Gold Problem 3: Fun with Functions

Copied from:
https://www.cs.hmc.edu/twiki/bin/view/CS5/FunctionFrenzyGold
on 3/20/2017

[50 points; individual or pair]

This problem asks you to use recursion to write several Python functions. (There are also extra-credit opportunities... .)

Starting your file: `hw1pr3.py`

Create a new plain-text file named `hw1pr3.py`, for example, using Sublime or another plain-text editor.

Here is a little bit of code to paste, if you'd like to use it to get started:

```
# CS5 Gold, hw1pr3
# Filename: hw1pr3.py
# Name:
# Problem description: Function Frenzy!

#
# leng example from class
#
def leng( s ):
    """ leng outputs the length of s
            input: s, which can be a string or list

    """
    if s == '' or s == []:   # if empty string or empty list
        return 0
    else:
        return 1 + leng( s[1:] )
```

In addition, the above code includes the `leng` example function we wrote in class. Here are all of the ClassExamples.

Please put all of your functions for this problem in this **single** `hw1pr3.py` file, and start your file with a comment that includes your name, the starting date, and the assignment/problem name - all good things to have in each of

your source-code files! All of the parts (functions) of this problem will be submitted in that single file.

Be sure to name your functions exactly as specified -- including capitalization!

Use recursion!

For this homework, the `mult`, `dot`, `ind`, `scrabbleScore`, and `transcribe` functions should all be done using *recursion*. Compare them to the `power`, `max`, `leng`, and `vwl` functions we did/saw in class this week. Those examples are linked at this page and beside each problem for you to test and use as the basis for your design.

Visualize recursion!

Some people have used this online Python visualizer to build intuition about how recursion works. A couple details: in order to visualize a recursive call, you'll need to (1) define your recursive function and then (2) make a test call to it, perhaps immediately underneath it.

Here is an example that shows how to use the online Python visualizer to test `mylen( 'cs5' )`, one of the examples from class. Paste this code into the visualizer linked above:

```
def mylen( s ):
    if s == '':
        return 0
    else:
        return 1 + mylen( s[1:] )

test = mylen( 'cs5' )
print('test is', test)
```

You can adapt this for other examples from class or from your own code, as well... . Try it!

Use docstrings!

Also, for each function be sure to include a docstring that indicates what the function's inputs mean and what its output means, i.e., what the function "does." (Omitting a docstring typically results in the function being *"doc'ed"* a couple of points!) Here's an example of a docstring, thorough if a bit verbose, that you are welcome to use for `mult` and as a template for the others:

```
def mult( n, m ):
    """ mult returns the product of its two inputs
            inputs: n and m are both integers
            output: the result upon multiplying n and m
    """
    code here ...
```

Warning! Notice that the docstring needs to be indented to the same level as body of the function it's in. (Python is picky about this...).

Test!

Be sure to test your functions! It's tempting to write a function and feel like it works, but if it hasn't been tested, it may have a small (or big!) error that causes it to fail... .

For this week's assignments, we provide a set of tests that you can (and should!) paste into your code. Then, when you run your file, the tests will run and you can check (by sight, in this case) whether any of the tests has not passed... .

For this week, if your functions pass the provided tests, they will pass all of the graders' tests, too. (In the extra credit and in future assignments, we may add more tests of our own... .)

Here's an example using the `flipside(s)` function from Lab 1. Paste this into your file and run it:

```
def flipside(s):
  """ flipside swaps s's sides
      input s: a string
  """
  x = len(s)//2
  return s[x:] + s[:x]
```

```
#
# Tests
#
print("flipside('carpets')    petscar ==", flipside('carpets'))
print("flipside('homework')  workhome ==", flipside('homework'))
print("flipside('flipside')  sideflip ==", flipside('flipside'))
print("flipside('az')              za ==", flipside('az'))
print("flipside('a')                a ==", flipside('a'))
print("flipside('')                   ==", flipside(''))
```

Be sure to paste the tests for the functions below, too - and run them!

The functions to write...

- **Function 1:** First, write `mult( n, m )`. Here is a full description of how it should work:

  `mult( n, m )` should output the product of the two integers `n` and `m`. Since this would be a bit *too* easy if the multiplication operator * were used, for this function, you are limited to using addition/subtraction/negation operators, along with recursion. (Use the `power` function we did in class as a guide.) Some examples:

```
In [1]: mult( 6, 7 )
Out[1]: 42

In [2]: mult( 6, -3 )
Out[2]: -18
```

  ClassExamples This link contains the recursive `power` function you wrote in class.

  Here are the tests to try:

```
#
# Tests
#
print("mult(6,7)     42 ==", mult(6,7))
print("mult(6,-7)   -42 ==", mult(6,-7))
print("mult(-6,7)   -42 ==", mult(-6,7))
print("mult(-6,-7)   42 ==", mult(-6,-7))
print("mult(6,0)      0 ==", mult(6,0))
print("mult(0,7)      0 ==", mult(0,7))
print("mult(0,0)      0 ==", mult(0,0))
```

- **Function 2:** Next, write `dot( L, K )`. Here is this function's description:

  `dot( L, K )` should output the dot product of the lists `L` and `K`. If these two input lists are not of equal length, `dot` should output `0.0`. If these two lists are both empty, `dot` also should output `0.0`. You should assume that the input lists contain only numeric values. (Compare this with the `mylen` example we did in class, but be sure to account for *both* lists -- and remember they're lists, not strings...! Here is the `leng` example, modified slightly to handle both lists and strings!

  What's the dot product? The dot product of two vectors or lists is the sum of the products of the elements in the same position in the two vectors. for example, the first result is `5*6` plus `3*4`, which is `42`. The result here is `42.0`, because we used a `float` of `0.0` in the base case... .

  You're welcome to use the multiplication operator `*` for this problem, for sure!

```
In [1]: dot( [5,3], [6,4] )
Out[1]: 42.0

In [2]: dot( [1,2,3,4], [10,100,1000,10000] )
Out[2]: 43210.0

In [3]: dot( [5,3], [6] )
Out[3]: 0.0
```

  Here are the tests to try:

```
#
# Tests
#
print("dot( [5,3], [6,4] )      42.0 ==", dot( [5,3], [6,4] ) )
print("dot( [1,2,3,4], [10,100,1000,10000] )  43210.0 ==", dot(
[1,2,3,4], [10,100,1000,10000] ) )
print("dot( [5,3], [6] )         0.0 ==", dot( [5,3], [6] ) )
print("dot( [], [6] )            0.0 ==", dot( [], [6] ))
print("dot( [], [] )             0.0 ==", dot( [], [6] ))
```

- **Function 3:** Next, write `ind( e, L )`. Here is its description:

  Write `ind(e, L)`, which takes in a sequence `L` and an element `e`. `L` might be a string, or L might be a list.

Your function `ind` should return the *index* at which `e` is first found in `L`. Counting begins at 0, as is usual with lists. If `e` is NOT an element of `L`, then `ind(e, L)` should return the integer equal to `len(L)`. You may **not** use the built-in `index` function of Python. Here are a few examples:

```
In [1]: ind(42, [ 55, 77, 42, 12, 42, 100 ])
Out[1]: 2

In [2]: ind(42, list(range(0,100)))
Out[2]: 42

In [3]: ind('hi', [ 'hello', 42, True ])
Out[3]: 3

In [4]: ind('hi', [ 'well', 'hi', 'there' ])
Out[4]: 1

In [5]: ind('i', 'team')
Out[5]: 4

In [6]: ind(' ', 'outer exploration')
Out[6]: 5
```

In this last example, the first input to `ind` is a string of a single space character, *not* the empty string.

Hint: Just as you can check whether an element is in a sequence with

```
if e in L:
```

you can also check whether an element is ***not*** in a sequence with

```
if e not in L:
```

This latter syntax is useful for the `ind` function! As with `dot`, `ind` is probably most similar - bot not identical - to `leng` from the ClassExamples.

Here are the tests to try:

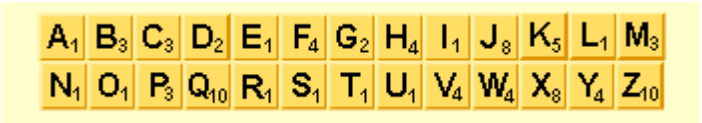
```
#
# Tests
#
print("ind( 42, [ 55, 77, 42, 12, 42, 100 ])  2 ==", ind( 42, [ 55, 77,
42, 12, 42, 100 ]))
```

```
print("ind(42, list(range(0,100)))           42 ==", ind(42,
list(range(0,100))))
print("ind('hi', [ 'hello', 42, True ])       3 ==", ind('hi', [
'hello', 42, True ]))
print("ind('hi', [ 'well', 'hi', 'there' ])   1 ==", ind('hi', [
'well', 'hi', 'there' ]))
print("ind('i', 'team')                       4 ==", ind('i', 'team'))
print("ind(' ', 'outer exploration')          5 ==", ind(' ', 'outer
exploration'))
```

- **Function 4:** Next, write `letterScore( let )`. (Watch for capitalization!) Here is its description:

  `letterScore( let )` should take as input a single-character string and produce as output the value of that character as a scrabble tile. If the input is not one of the letters from `'a'` to `'z'`, the function should return 0.

  To write this function you will need to use this mapping of letters to scores

  What!? Do I have to write 25 or 26 *if elif* or *else* statements? **No!** Instead, use the `in` keyword:

```
In [1]: 'a' in 'this is a string including a'
Out[1]: True

In [2]: 'q' in 'this string does not have the the letter before r'
Out[2]: False
```

  OK! ... but how does this help...?

  Consider a conditional such as this:

```
if let in 'qz':
  return 10
```

  One note: `letterScore` does **not** require recursion. But recursion *is* used in the next one... .

  Here are some examples of `letterScore` in action:

```
In [1]: letterScore('w')
Out[1]: 4

In [2]: letterScore('%')
Out[2]: 0
```

Tests? Write a few tests for this one yourself... it will also be tested in conjunction with the next function!

- **Function 5:** Next, write `scrabbleScore( S )`. (Again, watch for capitalization!) Here is `scrabbleScore`'s description: `scrabbleScore( S )` should take as input a string `S`, which will have only lowercase letters, and should return as output the scrabble score of that string. Ignore the fact that, in reality, the availability of each letter tile is limited. **Hint**: use the above `letterScore` function and recursion. (Compare this with the the `vwl` example we did in class, but consider adding *different* values for each letter. Here are the ClassExamples.

  Here are some examples:

```
In [1]: scrabbleScore('quetzal')
Out[1]: 25

In [2]: scrabbleScore('jonquil')
Out[2]: 23

In [3]: scrabbleScore('syzygy')
Out[3]: 25
```

  Here are the tests to try:

```
#
# Tests
#
print("scrabbleScore('quetzal'):  25 ==", scrabbleScore('quetzal'))
print("scrabbleScore('jonquil'):  23 ==", scrabbleScore('jonquil'))
print("scrabbleScore('syzygy'):   25 ==", scrabbleScore('syzygy'))
print("scrabbleScore('abcdefghijklmnopqrstuvwxyz'):  87 ==",
scrabbleScore('abcdefghijklmnopqrstuvwxyz'))
print("scrabbleScore('?!@#$%^&*()'):  0 ==",
scrabbleScore('?!@#$%^&*()'))
print("scrabbleScore(''):          0 ==", scrabbleScore(''))
```

- **Function 6:** Finally, write `transcribe( S )`. Here is its description: `transcribe( S )`

DNA -> RNA transcription In an incredible molecular feat called *transcription*, your cells create molecules of messenger RNA that mirror the sequence of nucleotides in your DNA. The RNA is then used to create proteins that do the work of the cell.

Write a recursive function `transcribe( S )`, which should take as input a string `S`, which will have DNA nucleotides (capital letter `A`s, `C`s, `G`s, and `T`s).

There may be other characters, too, though they should be ignored by your `transcribe` function by simply disappearing from the output. These might be spaces or other characters that are not really DNA nucleotides.

Then, `transcribe` should return as output the messenger RNA that would be produced from that string `S`. The correct output simply uses replacement:

- `A`s in the input become `U`s in the output.
- `C`s in the input become `G`s in the output.
- `G`s in the input become `C`s in the output.
- `T`s in the input become `A`s in the output.
- any other input characters should disappear from the output altogether

As with the previous problem, you will want a helper function that converts one nucleotide. Feel free to use this as a start for this helper function:

```
def one_dna_to_rna( c ):
    """ converts a single-character c from DNA
        nucleotide to complementary RNA nucleotide """
    if c == 'A': return 'U'
    # you'll need more here...
```

You'll want to adapt the `vwl` example, but adding together *strings*, instead of numbers! Here are the ClassExamples.

Here are some examples of `transcribe`:

```
In [1]: transcribe('ACGT TGCA')  # space should be removed
Out[1]: 'UGCAACGU'

In [2]: transcribe('GATTACA')
Out[2]: 'CUAAUGU'
```

```
In [3]: transcribe('cs5')   # lowercase doesn't count
Out[3]: ''
```

Not quite working? One common problem that can arise is that `one_dna_to_rna` lacks an `else` case to capture all of the non-legal characters. Since all non-nucleotide characters should be dropped, this can be fixed by include code similar to this:

```
    else:
        return ''     # return the empty string if it's not a legal
nucleotide
```

There are different ways around this, too, but this is one problem that has appeared a few times... . Note that the `else` above is only for `one_dna_to_rna`, *not* for `transcribe` itself.

Here are the tests to paste and try - ***note that the right-hand sides won't have quotes***:

```
#
# Tests
#
print("transcribe('ACGTTGCA'):   'UGCAACGU' ==",
transcribe('ACGTTGCA'))
print("transcribe('ACG TGCA'):   'UGCACGU' ==", transcribe('ACGTGCA'))
# note that the space disappears
print("transcribe('GATTACA'):     'CUAAUGU' ==", transcribe('GATTACA'))
print("transcribe('cs5') :               '' ==", transcribe('cs5'))
# note that the other characters disappear
print("transcribe('') :                  '' ==", transcribe(''))
```

Because Python `print`s strings without the enclosing quotes, the right-hand sides won't have quotes, which means that you will want your output to look like this:

```
transcribe('ACGTTGCA'):   'UGCAACGU' == UGCAACGU
transcribe('ACG TGCA'):   'UGCACGU' == UGCACGU
transcribe('GATTACA'):     'CUAAUGU' == CUAAUGU
transcribe('cs5') :               '' ==
transcribe('') :                  '' ==
```

Submit!

Remember to submit your file as `hw1pr3.py`.

If it's not too late, you might check out the optional extra-credit problems, described below.

Extra!

This week's extra-credit shows off a wonderful practice website for Python functions, called !CodingBat. In addition, it offers a challenge "pig-latin" function to write...

There are three opportunities (up to +4 pts each):

- practice with strings on CodingBat with Python strings
- practice with lists on CodingBat with Python lists
- write a pig-latin-izing function (more with strings...)

Extra #1: CodingBat for Python Strings

For +4 points, complete ***all of the Python string problems*** on CodingBat's "String-1" Python string page. Use as many attempts as you'd like. We will use the honor system here:

You'll receive +4 extra-credit points if

- you do complete ***all*** of those string problems successfully, and
- you paste at/near the bottom of your `hw1pr3.py` file this comment:

```
#
# I finished all of the CodingBat STRING problems.
#
```

Extra #2: CodingBat for Python Lists

If you like the CodingBat practice-problem site, try some more! For +4 points, complete ***all of the Python list problems*** on CodingBat's "List-1" Python list page. Use as many attempts as you'd like.

You'll receive +4 extra-credit points if

- you do complete all of those list problems successfully, and
- you paste at/near the bottom of your `hw1pr3.py` file this comment:
- ```
  #
  # I finished all of the CodingBat LIST problems.
  #
  ```

Extra #3: Pig Latin!

[up to another +4 points]

This problem asks you to write two functions that implement an Engligh-to-Pig Latin translator.

Be sure to name and test your functions carefully. Include in each a docstring, which should indicate what the function computes (outputs) and what its inputs are or what they mean.

This problem is inspired by

- **Warm up:**

Write `pigletLatin( s )`, which takes as input a string `s`. `s` will be a single word consisting of lowercase letters. Then, `pigletLatin` should output the translation of `s` to "piglet latin," which has these rules:

- If the input word has no letters at all (the empty string), your function should return the empty string
  ```

- If the input word begins with a vowel, the piglet latin output simply appends the string `'way'` at the end. `'y'` will be considered a consonant, and not a vowel, for this problem.
  Example: `pigletLatin('one')` returns `'oneway'`
- If the input word begins with a consonant, the piglet latin output is identical to the input, except that the input's initial consonant is at the end of the word instead of the beginning and it's followed by the string `'ay'`.
  Example: `pigletLatin('be')` returns `'ebay'`
- Of course, this is not full pig latin, because it does not handle words beginning with multiple consonants correctly. For example, `pigletLatin('string')` returns `'tringsay'`.
  You'll fix this next!

- **The real pig latin challenge**:

Create a function called `pigLatin( s )` that handles the rules above ***and*** hadles more than one initial consonant correctly in the translation to Pig Latin. That is, `pigLatin` moves *all* of the initial consonants to the end of the word before adding `'ay'`. (You may want to write and use a helper function to do this -- see the hint below.)

Also, `pigLatin` should handle an initial `'y'` either as a consonant **OR** as a vowel, depending on whether the `y` is followed by a vowel or consonant, respectively. For example, `'yes'` has an initial `y` acting as a consonant. The word `'yttrium'`, however, (element #39) has an initial `y` acting as a vowel. Here are some additional examples:

```
In [1]: pigLatin('string')
Out[1]: ingstray

In [2]: pigLatin('yttrium')
Out[2]: yttriumway

In [3]: pigLatin('yoohoo')
Out[3]: oohooyay
```

Tests? These we're leaving up to you!

Hint!

One way to use recursion to assist in this is to write a function

```
def initial_consonants( s ):
```

that returns a string of all of the initial consonants in the input string `s`. Thus, if `s` starts with a vowel, the empty string `''` will be returned.

If you think about this problem thoroughly, you'll find that not every possible case has been

accounted for - you are free to decide the appropriate course of action for those "corner cases." We won't test these... .

Oodgay ucklay!