numpy data types & file operations

array review (mostly)

- `import numpy as np` to gain access to numpy functions/modules/objects with `np.something`
- generating array:
    - from a list `np.array([1, 2, 3])`, or tuple `np.array((1, 2, 3))`
    - `np.arange()`, `np.zeros()`, `np.ones()`, `np.random.random()`
    - by combining multiple lists, tuples or arrays: `np.concatenate([a, b, c])`
- indexing:
    - single value
    - fancy indexing: integer and boolean

    ```
    a = np.arange(10)
    b = a > 5 # boolean indices
    ```

    - use `np.where()` to get integer indices from boolean indices
    - `i = np.where(a > 5)` # returns tuple of integer indices, one per dimension
    - `i = np.where(a > 5)[0]` pulls out integer indicies into first dimension
- vectorized math operators (`+`, `-`, `*`, `/`, `**`) and comparitors (`==`, `>`, `<`, `!=`)
    - `a = np.array([True, False, False])`
    - `b = np.array([True, True, False])`
    - normal boolean logic operators (`and`, `or`, `not`) don't work as vectorized operators on arrays
        - e.g., `a and b` gives an error
        - instead us vectorized boolean operators `&`, `|`, `~`
        - e.g. `a & b` and `a | b` and `~a` and `~b` work as you would expect
    - are all values in `a` True? `a.all()` or `np.all(a)`
    - are any values in `a` True? `a.any()` or `np.any(a)`
- common array math methods: `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
    ```

- `a.sort()` sorts in place, `b = np.sort(a)` creates a sorted copy of `a`
- `np.diff()` finds the difference between consecutive values in `a`
    - e.g., `np.diff([1, 4, 2, -3])` gives `np.array([3, -2, -5])`

more array exercises from last class:

5. Create an array `c` of 10 random numbers that range from 0 to 10 at most
6. Create an array `d` that has only the 2nd, 5th and 8th entries in `c` (one line of code!)
7. Create an array `e` that has only the values in `c` greater than 5
8. Use `np.where()` to get the integer indices of where `c` is greater than 5.
9. Check that all the values in `e` really are > 5 (one line of code!)
10. Create an array `f` of 10 random numbers that range from -1 to 1 at most
11. Create an array `g` that only has the values in `f` that fall between -0.5 and 0.5

12. Check that all the values in `g` really are between -0.5 and 0.5 (one line of code!)
13. Create an array `h` that has all the values of both `c` and `f`. How long do you expect it to be? Check its length.
14. Sort the values in `h` in-place. Use `np.diff()` in one line of code to check that `h` really is sorted.

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!

memory

- what's system memory (RAM)? computer's working memory (random access memory)
- what's a byte? 8 bits
- what's a bit?
  - a BInary digIT, numeric symbol for counting in base 2, can be 0 or 1
  - decimal digit: numeric symbol for counting in base 10, ranges 0 to 9
- 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)

- there are subtypes of both int and float, these are across programming languages, super important!

- **integers**: signed and unsigned

  - signed integers are symmetric around 0, unsigned integers are always >= 0
    - `n = 2**nbits` is the number of unique integers that can be represented by an integer data type:
    - unsigned integers range from `0` to `n-1`
    - signed integers range from `-n/2` to `n/2-1`
    - so, the bigger the integer data type (in bits and therefore bytes (`nbytes = nbits / 8`), the more integer numbers it can represent
  - `np.int8`, `np.int16`, `np.int32`, `np.int64` use 1, 2, 4 and 8 bytes, **signed**
  - `np.uint8`, `np.uint16`, `np.uint32`, `np.uint64` use 1, 2, 4 and 8 bytes, **unsigned**
  - can easily calculate max/min values of each int dtype, or use `np.iinfo()`, e.g. `np.iinfo(np.int8)`
    - access results using `.max` and `.min` attributes
  - init arrays to the desired data type by using the `dtype` kwarg:

        - `a = np.zeros(5, dtype=np.uint8)` - smallest unsigned int
        - `b = np.zeros(5, dtype=np.int8)` - smallest signed int
    - integer overflow when filling or doing in-place math:
        - `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 doing int math, numpy gives the result in the next biggest dtype if it won't fit in the existing dtypes
        - `a[:] = 200`, `b[:] = 100`, `a + b` gives result as `int16` dtype
    - when to use signed or unsigned? if in doubt, use signed!
    - 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`

- **floats**: always signed, and made of "mantissa + 10^exponent", e.g. `1.23456789e02`

    - some bits that make up a float are used for the mantissa, some for the exponent

    - bigger float data types have greater resolution (mantissa) and greater range (exponent), but use more memory

    - `np.float16`, `np.float32`, `np.float64` use 2, 4 and 8 bytes. Is there a `np.float8`?

    - to get max/min/resolution of a float dtype, use `np.finfo()`

        - e.g. `np.finfo(np.float16)` gives `finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04, dtype=float16)`
        - access results using `.max`, `.min` and `.resolution` attributes
        - note that resolution refers to the mantissa, not of the full mantissa + 10^exponent
        - as you get to bigger and bigger numbers, the effective resolution decreases because of the increasing exponent
            - `np.float16(1.234567e4)` gives `12344.0`
        - as you get to smaller numbers, effective resolution increases because of the decreasing exponent
            - `np.finfo(np.float16).tiny` gives `6.104e-05`, the smallest representable value

    - special values:

        - `np.inf` and `np.nan`, i.e. "infinity" and "not a number"
            - `np.inf` is used to represent out of range float values
            - `np.nan` is used to represent invalid float values, like 1/0
            - `inf` is signed (can be +/-), but `nan` has no sign
            - doing any math involving `inf` or `nan` always results in another `inf` or `nan`
            - `np.inf + np.nan` gives `np.nan`
            - comparing `nan` to anything, even itself, returns `False`, have to use `np.isnan()`

- numpy arrays default 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
    - initialize 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 `a`, `b` and `c` should be using, then check it using `.nbytes`

- special case: `bool` dtype
    - uses one byte per entry, just like int8 and uint8
    - `b = np.array([True, False, False])`
    - `b.dtype`, `b.nbytes`
    - could this be more efficient? yes, bool arrays could use single bits instead of a full byte for each value, but normal computers allocate memory no finer than a single byte

- **typecasting**: convert from one dtype to another
    - 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 when typecasting (converting between dtypes)!
    - especially from larger dtypes to smaller dtypes, 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

numeric data type exercises:

1. Create a sequence (tuple or a list) with the following entries: `3, 5, 1.7, -2.7, 1e2, -50`.
    i. What are the data types of the individual values?
    ii. What do you predict will happen if you convert this sequence to an array? What will the array's dtype be? How much memory will it use (in bytes)?
    iii. Now check your predictions. Convert the sequence to an array and check both its `.dtype` and its `.nbytes`.
2. You have integer data whose values span -2 to 2. Normally, you would use an integer array with numpy's default int64 dtype to store this data. But, the dataset is huge (1.5 billion entries) and your laptop only has 4 GB of RAM.
    i. How much memory would your data take up if you used the default int64 dtype in numpy?
    ii. Should you use an int or float dtype? Signed or unsigned?

   iii. What would be the optimal dtype to minimize the amount of memory used by your dataset? Will it fit into your 4 GB of RAM?

3. Repeat question 2. for integer values that span 0 to 10000.
4. Repeat question 2. for integer values that span -1 to 50000.