CS 1 Honor Roll: Introduction To Computer Programming, Fall 2013

Assignment 1h: Warmup

Due: *Monday, November 18, 02:00:00*

Before you begin

Please read the web page entitled "CS 1 Honor Roll Policies and Information" so that you know how the Honor Roll works and how your assignments are going to be graded. We will expect that you have read this in detail before attempting this assignment, so please don't send us questions that can be answered by reading that page; if you do we'll just redirect you to that page.

What to hand in

Put all of your files for this assignment into a single zip file, and submit that as `lab1h.zip`. This will include separate files for each of the problems, plus whatever test scripts or other files you want to include.

Submit your files to csman, but not to the usual CS 1 location. All of you should now have an account with the CS 1 Honor Roll "course" on csman, and you should submit your honor roll assignments there.

Background

This assignment consists of two (2) problems. Both of these are related to the miniproject in assignment 3. We don't anticipate that they will be particularly difficult or require a lot of work or a lot of coding. Think of this as a fairly gentle introduction to the Honor Roll way of doing things. Honor Roll problems will not be described in the same excruciating level of detail as normal CS 1 problems, and you will be given a fair amount of latitude in how you approach the problems. Since this code extends the assignment 3 miniproject, you need to include whatever code is needed from that miniproject in order to make your submission self-contained. You may modify the assignment 3 code if you want to or need to.

Problem 1: L-systems: Fractal plant drawings

For this problem, we are going to extend the L-systems miniproject from assignment 3 to handle *brackets*, which are a new special character that can occur in L-system strings.

Brackets

The L-systems described in assignment 3 are sufficient to generate a lot of interesting images. However, for some applications (notably, for simulating plant structures), it's useful to also have the two special bracket characters "[" and "]". These characters don't do anything when the string is being constructed; they don't occur on the left-hand side of any rules, for instance, so they are just copied from one L-system string to the next unchanged. However, they do do something when the string is being used as instructions for drawing. What the "[" character does is save (append) the current (x, y) coordinates of the turtle as well as the angle the turtle is facing onto a list. The "]" character restores the last saved (x, y) coordinates and angle of the turtle by removing the last element from the list (we call this *popping* a value from the end of the list, and lists used this way are often called *stacks*) and making the previous (x, y) coordinates and angle the current ones. The bracket characters allow the turtle to go back to a position and orientation where it was previously. When the turtle is restored to a previous position and orientation, a line is not drawn from the last position to the new (restored) position (in turtle graphics terms, the "pen is up" while the coordinates are adjusted, and then the "pen is down" again after the adjustment, so more drawing can resume). We can use the "[" and "]" characters to generate tree-like or plant-like images, and we'll do exactly that below.

New L-system code

We will use the following Python code to represent a plant structure as an L-system:

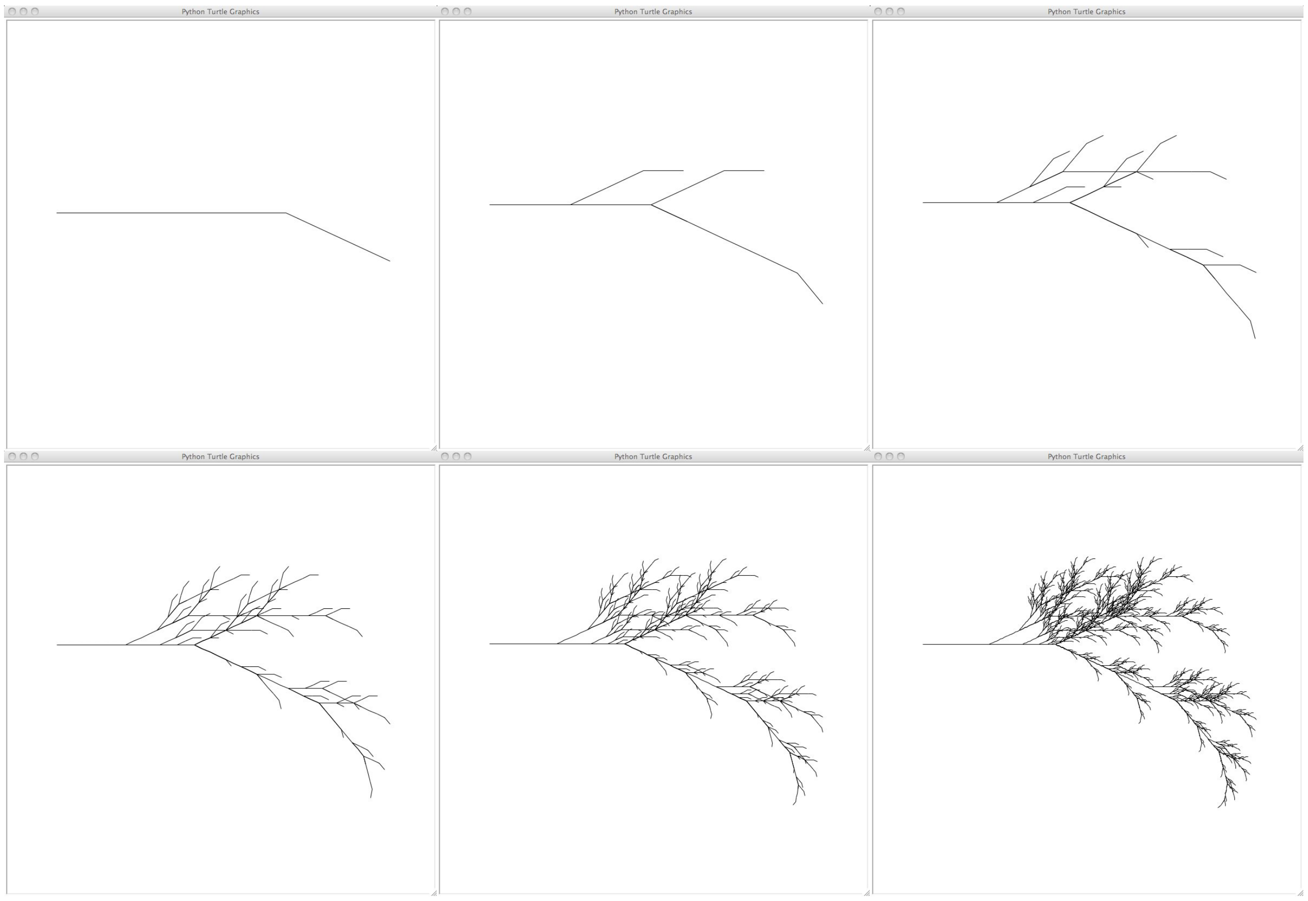
```
plant = { 'start' : 'X',
          'X'     : 'F-[[X]+X]+F[+FX]-X',
          'F'     : 'FF' }

plant_draw = { 'F' : 'F 1',
               '-' : 'L 25',
               '+' : 'R 25' }
```

Note that the `'X'` key in the `plant` dictionary is bound to a string value that contains the bracket characters `'['` and `']'`. We'll use this L-system as our example of using brackets in L-systems.

Plants pictures

Here are some plant pictures generated by using the plant L-system, from levels 1 to 6:



Program to write

You need to modify the `lab3b.py` code you submitted for assignment 3 so that it can work with the plant L-system as well. This is actually not that hard. Of the six functions you wrote for the assignment 3 miniproject, you will only need to modify these ones:

1. `lsystemToDrawingCommands`
2. `nextLocation`

The interface (number and name of arguments) of these functions should be unchanged. Of course, include all the other L-system functions as well; your submission should be self-contained and it should be possible to generate the plant drawing files shown above just by running the program from the command-line.

The drawing commands generated by the L-system code need to be extended to handle the brackets. A left bracket will save the current location in an internal data structure (appending to the end of a list works well), while a right bracket will remove the last saved location (popping from the end of a list using the `pop` method on lists works well) and output a "goto" instruction, which has this format:

```
G <x> <y> <angle>
```

where `<x>`, `<y>`, and `<angle>` are floating-point numbers representing the saved x-coordinate, y-coordinate, and angle with the horizontal, respectively. Angles should only be in the range [0, 360) degrees.

Note that any list used for saving/restoring locations/angles should be a local variable of a function, not a global variable. Note also that we don't specify the `'['` or `']'` characters as keys in the `plant_draw` dictionary, because there is no uniform mapping between these characters and specific drawing commands (the `'['` character doesn't generate any drawing commands, and the `']'` character generates a `G` command whose parameters depend on the last saved location).

Note that the code for this problem (other than test code) should be collected into a program that can be run as a stand-alone Python program from the (terminal) command line.

Once you've written your code, you should check that it can generate the above figures. Use the `lab3draw.py` program to draw your pictures; this program understands the `G` syntax for "goto" instructions. You're also free to write `nose`-style tests if you like; if you do this, you can use the assignment 3 tests as a template.

Problem 2: L-systems: Compressing drawing files

If you look at the output of the drawing files from the first problem, or if you watch them being drawn, you'll notice that there are lots of repetitions. For instance, at level 5, the drawing file looks something like this:

```
0 79.1586 -38.2663 20.2408
F 1
```

```
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
F 1
L 25
...
```

This means that you have a whole series of forward (`F`) commands coming one after another. Drawing these is inefficient because you can always replace them with a single F command, like this:

```
0 79.1586 -38.2663 20.2408
F 16
L 25
...
```

Similarly, sequences of left (`L`) and right (`R`) rotation commands can be replaced by a single command. It's easy to combine successive left and right rotations, and combining a left and a right rotation is almost as easy. Also, sequences of goto (`G`) commands can be replaced with the last such command.

For this problem, you are to write a stand-alone Python program called `compact.py` which takes a single filename argument on the command-line, reads in the file data, compacts them as described above, and writes out a new file with the compacted data, such that it can be drawn by the same program that could draw the original data (the `lab3draw.py` program), but faster since there are fewer lines to draw. You will need to read the Python documentation to learn how to access command-line arguments (we will cover this later in the class). The output file should have the same name as the input file, except with the extension `.compact`. An example run would look like this:

```
  % python compact.py plant_5
```

after which there would be a file called `plant_5.compact` in the same directory with the compacted drawing commands.

Note that the plant drawings only repeat the forward (`F`) commands, so they aren't really a good test of your program. Therefore, you should also write a test suite using the `nose` module, and call it `compact_tests.py`. Make sure that it's comprehensive and that your program passes the test suite! Also make use of the `if __name__ == '__main__': ...` idiom of Python to allow `compact.py` to be used either as a module (for testing) or as a stand-alone program. This is exactly why the `if __name__ == '__main__': ...` idiom was invented in the first place.
