Fractals.Py

A workbook on implementing L-system fractals using Python

Dr. Shibaji Banerjee
Department of Physics, St. Xavier's College

Introduction

Fractals are objects whose parts are similar to the whole structure at all length scales. They are geometrical objects which are the natural choice for modeling natural shapes like landscapes, shapes of trees, clouds and the like which exhibit this remarkable property of approximate self-similarity. Fractals find useful applications in diverse fields like Dynamical Systems, Computer Graphics or Cosmology This is a short primer on how to implement some algorithms that generate fractals using python. In this session we would work with a class of fractals known as **L-systems**. These fractals can be easily generated using few lines of python code. L-systems were introduced and developed in 1968 by the Hungarian theoretical biologist and botanist from the University of Utrecht, ***Aristid Lindenmayer***. It is most famously used to model the growth processes of plant development, but also able to model the morphology of a variety of organisms (Algorithmic botany). The purpose of the workbook which you are currently reading, is given below. In the order of reducing priorities, these are:

1. To understand and translate an algorithm into action,
2. cultivate an awareness for fractal patterns,
3. relate the algorithm to the effect produced, and,
4. to appreciate the algorithmic beauty of nature.

Each of the following sections is an exercise in self-learning. You're going to learn how to implement the algorithms leading to formation of (a) Algae duplication rules (b) Fibonacci Strings, (c) The Cantor dust, (d) A variant of a Koch curve and (e) The Sierpinski carpet. These appear in increasing order of complexity (and each requires a copy of the previous file ...)

Acknowledgements

1. The beauty of Fractals, Dr. Ananda Dasgupta, IISER, Kolkata
2. The wikipedia page on Lindenmayer systems

Prerequisites

Basic Python constructs, familiarity with turtle module. Note that from Python version 2.5 onwards you can now use the commands `fd(),bk(),lt(),rt(),pu(),pd()` etc directly without assigning them to internal names like `forward(),backward()\ldots` – the turtle module already does them for you!. Every workout introduces you to some new python vocabulary appropriate to the task at hand, so take your time to actually read through the text. You are required to read from the beginning of a section till you hit a thick line and then attempt to write the code.

1 Algae

Lindenmayer's original L-system for modeling the growth of algae consists of finding the rules behind the generation of string patterns like:

For the n values listed in the table. It is found that the pattern can be generated using the following rules:

1. start with `s='A'`
2. replace each `A` in the string by `AB` and each `B` in the string by `A`
3. go on doing this recursively for $n = 0 \ldots 7$.

n = 0	A
n = 1	AB
n = 2	ABA
n = 3	ABAAB
n = 4	ABAABABA
n = 5	ABAABABAABAAB
n = 6	ABAABABAABAABABAABABA
n = 7	ABAABABAABAABABAABABAABAABABAABAAB

Table 1: Growth of Algae, according to Lindenmayer

Write a program to duplicate the output of Table-1.
If you can do it successfully in the workspace, create a file called `algae.py` and write a function `next(s)` to package the transformations[1] and produce the required output. The code for this program is supplied right below, but please consult it only if you are totally stuck at some point, or have successfully generated the output and want to compare.

Notes

All python strings have the .replace() member function attached to them which can be used for the indicated replacements (first check it out on a test string). There is one catch however. As soon as you make the replacements for `'A'`, the `'B'` content of the string would also (usually) change(why?). Since we cannot allow this to happen; replacements **must** be made to the original string, not to the semi-modified string. An easy way to do it right would be to first change all the `'A'`s to `'a'`s and all `'B'`s to `'b'`s first and then replace the only lower cases by the capitalized replacement strings. A good way to get started with this would be to do some initial tests with the `.replace()` function and see how it responds (consider changing parts of your name)

Listing 1: Algae

```
#    variables : A B
#    constants : none
#    start  : A
#    rules  : (A > AB), (B > A)

a='A';b='B' #vars
s='A'       #initial state

def next(s):
        '''
        next(s)->s returns the next value of the iterated string
        '''
    s1=s.replace('A','a')
    s2=s1.replace('B','b')
    s3=s2.replace('a','AB')
    s4=s3.replace('b','A')
    return s4

nrange=range(8) #we would take 7 steps

#This is where the actual work is being done, the rest
#were all preparatory.
for i in nrange:
    print "n= ", i,s.center(30)
    s=next(s)
```

[1] i.e you are required to write a function that takes in a string and gives out the next in line, e.g next('ABA')='ABAAB'

2 Fibonacci

The Fibonacci sequence is defined by the grammer:

```
variables : A B
constants : none
start   : A
rules   : (A > B), (B > AB)
```

In some way this executes the reverse action of `Algae`. The length of the strings generated for a particular value of the iteration index are equal to the corresponding Fibonacci numbers. Generate the string values for $n = 1 \dots 12$ and check their length with the corresponding Fibonacci number defined through a function like `fibo(n)`. The output would look something like:

```
n=  1 A         len=  1         fibo=  1
n=  2 B         len=  1         fibo=  1
n=  3 AB        len=  2         fibo=  2
n=  4 BAB       len=  3         fibo=  3
n=  5 ABBAB     len=  5         fibo=  5
n=  6 BABABBAB  len=  8         fibo=  8
...
```

Notes

Python allows one to write functions that can call themselves(*recursive functions*). An example is the factorial function:

```
>>> def fact(n):
...   if n==0 or n==1:
...     return 1
...   else:
...     return n*fact(n-1)
...
>>> fact(5)
120
```

The Fibonacci numbers can be generated by the recursion rules: `fib(1)=fib(2)=1` and `fib(n)=fib(n-1)+fib(n-2)`

3 Cantor

The famous Cantor's fractal set on a real straight line is produced by the grammar:

```
variables : A B
constants : none
start   : A
rules   : (A > ABA), (B > BBB)
```

Where A mean "draw forward" and B mean "move forward"(lift pen as you move). Write a program to generate the cantor dust using a function `draw(s)` which takes in a string like `ABABBBABA` and can carry out the instructions from left to right. At each step reduce the step-size that the turtles take by 1/3(you can start with a step-size ~ 400). Draw several steps(maybe 5) together in the same diagram, bringing back the turtle at the left of the screen at each iteration and going down by some 30 units before initiating a fresh iteration. The output will appear as in Fig: 1

Notes

You may use the following function to initialize the turtle before each iteration.

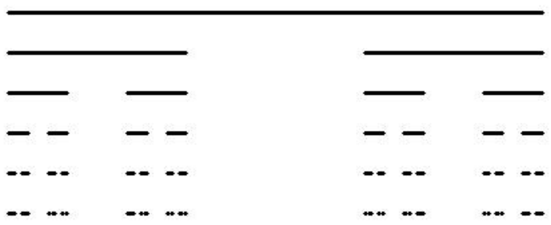

Figure 1: Cantor dust drawn by turtle

```
def penreset(n):
    delay(0.1)
    pu()
    home()
    fd(-200)
    #i stands for the i-th iteration step
    rt(90);fd(i*30);lt(90)
    pd()
```

Print the strings as well with each iteration. The functions `pu()` lifts the pen while the turtle moves, so no mark is made on the canvas. `pd()` puts it down again. The `delay` function (belonging to the turtle module) controls the turtle delay - it takes an argument in milliseconds. We have made it very fast so that the viewer may not be able to see the turtle sweep back and down before each call to draw. You can set the delay to a higher value in the draw routine. The width of the pen can be set using the `width()` function. We have used a width of 3. Use a call to `raw_input()` to suspend the GUI indefinitely.

4 A variant of Koch

A variant of the Koch curve which uses only right-angles is produced by the set of rules:

```
variables : F
constants : +,-
start  : F
rules  : F > F+F-F-F+F
```

Where the constants `+` stands for `lt(90)` and `-` stands for `rt(90)` respectively. This time generate one curve for each iteration. Request the user to input the number of iterations, step-size and delay for each drawing. It would be better to backtrack by about 200 steps before the drawing actually begins. Some pictures from a sample run is shown in Fig: 2.

4.1 Notes

Choose smaller values of step-sizes and delays for large iterations. You can go from $n = 0 \ldots 6$. Use stepsizes like 20,10,5,2.5,1.5 etc. Note the self-similarity of this and the following curve. The `draw()` function that I have used is given below:

```
def draw(s):
    for i in s:
        if i=='F':
            fd(step)
        elif i=='+':
            lt(90)
        elif i=='-':
            rt(90)
```

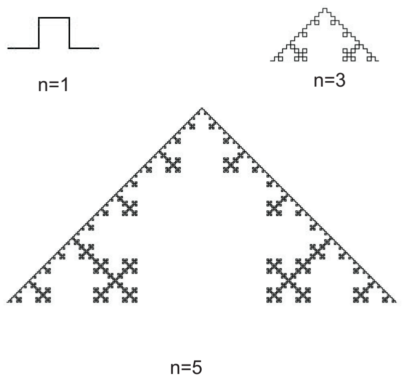

Figure 2: Koch curves drawn by turtle

To hide the turtle after completing the drawing, the `hideturtle` command can be quite useful.

5 Sierpinsky

The Sierpinski carpet, another space filling curve is produced by the directives:

```
variables : A,B
constants : +,-
start  : A
rules  : A > B - A - B; B > A + B + A
angle  : 60
```

Where the angle is the ‘turn-angle’ for left and right turns. Implement this in a program. Sample outputs are shown in Fig: 3

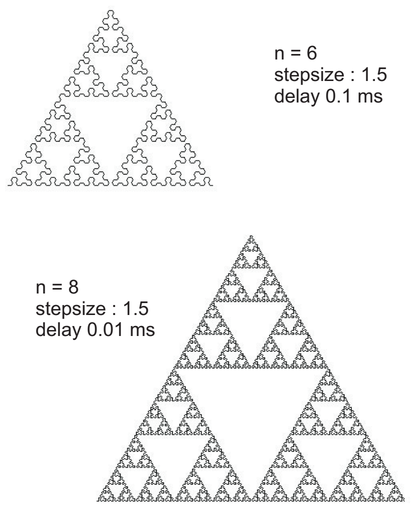

Figure 3: The Sierpinski carpets drawn by the turtle

6 Codes and Output

6.1 Algae

Code and ouput appears in the Sec. (1) of the text.

6.2 Fibonacci

Listing 2: Fibonacci

```
#     variables : A B
#     constants : none
#     start  : A
#     rules  : (A > B), (B > AB)

def fibo(n):
    if n > 2:
        return fibo(n-1)+fibo(n-2)
    elif n == 1 or n == 2:
        return 1

a='A';b='B'
s='A'

def next(s):
    s1=s.replace('A','a')
    s2=s1.replace('B','b')
    s3=s2.replace('a','B')
    s4=s3.replace('b','AB')
    return s4

nrange=range(1,12) #fibo cannot start from 0, so the range must start from 1
for i in nrange:
    print "n= ", i, s, 'len= ', len(s), 'fibo= ', fibo(i)
    s=next(s)
```

Sample output appears in Sec. (2) of the text.

6.3 Cantor

Listing 3: Cantor

```
#     variables : A B
#     constants : none
#     start   : A
#     rules   : (A > ABA), (B > BBB)
from turtle import *
reset()

a='A';b='B'
s='A'
STEPSIZE=400

def next(s):
    s1=s.replace('A','a')
    s2=s1.replace('B','b')
    s3=s2.replace('a','ABA')
    s4=s3.replace('b','BBB')
    return s4

def draw(s):
    delay(50)
    for char in s:
        if char=='A':
            pd();fd(STEPSIZE)
        elif char == 'B':
            pu();fd(STEPSIZE)

def penreset(n):
    delay(0.1)
    pu()
    home()
    fd(-200)
    rt(90);fd(i*30);lt(90)
    pd()

nrange=range(6)

width(3);
for i in nrange:
    print i,s
    penreset(i)
    draw(s)
    STEPSIZE=STEPSIZE/3.0
    s=next(s)

hideturtle()
raw_input('Enter to quit')
```

Sample output appears in Fig. (1).

6.4 Koch

Listing 4: Koch

```
#     variables : F
#     constants : +,-
#     start   : F
#     rules   : F > F+F-F-F+F

from turtle import *
f='F'
s='F'
step=20

def next(s):
    s1=s.replace('F','F+F-F-F+F')
    return(s1)

def draw(s):
    for i in s:
        if i=='F':
            fd(step)
        elif i=='+':
            lt(90)
        elif i=='-':
            rt(90)

n=input('order?:')
step=input('stepsize?:')
d=input('delay?:')


for i in range(n):
    s=next(s)

pu(); bk(200);pd()
delay(d)
draw(s)
hideturtle()

raw_input('Enter to quit.')
```

Sample output appears in Fig. (2).

6.5 Sierpinski

Listing 5: Sierpinski

```
#     variables : A,B
#     constants : +,-
#     start   : A
#     rules   : A > B - A - B; B > A + B + A
#     angle   : 60


from turtle import *
a='A';b='B';ang=60
s='A'
step=20

def next(s):
    s1=s.replace('A','a')
    s2=s1.replace('B','b')
    s3=s2.replace('a','B-A-B')
    s4=s3.replace('b','A+B+A')
    return(s4)

def draw(s):
    for i in s:
        if i=='A' or i=='B':
            fd(step)
        elif i=='+':
            lt(ang)
        elif i=='-':
            rt(ang)

n=input('order?:')
step=input('stepsize?:')
d=input('delay?:')



for i in range(n):
    s=next(s)

pu(); bk(200);pd()
delay(d)
draw(s)
hideturtle()

raw_input('Enter to quit.')
```
Sample output appears in Fig. (3).