Arrays-Order to the Universe

The types of data structures we have been using are (

Lists-We have worked with these and they are nice because they are mutable and easy to work with. But they are not great with numbers.

Tuples-We are not spending time on them. They are like lists with a funny name but they are immutable. So they can be good if you need something not to change. But we don't use them because they don't do much with numbers.

Dictionaries-We use these a little. They are thought of key:value pairs and are created with {}. We have used these already when defining our "props" on the graphs. It lets you pass a few keyword values at once. We won't use these much but you will come across them.

Numpy arrays-we have started using these. We have seen they are easy to plot and to do math with. But they are not great with large datasets with lots of different columns and with missing data. But they are the basis for a lot of things in python so you always build off of numpy.

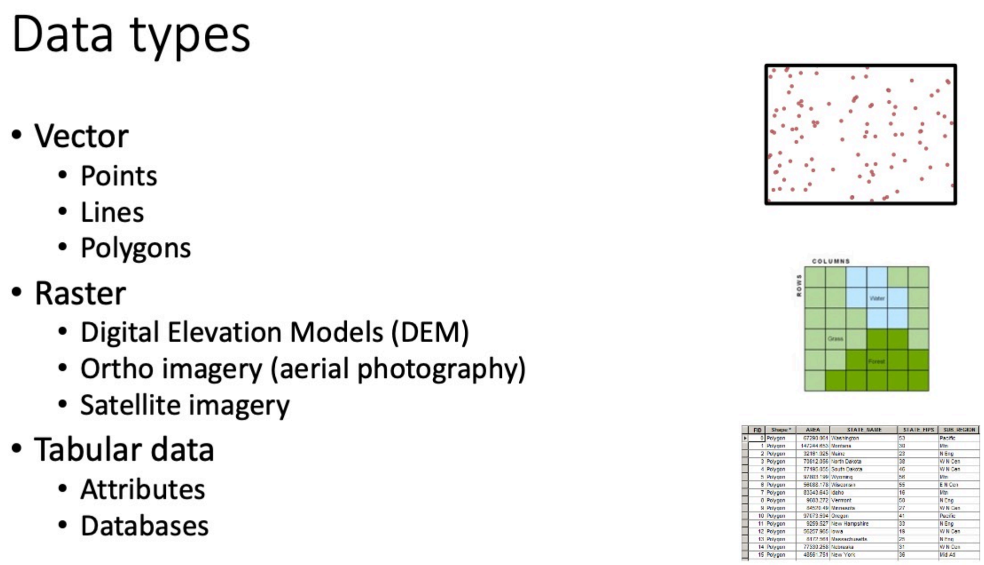
Pandas Dataframes-We have started these. these are like supercharged numpy arrays that give you a lot more information. If you could imagine that you could name the rows and columns in a numpy array it starts to get you there. Sort of like an excel sheet in the computer memory but more powerful. Plus they are good with dates and re-ordering. So these are good for complex datasets where we want to name variables. We will mainly be using these and building everything off of them.

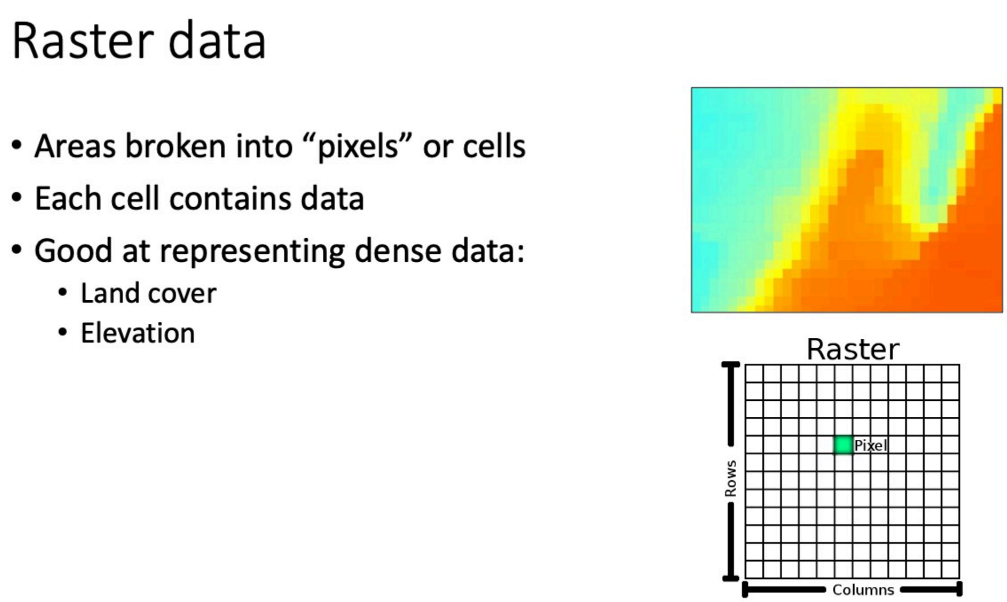
But how do we think about data?

When we usually think about data we think about tabular data. This is an excel sheet. Just a table of the data we have. this is the dataframe we read in. But there are really three data types we might run into

- Tabular Data
- Vector Data
- Raster Data

The picture below explain them





Vector data are used with GIS. They are for putting lines, points, or polygons on a map. For example putting a shoreline on a map or a road or a lake. We are not going to use them much this semester.

But first today we are going to talk about Raster data and two dimensional arrays. Raster data is really a 2d array.

Today lets just work with two dimensional numpy arrays. You can have arrays of as many dimensions as you want but I have trouble comprehending at three and more dimensions.

In [2]:
```
%matplotlib inline
import matplotlib.pylab as plt
import numpy as np
```

```
from scipy import stats
import pandas as pd
```

In [2]:
```
oneD=np.array([1,2,3,4,5,6,7,8,9,10])  # does anyone remember that band?
```

In [3]:
```
oneD
```

Out[3]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [4]:
```
#review-what will this print?
#oneD[0:10:2]
```

In [5]:
```
twoD=np.array([[1,2,3,4],[5,6,7,8]])
```

In [6]:
```
twoD
```

Out[6]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Do you see what I just did? It is 2 one dimensional arrays together to make a 2-dimensional array or table!

In [7]:
```
type(twoD)
```

Out[7]:
```
numpy.ndarray
```

In [8]:
```
len(twoD)
```

Out[8]:
```
2
```

In [9]:
```
twoD.shape
```

Out[9]:
```
(2, 4)
```

In [10]:
```
np.shape(twoD)
```

Out[10]:
```
(2, 4)
```

In [11]:
```
twoD.size
```

Out[11]:
```
8
```

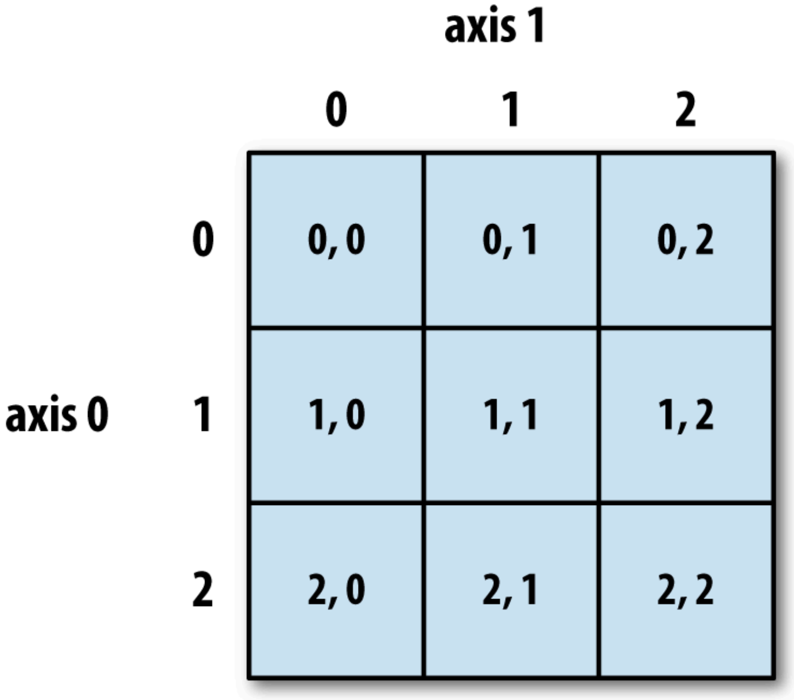
You can see what you can do with the np array by typing twoD. and then tab and you see the functions available. try one

In [108…
```
twoD.
```

Now lets try slicing. Remember it is rows and then columns. See if you can guess before uncommenting and running. This picture is just to help you and you don't need to load it.

In [12]:
```
from IPython.display import Image
Image(filename='array-axes.png',width=400)
```

Out[12]:



In [12]:
```
twoD[0,0]
```
Out[12]:
```
1
```
In [13]:
```
twoD[1,0]
```
Out[13]:
```
5
```
In [14]:
```
twoD[:,:]
```
Out[14]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```
In [16]:
```
#twoD[:,0]
```
In [111…
```
#twoD[:,1]
```
In [112…
```
#twoD[0,:]
```
In [113…
```
#twoD[1,:]
```
In [114…
```
#twoD[0,0]
```
In [115…
```
#twoD[3,3]
```
In [116…
```
#twoD[1,3]
```
np.vstack adds a row. So lets make our array bigger and keep going!

In [15]:
```
twoD=np.vstack((twoD,[9,10,11,12]))
```
In [16]:
```
twoD
```

Out[16]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [17]:
```
twoD=np.vstack((twoD,[13,14,15,16]))
```

In [18]:
```
twoD
```

Out[18]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

Now lets do some more slicing!

remember. For numpy it is

[start:stop:skip]

if you list multiple items and leaving one out assumes the last one is missing

so that means [1::] is one to the end by 1.

If you have a 2d array it will be [start:stop:skip,start:stop:skip] for the rows and then the columns

In [19]:
```
twoD[:,:]
```

Out[19]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

In [124…
```
#twoD[::2,::2]
```

In [125…
```
#twoD[2:,2:]
```

In [126…
```
#twoD[1:3,1:3]
```

In [23]:
```
#twoD[1::2,:]
```

Now you can set the numbers in different places.

In [20]:
```
twoD[3,3]=100
```

In [21]:
```
twoD
```

Out[21]:
```
array([[  1,   2,   3,   4],
       [  5,   6,   7,   8],
       [  9,  10,  11,  12],
       [ 13,  14,  15, 100]])
```

In [22]:
```
twoD[1:3,1:3]=55
```

In [23]:
```
twoD
```

Out[23]:
```
array([[  1,   2,   3,   4],
       [  5,  55,  55,   8],
       [  9,  55,  55,  12],
       [ 13,  14,  15, 100]])
```

you can use a function to set numbers!

In [24]:
```
twoD[0,:]=np.arange(10,14)
```

In [25]:
```
twoD
```

Out[25]:
```
array([[ 10,  11,  12,  13],
       [  5,  55,  55,   8],
       [  9,  55,  55,  12],
       [ 13,  14,  15, 100]])
```

you can reshape the array if you want to.

In [31]:
```
print (twoD.reshape(16,1))
```

```
[[ 10]
 [ 11]
 [ 12]
 [ 13]
 [  5]
 [ 55]
 [ 55]
 [  8]
 [  9]
 [ 55]
 [ 55]
 [ 12]
 [ 13]
 [ 14]
 [ 15]
 [100]]
```

In [33]:
```
print (np.reshape(twoD,(1,16)))
```

```
[[ 10  11  12  13   5  55  55   8   9  55  55  12  13  14  15 100]]
```

In [34]:
```
print (twoD.reshape(8,2))
```

```
[[ 10  11]
 [ 12  13]
 [  5  55]
 [ 55   8]
 [  9  55]
 [ 55  12]
 [ 13  14]
 [ 15 100]]
```

In [35]:
```
print (twoD)
```

```
[[ 10  11  12  13]
 [  5  55  55   8]
 [  9  55  55  12]
 [ 13  14  15 100]]
```

The shape is back to how we had it because we never changed becuase we only printed it. we never set it.

Now this is where we intersect with Raster Data

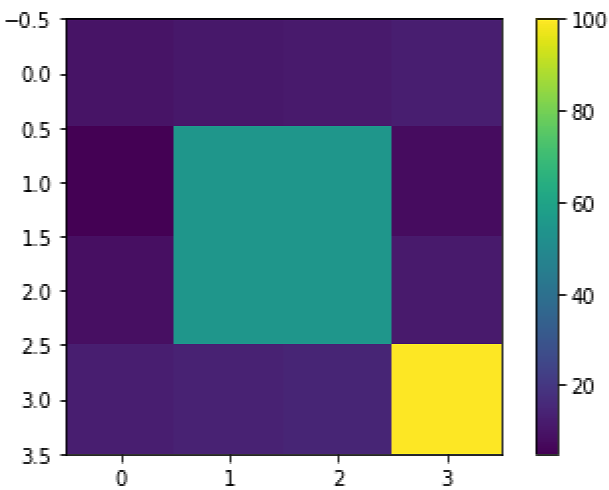
Add Color!

You can visualize the whole array! This just colors the array/grid we have by its values

This is raster data. It is like satelite data.

In [26]:
```
fig,ax=plt.subplots()
cax=ax.imshow(twoD)
fig.colorbar(cax)
```

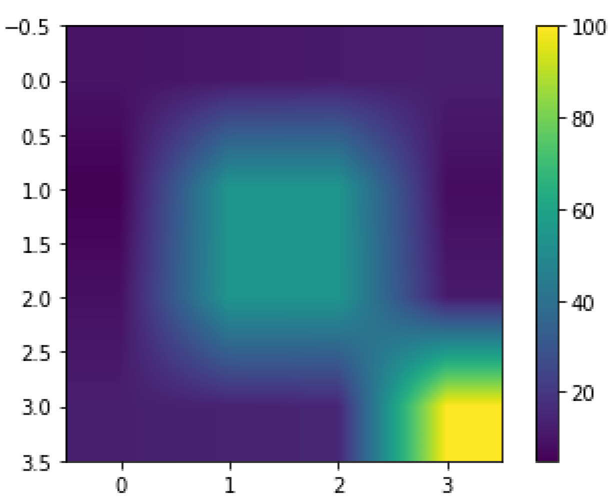
Out[26]:
```
<matplotlib.colorbar.Colorbar at 0x1cc020df390>
```



If you want to make the edges of each box smooth we need to interpolate the data. I chose interpolation='bilinear' but there are many options https://matplotlib.org/gallery/images_contours_and_fields/interpolation_methods.html

In [27]:
```
fig,ax=plt.subplots()
cax=ax.imshow(twoD,interpolation='bilinear')
fig.colorbar(cax)
```

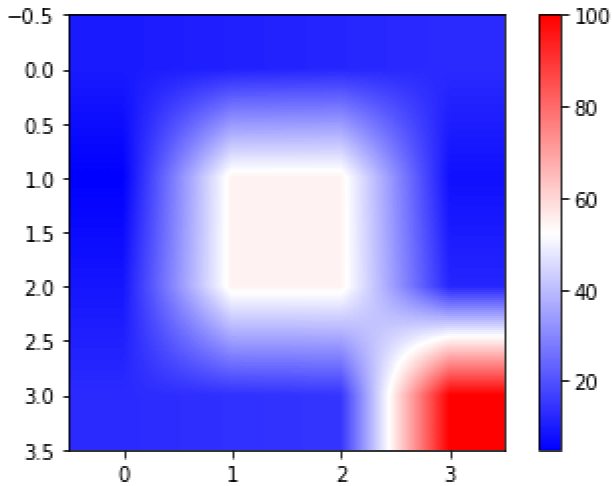
Out[27]:
```
<matplotlib.colorbar.Colorbar at 0x1cc021da898>
```



We can change the colorbar. This is another keyword argument

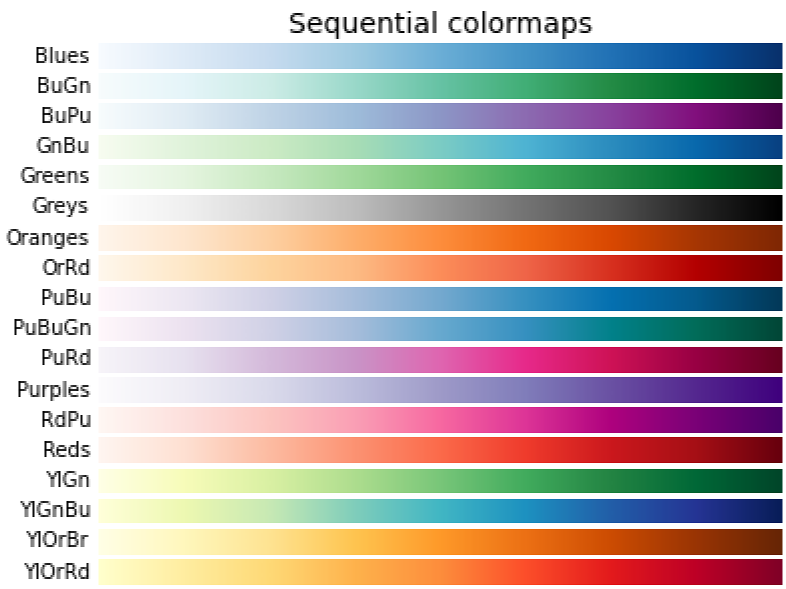
In [28]:
```
fig,ax=plt.subplots()
cax=ax.imshow(twoD,interpolation='bilinear',cmap='bwr')
fig.colorbar(cax)
```

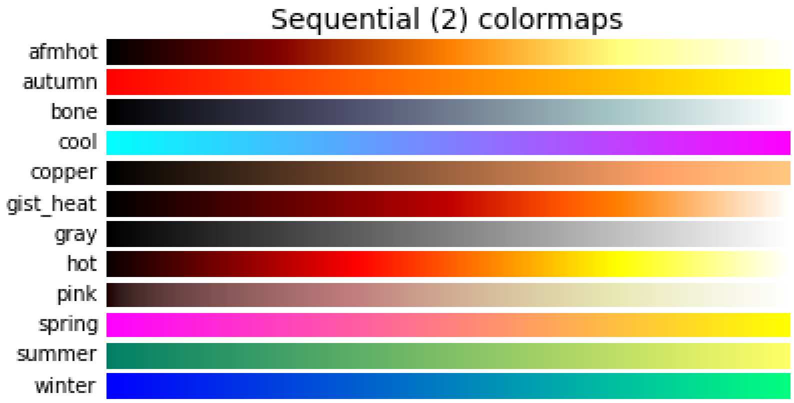
Out[28]: <matplotlib.colorbar.Colorbar at 0x1cc03259588>

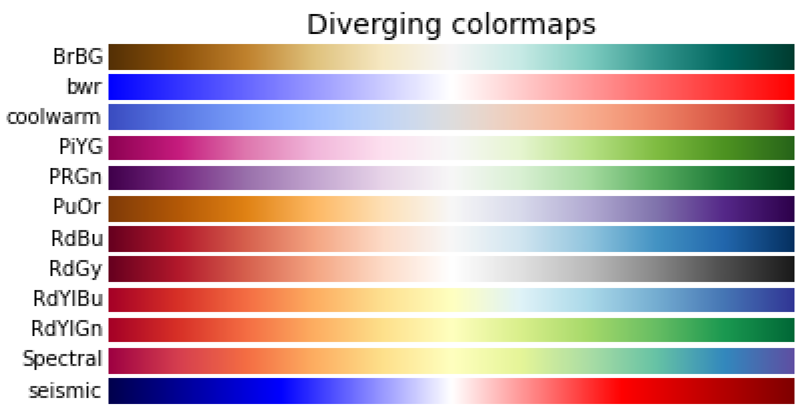


Now make your own. Here is just one list. You can google colormaps python.

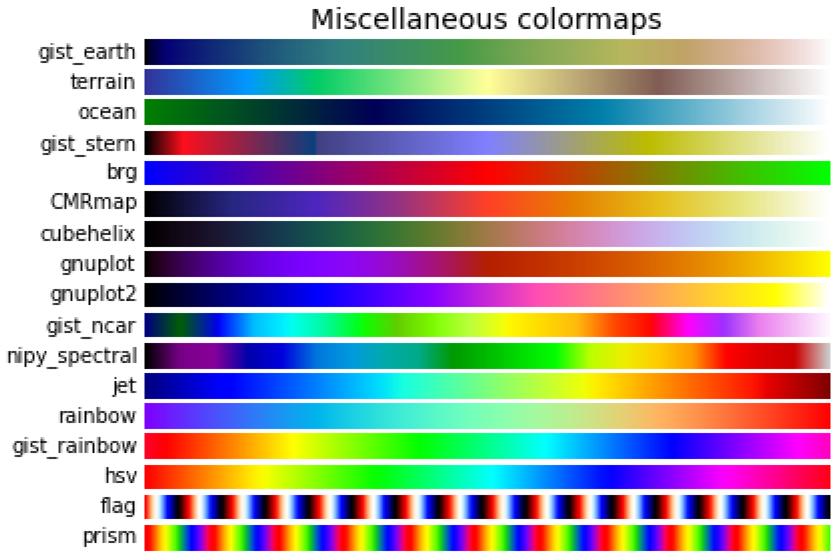
In [140…
```
#See below for the colormap code.  I also just ran the code off the web to make
```




Sequential (2) colormaps
afmhot
autumn
bone
cool
copper
gist_heat
gray
hot
pink
spring
summer
winter


Diverging colormaps
BrBG
bwr
coolwarm
PiYG
PRGn
PuOr
RdBu
RdGy
RdYlBu
RdYlGn
Spectral
seismic

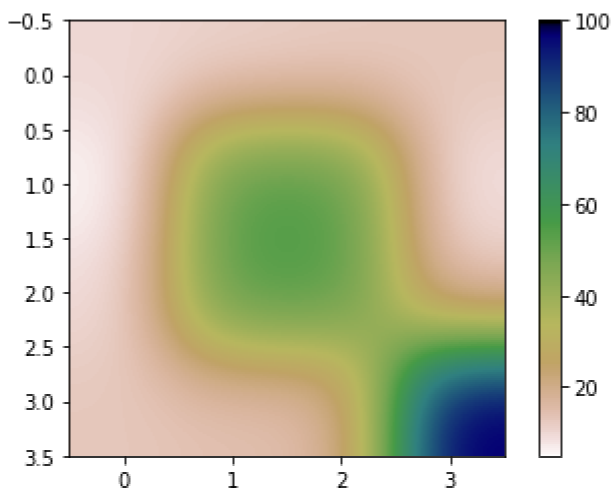

Qualitative colormaps
Accent
Dark2
Paired
Pastel1
Pastel2
Set1
Set2
Set3



- Plot the same plot with a color bar of your choice.
- reverse the color bar by adding _r to the name.

In [29]:
```
fig,ax=plt.subplots()
cax=ax.imshow(twoD,cmap='gist_earth_r',interpolation='gaussian')
fig.colorbar(cax)
```

Out[29]: <matplotlib.colorbar.Colorbar at 0x1cc032fbf98>



Now we are going to intersect back with pandas and dataframes.

We are going to

- read in a csv using pandas.
- this gives us a big dataframe which we can convert to a 2 dimensional array that is 100x100 is in size
- then we can plot/map it.

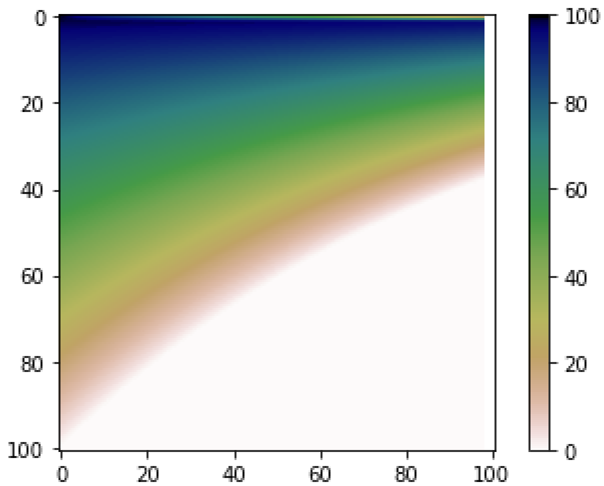
https://github.com/bmaillou/BigDataPython/blob/master/my_first_csv.csv

In [27]:
```
df=pd.read_csv('my_first_csv.csv',header=None)
```

In [30]:
```
twoD=df.values  # this takes the dataframe and turns it into an array
```

In [31]:
```
fig,ax=plt.subplots()
cax=ax.imshow(twoD,cmap='gist_earth_r',interpolation='gaussian')
fig.colorbar(cax)
```

Out[31]: <matplotlib.colorbar.Colorbar at 0x7fad32c9e580>

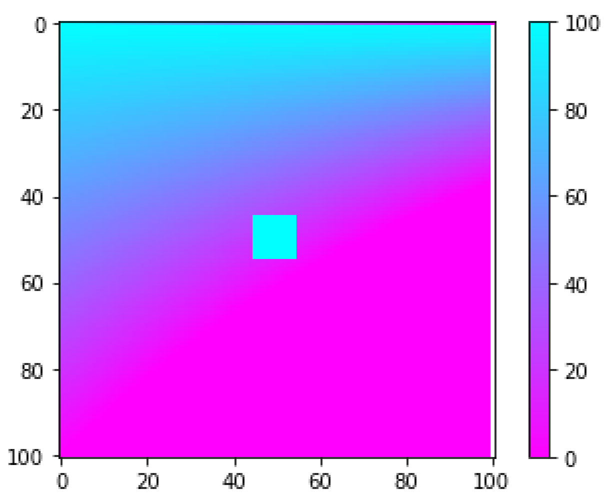


Remember we can alter the array. I am going to add a square in the middle of value 100...

In [36]:
```
twoD[45:55,45:55]=100
```

In [38]:
```
fig,ax=plt.subplots()
cax=ax.imshow(twoD,cmap='cool_r',interpolation='none')
fig.colorbar(cax)
```

Out[38]: <matplotlib.colorbar.Colorbar at 0x7fad33358940>



In [ ]:

Now you need to Read in Brian.csv.

It is a 2d array. Plot it with imshow and tell me what it looks like. you will be shocked at what it shows......

https://github.com/bmaillou/BigDataPython/blob/master/Brian.csv

In [ ]:

```

```

Homework hint: Think about how the data from Brian.csv is stored and how you can change it....

Now back to Tabular data.

For most of our data we usually work with tabular data. One example is if the first column is X and the remaining columns are all differnt Y's. Here is an example. Read in the oneX_manyY.csv file. print it to see it. Then plot all the values. This is really how we think of excel. But we give the columns nicer names

X Value	First Y Value	Second Y Value	Third Y Value	Fourth Y Value
1.0	1.0	10.0	4.0	6.0
2.0	2.0	9.0	4.0	6.0
3.0	3.0	8.0	4.0	6.0
4.0	4.0	7.0	4.0	6.0

https://github.com/bmaillou/BigDataPython/blob/master/oneX_manyY.csv

In [136…

```
manyY=pd.read_csv('oneX_manyY.csv')
manyY
```

Out[136…

	x_values	first_y_values	second_y_values	third_y_values	fourth_y_values
0	1	1	10	4	6
1	2	2	9	4	6
2	3	3	8	4	6
3	4	4	7	4	6
4	5	5	6	4	6
5	6	6	5	4	6
6	7	7	4	4	6
7	8	8	3	4	6
8	9	9	2	4	6
9	10	10	1	4	6

To show you how data sets/types are related we could strip off the column titles so it becomes a 2d array for numpy

In [137…

```
manyY=manyY.values
```

In [138…
```
manyY
```

Out[138…
```
array([[ 1,  1, 10,  4,  6],
       [ 2,  2,  9,  4,  6],
       [ 3,  3,  8,  4,  6],
       [ 4,  4,  7,  4,  6],
       [ 5,  5,  6,  4,  6],
       [ 6,  6,  5,  4,  6],
       [ 7,  7,  4,  4,  6],
       [ 8,  8,  3,  4,  6],
       [ 9,  9,  2,  4,  6],
       [10, 10,  1,  4,  6]])
```

Now you can do all of your array nomenclature. For example lets look at the first column

In [139…
```
manyY[:,0]
```

Out[139…
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

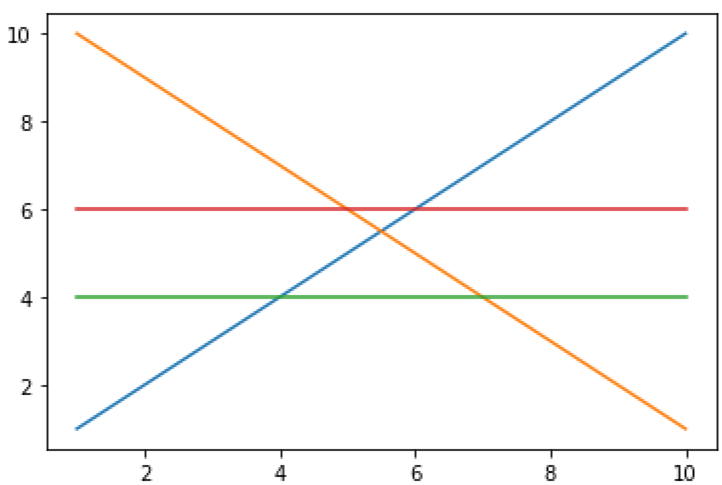
now we can plot the data. For plotting you just keep on listing the x and y pairs.

So plot

- column 0 versus column 1, X Value versus First Y Value
- column 0 versus column 2, X Value versus Second Y Value
- column 0 versus column 3, X Value versus Third Y Value
- column 0 versus column 4, X Value versus Fourth Y Value

In [58]:
```
fig,ax=plt.subplots()
ax.plot(manyY[:,0],manyY[:,1])
ax.plot(manyY[:,0],manyY[:,2])
ax.plot(manyY[:,0],manyY[:,3])
ax.plot(manyY[:,0],manyY[:,4])
```

Out[58]:
```
[<matplotlib.lines.Line2D at 0x7fad3398eb20>]
```

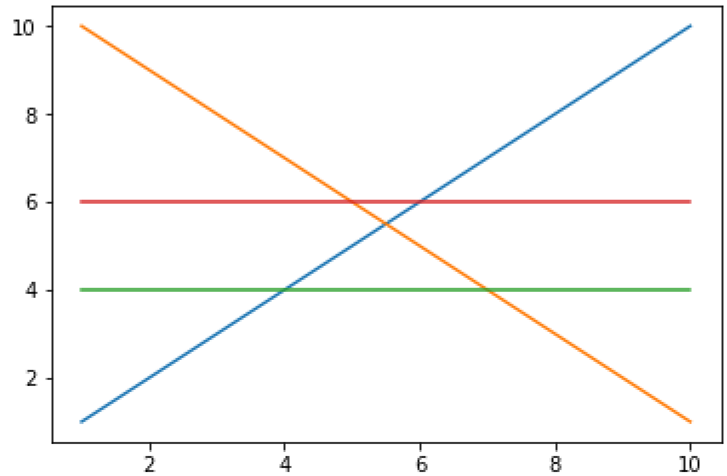


or you can do it in one call to ax.plot by listing x and y pairs but I find this hard to follow

In [47]:
```
fig,ax=plt.subplots()
ax.plot(manyY[:,0],manyY[:,1],manyY[:,0],manyY[0:,2],manyY[:,0],manyY[:,3],manyY
```

Out[47]:
```
[<matplotlib.lines.Line2D at 0x192edc8c6d8>,
 <matplotlib.lines.Line2D at 0x192edcaeac8>,
```

```
 <matplotlib.lines.Line2D at 0x192edcb92b0>,
 <matplotlib.lines.Line2D at 0x192edcb9710>]
```

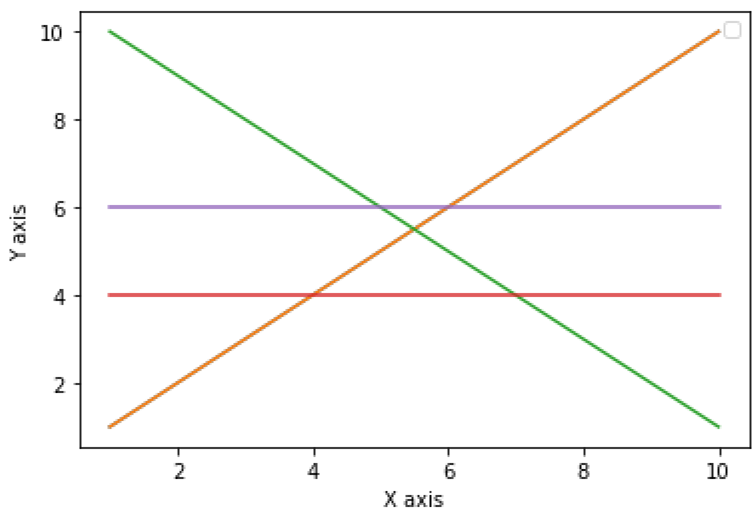


If you wanted to get fancy we could program a for loop to loop over the columns and plot them

In [63]:
```
fig,ax=plt.subplots()
for i in np.arange(5):
    ax.plot(manyY[:,0],manyY[:,i])
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
```

```
No handles with labels found to put in legend.
```

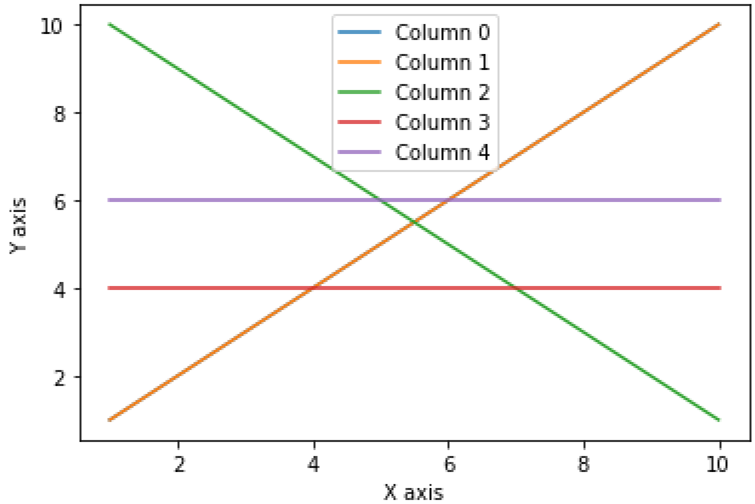
Out[63]:
```
Text(0, 0.5, 'Y axis')
```



And you can add a legend

In [64]:
```
fig,ax=plt.subplots()
for i in np.arange(manyY.shape[1]):
    labeltext='Column '+str(i)
    ax.plot(manyY[:,0],manyY[:,i],label=labeltext)
ax.legend(loc='best')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
```

Out[64]:
```
Text(0, 0.5, 'Y axis')
```



Compare a 2d array to a Pandas Dataframe

But now lets do it in pandas and see how it compares

In [94]:
```
df_manyY=pd.read_csv('oneX_manyY.csv')
df_manyY
```

Out[94]:

	x_values	first_y_values	second_y_values	third_y_values	fourth_y_values
0	1	1	10	4	6
1	2	2	9	4	6
2	3	3	8	4	6
3	4	4	7	4	6
4	5	5	6	4	6
5	6	6	5	4	6
6	7	7	4	4	6
7	8	8	3	4	6
8	9	9	2	4	6
9	10	10	1	4	6

Pandas is like an upgraded numpy array with column names.

This is going to make keeping track of data much nicer

using pandas we can use the column names

In [140…
```
df_manyY.columns
```

Out[140…
```
Index(['x_values', 'first_y_values', 'second_y_values', 'third_y_values',
       'fourth_y_values'],
      dtype='object')
```

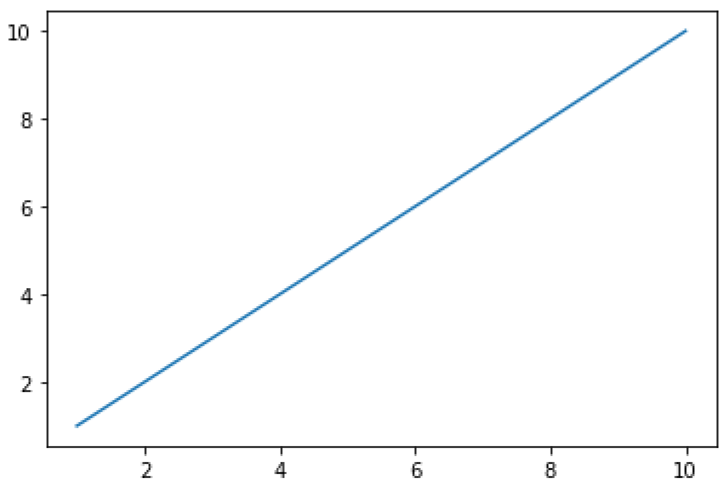
In [141…
```
df_manyY['x_values']
```

Out[141…
```
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10
Name: x_values, dtype: int64
```

making the plot in pandas

In [142…
```
fig,ax=plt.subplots()
ax.plot(df_manyY['x_values'],df_manyY['first_y_values'])
```

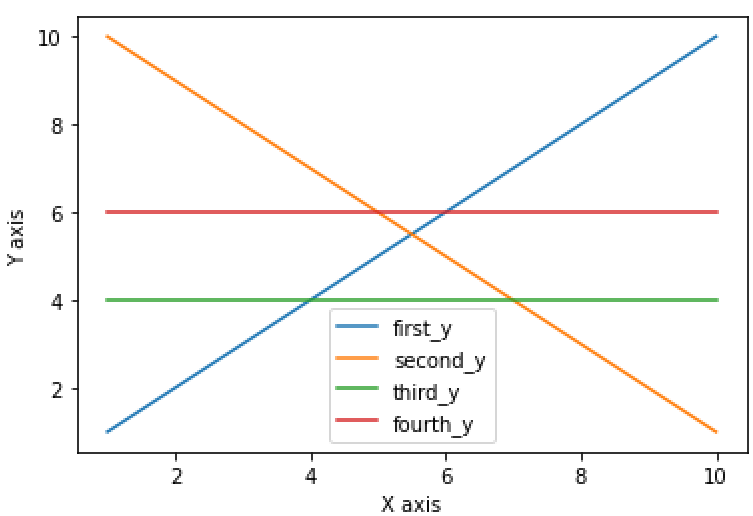
Out[142…
```
[<matplotlib.lines.Line2D at 0x7fad37397a30>]
```



Can you add the other columns and make a legend? Using Pandas?

In [86]:
```
```

Out[86]:
```
Text(0, 0.5, 'Y axis')
```



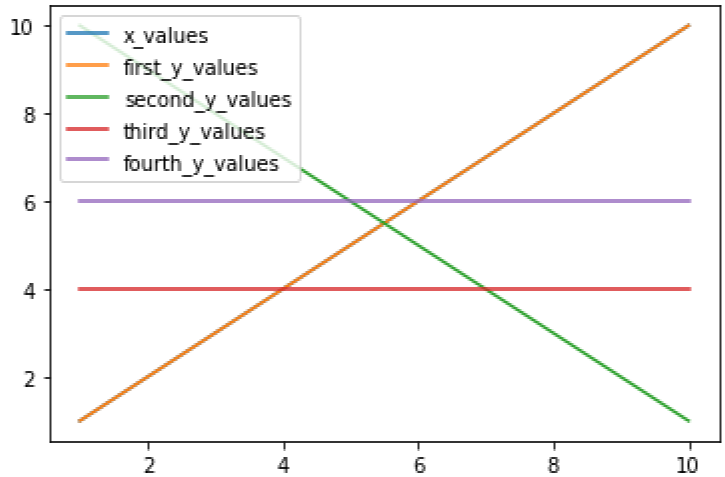
Pandas does some things to make your life easy. You can for loop over the columns. So the for loop returns the column name to col and you can pass that to ax.plot. We are going to be doing a lot more of this the next few weeks. So this is a sneak peak.

In [143…
```
fig,ax=plt.subplots()

for col in df_manyY:
    ax.plot(df_manyY['x_values'],df_manyY[col],label=col)

ax.legend()
```

Out[143…
```
<matplotlib.legend.Legend at 0x7fad373dbb80>
```



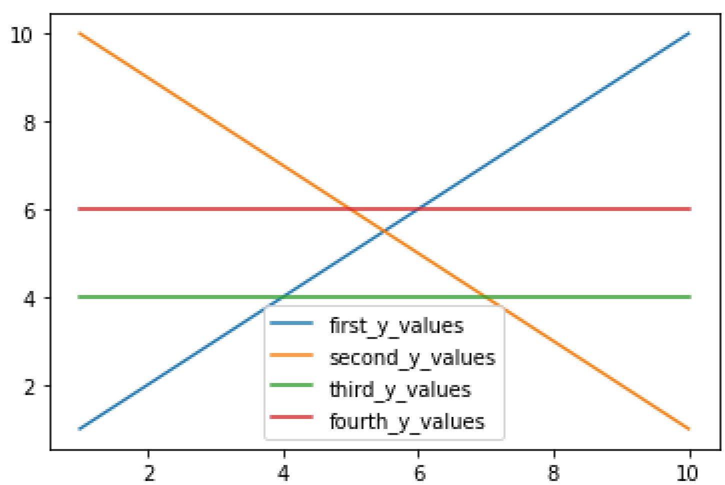
But this plots the first column versus itself. So we just need to only call from after the first column. We will learn how to do this next time. But here it is

In [144…
```
fig,ax=plt.subplots()

for col in df_manyY.iloc[:,1:]:
    ax.plot(df_manyY['x_values'],df_manyY[col],label=col)

ax.legend()
```

Out[144…
```
<matplotlib.legend.Legend at 0x7fad372bad90>
```



Mystery file

The file mystery.csv contains data in columns. Use what you know and plot the data! The first column is x values. The others are y values

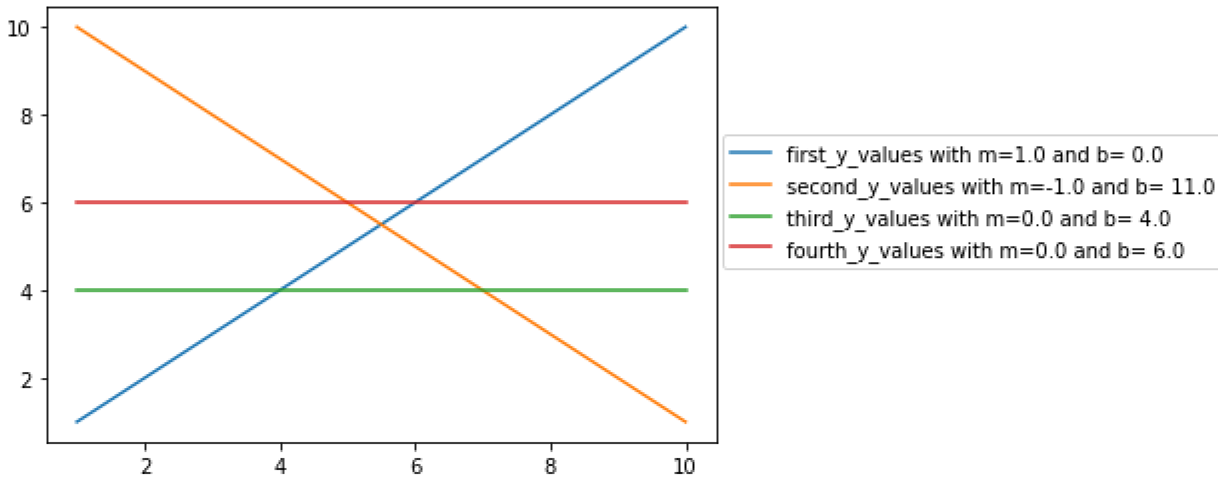
In [143…

Bonus.

If you got through this quickly see if you can go back to oneX_manyY.csv and get the equations for each line. You could do this in a for loop and adding each equation for a line to the legend...

In [145…

Out[145…
```
<matplotlib.legend.Legend at 0x7fad36eeac10>
```

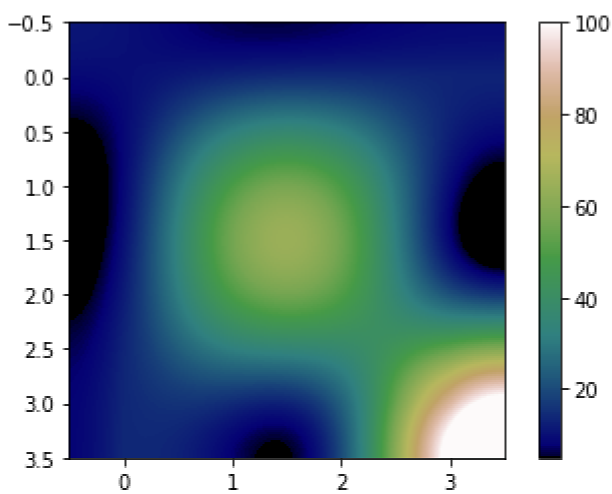


Answers

My own colorbar

In [49]:
```
plt.imshow(twoD,cmap='gist_earth',interpolation='bessel')
plt.colorbar()
```

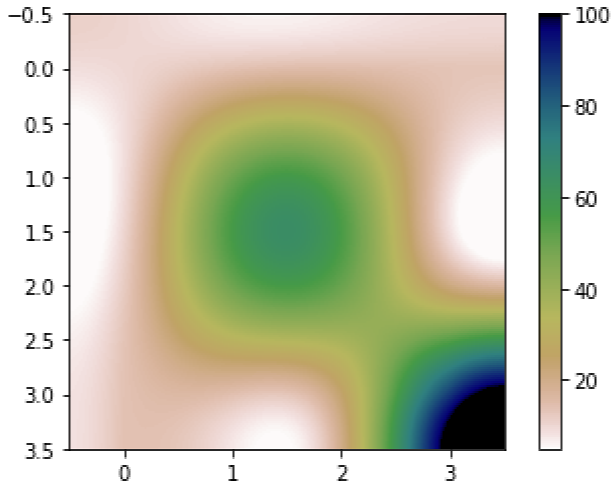
Out[49]:
```
<matplotlib.colorbar.Colorbar at 0x192eddb0358>
```



now reversed

In [50]:
```
plt.imshow(twoD,cmap='gist_earth_r',interpolation='bessel')
plt.colorbar()
```

Out[50]: <matplotlib.colorbar.Colorbar at 0x192ede435c0>



Brian result

In [34]:
```
Brian=pd.read_csv('Brian.csv',header=None)
Brian=Brian.values
plt.imshow(Brian,cmap='gnuplot',interpolation='none')
```

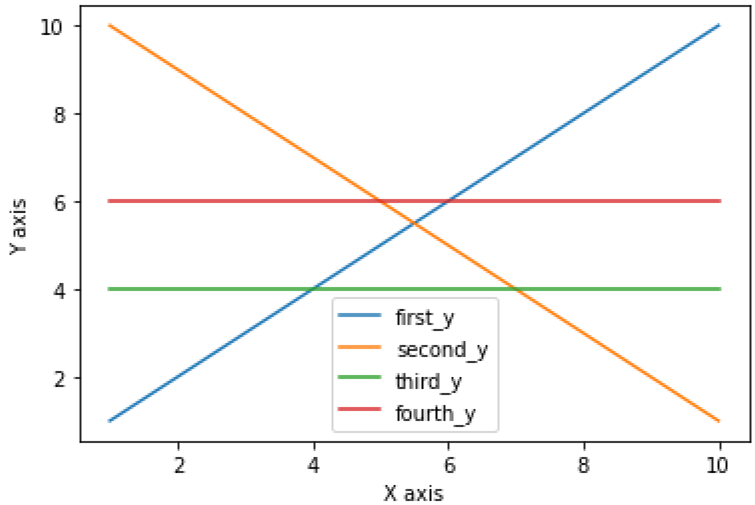
Out[34]: <matplotlib.image.AxesImage at 0x7fad31d6a070>

In [99]:
```
fig,ax=plt.subplots()

ax.plot(df_manyY['x_values'],df_manyY['first_y_values'],label='first_y')
ax.plot(df_manyY['x_values'],df_manyY['second_y_values'],label='second_y')
ax.plot(df_manyY['x_values'],df_manyY['third_y_values'],label='third_y')
ax.plot(df_manyY['x_values'],df_manyY['fourth_y_values'],label='fourth_y')

ax.legend(loc='best')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
```

Out[99]: Text(0, 0.5, 'Y axis')



Mystery File

In [3]:
```
df_mystery=pd.read_csv('mystery.csv')
df_mystery.columns
```

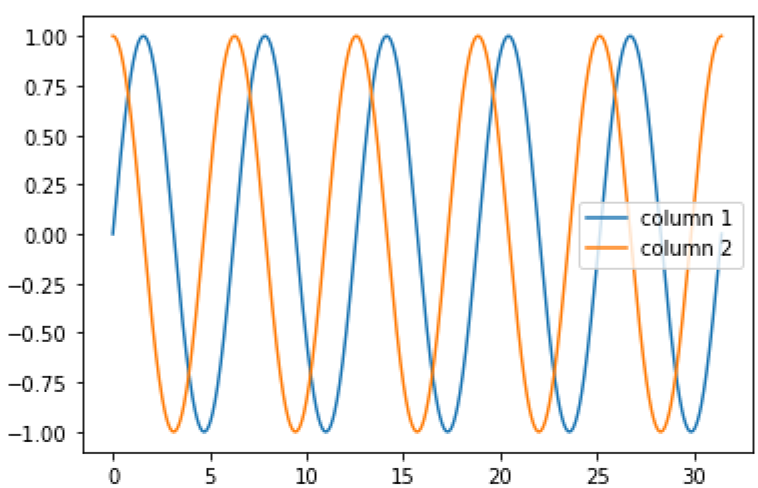
Out[3]: `Index(['Column_0', 'Column_1', 'Column_2'], dtype='object')`

In [4]:
```
df_mystery=pd.read_csv('mystery.csv')
fig,ax=plt.subplots()

ax.plot(df_mystery['Column_0'],df_mystery['Column_1'],label='column 1')
ax.plot(df_mystery['Column_0'],df_mystery['Column_2'],label='column 2')

ax.legend()
```

Out[4]: `<matplotlib.legend.Legend at 0x7f80cc49bee0>`



Using a for loop to get all the equations for the lines. Use linregress

In [133…
```
df_manyY=pd.read_csv('oneX_manyY.csv')

fig,ax=plt.subplots()

for col in df_manyY.iloc[:,1:]:
    x=df_manyY['x_values']
    y=df_manyY[col]
```

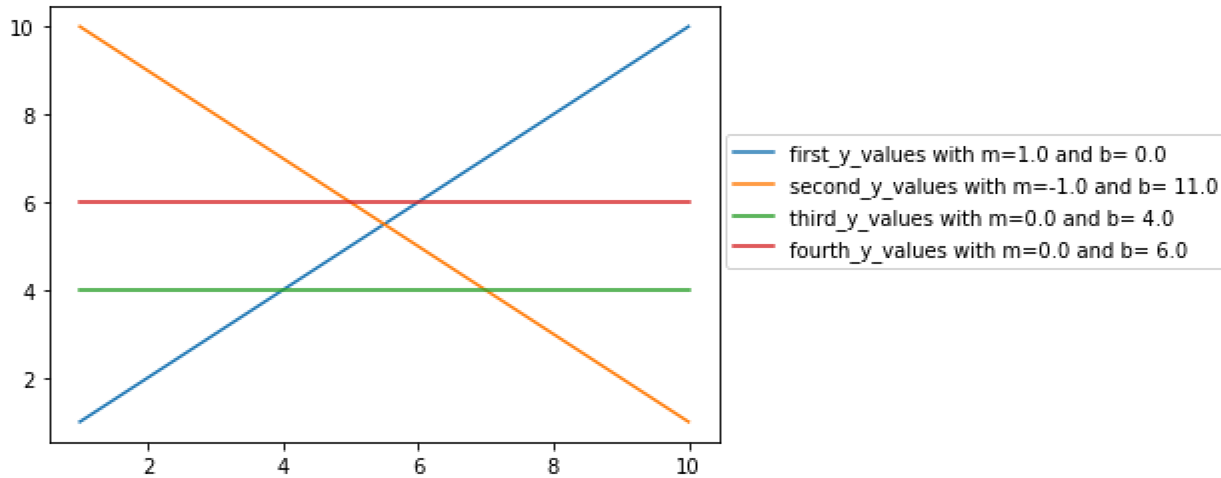
```
    slope, intercept, r_value,p_value,stderr = stats.linregress(x,y)
    label='{} with m={} and b= {}'.format(col,slope,intercept)

    ax.plot(df_manyY['x_values'],df_manyY[col],label=label)

ax.legend(loc=(1.01,0.4))
```

Out[133…

```
<matplotlib.legend.Legend at 0x7fad36ee2a00>
```



Code I copied from the web to show all the colormaps

In [ ]:

```
"""
Reference for colormaps included with Matplotlib.

This reference example shows all colormaps included with Matplotlib. Note that
any colormap listed here can be reversed by appending "_r" (e.g., "pink_r").
These colormaps are divided into the following categories:

Sequential:
    These colormaps are approximately monochromatic colormaps varying smoothly
    between two color tones---usually from low saturation (e.g. white) to high
    saturation (e.g. a bright blue). Sequential colormaps are ideal for
    representing most scientific data since they show a clear progression from
    low-to-high values.

Diverging:
    These colormaps have a median value (usually light in color) and vary
    smoothly to two different color tones at high and low values. Diverging
    colormaps are ideal when your data has a median value that is significant
    (e.g.  0, such that positive and negative values are represented by
    different colors of the colormap).

Qualitative:
    These colormaps vary rapidly in color. Qualitative colormaps are useful for
    choosing a set of discrete colors. For example::

        color_list = plt.cm.Set3(np.linspace(0, 1, 12))

    gives a list of RGB colors that are good for plotting a series of lines on
    a dark background.

Miscellaneous:
    Colormaps that don't fit into the categories above.
```

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


cmaps = [('Sequential',     ['Blues', 'BuGn', 'BuPu',
                             'GnBu', 'Greens', 'Greys', 'Oranges', 'OrRd',
                             'PuBu', 'PuBuGn', 'PuRd', 'Purples', 'RdPu',
                             'Reds', 'YlGn', 'YlGnBu', 'YlOrBr', 'YlOrRd']),
         ('Sequential (2)', ['afmhot', 'autumn', 'bone', 'cool', 'copper',
                             'gist_heat', 'gray', 'hot', 'pink',
                             'spring', 'summer', 'winter']),
         ('Diverging',      ['BrBG', 'bwr', 'coolwarm', 'PiYG', 'PRGn', 'PuOr',
                             'RdBu', 'RdGy', 'RdYlBu', 'RdYlGn', 'Spectral',
                             'seismic']),
         ('Qualitative',    ['Accent', 'Dark2', 'Paired', 'Pastel1',
                             'Pastel2', 'Set1', 'Set2', 'Set3']),
         ('Miscellaneous',  ['gist_earth', 'terrain', 'ocean', 'gist_stern',
                             'brg', 'CMRmap', 'cubehelix',
                             'gnuplot', 'gnuplot2', 'gist_ncar',
                             'nipy_spectral', 'jet', 'rainbow',
                             'gist_rainbow', 'hsv', 'flag', 'prism'])]


nrows = max(len(cmap_list) for cmap_category, cmap_list in cmaps)
gradient = np.linspace(0, 1, 256)
gradient = np.vstack((gradient, gradient))

def plot_color_gradients(cmap_category, cmap_list):
    fig, axes = plt.subplots(nrows=nrows)
    fig.subplots_adjust(top=0.95, bottom=0.01, left=0.2, right=0.99)
    axes[0].set_title(cmap_category + ' colormaps', fontsize=14)

    for ax, name in zip(axes, cmap_list):
        ax.imshow(gradient, aspect='auto', cmap=plt.get_cmap(name))
        pos = list(ax.get_position().bounds)
        x_text = pos[0] - 0.01
        y_text = pos[1] + pos[3]/2.
        fig.text(x_text, y_text, name, va='center', ha='right', fontsize=10)

    # Turn off *all* ticks & spines, not just the ones with colormaps.
    for ax in axes:
        ax.set_axis_off()

for cmap_category, cmap_list in cmaps:
    plot_color_gradients(cmap_category, cmap_list)

plt.show()
```

In [ ]:

In [ ]:

In [ ]: