more matplotlib, numpy matrices

go over previous matplotlib exercises and homework3

more matplotlib

- MATLAB style vs. OOP style:
  - we've learned the MATLAB "procedural" style of plotting:

```
import matplotlib.pyplot as plt
t = np.linspace(0, 4*np.pi, 100) # 100 evenly spaced timepoints, 2 cycles
s = np.sin(t) # calculate sine as a function of t
c = np.cos(t) # calculate cosine as a function of t
plt.plot(t, s) # plot points in t on x-axis vs. points in s on y-axis
plt.plot(t, c) # plot cosine as well
```

  - MPL also has an alternative, more Pythonic, object-oriented programming (OOP) style, with very similar commands
  - first, you explicitly create a figure and an axes
  - `f, ax = plt.subplots()` - by default creates a new figure with one set of x-y axes, and returns objects representing them
    - notice the `s` in `plt.subplots()`, `plt.subplot()` is a slightly different MATLAB-style procedural command which you shouldn't need to use
  - now, we can do most of our plot commands as methods of this particular axes `ax`:
    - `ax.plot(t, s)`
    - common formatting commands in OOP style:
      - `ax.set_xlim()`, `ax.set_ylim()`, `ax.set_xlabel()`, `ax.set_ylabel()`, `ax.set_title()`, `ax.legend()`
    - compare with MATLAB style:
      - `plt.xlim()`, `plt.ylim()`, `plt.xlabel()`, `plt.ylabel`, `plt.title()`, `plt.legend()`
    - OOP style is slightly more wordy, but much more explicit, gives better control over multiple figures
    - `spines` are more easily accessible through the OOP interface:
      - `ax.spines['top'].set_visible(False)`
      - `ax.spines['right'].set_visible(False)`
  - with multiple figures and axes open, we can refer to them directly by name, no longer have to worry about which is the "current" figure:
  - `f2, ax2 = plt.subplots()`
  - `ax2.hist(s)` - plot a histogram of sin(t) this time
  - to clear a particular axes: `ax.clear()`
  - if you created your figures/axes with the MATLAB style commands:
    - use `f = plt.gcf()` to get a reference to the current figure
    - use `ax = plt.gca()` to get a reference to the current axes
- subplots: create multiple axes in a single figure
  - `f, axs = plt.subplots(nrows=2, ncols=2)`
  - `axs` is now a 2D array, choose your axes by indexing into `axs` with row and col indices:
    - `axs[0, 1].plot(t, s) # plot s vs. t in axes in 1st row 2nd column`
    - `axs[1, 0].plot(t, c, color='r') # plot c vs. t in red in axes in 2nd row 1st column`
  - optional kwargs `sharex`, `sharey`

```
plt.close('all')
f1, ax1 = plt.subplots(2, 1, sharex=True, sharey=False) # ax1 is 1D array
ax1[0].plot(t, s) # plot s vs. t
ax1[1].plot(t, c, color='r') # plot c vs. t in red, shared x axis with sin plot
f2, ax2 = plt.subplots(2, 1, sharex=True, sharey=False) # ax2 is 1D array
ax2[0].hist(s) # plot hist of s
ax2[1].hist(c, color='r') # plot hist of c in red, shared x axis with sin hist
```

  - change the name of a figure, i.e. its window title bar and (maybe) its default filename in the save dialog box:

```
f1.canvas.set_window_title('time series')
f2.canvas.set_window_title('histograms')
```

- some other kinds of plots:
  - scatterplots
    - `x, y, c = np.random.random(100), np.random.random(100), np.random.random(100)`
    - `ax.scatter(x, y, c=c)` - very similar to `ax.plot()`
      - allows each point to be formatted differently (colour, marker, size)
      - defaults to not drawing a line between points
  - errorbar plot
    - `ax.errorbar(x, y, yerr=0.1, xerr=0.2)` - again similar to `a.plot()`, but with errorbars, each point can have a different sized pair of error bars
  - bar charts
    - `x, y = [1, 2, 3], [4, 5, 6]`
    - `ax.bar(x, height=y)` - vertical bars
    - `ax.barh(y, width=x)` - horizontal bars

matrices

- so far, we've (mostly) only dealt with 1D arrays, a.k.a. vectors
- numpy allows for N dimensional arrays, but most common are 2D arrays, a.k.a. matrices
- matrix is like an image: each entry has a (pixel) value stored at a row and column index
  - can also think of it as a nested list, i.e. a list of lists
- plotting matrices as images is a great way to visualize them
- initializing a 2D array is very similar to 1D arrays
  - explicitly, using a list of lists, or a tuple of tuples, convert to array:
  - `a = np.array([[1, 2, 3], [4, 5, 6]])` or `a = np.array(((1, 2, 3), (4, 5, 6)))`
  - using `np.arange()`, and then reshaping:
  - `a = np.arange(16).reshape((8, 2))`
    - creates a 1D array, but then reshapes it to 2D
    - 2D array shape tuples are always `(nrows, ncolumns)`
    - check the shape of an array: `a.shape`
    - `nrows * ncols` of the reshaped array have to equal the number of elements in the 1D array
    - get the total number of elements in an array, whether 1D or 2D or ND: `a.size`
    - what happens if you do `a.reshape((8, 3))`?
    - what happens if you do `a.reshape((4, 4))`?
    - can also change the shape of an existing array by assigning to `a.shape = 8, 2`
  - `a = np.zeros((8, 2))`
  - `a = np.ones((8, 2))`
  - `a = np.random.random((8, 2))`
  - `a = np.tile([1, 2], (8, 1))` - tile the pattern `[1, 2]` in 8 rows and 1 column
  - `a.fill(7)` fills the array with the number 7, but maintains its shape
  - `np.eye(5)` - create 5x5 identity matrix
- to get number of rows: `a.shape[0]`
- to get number of columns: `a.shape[1]`
  - for 1D arrays, `len()` gave number of elements in the array
  - for 2D arrays, `len()` is a shortcut for `a.shape[0]`, i.e. number of **rows** in `a`
- to convert 2D array to a 1D array, flatten it using `a.ravel()`
  - if `a = np.arange(16).reshape((8, 2))`, then `a.ravel()` gives the same as `np.arange(16)`
  - flattening and asking for the length of the result: `len(a.ravel())` gives same as `a.size`
- visualizing matrices:

```
f, ax = plt.subplots(figsize=(3, 3)) # set figure size in inches
im = ax.imshow(a) # returns an "image" object
f.canvas.set_window_title('imshow')
```

```
f.colorbar(im) # add a colorbar legend for the image
f.set_tight_layout(True) # make figure automatically resize contents
# resize the figure with the mouse
ax.set_xticks([]) # disable x ticks
ax.set_yticks([]) # disable y ticks
```

  - use different colormaps to change how values in an array map to colours in displayed image:
    - default is called `"viridis"`
    - another popular one is `"jet"`
    - all colormaps listed using `plt.colormaps()`
    - `im = ax.imshow(a, cmap='jet')` - set during `imshow` call, doesn't update exisiting colorbar (if any)
    - `im.set_cmap('viridis')` - modify existing image object, update colorbar
  - images covered in class 10
  - `scipy.ndimage` and `skimage` are great for all kinds of image manipulation
    - loading different image types as arrays
    - change contrast of an image
    - manipulate colours
    - thresholding, masking an image
    - image denoising/smoothing
    - image segmentation
    - see recent local Python talk by Joe Donovan for lots of examples:
      - https://github.com/superpythontalks/image_analysis/blob/master/image%20processing.ipynb
- 2D indexing and slicing
  - very similar to 1D arrays, but now we require two indices, not just one
  - `a = np.arange(16).reshape((8, 2))`
  - get element in 1st row, 1st column: `a[0, 0]`
  - get element in 3rd row, 2nd column: `a[2, 1]`
  - get element in last row, last column: `a[-1, -1]`
  - get element in 3rd row, 3rd column: `a[2, 2]` - IndexError! there is no 3rd column
  - get the first row, across all the columns: `a[0, :]`, or just `a[0]` for short
  - get the first column, across all the rows: `a[:, 0]`
  - get first 3 rows: `a[:3]`
  - get every other row: `a[::2]`
  - get first 2 columns: `a[:, :2]` - there are only 2 column anway, so this just returns `a`
  - flip matrix vertically by reversing order of rows: `a[::-1]`
  - flip matrix horizontally by reversing order of columns: `a[:, ::-1]`
  - rotate matrix in steps of 90 deg:
    - `np.rot90(a)` - counterclockwise
    - `np.rot90(a, -1)` - clockwise
    - `np.rot90(a, -2)` - 180 degrees clockwise
- arithmetic operations on 2D arrays
  - matrix & scalar, same as for 1D arrays, works elementwise
    - `a + 2`, `a - 2`, `a * 2`, etc. does what you'd expect
  - matrix & matrix also works elementwise, but both have to be the same shape:
    - `b = np.random.random(16).reshape((8, 2))`
    - `a + b`, `a - b`, `a * b`, `a / b` etc.
    - what happens if you try `b / a`? divide by zero warning, results in `np.inf` at [0, 0]
  - matrix `a` & 1D vector `x`, bit more complex:
    - still elementwise, but the last dimension of `a` has to have same length as `x`
    - `x = np.arange(8)`
    - `a * x` doesn't work, `a.T * x` does
    - called array "broadcasting"
  - can use many of the same methods as on 1D arrays:
    - `a.max()`, `a.min()`, `a.sum()`, `a.mean()`, `a.std()`, etc.

        - by default, these work on all elements in a 2D array, and return a single value
        - can be made to work across rows only, or columns only, by specifying the `axis` kwarg
        - `a.max(axis=0)` finds the max across the 1st dimension, i.e. across all rows, and returns one result per column
        - `a.max(axis=1)` finds the max across the 2nd dimension, i.e. across all colmuns, and returns one result per row
        - both of the above reduce dimensionality by 1, i.e. from 2D down to 1D

- matrix operations:
    - `a.transpose()` or its shortcut property `a.T` returns the transpose, i.e. swaps rows and columns
    - `a.diagonal()` returns the diagonal
    - `a.trace()` returns sum along the diagonal
    - inner product, results in new matrix, whose entry at (i, j) is sum of elementwise product of row i of `a` and column j of `b`
        - `np.dot(a, b)` raises error, ncols of `a` must equal nrows of `b`
        - `np.dot(a, b.T)` works, and so does `np.dot(a.T, b)`, but give different results
        - new matrix multiplication operator in Python 3.5 `@` does the same as `np.dot()`
    - outer product: take two vectors `x` and `y`, resulting matrix has (i, j)th entry that is `x[i] * y[j]`
        - `x = np.arange(10)`
        - `y = x.copy()`
        - `np.outer(x, y)`
        - builds a multiplication table!

- concatenating arrays in 2D:
    - `np.concatenate()` also has an `axis` kwarg which denotes which axis you want to lengthen
        - compare `np.concatenate([a, b], axis=0)` with `np.concatenate([a, b], axis=1)`
    - `np.hstack([a, b])` and `np.vstack([a, b])` are shortcuts for the above

- 3D arrays:
    - `np.stack([a, b])` gives a 3D array!
        - the two arrays have been stacked one on top of the other, like movie frames
    - can think of 3D arrays as movies, i.e. a sequence of images

```
movie = np.random.random(80).reshape((5, 4, 4)) # 5 frames, each 4 x 4 pixels
for framei, image in enumerate(movie):
    f, ax = plt.subplots()
    ax.imshow(image, cmap='jet')
    f.canvas.set_window_title('frame %d' % framei)
```

MPL and matrices exercises

1. Load in an array of data from the (binary) file `surprise.npy`. How many dimensions does it have? What's its shape?
2. If you print out the array, you have no chance of guessing what it might be. Numpy truncates big arrays when printing. But you can subsample the array. Pick every 80th row and column and print that instead. Any guesses? It's an image...
3. Plot the array. Are those the true colors? Does the array have color information? Play around a bit by giving it different colormaps: `'gray', 'jet', 'spring', 'cool'`. See https://matplotlib.org/examples/color/colormaps_reference.html. The default is `'viridis'`
4. Add a colorbar to the above figure.
5. Create a new figure with four axes. Choose how you want them to be arranged (2x2, 1x4, 4x1). Plot the array in each of the axes, but rotated 90 degrees with respect to the previous axes.
6. What do you predict the transpose of the array will look like? Plot the transpose in a new figure.
7. Create a new array that is a 2 x 2 version of the original, i.e. two copies in both x and y. Plot it in yet another figure.