Problem 1.5: Fun with Functions

This problem asks you to use recursion to write several Python functions.

When you're finished with this assignment, submit your code at the bottom of this page.

Start by making a copy of the trinket below. It includes the `mylen` example function we wrote in class.

In addition, you can find the rest of the class examples in this trinket.

Please put all of your functions for this problem in this **single** trinket, and start your trinket 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`, `mymax`, `mylen`, and `sajak` functions we did in class this week.

Those examples are linked in this trinket 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 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!

Functions to Write

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 the addition, subtraction, and negation operators, along with recursion. (Use the `power` function we did in class as a guide.) Some examples:

   ```
   >>> mult( 6, 7 )
   42

   >>> mult( 6, -3
   )
   -18
   ```

   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)
   ```

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!)

   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 below 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!

```
>>> dot( [5,3], [6,4] )
42.0

>>> dot( [1,2,3,4], [10,100,1000,10000]
)
43210.0

>>> dot( [5,3], [6] )
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( [], [] )
```

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, more generally, 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)`. Here are a few examples:

```
>>> ind(42, [ 55, 77, 42, 12, 42, 100
])
2
>>> ind(42, range(0,100))
42
>>> ind('hi', [ 'hello', 42, True ])
3
>>> ind('hi', [ 'well', 'hi', 'there'
])
1
>>> ind('i', 'team')
4
>>> ind(' ', 'outer exploration')
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 to `mylen` from the class examples.

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, range(0,100))                  42 ==", ind(42, 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')
```

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

`letterScore( let )` should take as input 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:

A - 1	B - 3	C - 3	D - 2	E - 1	F - 4	G - 2	H - 4	I - 1	J - 8	K - 5	L - 1	M - 3
N - 1	O - 1	P - 3	Q - 10	R - 1	S - 1	T - 1	U - 1	V - 4	W - 4	X - 8	Y - 4	Z - 10

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

```
>>> 'a' in 'this is a string including a'
True

>>> 'q' in 'this string does not have the the letter before
r'
False
```

Okay! ... 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:

```
>>>
letterScore('w')
4
>>>
letterScore('%')
0
```

Tests? Try this one yourself... it will be more formally tested in conjunction with the next 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 `sajak` example we did in class, but consider adding *different* values for each letter.)

Here are some examples:

```
>>>
scrabbleScore('quetzal')
25
>>>
scrabbleScore('jonquil')
23
>>> scrabbleScore('syzygy')
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('')
```

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

   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 will be ignored by your `transcribe` function—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 above, 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 `sajak` example, but adding together *strings*, instead of numbers!

   Here are some examples of `transcribe`:

```
>>> transcribe('ACGT TGCA')
'UGCAACGU'
>>> transcribe('GATTACA')
'CUAAUGU'
>>> transcribe('cs5')    # lowercase doesn't
count
''
```

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 sides won't have quotes:

```
#
# Tests
#
print "transcribe('ACGT TGCA'):  'UGCAACGU' ==", transcribe('ACGT
TGCA')
print "transcribe('GATTACA'):     'CUAAUGU' ==", transcribe('GATTACA')
print "transcribe('cs5') :              '' ==", transcribe('cs5')
print "transcribe('') :                 '' ==", transcribe('')
```

Submit Homework 3, Problem 1.5

50.0/50.0 points (graded)

To submit Homework 3, Problem 1.5, you'll need to copy your code from your trinket and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

```
def mylen( s
):
```

15

```
    """ mylen outputs the length of
s
16
                input: s, which can be a string or
list
17
18
"""
19
    if s == '' or s == []:
20
        return
0
21
else:
22
        return 1 + mylen( s[1:]
)
23
24
```

```
def mult(n,
m):
```

25

```
    """ mult returns the product of its two
inputs
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.