numpy file input/output, matplotlib

go over solutions to homework 3

numpy file input/output

- so far we've been generating values in code to fill arrays
- more often you load data from disk, and save results (and figures) back to disk
- two broad types of files: **text** and **binary**
  - **text files** are familiar, easy to view in a plain text editor, just a bunch of printable characters
    - what's a printable char? basically any available on your keyboard
    - like any other data, these chars are stored in bytes in memory and on disk
    - computers have to agree on which byte values represent which chars
    - an "encoding" is used to map byte values to characters
    - standard encoding is ASCII: American Standard Code for Information Interchange
    - ASCII uses 1 byte per character, but only uses the first 128 integer values (0 to 127) to represent printable characters, plus outdated "characters" that used to control direct output to printers and communications with old modems
    - see `ASCII-Conversion-Chart.pdf`
    - a newer increasingly common encoding is UTF-8, an extension of ASCII that can encode many more characters from more languages
    - in a text file, if you want to save the number `100`, you need to save 3 characters to disk (one `1`, two `0`s), so this takes up 3 bytes of space.
      - in reality, file sizes on disk are usually rounded up to the nearest multiple of 4 KB
    - what's the smallest integer data type that can represent `100`? How many bytes does it take up?
  - **binary files** are much more space efficient for storing numbers, faster to load/save, but require a hexadecimal ("hex") editor to view them directly
    - trying to open a binary file with a plain text editor will either show a bunch of nonesense text, or it will refuse to open it at all
    - open-source hex editors:
      - windows, mac and linux: wxHexEditor
      - windows: HexEdit, WinHex
      - mac: Hex Fiend
      - linux: ghex
    - mostly you load/save binary files programmatically anyway, no need to directly edit them
  - which file type to use depends on your data source, and your data size
  - for large numeric data sets, like imaging or electrophysiology, binary files are critical, text files aren't appropriate
  - text files are really just a subset of binary files
- **text files**: loading/saving arrays
  - this involves loading the entire text file into a big string, splitting the string up into lots of substrings (based on separators and line endings), then converting each substring into a

numeric value

- `np.loadtxt(fname)` - load from a text file, interpret as an array
  - use the `delimiter=','` keyword argument (kwarg) to handle e.g. comma separated values, see `test_1D.csv`
  - `test1D = np.loadtxt('test1D.csv', delimiter=',')` - interprets text file as comma separated values
  - dtype defaults to float64 for safety
  - use the `dtype` kwarg to force some other dtype, e.g. `dtype=int` or `dtype=np.int64`
    - `test1D = np.loadtxt('test1D.csv', delimiter=',', dtype=int)`
  - `test2D = np.loadtxt('test2D.csv', delimiter=',', dtype=int)` - 2D array!
- `np.savetxt(fname, a)` - save array `a` to a text file named `fname`
  - use the `delimiter=','` kwarg to create comma separated values between columns
  - `np.savetxt('test1D_out.txt', test1D)` - yucky formatting
  - `np.savetxt('test1D_out.txt', test1D, fmt='%d')` - better, but the row became a column
  - `np.savetxt('test1D_out.txt', (test1D,), fmt='%d')` - provide 2D-like object (1D array inside a tuple) - almost right, but missing commas
  - `np.savetxt('test1D_out.txt', (test1D,), fmt='%d', delimiter=',')` - perfect!
  - `np.savetxt('test2D_out.txt', test2D)` - yucky formatting
  - `np.savetxt('test2D_out.txt', test2D, fmt='%d', delimiter=',')` - perfect round-trip
  - file name extension is irrelevant, but is useful to humans to indicate contents
  - int vs float dtype information can be lost using the above, `fmt=%g'` kwarg helps
  - binary files handle dtype better...

- **binary files**: loading/saving arrays
  - `np.load(fname)` - from a binary numpy `.npy` file
    - `V = np.load('V.npy')` - load some fake voltage data
    - `t = np.load('t.npy')` - load corresponding timepoints
    - also loads `.npz` files, which are just `.zip` files containing multiple `.npy` files
  - `np.save('V2.npy', 2*V)` - to a binary `.npy` file named `V2.npy`
  - `np.savez()` & `np.savez_compressed()` - save multiple arrays to an uncompressed or compressed `.npz` file (just a `.zip` file with `.npy` files inside)

```
np.savez('Vt.npz', V=V, t=t) # save both arrays to .npz, pass as keyword argumen
d = np.load('Vt.npz') # load from .npz, returns something like a dictionary
list(d) # get list of keys in t he returned dictionary
V, t = d['V'], d['t'] # index into dictionary using the keys to get the values
```

  - reading and writing binary MATLAB `.mat` files
    - `scipy.io.loadmat()` and `scipy.io.savemat()` functions in the scipy package
    - read and writes using a dictionary, where each key:value pair is the variable_name:variable_value. Variable values are typically arrays

```
import scipy.io
d = scipy.io.loadmat('Vt.mat', squeeze_me=True)
V, t = d['V'], d['t'] # extract voltage and time from dict
```

```
d2 = {}
d2['V'] = 2*V # modify voltage
d2['t'] = t
scipy.io.savemat('Vt2.mat', d2) # save new voltage data to new file
```

- MATLAB likes to treat everything as a 2D array, even when it's only 1D. The `squeeze_me=True` "squeezes out" redundant dimensions that are of length 1. Without it, even if you stored a 1D array in the .mat file, you'd still get a 2D array out when reading it

- for loading/saving from/to **any** binary file (not just `.npy` or `.mat`):
  - `a = np.fromfile(fname)`
  - `a.tofile(fname)`
  - watch out: when loading you'll have to interpret the dtype and shape yourself, not ideal

numpy file input/output exercises:

1. Create an array `a = np.arange(10)`, and save it to a text file named `exercise.txt` using `np.savetxt()`. What folder will it be saved to? How can you check in IPython?
2. Open the file in your text editor. Does it look the way you'd expect? Resave it using the `fmt` kwarg (keyword argument), e.g. try `fmt='%g'` to format it as a general number that should look OK for both int and floats.
3. In your text editor, change the `9` to a `99`. Save it.
4. Create an array named `b` by loading in the data from `exercise.txt` using `np.loadtxt()`. Compare the contents of arrays `a` and `b`. Is `b` an integer array as you'd expect? Find a way to force `np.loadtxt()` to give you an integer array.
5. Save `a` again, but this time to a binary file named `exercise.npy` using `np.save()`
6. Create an array named `c` by loading in the data from `exercise.npy` using `np.load()`. Compare the contents of arrays `a` and `c`

plotting with matplotlib (MPL)

- main plotting library for Python, others exist, but often based on MPL
- can plot just about anything, see MPL sample plots
- typical usage: `import matplotlib.pyplot as plt`
  - now all the common plotting functions are available as `plt.something`
- line plots:
  - let's create a data array using `np.linspace()` to get a set of evenly spaced time points, and then `np.sin()` or `np.cos()` to create sinusoids as a function of time

```
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
plt.plot(t, s) # plot points in t on x-axis vs. points in s on y-axis
```

        - compare `np.linspace(start, stop, npoints)` with `np.arange(start, stop, step)`
            - `np.linspace()`
                - lets you specify the number of points you want to get out instead of step size
                - defaults to end-*inclusive* (`stop` value is included in the output)
            - `np.arange()`
                - lets you specify the step size between points instead of the number of points
                - is end-*exclusive* (`stop` value is excluded in the output)
        - `np.logspace()` is the logarithmic equivalent of `np.linspace()`, generates points equally spaced on a logarithmic scale instead of linear scale
    - if no existing plot window ("figure") exists, a new one will pop up, or will be embedded in your jupyter notebook
    - figure toolbar:
        - pan tool:
            - left button drag: pan horizontally and vertically
            - right button drag: zoom horizontally and vertically
            - back and forward buttons skip between recent views
        - home button returns to default view, forward and back buttons switch between recent views
        - magnifying glass: zoom to rectangle
            - left button drag to zoom to rectangle
            - right button drag to zoom out the view to fit rectangle
        - configure subplots: change borders, spacing between subplots (if any)
            - tight layout button minimizes borders and maximizes data, good for saving to file
        - edit plot params: titles, labels, limits, scales, line and marker formatting
        - save: save figure to disk, typically `.pdf` or `.png`
    - everything you can do interactively with the toolbar, you can also do programmatically in code
        - `plt.xlim()`, `plt.ylim()`, `plt.xlabel()`, `plt.ylabel()`, `plt.title()`
        - `plt.savefig()`
    - add another trace to the same plot:

```
c = np.cos(t) # calculate cosine as a function of the same timebase t
plt.plot(t, c) # plot points in t on x-axis vs. points in c on y-axis
```

    - by default, MPL adds the new line plot to the existing figure's axes, using a new color
    - create a new empty figure with `plt.figure()`
    - if you have multiple figures open, new plots go on most recently used figure
    - `plt.close()` closes current figure, `plt.close('all')` closes all figures
    - to specify color, marker type and line style with kwargs:
        - `color: 'red', 'green', 'blue'` etc.

        - `marker: '.', 'o', 'x', '+', '*'` , or `''` to turn off markers
        - `linestyle:` 'solid', 'dashed', 'dotted' , `or` "` to turn off the line
        - e.g. `plt.plot(t, c, color='red', marker='.', linestyle='solid')` `plt.plot(t, c, 'r.-')` is shorthand for the above
        - check `plt.plot?` docstring for more options, including color, marker and line abbreviations
        - `label` kwarg lets you give each line a name, which then shows up if you call `plt.legend()`

- histogram plot is useful for getting a graphical overview of the distribution of values in your data

    - `a = np.random.random(1000) - 0.5`
    - `plt.hist(a)` , defaults to 10 bins
    - `plt.hist(a, bins=100)` specifies desired number of bins
    - `plt.hist(a, bins='auto')` automatically finds a reasonable number of bins
    - can also specify the exact locations of all the bin edges by passing a sequence instead of a scalar to `bins` :
        - `plt.hist(a, bins=np.arange(-1, 1, 0.1))`
    - similar plotting options as for `plt.plot()` for controlling e.g. color

- anatomy of a MPL figure

    - axes, markers, lines, labels, titles, legends, ticks, grids, spines
    - spines are inexplicably complicated to toggle, for now use:
        - `plt.gca().spines['top'].set_visible(False)`
        - `plt.gca().spines['right'].set_visible(False)`
        - can set them permanently in your `matplotlibrc` file (see below)
    - `plt.axhline(y)` let's plot a horizontal line at `y` across the whole plot
    - `plt.axvline(x)` does the same for a vertical line
    - `plt.text(x, y, s)` places text `s` at (x, y)
    - `plt.Circle(xy, radius)` draws a circle at (x, y)
    - `plt.annotate()` should let you "annotate" a particular (x, y) point with some text (currently broken?)

- other kinds of plots:

    - scatterplots: `plt.scatter()`
    - error bars: `plt.errorbar()`
    - bar plots: `plt.bar()`

- 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 `%matplotlib` (or failing that, `plt.ion()`)
        - permanently enable interactive mode in matplotlib settings file:
            - this is the normal location for `matplotlibrc` :

        - linux: `~/.config/matplotlib/matplotlibrc`
        - mac + windows: `~/.matplotlib/matplotlibrc`
      - however, Anaconda seems to use a different version of the file in a weird path
        - to find out what that path is, type `plt.__file__` . Navigate to that folder, then go one folder deeper into the `mpl-data` folder, where you should find the `matplotlibrc` file
        - the path will probably look something like: `...something.../Anaconda3/site-packges/matplotlib/mpl-data/`
      - open the `matplotlibrc` file with a plain text editor (e.g. notepad in windows, textedit in mac)
      - uncomment `#interactive: False` line and set to `True` instead
    - if all else fails, you can always call `plt.show()` when you're done with plot commands, to show the figure, but then your command line might be inactive until you close the figure again :(
  - 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

matplotlib exercises:

1. Create three arrays:
   i. an array `t` that represents time in seconds, from `0` to `4*np.pi` in steps of 0.1.
   ii. an array `s` that is two times the sine of t
   iii. an array `c` that is the cosine of `t` plus 2
2. Plot both `s` and `c` vs. `t` in the same figure, using `plt.plot()`
3. Give the figure a title, and label the x axis `Time (s)` and the y axis `Position` , and save it to disk as either a PNG or a PDF. If you're missing the save button on the toolbar, try the `plt.savefig()` function
4. Create a new figure with `plt.figure()` and repeat 2., but now give each line a label by specifying the `label` kwarg when calling `plt.plot()` . Then call `plt.legend()` to add a legend to the figure. Now give it a title, label the axes, and save it as in 3.
5. Put all of your plot commands for 4. in a script named `plot_exercise.py` so you don't have to keep re-typing them. Make sure it runs when you type `run plot_exercises.py` in IPython or Jupyter.
6. By default, your above plots (might) have markers (dots) at each data point. Repeat 4 (use your script) but now turn off/on the markers by specifying the `marker` kwarg (hint: give it an empty string to turn them off).
7. Initialize a second figure in your script with `plt.figure()` after your code for the first figure.
8. Plot a histogram of `s` with 20 bins, and give it a label during the `plt.hist()` call. Now do the same for `c` . Label the x and y axes and add a legend.

9. Add two `plt.savefig()` calls in the appropriate places in your script to automatically save your two figures to file every time the script is run. Give the files the names `plot_example.png` and `hist_example.png` respectively. What folder will these files be saved to by default? What happens when you run the script multiple times?