You’ve got the basics, now let’s unleash the power!
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:

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

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

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

**PRO TIP:**
Saving to EPS doesn’t support transparency.
Images (when stored in an array) are in a different order than in the Cartesian sense. For instance, finding coordinate (3,2):

<table>
<thead>
<tr>
<th>Image Coordinates</th>
<th>Cartesian Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
</tr>
<tr>
<td>2,0</td>
<td>2,1</td>
</tr>
<tr>
<td>3,0</td>
<td>3,1</td>
</tr>
<tr>
<td>4,0</td>
<td>4,1</td>
</tr>
</tbody>
</table>
Images (when stored in an array) are in a different order than in the Cartesian sense. For instance, finding coordinate (3,2):
Images (when stored in an array) are in a different order than in the Cartesian sense. For instance, finding coordinate (3, 2):

<table>
<thead>
<tr>
<th>Image Coordinates</th>
<th>Cartesian Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,4</td>
</tr>
<tr>
<td>0,1</td>
<td>1,4</td>
</tr>
<tr>
<td>0,2</td>
<td>2,4</td>
</tr>
<tr>
<td>0,3</td>
<td>3,4</td>
</tr>
<tr>
<td>1,0</td>
<td>0,3</td>
</tr>
<tr>
<td>1,1</td>
<td>1,3</td>
</tr>
<tr>
<td>1,2</td>
<td>2,3</td>
</tr>
<tr>
<td>1,3</td>
<td>3,3</td>
</tr>
<tr>
<td>2,0</td>
<td>0,2</td>
</tr>
<tr>
<td>2,1</td>
<td>1,2</td>
</tr>
<tr>
<td>2,2</td>
<td>2,2</td>
</tr>
<tr>
<td>2,3</td>
<td>3,2</td>
</tr>
<tr>
<td>3,0</td>
<td>0,1</td>
</tr>
<tr>
<td>3,1</td>
<td>1,1</td>
</tr>
<tr>
<td>3,2</td>
<td>2,1</td>
</tr>
<tr>
<td>3,3</td>
<td>3,1</td>
</tr>
<tr>
<td>4,0</td>
<td>0,0</td>
</tr>
<tr>
<td>4,1</td>
<td>1,0</td>
</tr>
<tr>
<td>4,2</td>
<td>2,0</td>
</tr>
<tr>
<td>4,3</td>
<td>3,0</td>
</tr>
</tbody>
</table>
Images (when stored in an array) are in a different order than in the Cartesian sense:

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

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

```python
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.
**IMSHOW**

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

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

But this likely doesn’t do what you want it to, so there are many optional arguments to use.
Moving to Cartesian coordinates manually:

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

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

```python
plt.imshow(arr1.T, origin='lower')
```
Moving to Cartesian coordinates manually:

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

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

```python
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.
The fuzziness is due to interpolation between pixels. The default is "bilinear". To see the pixels:

```python
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:

```python
plt.imshow(
    ..., extent=[0, 5, 0, 10]
)
```
By default, the image is placed such that the pixels are centred on their pixel number. This can be changed using the “extent” argument:

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

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

**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:

```python
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:

```python
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:

```python
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:

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

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

**PRO TIP:**
If the array contains NaNs, the autoscaling will fail. In which case, you need to manually set `vmin/vmax` values.
We can also change the colourmap used to turn floating point values into colours:

```python
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:

```python
plt.imshow(...,
cmap=plt.cm.gray)
```
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.
COLOURMAPS

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

Choose the one that works best for your purpose.
You can create a simple colour bar using the convenience function `plt.colorbar`:

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

```python
plt.colorbar(
    orientation="horizontal"
)
```
If you don’t want to worry about the orientation issues or have images with varying pixel sizes, you can use the pcolor function instead of imshow:

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

‘xvals’ and ‘yvals’ are arrays with the values of the x and y pixel edges.
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`:

```python
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 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):

```python
plt.contour(...,
    colors=('r','b'))
```
CONTOURS

Setting the number of contours:

```python
plt.contour(arr, 20, ...)
```
Setting the specific location of the contours:

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

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

```python
import matplotlib.pyplot as plt
c1 = plt.contour(...) plt.clabel(c1)
```
Matplotlib also provides robust histogram capabilities:

```python
plt.hist(arr)
```
Matplotlib also provides robust histogram capabilities:

\texttt{plt.hist(arr)}

**PRO TIP:**
The histogram function takes a one-dimensional array. If it isn’t already, flatten it!
Choosing the number of bins:

```python
plt.hist(..., bins=20)
```
HISTOGRAMS

Choosing the number of bins:

```python
plt.hist(..., bins=20)
```

Or specific location of bin edges:

```python
plt.hist(..., bins=bin_edges)
```
Choosing steps instead of bars:

```python
plt.hist(...,
         histtype='stepfilled')
```
HISTOGRAMS

Or maybe you’d prefer just the line?

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

There is also a `hist2d` command that histograms 2D data into an image.
Or maybe you’d prefer just the line?

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

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

`plotnum` starts at 1.
Making subplots are quite easy using the convenience function “subplot”:

```python
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:

```python
plt.subplot(321)
```
where this axis is the first in a grid of 3 rows and 2 columns.
More complicated plots can be made by adding specific axes:

```python
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.
Adding text to axes is simple using the “text” command:

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

Or if adding to the figure:

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

Where these coordinate go from 0 to 1 in fractions of the figure.
Anywhere you have text, you can use LaTeX by enclosing the text in dollar signs ($)

```python
plt.text(...
    "x+y=\sqrt{z}"
)
```

$x + y = \sqrt{z}$
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 `\textdefault{...}` environment.
Adding additional shapes to the plot is called adding a “patch”. There are a variety of patches available by importing:

```python
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:

```python
ax1.add_patch(p1)
# Or if you haven’t created a variable for your axis
plt.gca().add_patch(p1)
```
Looking at a ‘Circle’ patch:

```python
p1 = patches.Circle((xloc, yloc),
    radius=3,
    edgecolor='g',
    facecolor='r',
    linewidth=4
)
```
Looking at a ‘Circle’ patch:

```python
p1 = patches.Circle((xloc, yloc), radius=3,
                     edgecolor='g',
                     facecolor='None',
                     linewidth=4)
```
Looking at a ‘Circle’ patch:

```python
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:

```python
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.