Code Review Stack Exchange is a question and answer site for peer programmer code reviews. It only takes a minute to sign up.

Sign up to join this community

Anybody can ask a question

Anybody can answer

The best answers are voted up and rise to the top

CODE REVIEW

9

1

L-systems are basically rules for recursively rewriting a string, which can be used to characterize e.g. some fractal and plant growth.

I wrote a small class to represent deterministic L-systems and used it for two examples. Any comments would be greatly appreciated, especially about the class design, the structure of the second example, and how to make things more pythonic. I'm new to Python and don't have any training in "grammars", this is just a hobby.

The class `LSystem.py`:

```
class LSystem:
    """ Lindenmayer System
    LSystem( alphabet, axiom )
      axiom: starting "string", a list of Symbols
      rules: dictionary with rules governing how each Symbol evolves,
             keys are Symbols and values are lists of Symbols
    """
    def __init__(self, axiom, rules):
        self.axiom = axiom
        self.rules = rules

    """ Evaluate system by recursively applying the rules on the axiom """
    def evaluate(self,depth):
        for symbol in self.axiom:
```

```
        if depth <= 0 or symbol not in self.rules:
            symbol.leaf_function()
        else:
            for produced_symbol in self.rules[symbol]:
                self.evaluate_symbol( produced_symbol, depth - 1 )


class Symbol:
    """ Symbol in an L-system alphabet
    Symbol( leaf_function )
      leaf_function: Function run when the symbol is evaluated at the final
                     recursion depth. Could e.g. output a symbol or draw smth.
    """
    def __init__(self, leaf_function ):
        self.leaf_function = leaf_function
```

Example: Algae growth (example 1 from the wikipedia article)

```
import LSystem

# define symbols. their "leaf function" is to print themselves.
A = LSystem.Symbol( lambda:print('A',end='') )
B = LSystem.Symbol( lambda:print('B',end='') )
# define system
algae_system = LSystem.LSystem(
    axiom = [A],
    rules = { A: [A,B], B: [A] }
```

Example: Draw a Koch snowflake (as in my previous question)

```
import LSystem
import pygame
from math import pi, sin, cos

# some constants
WINDOW_SIZE = [300,300]
LINE_WIDTH = 1
LINE_LENGTH = 1

# global variables for "turtle drawing"
# maybe I should pass around a turtle/cursor object instead?
turtle_angle = 0
turtle_x = 0
turtle_y = WINDOW_SIZE[1]*3/4

# define drawing functions used to draw the Koch snowflake
def draw_forward():
    global turtle_angle, turtle_x, turtle_y
    start = [turtle_x, turtle_y]
    turtle_x += LINE_LENGTH * cos(turtle_angle)
    turtle_y += LINE_LENGTH * sin(turtle_angle)
    end = [turtle_x, turtle_y ]
    pygame.draw.line(window, pygame.Color('black'), start, end, LINE_WIDTH )
def turn_left():
    global turtle_angle
    turtle_angle += pi/2
```

```
# symbols in the L-system
Line = LSystem.Symbol( draw_forward )
Left = LSystem.Symbol( turn_left )
Right = LSystem.Symbol( turn_right )

# L-system axiom and rules
koch_curve_system = LSystem.LSystem(
    axiom = [ Line, Right, Right, Line, Right, Right, Line ],
    rules = { Line: [ Line, Left, Line, Right, Right, Line, Left, Line ] }
)

# init pygame
pygame.init()
window = pygame.display.set_mode(WINDOW_SIZE)
window.fill(pygame.Color('white'))

# evaluate the L-system, which draws the Koch snowflake
# (recursion depth was chosen manually to fit window size and line length)
koch_curve_system.evaluate(5)

# display
pygame.display.flip()

# wait for the user to exit
while pygame.event.wait().type != pygame.QUIT:
    1
```

python beginner pygame fractals

Community ♦
1

Anna
180 6

1 Answer

5

This is a neat implementation of Lindenmayer systems. I have some suggestions for simplifying and organizing the code.

1. The docstring for a method or a function comes *after* the `def` line (not before, as in the code here). So you need something like:

```
def evaluate(self, depth):
    """Evaluate system by recursively applying the rules on the axiom."""
    for symbol in self.axiom:
        self.evaluate_symbol(symbol, depth)
```

and then you can use the `help` function from the interactive interpreter:

```
>>> help(LSystem.evaluate)
Help on function evaluate in module LSystem:

evaluate(self, depth)
```

2. The `Symbol` class is redundant — it only has one attribute, and doesn't have any methods other than the constructor. Instead of constructing `Symbol` objects, you could just use functions:

```
def A():
    print('A', end='')

def B():
    print('B', end='')
```

and instead of calling `symbol.leaf_function()`, you could just call `symbol()`.

In the Koch example, you already have functions so you can just omit the construction of the `Symbol` objects and write:

```
koch_curve_system = LSystem(
    axiom = [draw_forward, turn_right, turn_right, draw_forward, turn_right,
             turn_right, draw_forward],
    rules = {
        draw_forward: [draw_forward, turn_left, draw_forward,
                       turn_right, turn_right, draw_forward, turn_left,
                       draw_forward],
    }
)
```

3. The code in `evaluate` is very similar to the code in `evaluate_symbol`. This suggests that it would result in simpler code if you described the Lindemayer system in a different way, giving an *initial symbol* instead of an *initial list of symbols*. (And possibly giving an extra rule mapping the initial symbol to a list.)

   If you try this, then you'll find that the `LSystem` class is redundant too: the only thing you can do with it is to call its `evaluate` method, so you might as well just write it as a function:

```
def evaluate_lsystem(symbol, rules, depth):
    """Evaluate a Lindenmayer system.

    symbol: initial symbol.
    rules: rules for evolution of the system, in the form of a
        dictionary mapping a symbol to a list of symbols. Symbols
        should be represented as functions taking no arguments.
    depth: depth at which to call the symbols.

    """
    if depth <= 0 or symbol not in rules:
        symbol()
    else:
        for produced_symbol in rules[symbol]:
            evaluate_lsystem(produced_symbol, rules, depth - 1)
```

4. In the snowflake example, there is persistent shared state (the position and heading of the turtle). When you have persistent shared state it makes sense to define a class, something like this:

```
class Turtle:
    """A drawing context with a position and a heading."""
    angle = 0
    x = 0
    y = WINDOW_SIZE[1]*3/4

    def forward(self, distance):
        """Move forward by distance."""
        start = [self.x, self.y]
        self.x += distance * cos(self.angle)
        self.y += distance * sin(self.angle)
        end = [self.x, self.y ]
        pygame.draw.line(window, LINE_COLOR, start, end, LINE_WIDTH)

    def turn(self, angle):
        """Turn left by angle."""
        self.angle += angle
```

and then:

```
turtle = Turtle()
forward = lambda: turtle.forward(1)
```

```
    initial: [forward, right, right, forward, right, right, forward],
    forward: [forward, left, forward, right, right, forward, left, forward],
}
evaluate_lsystem(initial, rules, 5)
```

answered May 30 '16 at 13:31

Gareth Rees

46.5k 3 108 192

Thank you very much for your feedback!! I've learnt so much from my two posts on this SE. 1. Didn't know these descriptions were actually parsed. Also thanks for letting me know they're called docstrings. 2. Clever to just pass the function, I didn't think of that! 3&4. Thank you, very concise and beautiful! – Anna May 31 '16 at 8:33