'''Exercises from Math Adventures with Python'''

#Ex. 1-1

from turtle import \*

for i in range(4):

forward(100)

right(90)

#Ex. 1-2:

from turtle import \*

def square():

for i in range(4):

forward(100)

right(90)

for i in range(60):

square()

right(5)

#Ex. 1-3

def triangle(sidelength):

for i in range(3):

forward(sidelength)

right(120)

#Ex. 1-4

def polygon(sides):

for i in range(sides):

forward(100)

right(360/sides)

#Ex. 1-5

def square(sidelength):

'''Draws a square of a

given sidelength'''

for i in range(4):

forward(sidelength)

right(90)

def spiral():

length = 5

for i in range(60):

square(length)

right(5)

length += 5

#Ex. 1-6

def star(sidelength):

'''Draws a 5-pointed star of a

given sidelength'''

for i in range(5):

forward(sidelength)

right(144) #why this angle??

def starSpiral():

'''Draws a spiral of stars'''

length = 5

for i in range(60):

star(length)

right(5)

length += 5

#Ex. 2-1

def mySum2(num):

running\_sum = 0

for i in range(1,num+1):

running\_sum += i

return running\_sum

mySum2(100) # 5050

mySum2(1000) # 500500

#Ex. 2-2

def average3(numList):

return sum(numList)/len(numList)

d = [53, 28, 54, 84, 65, 60, 22, 93, 62, 27, 16, 25, 74, 42, 4, 42,

15, 96, 11, 70, 83, 97, 75]

print(average3(d)) #Answer: 52.08695652173913

Ex. 3-1

def factors(num):

'''returns a list of the factors of num'''

factorList = []

for i in range(1,num+1):

if num % i == 0:

factorList.append(i)

return factorList

def gcf(a,b):

'''Returns the Greatest Common Factor of a and b'''

#first put all the factors of a and b in lists:

factors\_of\_a = factors(a)

factors\_of\_b = factors(b)

#create a list of "common factors"

common\_factors = []

#go through all the factors of a

for f in factors\_of\_a:

#if the factor is also a factor of b

if f in factors\_of\_b:

#add it to the common factors list

common\_factors.append(f)

return max(common\_factors) # or common\_factors[-1]

print(gcf(150,138))

def average(a,b):

return (a + b)/2

def squareRoot(num,low,high):

'''Finds the square root of num by

playing the Number Guessing Game

strategy by guessing over the

range from "low" to "high"'''

for i in range(20):

guess = average(low,high)

if guess\*\*2 == num:

print(guess)

elif guess\*\*2 > num: #"Guess lower."

high = guess

else: #"Guess higher."

low = guess

print(guess)

Ex. 3-2

squareRoot(200,1,200) # 14.142157554626465

squareRoot(1000,30,40) # 31.62278175354004

squareRoot(50000,100,500) # 223.6064910888672

Ex. 4-1

def equation(a,b,c,d):

''''solves equations of the

form ax + b = cx + d'''

return (d - b)/(a - c)

print(equation(12,18,-34,67)) # Answer: 1.065217391304348

Ex 4-2

print(equation(1/2,2/3,1/5,7/8)) # Answer: 0.6944444444444446

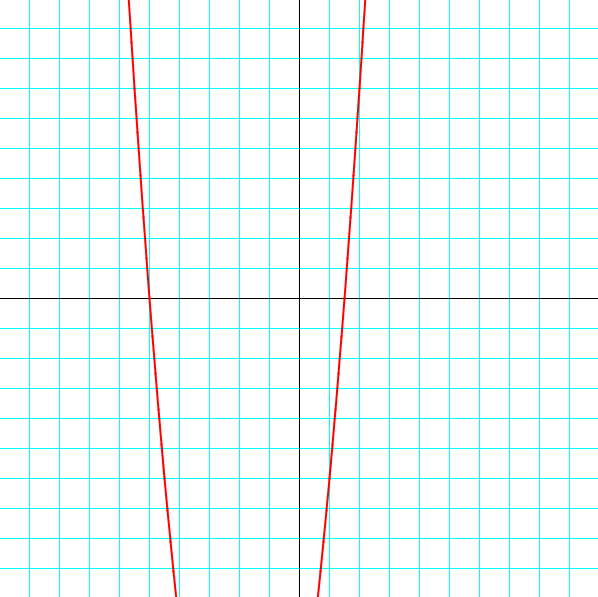
Ex. 4-3

In grid.pyde and guess.py, change f(x) to this:

def f(x):

return 2\*x\*\*2 + 7\*x – 15

Run grid.pyde. The graph shows 2 solutions:



Looks like the first one is x = -5. Is it?

>>> f(-5)

0

Yes. The second solution is between x = 1 and x = 2. Change the guess.py code to this:

'''The guess method'''

def f(x):

return 2\*x\*\*2 + 7\*x - 15

def average(a,b):

return (a + b)/2.0

def guess():

lower = 1

upper = 2

for i in range(20):

midpt = average(lower,upper)

if f(midpt) == 0:

return midpt

elif f(midpt) > 0: # changed to "greater than"

upper = midpt

else:

lower = midpt

return midpt

x = guess()

print(x,f(x))

The output is

1.5 0.0

So the other solution is x = 1.5. If you check it using the quad function from Listing 4-4:

def quad(a,b,c):

'''Returns the solutions of an equation

of the form a\*x\*\*2 + b\*x + c = 0'''

x1 = (-b + sqrt(b\*\*2 - 4\*a\*c))/(2\*a)

x2 = (-b - sqrt(b\*\*2 - 4\*a\*c))/(2\*a)

return x1,x2

print(quad(2,7,-15))

The output is (1.5, -5.0). Check!

Ex. 5-1

Take the code from geometry.pyde and add the tri function from triangles.pyde:

t = 0

def tri(length):

'''Draws an equilateral triangle

around center of triangle'''

triangle(0,-length,

-length\*sqrt(3)/2, length/2,

length\*sqrt(3)/2, length/2)

def setup():

size(600,600)

rectMode(CENTER)

def draw():

global t

#set background white

background(255)

translate(width/2,height/2)

for i in range(12):

pushMatrix() #save this orientation

translate(200,0)

rotate(radians(t))

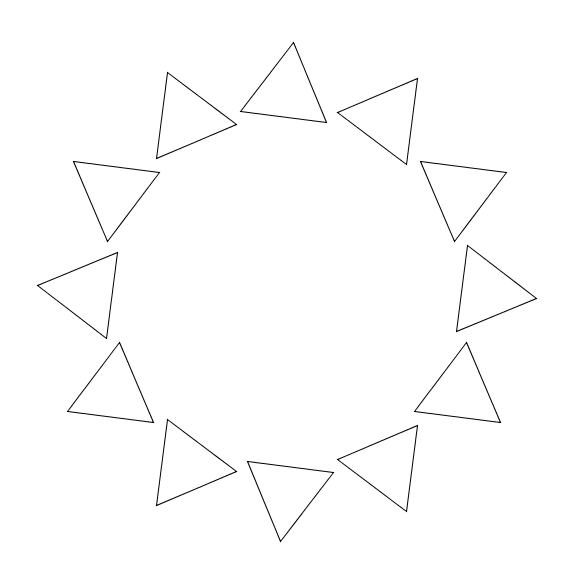
tri(50) #this line is changed from geometry.pyde

popMatrix() #return to the saved orientation

rotate(radians(360/12))

t += 1

Here’s the output:



Ex. 5-2

I added a few lines in triangles.pyde to make the lines thicker, the colorMode HSB, and finally to add color to each triangle according to where it is in the for i in range loop.

def setup():

size(600,600)

rectMode(CENTER)

colorMode(HSB) #for the rainbow colors

strokeWeight(2) #a little thicker line

t = 0

def draw():

global t

background(255)#white

translate(width/2,height/2)

for i in range(90):

#space the triangles evenly

#around the circle

rotate(radians(360/90))

pushMatrix() #save this orientation

#go to circumference of circle

translate(200,0)

#add color to each triangle

stroke(2\*i,255,255)

#spin each triangle

rotate(radians(t+2\*i\*360/90))

#draw the triangle

tri(100)

#return to saved orientation

popMatrix()

t += 0.5

def tri(length):

noFill() #makes the triangle transparent

triangle(0,-length,

-length\*sqrt(3)/2, length/2,

length\*sqrt(3)/2, length/2)

Ex. 7-1

Change the complex number c to [0.285,0.01]. Here’s the whole code for julia.pyde:

#range of x-values

xmin = -2

xmax = 2

#range of y-values

ymin = -2

ymax = 2

#calculate the range

rangex = xmax - xmin

rangey = ymax - ymin

def cAdd(a,b):

'''adds two complex numbers'''

return [a[0]+b[0],a[1]+b[1]]

def cMult(u,v):

'''Returns the product of two complex numbers'''

return [u[0]\*v[0]-u[1]\*v[1],u[1]\*v[0]+u[0]\*v[1]]

def magnitude(z):

return sqrt(z[0]\*\*2 + z[1]\*\*2)

def julia(z,c,num):

'''runs the process num times

and returns the diverge count'''

count = 0

#define z1 as z

z1 = z

#iterate num times

while count <= num:

#check for divergence

if magnitude(z1) > 2.0:

#return the step it diverged on

return count

#iterate z

z1 = cAdd(cMult(z1,z1),c)

count += 1

return num

def setup():

global xscl, yscl

size(600,600)

colorMode(HSB)

noStroke()

xscl = width/float(rangex)

yscl = height/float(rangey)

def draw():

#origin in center:

translate(width/2,height/2)

#go over all x's and y's on the grid

x = xmin

while x < xmax:

y = ymin

while y < ymax:

z = [x,y]

c = [0.285,0.01]

#put it into the julia program

col = julia(z,c,100)

#if julia returns 100

if col == 100:

fill(0)

else:

#map the color from 0 to 100

#to 0 to 255

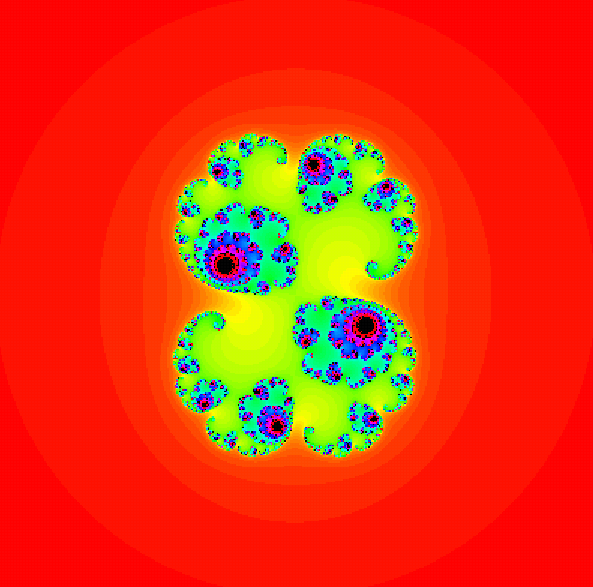
#coll = map(col,0,100,0,300)

fill(3\*col,255,255)

rect(x\*xscl,y\*yscl,1,1)

y += 0.01

x += 0.01



Ex. 8-1

In matrices.pyde, in the draw function, multiply the f-matrix by a, b and c. Here’s the whole code:

#set the range of x-values

xmin=-10

xmax=10

#range of y-values

ymin = -10

ymax = 10

#calculate the range

rangex = xmax - xmin

rangey = ymax - ymin

def setup():

global xscl, yscl

size(600,600)

#the scale factors for drawing on the grid:

xscl= width / rangex

yscl= -height / rangey

def draw():

global xscl, yscl

background(255) #white

translate(width/2,height/2)

grid(xscl, yscl)

#multiply fmatrix by chosen transformation matrix

newmatrix = multmatrix(fmatrix,a)

noFill()

strokeWeight(1) #thinner line

stroke(0) #black

graphPoints(fmatrix)

strokeWeight(2) #thicker line

stroke(255,0,0) #red

graphPoints(newmatrix)

fmatrix = [[0,0],[1,0],[1,2],[2,2],[2,3],[1,3],[1,4],

[3,4],[3,5],[0,5]]

transformation\_matrix = [[0,-1],[1,1]]

a = [[1,0],[0,-1]]

b = [[0,-1],[-1,0]]

c = [[-1,1],[1,1]]

def graphPoints(matrix):

#draw line segments between consecutive points

beginShape()

for pt in matrix:

vertex(pt[0]\*xscl,pt[1]\*yscl)

endShape(CLOSE)

def multmatrix(a,b):

#Returns the product of two matrices

#b is a 2x2

newmatrix = []

for i in range(len(a)):

newmatrix.append([])

for j in range(2):

newmatrix[i].append(a[i][0]\*b[0][j]+a[i][1]\*b[1][j])

return newmatrix

def graphPoints(matrix):

strokeWeight(2) #thicker line

#stroke(255,0,0) #red

#draw line segments between consecutive points

for i,pt in enumerate(matrix):

if i < len(matrix) - 1:

line(pt[0]\*xscl,pt[1]\*yscl,

matrix[i+1][0]\*xscl,matrix[i+1][1]\*yscl)

else: #connect first and last point

line(matrix[0][0]\*xscl,matrix[0][1]\*yscl,

matrix[-1][0]\*xscl,matrix[-1][1]\*yscl)

def grid(xscl, yscl):

'''Draws a grid for graphing'''

#cyan lines

strokeWeight(1)

stroke(0,255,255)

for i in range(xmin, xmax + 1):

line(i\*xscl, ymin\*yscl, i\*xscl, ymax\*yscl)

for i in range(ymin,ymax+1):

line(xmin\*xscl, i\*yscl, xmax\*xscl, i\*yscl)

stroke(0) #black axes

line(0,ymin\*yscl,0,ymax\*yscl)

line(xmin\*xscl,0, xmax\*xscl,0)

Here’s what the transformations look like:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Ex. 8-2

We’re plugging a 4x5 matrix into our “gauss” function and solving.

def gauss(A):

'''Converts a matrix into the identity

matrix by Gaussian elimination, with

the last column containing the solutions

for the variables'''

m = len(A)

n = len(A[0])

for j,row in enumerate(A):

#diagonal term to be 1

#by dividing row by diagonal term

if row[j] != 0: #diagonal entry can't be zero

divisor = row[j]

for i, term in enumerate(row):

row[i] = term / divisor

#add the other rows to the additive inverse

#for every row

for i in range(m):

if i != j: #don't do it to row j

#calculate the additive inverse

addinv = -1\*A[i][j]

#for every term in the ith row

for ind in range(n):

#add the corresponding term in the jth row

#multiplied by the additive inverse

#to the term in the ith row

A[i][ind] += addinv\*A[j][ind]

return A

def print\_matrix(A):

for row in A:

print(row)

#Here’s the matrix with the coefficients from the system of equations:

X = [[2,-1,5,1,-3],

[3,2,2,-6,-32],

[1,3,3,-1,-47],

[5,-2,-3,3,49]]

print\_matrix(gauss(X))

The output will be:

[1.0, 0.0, 0.0, 0.0, 2.000000000000001]

[0.0, 1.0, 0.0, 0.0, -12.0]

[0.0, 0.0, 1.0, 0.0, -4.0]

[0.0, 0.0, 0.0, 1.0, 1.0000000000000004]

The last values in every row are the solutions:

w = 2

x = -12

y = -4

z = 1

Ex. 9-1

Add this line to the \_\_init\_\_ method of the Ball class:

self.sz = random(5,50)

Ex. 9-2

Add this line to the \_\_init\_\_ method of the Sheep class to create an age property:

self.age = 0

Add this code to the update method of the Sheep class to increment the age and delete the sheep if they’re over 100 units old:

self.age += 1

if self.age > 100:

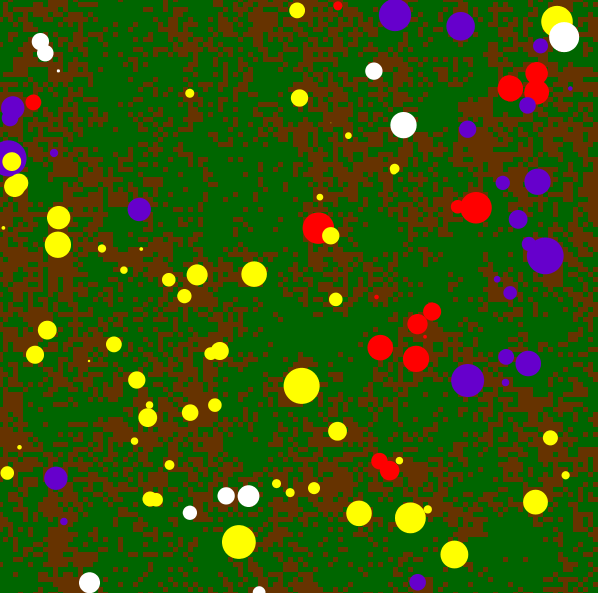
sheepList.remove(self)

Ex. 9-3

Add this line to the update method of the Sheep class just before drawing the ellipse:

self.sz = 0.75\*self.energy

It should look like this, with the Sheep having more energy looking bigger!



Ex. 11-1

Add the mouseClicked function at the end of the code to update the cellList manually:

GRID\_W = 51

GRID\_H = 51

#size of cell

SZ = 18

class Cell:

def \_\_init\_\_(self,c,r,state=0):

self.c = c

self.r = r

self.state = state

def display(self):

if self.state == 1:

fill(0) #black

else:

fill(255) #white

rect(SZ\*self.r,SZ\*self.c,SZ,SZ)

def checkNeighbors(self):

if self.state == 1: return 1 #on Cells stay on

neighbs = 0

#check the neighbors

for dr,dc in [[-1,0],[1,0],[0,-1],[0,1]]:

try:

if cellList[self.r + dr][self.c + dc].state == 1:

neighbs += 1

except IndexError:

continue

if neighbs in [1,4]:

return 1

else:

return 0

def setup():

global SZ, cellList

size(600,600)

SZ = width // GRID\_W + 1

cellList = createCellList()

def draw():

global cellList

for row in cellList:

for cell in row:

cell.display()

def createCellList():

'''Creates a big list of off cells with

one on Cell in the center'''

newList=[]#empty list for cells

#populate the initial cell list

for j in range(GRID\_H):

newList.append([]) #add empty row

for i in range(GRID\_W):

newList [j].append(Cell(i,j,0)) #add off Cells or zeroes

#center cell is set to on

newList [GRID\_H//2][GRID\_W//2].state = 1

return newList

def update(cellList):

newList = []

for r,row in enumerate(cellList):

newList.append([])

for c,cell in enumerate(row):

newList[r].append(Cell(c,r,cell.checkNeighbors()))

return newList[::]

def mouseClicked():

global cellList

cellList = update(cellList)

Ex. 11-2

Simply change the ruleset list in the elementaryCA.pyde file to this:

ruleset = [0,1,0,1,1,0,1,0] #rule 90

Here’s what the whole code looks like:

w = 3

rows = 1000

cols = 1000

#ruleset = [0,0,0,1,1,1,1,0]

ruleset = [0,1,0,1,1,0,1,0] #rule 90

def rules(a,b,c):

return ruleset[7-(4\*a+2\*b+c)]

def generate():

for i, row in enumerate(cells):#look at first row

for j in range(1,len(row)-1):

left = row[j-1]

me = row[j]

right = row[j+1]

if i < len(cells) - 1:

cells[i+1][j] = rules(left,me,right)

return cells

def setup():

global cells

size(600,600)

noStroke()

#first row:

cells = []

for r in range(rows):

cells.append([])

for c in range(cols):

cells[r].append(0)

cells[0][cols//2] = 1

cells = generate()

def draw():

global cells

background(255) #white

#draw the CA

for i, cell in enumerate(cells): #rows

for j, v in enumerate(cell): #columns

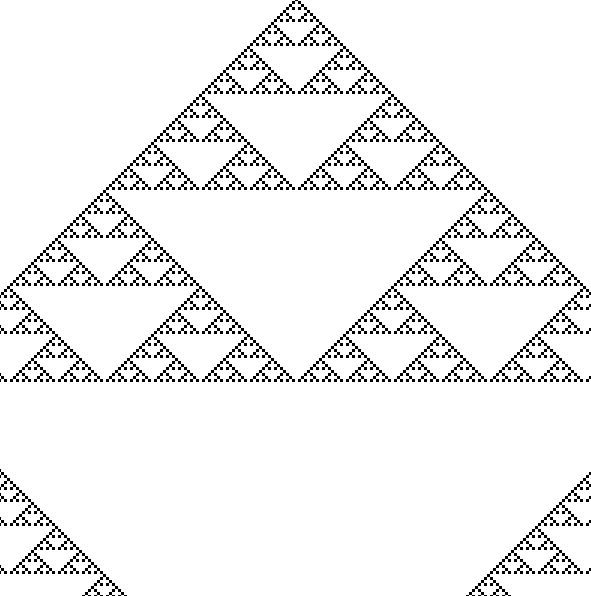
if v == 1:

fill(0)

else: fill(255)

rect(j\*w-(cols\*w-width)/2,w\*i,w,w)

Run this and the CA will change to look like the Sierpinski Triangle:



Exercise 11-3

Add the keyPressed function to the end of elementaryCA.pyde to change w, the size of each cell.

def keyPressed():

global w

if key == CODED:

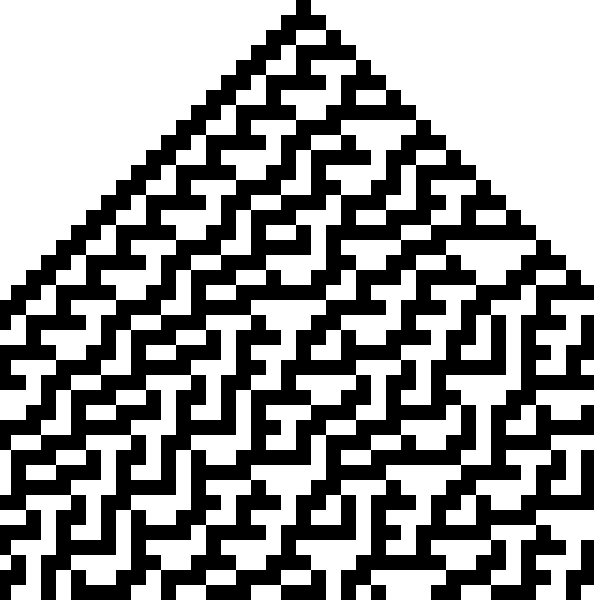
if keyCode == UP:

w += 1

if keyCode == DOWN:

w -= 1

Run it and you’ll be able to zoom in to the CA. (First you might have to click in the display window.)



The down arrow will reduce the size of each cell and let you zoom out:

