CISC106H - Fall 2017
Lab 3, due Sunday 10/1

1. Write a function **worker** that multiplies its single numeric parameter by two and returns the product. Then write **manager** that takes two parameters, a function and a number, and simply returns the value of the function parameter called on the number parameter. Call **manager** on **worker**. Play with this example and ask your TA questions.

Problems for submission:

2. Here is a fun way to practice writing functions for the exam! Use the template provided to write a function, chomper, that uses multiple simple helper functions to achieve its high level goals of moving and eating, while the helper functions process a string world.

   The purpose of this function is to mimic the actions of a simple cybernetic animal that lives in a one-dimensional world of zeros and ones[1]. Ones represent things to eat, and the animal seeks them out and eats them until none are left. An interaction with chomper will look like this:

```
>>> chomper('1 0 1 0 V 1 0 1 0')
1 0 1 0 V 1 0 1 0
1 0 1 0 > 1 0 1 0
1 0 1 > 0 1 0 1 0
1 0 > 0 0 1 0 1 0
1 > 0 0 0 1 0 1 0
> 0 0 0 0 1 0 1 0
V 0 0 0 0 1 0 1 0
< 0 0 0 0 1 0 1 0
0 < 0 0 0 1 0 1 0
0 0 < 0 0 1 0 1 0
0 0 0 < 0 1 0 1 0
0 0 0 0 < 1 0 1 0
0 0 0 0 0 < 0 1 0
0 0 0 0 0 0 < 1 0
0 0 0 0 0 0 0 < 0
0 0 0 0 0 0 0 V 0
All Done!
>>> chomper("0 0 V 0")
0 0 V 0
All Done!
>>>
```

   Let's notice a couple of things about the Chomper. First, Chomper itself is represented in one of three ways, depending on its current state. If Chomper is not currently moving in a direction, it is represented as a V. If it is moving left, it is represented as a >, and if it is moving right, a <. Thus the string given to the chomper function consists of a series of zeros and ones and one instance of Chomper in one of its three states.

[1]This problem adapted with permission from Kathy McCoy.

Suppose Chomper is in its V state. If there is any food to its left (i.e., there are any ones to its left) then it will switch to a state in which it is moving left. If there is no food to its left (i.e., there are NO ones to its left) but there is food to its right, it will switch to a state in which it is moving right. If there is no food left for it to consume (when it is in its V state), it will print out a message.

Once Chomper is started in one direction or another, it will continue in that direction as long as there is any food in that direction at all. When food runs out in that direction, Chomper will not move but will switch into its V state. As long as there is food in a particular direction, Chomper will move one place in that direction, converting ones into zeros as they are met. The state of Chomper is printed out with each change to its state.

The highest level function, chomper, will return the first state in which there is no food and Chomper is resting.

All functions for this problem must be written with tools from class, e.g. recursion, if. Any use of for, while, in, dot functions, mutators, etc, will result in large penalties. Ask well in advance if you are not sure.

Here are the functions to write, and a small, insufficient number of tests:

```
def look_left(world):
    """
    Returns True if there are any 1s found in list before V is found in world.
    """

def find_chomper(world):
    """
    Returns the index of chomper in parameter list. May have a helper.
    """

def look_right(world):
    """
    Returns true if there are any 1s to right of chomper in world.
    """

def replace(old, new, world):
    """
    Returns world string with first instance of old replaced with new.
    Assumes one space between all other elements.
    Precondition: old is in world.
    """

def chomper(world):
    """
    Looks to one direction, if it sees food it
    calls helper with appropriate state. Then checks other
    direction. If no food is found, the program ends.
    """

def chomp_helper(world, state):
    """
    Based on chomper state (< or >), looks in one direction for food,
    calls itself on world after one move to left or right. If no food
    is seen in that direction, calls chomper.
    """

def move_one_place_left(world):
    """ Precondition: there is food to left of chomper. """

def move_one_place_right(world):
    """ Precondition: there is food to right of chomper. """

def test():
    """
    The tests are not sufficient. Identify boundary and other untested
    conditions and write your own tests to augment these.
    """

    assertEqual( move_one_place_left('1 >'), '> 0')
    assertEqual( move_one_place_left('1 0 >'), '1 > 0')
    assertEqual( move_one_place_left('1 > 0'), '> 0 0')

    assertEqual( move_one_place_right('< 1'), '0 <')
    assertEqual( move_one_place_right('< 0 0 0'), '0 < 0 0')

    assertEqual(replace('V','>','V'), '>')
    assertEqual(replace('V','>','V 0 0'), '> 0 0')

    assertEqual(look_right('V 1'), True)
    assertEqual(look_right('V'), False)
    assertEqual(look_right('V 0 0 1'), True)

    assertEqual(find_chomper('V'), 0)
    assertEqual(find_chomper('>'), 0)
    assertEqual(find_chomper('<'), 0)
    assertEqual(find_chomper('0 0 V'), 4)

    assertEqual(look_left('1 0'), True)
    assertEqual(look_left('0 1'), True)
    assertEqual(look_left('V'), False)
    assertEqual(look_left('0 0 0 0 0 V'), False)
    assertEqual(look_left('> 0 0 0 0 0 1'), False)

    chomper('0 0 0 V 0 0 0')
    chomper('1 0 0 V 0 0 0')
    chomper('0 0 1 V 1 0 0 0')
    chomper('1 1 V 1 1')
```

```
test()
```

3. Write a function **sum** of two parameters that will work with accumulate to sum a list. Note that all list operations are performed in accumulate, not the helper you are writing. Write three good tests for sum and three good tests for accumulate using your sum.

4. As in problem 3, but for a helper **mult_five** so that accumulate can multiply all elements in the list by five.

5. As in problem 3, but for a helper **reverser** so that accumulate can reverse a list.

6. As in problem 3, but for a helper **min** so that accumulate can find the min value in a list. Note: for this accumulate call, pass in the first elt of the list as your init parameter (why is this important?).

Post-exam problems:

7. Write a function **sum_n(n)** that uses a for loop and a range to compute the sum of all numbers 1-n, inclusive.

8. Write a function **evens_n** that uses mod (%) and a for loop to collect all the even numbers from 0-n, inclusive, in a list. Use range with only one parameter. (How could you achieve this more easily using the range function?)

9. Write a function **contains(key, alist)** that uses a for loop to return True if key is in the list, and False otherwise. Show it working with strings and numbers.

10. Write a function **memory(n, m, alist)** that replaces a list element with m if either of the preceding two elements was originally n, e.g.

    memory(5, 7, [5,5,6,3,5,2,3,4,5]) $\rightarrow$ [5,7,7,7,5,7,7,4,5]

Assignment Submission:

Make sure both names of your programming pair are in each file.

Have your python functions in a single .py file, with all the relevant assertEqual tests in the same file.

Submit on Canvas: One python file containing code and tests. Follow the TA's instructions if they vary from this document.