15-112: Fundamentals of Programming and Computer Science, Fall 2019

Homework 6 Programming: String processing, dictionaries, and recursion

Due: Tuesday, October 22, 2019 by 22:00

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

Your submission will be made through the web interface of Autolab. We will grade you on the overall structure of your code so make sure you make good design decisions. Write all functions in the same file and call that file YourAndrewIDhw6.py. You should not have any test code in this file besides the function definitions plus any other helper functions you want to write. You should not have any main code that is executed. Your functions should be named according to the specifications given in the questions below. Again, if you want to write helper functions within the same file to help you organize your code, you are more than welcome to do so and you can name them whatever you want. You should submit this python file at:

```
https://autolab.cs.cmu.edu/15112q-f19
```

1 Quick Drawing Tutorial on Turtle Graphics - A Review

For the purposes of this homework, we will be using the Turtle module to draw. It is really simple to use. In particular, you start your code by importing the module into your editor, as follows:

```
import turtle
```

Once you have imported the module, imagine you have a robotic turtle on the screen, and you can order it to do any of the functions listed on the module reference page:

http://docs.python.org/library/turtle.html

For example, if you want the turtle to move forward 50px, type:

```
turtle.forward(50)
```

You can also set the position of the turtle by using the setpos() function.

```
turtle.setpos(x_coordinate, y_coordinate)
```

Python treats the turtle drawing environment as a Cartesian plane, where the center of the drawing window has x_coordinate = 0 and y_coordinate = 0.

Other functions that you might want to look at include

- up() and down() - used to indicate whether the turtle should be drawing or not
- right() and left() - used to turn the direction of the turtle
- color() - for making the turtle draw in different colors

2 A New Turtle Language

Your task, in this homework, is to execute a turtle program. The turtle program is written in Turtle Language and consists of commands for a turtle to execute using turtle graphics. Our turtle programs will be strings consisting of a sequence of commands. The individual commands consist of a single letter, which are usually followed by a number. For example, the command F 120 asks the turtle to move forward 120 pixels in the direction it is facing. The command L90 asks the turtle to turn left 90 degrees. A program is simply a sequence of these commands. The program:

```
F120 L90 F 120 L90 F120 L 90 F 120
```

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

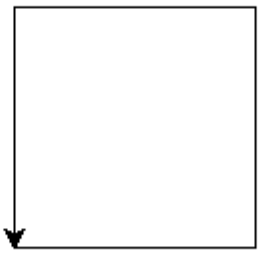

Figure 1: Program output for F120 L90 F120 L90 F120 L90 F120

The turtle language 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 two squares looks like this:

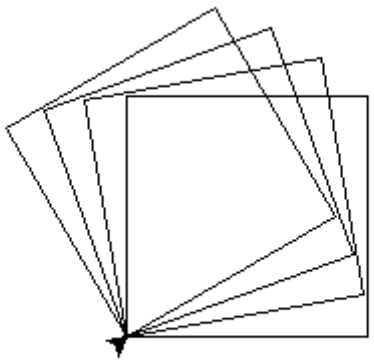

Figure 2: Program output for X4 X4 F120 L90 L10

Lastly, the turtle language also includes functions. You can define 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 specified 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. Following 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.

Task 1 (20 pts) *Write a function called Tokenize(program). This function should take a string consisting of a turtle program as input. The function should break up the input string into individual commands and return a list containing commands as strings. he 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. The digits, however, are optional. If you supply the L or R command without any digits, the TurtleGraphics application assumes that you want the turtle to turn 90 degrees in the indicated direction. Similarly, executing the F command without any digits moves forward 50 pixels as a default. 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*

```
F120LF120LF120LF120L
```

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

If you use this string as input to this function, the function should return a list of eight tokens - ["F120","L","F120","L","F120","L","F120","L"].

For the X command that repeats a block of code, it is necessary to define an additional type of token. For example, the program that uses the X command to draw a square looks like this:

```
X4 {F120 L90}
```

When you write the code to execute this program, you need to recognize that everything inside the braces needs to be repeated. One of the easiest ways to implement tokens enclosed in braces is to have the function consider the entire block after X command as part of the X command.

Task 2 (20 pts) *Write a function execute(commands,functions) that will take a list of strings containing tokens for a turtle 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 execute function has the responsibility of taking the tokens and translating each token to the appropriate command for Turtle. For example, given the program tokens:*

```
["F120", "L90", "F120", "L90", "F120", "L90", "F120", "L90"]
```

The execute method will have to translate each token into the appropriate function call to the Turtle object. Thus, executing the F120 token needs to invoke the method call turtle.forward(120). Similarly, executing the L90 token needs to invoke a call to turtle.left(90)

Your program is required to implement the command forms shown in the following table:

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	*Execute the function A. A is just a placeholder, you can use any character to define a function*

Note: A short explanation of the functions argument that is passed to this function. Recall that functions are defined as part of the program in turtle language. Hence as you are executing command you will come across function definitions and you will add these definitions to the functions dictionary. When we execute your code at the very beginning, we will pass in an empty dictionary. For example to execute the code that defines a function S and then calls the function to draw five squares in a line will be called as follows:

```
cmds = Tokenize("MS {X4 {L90 F50}} X5 { S F100}")
# empty dictionary
funs = { }
execute(cmds,funs)
```

Task 3 (5 pts) *In this assignment, you will be graded on style. Since you have been working on style for last several homeworks, we will be strict in grading style points for this assignment. Make sure your code meets the style guidelines given in the previous homeworks: We will use the following rubric*[1] *for grading style points.*

- *You must include your name and andrewId in a comment at the top of every file you submit.*
- *This is good practice for later in life, when you will want to document all code that you contribute to projects.*
- *2-point error: not writing your name/andrewId in a submitted file*

Comments

- *You should write concise, clear, and informative comments that supplement your code and improve understanding.*
- *Comments should be included with any piece of code that is not self-documenting.*

[1] Adopted with minor modifications from `https://www.cs.cmu.edu/~112/notes/notes-style.html`

- *Comments should also be included at the start of every function (including helper functions).*
- *Comments should not be written where they are not needed.*
- *5-point error: not writing any comments at all.*
- *2-point error: writing too many or too few comments, or writing bad comments.*

Helper Functions (Top-Down Design)

- *You should use top-down design to break large programs down into helper functions where appropriate.*
- *This also means that no function should become too long (and therefore unclear).*
- *5-point error: not using any helper functions (where helper functions are needed).*
- *2-point error: using too many or too few helper functions.*
- *2-point error: writing a function that is more than 20 lines long. Exceptions: blank lines and comments do not count towards this line limit, and this rule does not apply to graphics functions and init()/run() functions in animations.*

Variable Names

- *Use meaningful variable and function names (whenever possible).*
- *Variables and functions should be written in the camelCase format. In this format, the first letter is in lowercase, and all following words are uppercased (eg: tetrisPiece).*
- *Variable names should not overwrite built-in function names; for example, str is a bad name for a string variable. Common built-in keywords to avoid include dict, dir, id, input, int, len, list, map, max, min, next, object, set, str, sum, and type.*
- *5-point error: not having any meaningful variable names (assuming variables are used).*
- *2-point error: using some non-meaningful variable names. Exceptions: i/j for index/iterator, c for character, s for string, and n/x/y for number.*
- *2-point error: not using camelCase formatting.*
- *2-point error: using a built-in function name as a variable.*

Unused Code

- *Your code should not include any dead code (code that will never be executed).*
- *Additionally, all debugging code should be removed once your program is complete, even if it has been commented out.*
- *2-points error: having any dead or debugging code.*

Formatting

- *Your code formatting should make your code readable. This includes:*
  - *Not exceeding 79 characters in any one line (including comments!).*
  - *Indenting consistently. Use spaces, not tabs, with 4 spaces per indent level (most editors let you map tabs to spaces automatically).*
  - *Using consistent whitespace throughout your code.*
  - *Good whitespace: x=y+2, x = y+2, or x = y + 2*
  - *Bad whitespace: x= y+2, x = y +2, or x = y + 2*
- *2-point error: having bad formatting in any of the ways described above.*