

## numpy 1D arrays

- review of collections:
  - sequences:
    - tuples and lists
    - which are mutable/immutable?
    - how to initialize them?
    - `val in sequence` keyword asks whether a value can be found in a sequence
      - `1 = [1, 2, 3]`
      - `2 in 1` returns True, `5 in 1` returns False
  - mappings:
    - dict, OrderedDict
    - map keys to values
    - add new key:value pairs with `d[key] = value`
      - what happens if key already exists?
    - access key:value pairs with `d[key]`
      - what happens if key doesn't exist in d? get a `KeyError`
    - remove an existing key:value pair with `del d[key]`
      - what happens if key doesn't exist in d? get a `KeyError`
    - dictionary methods, these were skipped last class:
      - `list(d.keys())` returns a list of d's keys
      - `list(d.values())` returns a list of d's values
      - `list(d.items())` returns a list tuples of d's (key, value) pairs
      - `d[key].pop()` returns the value of `d[key]` and also removes the key and its val from d
    - `val in dict` works as for sequences, but asks whether val is a key in dict
- numpy: main numerical library in Python
  - basis for many other scientific Python libraries
  - numpy provides the `ndarray` object + lots of array functions
  - arrays are a type of sequence, like lists and tuples, but faster and much more memory efficient
  - arrays are 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
  - like a tuple, array length generally **can't** change, but like a list, its values **can** be changed, so it's "semi-mutable"
  - typical usage: `import numpy as np`
- 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
  - `a = np.zeros(10)`

- `a = np.ones(10)`
- `a = np.empty(10)` inits an array without resetting the values, `np.zeros` is preferred
- `a = np.random.random(10)`
- `a = np.tile([1, 2], 5)`
- `a.fill(7)` fills the array with the number 7
- array methods tend to operate on the array in-place, while numpy functions tend to return a new array, but there are lots of exceptions
- exercise: use a for loop to build a list of 3 arrays, each array of length 5, initialized to zeros
- 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`, multidimensional arrays covered later
  - length of the first dimension is `a.shape[0]`, identical to `len(a)`
- 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:
  - two types: integer & boolean fancy indexing
  - both are kind of hybrid between normal indexing and slicing
  - allow you to ask for multiple values from an array in a single call
  - integer fancy indexing

```
i = [3, 7, 5, 2, 7] # create a list of indices
vals = a[i] # this is integer fancy indexing
a[i] = -1 # assignment 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 `l[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]` returns only those entries in `a` that are `> 5`
  - 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 (`==`, `>`, `<`, `!=`)
  - what does vectorized mean? they work on all values of an array at the same time
  - `a = np.array([1, 2, 3])`
  - `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
  - arrays & scalars, vs. arrays & arrays
    - `a + 2` returns an array with 2 added to all the values in `a`
- array methods
  - `a.max()`, `a.min()`, `a.ptp()`, `a.sum()`, `a.mean()`, `a.std()`
  - `a.sort()` - in place!
  - `a.tolist()` returns list equivalent of `a`, same as `list(a)` if `a` is 1D
  - many have an equivalent numpy function, e.g. `np.max()`, `np.min()`, etc.
  - gotcha: in-place vs copy
- loading/saving arrays from/to text files:
  - text vs. binary files?
    - text files are easier to view in a text editor
    - binary files require a "hex" editor, harder to view and edit, but are much more space efficient and faster
      - same amount of data can be stored using less disk space
    - which one to use depends on how your data are saved
    - for large data sets, like images or electrophysiology, binary files are critical
  - `np.loadtxt(fname)` - recommended way to load from a text file
  - `np.savetxt(fname, a)` - recommended way to save to a text file
    - notice that dtype information can be lost using the above
    - saving to and loading from binary files can prevent that, covered later
- basic numpy data types (dtype)
  - used across programming languages, correspond to underlying C data types
  - integers
    - signed integers are symmetric around 0, unsigned integers are `>= 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
  - what's a byte? 8 bits
  - `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 **unsigned**
  - find max/min value of each type using `np.iinfo()`, e.g. `np.iinfo(np.int8).max`
  - when to use signed or unsigned?
  - integer overflow and underflow
- floats - always signed, and made of "mantissa +  $10^{\text{exponent}}$ "
  - bigger floats have greater precision
  - `np.float16`, `np.float32`, `np.float64` - 2, 4 and 8 bytes floats
- can convert from one dtype to another by using the dtype as a function:
  - e.g., `np.float64(a)` converts `a` to float64 dtype
- 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
    - float64 to int16 conversion 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
- commonly used array attributes:
  - `a.shape`, `a.ndim`, `a.dtype`, `a.nbytes`
- exercise: create a 1D array of length 1 million that's suitable for storing integer values ranging from 0 to 1000, while using as little memory as possible
  - how many bytes of memory do you predict it will use? how many does it actually use?
  - is it safe to add/subtract two such arrays to/from each other?
  - unless you absolutely need the extra double max value, it's safer to use signed integers, in case of subtraction
- 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 from a sequence
    - 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!