6. ADVANCED PLOTTING

JHU Physics & Astronomy
Python Workshop 2017

Lecturer: Mubdi Rahman

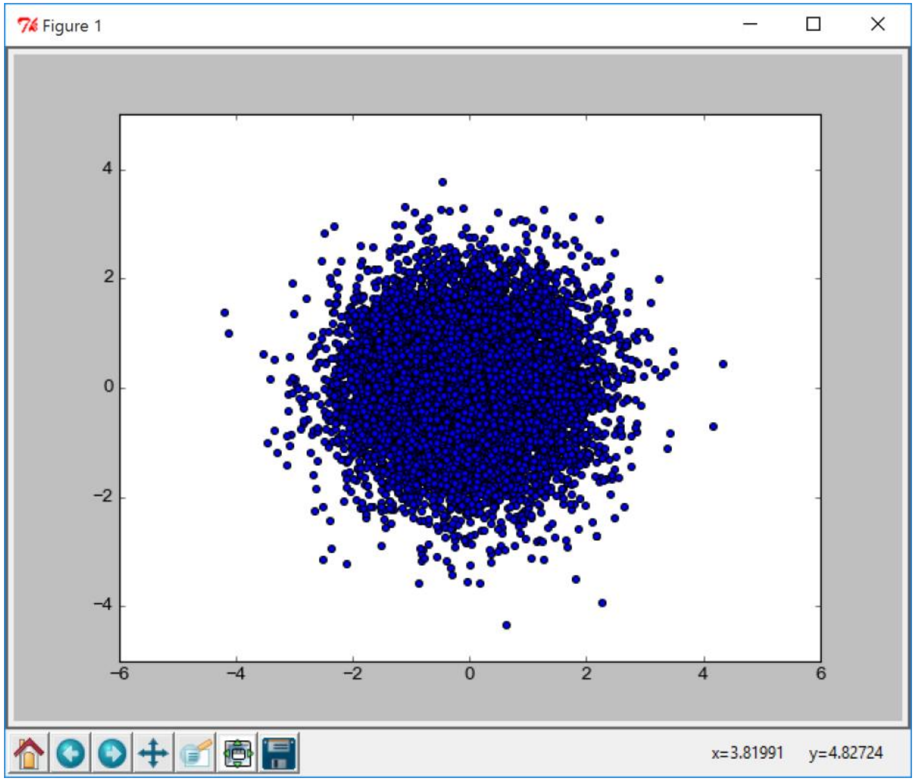
MATPLOTLIB REDUX

You've got the **basics,** now let's unleash the **power**!

ALPHA/TRANSPARENCY

Every plotting function in matplotlib accepts the “alpha” parameter. This parameter goes from 0 to 1, where 0 indicates fully transparent to 1 meaning fully opaque. For instance:

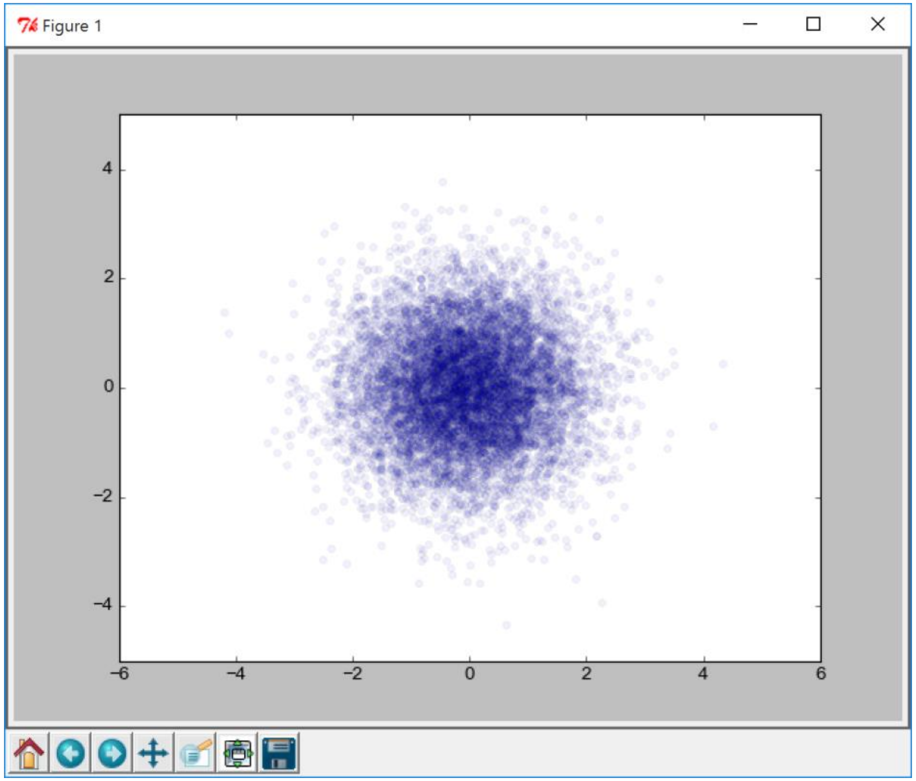
```
plt.scatter(
  x, y, alpha=1
)
```



ALPHA/TRANSPARENCY

Every plotting function in matplotlib accepts the “alpha” parameter. This parameter goes from 0 to 1, where 0 indicates fully transparent to 1 meaning fully opaque. For instance:

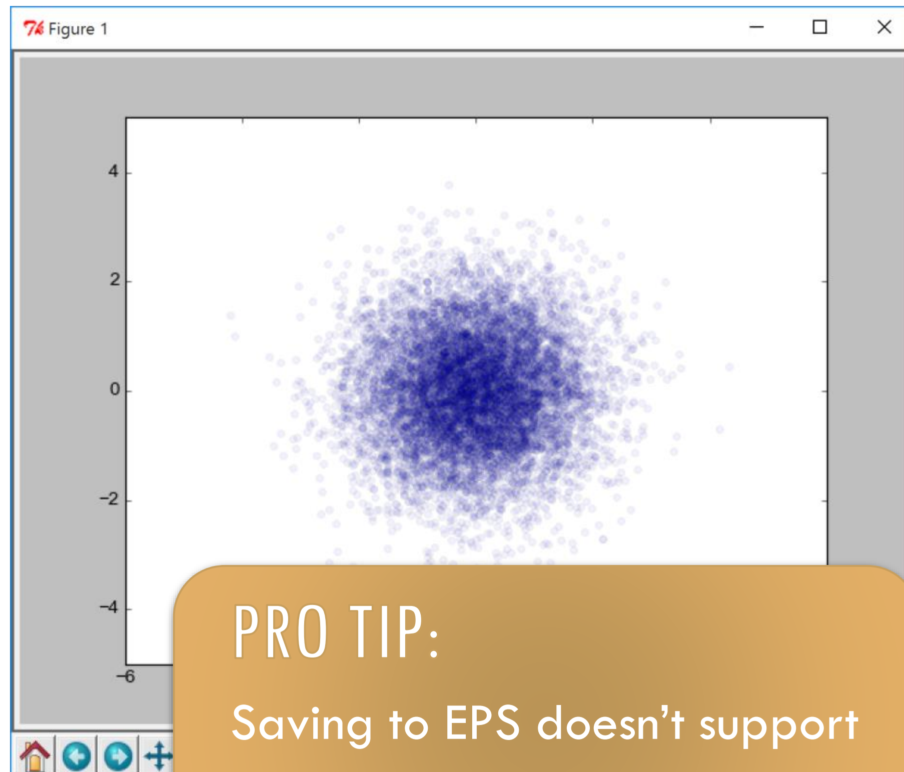
```
plt.scatter(
  x, y, alpha=0.05
)
```



ALPHA/TRANSPARENCY

Every plotting function in matplotlib accepts the “alpha” parameter. This parameter goes from 0 to 1, where 0 indicates fully transparent to 1 meaning fully opaque. For instance:

```
plt.scatter(
  x, y, alpha=0.05
)
```



PRO TIP:

Saving to EPS doesn’t support transparency.

IMAGES

Images (when stored in an array) are in a different order than in the Cartesian sense. For instance, finding coordinate (3,2):

Image Coordinates

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

Cartesian Coordinates

0,4	1,4	2,4	3,4
0,3	1,3	2,3	3,3
0,2	1,2	2,2	3,2
0,1	1,1	2,1	3,1
0,0	1,0	2,0	3,0

IMAGES

Images (when stored in an array) are in a different order than in the Cartesian sense. For instance, finding coordinate (3,2):

Image Coordinates

First Axis

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

Second Axis

Cartesian Coordinates

0,4	1,4	2,4	3,4
0,3	1,3	2,3	3,3
0,2	1,2	2,2	3,2
0,1	1,1	2,1	3,1
0,0	1,0	2,0	3,0

IMAGES

Images (when stored in an array) are in a different order than in the Cartesian sense. For instance, finding coordinate (3,2):

Image Coordinates

First Axis

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

Second Axis

Cartesian Coordinates

0,4	1,4	2,4	3,4
0,3	1,3	2,3	3,3
0,2	1,2	2,2	3,2
0,1	1,1	2,1	3,1
0,0	1,0	2,0	3,0

Second Axis

First Axis

IMAGES

Images (when stored in an array) are in a different order than in the Cartesian sense:

```
array([[0, 0, 0],
       [1, 0, 0],
       [2, 0, 0]])

arr[:,0] =
  array([0, 1, 2])
```

If you want matplotlib to show your image in Cartesian coordinates, you will need to flip and reverse your array.

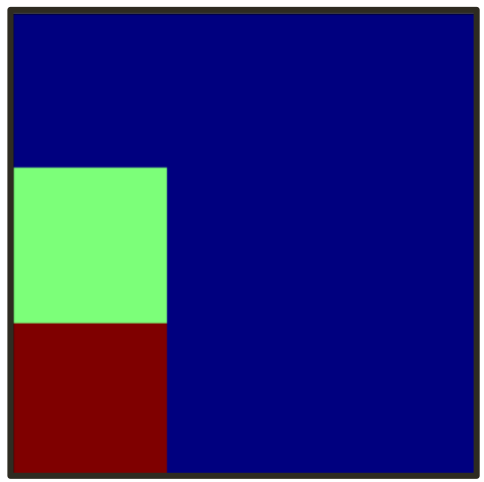
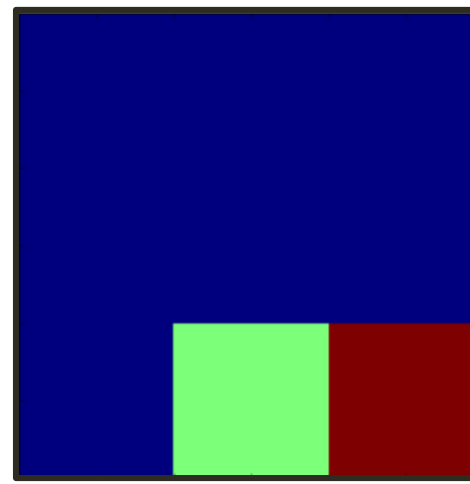


Image Coordinates



Cartesian Coordinates

IMAGES

Images (when stored in an array) are in a different order than in the Cartesian sense:

```
array([[0, 0, 0],
       [1, 0, 0],
       [2, 0, 0]])

arr[:,0] =
  array([0, 1, 2])
```

If you want matplotlib to show your image in Cartesian coordinates, you will need to transpose and reverse your array.

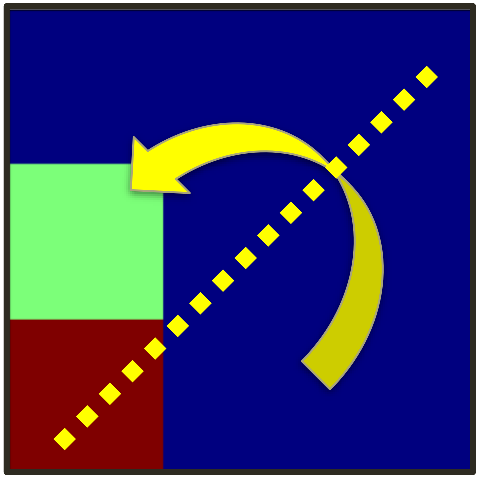
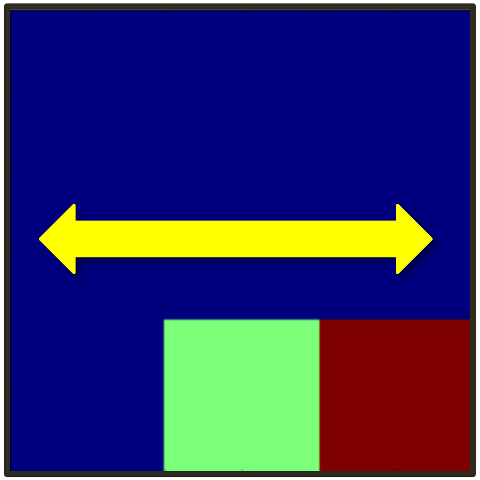


Image Coordinates



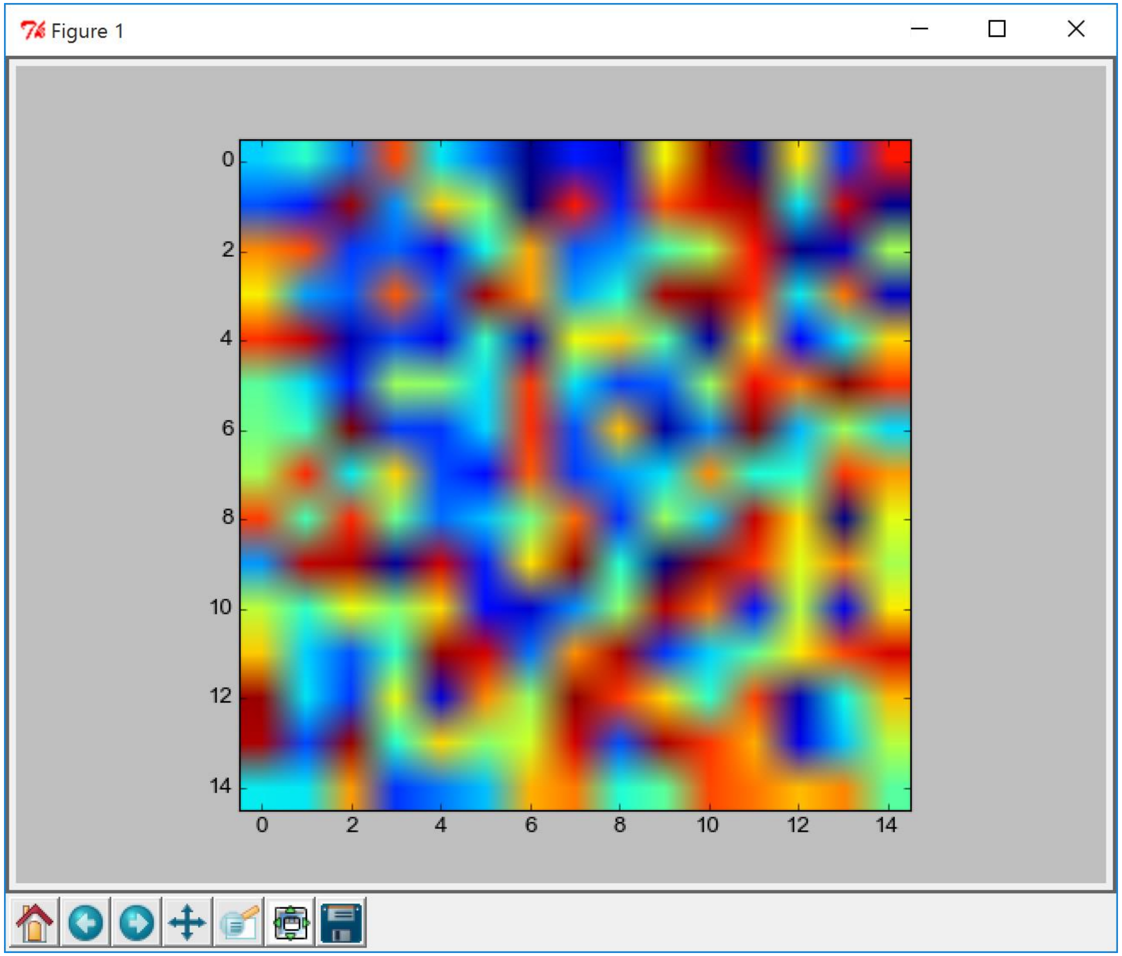
Cartesian Coordinates

IMSHOW

Imshow is the go-to image plotting function in matplotlib. The basic syntax is:

```
plt.imshow(arr1)
```

But this likely doesn't do what you want it to, so there are many optional arguments to use.



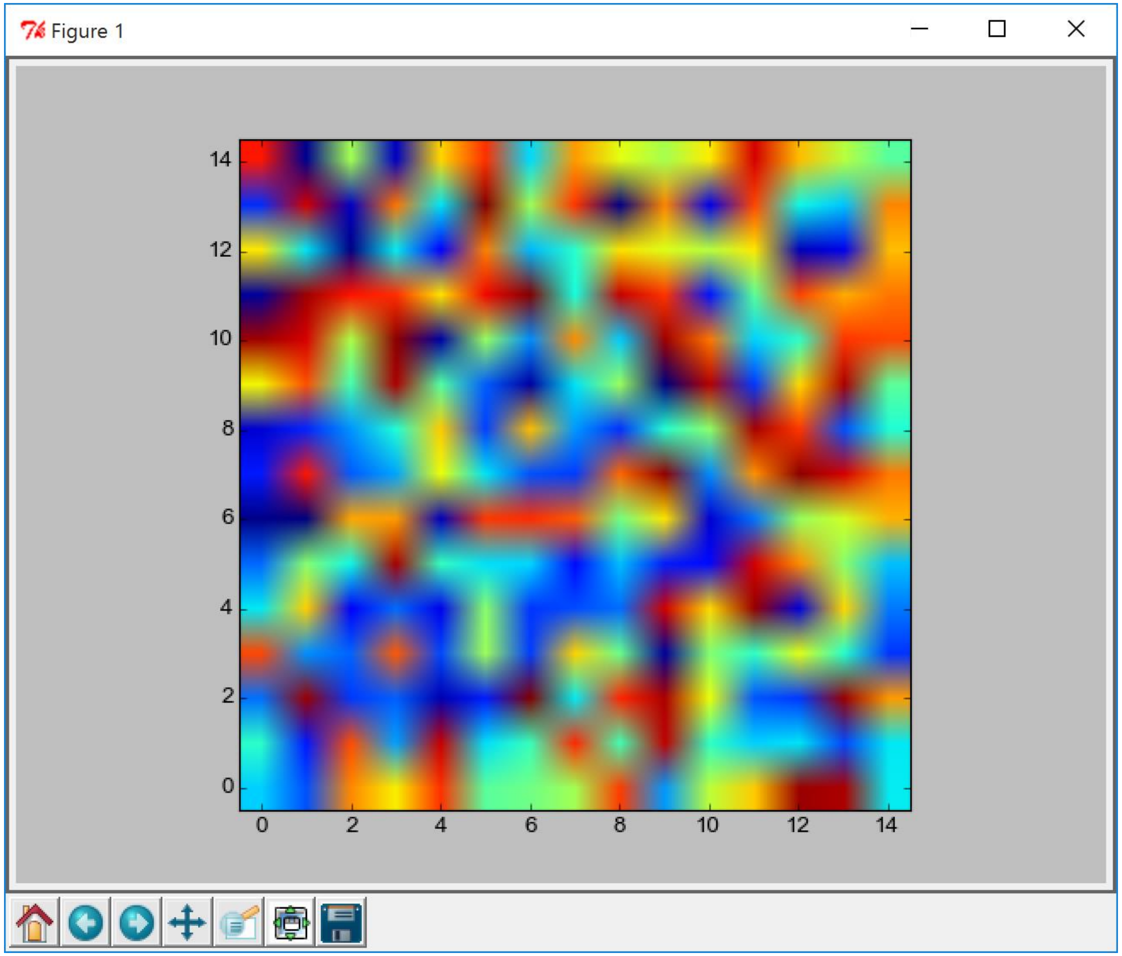
IMSHOW

Moving to Cartesian coordinates manually:

```
plt.imshow(
  arr1[:,::-1].T
)
```

or if you want to make it a little more automated:

```
plt.imshow(
  arr1.T,
  origin='lower'
)
```



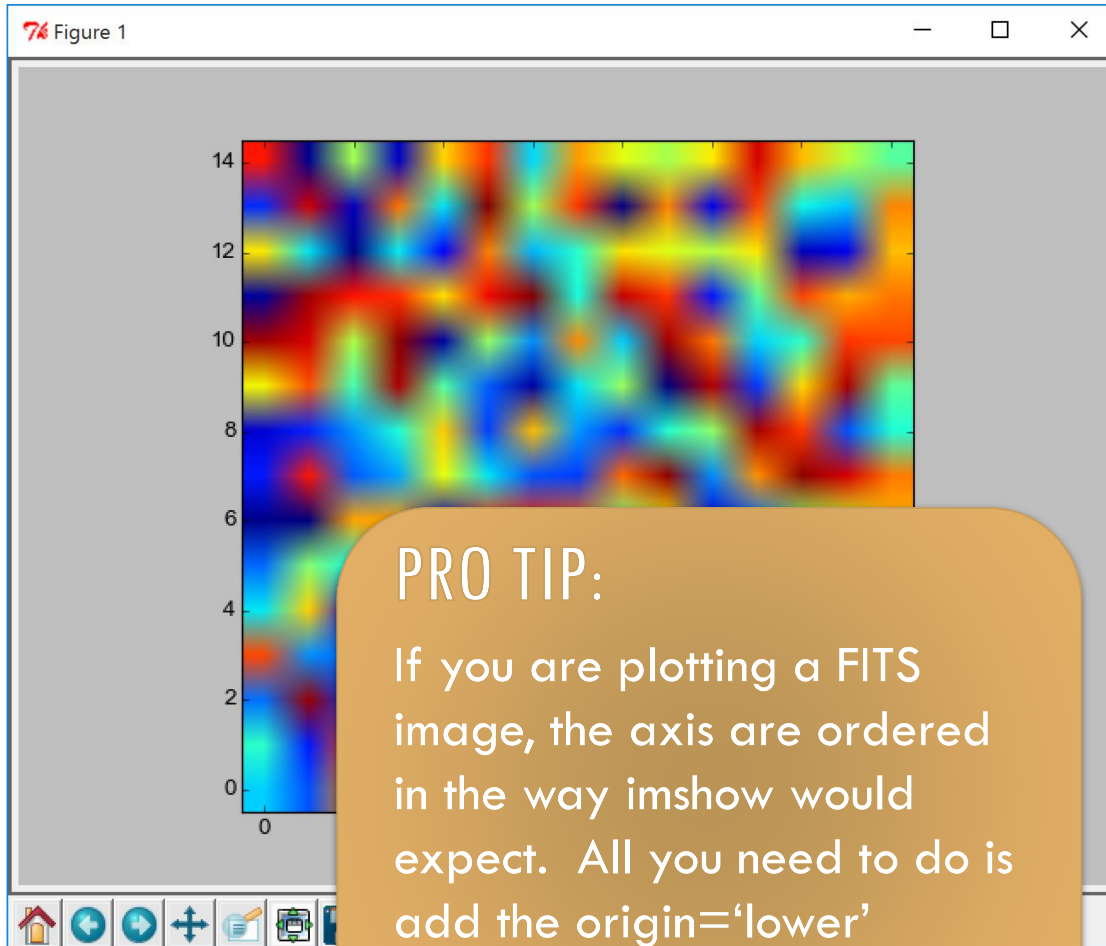
IMSHOW

Moving to Cartesian coordinates manually:

```
plt.imshow(
  arr1[:,::-1].T
)
```

or if you want to make it a little more automated:

```
plt.imshow(
  arr1.T,
  origin='lower'
)
```



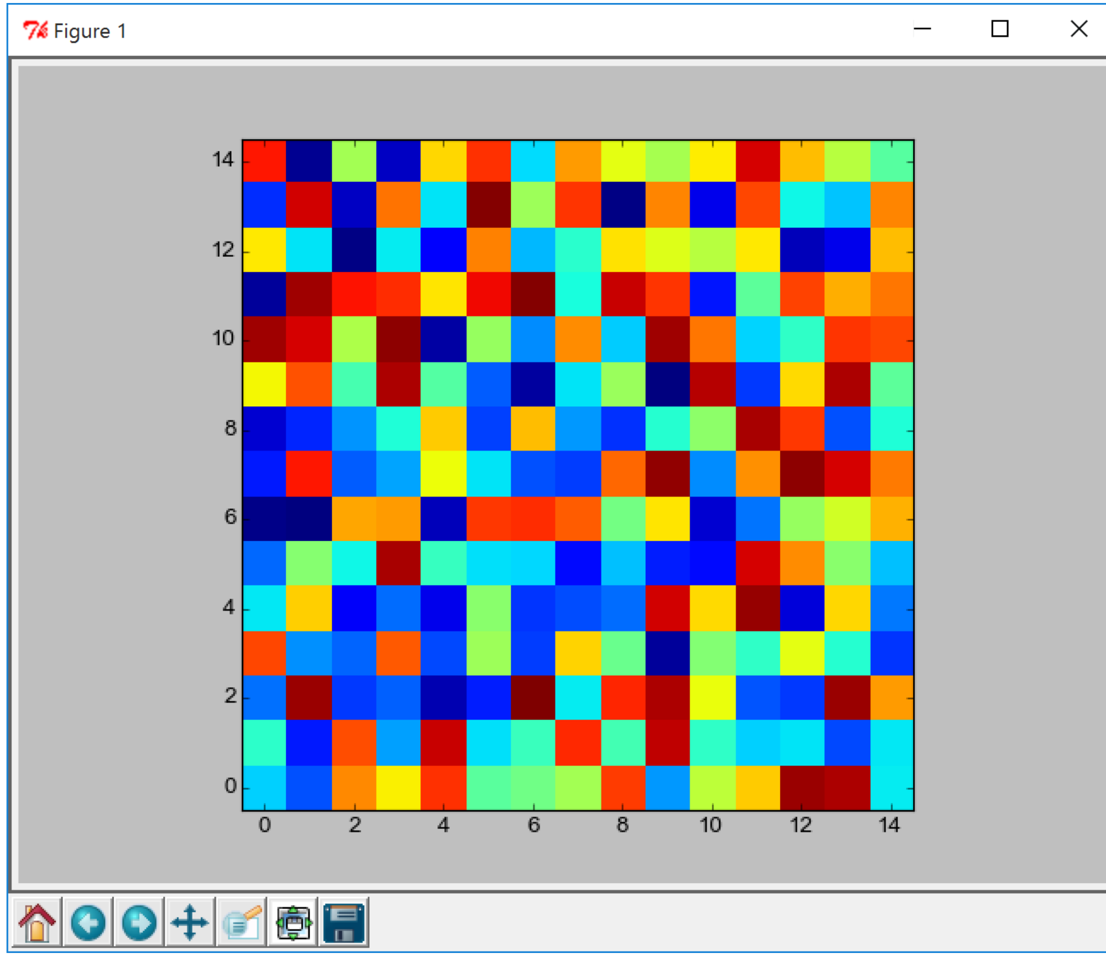
PRO TIP:

If you are plotting a FITS image, the axis are ordered in the way imshow would expect. All you need to do is add the origin='lower' keyword.

IMSHOW

The fuzziness is due to interpolation between pixels. The default is “bilinear”. To see the pixels:

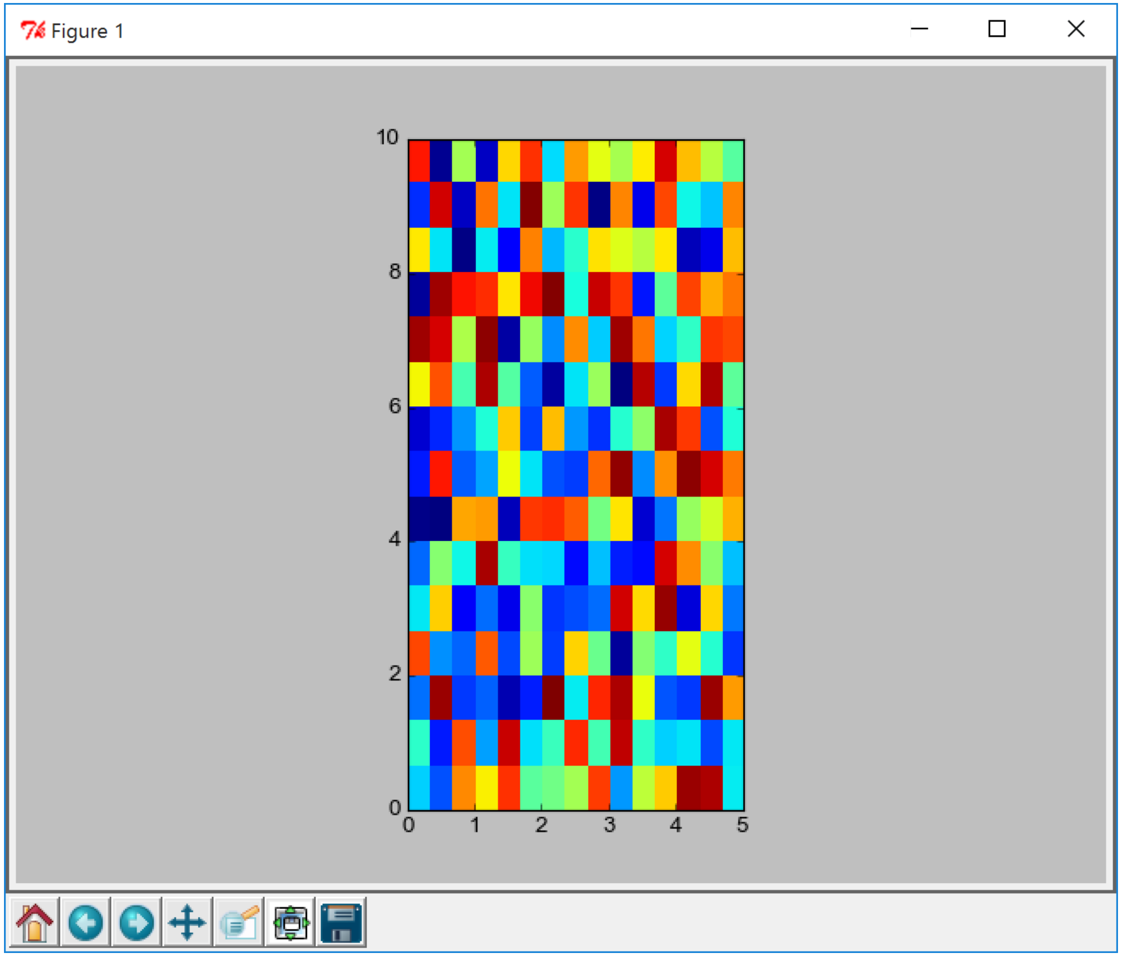
```
plt.imshow(
  arr1.T,
  origin=‘lower’,
  interpolation=
  ‘nearest’
)
```



IMSHOW

By default, the image is placed such that the pixels are centred on their pixel number. This can be changed using the “extent” argument:

```
plt.imshow(
  …, extent=[0,
5, 0, 10]
)
```

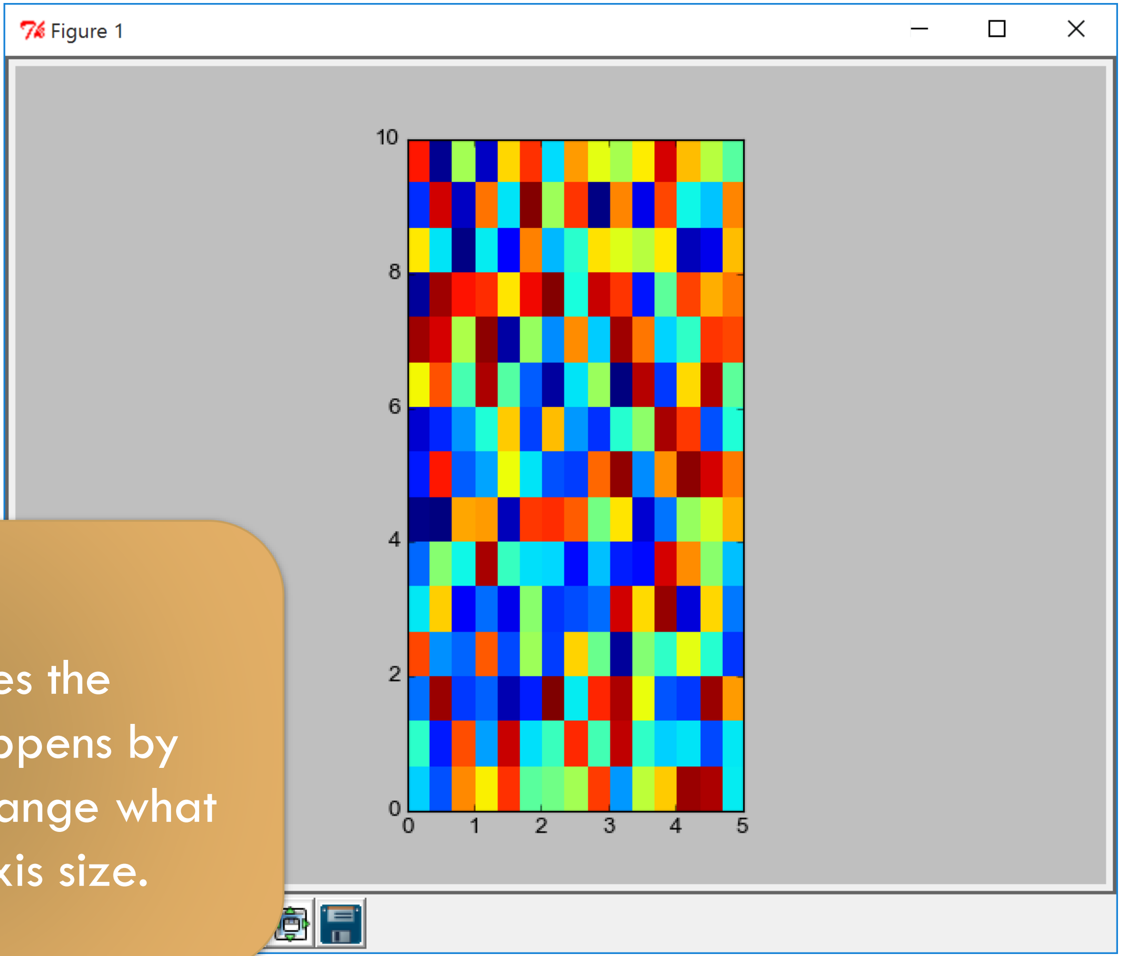


IMSHOW

By default, the image is placed such that the pixels are centred on their pixel number. This can be changed using the "extent" argument:

pl

5,
)



PRO TIP:

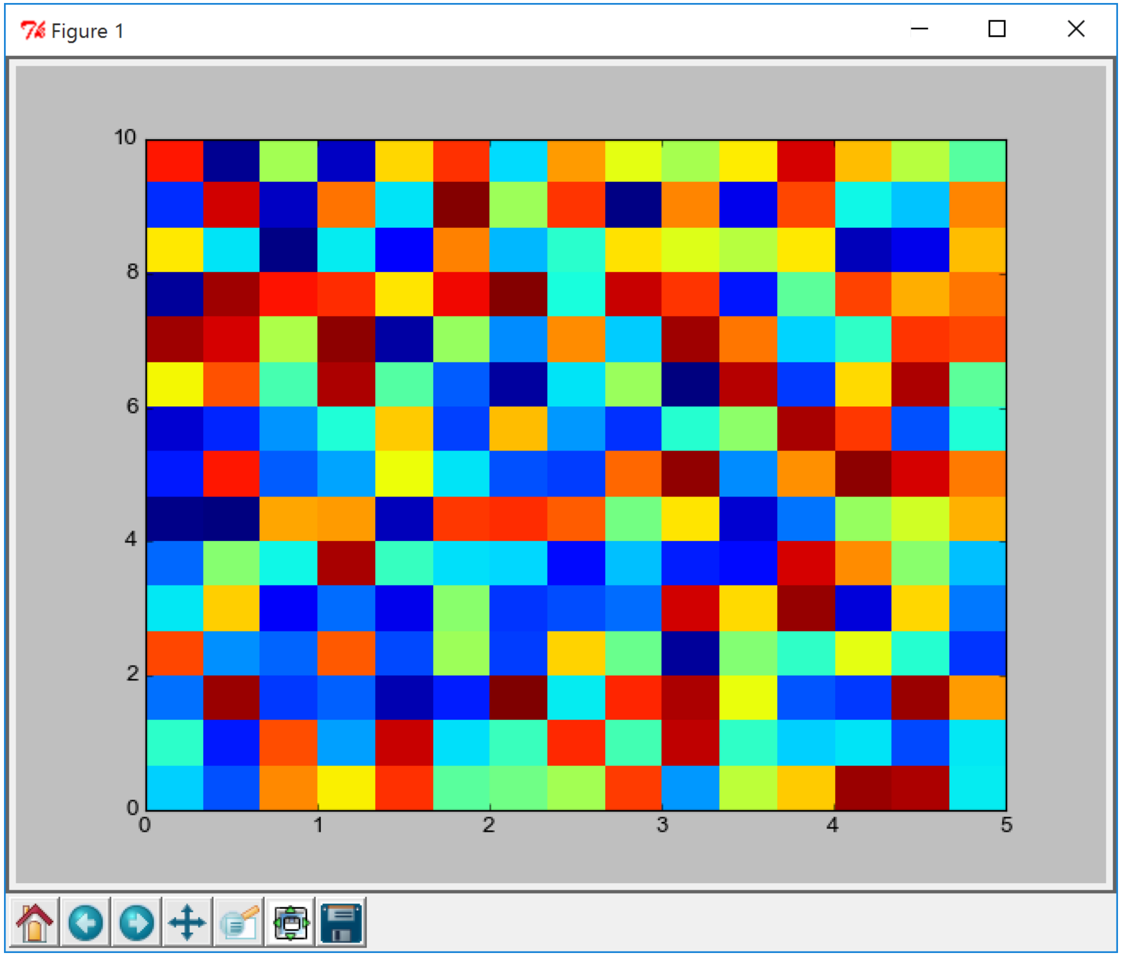
Note that this changes the aspect ratio. This happens by default, and may change what you've set as your axis size.

IMSHOW

By default, the axis ratio of the pixels is unity. You can change this manually or automatically using the “aspect” argument:

```
plt.imshow(…,
  aspect=‘auto’
)
```

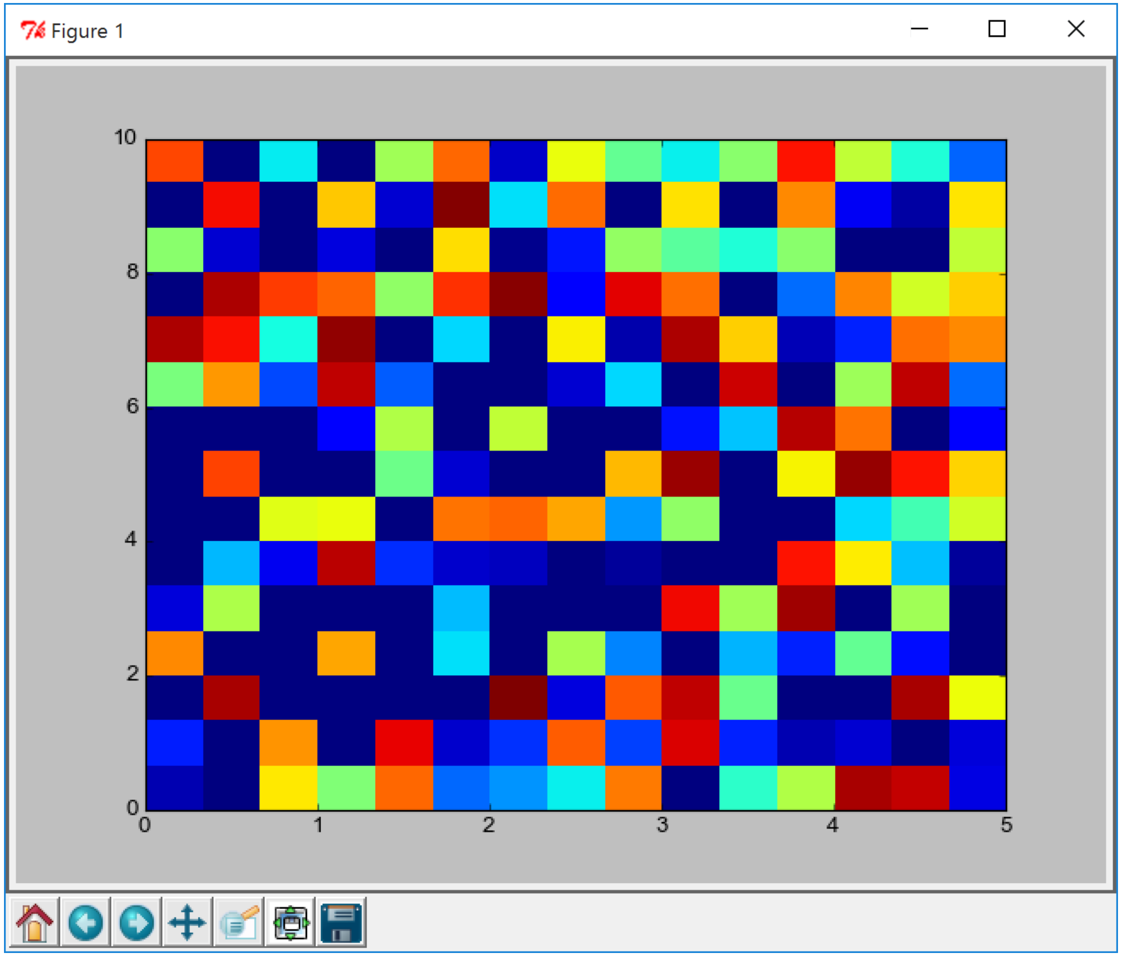
‘auto’ ensures that the axes doesn’t change its size or location.



IMSHOW

imshow will try to autoscale the image. If you want a different min or max value, you can change the “vmin” or “vmax” values:

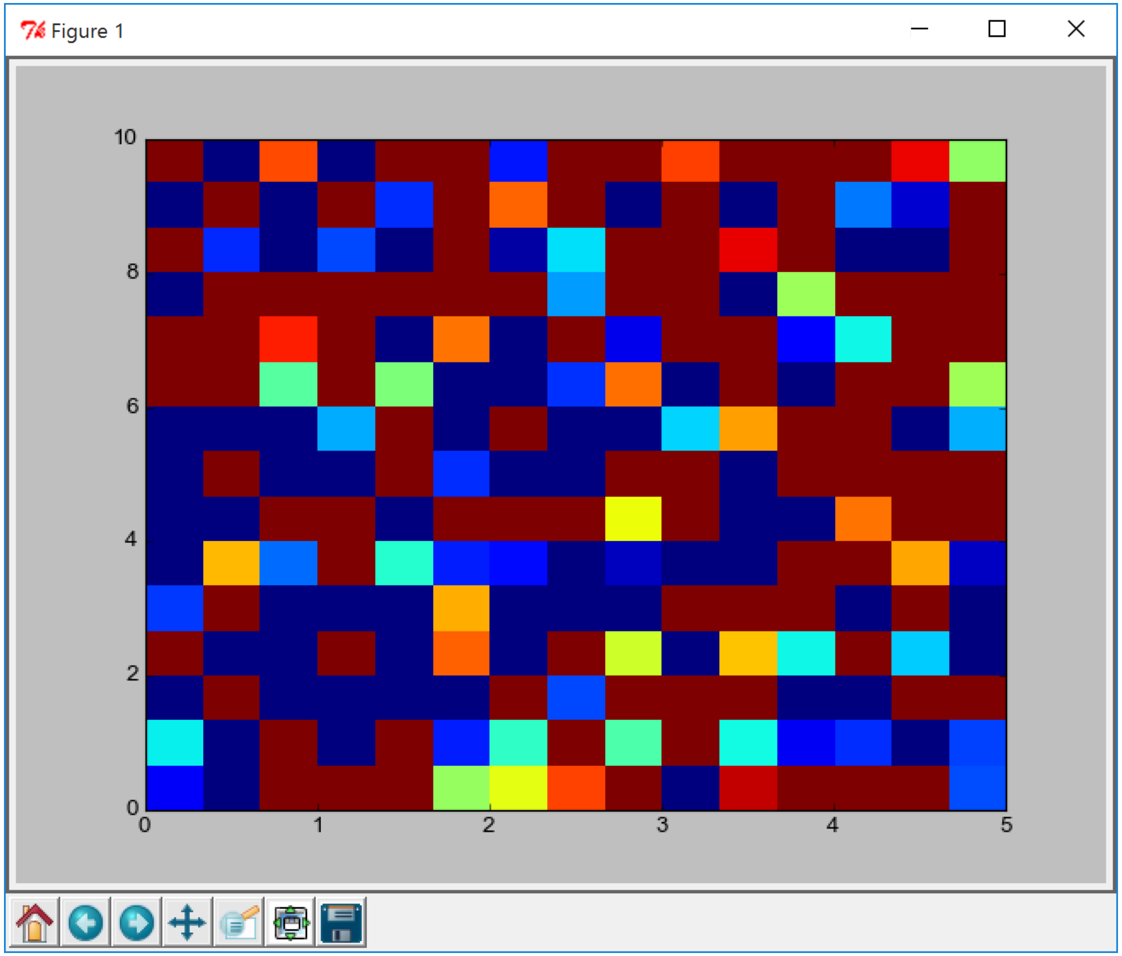
```
plt.imshow(…,
  vmin=0.3
)
```



IMSHOW

imshow will try to autoscale the image. If you want a different min or max value, you can change the "vmin" or "vmax" values:

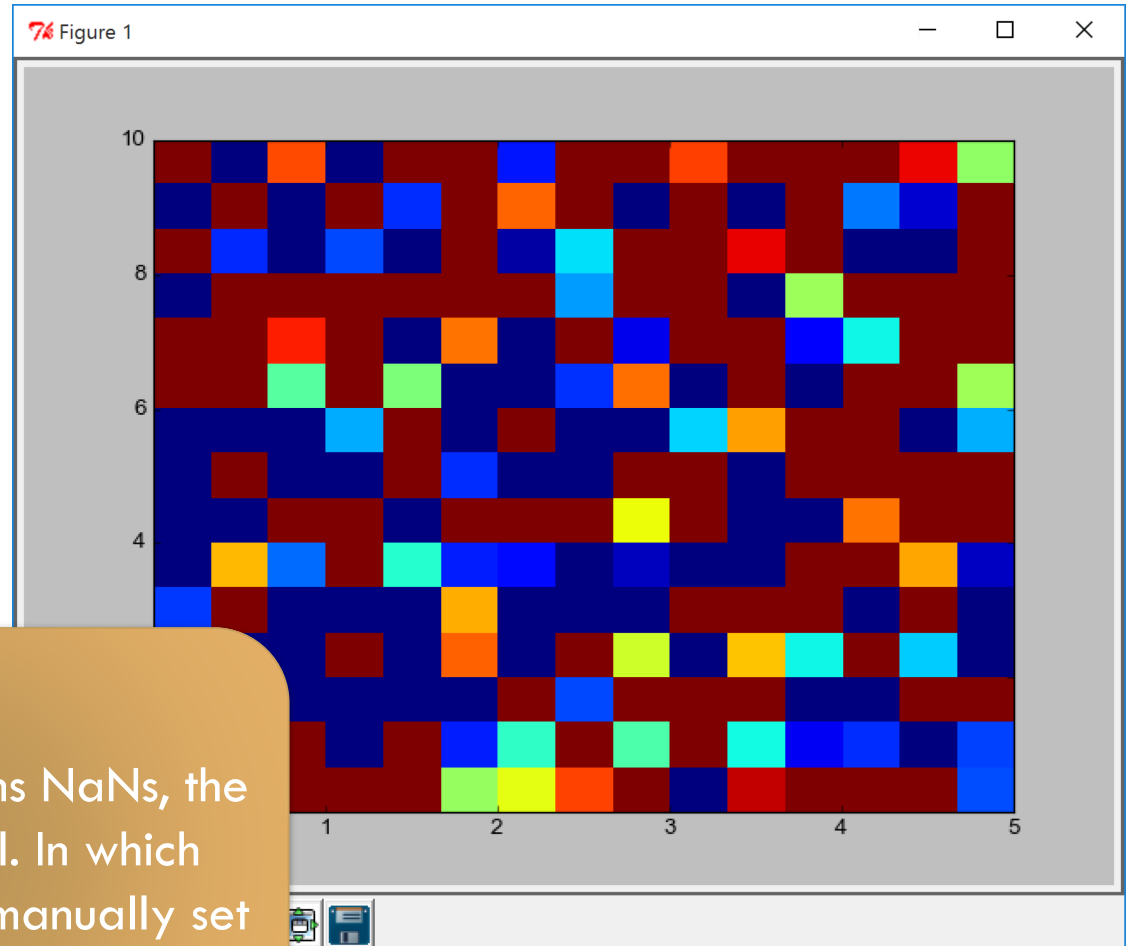
```
plt.imshow(…,
  vmin=0.3,
  vmax=0.6
)
```



IMSHOW

imshow will try to autoscale the image. If you want a different min or max value, you can change the "vmin" or "vmax" values:

```
plt.imshow(…,
  vm
  v
)
```



PRO TIP:

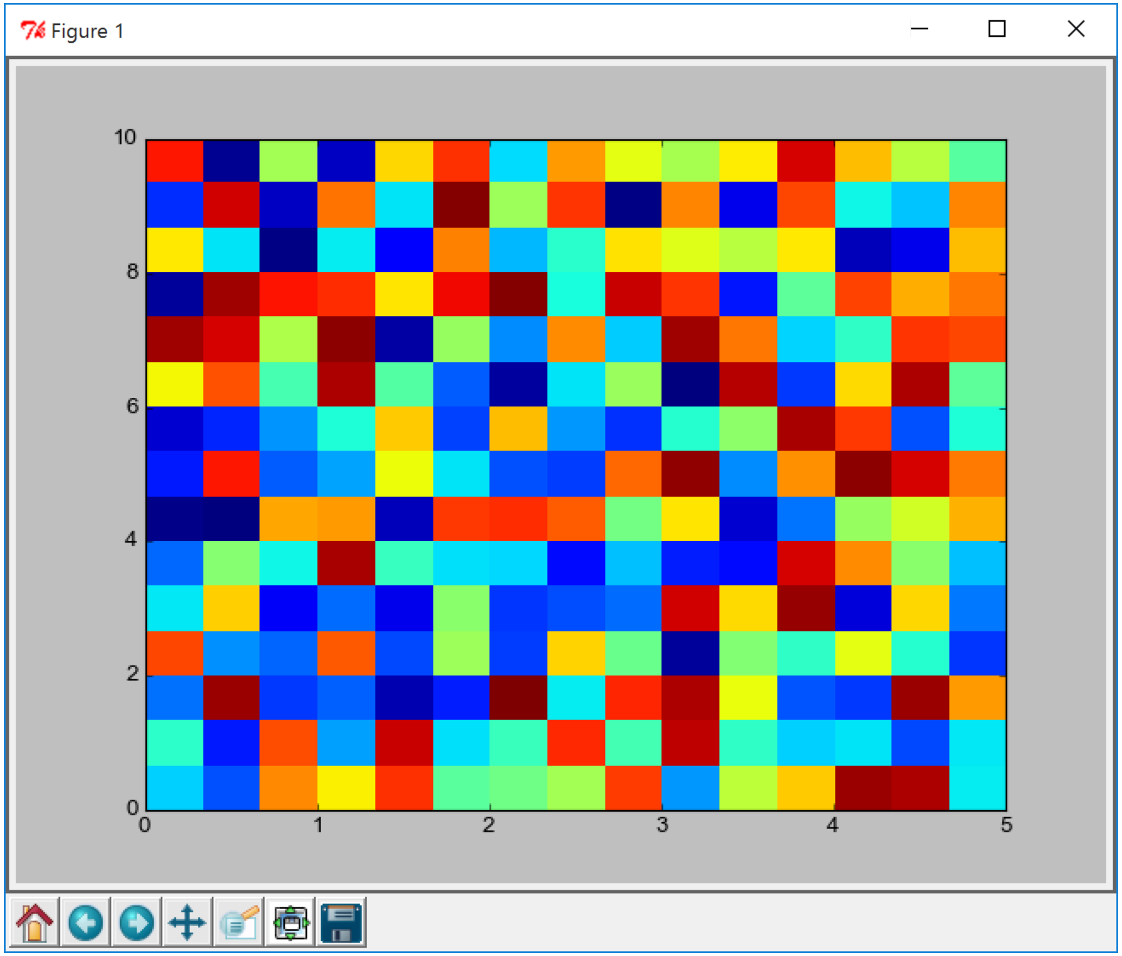
If the array contains NaNs, the autoscaling will fail. In which case, you need to manually set vmin/vmax values.

IMSHOW

We can also change the colourmap used to turn floating point values into colours:

```
plt.imshow(…,
  cmap=plt.cm.jet
)
```

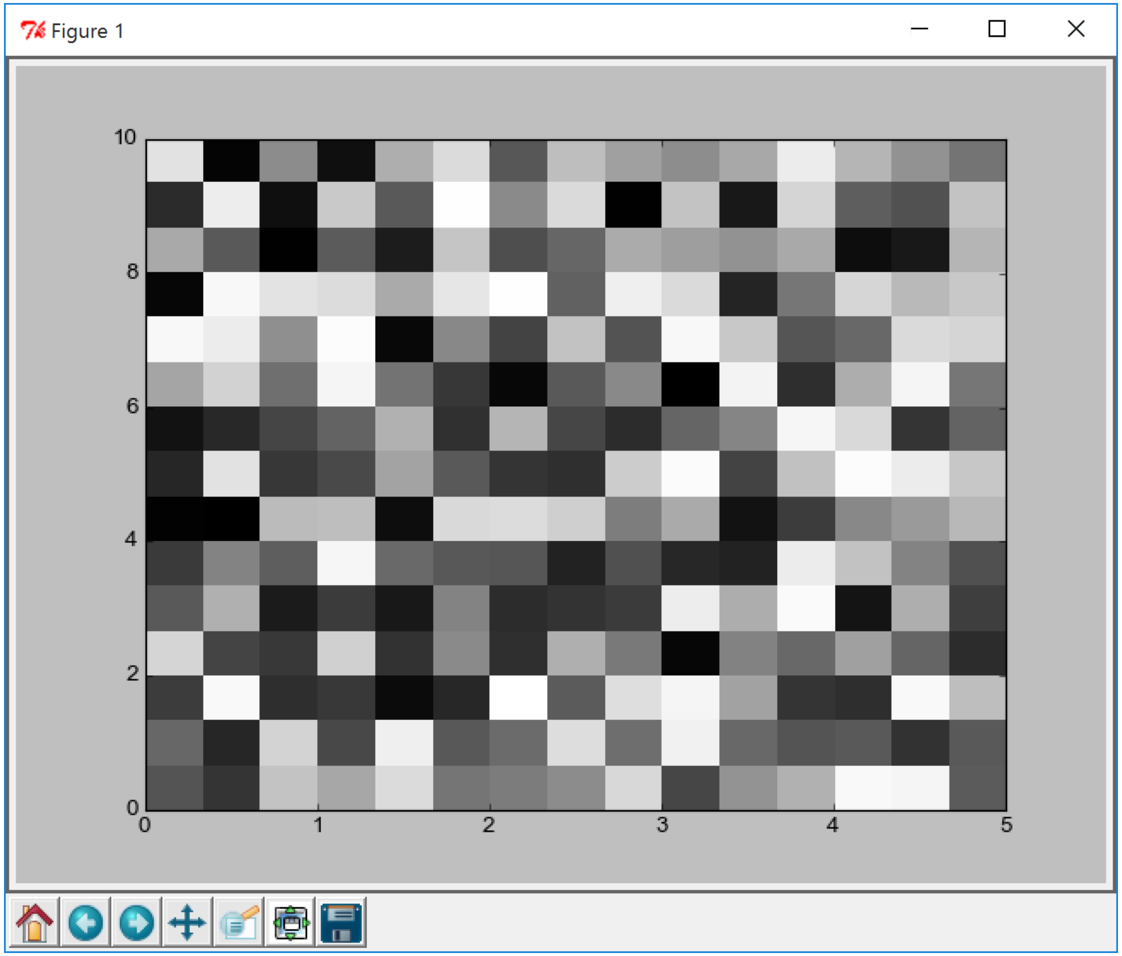
This is the default colourmap



IMSHOW

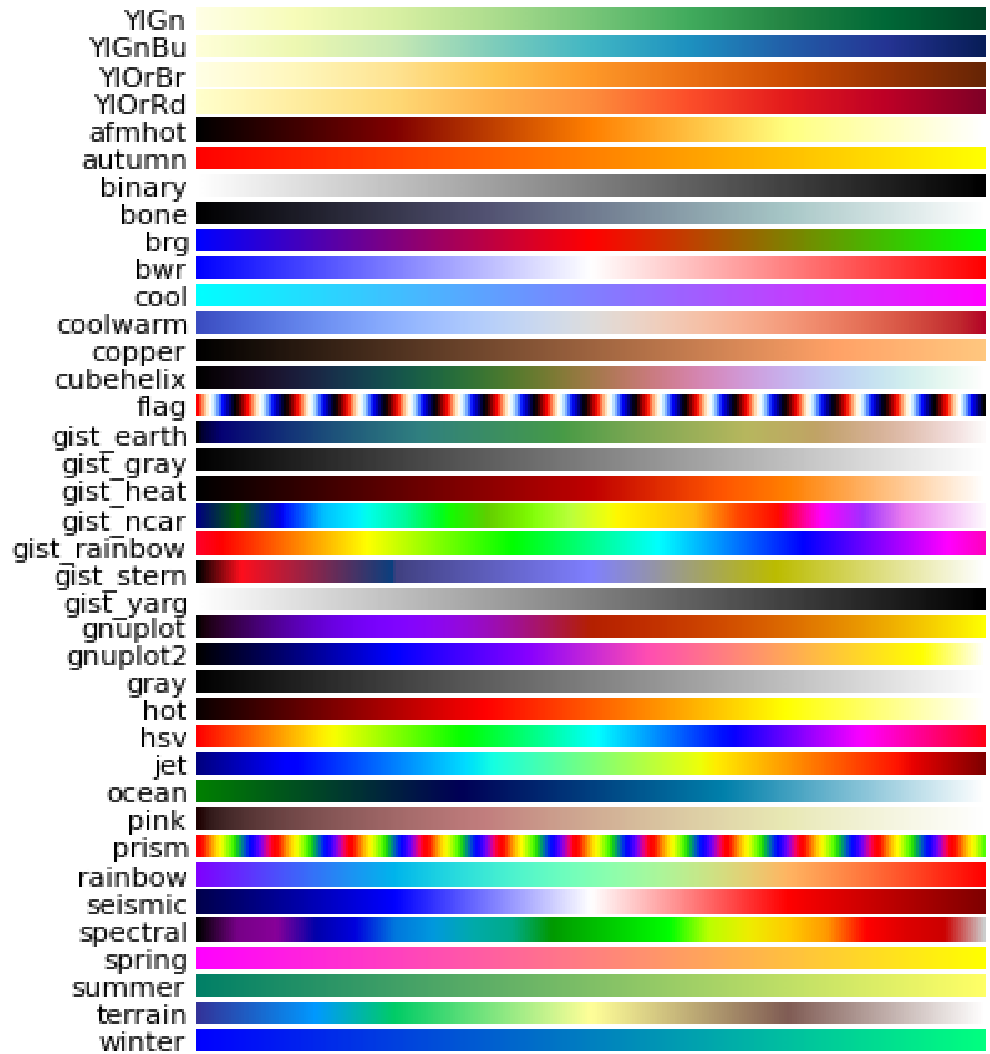
We can also change the colourmap used to turn floating point values into colours:

```
plt.imshow(…,
 cmap=plt.cm.gray
)
```



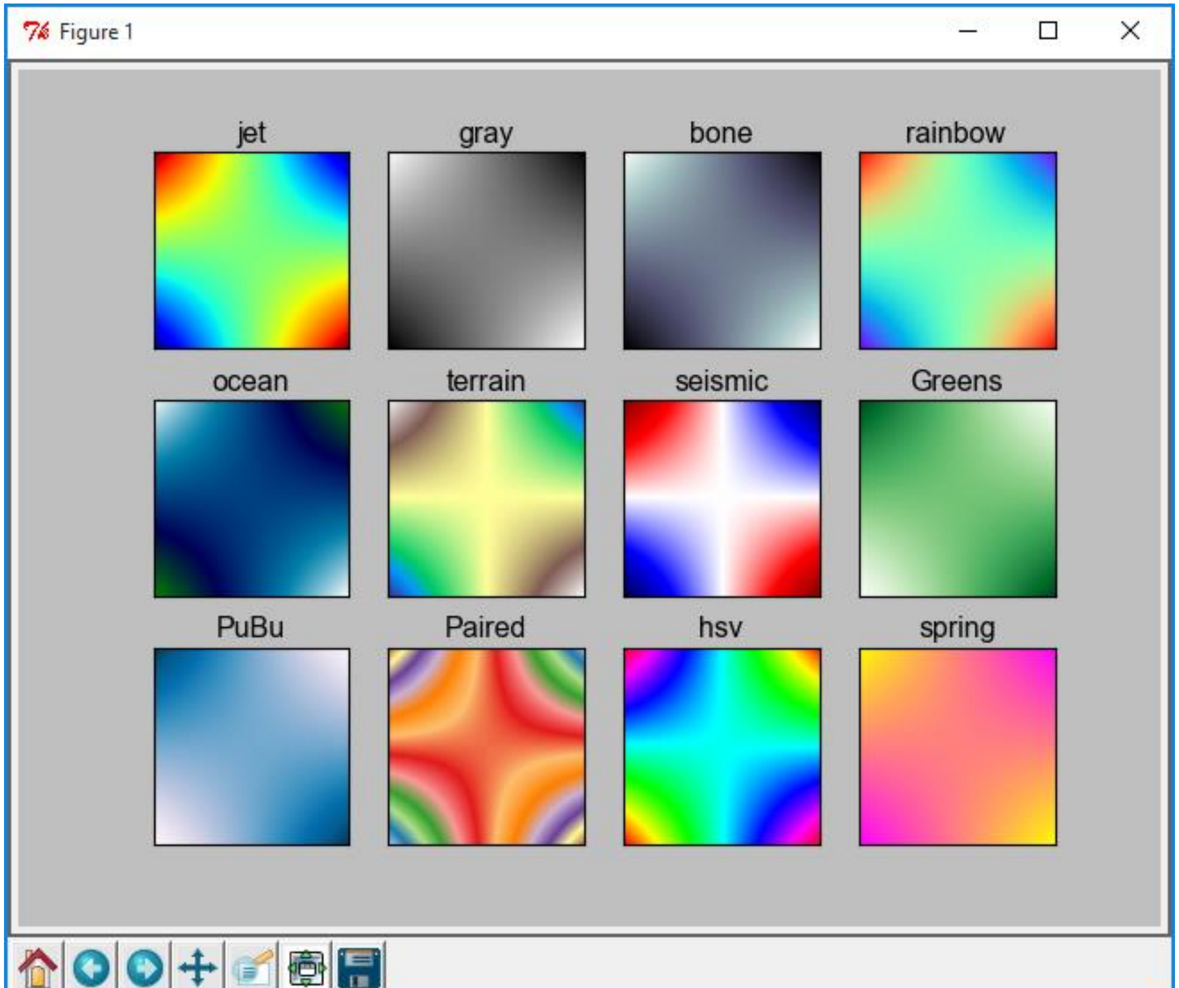
COLOURMAPS

Matplotlib has a large selection of colourmaps available. You can also code your own! All of the colourmaps are located in the plt.cm module.



Just a selection of built-in colour maps

COLOURMAPS



A general selection of colourmap. Your choice of colourmap does matter.

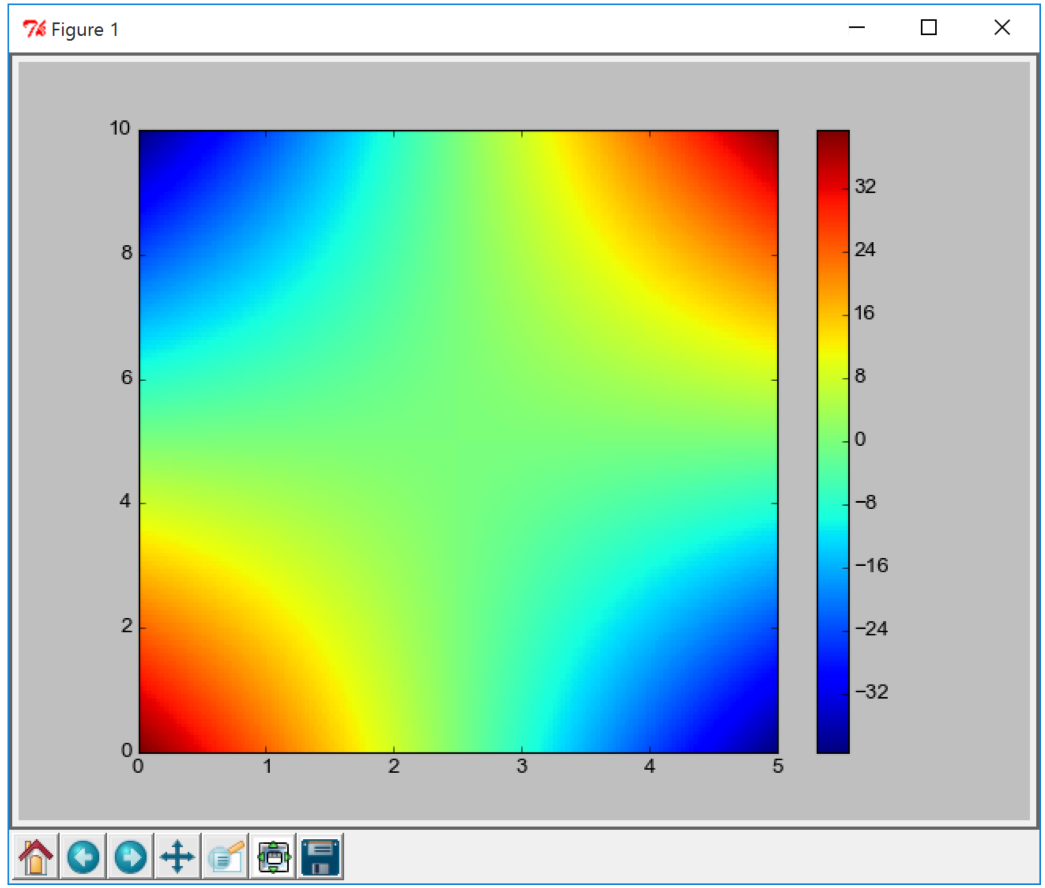
Choose the one that works best for your purpose.

COLOUR BARS

You can create a simple colour bar using the convenience function plt.colorbar:

```
plt.colorbar()
```

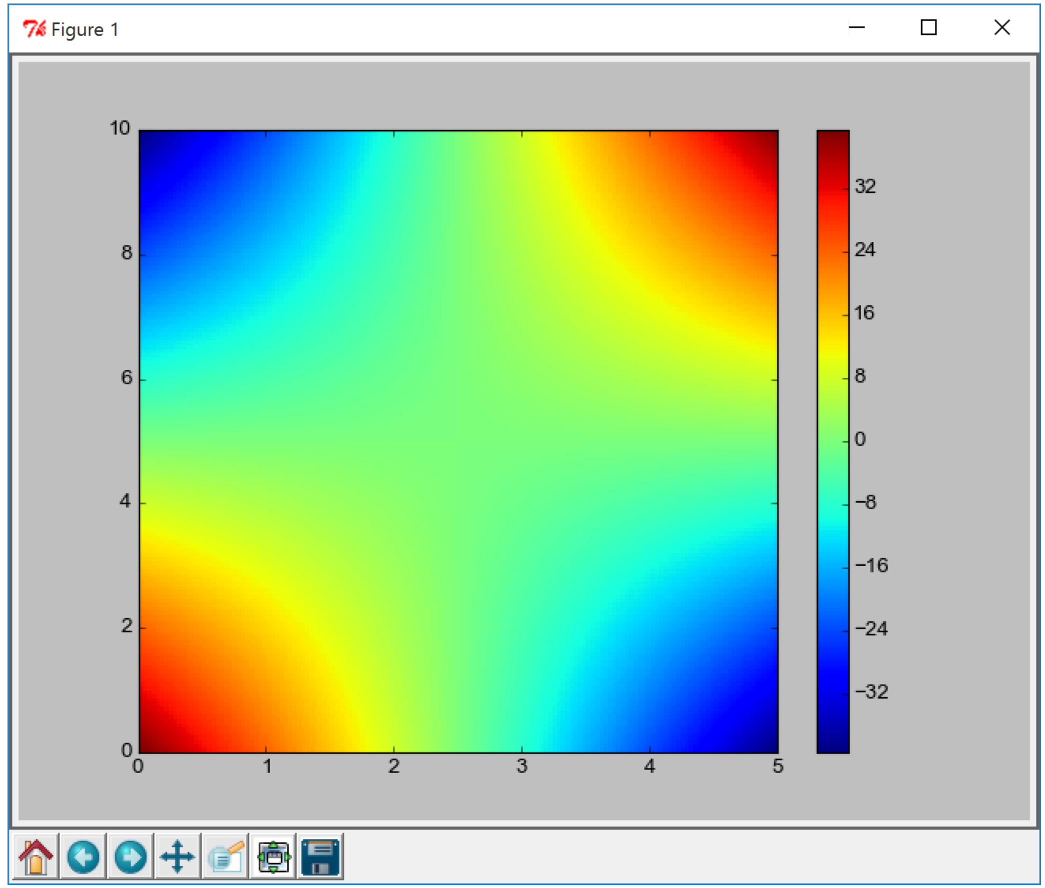
This will create a colour bar that takes some space from the current axis.



COLOUR BARS

If you have a specific location you want to put the colour bar, use the "cax" keyword

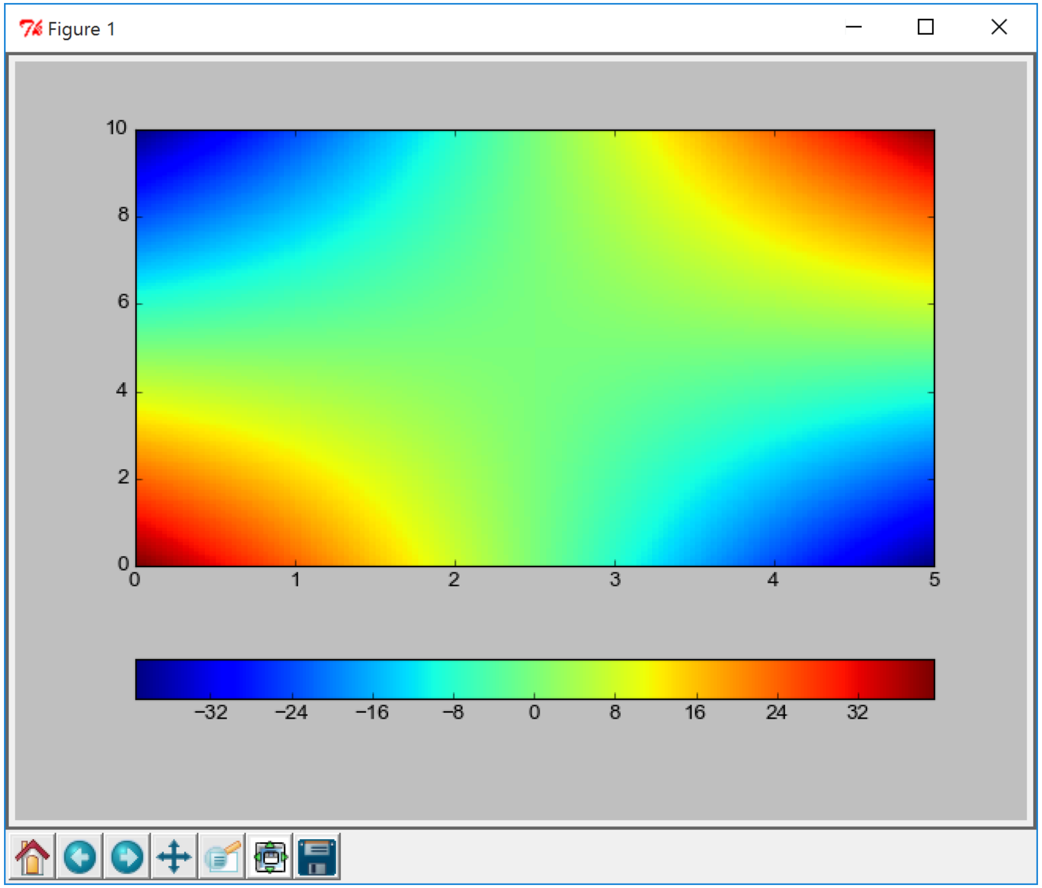
```
cbax =
fig.add_axes(loc)

plt.colorbar(
  cax=cbax
)
```



COLOUR BARS

You can choose to have the colour bar oriented horizontally as opposed to vertically:

```
plt.colorbar(
  orientation=
  “horizontal”
)
```

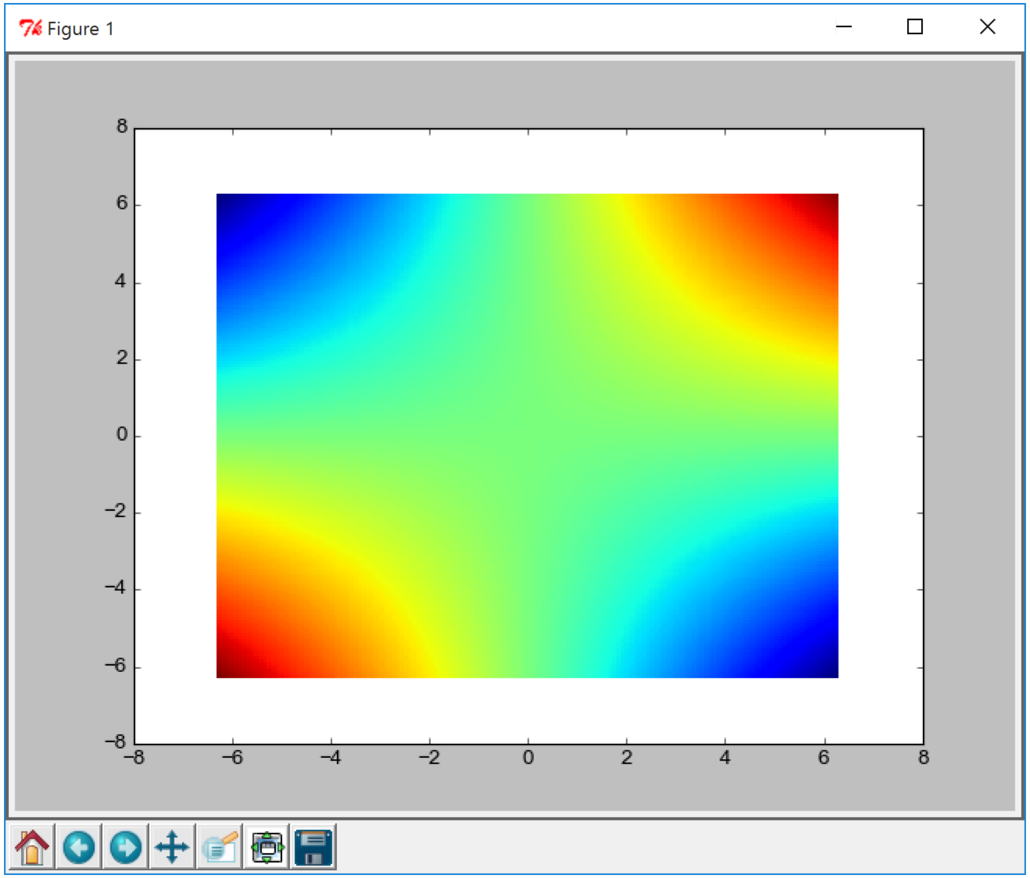


PCOLOR

If you don't want to worry about the orientation issues or have images with varying pixel sizes, you can using the pcolor function instead of imshow:

```
plt.pcolor(
  xvals, yvals,
  array
)
```

'xvals' and 'yvals' are arrays with the values of the x and y pixel edges.

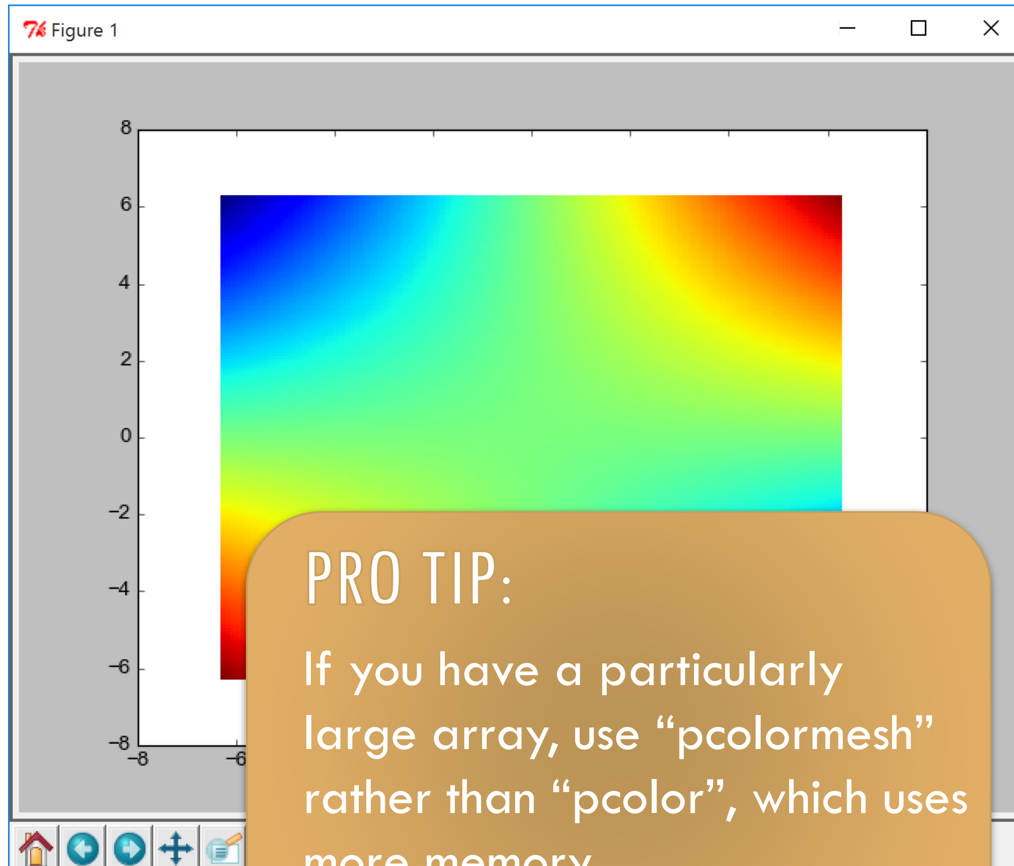


PCOLOR

If you don't want to worry about the orientation issues or have images with varying pixel sizes, you can using the pcolor function instead of imshow:

```
plt.pcolor(
  xvals, yvals,
  array
)
```

'xvals' and 'yvals' are arrays with the values of the x and y pixel edges.



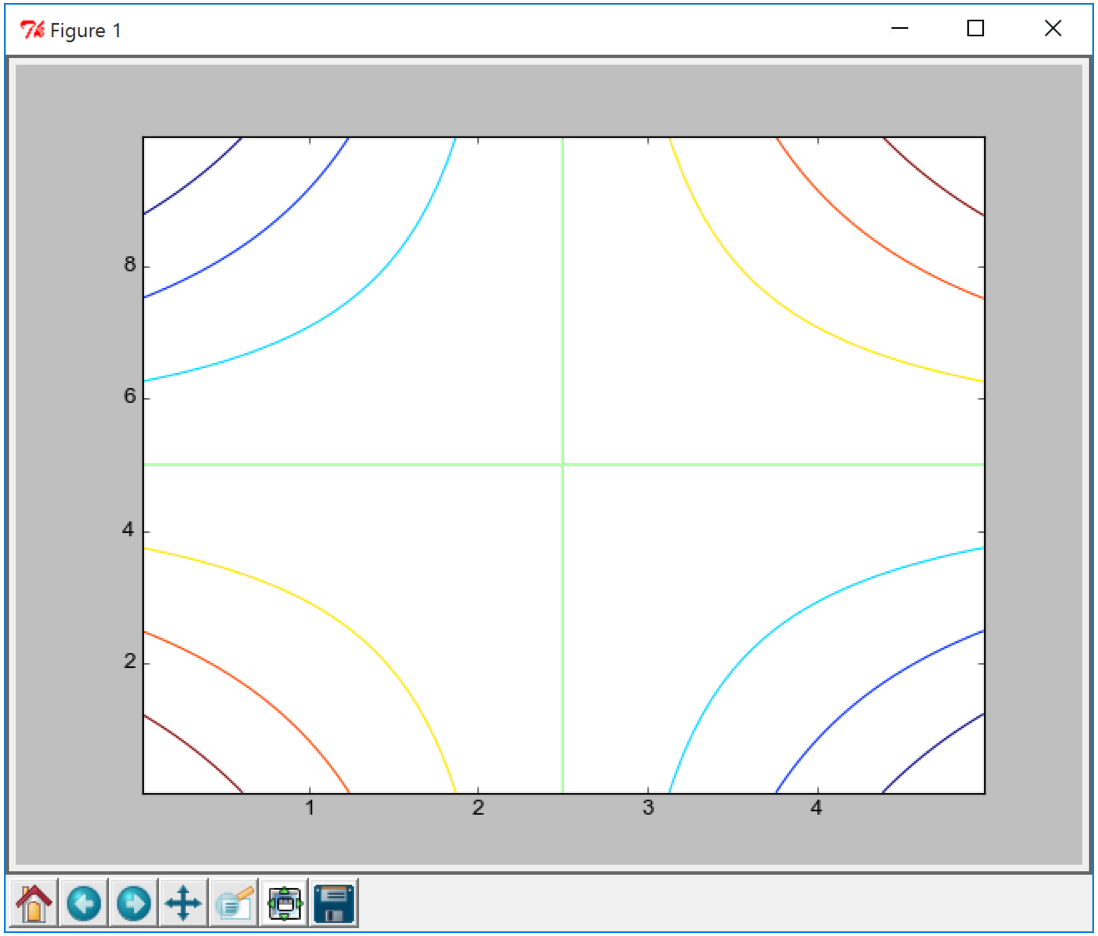
PRO TIP:

If you have a particularly large array, use "pcolormesh" rather than "pcolor", which uses more memory.

CONTOURS

Contours takes the same arguments as imshow, and by default produces contours with a jet colourmap:

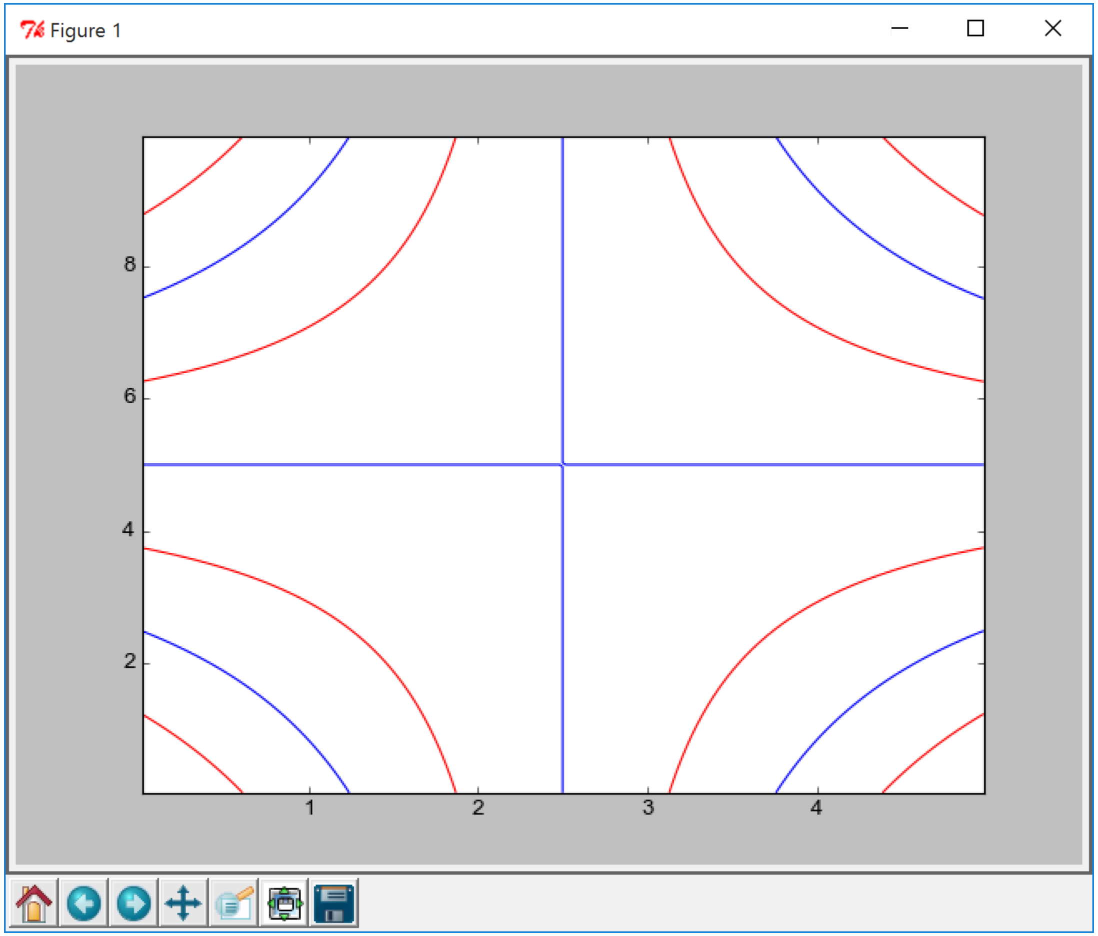
```
plt.contour(…)
```



CONTOURS

You can set the colour (or sequence of colours) of the contours (so that they are uniform):

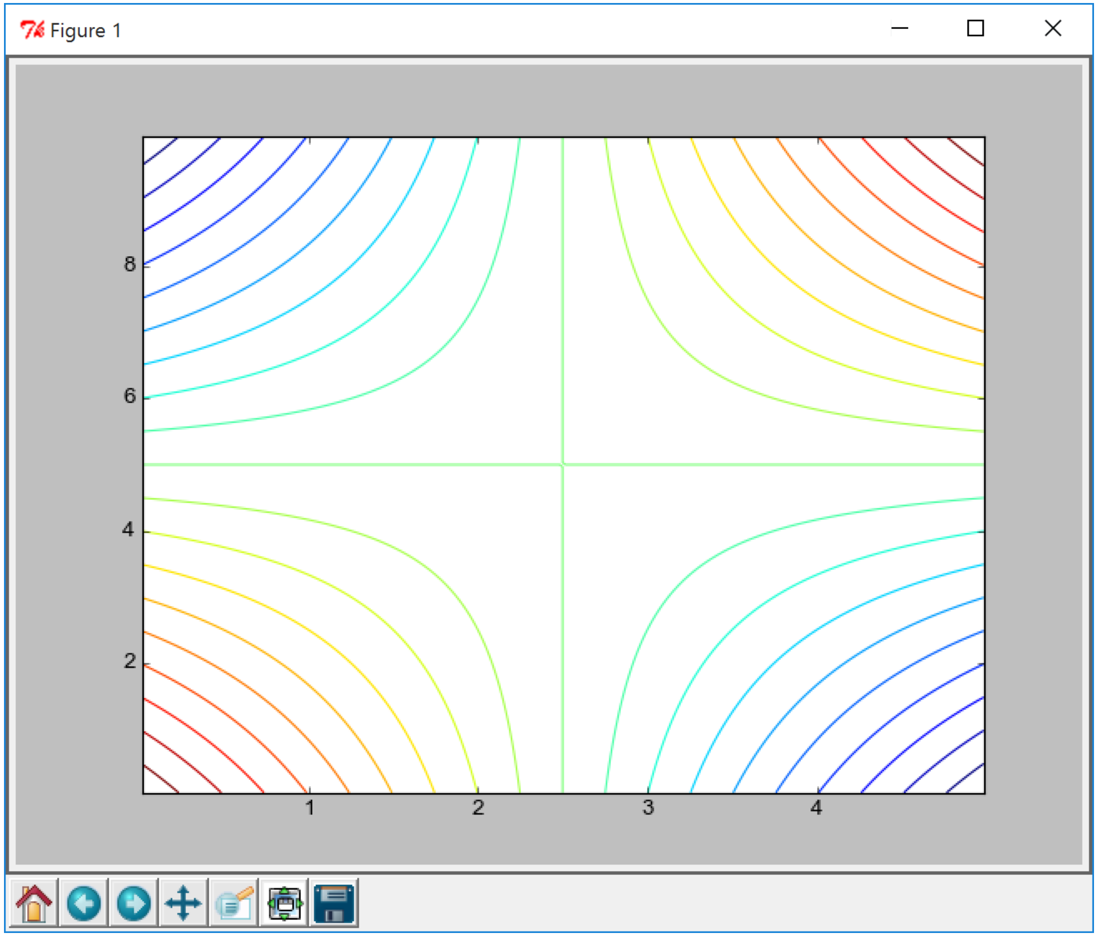
```
plt.contour(…,
 colors=(‘r’,’b’)
)
```



CONTOURS

Setting the number of contours:

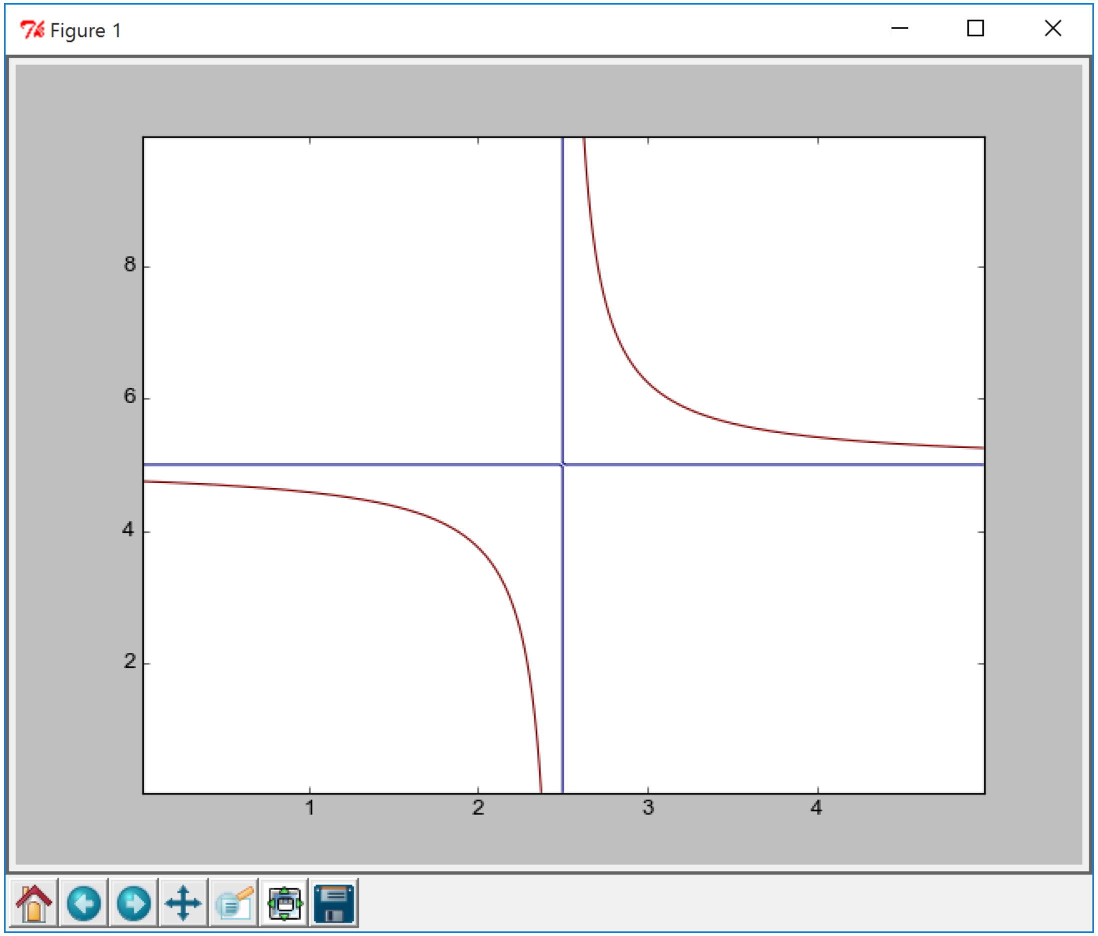
```
plt.contour(
  arr, 20, …
)
```



CONTOURS

Setting the specific location of the contours:

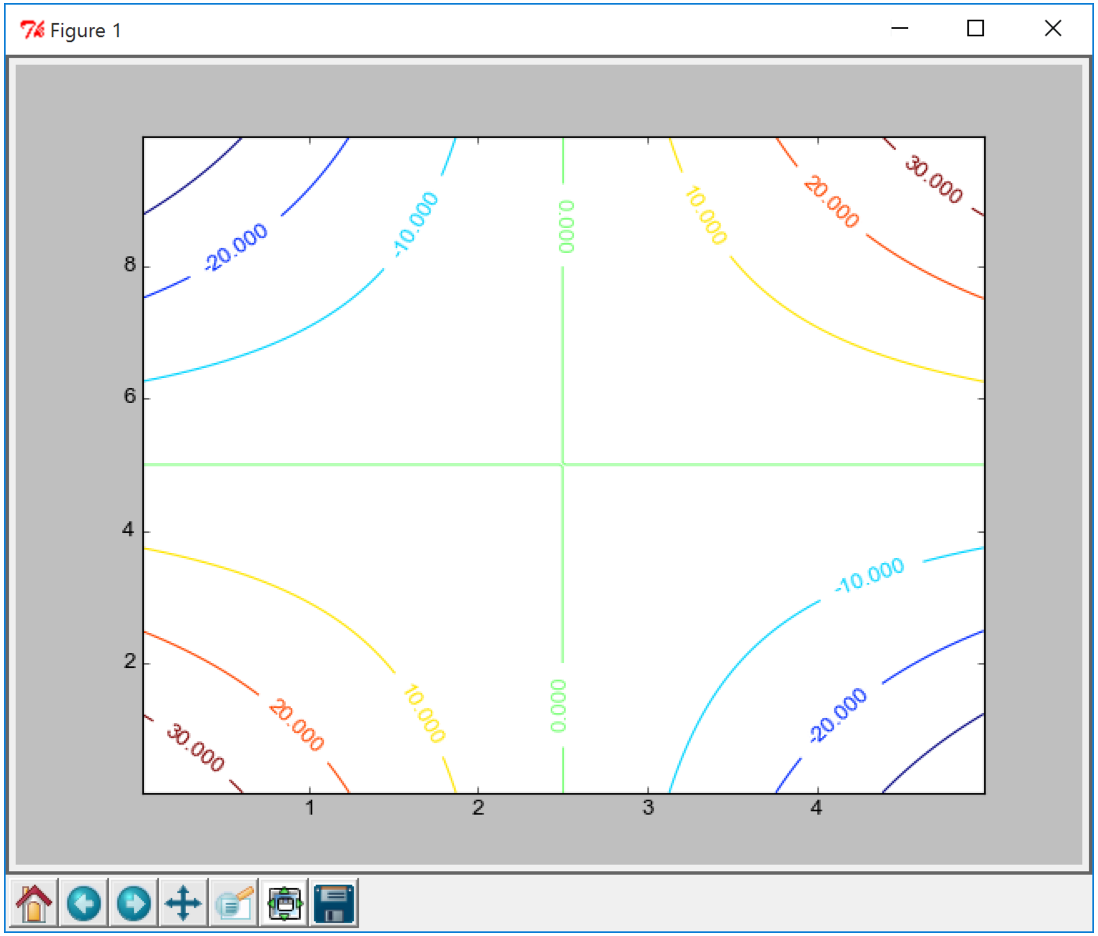
```
plt.contour(…,
  levels=[0, 2.0]
)
```



CONTOURS

You can set labels on the contours using the "clabel" function:

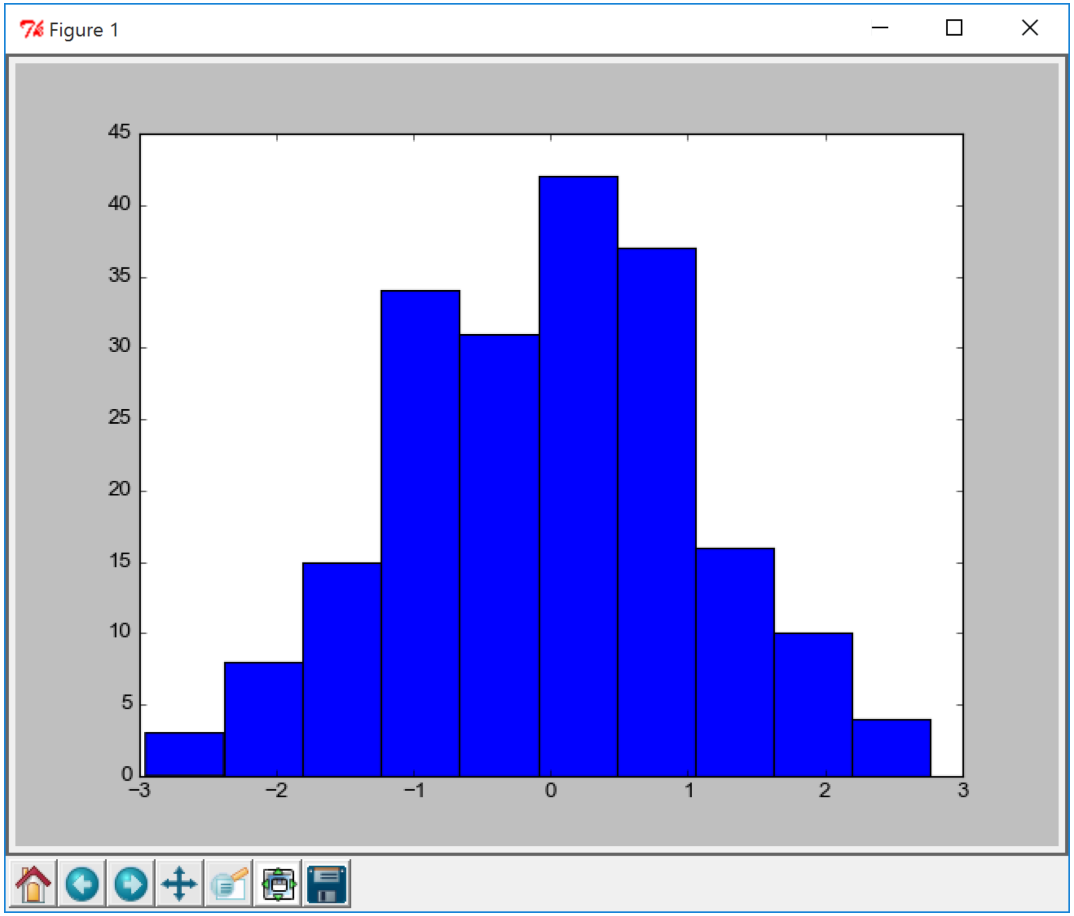
```
c1 =
plt.contour(…)
plt.clabel(c1)
```



HISTOGRAMS

Matplotlib also provides robust histogram capabilities:

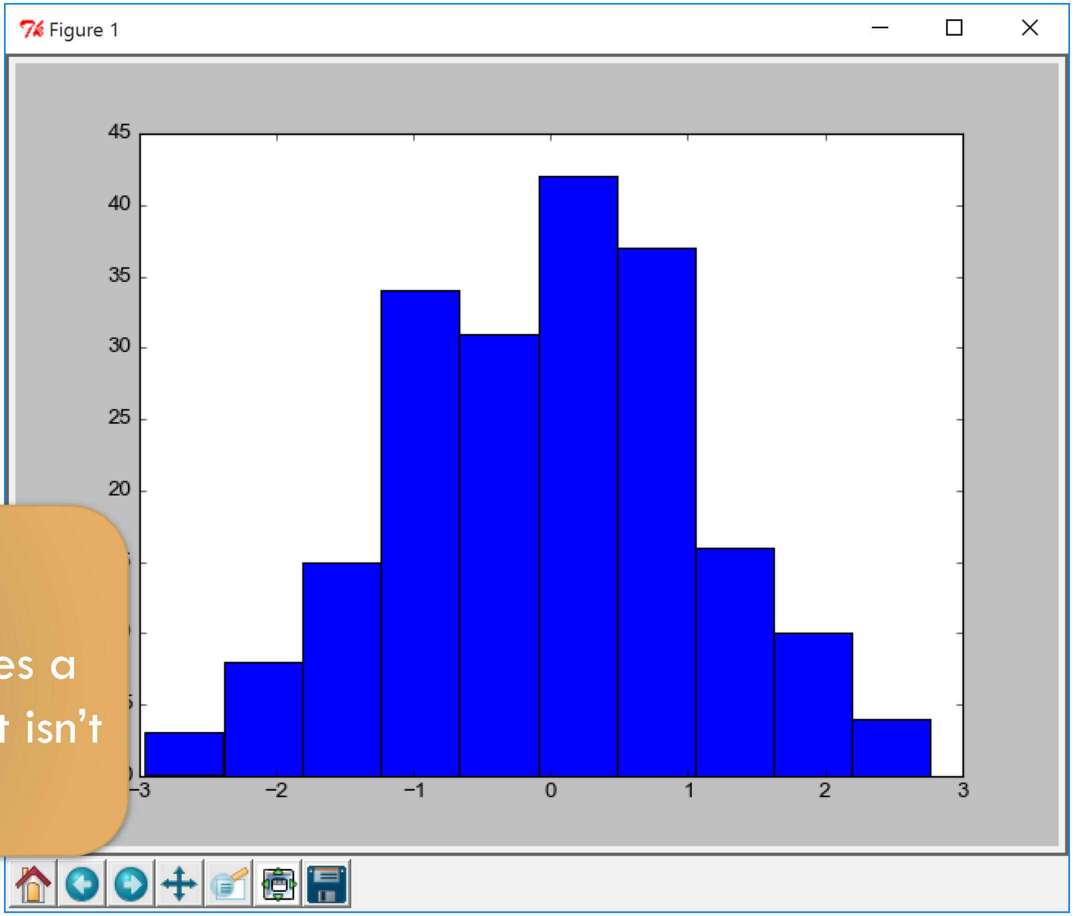
```
plt.hist(arr)
```



HISTOGRAMS

Matplotlib also provides robust histogram capabilities:

```
plt.hist(arr)
```



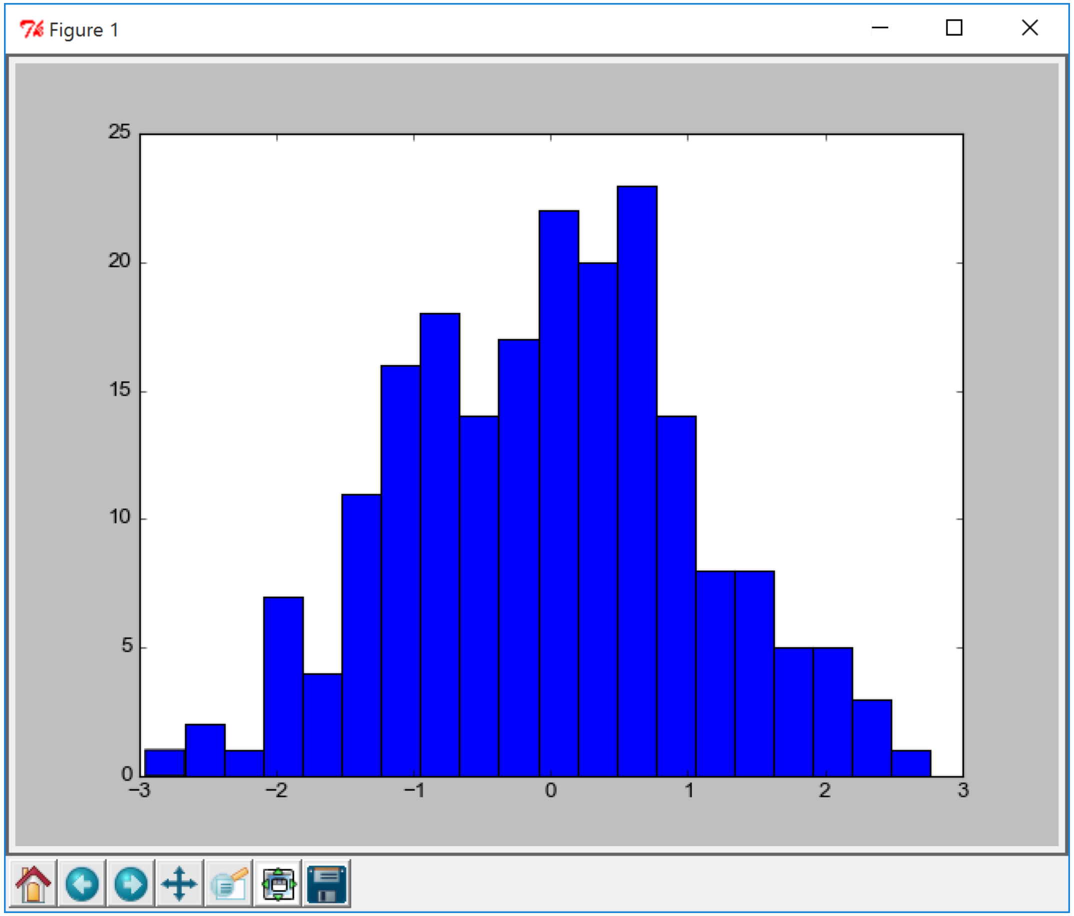
PRO TIP:

The histogram function takes a one-dimensional array. If it isn't already, flatten it!

HISTOGRAMS

Choosing the number of bins:

```
plt.hist(…,
  bins=20
)
```



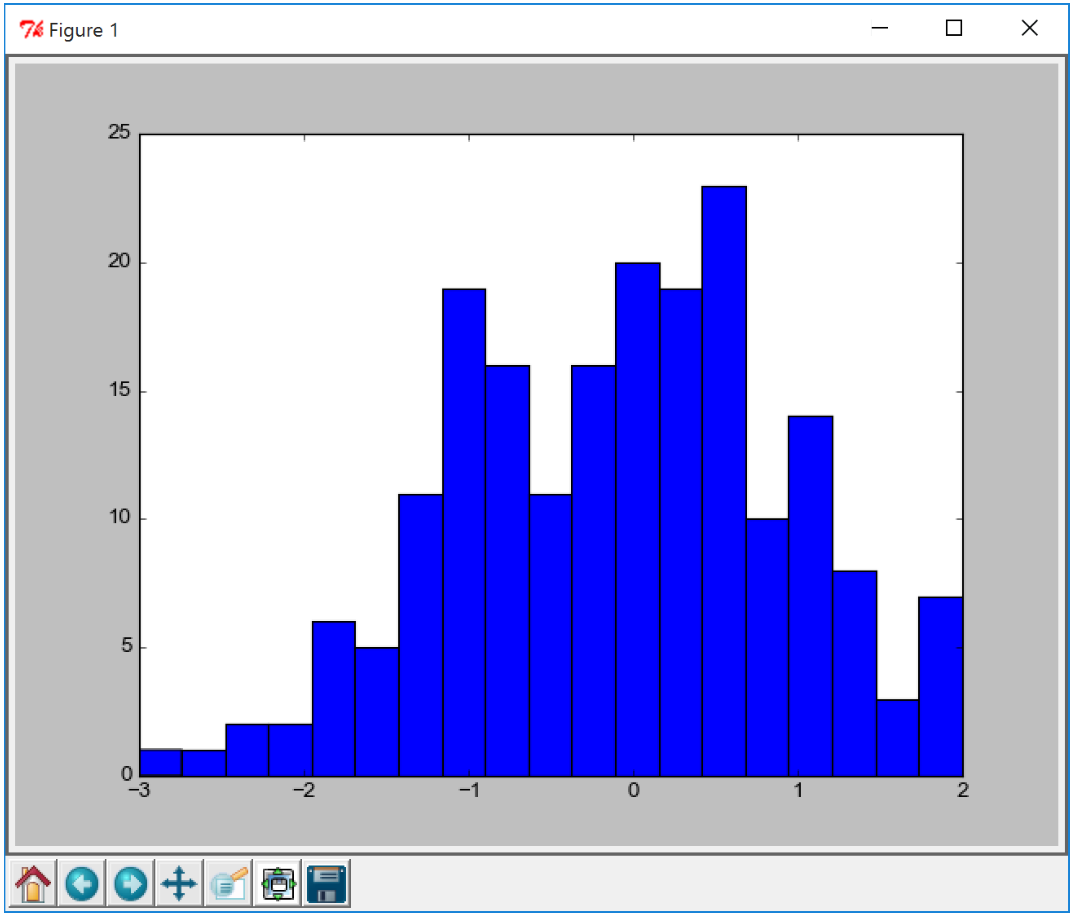
HISTOGRAMS

Choosing the number of bins:

```
plt.hist(…,
  bins=20
)
```

Or specific location of bin edges:

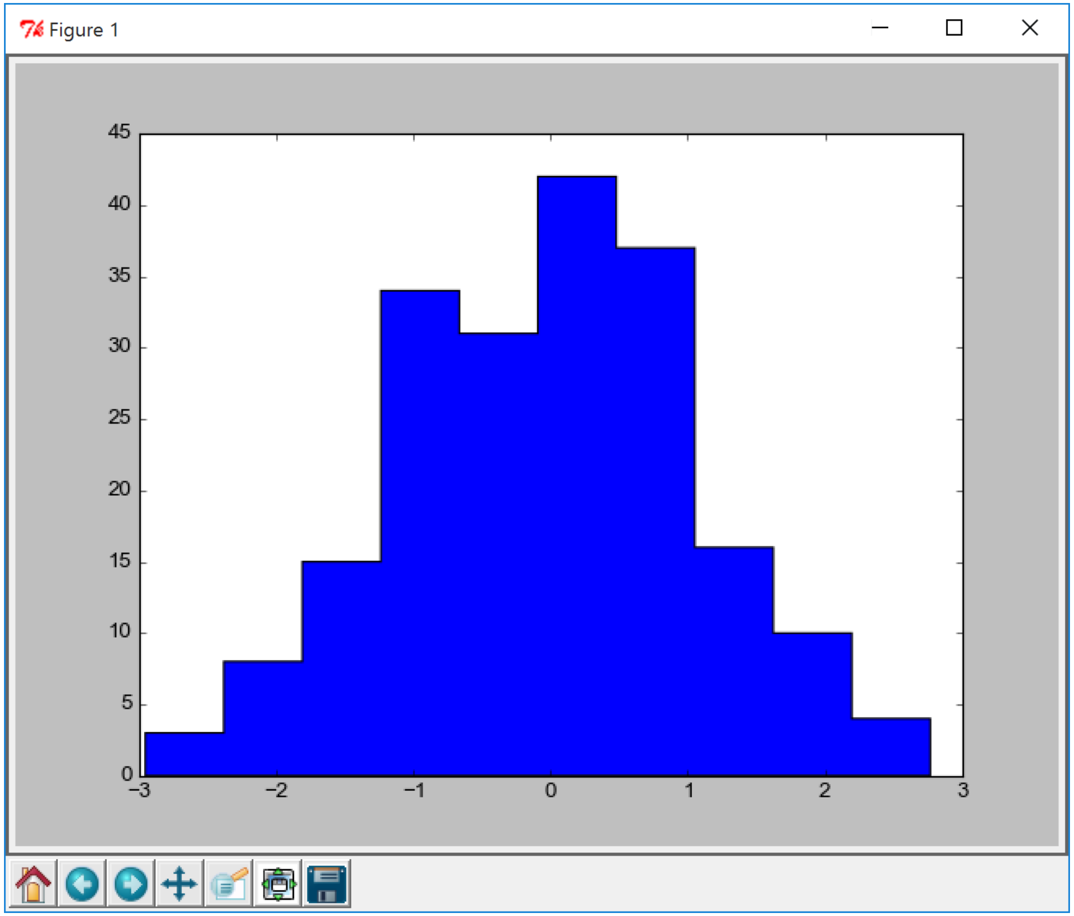
```
plt.hist(…,
  bins=bin_edges
)
```



HISTOGRAMS

Choosing steps instead of bars:

```
plt.hist(…,
  histtype=
  'stepfilled'
)
```

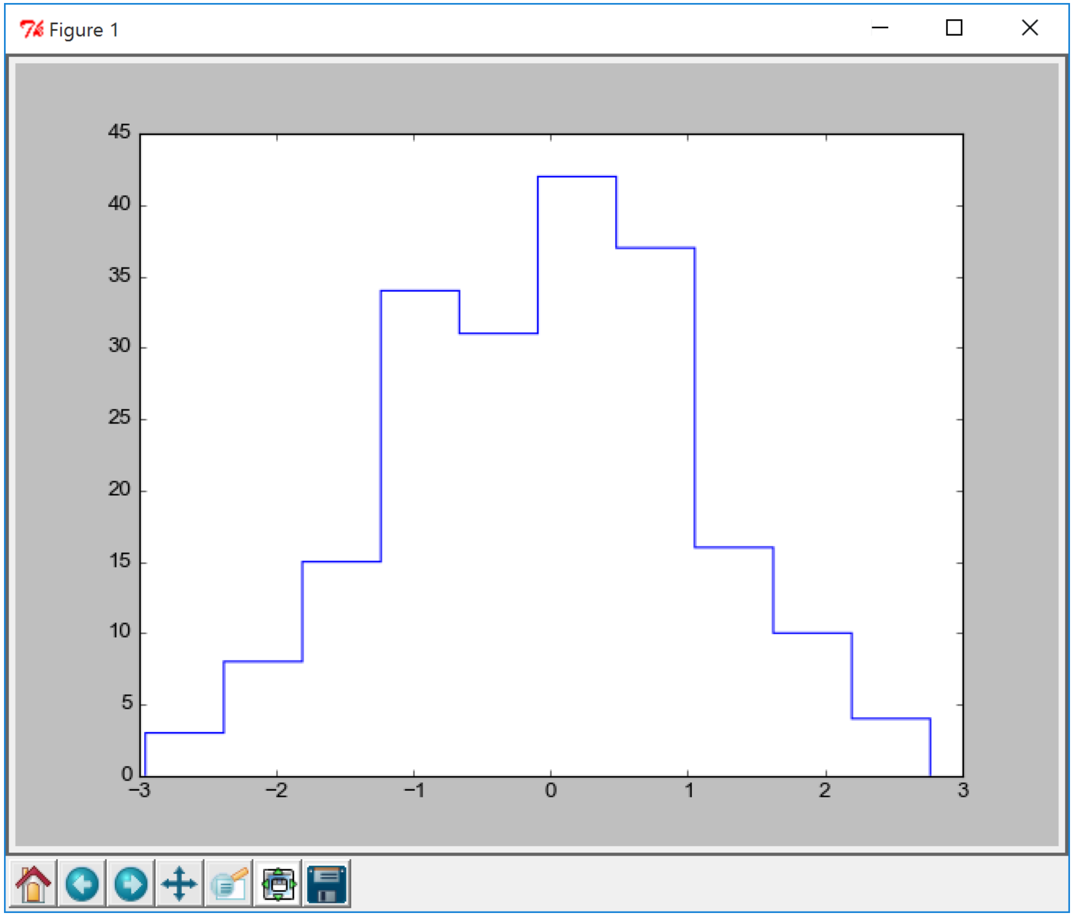


HISTOGRAMS

Or maybe you'd prefer just the line?

```
plt.hist(…,
  histtype=
  'step'
)
```

There is also a hist2d command that histograms 2D data into an image.

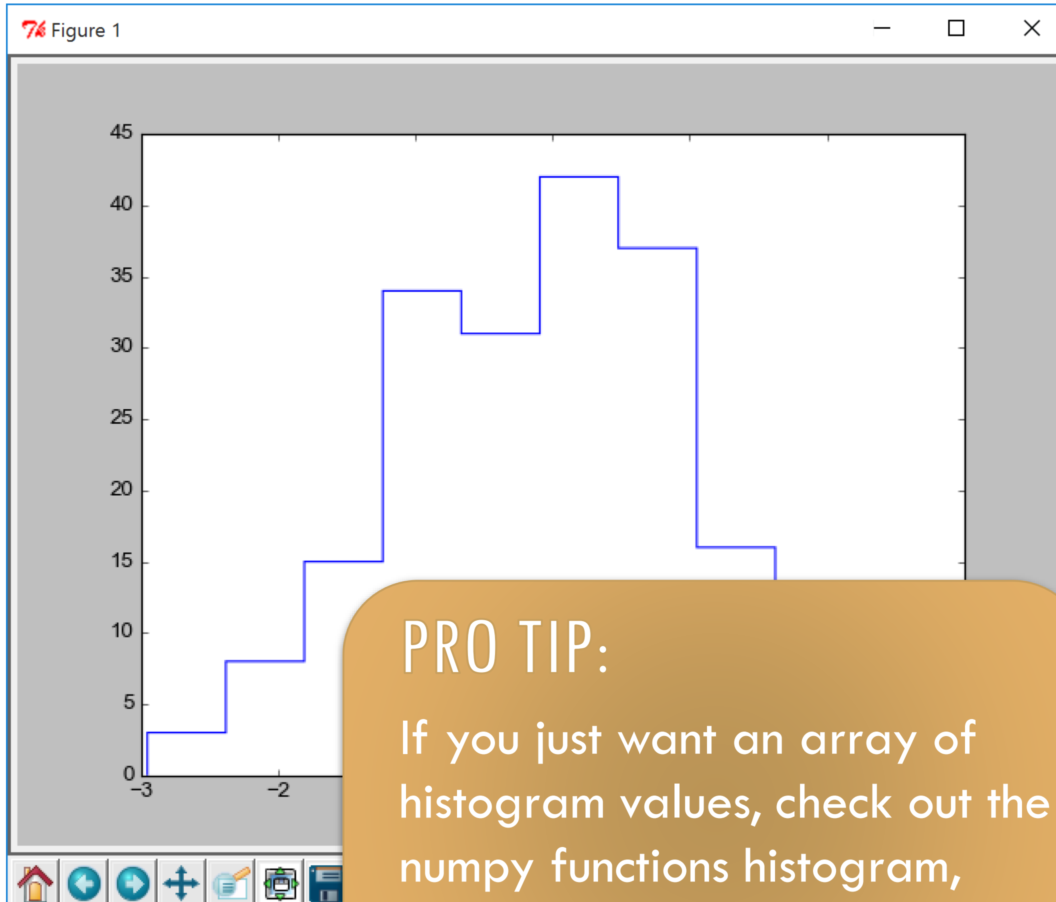


HISTOGRAMS

Or maybe you'd prefer just the line?

```
plt.hist(…,
  histtype=
  ‘step’
)
```

There is also a hist2d command that histograms 2D data into an image.



PRO TIP:

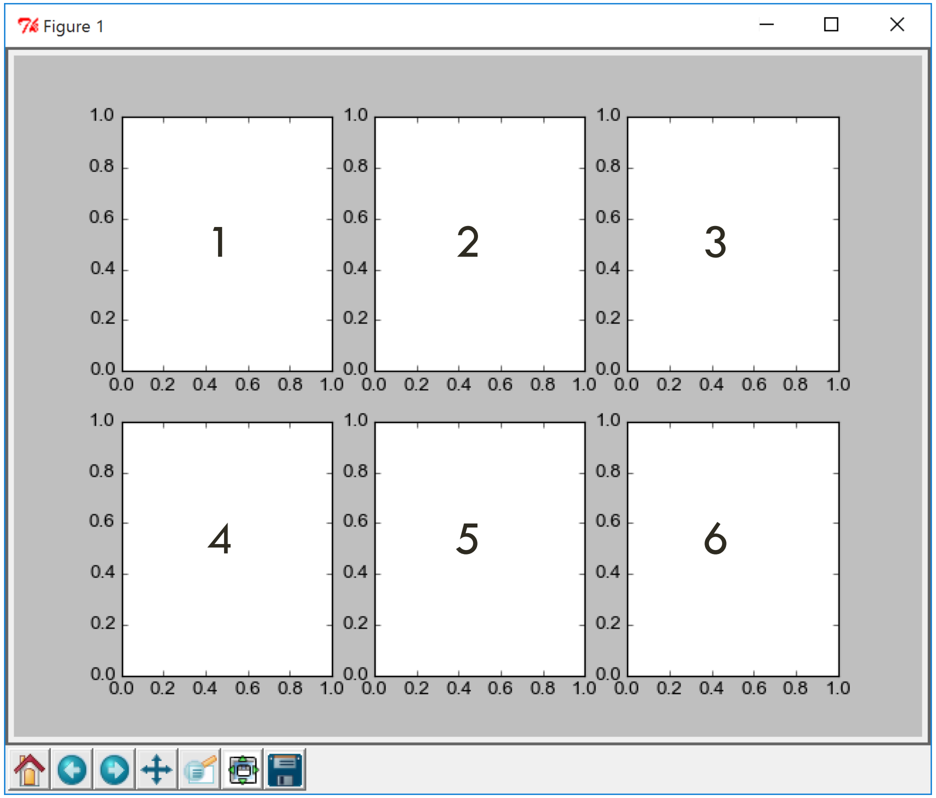
If you just want an array of histogram values, check out the numpy functions histogram, histogram2d, and histogramdd

SUBPLOTS/MULTIPLE PLOTS

Making subplots are quite easy using the convenience function “subplot”:

```
ax1 = plt.subplot(
  nrows, ncols,
  plotnum
)
```

plotnum starts at 1.

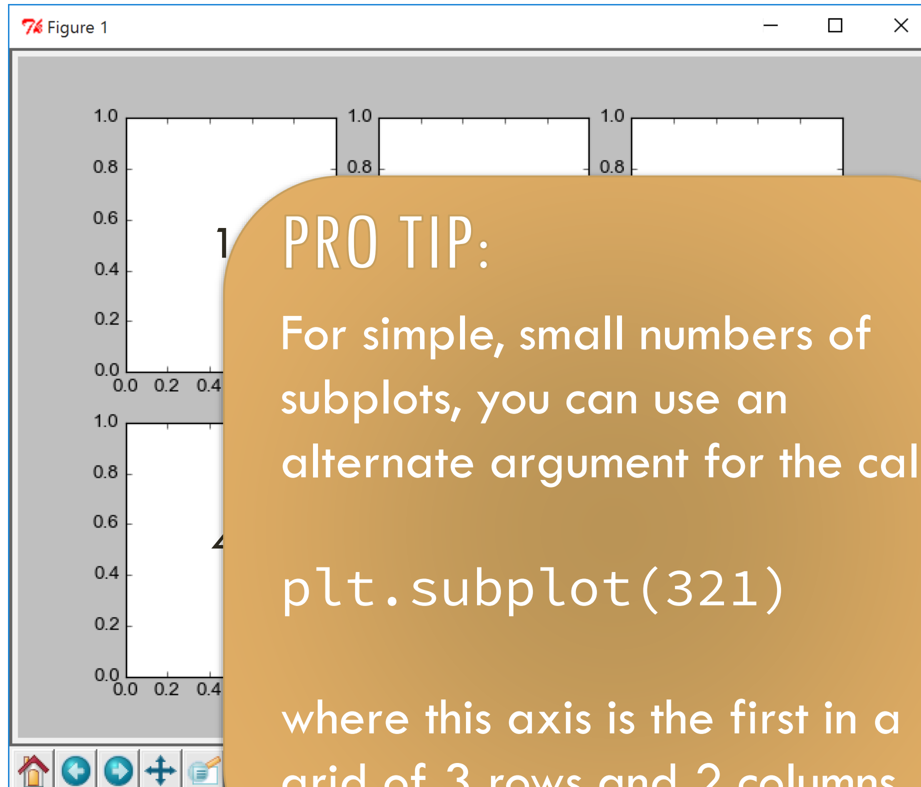


SUBPLOTS/MULTIPLE PLOTS

Making subplots are quite easy using the convenience function “subplot”:

```
ax1 = plt.subplot(
  nrows, ncols,
  plotnum
)
```

plotnum starts at 1.



PRO TIP:

For simple, small numbers of subplots, you can use an alternate argument for the call:

```
plt.subplot(321)
```

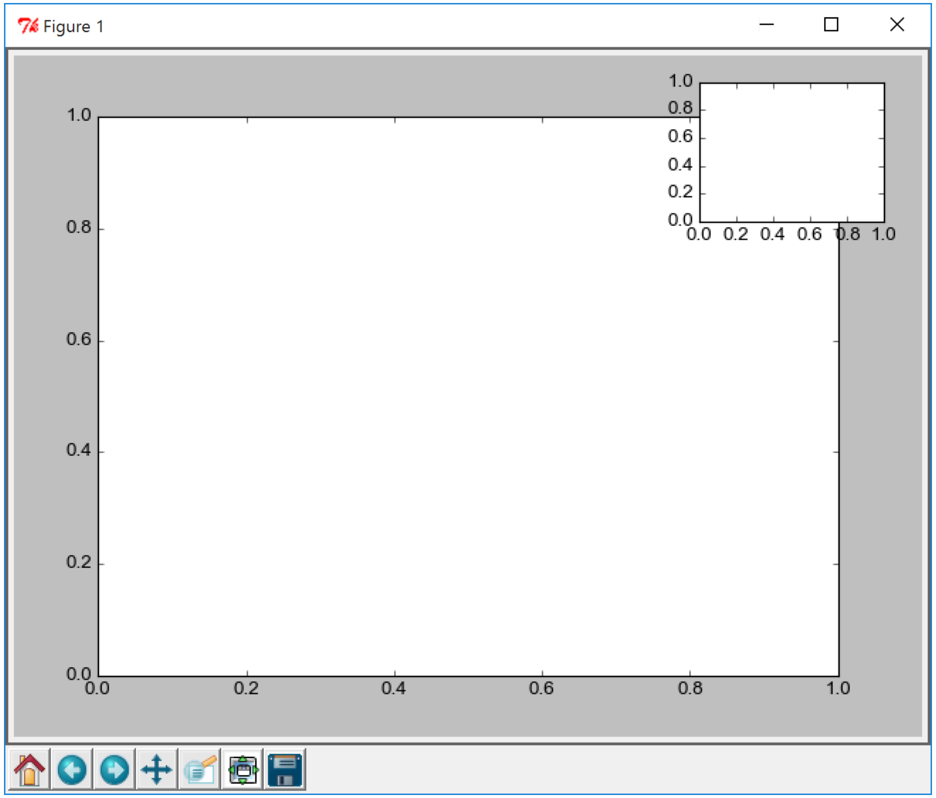
where this axis is the first in a grid of 3 rows and 2 columns.

SUBPLOTS/MULTIPLE PLOTS

More complicated plots can be made by adding specific axes:

```
ax1 = plt.axes(
[0.1, 0.1, 0.8, 0.8]
)

ax2 = plt.axes(
[0.75, 0.75, 0.2,
0.2]
)
```

I prefer this method.



ANNOTATIONS

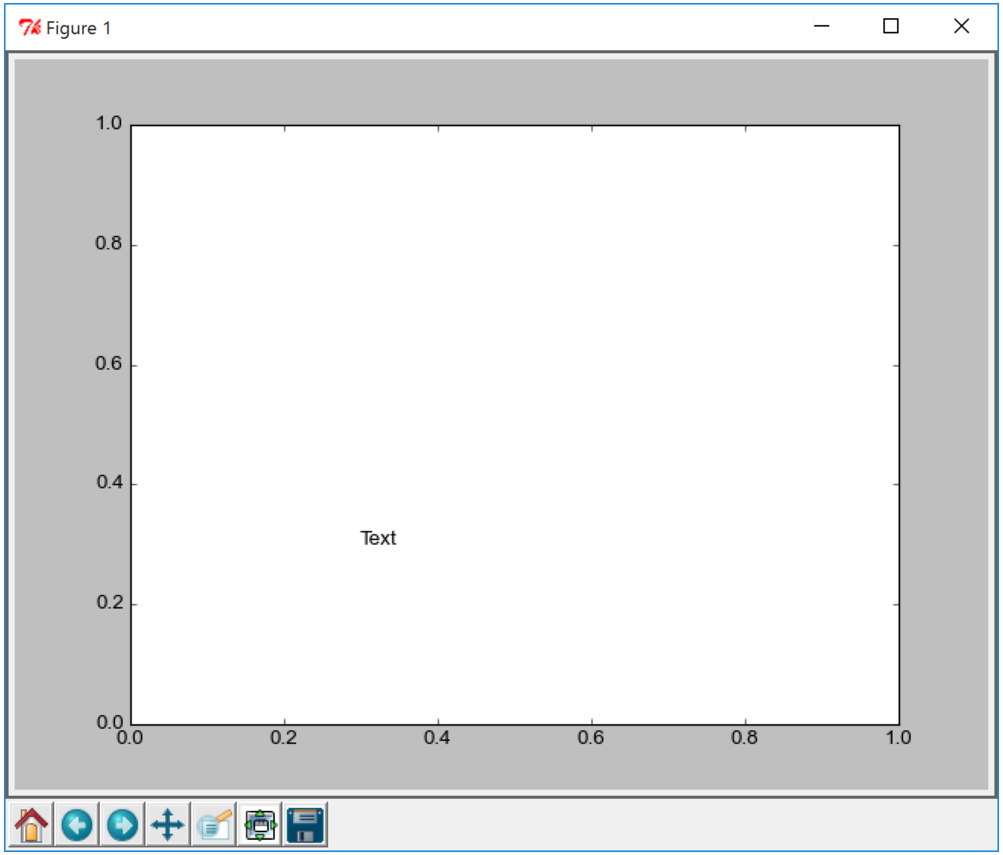
Adding text to axes is simple using the "text" command:

```
plt.text(
  x, y, "Text"
)
```

Or if adding to the figure:

```
plt.figtext(
  x, y, "Text"
)
```

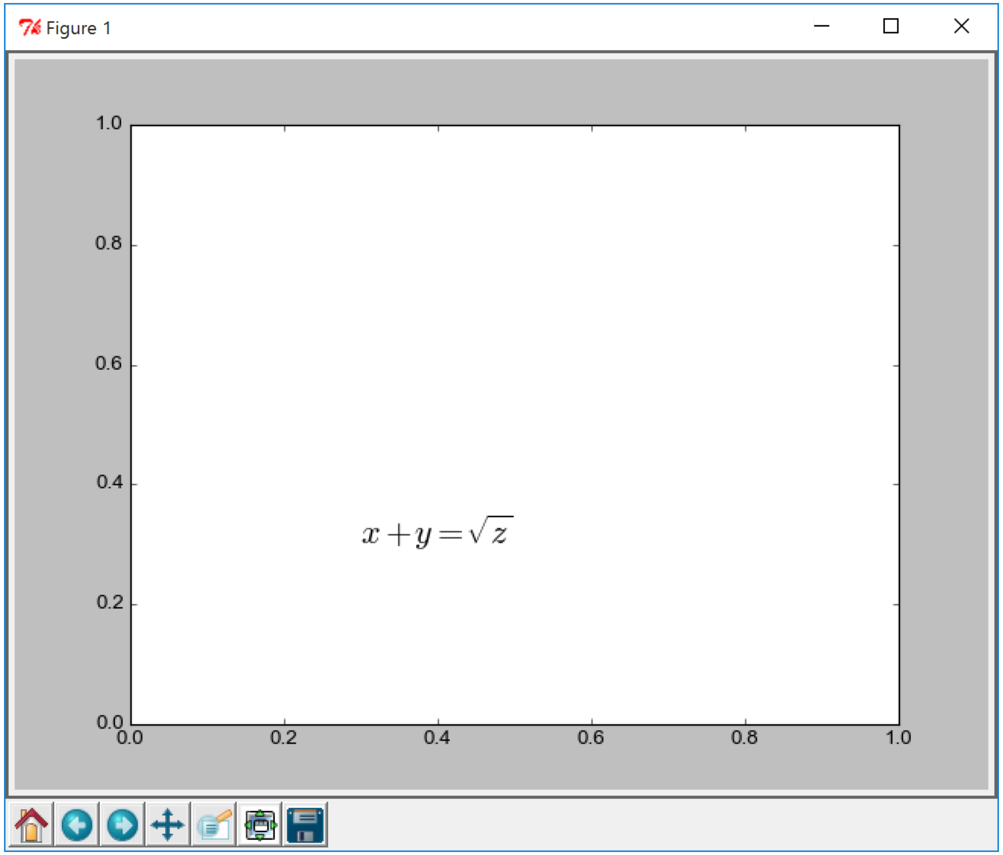
Where these coordinate go from 0 to 1 in fractions of the figure.



ANNOTATIONS

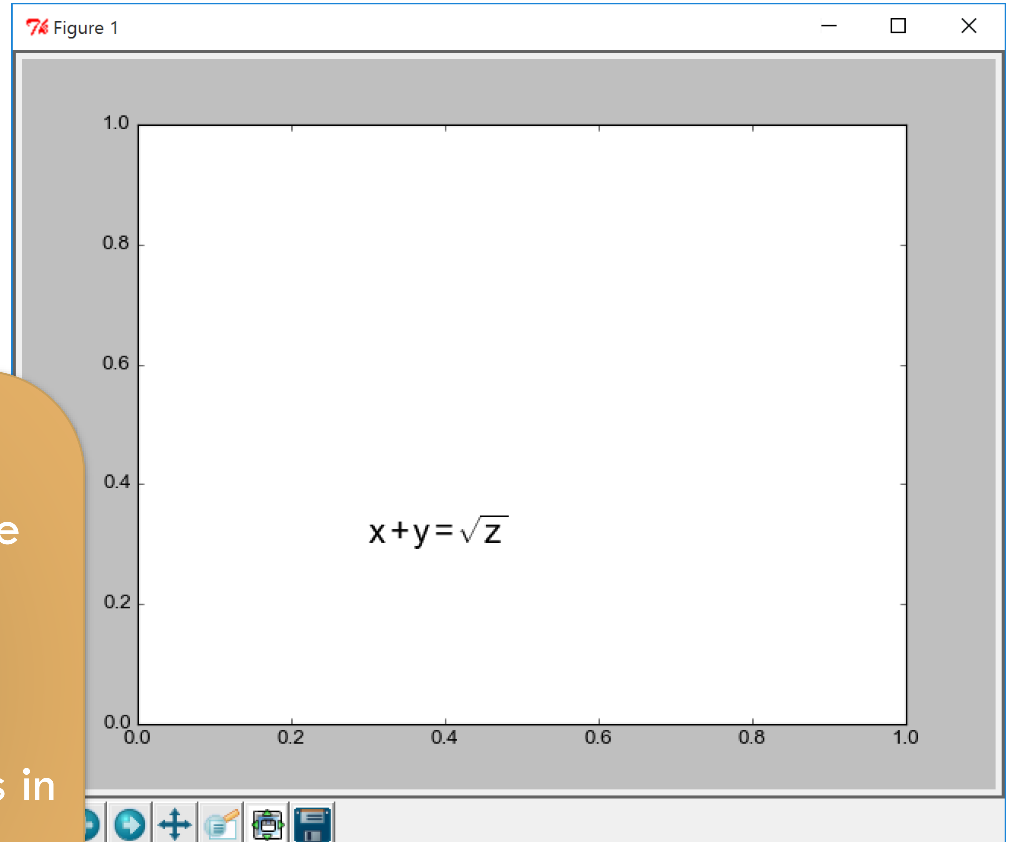
Anywhere you have text, you can use latex by enclosing the text in dollar signs ($)

```
plt.text(…
  “$x+y=\sqrt{z}$”
)
```



ANNOTATIONS

Anywhere you have text, you can use latex by enclosing the text in dollar signs ($)



PRO TIP:

If you want to avoid using the (*ugly*) computer modern font and just use whatever font you've set matplotlib to use, embed your latex commands in the `\mathdefault{…}` environment.

ANNOTATIONS: PATCHES

Adding additional shapes to the plot is called adding a "patch". There are a variety of patches available by importing:

```
from matplotlib import patches
```

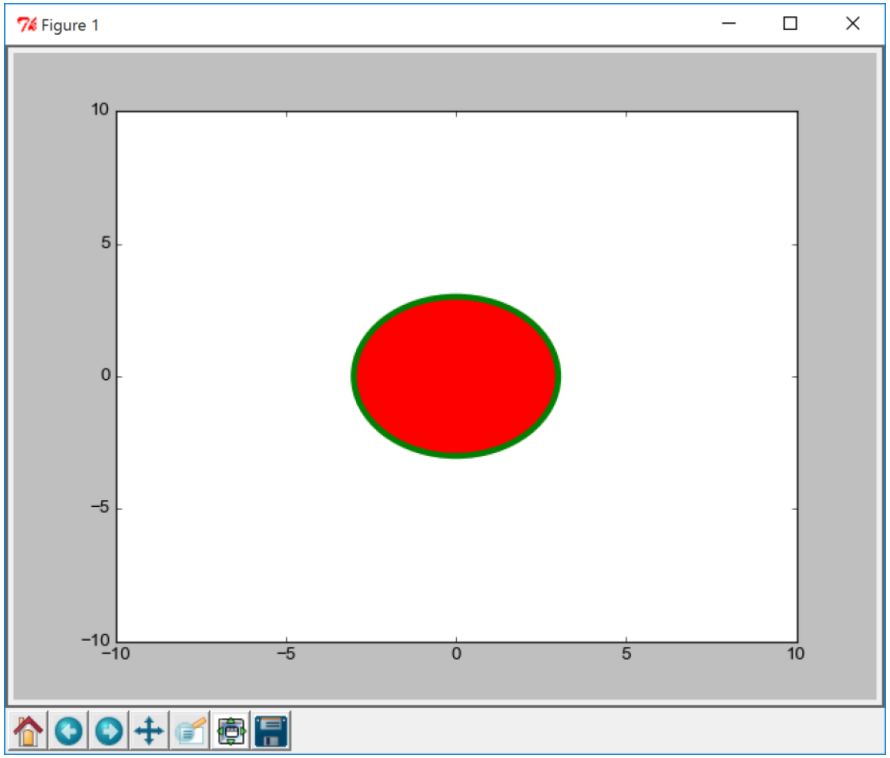
There are a large number of various patches, including Rectangles, Circles, Ellipses, and many more. Once a patch has been made using its declaration (i.e., `p1=patches.Circle(…)`), it needs to be added by:

```
ax1.add_patch(p1)
# Or if you haven't created a variable for your axis
plt.gca().add_patch(p1)
```

ANNOTATIONS: PATCHES

Looking at a 'Circle' patch:

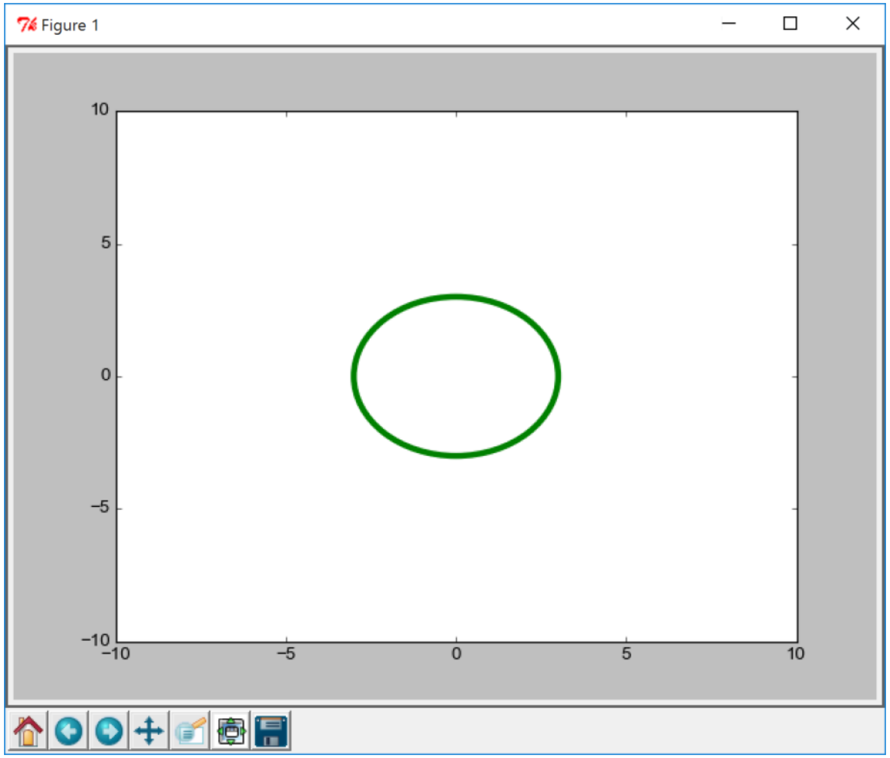
```
p1 = patches.Circle(
   (xloc, yloc),
   radius=3,
   edgecolor='g',
   facecolor='r',
   linewidth=4
)
```



ANNOTATIONS: PATCHES

Looking at a 'Circle' patch:

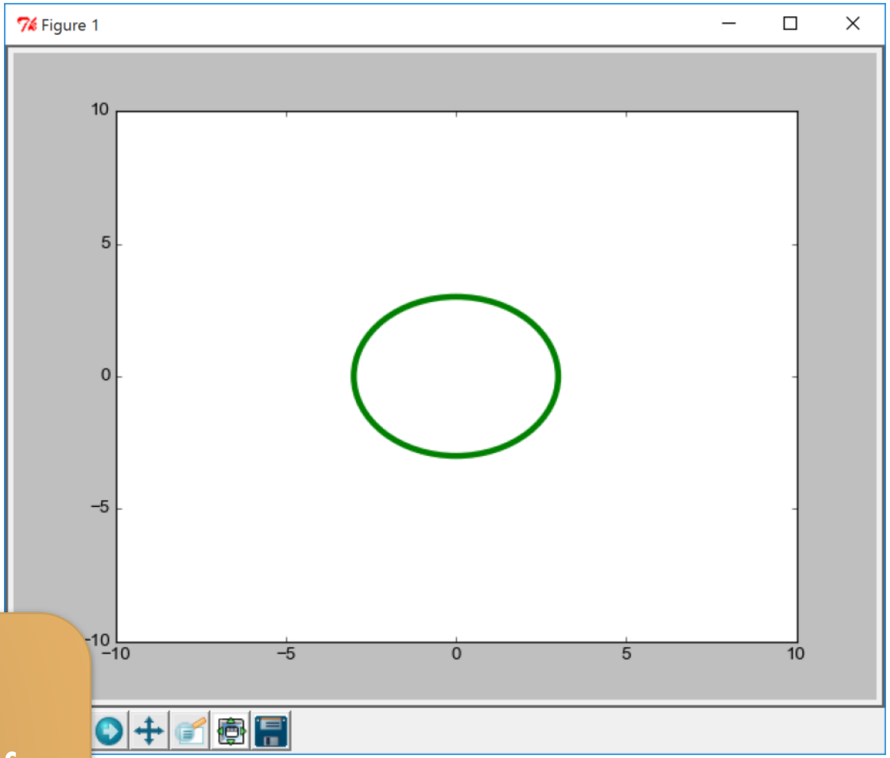
```
p1 = patches.Circle(
   (xloc, yloc),
   radius=3,
   edgecolor='g',
   facecolor='None',
   linewidth=4
)
```



ANNOTATIONS: PATCHES

Looking at a 'Circle' patch:

```
p1 = patches.Circle(
   (xloc, yloc),
   radius=3,
   edgecolor='g',
   facecolor='None',
   linewidth=4
)
```



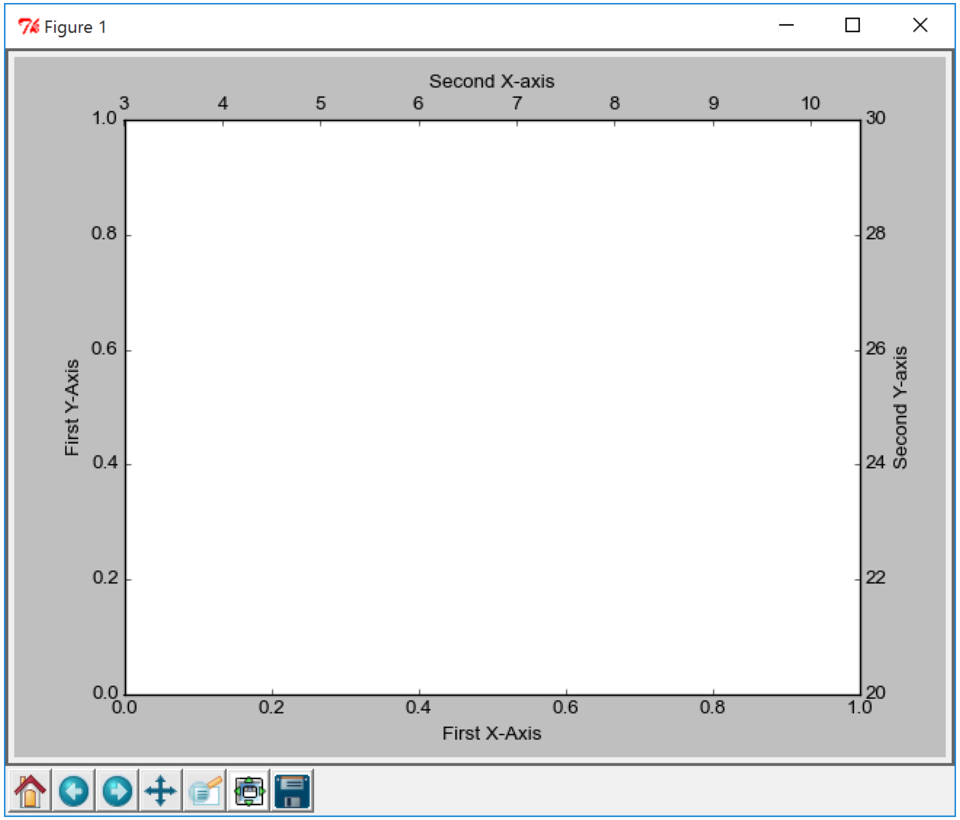
PRO TIP:

Circles will only look circular if the aspect ratio of the axis is 1

MULTIPLE AXES ON A SINGLE PLOT

You can create a second x or y axis on the same plot (which will be shown either on the top or the right) using the twinx or twiny methods:

```
ax2 = ax1.twinx()
ax3 = ax1.twiny()

ax2.set_ylim(20,30)
ax3.set_xlim(3,10.5)
```



EXERCISE TIME!

At least I can say that I've tried.