CS 1: Introduction To Computer Programming, Fall 2015

Assignment 2: The Lizard-Spock Expansion

Due: *Thursday, October 22, 02:00:00*

Coverage

This assignment covers the material up to lecture 7 (as well as the `range` function covered in lecture 8).

What to hand in

You will be handing in two files for this assignment.

You should collect the answers to sections B and C into a file called `lab2a.py`. For the questions in section C, you should write your answers in comments, but write the corrected code outside of comments.

For section D, you should hand in the file `lab2b.py` containing your Mastermind code. You should **not** hand in the `lab2b_tests.py` file.

Both files should be handed in to csman as usual.

Part A: Installing the "nose" package

For this section, we're going to assume that you already installed Python 2.7.10 and WingIDE last week. There are still several more Python packages that we will be using in this course, and so Part A of the labs will cover installation of these packages. Naturally, you will not be graded on this section.

For this assignment, we will install the `nose` testing framework. This will provide Python modules which will make it easier to test the code that you write. In particular, this will make it easier for you (and your TA!) to figure out if your code is correct or not. For this assignment, and for many of the subsequent assignments, we will provide the testing code for you. Later we will get you to write some of your own tests.

The `nose` framework is already installed on the CMS cluster, so if you can't be bothered doing this you can always run your programs there. The `nose` documentation is here, but you don't need to read it (it will probably be more confusing than helpful to you at this stage). We will only be using the most basic features of `nose` for the next few assignments. Note that the documentation link is for the most recent version of `nose`, which may not be the same as the version you have; it shouldn't make any difference.

The rest of the instructions depend on which operating system you're running on your computer. We will cover Windows, Mac OS X, and Linux; if you're running some other operating system, you're on your own.

The easiest way to install nose is using a Python installation system called `easy_install`. We'll do that here.

Windows

Step 1

Install `setuptools`. This will install `easy_install` for you. The `setuptools` installer can be found here. Click on this, follow the instructions, and select all the default values to install the software. If you're running Windows Vista, the operating system will ask you several times if you trust the software enough to install it; reassure it that you do.

Step 2

Run the DOS Command Prompt (under Accessories -> Command Prompt). This is something like a terminal program on Unix. At the prompt, type:

```
C:\Python27\Scripts\easy_install nose
```

This will install `nose`. (For this step to work, you must be connected to the internet.)

Mac OS X

Step 1

Install `setuptools` by going to this web site and following the instructions. Most likely, you will be doing something like the following. Start a terminal (*e.g.* `Terminal.app`) and at the prompt, type:

```
% sudo curl https://bootstrap.pypa.io/ez_setup.py -o - | python
```

You may need to type in your password if you use `sudo`.

If this doesn't work, enable the root user by following these directions, and then type:

```
% su
[enter the root password]
# curl https://bootstrap.pypa.io/ez_setup.py -o - | python
# exit
```

Note that the prompt changes to `#` when you enter the root password. **Do not do anything else as root!** Careless use of the root account can severely damage your computer.

Step 2

In a terminal, type:

```
% easy_install nose
```

This will install `nose`. (For this step to work, you must be connected to the internet.) If this doesn't work and you get an error message containing the words `permission denied`, try this:

```
% sudo easy_install nose
```

Linux

The instructions for installing nose on Linux are exactly the same as for installing nose on Mac OS X.

Checking that it installed correctly

Start up WingIDE, and in the Python Shell type:

```
>>> import nose
```

If it gives you the prompt again without reporting any errors, you've installed `nose`. If not, ask your TA for help. To see which version of `nose` you're running, type:

```
>>> print nose.__version__
```

Anything above version 0.11 should be fine.

Part B: Exercises

Note: In many of the exercises below, we want you to not only solve the problem, but to solve the problem in a particular way. So if we say to solve a problem using a particular function or without doing something (for instance) we want it done that way. Alternative solutions, even correct ones, are not acceptable.

Make sure you write a docstring for every function we ask you to write. It should describe what the function does, what the arguments represent, and what the return value represents.

1. [**15**] Write a function which takes as its only argument a string composed of only the letters 'A', 'C', 'G', and 'T' (representing a DNA string). This function will return another string composed of the same four letters, representing the *complement* of the original string. This means that the result string will have an 'A' where the original string has a 'T', a 'C' where the original string has a 'G', a 'G' where the original string has a 'C', and a 'T' where the original string has an 'A'.

   Examples:

   ```
   >>> complement('ACGGATC')
   'TGCCTAG'
   ```

   Use string operations only (don't convert the string to a list). You may find the `+=` operation on strings to be useful. Note that you can write a `for` loop to loop over the characters in a string, as described in class.

2. [**15**] Now assume that we're representing DNA using lists of individual base pairs, represented as the characters 'A', 'C', 'G', and 'T'. So a DNA strand might be *e.g.* something like `['A', 'A', 'C', 'G', 'T', 'G', 'T']`. Write a function that takes as its only argument a list of this type, and changes the list so that all the 'A's become 'T's, the 'T's become 'A's, the 'C's become 'G's, and the 'G's become 'C's. In this function you will be changing the list that is passed as an argument to the function, so you will not need a `return` statement (and there shouldn't be one).

   Examples:

   ```
   >>> dna = ['A', 'C', 'G', 'G', 'A', 'T', 'C']
   >>> list_complement(dna)
   >>> dna
   ['T', 'G', 'C', 'C', 'T', 'A', 'G']
   ```

3. [**15**] You know that you can sum a list of numbers using the built-in `sum` function. Write a function called `product` which will take one argument, which will be a list of numbers. It will compute the product of the entire list and return it. If the list is empty, it should return 1.

   Examples:

```
>>> product([])
1
>>> product([2, 2, 3, 3, 10])
360
```

4. [**15**] The factorial of a non-negative integer is defined to be the product of all the integers from 1 up to the integer. As a special case, the factorial of 0 is defined to be 1. Use the `range` function and the `product` function you just defined to define a function called `factorial` which takes a single argument (a non-negative integer) and returns the factorial of that integer.

   Examples:

   ```
   >>> factorial(0)
   1
   >>> factorial(5)  # 5 * 4 * 3 * 2 * 1
   120
   ```

5. [**15**] You are writing a computer game which simulates a game of Dungeons and Dragons. You need a way to simulate the rolling of dice, which can have different numbers of sides (regular six-sided dice, four-sided dice, 10-sided dice, 20-sided dice, etc.). Write a function `dice(m, n)` which takes two arguments: the "sidedness" of the dice (`m`) and the number of dice rolled (`n`). This function will use the `random.choice()` function (so you will need to put the line `import random` in your file before using it, ideally at the top) and the `range()` function to compute the sum of all the random dice values.

   Examples:

   ```
   >>> dice(4, 1)
   3
   >>> dice(4, 2)
   5
   >>> dice(4, 3)
   8
   >>> dice(6, 2)
   11
   >>> dice(6, 3)
   14
   >>> dice(6, 6)
   22
   >>> dice(20, 1)
   13
   >>> dice(20, 3)
   42
   ```

6. [**20**] The `remove` method of lists is useful, but it has the limitation that it will only remove the first matching element it finds in a list. Write a function called `remove_all` which takes two arguments: a list (which you can assume to be a list of integers) and a value (which you can assume to be an integer). It will remove *all* copies of the value from the list. It will do this by using a `while` loop and the `count` method on lists to determine if there are any more matching elements in the list. Each time through the list, the `remove` method will be called to remove one more matching element from the list. The function won't return anything, because the list will be changed in place.

   Examples:

   ```
   >>> lst = [1, 1, 3, 1, 4, 1, 5, 1, 6, 1, 1, 7, 1]
   >>> remove_all(lst, 1)
   >>> lst
   [3, 4, 5, 6, 7]
   ```

7. [**20**] The use of the `count` method in the previous problem is inefficient because you have to count all of the occurrences of a particular value in a list just to see if there are any values to be removed. Once you

know that there is at least one value to be removed, you could stop counting, but the `count` method will go ahead and look at every value in the list to see if it matches the value being counted, even if that value has already been encountered earlier in the list. There are two obvious ways to fix this:

1. Count the number of values to be removed once, and then use the `remove` method that many times.
2. Use some other way to find out if a value is in a list, and remove elements until there are no more elements.

The first way is straightforward; let's talk about the second way. You've learned the Python syntax `for <item> in <list>` in class. But as we've mentioned several times, Python likes to overload its syntax to mean different things. The `in` keyword, in particular, can be used not just as part of a `for` loop but also to test for membership in a list. For instance:

```
>>> 4 in [1, 2, 3, 4, 5]
True
>>> 'foo' in ['foo', 'bar', 'baz']
True
>>> 'cat' in ['fluffy', 'morris', 'sylvester']
False
```

In each case, the interpretation is the same: is this item in this list? When we use the `in` keyword in this way we often refer to it as the "`in` operator". The nice thing about the `in` operator is that once it finds that a particular is in the list, it stops looking.

Given what you now know, write two efficient versions of `remove_all`, called `remove_all2` and `remove_all3`, which implement the two new strategies for removing all copies of a particular value from a list. Use a `for` loop for `remove_all2` (the function which counts the number of repetitions first), and a `while` loop for `remove_all3` (the function which uses the `in` operator). The way these functions are used is exactly the same as for `remove_all`; only the implementation is different. This is a common theme in programming: rewriting an inefficient function into a more efficient form without changing the interface to the function (the inputs or the outputs).

8. [**20**] Let's use the `in` operator described in the previous problem to write a function called `any_in`. It will take two lists as arguments, and will return `True` if any of the elements of the first list are equal to any of the elements of the second one.

   Examples:

   ```
   >>> any_in([1, 2, 3], [1, 5, 9, 11, 43])
   True
   >>> any_in([1, 2, 3], [])
   False
   >>> any_in([], [1, 2, 3])
   False
   >>> any_in([], [])
   False
   >>> any_in(['foo', 'bar', 'baz'], ['to', 'be', 'or', 'not', 'to', 'be'])
   False
   >>> any_in(['and', 'or', 'not'], ['to', 'be', 'or', 'not', 'to', 'be'])
   True
   ```

   The `any_in` function can be written using only a few lines of code, so if your function is long, you're probably doing something wrong (or at least, doing something inefficiently).

   Hint: It's legal to `return` from a function before the end of the function (say, when some condition is met).

Part C: Pitfalls

In this section we're going to review some aspects of Python that can confuse new programmers (and sometimes also experienced programmers). We encourage you to experiment with these examples in WingIDE to get a better understanding of what's going on.

1. [**30**] What is wrong with the following snippets of Python code? Write a sentence or two describing what the problem is, why it's incorrect, and how to fix it.

   a.
   ```
   a = 20
   # ... later in the program, test to see if a has become 0.
   if a = 0:
         print 'a is zero!'
   ```

   b.
   ```
   def add_suffix('s'):
       '''This function adds the suffix '-Caltech' to the string s.'''
       return s + '-Caltech'
   ```

   c.
   ```
   def add_suffix(s):
       '''This function adds the suffix '-Caltech' to the string s.'''
       return 's' + '-Caltech'
   ```

   d.
   ```
   # We want to add the string 'bam' to a list of strings, changing the original
   # list.
   lst = ['foo', 'bar', 'baz']
   lst = lst + 'bam'
   ```

   e.
   ```
   def reverse_and_append_zero(lst):
       '''This function reverses a list of numbers and then
       appends the number 0 to the end of the list.'''
       lst2 = lst.reverse()
       return lst2.append(0)
   ```

   f.
   ```
   def append_string_letters_to_list(list, str):
       '''This function converts a string 'str' to a list and appends
       the letters of the string to the list 'list'.'''
       letters = list(str)
       list.append(letters)
   ```

2. [**10**] Write a sentence or two explaining why the following code:

   ```
   a = 10
   b = 20
   c = b + a
   a = 30
   print c
   ```

   prints `30` and not `50`.

3. [**20**] Consider the following two functions:

   ```
   def add_and_double_1(x, y, z):
       result = 2 * (x + y + z)
       return result

   def add_and_double_2(x, y, z):
       result = 2 * (x + y + z)
       print result
   ```

   Now let's say that you wanted to use the result of these functions somewhere else in your program. Explain why this would work:

```
n = 2 * add_and_double_1(1, 2, 3)
```

and why this would not work:

```
n = 2 * add_and_double_2(1, 2, 3)
```

Write a sentence or two explaining the difference between printing a result and returning a result from a function.

4. [**20**] Consider the following two functions:

```
def sum_of_squares_1(x, y):
    result = x * x + y * y
    return result

def sum_of_squares_2():
    x = int(raw_input('Enter x: '))
    y = int(raw_input('Enter y: '))
    result = x * x + y * y
    return result
```

Now let's say that you wanted to use the result of these functions somewhere else in your program. Explain why this would work:

```
n = 2 * sum_of_squares_1(2, 3)
```

and why this would not work:

```
n = 2 * sum_of_squares_2(2, 3)
```

Write a sentence or two explaining the difference between passing a value as an argument to a function and getting it interactively using `raw_input`.

5. [**10**] Why won't this function work?

```
def capitalize(s):
    '''This function capitalizes the first letter of the string 's'.'''
    s[0] = s[0].upper()
```

6. [**10**] Why won't this function work? **Warning:** This is a bit tricky.

```
def double_list(lst):
    '''This function doubles each element in a list in-place.'''
    for item in lst:
        item *= 2
```

Part D: Miniproject: The game of Mastermind

Mastermind is a simple board game for two players. The rules are listed here, but here is a quick summary. One player (the "codemaker") picks a secret code consisting of four *code pegs* which can be any of six colors (we'll use the colors red, green, blue, yellow, orange, and white, which we'll abbreviate as 'R', 'G', 'B', 'Y', 'O', and 'W'). The order of the pegs is important, so a code of "RGBB" is not the same as a code of "BRGB". Colors can be duplicated in a code, but they don't have to be. The codemaker puts the four pegs representing his/her code in a secret location where the other player can't see them. That player (the "codebreaker") tries to guess the code by laying down four code pegs that he/she thinks might be the correct code. The codemaker compares the guess with the real code, and puts down zero to four *key pegs* which indicate how close the guess is to the solution. For every code peg which is the correct color in the correct location, a black key peg is put down. For every code

peg which is a correct color but not in a correct location, a white peg is put down. Victory occurs when four black pegs are put down, which means that all the code pegs in the guess are the same as those in the secret code. The objective is to guess the code in as few moves as possible.

Here's an example of a sample game. Let's say that the secret code was 'ROOB' (Red, Orange, Orange, Blue). We'll show the code pegs on the left and the key pegs on the right (as black ('b'), white ('w'), or blank ('-')).

```
Guess    Result
-----    ------

RGBY     bw--   (R is in the correct location, B is not, other colors wrong)
OOWW     bw--   (One O is in the right location, one isn't, other colors wrong)
RBOR     bbw-
RBBW     bw--
RBOO     bbww   (All colors correct, but not in the right order)
ROOB     bbbb   (All colors correct and in the right order; victory!)
```

Your job will be to write a computer program to play Mastermind against you. The computer will be the codemaker and will choose a random code. You will be the codebreaker. The computer has to take your guesses and report how well you are doing. An interaction with the computer will look like this:

```
New game.
Enter your guess: RGBW
    Result: bw--
Enter your guess: OOWW
    Result: bw--
Enter your guess: RBOR
    Result: bbw-
Enter your guess: RBBW
    Result: bw--
Enter your guess: RBOO
    Result: bbww
Enter your guess: ROOB
    Result: bbbb
Congratulations!  You cracked the code in 6 moves!
```

We will build up to this in stages. All of the code for this miniproject should be in a file called `lab2b.py`. All of your functions should have docstrings in which you state what the function does, what the arguments mean, and what the return value represents. Import whatever modules you need to solve the problems.

None of these functions need to be very long, so if you find yourself writing a lot of code for any function (say, more than 20 lines), you are probably going about the task the wrong way and need to speak to a TA.

1. [**15**] Write a function called `make_random_code` which takes no arguments and returns a string of exactly four characters, each of which should be one of 'R', 'G', 'B', 'Y', 'O', or 'W'. You may find the `random.choice` function very useful for this problem. Do:

   ```
   >>> help(random.choice)
   ```

   from inside Python to learn more about it. (Make sure you `import` the `random` module first!)

2. [**15**] Write a function called `count_exact_matches` which takes two arguments which are both strings of length 4 (you don't have to check this). It returns the number of places where the two strings have the exact same letters at the exact same locations. So if the two strings are identical, it would return 4; if the first and third letters of both strings are the same but the other two are different, it would return 2; and if none of the letters are the same at corresponding locations it would return 0.

   Examples:

```
count_exact_matches('ABCD', 'ABCD')  # --> 4
count_exact_matches('ABCD', 'EFGH')  # --> 0
count_exact_matches('ABCD', 'BCDA')  # --> 0  (not in the same locations in string)
count_exact_matches('AXBX', 'AABB')  # --> 2
```

Use a `for` loop, which should iterate over the indices of the string (*i.e.* the list `[0, 1, 2, 3]`). Recall that you can access letters of a string using the same syntax you use to access elements of a list (because they're both sequences).

3. [**30**] Write a function called `count_letter_matches` which is like `count_exact_matches` except that it doesn't care about what order the letters are in; it returns the number of letters of the two strings that are the same regardless of order.

   Examples:

   ```
   count_letter_matches('ABCD', 'ABCD')  # --> 4
   count_letter_matches('ABCD', 'EFGH')  # --> 0
   count_letter_matches('ABCD', 'BCDA')  # --> 4  (the order doesn't matter)
   count_letter_matches('AXBX', 'AABB')  # --> 2
   ```

   This function is a bit tricky, so here are some hints. Of course, you don't have to follow our hints; you can solve this problem any way you like as long as the solution isn't ridiculously long or convoluted.

   1. Convert the two strings into lists using the `list` built-in function before doing anything else.

   2. Go through one list character by character using a `for` loop. For each character that is in the other list, increment a counter variable and then remove the character from the second list (not the list you are iterating over) using the `remove` method on lists. Use the form `<x> in <list>` (see section B) to find out if a character is in a list.

   3. Once you have gone through the list, the counter variable is the answer.

4. [**30**] Write a function called `compare_codes` which takes two arguments which are both strings of length 4. The first argument is called `code` and will represent the secret code chosen by the codemaker. The second argument is called `guess` and will represent the guess of the codebreaker. The function will output a string of length four consisting only of the characters `'b'`, `'w'`, and `'-'` (for black, white, and blank key pegs). This represents the key pegs *i.e.* the evaluation of the guess (so `'bbbb'` would be a perfect guess, `'wwww'` would mean all colors are correct but in the wrong order, and `'----'` would mean that no colors are correct).

   This function is the heart of the game. Fortunately, if you've written the two previous functions correctly, writing this one is a piece of cake. Here is the algorithm (solution method) you should use:

   1. The count of black pegs can be obtained just by calling the function `count_exact_matches`.

   2. The count of white pegs can be obtained by calling both the functions `count_letter_matches` and `count_exact_matches` and subtracting the result of the second function call from the first.

   3. The count of blank pegs is the difference between 4 and the sum of the count of black and white pegs.

   You will need to take these three counts and create a string for the result. Make sure that the string includes the `'b'`s first, then the `'w'`s, then the `'-'`s.

   Note that this function doesn't print anything; it just returns a string.

   Examples:

```
compare_codes('ABCD', 'ABCD')  # --> 'bbbb'
compare_codes('ABCD', 'EFGH')  # --> '----'
compare_codes('ABCD', 'BCDA')  # --> 'wwww'
compare_codes('AXBX', 'AABB')  # --> 'bb--'
compare_codes('ABDC', 'ABCD')  # --> 'bbww'
```

5. [**30**] Now we're ready to write the function that actually runs the game. It will be called `run_game`, and it will take no arguments. When called, this function will do the following:

    1. Print "`New game.`".
    2. Select a secret code using the `make_random_code` function you defined above, and store this in a variable.
    3. Prompt the user for a guess using the `raw_input` function and the prompt string "`Enter your guess:` ".
    4. Evaluate how well the guess matches the stored code using the `compare_codes` function and print out the result in the format "`Result: <result string>`".
    5. If the result string is "`bbbb`", then the function will print "`Congratulations! You cracked the code in <N> moves!`" and exit (where `<N>` is the number of moves since the game began). Otherwise, the function will go back to step 3 above.

    In other words, the interaction with the user should be the same as what was described above at the beginning of this section. Here are a few hints:

    1. Every time the function asks for a guess, a counter which holds the number of moves should be incremented.
    2. Consider using an infinite loop (with a `while` statement) and a `break` statement inside the loop once the exit condition applies.

6. [**15**] To make it easier for you to decide if your code is working correctly, we have written a "test suite" which will automatically test most of the above functions (all but the last one, to be exact). This test suite is in a Python file called `lab2b_tests.py` and can be downloaded here. Once you've downloaded it, you should put it in the same directory as your `lab2b.py` file and load it into WingIDE. Then hit the "Run" button. If all is well it will report that it ran 4 tests and it will print "OK". If there is a test failure, it will report that as well and indicate which function failed. If that happens, you have some bugs to fix!

    Note that even though there is nothing to hand in for this problem, we still require that you do it. Good programmers get into the habit of testing their code thoroughly every time they write code. Another way to say it is that code that has not been tested is wrong by definition.
