numpy 1D arrays

go over solutions to homework 2

Gotcha: comparing by reference vs. value

- for mutable sequences (like lists or dictionaries), be aware of difference between a reference and a copy:

1. `a = [1, 2, 3]; b = a`
    - `a` and `b` point to (reference) the same object in memory, the list `[1, 2, 3]`
    - if we set `b[2] = 100`, what happens to `a`?
2. `a = [1, 2, 3]; b = a.copy()` - lists have a `.copy()` that copies the list to a new memory location
    - now `a` and `b` still have the same value, but point to (reference) different objects in memory
    - now if we set `b[2] = 100`, what happens to `a`?

- `is` keyword vs. `==` operator
    - `a = [1, 2, 3]; b = a`
    - `a == b` returns True
    - `a is b` returns True
    - `a = [1, 2, 3]; b = a.copy()`
    - `a == b` still returns True
    - `a is b` returns False
    - `a is [1, 2, 3]` also returns False
    - `b is [1, 2, 3]` also returns False
    - `==` checks for value, i.e. whether two variables have the same value
    - `is` keyword checks for identity, i.e., whether two variables point to the same object stored in memory, i.e. whether they point to the exact same spot (*address*) in memory
    - generally, it's safer and less confusing to use `==` than `is`, but good to know the difference.

numpy

- numpy is the main numerical library in Python, is the 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

        - there are different kinds of integer and float data types (see class 05), 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"

- lots of ways to initialize an array

    - explicitly: first create a list or a tuple, then convert to an array:
    - `a = np.array([1, 2, 3])` or `a = np.array((1, 2, 3))`
    - `a = np.arange(10)` returns a range of integers
        - very similar to `list(range(10))`, but returns an array instead of a list
    - `a = np.zeros(10)` - an array with 10 entries, all `0.0`
    - `a = np.ones(10)` - an array with 10 entries, all `1.0`
    - `a = np.random.random(10)` - 10 random numbers uniformly distributed between 0 and 1
    - `a = np.tile(5, 10)` - 10 copies of the integer 5
    - `a = np.tile([1, 2], 5)` - 5 copies of the sequence `[1, 2]`
    - `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(b)
        ```

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

- 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)`
    - `a.ndim` tells you the number of dimensions - 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]`
- with arrays, you can also *assign* values to a slice:
    - assign to the last 5 entries
        - `a[5:10] = 7` or `a[5:] = 7`
    - assign to all entries with `:`, i.e. slice from start to end
        - `a[:] = 8`, same as `a.fill(8)`
    - what happens if you do `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 of fancy indexing 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:

    ```
    l = list(range(10))
    l[i] # TypeError: list indices must be integers or slices, not list
    ```

    - boolean fancy indexing
        - ask a question of values of the array, get an answer back made up of boolean values of same length as original array
        - `i = a >= 5` returns an array of booleans, which can be used for indexing
        - `a[i]` or `a[a >= 5]` 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: `l[i] # TypeError`
- **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 entries 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 vector difference between the following two arrays and assign it to a new array called `d`:

    ```
    a = np.array([10, 20, 30, 40, 50])
    b = np.array([5, 12, 18, 31, 45])
    ```

3. "Reduce" `d` to a single number by using the function `np.mean()` or the method `d.mean()`
4. Write a function called `rms()` that calculates the RMS (root mean square) of an input array. RMS is the the square root of the mean of the square of a signal. To calculate square root, use the function `np.sqrt()`. This can be done in a single line.
5. Use your `rms()` function to calculate the RMS of the difference between the two arrays in 2.
6. Concatenate `a` and `b` into a new array called `c`. Now sort `c`, either using the function `np.sort()` or the method `c.sort()`
7. Create a boolean array `i` that describes where the values in `c` fall between 10 and 20 (inclusive)
8. Use `i` to extract the corresponding values from `c`
9. Use *integer* fancy indexing to set the 1st, 3rd and 4th entries in `c` to 0. Check it.
10. Use *boolean* fancy indexing to set the 1st and last entries in `a` to -1. Check it.

<go over solutions>