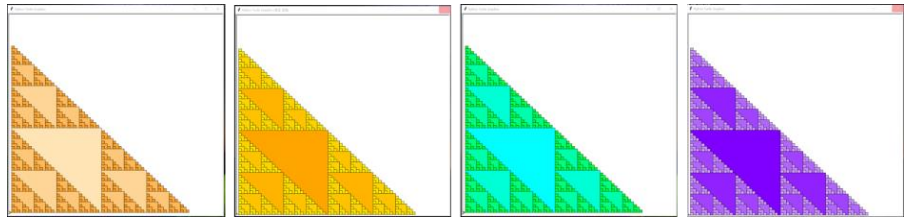
Data Structures in Python
Chapter 4

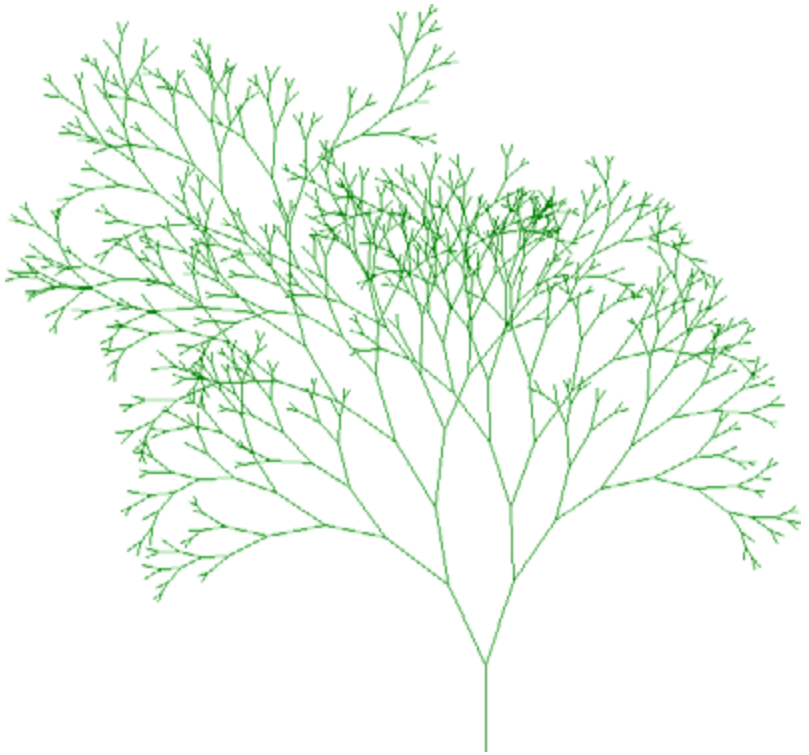
1. Recursion Concepts
2. Recursion Stack and Memoization
3. Recursive Algorithms
4. **Recursive Graphics**
   Exercise - Stacking boxes



Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

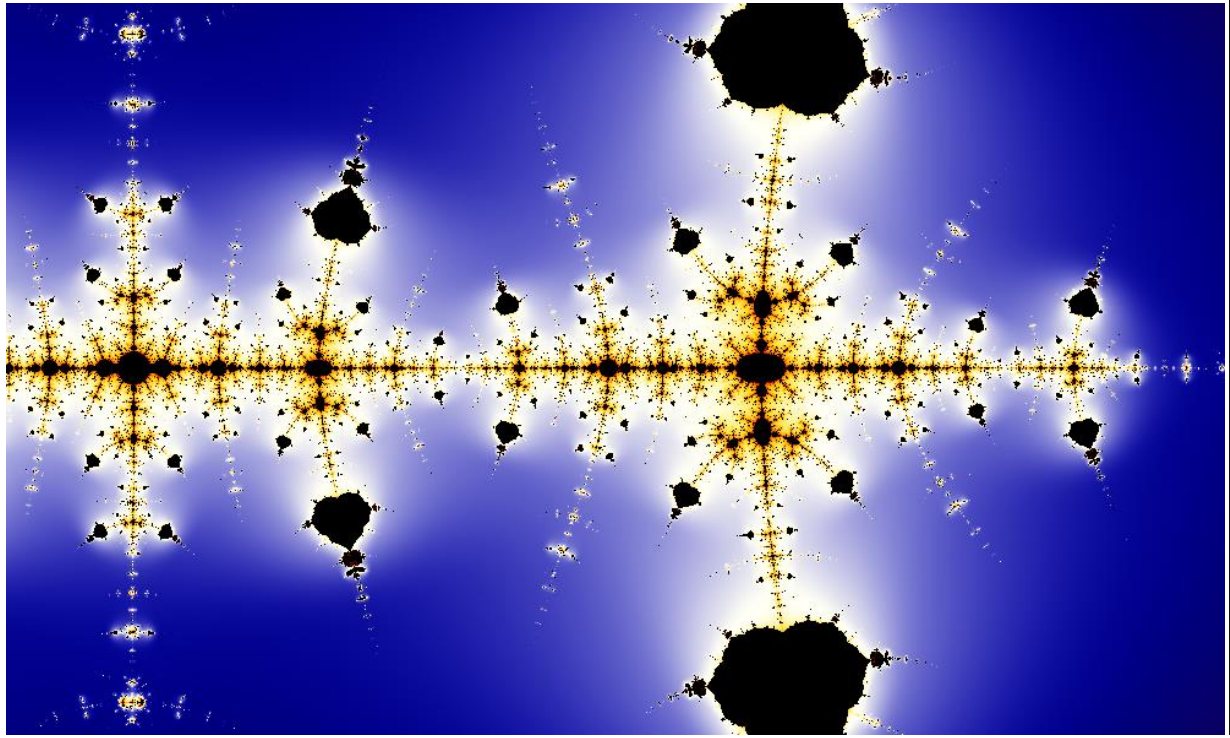
Agenda & Readings

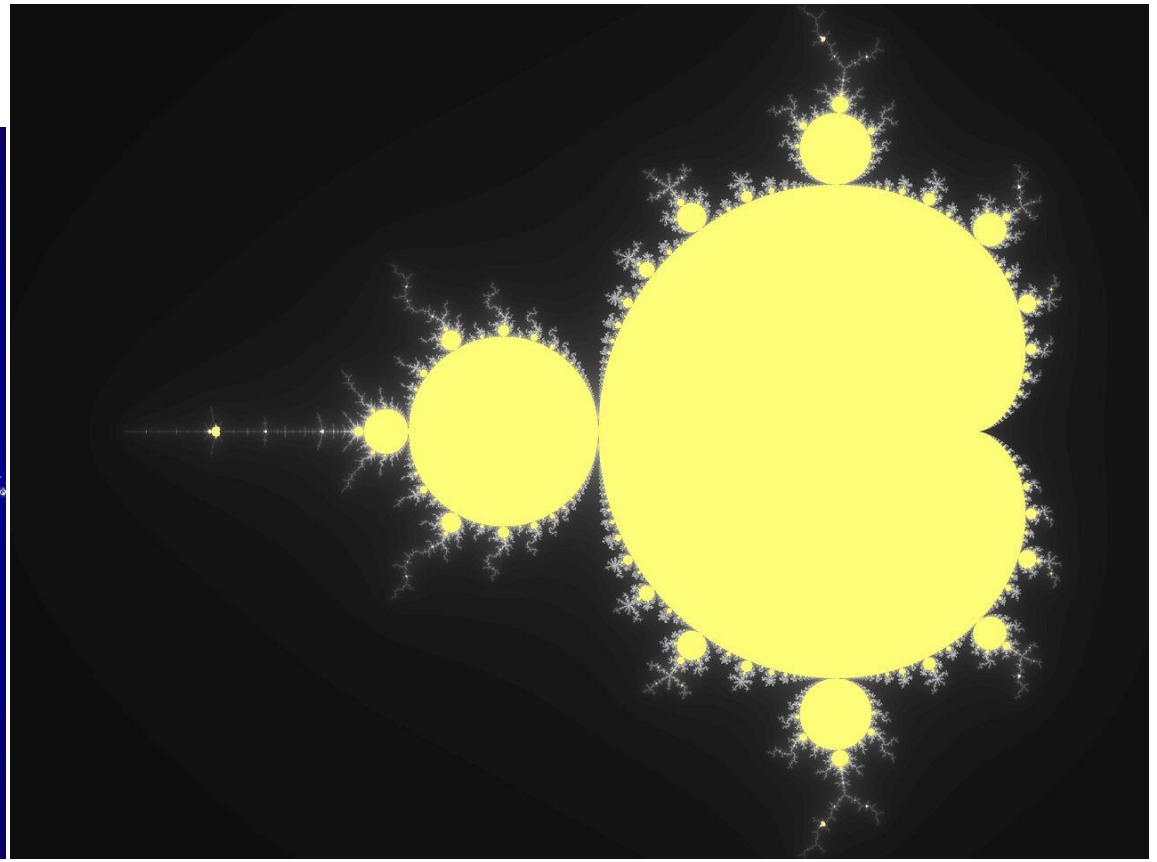
- Introduction
- Using the Python Turtle
- Drawing the Koch Snowflake
- Using recursive drawing



Introduction - Self-similarity

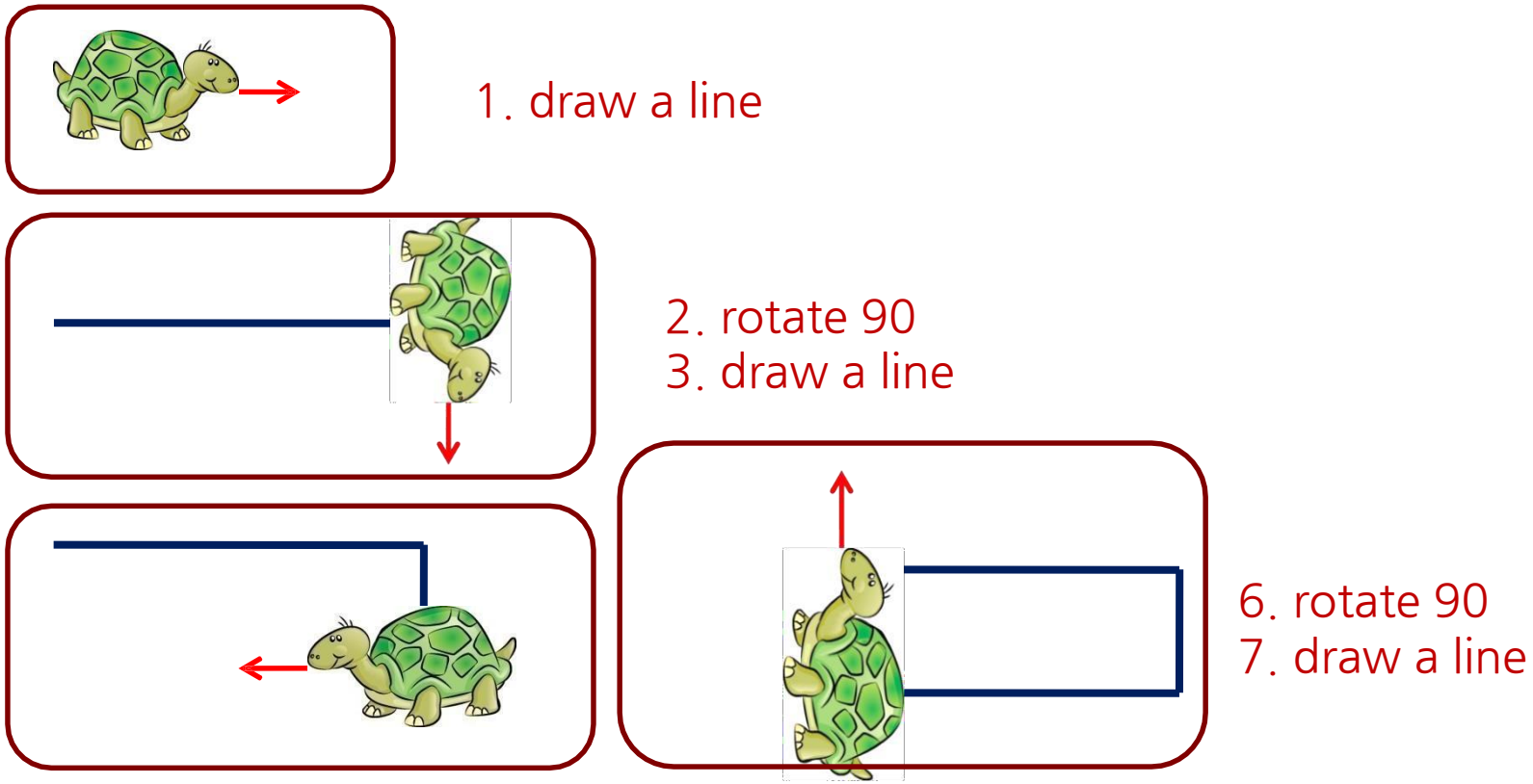
- A **fractal** is a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole.
  - This a property is called **self-similarity** (자기 유사성).





Python Turtle Class

- Can be drawn using a “Turtle”
  - Named after Logo programming language
  - Pen location used to draw on the screen
  - Commands
    - Pen up
    - Pen down
    - Rotate …



Python Turtle Class

- Steps:
  - Import the turtle module which defines the Turtle and the Screen types.

```
import turtle
```

  - Create and open a window.

```
win = turtle.Screen()
```

    - The window contains a canvas, which is the area inside the window on which the turtle draws.
  - Create a turtle object which can move forward, backwards, turn left, turn right, the turtle can have its tail up/down.

```
tom = turtle.Turtle()
```

  - If the tail is down, the turtle draws as it moves.
    - The width and color of the turtle tail can be changed.
  - When the user clicks somewhere in the window, the turtle window closes, and execution of the Python program stops.

```
turtle.exitonclick()
```

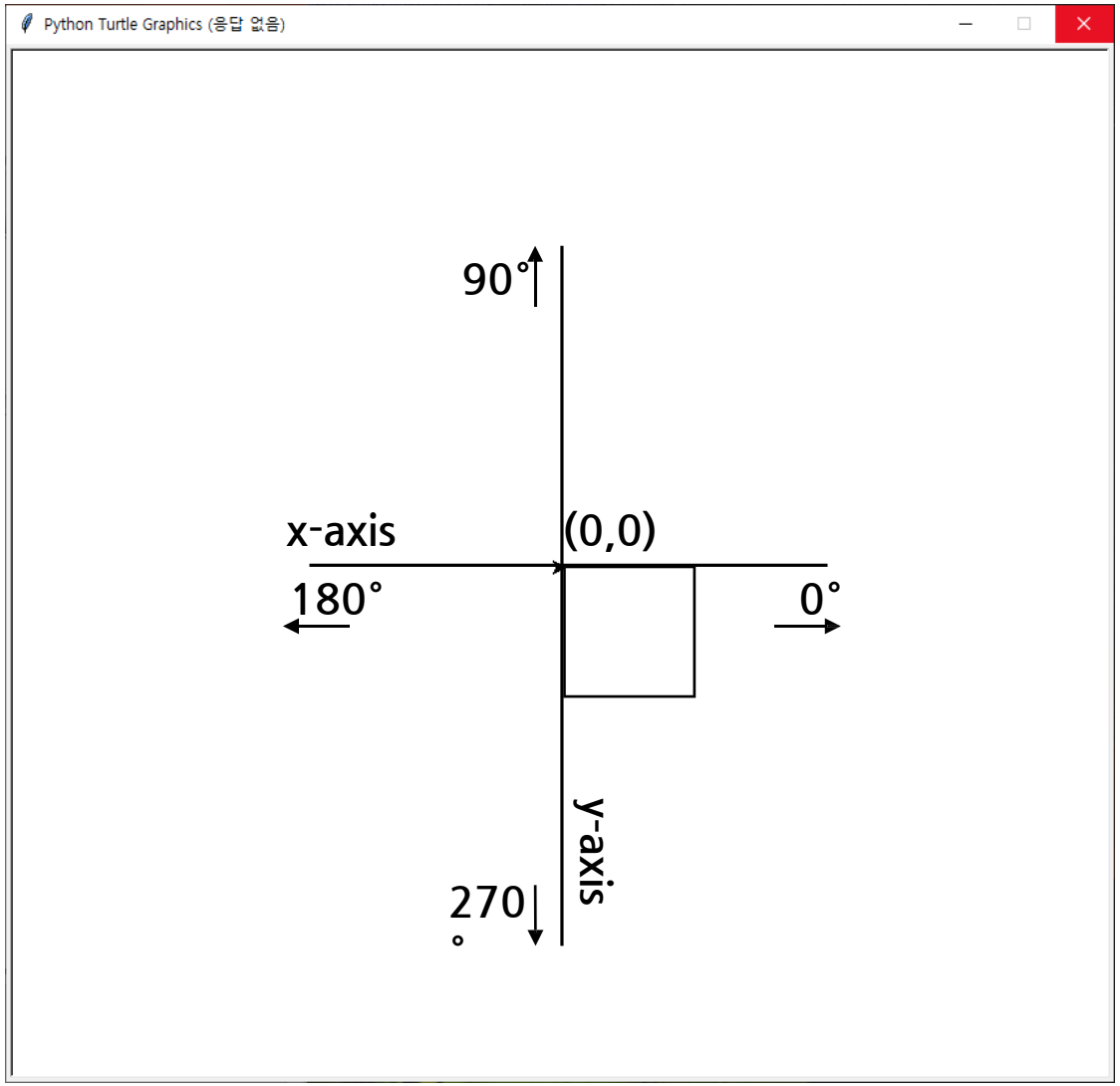
```
turtle.done()
```

Python Turtle Class

- Instantiate a Turtle object:

```
tom = turtle.Turtle()
```

- The turtle appears as an icon
  - Initial position: (0, 0)
  - Initial direction: East (0°)
  - Color: black
  - Line width: 1
  - pixel Pen: down (ready to draw)

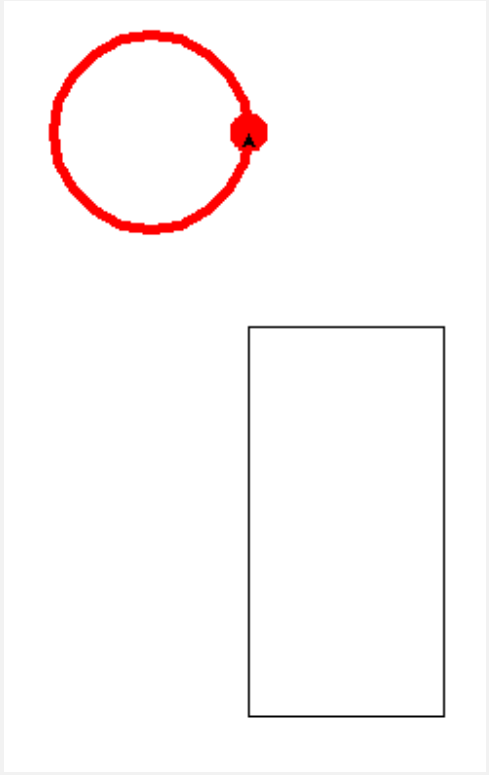


Python Turtle Class - Methods

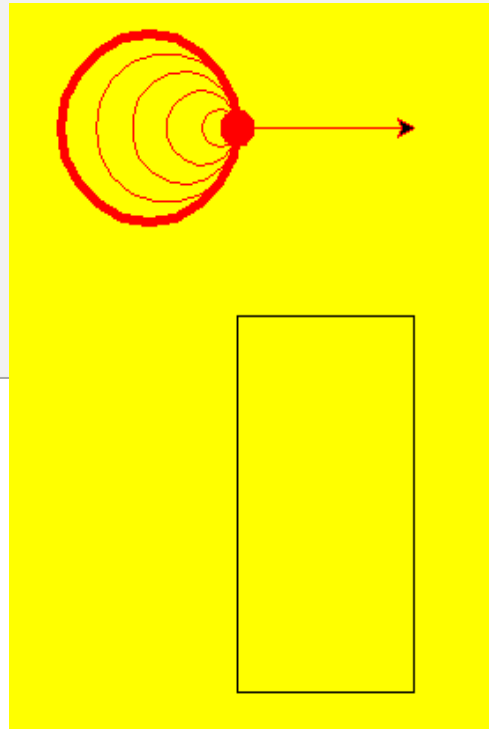
- `forward(distance)` - move the turtle forward
- `backward(distance)` - move the turtle backwards
- `right(angle)` - turn the turtle clockwise
- `left(angle)` - turn the turtle anti-clockwise
- `up()` - puts the turtle tail/pen up, i.e., no drawing
- `down()` - puts the turtle tail/pen down, i.e., drawing
- `pencolor(color_name)` - changes the color of the turtle's tail
- `heading()` - returns the direction in which the turtle is pointing
- `setheading(angle)` - set the direction in which the turtle is pointing
- `position()` - returns the position of the turtle
- `goto(x, y)` - moves the turtle to position x, y
- `speed(number)` - set the speed of the turtle movement

Python Turtle - Drawing Examples

```
import turtle
tom = turtle.Turtle()
tom.forward(100)
tom.right(90)
tom.forward(200)
tom.right(90)
tom.forward(100)
tom.right(90)
tom.forward(200)

tom.up()        # pen up
tom.forward(100)

tom.down()      # pen down
tom.pensize(5)
tom.pencolor('red')
tom.circle(50) # radius=50

tom.dot(20)
```

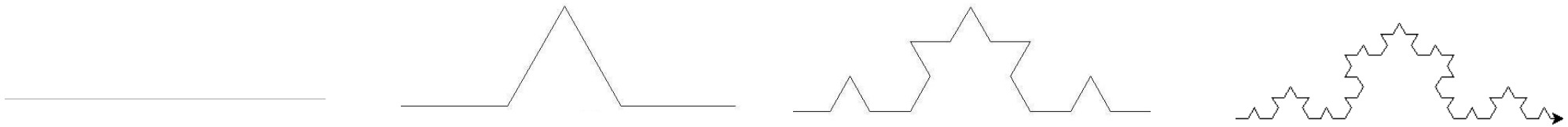


```
turtle.bgcolor('yellow')
tom.pensize(1)
n = 10
while n < 50:
    tom.circle(n)
    n += 10
tom.rt(90)
tom.fd(100)
```



Recursive graphics - Koch curves

- A Koch curve of order 0 is a straight line.
- To form a Koch curve of order n, draw a Koch curve order n - 1, turn left 60 degrees, draw a second Koch curve of order n - 1, turn right 120 degrees (left -120 degrees), draw a third order n - 1.
- These recursive instructions lead immediately to turn into a turtle code.
- These recursive schemes have proved useful in modeling self-similar patterns found in nature, such as snowflakes, trees, etc.

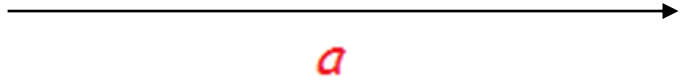


order 0 order 1 order 2 order 3

Recursive graphics - Koch curves

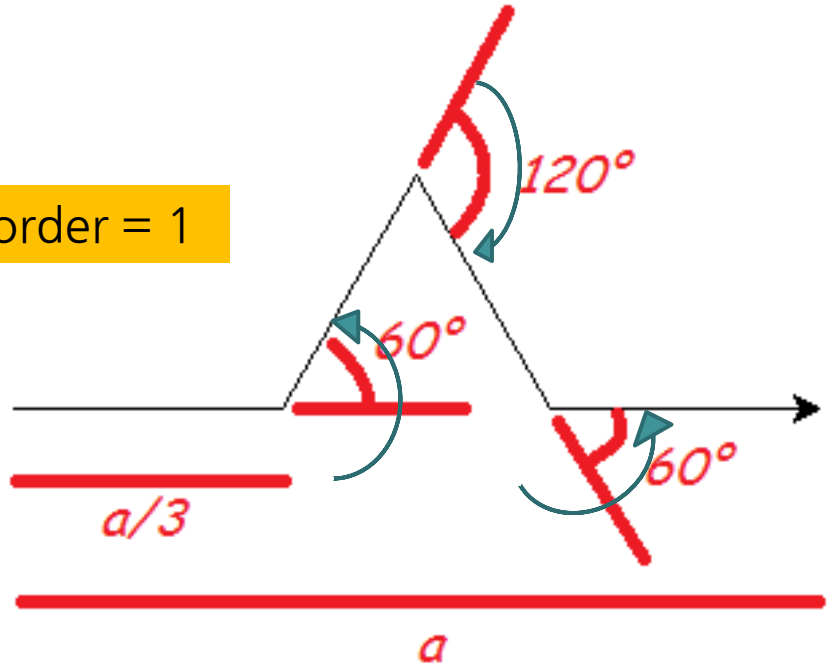
- We're going to define a function that either draws a line with a kink in it or draws a straight line the same length.
  - Simplify the "order=1 & 2" cases using a function and recursion:

order = 0




```
forward(a)
```

order = 1

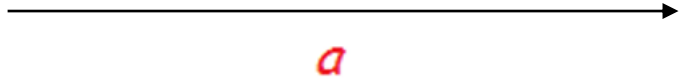


Recursive graphics - Koch curves

- Simplify the "order=1 & 2" cases using a function:

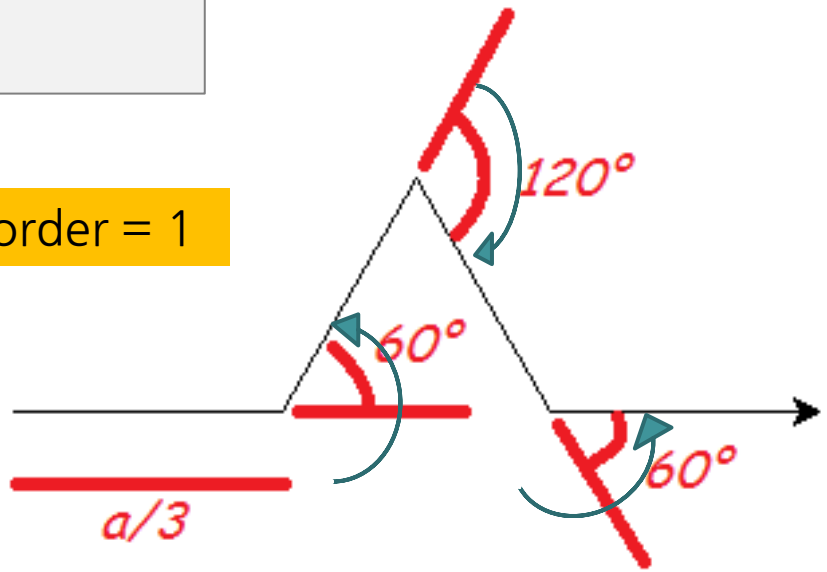
```
from turtle import *

def koch(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            forward(a/3)
            left(t)
    else:
        forward(a)
```

order = 0




```
forward(a)
```

order = 1




```
from turtle import *
a = 100

for t in [60, -120, 60, 0]:
    forward(a/3)
    left(t)
```

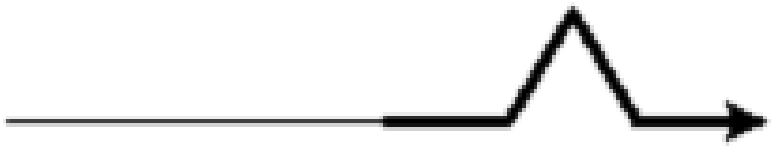
```
from turtle import *
a = 100

forward(a/3)
left(60)

forward(a/3)
right(120)

forward(a/3)
left(60)

forward(a/3)
```

Recursive graphics - Koch curves

- Sample Run:

```
from turtle import *

def koch(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            forward(a/3)
            left(t)
    else:
        forward(a)
```

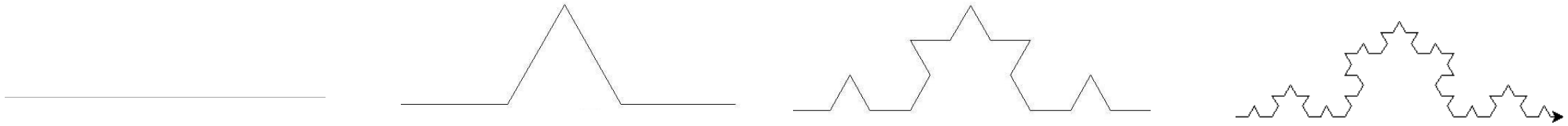
```
# Sample Run
koch(100, 0)
pensize(3)
koch(100, 1)
```



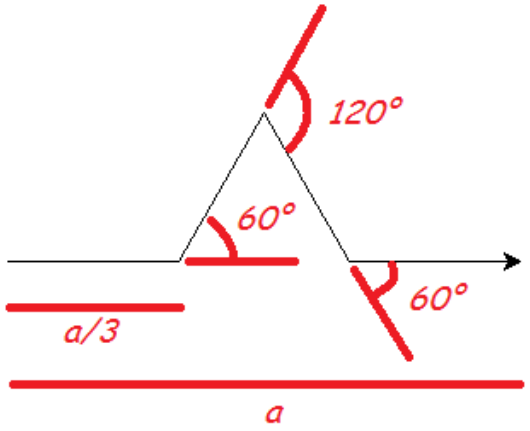
koch(100, 0) pensize(3)
koch(100, 1)

Recursive graphics - Koch curves

- Idea: recursively applying a simple rule to each of the triangles sides
- Examples:
  - The pattern:



order 0 order 1 order 2 order 3



Each of the 4 lines is `a/3` long, and because of the choice of angles, the distance between the ends of the line is `a`.

Recursive graphics - Koch curves

- **Idea:** recursively applying a simple rule to each of the triangles sides
  - Replace the **`forward(a/3)`** with another call to **`koch(a/3, order-1)`**, to draw a kinked line of the same length.
  - The variable `order` goes down by one each time, and when it hits zero, we just draw a line. Change the function **koch**, just a little:

```
from turtle import *

def koch(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            koch(a/3, order - 1)
            left(t)
    else:
        forward(a)
```

```
# Sample Run
reset()
clear()
koch(300, 2)
```

Recursive graphics - Koch curves

- **Idea:** recursively applying a simple rule to each of the triangles sides
  - Replace the **`forward(a/3)`** with another call to **`koch(a/3, order-1)`**, to draw a kinked line of the same length.
  - The variable **`order`** goes down by one each time, and when it hits zero, we just draw a line. Change the function **koch**, just a little:

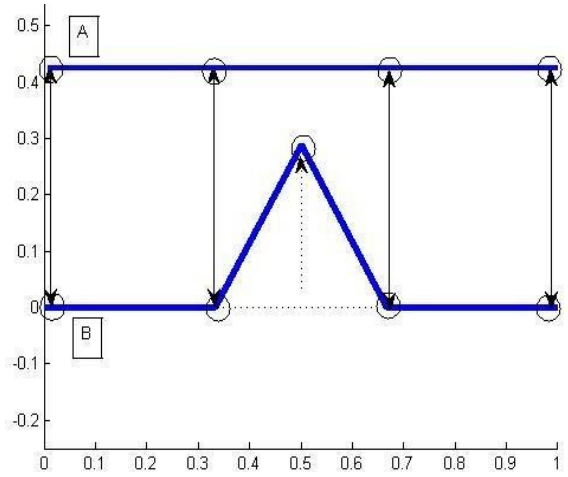
```
from turtle import *

def koch(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            koch(a/3, order - 1)
            left(t)
    else:
        forward(a)
```

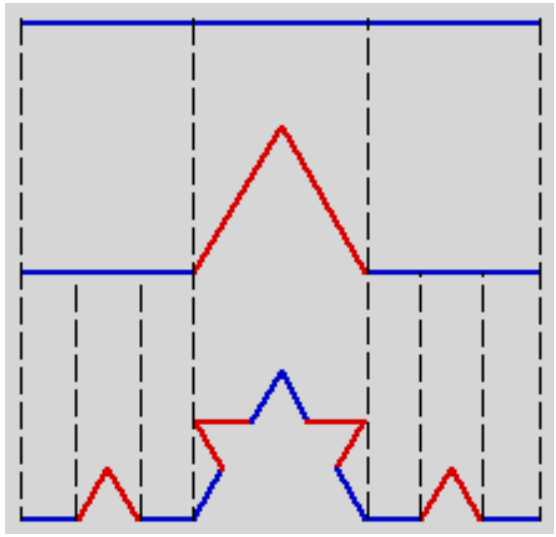
```
# Sample Run
reset()
clear()
koch(300, 2)
```

- Which function does the actual drawing?
- What is the value of **`a`** in "**`else`**" part?
- How many times is **`a`** divided by **`3`**?
- How many times is **`koch(order=2)`** invoked?
  How many times is **`koch(order=1)`** invoked?
  How many times is **`koch(order=0)`** invoked?

Recursive graphics - Koch curves

- Idea: recursively applying a simple rule to each of the triangles sides
- Examples:



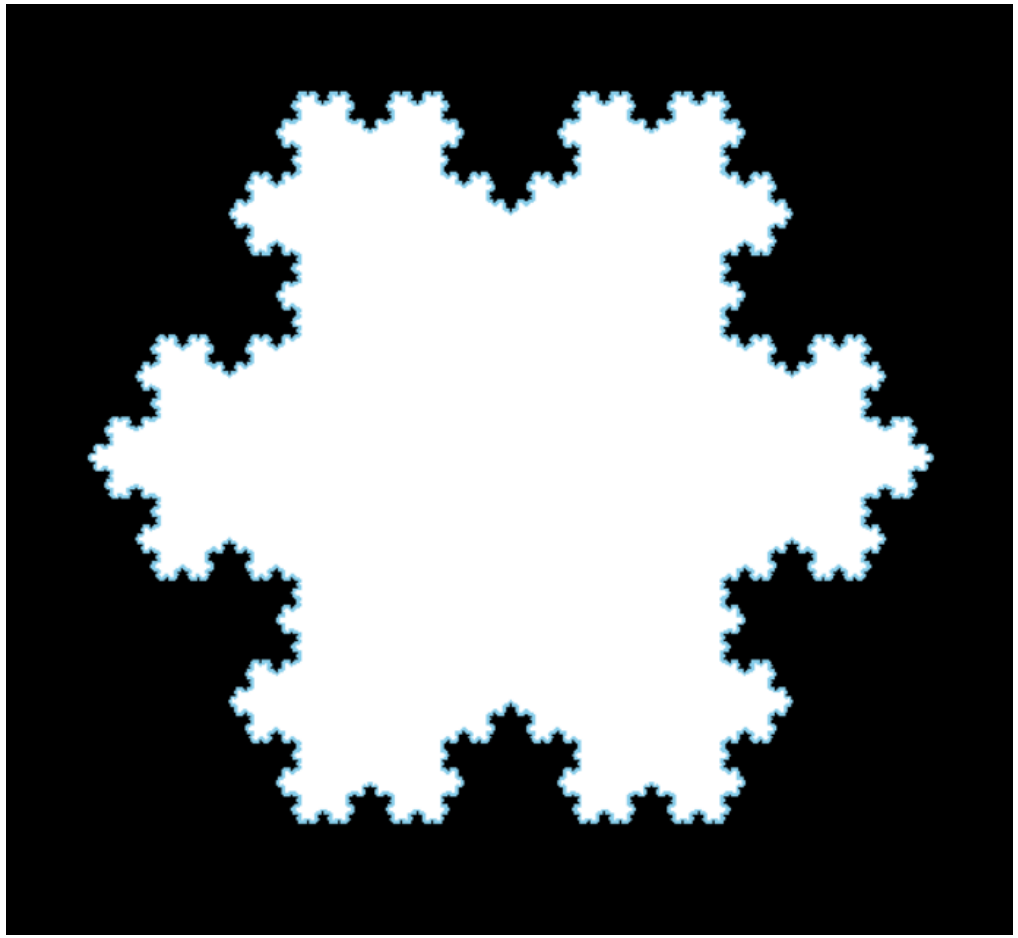


- The pattern:
  - Let us set the side length = SIDE.
  - Cut the SIDE into 3 equal parts SIDE/3.
  - Replace the center part with 2 sides of length SIDE/3, such that it forms a spike.
  - Repeat the process for each of the 4 sides, until the length of each side is smaller than a given value.

Python Turtle - The Koch Snowflake using recursion

- The **algorithm** for drawing a Koch snowflake with n sides is:
  - for each side:
    - draw a Koch curve of the appropriate length and level
    - turn right 360.0/n degrees
- This is an example of a Koch snowflake made with **three** Koch curves.

```
for i in range(3):
    koch(size, order)
    right(120)
```



Python Turtle - The Koch Snowflake using recursion

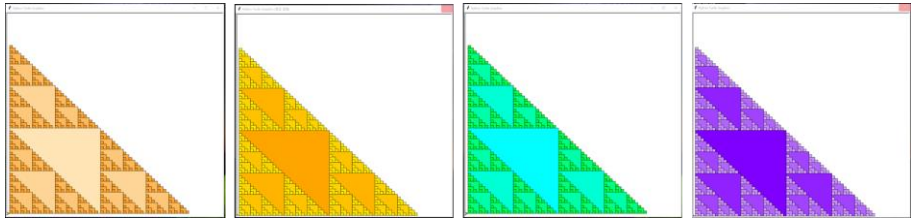
- Code:

```
from turtle import *

def koch(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            koch(a/3, order - 1)
            left(t)
    else:
        forward(a)
```

```
reset()

# Choose colors and size
color("sky blue", "white")
bgcolor("black")
size = 400
order = 7

# Ensure snowflake is centered
penup()
backward(size/1.732)
left(30)
pendown()

begin_fill() # Three Koch curves
for i in range(3):
    koch(size, order)
    right(120)
end_fill()
update()     # Make the last parts appear
```

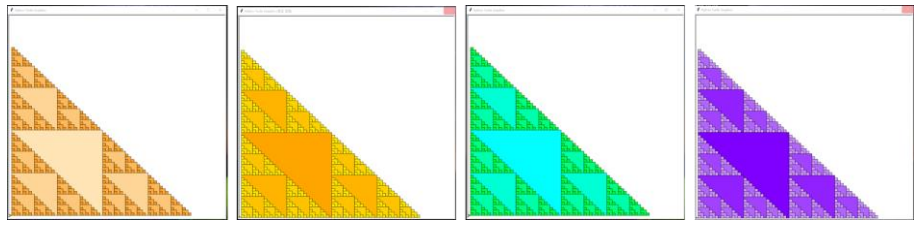
Summary

- We visually experienced how the recursion works through recursive graphics.
- All good recursion must come to an end. Sooner or later method must NOT call itself recursively. It must have the base case(s).
- Recursion is a powerful tool!



Data Structures in Python
Chapter 4

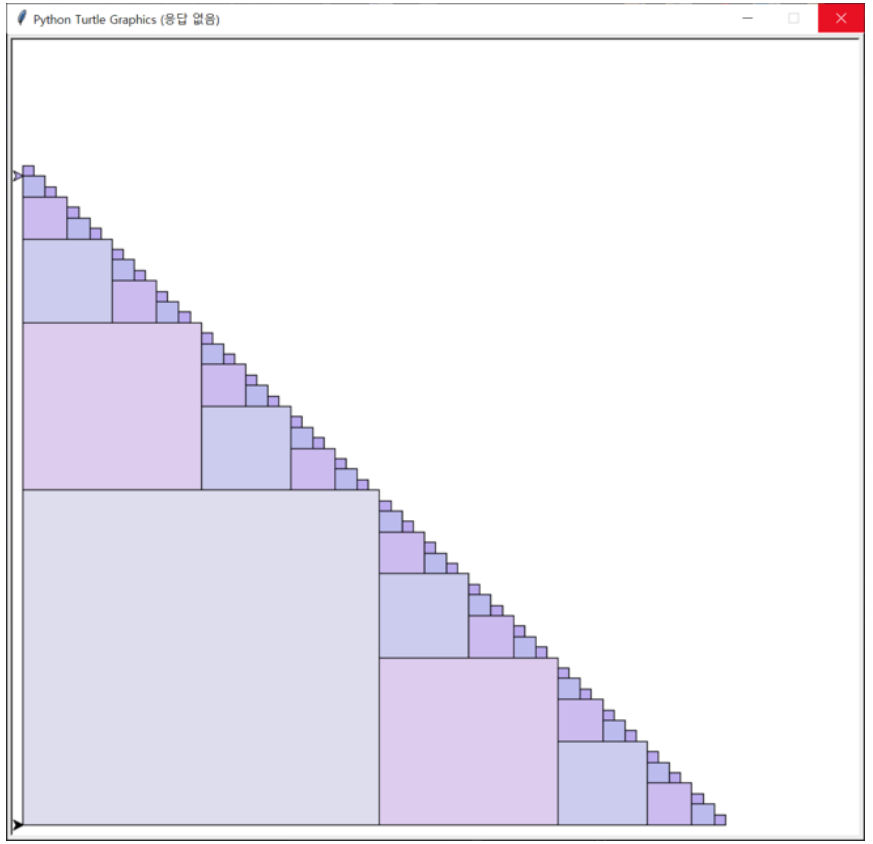
1. Recursion Concepts
2. Recursion Stack and Memoization
3. Recursive Algorithms
4. Recursive Graphics
   Exercise - Stacking boxes



Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

Exercise: Stacking boxes

- Draw a stack of boxes using the recursion and turtle.
  - These boxes consists of a large box in the lower left, a smaller box just above it, and a smaller box to the right of it.



Exercise: Stacking boxes

- Step 1: Draw boxes at the origin in two ways.
  - **Iterative method:** In `stack_boxes()`, invoke `draw_boxes()` **repeatedly** such that all boxes are drawn at the origin.

- The bottom left corner of the window is set to the origin(0, 0).
- The degree determines the number of boxes to draw.
- The size of boxes are set to a power of 2.
- The maximum degree is limited to 8 for easy coloring.
- draw_box() starts and finishes its drawing at the origin given.
- fillcolor() is set by colormap[degree].

```
if __name__ == '__main__':
    Turtle()
    setworldcoordinates(0, 0, 600, 600)
    stack_boxes(0, 0, 256, 6)
```

Exercise: Stacking boxes

- For this exercise, only two functions shown below will do the job.

```
%%writefile box.py
from turtle import *
def draw_box(x, y, side, color):
    """x, y - the origin of the box to draw at lower left corner
       side - the length of the side of the box
       color - fill color for the box """

    # your code here
```

```
from turtle import *
from box import *
def stack_boxes(x, y, side, degree):
    """x, y - the origin of the box to draw at lower left corner
       side - the length of the side of the box
       degree - number of boxes to stack side or top.
            assume the max degree is limited to 8. """
    colormap = ["#unused", "#BBAAEE", "#BBBBEE", "#CCBBEE", \
                "#CCCCEE", "#DDCCEE", "#DDDDEE", "#EEDDEE", "#EEEEEE"]

    # your code here
```

```
if __name__ == '__main__':
    Turtle()
    setworldcoordinates(0, 0, 600, 600)
    stack_boxes(0, 0, 256, 6)
```

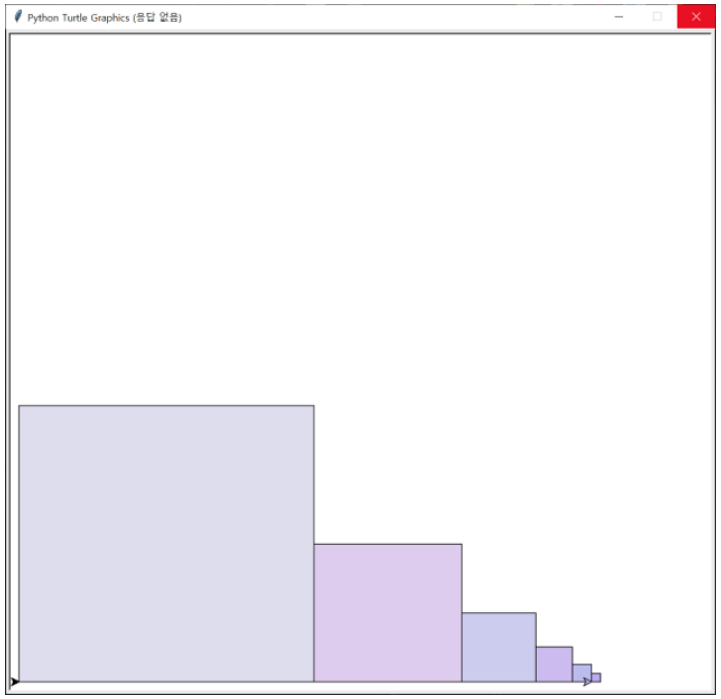
Exercise: Stacking boxes

- Step 2: Draw boxes at the origin in two ways.
  - **Iterative method:** In `stack_boxes()`, invoke `draw_boxes()` **repeatedly** such that all boxes are drawn at the origin.
  - **Recursive method:** In `stack_boxes()`, invoke `draw_boxes()` **only once**, but `stack_boxes()` invoked **recursively** makes `draw_boxes()` called many times such that all boxes are drawn at the origin.

```
if __name__ == '__main__':
    Turtle()
    setworldcoordinates(0, 0, 600, 600)
    stack_boxes(0, 0, 256, 6)
```

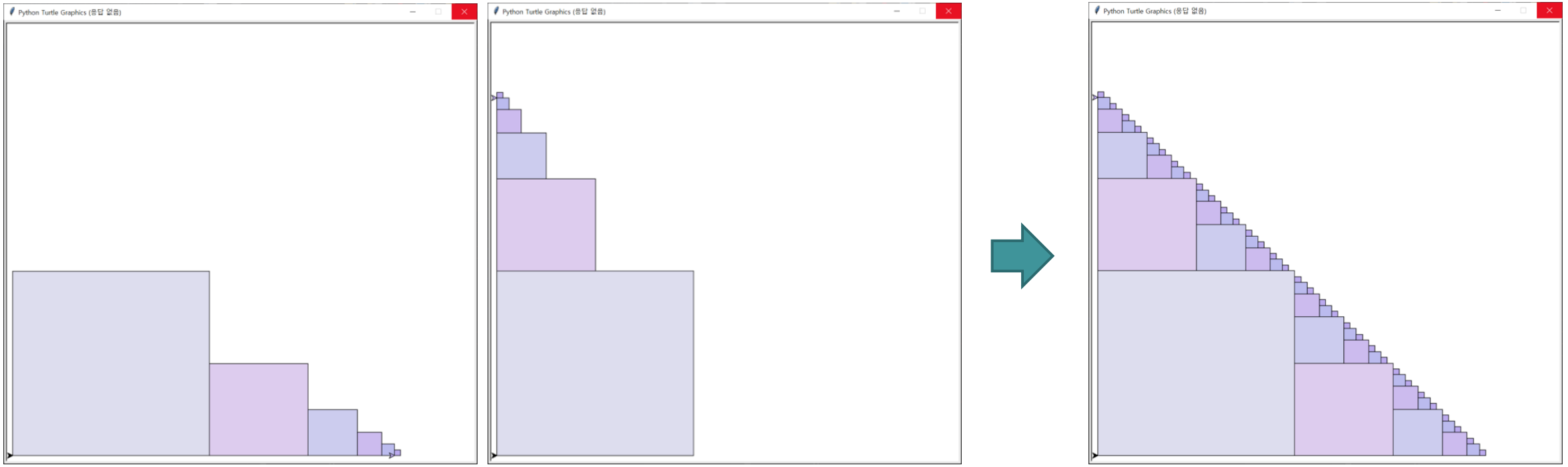
Exercise: Stacking boxes

- Step 3: Draw a stack of boxes at the bottom first and at the top next, recursively.
  - It is a matter of setting a new origin of the boxes to draw.



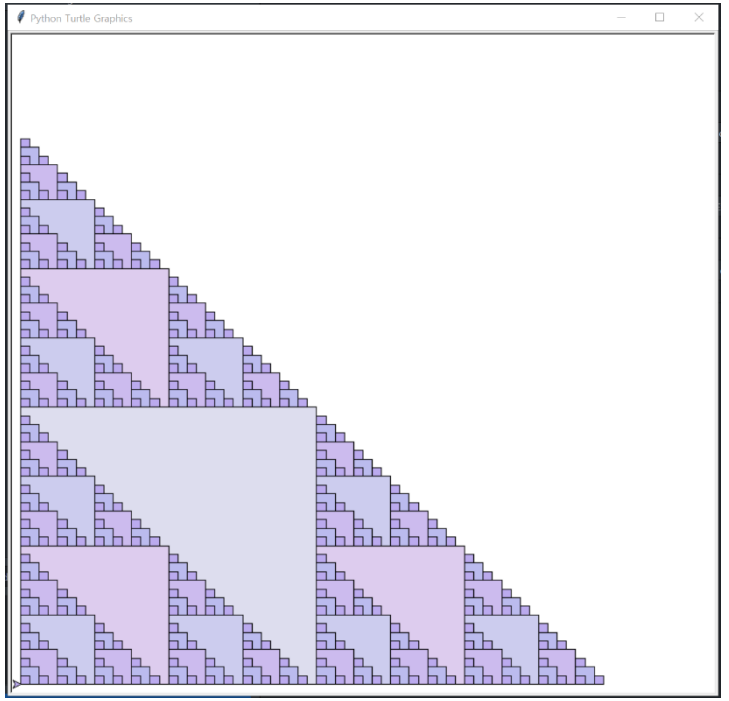
Exercise: Stacking boxes

- Step 4: Invoke stack_boxes() twice such that boxes, one to the right and one for at the top, can be recursively drawn, by setting the new origin.
  - Stop if the degree is less than 1.



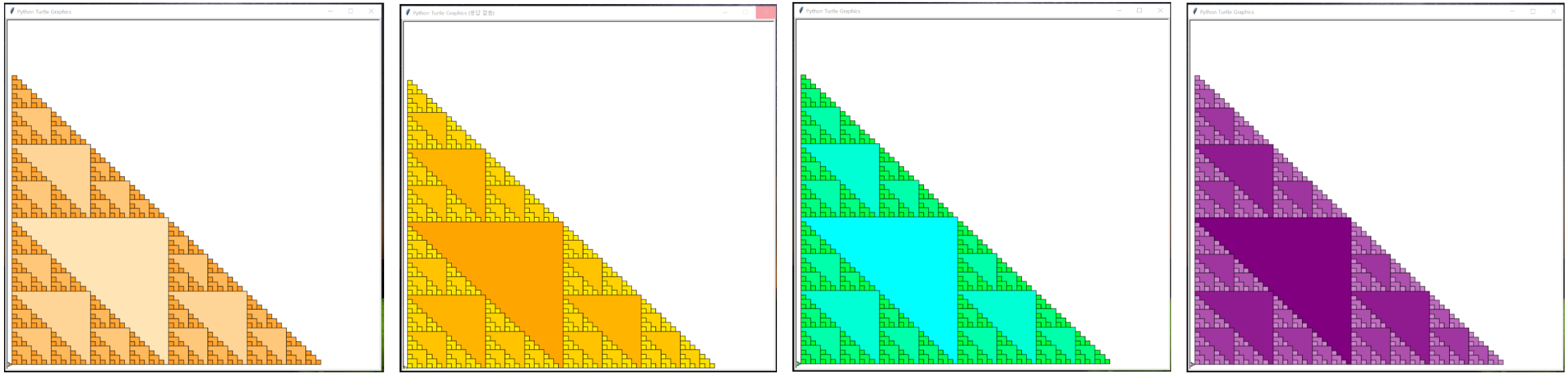
Exercise: Stacking boxes

- Step 5: Pile up more boxes in boxes.
  - Pile up more boxes inside boxes by invoking one more recursive call.



Exercise: Stacking boxes

- Step 6: Coloring boxes.
  - We want to replace the colormap which is hard-coded using a fixed (magic) number.
  - Before invoking stack_boxes(), make a colormap based on degree using color_fader(). color_fader() returns a color between two colors mixed (0 to 1 ratio).
  - Pass this new colormap as an additional argument of stack_boxes().

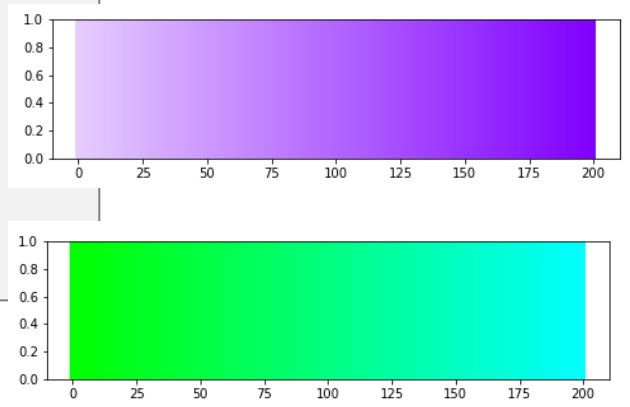


Exercise: Stacking boxes

- Step 6: Coloring boxes.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy

def color_fader(c1,c2,mix=0):
    #fade (linear interpolate) from color c1 (at mix=0) to c2 (mix=1)
    c1=numpy.array(mpl.colors.to_rgb(c1))
    c2=numpy.array(mpl.colors.to_rgb(c2))
    return mpl.colors.to_hex((1-mix)*c1 + mix*c2)

if __name__ == '__main__':
    c1 = '#e6ccff'    #light purple      c1 = 'lime'
    c2 = '#8000ff'    #dark purple       c2 = 'cyan'
    n=200
    fig, ax = plt.subplots(figsize=(8, 2))
    for x in range(n+1):
        ax.axvline(x, color=color_fader(c1, c2, x/n), linewidth=4)
    plt.show()
```

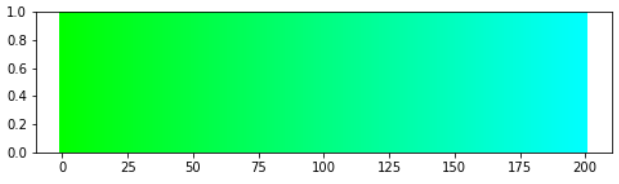
You may use color names such as 'lime' or 'cyan'.

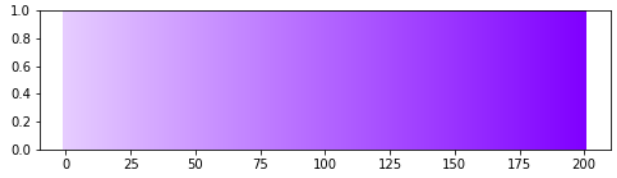


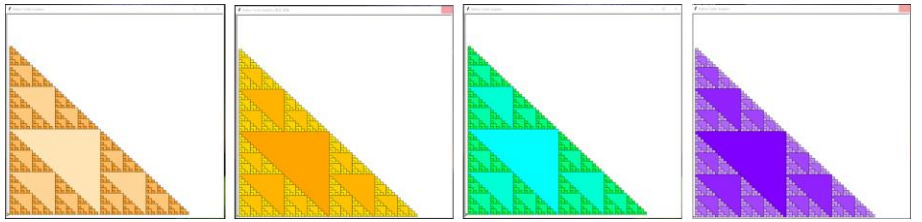
Exercise: Stacking boxes

- Step 6: Coloring boxes.
  - We want to replace the colormap which is hard-coded using a fixed (magic) number.
  - Before invoking stack_boxes(), make a colormap based on degree using color_fader(). color_fader() returns a color between two colors mixed (0 to 1 ratio).
  - Pass this new colormap as an additional argument of stack_boxes().

```
if __name__ == '__main__':
    Turtle()
    setworldcoordinates(0, 0, 600, 600)
    side = 256
    degree = 6
    cmap = [None]     # list of colors by degree, use color_fader(), list comprehension
    # ...
    stack_boxes(0, 0, side, degree, cmap)
```







Summary

- We visually experienced how the recursion works through recursive graphics.
- All good recursion must come to an end. Sooner or later method must NOT call itself recursively. It must have the base case(s).
- Recursion is a powerful tool!