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

Asked 2 years, 9 months ago Active 6 months ago Viewed 1k times

2

An L-system is a rewriting system that can be used to generate fractals and space filling curves, because of its recursive nature.

Some L-systems for mathematical curves can be found here.

An example L-system:

Axiom:

$$FX + FX+$$

Production rules:

$$X \to X + YF$$

$$Y \to FX - Y$$

Angle:

$$\theta = 90$$

In this example, F means "move forward", "+" means "turn right by angle theta, and "-"

The axiom states "move forward, turn right, move forwards, turn right". After applying the production rules, the command string becomes

$$FX + YF + FX + YF +$$

Performing many iterations produces long, complicated chains of commands. As the turtle follows the commands, very interesting shapes can be produced.

```
import matplotlib.pyplot as plt
import numpy as np
from math import sin, cos, atan2, radians

class turtle:
    """
    A turtle is a simple object with a direction and a position.
    It can follow two basic commands: move forward and turn by an angle
    """
    def __init__(self):
        self._direction = np.array([1, 0]) # 2D direction vector
        self._position = np.array([0, 0]) # 2D position vector
    def forward(self):
        """
        Move turtle forward by one unit.
        """
        pos = self._position
        dirn = self._direction
        self._position = np.add(pos, dirn)
```

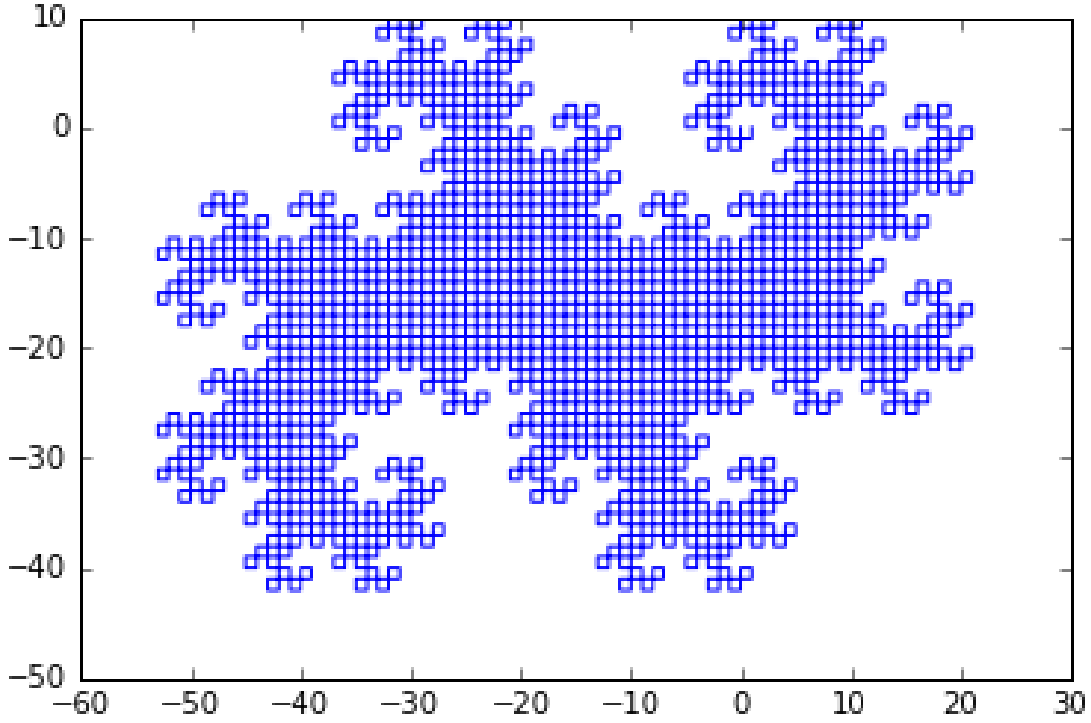
```
        (x, y) = self._direction
        current_angle = atan2(y, x)
        new_angle = current_angle + radians(theta)

        self._direction = [cos(new_angle), sin(new_angle)]


def L_system(commands, axiom, production_rules, theta, n_iterations):
    """
    Executes the commands of an L-system on a turtle object,
    and returns the resulting positions.

    Beginning with a string of simple commands, this string is made longer by
    replacing single characters with longer strings, in a recursive manner.

    By completing a number of iterations of this process, a long command string
    is generated. A 'turtle' object then follows these commands in order.
    It can only move forward or change its direction. The positions of the turtle
are
    returned in a matrix.

    Parameters
    ----------
    commands : dict
        Maps single characters to function calls written as strings
        The functions are performed on a turtle object
        e.g. {'+': 't.rotate(-theta)', '-': 't.rotate(theta)', 'F':
't.forward()'}
```

```
            e.g. 'FX+FX+'

        production_rules : dict
            Maps single character strings to more complicated strings of characters
            The value strings replace the key strings on each new iteration
            e.g. {'X': 'X+YF', 'Y': 'FX-Y'}

        theta : int
            Angle of rotation, in degrees
            e.g. 90

        n_iterations : int
            Number of iterations for the L system
            e.g. 5

        Returns
        -------
        positions : numpy matrix
            The positions of the turtle, while following commands in the final
    command string

        """
        command_string = axiom # Begin commands with only the axiom
        for iteration in range(n_iterations):
            new_command_string = str()
            for char in command_string:
                if char in production_rules:
                    new_command_string += production_rules[char]
                else:
```

```
    t = turtle() # Initialize a turtle at position [0, 0]

    positions = np.zeros((n_commands, 2))

    for i, command in enumerate(command_string):
        if command in commands:
            exec(commands[command]) # Perform command on turtle
        positions[i, :] = t._position

    return positions


commands = {
    'F': 't.forward()',
    '+': 't.rotate(-theta)',
    '-': 't.rotate(theta)',
    }

axiom = 'FX+FX+'

production_rules = {
    'X': 'X+YF',
    'Y': 'FX-Y'
    }

n_iterations = 11

theta = 90
```

```
positions = L_system(commands, axiom, production_rules, theta, n_iterations)
plt.plot(positions[:, 0], positions[:, 1])
```

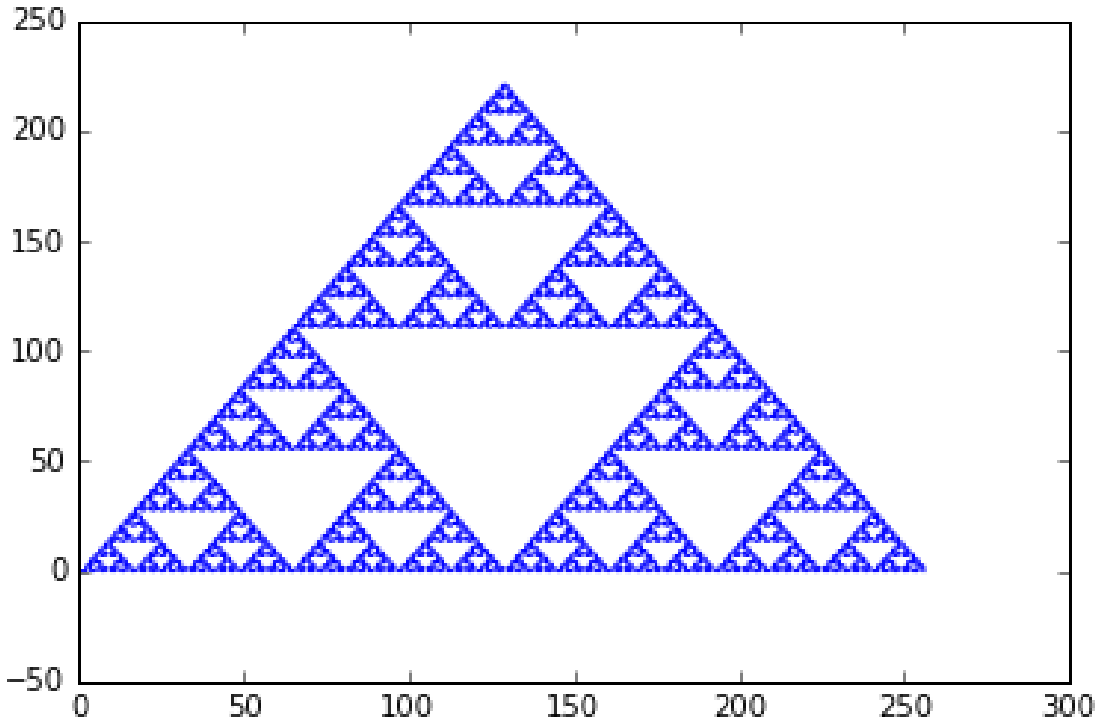
The output of this code is the Twin Dragon fractal:



Other examples include the Sierpinski Arrowhead curve:

```
axiom = 'FX'
```

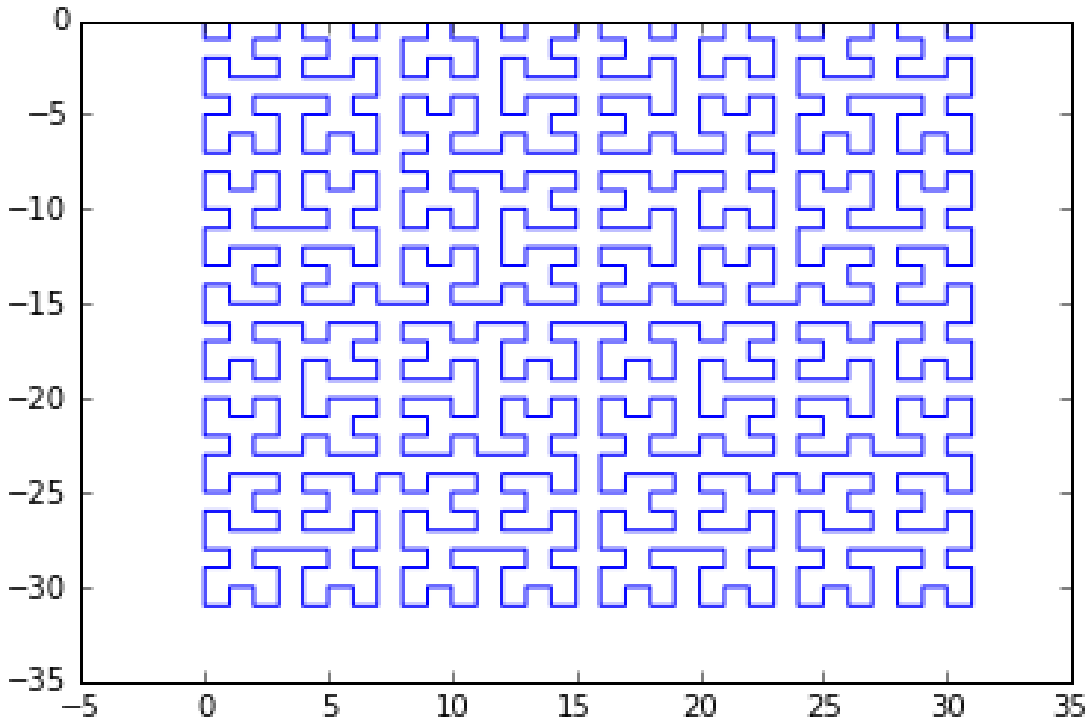
```
n_iterations = 8

theta = 60
```



The Hilbert curve:

```
axiom = 'X'
```

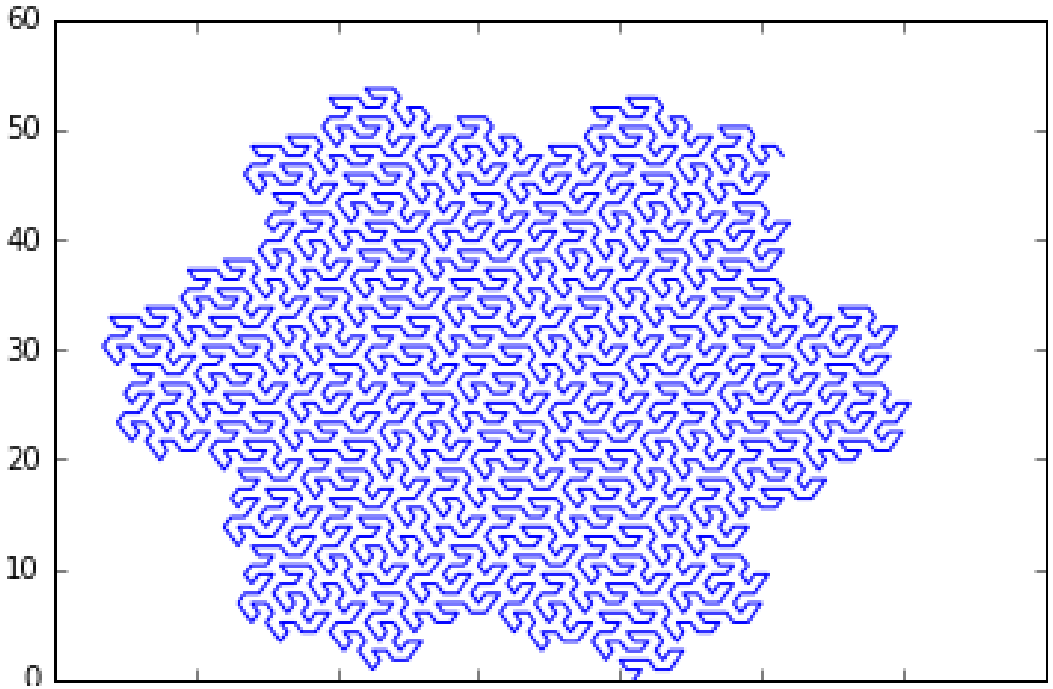
```
n_iterations = 5

theta = 90
```



And the Gosper Curve:

```
commands = {
    'A': 'f forward()'
```

```
axiom = 'A'

production_rules = {
    'A': 'A-B--B+A++AA+B-',
    'B': '+A-BB--B-A++A+B'
    }

n_iterations = 4

theta = 60
```



asked Oct 25 '16 at 2:16

Vermillion

530 1 6 20

2 Answers

1

Your substituting part should be put into its own function as well. It can also be greatly simplified using `dict.get` :

```
rules = {"X": "X+YF", "Y": "FX-Y"}

def apply_rules(text, rules):
    return "".join(rules.get(c, c) for c in text)

>>> apply_rules("FX+FX+", rules)
'FX+YF+FX+YF+'
```

Which you can embed in your code like this:

```
command_string = axiom # Begin commands with only the axiom
for   in range(n iterations):
```

I would put your example code at the end into a `if __name__ == '__main__':` guard to allow your module to be imported without this code being run.

The variable `n_commands` is used exactly once, so it is not necessary to declare it.

Your function is potentially very dangerous, because I could put *any* command in commands, even system calls.

edited Oct 25 '16 at 13:21

answered Oct 25 '16 at 6:59

Graipher

29.6k 6 47 104

Is there another way to call these functions without using exec() ? – Vermillion Oct 25 '16 at 16:55

1 @Vermillion I was trying to find a better way, but they all suffered the same problem (one would be writing a class deriving from your turtle class with a method handling the translation. But then this class can have arbitrary code in it, too...). – Graipher Oct 25 '16 at 17:38

Is this the best way to move the turtle then? – Vermillion Oct 25 '16 at 18:36

@Vermillion I don't think so, but I need some more time to think of a better one. – Graipher Oct 25 '16 at 18:51

equivalent of a switch statement to dispatch the command? The switch would embody what is now the `commands` dictionary. That way, the turtle class strictly controls which commands it will accept (and ignores all others). If you want flexibility on which variables (A, B, X, Y, F, but not L, R) get interpreted as `forward()`, allow the caller to pass in a list of such symbols. – LarsH Dec 14 '16 at 20:26

2

`L_system` performs two unrelated actions:

- It computes the final string containing only terminals, and
- It commands the turtle to draw the string.

I strongly recommend to split it into two methods, e.g.

```
def L_system(axioms, productions, iterations):
```

and

```
def draw_path(L_string, theta, step):
```

vnp
42.6k 2 35 109