more numpy 1D arrays, numpy file operations, plotting with matplotlib

more numpy 1D arrays

- `import numpy as np` to gain convenient access to numpy functions/modules/objects with `np.something`
- array methods
  - `a = np.random.random(10)` - array of random float values between 0 and 1
  - `a.max()`, `a.min()`, `a.ptp()`, `a.sum()`, `a.mean()`, `a.std()`
  - how can we shift all these values to have zero mean and a standard deviation of 1?

```
a -= a.mean() # now mean is very close to 0
a /= a.std() # now std is also very close to 0
```

    - note that e.g. `-3.271e-17` is shorthand for `-3.271 x 10^-17`
    - what if you now want the sum of the values to be 1?
  - `a.sort()` - sorts in place! same as for a list
  - `a.tolist()` returns list equivalent of `a`, same as `list(a)` if `a` is 1D
  - many array methods have an equivalent numpy function, e.g. `np.max()`, `np.min()`, etc., which you can use directly on lists or tuples
- review integer dtypes:
  - what's a digit? numeric symbol used to represent numbers
  - "digiti": Latin for "fingers", decimal digits are numeric symbols for counting in base 10, values 0 to 9
  - what's a bit? bit = binary digit, bits are numeric symbols for counting in base 2, values 0 and 1
  - what's a byte? 8 bits: 00000000, 00000001, 00000010, 00000011 ... == 0, 1, 2, 3, ...
  - 8 bit integer takes up 1 byte of memory
  - 16 bit takes up 2 bytes
  - 32 bit takes up 4 bytes
  - 64 bit takes up 8 bytes
  - total number of values expressable by an int: `2**nbits`
  - max value for unsigned int: `2**nbits - 1`
  - min/max val for signed int: `-(2**nbits)/2, (2**nbits)/2 - 1`
    - or, use `np.iinfo()`
    - usually safest to use signed integers instead of unsigned, especially for subtraction
  - what's overflow? it's what happens when you exceed the max or min value
  - when converting between dtypes, numpy *warns* about overflow but doesn't stop execution:
    - what's the max value expressable by int8?
    - `np.int8(120) + np.int8(10)` warns
    - get numpy error settings using `np.geterr()`
    - set numpy error setting using e.g. `np.seterr(over='raise')` to "raise" an *error* on overflow, which stops execution of your code, more strict than a *warning*
      - `np.seterr()` returns old settings before setting new ones

- exercise: create an empty 1D array (using `np.zeros()`) of length 1 million that can store values integer values from -1000 to 1000
    - which dtype can do so while using the least memory?
    - how many bytes of memory do you predict it will use?
    - check `a.nbytes` to see if you got it right
    - is it safe to add/subtract two such arrays to/from each other?
- deciding between lists and arrays:
    - use a list when:
        - have heterogenous data types you want to store together in a sequence
        - want to easily add and remove items to/from it
        - don't have to store a very large number of items, memory use isn't an issue
        - don't have to do vectorized operations on the sequence, e.g. adding two of them together
    - otherwise, use an array!

plotting with matplotlib (MPL)

- main plotting library for python, others exist, but often based on MPL
- 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 a nice sinusoid 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
                - is end-inclusive (`stop` value is included in the output)
            - `np.arange()`
                - lets you specify the step size between points
                - is end-exclusive (`stop` value is excluded in the output)
        - `np.logspace()` is the logarithmic equivalent of `np.linspace`, but creates requested number of 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
        - 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()`
- add another line 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 colour
- create a new empty figure with `plt.figure()`
- 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', '+', '*'`
    - `linestyle:`'solid', 'dashed', 'dotted', 'None'`
    - 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 in `plt.legend`

- histogram plot is useful for getting a graphical overview of all the values in your data
    - `plt.hist(a)`, defaults to 10 bins
    - `plt.hist(a, bins=100)` specifies desired number of bins
    - similar plotting options as for `plt.plot()` for controlling e.g. colour
    - see documentation for lots of details

- anatomy of a MPL figure
    - http://matplotlib.org/examples/showcase/anatomy.html
    - axes, markers, lines, labels, titles, legends, ticks, grids, spines
    - annotate, text, circle

- plotting issues:

    - 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 icon for "edit axes/curve/image params" in figure window might be missing

- exercise:

    - create 1D array using `np.sin()` or `np.cos()`
    - plot it with `plt.plot()` to see what it looks like
    - give it some labels, save the plot to disk
    - save the array to a text file with `np.savetxt()`
    - examine the text file in your text editor
    - now save the same array to binary file using `np.save()`
    - exit ipython/jupyter
    - compare the size of the text and binary file
    - restart ipython/jupyter
    - load array twice: from the text file & from the binary file, save to two different names
    - plot both arrays, compare them to each other, compare to saved plot to make sure they look the same

- other kinds of plots:

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

numpy file operations

- so far we've been using mostly made-up values to fill arrays, generated in code
- in reality you have to load data from disk, and save results (and figures) back to disk
- loading/saving arrays from/to files:
- 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 bytes represent which chars
            - 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 various characters, plus outdated "characters" that controlled direct output to printers and communications with old modems
            - see `ASCII-Conversion-Chart.pdf`
            - a newer increasingly common one 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.
        - 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 "hex" editor to directly view them
        - 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: HexEdit
            - mac: Hex Fiend
            - linux: ghex, bless
        - mostly you load/save them programmatically anyway, no need to directly edit them
    - which file type to use depends on your data source, and your data size
    - for large data sets, like images or electrophysiology, binary files are critical, text files aren't appropriate

- loading/saving arrays from/to text files

    - `np.savetxt(fname, a)` - save to a text file
        - use the `delimiter=','` kwarg to create comma separated values
        - notice that dtype information can be lost using the above, `fmt=%g'` kwarg helps
        - saving to and loading from binary files handles metadata better
    - `np.loadtxt(fname)` - load from a text file
        - use the `delimiter=','` kwarg to handle e.g. comma separated values, see `test.csv`

- loading/saving arrays from/to binary files:

    - to see hex representation of bytes in memory for array `a` : `a.tobytes()`
    - `np.save(fname, a)` - to a binary `.npy` file
    - `np.load(fname)` - from a binary `.npy` file, or a `.zip` file containing multiple `.npy` files
    - can inspect binary files with hex editor
    - `np.savez('experiments', exp1=a, exp2=b)` & `np.savez_compressed()` - save multiple arrays to an uncompressed or compressed `.zip` file
    - optional: for loading/saving raw binary representation of array data from/to files, for use with other systems:
        - `np.fromfile()`
        - `a.tofile()`