

more matplotlib, numpy matrices

more matplotlib

- plotting issues:

```
import matplotlib.pyplot as plt
plt.plot() # should pop up a figure window with axes
```

- figures not popping up in **ipython**?
 - turn on interactive mode by calling `plt.ion()`
 - permanently enable interactive mode in matplotlib settings file:
 - linux: `~/.config/matplotlib/matplotlibrc`
 - mac + windows: `~/.matplotlib/matplotlibrc`
 - uncomment `#interactive: False` line and set to `True` instead
 - figures in **jupyter** not automatically displaying inline?
 - type `%matplotlib inline` in a cell, all cells that follow will do inline plots
 - make this setting permanent in `~/.ipython/ipython_config.py` file
 - uncomment `#c.InteractiveShellApp.matplotlib = None` line and set to `'inline'`
 - for interactive plotting in jupyter, type `%matplotlib notebook`
 - make this setting permanent with `c.InteractiveShellApp.matplotlib = 'notebook'` in `ipython_config.py` file
 - NOTE: this only works in more recent versions of matplotlib/jupyter?
 - quite a bit slower than interactive plots in ipython
 - missing toolbar?
 - set toolbar : `toolbar2` in `matplotlibrc` file
 - the button for "edit axes/curve/image params" in figure window might be missing, not sure why
- MATLAB style vs. OOP style:

- last week we 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 (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
 - one more useful figure property worth formatting is `spines`, only 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()`

- 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 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:
 - `ax.scatter(x, y)` - 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=5, xerr=2)` - again similar to `a.plot()`, but with errorbars
- bar charts
 - `ax.bar(left, height)` - vertical bars, left and height are sequences
 - `ax.bar(bottom, width)` - 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, ncols)
 - 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)`

- `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()`
 - `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
 - `im.set_cmap('viridis')` - modify existing image object
- `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
 - 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!
 - 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]`
 - 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
- 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` & 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 columns, and returns one result per row
- matrix operations:
 - `a.transpose()` or its shortcut property `a.T` - swaps rows with 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.stack()`, `np.hstack()`, `np.vstack()`
 - 3D arrays:
 - can think of them 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)

```