collections: tuples, lists, dictionaries

- review:

    - different ways of running code/scripts
        - in terminal/command line: type `python myscript.py`
            - exits immediately back to terminal when done, can't inspect variables or plots
            - add `input()` to last line of script to prevent exiting
        - run `ipython`, type `run myscript.py`, then you can inspect variables when it's done
        - run `python` or `ipython` interpreter, copy and paste code from editor to interpreter
            - IPython handles pasted code better than plain Python
        - Jupyter: like IPython, but in a web browser
    - IPython tips:
        - tab completion, saves time, mistakes, frustration - use it!

          ```
          verylong<TAB> -> verylongvariablename
          import math
          math.<TAB>
              .acos .acosh, .asin, .asinh, etc.
          math.fac<TAB> -> math.factorial
          cd lon<TAB> -> cd long\ pathname\ with\ spaces
          ```

        - `obj?` - get help about obj (variable, function, etc.)
        - separately numbered input and output lines
        - `_`, `__`, `___` - return last/2nd last/3rd last output
        - `_5` - return output of output line 5
        - `ipython_config.py` file in your hidden `~/.ipython` directory for changing defaults
    - functions - why bother writing them?
        - code reuse: replace multiple lines of code with just one

      ```python
      def rms(a, b):
          """Return root mean square of the inputs a and b """
          import math
          sumsquare = a**2 + b**2
          meansquare = sumsquare / 2
          return math.sqrt(meansquare)
      ```

- ways of installing python libraries/packages/modules, for familiarity, in decreasing order of ease:

    - `conda install`
    - `pip install`
    - less recommended: binaries (.exe, .zip), especially in windows, .dmg on Mac
    - ubuntu/deb repositories
    - advanced: from original source code, might require compiling

- collections

    - data types for storing multiple values together as one variable

  - sequences
    - tuples
    - lists
  - mapping
    - dictionary
    - ordered dictionary

- sequences:

  - tuples
    - comes from words like "quadruple, quintuple, etc"
    - wiki: "A tuple is a finite ordered list of elements"
    - denoted by **parentheses**, contain comma separated list of objects
    - can hold anything: integers, floats, strings, etc.
    - once declared, **cannot** be modified: "immutable"
    - e.g. `t = [1, 2, 3]` or `t = ['a', True, 3.14]`
      - often the parentheses are optional: `t = 1, 2, 3`
    - tuple expansion allows for multiple assignment:
      - `a, b, c = (1, 2, 3)` or simply `a, b, c = 1, 2, 3`
    - methods:
      - `t.count(val)` returns number of occurrences of val
      - `t.index(val)` returns index of first occurence of val
    - tuples are often used to `return` multiple values from a function

```
def myfunc(a):
    return a, 2*a, 3*a
a, b, c = myfunc(2)
```

  - lists
    - denoted by **square brackets**, contain comma separated list of objects
    - can hold anything: integers, floats, strings, etc.
    - once declared, **can** be modified: "mutable"
    - e.g. `l = [1, 2, 3]` or `l = ['a', True, 3.14]`
    - initialize empty list with `l = []` or `l = list()`
    - same methods as tuple, plus these ones:
      - `l.append(val)`
      - `l.extend(anotherlist)`, or `l + [4, 5, 6]`
      - `l.reverse()`
      - `l.sort()`
        - does `.sort()` work for lists of objects of different types?
      - `l.clear()`
      - all the above methods operate *in place*, i.e. they modify the list, but don't return anything
    - typical way to build up a list:

```
l = []
for i in range(10):
    l.append(i)
```

or `l = list(range(10))`
            - convert a tuple to a list with `list()`
                - `l = list((1, 2, 3))`
    - indexing for tuples and lists is 0-based, same as for strings:
        - `t[0]` returns the first index, `t[n-1]` returns the last
        - negative indices denote distance from end, starting with -1:
        - last value is `t[-1]`, second last is `t[-2]`, etc.
        - delete entries from a list with `del` by specifying their index: `del l[2]`
    - slicing
        - `a[start:stop:step]`
        - fencepost analogy, slicing from one fencepost to another, not from one slot to another
        - negative indices also work for slices
        - colon `:` can be used as placeholder for start or stop if you don't want to specify them
    - iterating over sequences
        - `for val in sequence:`
            - when iterating over a sequence using `enumerate()`, you also get the index of each value, which can be useful inside the loop
                - `for index, val in enumerate(sequence):`
        - list comprehension: handy for doing something repetitive to build up a list in a single line of code
            - `doubledlist = [ 2*val for val in sequence ]`
    - common functions: `min(), max(), mean(), sorted(), tuple(), list()`

- mappings:

    - what if you want to store your values by name, instead of by index?
    - a "mapping" maps keys (names) to values
    - dictionaries
        - init with `{}` or `dict()`
        - add new key:value pairs with `d[key] = value`
            - what happens if key already exists?
        - various methods
        - iterating over dicts
            - `for key in list(d):`
            - `for key, val in d.items():`
            - `for val in d.values():`
            - dict comprehension:
                - `doubleddict = { key:2*val for (key, val) in d.items() }`
        - NOTE: order of keys in dict is not preserved! because dict is a mapping, from keys to values, not just a sequence of things, like a tuple or list
    - OrderedDict
        - OrderedDict is a hybrid of mapping and a sequence, preserves key order
        - `from collections import OrderedDict as odict`

- combining tuples, lists, dicts, any combination is possible, can be nested

    - common ones: list of tuples, dict of lists

- reference vs. a copy for mutable sequences, things get tricky!:
    1. `a = [1, 2, 3]; b = a`
        - `a` and `b` point to the same object in memory
    2. `a = [1, 2, 3]; b = a.copy()`
        - `a` and `b` have the same value, but point to different objects in memory
    - if we set `b[2] = 666`, what's the value of `a` in the above two cases?
    - `is` and `is not` operators
        - `a == [1, 2, 3]` returns True
        - `b == [1, 2, 3]` returns True
        - `a is b` returns False
        - `a is [1, 2, 3]` also returns False
        - `is` and `is not` operators check for identity, i.e., whether two variables point to the same object stored in memory
        - `==` checks for value, i.e. whether two variables have the same value
        - generally, it's safer and less confusing to use `==` than `is`, but good to know about