About Lab 8

Lab 8 asks you to write 3 programs.

The first program is a recursion that iterates through the numbers of a sequence. Your job is to count the number of elements in this sequence, which is the same as the number of times this function is called.

The secret to this is to make the function return the number of times it is called. To do this with a different function, consider this:

```
def f(n):
    print(n, end=" " )
    if n > 0:
        f(n//2)
```

If we call f(10) this will print 10 5 2 1 0

```
def f(n):
        print(n, end=" " )
        if n > 0:
                f(n//2)
```

Question: if f(n) should return the number of times f is called, what is the recursion relating f(n) and f(n//2)??

A. f(n) = 2*f(n//2)
B. f(n) = f(n//2)
C. f(n) = 1+ f(n//2)
D. f(n) = 2+f(n//2)

The second part of the lab involves writing a translator for Morse code. I give you a file of code, you need to translate it into standard characters of the alphabet. Here is a sample of code

```
-...  ---  -...   -.-.  .-  -
 B    O    B     C    A   T
```

Each letter is represented by a sequence of dots and dashes. The letters are separated by spaces, the words are separated by two spaces.

The idea is to break the file into lines (we can read it with a for-loop for that), then break the line into words, then break each word into its individual character codes. We then just need to look the representation of each letter up in a table of all of the Morse codes.

Our words are separated by two spaces. We can use the split( ) method for strings to split the line into individual words.

For example, if s is the string "Here comes the sun." then s.split(" ") is the list

["Here", "comes", "the", "sun."]

(It is hard to see in print, but the words in s are separated by 2 spaces and the argument to split( ) is a string with 2 spaces.)

Similarly, if t is the string

"-... --- -..."

then t.split(" ") is the list ["-...", "---", "-..."]

Our Morse Code table is formatted

A .-

B -...

C -.-.

D -..

There is a space after each letter, before the code.

We need to read this into a structure that is easy to search. One way to do this is by reading the file one line at a time. Strip the end-of-line marker off the line (line=line.strip()), then say

Table.append(line.split(" "))

where Table is a list.

Question: Suppose variable line is the string

"C -.-."

and we say Table.append(line.split(" ")). What does this entry of Table look like?

A. ["C", "-.-."]
B. ["C -.-."]
C. ("C", "-.-.")
D. "C", "-.-."

Each entry of Table is a list with 2 elements. This means we can walk through table with a for-loop like this to find the letter that a given code c represents:

```
for [letter, code] in Table:
        if code == c:
                return letter
```

The third program you need to write works with a different kind of code. This gives you a file encrypted with a "Caesar cypher". This shifts each letter by a certain number of characters. For example the shift of size 2 has

letter	a	b	c	d	e	f	g	h	i
code	c	d	e	f	g	h	i	j	k

....

u	v	w	x	y	z
w	x	y	z	a	b

This encodes "cab" as "ecd". We can decode "hkxg" as "five".

The problem with decoding Caesar cyphers is that you usually aren't told the size of the shift. Of course, you could try all 26 shifts and see which turns the code into English. A nicer alternative is to use the fact that in most samples of English text the most common letter is "e". So if we count the number of instances of each letter, the most common letter is probably what "e" was shifted . For example, if the most common letter in a coded piece of text is "h", the shift was probably size 3 -- shifting "e" to "h", shifting "f" to "i", and so forth.

In the third program you will read a file of encoded text twice. On the first pass you will just count the number of instances of each letter. For this start with a list Counts of 26 0's. Each time you see an "a" increment Counts[0]; each time you see "b" increment Counts[1], and so forth. You can ignore any characters that aren't lower-case letters.

After you have read the whole file you can calculate the size of the shift by finding the most common letter. You then reset the file and read it again, shifting each letter in the opposite direction to return it to its uncoded state.

You need to find indexes for letters -- 0 for "a", 1 for "b", and so forth. One way to do this is to use an alphabet string. To find the index for character c:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
if c in alphabet:
        i = alphabet.index(c)
```

To go the other way,

```
alphabet[3]
```

is the letter "d".

An alternative is to use the ord( ) and chr( ) functions.

ord(letter)

is the index of letter in the standard table of characters. ord("a") is 97, ord("b") is 98 and so forth. If x is a lower-case letter,

i = ord(x)-ord("a")

is the index of x in our Counts list.

chr(i)

is the ith letter in the standard table of characters.
So chr(ord("a") + 3) is "d".

Notice that ord("y") + 3 gets an index that is too big. This is where the table wraps around. Since

ord("y") + 3 > ord("z")

we shift "y" to

chr(ord("y") + 3 - 26)

which is "b".

Similarly, if we are shifting by -3

ord("c") - 3 < ord("a")

so we add 26 and shift "c" to

chr(ord("c") - 3 + 26)

which is "z".