import antigravity

Why bother when all that is needed is an "import antigravity". Circumventing gravity can't possibly get any simpler. Here's to all things pythonic..!
A Python Code blog with insightful(hopefully) snippets illustrating its power and simplicity, along the long and treacherous path to finally conquering the language.
And while we're on it, do enjoy squeezing the juice out of Python[the one with the capital P]

Saturday, August 30, 2008

One liners, lambda tricks and other silly stuff

Primes below a bound

Printing all prime less than or equal to a given number. Doesn't get simpler than this. Just to get you warmed up.

```
z = lambda n : [x for x in range(2, n + 1) if len([i for i in ra
nge(2, x) if x%i == 0]) == 0]
```

z(41) looks like

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
```

Matrix Multiplication

Again, an extremely concise lambda function will do.

```
z = lambda x, y : [[ sum([x[i][k]*y[k][j] for k in range(len(x[0
]))]) for j in range(len(y[0]))] for i in range(len(x))]
```

Multiplying [[41,21,2], [-2,83,-4], [-1,22,10]] and [[5,-1,23], [-12,34,1], [23,97,-2]] yields

```
[[-1, 867, 960], [-1098, 2436, 45], [-39, 1719, -21]]
```

About Me

CHILLU

View my complete profile

Contact Me

quan...@gmail.com

Blog Archive

- ► 2011 (1)
- ▼ 2008 (11)
  - ► November (1)
  - ► September (2)
  - ▼ August (3)
    - One liners, lambda tricks and other silly stuff
    - Say twice, say twice
    - The Self-Describing Sequence

Skewed Triangles

A program to print skewed triangles. I'm yet to find a use for it. First argument controls the height and the second controls skew. Positive values skew right while negative ones skew left, subject, of course, to way you look at it.

```
x = lambda n,t=1 : sys.stdout.write('\n'.join([' '*((((t>0) and
n or 1)-i)*t) + '*'*i for i in range(1,n+1)]))
```

here's x(10,2) and x(12,-3)

```
                  *
                **
              ***
            ****
          *****
        ******
      *******
    ********
  *********
**********
```

```
*
   **
      ***
         ****
            *****
               ******
                  *******
                     ********
                        *********
                           **********
                              ***********
                                 ************
```

The Sierpinski Fractal

One of my better exploits. Creates the sierpinksi fractal using '*'s and '.'s You generally want to restrict yourself to around level 7. The code just calls itself recursively and pastes 3 copies of n - 1 level fractals surrounding a dotted triangular region.

```
z = lambda n : n==1 and '\n'.join('*'*i for i in range(1,4)) or
z(n-1)+'\n'+'\n'.join(([z(n-1).splitlines()[i] + '.'*(3*pow(2,n-
2)-i-1) + z(n-1).splitlines()[i] for i in range(3*pow(2,n-2))]))
```

► July (5)

Subscribe and I'll kee

Posts

Comments

Label Cloud

Amarok BeautifulSoup binary search bisect cmus conky DFS dictionary discrete time convolution faster generators golomb sequence hashcash iterators lambda leoLyricsAPI list list comprehensions lyrics one liners optimize parallel python. swig psyco pyrex python quine quizzy&#39;s word challenge recursion reverse indent script sierpinski fractal solver speedup the-end yield

Content licensed under

Instead of recalculating the n - 1 level fractals thrice every step a little memoization speeds up the whole thing considerably. This is version 2.

```
z = lambda n : n==1 and '\n'.join('*'*i for i in range(1,4)) or (lambda s=z(n-1) : s+'\n'+'\n'.join(([s.splitlines()[i] + '.'*(3*pow(2,n-2)-i-1) + s.splitlines()[i] for i in range(3*pow(2,n-2))])))()
```

Meet z(5)

```
*
**
***
*..*
**.**
******
*.....*
**....**
***...***
*..*..*..*
**.**.**.**
************
*...........*
**..........**
***.........***
*..*........*..*
**.**.......**.**
******......******
*.....*.....*.....*
**....**....**....**
***...***...***...***
*..*..*..*..*..*..*..*
**.**.**.**.**.**.**.**
************************
*.......................*
**......................**
***.....................***
*..*....................*..*
**.**...................**.**
******..................******
*.....*.................*.....*
**....**................**....**
***...***...............***...***
*..*..*..*..............*..*..*..*
**.**.**.**.............**.**.**.**
************............************
*...........*...........*...........*
**..........**..........**..........**
***.........***.........***.........***
*..*........*..*........*..*........*..*
**.**.......**.**.......**.**.......**.**
******......******......******......******
*.....*.....*.....*.....*.....*.....*.....*
**....**....**....**....**....**....**....**
```

```
***...***...***...***...***...***...***...***
*..*..*..*..*..*..*..*..*..*..*..*..*..*..*..*
**.**.**.**.**.**.**.**.**.**.**.**.**.**.**.**
***********************************************
```

Zig-Zag-ing your way through a 2D Matrix

We had a lab assignment for a CS course that required us to subdivide a given region into 4 square blocks and traverse them in a Z sort of fashion. The base case is the single block which is trivially traversed. If the blocks were represented by numbers then print the order in which the numbers are visited. The regions are guaranteed to be square with sides of the form 2^n. A long C program in one line in python. Even if this happens pretty frequently there's still no reason not to say it.

```
y = lambda l, n, rs = 0, cs = 0 : n == 2 and [l[rs][cs], l[rs][c
s + 1], l[rs + 1][cs], l[rs+1][cs+1]] or y(l, n/2, rs, cs) + y(l
, n/2, rs, cs + n/2) + y(l, n/2, rs + n/2, cs) + y(l, n/2, rs +
n/2, cs + n/2)
z = lambda n : (lambda x : y([[1+i+j*x for i in range(0, x)] for
 j in range(0,x)], x, 0, 0))(pow(2,n))
```

Not a one liner. But two. The second function is a wrapper to build the region and send it to the first.
For a region like

```
  1   2   3   4   5   6   7   8
  9  10  11  12  13  14  15  16
 17  18  19  20  21  22  23  24
 25  26  27  28  29  30  31  32
 33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48
 49  50  51  52  53  54  55  56
 57  58  59  60  61  62  63  64
```

Traversing it is as follows

```
1 2 9 10 3 4 11 12 17 18 25 26 19 20 27 28 5 6 13 14 7 8 15 16 21 22 29 30
23 24 31 32 33 34 41 42 35 36 43 44 49 50 57 58 51 52 59 60 37 38 45 46 39
40 47 48 53 54 61 62 55 56 63 64
```

Discrete Time Convolution

Performs the convolution of two discrete time signals and returns the output. The output signal starts at the sum of the starting points of the two input signals and has length equal to len(x) + len(h) - 1

```
h, x = [1,1,1,1,-1,-1,-1,-1], [1,1,1,1,1]
z = lambda x, h : [sum((x[i - j] if i - j >= 0 and i - j < len(x
) else 0)*h[j] for j in range(len(h))) for i in range(len(x) + l
en(h) - 1)]
```

yields an output signal x*h

```
[1, 2, 3, 4, 3, 1, -1, -3, -4, -3, -2, -1]
```

Hoping I've inspired someone to try out devious tricks with lambdas

Posted by chillu at 7:53 AM

Labels: discrete time convolution, lambda, list comprehensions, one liners, python, sierpinski fractal

2 comments:

jamesw said...

And, how exactly do you run these on one line? example python command line, perhaps? The equivalent of perl-e'perl_one_liner_code'

November 16, 2010 at 6:01 AM

Anonymous said...

I think it's python -c 'CODE'
But i am no sure... try python.org ;)

December 7, 2010 at 3:13 AM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)