ADVANCED PLOTTING

DATA 601
Lecture 03
Part II

MATPLOTLIB REDUX

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

NUMPY.RANDOM & DISTRUBUTIONS

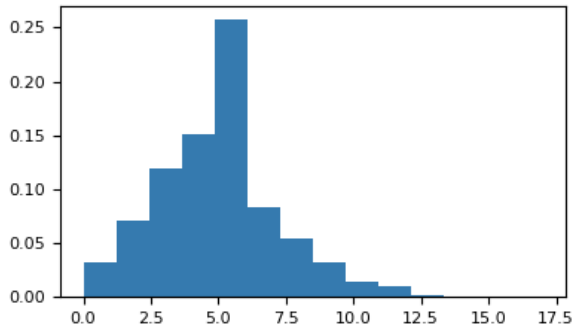
There are tens of different distribution options available in Numpy

Normal

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Poisson

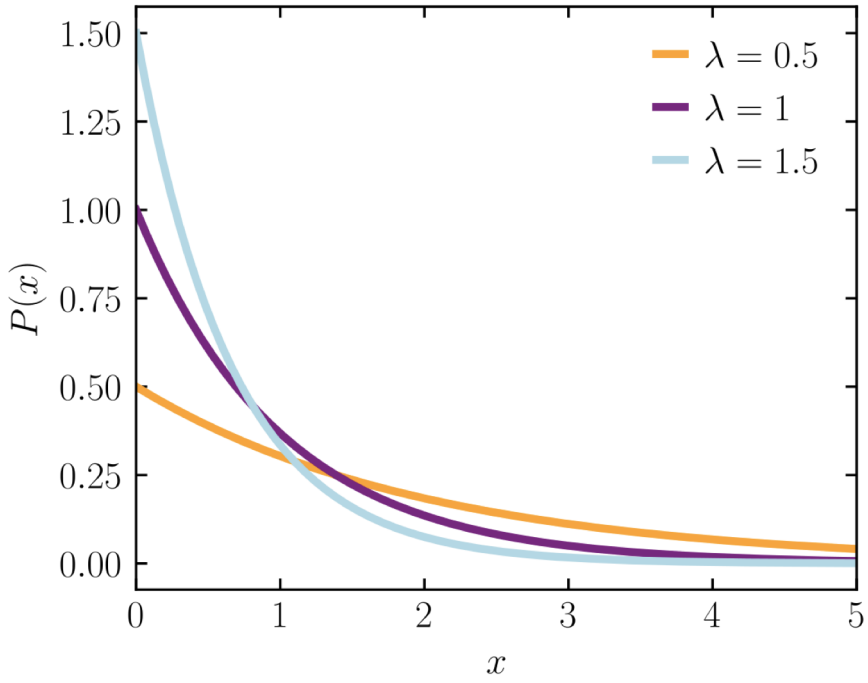
$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$



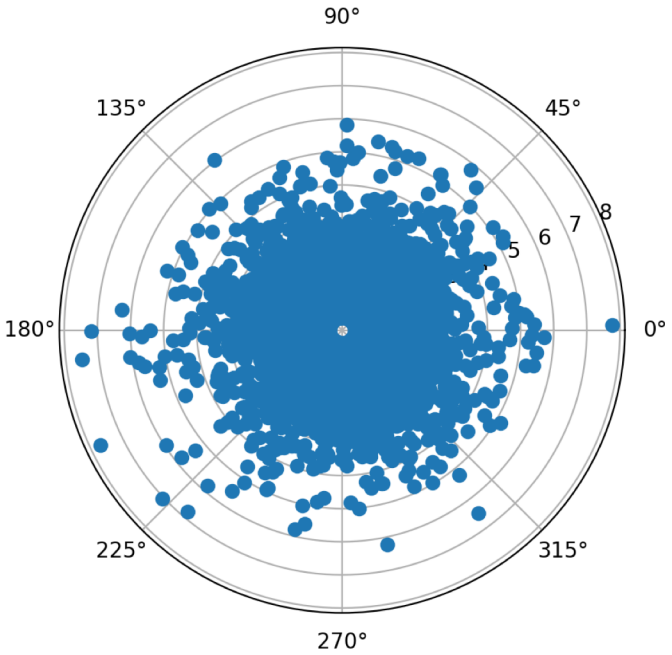
NUMPY.RANDOM & DISTRUBUTIONS

Exponential

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta}\exp(-\frac{x}{\beta})$$

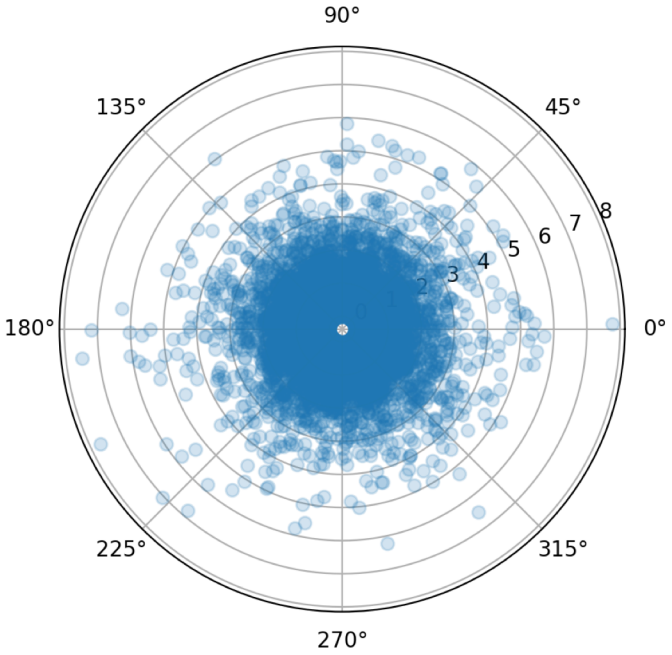


ALPHA/TRANSPARENCY




```
import numpy as np
import matplotlib.pyplot as plt

# Cluster data
a = np.random.uniform(0, 2*np.pi, size=10000)
r = np.random.exponential(size=10000)

# Make a polar plot.
plt.polar(a, r, linestyle='none', marker='o')
```

ALPHA/TRANSPARENCY



PRO TIP:

Saving to EPS doesn't support transparency.

```
# Alpha controls transparency
# 0 = transparent
# 1 = opaque

# Make markers transparent.
plt.clf()
plt.polar(a, r, linestyle='none', marker='o', alpha=0.2)
```

IMAGES

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

origin

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

origin

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.**

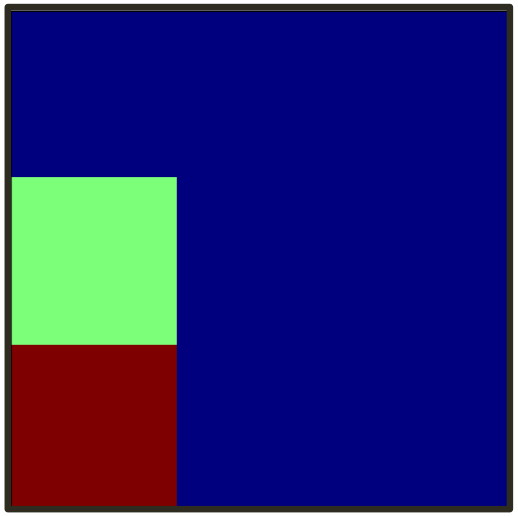
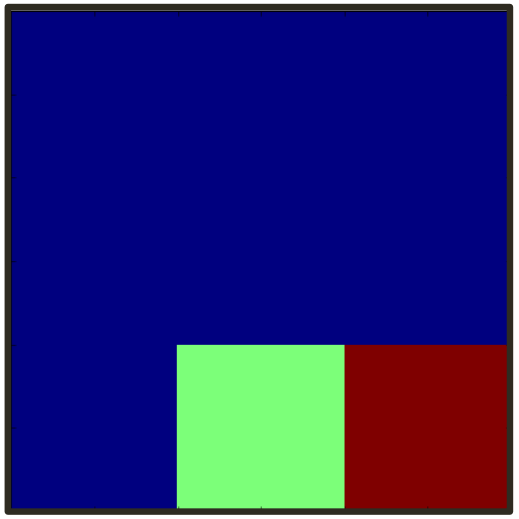


Image Coordinates



Cartesian Coordinates

NUMPY OUTER FUNCTION

FORMAT: numpy.outer(a, b, out=None)

FUNCTION: Compute the outer product of two vectors.

Given two vectors, a = [a0, a1, ..., aM] and b = [b0, b1, ..., bN], the outer product

[[a0*b0 a0*b1 ... a0*bN]

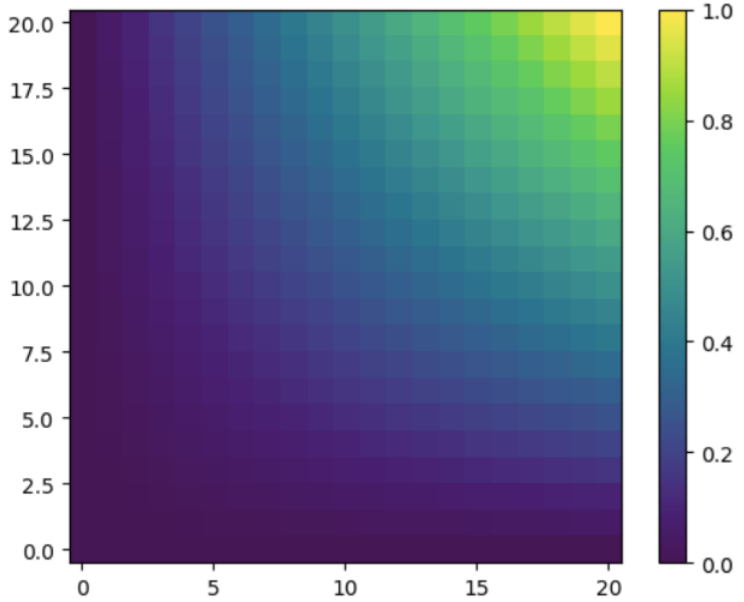
[a1*b0 .

[... .

[aM*b0 aM*bN]]

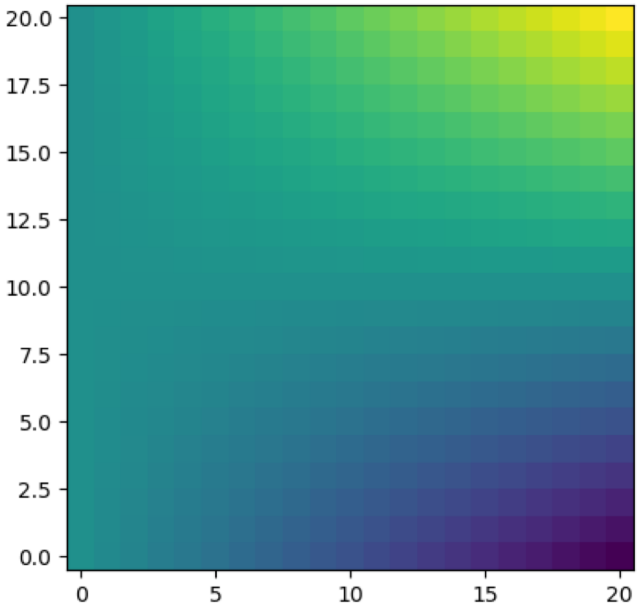
IMSHOW

```
# Imshow is the go-to image plotting
# function in matplotlib.

# Small parabolic image: z=x*y

# Show it.
plt.clf()
plt.imshow(z)

# But this is probably not quite
# what you want... Remember the
values of x and y, compare them to
the right ➔
```



IMSHOW

```
# Manually convert to Cartesian
plt.clf()
plt.imshow(z[:,::-1].T)

# Or do it the easy way...
plt.clf()
plt.imshow(z.T, origin='lower')
```

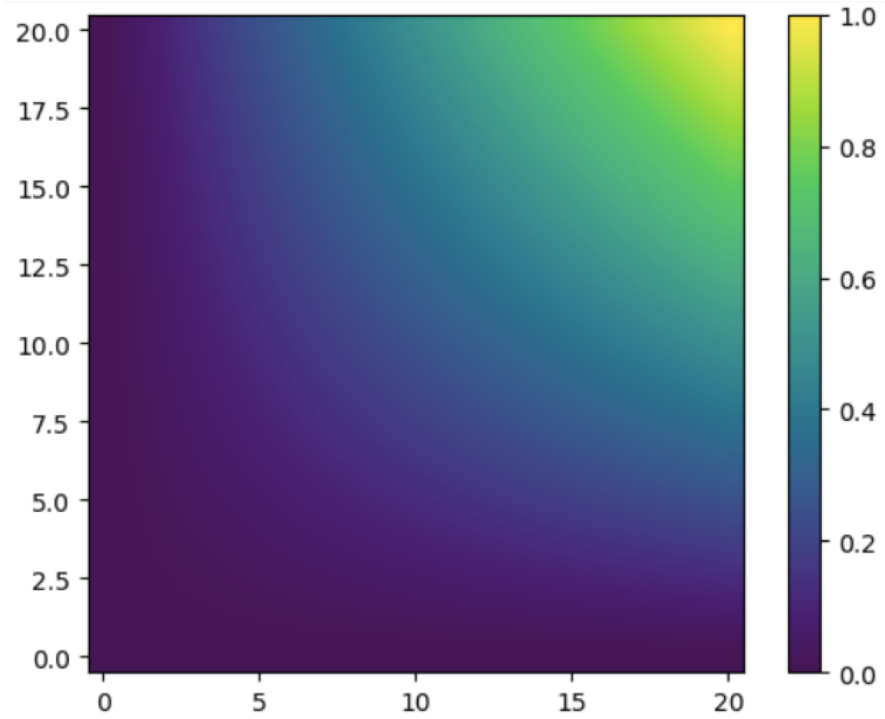


PRO TIP:

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

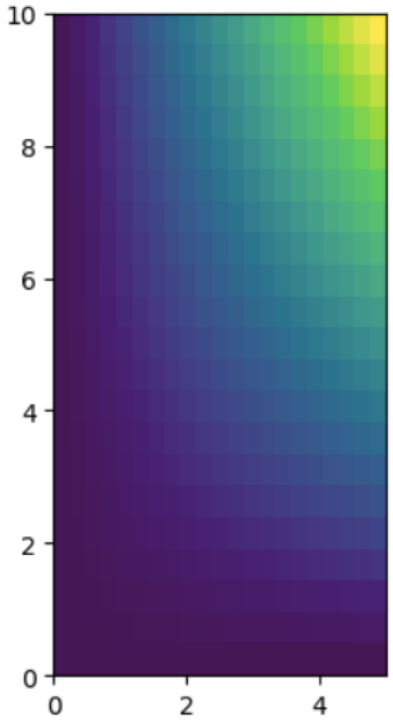
IMSHOW

```
# Smooth with bilinear
# interpolation.
plt.clf()
plt.imshow(z.T, origin='lower', interpolation='bilinear')
```



IMSHOW

```
# Pixels centered on pixel number.
# Change the ranges to stretch.
plt.clf()
plt.imshow(z.T, origin='lower', extent=[0, 5, 0, 10])
```

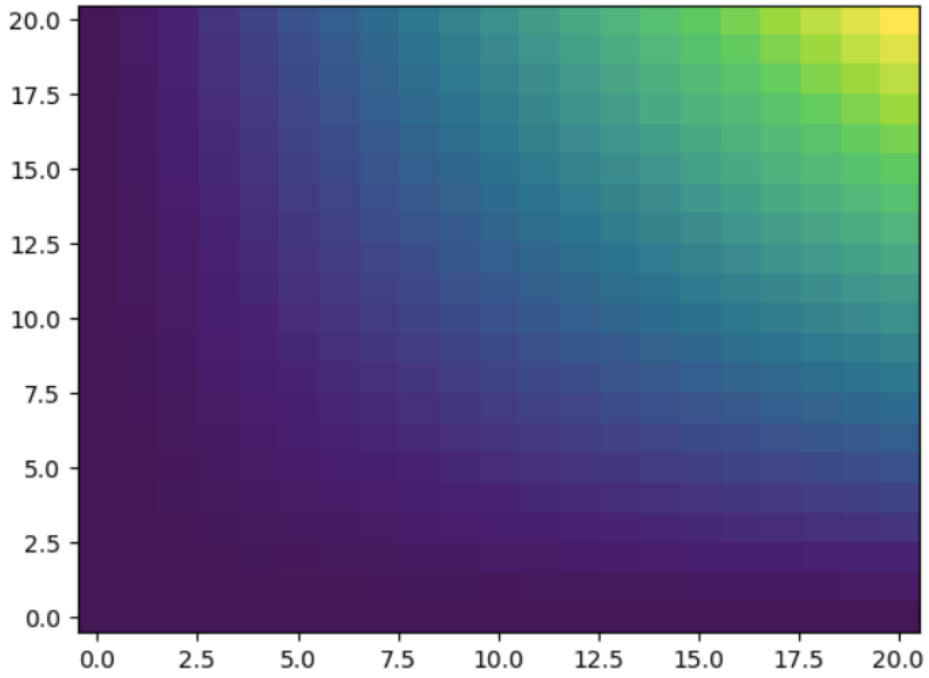


PRO TIP:

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

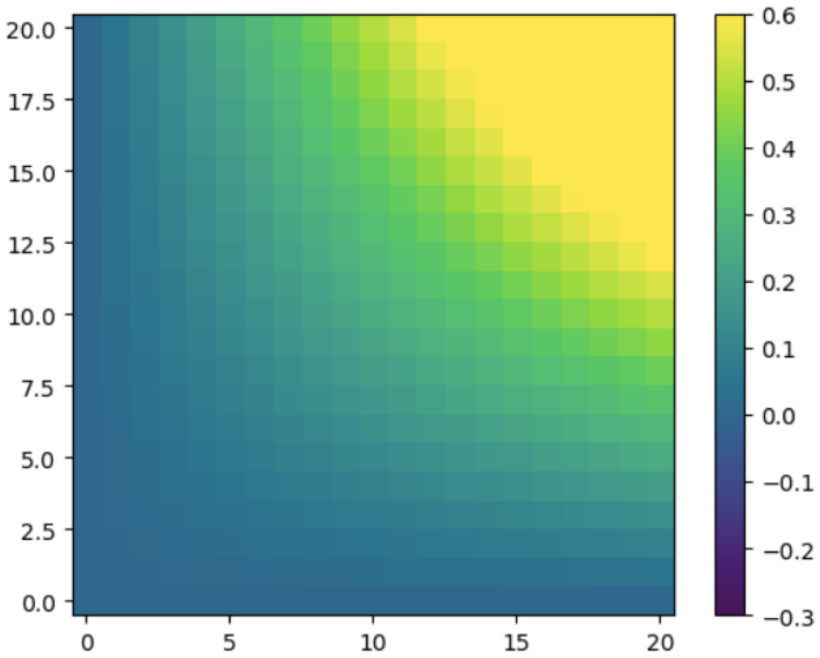
IMSHOW

```
# Change pixel aspect ratio to match.
# axes aspect ratio (default is 1:1).
plt.clf()
plt.imshow(z.T, origin='lower', aspect='auto')
```



IMSHOW

```
# imshow autoscales to color map.

# Change range of data mapped.
plt.clf()
plt.imshow(z.T, origin='lower', vmin=-0.3, vmax=0.6)
```

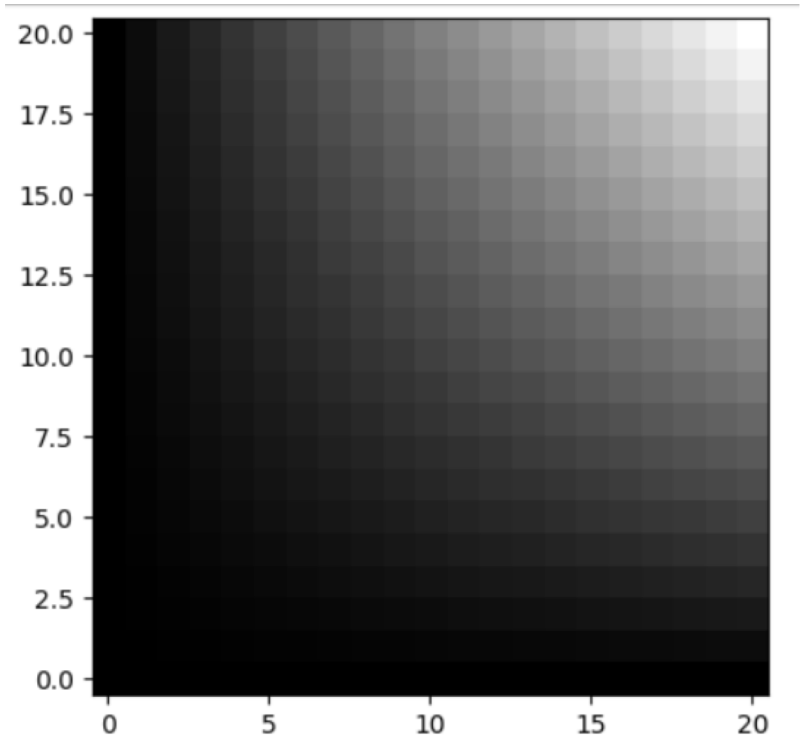


PRO TIP:

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

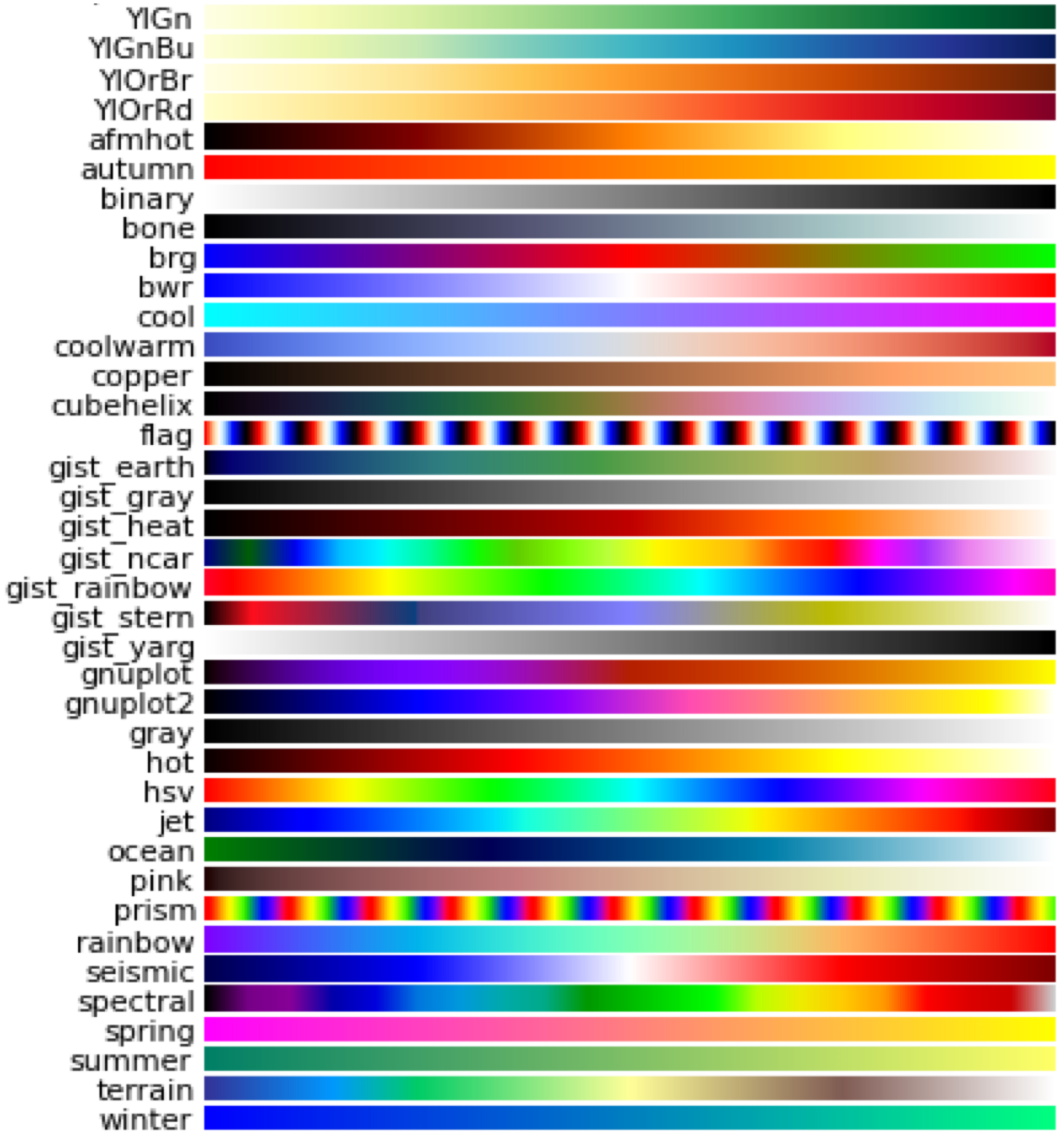
IMSHOW

```
# Change the color map.
plt.clf()
plt.imshow(z.T, origin='lower', cmap=plt.cm.gray)
```



COLORMAPS

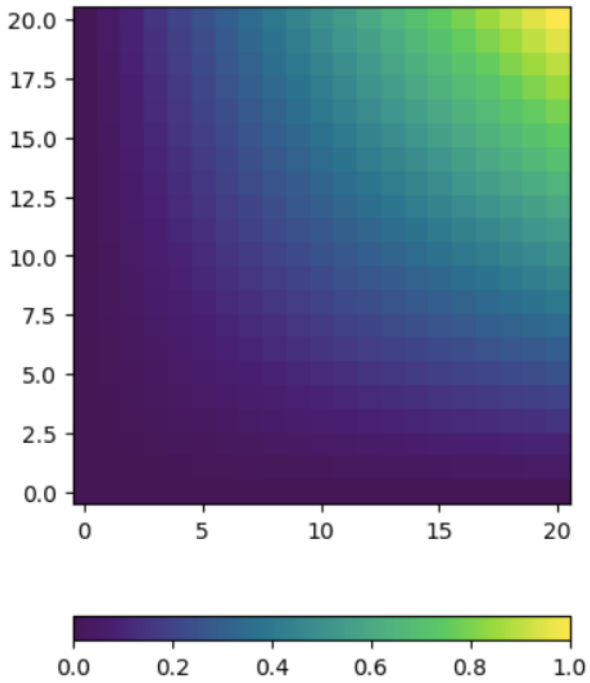
Matplotlib has a large selection of colormaps available. You can also code your own! All of the colormaps are located in the plt.cm module.



Just a selection of built-in colour maps

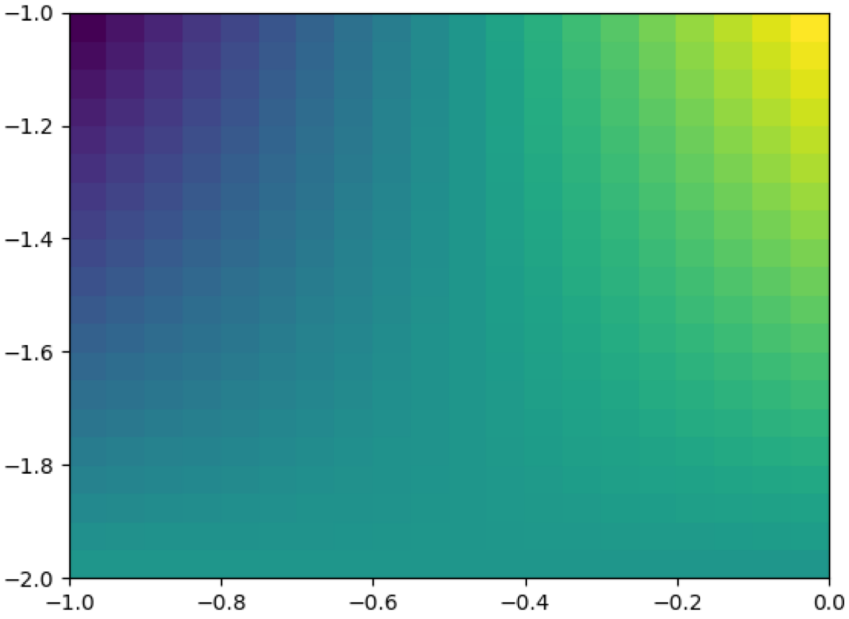
COLORBARS

```
# Change the colorbar orientation.

plt.clf()
plt.figure(figsize=(4,6))
plt.imshow(z.T, origin='lower')
plt.colorbar(orientation='horizontal')
# Other options allow you to specify the size and location of the colorbar.
```



PCOLOR

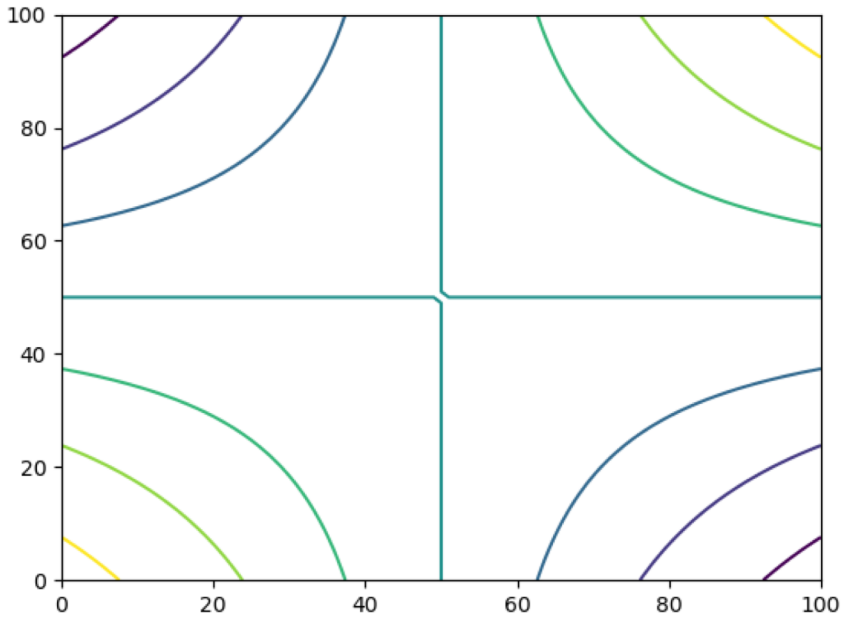
```
# Useful for arbitrary orientations
# and pixel sizes.

# Make new x and y values for pixel
# edges.
xa = np.linspace(-1, 0, 21)
ya = np.linspace(-2, -1, 21)

# Replot with these axes.
plt.clf()
plt.pcolor(xa, ya, z)
```



PRO TIP:
If you have a particularly large array, use “pcolormesh” rather than “pcolor”, which uses more memory.

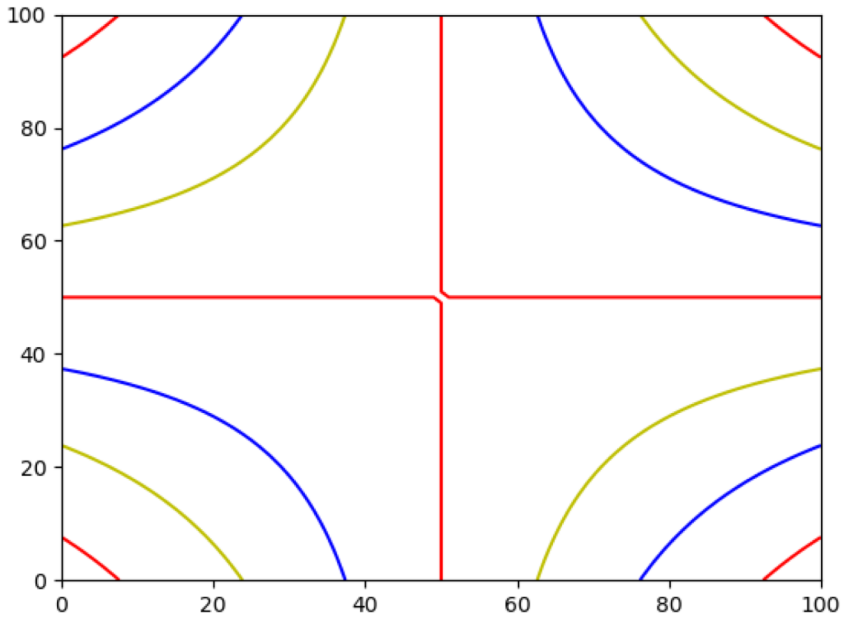
CONTOURS

```
# Same options as imshow.

# Make new data.
x = np.linspace(-1, 1, 101)
y = np.linspace(-1, 1, 101)
z = np.sin(np.outer(x, y))

# Contour plot the new data.
plt.clf()
plt.contour(z)
```



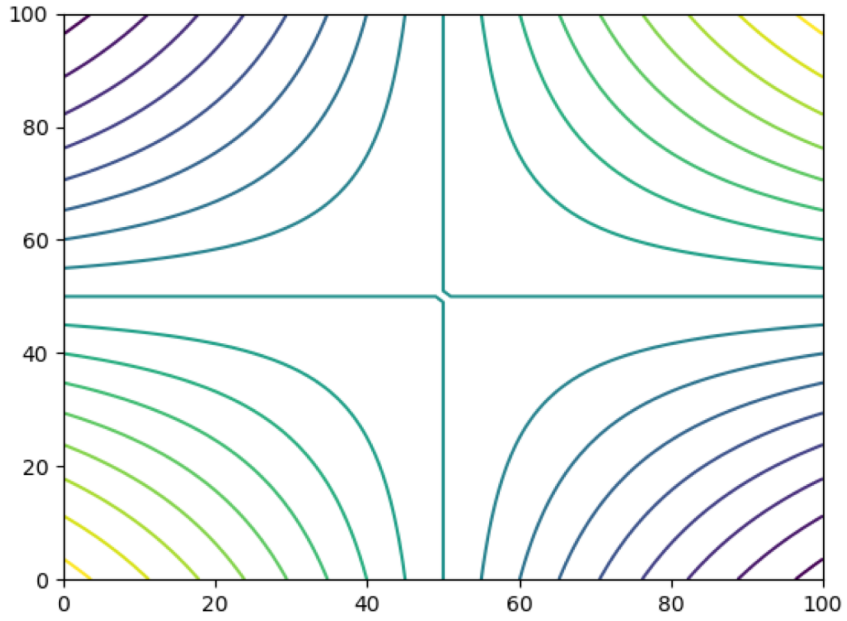
CONTOURS

```
# Specify contour line colors.
plt.clf()
plt.contour(z, colors=('r', 'b', 'y'))
```



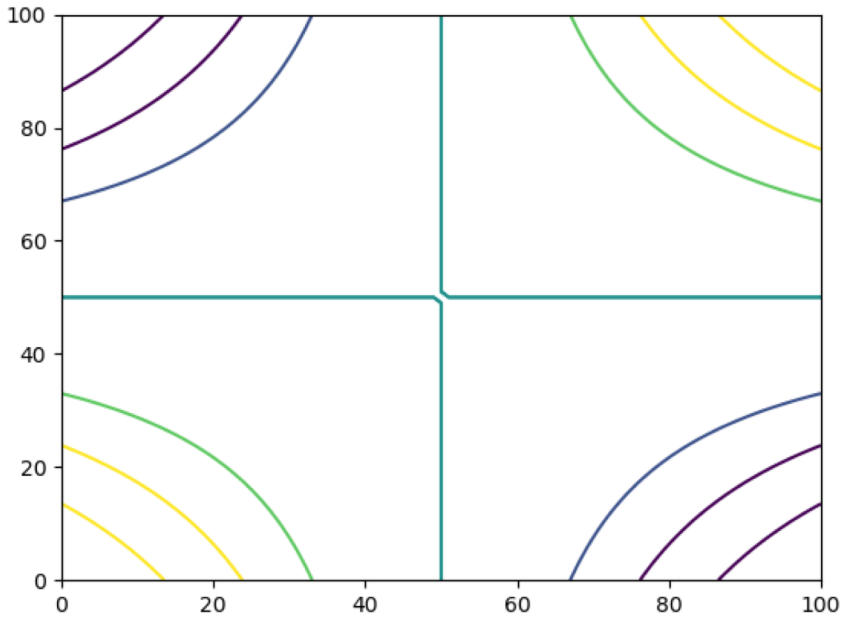
CONTOURS

```
# Specify the number of contour lines.
plt.clf()
plt.contour(z, 20)
```



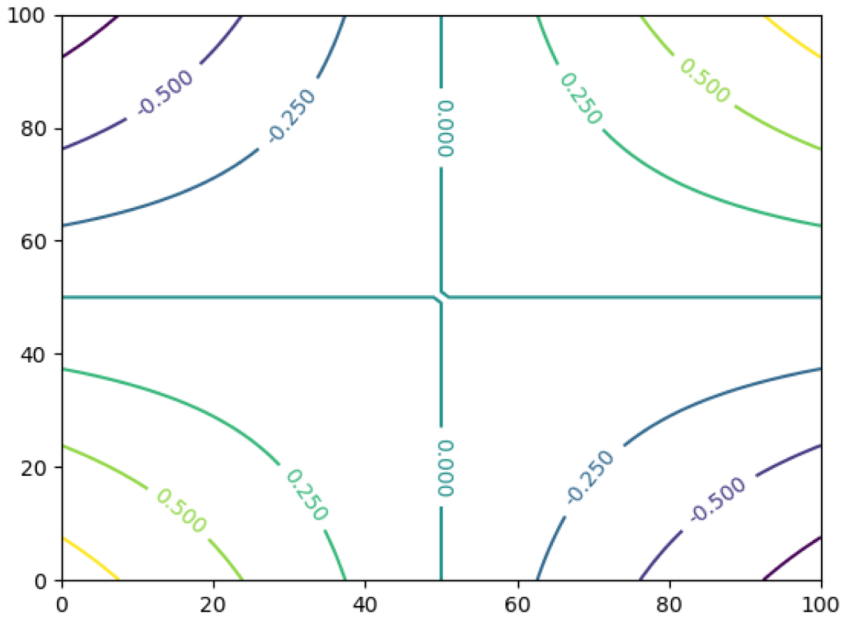
CONTOURS

```
# Specify the contour levels.
plt.clf()
lvls = np.linspace(-1, 1, 7)
plt.contour(z, levels=lvls)
```



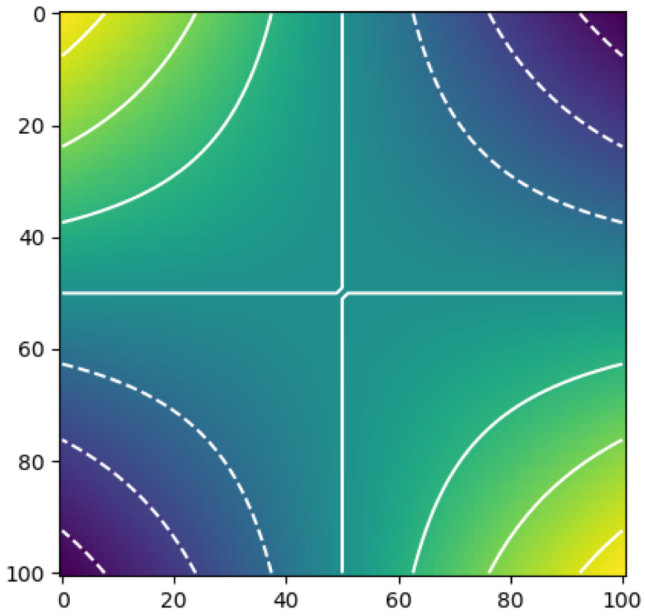
CONTOURS

```
# Add contour labels.
plt.clf()
c1 = plt.contour(z)
plt.clabel(c1)
```



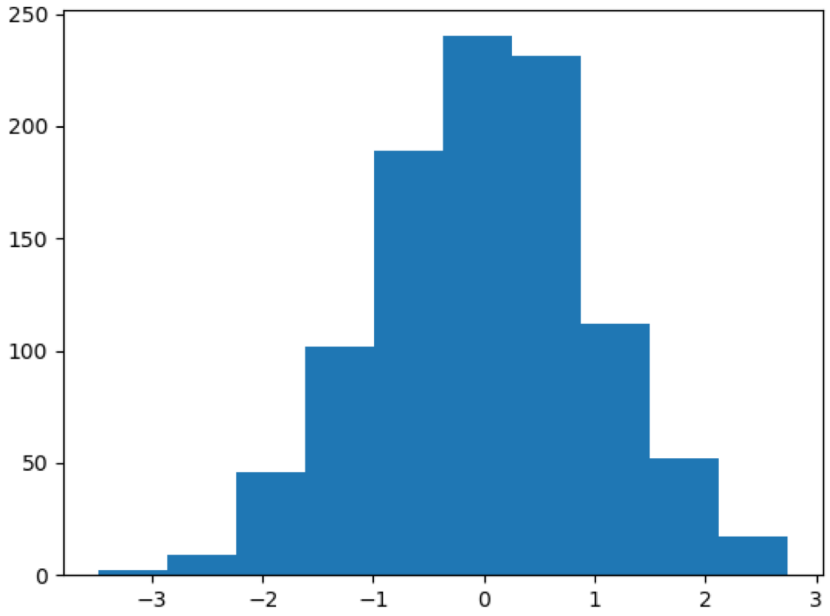
CONTOURS

```
# Overlay contours on images.
plt.clf()
plt.imshow(z)
plt.contour(z, colors='w')
```



HISTOGRAMS

```
# Gaussian random data
y = np.random.normal(size=1000)

# Default histogram (10 bins)
plt.clf()
plt.hist(y)
```

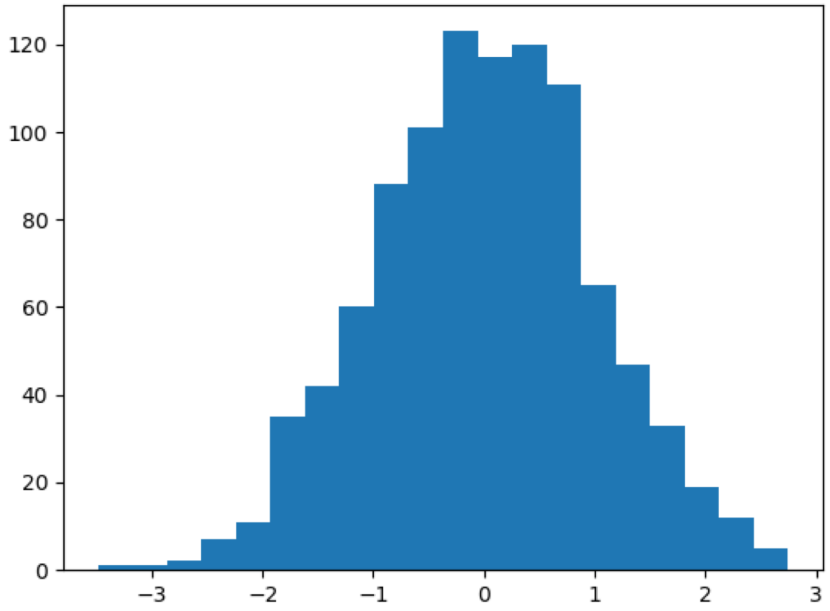


PRO TIP:

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

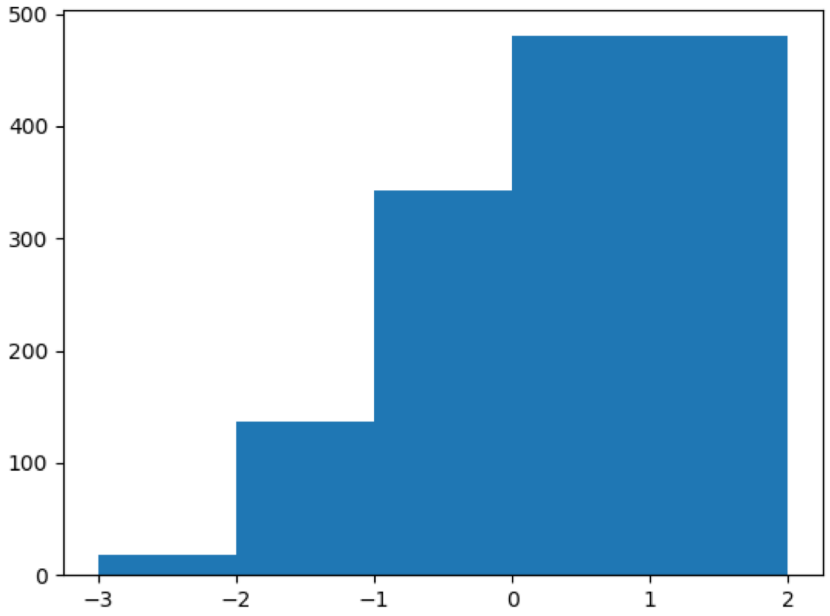
HISTOGRAMS

```
# Change the number of bins.
plt.clf()
plt.hist(y, bins=20)
```



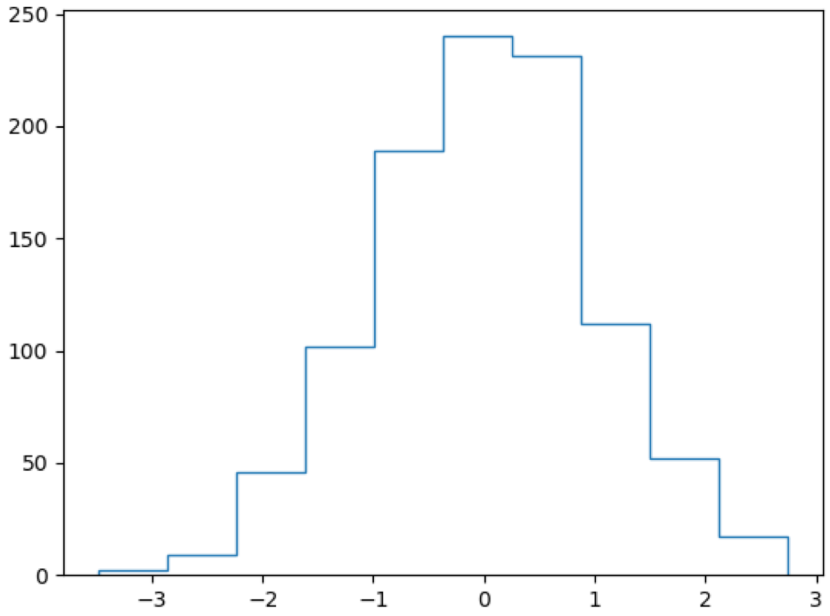
HISTOGRAMS

```
# Change the bin edges.
plt.clf()
edges=[-3, -2, -1, 0, 2]
plt.hist(y, bins=edges)
```



HISTOGRAMS

```
# Just show the lines.
plt.clf()
plt.hist(y, histtype='step')
```



PRO TIP:

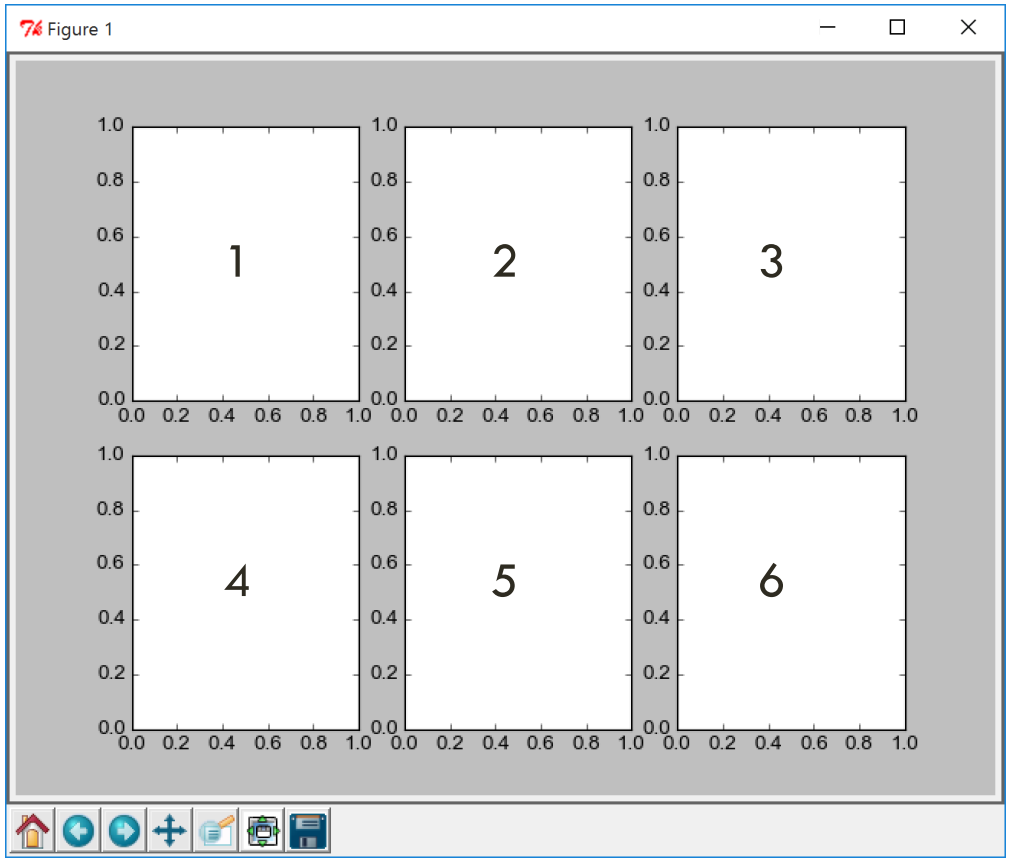
If you just want an array of histogram values, check out the numpy functions histogram, histogram2d, and histogramdd

SUBPLOTS/MULTIPLE PLOTS

Making subplots are quite easy using the convenience function "subplot":

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

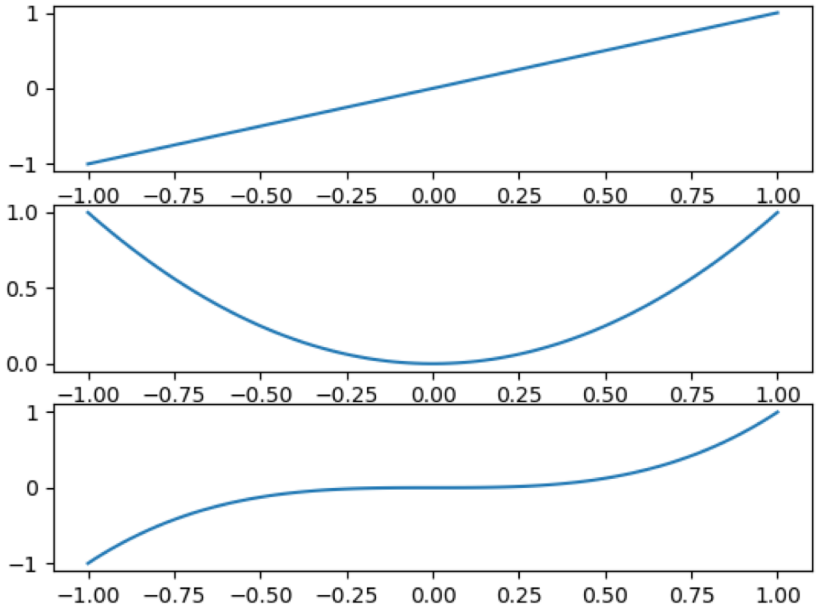
plotnum starts at 1.




```
# Create a 2x3 grid of plots.
ax1 = plt.subplots(2, 3)
```

SUBPLOTS/MULTIPLE PLOTS

```
# 3x1 grid of plots.
plt.clf()
x = np.linspace(-1, 1, 100)
y1 = x
y2 = x**2
y3 = x**3

# Make the plots one at a time.
ax1 = plt.subplot(3, 1, 1)
ax1.plot(x, y1)
ax2 = plt.subplot(3, 1, 2)
ax2.plot(x, y2)
ax3 = plt.subplot(3, 1, 3)
ax3.plot(x, y3)
```



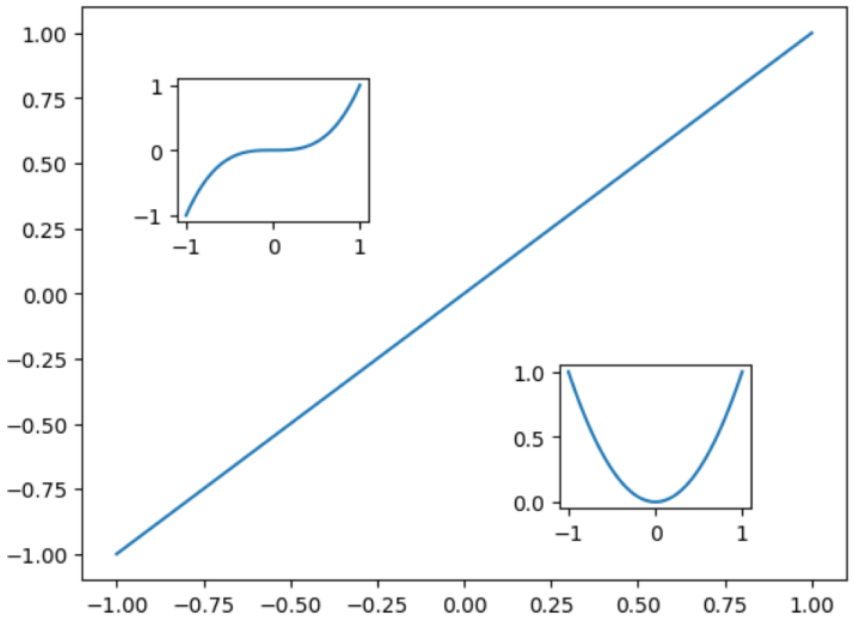
For simple, small numbers of subplots, you can use an alternate argument for the call:

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

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

SUBPLOTS/MULTIPLE PLOTS

```
# Arbitrary subplot placement.
plt.clf()

# Make the plots one at a time.
ax1 = plt.axes([0.1, 0.1, 0.8, 0.8])
ax1.plot(x, y1)
ax2 = plt.axes([0.6, 0.2, 0.2, 0.2])
ax2.plot(x, y2)
ax3 = plt.axes([0.2, 0.6, 0.2, 0.2])
ax3.plot(x, y3)
```



ANNOTATIONS: TEXT

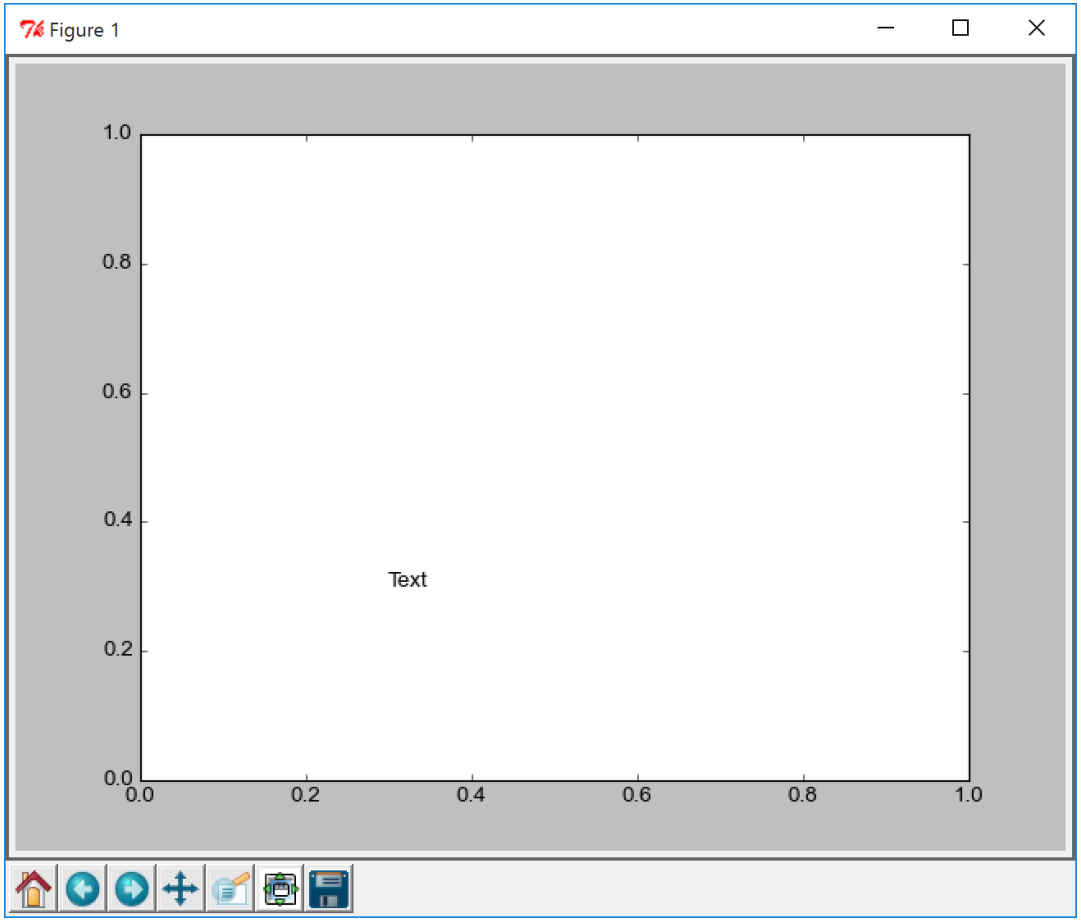
Adding text to axes is simple using the "text" command:

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

Or if adding to the figure:

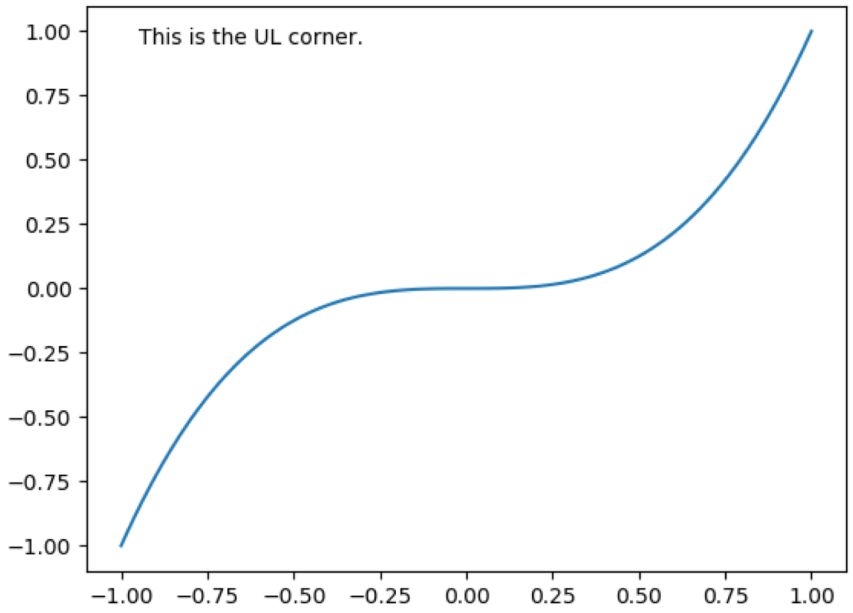
```
plt.figtext(
   x, y, "Text"
)
```

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



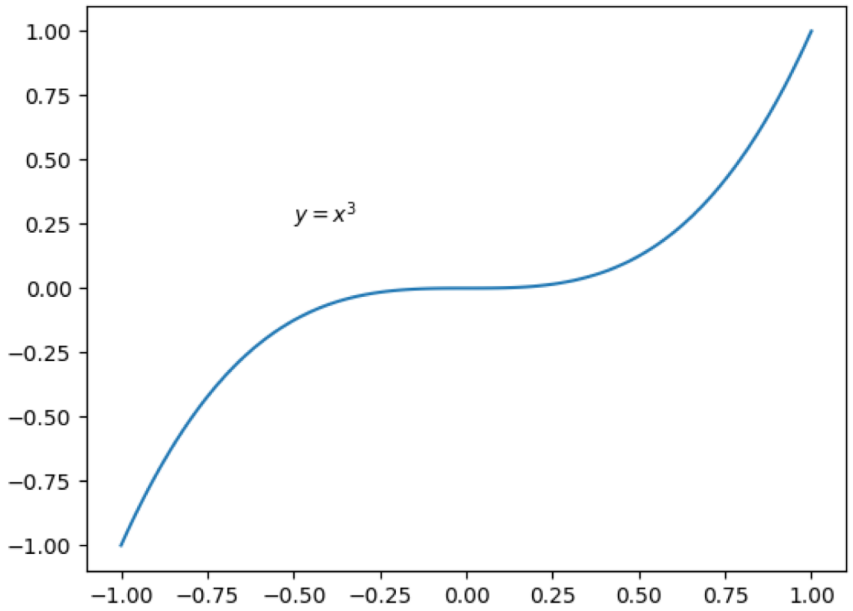
ANNOTATIONS: TEXT

```
# Just specify location and text.
# Location in axis units!
plt.clf()
plt.plot(x, y3)
plt.text(-0.95, 0.95, 'This is the UL corner.')
```



ANNOTATIONS: TEXT

```
# You can use Latex!
plt.clf()
plt.plot(x, y3)
plt.text(-0.5, 0.25, '$y=x^3$')
```



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
```

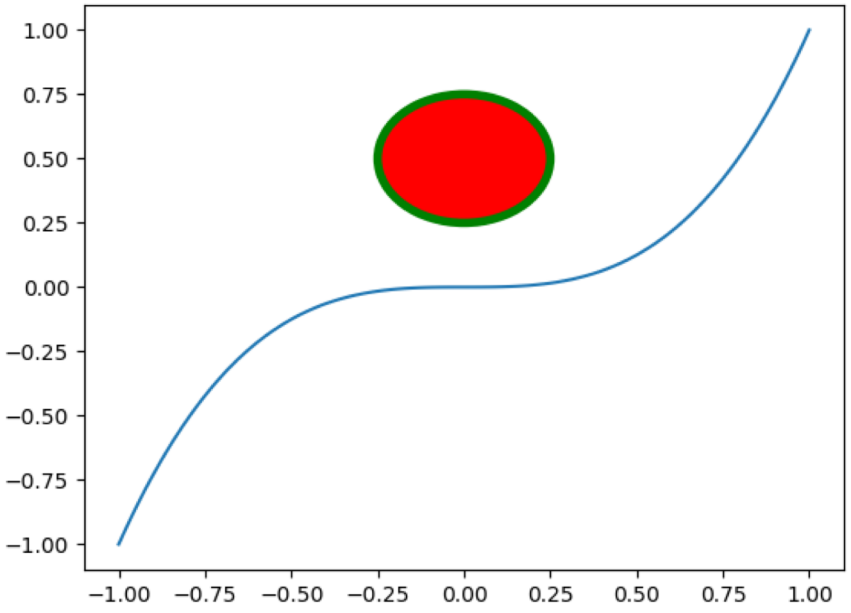
There are a large number of various patches, including Rectangles, Circles, Ellipses, and many more. Once a patch has been made using its declaration (i.e., `p1=patches.Circle(...)`), it needs to be added by:

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

"get current axes"

ANNOTATIONS: PATCHES

```
from matplotlib import patches

# Make a red circle with a thick
# green border.
p1 = patches.Circle((0, 0.5), 0.25,
edgecolor='g', facecolor='r',
linewidth=4)

# Plot the data.
plt.clf()
plt.plot(x, y3)

# Add the circle (seen as oval - why?)
plt.gca().add_patch(p1)
```

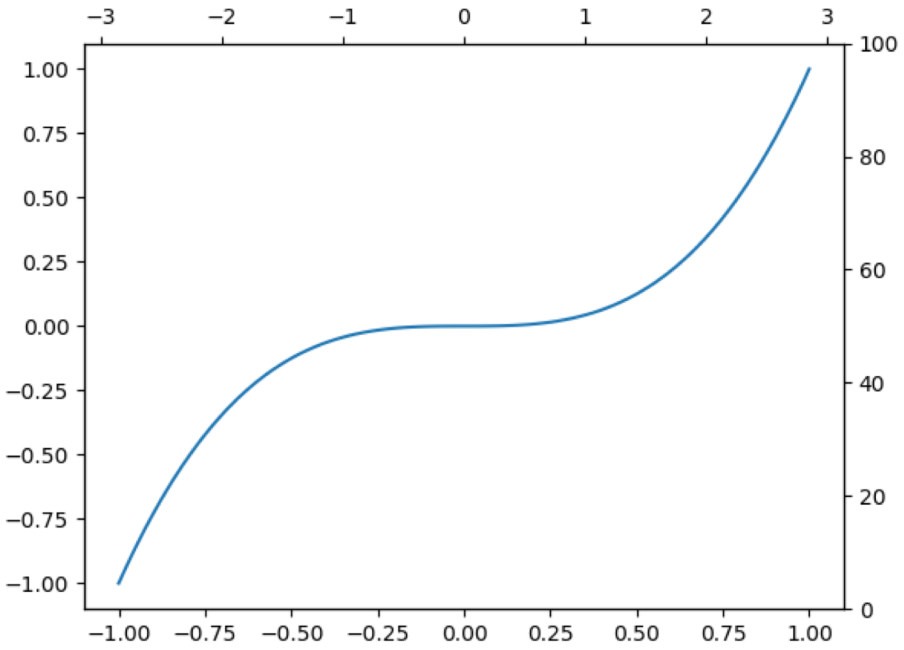


PRO TIP:

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

MULTIPLE AXES ON A SINGLE PLOT

```
# Plot the data.
plt.clf()
plt.plot(x, y3)

# Add extra x and y axes.
# WATCH CAREFULLY!
xaxis2 = plt.gca().twiny()
yaxis2 = plt.gca().twinx()

# Set new values for new axes.
xaxis2.set_xlim(-np.pi, np.pi)
yaxis2.set_ylim(0,100)
```



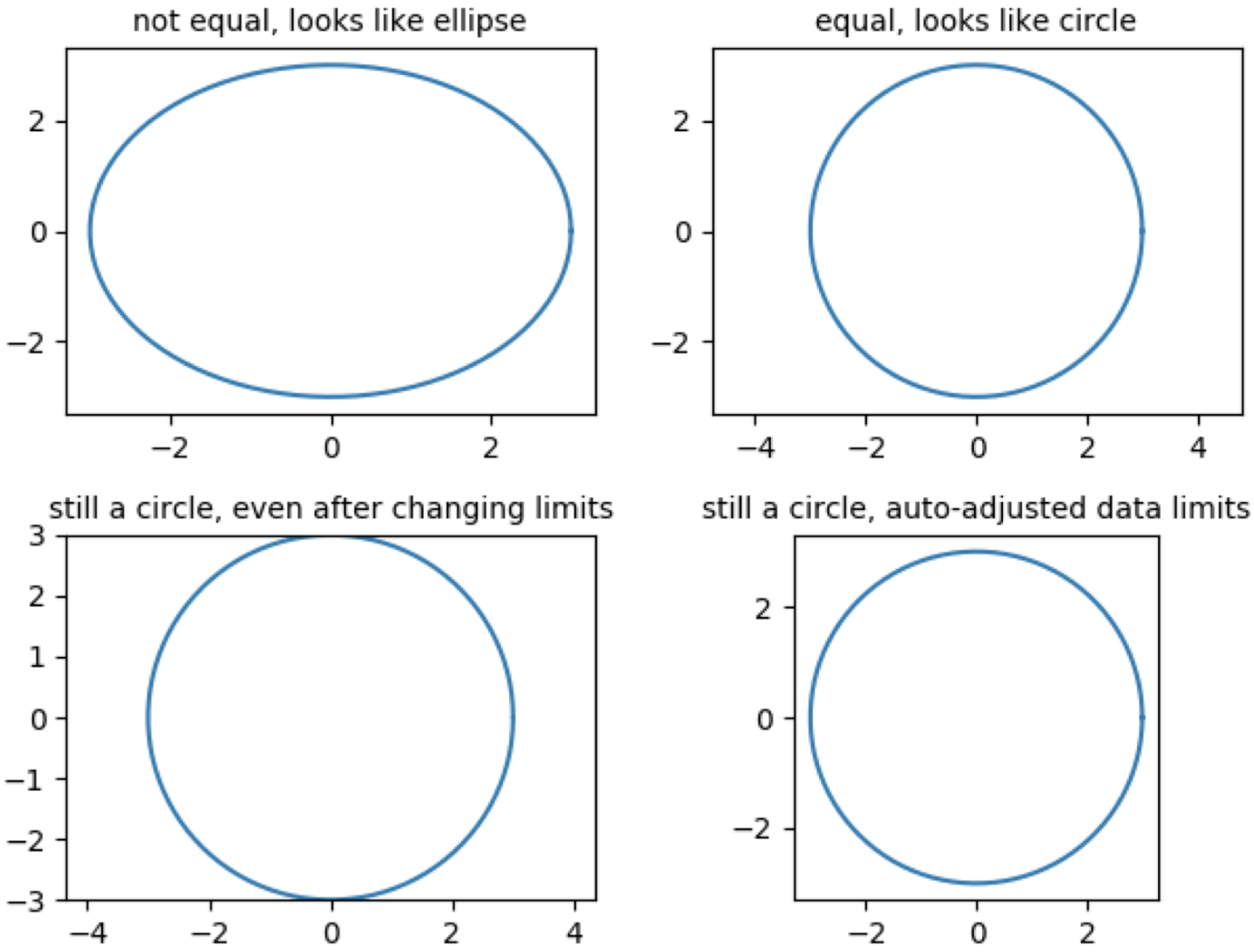
VERY ADVANCED PLOTTING! (1 OF 3)

```
# EQUAL AXES
import matplotlib.pyplot as plt
import numpy as np

# Let's create a circle of radius 3.
an = np.linspace(0, 2 * np.pi, 100)
x = 3 * np.cos(an)
y = 3 * np.sin(an)
```

VERY ADVANCED PLOTTING! (2 OF 3)

```
fig, axs = plt.subplots(2, 2)

axs[0, 0].plot(x, y)
axs[0, 0].set_title('not equal, looks like ellipse', fontsize=10)

axs[0, 1].plot(x, y)
axs[0, 1].axis('equal')
axs[0, 1].set_title('equal, looks like circle', fontsize=10)

axs[1, 0].plot(x, y)
axs[1, 0].axis('equal')
axs[1, 0].axis([-3, 3, -3, 3])
axs[1, 0].set_title('still a circle, even after changing limits', fontsize=10)

axs[1, 1].plot(x, y)
axs[1, 1].set_aspect('equal', 'box')
axs[1, 1].set_title('still a circle, auto-adjusted data limits', fontsize=10)

fig.tight_layout()

plt.show()
```

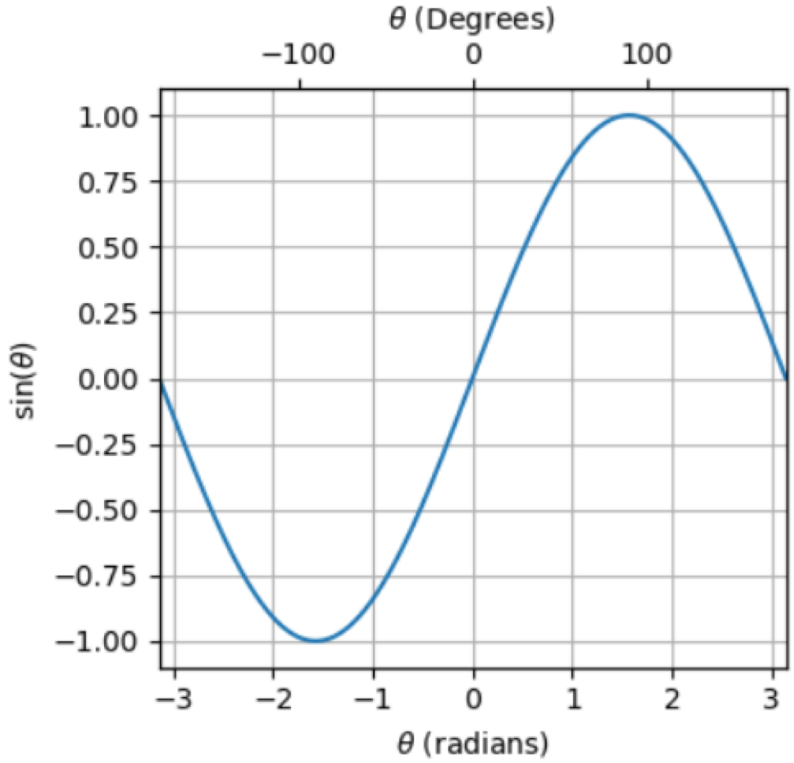
VERY ADVANCED PLOTTING! (3 OF 3)

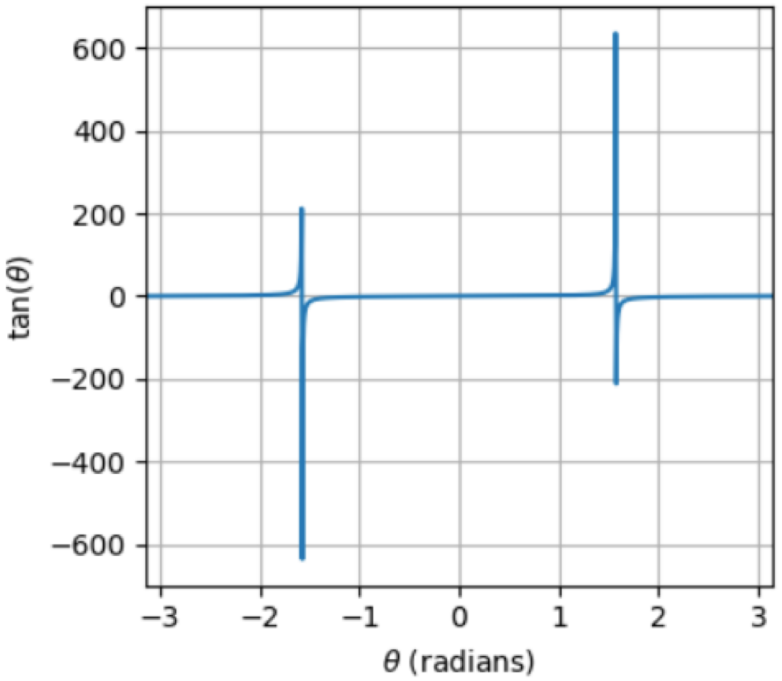


EXERCISE TIME!

GENERATE THIS FIGURE

```
x = np.linspace(-np.pi, np.pi, 1000)
y1 = np.sin(x)
y2 = np.tan(x)
```





SOLUTION

```
x = np.linspace(-np.pi, np.pi, 1000)
y1 = np.sin(x)
y2 = np.tan(x)

fig, (ax1, ax2) = plt.subplots(1, 2)
fig.set_size_inches(8, 4)

ax1.plot(x, y1)
ax1.grid()
ax1.set_xlabel('$\\theta$ (radians)', fontsize=10)
ax1.set_ylabel('$\\sin(\\theta)$', fontsize=10)
ax1.set_xlim(-np.pi, np.pi)
twin_axes = ax1.twiny()
twin_axes.set_xlabel('$\\theta$ (Degrees)', fontsize=10)
twin_axes.set_xlim(-180, 180)

ax2.plot(x, y2)
ax2.grid()
ax2.set_xlim(-np.pi, np.pi)
ax2.set_xlabel('$\\theta$ (radians)', fontsize=10)
ax2.set_ylabel('$\\tan(\\theta)$', fontsize=10)

fig.tight_layout()
```