Review

- Converting color images to grayscale
- Thresholding
- Other single-pixel filters: Negative, Sepia, …
- Histogram Equalization
- Spatial Filtering
  - Identity, Average, High Pass, Low Pass (Gaussian Blur)
  - Erosion, Dilation – Useful for noise removal and feature detection
- Applications
  - Measuring Cell Culture Confluency
  - Vision-Guided Robotics
  - Detecting Error Conditions
  - Facial Recognition

Assignment 5

- ImageProcessing.py:
  - Collected image processing functions
  - Each takes two images: img1 and img2
  - img1 is processed, img2 is the result of image processing

```
grayscale(img1, img2)
threshold(cutoff, img1, img2)
negative( img1, img2 )
sepia( img1, img2 )
spatial( matrix, img1, img2 )
erode(img1, img2)
dilate(img1, img2)
```

- To use:

  Copy ImageProcesing.py to same folder as your program, then import.

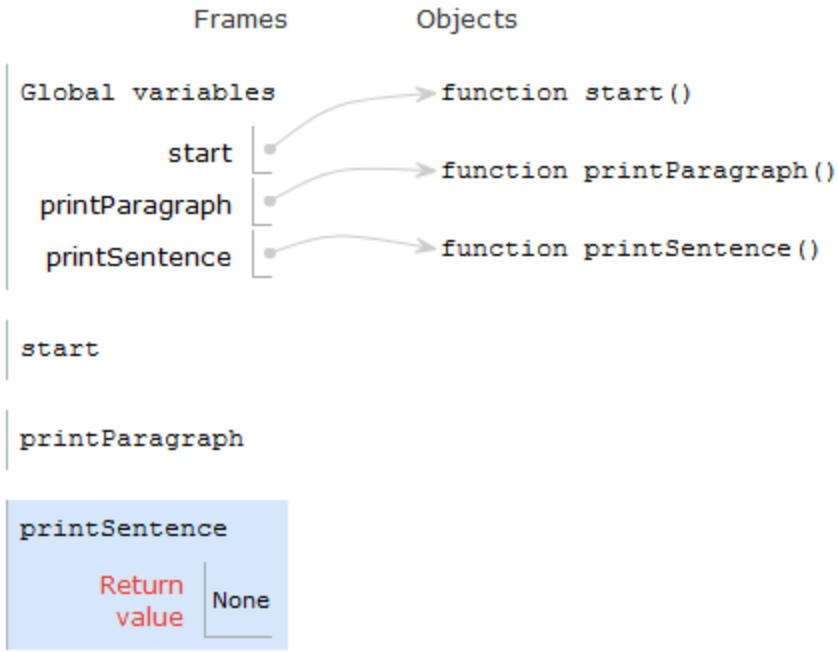
```
from ImageProcessing import *
```

Objects vs. Functions

Objects	Functions
class statement defines how to create a new object	def statement defines how to create a new function
Objects have state: • instance variables + values	Function "instances" have state: • local variables + values • current line being executed
New object instances are created by invoking constructor	New function "instances" are created when function is called

Function Call Tracing

```
1  # paragraph.py
2  def start():
3    print("Before paragraph.")
4    printParagraph()
5    print("After paragraph.")
6
7  def printParagraph():
8    print("Before sentence.")
9    printSentence()
10   print("After sentence.")
11
12 def printSentence():
13   print("Go!")
14
15 start()
```




```
Before paragraph.
Before sentence.
Go!
After sentence.
After paragraph.
```

paragraph.py

Factorial

- The factorial of a positive integer N is computed as the product of N with all positive integers less than or equal to N.

  $4! = 4 \times 3 \times 2 \times 1 = 24$

  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

  $30! = 30 \times 29 \times \ldots \times 2 \times 1 =$
  $265252859812191058636308480000000$

Factorial – Iterative Implementation

```
1.   def factorial( N ):
2.       F = 1
3.       i = N

4.       while i > 1:
5.           F = F * i
6.           i -= 1

7.       return F

8.   B = factorial(5)
9.   print( B )
```

Trace it.

factorialIterative.py

$5! = 5 \times 4 \times 3 \times 2 \times 1$

$4! = 4 \times 3 \times 2 \times 1$

$5! = 5 \times 4!$

$N! = N \times (N-1)!$

Factorial can be defined in terms of itself

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$5! = 5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1$

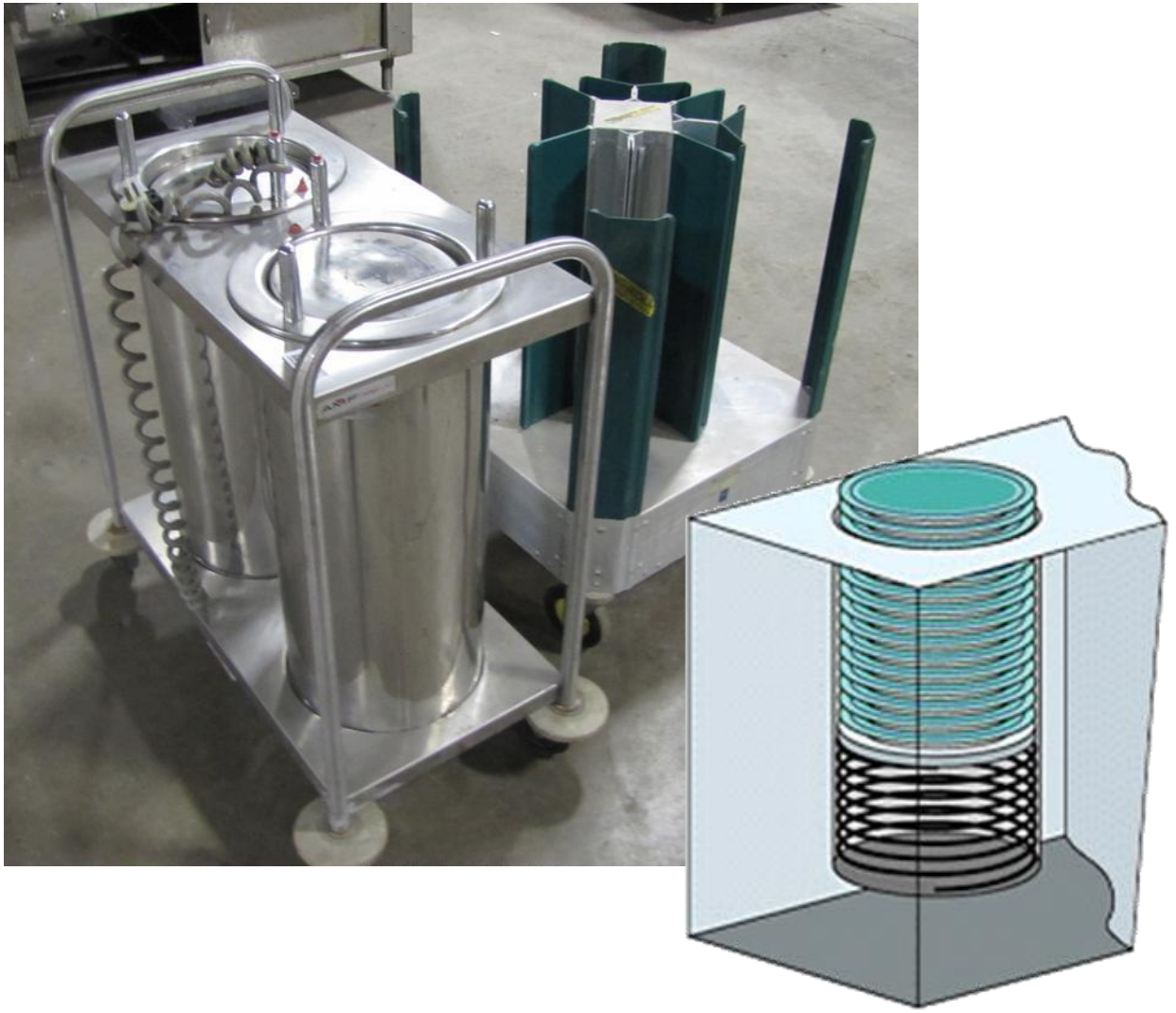
Factorial – Recursive Implementation

```
1.   def factorial( i ):

2.       if i == 0:
3.           return 1
4.       else:
5.           return i*factorial(i-1)

6.   f = factorial(10)
7.   print(f)
```

Trace it.

factorial.py

Last In First Out (LIFO) Stack of Plates



```
1  def factorial( i ):
2      if i == 0:
3          return 1
4      else:
5          return i*factorial(i-1)
6
7  f = factorial(5)
8  print(f)
```

Edit code

<< First | < Back | Step 14 of 21 | Forward > | Last >>

that has just executed
line to execute

output:

Frames Objects

Global variables
factorial → function factorial(i)

factorial
i 5

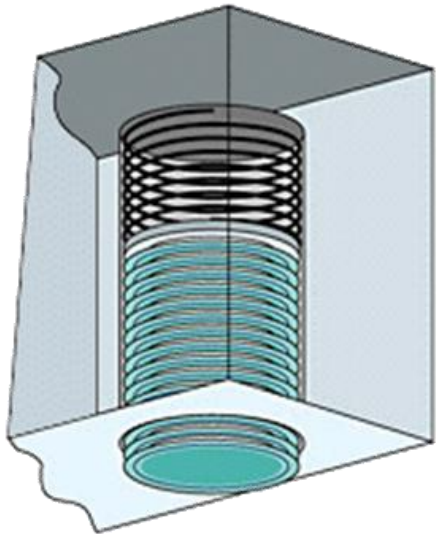
factorial
i 4

factorial
i 3

factorial
i 2

factorial
i 1

factorial
i 0



factorial.py

The Call Stack keeps track of …

1. all functions that are suspended, in the reverse order in which they were suspended (LIFO)
2. the point in the function where execution should resume after the invoked subordinate function returns
3. a snapshot of all variables and values within the scope of the suspended function so these can be restored upon continuing execution

```
# fibonacci.py
# Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
# Compute Fibonocci sequence recursively

# Compute and return the nth Fibonacci number
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        f = fibonacci(n-1) + fibonacci(n-2)
        return f

f = fibonacci(12)
print(f)
```

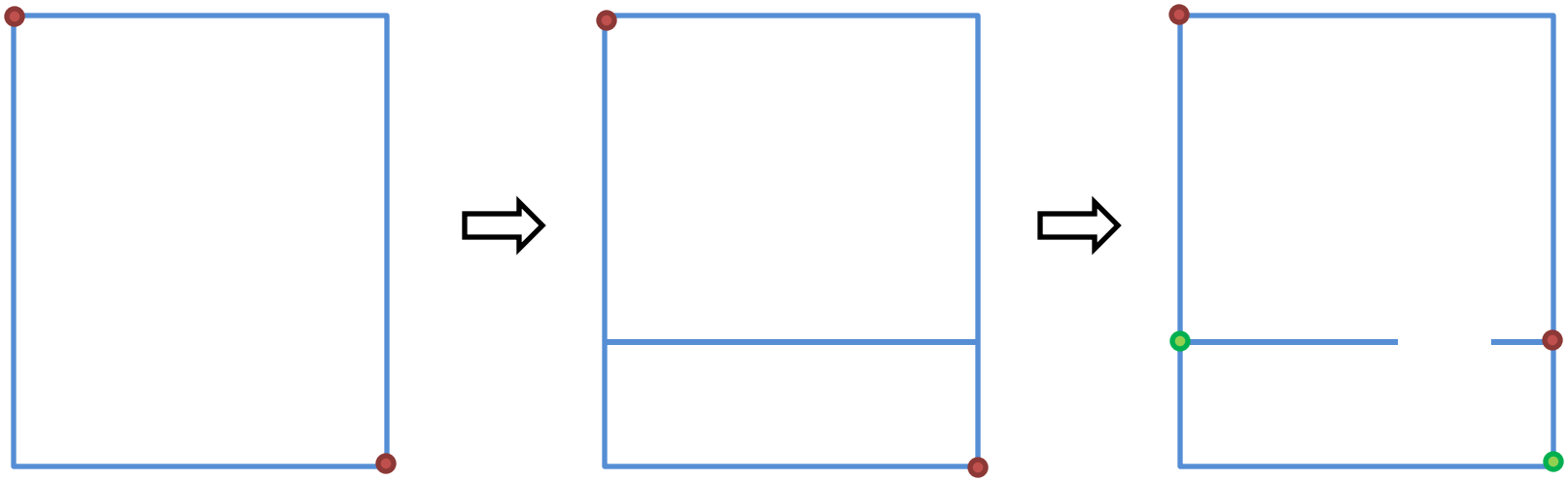
fibonacci.py

Recursive Function Model

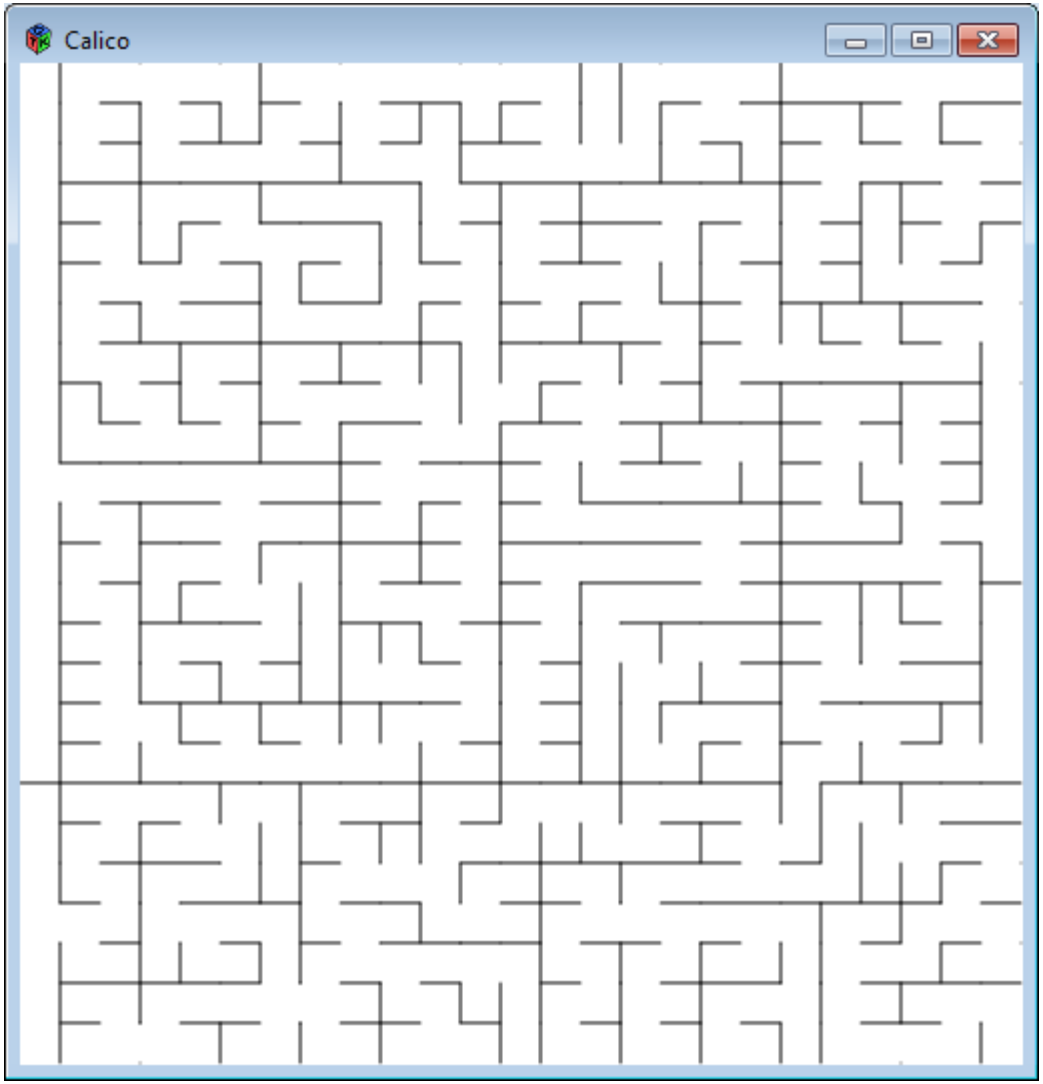
```
# Model of a recursive function

def recursiveFunction( args ):

    if base_case_condition :
        return base_case_value

    else:
        modified_args = (some modification of args)
        f = recursiveFunction( modified_args )
        return f

f = recursiveFunction( args )
print(f)
```

Creating a maze, recursively

1. Start with a rectangular region defined by its upper left and lower right corners
2. Divide the region at a random location through its more narrow dimension
3. Add an opening at a random location
4. Recursively, repeat on two rectangular subregions



Inspired by http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm



recursiveMaze.py

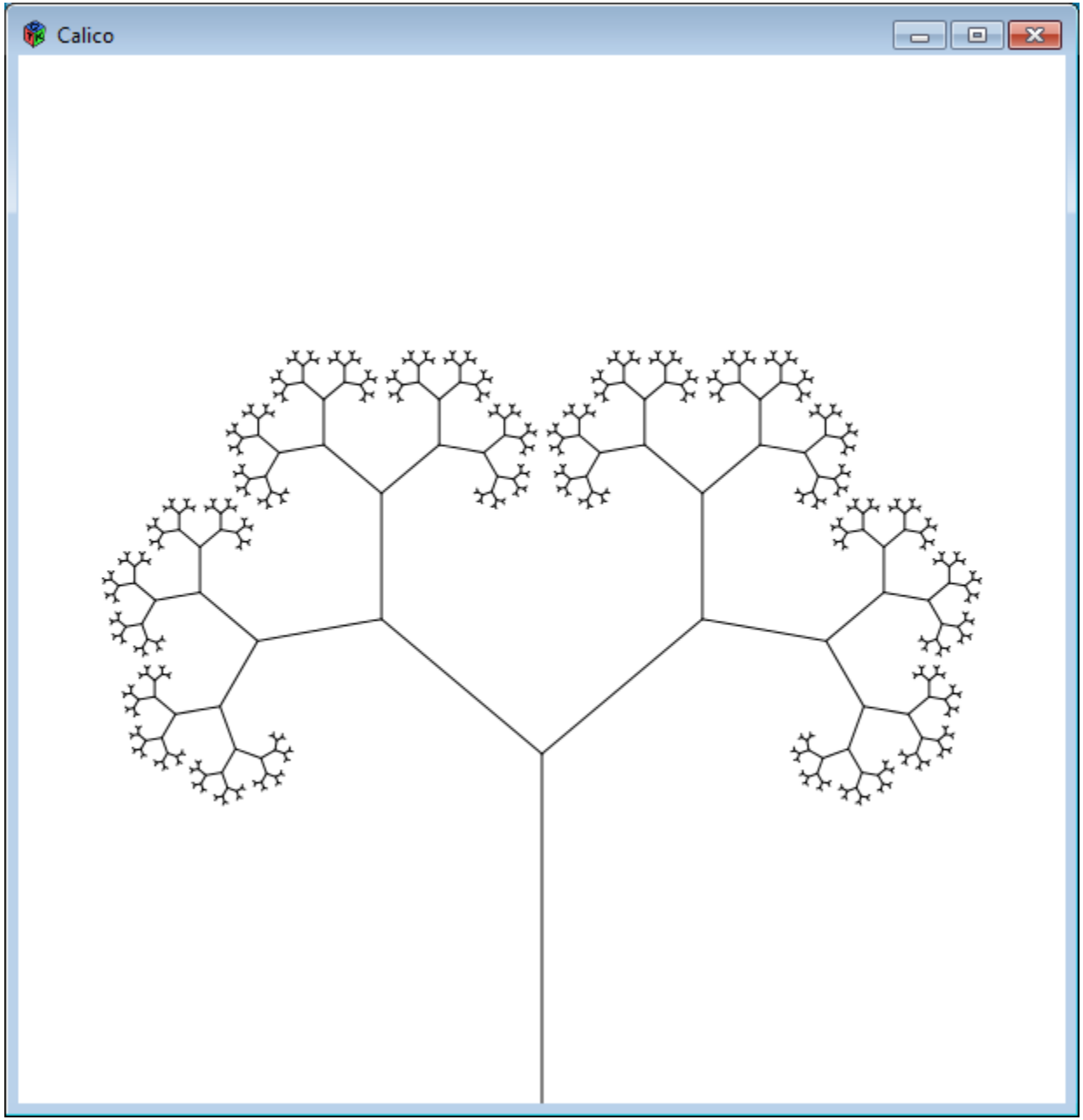
```
# RecursiveMaze
from Processing import *
N = 25          # Grid dimension (# cells)
gsize = 20      # Grid cell size (pixels)
vertical = 1    # Vertical direction constant
horizontal = 2  # Horizontal direction constant

window( N*gsize+1, N*gsize+1 )
noLoop()
background(255)
stroke(0)

# Determine the direction for dividing the region
# based on which dimension is smaller. Stop when too small.
def divDirection(r1, c1, r2, c2):
    dr = r2 - r1                # Calculate deltas
    dc = c2 - c1
    if dr <= 1 or dc <= 1:      # Too small
        return 0                # No division
    elif dr < dc:               # Flat and wide
        return vertical         # Vertical division
    else:                       # Tall and narrow
        return horizontal       # Horizontal division

# Return a random integer in the range [min, max]
def randomInt(min, max):
    return round(random(min-0.5, max+0.5))

# Draw a line on a grid segment given grid points
def gridLine(r1, c1, r2, c2):
    line(r1*gsize, c1*gsize, r2*gsize, c2*gsize)
```

recursiveMaze.py

```
# Divide the region given upper left and lower right grid corner points
def divide(r1, c1, r2, c2):

    direction = divDirection(r1, c1, r2, c2) # Get divide direction (V, H or 0)

    if direction == vertical:                # Divide in vertical direction
        cr = randomInt(c1+1, c2-1)           # Calculate wall and opening locations
        rr = randomInt(r1, r2-1)

        gridLine(cr,r1,cr,rr)                # Draw wall
        gridLine(cr,rr+1,cr,r2)

        divide(r1,c1,r2,cr)                  # Recursively divide two subregions
        divide(r1,cr,r2,c2)

    elif direction == horizontal:            # Divide in horizontal direction
        cr = randomInt(c1, c2-1)             # Calculate wall and opening locations
        rr = randomInt(r1+1, r2-1)

        gridLine(c1,rr,cr,rr)                # Draw wall
        gridLine(cr+1,rr,c2,rr)

        divide(r1,c1,rr,c2)                  # Recursively divide two subregions
        divide(rr,c1,r2,c2)

     else:                                   # No division. We're done.
        return

# Kick off the recursive divide on the whole sketch
divide(0,0,N,N)
```

recursiveMaze.py

Recursive Object Construction



Recursive Object Construction

```
# fractalTree.py
from Processing import *
window(600, 600)
background(255)

class FractalTree:
    def __init__(self, length, depth):
        self.length = length
        self.left, self.right = None, None

        # Not leaf. Grow further.
        if depth > 1:
            depth -= 1
            self.left  = FractalTree(0.6*length, depth)
            self.right = FractalTree(0.6*length, depth)

…

f = FractalTree(-200, 10)

translate(300, 600)
f.draw(50)

print(f.countBranches())
```

fractalTree.py

Recursive Object Rendering

```
# Recursively draw tree
def draw(self, angle):
    stroke(0)
    line(0, 0, 0, self.length)
    if self.left != None and self.right != None:
        translate(0, self.length)
        pushMatrix()
        rotate( radians(angle) )
        self.left.draw(angle)
        popMatrix()
        pushMatrix()
        rotate( radians(-angle) )
        self.right.draw(angle)
        popMatrix()

# Recursively count the depth of the tree
def countBranches(self):
    if self.left == None or self.right == None:
        return 1
    else:
        countLeft = self.left.countBranches()
        countRight = self.right.countBranches()
        return 1 + countLeft + countRight
```

fractalTree.py

Recursive Function Model

```
# Model of a recursive function

def recursiveFunction( args ):

    if base_case_condition :
        return base_case_value

    else:
        modified_args = (some modification of args)
        f = recursiveFunction( modified_args )
        return f

f = recursiveFunction( args )
print(f)
```