numpy 1D arrays

go over solutions to homework 2

numpy

- numpy is the main numerical library in Python, basis for many other scientific Python libraries
  - typical usage: `import numpy as np`
  - numpy provides: 1. the `ndarray` object, 2. lots of numerical and array functions
  - arrays are sequences, like lists and tuples, but faster and much more memory efficient
    - ideal for large datasets!
  - unlike lists, can explicitly be multidimensional - useful for e.g. images and movies
  - tradeoff: not as flexible as lists - for efficiency, each entry in an array has to be of the same data type
    - you can have an array of ints, or floats, or strings or booleans, but not a mixture
    - so far, we've seen that there are two main numeric data types: int and float
    - different kinds of integers and floats (see later), each array can contain only one kind
  - like a tuple, array length generally **can't** change, but like a list, its values **can** be changed, so it's "semi-mutable"

- initializing an array
  - explicitly, using a list or a tuple, convert to array:
  - `a = np.array([1, 2, 3])` or `a = np.array((1, 2, 3))`
  - `a = np.arange(10)`
    - very similar to `list(range(10))` , but returns an array instead of a list
  - `a = np.zeros(10)`
  - `a = np.ones(10)`
  - `a = np.random.random(10)` - 10 random numbers uniformly distributed between 0 and 1
  - `a = np.tile([1, 2], 5)`
  - `a.fill(7)` fills the existing array `a` with the number 7
  - array methods (e.g. `a.fill()`) usually operate on the array in-place, while numpy functions (e.g. `np.zeros()`) usually return a new array
  - here's an exception: `b = a.copy()`
  - numpy functions often have array method counterparts (and vice versa)
    - `copy()` and `sort()` are two examples:

    ```
    a = np.random.random(10)
    b = a.copy()
    c = np.copy(b)
    ```

    - are `a` , `b` and `c` equal? test with `==` , get a boolean answer for each entry
    - are `a` , `b` and `c` the same objects? test with `is` , get a single bool answer

    ```
    d = a
    d.sort() # in-place
    e = np.sort(a)
    ```

    - are `a` , `d` and `e` equal? are they the same objects?

- like other sequences (tuples & lists), get length of array using `len(a)` , but can also get array shape using `a.shape` attribute
    - shape returns the length along all dimensions of `a`
    - length of the first dimension is `a.shape[0]` , identical to `len(a)`
    - get num dims with `a.ndim` , multidimensional arrays covered later
- indexing in 1D is the same as for tuples & lists: 0-based, -ve indices count from the end
    - `a[0] = 7` assigns `7` to 1st entry
    - `a[1] = 7` assigns `7` to 2nd entry
    - `a[-1] = 7` assigns `7` to last entry
    - `a[-2] = 7` assigns `7` to 2nd last entry
- slicing in 1D is also the same as for tuples and lists
    - retrieve a slice: the first 5 entries
        - `b = a[0:5]` or `b = a[:5]`
    - assign to a slice: the last 5 entries
        - `a[5:10] = 7` or `a[5:] = 7`
    - assign to a slice: all entries
        - `a[:] = 8` , same as `a.fill(8)`
        - what happens if you go `a = 8` ?
- arrays also have "fancy" indexing:
    - allow you to ask for multiple values from an array at once
    - two types: integer & boolean fancy indexing
    - both are kind of hybrid between normal indexing and slicing
        - benefit over slicing is that you can specify any sequence of indices, not just evenly spaced ones
        - you can even specify the same index multiple times
    - integer fancy indexing
        - ```
          a = np.random.random(10) # init an array of random data
          i = [3, 7, 5, 2, 7] # create a list of indices
          vals = a[i] # index into array using integer fancy indexing
          a[i] = -1 # assign -1 at multiple locations using integer fancy indexing
          ```
        - can ask for array values in arbitrary order
        - can ask for the same value repeatedly
        - can't do this with lists: try `list(range(10))[i]`
    - boolean fancy indexing
        - ask some question of values of the array, get an answer back of boolean values of same length as original array
        - `i = a > 5` returns an array of booleans, which can be used for indexing
        - `a[a > 5]` or `a[i]` returns only those entries in a that are > 5
        - i.e., where `i` is True, return the value in `a` at that index
        - what if you have another array `b` that is of different length? can you also index into it with the above `i` ? no!
        - again, can't do this with lists: try `l[i]`
- **vectorized** math operators (`+` , `-` , `*` , `/` , `**`) and comparitors (`==` , `>` , `<` , `!=`)
          ```

    - vectorized: work on all values of an array at the same time
    - `a = np.array([1, 2, 3])`
    - arrays & scalars
        - `a + 1` returns a new array with 1 added to all the values in `a`
        - `a += 1` increments a in-place by 1, doesn't return anything
        - `a -= 1` decrements a in-place by 1, doesn't return anything
    - `b = np.array([4, 5, 6])`
    - `a + b` returns another array each of whose values are the sum of the corresponding two values in `a` and `b`
        - in comparison, what does `+` do for strings and lists?
        - use `np.concatenate([a, b])` or `np.concatenate((a, b))` to combine arrays
    - what happens if you try to do one of the above vectorized operations on two arrays of different length?

array exercises:

1. Use a for loop to build a list of 3 arrays, each array of length 5, initialized to zeros
2. Find the average difference between the following two arrays: `a = np.array([10, 20, 30, 40, 50])`, `b = np.array([5, 10, 15, 20, 25])`
3. Write a function called `rms()` that calculates the RMS (root mean square) of an input sequence (array, list, tuple). RMS is the the square root of the mean of the square of a signal.
4. Use your `rms()` function to calculate the RMS of the difference between the two arrays

<go over solutions>

5. Create an array `c` of 10 random numbers between 0 and 10.
6. Create an array `d` that consists of the values in `c` greater than 7.
7. Create an array `e` of 10 random numbers between -5 and 5.
8. Create an array `f` that consists of values in `e` greater than 1 and less than -1.
9. Create an array `g` that has all the values of both `d` and `f`. How long is it?
10. Create an array `h` that has only the 3rd, 8th and 11th entries in `g`

- memory
    - what's system memory (RAM)? computer's working memory
    - what's a byte? 8 bits
    - what's a bit? a binary digit, can be a 0 or 1
    - different numeric values are expressed using different combinations of bits
        - 1 byte, 8 bits allow for 2**8 = 256 different numeric values to be expressed
        - `00000000, 00000001, 00000010, 00000011 ... == 0, 1, 2, 3, ...`
    - how much memory does my array use?
        - `a.nbytes`
        - memory use depends on the number of elements in the array, times the size of each element
        - element size depends on the data type (dtype) of the array - `a.dtype`
        - `a.nbytes == len(a) * a.dtype.itemsize` for 1D arrays
- array data type (dtype)

- a common set of numeric data types are used across programming languages, super important!

- integers: signed and unsigned

    - signed integers are symmetric around 0, unsigned integers are always >= 0
        - if `n` is the number of unique integers that can be represented by an integer data type:
        - signed integers range from `-n/2` to `n/2-1`
        - unsigned integers range from `0` to `n-1`
        - `n = 2**nbits`
        - so, the bigger the integer data type (in bits or bytes), the more integer numbers it can represent
    - `np.int8` , `np.int16` , `np.int32` , `np.int64` - 1, 2, 4 and 8 byte signed
    - `np.uint8` , `np.uint16` , `np.uint32` , `np.uint64` - 1, 2, 4 and 8 byte **un**signed
    - can easily calculate max/min values of each int dtype, or use `np.iinfo()` , e.g. `np.iinfo(np.int8).max`
    - init arrays to the desired data type by using the `dtype` kwarg:
        - `a = np.zeros(10, dtype=np.uint8)`
        - `b = np.zeros(10, dtype=np.int64)`
    - integer overflow:
        - `a[:] = 255` is fine, but `a[:] = 256` and `a[:] = -1` both wrap overflow (wrap around)
        - `b[:] = 127` is fine, but `a[:] = -128` isn't
        - `b[:] = -128` is fine, but `b[:] = -129` isn't
    - when to use signed or unsigned? if in doubt, use signed!

- floats - always signed, and made of "mantissa + 10^exponent"

    - bigger floats have greater precision
    - `np.float16` , `np.float32` , `np.float64` - 2, 4 and 8 bytes floats

- by default, arrays init to the biggest dtypes, either `np.float64` or `np.int64` :

    - `a = np.array([1, 2, 3])` , `a.dtype` -> int64
    - `b = np.array([1.1, 2.2, 3.3])` , `b.dtype` -> float64

- init arrays to the desired data type by using the `dtype` kwarg:

    - `a = np.zeros(10, dtype=np.int8)`
    - `b = np.zeros(10, dtype=np.int64)`
    - `c = np.zeros(10, dtype=np.float64)`
    - calculate how much memory do `a` , `b` and `c` take, then check it using `.nbytes`

- how much memory would `a = np.zeros(20000000000, dtype=np.uint8)` use? what would happen if I tried this on my 16 GB laptop? `MemoryError`

- can convert from one dtype to another by using the dtype as a function:

    - `a = np.array([1, 2, 3])`
    - `np.float64(a)` converts `a` to float64 dtype
        - similar to basic Python: `float(val)`
    - `a = np.array([1.1, 2.2, 3.3])`
    - `np.int64(a)` converts `a` to int64 dtype, but it truncates!
        - this is similar to basic Python: `int(val)`

        - use `np.int64(np.round(a))` to round to the nearest integer instead
    - check array data type with `a.dtype`
  - usually only need to worry about int vs float dtype, stick to the defaults `int64` and `float64`
    - only consider going down to smaller dtypes if you have lots of data and not enough memory on your machine
  - take care converting between dtypes!
    - especially from larger ones to smaller ones, and from floats to ints
    - a number that can be represented in one data type might not be possible to represent in another
    - dramatic example: Ariane 5 1996 failure
      - code adapted from Ariane 4 tried to convert a large float64 to int16, resulted in integer overflow, caused computer to think it was suddenly way off course, tried to correct by rapidly changing direction, high G-forces caused it to start to disintegrate, which triggered self-destruct. Cost: $370M

quickie numeric data type exercises:

1. Create a tuple or a list with the following entries: `3, 5, 1.7, -2.7, 1e2, -50`. Now convert it to an array. What happens?
2. You have integer data whose values span -100 to 100. Normally, you would use an int64 array to store this data, except the dataset is huge (1 billion entries) and your laptop has a measly 4 GB of RAM. How much memory would you need to store your data using the default int64 dtype? What would be the optimal dtype to use to minimize memory use by your dataset? Will it fit into your 4 GB of RAM?