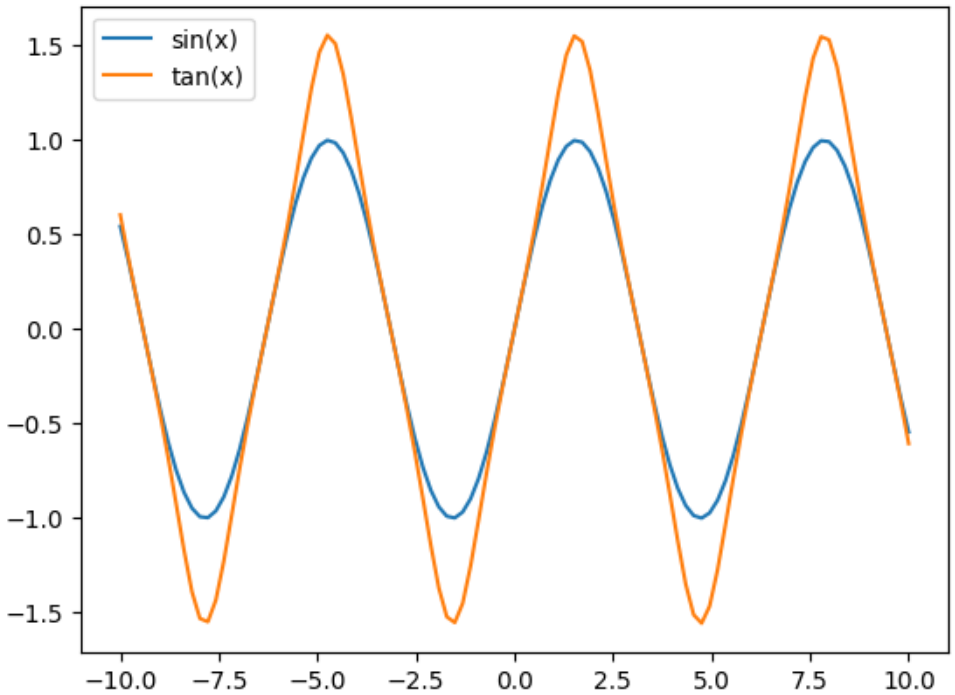

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−10
−5
0
5
10
1.0
0.9
0.8
0.7
0.6
−10
−5
0
5
10

Adding a legend

Or we could put them onto the same plot. Matplotlib will automatically give them a different colour. If we use the `label` argument to `plot`, we can also give them a name that will appear when we call `legend()`.

```
37 x = np.linspace(-10, 10, 100)
38 y = np.sin(x)
39 z = np.tan(y)
40 fig, ax = plt.subplots()
41 ax.plot(x, y, label="sin(x)")
42 ax.plot(x, z, label="tan(x)")
43 ax.legend()
```

Adding a legend



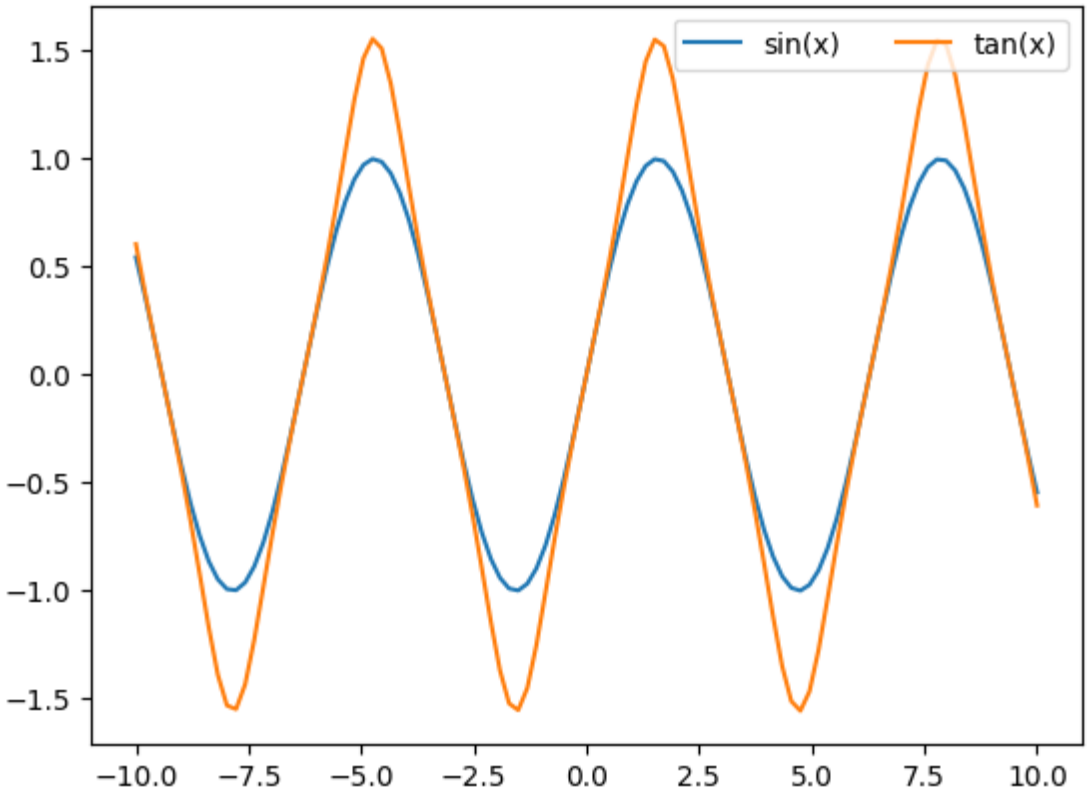
Position the legend in different places

We can change the position of the legend by specifying a different integer value for the `loc` argument (or string values such as 'upper left', 'upper right', ...). Additionally, we can change the number of columns the legend has with the `ncol` argument.

```
44  x = np.linspace(-10, 10, 100)
45  y = np.sin(x)
46  z = np.tan(y)
47
48  fig, ax = plt.subplots()
49  ax.plot(x, y, label="sin(x)")
50  ax.plot(x, z, label="tan(x)")
51  ax.legend(loc=1, ncol=2)
```

You can find the API reference for the different arguments to legend at: `https://matplotlib.org/stable/api/legend_api.html?highlight=legend#module-matplotlib.legend`

Position the legend in different places

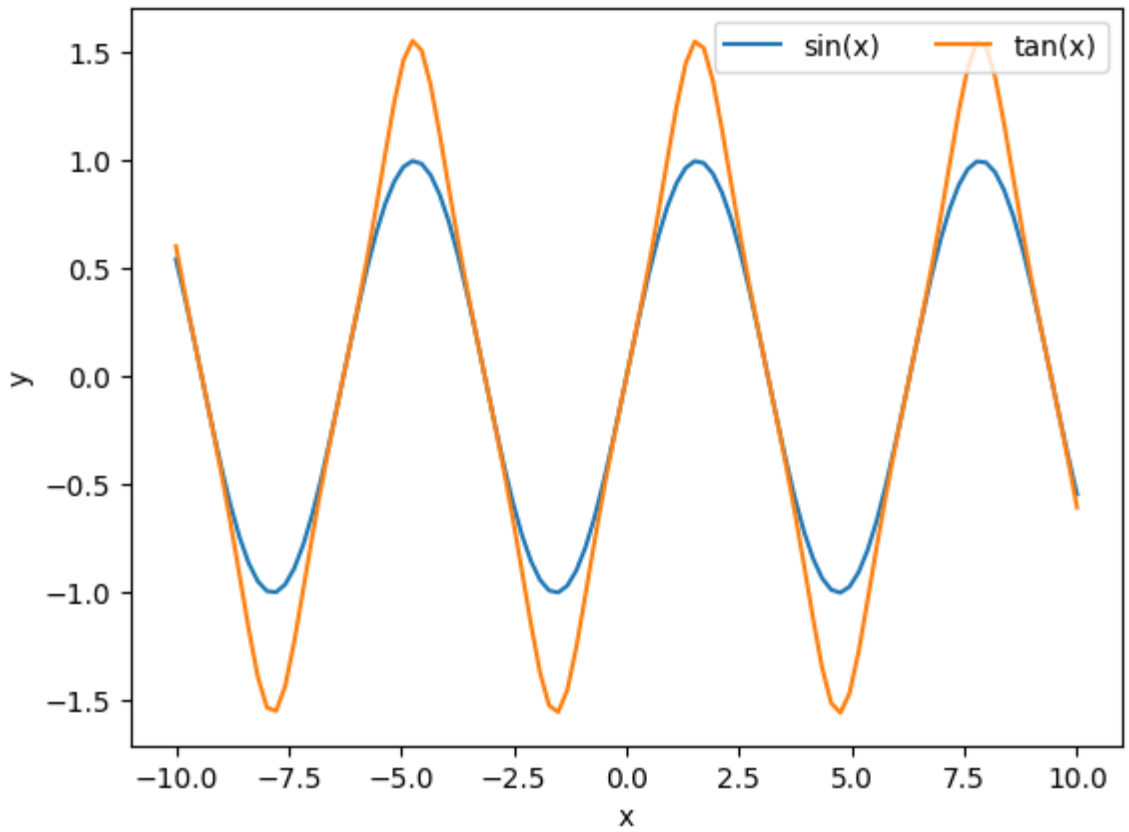


Modifying the x/y axis

Good graphs always have their axis's labelled. To do this in matplotlib, if we have a subplot object, we use `set_xlabel`, or we can use `plt.xlabel(...)`. Here is an example with an subplot object:

```
52  x = np.linspace(-10, 10, 100)
53  y = np.sin(x)
54  z = np.tan(y)
55
56  fig, ax = plt.subplots()
57  ax.plot(x, y, label="sin(x)")
58  ax.plot(x, z, label="tan(x)")
59  ax.legend(loc=1, ncol=2)
60  ax.set_xlabel("x")
61  ax.set_ylabel("y")
```

Modifying the x/y axis



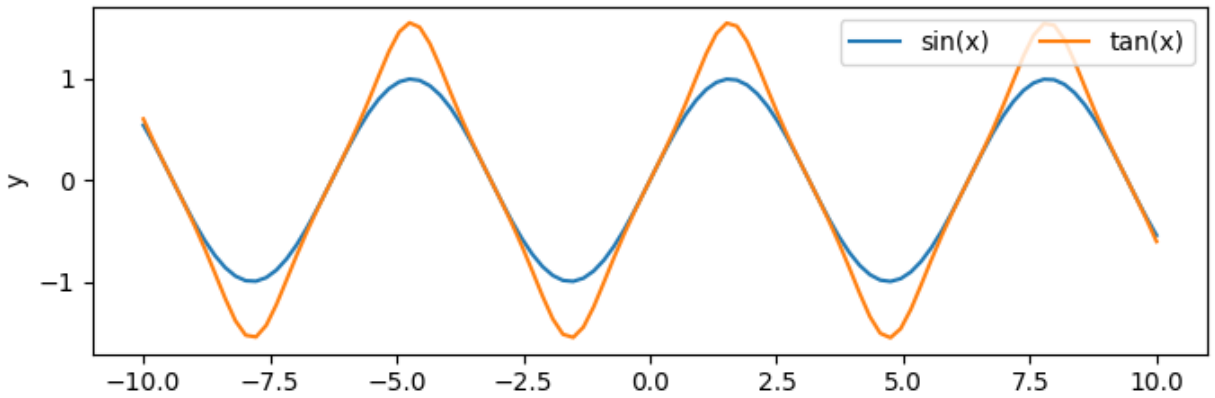
Changing figure size

A common change you may want to make to your figure is to change its size or aspect ratio. `figure()` or `subplots()` take an optional argument called `figsize`. This argument expects a tuple representing the width and height of the figure in inches.

```
62  fig = plt.figure(figsize=(8, 2.5))
63
64  # or most likely
65  fig, ax = plt.subplots(figsize=(8, 2.5))
66  x = np.linspace(-10, 10, 100)
67  y = np.sin(x)
68  z = np.tan(y)
69  ax.plot(x, y, label="sin(x)")
70  ax.plot(x, z, label="tan(x)")
71  ax.legend(loc=1, ncol=2)
72  ax.set_xlabel("x")
73  ax.set_ylabel("y")
```

Here we are creating a figure with 8 inches of width, and 2.5 inches of height.

Changing figure size

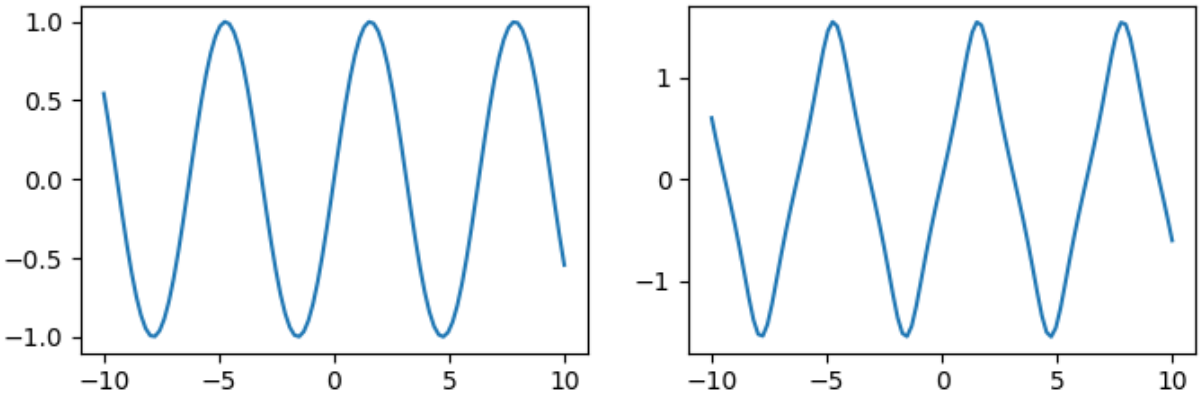


Changing figure size

This is especially useful when you have many sub-figures, as by default, they will be 'squashed' into the default aspect ratio. We can 'give them more space' by modifying this `figsize` argument when creating the many sub-figures.

```
74  fig, ax = plt.subplots(1, 2, figsize=(8, 2.5))
75  x = np.linspace(-10, 10, 100)
76  y = np.sin(x)
77  z = np.tan(y)
78  ax[0].plot(x, y, label="sin(x)")
79  ax[1].plot(x, z, label="tan(x)")
```

Changing figure size



Line properties

When creating a plot, there are many different properties you can change. Some of these include:

- color – the colour of the line
- alpha – the amount of transparency (1.0 is opaque, 0.0 is transparent)
- linewidth, lw – the width of the stroke width
- linestyle, ls – the style of the line (i.e. a dotted line)

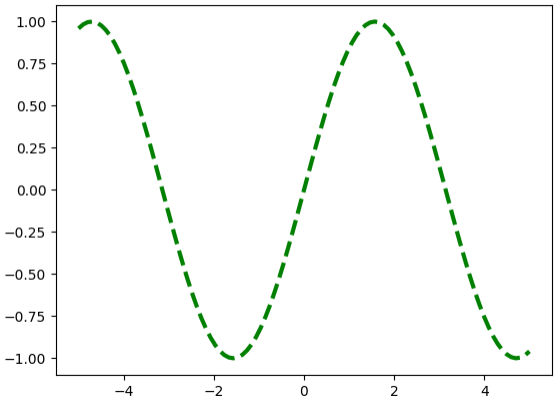
There are also some properties for the markers, i.e. the circles in the scatter plot. These properties are:

- marker – the type of marker (you can use different shapes instead of a circle
- markersize – the size of the mark
- markerfacecolor – colour of the marker
- markeredgewidth – outline width of the marker.

Line properties

If this example we are modifying some of the line properties that include the color (c), setting it to a string value of "green". The linewidth (lw) to be thicker, and making the line to be a dotted line by specifying the linestyle (ls) to "=–={".

```
80  fig = plt.figure()
81  x = np.linspace(-5, 5, 100)
82  y = np.sin(x)
83  plt.plot(x, y,
84           c="green", # or color
85           lw=3, # or linewidth
86           ls="--")
```



Colormaps

When we create a heatmap using `imshow`, the gradients of colour are automatically set. Yet, we can control the colour gradient using a colour map. First we must import `cm` from matplotlib:

```
87 from matplotlib import cm
```

Then we can get a colour map with 10 levels using `get_cmap`:

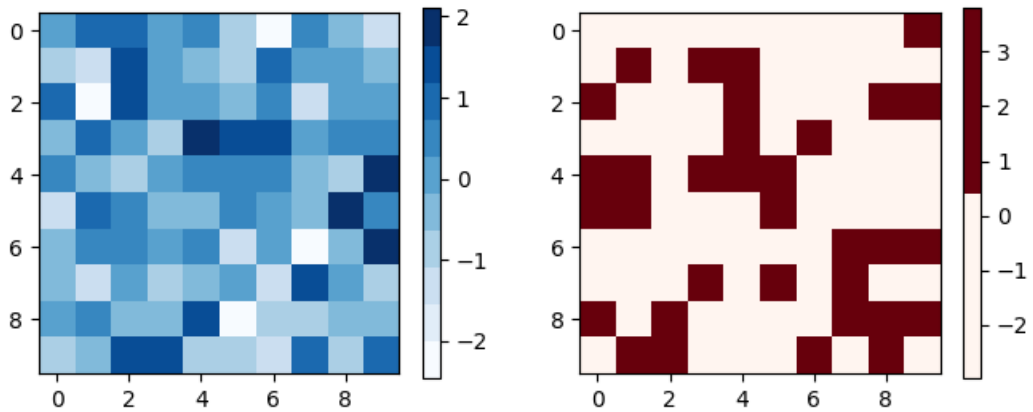
```
88 blues = cm.get_cmap("Blues", 10) # 10 levels
89 reds = cm.get_cmap("Reds", 2) # 2 levels
```

You can find a full list of different colour maps at: `https://matplotlib.org/stable/tutorials/colors/colormaps.html`

Colourmaps

Now that we have our new colour maps, we can pass it as an `cmap` argument when we create a plot.

```
90  x = np.random.randn(10, 10)
91  y = np.random.randn(10, 10)
92  fig, ax = plt.subplots(1, 2, figsize=(8, 3))
93  p1 = ax[0].imshow(x, cmap=blues)
94  p2 = ax[1].imshow(y, cmap=reds)
95  fig.colorbar(p1, ax=ax[0])
96  fig.colorbar(p2, ax=ax[1])
```

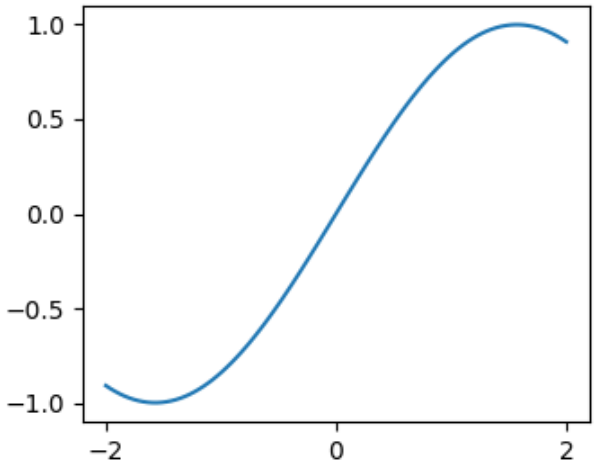


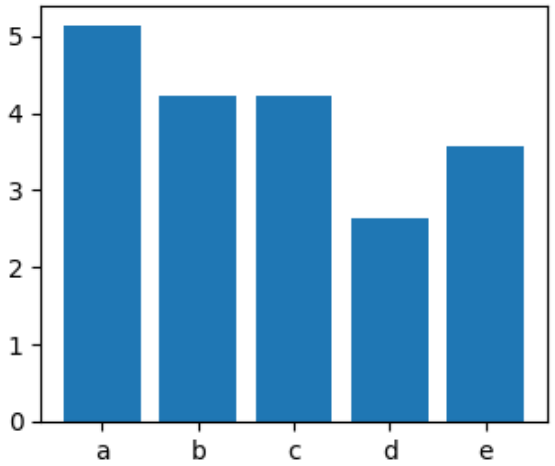
Ticks

If we want to customise the numbers along each axis, we use the `set_xticks` for the x-axis and `set_yticks` for the y-axis. These functions take the list of locations for each 'tick', and optionally a list of labels to use instead of the numbers.

```
97  x = np.linspace(-2, 2, 100)
98  y = np.sin(x)
99
100 bx = np.arange(2, 7)
101 by = np.random.uniform(2, 7, len(bx))
102
103 fig, ax = plt.subplots(1, 2, figsize=(8, 3))
104 ax[0].plot(x, y)
105 ax[0].set_xticks([-2, 0, 2])
106 ax[1].bar(bx, by)
107 ax[1].set_xticks(bx, ["a", "b", "c", "d", "e"])
```

Ticks



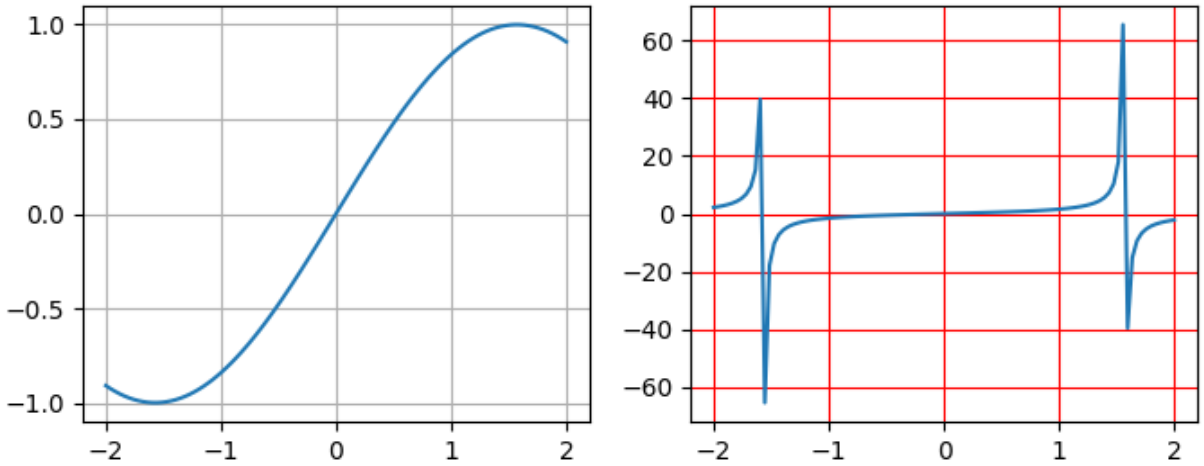


Grids

In all of the previous plots, the background has no grids, they are simply white. If we wanted to add grid lines to the plot we use the `.grid()` method. This function, by default, adds the major grid lines.

```
108 x = np.linspace(-2, 2, 100)
109 y = np.sin(x)
110 z = np.tan(x)
111 fig, ax = plt.subplots(1, 2, figsize=(8, 3))
112 ax[0].plot(x, y)
113 ax[0].grid()
114 ax[1].plot(x, z)
115 ax[1].grid(which="both", color="r")
```

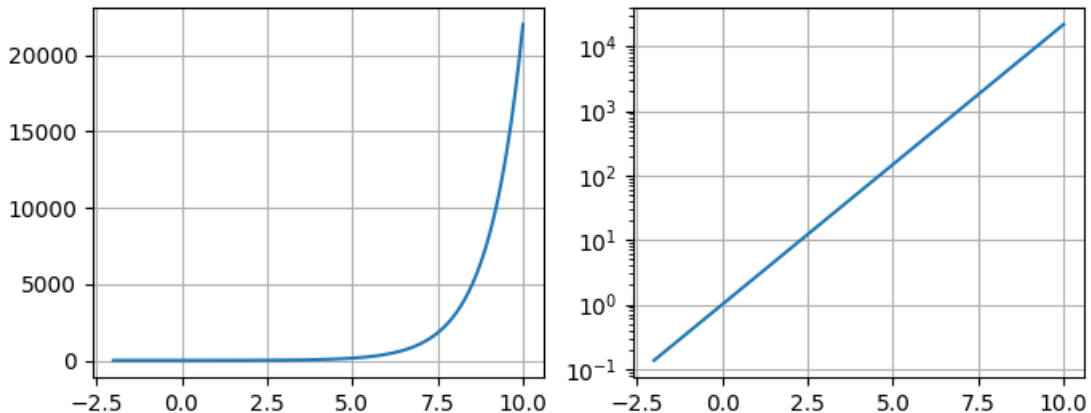
Grids



Scale

The default behaviour of matplotlib is to plot using a linear scale. In certain situations, we want view the plot using a different scale. For this we can use `set_yscale`.

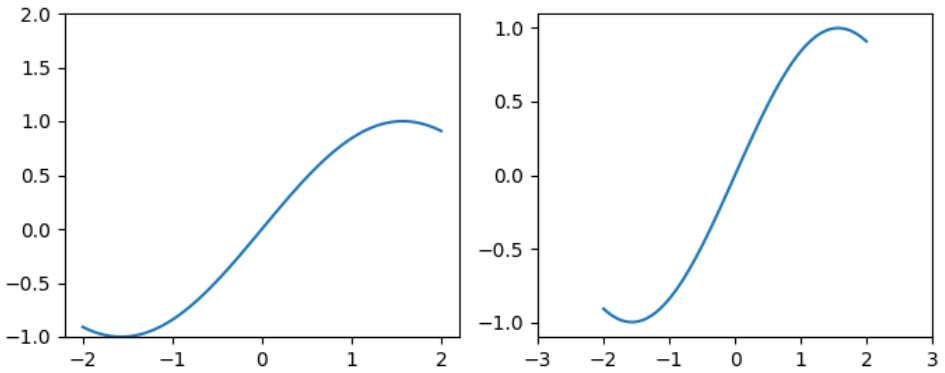
```
116  x = np.linspace(-2, 10, 100)
117  y = np.exp(x)
118  fig, ax = plt.subplots(1, 2, figsize=(8, 3))
119  ax[0].plot(x, y)
120  ax[0].grid()
121  ax[1].plot(x, y)
122  ax[1].set_yscale('log')
123  ax[1].grid()
```



Setting the plot limits

By default, matplotlib will calculate the minimum and maximum values of the data, and use those values to set the limits of the plot. Using `set_xlim` and `set_ylim` we can change this default behaviour.

```
124  x = np.linspace(-2, 2, 100)
125  y = np.sin(x)
126  fig, ax = plt.subplots(1, 2, figsize=(8,3))
127  ax[0].plot(x, y)
128  ax[0].set_ylim(-1, 2)
129  ax[1].plot(x, y)
130  ax[1].set_xlim(-3, 3)
```



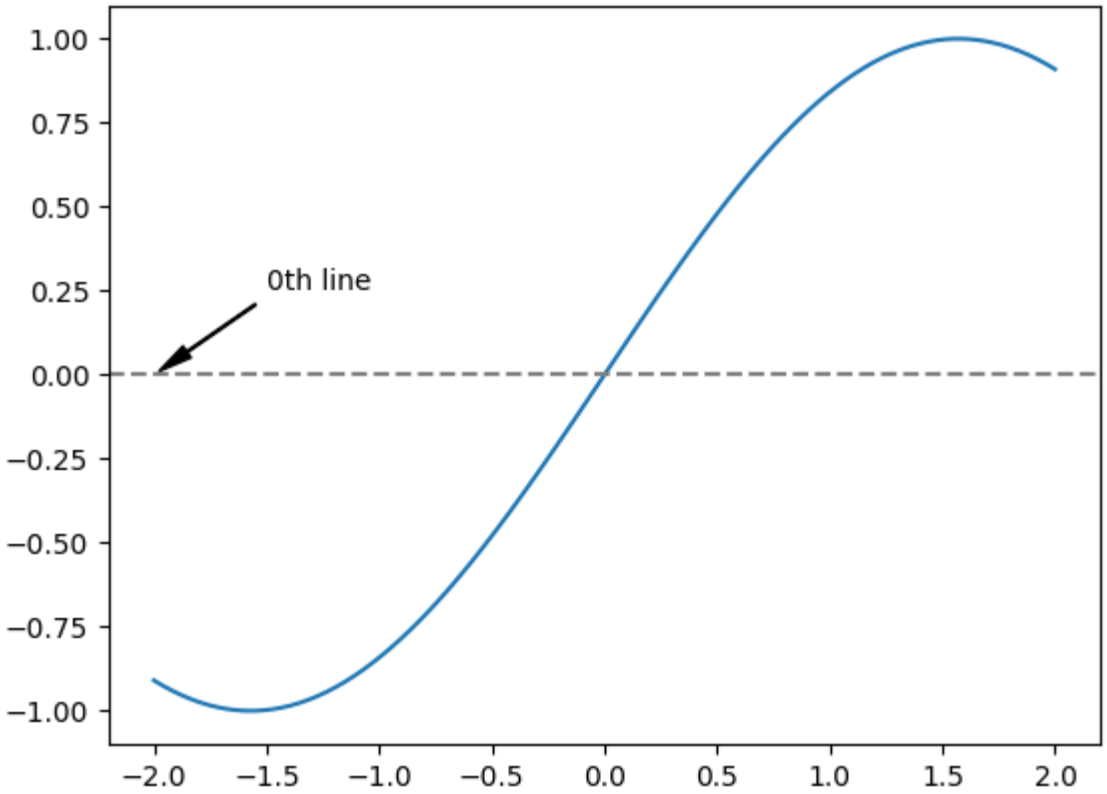
Annotations

We can annotate our plot in a number of way:

- .axhline – plot a horizontal line (axvline for vertical lines)/
- .annotate – add text to the plot at a certain position.

```
131 x = np.linspace(-2, 2, 100)
132 y = np.sin(x)
133 fig, ax = plt.subplots()
134 ax.plot(x, y)
135 ax.axhline(0, c='gray', ls='--')
136 ax.annotate("0th line", (-2, 0), xytext=(-1.5, 0.25),
137             arrowprops=dict(facecolor='black', shrink=0.05,
138                             width=0.5, headwidth=5.0))
```

Annotations

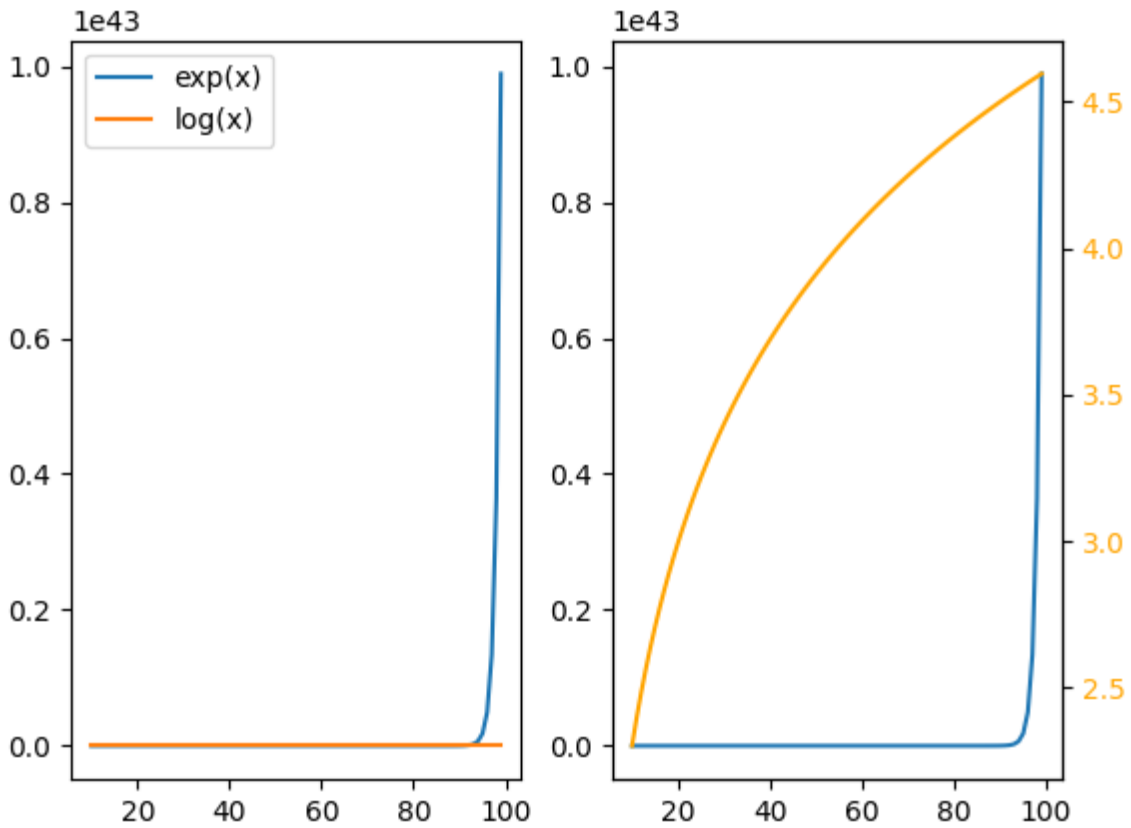


Creating a twin axes plot

Sometimes you will want to display multiple sub-plots on the same plot, but where each have a very different range in values. Instead of having a single y-axis, with `twinx()` we can create a two y-axis plot.

```
139  x = np.arange(10, 100)
140  y = np.exp(x)
141  z = np.log(x)
142
143  fig, ax = plt.subplots(1, 2)
144  ax[0].plot(x, y, label="exp(x)")
145  ax[0].plot(x, z, label="log(x)")
146  ax[0].legend()
147
148  ax2 = ax[1].twinx()
149  ax[1].plot(x, y)
150  ax2.plot(x, z, color="orange")
151  ax2.tick_params(axis="y", labelcolor="orange")
```

Creating a twin axes plot



Learn more

There are many many more types of plots you can create with matplotlib. I would recommend that you read the documentation to fully appreciate everything that it can visualise:

- Gallery – `https://matplotlib.org/stable/gallery/index.html`
- Plotting tutorials – `https://matplotlib.org/stable/tutorials/index.html`
- Basic plot types – `https://matplotlib.org/stable/plot_types/index.html`