15-112: Introduction to Programming and Computer Science, Fall 2020

Homework 6: Dictionaries and Recursion

Due: Tuesday, October 20, 2020 by 22:00

This assignment has 5 questions, for a total of 50 points.
To start this homework....

1. Create a folder named "week6"
2. Create a file called hw6.py and write all your code in that file.
3. When you have completed and fully tested hw6, submit hw6.py to Gradescope. For this hw, you may submit up to 12 times, but only your last submission counts.

Some important notes:

1. Some problems on this assignment are labeled as **recursive**. For these problems, using a for loop, a while loop, a list/set/dictionary comprehension, or a generator will result in the code failing on Gradescope. You also may not use inherently iterative functions. These include range, sum, max, min, reversed, sorted, list.count, list.sort, list.reverse, str.replace, and str.join.
2. You may not use try/except or classes on this homework.
3. After you submit to Gradescope, make sure you check your score. If you aren't sure how to do this, then ask a CA or Professor.
4. There is no partial credit on Gradescope testcases. Your Gradescope score is your Gradescope score.
5. Read the last bullet point again. Seriously, we won't go back later and increase your Gradescope score for any reason. Even if you worked really hard and it was only a minor error...
6. Do not hardcode the test cases in your solutions.
7. Remember the course's academic integrity policy. Solving the homework yourself is your best preparation for exams and quizzes; cheating or short-cutting your learning process in order to improve your homework score will actually hurt your course grade long-term.
8. Your code will be graded for style. Check the style notes on the website for details.

1. [5 points] `movieAwards(oscarResults)`

   Write the function `movieAwards(oscarResults)` that takes a set of tuples, where each tuple holds the name of a category and the name of the winning movie, then returns a dictionary mapping each movie to the number of the awards that it won. For example, if we provide the set:

```
{
  ("Best Picture", "The Shape of Water"),
  ("Best Actor", "Darkest Hour"),
  ("Best Actress", "Three Billboards Outside Ebbing, Missouri"),
  ("Best Director", "The Shape of Water"),
  ("Best Supporting Actor", "Three Billboards Outside Ebbing, Missouri"),
  ("Best Supporting Actress", "I, Tonya"),
  ("Best Original Score", "The Shape of Water")
}
```

   the program should return:

```
{
  "Darkest Hour" : 1,
  "Three Billboards Outside Ebbing, Missouri" : 2,
  "The Shape of Water" : 3, "I, Tonya" : 1
}
```

   Note: Remember that sets and dictionaries are unordered! For the example above, the returned set may be in a different order than what we have shown, and that is ok.

2. [5 points] **Recursive** `alternatingSum(lst)`

   Write the function `alternatingSum(lst)` that takes a possibly-empty list of numbers, `lst`, and returns the alternating sum of the list, where every other value is subtracted rather than added. For example:

   ```
   alternatingSum([1,2,3,4,5]) returns 1-2+3-4+5 (that is, 3).
   ```

   If lst is empty, return 0.

3. [5 points] **Recursive** `generateCharacterString(s)`

   Write the function `generateCharacterString(s)` that takes a two-character string and returns a new string that contains the all of the characters (in ascii order) between the first character and the second character.

   For example, `generateCharacterString("ko")` would return `"klmno"`. This should also work backwards, so `generateCharacterString("ME")` would return `"MLKJIHGFE"`.

   If the initial provided string is not two characters long, return the empty string; if the provided string contains two identical characters (for example, "22"), return that character ("2").

4. [5 points] **Recursive** `powersOf3ToN(n)`

   Write the function `powersOf3ToN(n)` that takes a possibly-negative float or int `n`, and returns a list of the positive powers of 3 up to and including n.

   As an example, `powersOf3ToN(10.5)` returns `[1, 3, 9]`. If no such powers of 3 exist, you should return the empty list.

   Hint: in the recursive case, consider using a partial result to construct a full result.

5. [30 points] **Turtle Script Converter**

   This problem is designed to get you more practice with using dictionaries, processing strings and get you thinking about recursion. This homework has been adapted from the presentation of Eric Roberts for nifty assignments at SIGCSE 2013.

   Turtle Refresher

   `turtle` is a Python module that allows for drawing of simple figures. In order to use it, you give an imaginary on-screen turtle commands regarding where to go, and it draws a line along its path.

   The details of turtle library can be found at http://docs.python.org/library/turtle.html

   Consider the following very simple turtle program which makes use of some basic turtle functionality to draw two squares:

```
import turtle
turtle.pendown()
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.penup()
turtle.forward(100)
turtle.pendown()
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.penup()
turtle.forward(100)
```

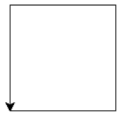
   (You should run this in Python and convince yourself you understand how it works.)

Turtle Script

In order to simplify the usage of turtle, we have designed a new language to writing turtle programs. We call it Turtle Script. A Turtle Script program is specified as a string consisting of a sequence of commands. For example, consider the following Turtle Script program:

```
F120 L90 F120 L90 F120 L90 F120
```

This program moves the turtle in a square 120 pixels on a side, ending up in the same position and orientation as when it started, like so:



Repetition

Turtle Script also includes the concept of repetition. If you enclose a sequence of commands in curly braces, you can repeat that entire sequence any number of times by preceding that block of commands with the letter X followed by the desired number of repetitions. The program to draw a square can therefore be simplified like this:

```
X4 {F120 L90}
```

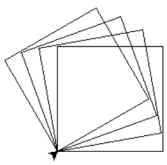
In English pseudocode form, this program therefore has the following effect:

```
Repeat the following sequence of commands 4 times
  Move forward 120 pixels.
  Turn left 90 degrees.
```

Repetitions can be nested to any level. You could, for example, use the program:

```
X4 {X4 {F120 L90} L10}
```

to draw two squares with a 10-degree left turn in the middle.The figure after drawing these four squares looks like this:



Functions

Turtle Script also includes functions. You can defne a function by enclosing the body of the function in braces and then preceding the braces with the command M followed by a single character name for the function. The following example shows a definition of a function called S that draws a square:

```
MS {F120 L90 F120 L90 F120 L90 F120 L90}
```

The keyword M specifies that the following is a function name, followed by function body enclosed in braces. Remember that M command only defines a function but does not execute it. After this function has been defined, you can use the function name to call this function. Here is an example of defining a function and then calling it two times:

```
MS { X4 { L90 F50}} S U F100 D S
```

This program will create two squares in a single line separated by 50 pixels.

Turtle Command Summary

Command	Description
`Fn`	Move the turtle forward by n pixels. If n is missing move by default value of 50 pixels.
`Ln`	Turn the turtle left by n degrees, where n defaults to 90
`Rn`	Turn the turtle right by n degrees, where n defaults to 90
`D`	Calls the pendown function from Turtle
`U`	Calls the penup function from Turtle
`Xn cmds`	Repeats the specified block of commands n times.
`MA cmds`	Defines a function A where A is any single character not already reserved. The function A consists of the commands specified in the block. A cannot be one of the other commands (F, L, R, D, U, X, or M).
`A`	Execute the function A. A is just a placeholder, you can use any character to define a function. A cannot be one of the other commands (F, L, R, D, U, X, or M).

(a) **Problem 1:** `tokenizeTurtleScript(program)`

Write a function called `tokenizeTurtleScript(program)` which, given a Turtle Script program, return a list of commands in that program. The function should break up the input string into individual commands and return a list containing commands as strings.

The most common kind of token in a turtle program is a command character (typically a letter, although any non-whitespace character is legal), which is typically followed by a sequence of decimal digits. For example, the command F120, which moves the turtle forward 120 pixels, is a single token in a turtle program. All command listed in the table above are individual tokens, including repetition and function definitions.

In addition, spaces are not required between tokens but are permitted for readability. These rules mean, for example, that you could rewrite the program that draws a square as:

```
F120L90F120L90F120L90F120
```

even though doing so makes the program much more difficult for people to read.

Consider the following examples:

```
tokenizeTurtleScript("F120L90F120L90F120L90F120") returns
['F120', 'L90', 'F120', 'L90', 'F120', 'L90', 'F120']

tokenizeTurtleScript("X4 {F120 L90} U F200 X4 {F120 L90}") returns
['X4{F120L90}', 'U', 'F200', 'X4{F120L90}']

tokenizeTurtleScript("MS { X4 { L90 F50}} S U F100 D S") returns
['MS{X4{L90F50}}', 'S', 'U', 'F100', 'D', 'S']
```

Note: This problem is made much easier if you plan ahead by choosing good helper functions.

(b) **Problem 2:** `convertTurtleScript(program, funcs)`

Write a function `convertTurtleScript(program, funcs)` that will convert a Turtle Script program to a Python program. It should take as arguments a Turtle Script program and a dictionary consisting of all the functions defined so far in the program. The dictionary will contain function names as keys and function code as values.

The convertTurtleScript function has the responsibility of taking the program and translating each token to the appropriate command for turtle in Python. For example, given the program tokens:

```
['F120', 'L90', 'F120', 'L90', 'F120', 'L90', 'F120']
```

The convertTurtleScript function will have to translate each token into the appropriate function call in Python. Thus, executing the `F120` token needs to invoke the function call `turtle.forward(120)`. Similarly, executing the `L90` token needs to invoke a call to `turtle.left(90)`

Notes:

- You should call tokenizeTurtleScript in order to convert the program to tokens, and then execute the tokens one at a time.
- This function does not return anything. Instead, it prints the Python program.
- You should make this function recursive when necessary. Recursion makes handling the repetition tokens (X) and the function tokens (M) much easier.
- The `funcs` argument that is passed to this function is a dictionary to map function names to function code. Recall that functions are defined as part of Turtle Script. Hence, as you are executing commands you will come across function definitions. When you do, add those functions to the `funcs` dictionary. Later, if a function is being called, you can retrieve that function's commands from the dictionary. For the initial call to convertTurtleScript, you should just pass an empty dictionary.

Consider the following examples.

First,

```
p = "F120 L90 F120 L90 F120 L90 F120"
funcs = dict()
convertTurtleScript(p, funcs)
```

will print out:

```
turtle.forward(120)
turtle.left(90)
turtle.forward(120)
turtle.left(90)
turtle.forward(120)
turtle.left(90)
turtle.forward(120)
```

Second,

```
p = "MS { X4 { L90 F50}} S U F100 D S"
funcs = dict()
convertTurtleScript(p, funcs)
```

will print out:

```
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.penup()
turtle.forward(100)
turtle.pendown()
turtle.left(90)
turtle.forward(50)
```

```
        turtle.left(90)
        turtle.forward(50)
        turtle.left(90)
        turtle.forward(50)
        turtle.left(90)
        turtle.forward(50)
```