collections: tuples, lists, dictionaries

solutions to homework 1

- general
  - printing a result vs. returning a result
    - what is `return` ? A function? No, it's a "keyword", like `if` , `for` , `in` , etc.
    - normally you write `return something` , not `return(something)` . The latter works, but only by accident
    - parentheses are used for other things besides function calls, such as controlling order of operations, which in this case does nothing (there's no operations to order)
    - when defining or calling a function, never leave space between the `functionname` and the opening `(` .
    - Bad: `functionname (arg1, arg2)` . Good: `functionname(arg1, arg2)` .
    - bad example works, but the good example makes it visually obvious that you're defining or calling a function, and not doing something else
    - BTW, what's the difference between defining a function and calling a function?
  - no need for semicolons at the end of each line! although adding one doesn't raise an error, it's a Matlab habit that's best to break free of
  - style: leave one blank line between the end of one function and the start of the next, easier to see where each function ends

1. Write a function called `vowelcount()` that takes a string as an argument, and returns the number of vowels in the string. Test it, e.g. `vowelcount('hEllo')` , `vowelcount('wOrld')` . It should ignore whether the vowels are captial or lowercase.

   - `s.lower()` doesn't affect `s` "in-place", it returns a new string - have to overwrite existing string: `s = s.lower()`
   - use a loop whenever possible to reduce the amount of duplicated code
     - loop over a string of vowels instead of calling `s.count()` separately for each vowel
     - use `lower()` and loop over the vowels once, regardless of case
   - if possible, do calculation/operation (in this case, `.lower()`) once outside of the loop and reuse it, instead of unnecessarily re-calculating on every iteration of the loop
   - loop over a string directly, instead of the indices of the string. Compare:

```
for c in s:
    # do stuff
```

```
for i in range(len(s)):
    c = s[i]
    # do stuff
```

     - this is a very common Matlab habit, and is much harder to read, and more prone to error
   - no real need to check the type of the input. If it's not a string, you'll usually get an error that gives you a hint, e.g. if you give an int instead of a string, you'll get `AttributeError: 'int' has no attribute 'lower()'` when trying to get the lowercase version
   - if you really do want to check the type, it's better to exit ASAP:

```
if type(s) != str:
    print('s is not a string')
```

```
        return
    # do stuff
    return count
```

at the very top, instead of:

```
    if type(s) == str:
        # do stuff
        return count
    else:
        print('s is not a string')
        return
```

The second one forces all the code that actually does stuff to be indented one extra level, unnecessarily complicated!

- what happens when you don't return anything? What's the returned value?
- most people iterated over the input string. What if instead you iterated over a list of all vowels?
- `.count()` is more useful than `.find()` or `.index()` in this case. All you care about is the vowel count, not *where* the vowels happen to be

2. Write a function called `metric()` that takes two numbers `x` and `y`, prints their difference and sum in a single clear message (e.g. `difference is 1, sum is 5`), and returns the difference divided by the sum. Test it, e.g. `metric(2, 3)`, `metric(10, 0.1)`. What happens if the sum is 0? What can you do to handle that case?

    - difference/sum, not sum/difference! The 1st is commonly used (goes from -1 to 1), the 2nd is unbounded and probably meaningless
    - can do `abs()` on the difference, which is fine, but this changes the output metric
    - nice, but not always necessary to assign your result to a variable (e.g. `result`) before returning it, you can leave out the assignment and save a line

3. Write a function called `multtable()` that takes a number `n` and prints out the multiplication table for integers 1 through `n`. Hint: use two `for` loops, each with a different loop variable. Bonus: check the help for `print()` to figure out how to print each row in the table in a single horizontal line.

    - what's a multiplication table?
    - do you need to check with an `if` statement if you've reached the end of the inner loop before printing a newline character?

more string methods

- `s.join()` is useful for joining a bunch of strings together, separating them by the contents of `s`
    - e.g., `','.join(s1, s2, s3)` concatenates three strings together, separating them with `,`

collections

- data types for storing multiple values together under a single name

- choosing the right way to store your data depends on what you want to do with it, and directly affects how efficient and readable your code will be. Choose wisely
- sequences - integer indices only
  - tuples
  - lists
  - numpy arrays (next classsequences)
- mapping - allows non-integer indices, or "key", e.g. strings
  - dictionary
- hybrid of sequence and mapping
  - Pandas `DataFrame` (class 08)

sequences: tuples and lists

- tuples

  > "A tuple is a finite ordered list of elements" -- Wikipedia

  - comes from words like "quadruple, quintuple, etc"
  - denoted by **parentheses** `()` , contain comma separated list of objects
  - can hold anything: integers, floats, strings, booleans, Dogs, Cats, whatever
  - by design, once declared, **cannot** be modified: "immutable"
  - e.g. `t = (1, 2, 3)` or `t = ('a', True, 3.14)`
    - parentheses are often optional: `t = 1, 2, 3`
    - tuple expansion/unpacking allows for multiple assignment:
      - `a, b, c = (1, 2, 3)` or simply `a, b, c = 1, 2, 3`
    - tuples are often used to `return` multiple values from a function

    ```
    def mult123(x):
        return x, 2*x, 3*x

    a, b, c = mult123(2)
    ```

    - note that `return (x, 2*x, 3*x)` works just as well, but takes extra typing, so less common
  - as with strings, get length of a tuple (or any other sequence) with the `len()` function
    - `len(t)` gives 3
  - indexing and slicing of tuples works as it does with strings:
    - `t[0]` gives `1`
    - `t[-1]` gives `3`
    - `t[:2]` gives `(1, 2)`
  - what happens if you try to assign to a particular entry in a tuple?
    - `t[0] = 4` - gives `TypeError` - tuples are immutable!
  - methods:
    - `t.count(val)` returns number of occurrences of val
    - `t.index(val)` returns 0-based index of first occurence of val

- 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 that can modify the list:
  - `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. This is different from string operations, that generally *don't* modify the string, but *do* return something, typically a new string
- typical way to build a list is start with an empty one, use a `for` loop to append stuff to it:

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

- if you just want a list of regularly spaced numbers, use range directly: `l = list(range(10))`
- convert a tuple to a list with `list()`
  - `list((1, 2, 3))`
- convert a list to a tuple with `tuple()`
  - `tuple(l)`
- indexing for lists is the same as for tuples and strings:
  - `l[0]` returns the first index, `l[n-1]` or `l[-1]` returns the last
  - delete entries from a list with `del` keyword by specifying the entry to delete: `del l[2]`
- slicing for lists is the same as for tuples and strings:
  - `l[::3]` gives every 3rd entry in the list, `l[::-3]` gives the reverse

- check contents of tuples and lists using `in`, same as for strings `3 in t` returns `True`, `5 in l` returns `False`

- 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 simple but repetitive, build up a list in a single line of code
    - `doubledlist = [ 2*val for val in sequence ]`

- common functions for use on sequences: `min(), max(), sum(), sorted(), tuple(), list()`

  - `sorted()` also works on strings

sequences exercise:

1. Create a tuple with the following entries: `3, 5, 1.7, -2.7, 1e2, -50`
2. In a single line, make a new tuple that only contains every 2nd entry
3. Convert the original tuple in 1. to a list, assign it a name `l`

4. Sort the list in-place. Prove to yourself that it really is sorted. What happens if you sort it in-place again? What happens if you call `sorted()` on it?
5. Append the value `'blah'` to the list. What do you expect will happen if you try sorting it again? Try it!
6. Remove the `'blah'` from the list, and sort it in reverse order (multiple ways to do this)
7. Now make a new list by doubling the value of each entry in the tuple in 1. First do this using a `for` loop. Then do it again in a single line using list comprehension
8. Convert your code in 7. into a function called `multlist(seq, x)` that takes a sequence (tuple or list) and a multiplication factor `x` and returns a list of `x` times the value of every entry. Ideally, the body of the function should only be a single line

dictionaries

- what if you want to store your values by name, instead of by numerical index?
    - e.g., you have an animal ID that is a mix of letters and numbers
- a "mapping" maps keys (names) to values
- dictionaries are the main mapping object in Python
    - denoted by **curly brackets** `{}` , contain comma separated list of key:value pairs
    - init an empty dictionary with `d = {}` or `d = dict()`
    - init a dict with some predefined key:value pairs:

```
names2ages = {'Alice':25, 'Bob':20, 'Carol':32}
```

    - keys don't have to be strings, they can be int, float, bool, etc. Same goes for values:

```
ages2names = {25:'Alice', 20.5:'Bob', 32:'Carol'}
```

    - add new key:value pairs with `d[key] = value` , e.g. `d['a'] = 1`
        - what happens if a key already exists? Its value is overwritten!
    - access existing key:value pairs with `d[key]`
        - what happens if key doesn't exist in d? `KeyError`
    - remove an existing key:value pair with `del d[key]`
        - what happens if key doesn't exist in d? `KeyError`
    - dictionary methods
        - `list(d)` or `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 of 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
    - iterating over dicts
        - `for key in d:` or `for key in d.keys():`
        - `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! The idea is that a dict is purely a mapping from keys to values, in no particular sequence, unlike a tuple or list. However, as of Python 3.6, order **is** now preserved, but most existing Python code still assumes it doesn't

- combining tuples, lists, dicts, any combination is possible, can be nested as deeply as you want
- common ones:
  - list of tuples: `[(1, 2), (3, 4), (5, 6)]`
  - dict of lists: `{'a':[1, 2, 3], 'b':[4, 5, 6]}`

dictionaries exercise:

1. Describe this nested data structure in words: `[{'a':1, 'b':2}, {'c':3, 'd':4}]`
2. Assign the above structure to the name `d`. Index into `d` to print out only the second dictionary
3. Add a 3rd key:value pair `'e':5` to the second dictionary
4. Delete the key `'a'` from the first dictionary in `d`

Gotcha: compare by reference vs. value

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

1. `a = [1, 2, 3]; b = a`
   - `a` and `b` point to the same object in memory, the list `[1, 2, 3]`
2. `a = [1, 2, 3]; b = a.copy()`
   - `a` and `b` have the same value, but point to different objects in memory that happen to have the same value

- if we set `b[2] = 666`, what's the value of `a` in the above two cases?
- `is` and `is not` operators vs. `==` and `!=`
  - `a = [1, 2, 3]; b = a.copy()`
  - `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

Homework 2 will be due May 8

- no class next week (May 1 holiday), but homework on sequences and dictionaries will be emailed out around then and due before class on May 8