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
  - only deal with 1D for now
  - 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 with int 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])`. Use the function/method called `np.mean()` or `a.mean()`
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. Use the function/method called `np.sqrt()` or `a.sqrt()`
4. Use your `rms()` function to calculate the RMS of the difference between the two arrays

<go over solutions>