Lecture 07: Matplotlib and object-oriented plotting

In lecture 6 we explored how classes are constructed. This forms the foundation for a lot of object-oriented programming. We explore this in more detail in tutorial 3.

There are multiple reasons for this. The most obvious reason is that it will benefit you greatly to be able to build more kinds of programmes for different uses.

The other major reason is it allows you to understand how a lot of other object-oriented programmes work. Almost all of Python operates in this way somehow.

Today we'll explore plotting with `Matplotlib`. This is a major package for making figures in Python. It can be confusing at first. A major reason for this is it's object-oriented architecture.

Before understanding plotting, it helps to understand a few simple things about how `jupyter` notebooks work.

Defining variables:

In [1]:

```
x = 1
y, z = 2, 3
```

With the assignment, nothing came up after the cell.

We can try the following:

In [2]:

```
x
y
z
```

Out[2]:

```
3
```

This time we got out `z`.

What about?

In [3]:

```
print(x)
y
z
```

```
1
```

Out[3]:

```
3
```

So now `x` is printed below the cell, and we *also* get `Out[#]: z`

Or

In [4]:

```
print(x)
y
print(z)
w=4
```

```
1
3
```

Or

In [5]:

```
print(x)
print(y)
print(z)
w
print(x+y)
```

```
1
2
3
3
```

Hopefully this illustrates an important aspect of the way `jupyter` works.

The `Out[number]` corresponding to an `In[number]` is only the last thing that isn't an assignment, nor a `print` statement.

This might seem trivial. But it allows us to diagnose other behaviours.

Let's try some **plotting**

In [6]:

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

Let's try something we've seen before. But in the past we've used the magic words `%matplotlib inline`. Let's leave these our for now. It won't work, but understanding why will give us hints about what's going on.

In [7]:

```
x = np.linspace(0,1,1000)
y = np.sin(np.pi*x)
plt.plot(x,y)
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x7f1f91f9b320>]
```

Notice there was an `Out[number]:  [<matplotlib.lines.Line2D at ...>]`. No plot. Remember `jupyter` returns the last thing that's not an assignment or a `print` statement.

We can test this.

In [8]:

```
x = np.linspace(0,1,1000)
y = np.sin(np.pi*x)
plt.plot(x,y)
print(x[0])
```

```
0.0
```

This time no `Out[number]:`. We also note that the words that come out of the plot look a lot like the stuff we get with `functions` and `classes` etc. For example:

In [9]:

```
np.sin
```

Out[9]:

```
<ufunc 'sin'>
```

It might have taken a while to get to the point. But now we know something important:

```
Plots in Python are objects. Just like everything else.
```

Therefore we need to think of differently. The `plot` is going somewhere we can't see it. To make it show up we have to ask for it.

This is the magic words that tells the plots to come to the screen as output:

In [10]:

```
%matplotlib inline
```

It's not needed if you are plotting your figure to a file. But it is if you want to see it first:

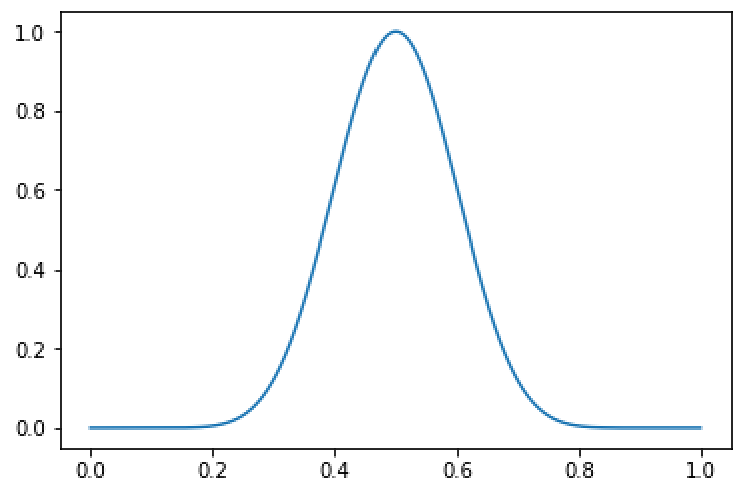
This might seem like a waste. But there are reasons. For example, let's enter another plot.

In [11]:

```
plt.plot(x,y**10)
```

Out[11]:

```
[<matplotlib.lines.Line2D at 0x7f1f91eae2e8>]
```



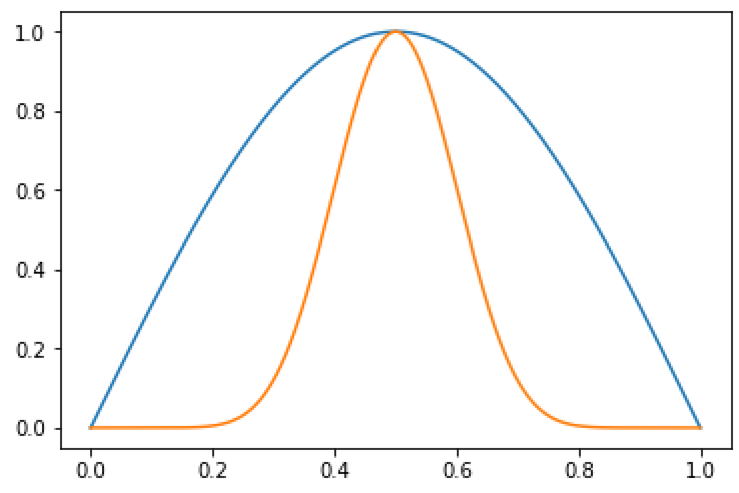
Or

In [12]:

```
plt.plot(x,y)
plt.plot(x,y**10)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x7f1f91ec9978>]
```

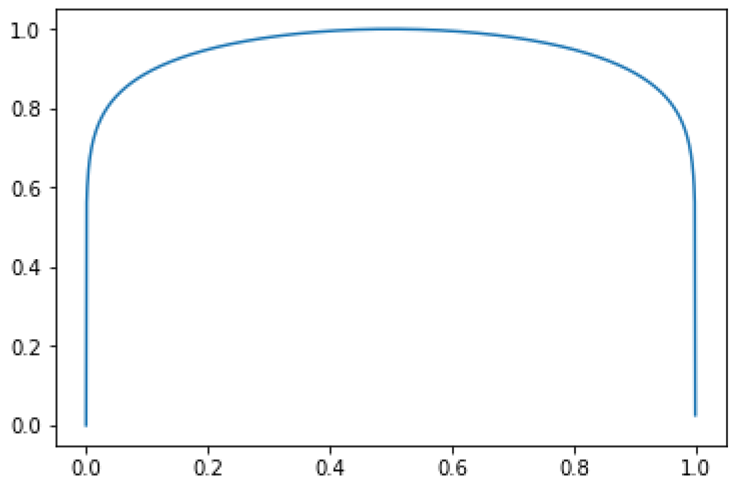


In [13]:

```
plt.plot(x,y**0.1)
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x7f1f91e1a3c8>]
```



Here's the summary so far:

Plots are objects. The `plt.plot` sends that object to some random buffer. We can even put multiple things in the buffer. Getting the object out of the buffer requires `plt.show()`, or `%matplotlib inline`. But doing this clears out buffer and we have to start over.

What else can we do?

Figures as objects

This makes a variable called `fig` that stores an object from the class `figure`.

In [14]:

```
fig = plt.figure()              # a new figure window
```

```
<matplotlib.figure.Figure at 0x7f1f91dfd400>
```

In [15]:

```
fig
```

Out[15]:

```
<matplotlib.figure.Figure at 0x7f1f91dfd400>
```

The advantage of this is that we can change the attributes of `fig` using methods of the class.

This creates a set of *axes* for `fig`. It's common to use the name `ax` for this.

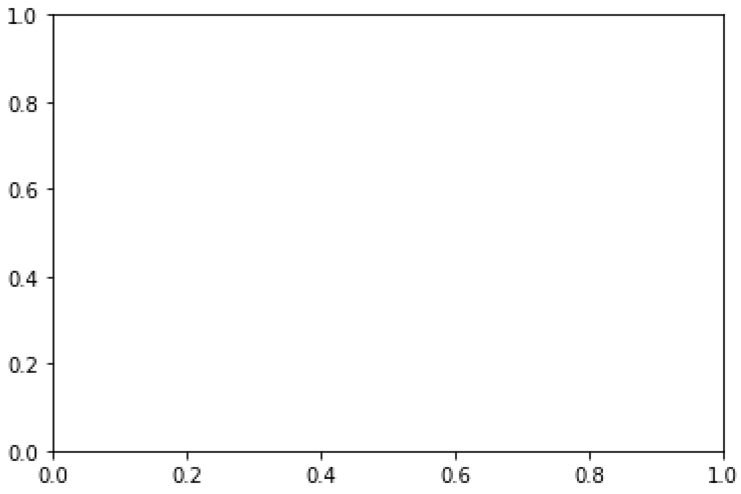
In [16]:

```
ax = fig.add_subplot(1, 1, 1)  # specify (nrows, ncols, axnum)
```

In [17]:

```
fig
```

Out[17]:



We can now put things on `ax`.

In [18]:

```
ax.plot(x,y,color='blue')
```

Out[18]:

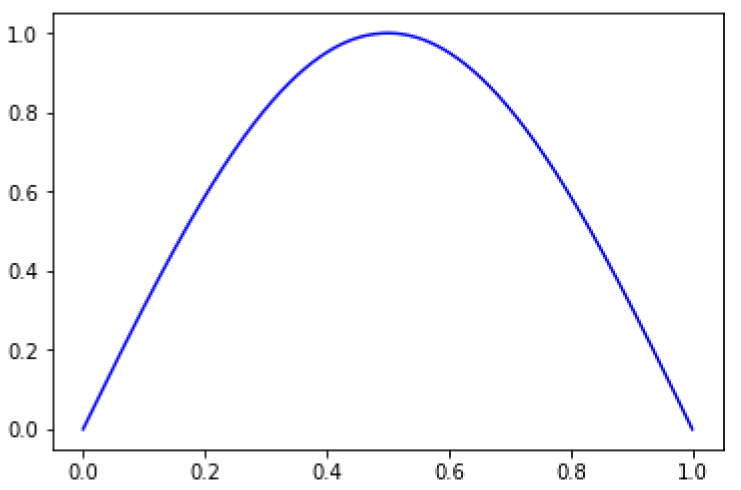
```
[<matplotlib.lines.Line2D at 0x7f1f91d796a0>]
```

Now `fig` is more interesting:

In [19]:

```
fig
```

Out[19]:



The great advantage is that we can keep adding things to `ax` and `fig`. We don't have to get it all right the first time.

In [20]:

```
ax.plot(x,y**10,color='red')
```

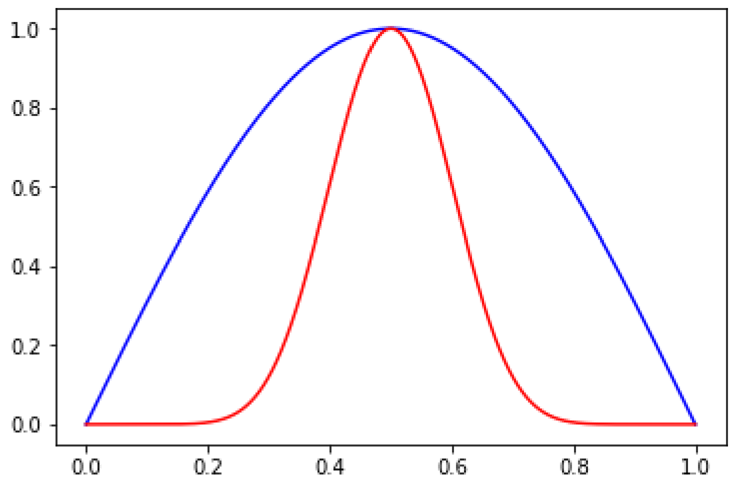
Out[20]:

```
[<matplotlib.lines.Line2D at 0x7f1f91d79550>]
```

In [21]:

```
fig
```

Out[21]:



In [22]:

```
g = ax.plot(x,y**0.1,color='green')
```

In [23]:

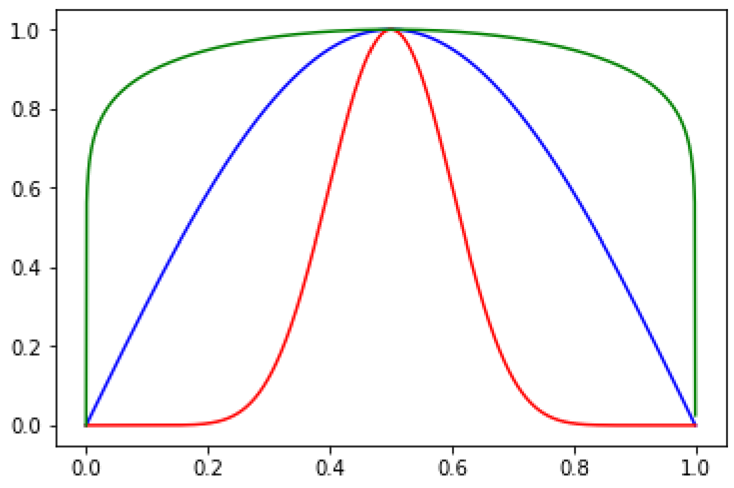
```
g.del()
```

```
  File "<ipython-input-23-3adecd66ad00>", line 1
    g.del()
        ^
SyntaxError: invalid syntax
```

In [24]:

```
fig
```

Out[24]:



Adding more lines is fun. But there are some obvious things we need to do. As you high school teacher said "`label` your ax!"

$$e^{i\pi} + 1 = 0$$

In [25]:

```
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_title("You can use Markdown this way: $\sin(\pi x)^{k}$")
```

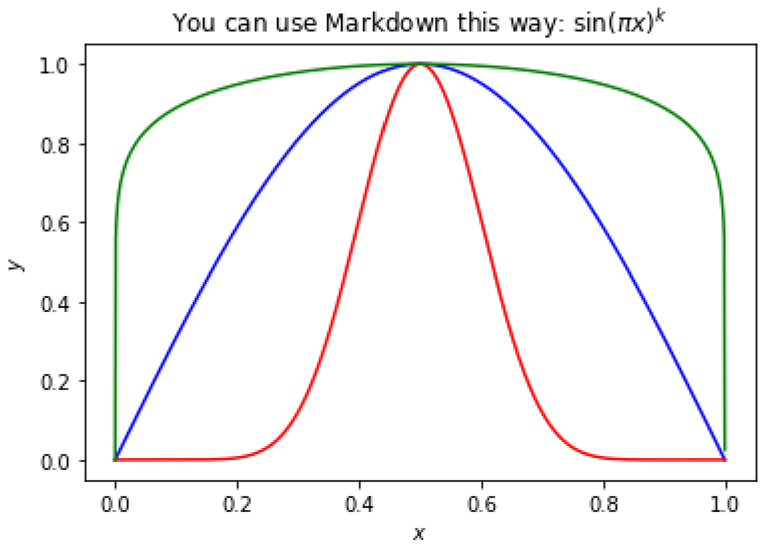
Out[25]:

```
Text(0.5,1,'You can use Markdown this way: $\\sin(\\pi x)^{k}$')
```

In [26]:

```
fig
```

Out[26]:

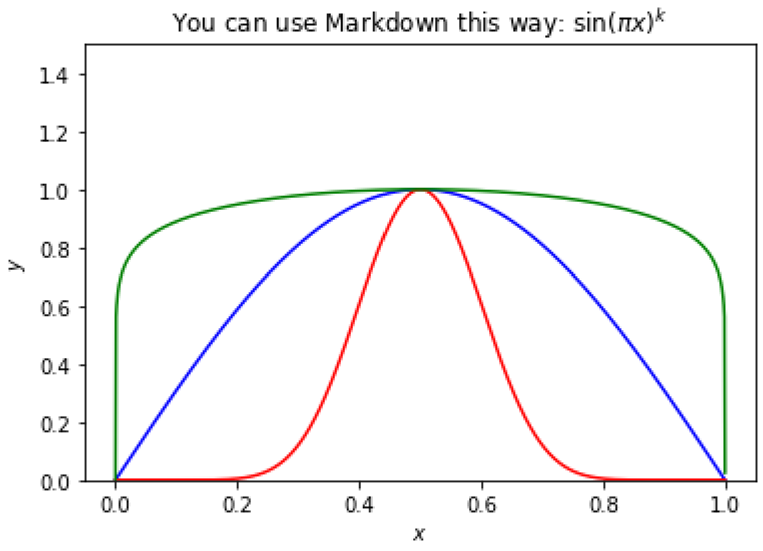


You can do more

In [27]:

```
ax.set_ylim(0, 1.5)
fig
```

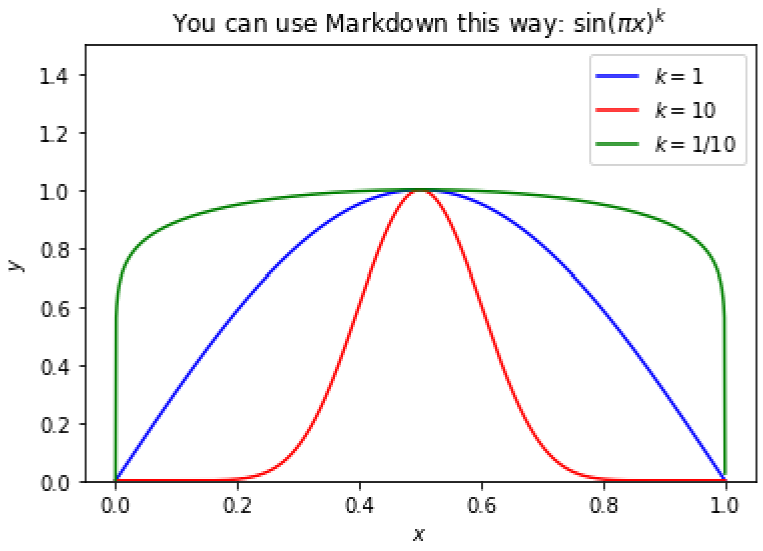
Out[27]:



In [28]:

```
ax.legend(['$k=1$', '$k=10$', '$k=1/10$'])
fig
```

Out[28]:

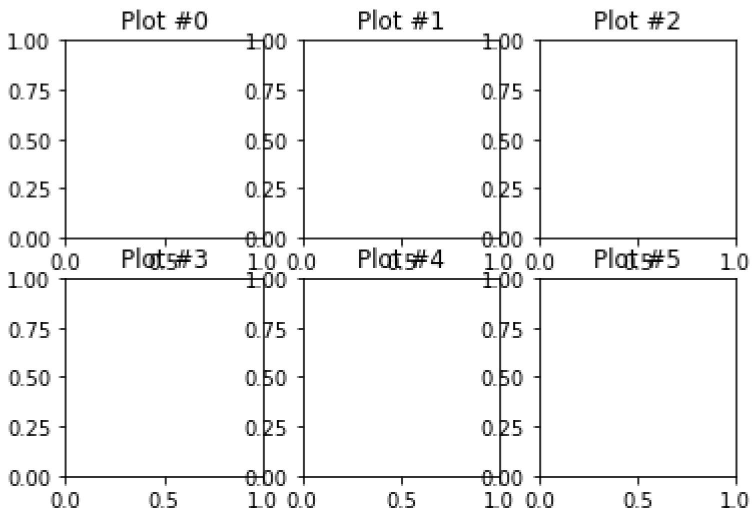


Plot arrays

This makes 2 rows and 3 columns of axes.

In [29]:

```
fig = plt.figure()
for i in range(6):
    ax = fig.add_subplot(2, 3, i + 1)
    ax.set_title("Plot #%i" % i)
```



Ugly! But:

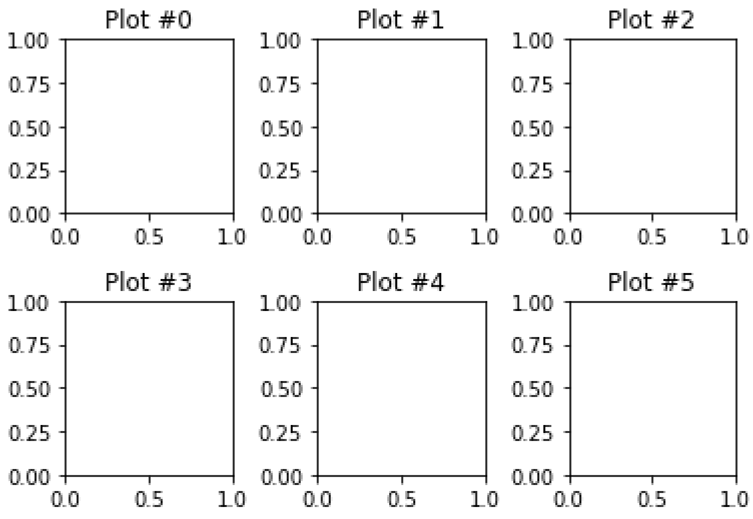
In [30]:

```
fig?
```

In [31]:

```
fig.subplots_adjust(wspace=0.5, hspace=0.5)
fig
```

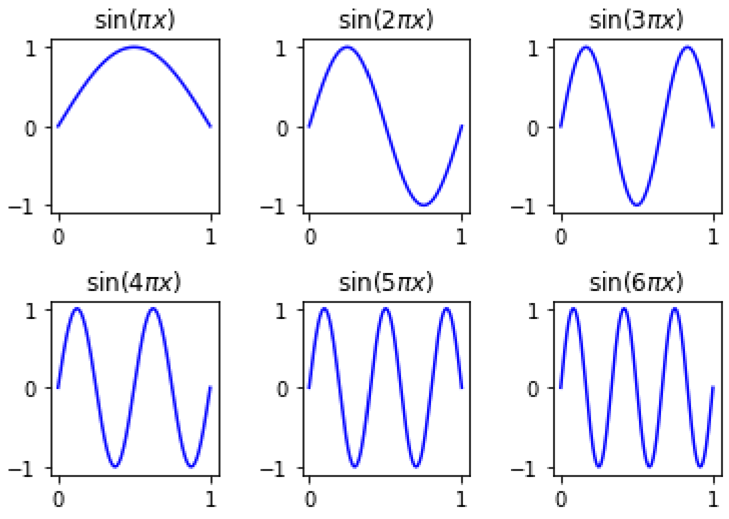
Out[31]:



In [32]:

```
x = np.linspace(0,1, 200)
for i in range(6):
    fig.axes[i].plot(x, np.sin((i+1) * np.pi*x),color='blue')
    fig.axes[i].set_ylim(-1.1, 1.1)

fig.axes[0].set_title("$\sin(\pi x)$")
for i in range(1,6):
    fig.axes[i].set_title("$\sin(%i \pi x)$" %(i+1))


fig
```

Out[32]:



Like most things in life, there is more than one way to do similar things. This might be a nicer way of plotting:

We can define the `figure` and `axes` at the same time.

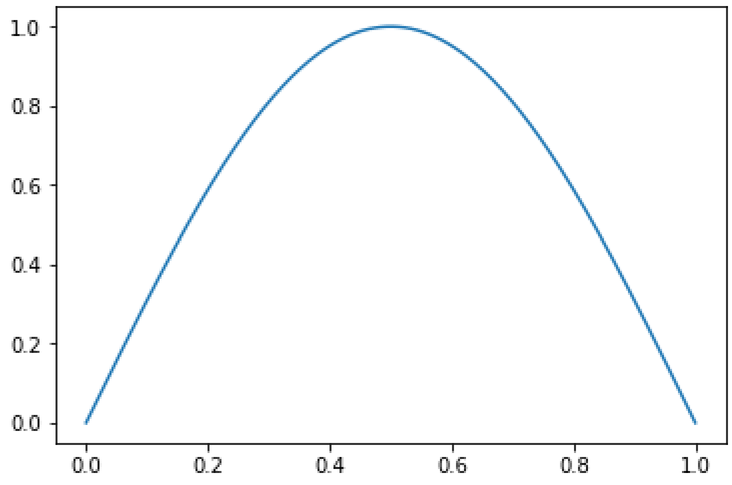
For a single plot:

In [33]:

```
fig, ax = plt.subplots()

ax.plot(x, np.sin(np.pi*x))
```

Out[33]:

```
[<matplotlib.lines.Line2D at 0x7f1f91ce5358>]
```

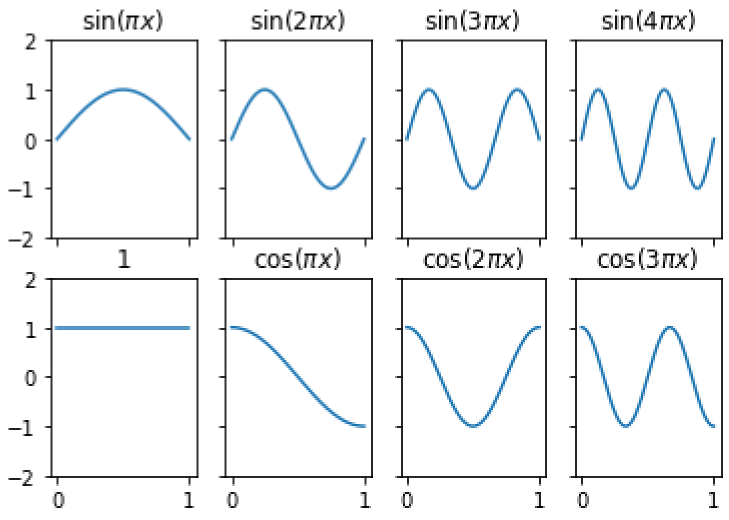


For a 2 by 4 grid:

This has some extra options. For example, we can take away the redundant labels.

In [34]:

```
fig, ax = plt.subplots(2, 4,sharex=True, sharey=True)

for j in range(4):
    ax[0,j].plot(x,np.sin((j+1)*np.pi*x))
    ax[1,j].plot(x,np.cos(j*np.pi*x))
    ax[0,j].set_ylim(-2, 2)
    ax[1,j].set_ylim(-2, 2)

ax[0,0].set_title("$\sin(\pi x)$")
ax[0,1].set_title("$\sin(2\pi x)$")
ax[1,0].set_title("$1$")
ax[1,1].set_title("$\cos(\pi x)$")
for j in range(2,4):
    ax[0,j].set_title("$\sin(%i \pi x)$" %(j+1))
    ax[1,j].set_title("$\cos(%i \pi x)$" %(j))
```



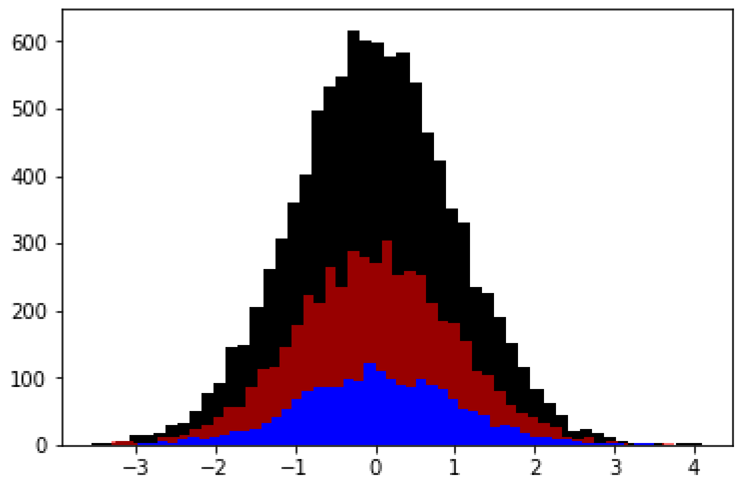
Lots of different kinds of plots:

HISTOGRAMS: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html)

Play around with the options. `alpha` changes the color opacity. You can see that `'red' + 'blue' = 'purple'`. This wouldn't be the case with other values of `alpha`.

In [35]:

```
fig, ax = plt.subplots()

x = np.random.normal(size=10000)
H = ax.hist(x, bins=50, alpha=1.0, histtype='stepfilled',color='black')

x = np.random.normal(size=5000)
H = ax.hist(x, bins=50, alpha=0.6, histtype='stepfilled',color='red')

x = np.random.normal(size=2000)
H = ax.hist(x, bins=50, alpha=1.0, histtype='stepfilled',color='blue')
```



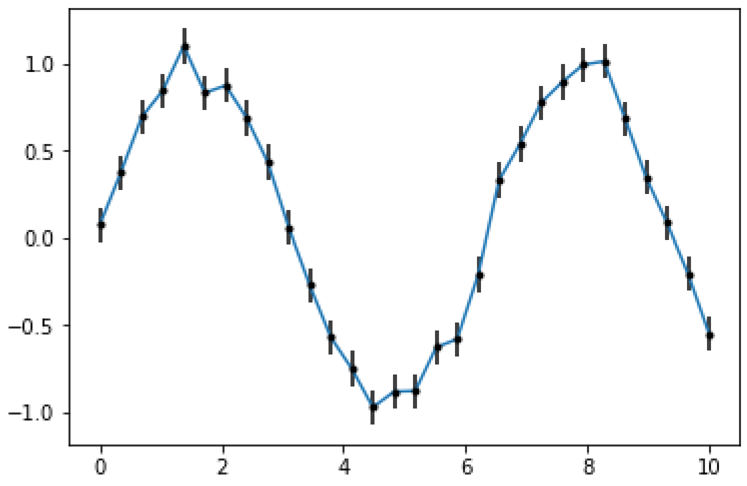
ERROR BARS https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html)

In [36]:

```
x = np.linspace(0, 10, 30)
dy = 0.1
y = np.random.normal(np.sin(x), dy)

fig, ax = plt.subplots()
ax.errorbar(x, y, dy, fmt='.k')
ax.plot(x, y)
```

Out[36]:

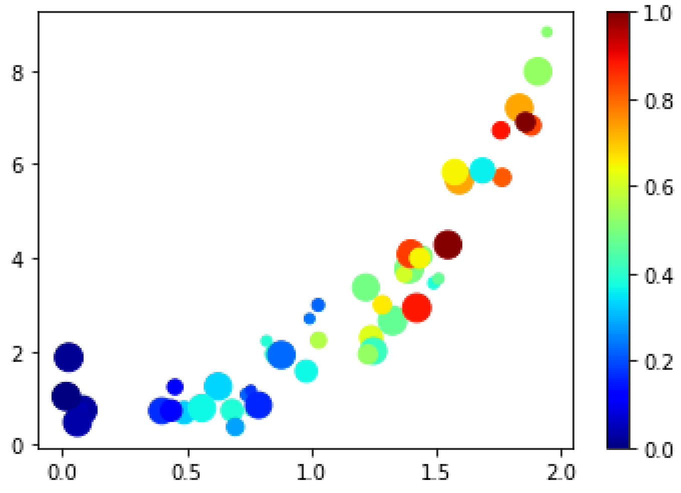
```
[<matplotlib.lines.Line2D at 0x7f1f917420f0>]
```



SCATTERPLOT https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html)

In [37]:

```
x = 2*np.random.random(50)
y = x**3+2*np.random.random(50)
c = 0.2*x+0.5*x*np.random.random(50)          # color of points
s = x+ 200*np.random.random(50)     # size of points

fig, ax = plt.subplots()
im = ax.scatter(x, y, c=c, s=s, cmap=plt.cm.jet)

# Add a colorbar
fig.colorbar(im, ax=ax)

# set the color limits - not necessary here, but good to know how.
im.set_clim(0.0, 1.0)
```



CONTOUR PLOTS

We can first see a useful trick for making 2D arrays from 1D arrays.

In [38]:

```
x = np.linspace(0, 10*np.pi, 100)
y = np.linspace(0, 10*np.pi, 100)

print(x.shape)
print(y.shape)

print(x[0:5])
print(y[0:5])
```

```
(100,)
(100,)
[0.         0.31733259 0.63466518 0.95199777 1.26933037]
[0.         0.31733259 0.63466518 0.95199777 1.26933037]
```

Make a small change:

In [39]:

```
x.shape = (100,1)
y.shape = (1,100)

print(x.shape)
print(y.shape)

print(x[0:5])
print(y[0:5])
```

```
(100, 1)
(1, 100)
[[0.        ]
 [0.31733259]
 [0.63466518]
 [0.95199777]
 [1.26933037]]
[[ 0.          0.31733259  0.63466518  0.95199777  1.26933037  1.586
66296
   1.90399555  2.22132814  2.53866073  2.85599332  3.17332591  3.490
6585
   3.8079911   4.12532369  4.44265628  4.75998887  5.07732146  5.394
65405
   5.71198664  6.02931923  6.34665183  6.66398442  6.98131701  7.298
6496
   7.61598219  7.93331478  8.25064737  8.56797996  8.88531256  9.202
64515
   9.51997774  9.83731033 10.15464292 10.47197551 10.7893081  11.106
64069
  11.42397329 11.74130588 12.05863847 12.37597106 12.69330365 13.010
63624
  13.32796883 13.64530142 13.96263402 14.27996661 14.5972992  14.914
63179
  15.23196438 15.54929697 15.86662956 16.18396215 16.50129475 16.818
62734
  17.13595993 17.45329252 17.77062511 18.0879577  18.40529029 18.722
62289
  19.03995548 19.35728807 19.67462066 19.99195325 20.30928584 20.626
61843
  20.94395102 21.26128362 21.57861621 21.8959488  22.21328139 22.530
61398
  22.84794657 23.16527916 23.48261175 23.79994435 24.11727694 24.434
60953
  24.75194212 25.06927471 25.3866073  25.70393989 26.02127248 26.338
60508
  26.65593767 26.97327026 27.29060285 27.60793544 27.92526803 28.242
60062
  28.55993321 28.87726581 29.1945984  29.51193099 29.82926358 30.146
59617
  30.46392876 30.78126135 31.09859394 31.41592654]]
```

Now we can combine them like they are 2D arrays

In [40]:

```
print((x+y).shape)
print((x+y)[0:5,0:5])
```

```
(100, 100)
[[0.         0.31733259 0.63466518 0.95199777 1.26933037]
 [0.31733259 0.63466518 0.95199777 1.26933037 1.58666296]
 [0.63466518 0.95199777 1.26933037 1.58666296 1.90399555]
 [0.95199777 1.26933037 1.58666296 1.90399555 2.22132814]
 [1.26933037 1.58666296 1.90399555 2.22132814 2.53866073]]
```

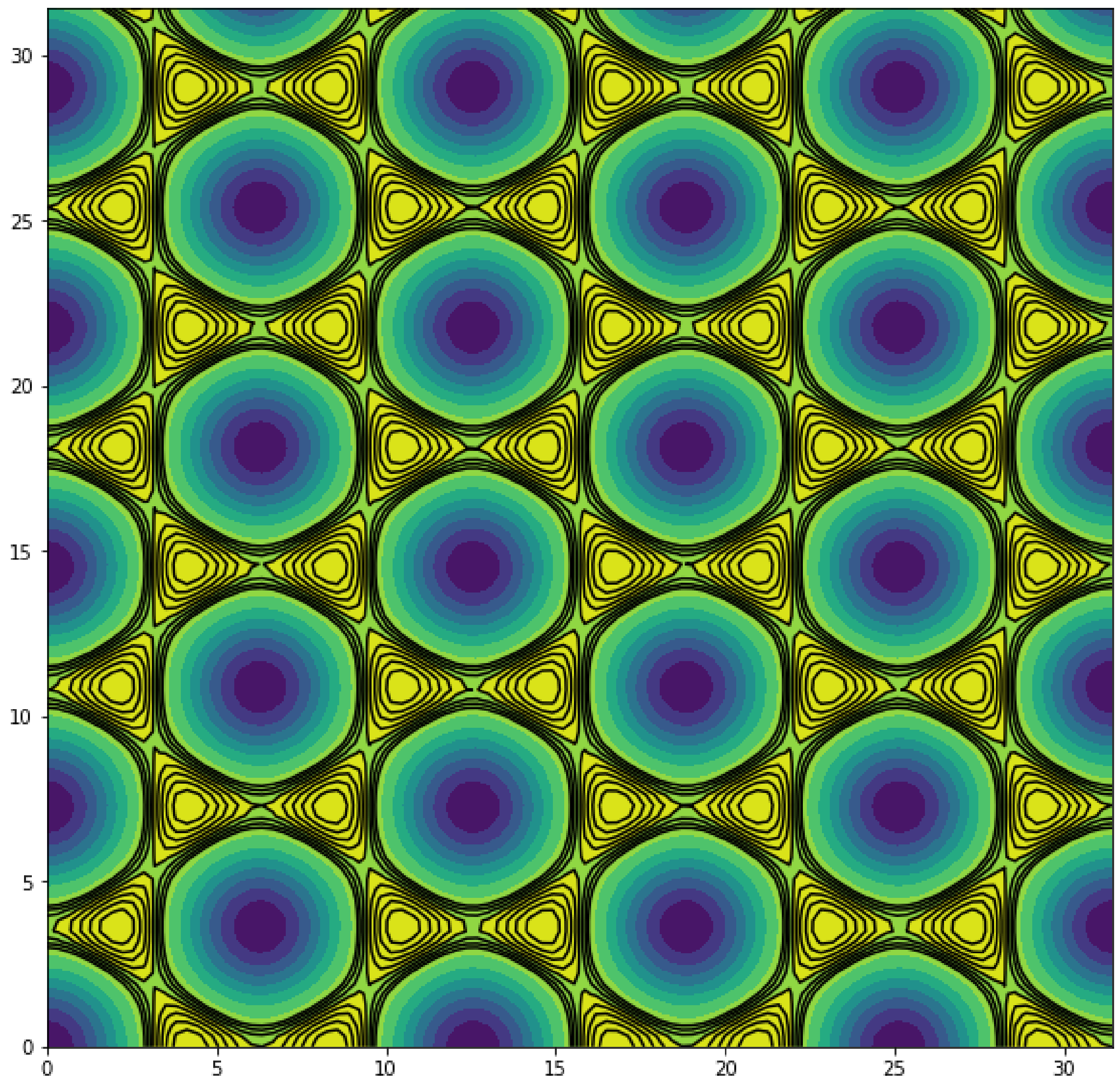
In [41]:

```
z = (np.cos(x) + 2*np.cos(x/2)*np.cos(np.sqrt(3)*y/2))

#z = np.cos(np.sqrt(3)*y+x)

x, y = x+0*y, y+0*x


print(x.shape)
print(y.shape)
print(z.shape)
```

```
(100, 100)
(100, 100)
(100, 100)
```

In [42]:

```
fig, ax = plt.subplots(figsize=(10, 10))

# contour lines
im0 = ax.contour(x, y,-z,(0.7,0.8,0.9,1.0,1.1,1.2,1.3,1.4),colors='k')

# filled contours
im1 = ax.contourf(x, y, -z, 10 )

#fig.colorbar(im1, ax=ax)
#ax.set_aspect('equal')
```

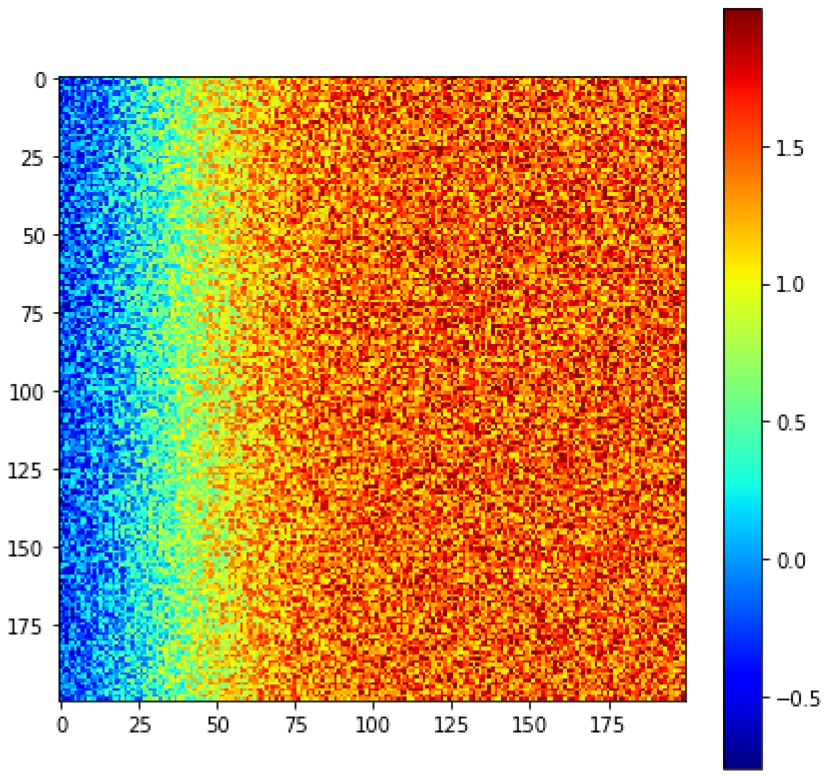


2D COLOR PLOTS

Here is an example of `imshow` https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html)

In [43]:

```
I = np.random.random((200, 200))

I += np.tanh(np.linspace(-1, 5, 200))

fig, ax = plt.subplots(figsize=(7, 7))

im = ax.imshow(I, cmap=plt.cm.jet)

fig.colorbar(im, ax=ax)
ax.set_aspect('equal')
```



Here is another example using `pcolormesh`, which needs a grid of x and y values. [https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.pcolormesh.html (https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.pcolormesh.html)](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.pcolormesh.html)

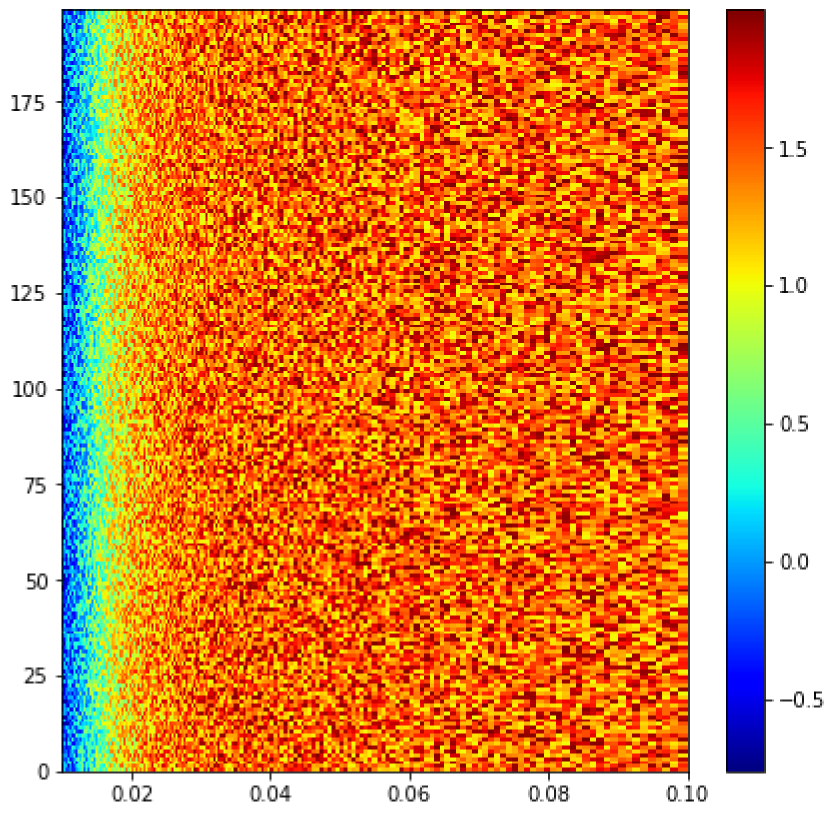
The good thing about it is that it can work for data on a non-uniform grid.

In [44]:

```
I = np.random.random((200, 200))
x = np.logspace(-2,-1,200)
y = np.arange(200)

I += np.tanh(np.linspace(-1, 5, 200))

fig, ax = plt.subplots(figsize=(7, 7))

im = ax.pcolormesh(x,y,I, cmap=plt.cm.jet)

fig.colorbar(im, ax=ax)
# ax.set_aspect('equal')
```

Out[44]:

```
<matplotlib.colorbar.Colorbar at 0x7f1f91da8160>
```



`matplotlib` is a very useful library, but is sometimes annoying.

The good news is that because a lot of people use it, a lot of people get annoyed. And they use the internet. So stack exchange is a great resource for getting `matplotlib` to do what you want.

Happy plotting!