tu08_re_MatPlotLib

February 21, 2023

1 Numpy and MatPlotLib

Data really come alive when plotted. Visualizing data allows us see trends in data that would otherwise be hard to uncover. While Numpy is a great package for storing and manipulating data, it has no plotting functionality. There is a second package, MatPlotLib, that was specifically written to plot Numpy data. (This separation of functions – data storage and manipulation on the one hand and data vizualization on the other – is intentional and is good programming practice.)

The first thing we'll do is import numpy and the main module of matplotlib, pyplot.

Just as numpy is conventionally imported as "np", matplotlib.pyplot is conventionally imported as "plt".

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.1 Making a basic matplotlib plot

Let's plot a single cycle of a sine wave. We'll first make an x-axis from 0 to 2*pi using `np.linspace()` (Somewhat ironically, **Py**thon does not have **pi**, but numpy does!)

[3]:
```
x = np.linspace(0, 2*np.pi, 100)
```

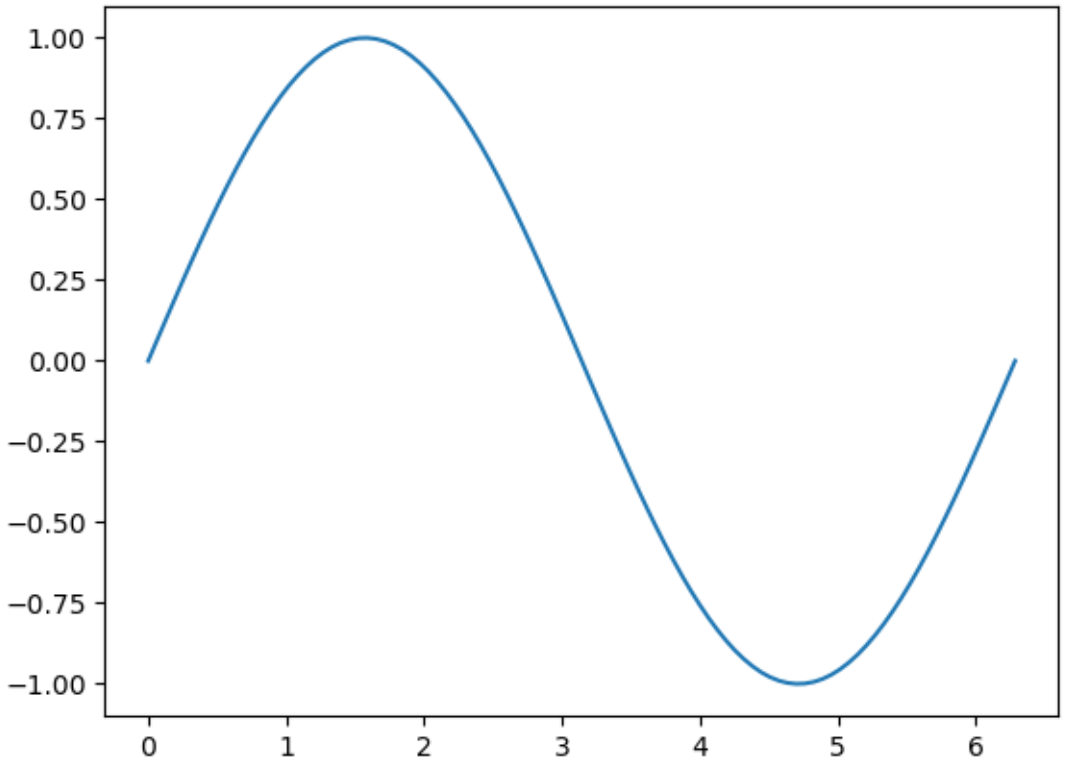
And then we'll compute the sine of x (numpy, being a numerical computing package, comes with all the goodies you would expect, such as trigonometry functions).

[4]:
```
y = np.sin(x)
```

Now we can plot this with `plt.plot()`.

[5]:
```
plt.plot(x, y)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x7f984a93c550>]
```

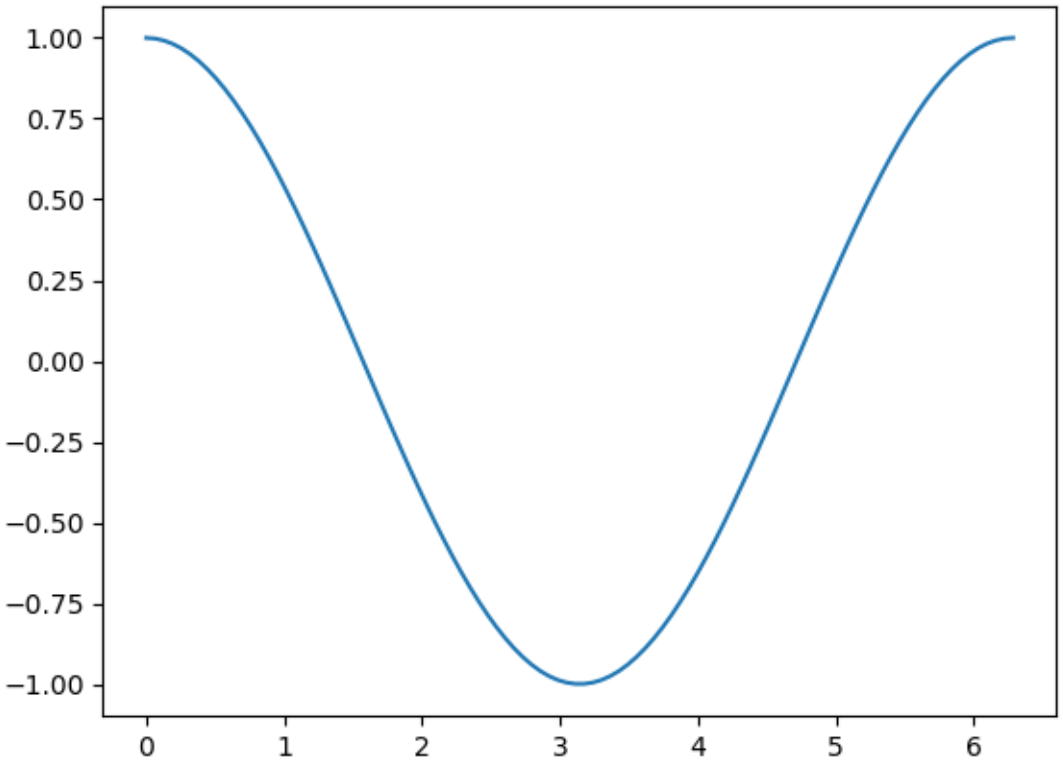


(Some people say "pea el tea" for `plt` but we pronounce it "plit" so it rhymes with "it" – so we can pronounce the whole thing as either "plit plot" or "plit dot plot" – we'll be saying these a lot!).

Use the code cell below to plot the cosine of x.

[6]:
```
y = np.cos(x)
plt.plot(x, y)
```

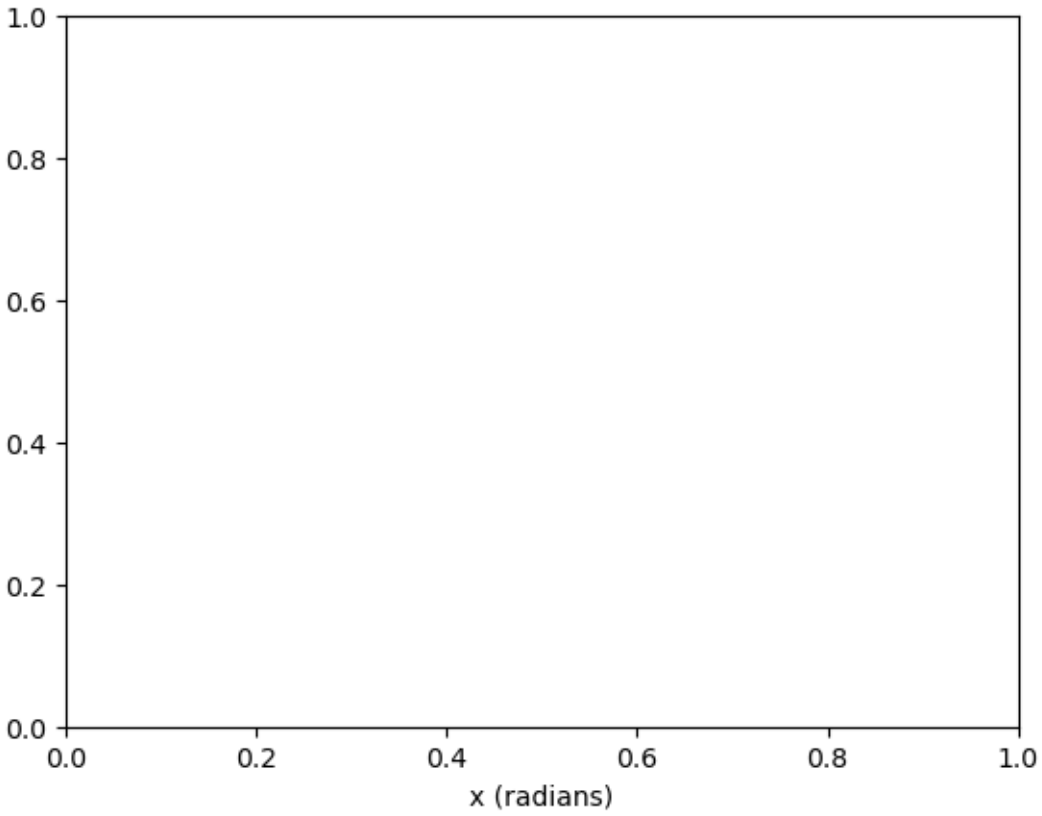
[6]:
```
[<matplotlib.lines.Line2D at 0x7f98789607c0>]
```



Even though we're just playing with plotting, we should get in the habit of adding some important things. Any plot that is ever going to have eyeballs on it that don't belong to present-you needs to have, at a minimum, axis labels. These are easy to add with `plt.xlabel()`, and `plt.ylabel()`.

[7]:
```
plt.xlabel('x (radians)')
```

[7]:
```
Text(0.5, 0, 'x (radians)')
```



Uh oh. What happened there?

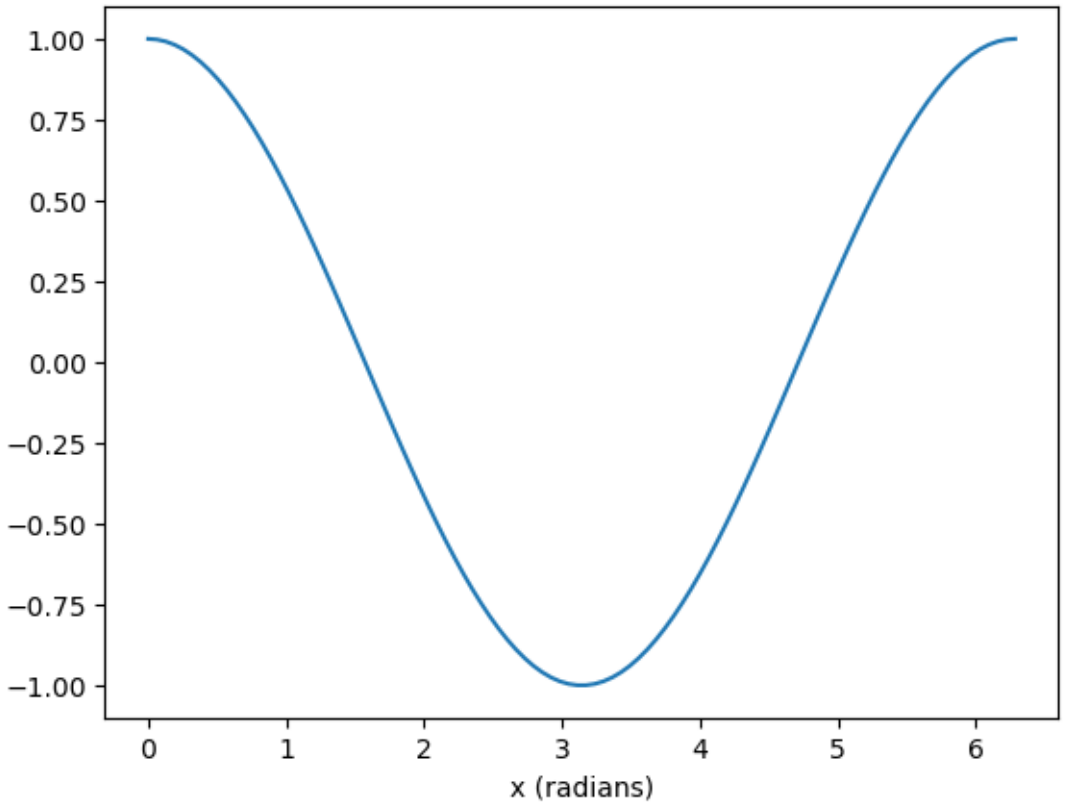
In Jupyter notebooks, all plotting happens in the current code cell. So when we asked `plt` to give us an x label, `plt` shrugged and thought "Um, okay, I guess you know what you are doing…" Since there was no plot already created on which to put an x label, `plt` went ahead and created one for us (how nice!).

Notice that `plt.plot` did the same thing above. When we requested a graph of the sine or cosine, it went ahead and created a plotting area with an x and y axis for us (an `axes` in matplotlib lingo) and then graphed our function onto it.

So what we need to do is just recycle our code from above and put everything into one code cell.

[8]:
```
plt.plot(x, y)
plt.xlabel('x (radians)')
```

[8]:
```
Text(0.5, 0, 'x (radians)')
```

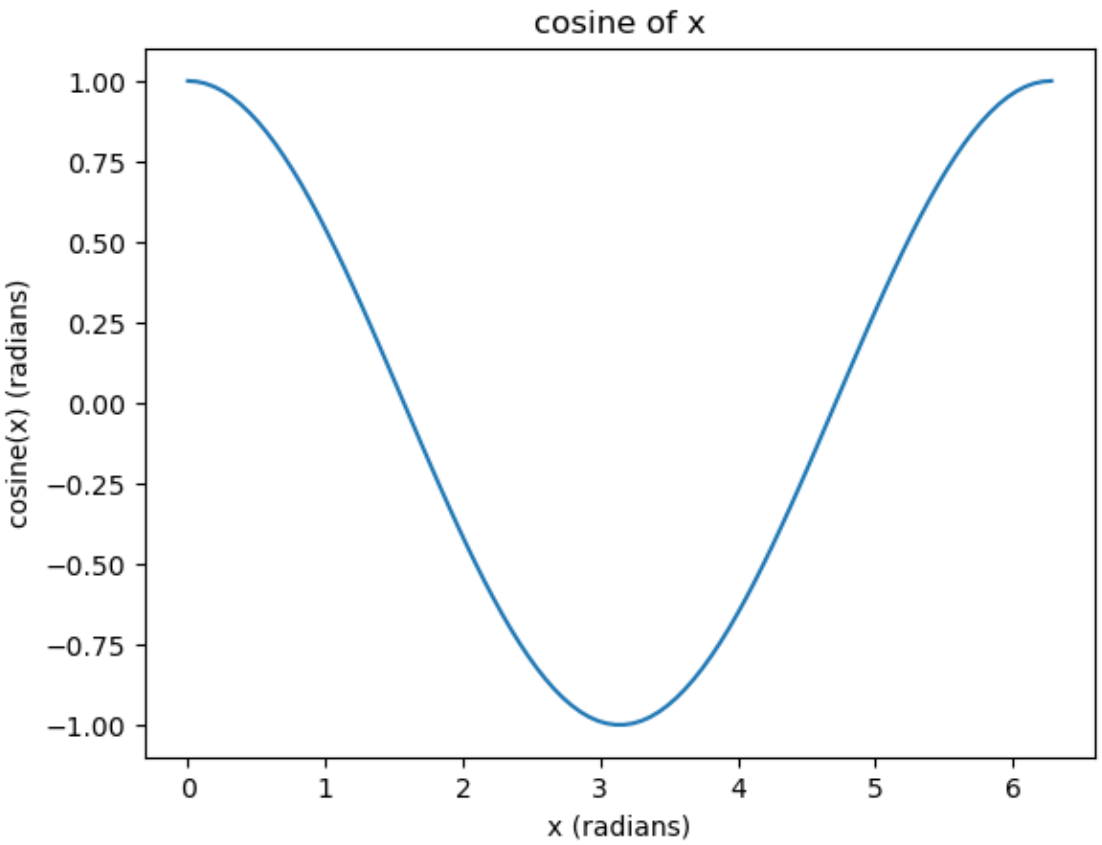


In the cell below, make the same plot with the addition of a y label and a title.

(If we add an x label with `plt.xlabel()`, we'd add a title with… what?)

[9]:
```
plt.plot(x, y)
plt.xlabel('x (radians)')
plt.ylabel('cosine(x) (radians)')
plt.title('cosine of x')
```

[9]:
```
Text(0.5, 1.0, 'cosine of x')
```

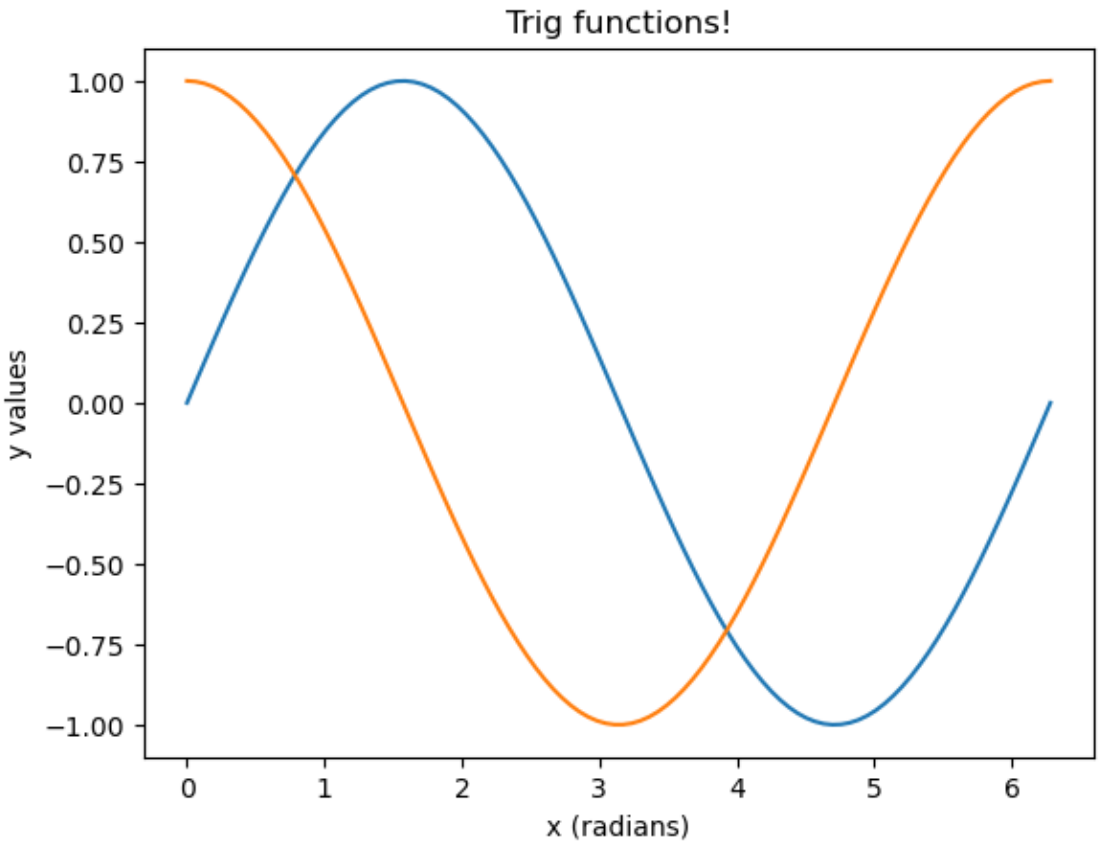


We can graph more that one thing at a time just by calling `plt.plot()` multiple times. Let's put the code for everything in one cell (which is what we'll be generally doing to make "real" figures).

[10]:
```
# plot the sine and cosine together

# make the x axis
x = np.linspace(0, 2*np.pi, 100)

# compute the sin and cosine
y = np.sin(x)
y2 = np.cos(x)

# make the plots
plt.plot(x, y)
plt.plot(x,y2)

# annotate
plt.xlabel('x (radians)')
plt.ylabel('y values')
```

```
plt.title('Trig functions!')
```

[10]:
```
Text(0.5, 1.0, 'Trig functions!')
```



Great! Now, there is only one thing missing from this figure to make it complete. Can you figure out what one other annotation we should really have here?

If you use trig a lot (like we do), you can tell at a glance which graph is the sine and which is the cosine but, if you don't, you probably can't. So, whenever we graph more than one function or plot more than one set of data, we consider having a *legend*.

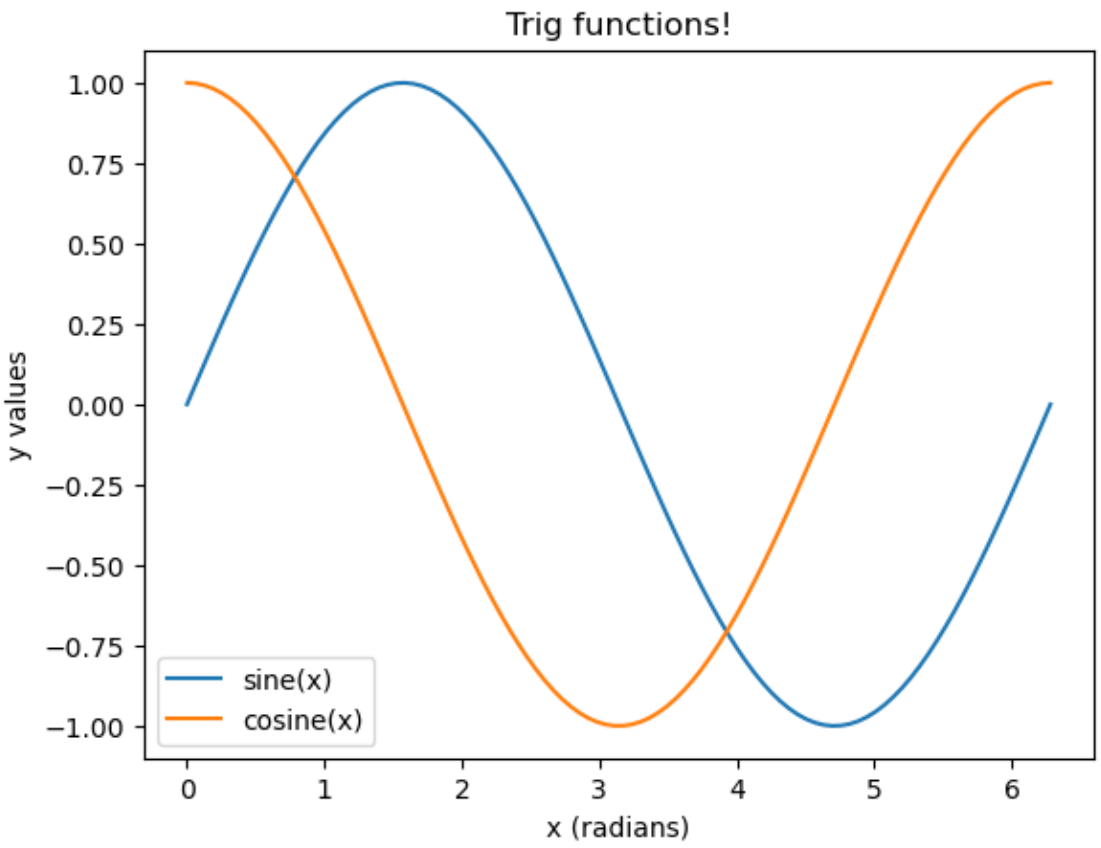
Copy the code from above and paste it in the code cell below, then add a `plt.legend()` at the very bottom (note that, strictly speaking, we only needed to copy the code starting at `# make the plots` to make new plots, but we might as well keep everything together as a tidy little program).

Run your code.

[11]:
```
# make the plots
plt.plot(x, y)
plt.plot(x,y2)

# annotate
```

```
plt.xlabel('x (radians)')
plt.ylabel('y values')
plt.title('Trig functions!')

# legend
plt.legend(['sine(x)','cosine(x)'])
```

[11]: <matplotlib.legend.Legend at 0x7f984a9ada00>



What happened? Some complaint about "artists"...

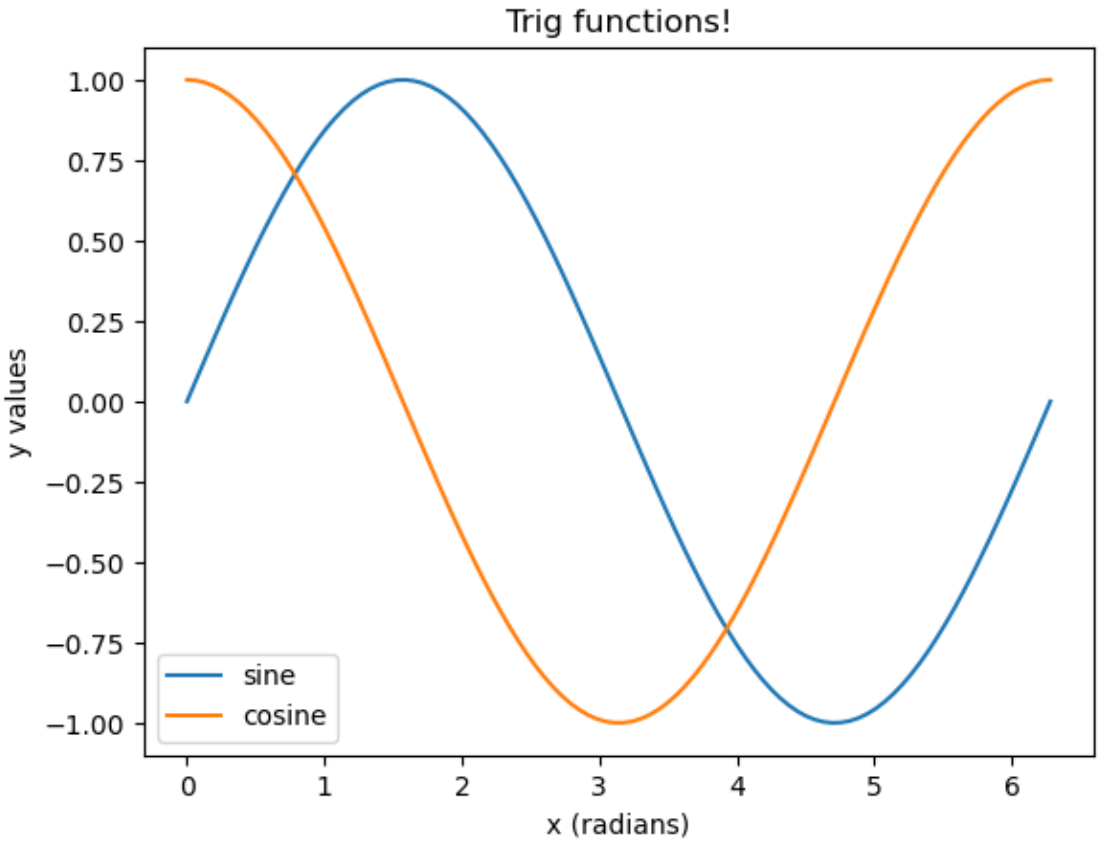
In matplotlib, everything is ultimately an "artist", which just means "a thing that knows how to draw itself". Here, `legend()` is complaining that the "artists" created by `plt.plot()` don't have labels, so it doesn't know how to make a legend. Yes, the error message could have been helpful...

But we can fix this by giving our sine and cosine "artists" labels for their work:

[12]:
```
# plot the sine and cosine together

# make the x axis
x = np.linspace(0, 2*np.pi, 100)
```

```
# compute the sin and cosine
y = np.sin(x)
y2 = np.cos(x)

# make the plots
plt.plot(x, y, label = 'sine')
plt.plot(x,y2, label = 'cosine')

# annotate
plt.xlabel('x (radians)')
plt.ylabel('y values')
plt.title('Trig functions!')
plt.legend()
```

[12]: <matplotlib.legend.Legend at 0x7f98789c4fa0>



And a legend is born.

Now we have a figure that is complete, in that it has

- x and y axis labels
- a title
- a legend so we can tell what is what

1.2 Modifying line and point appearance.

The matplotlib package makes it fairly easy to alter the appearance of lines and points.

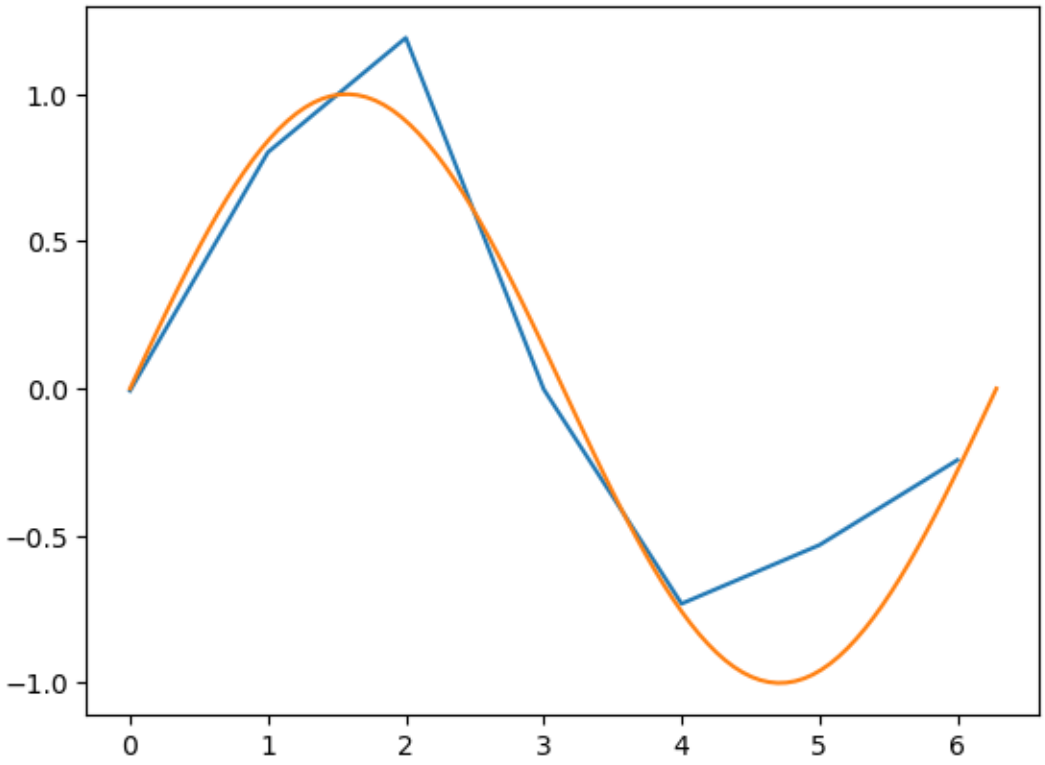
First, let's make a basic figure with some simulated data and a candidate function for the data.

[13]:
```
# make some simulated data
dat_x = [0, 1, 2, 3, 4, 5, 6]
dat_y = np.sin(dat_x) + 0.25*np.random.randn(len(dat_x))

# make the fitting function
x = np.linspace(0, 2*np.pi, 100)
# compute the sine
y = np.sin(x)
```

Now let's plot these, letting matplotlib pick the default appearance.

[14]:
```
plt.plot(dat_x, dat_y)
plt.plot(x, y)
```

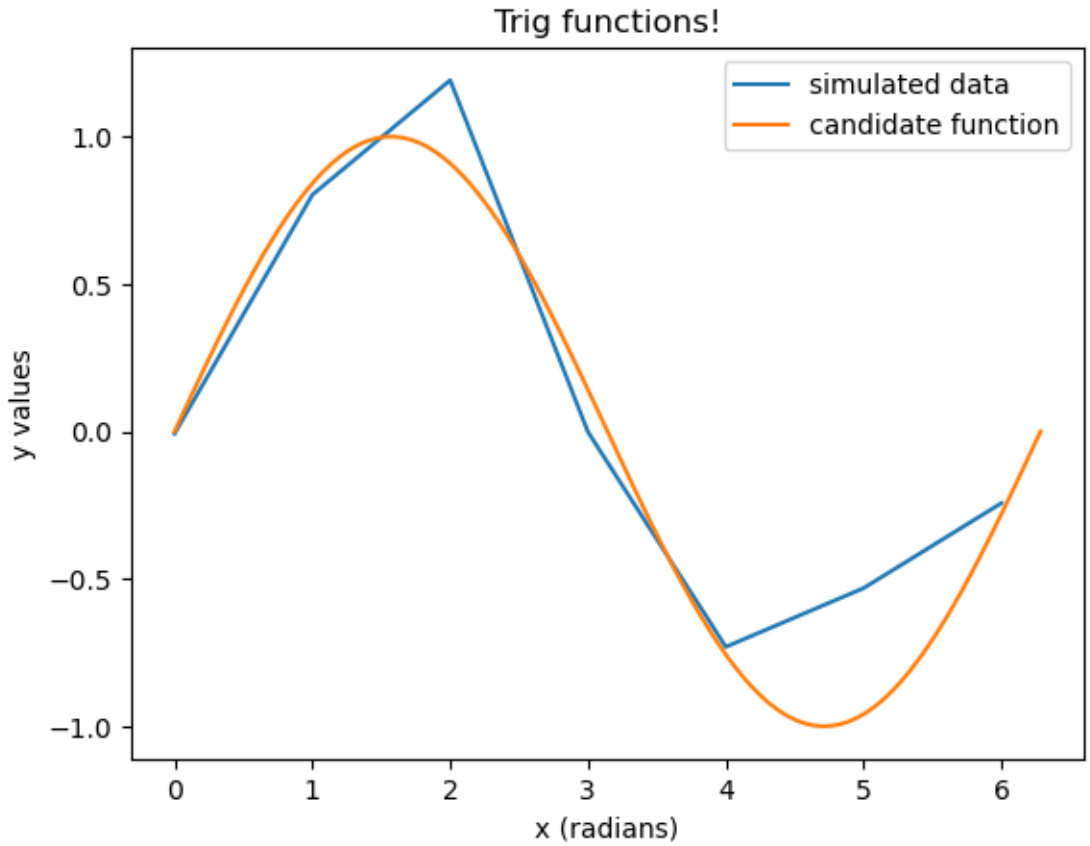
[14]:
```
[<matplotlib.lines.Line2D at 0x7f98585aed90>]
```



There are some key elements that this figure are missing, as covered above, so use the code cell below to make a figure with those added.

[15]:
```
# plots
plt.plot(dat_x, dat_y, label = 'simulated data')
plt.plot(x, y, label = 'candidate function')

# annotate
plt.xlabel('x (radians)')
plt.ylabel('y values')
plt.title('Trig functions!')
plt.legend()
```

[15]:
```
<matplotlib.legend.Legend at 0x7f98683ebaf0>
```

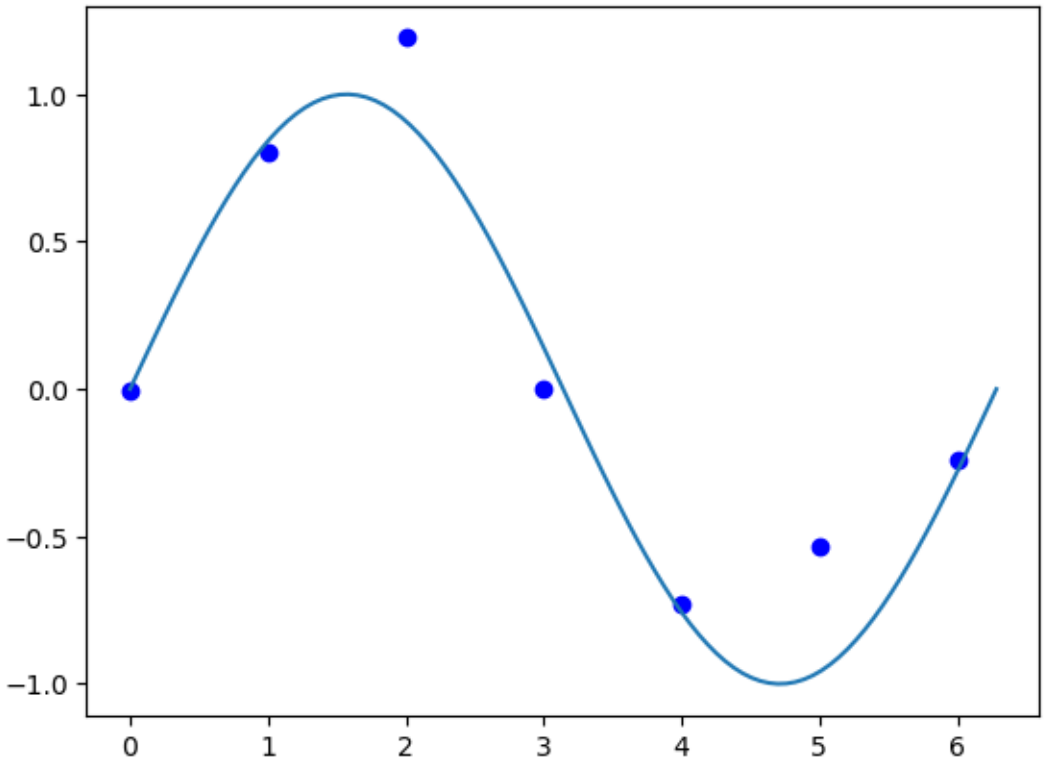


So far so good, but we can do better. The data are sparse – they're only 7 points or so – and they are a bit noisy. So let's let them be points, and make them blue.

Just run the code and we can worry about labels and such in a bit.

[16]:
```
plt.plot(dat_x, dat_y, 'bo')
plt.plot(x, y)
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7f98686789a0>]
```



This is much better. We have individual data points that almost say out loud "Here's a small sample of data" and a continuous line that says "Here's a smooth function that might describe the data pretty well."

The `'ob'` inside `plt.plot()` is a ***format string*** for plots. This one says "I want my data plotted as blue (the 'b') circles (the 'o').

The general format for these format strings is `[color][marker][line]` but any can be omitted. Here are some common ones:

- colors: `b`, `g`, `r`, `c`, `m`, `y`, `k`
- markers: `.` , `o`, `+`, `x`, `*`, `D`
- lines: `-` , `--` , `-.` , `:`

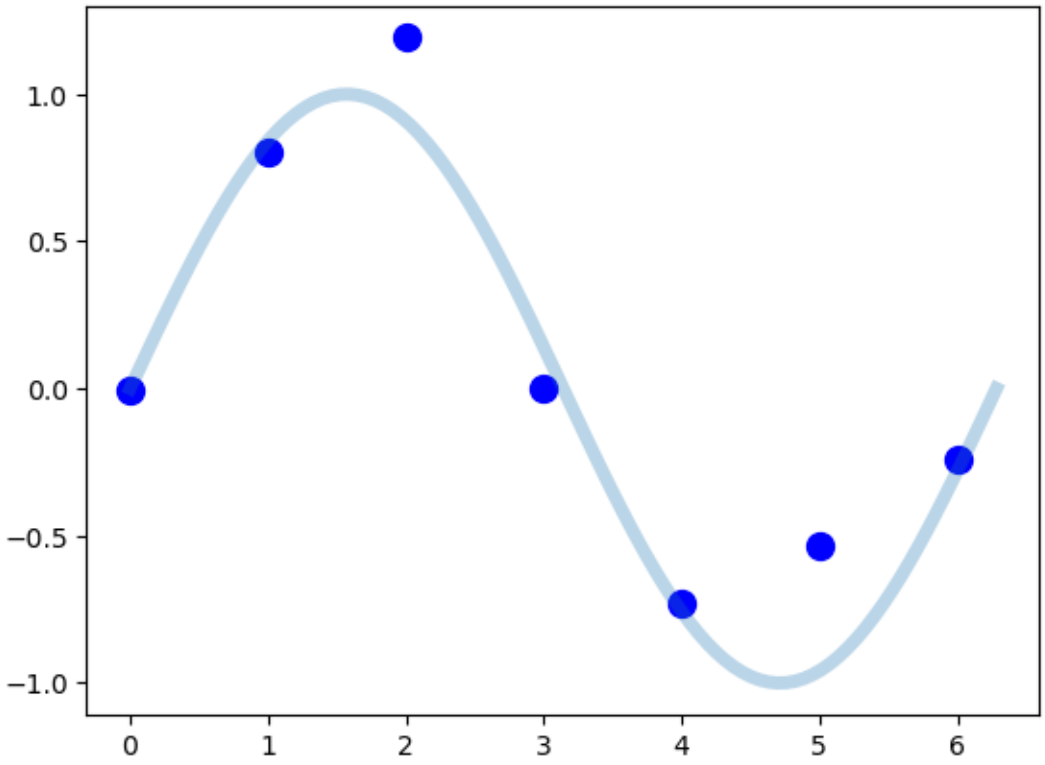
Here's a secret: you can generally put the elements of the format string in any order you want.

One less thing to remember!

You can also play around with things like the size of the markers, width of the lines, and transparency ("alpha" in computer graphics lingo).

[17]:
```
plt.plot(dat_x, dat_y, 'bo', markersize = 10)
plt.plot(x, y, linewidth = 5, alpha = 0.3)
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f984ac16250>]
```

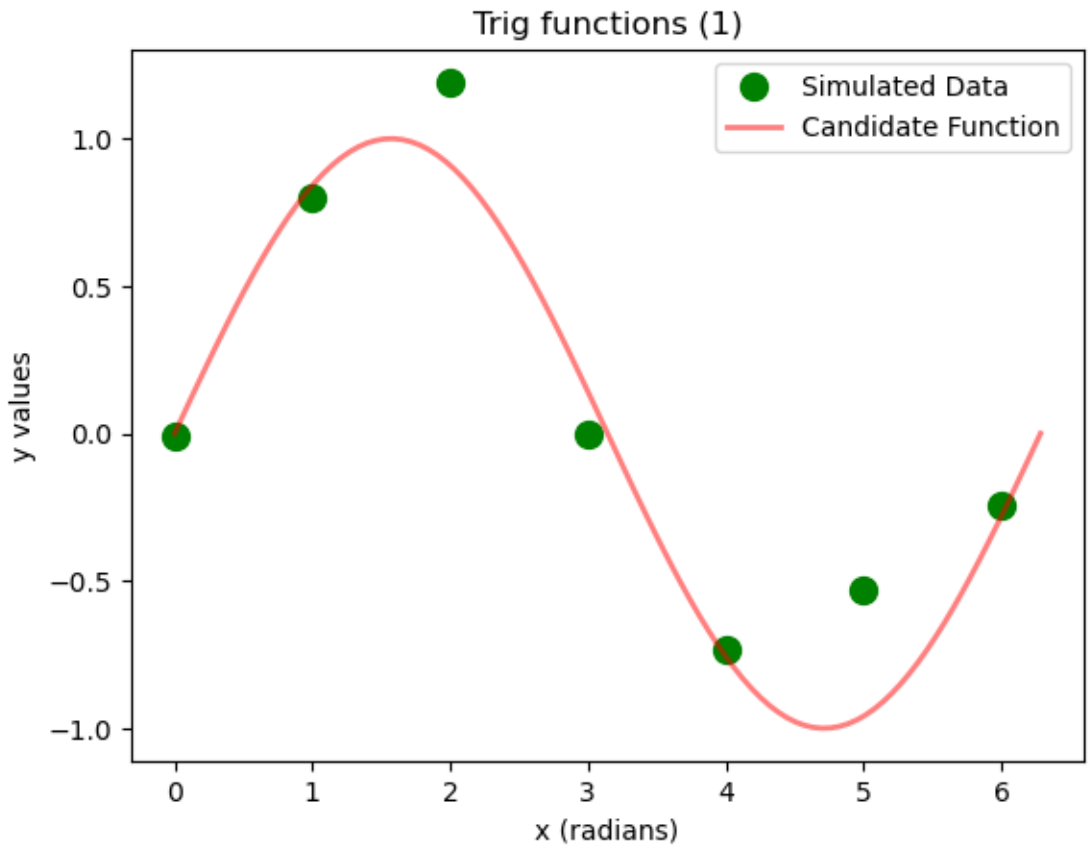


You can find more information on what markers, lines, and colors are available and other tips and tricks here.

Use the code cells below to make three plots with different markers, lines, etc. Make sure to add axis labels and legends.

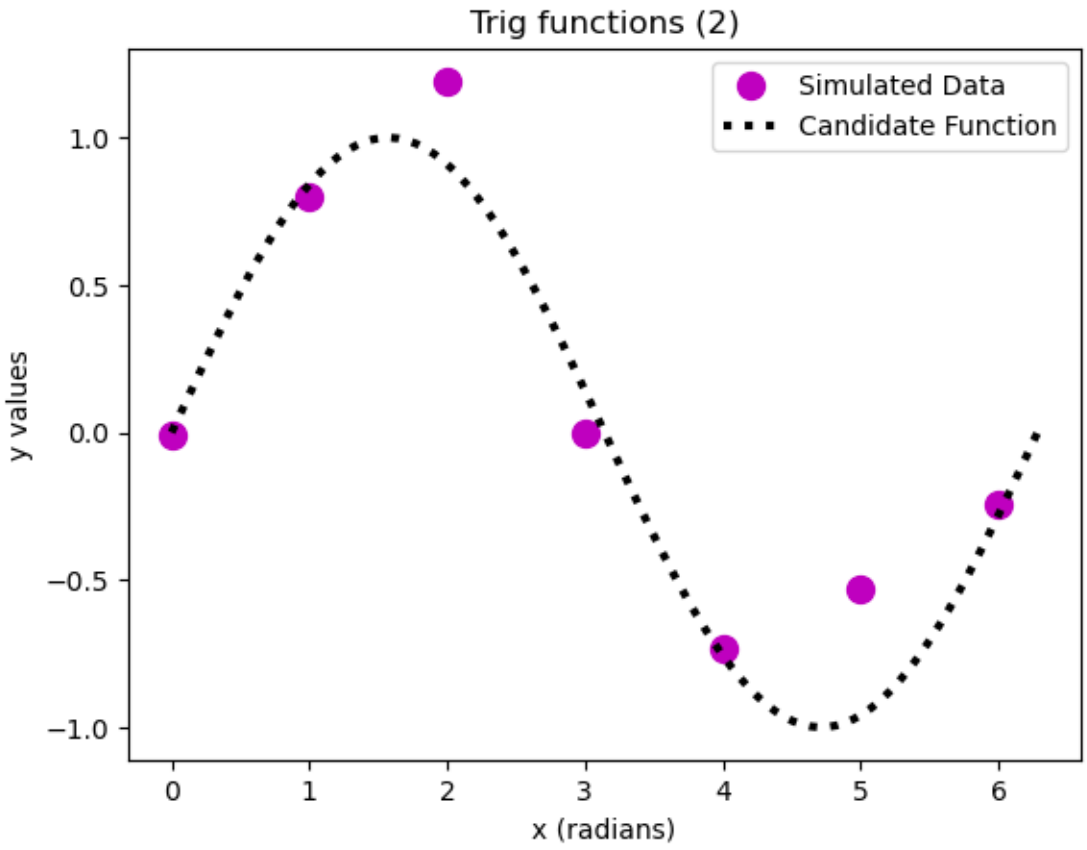
[18]:
```
plt.plot(dat_x, dat_y, 'g.', markersize = 20, label = 'Simulated Data')
plt.plot(x, y, 'r', linewidth = 2, alpha = 0.5, label = 'Candidate Function')
plt.xlabel('x (radians)')
plt.ylabel('y values')
plt.title('Trig functions (1)')
plt.legend()
```

[18]: <matplotlib.legend.Legend at 0x7f985861a580>



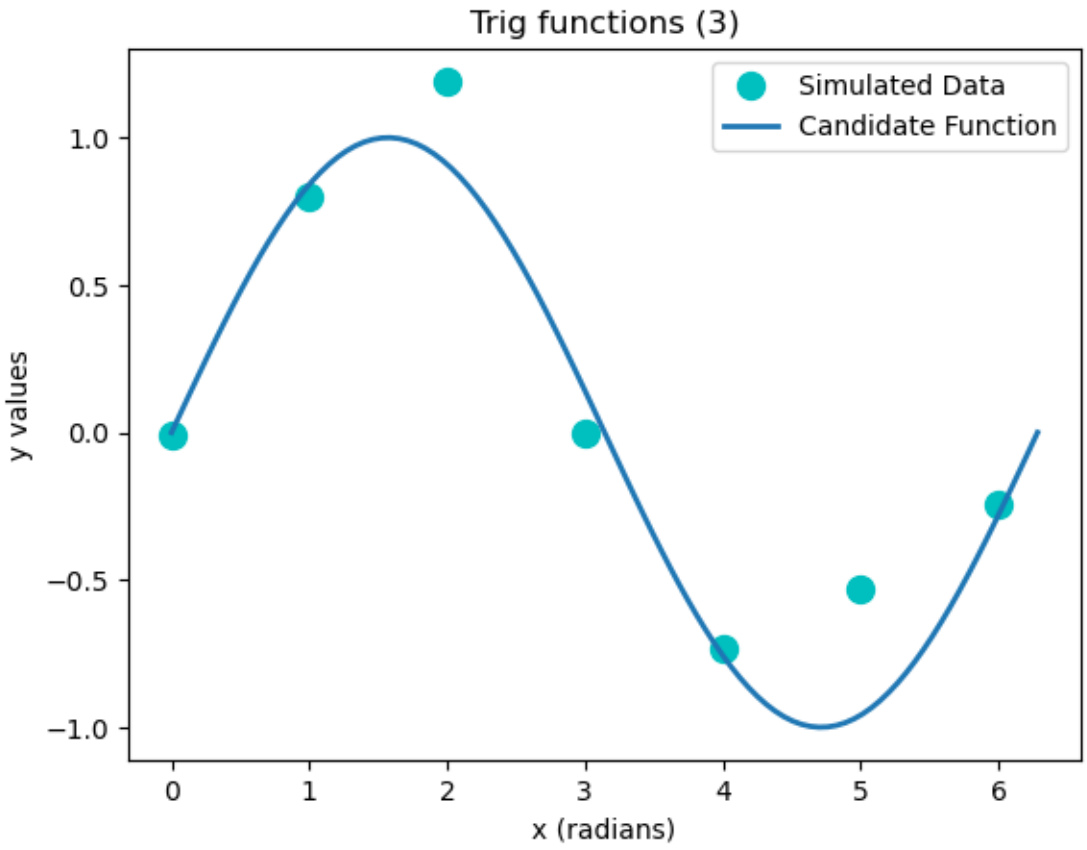
[19]:
```
plt.plot(dat_x, dat_y, 'mo', markersize = 10, label = 'Simulated Data')
plt.plot(x, y,'k:', linewidth = 3, label = 'Candidate Function')
plt.xlabel('x (radians)')
plt.ylabel('y values')
plt.title('Trig functions (2)')
plt.legend()
```

[19]: <matplotlib.legend.Legend at 0x7f9858607d30>



[20]:
```
plt.plot(dat_x, dat_y, 'co', markersize = 10, label = 'Simulated Data')
plt.plot(x, y, linewidth = 2, label = 'Candidate Function')
plt.xlabel('x (radians)')
plt.ylabel('y values')
plt.title('Trig functions (3)')
plt.legend()
```

[20]: <matplotlib.legend.Legend at 0x7f9878bd7d90>



1.3 Some basic plot types

Some basic plot types in matplotlib are

- *histograms* – the distributions of a single variable
- *line plots* – two variables with the x axis in numerical order
- *scatterplots* – two variables with potentially more mapped to color and or size

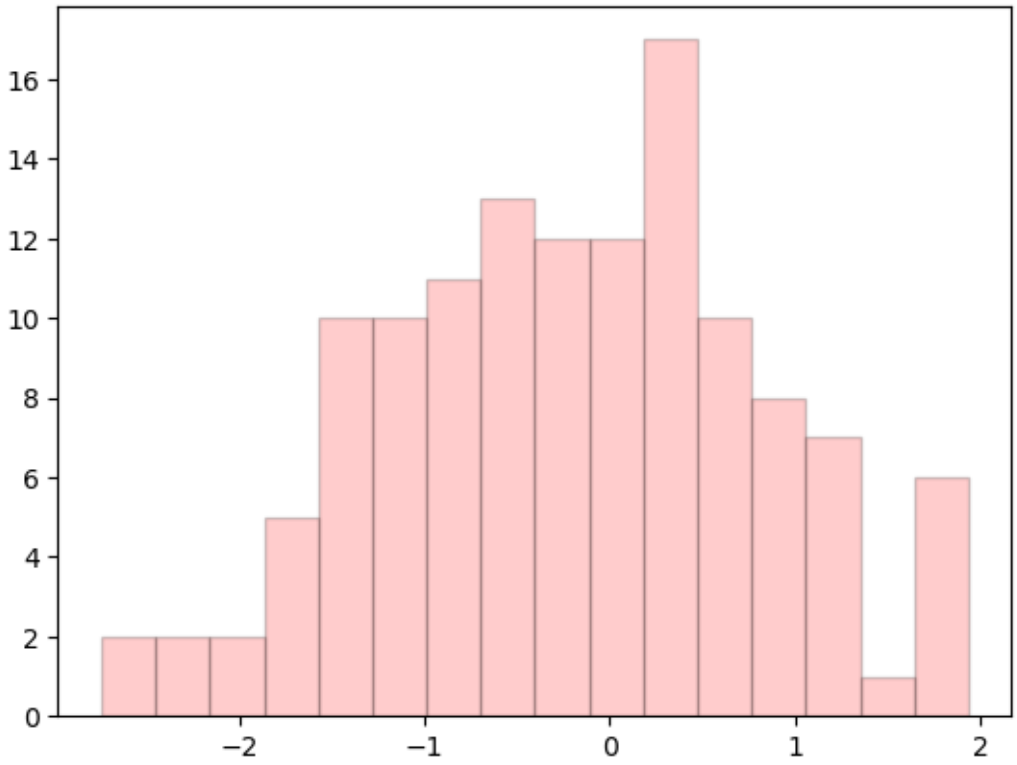
1.3.1 histograms

Here's an example of plotting a histogram.

[21]:
```
data = np.random.randn(128)
plt.hist(data, bins = 16, color = 'r', edgecolor = 'k', alpha = 0.2)
```

[21]:
```
(array([ 2.,  2.,  2.,  5., 10., 10., 11., 13., 12., 12., 17., 10.,  8.,
         7.,  1.,  6.]),
 array([-2.74438867, -2.45183927, -2.15928987, -1.86674048, -1.57419108,
        -1.28164168, -0.98909229, -0.69654289, -0.40399349, -0.1114441 ,
```

```
        0.1811053 ,  0.4736547 ,  0.7662041 ,  1.05875349,  1.35130289,
        1.64385229,  1.93640168]),
 <BarContainer object of 16 artists>)
```



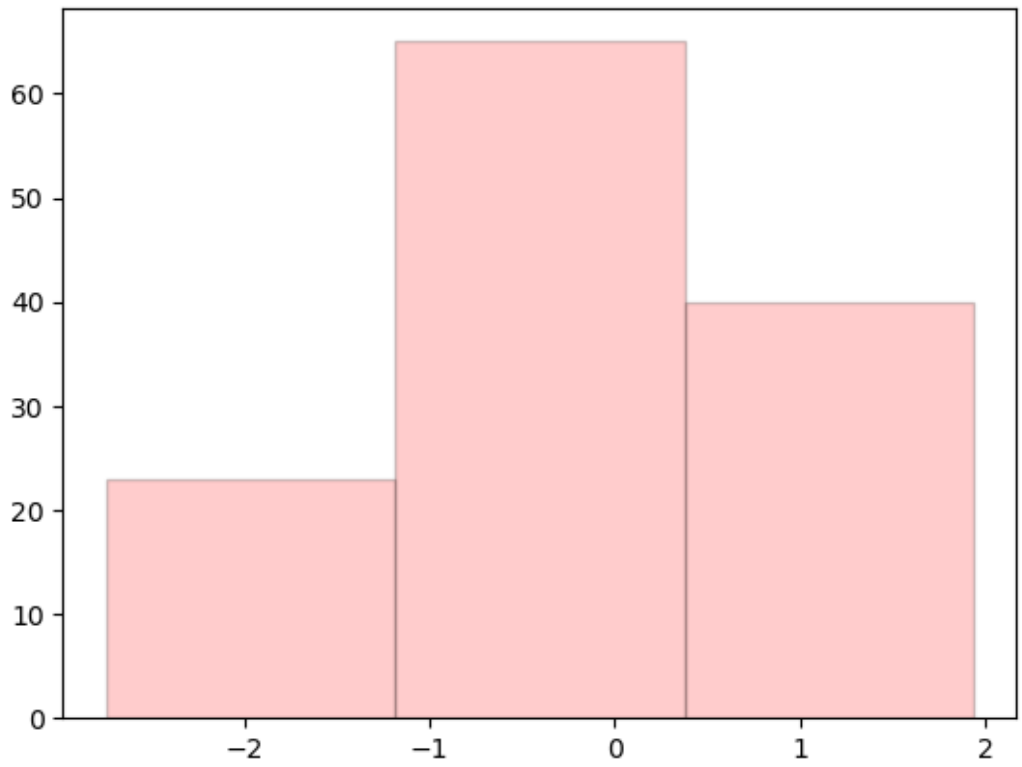
Sometimes, a plot command outputs some annoying text on top of our plot. We can suppress it with a semicolon at the end of the command. Go back up to the above cell and re-run it with the semicolon after the `plt.hist()` command.

Ah! So much better!

Play around with the `bins` parameter for the above plot. Use the cells below to make histograms that you think have 1) to few bins, 2) to many bins and, 3) a Goldilocks number of bins.

[22]:
```
plt.hist(data, bins = 3, color = 'r', edgecolor = 'k', alpha = 0.2)
```

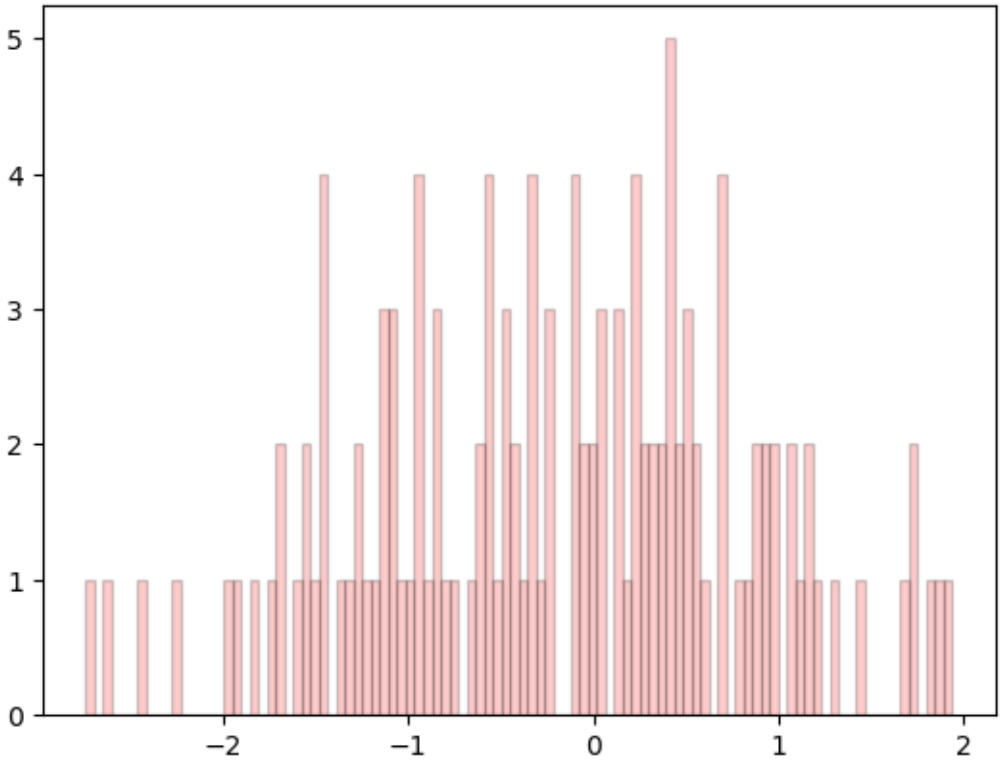
[22]:
```
(array([23., 65., 40.]),
 array([-2.74438867, -1.18412522,  0.37613823,  1.93640168]),
 <BarContainer object of 3 artists>)
```



[23]:
```
plt.hist(data, bins = 100, color = 'r', edgecolor = 'k', alpha = 0.2)
```

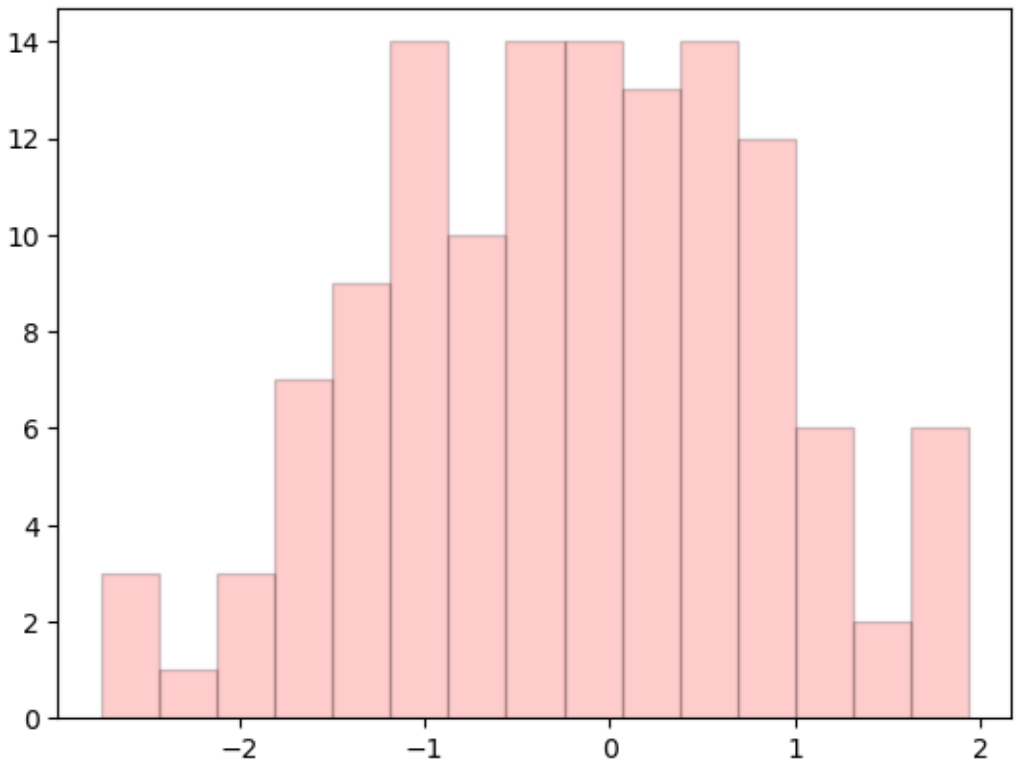
[23]:
```
(array([1., 0., 1., 0., 0., 0., 1., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.,
        1., 0., 1., 0., 1., 2., 0., 1., 2., 1., 4., 0., 1., 1., 2., 1., 1.,
        3., 3., 1., 1., 4., 1., 3., 1., 1., 0., 1., 2., 4., 1., 3., 2., 1.,
        4., 1., 3., 0., 0., 4., 2., 2., 3., 0., 3., 1., 4., 2., 2., 2., 5.,
        2., 3., 2., 1., 0., 4., 0., 1., 1., 2., 2., 2., 0., 2., 1., 2., 1.,
        0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 2., 0., 1., 1., 1.]),
 array([-2.74438867, -2.69758076, -2.65077286, -2.60396496, -2.55715705,
        -2.51034915, -2.46354125, -2.41673334, -2.36992544, -2.32311754,
        -2.27630963, -2.22950173, -2.18269383, -2.13588592, -2.08907802,
        -2.04227012, -1.99546221, -1.94865431, -1.90184641, -1.8550385 ,
        -1.8082306 , -1.76142269, -1.71461479, -1.66780689, -1.62099898,
        -1.57419108, -1.52738318, -1.48057527, -1.43376737, -1.38695947,
        -1.34015156, -1.29334366, -1.24653576, -1.19972785, -1.15291995,
        -1.10611205, -1.05930414, -1.01249624, -0.96568833, -0.91888043,
        -0.87207253, -0.82526462, -0.77845672, -0.73164882, -0.68484091,
        -0.63803301, -0.59122511, -0.5444172 , -0.4976093 , -0.4508014 ,
        -0.40399349, -0.35718559, -0.31037769, -0.26356978, -0.21676188,
        -0.16995397, -0.12314607, -0.07633817, -0.02953026,  0.01727764,
         0.06408554,  0.11089345,  0.15770135,  0.20450925,  0.25131716,
         0.29812506,  0.34493296,  0.39174087,  0.43854877,  0.48535667,
```

```
        0.53216458,  0.57897248,  0.62578038,  0.67258829,  0.71939619,
        0.7662041 ,  0.813012  ,  0.8598199 ,  0.90662781,  0.95343571,
        1.00024361,  1.04705152,  1.09385942,  1.14066732,  1.18747523,
        1.23428313,  1.28109103,  1.32789894,  1.37470684,  1.42151474,
        1.46832265,  1.51513055,  1.56193846,  1.60874636,  1.65555426,
        1.70236217,  1.74917007,  1.79597797,  1.84278588,  1.88959378,
        1.93640168]),
 <BarContainer object of 100 artists>)
```



[24]:
```
plt.hist(data, bins = 15, color = 'r', edgecolor = 'k', alpha = 0.2)
```

[24]:
```
(array([ 3.,  1.,  3.,  7.,  9., 14., 10., 14., 14., 13., 14., 12.,  6.,
         2.,  6.]),
 array([-2.74438867, -2.43233598, -2.12028329, -1.8082306 , -1.49617791,
        -1.18412522, -0.87207253, -0.56001984, -0.24796715,  0.06408554,
         0.37613823,  0.68819092,  1.00024361,  1.3122963 ,  1.62434899,
         1.93640168]),
 <BarContainer object of 15 artists>)
```



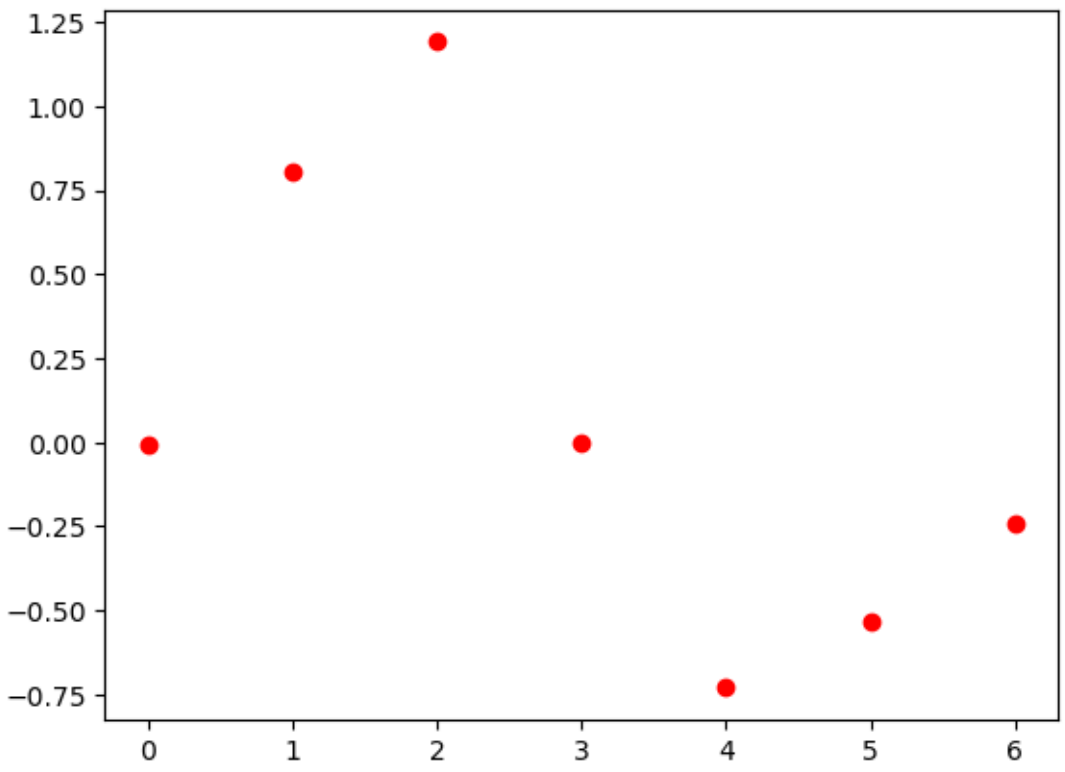
1.3.2 "plots" (line plots)

We've already played with line plots a bit – these are generated by `plt.plot()`.

The plot below (after you run the code cell) is technically a "line plot". It's so named because the data points *could* be connected with an interpolating line, even if they're not at the moment.

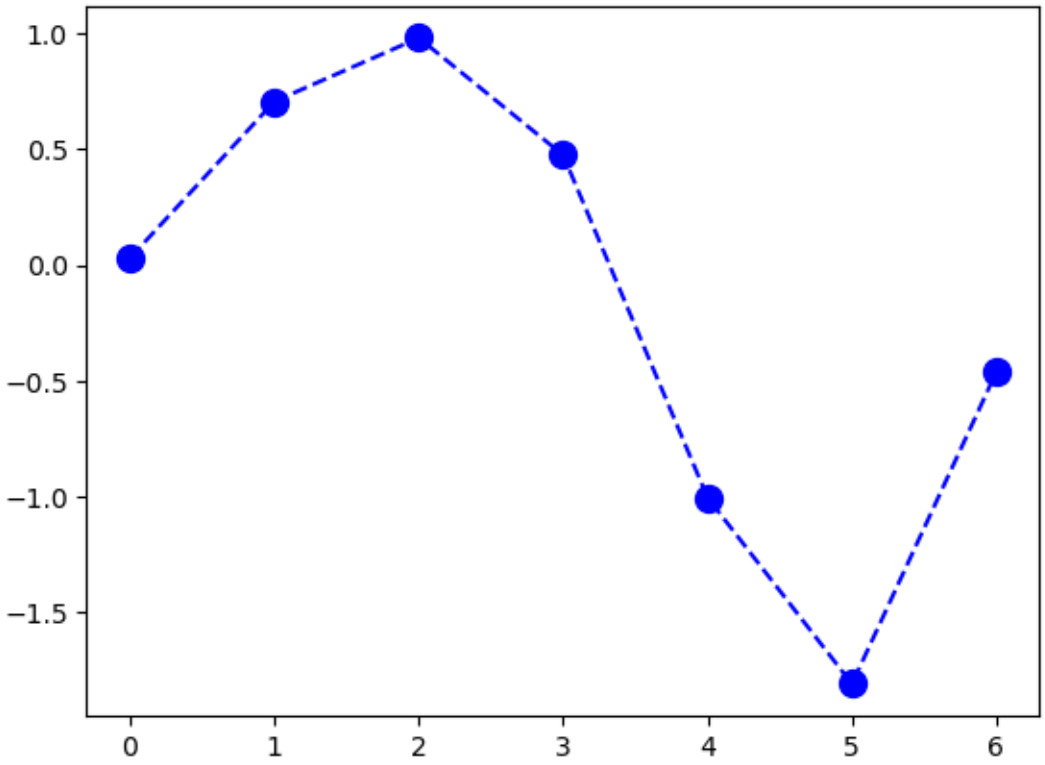
[25]:
```
plt.plot(dat_x, dat_y, 'ro')
```

[25]:
```
[<matplotlib.lines.Line2D at 0x7f987937b400>]
```



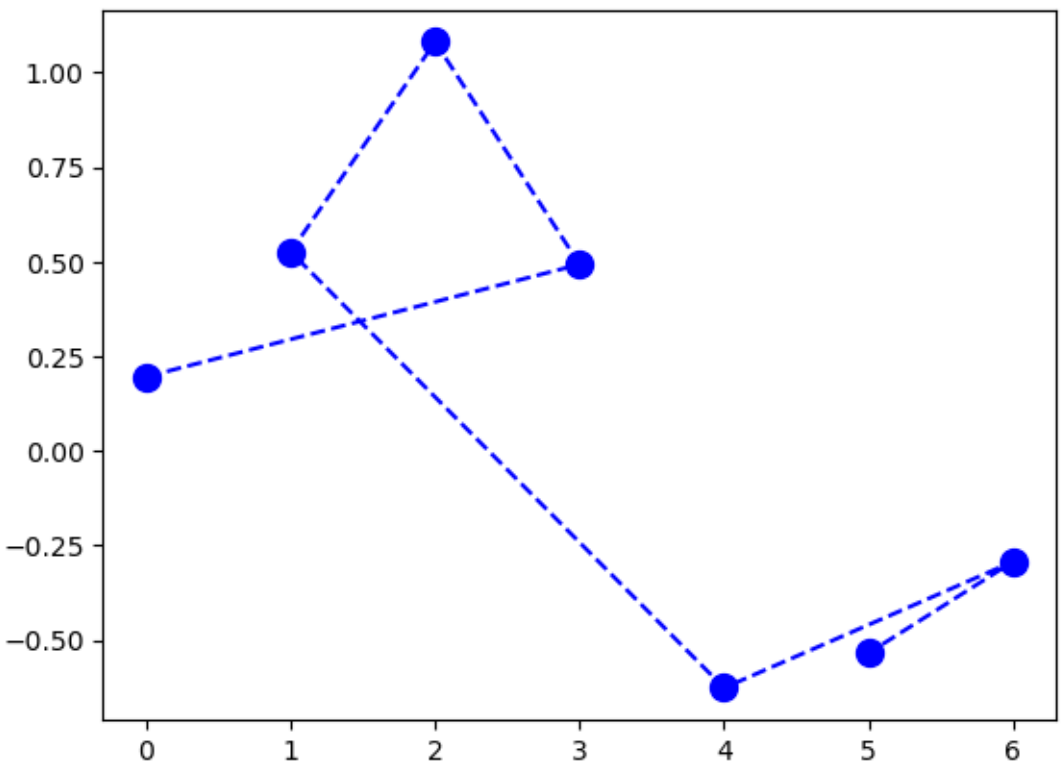
Like this.

[26]:
```
dat_x = [0, 1, 2, 3, 4, 5, 6] # same as above
dat_y = np.sin(dat_x) + 0.25*np.random.randn(len(dat_x))

plt.plot(dat_x, dat_y, 'bo--', markersize = 10);
```



But now let's see what happens when the x values are not ordered.

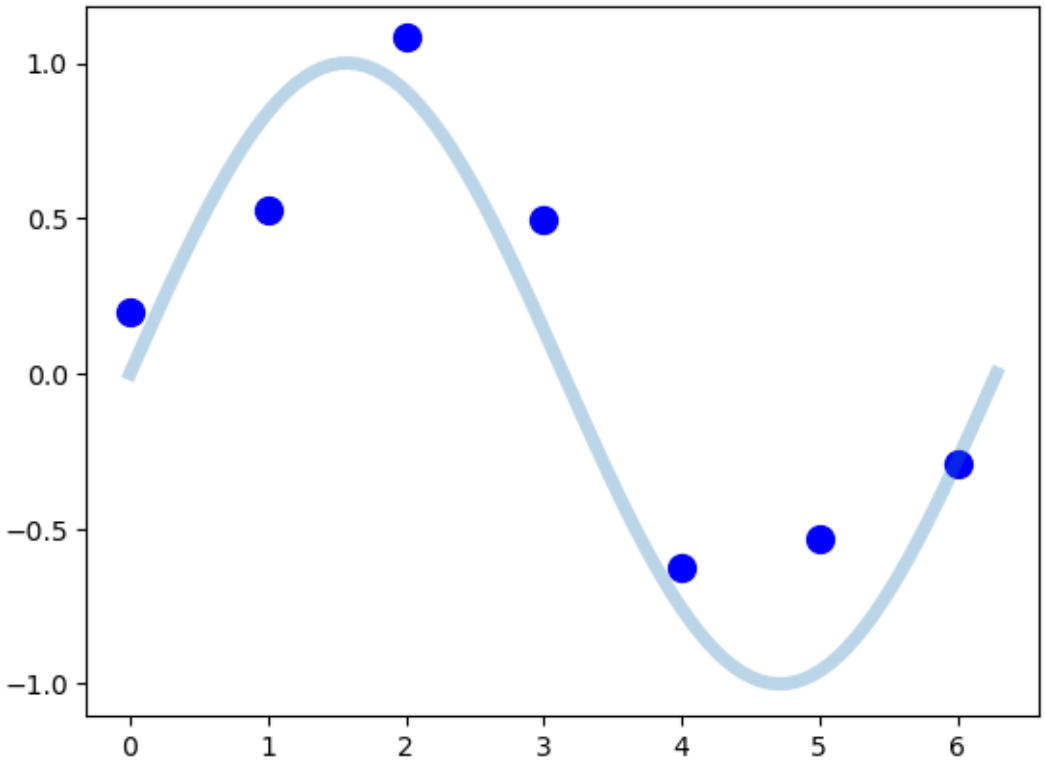
[27]:
```
dat_x = [0, 3, 2, 1, 4, 6, 5]
dat_y = np.sin(dat_x) + 0.25*np.random.randn(len(dat_x))

plt.plot(dat_x, dat_y, 'bo--', markersize = 10);
```



Hm. Yeah, not good. Let's re-plot with the code below.

[28]:
```
plt.plot(dat_x, dat_y, 'bo', markersize = 10);
plt.plot(x, y, linewidth = 5, alpha = 0.3)
```

[28]:
```
[<matplotlib.lines.Line2D at 0x7f98588326a0>]
```



We can see that a sine is still a pretty good description of the data, but the line connecting the data points in the plot before this one was very misleading because it connected the dots in the order that the data were listed.

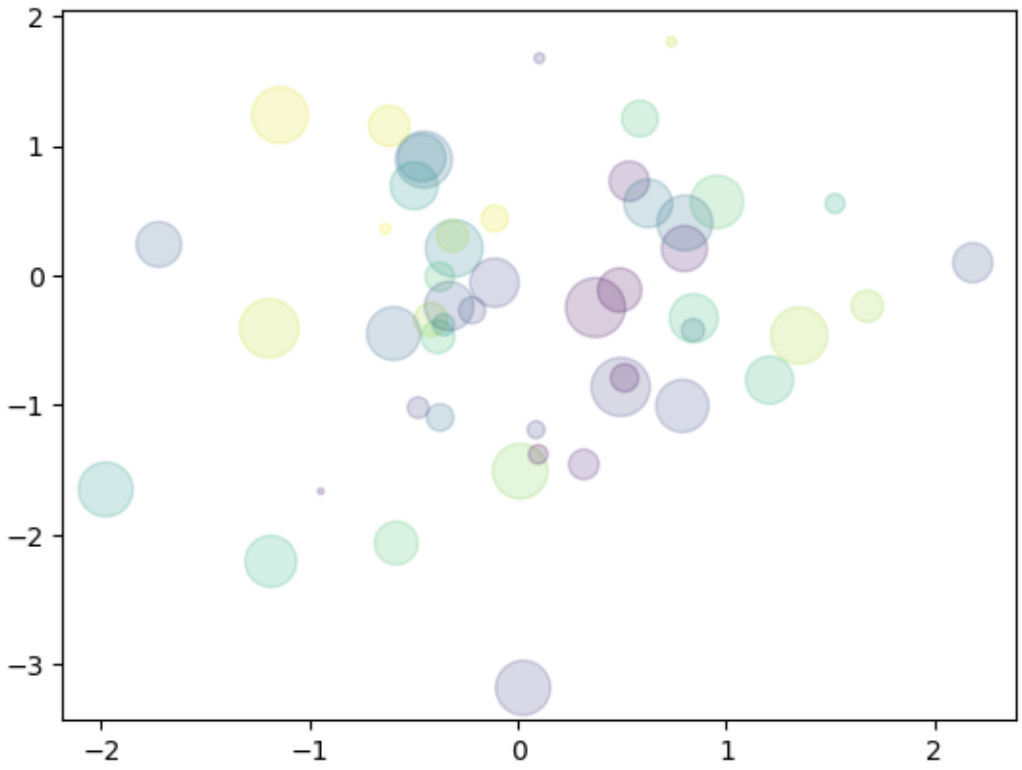
1.3.3 scatterplots

Scatter plots are like line plots in that they plot one variable on the y axis against another variable on the x axis. They don't offer an option to connect the points with a line. But, unlike line plots, *scatterplots allow you to map other variables onto the size or color of the points.*

Here's a scatterplot of totally random data showing how variables can be used to change color and size. Inside the `plt.scatter()` function, the `s` and `c` parameters are used to set the sizes and colors of the data points.

[29]:
```
my_x = np.random.randn(50) # normally distributed x
my_y = np.random.randn(50) # and y
my_sizes = 500 * np.random.rand(50) # random sizes
my_colors = np.random.rand(50) #  colormap numbers 0 to 1

plt.scatter(my_x, my_y, s = my_sizes, c = my_colors, alpha = 0.2)
```

[29]:
```
<matplotlib.collections.PathCollection at 0x7f984b870940>
```



You can use different colormaps to map the data to various colors using the `cmap` parameter.

[30]:
```
my_x = np.random.randn(50) # normally distributed x
my_y = np.random.randn(50) # and y
my_sizes = 500 * np.random.rand(50) # random sizes
my_colors = np.random.rand(50) #  colormap numbers 0 to 1

plt.scatter(my_x, my_y, s = my_sizes, c = my_colors, alpha = 0.2, cmap =␣
 ↪'copper')
```

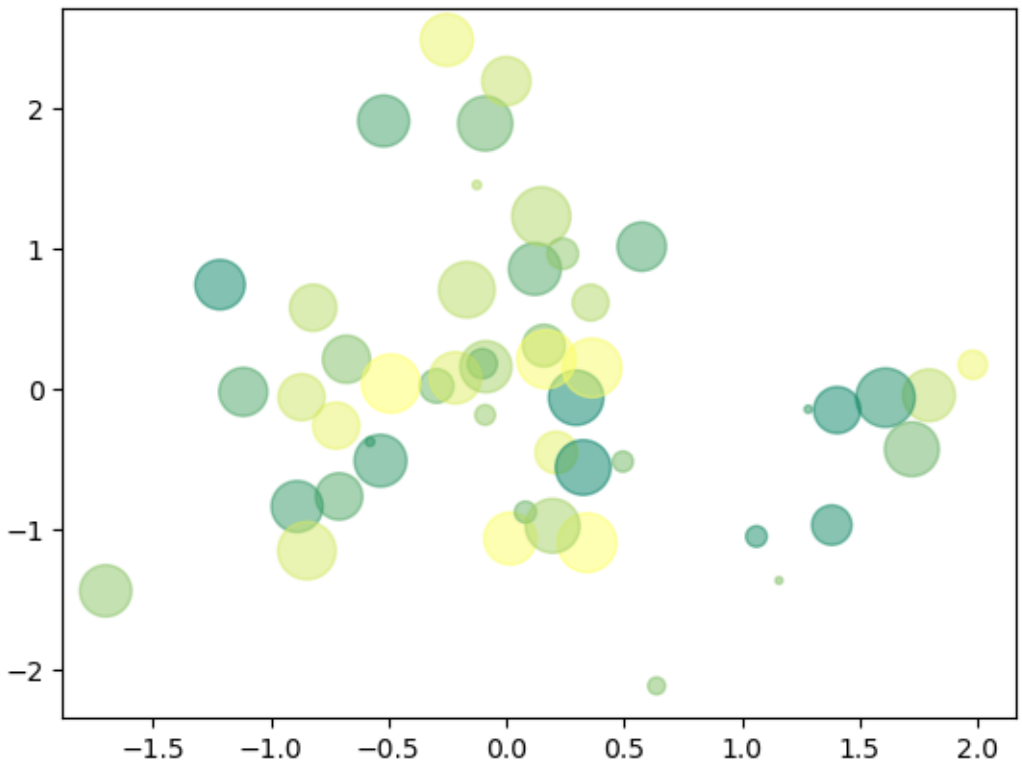
[30]:
```
<matplotlib.collections.PathCollection at 0x7f984b744d00>
```

You can check out the different built-in colormaps here.

Play around with some different colormaps and show us a scatterplot using your favorite.

[31]:
```
plt.scatter(my_x, my_y, s = my_sizes, c = my_colors, alpha = 0.5, cmap =␣
↪'summer')
```

[31]:
```
<matplotlib.collections.PathCollection at 0x7f987969b400>
```

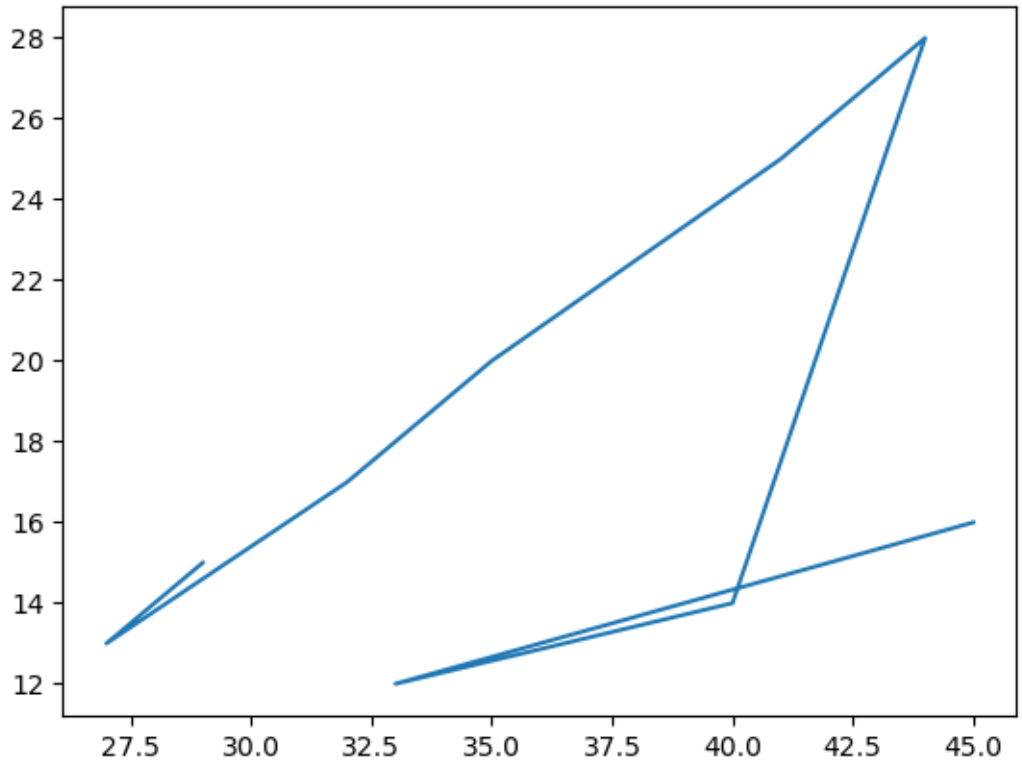


As another little example, here are some toy data on the leg length of 9 dogs, the body length of the same dogs, and a third variable coding the breed.

[32]:
```
leg_len = np.array([15, 13, 17, 20, 25, 28, 14, 12, 16])
body_len = np.array([29, 27, 32, 35, 41, 44, 40, 33, 45])
breed = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

[33]:
```
plt.plot(body_len, leg_len)
```

[33]:
```
[<matplotlib.lines.Line2D at 0x7f98687f3af0>]
```

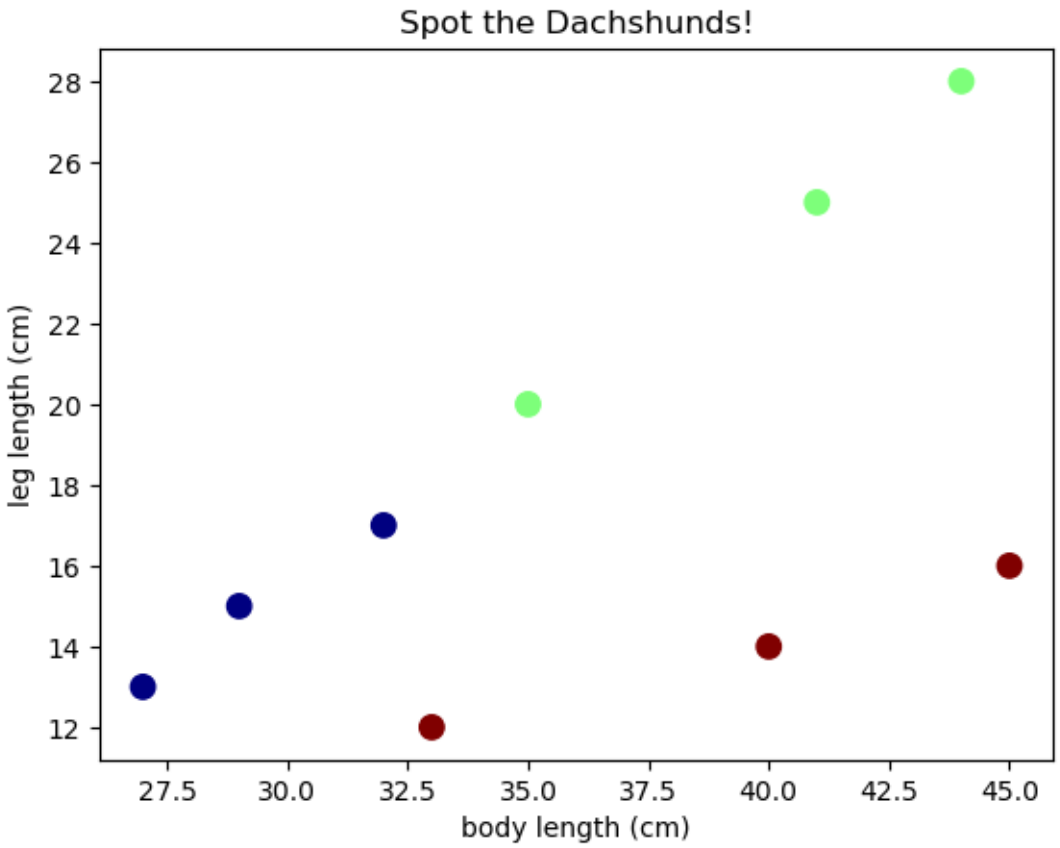


Hm, well, that's not very useful.

Let's give it another go with a scatter plot, mapping the breed variable to data point color.

[34]:
```
plt.scatter(body_len, leg_len, c = breed/2, s = 80, cmap = 'jet')
plt.xlabel('body length (cm)')
plt.ylabel('leg length (cm)')
plt.title('Spot the Dachshunds!')
```

[34]:
```
Text(0.5, 1.0, 'Spot the Dachshunds!')
```

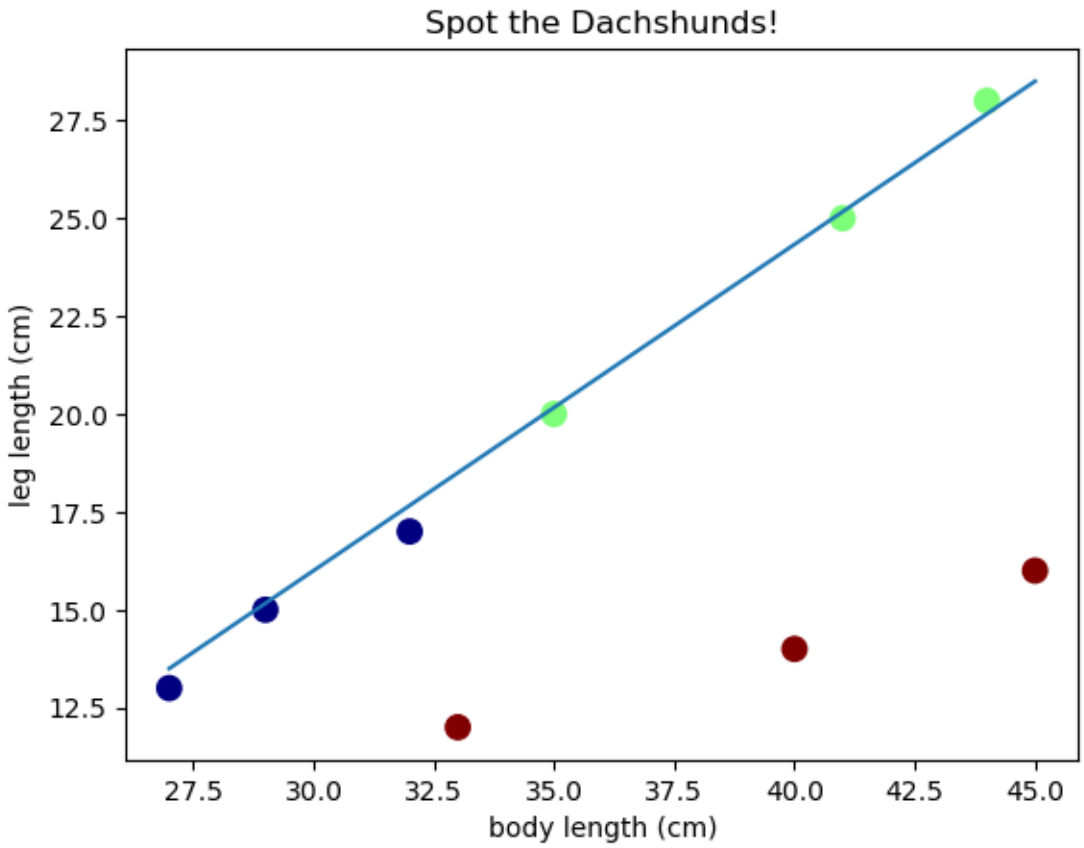


Okay, now it seems pretty clear that two of the breeds fall on a single trend line relating leg length to body length, whereas another breed is on a different line altogether.

We can make this more clear by adding a diagonal reference line to the plot.

[35]:
```
plt.scatter(body_len, leg_len, c = breed/2, s = 80, cmap = 'jet')

# add a reference line
x_ref_line = np.linspace(27, 45) # x value endpoints
y_ref_line = x_ref_line/1.2 - 9  # y calculated using "by-hand" slope and y-int␣
  ↪estimates
plt.plot(x_ref_line, y_ref_line)
plt.xlabel('body length (cm)')
plt.ylabel('leg length (cm)')
plt.title('Spot the Dachshunds!')
```

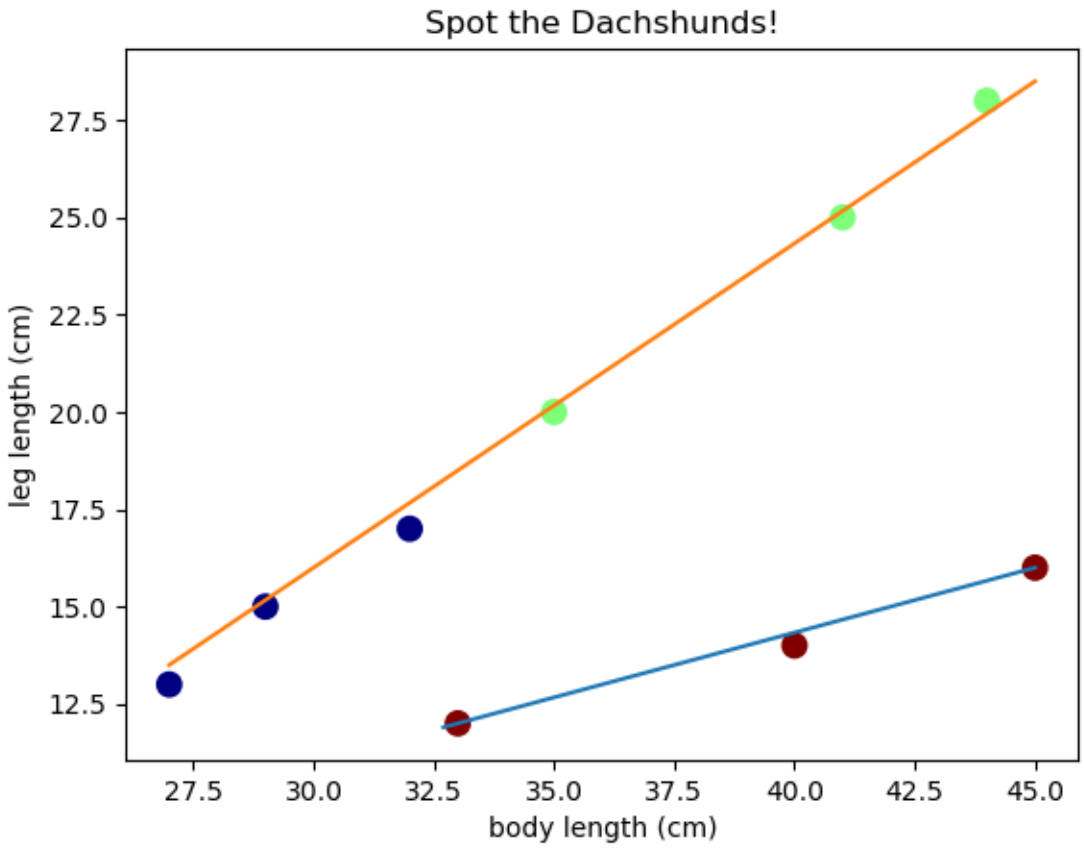
[35]:
```
Text(0.5, 1.0, 'Spot the Dachshunds!')
```



In the cell below, see if you can add a second reference line to the doggy plot to capture the Dachshund data.

[36]:
```
plt.scatter(body_len, leg_len, c = breed/2, s = 80, cmap = 'jet')

# add a reference line
x_ref_line = np.linspace(27, 45) # x value endpoints
y_ref_line = x_ref_line/1.2 - 9  # y calculated using "by-hand" slope and y-int␣
 ↪estimates
ref_x = np.linspace(32.7, 45)
ref_y = ref_x/3+1
plt.plot(ref_x, ref_y)
plt.plot(x_ref_line, y_ref_line)
plt.xlabel('body length (cm)')
plt.ylabel('leg length (cm)')
plt.title('Spot the Dachshunds!')
```

[36]:
```
Text(0.5, 1.0, 'Spot the Dachshunds!')
```



1.4 Anatomy of a plot

Trust us, understanding how matplotlib creates figures will save you a lot of time and aggravation in the long run! The basic anatomy of a matplotlib plot is shown in the first figure.

As we mentioned above, everything is an "artist". Some artists are "containers" because they contain other things, and some are "primitives" that draw themselves inside containers.

The containers are

- **figures** - figures contain 1 or more axes
- **axes** - axes contain an x axis, a y axis, and your curves, data points, etc.
- **axis** - an axis contains tick labels and tick marks

The "primitives" here are the

- **Line2D** - the line plotting the sine wave (inside an axes)
- **annotations** - x and y axis labels and title (inside an axes)

Let's make a few of these things

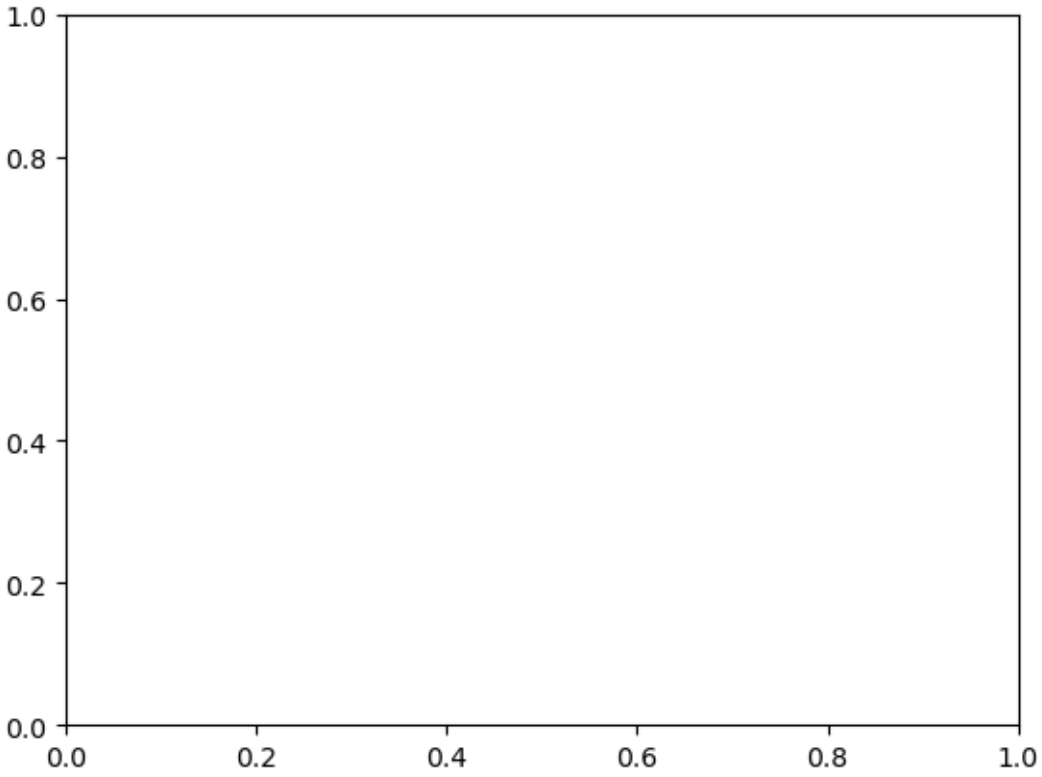
[37]:
```
my_fig = plt.figure()
```

```
<Figure size 640x480 with 0 Axes>
```

Well, that wasn't very exciting. It seems a figure was created in memory, but nothing is being displayed. This is actually just an oddity of Jupyter notebooks. If you do this in iPython at the terminal or in IDLE and do a plt.show(), a blank figure will appear.

Let's add an axes inside of it. Even though we don't usually do it this way, figures know how to make and draw axes inside of themselves. Don't worry about the arguments to `subplot()` just yet.

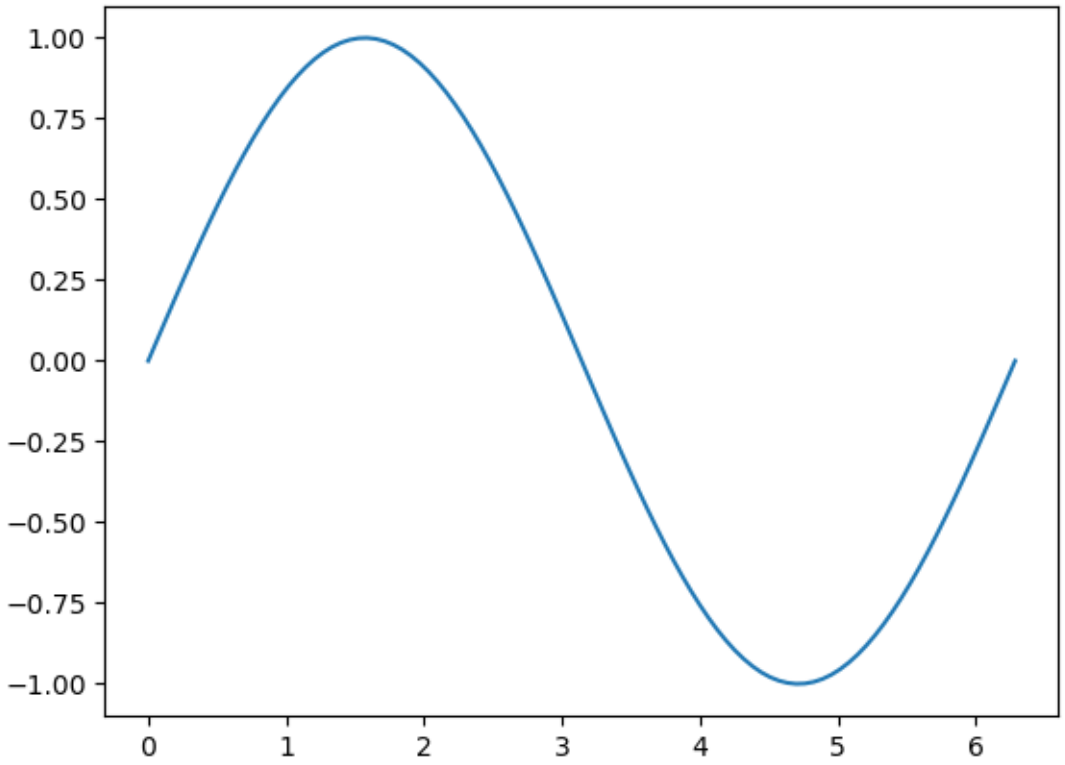
[38]:
```
my_fig = plt.figure()
my_ax = my_fig.add_subplot(1,1,1)
```



Once we have an axes, we can plot stuff in it:

[39]:
```
my_fig = plt.figure()
my_ax = my_fig.add_subplot(1,1,1)
my_ax.plot(x, y, label = 'sine')
```

[39]:
```
[<matplotlib.lines.Line2D at 0x7f9858cb4280>]
```



In this markdown cell, list the "artists" in this thing we just created above.

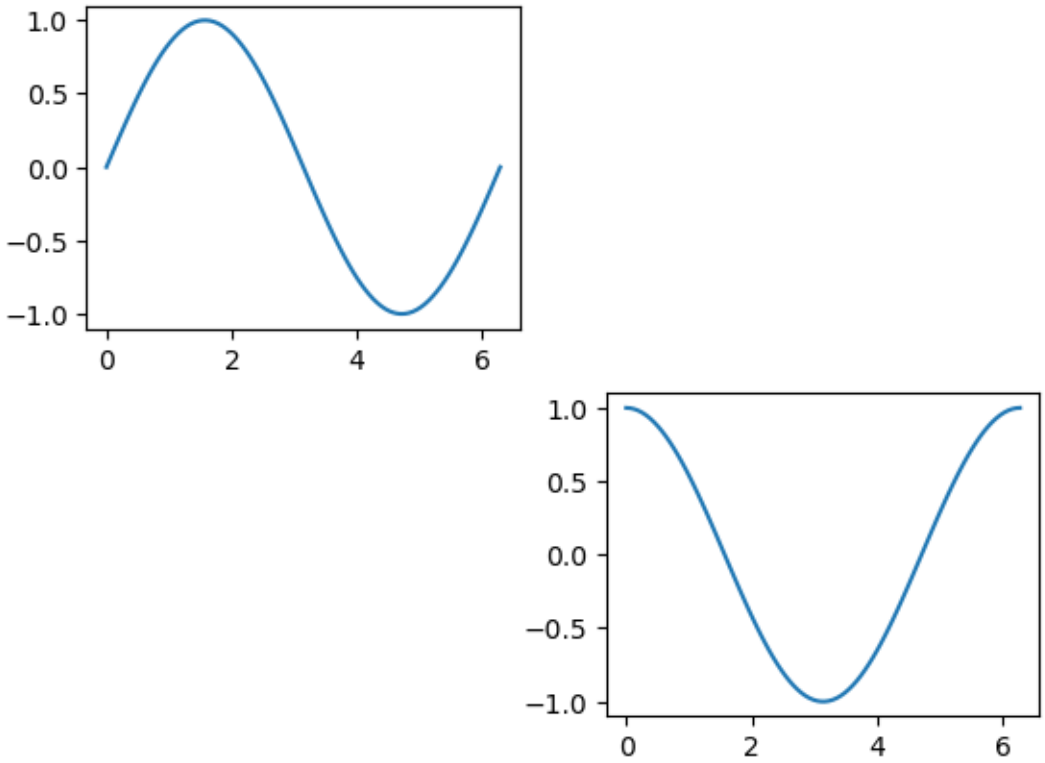
1.5 Multi-panel figures

We often want to display more than on graphic (or 'axes' in matplotlib terms) in a single figure. In general, these are referred to as "multi-panel figures" and the panels are referred to as "subplots".

In the section above, we created a figure, made the figure create an axes inside of itself, and then made the axes plot the graph to illustrate the hierarchy. Alternatively, we make `plt` handle the figure and axes creation for us with its `subplot()` method.

[40]:
```
plt.subplot(2,2,1)
plt.plot(x, y, label = 'sine')
plt.subplot(2,2,4)
plt.plot(x, y2, label = 'cosine')
```

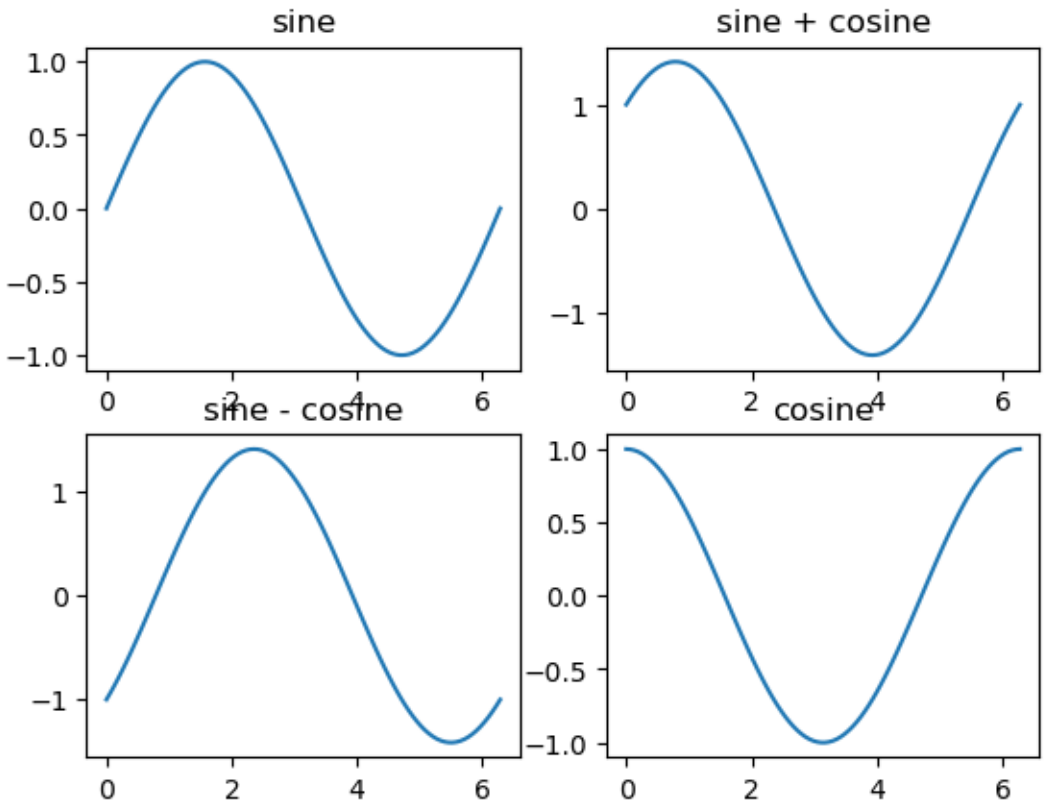
[40]:
```
[<matplotlib.lines.Line2D at 0x7f98795efa90>]
```



In the cell below, recreate the figure above with the sum and difference of the sine and cosine in the remaining two cells, and put a title on each panel (i.e. subplot, i.e. “axes”).

[41]:
```
y3 = np.sin(x) + np.cos(x)
y4 = np.sin(x) - np.cos(x)
plt.subplot(2,2,1)
plt.plot(x, y, label = 'sine')
plt.title('sine')

plt.subplot(2,2,2)
plt.plot(x, y3, label = 'sum of sine and cosine')
plt.title('sine + cosine')

plt.subplot(2,2,3)
plt.plot(x, y4, label = 'difference of sine and cosine')
plt.title('sine - cosine')

plt.subplot(2,2,4)
plt.plot(x, y2, label = 'cosine')
plt.title('cosine')
```

[41]: `Text(0.5, 1.0, 'cosine')`



Fancier arrangements are possible, but we'll leave it here for now.

[ ]: