Week 3 Notes

We'll be talking about multiple plots with the same figure, interaction, animation and a few more kinds of plots that you may find useful in your data science journey. Do check out the Matplotlib mailing list as well. It's pretty common with open source projects to have 2 mailing lists - one for developers and one for users. The users list is where most of the question and answering happens, but you should take a look at the developer list sometimes as well. A further reading will be provided for this as well.

03-01: Subplots

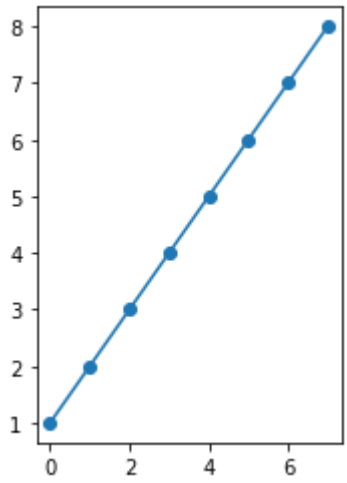
Sometimes its useful to show 2 figures side by side for viewers to compare.

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

plt.subplot?
```

When we look at the documentation, we see this:

```
    Call signatures::


subplot(nrows, ncols, index, kwargs)
   subplot(pos, kwargs)
   subplot(**kwargs)
   subplot(ax)
</code>
We see that the first argument is the number of rows, 2nd is the number
of columns, and third is the plot number.
```

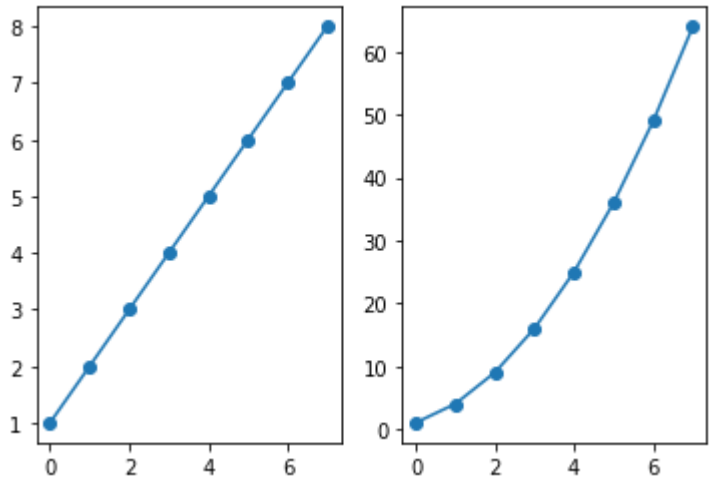
In [5]:
```
plt.figure()
# subplot with 1 row, 2 columns, and set the first axes to the current axis.
plt.subplot(1, 2, 1)

linear_data = np.array([1,2,3,4,5,6,7,8])

plt.plot(linear_data, '-o')
plt.show()
```



Shifting over to the other side

In [6]:

```
plt.figure()
# subplot with 1 row, 2 columns, and set the first axes to the current axis.
plt.subplot(1, 2, 1)

linear_data = np.array([1,2,3,4,5,6,7,8])

plt.plot(linear_data, '-o')
exponential_data = linear_data**2

# subplot with 1 row, 2 columns, and current axis is 2nd subplot axes
plt.subplot(1, 2, 2)
plt.plot(exponential_data, '-o')
plt.show()
```



The norm with Matplotlib is that you **store the axes object** that you get back from the subplot. But you can call subplot again by all means, in order to get back a given axes.

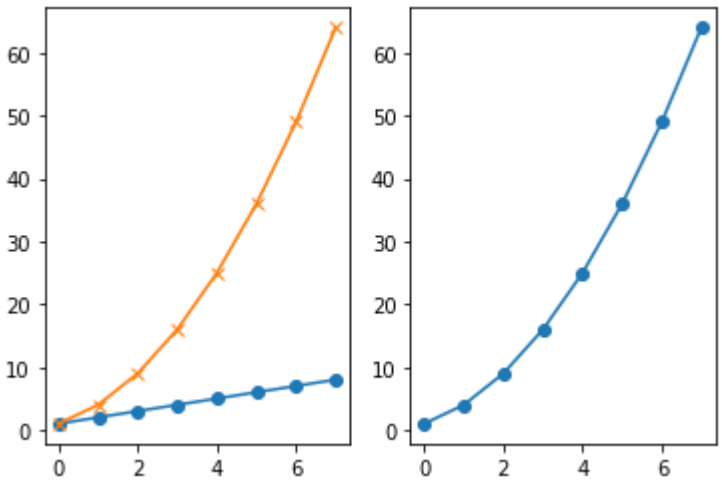
In [8]:

```
plt.figure()
# subplot with 1 row, 2 columns, and set the first axes to the current axis.
plt.subplot(1, 2, 1)

linear_data = np.array([1,2,3,4,5,6,7,8])

plt.plot(linear_data, '-o')
exponential_data = linear_data**2

# subplot with 1 row, 2 columns, and current axis is 2nd subplot axes
plt.subplot(1, 2, 2)
```

```
plt.plot(exponential_data, '-o')
# plot exponential data on 1st subplot axes
plt.subplot(1, 2, 1)
plt.plot(exponential_data, '-x')
```

```
<ipython-input-8-3824c70bea20>:14: MatplotlibDeprecationWarning: Adding an axe
s using the same arguments as a previous axes currently reuses the earlier ins
tance.  In a future version, a new instance will always be created and returne
d.  Meanwhile, this warning can be suppressed, and the future behavior ensure
d, by passing a unique label to each axes instance.
  plt.subplot(1, 2, 1)
```

Out[8]: `[<matplotlib.lines.Line2D at 0x7fd30dbea1c0>]`



Mislead User Warning

This demonstrates a common problem. It **looked like** linear had roughly the same area under the line on the chart until we asked matplotlib to put them into 1 graph, then the y axis was refreshed. This would be an unconscious attempt to mislead a reader if we did not find a way to lock axis between 2 plots.

Sharing Axes using `sharex=` and `sharey=` parameters

We create a subplot on the left hand side and store it in ax1. When we create the subplot for the right hand side we explicitly state we want to share the y axis using `sharey=ax1`. Note that we don't have to store the subplot to a variable like we did with `ax1`.

Remember that when you use the scripting interface, pyplot is going to **get current axis** via the function `gca()` underneath. So `calling pyplot.plot()` will work given the current axes that we were using when we called `pyplot.subplot()` the 2nd time.

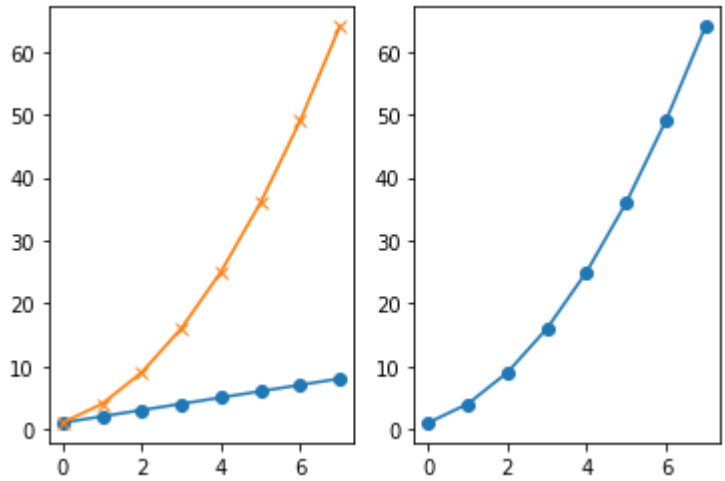
In [9]:

```
plt.figure()
# subplot with 1 row, 2 columns, and set the first axes to the current axis.
plt.subplot(1, 2, 1)

linear_data = np.array([1,2,3,4,5,6,7,8])

plt.plot(linear_data, '-o')
exponential_data = linear_data**2

# subplot with 1 row, 2 columns, and current axis is 2nd subplot axes
plt.subplot(1, 2, 2)
plt.plot(exponential_data, '-o')
# plot exponential data on 1st subplot axes
```

```
plt.subplot(1, 2, 1)
plt.plot(exponential_data, '-x')
```

```
<ipython-input-9-3824c70bea20>:14: MatplotlibDeprecationWarning: Adding an axe
s using the same arguments as a previous axes currently reuses the earlier ins
tance.  In a future version, a new instance will always be created and returne
d.  Meanwhile, this warning can be suppressed, and the future behavior ensure
d, by passing a unique label to each axes instance.
  plt.subplot(1, 2, 1)
```

Out[9]: `[<matplotlib.lines.Line2D at 0x7fd30dd39070>]`



Mislead User Warning

This demonstrates a common problem. It **looked like** linear had roughly the same area under the line on the chart until we asked matplotlib to put them into 1 graph, then the y axis was refreshed. This would be an unconscious attempt to mislead a reader if we did not find a way to lock axis between 2 plots.

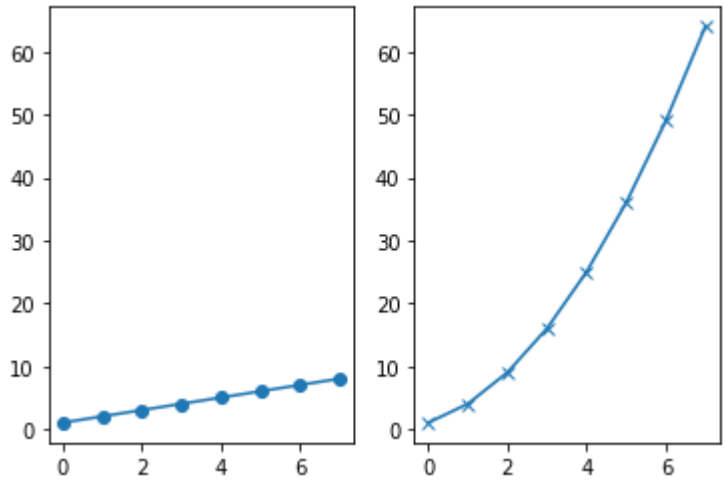
Sharing Axes using `sharex=` and `sharey=` parameters

We create a subplot on the left hand side and store it in ax1. When we create the subplot for the right hand side we explicitly state we want to share the y axis using `sharey=ax1`. Note that we don't have to store the subplot to a variable like we did with `ax1`.

Remember that when you use the scripting interface, pyplot is going to **get current axis** via the function `gca()` underneath. So `calling pyplot.plot()` will work given the current axes that we were using when we called `pyplot.subplot()` the 2nd time.

In [10]:
```
plt.figure()
ax1 = plt.subplot(1, 2, 1)
plt.plot(linear_data, '-o')
# pass sharey=ax1 to ensure the two subplots share the same y axis
ax2 = plt.subplot(1, 2, 2, sharey=ax1)
plt.plot(exponential_data, '-x')
```

Out[10]: `[<matplotlib.lines.Line2D at 0x7fd30de352e0>]`



Shorthand Call to Subplot

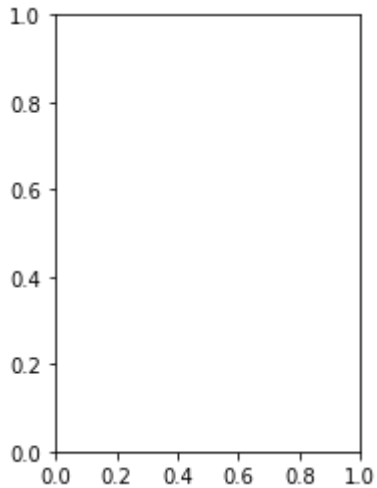
- Hundred place is rows
- Tens place is the columns
- Ones digit is the plot number.

Remember that the subplot count starts at position 1 (not 0). So when you are iterating through subplots, remember to end at `No. of subplots +1`

In [11]:
```
plt.figure()
# the right hand side is equivalent shorthand syntax
plt.subplot(1,2,1) == plt.subplot(121)
```

```
<ipython-input-11-2c808a67d888>:3: MatplotlibDeprecationWarning: Adding an axe
s using the same arguments as a previous axes currently reuses the earlier ins
tance.  In a future version, a new instance will always be created and returne
d.  Meanwhile, this warning can be suppressed, and the future behavior ensure
d, by passing a unique label to each axes instance.
  plt.subplot(1,2,1) == plt.subplot(121)
```

Out[11]: True

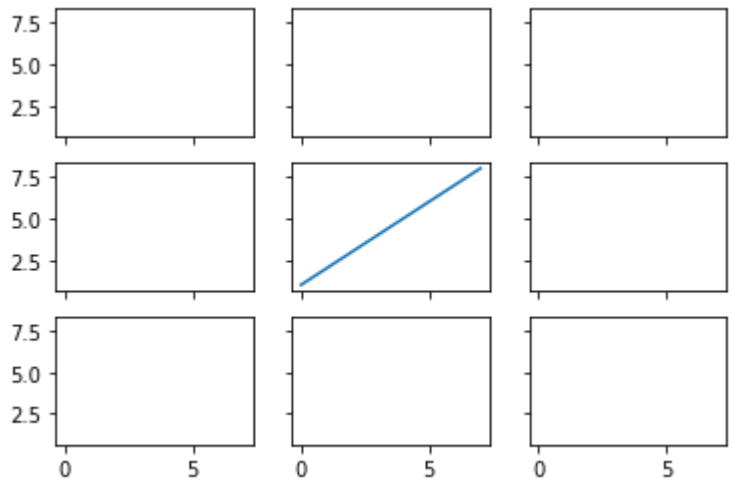


Function subplotS <-- Note the plural.

Let's say we want to get a 3x3 grid with all of the axis x and y ranges locked, we can do so like this. Note that this, by default turns of all the ticklabels except those at the left and bottom of the figure.

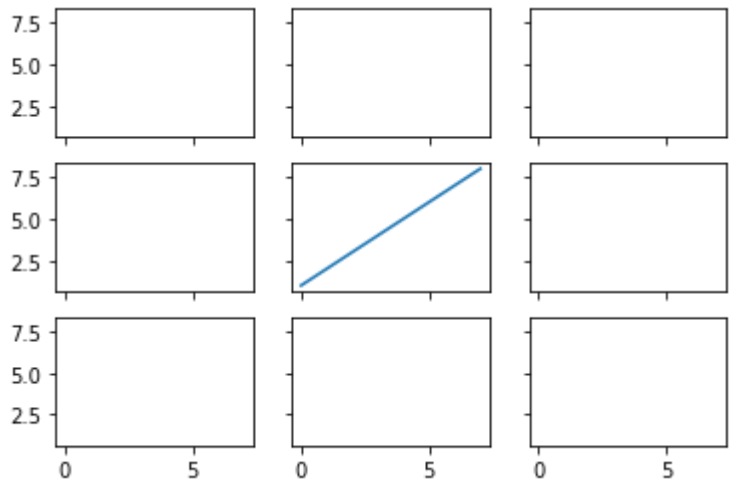
In [12]:
```
# create a 3x3 grid of subplots
fig, ((ax1,ax2,ax3), (ax4,ax5,ax6), (ax7,ax8,ax9)) = plt.subplots(3, 3, share
# plot the linear_data on the 5th subplot axes
ax5.plot(linear_data, '-')
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x7fd30e169130>]
```



Turning the labels back on...

In [13]:
```
# create a 3x3 grid of subplots
fig, ((ax1,ax2,ax3), (ax4,ax5,ax6), (ax7,ax8,ax9)) = plt.subplots(3, 3, share
# plot the linear_data on the 5th subplot axes
ax5.plot(linear_data, '-')
# set inside tick labels to visible
for ax in plt.gcf().get_axes():
    for label in ax.get_xticklabels() + ax.get_yticklabels():
        label.set_visible(True)
plt.show()
```



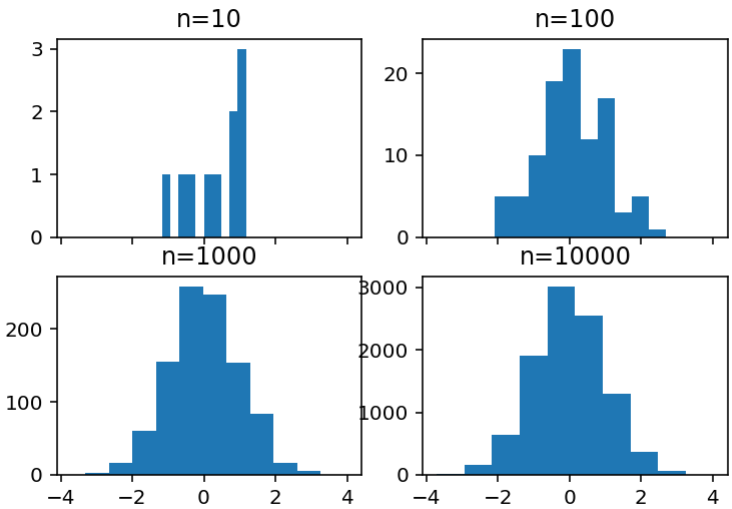
In [14]:
```
# necessary on some systems to update the plot
plt.gcf().canvas.draw()
```

```
<Figure size 432x288 with 0 Axes>
```

03-02: Histograms

Plotting Histograms using `axis.hist(sample data)`

In [15]:
```
%matplotlib notebook
# create 2x2 grid of axis subplots
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True)
axs = [ax1,ax2,ax3,ax4]

# draw n = 10, 100, 1000, and 10000 samples from the normal distribution and
for n in range(0,len(axs)):
    sample_size = 10**(n+1)
    sample = np.random.normal(loc=0.0, scale=1.0, size=sample_size)
    axs[n].hist(sample)
    axs[n].set_title('n={}'.format(sample_size))
```



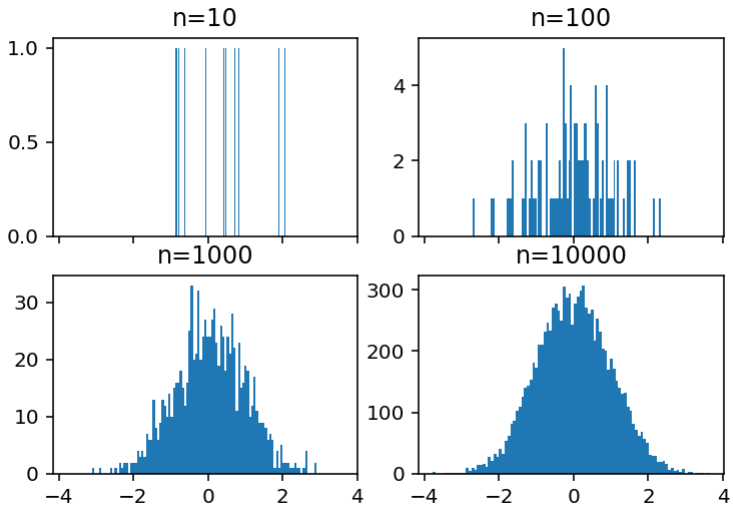
Data Bins in Matplotlib

You realise that the bars for the graph where n = 10000 is wider than the graph where n = 10. You'll also realise that by default, the histogram in Matplotlib uses 10 bins, that is, for 10 different bars. Here, we created a shared x axis, and as we sample more from the distribution, we're more likely to get outlier values. Thus, 10 bins for n = 10 is just nice to capture 10 different values, while for n = 10000, many values have to be combined into a single bin.

Changing the bin size to 100

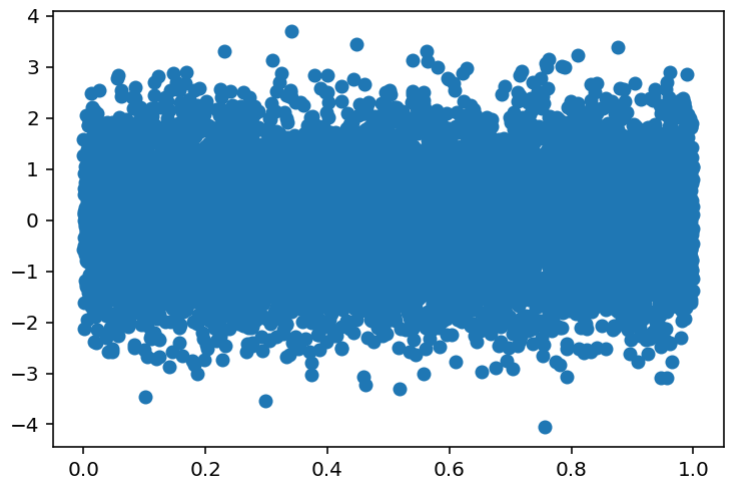
This is testament to the ongoing analysis of how much granularity you need for a specific set of data.

In [16]:
```
# repeat with number of bins set to 100
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, sharex=True)
axs = [ax1,ax2,ax3,ax4]

for n in range(0,len(axs)):
    sample_size = 10**(n+1)
    sample = np.random.normal(loc=0.0, scale=1.0, size=sample_size)
    axs[n].hist(sample, bins=100)
    axs[n].set_title('n={}'.format(sample_size))
```



Using GridSpec

In [17]:
```
plt.figure()
Y = np.random.normal(loc=0.0, scale=1.0, size=10000)
X = np.random.random(size=10000)
plt.scatter(X,Y)
```



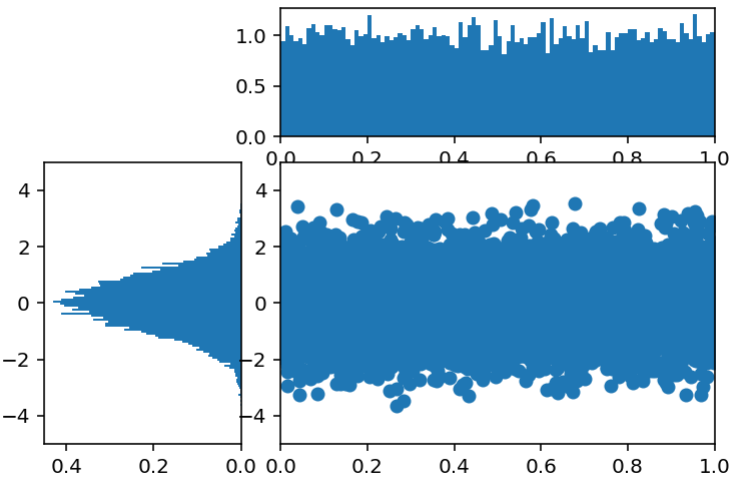
Out[17]: `<matplotlib.collections.PathCollection at 0x7fd30ea66ca0>`

To use the gridspec, we first import it, then create a new GridSpec to the overall shape that we want. When we add new items with the subplot, instead of specifying the three numbers of row, column and position, we pass in the elements of the GridSpec object which we wish to cover. **Because we are using the elements of a list, all of the indexing starts at 0**, and it is very reasonable to **use slicing** for the beginning or ends of lists.

The gridspec is indexed by the **indexing operator** `[ rows , columns]`

In [18]:
```
# use gridspec to partition the figure into subplots
import matplotlib.gridspec as gridspec
```

```
plt.figure()
gspec = gridspec.GridSpec(3, 3)

top_histogram = plt.subplot(gspec[0, 1:]) # takes up the width of 2 plots hor
side_histogram = plt.subplot(gspec[1:, 0]) #takes up the width of 2 plots ver
lower_right = plt.subplot(gspec[1:, 1:]) # takes up the width of 2 plots vert
```



Put in some values

In [19]:

```
Y = np.random.normal(loc=0.0, scale=1.0, size=10000)
X = np.random.random(size=10000)
lower_right.scatter(X, Y)
top_histogram.hist(X, bins=100)
s = side_histogram.hist(Y, bins=100, orientation='horizontal')
```

Clearing a histogram

In [20]:

```
# clear the histograms and plot normed histograms
top_histogram.clear()
top_histogram.hist(X, bins=100, density=True)
side_histogram.clear()
side_histogram.hist(Y, bins=100, orientation='horizontal', density=True)
# flip the side histogram's x axis
side_histogram.invert_xaxis()
```

In [21]:

```
# change axes limits
for ax in [top_histogram, lower_right]:
    ax.set_xlim(0, 1)
for ax in [side_histogram, lower_right]:
    ax.set_ylim(-5, 5)
```

MOOC DATA

03-03: Box and Whisker Plots

Sometimes called box and whisker plots, it is a method of showing aggregate statistics of various samples **in a concise manner**. The box plot simultaneously shows, for each sample,

the median of each value, the minimum and maximum of the samples, and the **interquartile range**. Let's create three different samplings from NumPy. One for the normal distribution, one for random, and one from a gamma distribution and load it into the a pandas dataframe.

In [22]:
```
import pandas as pd
normal_sample = np.random.normal(loc=0.0, scale=1.0, size=10000)
random_sample = np.random.random(size=10000)
gamma_sample = np.random.gamma(2, size=10000)

df = pd.DataFrame({'normal': normal_sample,
                   'random': random_sample,
                   'gamma': gamma_sample})
```

Summary of Dataframe using `df.describe()`

In [23]:
```
df.describe()
```

Out[23]:

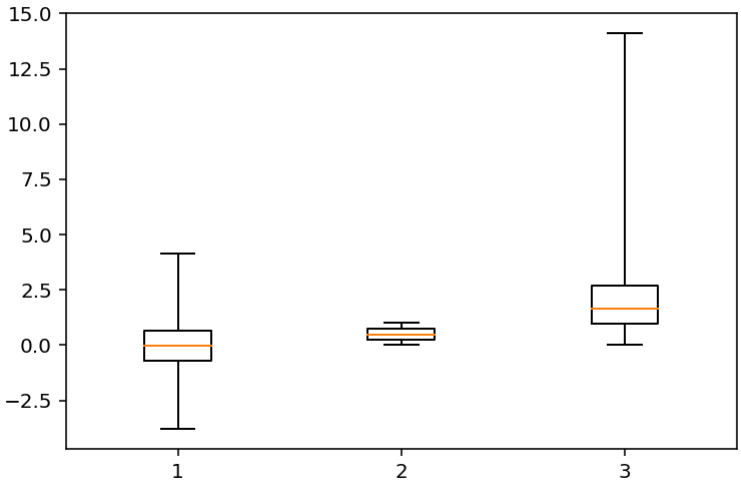
	normal	random	gamma
count	10000.000000	10000.000000	10000.000000
mean	-0.015152	0.492368	1.980459
std	1.006779	0.288869	1.402539
min	-3.797241	0.000047	0.012793
25%	-0.694599	0.243916	0.951711
50%	-0.019684	0.486142	1.658678
75%	0.664063	0.744095	2.672370
max	4.130899	0.999963	14.115630

Creating a Boxplot

Visualising the normal data. Syntax: `plt.boxplot(col we want to visualise, whis = [lower end,higher end]*` such that the **whiskers** of the plot extend to the maximum and minimum of the dataset. In addition, to suppress output, we assign the boxplot function to a variable, which is just an underscore.

*[0,100] represents the entire range of data.

In [24]:
```
plt.figure()
# create a boxplot of the normal data, assign the output to a variable to sup
_ = plt.boxplot(df['normal'], whis=[0,100])
```



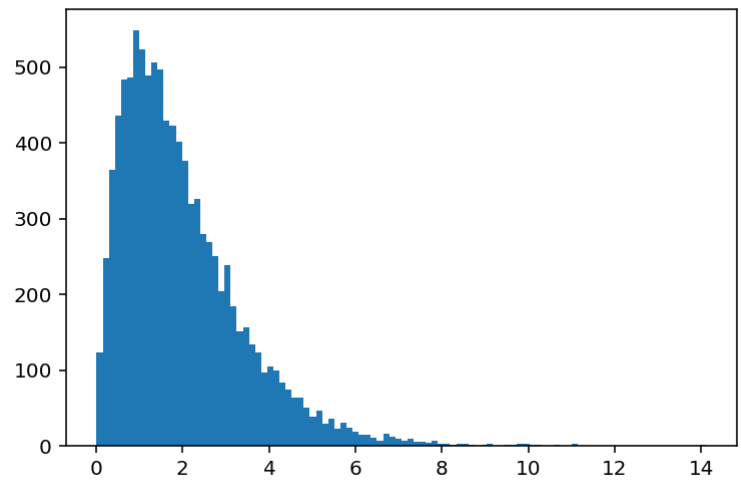
Creating Three Boxplots

In [25]:
```
# clear the current figure
plt.clf()
# plot boxplots for all three of df's columns
_ = plt.boxplot([ df['normal'], df['random'], df['gamma'] ], whis=[0,100])
```

Observe this in A Histogram

Plotting the gamma distribution on a histogram.

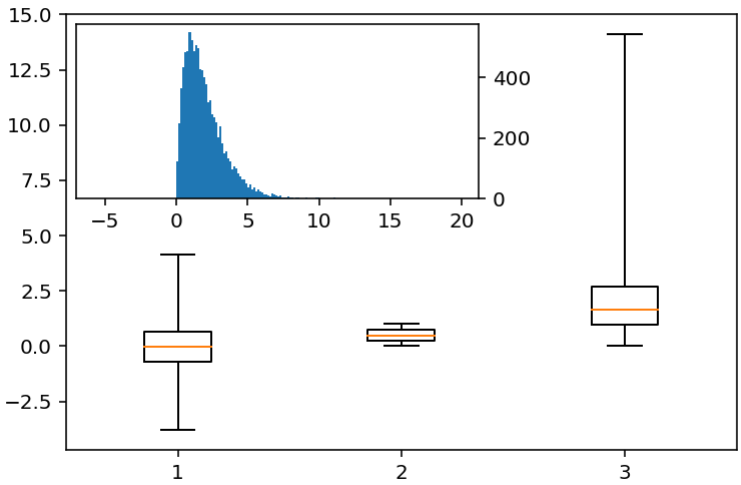
In [26]:
```
plt.figure()
_ = plt.hist(df['gamma'], bins=100)
```



Adding The Histogram to our Boxplot using `inset_axes()`

We can actually overlay an axes on top of another within a figure. This functionality isn't in the basic matplotlib space, but its in the toolkits, which tend to ship with matplotlib. There are several different toolkits available, and while they are considered *packaged*, they aren't considered core.

In [27]:
```
import mpl_toolkits.axes_grid1.inset_locator as mpl_il

plt.figure()
plt.boxplot([ df['normal'], df['random'], df['gamma'] ], whis=[0,100])
# overlay axis on top of another
ax2 = mpl_il.inset_axes(plt.gca(), width='60%', height='40%', loc=2)#loc 2 pl
ax2.hist(df['gamma'], bins=100)
ax2.margins(x=0.5)
```



Changing the y-axis ticks and tick labels

In [28]:
```
# switch the y axis ticks for ax2 to the right side
ax2.yaxis.tick_right()
```

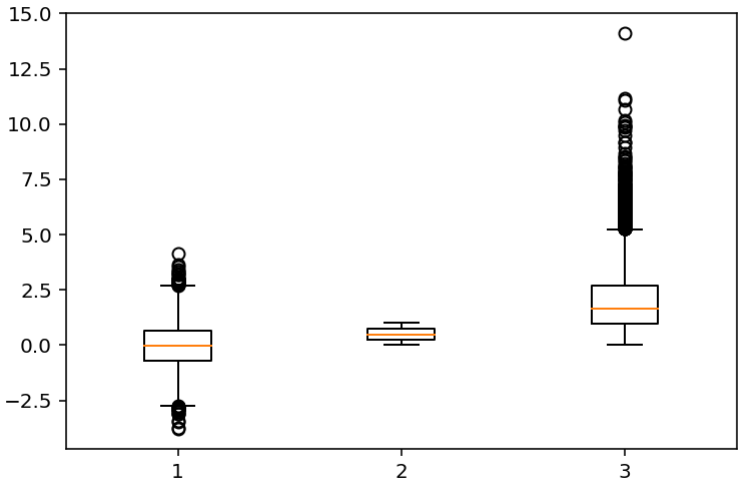
Default `whis` tendencies

If you don't supply the whis argument, the whiskers actually only go out to halfway between the interquartile range. You can figure this out through the top of the box minus the bottom of the box and multiply the value by 1.5.

Spotting Outliers

By not putting a whis argument, this can actually help us to detect fliers. You can see how this method of outlier detection differs with respect to our three distributions. You can also plot the confidence interval in a couple of different ways on the data. The most common is to **add notches** to the box plot represent the 95% confidence interval of the data and there are lots of other ways to customise the box plot.

In [29]:
```
# if `whis` argument isn't passed, boxplot defaults to showing 1.5*interquart
plt.figure()
_ = plt.boxplot([ df['normal'], df['random'], df['gamma'] ] )
```



Summary

The boxplot is one of the more common plots you might use as a data scientist, and matplotlib has significant support for different kinds of box plots. Here the matplotlib documentation is key. You can find links in the course resources to the API, which describes the box plot functionality.

03-04: Heatmaps (2-D Histogram)

Heatmaps are a way to visualise **3D data** and to take advantage of **spatial proximity** of those dimensions.

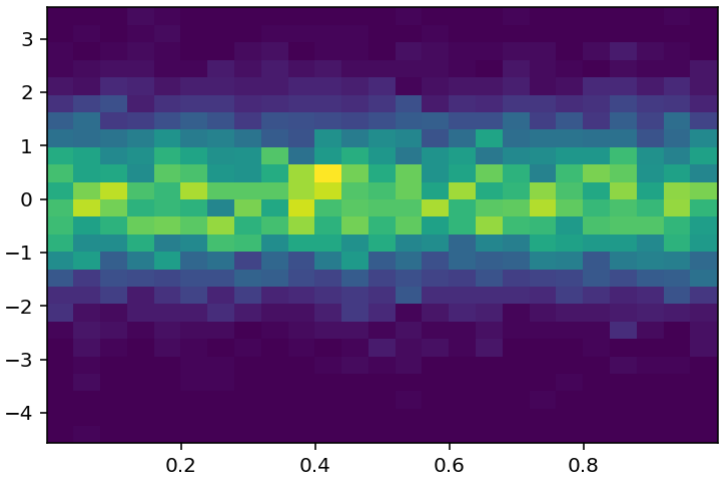
For instance, you have 2 dimensions, *latitude* and *longitude*, and you can overlay a third dimension, say *temperature* using **color** to denote its intensity.

When not to use heatmaps

Using a heatmap for categorical data, for instance, is just plain wrong. It misleads the user to looking for patterns and ordering through spatial proximity, and any such patterns would be purely spurious.

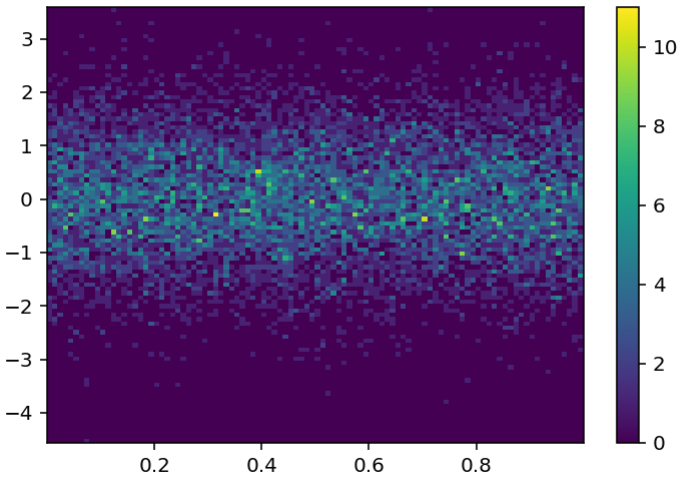
Creating a Heatmap using `hist2d`

In [30]:
```
plt.figure()

Y = np.random.normal(loc=0.0, scale=1.0, size=10000)
X = np.random.random(size=10000)
_ = plt.hist2d(X, Y, bins=25)
```



Increasing the bin size on a heatmap

In [31]:
```
plt.figure()
_ = plt.hist2d(X, Y, bins=100)
```



Adding a Colour Legend using `plt.colorbar()`

When using pyplot as a scripting layer, this is as easy as calling the `colorbar()` function. Be warned, there is **A LOT** of accounting and bookkeeping happening under the hood.

In fact, pyplot is 1) finding the current rendered image,

2) generating the color bar,

3) resizing the existing axes,

4) using **gridspec** to add a new axes the size of the color bar.

In [32]:
```
# add a colorbar legend
plt.colorbar()
```

Out[32]: <matplotlib.colorbar.Colorbar at 0x7fd310a3ea90>

03-05: Animations

So far we focused on static images, but matplotlib does have some support for both animation and interactivity.

We recall that the backend that renders the plot to the screen. Animation and interactivity heavily depend on support from this backend layer. The `nbagg` backend or the `%matplotlib notebook` magic function does provide some interactivity, and we can leverage that here.

Module `matplotlib.animation`

The matplotlib.animation module contains important helpers for building animations. For our discussion, the important object here is to call `FuncAnimation`, and it builds an animation by **iteratively calling a function which you define**. Essentially, your function will either clear the axis object and redraw the next frame, which you want users to see or will return a list of objects which need to be redrawn.

In [33]:
```
import matplotlib.animation as animation

n = 100
x = np.random.randn(n)
x
```

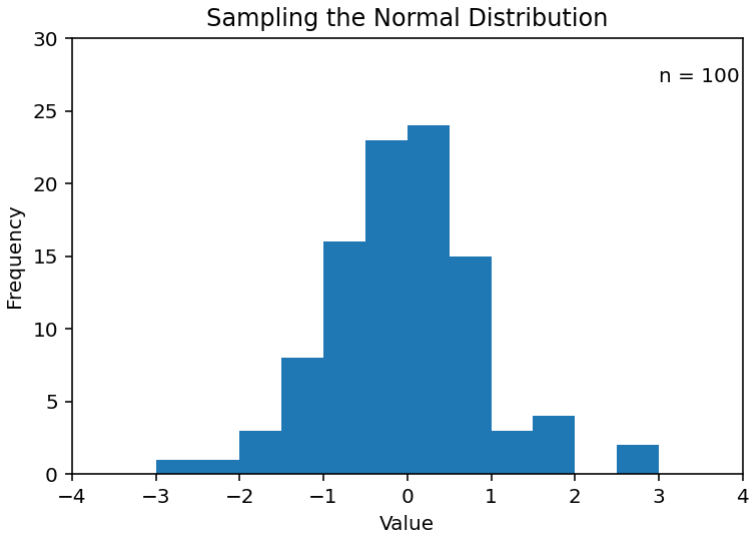
Out[33]:
```
array([-0.74517446, -1.2775357 , -1.34161355,  0.30484248,  0.25074006,
       -0.31578603, -0.50212037, -2.30234154,  0.17799653,  0.35501142,
       -0.35888391, -0.79767001,  0.17749595, -0.57894642, -0.03393466,
        0.29420338,  0.04514441, -0.06719317,  0.60573436, -0.46298592,
        2.6138228 ,  0.24886981, -0.3366251 ,  0.41617401, -0.98007793,
       -0.1392359 ,  0.73353166,  0.57022669, -0.88024503, -0.56694939,
       -0.05296663,  0.17026272, -0.123445  , -1.16006723,  0.58733743,
       -0.49458327,  1.07920872, -0.62638859,  1.6826339 ,  0.88705187,
        1.11856736, -1.60645951, -0.31844167, -0.18658876, -0.48147123,
        0.2678201 ,  0.25745922,  0.87024596,  0.90908562,  0.94872929,
       -1.28983787,  0.41763449,  0.38226099, -1.83890758,  0.2636709 ,
        0.69441339,  0.09103246, -0.09397129, -0.59702115, -0.59699108,
        1.24969745, -0.18380215, -0.41356675, -0.9688717 ,  0.57903809,
        1.66615563, -0.19466727, -0.14397562,  0.42651128,  0.73052082,
       -0.78615044,  2.7878789 ,  0.62369108, -0.54570119,  0.16965874,
       -0.40941711, -0.14031968,  0.28184052,  0.55810434,  0.23969515,
        1.56203287,  0.29462025, -2.68341542, -0.64863624, -1.48190454,
       -1.28585197, -0.0066058 , -1.37341382, -0.32950465,  0.98791678,
       -1.57194505,  0.32538264,  0.40743851, -0.8427985 ,  1.70066222,
       -0.10842604, -1.19819176,  0.80409686, -0.58312655,  0.10112116])
```

In [34]:
```
# create the function that will do the plotting, where curr is the current fr
def update(curr):
    # check if animation is at the last frame, and if so, stop the animation
    if curr == n:
        a.event_source.stop()
    plt.cla() #clear current axis
    bins = np.arange(-4, 4, 0.5) # last parameter is the spacing between bins
    plt.hist(x[:curr], bins=bins)
    plt.axis([-4,4,0,30])  # also you don't want the axis to change, so hard
    plt.gca().set_title('Sampling the Normal Distribution')
    plt.gca().set_ylabel('Frequency')
```

```
    plt.gca().set_xlabel('Value')
    plt.annotate('n = {}'.format(curr), [3,27])
```

In [41]:

```
fig = plt.figure()
a = animation.FuncAnimation(fig, # figure
                            update, # function reference
                            interval=100) # time in milliseconds
```



Summary

A figure animation with 4 subplots, one for each kind of distribution we might be interested in understanding, could be pretty neat. We could plot the samples for the normal distribution in one, for a gamma distribution in another, then maybe a couple of parameterised distributions like the normal distribution with different levels of standard deviation. This would be a great way to practice the skills that you've learned in this module, as iit would require that you manage subplots within an animation using histograms.

03-06: Interactivity

Interactivity and animation are very similar in Matplotlib. For interactivity though, we have to head down to the artist layer a bit more. In particular, we have to reference the `Canvas` object of the current figure.

The `Canvas` Object

The *canvas* object handles all of the drawing events and it's tightly connected with a given backend.

Events

If event listening is something you're not familiar with, it can be a bit of a tough concept to grasp. The abstraction used was largely focused on the notion of events. This could be, for instance, moving a mouse pointer would create an event, clicking will create an event, pressing on keys on the keyboard would create an event, and this didn't only happen at the

hardware level such as IRQ interrupts but at the software level as well. In fact, **event driven programming** has infiltrated most of the ways computer programmers regularly engage with software. From HTML and JavScript down to lower level C code.

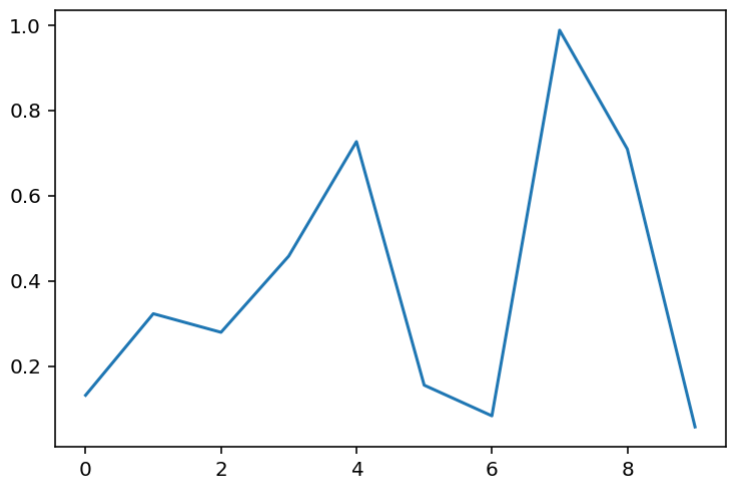
You can think of an event as a piece of data which is associated with a function call. And when the event happens, the software environment, in our case this is the **Matplotlib's backend**, will call the function with the relevant data.

Example: Event `onclick()`

For this example function onclick, we'll clear the current axis then plot our data, then set the title of the plot to be the location of the mouse. Finally, we have to connect this events to an event listener and this process is called **wiring it up**. In this case, it's very easy, get the current figure and its canvas subject then call the mpl_connect function. Passing in the string for button_press_event as well as reference to the function `onclick()`, which will be called when the event is detected.

Now when we click our plot we see the information printed to the title.

In [36]:
```
plt.figure()
data = np.random.rand(10)
plt.plot(data)

def onclick(event):
    plt.cla()
    plt.plot(data)
    plt.gca().set_title('Event at pixels {},{} \nand data {},{}'.format(event

# tell mpl_connect we want to pass a 'button_press_event' into onclick when t
plt.gcf().canvas.mpl_connect('button_press_event', onclick)
```



Out[36]: 9

The matplotlib documentation describes the type of events you can listen from. But whether they work or not **depends on the backend you're using**, and some backends are not interactive.

Pick() Event

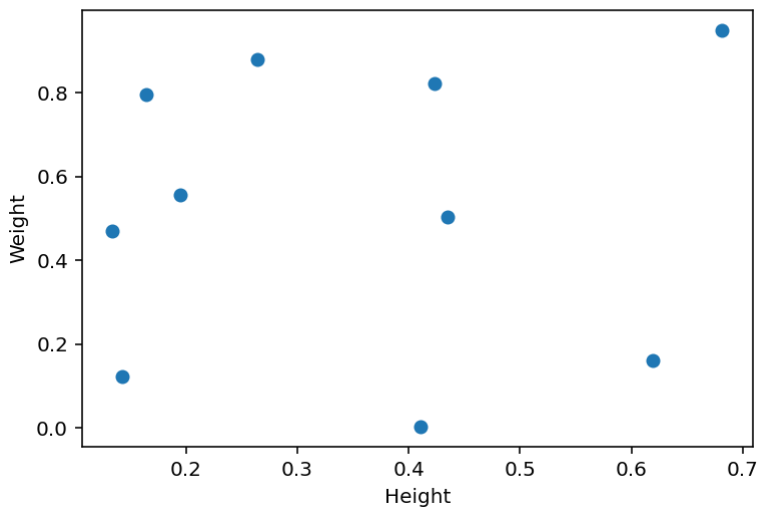
In [37]:
```
from random import shuffle
origins = ['China', 'Brazil', 'India', 'USA', 'Canada', 'UK', 'Germany', 'Ira

shuffle(origins)

df = pd.DataFrame({'height': np.random.rand(10),
                   'weight': np.random.rand(10),
                   'origin': origins})
df
```

Out[37]:

	height	weight	origin
0	0.619288	0.160845	Brazil
1	0.142874	0.122175	Chile
2	0.435068	0.504439	China
3	0.423570	0.821940	India
4	0.164174	0.795532	UK
5	0.681260	0.948988	Mexico
6	0.194701	0.556152	Canada
7	0.263972	0.878910	Germany
8	0.134198	0.470544	Iraq
9	0.410497	0.002762	USA

Plotting with an added argument `picker`

In [40]:
```
plt.figure()
# picker=5 means the mouse doesn't have to click directly on an event, but ca
plt.scatter(df['height'], df['weight'], picker=10)
plt.gca().set_ylabel('Weight')
plt.gca().set_xlabel('Height')
```



Out[40]: Text(0.5, 0, 'Height')

Wiring It Up

We'll create a function onpick(), which takes on an event. This event is a pick event which has different data than the `most` event. In particular, **it's got an index value** which happens to correspond to our index and the dataframe.

Matplotlib isn't aware of the dataframe, but it renders the data in the same order. So we can us the dataframe *iloc* indexer to pull out the origin information.

In [39]:
```
def onpick(event):
    origin = df.iloc[event.ind[0]]['origin']
    plt.gca().set_title('Selected item came from {}'.format(origin))

# tell mpl_connect we want to pass a 'pick_event' into onpick when the event
plt.gcf().canvas.mpl_connect('pick_event', onpick)
```

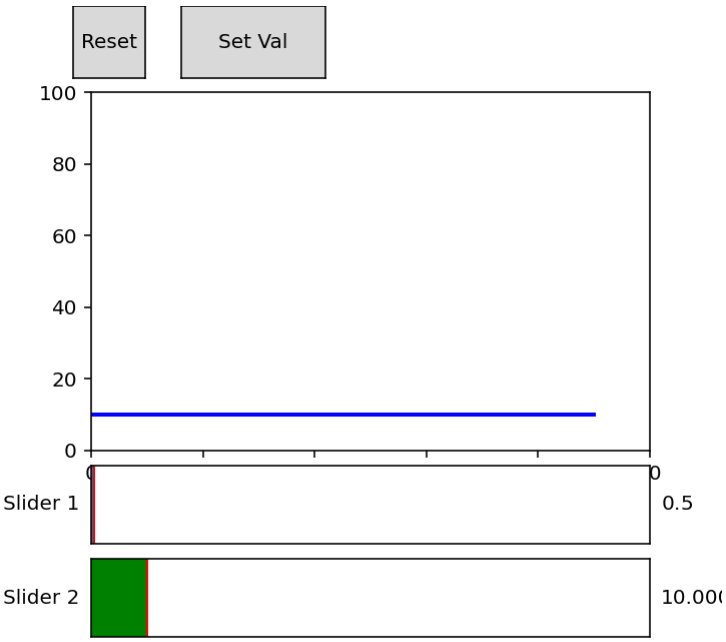
Out[39]: 9

03-07: Matplotlib Widgets (Supplementary)

Slider & Button Widget

In [98]:
```
%matplotlib notebook
import matplotlib.gridspec as gridspec
from matplotlib.widgets import Slider, Button

# Creating sample data
x = np.arange(0,10,1)
y = np.ones(10)*10
fig = plt.figure(figsize = (5,5))
gs = gridspec.GridSpec(6,2)
ax1 = plt.subplot(gs[0:4,:])
ax2 = plt.subplot(gs[4:5,:])
ax3 = plt.subplot(gs[5:6,:])
_ , = ax1.plot(x,y, linewidth=2, color = 'blue')
ax1.axis([0,10,0,100])

# Dummy Slider Example
slider1 = Slider(ax2, 'Slider 1', valmin = 0, valmax = 100)

# Slider that changes graph
slider2 = Slider(ax = ax3,
                 label = 'Slider 2',
                 valmin = 0,
                 valmax = 100,
                 valinit = 10, ## starting value when the graph loads
                 valfmt =  '%1.3f', ## Set value format to 3 decimal points
                 ## The closedmin and closedmax properties indicate whether
                 ## the slider is closed on either end.
                 #slidermax = slider1,
                 closedmax = False,
                 # each time the value changes it changes by 10.
                 # At this point, the slider has not changed the graph
                 color = 'green',
                 #alpha = 0.5
                )
# Function used for wiring up the event.
def val_update(val):
    print(val)
```

```
    yval = slider2.val
    _.set_ydata(np.ones(10)*yval)

cid = slider2.on_changed(val_update)

## Create a button (another widget)
axButton1 = plt.axes([0.1,0.9,0.1,0.1])
btn1 = Button(axButton1,'Reset')

axButton2 = plt.axes([0.25,0.9,0.2,0.1])
btn2 = Button(axButton2, 'Set Val')

def resetSliders(event):
    slider1.reset()
    slider2.reset()
btn1.on_clicked(resetSliders)

def setValue(val):
    slider2.set_val(50)
btn2.on_clicked(setValue)
# At this point, the slider has not changed the graph.
plt.show()
```



Creating the Slider Widget

Summary

And there we have it, a fairly straight forward example of adding interactivity to your graphs and plots. Now I'll be honest, this looks very little code to write and this lecture was pretty short. But there is **a lot of hunting through documentation and forms** that you have to do in order to understand the details of the events which are being passed around. Python's **lack of static typing** has an unfortunate side effect and it matches the documentation for features like events is buried or missing.

In [ ]:

In [ ]:

In [ ]: